

Cybernetics Oriented Programming (CYBOP)

An Investigation on the Applicability of Inter-Disciplinary Concepts
to Software System Development

Christian Heller

Cybernetics Oriented Programming (CYBOP)

An Investigation on the Applicability of Inter-Disciplinary Concepts
to Software System Development



Ilmenau

Cataloging-in-Publication Data

Christian Heller.

Cybernetics Oriented Programming (CYBOP):

An Investigation on the Applicability of Inter-Disciplinary Concepts
to Software System Development

Ilmenau: Tux Tax, 2006

ISBN-10: 3-9810898-0-4

ISBN-13: 978-3-9810898-0-6

Information on Ordering this book

<http://www.tuxtax.de>, <http://www.cybop.net>

Written as Dissertation

Supervisor 1: Prof. Dr.-Ing. habil. Ilka Philippow (Chair), Technical University of Ilmenau

Supervisor 2: Prof. Dr.-Ing. habil. Dietrich Reschke, Technical University of Ilmenau, Germany

Supervisor 3: Mark Lycett (PhD), Brunel University, Great Britain

Submission: 2005-12-12; Presentation: 2006-10-04

Copyright © 2002-2006. Christian Heller. All rights reserved.

Cover Illustration: TSAMEDIEN, Düsseldorf

Printing and Binding: Offizin Andersen Nexö, Leipzig/ Zwenkau

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Trademark Credits

Most of the software-, hardware- and product names used in this document are also trademarks or registered trademarks of their respective owners.

Donations

Companies planning to publish this work on a grand scale are asked to notify the author <christian.heller@tuxtax.de> and to consider donating some of their sales revenues, which will be used exclusively for the CYBOP and Res Medicinae free software projects.

Text printed on recycled and acid-free paper.

Printed in Germany

To all kind-hearted People who contribute to Humanity;
against Those whose only Aim in Life is to amass Money

Contents

Preface	xv
1 Introduction	1
1.1 Information Science	1
1.2 Software Crisis	2
1.3 Motivation	4
1.4 Cybernetics	5
1.5 Method	5
1.6 Example	7
1.7 Structure	8
I Basics	11
2 Software Engineering Process	13
2.1 Waterfall Process	14
2.2 Iterative Process	14
2.3 Agile Methodologies	16
2.4 Extreme Programming	17
2.5 Method Maturity	19
2.6 Abstraction Gaps	19
2.7 Software Architecture	22
3 Physical Architecture	25
3.1 Process	26
3.2 Application Server	27

3.3	Database Server	28
3.4	Presentation Client	30
3.5	Web Client and Server	31
3.6	Local Process	32
3.7	Human User	33
3.8	Peer Node	34
3.9	Remote Server	35
3.10	Legacy Host	36
3.11	Systems Interconnection	37
3.12	Scalability	39
3.13	Misleading Tiers	40
4	Logical Architecture	43
4.1	Paradigm and Language	45
4.1.1	Language History	45
4.1.2	Paradigm Overview	47
4.1.3	Hardware Architecture	48
4.1.4	Machine Language	51
4.1.5	Assembly Language	51
4.1.6	Structured- and Procedural Programming	51
4.1.7	System Programming	57
4.1.8	Typeless Programming	58
4.1.9	Functional Programming	58
4.1.10	Logical Programming	60
4.1.11	Data Manipulation Language	61
4.1.12	Markup Language	61
4.1.13	Page Description Language	66
4.1.14	Hardware Description Language	67
4.1.15	Object Oriented Programming	68
4.2	Pattern	79
4.2.1	Architectural	81
4.2.2	Design	97
4.2.3	Idiomatic	103
4.2.4	Framework	107
4.3	Component Oriented Programming	109
4.3.1	Inversion of Control	110

4.3.2	Component Lifecycle	111
4.3.3	Interface and Implementation	112
4.3.4	Separation of Concerns	113
4.3.5	Spread Functionality	115
4.3.6	Aspect Oriented Programming	117
4.3.7	Agent Oriented Programming	120
4.4	Domain Engineering	122
4.4.1	Tool & Material	124
4.4.2	Generics	124
4.4.3	Domain Specific Language	125
4.4.4	Specification Language	127
4.4.5	Generative Programming	131
4.4.6	Model Driven Architecture	131
4.4.7	Model and Code	133
4.5	Knowledge Engineering	135
4.5.1	Representation Principles	137
4.5.2	Date and Rule	137
4.5.3	Agent Communication Language	138
4.5.4	Semantic Web	141
4.6	Conceptual Network	143
4.6.1	Ontos and Logos	144
4.6.2	Applicability	145
4.6.3	Two Level Separation	145
4.6.4	Building Blocks	146
4.6.5	Terminology	148
4.6.6	Schemes	149
4.6.7	Ontology	152
4.6.8	Archetype	153
4.6.9	Dual Model Approach	155
4.7	Modelling Mistakes	158
5	Extended Motivation	161
5.1	Idea	162
5.2	Recapitulation	163
5.3	Approach	165

II Contribution	171
6 Statics and Dynamics	173
6.1 Virtual- and Real World	173
6.1.1 Mind and Body	173
6.1.2 Brain Regions	176
6.1.3 Cell Division	177
6.1.4 Short- and Long-Term Memory	178
6.1.5 Information Processing Model	180
6.1.6 Persistent and Transient	181
6.2 System and Knowledge	182
6.2.1 Configurable or Programmable	182
6.2.2 Code Reduction	184
6.2.3 Base- and Meta Level	185
6.2.4 Reference- and Archetype Model	185
6.2.5 Common- and Crosscutting Concerns	186
6.2.6 Application and Domain	187
6.2.7 Platform Specific and -Independent	189
6.2.8 Agent with Mental State	189
6.2.9 Data Garden	190
6.3 Knowledge Management System	192
6.3.1 Hardware Connection	192
6.3.2 Memory	194
6.3.3 Processing	194
6.3.4 Lifecycle	196
7 Knowledge Schema	199
7.1 Human Thinking	199
7.1.1 Basic Behaviour	199
7.1.2 Conglomerate	201
7.1.3 Abstraction	202
7.1.4 Interaction	207
7.1.5 Intrinsic or Extrinsic Properties	210
7.1.6 Language	210
7.1.7 Quality and Quantity	213
7.2 Design Reflections	214

7.2.1	Pattern Systematics	214
7.2.2	Recommendation	216
7.2.3	Model Metamorphosis	217
7.2.4	Structure by Hierarchy	221
7.2.5	Association Elimination	222
7.2.6	Hierarchical Algorithm	224
7.2.7	Framework Example	225
7.2.8	Categorisation versus Composition	229
7.3	Knowledge Representation	230
7.3.1	Knowledge Ontology	230
7.3.2	Schema	234
7.3.3	Double Hierarchy	235
7.3.4	Modelling Example	237
7.3.5	Container Unification	239
7.3.6	Universal Memory Structure	239
8	State and Logic	243
8.1	A Changing World	243
8.1.1	Change follows Rules	243
8.1.2	From Philosophy to Mathematics	244
8.1.3	System	247
8.1.4	Self Awareness	250
8.1.5	Communication	253
8.2	Translator Architecture	257
8.2.1	Interacting Systems	257
8.2.2	Basic Patterns	259
8.2.3	Placement	261
8.2.4	Simplification	262
8.2.5	Communication Model	263
8.3	Knowledge Abstraction and -Manipulation	265
8.3.1	Algorithm	265
8.3.2	Operations	266
8.3.3	Primitives	266
8.3.4	Logic Manipulates State	267
8.3.5	Without Capsules?	269

III Proof	271
9 Cybernetics Oriented Language	273
9.1 Formality	273
9.2 Definition	274
9.2.1 Syntax	275
9.2.2 Vocabulary	276
9.2.3 Semantics	281
9.2.4 Tag-Attribute Swapping	283
9.3 Constructs	284
9.3.1 State Examples	284
9.3.2 Logic Examples	289
9.3.3 Special Examples	293
9.3.4 Inheritance as Property	299
9.3.5 Container Mapping	300
9.3.6 Hidden Patterns	301
9.4 Comparison	301
9.4.1 RDF	302
9.4.2 OWL	303
9.5 Tool Support	305
9.5.1 Template Editor	305
9.5.2 Knowledge Designer	306
9.5.3 Model Viewer	310
10 Cybernetics Oriented Interpreter	313
10.1 Architecture	313
10.1.1 Overall Placement	313
10.1.2 Inner Structure	314
10.1.3 Pattern Merger	316
10.1.4 Kernel Concepts	317
10.1.5 Security	319
10.2 Functionality in Detail	321
10.2.1 Process Launching	322
10.2.2 Lifecycle Management	322
10.2.3 Signal Checking	323
10.2.4 Signal Handling	323

10.2.5	Operation Execution	324
10.2.6	Model Transition	324
10.2.7	Data Creation	325
10.3	Implementation	327
10.3.1	Simplified C	327
10.3.2	Corrected C	328
10.3.3	Used Libraries	328
10.3.4	Development Environment	329
10.3.5	Error Handling	329
10.3.6	Distribution and Installation	330
11	Res Medicinae	331
11.1	Project	331
11.1.1	Free and Open Source Software	331
11.1.2	Portals and Services	332
11.1.3	Tools	333
11.1.4	Contributors	334
11.2	Analysis	335
11.2.1	Requirements Document	335
11.2.2	EHR & Co.	335
11.2.3	Episode Based	337
11.2.4	Evidence Based	338
11.2.5	Continuity of Care	339
11.2.6	Core Model	339
11.3	Standards	341
11.3.1	Overview	341
11.3.2	Record Modelling	342
11.3.3	Messaging and Communication	344
11.3.4	Terminology Systems	347
11.3.5	Further Standards	352
11.3.6	Standards Development	355
11.3.7	Implication	356
11.4	Realisation	357
11.4.1	Student Works	357
11.4.2	First Trial	359
11.4.3	Knowledge Separation	360

11.4.4 Reimplementation	362
11.4.5 Module Modelling	363

IV Completion 367

12 Review 369

12.1 Validation	369
12.1.1 Distinction of Statics and Dynamics	370
12.1.2 Usage of a Double-Hierarchy Knowledge Schema	372
12.1.3 Separation of State- and Logic Knowledge	373
12.2 Evaluation	374
12.2.1 Knowledge Triumvirate	374
12.2.2 Common Knowledge Abstraction	376
12.2.3 Long-Life Software System	377
12.3 Limits	378

13 Summary and Outlook 381

13.1 Summary	381
13.2 Future Works	383
13.3 Fiction	387

14 Appendices 389

14.1 Abbreviations	389
14.2 References	409
14.3 Figures	437
14.4 Tables	443
14.5 History	445
14.6 Migration to CYBOL	453
14.7 Call for Developers	455
14.8 Abstract	457
14.9 Kurzfassung	459
14.10 Licences	461
14.10.1 GNU General Public License	461
14.10.2 GNU Free Documentation License	469
14.11 Index	479

Preface

*I slept and dreamt that Life was Joy.
I awoke and saw that Life was Service.
I acted and behold, Service was joy.*

RABINDRANATH TAGORE

Prologue

To me, basically, there are two ways to deal with a scientific subject:

1. The deepened investigation on a special area aiming to find completely new phenomena
2. The systematic subsumption of multiple known aspects of one or many disciplines aiming to find new cross-correlations and ideas

Both approaches may lead to new theories, methods and concepts. And both may use laboratory trials to find and prove their theories. This work follows the second approach. The idea behind is, simply spoken, to steal ideas from nature and various fields of science, and to apply them to software design.

Laboratory Trials are what *Coding* is in informatics – experiment and proof of operability, at the same time. Some information scientists have the opinion that coding weren't *scientific* enough and not necessary to create new theories or to achieve good results. I doubt this. In my opinion, there are things that can only be found when actually implementing ideas in a computer language. And in the end, a theory is worth much more when having been proven in practice. This document contains proven ideas that were growing in my mind over the last few years, while dealing with topics such as:

- Structured- and Procedural Programming
- Object Oriented Programming
- Design Patterns and Frameworks
- Component Based Design and Agents
- Ontology Structured Domain Knowledge
- Document- and User Interface Markup
- Persistence Mechanisms
- System Communication
- Operating System Concepts

The usage of typical buzzwords could not quite be avoided in this work, yet do I hope that the ideas and results are nevertheless explained straightforward and well enough to be really useful to some other developers out there.

This document claims to be an *Academic Paper*. To all practitioners who do not want to read it for that reason, I would like to point out that each and every concept in it arose from practice, that is coding. Like most developers, I started up with only a few lines of code in one Java class, later extended to more classes, a whole framework and so on. Whenever I stumbled over difficulties, I thought through and improved my current design by applying patterns recommended by several software development Gurus. It was only when I realised that even those concepts were not sufficient, that I made up my own. They are entitled *Cybernetics Oriented Programming* (CYBOP), because most ideas behind them stem from nature.

Finally, this document has become my thesis, written to earn a doctorate (Dr.-Ing./ PhD) in Informatics/ Software Engineering. You may wonder why I release it under the *Free Documentation License* (FDL). Well, I'm a full supporter of the idea of *Free Knowledge*, *Free Software*, a *Society free of Patents* which are only hindering its development. There are three reasons that have contributed to my decision:

1. Hope to get helpful *Feedback* from readers
2. Trust in the scientific *Fairness* of colleagues, worldwide, to properly reference this document even though it is licensed under the FDL
3. Wish to contribute to the open source movement *now* (and not in some years when the document might reach a more stable version), to speed up its successful development

This is a growing document undergoing steady development. It is not and doesn't claim to be free of errors nor to contain the only possible way for application system development. So, if you find errors of whatever kind or have any helpful ideas or constructive critics, then please contribute them to <christian.heller@tuxtax.de> or to the CYBOP developers mailing list <cybop-developers@lists.berlios.de>!

Scientific Progress

An *Abstraction* allows to capture the real world by representing it in simplified models. Such models contain only the essential aspects of a special domain. Any unimportant nuances, in the considered context, are neglected. Correct abstract models is what makes science easy. Good science *can* be *easy*. If it is not, then probably either:

- there is a *mistake* in the model
- it is not fully *understood* by the scientist him/ herself
- the explaining person wants to *keep back* knowledge, making others look clueless

One of the biggest hindrances to scientific progress is too much or false respect for existing solutions. No theory/ model/ concept is ever finished; no document/ software/ product is ever fully completed. There is always room for improvements. In the end, it is all just a person's subjective perception and an arbitrary, abstract extract of the real world.

It is always worth reviewing and questioning everything in depth, again and again. Stand-still means regress. The best example showing how to work around these critics is the *Free and Open Source Software* (FOSS) movement where all the time, existing solutions are rewritten, to be improved.

Software Patents

This work is about software. Software abstracts the real world, its items and processes, and it can store these information and their relations which make up actual *Knowledge*. In the modern, so-called *Information Society*, it becomes more and more important to have *free* access to external knowledge. This is an *essential* human right and will decide about the future living quality of people.

So much the important it is to *prohibit* the application of patents to software! They make an exclusive club of large companies own the rights on banal, ordinary, day-to-day algorithms and methods that many people use. And, they thereby kill any new ideas and hinder research efforts that depend on these basic algorithms. If *Software Patents* and patents on *Computer Implemented Inventions* (CII) got introduced, any free software developer and especially *Small- and Medium Sized Enterprises* (SME), the driving force of innovation, could not unfold their full potential anymore, since much of their time and effort would then have to go into patent inquiries and costly legal disputes.

Software patents are dangerous for the free development of thoughts! Certain lobbies exert an increasing influence on politics and push members of parliaments to agitate and vote in their interest. Since probably every reader of this document has an interest in informatics, every reader is also affected by the software patent enforcement. But everybody can do something about it, not only in Europe! Express your protest and sign the petition at [91]!

Free Publishing

Reputation in the scientific world strongly depends on the number of publications in scientific journals, conference proceedings, magazines etc., of which some have greater kudos, some less. A *Philosophiae Doctor* (PhD) student, for example, is expected to publish in some of the *acknowledged* journals, in order to be conferred a doctorate. The grant of project fundings by local-, national- or *European Union* (EU) governments and sponsorship of a professor's department at university depend on it as well. Some unfair practices and shortcomings of the current system of publication shall therefore be mentioned here. There are at least four disadvantages of publishing in scientific journals. An author:

- is almost always forced to assign his copyright to the publisher;
- has very little chance of publishing completely new ideas, since evaluators (which are to guarantee a certain *scientific level*) sieve those which seem too crazy or are unknown to them and do not match state-of-the-art science, so that really new ideas can hardly become popular in this way;
- has to wait many months before being informed about article acceptance, sometimes further months to presentation at a conference and yet more months until a journal/proceedings are finally available – which, besides the unfine delay, is enough time for an evaluator to adapt the best ideas and publish them in a modified form before;

- and everyone else have to pay money for receiving journals (even for the one containing the author's own work), or become a member of certain scientific societies for some discount – which means that the work is not freely accessible.

Further, there is something often labelled *Citation Mafia*. Whether an article gets published in a journal or not depends on it being accepted by a number of reviewers (normally three). In order to avoid personal battles, the article author never gets to know the evaluators' names or proficiency and has to blindly rely on the *good taste* of a conference's program committee. However, evaluators, although tied to ethical standards, often seem to have their list of *friends* or seem to just prefer authors who have already published elsewhere, leading to circles of scientists citing each other, quite independent from the quality of their papers. Logically, also here, there are a number of disadvantages:

- Young scientists have a hard life and need a long time for getting their articles accepted, independent from how innovative they are.
- Mafioso scientists often warm up old stories or deliver well-formulated, but rubbish articles not earning the predicate *scientific*.

Don't ask for proof – I don't have it. But almost everybody in the scientific business knows about these issues. Unfortunately, only few people [166] talk about- or try to change them. Obviously, many scientists prefer to either play the same old game or are scared of personal disadvantages. However, it feels like increasingly more researchers, in particular the new generation, become aware that these drawbacks hinder scientific progress and new solutions need to be found. Well, there is free online journals such as the *Journal of Free and Open Source Medical Computing* (JOSMC) [226] or the *BioMed Central* (BMC) [326] publisher, where research articles are: *free to access immediately, peer reviewed, citation-tracked ...*

Although this document cannot deliver solutions to the above-mentioned problems, it mentioned those to inform the reader and spur further discussion. Supportive actions in this process would be that:

- scientists acknowledge no-cost entry open source conferences like LinuxTag & Co. [82] as alternatives to traditional ones
- professors more readily accept citations of free knowledge sources such as Wikipedia [60] in scientific works of their students
- students and scientists publish their works (code and documentation) under open source licenses

New Science

It was end of October 2004 that I discovered Stephen Wolfram's book *A New Kind of Science* [344] (published in 2002), through a link in Wikipedia [60]. By that time, I was already heavily writing on my own work.

During those years of thinking about software systems, nature, the universe – I felt pretty similar to how Wolfram describes it in the preface of his book. Starting with an inspection of state-of-the-art techniques, diving deeper and deeper into several topics, I soon realised that they all could not deliver a *coherent, conclusive* solution to software modelling. Each had its own drawbacks that made workarounds necessary. And, the more I dived into the different technologies, the more complex, complicated, intransparent they got – but still, none seemed to provide an *overall* solution.

It was only when I got more and more distance to existing solutions and moved away from current thinking, towards a more universal approach and a view at software systems through the eyes of nature, that I found the basic principles described in this work.

Now, after having read *A New Kind of Science*, I am glad that Wolfram did not already write down everything I want to say, so that there is something left for me to contribute, by delivering this work :-). There is one difference that soon became obvious to me: Wolfram argues, that it is possible to study the abstract world of simple programs, and take lessons from what kinds of things occur there and have them in mind when investigating natural systems [60]. My work follows the exact opposite way, in that it observes phenomenons of nature and concepts used in other sciences, and tries to apply them to the design of software systems.

This is *not* to say that *CYBOP* does provide *the* overall solution. But what it surely wants to reach is to encourage people to think in more general terms, across disciplines, to possibly find new concepts. And for that, this work hopes to deliver some ideas. And I certainly do hope that the more you, as readers, think about these ideas, the more sense they will make to you, too.

Stylistic Means and Notation

The language of choice in this document is *British English*, more precisely known as *Commonwealth English*. Exceptions are citations or proper names like *Unified Modeling Lan-*

guage, stemming from *American English* sources. (In Oxford English, *Modelling* would be written with double letter *l*). I am thinking about writing a *German* version of this document, but am not sure if it will be worth the effort. If you as reader are interested in a translation, send me a short note! The more emails I receive, the more convinced I will be.

Correctly, masculine *and* feminine forms are used in a work. When describing a patient's record, for example, one would write: *his or her record*. In order to improve readability, and exclusively because of this reason, only masculine forms are used in this work.

The document sticks to the widespread *Unified Modeling Language* (UML) [235] standard notation for describing classical software concepts in diagrams, wherever suitable. Minor simplifications are applied wherever these result in a clearer illustration with better overview.

Pieces of software source code are displayed in **Typewriter Typeface**. Emphasised words are *italicised*.

Footnotes are not used on purpose. In my opinion, they only interrupt the flow-of-reading. Remarks are placed in context instead, sometimes enclosed in parentheses.

To all authors and contributors of the Wikipedia Encyclopedia:

I have cited so many Wikipedia articles in this work, that it would not have been possible to create an extra bibliography entry for each of them, without letting the frame of this work explode. Therefore, I have just referenced Wikipedia in general, whenever one of its articles was used.

Some scientists still label Wikipedia a *Pseudo Encyclopedia* not worth being mentioned in scientific works. However, it is my firm believe that this will change in the near future and one day, it will be hard to write any work without referencing Wikipedia knowledge, which will then (if not already now) be of best quality.

Acknowledgements

Certainly, first thanks is due my wife *Kasia* and my *Parents* and *Sisters*, being always with me, in good as in bad times. Not less important to me are my aunt *Maria Kosiza*, my great *Relatives* and our former chaplain *Johannes Preis*, who have helped shaping me the way I am.

I would like to thank my professor, *Ilka Philippow*, for greatly encouraging me during my work while leaving enough room to develop my own ideas. Equal thanks is due my supervi-

sors *Dietrich Reschke* and *Mark Lycett*. *Detlef Streitferdt* and *Bernd Däne* gave numerous hints improving the quality of the first part of my work. Consultation with *Bernd and Wolfgang Fengler* helped me understand Petri Net diagrams and their hardware background as well as Assembler programming. Whenever I got doubts about what I was doing, I was very lucky to receive good motivation from my colleagues *Volker Langenhan*, *Oswald Kowalski*, *Todor Vangelov* and *Kai Böllert*. Oswald's talks about hardware concepts made me find useful parallels to software. Alexander Fleischer helped out when I was struggling with L^AT_EX's paper size option.

My thanks go to my students *Jens Bohl*, *Torsten Kunze*, *Dirk Behrendt*, *Kumanan Kanasabapathy*, *Jens Kleinschmidt*, *Martin Fache*, *Karsten Tellhelm*, *Marcel Kiesling*, *Teodora Kikova*, *Dennis Reichenbach*, *Stefan Zeisler*, *Michael Simon*, *Henrik Brandes* and *Saddia Malik* for contributing their theses, tutorials or source code to the project. Special thanks to *Rolf Holzmüller* who brought in some innovative ideas for CYBOL, in the final phase of my work, and helped cleaning many bugs in CYBOI.

Reminiscences on good times go to my former colleagues of *OWiS Software* who, together with the *Technical University of Ilmenau* (TUI), have contributed with great commitment to the development of the *Object Technology Workbench* (OTW) UML tool which I would have liked to use in the early stages of my work. Pity it hasn't gone Open Source after its development was stopped in 2000 :-). Thanks to *Martin Wolf*, *Rene Preißel*, *Dirk Henning* and all colleagues who have been patient and well-explaining teachers!

I would like to acknowledge the contributors of *CYBOP* [256] and *Res Medicinae* [266], especially all medical doctors, e.g. *Claudia Neumann* and *Karsten Hilbert*, who supported the second project with their analysis work [135] and mailing list discussions. Furthermore, I want to mention *Thomas Beale* from the *OpenEHR* project [22] whose freely published design document (back in 2001) gave me some initial ideas in the early stage of my work. Acknowledged be all these brave *Enthusiasts* of the *Free/ Libre Open Source Software* (FLOSS) community, who have provided me with a great amount of knowledge through a comprising code base to build on. I shall mention the contributors of FLOSS projects such as *Scope* [267], *Apache Jakarta* [253], *JOS* [261], *JDistro* [260], the *OpenHealth* [168] mailing list readers, the *OSHCA* [241] members and all other supporters of our projects and ideals.

Great thanks goes to the *Urban und Fischer* publishing company, for providing anatomical images from their *Sobotta: Atlas der Anatomie* [319] and to the *Open Clip Art* project [103] for its wonderful library of free art! Similarly, I have to thank the free online dictionaries of *LEO* [72] and the *Technical University of Chemnitz* [51].

I am grateful to all people who openly publish their knowledge on the web. Without the numerous free sources, I would have never been able to accomplish this work. Especially in the state-of-the-art part, I had to heavily rely on existing sources. It is also therefore that I have decided to put my work under the *GNU FDL* licence [104]. I would be happy to see large parts of it copied in *Wikipedia* [60]!

Let me finish this preface with ARTHUR SCHOPENHAUER's words:

All truth passes through three stages:

First, it is ridiculed.

Second, it is violently opposed.

Third, it is accepted as being self-evident.

Thank you for reading!

Ilmenau, October 2006

Christian Heller <christian.heller@tuxtax.de>

1 Introduction

Even a Way of a thousand Miles begins with one Step.

SAYING

Information Technology is gaining more and more importance in modern society. Some people even talk of the *Information Age*. What *Electricity* was for the *Industrial Age*, *Information* is for today's society.

And *Software* plays one of the, if not the most important role thereby.

1.1 Information Science

Science is one form in which humans express their aspiration for *Perception*. It should – but unfortunately not always does – serve the well-being of people. Similarly, scientific *Inventions* usually are to ease human's life.

The results of many technical inventions are *Tools*, *Machines* or *Robots* (figure 1.1). A passive tool is a mostly simple device used by humans to carry out a task better. The word machine is used to describe advanced, active tools which can run by themselves, only driven by an external force like steam or electrical energy. A robot, finally, is an enhanced machine which may imitate human behaviour (humanoid) or take over (industrial) tasks that are too dirty, dangerous, difficult, repetitive or dull for humans [60]. Its parts are often called *Hardware*. It does not necessarily have the same shape as the human body but can come very close. Also, it contains some pieces of rudimentary *Intelligence* that lets it act alone (autonomous). The intelligence basically controls the way in which the robot functions what is sometimes called *Workflow* or *Program*. That must be encoded, for example in form of a *Punchcard* or pieces of *Software*, kept as pure text or binary data in some electronic memory

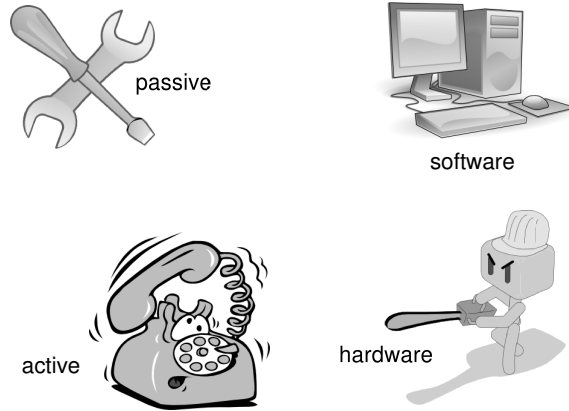


Figure 1.1: Scientific Inventions

or on a storage medium.

A *Computer* can be seen as handicapped robot that can think but not move. Essentially, it represents the intelligent parts of a robot and is able to process (*compute*) *Information* (data content of a message [71]). But its hardware is pruned to pure information input and output. While the importance of robots lies in their *Movement* actions, it lies in problem *Solving* and system *Simulation* for computers [60]. Software plays the biggest role thereby. It contains the programs after which a computer is run, after which it acts.

One important area the science of information, called *Informatics*, deals with is software – the art of *representing* and *processing* information. As such, one of its major aims is to find *Abstract Models* which represent the real world best. The better this is done and the better information can be stored and processed, the better software can assist its human users.

1.2 Software Crisis

An early question in software engineering was how to write programs that control a computer system's *Hardware* correctly and efficiently. Over time, the importance of hardware shifted in favour of *Software* which nowadays contains most of the logic needed to run an application on a computer system. Consequently, much more research emphasis is now placed on the

finding of clever modelling concepts that help writing correct and effective, stable and robust, flexible and maintainable, secure software. Another objective is to increase the effectiveness and lessen the expenditure of cost and time in software development projects, by *reusing* (pieces of) software.

The past 40 years have delivered numerous helpful concepts, for instance *Structure* and *Procedure*, *Class* and *Inheritance*, *Pattern* and *Framework*, *Component* and *Concern*, and many more. They undoubtedly have moved software design far forward. Nevertheless, the dream of true componentisation and full reusability has not been reached. Czarnecki [66] identifies problems in the four areas: *Reuse*, *Adaptability* (in this work also called *Flexibility*), management of *Complexity* and *Performance*.

Modern software is very *complex*. It runs on different hardware platforms, uses multiple communication paradigms and offers various user interfaces. Many tools and methods assist experts as well as engineers in creating and maintaining software but do they not seem sufficient to cope with the complexity so that often, systems still base on buggy source code causing:

- False Results
- Memory Leaks
- Endless Loops
- Weak Performance
- Security Holes

Are these exclusively the fault of software developers? Or, are the used concepts perhaps insufficient? Using the same, allegedly unsatisfying concepts caused some people to talk about an ongoing *Software Crisis*, sometimes *Complexity Crisis*, affecting not only high-level application programming, but also low-level microchip design [67].

However, answers are not easy to find. Software design is *Arts* and *Engineering*, at the same time. Not everything is or can be regulated by rules. It is true, developers have to stick to a set of design rules – and tools that support their usage exist – but they also have to be very creative. All the time, they have to have new, innovative ideas and apply them to software. This is what makes the creation, integration, test and maintenance of software so difficult. There is not really a uniform way of treating it.

1.3 Motivation

To the issues that this work has with some state-of-the-art solutions belong in particular three things:

1. *Abstraction Gaps* in Software Engineering Process (chapter 2)
2. *Misleading Tiers* in Physical Architecture (chapter 3)
3. *Modelling Mistakes* in Logical Architecture (chapter 4)

The traversing of abstraction gaps in a software engineering process belongs to the main difficulties in software development, and causes considerable cost- and time effort. It necessitates a steady synchronisation between domain experts and application system developers, because their responsibilities cannot be clearly separated and interests often clash. A first objective of this work is therefore to contribute to closing these gaps, especially the one existing between a designed system architecture and the implemented source code.

The misinterpretation of the physical tiers in an information technology environment often leads to wrong-designed software architectures. Logical layers are adapted to physical tiers (frontend, business logic and backend) and differing patterns are used to implement them. Instead, systems should be designed in a way that allows the usage of a unified translator architecture, so to give every application system the capability to communicate universally by default, which is the second objective of this work.

Several well-known issues exist with the modelling of logical system architectures, for example: fragile base class problem, container inheritance, bidirectional dependencies, global data access. These and others more result from using wrong principles of knowledge abstraction, like the bundling of attributes and methods in one class, as suggested by object oriented programming, or the equalising of structural- and meta information in a model. A third and final aim of this work is therefore to closer investigate the basic principles and concepts after which current software systems are created, and to search for new ideas and concepts, with the objective of finding a universal type structure (knowledge schema).

On its search for new ideas, this work intentionally tries to cross the borders to other scientific disciplines. It can therefore also be called an *inter-disciplinary* effort. Results from many different sciences are applied to software engineering. Most emphasis, however, is placed on the comparison between human- and computer systems. Nature has always been a good teacher and its principles have often been copied; so does this work.

1.4 Cybernetics

One scientific subject being inter-disciplinary since its creation is *Cybernetics*. Its name stems from the ancient Greek word *Kybernetes* meaning *Steersman* and it has many definitions [134]. One that was coined in 1948 by *Norbert Wiener* sees *Cybernetics* as the science of information and control, no matter whether it is about living things or machines. The *American Heritage Dictionary of the English Language* [251] defines it as *the theoretical study of communication and control processes in biological, mechanical, and electronic systems, especially the comparison of these processes in biological and artificial systems*.

The closely related subject of *Bionics* is a specialisation of cybernetics (*Bionics* = *Bio-Cybernetics*) [73]. It can be defined as *the application of biological principles to the study and design of engineering systems* [251].

Other related fields which are not considered further in this work are morphology (structure-function), general systems theory (complexity, isomorphic relationships), biomechanics (prosthetics), biomimetics, robotics and artificial intelligence. However, the results described in this document might also be of importance in those areas.

Since *Software Engineering* is a kind of *Systems Engineering*, the consideration of systems as a whole gains in importance. *Cybernetics* as science of observing, comparing and controlling biological and technical systems is of great importance in the document on hand. Using models inspired by biology and psychology (but also further disciplines such as philosophy or physics), the science of *Bionics* plays an important role, too.

Sticking to the system idea of *Wiener* and in the fashion of the science of *Bionics*, this work and the new concepts described therein are called *Cybernetics Oriented Programming* (CYBOP).

1.5 Method

Despite all scientific methodology, research is mostly a journey into the blue. Likewise did this work not follow a linear way of progression, but rather a zigzag course between theory and practice (figure 1.2), which may be labelled *Constructive Development*.

At the beginning, there was the wish to create a software application for use in medicine. Development started off by using classical programming techniques. Whenever a problem occurred, it was solved by applying yet more up-to-date techniques and latest software design

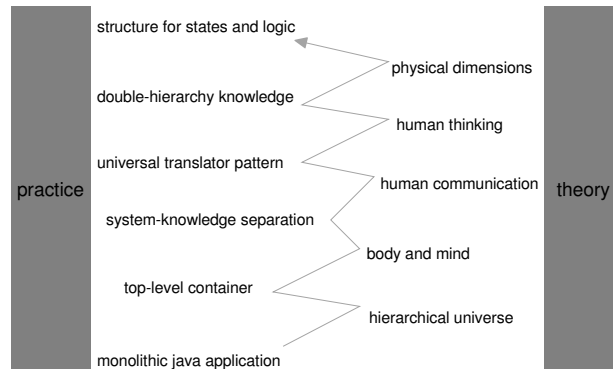


Figure 1.2: Constructive Development

principles, such as *Patterns*. This worked out well until the point at which the complexity of the software could not be handled easily anymore and new ideas were demanded.

It was only when state-of-the-art concepts got more and more unsatisfying and insufficient to maintain a clear architecture, that new ones had to be found. After some time of reflexion, the principles of human thinking for abstracting the real world in artificial models could be identified as source of new ideas for software design. Further ideas were later taken over from other phenomenons of nature and various scientific disciplines. The obvious similarities between human- and computer systems (information input, -storage, -processing, -output) should be rationale enough for an inter-disciplinary approach.

The concepts resulting from both, traditional and new ideas, got finally merged and developed towards the CYBOP theory (figure 1.3). For this new kind of programming, the distinction of *Statics and Dynamics*, a special *Knowledge Schema* and the separation of *State and Logic* are necessary. Chapter 5 will define these in greater detail.

This work reports about the progress of finding new ideas for software design. However, since problems did not occur in a predictable way, while developing the mentioned application, their presentation in order of appearance would be rather confusing. A systematised structure of sections is therefore used in this work to organise most problems after the programming paradigm they belong to. For the interested reader, chapter 11 describes the

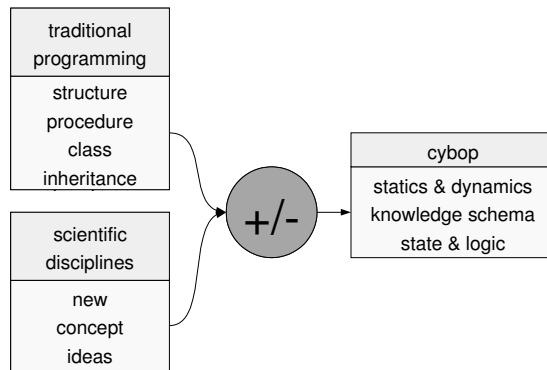


Figure 1.3: Merger of traditional and new Concepts

stepwise construction and taken design decisions of the prototype anyway.

1.6 Example

In the course of this work, most different solutions, frameworks and models have been developed, which is why it turns out to be rather difficult to deliver a continuous example here.

Some traditional concepts and many new ideas of this work are demonstrated on examples taken from a *Medical Information System* environment, with focus on the *Electronic Health Record* (EHR). This counts for the theoretical models of the first and second part as well as for the practical examples in part III. Many other examples and models, though, were picked arbitrarily, depending on their adequacy for demonstrating a corresponding concept or idea.

The actual application of the CYBOP concepts is described in chapter 11 where a prototype software project called *Res Medicinae* gets introduced. It is to validate the new concepts and to give the proof of their operability.

1.7 Structure

This document is divided into fourteen chapters. Neglecting this introduction, thirteen chapters remain which are organised in four parts. They are illustrated in figure 1.4.

basics	contribution	proof	completion
software engineering process	statics & dynamics	cybol	review
physical architecture	knowledge schema	cyboi	summary & future
logical architecture	state & logic	res medicinae	appendices
extended motivation			

Figure 1.4: Document Structure

Part I considers basic concepts of software development (*State of the Art*), before the then following part II contributes new concept ideas. Practical proof of their operability is given in part III. And part IV finally completes the work with a review, summary and outlook into the future.

Software Engineering Processes (SEP) (chapter 2) have to be briefly described to be able to estimate the effects of abstraction changes on the actual SEP phases. The *Physical Architecture* (chapter 3) of a standard *Information Technology* (IT) environment is necessary background knowledge for later reflections on the design of software systems and their communication paradigms. Finally, the *Logical Architecture* (chapter 4), that is conceptual solutions for structuring software systems, is investigated, to later be able to possibly find *Pros* and *Cons*.

A short *Recapitulation* of introduced state-of-the-art concepts and the idea of an *interdisciplinary, cybernetics-oriented* approach lead to an *Extended Motivation* (chapter 5) whose results and solutions are described in the remaining parts of the work.

A first description focuses on the distinction of *Statics and Dynamics* (chapter 6). In a second step, a new kind of *Knowledge Schema* gets introduced (chapter 7). Thirdly, *State and Logic* are described as to-be-separated knowledge models (chapter 8).

The application of the merged traditional and new design concepts results in the XML-based *Cybernetics Oriented Language* (CYBOL) (chapter 9). A corresponding *Cybernetics Oriented Interpreter* (CYBOI) (chapter 10) is needed to execute systems defined in that language. The *Res Medicinae* prototype application (chapter 11) is written in CYBOL and executed by CYBOI.

One might argue that chapters 9 (CYBOL) and 10 (CYBOI) should rather belong to part II, called *Contribution*, since they contain newly developed technologies. However, as they were needed for the practical proof, and in order to keep the chapter symmetry, they were placed in part III, called *Proof*.

After a *Review* validating and evaluating the CYBOP programming philosophy in comparison to the original motivation (chapter 12), a *Summary* and recommendations for *Future* research are given (chapter 13). The *Appendices* (chapter 14) contain used abbreviations, references to literature and the usual lists of figures and tables. A glossary was omitted since this document does not want to be a lexicon. All terms are explained at their first appearance in the text. A short history of thoughts that lead to the creation of this document and recommendations for a migration to CYBOL as well as some licences in full text follow. Caution! The page numbers behind an index entry at the end of this document refer to the *Beginning* of the section in which the entry appeared.

Part I

Basics

2 Software Engineering Process

The Way is the Aim.

CONFUCIUS

Software does not only contain and process information, it is information itself. Its creation, existence, growing old and death are called *Lifecycle*. Software stands at the end of a sequence of abstractions which is often called a *Software Engineering Process* (SEP). Besides the single steps of work and methodology to follow, a SEP often specifies the tools to be used and the roles of people involved [14]. Software development history has shown plenty of different forms of such processes, but most can be categorised into one of the following:

- Waterfall Process
- Iterative Process
- Agile Software Development
- Extreme Programming

This work is not exactly about software engineering processes, nor does it want to introduce yet another one. Its main purpose is to deal with the results of a SEP's phases: *Abstractions*. Three forms of abstraction are common to most processes:

- Requirements Analysis Document
- Architecture Design Diagrams
- Implementation Source Code

In order to have a common base of understanding and to be able to estimate the effects of abstraction changes on the actual software development phases, it is necessary to briefly describe some processes, which is done in the following sections.

2.1 Waterfall Process

The *Waterfall Process* (figure 2.1) is the classical way to develop a product. It assumes that the requirements are clear and do not change during a project. Waterfall software development is pretty straightforward and usually consists of the sequenced phases *Requirements*, *Analysis*, *Design*, *Implementation* (Realisation, Coding), *Test* and *Integration* (Release).

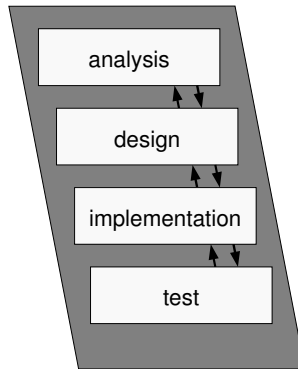


Figure 2.1: Waterfall Process with Back Flow

Numerous variations of waterfall processes exist. The simplest ones deliver their product at once, at the end of the project, what is often called *Big Bang Delivery* [205]. Others integrate some kind of *Back Flow* [302] that allows to consider test results in further development. One example that has combined software development- and testing activities is the *V-Modell* 97 [147] (figure 2.2). Its name stands for its shape: the left-hand (downhill) side of the *V* represents the development; the right-hand (uphill) side represents the corresponding test activities.

2.2 Iterative Process

An *Iterative Process* (figure 2.3) contains phases as known from the waterfall process, supplemented by the new idea of a *Reentrant Structure* (*Feedback Loop*). All phases are gone

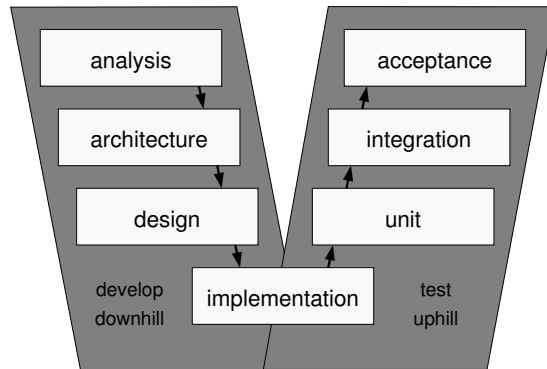


Figure 2.2: V-Model

through repeatedly, as long as the product is not satisfying. Whenever new requirements show up, also after completion, new features can be added to the system by reiterating a new project cycle.

Also here, many variations exist. They are called *incremental*, *evolutionary*, *staged*, *spiral* or *whirlpool*, or similarly. In the end, they all have their roots in some kind of *Iteration* which should *frequently produce working versions of the final system that have a subset of the required features*, as Fowler [98] writes.

A famous representative is the *Rational Unified Process* (RUP) [181]. Developed by Philippe Kruchten, Ivar Jacobson and others, RUP is the process complement to the *Unified Modeling Language* (UML). Its strength of being a process framework that can accommodate a wide variety of processes is its weakness, at the same time. Fowler [98] criticises this as follows:

As a result of this process framework mentality, RUP can be used in a very traditional waterfall style or in an agile manner (explained in section 2.3). So as a result you can use RUP as an agile process, or as a heavyweight process – it all depends on how you tailor it in your environment.

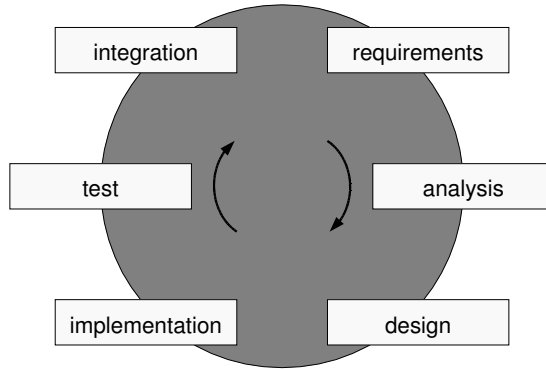


Figure 2.3: Iterative Process

2.3 Agile Methodologies

The principles of *Agile Methodologies* (figure 2.4) are applied by a group of so-called *lightweight, adaptive* software development processes with few bureaucracy, less predictability, less process- and document-orientation, but more emphasis on people and their skills, and on source code – which is considered the key part of documentation.

Besides *Extreme Programming* (XP) and *Open Source Software* (OSS) development, both described in section 2.4, there are several other methodologies that fit under the *Agile* banner. Fowler explains some of them in [98], which contains Alistair Cockburn’s *Crystal Family*, Jim Highsmith’s *Adaptive Software Development* (ASD), *Scrum*, *Feature Driven Development* (FDD) by Jeff De Luca and Peter Coad, the *Dynamic System Development Method* (DSDM) specified by a consortium of British companies and some remarks on *Context Driven Testing*.

For the purpose of this paper, further investigation on details of the mentioned methodologies is not needed. The general principles of agile software development (manifesto) are the important part to recognise, because they suggest a different, more *agile* approach to software engineering. Although many techniques of agile methodologies had been known and used for long, at least in OSS development, they had not been investigated, documented and promoted for business use in this form before. This is the great achievement of the *Agile Alliance* [4].

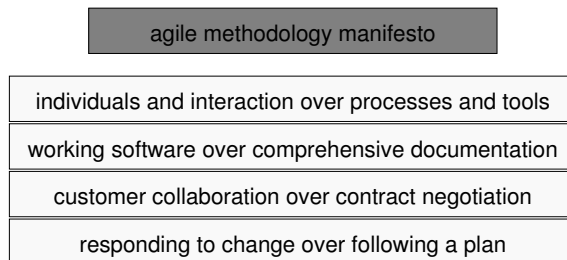


Figure 2.4: Agile Manifesto

2.4 Extreme Programming

Extreme Programming (XP) uses the idea of an iterative structure as explained in section 2.2, with the difference that it contains not only one but many cycles to assure sufficient feedback. The whole process can be cascaded and split into more fine-grained processes, for example *Iteration*, *Development* and *Collective Code Ownership*. Figure 2.5 shows a strongly simplified view of the XP methodology, with emphasis on its nested structure and multiple iterations. Better and more detailed overviews are given in [340].

In some way or the other, the classical process phases as first mentioned in section 2.1 also appear in XP, although they may have different names or a modified meaning. The requirements document, for example, is replaced by so-called *User Stories*, which are similar to usage scenarios (except that they are not limited to describing a user interface), but not to be mixed up with use cases [340]. Also, a number of new phases like *Release Planning* appear and more fine-granular activities like *Learn and Communicate* or *Stand Up Meeting* are added. The basis and starting point of each XP project are four common values:

- Communication
- Feedback
- Simplicity

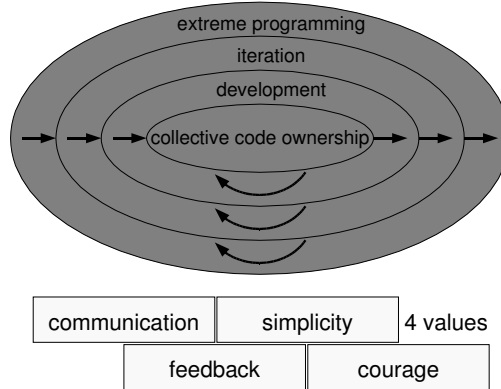


Figure 2.5: Extreme Programming (strongly simplified)

- Courage

The last decade has shown several pushes and increasingly greater support for *Free and Open Source Software* (FOSS) [242]. What makes this software so successful, besides the fact that its source code is open and freely available, is its astonishingly fast development model. Well, there surely are as many different development methodologies as there are FOSS projects out there, but many of them would probably, at least partly, match into XP. Additionally, however, there are a few significant differences that characterise *Open Source Development*. Among them are [98]:

- Collaboration between physically distributed teams
- Maintainer responsible for overall coordination and design
- Highly parallelisable debugging

In his book *The Cathedral and the Bazaar* [271], Eric S. Raymond provides further insights. Popular slogans taken from it are: *Release early and often!*, *Delegate everything you can!* or: *Be open to the point of promiscuity!* He recommends to foster a community of developers, lead by *Doing* and *Good Humour*. Yet Tim Churches reminds people not to take Raymond's recommendations as dogma [168]:

Although Eric S. Raymond's ... essay brought one particular FOSS development paradigm to a lot of people's attention, it may have also done the FOSS movement a disservice by making people think that the 'bazaar' approach is the only way in which FOSS can be developed.

Instead, each project should pick a methodology that best suits its needs, may it be cathedral- or bazaar-like.

2.5 Method Maturity

Numerous research efforts try to find the ideal software development paradigm and many academic papers were written on the topic. In order to be able to compare the resulting methodologies, a couple of which were described in the previous sections, some kind of measure is needed.

The *Capability Maturity Model for Software* (SW-CMM) [248] is such a measure. The newer CMM version is called *Capability Maturity Model Integration* (CMMI). Intended to help organisations improve the maturity of their software processes, it describes underlying principles and practices in terms of an evolutionary path [45]. The CMM is organised into five levels describing a software process' maturity:

1. *Initial*: ad hoc, occasionally even chaotic, scarcely defined
2. *Repeatable*: established discipline for repetition of earlier successes
3. *Defined*: documented, standardised activities for organisation
4. *Managed*: detailed quality measures, quantitative understanding
5. *Optimising*: continuous improvement through feedback

Two examples using the CMM for process evaluation are described in [284] which considers the *V-Model* and in [247] which investigates *XP*.

2.6 Abstraction Gaps

Software has to be developed in a creative process (methodology) called *Software Engineering Process* (SEP). As the previous sections tried to show, many different forms of such processes exist. Every project, consciously or not, follows a SEP that sooner-or-later, in one

form or the other, goes through the three common phases *Analysis*, *Design* and *Implementation* (figure 2.6). Each phase creates its own (ideally equivalent) model of what is to be abstracted in software and it is the differences in exactly these models that often, actually always cause complications.

The analysis mostly results in a *Requirements Document* which investigates the problem domain and uses expert knowledge to specify the functionality of the software to be created. This specification is mostly *informal*, that is an ordered collection of textual descriptions. Sometimes, *semi-formal* descriptions such as tables or graphics are used additionally.

It is the aim of the design phase to deliver a clear system architecture with little redundancies and only few interdependencies, which it may specify by help of *semi-formal Diagrams*. Recent years showed an increased use of the *Unified Modeling Language* (UML), a collection of diagram specifications for representing static or dynamic aspects of a system. Normally, a *top-down* approach is chosen for the design of a system. Hereby, the overall architecture is considered first, before moving into details. The less common *bottom-up* design would start the other way round and first try to build small components to construct the whole system from. A third possible approach, called *Yo-Yo* [41], would mix the two above-mentioned kinds.

Finally, implementation of a system is done *formally*, in (one or more) programming languages. The retrieved *Source Code* represents the temporally final abstraction, the software that was to be built.

It is obvious that at least two gaps (figure 2.6) have to be crossed when using the described phases:

1. Requirements Document – Architecture Diagrams
2. Architecture Diagrams – Source Code

Many efforts try to minimise the first gap by telling their analysis experts to specify use cases, workflows and static structures using the corresponding diagrams provided by the *Unified Modeling Language* (UML). Other efforts like the *Feature Modelling* that became especially popular in the area of *Product Line-/ System Family Engineering* [30] introduce an intermediate step of abstraction. The feature modelling itself is part of the analysis but can logically be placed between analysis and design. The results it delivers are called *Feature Models* [300, 246]. They provide hierarchical structures of the design properties of the system to be built. By applying feature models, the former big abstraction gap is broken

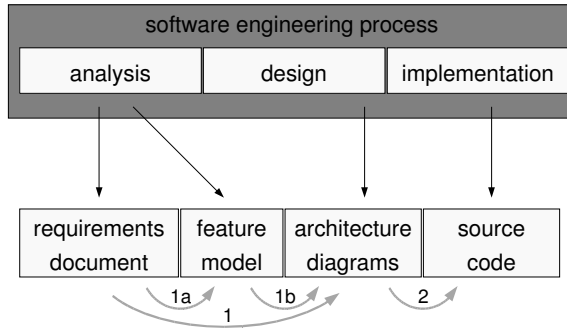


Figure 2.6: Abstraction Gaps

down into two smaller ones (figure 2.6), that are easier to cross:

1a Requirements Document – Feature Model

1b Feature Model – Architecture Diagrams

2 Architecture Diagrams – Source Code

That way, the *Traceability* between concrete requirements and architecture components can be improved. Moreover, the *Communication* between stakeholders in the development process can profit from feature models because of their closeness to both, analysis and design. Yet has the usage of feature models one disadvantage, too: another gap in abstraction is created through them.

The same happens in [107], where a special knowledge level called *Conceptual Ontology Representation*, comparable in its aims to the feature model, gets introduced as additional abstraction step. The aim of becoming more independent from implementation code for retrieving a human-readable form of knowledge, to improve communication between domain experts and engineers may well be reached, but sooner-or-later, also these models have to be transferred into program source code, by bridging the classical abstraction gaps mentioned above.

Bridging or closing these abstraction gaps (sometimes called *Semantic Gaps* or *Conceptual*

- *Module View*: represent the decomposition of a system into modules that are grouped in layers
- *Code View*: organise source code into object code, libraries and binaries, and into corresponding version files and directories
- *Execution View*: map software to hardware and distribute software components

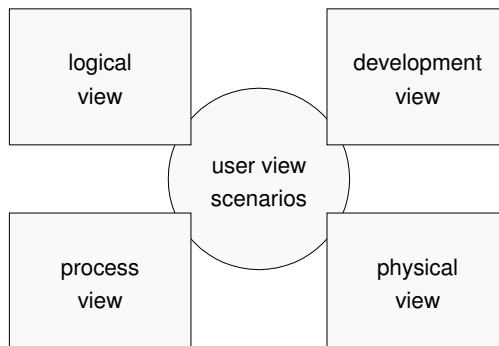


Figure 2.8: The 4+1 View Model of Architecture [182]

Architectures of systems that are implemented in an object-oriented way are better represented by the *4+1 View* model (figure 2.8), proposed by [182] and embraced as part of the *Rational Unified Process* (RUP) [181]. It separates static and dynamic aspects and consists of five different views, with the following purposes:

- *Logical (Design) View*: map required system functionality to architecture elements
- *Development (Implementation) View*: focus on the actual software module organisation
- *Physical (Deployment) View*: assign software elements to concrete hardware nodes
- *Process View*: describe dynamic runtime behavior of the executed system
- *Scenarios (Use Case +1 View)*: collect domain knowledge, from a user's view, and use them to validate and unify the other four views

Many other architecture modelling approaches like for instance the *Architecture Description Languages* (ADL), some representatives of which are described in [109, 53], exist but are outside the scope of this work and not elaborated further here, because the following two chapters were created according to the *4+1 View Model of Architecture*. Two simplifications are made, however: The *Physical-* also includes the *Process View* (chapter 3), because processes are considered as communicating systems running on physical machines, and the *Logical-* also contains the *Development View* (chapter 4), because logical models represent abstractions at different stages of software development. *Scenarios* are not considered since they belong to requirements engineering whose techniques are not a topic of this work.

3 Physical Architecture

Simplicity is the Result of Maturity.

JOHANN CHRISTOPH FRIEDRICH VON SCHILLER

Software provides the functionality through which robots act and computers represent and process information. Both are special kinds of machines which only get useful for humans if they can be controlled and communicated with. *Communication* is an essential ability for almost any kind of system. *Autonomous* systems exist and may well be useful, but it is nearly always the *Interaction* and *Cooperation* that makes technical systems (from now on called *Computer* in this work) so interesting and helpful to humans.

In many cases, systems are limited to one role: *Client* or *Server*. Clients ask questions which servers answer. But both are able to send as well as to receive information. One-way communication without any feedback is rarely useful. Besides the mentioned client- and server-, there are other roles that a computer system can take on when talking with so-called *Communication Partners*.

The following sections will stepwise build up- and briefly investigate some examples of well-known system constellations and possible communication languages that are commonly used in a general *Information Technology* (IT) environment. Because physical systems and their interactions are considered without any knowledge about their inside, one also talks of this as the *Physical Architecture* of an IT environment. Its understanding is important for later reflections on the inner architecture of software systems (chapter 4). Also will chapter 8 come back to system communication principles and introduce a translator architecture for universal communication.

3.1 Process

The most common word used to describe a running computer program is *Process*. Tanenbaum [304] defines it as an abstract model based on two independent concepts: *Resource Grouping* (space) and *Execution* (time).

He writes that *Resource Grouping* meant that a process had an address space containing program text and data, as well as other resources. A *Thread of Execution*, on the other hand, were the entity scheduled for execution on the *Central Processing Unit* (CPU). It had a program counter (keeping track of which instruction to execute next), registers (holding its current working variables) and a stack (containing the execution history, with one frame for each procedure called but not yet returned from). Although a thread would have to execute in some process, the thread and its process were different concepts and could be treated separately.

A slightly different explanation is given in [159]:

A thread is the path a program takes while it runs, the steps it performs, and the order in which it performs the steps. A thread runs code from its starting location in an ordered, predefined sequence for a given set of inputs. The term *Thread* is shorthand for *Thread of Control*. (One) can use multiple threads to improve application performance by running different application tasks simultaneously.

Abstract Concept	Explanation	Synonyms
Session	Bundle of processes of one user	
Process Group	Collection of one or more processes	Job
Process	Container for related Resources	System, Application, Task
Thread	Schedulable Entity	Lightweight Process

Table 3.1: Systematics of Abstract System Concepts

There are other abstract concepts which are of importance, especially in an *Operating System* (OS) context. A terminal in the *Linux* OS [167], for example, may control a *Session* consisting of *Process Groups* which in turn contain many *Processes* providing resources for the threads running in them. Table 3.1 shows one possible systematics of these concepts.

Some ambiguities exist, however. The term *Job* which, some decades ago, still stood for a program or set of programs, is nowadays used to label a process group in *Windows 2000*

[304, p. 7, 796] and similarly in *Linux* [167, p. 125, 237]. The notion of a *Task* is sometimes used equivalent to thread [67], but other times refers to a process or even process group [167, p. 113]. Additionally, some sources use the term in the meaning of a signal or event belonging to a work queue called *Task Farm* or *Task Bag* [305, p. 548, 606].

This document uses the more general word *System* to write about a process that manages the input, storage, processing and output of data in a computer. This is contrary to some other works which mean a whole computer, including its hardware and software programs running on it, when talking about systems. In the understanding of this work, once again, a *System* is a *Process* (software system) running on a *Computer* (hardware system).

3.2 Application Server

One well-known system, nowadays, is the *Application Server*. The name implies that this system is to *serve* other systems, so-called *Presentation Clients* (section 3.4). It may be programmed in languages like *Java*, *Python*, *Smalltalk*, *C++*, *C* or others more.

On the other hand, there are systems running all by themselves, without any access to/from another system – so-called *Standalone Systems*. In reality, they hardly exist since most applications run in a surrounding *Operating System* (OS) and are thus not really *alone*. An OS may be called *standalone* but mostly, even that consists of a number of sub processes solving background tasks. That is why the name *standalone* is used when one wants to place emphasis on the system itself, neglecting its communication with others.

Many kinds of application servers exist. Multiple services are offered by them, for example storage or persistence handling but also application- and domain specific functionality. A healthcare environment, as example, may contain several servers, each fulfilling one task such as person identification, resource access decision, image access and so on – just like people in real life have abilities and professions.

Systems of an IT environment are structured into so-called *Layers*, another name for which is *Tier*. The application server alone represents a *1 Tier* environment. The more systems of different type (presentation client, application server, database server) are added to an environment, the more tiers are added. For that reason, distributed client-server environments are called *n Tier*.

When people talk about a *Server*, they very often mean a *Computer* on which a *Server Process* is running. This is neither completely wrong nor absolutely correct. A computer

can run many different processes, only some of which may be servers. Hence, the computer can act as *Server* but also as *Client*, at the same time.

3.3 Database Server

Another popular kind of server system, besides the application server, is the *Database Server*, also called *Database Management System* (DBMS). It manages structured data called a *Database* (DB) and serves clients with *persistent* data. The arrow in figure 3.1 points in the direction into which the application server sends its queries to the database server, in order to retrieve data. Example DBMS representatives are *PostgreSQL*, *MySQL*, *DB2*, *ORACLE*, *ObjectStore*, *POET* or *Versant*.

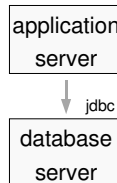


Figure 3.1: Database Server (2 Tiers)

Persistent Data are those that live longer than the system working on them. Very often, this is domain-specific- but also configuration information. These are stored in a filesystem or database [350]. *Transient Data*, on the other hand, is temporary information that a system holds during its lifetime, to function correctly. They get destroyed together with the system which created them.

Managing persistent data implies a number of quite complex tasks, the details of which will not be part of this document. To these aspects of database servers belong:

- Querying
- Transaction Handling
- Locking

Different types of database systems exist. The major ones are:

- *Hierarchical and Network DBMS*
- *Relational DBMS* (RDBMS)
- *Object-Relational DBMS* (ORDBMS)
- *Object-Oriented DBMS* (OODBMS)

Hierarchical DBMS were the first (electronic) databases ever used. They managed their data in tree structures, starting each access from the root node. Network DBMS went one step further: data could be associated at will [350, p. 128]. Relational DBMS are based on tabular data structures which can have relations. They were the first to accomplish a true separation between application and data. Special languages were created to define and query such data sources: The *Data Definition Language* (DDL) and the *Structured Query Language* (SQL). Object-Relational DBMS were to fill the semantic gap between *Object-Oriented Model* (OOM) and *Entity-Relationship Model* (ERM) structures. Their extensions introduced a number of user-defined data types. Object-Oriented DBMS conclusively close the semantic gap between object-oriented applications and data. Their programming interface is often integrated into a framework. The new SQL-based *Object Query Language* (OQL) [350, p. 138] was created for them.

The communication between systems can be eased with special techniques. After Tanenbaum [306], these were often called *Middleware* since they are placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems. In the case of database systems, one such mechanism is the *Java Database Connectivity* (JDBC) [121, 178]; another one the *Open Database Connectivity* (ODBC) [350, p. 170, 177]. They provide a common interface for many different relational databases.

Another technique are *Enterprise Java Beans* (EJB) and comparable mechanisms. They represent so-called *Business Objects* (BO) and hence actually belong to the previous section describing application servers. However, the containers in which EJBs live also contain functionality for persistence- and transaction handling which is why they are mentioned here. Further documentation can be found in the corresponding literature [119] and sources [29, 112].

3.4 Presentation Client

A system is called *Client* when it uses services of a server. Most modern applications incorporate abilities to communicate with server systems which may run on the same computer as the client or on a remote machine that has to be accessed over network.

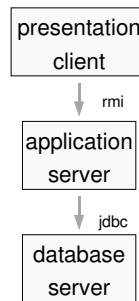


Figure 3.2: Presentation Client (3 Tiers)

But also clients can offer services as well as servers can use external services and such become clients themselves. The application server in figure 3.1 becomes a client when accessing the database system. As can be seen – *Client* and *Server* are quite arbitrary terms to characterise systems.

Figure 3.2 illustrates the communication between a presentation client and application server over network. Again, various mechanisms such as *Remote Method Invocation* (RMI), outside the Java world rather called *Remote Procedure Call* (RPC), exist to ease the way two remote systems talk with one another.

Frequently, people distinguish between *Thin Client* and *Fat Client* (the latter also called *Rich Client*) [350, p. 176]. While a thin client's task is nothing else than to display information coming from some server, a fat client also takes over part of the business data processing which is otherwise done by the server only.

3.5 Web Client and Server

With the emerge of the *Internet*, several new kinds of services like *Email*, *File Transfer*, *Web* etc. became popular. The web service allows information to be published in form of a *Web Page*. Web pages can be written in markup formats like *Hypertext Markup Language* (HTML) and *Extensible Markup Language* (XML) or, using special tags, as *Java Server Pages*- (JSP) and *PHP Hypertext Preprocessor*- (PHP) instructions. Before being displayed, the latter two need to be translated by a preprocessor inside the web server, into HTML.

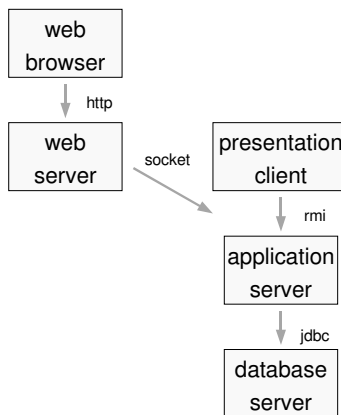


Figure 3.3: Web Client and Server

The principle as shown in figure 3.3 is easy: A *Web Server* stores web pages which can be accessed by clients called *Web Browser*. Browsers extract and translate (render) the (graphical) information given in form of a web page and display them. But they are also able to handle actions such as keyboard input or mouse click, and send these information back to the web server.

Moreover, browsers can locally execute small programs called *Applets* which were downloaded from the web server. Their counterpart are *Servlets* which are executed in multiple threads on the web server, offering the actual services.

Web communication is based on standards like the *Transfer Control Protocol/ Internet Protocol* (TCP/IP) and the *Hypertext Transfer Protocol* (HTTP) [303]. Section 3.11 will

systematise them together with other standards for system interconnection. The socket mechanism may be used to connect a web server to an application server.

Many other aspects are important when talking about internet services. There is the issue of security, there is performance, user-friendliness and many more which will not be discussed further here, since it would exceed the frame of this work.

3.6 Local Process

Not all software systems run on physically separated computers, also called *Nodes*. And not all communication happens over network. As well, one *Local Process* can talk to a second on the same machine (figure 3.4). In fact, all applications have this ability, at least for talking with the surrounding operating system.

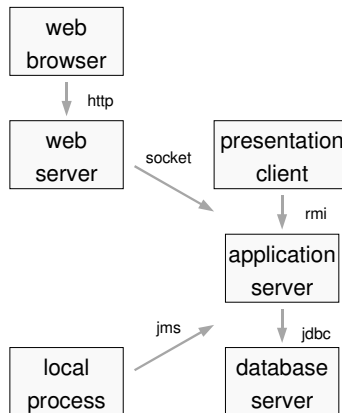


Figure 3.4: Local Process

Sometimes, local processes are needed by the operating system itself. Those are running in the background then which is why they are often called *Daemon*. Because they offer special services, daemons are nothing else than small servers. They fulfil tasks like managing all printing or email delivery of a system, or similar things [304, p. 74].

Very often, it is useful to let local client applications talk with each other. One part of a document (for instance a diagram) that was created by help of a special application

may want to get integrated into another document (for instance a letter) which is edited by another application. A number of mechanisms were created to solve this *Inter-Process Communication* (IPC) task, for example:

- *Dynamic Data Exchange* (DDE) [203]
- *Object Linking and Embedding* (OLE/ OLE2) and *ActiveX*, both now based on the *Component Object Model* (COM) [350, 119]
- *Java Message Service* (JMS) [112]
- *Desktop Communication Protocol* (DCOP) [81]
- *Bonobo* [96]
- *Pipes* [167, 304]

Although usually used for local communication (on the same node), some of these also function over network. Again, this document will not discuss their inside functionality. Plenty of books were written about that.

3.7 Human User

One system that needs special consideration is the *Human User*. In the first instance, it can be seen as normal system that is able to communicate with other humans but also with artificial software systems running on machines such as computers (figure 3.5).

At the second view, one realises that due to the difference in construction, human systems rely on other kinds of communication signals. While network cards are usually enough for two computers to exchange data, additional input/ output devices are needed to let human beings and computers talk to each other. To these devices count: *Keyboard*, *Mouse*, *Screen*, *Printer* and many more. They are made to suit the five human senses, that is to generate and understand optical, acoustical, mechanical and similar signals.

The optical information displayed on a screen is often systematised into character-based *Textual User Interface* (TUI) and window-based *Graphical User Interface* (GUI).

The scientific subject dealing with those issues in more detail is called *Human-Computer Interaction* (HCI). One working definition given in [133] states:

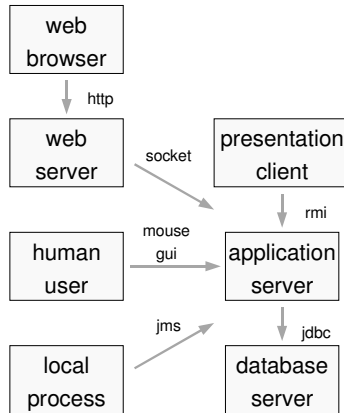


Figure 3.5: Human User

Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.

3.8 Peer Node

Tanenbaum and Steen [306] define a *Distributed System* as a collection of independent computers that appear to its users as a single coherent system. With *System* referring to a process rather than only hardware, as defined in section 3.1, it seems appropriate to rephrase and use this for the definition of a general *Distributed Computing Environment* (DCE):

A distributed computing environment consists of at least two systems that work together over a network but run on independent computer hardware (nodes).

Besides the previously mentioned client/ server (c/s) environments, so-called *Peer-to-Peer* (P2P) computer networks latterly became popular. In them, nodes do not have just one role, but act as client and server at the same time (figure 3.6), thus sharing their computing power and bandwidth. Common P2P protocols are: *Freenet*, *Gnutella2*, *BitTorrent*, *eDonkey*, *FastTrack* or *Napster* [60]. Many more exist.

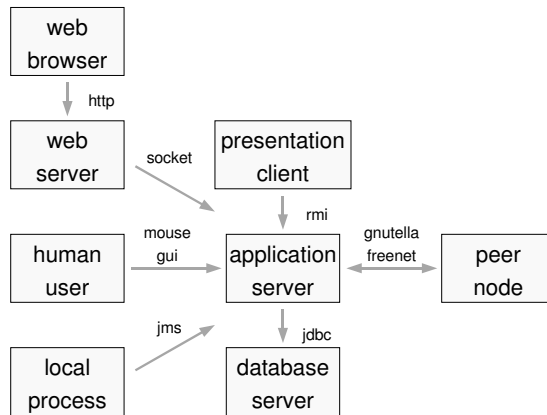


Figure 3.6: Peer-to-Peer Node Communication

Just like nodes in a P2P network, human beings are capable of communicating both ways, taking the role of a client or server. The organs that are needed to do so are put into comparison with the corresponding devices of a computer system, in chapter 8.

3.9 Remote Server

Figure 3.7 introduces a *Remote Server* to the illustrated example environment. It may access a database system – similarly to the already existing application server. In this example, however, it just works on simple local files, using *Streams*.

Like the previously introduced kinds of systems, remote systems need to rely on a number of standards and mechanisms, in order to be able to communicate over network. A comparison of some of these is given in [233, 37, 144]. In the following is a list of common techniques that were not yet mentioned before:

- *Common Object Request Broker Architecture* (CORBA) [234, 327, 119]
- *Simple Object Access Protocol* (SOAP) [331]
- *Network Dynamic Data Exchange* (NetDDE) [203]
- *Distributed Component Object Model* (DCOM/ COM+) [119]

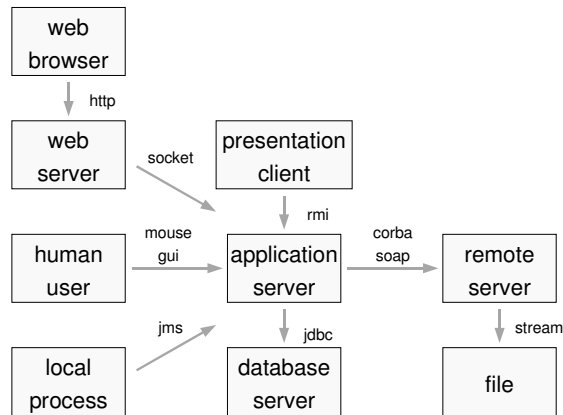


Figure 3.7: Remote Server

- *KParts* [81]
- *Universal Network Objects* (UNO) [239]

3.10 Legacy Host

Finally, there is often a need to integrate *Legacy Systems*, which are a special variant of remote software systems running on computers with an older architecture. Those computers are also named *Host*, as in the example of figure 3.8, or *Mainframe*. The applications running on them are programmed in languages like the *Common Business Oriented Language* (COBOL) or *Programming Language One* (PL/I) [89], the latter developed as an *International Business Machines* (IBM) [151] product in the mid 1960's.

Host computers manage nearly everything an ancient information technology environment needs. They are responsible for persistence and processing of data. Often, they contain hierarchical databases [124] using flat files like the *Virtual Storage Access Method* (VSAM) format. True clients do not exist here. Character-based terminals are the way to communicate with the host which controls all interaction (including keyboard and screen), within a *Third Party Maintenance* (TPM) *Customer Information Control System* (CICS) runtime environment.

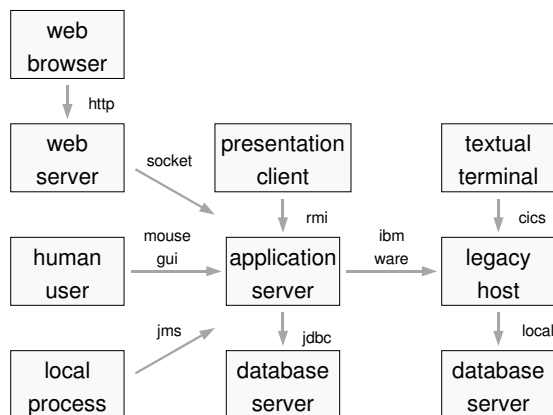


Figure 3.8: Legacy Host

3.11 Systems Interconnection

Communication is essential to an IT environment as described before. To enable and ease communication across different systems, special solutions have been developed and accepted as *de facto* or *de jure* standards. One such specification is the well-known *Open Systems Interconnection* (OSI) reference model, defined by the *International Organization for Standardization* (ISO). Numerous books [303] and documents on the web [249] describe this model and its protocols.

Figure 3.9 organises the seven layers of the model in table form, with one row representing one layer. The first column contains a layer's name, the second examples of typical network protocols and the third devices in which the protocols are used. *Simple Mail Transfer Protocol* (SMTP), *Telephone Network* (Telnet), *File Transfer Protocol* (FTP), *Hypertext Transfer Protocol* (HTTP) and *Domain Name Service* (DNS) are standard protocols used directly in software applications and -tools. *X.226* is a recommendation defining the OSI presentation protocol. The *Remote Procedure Call* (RPC) and *Network Basic Input/ Output System* (NetBIOS) may be sorted into the session layer. *Transfer Control Protocol* (TCP), *User Datagram Protocol* (UDP), *Transport Protocol Class 4* (TP4) and *Sequence Package Exchange* (SPX) do belong to the transport layer. The *Internet Protocol* (IP) is used in two versions: 4 and 6. Both of them are situated on the network level of the OSI model,

	layer	protocol	device
7	application	smtp, telnet, ftp, http, dns	gateway
6	presentation	x.226	gateway
5	session	rpc, netbios	gateway
4	transport	tcp, udp, tp4, spx	gateway
3	network	ipv4, ipv6, ipx	router
2	link	ppp, slip, fr	bridge, switch
1	physical	ethernet, token ring, fddi	repeater, hub

Figure 3.9: ISO OSI Reference Model

just like the *Internet Packet Exchange* (IPX) protocol. The link level contains the *Point-to-Point Protocol* (PPP), *Serial Line Internet Protocol* (SLIP) and *Frame Relay* (FR), the latter being a replacement for veterans like *X.25*. To the physical level transmitting raw Bits finally, belong *Ethernet*, *Token Ring* and *Fiber Distributed Data Interface* (FDDI).

Many of the mentioned protocols may be assigned to more than just one layer. But it is *not* the intention of this work to deal with such details. The overall ISO OSI model, however, is mentioned because it is a good example of a structure whose layers represent increasing levels of abstraction, what will later in this work be called an *Ontology* (chapters 4 and 7). Also, the *Health Level Seven* (HL7) medical standard, which gets introduced in chapter 11, received its name from referring to OSI's seventh level – the application level [276].

While the ISO OSI model defines seven abstract communication layers, the popular *TCP/IP* model uses solely four. Web communication as described in section 3.5 is based on it. Today, TCP/IP has become the standard in network management systems. A majority of them run the *Universal Interactive Executive* (UNIX) *Operating System* (OS), of which TCP/IP is an integral part. Margarete Payer [249] writes: *Although the OSI Model is affected with various deficiencies, it is well suitable for didactic purposes*. Further, she mentions that since some time, Andrew S. Tanenbaum uses a hybrid model for structuring his standard book on computer networks [303], which stucked to neither OSI nor TCP/IP.

3.12 Scalability

The previous sections demonstrated that there are many different ways to organise a distributed information technology environment. The physical distribution of systems is often a user requirement, either to connect different locations or to reach better performance by sharing the work load. The degree to which a system can be distributed to different hardware is often called its *Scalability*. Two models of scaling can be distinguished: *vertical* and *horizontal* computing (figure 3.10), whose key characteristics are only described briefly here.

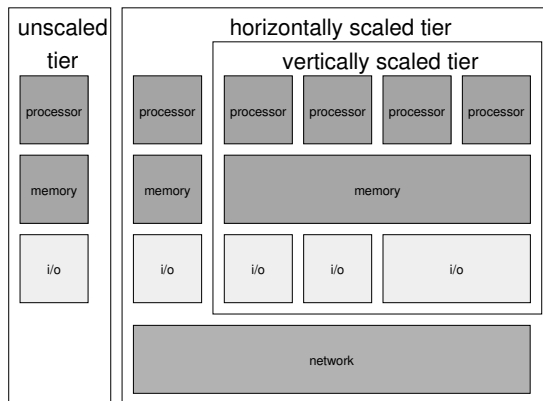


Figure 3.10: Vertical and Horizontal Scaling

Vertical servers are large *Symmetric Multiprocessing* (SMP) systems with more than four *Central Processing Units* (CPU) that share one common memory. One single *Operating System* (OS) instance covers the processors, the memory and input/ output (i/o) components. Vertical servers provide high availability by building numerous *Reliability, Availability, Serviceability* (RAS) features into the individual server, to minimise un-/planned downtime.

The alternative horizontal scaling connects many systems over network, which is often called *Clustering*. A cluster contains computing nodes having one to four processors and a memory each. The input/ output devices may belong to just one node or be shared by many. Each node has an OS instance. *Horizontal servers do not build RAS features into the individual servers but get high RAS by replication and deployment of many servers*, as Atwood [11] writes.

Vertical System	Horizontal System
Large Database	Web Server
Transactional Database	Firewall
Data Warehouse	Proxy Server
Data Mining	Directories
Application Server	Application Server
High Performance Technical Computing (HPTC) application (non-partitionable)	High Performance Technical Computing (HPTC) application (partitionable)
	Media Streaming
	Extensible Markup Language (XML) Processing
	Java Server Pages (JSP) Application
	Secure Socket Layer (SSL)
	Virtual Private Network (VPN)

Table 3.2: Vertical and Horizontal Application Types [11]

Table 3.2 states some typical applications for vertical and horizontal computing. The key difference, that after [11] affected both, their price and performance, is the *Interconnect* used with each architecture. Horizontal servers use a loosely-coupled *external* interconnect. Vertical servers use a tightly-coupled *internal* interconnect that makes data communications faster.

3.13 Misleading Tiers

When distinguishing human- and technical systems, three kinds of *Communication* (in this respect also called *Interfaces*) can be identified:

- Human \leftrightarrow Human
- Human \leftrightarrow Computer
- Computer \leftrightarrow Computer

Each of these relies on different communication techniques, transport mechanisms, languages (protocols) and so on. But the general principle after which communication works, is always the same – no matter whether technical *Computer* systems or their biological prototype, the *Human Being*, are considered: Information is *received*, *stored*, *processed* and *sent*. De-

spite these common characteristics, today's IT environments treat communication between a computer system and a human being differently than that *among* computer systems.

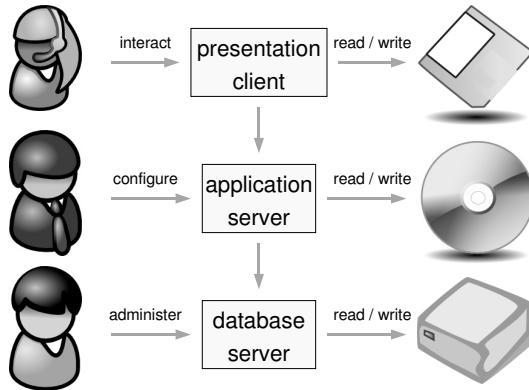


Figure 3.11: Universal Communication between Humans and Computers

Figure 3.11 shows a three-tier environment, as described in the previous sections: tier 1 represents the *Presentation Layer* (mostly scaled horizontally, using smaller servers); tier 2 stands for the *Application Layer* (where both, vertical and horizontal architectures are common); tier 3 is the *Database Layer* (dominated by vertical servers). Typical synonyms are, in this order: *Frontend*, *Business Logic* and *Backend*. The tiers (layers) serve two needs: connect different locations and share work load, as elaborated in section 3.12. However, the split into tiers of that kind is often misleadingly interpreted, since it raises two illusions:

1. *Users only interact with clients in the presentation layer:* Indeed, that layer was especially introduced for end-user communication but – systems of the other layers need to be controlled as well, by humans! Databases have to be administered; application servers configured.
2. *Persistent data are only stored in databases:* The majority of systems relies on some kind of locally available, persistent data. Even database management systems themselves use configuration files, for example.

Many IT architectures, or at least their illustrations, neglect the fact that in reality *all* systems need a *User Interface* (UI) and *almost* all systems store some of their persistent data

outside a database. This is not necessarily a problem for the physical IT environment as such, but it is for the internal architecture of software systems. Special solutions have to deal with frontend (UI framework), business logic (domain patterns) and backend (data mapping). Additionally, most modern systems contain several mechanisms that permit to communicate with other – local or remote – systems. The serious differences in these design solutions are one root of well-known problems like multi-directional inter-dependencies between system parts, that make software difficult to develop and hard to maintain.

One aim of this work is to investigate possibilities for a *unification* of communication paradigms, that is high-level design paradigms (like patterns) rather than low-level protocols, in order to architect software in a way that allows the computer systems it runs on to communicate *universally*. The following chapter therefore inspects the inner structure, also called *Logical Architecture*, of software systems as well as state-of-the-art techniques for its development.

4 Logical Architecture

*Because nothing is more difficult and
nothing requires more Personality,
than to be in open Opposition to current Time
and loudly to say: NO.*

KURT TUCHOLSKY

While the previous chapter had a look at the *Physical Architecture* of an IT environment, that is the systems and their communication, this chapter will discuss the *Logical Architecture*, that is the *Inside* of a software system.

The program source code of every system is – or at least should be – separated into logical parts like *Layers*, for example (figure 4.1). Current systems distinguish *Presentation*-, *Domain*- and *Data Source* layer [101]. Each of them contains functionality for a specific task: the presentation layer for user interaction; the domain for business logic; the data source for database communication.

Just like physical tiers can be scaled vertically and horizontally, the logical layers within a software system can be shared in a similar way. Figure 4.1 splits the horizontal business logic layer of a healthcare environment into the vertical domains *Documentation*, *Laboratory*, *Reporting*, *Billing*, *Administration*, *Imaging*, *Devices*.

One must not mix logical layers with the physical tiers that were introduced in chapter 3! It is true, logical layers may be distributed to physically separated systems – the presentation layer, for example, may be situated on the physical client tier (frontend). But as section 3.13 pointed out: In the end, *all* systems (not only the client tier) will have to interact with users and further systems in some way and thus cannot only implement one functionality but need to be able to communicate *universally*. More on that in part II.

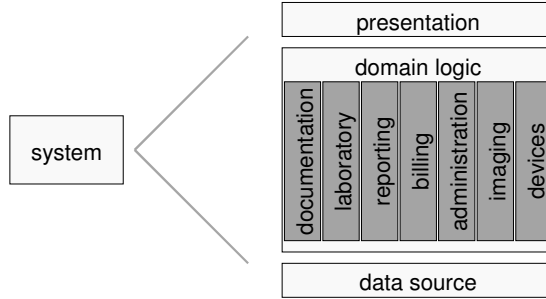


Figure 4.1: System with Logical Layers

Layers are just one concept aiming to improve a system's architecture. There are many more. The introduction of *Object Oriented Programming* (OOP) and the *Unified Modeling Language* (UML), for example, animated and enabled software developers to structure their program code more and more clearly. *Patterns*, *Frameworks*, *Components* and *Ontologies* are further techniques which delivered many new concepts and solutions. They all represent the state-of-the-art in software design and will be investigated together with their *Pros* and *Cons* in the following sections. The most general concepts, however, are still provided by computer languages and programming paradigms, which is why they are described first.

Over the years, several terms and synonyms describing architectural elements were introduced. Following are some examples, grouped arbitrarily into those that represent a kind of *State* and others that manipulate states according to certain rules of *Logic*. Both will be called *Statics*, later in this work (parts II and III). Besides these, there are terms for elements that describe the runtime behaviour of a system – its *Dynamics*, and others for some *Structural* elements. They all appear in one form or another in the following sections.

- Statics:

- State: Operand, Data, Value, Parameter, Attribute
- Logic: Operation, Operator, Function, Procedure, Method, Algorithm, Activity, Workflow

- Dynamics: Allocated Memory, Array, Instance, Object, Property, Process, Signal, Event, Action
- Structure: Class, Component, Module, Library, Package, Layer

4.1 Paradigm and Language

Manifold instructions exist that allow humans to program a computer. A set of such instructions is called *Programming Language* and is one of many groups of different *Computer Languages*. Other groups are for example *Markup Languages*, *Data Manipulation Languages* (DML), *Page Description Languages* or *Specification Languages* [60].

4.1.1 Language History

Just as a software engineering school advocates its very own *Methodology* (chapter 2), each programming language advocates a special *Programming Paradigm* [60] (sometimes also more than one). Some efforts categorise languages or their paradigms historically [297]. Eric Levenez' *Computer Languages Timeline* [193] captures common programming languages from a historical perspective. Some of them are shown in the simplified figure 4.2 (whose columns have no meaning). A much more comprehensive overview listing more than 2500 languages is given in the *Language List* [176] of Bill Kinneresley.

A lineage can be identified for every language, some popular of which are shown in the following list, the corresponding language name mentioned at first, being followed by the names of the language's ancestors. The right-most language represents the oldest ancestor. Only *one* lineage of arbitrary choice is considered for each language; most languages have *further* ancestors that are not mentioned here:

- *Java 2*: Java 1, Oak, Cedar, Mesa, Algol, IAL, Fortran
- *C#*: C++, C with Classes, C, B, BCPL, CPL, Algol
- *VB.NET*: Visual Basic, MS Basic, Basic, Algol
- *Delphi*: Object Pascal, Pascal, Algol
- *Oberon*: Modula, Pascal
- *Self*: Smalltalk, Simula, Algol

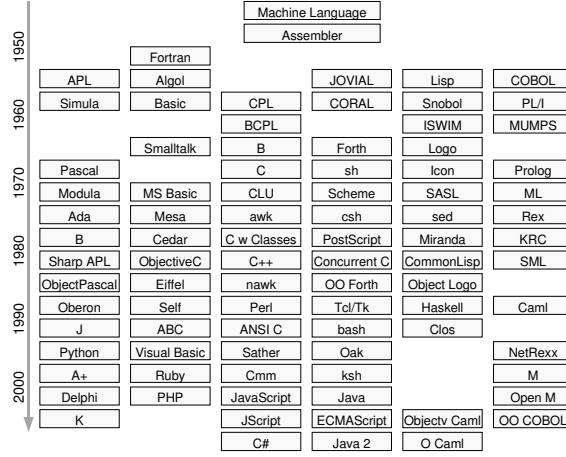


Figure 4.2: Programming Language History

- *Tcl/Tk*: Tcl
- *Python*: ABC, B
- *Perl*: nawk, awk, Icon, Snobol
- *PHP*: PHP/FI, Perl
- *Ruby*: CLU, Pascal
- *Haskell*: Miranda, KRC, SASL, ISWIM
- *O Caml*: Objective Caml, Caml, SML, ML
- *OO COBOL*: COBOL, Flow-Matic, B-O
- *NetRexx*: Object Rexx, Rexx, Rex, PL/1 ANS, PL/M, PL/I, COBOL
- *Open M*: M, MUMPS
- *Scheme*: Common Lisp, Lisp
- *PostScript*: OO Forth, Forth

Other historical approaches assign each programming language to a special *Generation*. Commonly used programming language generations and some of their representatives are shown following [60]:

- *First Generation Language*: machine-level language

- *Second Generation Language*: assembly language
- *Third Generation Language (3GL)*: Fortran, Algol, COBOL, Basic, C, C++
- *Fourth Generation Language (4GL)*: SQL, Mathematica, SAS, VB, MATLAB
- *Fifth Generation Language*: Prolog, OPS5, Mercury

4.1.2 Paradigm Overview

Several other systematics, besides the historical one shown in the previous section, exist to categorise programming languages and their paradigms. Some authors, for example, divide computer languages into those that have to be compiled before being executed and those which are interpreted at runtime. Figure 4.3 shows yet another arbitrary, tree-like systematics that was assembled on the basis of [60] and [176].

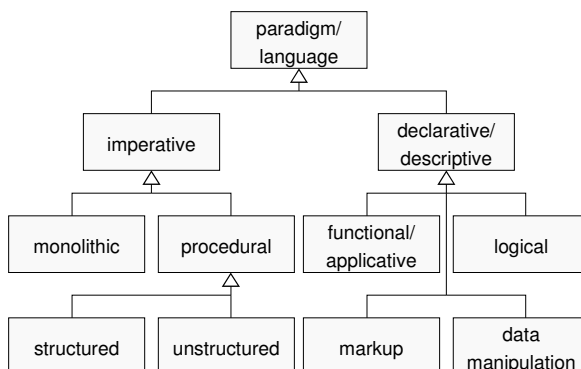


Figure 4.3: Programming Paradigm Systematics

Machine- and *Assembly Language* as well as *System Programming* are *imperative* (command-oriented). *Functional-* and *Logical Programming*, on the other hand, are *declarative*, just as most *Scripting Languages* used for *Typeless Programming*. The boundaries tend to be vague, however. Many of the new languages borrow features from more than one programming paradigm. Similarly, the concepts of *Structured* and *Procedural Programming* (SPP)

and *Object Oriented Programming* (OOP) are not only used in system programming-, but also in scripting languages.

It is important to note that it is extremely difficult, if not impossible, to arrange all programming languages into just one tree of categories. Kinnersley [176] writes that *for every classification scheme there will be a large proportion of languages that do not fit ... most languages are not purely one or the other*. The *Logo* language, for example, is an adaptation of the functional language *Lisp*, that is non-imperative, yet procedural [60]. Figure 4.3 can therefore only be seen as trial to create a systematics of the most common programming paradigms. In order to avoid miscategorisation, the Wikipedia Encyclopedia [60] prefers to list programming paradigms as contrasting pairs, for example:

- Procedural vs. Functional
- Imperative vs. Declarative
- Structured vs. Unstructured
- Value-level vs. Function-level
- Flow-driven vs. Event-driven
- Scalar vs. Array
- Class-based vs. Prototype-based
- Rule-based vs. Constraint

Not all items of the list are explained in this work since this would break its frame and focus. However, some of the most important programming language concepts in use today are described in the following sections.

4.1.3 Hardware Architecture

In his very clear book, Tanenbaum [305] organises instructions in abstract *Levels* (figure 4.4), which he also calls *Virtual Machines* (VM), since each level could be seen as hypothetic computer with an own language. Further on, he considers hardware and software to be *logically equivalent* because one could replace the other.

The next sub sections are based on this structure. They describe lower levels, close to hardware. Later sections then place more emphasis on concepts introduced by higher-level *Problem Oriented Languages* (POL).

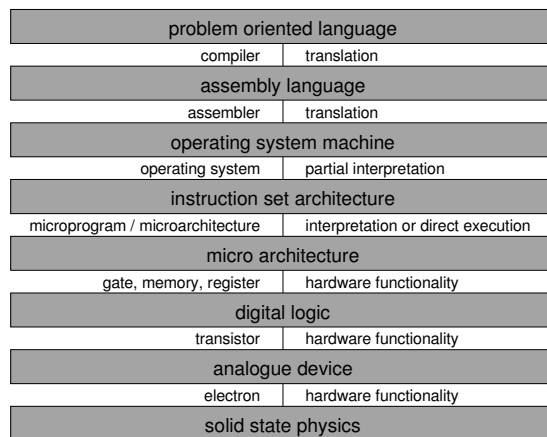


Figure 4.4: Computer Structure (adapted from [305])

Digital Logic

As mentioned in chapter 1, it is *Abstractions* that enable humans to capture the surrounding real world in a simplified way. *All* information is abstract. Software is information and the data it processes are information, too.

With the emerge of *Digital Computers*, *Digital Logic* gained more importance and the new field of science *Informatics* was born whose job in essence is to break down (abstract) every piece of information to just two states: *0* and *1*, represented by one *Binary Digit* (Bit). This is accomplished through the use of digital electronic components called *Gate* which transfer one or more input signals (states) into a defined output signal by applying simple logic functions like *AND* or *OR*. Many gates can form a 1-Bit *Memory* that is able to store the states *0* or *1*. Memories can be grouped so to form *Registers* [130] which are able to store one or many Bits.

Internally, gates consist of analogue electronic devices like *Transistors*, the functionality of which is out of the scope of this document. Any other details of what is going on inside analogue electronic components belong to the field of *Solid State Physics*.

One might ask why exactly *0* and *1* and no other states (for example *0.1*, *0.2* etc.) between them were chosen. The answer needs some background information. When talking about a

Signal in hardware computer science, people mean electric voltage. *Zero* and *One* correspond to *Low* and *High* voltage in electronic circuits. These minimum and maximum values of voltage are reached in rarest cases – mostly, the voltage lies somewhere between. This is due to environmental influences called *Noise* which pollute a signal (voltage). Therefore, each signal has to be interpreted as being rather high or low. The better the *Signal to Noise Ratio* (SNR), the more exact this interpretation can be.

With only two possible states, interpretation failures are very rare and digital technique has already proven to be quite error-tolerant. How much more difficult would it be to guess a signal's state if there were four, ten or more! That is why breaking down all information to only *High* and *Low* (also labeled *True* and *False* or *On* and *Off*) provides the most reliable abstraction.

There are efforts to develop *Quantum Computers* that use *Qubits* to measure data. While a traditional Bit represents just one state, that is either *zero* or *one*, a Qubit can hold a *zero*, or a *one*, or a superposition of these and represent more than one state, at one time instant. Qubits can be implemented using elementary particles with two spin states, for example represented by *Quarks*. Quantum computers are believed to solve certain problems faster than any classical computer [60]. However, this is the future of computing and not part of this work.

Micro Architecture

The *Micro Architecture* level contains a number of memories and the so called *Arithmetic Logic Unit* (ALU) which is an *Integrated Circuit* (IC) that is able to execute simple arithmetic operations. The arithmetic logic unit and registers exchange data across the *Data Path*. The data path is controlled either directly by hardware or by a special *Micro Program* which interprets instructions from the next higher *Instruction Set Architecture* (ISA) level.

Instruction Set Architecture

The *Instruction Set Architecture* (ISA) essentially summarises the instructions that can be carried out by the micro architecture hardware (or interpreted by its micro program software). Computer manufacturers usually publish a handbook describing the whole set of instructions.

4.1.4 Machine Language

The new features in this level (for example memory organisation or parallel execution of programs) are normally provided by an interpreter program that is running on the lower instruction set architecture level. This is the *Operating System* (OS).

Instructions which are identical to those of the instruction set architecture, however, are executed directly by the yet lower micro architecture/ micro program level, and not by the operating system. Therefore, the *Machine Language* level is also called *hybrid*.

4.1.5 Assembly Language

Languages of the layers described to here are numeric. That is, programs written in them consist of long numerical series adapted to what a machine expects. Starting with the level of *Assembly Language*, programs contain special *Keywords*, symbols and abbreviations which are meaningful to humans. While programs of the former levels are written by *System Programmers*, it is *Application Programmers* who use assembly- and higher-level languages to write a program.

Instructions of lower levels are always interpreted. The corresponding program is called *Interpreter*. It is running on the level below the one the instructions stem from. An interpreter executes an instruction directly, without generating a translated program. Higher-level languages, on the other hand, get translated into lower-level instructions before being executed. Such translator programs are called *Assembler* or *Compiler*. New forms of programs (like those written in Java) also use a combination of both, being first compiled into a special byte code and then interpreted at runtime.

4.1.6 Structured- and Procedural Programming

Computer history has produced a whole plethora of high-level languages (an overview is given in section 4.1.1). They are to ease the programming of applications which solve problems of an arbitrary domain. Nearly all of them make use of a number of techniques that stem from the so-called *Structured- and Procedural Programming* (SPP).

These techniques arise from firstly the reduction of *Control Structures* to a minimal set of elements which can be combined arbitrarily in *Sequenced Steps*. Secondly, repeating algorithms can be defined as *Procedure* and called as subroutine. That way, wild *Jumps*

from one part of a program to another are avoided. A procedure can also call itself which is known as *Recursion* [250].

It is possible to *hierarchically modularise* all control structures, with each structure having a defined *Entrance* and *Exit*. When procedures are grouped together in a separate file, then this file is often called *Module* or *Library*. Modules can contribute greatly to the reuse and creation of clear program code.

Two kinds of diagrams are typically used to describe a (part of a) procedural program semi-formally: *Program Flow Chart* and *Structure Chart*. Both representations are based on sequences of control structures. The former differs from the latter in the existence and appearance of certain graphical elements; *GoTo* instructions, for example, do not exist in structure charts.

Following is a brief description of the most important control elements of SPP, given in form of both, diagrams [280] and C program code [309]. These basic control techniques are: *Assignment*, *Branching* and *Looping*.

Assignment

A *Statement* (figure 4.5) is a sequence of operators and operands [106], to be evaluated (executed) by (the next lower abstraction level of) a computer. It is also called an *Expression*. The *Operator* represents the actual *Operation*, an active instruction to the computer. It uses and works on passive data – the *Operands*, also called *Variables*. Following a statement in C code:

```
operand++;
```

A *Variable* is a placeholder for an abstracted *Data Value*. It occupies space in memory which is why this space has to be reserved before it can be used. The reservation is called *Allocation* or *Declaration* and it states the variable's *Type* and an *Identifier*. Commonly, variables also get initialised through the *Assignment* of an *Initial Value*. Here an example for declaration and initialisation through assignment in C code:

```
type identifier = value;
```

Many statements which belong together can form a *Block*, also called *Compound Statement*. Variables declared in a block are called its *Local Variables* and lose their validity outside

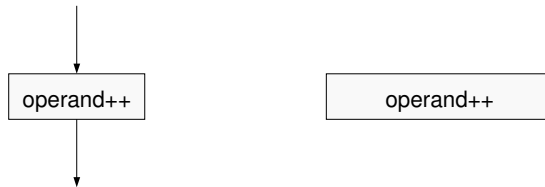


Figure 4.5: Statement as Program Flow Chart and Structure Chart

that block. Blocks have an opening and a closing symbol. Following once more an example in *C* programming language source code, showing a block with two statements:

```
{
    statement1;
    statement2;
}
```

Branching

A block of statements that get only executed at special occasions is called a *Branch*. Two kinds of branching exist: *Conditional Branching* and *Unconditional Branching*. An implementation of the latter is the well-known but also disliked *goto* (*jump*) command. The former depends on a *Condition*, also called *Alternative* or *Choice* (figure 4.6), that is its statements are only executed if the condition's result is true. That way, a condition can change the flow of a program. A code example follows; it shows conditional branching:

```
if (condition) {
    statements;
} else {
    statements;
}
```

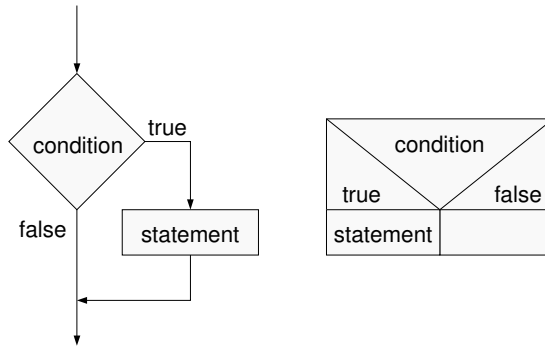


Figure 4.6: Condition as Program Flow Chart and Structure Chart

Many programming languages offer a *Multiple Condition* control structure like *switch* or *case*. It is a comfortable possibility to let a program make a choice out of many alternatives:

```
switch (condition) {
    case constant1:
        statements;
    case constant2:
        statements;
    default:
        statements;
}
```

Essentially, however, it is a subsumption of a number of simple conditions which are mostly called *if-else*, and therefore replaceable by such, as shown following:

```
if (condition == constant1) {
    statements;
} else if (condition == constant2) {
    statements;
} else {
    statements;
}
```

The multiple condition is conceptually no innovation in comparison with the simple condition and hence pure convenience for the programmer. The interpreter described in chapter 10 uses solely if-then statements.

Looping

The *Loop* (figure 4.7) is a control element that allows to iterate through statements, in other words to execute them repeatedly, several times. Its concept is quite simple – a jump backwards in the program. However, this low-level jump is hidden to the application programmer using a higher-level SPP language. The loop is indicated by a special keyword instead, for example:

```
while (condition) {  
    statements;  
}
```

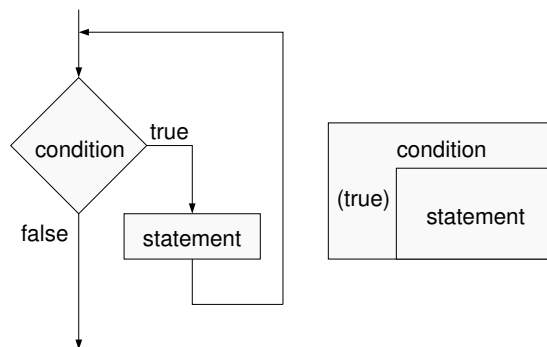


Figure 4.7: Loop as Program Flow Chart and Structure Chart

Most programming languages offer three different loop styles, as there are:

- Pre-test loop: *while*, *while-do*
- Post-test loop: *do-while*, *repeat-until*

- Counting Loop: *for*, *for-next*

A *Pre-Test Loop* is used when one wants to check a condition before the statements in the loop body are executed:

```
int i = 0;
while (i < 1) {
    statements;
    i++;
}
```

The *Post-Test Loop*, on the other hand, repeats all loop-body statements until a condition is met:

```
int i = 0;
do {
    statements;
    i++;
} while (i < 1);
```

A *Counting Loop*, finally, can be applied when the number of necessary repetitions of the loop-body statements is known in advance:

```
int i;
for (i = 0; i < 1; i++) {
    statements;
}
```

The statements in all three loop examples are only executed once. It is not difficult to see that the *for* loop can be replaced with a *while* loop by initialising the *i* variable in its declaration line and moving the increment statement into the loop's block. But also the *do-while* loop can be replaced with a *while* loop. If the behaviour does not match (for example a while block is not executed even once), then changing the initial loop variable value can solve this problem. Otherwise, modifying the statements (algorithm) in the block, without changing it logically, will do.

As can be seen: Most variations of the *Looping* concept are just a convenience for the programmer. They are conceptually identical and can be lead back to a simple loop with break condition, each. The interpreter described in chapter 10 uses just one kind of loop.

4.1.7 System Programming

After John K. Ousterhout [244], *System Programming Languages* such as *PL/1*, *Pascal*, *C* or *C++* or *Java* (which evolved from higher level languages such as *LISP*, *Fortran* or *Algol* – see section 4.1.1) had been introduced as an alternative to *Assembly Languages* and both would differ in two ways. While in an assembly language, virtually every aspect of a machine were reflected in the program, each statement representing a single machine instruction so that programmers had to deal with low-level details such as register allocation and procedure calling sequences, a system programming language were:

1. *higher level* because its statements did not correspond exactly to machine instructions; a compiler would translate each statement in the source program into a sequence of binary instructions and handle register allocation;
2. *strongly typed* because programmers needed to declare how each piece of information would be used; the language would prevent the information from being used in any other way.

Ousterhout uses the term *Typing* to: *refer to the degree to which the meaning of information is specified in advance of its use*. After him, the strong typing (also called *Static Typing*) of today's system programming languages had several advantages, such as:

- Better manageability of large programs by differentiating between things that must be treated differently
- Possible error detection by using type information in compilers
- Improved performance by allowing compilers to generate specialized code

But there were also a number of disadvantages when using system programming languages:

- Need to declare each variable with a particular type and to use it in ways that are appropriate for the type
- Difficulty to create new code on the fly due to total segregation of data and code
- Impossibility to use an object of one type where an object of a different type is expected, because variables are collected in objects with well-defined substructure and procedures to manipulate them

4.1.8 Typeless Programming

Scripting Languages (formerly also called *Job Control-* or *Batch Languages* [60]) such as Perl [334], Python [204], Rexx [232], Tcl [243], *Visual Basic* (VB) and the *Universal Interactive Executive* (UNIX) shells overcome the disadvantages of system programming languages (section 4.1.7) by being *typeless* and *interpreted*. The paradigm behind them is sometimes called *Script Oriented Programming* (SOP).

Ousterhout [244] writes that modern computers were fundamentally typeless: any word in memory could hold any kind of value, such as an integer, a floating-point number, a pointer, or an instruction. The meaning of a value were determined by how it is used: if the program counter pointed at a word of memory then it were treated as an instruction; if a word were referenced by an integer *add* instruction then it were treated as an integer; and so on. The same word could be used in different ways at different times (what is also called *Dynamic Typing*).

Because scripting languages are intended primarily for plugging together existing components, they are also referred to as *System Integration Languages* or *Glue Languages*. They provide a higher level of programming than assembly- or system programming languages. Through their usage, integrated applications, after [244], could be developed five to ten times faster than with system programming languages. Scripting languages sacrifice execution speed to improve development speed.

The interpreter described in chapter 10 is able to handle data (knowledge) without knowing about their type (kind of abstraction) in advance, that is at compilation time. Although itself written in the system programming language *C*, the interpreter is very flexible when it comes to processing knowledge.

4.1.9 Functional Programming

Many languages such as *Lisp* and its relatives cannot be characterised cleanly as system programming language or scripting language; they are situated somewhere between. Concepts like *Interpretation* and *Dynamic Typing*, now common in scripting languages, stem from Lisp [244]. Others like *Automatic Storage Management* and *Integrated Development Environments*, now used in both scripting- and system programming languages, were introduced by Lisp as well. Peter Norvig writes in [223]:

There is a myth that Lisp (and Prolog) are *Special Purpose Languages* (SPL), while languages such as Pascal and C are *General Purpose* (GPL). Actually, just the reverse is true. Pascal and C are special-purpose languages for manipulating the registers and memory of a *von Neumann*-style computer. The majority of their syntax is devoted to arithmetic and Boolean expressions, and while they provide some facilities for forming data structures, they have poor mechanisms for procedural abstraction or control abstraction. In addition, they are designed for the *state-oriented* style of programming: computing a result by changing the value of variables through assignment statements.

The *Frequently Asked Questions* (FAQ) edited by Graham Hutton [146] distinguish between *Imperative Languages* and *Functional Languages*. System programming languages as introduced in previous sections belong to the first group. To calculate the sum of the integers from 1 to 10, for example, they would probably use a simple loop and repeatedly update the values held in an accumulator variable *total* and a counter variable *i*:

```
int total = 0;
for (int i = 1; i <= 10; ++i) {
    total += i;
}
```

A functional language like *Haskell* would express the same program *without* any variable updates, by evaluating an expression, as shown below. Variable updates, that is *computational effects caused by expression evaluation that persist after the evaluation is completed* [146] are called *Side Effects*.

```
sum [1..10]
```

The following two examples [146] show the same program in two other functional languages, namely *SML* and *Scheme*:

```
let fun sum i tot = if i = 0 then tot else sum (i - 1) (tot + i)
in sum 10 0
end
```

```
(define sum
(lambda (from total)
  (if (= 0 from)
      total
      (sum (- from 1) (+ total from)))))
```

```
(sum 10 0)
```

The *Association of Lisp Users* [227] points out the absence of side effects and explains *Functional Programming* as follows:

Functional programming describes all computer operations as mathematical functions on inputs. Typically, a function can be created and returned from other functions as first-class data. This function object may then be passed as input to other functions, perhaps be composed with other functions, and eventually, applied to inputs to produce a value. Objects can be defined in terms of functions that encapsulate certain data, and operations on objects can be defined by functions encapsulating the objects. Purely functional languages do not have assignment, as all side-effecting can be defined in terms of functions that encapsulate the changed data. Procedural languages essentially perform everything as side-effects to data structures. A purely procedural language would have no functions, but might have subroutines of no arguments that returned no values, and performed certain assignments and other operations based on the data it found stored in the system.

The interpreter described in chapter 10 manipulates *all* data (knowledge) as if it would be one huge side effect. Data (knowledge models) are not bundled with-, but kept completely outside any functions/ procedures. Only references to these data are handed over as parameters.

4.1.10 Logical Programming

Functional Programming as introduced in the previous section is one kind of *Declarative Programming*, which describes to the computer a set of conditions and lets the computer figure out how to satisfy them [60]. Another kind is *Logical Programming*. It specifies a set of attributes that a solution should have – rather than a set of steps to obtain such a solution. Schematically, the logical programming process follows the equation:

```
facts + rules = results
```

Logical programming was strongly influenced by *Artificial Intelligence* (AI) and is applied in domains such as *Expert Systems*, where the program generates a recommendation or answer

from a large model of the application domain, and *Automated Theorem Proving*, where the program generates novel theorems to extend some existing body of theory. [60]

The *Monkey and Banana Problem* is a famous example studied in the community of logical programming [60]: *Instead of the programmer explicitly specifying the path for the monkey to reach the banana, the computer actually reasons out a possible way that the monkey reaches the banana.*

A prominent logical language representative is *Prolog*; a more recent one is *Mercury*; an *Open Source Software* (OSS) one is *TyRuBa*.

4.1.11 Data Manipulation Language

A *Data Manipulation Language* (DML), after [60], is: *a family of computer languages used by computer programs or database users to retrieve, insert, delete and update data in a database.* As most popular DML, the source mentions the *Structured Query Language* (SQL) that was originally developed as *Structured English Query Language* (SEQUEL) by *International Business Machines* (IBM), after the model described by Edgar F. Codd in [55].

Technically, SQL is a set-based, declarative computer language that, after [60], could be used to create, modify and retrieve data from *Relational Database Management Systems* (RDBMS). Its keywords are often shared into the three groups:

- *Data Manipulation Language* (DML): SELECT, INSERT, UPDATE, DELETE
- *Data Definition Language* (DDL): CREATE, DROP
- *Data Control Language* (DCL): GRANT, REVOKE

Since the details of that language are outside the scope of this work, they are not elaborated further here, but can be learned at for example [282].

4.1.12 Markup Language

At latest with the distribution of the *World Wide Web* (WWW), *Markup Languages* increasingly gained in popularity. A markup language separates the presentation *Style* of a document from its logical *Structure* and *Content*. Well-known representatives of markup languages, two famous of which being described in the following sections, are [60]:

- *T_EX* / *Lamport T_EX* (*L^AT_EX*, *L^AT_EX 2_ε*)
- *Scribe*
- *Standard Generalized Markup Language* (SGML)
- *Extensible Markup Language* (XML)
- *Hypertext Markup Language* (HTML)
- *DocBook*
- *Text Encoding Initiative* (TEI)

Recently, more and more projects appear that try to use markup languages not just for document markup, but also for declarative programming. Before coming to the actual markup languages, the paradigm of *Literate Programming* and its idea to use markup tokens for distinguishing source code and documentation, is investigated.

Literate Programming

Ross Williams writes in [343, section 1.1]:

A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments which describe the various parts of the code. In *Literate Programming*, the emphasis is reversed. Instead of writing code containing documentation, the literate programmer writes documentation containing code.

In other words, *Literate Programming* pays more attention to proper source code documentation than classical programming languages do. It mostly offers special *Token* characters like the *Commercial At* character @ for example, which serve as code delimiters. The delimited blocks are determined by particular tools such as a preprocessor that filters out program code to be processed further. All source information together (input document, commentaries, program code) is used to generate typeset documentation files in one or more formats.

Williams [343] means that the literate programming system provided far more than: *just a reversal of the priority of comments and code*. In its full-blown form, a good literate programming facility could provide:

- *Re-ordering of Code*: Some programming languages force the programmer to give the various program parts in a particular order.
- *Typeset Code and Documentation*: Because a literate programming utility sees all the code, it can use its knowledge of the programming language and the features of the typesetting language to typeset the program code as if it were appearing in a technical journal.
- *Cross referencing*: Because a literate tool sees all the code and documentation, it is able to generate extensive cross referencing information in the typeset documentation, which makes the printed program document more easy to navigate and partially compensates for the lack of an automatic searching facility when reading printed documentation.

It is true, the actual instructions and algorithms in between commentaries are written in (or translated into) a system programming- or other kind of language. But literate programming places its focus on source code *Documentation* for which it uses *Markup* tokens, which is why it was classified under *Markup Language* in this work.

Although literate programming itself has not gained that much popularity, its idea of using markup tokens to generate more expressive source code documentation has. Several up-to-date programming environments make use of it. A well-known example is the *JavaDoc* tool [154]; other systems are *Doxygen* [324] or *DOC++* [1].

TeX and LaTeX

The special-purpose T_EX [179] language is the centre-piece of a typesetting system which, due to its well-formatted output of complex mathematical formulas and generally high-quality typesetting, is especially popular among academic circles of mathematicians, physicists and computer scientists [316]. The Wikipedia encyclopedia [60] writes:

T_EX is a macro and token based language: many commands, including most user-defined ones, are expanded on the fly until only unexpandable tokens remain which get executed. Expansion itself is practically side-effect free. Tail recursion of macros takes no memory, and if-then-else constructs are available.
...

The T_EX system has precise knowledge of the sizes of all characters and symbols, and using this information, it computes the optimal arrangement of letters per

line and lines per page. It then produces a *Device Independent* (DVI) file containing the final locations of all characters. The DVI file can be printed directly given an appropriate printer driver, or it can be converted to other formats.

The *PDFTeX* translator program is often used to bypass all DVI generation, by creating *Portable Document Format* (PDF) files directly.

Nowadays, $\text{T}_{\text{E}}\text{X}$ is mostly used with a template extension called *Lamport $\text{T}_{\text{E}}\text{X}$* ($\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$) [188]. The Indian *$\text{T}_{\text{E}}\text{X}$ Users Group* (TUG) writes [316]: *L^AT_EX is a document preparation system which adds a set of functions that make the $\text{T}_{\text{E}}\text{X}$ language friendlier than using the primitives provided by it.* It offers programmable *Desktop Publishing* (DTP) features and extensive facilities for automating most aspects of typesetting. [60]

The most important feature of $\text{T}_{\text{E}}\text{X}$ for this work is its kind of relating meta- with structural information. Two examples may help here:

```
\documentclass[a4paper,12pt]{book}
\includegraphics[scale=0.3]{path/file.pdf}
```

The first statement determines *book* as document class for a document to be written. It contains additional information such as paper- and font size, in square brackets. The second statement refers to a graphics file to be included. The additional information given in square brackets here is the scale factor. With a different syntax, but in a comparable manner, the knowledge modelling language introduced in chapter 9 does relate structural- with meta information.

Extensible Markup Language

A popular, very flexible, yet simple language playing an increasingly important role in the exchange of a wide variety of data on the *World Wide Web* (WWW) and elsewhere is the *Extensible Markup Language* (XML) [345], defined by the *World Wide Web Consortium* (W3C) [330]. Being a text format derived as simplified subset (dialect) of the *Standard Generalized Markup Language* (SGML) [160], it allows to structure and store information hierarchically as *Document* file. Norman Walsh [335] writes:

A markup language is a mechanism to *identify structures* in a document. The XML specification defines a standard way to *add markup* to documents.

And the XML Cover Pages [142] state:

Both SGML and XML are *meta* languages because they are used for defining *markup* languages. A markup language defined using SGML or XML has a specific vocabulary (labels for elements and attributes) and a declared syntax (grammar defining the hierarchy and other features).

Historically, markup languages became widely known through the *Hypertext Markup Language* (HTML) as language of the Web. To overcome its limitations, XML was originally *designed to meet the challenges of large- scale electronic publishing* [345]. Today, XML is applied in many different areas, for example:

- Document Publishing [336]
- Data Transfer [331]
- GUI Design [262, 57, 292, 273]
- Workflow Composition [136, 161]
- Database Storage [210, 70]
- Domain Modelling [295, 114]

Yet in the opinion of Robin Cover [65], the usability of XML for domain modelling is limited. He writes:

Just like its parent metalanguage (SGML), XML has no formal mechanism to support the declaration of semantic integrity constraints, and XML processors have no means of validating object semantics even if these are declared informally in an XML DTD. XML processors will have no inherent understanding of document object semantics because XML (meta-)markup languages have no predefined application-level processing semantics. XML thus formally governs syntax only – not semantics.

In fact, XML syntax is designed for representing an encoded serialization, and thus has a very limited range of expression for modeling complex object semantics, where *Semantics* fundamentally means an intricate web of constrained relationships and properties. Otherwise stated: XML is a poor language for data modelling . . .

The XML-based language described in chapter 9 proves the opposite. By applying a common knowledge modelling schema (chapter 7), it allows to model arbitrary meta information (complex object semantics). Robin Cover continues [65]:

The notion of *Attribute* might have been more useful except that XML supports only a flat data model for the value of an attribute in a name-value pair (essentially *String*). This flat model cannot easily capture complex attribute notions such as would be predicated of abstracted real world objects, where attribute values are themselves typically represented by complex objects, either owned or referenced.

This criticism of Cover is absolutely correct. It can be circumvented, though. The language introduced in chapter 9 permits one attribute to store a (file) path to an external compound knowledge template and is thus capable of representing compound properties (complex attribute notions). One problem remains, however: When serialising compound knowledge models consisting of other compound models, the quotation mark as attribute value delimiter is not sufficient, because the beginning and end of an attribute value may get mixed up. Solving it, the XML standard needs to be injured (chapter 9).

4.1.13 Page Description Language

In order to be (more or less) complete in the language overview given in this work, the *Page Description Language* (PDL) as further category shall be mentioned here as well. It describes the contents and appearance (text, graphical shapes, images) of a page to be printed in a device-independent, higher-level way than an actual output bitmap [60]. It may therefore serve as an: *interchange standard for (the) transmission and storage of printable documents* [143]. Well-known PDL representatives are:

- *Device Independent* (DVI) format
- *Printer Control Language* (PCL)
- *PostScript* (PS)
- *Portable Document Format* (PDF)

After [143], *PostScript* is a: *full programming language, rather than a series of low-level escape sequences*. It is stack-based and interpreted. These properties made it the: *language*

of choice for graphical output, until PDF appeared. The following PostScript code example [60] computes $(3 + 4) * (5 - 1)$:

```
3 4 add 5 1 sub mul
```

4.1.14 Hardware Description Language

Hardware Description Language (HDL) is an umbrella term for any computer language formally describing *Electronic Circuits*, that is their design and operation, as well as tests to verify their operation by means of *Simulation* [60]. HDLs used for the design of digital circuits like *Application Specific Integrated Circuits* (ASIC) or *Field Programmable Gate Arrays* (FPGA) include:

- *Very High Speed Integrated Circuit* (VHSIC) HDL (VHDL) [290, standard 1164]
- *Verilog HDL* [290, standard 1364-2001]
- *SystemC* [74]

Although being similar, HDLs are not programming languages. *HDL's syntax and semantics include explicit notations for expressing time and concurrency which are the primary attributes of hardware*, as [60] writes and adds:

An HDL compiler often works in several stages, first producing a logic description file in a proprietary format, then converting that to a logic description file in the industry-standard *Electronic Data Interchange Format* (EDIF), then converting that to a *Joint Electron Device Engineering Council* (JEDEC) format file. The JEDEC file contains instructions to a *Programmable Logic Device* (PLD) programmer for building logic. On the other hand, a software (programming language) compiler generates instructions to a microprocessor for moving data.

The following sections of this chapter will be about programming-, not hardware description concepts.

4.1.15 Object Oriented Programming

With the emerge of *Object Oriented* (OO) languages, an additional programming paradigm got introduced. That is, many principles such as *Structured and Procedural Programming* (SPP) were still holding true but got extended through *Object Oriented Programming* (OOP). Examples of OOP languages, often defined by simply extending an existing language, are:

- *Smalltalk* [202]
- *C++*, extending the *C* system programming language (section 4.1.7)
- *Python*, as typeless programming language (section 4.1.8)
- *Common Lisp Object System* (CLOS), extending the *Common Lisp* (CL) dialect of the *Lisp* functional programming language (section 4.1.9)

The following sections describe the main concepts behind OOP in brief. Although many of them represent improvements to SPP, this work will point out their weaknesses, too. The merger of attributes (data) and methods (operations) into one common data structure called *Class*, for example, will be criticised and eliminated later in this work (chapter 8).

Code examples in the *Java* programming language are given as well as *Unified Modeling Language* (UML) diagrams. The UML is a semi-formal, graphical description language that offers elements for the concepts of *Object Orientation* (OO). To some extend, programs can be designed, generated and documented using special applications called *UML Tools*.

Classification

The main idea of object oriented programming is to structure program code into *Classes* owning *Attributes* and *Methods* (figure 4.8). They are comparable to the structured data types (*struct*, *record*) of *Structured and Procedural Programming* (SPP) (section 4.1.6) that can own fields representing properties, but not behaviour. A class definition in *Java* source code looks like this:

```
public class Example {
    private Type attribute;
    public void method(Type parameter) {
    }
}
```

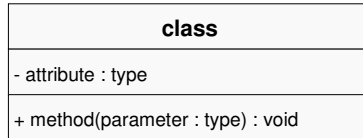


Figure 4.8: Classification as UML Diagram

While procedures and many variables in SPP are global, that is only exist once, classes are treated as types of which many *Instances* (also called *Objects*) can be created, including attributes and methods. In OOP, such memory allocation is called *Instantiation*.

Two related data types are *Abstract Class* and *Interface*. An abstract class can hold attributes and (partly abstract) methods. Just like interfaces, abstract classes cannot be instantiated. An interface is yet more restricted in that it can only have constants but not attributes and only declarations but not actual implementations of methods. Interfaces are commonly used to [297]:

- Realise multiple inheritance (section 4.1.15)
- Encapsulate components (section 4.3.3)
- Pool common methods (section 4.3.4)

Specialities like *Inner Classes* [112] with limited scope of validity are of minor importance to the argumentation of this document and not further explained here.

The *Bundling* of attributes and methods (state and logic) causes more system interdependencies and complications than were predictable. It is a big disadvantage that affects all modern object-oriented systems. [125] Certainly, the bundling stems from best intentions to receive cleaner code by keeping not only attributes but also methods in a common module,

such avoiding *wild* and *global* procedures. But now, modules not only have to refer to other modules for accessing their state data; the same is needed for accessing their logic in form of method calls.

With OOP, the number of cross-relations between modules, and inter-dependencies between system layers may rise dramatically. In reality, state- and logic properties are two *different* things that have to be kept in different places! Both can have a similar, hierarchical structure but each is a concept on its own, as chapter 8 will show.

Encapsulation

One recommendation of object oriented programming is that the properties of an object created as instance of a class be protected through special *Access Methods* (figure 4.9). A *Java* code example can be found below. The intention is not to expose class attributes to other classes by minimising direct access to them and such to provide some security by preventing illegal access to an object's interna. Therefore, this paradigm is called *Encapsulation* or *Information-/ Data Hiding*. Another advantage is that if an attribute changes its name, then only one place in the code (the access method), instead of hundreds, needs to be updated.

```
public class example_class {  
    private Type attribute;  
    public void set_attribute(Type a) {  
        this.attribute = a;  
    }  
    public Type get_attribute() {  
        return this.attribute;  
    }  
}
```

Special keywords are necessary to ensure proper encapsulation by making attributes and methods *visible* to only certain outside objects. These keywords are: *public*, *protected* and *private*. In the *Java* programming language [112], an additional package protection level is applied when none of the aforementioned keywords is found. The *Delphi* language [337] knows an additional *published* keyword that makes properties visible in its object-inspector tool. Other languages may contain further variations of access limitations.

The recommendation to encapsulate attributes produces thousands of lines of source code whose usefulness is at least questionable [126]. In about 90% of cases (practical experience

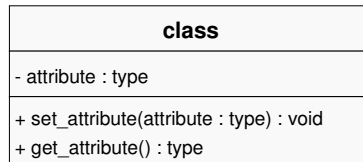


Figure 4.9: Encapsulation as UML Diagram

of the author of this document), the *set* and *get* methods consist of only one single line accessing an attribute value which in the end is the same as accessing that attribute directly. Sometimes, additional lines with a trigger function to update other parts of the system are added. They get invoked whenever an attribute value of the called object is changed by a *set* method:

```
public void set_attribute(Type a) {
    this.attribute = a;
    get_update_manager().update(this);
}
```

But, as shown below, this update notification could as well be taken over by the object that was calling the *set* method:

```
public void method() {
    example_object.set_attribute(a);
    get_update_manager().update(example_object);
}
```

The argumentation that *in this case a lot of redundant code would be produced since the update function has to be implemented in every calling object, instead of just once in the called object* does not really hold true when looking into programming practice. The number

of external objects calling an object is mostly very well manageable. It finally seems that thousands of *set/ get* access methods could be eliminated which would lead to a tremendous code reduction and improved clarity.

The language introduced in chapter 9 does not use encapsulation and the attributes (state knowledge) and methods (logic knowledge) modelled in it are not bundled together.

Inheritance

Inheritance allows for code minimisation by letting classes inherit attributes and methods from their *superior* (sometimes called *parent*) class (figure 4.10). Redundant code can such be avoided and existing code can be reused. An inheriting class in *Java* source code looks like this:

```
public class example extends super_class {
}
```

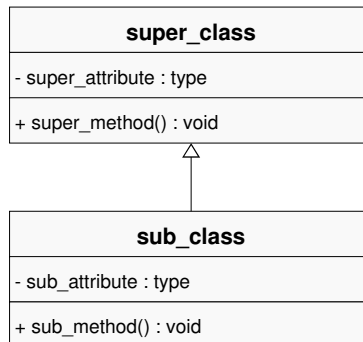


Figure 4.10: Inheritance as UML Diagram

Some object oriented programming languages (such as *C++*) permit *Multiple Inheritance*. Classes written in those languages can have more than one superior class. Other languages (such as *Java*) that have *Single Inheritance* only, sometimes offer to *inherit* (*realise/ implement*) multiple interfaces. An interface forces its subclasses to implement all methods

it declares (more on this in section 4.3.3) and can such provide a common *Application Programming Interface* (API) which makes classes interchangeable and hence encourages reuse.

Fragile Base Class

Despite the possible code reduction through class inheritance, there are some negative effects that hinder just this code reduction and reuse. John K. Ousterhout writes in his article [244]:

Implementation inheritance, where one class borrows code that was written for another class, is a bad idea that makes software harder to manage and reuse. It binds the implementations of classes together so that neither class can be understood without the other: a subclass cannot be understood without knowing how the inherited methods are implemented in its superclass, and a superclass cannot be understood without knowing how its methods are inherited in subclasses. In a complex class hierarchy, no individual class can be understood without understanding all the other classes in the hierarchy. Even worse, a class cannot be separated from its hierarchy for reuse. Multiple inheritance makes these problems even worse. Implementation inheritance causes the same intertwining and brittleness that have been observed when goto statements are overused. As a result, object-oriented systems often suffer from complexity and lack of reuse.

Unwanted dependencies caused simply by the usage of inheritance are called *Fragile Base Class Problem* [41, section *Layers*; p. 48]. The source code changes resulting from base class manipulation are also called *Cascade of Change* [119, Vorwort]. They are just the opposite of what inheritance was actually intended to be for: *Reusability*. Leonid Mikhajlov and Emil Sekerinski [213] write:

This problem occurs in open object-oriented systems employing code inheritance as an implementation reuse mechanism. System developers unaware of extensions to the system developed by its users may produce a seemingly acceptable revision of a base class which may damage its extensions. The fragile base class problem becomes apparent during maintenance of open object-oriented systems, but requires consideration during design.

They identify the following *Restrictions* [213] disciplining the code inheritance mechanism, thus avoiding the *Fragile Base Class Problem*, but on the cost of general *Flexibility*:

- *No cycles*: A base class revision and a modifier should not jointly introduce new cyclic method dependencies.
- *No revision self-calling assumptions*: Revision class methods should not make any additional assumptions about the behaviour of the other methods of itself. Only the behaviour described in the base class may be taken into consideration.
- *No base class down-calling assumptions*: Methods of a modifier should disregard the fact that base class self-calls can get redirected to the modifier itself. In this case bodies of the corresponding methods in the base class should be considered instead, as if there were no dynamic binding.
- *No direct access to base class state*: An extension class may not access the state of its base class directly, but only through calling base class methods.
- *No modifier invariant function*: A modifier should not bind values of its instance variables with values of the intended base class instance variables to generate an invariant.

In order to remain highly flexible and to avoid the fragile base class problem, the language described in chapter 9 does not use inheritance, although it could be extended to do so. In this case, of course, its interpreter (chapter 10) would have to be adapted as well.

Polymorphism

Another object oriented feature that comes with inheritance is *Polymorphism*. It allows methods to be *overloaded* (sometimes called *overridden*). That is, on two objects created from different classes inheriting from each other, the right equally named method will be called by the language interpreter program (figure 4.11), which leads to different behaviour depending on the current object context. Following is a *Java* code example overloading a method to gain polymorphic behaviour:

```
public class super_class {
    public void method() {
        do_something();
    }
}

public class sub_class extends super_class {
```

```

public void method() {
    do_something_else();
}
}

```

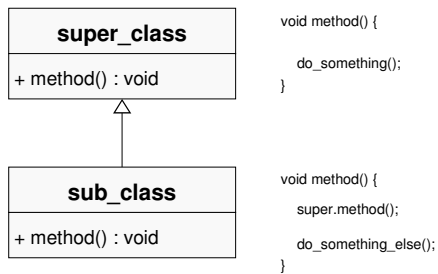


Figure 4.11: Polymorphism as UML Diagram

If objects instantiated from a sub class want to make use of the functionality contained in the super class' equally named method, the sub class' method needs to call the super class' method explicitly using the keyword *super*:

```

public class sub_class extends super_class {
    public void method() {
        super.method();
        do_something_else();
    }
}

```

Container

An object that got created through instantiating a class represents an allocated area in a computer's memory which needs to be referenced in order to be able to work with it, and to finally destroy it. The size of that area may change *dynamically*, depending on how

the properties of the object are manipulated. Primitive types like *integer* or *double* also reserve memory space, only that the size of that space is *not* dynamic; it is pre-defined by the programming language, for each type. All *Structured- and Procedural Programming* (SPP) languages and some *Object Oriented Programming* (OOP) languages, like Java, offer standard primitive types.

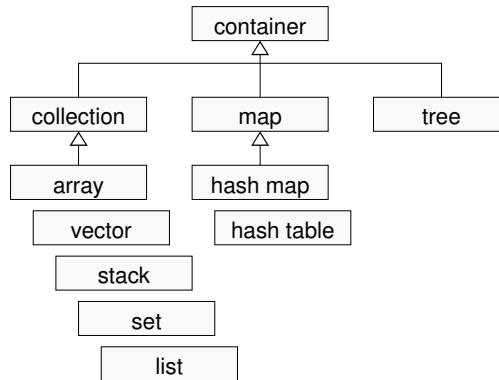


Figure 4.12: Java Container Framework Systematics

One way to store references to more than one dynamic element in memory, or primitive data, is a *Container*. Modern programming languages offer many different kinds of containers. Figure 4.12 shows a systematics of the Java container framework [112], as example, which gets briefly introduced in the following paragraphs. Its main categories of systematisation are *Collection*, *Map* and *Tree*.

A similar library of container classes, algorithms and iterators exists for the C++ programming language. It is called *Standard Template Library* (STL) [153] and it talks of *Sequence* and *Associative Container*, where Java says *Collection* and *Map*.

Collection The *Array* is the most basic form of a container. It represents an allocated area in the computer's *Random Access Memory* (RAM). A *Vector* implements a dynamically growable array of objects. The *Stack* class extends the vector class and represents a *Last-In-First-Out* (LIFO) stack of objects. A collection that contains no duplicate elements is

called a *Set*. Unlike sets, *Lists* typically allow duplicate elements. Synonyms for list are *Ordered Collection* and *Sequence*.

Objects that can generate a series of elements, one at a time, implement the *Enumeration* interface. Successive calls to the *nextElement* method return successive elements of such a series. In recent releases of the *Java Development Kit* (JDK) [112], *Iterator* takes the place of enumeration, in the collections framework. An iterator over a collection differs from an enumeration in that it allows the caller to remove elements, with well-defined semantics, from the underlying collection during an iteration.

Map A *Map* (also called *Dictionary* or *Table*) is an object that maps *Keys* to *Values*. It cannot contain duplicate keys; each key can map to at most one value. Java offers two kinds of a map: *Hash Map* and *Hash Table*. The former is roughly equivalent to the latter, except that it permits null values and the null key [112, 120].

Tree A *Tree*, or more exact *Tree Node*, is a further kind of container. Many tree nodes, in hierarchical order, may form a tree. A tree node may represent a *Leaf* with no children or a *Branch* with one or more children. The top-most tree node is usually called *Root*.

Falsifying Polymorphism

Problems can occur when inheriting containers. This is now demonstrated on a Java example adopted from [222].

A class *ExtendedHashtable* extends the standard container *Hashtable* (figure 4.13). The *ExtendedHashtable* overrides the *put* method and lets it do two calls to the *put* method of the superior class *Hashtable*, the second of these calls adding the letter *s* to the key.

A first object of type *ExtendedHashtable* gets filled by calling the *put* method which adds two identical element values with the two different keys *ball* and *balls* to the container. When the container is full, a new one with extended size gets created and all values of the old have to be copied into the new container, which is again of type *ExtendedHashtable*.

If the *put* method is now used to accomplish this, a falsified container with more elements than the original one will be retrieved. The copying of the first element *ball* results in two elements *ball* and *balls*, placed in the new container. The copying of the second element

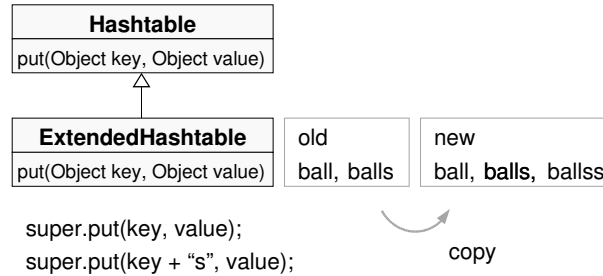


Figure 4.13: Falsified Contents with Container Inheritance

balls adds two further elements *balls* and *ballss*, whereby the *balls* key stemming from the copying of the first element gets overwritten.

This example demonstrates only the principle of how the automatic size extension of inherited container objects with element copying using container-owned methods can incorrectly modify the container contents. The Java language's *Hashtable* class uses a slightly different mechanism, handing over the hashtable object as parameter of a copy constructor which internally calls a *putAll* method which finally calls the *put* method. Other OOP languages may use different mechanisms. Of course, there are workarounds to avoid the described troubles. But as a matter of fact, container inheritance may – due to polymorphism – cause unpredictable behaviour leading to *falsified* container contents.

The language and interpreter introduced in chapters 9 and 10 base on just one container structure for knowledge representation, that covers many of the traditional forms of containers.

Conclusion

As could be seen in the previous sections, OOP contributed many new concepts to software design, thus trying to improve SPP. Most importantly, SPP data structures (struct, record)

got extended towards the *Class* which does not only hold data (attributes), but also operations (methods). This brought with the concept of *Encapsulation*, which permits only special methods of an *Object* (class instance) to access the data (properties) of that same object. The next innovation was *Inheritance*, which allows a class to reuse the attributes and methods of its super class(es). Finally, inheritance was used to introduce the concept of *Polymorphism*, which lets objects react differently, depending on the class they were instantiated with.

All of these concepts were true innovations as compared with traditional SPP techniques. However, they have their own drawbacks: growth of the number of dependencies within a system (links between classes), caused by the bundling of attributes and methods; fragile base class problem; falsified container contents with container inheritance. This work will not just revise these concepts, but turn them upside down. Data (attributes) and operations/algorithms (methods) are not bundled any longer; the resolution of inheritance relationships at runtime gets eliminated and with it polymorphism; container inheritance is not necessary any longer, since only one global container structure (knowledge container) is used in a system. More on that in part II of this work.

4.2 Pattern

The previous sections investigated basic concepts offered by today's programming languages and -paradigms. The following and all later sections of this chapter describe design techniques that belong to a higher conceptual level. *Patterns*, in a more correct form called *Software Patterns*, are the first technique dealt with. They became popular through *Object Oriented Programming* (OOP), but their use is not limited to OOP languages. Patterns represent solutions for recurring software design problems and can be understood as recommendations for how to build software in an elegant and efficient way. In the past, more detailed definitions have been given by meanwhile well-known authors.

Christopher Alexander, an architect and urban planner, writes [3]: *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* He gave this definition primarily for problems occurring in architecture, construction, and urban/regional planning, but it can be applied in the same manner to software design, as done first by Ward Cunningham and others [148].

The systems designer Swift [73] sees a pattern as: *essentially a morphological law, a relationship among parts (pattern components) within a particular context. Specifically, a pattern expresses a relationship among parts that resolves problems that would exist if the relationship were missing. As patterns express these relationships, they are not formulae or algorithms, but rather loose rules of thumb or heuristics.*

The *Gang of Four* (GoF) (Erich Gamma et al.) applied Alexander's definition to object oriented software and created a whole catalogue of design patterns [108]. After them, patterns are: *Structured models of thinking that represent reusable solutions for one-and-the-same design problem. They shall support the development, maintenance and extension of large software systems, while being independent from concrete implementation languages.* The experts identified four basic elements of each pattern: *Name, Problem, Solution* and *Consequences* (advantages and disadvantages).

For Frank Buschmann et al., software patterns contain the knowledge of experienced software engineers and help to improve the quality of decision making [41]. In his opinion, they are basic solutions for problems that already occurred in a similar way before. Therefore, the author talks of *Problem Solution Pairs*.

Martin Fowler means that: *A pattern is some idea that already was helpful in a practical context and will probably be useful in other contexts, too.* [97]. After him, patterns, however they are written, have four essential parts: *Context, Problem, Forces* and *Solution*.

Depending on their experience, software developers can produce good or bad solutions, in every domain. One possibility to improve less well-done designs or to extend legacy systems are the so-called *Anti-Patterns* (telling how to go from a problem to a bad solution), or the contrasting *Amelioration Patterns* (telling how to go from a bad solution to a good solution) [148]. Both help finding patterns in wrong-designed systems and give advice for their improvement.

There are efforts to combine patterns to form a *Pattern Language*, also called *Pattern System* [41]. Such systems describe dependencies between patterns, specify rules for pattern combination and show how patterns can be implemented and used in software development practice.

Several schemes of *Pattern Classification* exist. One possible is shown in figure 4.14. Considering the level of abstraction (granularity), it distinguishes between *Architectural*-, *Design*- and *Idiomatic* patterns [41]. Design patterns, in turn, are divided after their functionality (problem category) into *Creational*-, *Structural*- and *Behavioural* patterns [108]. The

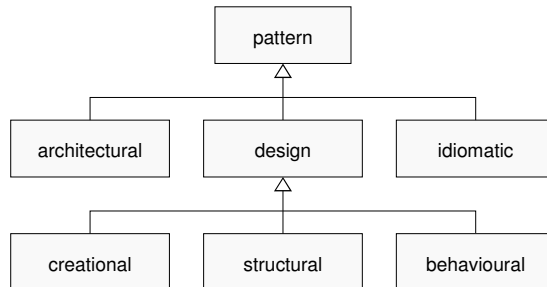


Figure 4.14: Software Pattern Classification

Wikipedia Encyclopedia [60] mentions three further problem categories: *Fundamental*-, *Concurrency*- and *Real-time* patterns. Other criteria (dimensions) of classification exist. Fowler introduces a completely different category which he calls *Analysis Patterns* [97]. These are applicable early in the software engineering process (chapter 2). And he defines patterns that are more often used for describing the modelling *Language* than the actual *Models* as *Meta Model Patterns*.

In the following sections, a greater number of known patterns will be described briefly. They form the scientific basis for the ideas following in part II of this work and some of them appear in a modified form in the language and interpreter introduced in part III. Chapter 7 moreover introduces a new pattern systematics for which it references common patterns as introduced here. However, since the next sections do not want to copy the work accomplished by the above-mentioned authors, they refer to the corresponding literaric source for more detailed explanation.

4.2.1 Architectural

Architectural Patterns are templates for the gross design of software systems. They describe concrete software architectures and provide basic structuring (modularisation) principles.

Layers

The *Layers* pattern [41] is one of the most often used principles to subdivide a system into logical levels. One variant was shown in figure 4.1, at the beginning of this chapter. It contained the three layers *Presentation*, *Domain Logic* and *Data Source*. A more general illustration can be seen in figure 4.15. It shows a client using the functionality encapsulated in a layer. That top-most layer delegates subtasks to lower-level layers which are specialised on solving them. Another well-known example making use of this pattern is the *ISO OSI* model as introduced in section 3.11.

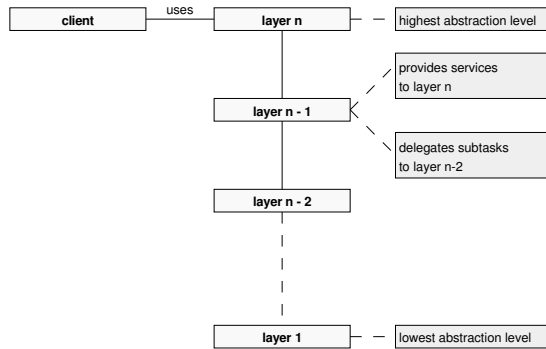


Figure 4.15: Layers Pattern

One variant of this pattern, mentioned by Buschmann [41], is the *Relaxed-Layered-System*. It permits a layer to not only use the services of its direct base layer, but also of yet lower-situated layers. The base layer, in this case, is called *transparent*.

The ontology examples in chapter 7 are organised according to the *Layers* pattern. Their layers represent levels of growing granularity.

Layer Supertype

The *Layer Supertype* pattern [101] is a rather simple but quite useful one. It assumes that a system is structured using the *Layers* pattern. What the pattern proposes is to add a

(possibly abstract) class that all other classes in its layer inherit from (figure 4.16). The reason is that basic functionality common to all classes in a layer, for example persistence- or logging capabilities, can be provided once by the supertype, such avoiding redundancies.

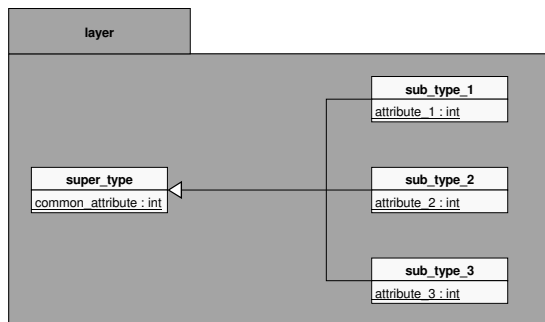


Figure 4.16: Layer Supertype Pattern

The language introduced in chapter 9 does not use inheritance and thus cannot use super knowledge templates in the meaning of the *Layer Supertype* pattern. Nevertheless, the pattern is important because of its idea to categorise similar knowledge, such as all templates of: a *Textual User Interface* (TUI), a *Graphical User Interface* (GUI), a *Domain Model* etc.

Domain Model

One of the three layers in figure 4.1 shown at the beginning of this chapter is the *Domain Model*. Fowler [101] proposed it as singular pattern because of its importance in large-scale business systems. Figure 4.17 shows an imaginary business domain model. The actual focus, however, should not be put on the inside structure of this example model, but on the fact that the domain model represents a layer on its own.

This separation cannot be found in all systems and in fact, it does not make sense for *all* systems. Small solutions let their user interface or application control, respectively, access a database directly which avoids the rather big effort of creating a special domain model. But

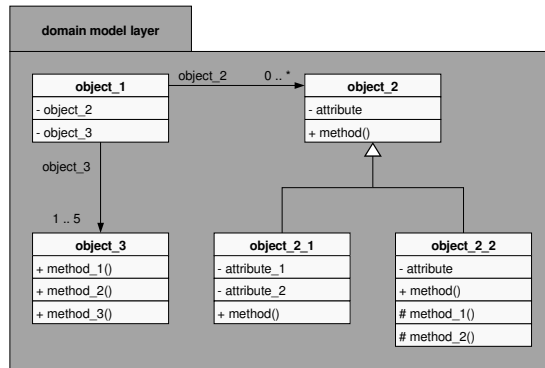


Figure 4.17: Domain Model Pattern

the larger the system to be created and the more clear the desired architecture shall be, the more recommendable it is to use the *Domain Model* pattern.

It will be helpful to have heard about this pattern when reading chapter 8 dealing with domain-, user interface- and other models and their translation into each other.

Data Mapper

Besides the *Domain Model*, figure 4.1 contained a layer called *Data Source* which may for example represent a database. Normally, both layers need to exchange data. Modern systems use OOP methods to implement the domain model. Database models, on the other hand, are often implemented on the basis of an *Entity Relationship Model* (ERM). In order to avoid close coupling and a mix-up of both layers, the introduction of an additional *Data Mapper* layer [101] in between the two others may be justified (figure 4.18). The most important idea of this pattern is to abolish the interdependencies of domain- and persistence model (database).

The dashed arrows in figure 4.18 indicate dependencies. The data mapper layer knows the domain model- as well as the data source layer, via *unidirectional* relations. Its task is to *translate* between the two, in both directions. Domain model and data source know nothing

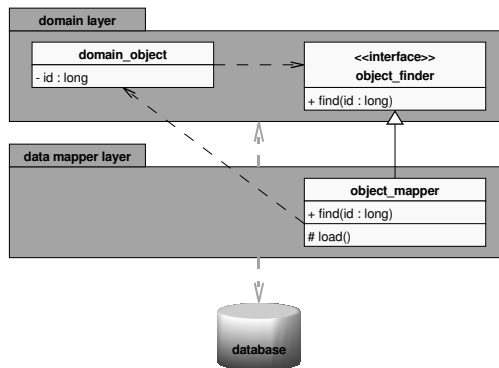


Figure 4.18: Data Mapper Pattern

from each other. Each domain model class knows its appropriate *object_finder* interface but does not know the implementation of the same. That is, persistence- and data retrieval mechanisms are hidden in front of the domain model. The *object_mapper* implementation is part of the mapping package and also implements all finder methods. It maps data of the received result sets to the special attributes of the domain model objects.

The *Mediator* pattern [108] is similar to the *Mapper*, in that it is used to decouple different parts of a system. Fowler [101] writes: *... the objects that use a mediator are aware of it, even if they aren't aware of each other; the objects that a mapper separates aren't even aware of the mapper.*

Although the *Data Mapper* pattern is very helpful at implementing OO systems, two things are to be criticised: Firstly, since the *object_finder* relies on functionality specific to the retrieval of persistent data, it does actually belong into the data mapper layer what, if done, would create bidirectional dependencies between the domain model- and data mapper layer. But also with the *object_finder* remaining in the domain model layer, dependencies are not purely unidirectional. It is true that from an OO view, they are. Internally, however, a super class or interface relates to its inheriting classes, so that it can call their methods to satisfy the polymorphic behaviour.

Secondly, the layers do not truly build on each other. Taken an architecture similar to the

one in figure 4.1, consisting of the following five instead of only three layers:

1. Presentation
2. Application Process
3. Domain Model
4. Data Mapper
5. Data Source

... the application process does not only access the domain model layer, it also has to manage (create and destroy) the objects of the data mapper layer. In other words, it surpasses (disregards) the domain model layer when accessing the data mapper layer directly.

Chapter 8 will describe how a strict separation of state- and logic knowledge allows to access and translate runtime models unidirectionally.

Data Transfer Object

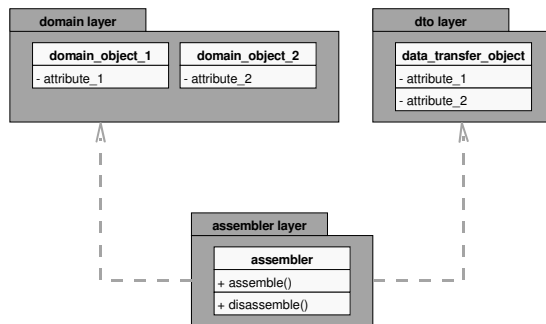


Figure 4.19: Data Transfer Object Pattern

It is a well-known fact that many small requests between two processes, and even more between two hosts in a network need a lot of time. A local machine with two processes has to permanently change the *Program Context*; a network has a lot of *Transfers*. For each

request, there is a necessity of at least *two* transfers – the *Question* of the client and the *Answer* of the server. Transfer methods are often expected to deliver common data such as a Person's address, that is surname, first name, street, zip-code, town and so on. These information is best retrieved by only *one* transfer call. That way, the client has to wait only once for a server response and the server does not get too many single tasks. All address data (in this example) would best be packaged together and sent back to the client.

A scenario of that kind is exactly what the *Data Transfer Object* pattern [101] proposes a solution for: A central *Assembler* object takes all common data of the server's domain model objects and assembles them together into a special *Data Transfer Object* (DTO), which is a flat data structure (figure 4.19). The server will then send this DTO over network to the client. On the client's side, a similar assembler takes the DTO, finds out all received data and maps (disassembles) them to the client's domain model. In that manner, a DTO is able to drastically improve the communication performance.

Both, *Data Mapper*- and DTO pattern translate one model into another. Due to this similarity, chapter 8 will try to merge them into a common *Translator* architecture.

Model View Controller

After having had a closer look at design patterns for persistence (*Data Mapper*) and communication (*Data Transfer Object*), this section considers the presentation layer of an application (figure 4.1), which is often realised in form of a *Graphical User Interface* (GUI). Nowadays, the well-known *Model View Controller* (MVC) pattern [41, 101] is used by a majority of standard business applications. Its principle is to have the *Model* holding domain data, the *View* accessing and displaying these data and the *Controller* providing the workflow of the application by handling any action events happening on the view (figure 4.20). This separation eases the creation of applications with many synchronous views on the same data. Internally, the MVC may consist of design patterns like:

- *Observer* (section 4.2.2) which notifies the views about data model changes
- *Strategy* [108] which encapsulates exchangeable functionality of the controller
- *Wrapper* (section 4.2.2) which delegates controller functionality to the *Strategy*
- *Composite* (section 4.2.2) which equips graphical views with a hierarchical structure

Some MVC implementations like parts of the *Java Foundation Classes* (JFC) use a simplified version not separating controllers from their views. The *Microsoft Foundation Classes*

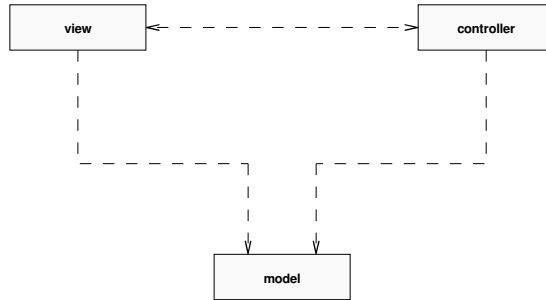


Figure 4.20: Model View Controller Pattern

(MFC) C++ library calls its implementation *Document-View*.

Besides the above-mentioned patterns *Data Mapper* and *DTO*, *MVC* is the third one getting merged into a common *Translator* architecture, in chapter 8.

Hierarchical Model View Controller

There exist several extensions of the *MVC* pattern, one of them being the *Hierarchical Model View Controller* (HMVC) [43]. It combines the patterns *Composite* (section 4.2.2), *Layers* (section 4.2.1) and *Chain of Responsibility* (section 4.2.2) into one conceptual architecture (figure 4.21). This architecture divides the presentation layer into hierarchical sections containing so-called *MVC Triads*. The triads conventionally consist of *Model*, *View* and *Controller*, each. They communicate with each other by relating over their controller object. Following the *Layers* pattern, only neighbouring layers know from each other.

As a practical example, the upper-most triad could represent a graphical *Dialogue* and the next lower one a *Panel*. Being a container, too, the panel could hold a third triad like for example a *Button*. Events occurring at the button are then normally processed by the corresponding controller belonging to the button's triad. If, however, the button controller cannot handle the event, that is forwarded along the chain of responsibility to the controller

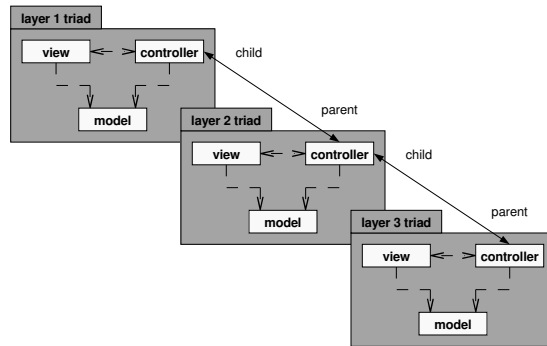


Figure 4.21: Hierarchical Model View Controller Pattern

of the higher-next layer. If also the panel controller does not know how to handle the event, the final responsibility falls to the controller of the dialogue's triad.

The HMVC is similar to the *Presentation Abstraction Control* (PAC) pattern [41]. A *PAC Agent* is comparable to an *HMVC Triad*.

Chapter 7 will apply the principle of *Hierarchy* not only to logic- (controller), but also to user interface- (view), domain- and further models.

Microkernel

The *Microkernel* pattern [41] allows to keep a system flexible and adaptable to changing requirements or new technologies. A minimal functional *Kernel* gets separated from extended functionality. The kernel may call internal- or external servers (figure 4.22) to let them solve special tasks which do not belong to its own core responsibility. Internal servers, also called *Daemons*, were already mentioned in section 3.6.

This pattern provides a *Plug & Play* environment and serves as base architecture for many modern *Operating Systems* (OS). Andrew S. Tanenbaum recommends its use as well [304]. And also the interpreter that will be described in chapter 10 uses this pattern in its own adapted form.

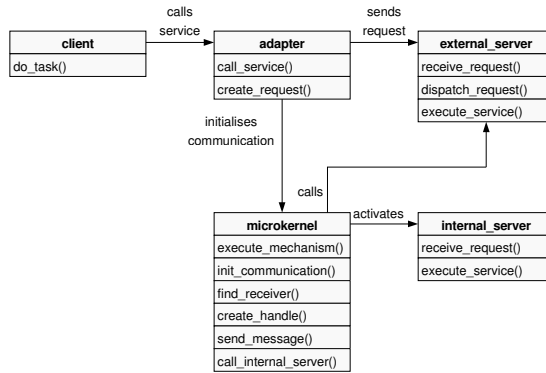


Figure 4.22: Microkernel Pattern

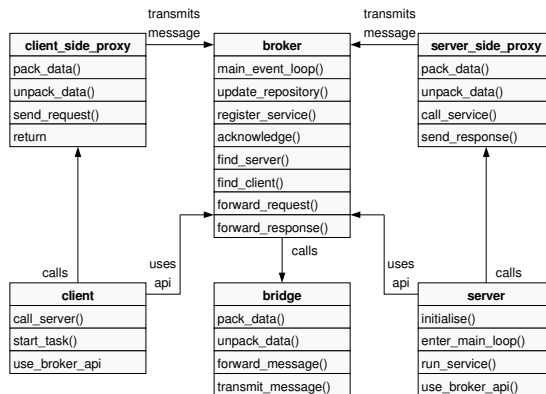


Figure 4.23: Broker Pattern

Broker

The *Broker* pattern [41] may support the creation of an IT infrastructure for distributed applications. It connects decoupled components which interact through remote service invocations (figure 4.23). The broker is responsible for coordinating all communication, for forwarding requests as well as for transmitting results and exceptions.

Chapter 10 introduces an interpreter program being able to act as broker.

Pipes and Filters

Systems that process streams of data may make use of the *Pipes and Filters* pattern [41]. It encapsulates every processing step in an own *Filter* component and forwards the data through channels which are called *Pipeline* (figure 4.24). The data forwarding can follow various scenarios:

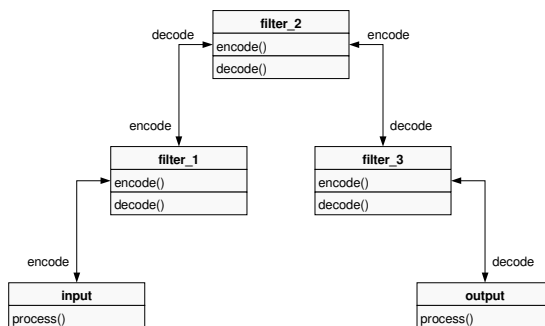


Figure 4.24: Pipes and Filters Pattern

- *Push*: active filter pushes data to passive filter
- *Pull*: active filter pulls data from passive filter
- *Mixed Push-Pull-Pipeline*: all filters may push or pull data
- *Independent Loops*: all filters are active and access pipeline data

Families of related systems can be formed by changing the single filter positions. Special communication filters are also used in the interpreter program of chapter 10. Its filters belong to neither of the above-listed forms of data forwarding, because they are all passive, controlled from an outside entity which is not a filter itself.

Reflection

The *Reflection* pattern [41] (also known under the synonyms *Open Implementation* or *Meta-Level Architecture*) provides a mechanism to change the structure and behaviour of a software system *dynamically*, that is at runtime, which is why that mechanism is sometimes called *Run Time Type Identification* (RTTI). A reflective system owns information about itself and uses these to remain changeable and extensible.

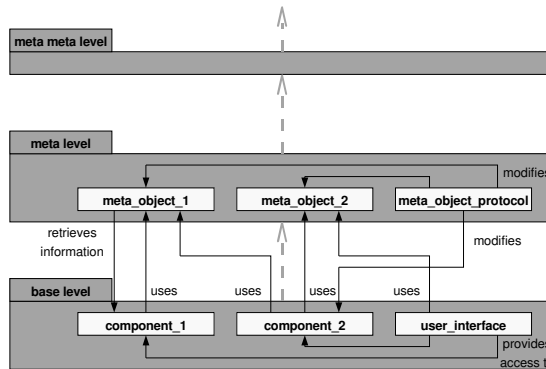


Figure 4.25: Reflection Pattern

Reflective information *about* something is called *Meta Information*. Therefore, the level above the *Base Level* in figure 4.25 is labelled *Meta Level*. The base level depends on the meta level, so that changes in the meta level will also affect the base level. All manipulation of meta objects happens through an interface called *Meta Object Protocol* (MOP), which is responsible for checking the correctness of- and for performing a change. If a further level holds information about the meta level, then that additional level is called *Meta Meta Level*, and so forth.

Many examples of meta level architectures exist. In his book *Analysis Patterns* [97], Fowler uses them extensively. He talks of *Knowledge Level* (instead of meta level) and *Operational Level* (instead of base level). Elements of the *Unified Modeling Language* (UML) are defined in an own meta model [235]. And the principles of reflection are also supported by several programming languages, such as *Smalltalk* [202] and *Java* [112].

Classes (types) in a system have a static structure, as defined by the developer at design time. Normally, most classes belong to the base level containing the application logic. As written before, one way to change the structure and behaviour of classes at runtime is to introduce a meta level providing type information, in other words functionality that *all* application classes need. This helps avoid redundant implementations of the same functionality.

Looking closer at functionality, it turns out that some basic features like persistence and communication occur repeatedly in almost all systems, while other parts are very specific to one concrete application. Traditionally, the application classes in the base level have to cope with the general system functionality although that is not in their original interest. It therefore seems logical to try to divide application- and system functionality. Chapter 6 will deal with this thought in more detail.

Broken Type System

This section does not describe another pattern. Instead, it wants to come back to reflective mechanisms as described before and elaborate their negative effects a bit more. Although the following review concentrates on the example of *Java*, many points surely count for other OO languages as well. Languages like *Smalltalk* or the *Common Lisp Object System* (CLOS) offer reflective mechanisms [41]. The *C++ Standard Library*, also known as *libstdc++* [195], has a *type_info* class providing meta information that *C++* innately does not have. In the *Java* framework [112], finally, the basic *java.lang.** package contains the top-most super class *java.lang.Object*. All other classes in the framework inherit from it. Additionally, the package contains a class *java.lang.Class* which, among others, keeps reflective (meta) type information about a *Java* class':

- Package
- Name
- Superior Class
- Interfaces

- Fields
- Methods
- Constructors
- Modifiers
- Member Classes

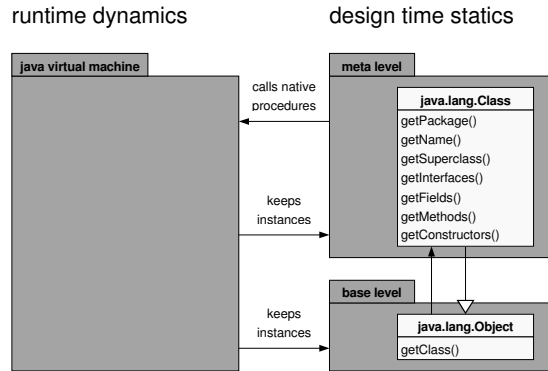


Figure 4.26: Java Type System

Via the `getClass()` method which they inherit from *java.lang.Object* (figure 4.26), all Java classes have access to that reflective information in their meta class. The meta class *java.lang.Class* itself uses so-called *native* methods to access the information in the *Java Virtual Machine* (JVM). The JVM operates on a level underneath the actual application, close to the *Operating System* (OS). It interprets the Java application source code and resolves all object-oriented- into procedural structures, and finally low-level system instructions. All runtime objects, that is class instances, are hold in dynamic structures internal to the JVM. That is why *native* methods need to be used to access and change the runtime structure or behaviour of objects.

One problem that becomes obvious when inspecting figure 4.26 is the existence of a *Bidirectional Dependency* (*Circular Reference*). The two sub dependencies causing it are:

1. *Inheritance of java.lang.Class from java.lang.Object* which is due to the rule that all Java classes need to inherit from the top-most framework class

2. *Association* from *java.lang.Object* to *java.lang.Class* which enables every object to access its meta class using the *getClass()* method

The avoidance of circular references is one of the most basic principles of computer programming (section 4.2.2). The disadvantage of bidirectional dependencies between meta and basic level is also mentioned by Buschmann [41]. If meta classes in the kind of *java.lang.Class* define the structure and behaviour of all basic classes inheriting from *java.lang.Object*, then those meta classes in turn should *not* inherit from *java.lang.Object* themselves. The new language described in chapter 9 permits applications to be programmed without bidirectional dependencies. Functionality that could be put into a meta level is provided by a low-level interpreter instead (chapter 10).

Another problem is the mixed and redundant storage of meta information which Jonathon Tidswell [132] even calls a *Broken Type System*. He writes: *A careful examination of the classes in the standard runtime will show that they are not strictly instances of java.lang.Class (hint: statics)*. Gilbert Carl Herschberger II [132] calls the separation of reflection and wrappers an *Inconsistent Design*. Java classes are based on many different kinds of type information:

- Structure applied by the JVM through the usage of the *class* keyword
- Meta information supplied by the *java.lang.Class* class
- Reflective information provided by *java.lang.reflect.**
- Wrapper classes for primitive types in *java.lang.**
- Dynamically created array classes, without having an array class file

The fact that the *java.lang.Class* class which is to provide meta information *about* classes is a *Class* itself is an antagonism. It is true that that meta class is made *final* to avoid its extension by inheriting subclasses. But correctly, it should not be a class at all. Yet how can this paradoxon be resolved? Obviously, one of the two dependencies between *java.lang.Object* and *java.lang.Class* needs to be cut. But then either the *java.lang.Object* class would not be able to access its meta information anymore or the *java.lang.Class* class would not be available as runtime object to other polymorphic data structures. One solution could be to merge both classes, so that each object, by default, has the necessary methods to access its meta information. But as it turns out, this would not be a real solution, just a *Shift* of the problem to another level. As mentioned above, the JVM keeps all instances (objects) in internal, dynamic structures. If objects were allowed to access these internal structures

via native methods (procedures), a similar kind of bidirectional dependency, between the JVM and its stored objects, would occur. Gilbert Carl Herschberger II writes [132]:

A purist *Super Platform* does not bleed into the *Sub Platform*. In practice, this doesn't make people happy because transition is more difficult. Java itself is a super platform. It bleeds into the sub platform in numerous ways, including *Java Native Interface* (JNI) and *Runtime.exec()*, leaving us with a security headache. Reasonable security can be achieved; but, it is far from automatic.

One finally has to ask whether the usage and manipulation of meta information is really necessary at all! If objects did not have a *static* structure consisting of certain attributes and methods, as defined by the software developer at design time, but instead based on a uniform, *dynamically* changeable structure – the need to use reflective mechanisms might disappear. More on this topic in chapter 6.

There are other Java-related points to be criticised, that have their reason in the application of the *Reflection* pattern. Although it is worth noting they exist, these are *not* explained in detail here, since this document wants to focus on general concepts. Gilbert Carl Herschberger II [132] mentions the problematic issue of *Pre-Conditions*, leading to corresponding *Assumptions*. After him, such work-arounds were necessary to break circular references in Java:

- Each JVM must pre-define an *Internal Meta Class*, implemented in machine code and *not* available as Java bytecode in a class file. The *java.lang.Class* as base meta class for all Java classes depends on that internal meta class and assumes its existence.
- A JVM pre-defines one *Primordial Class Loader*, implemented in machine code and resolved at compile-time. Since additional class loaders need to know their meta class when being created, they have to assume the primordial class loader exists so that, using it, their meta class can be created first.

Further, Jonathon Tidswell [132] is of the opinion that there are a number of security issues related to the design of Java, for example:

- Global names not local references are used for security
- Wrappers and names are used for reflection

Even though most of the issues raised in this section are rather Java-specific, many of them apply to other programming languages as well. *Smalltalk* [202] and *CLU* [198], for example,

make primitive types look like classes and do not need special *Wrapper* classes like Java. But when digging deep enough, one will find that this is *Syntactic Sugar*, as Peter J. Landin used to call additions to the syntax of a computer language that do not affect its expressiveness but make it *sweeter* for humans to use [60].

The interpreter described in chapter 10 uses only one single type called its general *Knowledge Schema* (chapter 7), and thus circumvents any troubles with broken type systems. Reflective techniques are not needed because general functionality contained in the interpreter is separated from domain-specific knowledge which is stored externally to the interpreter, in a special language (chapter 9).

4.2.2 Design

Gamma et al. [108] define a design pattern as: *description of collaborating objects and classes which are tailored to solve a general design problem in a special context*. Mostly, patterns are in relation to each other. They can be combined to master more complex tasks.

Command

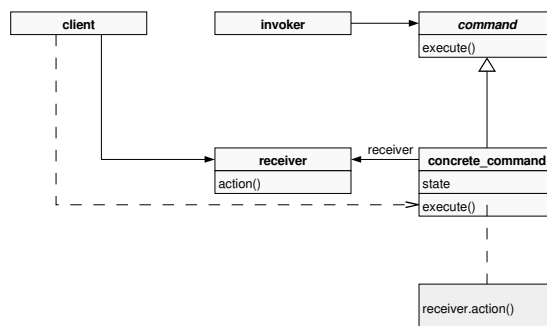


Figure 4.27: Command Pattern

The *Command* pattern [108], also known as *Action* or *Transaction*, sometimes also *Signal*,

encapsulates a command in form of an object. That way, operations can get parameterised; they can be put in a queue, be made undone or traced in a log book. Figure 4.27 shows the structure of the pattern.

Chapter 8 uses the idea of representing operations and algorithms (logic knowledge) as independent models, similar to encapsulated commands.

Wrapper

The *Wrapper* pattern [108] allows otherwise incompatible classes to work together. It can be seen as skin object enclosing (wrapping) an inner core object, to which it provides access. In other words: It adapts the interface of a class which is why Gamma et al. call the pattern *Adapter*. As can be seen in figure 4.28, this pattern makes heavy use of *Delegation*, where the *Delegator* is the adapter (or wrapper) and the *Delegatee* is the class being adapted [148].

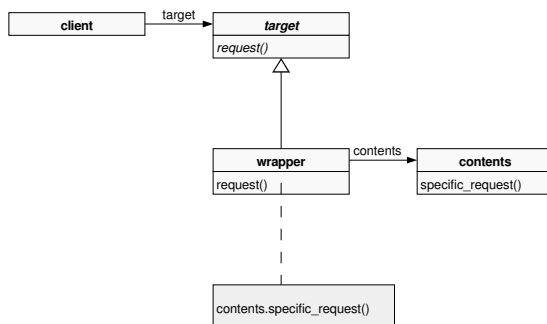


Figure 4.28: Wrapper Pattern

Knowledge templates created in the language described in chapter 9 wrap the more fine-granular templates they consist of.

Whole-Part

Whenever many components form a semantic unit, they can be subsumed by the *Whole-Part* pattern [41]. It encapsulates single part objects (figure 4.29) and controls their cooperation. Part objects are not addressable directly. Almost all software systems contain some components or sub systems which could be organised by help of this pattern. In some way, it is quite similar to the previously introduced *Wrapper* pattern, only that not just one but many objects are wrapped.

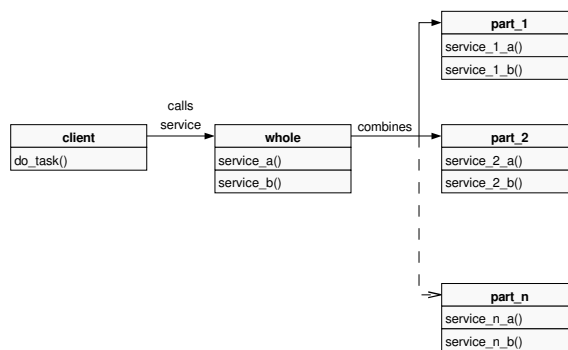


Figure 4.29: Whole-Part Pattern

The principal structure of the new language introduced in chapter 9 is based on the *Whole-Part* pattern. One knowledge template (whole) may consist of zero, one or many other templates (parts).

Composite

A hierarchical object structure, also called *Directed Acyclical Graph* (DAG) or *Tree*, can be represented by a combination of classes called *Composite* pattern [108]. It describes a *Component* that may consist of *Children* (figure 4.30), which makes it comparable to the *Whole-Part* pattern. The difference is that the *Composite* is a more generalised version, with a dynamically extensible number of child (part) objects. The *Composite* is a pattern

based on *Recursion*, which is one of the most commonly used programming techniques at all. The pattern's split into *Leaf*- and *Composite* sub classes helps distinguish primitive- from container objects. A composite tree node holds objects of type *Component*.

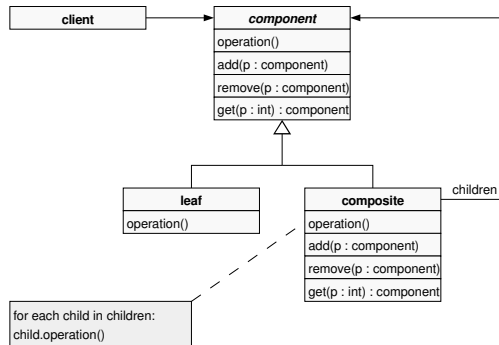


Figure 4.30: Composite Pattern

The knowledge schema introduced in chapter 7 has container capabilities, like the composite pattern. It does, however, not distinguish between composite and leaf nodes, and not use inheritance.

Chain of Responsibility

The *Chain of Responsibility* pattern [108] is similar to the *Composite*, in that it represents a recursive structure as well. Objects destined to solve a task are linked with a corresponding *Successor* (figure 4.31), such forming a chain. If an object is not able to solve a task, that task is forwarded to the object's successor, along the chain.

The pattern found wide application, for example in help systems, in event handling frameworks or for exception handling. Its *Handler* class is known under synonyms like *Event Handler*, *Bureaucrat* or *Responder*. Frequently, the pattern gets misused by delegating messages not only to children but also to the parent of objects. The *Hierarchical Model View Controller* (HMVC) pattern (section 4.2.1) is one example for this. It causes unfavourable

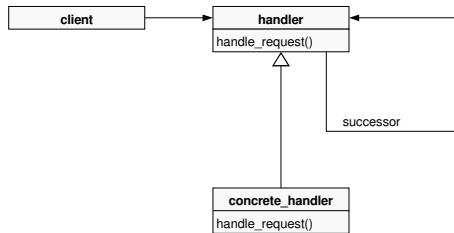


Figure 4.31: Chain of Responsibility Pattern

bidirectional dependencies and leads to stronger coupling between the layers of a framework, because parent- and child objects then reference each other.

Much like state knowledge (data structures) is representable by the knowledge schema being described in chapter 7, also logic knowledge (algorithms, operations) is. A compound logic model may contain further logic models, which it calls or sends as signal in a manner similar to the *Chain of Responsibility* principle.

Observer

Another pattern that found wide application is the *Observer* [108], an often-used synonym for which is *Publisher-Subscriber*. It provides a notification mechanism for all objects that registered as *Observer* at a *Subject* in whose state changes they are interested, leading to an automatic update of all dependent objects (figure 4.32).

Similar notification mechanisms are used for *Callback* event handling in frameworks (section 4.2.4), where the framework core calls functionality of its extensions. The *Model View Controller*- (MVC) (section 4.2.1) uses the *Observer* pattern to let the model notify its observing views about necessary updates (figure 4.33). The disadvantage of the *Observer* pattern is that it relies on bidirectional dependencies, so that circular references can occur,

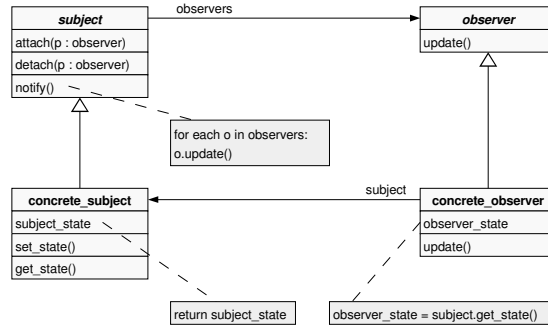


Figure 4.32: Observer Pattern

when a system is not programmed very carefully.

The new pattern systematics presented in chapter 7 classifies the *Observer* as not recommendable pattern. The language and interpreter described in chapters 9 and 10 do avoid its usage.

Bidirectional Dependency

Bidirectional References are a nightmare for every software developer. They cause *Inter-Dependencies* so that changes in one part of a system can affect multiple other parts which in turn affect the originating part, which may finally lead to cycles or even endless loops. Also, the actual program flow and effects of changes to a system become very hard to trace. Therefore, the avoidance of such dependencies belongs to the core principles of good software design.

A *Tree*, in mathematics, is defined as *Directed Acyclic Graph* (DAG), also known as *Oriented Acyclic Graph* [231]. It has a *Root Node* and *Child Nodes* which can become *Parent Nodes* when having children themselves; otherwise they are called *Leaves*. Children of the same node are *Siblings*. A common constraint is that no node can have more than one parent, as [143] writes and continues: *Moreover, for some applications, it is necessary to consider*

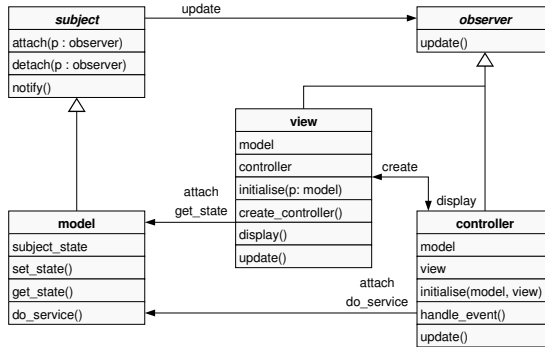


Figure 4.33: MVC- using Observer Pattern

a node's children to be an ordered list, instead of merely a set. A graph is *acyclic* if every node can be reached via exactly one path, which then also is the shortest possible.

In computing, trees are used in many forms, for example as *Process Tree* of an operating system [258, 149, 313] or as *Object Tree* of an object-oriented application. They represent *Data Structures* in databases or file systems and also the *Syntax Tree* of languages. The violation of the principle of the *Acyclic Graph* can lead to the same loops, also called *Circular References*, as mentioned above, which can result in the crossing of memory limits and is a potential security risk. Therefore, one of the main aims in the creation of the new concepts introduced in part II of this work was the avoidance of bidirectional relations.

4.2.3 Idiomatic

An *Idiom* is a pattern on a low abstraction level. It describes how certain aspects of components or the relations between them can be implemented using the means of a specific programming language. Idioms can thus be used to describe the actual realisation of design patterns. Besides the *Counted-Pointer* pattern, Buschmann [41, p. 377] also categorises *Singleton*, *Template Method*, *Factory Method* and *Envelope-Letter* [62] as *Idiom*.

Template Method

The *Template Method* pattern [108], also called *Hook Method*, is an abstract definition of the *Skeleton* of an algorithm. The implementation of one or more steps of that algorithm is delegated to a sub class (figure 4.34).

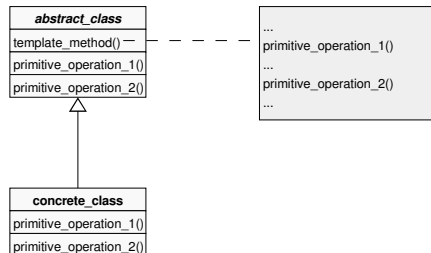


Figure 4.34: Template Method Pattern

The idea of algorithm (method) templates was taken over in the design of the new language described in chapter 9. The single template parts, however, are not inherited but implemented in *part* templates referenced by their corresponding *whole* template, which is actually more similar to the previously described *Whole-Part* pattern (section 4.2.2).

Counted Pointer

The *Counted Pointer* pattern [41] supports memory management in the *C++* programming language, by counting references to dynamically created objects (figure 4.35). That way, it can avoid the destruction of an object through one client, while still being referenced by other clients. Also, it helps avoiding memory leaks by cleaning up forgotten objects in the style of a *Garbage Collector*.

Since all of its knowledge is kept in just one huge tree structure, the interpreter introduced in chapter 10 has memory management and reference counting capabilities built in by default.

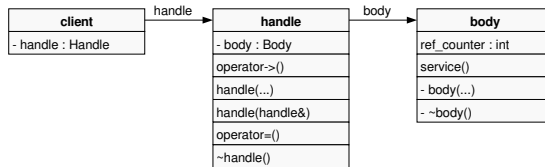


Figure 4.35: Counted Pointer Pattern

Singleton

Whenever an object-oriented system wants to ensure that only one instance of a certain class exists, the *Singleton* pattern [108] can be used. It essentially is a class which encapsulates its instance's data and provides global access to them, via *static*, sometimes called *class* methods (figure 4.36).

A *Registry* object as described by Fowler [101] often uses the *Singleton* pattern, to be unique and to become globally accessible. Similarly do many so-called *Manager* objects, for example change managers which are also responsible for the caching of objects.

Global, that is static access – the main purpose of the *Singleton* pattern, is its main weakness, at the same time. One obvious solution to avoid singleton objects could be to forward global information as instances from component to component, possibly using an own *Lifecycle Method* (section 4.3.2). This approach, however, might bring with a rather large number of parameters to be handed over. It therefore seems easier to use another alternative – the central tree of knowledge instances, as done in the interpreter of chapter 10.



Figure 4.36: Singleton Pattern

Global Access

Statically accessible (manager) classes are often introduced to frameworks (section 4.2.4) because some instances can not be reached anymore along normal object associations. With instance models having a tree structure being directed and acyclic, each object can be reached anytime in a consistent way, by navigating down the tree branches. A pure instance tree in a computer's *Random Access Memory* (RAM) represents an unidirectional structure that permits data access along *well-defined* paths.

Global access via static types, on the other hand, allows *any* instance to address data in memory *directly*, which not only complicates software development and maintenance, but, due to uncontrollable access, is a potential security risk. The usage of static objects accessible by any other part in a system is an *Anti Pattern* to *Inversion of Control* (IoC) (section 4.3.1), highly insecure and hence undesirable. Chapter 7 does therefore classify all patterns using global (static) access as not recommendable. And the language and interpreter introduced in chapters 9 and 10 do avoid their usage.

4.2.4 Framework

In the past decade, *Software Frameworks* have gained in importance. Patterns are considered their elementary building blocks. Yet while patterns are solutions for recurring design problems, frameworks represent the base architecture for a family of systems [252]. Because both concepts depend on each other, frameworks are described within the main section *Patterns*.

A *Framework* essentially is a reusable collection of a number of cooperating abstract and concrete classes, in a special constellation. It represents an incomplete software system which still needs to be extended and instantiated, to be executable. A conventional *Library* is used by calling the procedures provided by it; the main part of each application is then designed and realised by the developer. A *Framework* already represents the actual main part of a system. Functionality added by the application developer is reversely called and used by the framework itself. This principle carries the name *Callback Mechanism*. Extensions are mostly realised through *Inheritance* and *Polymorphism* (section 4.1.15).

But not all parts of a framework are intended to be extended. After W. Pree [252], there are *Static Constraints* and *Dynamic Parts*. Buschmann [41] calls them *Frozen Spots* and *Hot Spots*; the *Apache Jakarta Avalon* framework [17] labels static parts *Contracts*. When the abstract state of a framework is turned into a functioning application by instantiating its classes, static elements remain unchanged. They form the basic structure for all derived applications. Application-specific behaviour, on the other hand, is determined by specialising adaptable framework parts.

The *Jakarta Avalon* documentation [17] defines a framework as:

1. A supporting or enclosing structure.
2. A basic system or arrangement as of ideas.

It distinguishes between *Vertical Market Frameworks* which focused on a single industry like medical systems or telecommunications and would not work well in other industries, and *Horizontal Market Frameworks* which were generic enough to be used across multiple industries. Vertical market frameworks could be built on top of horizontal market frameworks.

Just like patterns, frameworks provide higher flexibility to software components, prevent code duplication and lower development efforts [252]. Developers are freed from frequently

reinventing the same solutions and can concentrate on actual application development. The similar structure of applications that base on the same framework ensures consistency and eases their maintenance, and also reduces the time it takes for a developer to learn how the software works. Of course, the necessary adjustment for new developers should not be underestimated; comprehensive documentation is necessary. But once the principles behind a framework are understood, one will be able to comprehend any system built upon it.

The *Java Development Kit* (JDK) [112], for example, offers a number of special *Collection* containers (section 4.1.15) which it calls *Collection Framework*; there is also an *Input Method Framework* and so on. Over the years, however, the framework definition has become a bit fuzzy here-and-there.

The price of framework reusability is *Lower Flexibility*, which is due to the above-mentioned static parts. Besides this, applications are subject to the evolution of the underlying framework. However, that disadvantage shouldn't be too bold, if the framework is designed general and clever enough.

Framework callback mechanisms rely on *Bidirectional Dependencies* and bring with all their disadvantages (section 4.2.2). To explain this briefly: Instances that want to be called by the framework need to register at a caller before, as was explained in section 4.2.2, on the example of the *Observer* pattern. In order to be able to register themselves, callees need to know about the caller. Once callees are registered, the caller knows about them in turn.

Frequently, statically accessible classes, also called *Managers*, have to be introduced to a framework, mostly due to unforeseen requirements. They often use the *Singleton* pattern (section 4.2.3) to become unique within a system. Managers of that kind serve as gateway to certain areas of the framework that are not easily reachable anymore through normal navigation along object associations. A number of negative effects related to static object access were already mentioned (section 4.2.3).

Chapter 7 introduces a structure called *Knowledge Schema* which, although being static, is capable of representing general knowledge, thus allowing the creation of flexible application systems. Bidirectional dependencies and global model (object) access are not an issue in the new language and interpreter introduced in chapters 9 and 10, because any runtime knowledge model may be accessed along well-defined paths in a simple tree-like structure.

4.3 Component Oriented Programming

In the not-so-distant past, there has been a lot of buzz in the industry touting *Component Based Design* (CBD). Much of it was more marketing than the actual application of new design principles and it is also no surprise that the definition of a *Software Component* differs between authors, companies and projects. A rather formal one is that of the *Avalon* framework [17] of the *Apache Jakarta* project [253], which says:

Component Oriented Programming takes object-oriented programming one step further. Regular object-oriented programming organizes data objects into entities that take care of themselves. There are many advantages to this approach. ... But it also has a big limitation: that of object *Co-Dependency*. To remove that limitation, a more rigid idea had to be formalized: the *Component*. The key difference between a regular object and a component is that a component is completely replaceable (*Black Box Reuse*).

Many kinds of software components exist: There are *Graphical User Interface* (GUI) components known from Delphi's *Visual Component Library* (VCL) [63], from *Visual Basic* (VB) [64] or Java's *Abstract Window Toolkit* (AWT) [112], which calls its components *Beans*; there are *Business* components containing domain knowledge and *Technical* components interacting with infrastructure systems like *CORBA*-/ *EJB*-/ *DCOM* middleware, with a *Database* (DB) or *Operating System* (OS) (sections 3.3, 3.9); there are general *Application* components as investigated by *Avalon* [17]. Whole systems, that is self-acting components, are often called *Agents* (section 4.3.7).

The following sections deal with application components whose principles are investigated on the example of the *Avalon* framework, because that is an *Open Source Software* (OSS) project providing all sources for free. After *Avalon* [17], *Component Oriented Programming* (COP) were not necessarily the same as *Component Based Design* (CBD). The latter referred to how a system is *designed*; the former to how it is *implemented*. In practice, one could not implement COP without first designing with components in mind.

By following CBD rules, software projects try to integrate most diverse systems into one environment. Although the systems should ideally use COP for the implementation of their components, this is not a must. Legacy systems (section 3.10) can be encapsulated as well [36], acting as one component to the outside environment. A *Service* offered by a legacy system written in *PL/1* [89], for example, may be designed and treated as component. It

may be accessed via *Interface Definition Language* (IDL) interfaces and use a *Common Object Request Broker Architecture* (CORBA) for communication. The interna of such a system, however, do neither have to follow component- nor object-oriented programming principles.

The following sections describe popular COP techniques. Although being researched in an own scientific field, *Aspect Oriented Programming* (AOP) principles are added as sub-section, because *Aspects* are comparable to the *Concerns* of COP. For similar reasons, *Agent Oriented Programming* (AGOP) techniques are described following, because *Agents* can be seen as self-acting *Components*. All of them, that is COP's concerns, AOP's aspects and AGOP's agents owning a knowledge base will be recalled once again in part II of this work.

4.3.1 Inversion of Control

The *Avalon* documentation [17] defines a *Component* as a *passive entity that performs a specific role*. According to this definition, a passive entity has to employ a *passive Application Programming Interface* (API) which, after [17], were one that is acted upon, versus one that acts itself.

As could be seen in section 4.2.4, it is often desirable to let a framework play the role of the main program coordinating and sequencing events and application activity. *Observers* (section 4.2.2) are applied to realise a callback mechanism; a *Chain of Responsibility* (section 4.2.2) is often set up among objects so that they can react to certain messages in a delegation hierarchy. Both patterns rely on bidirectional dependencies whose negative effects such as *Strong Coupling* were mentioned before (section 4.2.2).

The *Inversion of Control* (IoC) pattern [17], in a recent article of Martin Fowler also called *Dependency Injection* [100], shows a way out. It is mentioned here (and was not in the pattern section 4.2) because of its importance to COP. The pattern refers to a major semantic detail: a parent object controlling child objects (components) through their passive API, but *not* vice-versa, which results in solely *unidirectional* dependencies. As a side-effect, this principle enforces *Security by Design* in that the flow of control (object access) is always directed *top-to-bottom*. A parent instantiates its child components, hands over important parameters (which were configured by the parent), and then calls the component's methods. Components are not autonomous. They have no state apart from the parent and also have no way of obtaining a reference to implementations of parent parameters without the parent giving them the implementation they need.

Parent objects that have the ability to host child objects are often called *Container*. A container can provide a passive API itself which allows yet other containers to control that container. A hierarchical container example can be found in the *Pico Container* project [56]. The container manages things like loading of configuration files, resolution of dependencies, component handling and isolation, and lifecycle support.

4.3.2 Component Lifecycle

In a component environment, the container and the components living within it have concluded a *Contract* which stipulates that the container is required to take its components through what is called their *Lifecycle*.

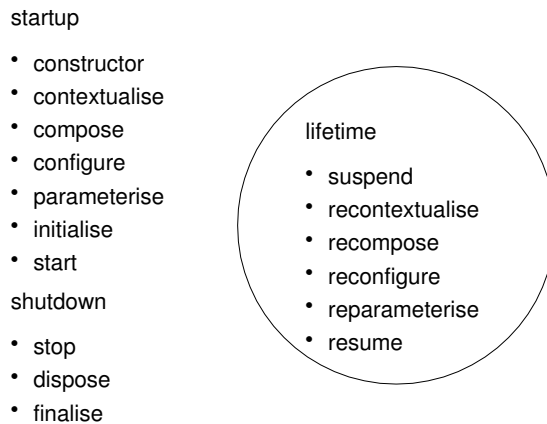


Figure 4.37: Component Lifecycle Methods

The lifecycle of a component specifies the methods that can be called on it, and the order in which this may happen. The corresponding methods are called *Lifecycle Methods*. Some of them can be called only once in a specific phase of a component's lifecycle, others may be called multiple times. The methods in figure 4.37 are examples taken from the *Avalon* [17] project. They represent three phases in a component's lifecycle: *Startup*, *Lifetime* and *Shutdown*.

Lifetime phase methods can be called repeatedly, at various times after startup and before shutdown. The *constructor* is called as a consequence of instantiation. Its counterpart

destructor is not considered in the project; since *Avalon* is a Java-based framework, it was omitted because a *Garbage Collector* destroys instances at some indeterminate moment.

The order in which the various lifecycle methods are called is very specific. While none are required (it is possible to have a component implementing none of the lifecycle methods, although the use of that would be limited), some can only be used when others are as well. It is up to each container to indicate which lifecycle methods it will honour, as [17] writes. This should be clearly documented together with the description of the container.

A component lifecycle allows to forward parameters throughout a whole framework, which makes static objects such as the singleton managers criticised in section 4.2.3 superfluous. The *configure* method, for example, may forward a *Configuration* object containing parameters that otherwise would have to be made global.

4.3.3 Interface and Implementation

The *Separation of Interface and Implementation* is a core concept of many programming languages. *Java*, for example, distinguishes *Interface* and *Class* (section 4.1.15). Mark Grand's book *Patterns in Java* [113] refers to that separation simply as *Interface* pattern, one of whose uses, after him, were to *encapsulate components*, that is to:

- Force decoupling of different components
- Ensure easy changes of interface implementations
- Enable users to read interface documentations without having the implementation details clutter up their perception
- Increase the reuse possibility in larger applications

The Java source code example below shows how the *sayHello* method whose header is declared in the *HelloWorld* interface can be used as *Application Programming Interface* (API) without knowing anything about the underlying implementation. Depending on the instance, the method may be processed on a local or a remote system.

```
package helloworld;
public interface HelloWorld {
    void sayHello(String greeting);
}

package helloworld.impl.default;
```



```
public class DefaultHelloWorld implements HelloWorld {
    void sayHello(String greeting) {
        System.out.println("HelloWorld Greeting: " + greeting);
    }
}

package helloworld.impl.remote;
public class RemoteHelloWorld implements HelloWorld {
    private RemoteMessenger remoteMessenger;
    public RemoteHelloWorld(RemoteMessenger rm) {
        remoteMessenger = rm;
    }
    void sayHello(String greeting) {
        rm.sendMessage("HelloWorld Greeting: " + greeting);
    }
}
```

Further details and recommendations for using interfaces, on examples specific to the *Java Development Kit* (JDK) [112], are given in [17].

4.3.4 Separation of Concerns

The *Avalon* project [17] writes:

Separation of concerns in its simplest form is separating a problem into different points of view. Every time one uses interfaces within object- or component oriented programming, the *Separation of Concerns* (SoC) pattern is applied. The interface separates the implementation concern from the concern of the user of the interface.

Interfaces *pool* common methods (section 4.1.15). Inheriting an interface, components indicate to their surrounding container which methods they implement so that the container can use and rely on these. Therefore, one often talks of a *Contract* between container and component. The contract defines what the container (as user of the component) must provide and what the component produces. In the end, the separation of *Interface and Implementation* (section 4.3.3) could be more correctly called separation of *Contract and Implementation*.

The contract was mentioned in section 4.3.2 which explained that a container is responsible for taking its components through a lifecycle. The *Avalon* project [17] specifies a number of

concerns which enforce the implementation of one or more lifecycle methods. Here is a list of some concerns referring to the methods of section 4.3.2:

- Loggable
- Contextualizable
- Composable
- Configurable
- Initializable
- Startable
- Suspendable

For example, an object that can be configured implements the *Configurable* interface. The contract surrounding that interface is that the container as instantiator of the object passes a *Configuration* object to the component being a configurable object. Just what the configurable object does with the passed configuration object is irrelevant to the instantiator.

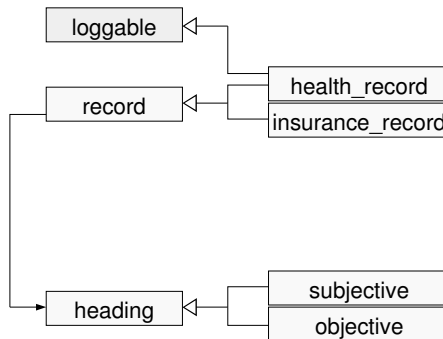


Figure 4.38: Class Inheriting Loggable Concern Interface

Figure 4.38 shows an *Electronic Health Record* (EHR) component implementing the *Loggable* interface which indicates that the component offers functionality for the logging of messages. To fully explain the figure: The EHR inherits from a general *Record* class that references

so-called *Heading* objects which may be a *Subjective* description of a patient or an *Objective* result of an examination. Of course, these objects may be programmed as components as well.

A common comparison used in component oriented programming is that of the *Role* coming from theater [17]. The function or action of a component's role within a system, as well as its contracts, are defined by its script – the interface. Any object that implements the component interface must comply with the contracts. This allows developers to manipulate components using a standard interface, without worrying about the semantics of the implementation. They are separate concerns.

A *Service Manager* is often used to get an instance of the needed component. The manager's *lookup* method identifies the component based on the *Fully Qualified Name* (FQN) of the role (interface). If several components functioning in the same role exist, a *Component Selector* may be applied to choose the right one. More details are given in [17].

4.3.5 Spread Functionality

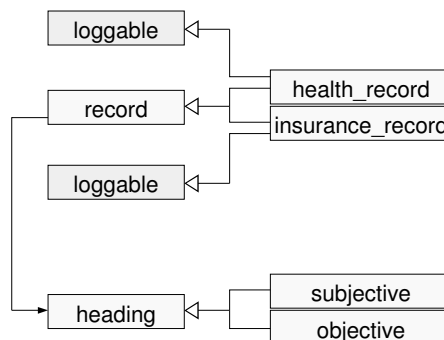


Figure 4.39: Redundant Code through Usage of Concerns

Separating concerns does not avoid *Redundant Code*. If two independent components want to target the same concern what they expose by implementing the corresponding interface,

they will both have to implement the required methods redundantly. This may not be a problem with just two records as in the example of figure 4.39, but will become an issue as soon as other objects are to be programmed as component as well.

Another unwanted effect when using concern interfaces is the *Overlapping* of concerns (figure 4.40). It may happen that a superclass implements a number of lifecycle methods and their corresponding interfaces without knowing if its subclasses eventually implement exactly one of these, too. In such a case, redundant code would appear and the principle of efficient programming would be injured once more.

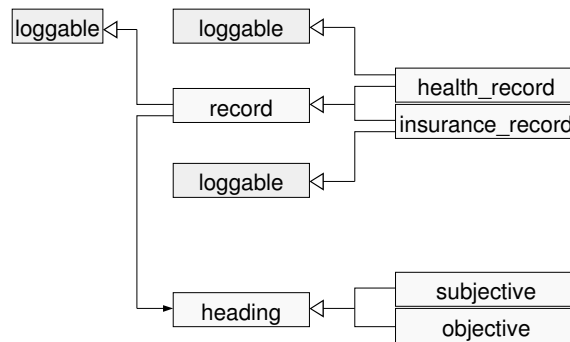


Figure 4.40: Overlapping Code through Usage of Concerns

A piece of source code holding a reference to a component instance in form of a concern interface can *only* call the methods of *that* concern. Mostly, however, other methods have to be called as well. In this case, a *Downcast* from the concern interface to the component class implementing the interface becomes necessary. Yet to be able to downcast, the class (type) to downcast to needs to be known anyway. In the end, the usability of concern interfaces turns out to be limited, sometimes even useless, since they only allow a few methods to be called and information about the real class is not available.

From the viewpoint of reuse, it seems far better to inherit all components from one common super-class, as suggested by the *Layer Supertype* pattern (section 4.2.1). This class would implement necessary lifecycle methods just once, being available for all inheriting classes.

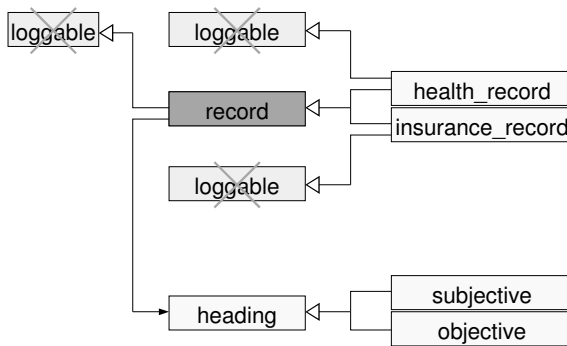


Figure 4.41: Concerns Spread Functionality, an Ontology Bunches it

For the used example this would mean to eliminate all *Loggable* concern interfaces (figure 4.41) and put the logging functionality into the *Record* super class.

Finally, the only way out of the misery of redundant code caused by concern interfaces seems to be *Concern-less* software development using *pure* class hierarchies. And in fact, this is what *Ontologies* (section 4.6.7) are proposing. Section 4.3.4 mentioned that interfaces help *pooling* common methods; but in the big system picture, they actually *spread* them. While concerns represented by interfaces *spread* functionality, away from the classes that were actually made to *keep* it, an ontology *bunches* functionality. Chapter 7 will show some ontology examples and introduce a knowledge schema for their hierarchical representation, including necessary meta information.

4.3.6 Aspect Oriented Programming

Another alternative avoiding redundant code caused by the implementation of concern interfaces (section 4.3.4) is the so-called *Aspect Oriented Programming* (AOP), which is an extension to *Object Oriented Programming* (OOP). *Aspects* are a possibility to separate and define concern areas addressed in program code. Wikipedia [60] writes that:

Aspects emerged out of object-oriented programming and have functionality

similar to using a meta-object protocol (section 4.2.1). Aspects relate closely to programming concepts like subjects, mixins, and delegation.

The Avalon documentation [17] means that AOP were: *the next logical step after separation of concerns*. Many concerns could not be centrally addressed using the standard mechanisms of programming. Using AOP, it were possible to do so in a simple fashion.

The AspectJ documentation [254] writes that the motivation for AOP had been the realisation that there are issues or concerns (like a security policy) that cut across many of the natural units of modularity of an application and are not well captured by traditional programming methodologies. For *Object Oriented Programming* (OOP) languages, the natural unit of modularity were the *Class*. But in these languages, some concerns were not easily turned into classes because they'd *cut across* classes. So these weren't reusable, couldn't be refined or inherited, were spread throughout the program in an undisciplined way and, in short, were difficult to work with. AOP were a way of modularising *Crosscutting Concerns* much like OOP were a way of modularising *Common Concerns*. The later chapter 6 will come back to these two kinds of concerns in short.

As shown in the previous sections, the *Avalon* project [17] uses concern interfaces (sometimes called *Aspect Marker Interfaces*) and *Component Oriented Programming* (COP) to define its concerns, what frequently leads to redundant implementations. Other projects, for instance *AspectJ* [254] and *AspectWerkz* [255], provide AOP facilities whose aim is the clean modularisation of crosscutting concerns such as those in the following list, which AspectJ divides into *Development Aspects*:

- Tracing
- Profiling and Logging
- Pre- and Post-Conditions
- Contract Enforcement
- Configuration Management

and *Production Aspects*:

- Change Monitoring
- Context Passing
- Providing Consistent Behaviour

The AspectJ project as an implementation of AOP for Java adds just one new concept to that language – the *Join Point*, which is a well-defined point in the program flow. A number of new constructs are introduced as well: A *Pointcut* picks out certain join points and values at those points; an *Advice* is a piece of code that is executed when a join point is reached. Both do dynamically affect the program flow. *Inter-Type Declarations*, on the other hand, statically affect a program’s class hierarchy, namely the members of its classes and the relationships between classes. AspectJ’s *Aspect*, finally, is the unit of modularity for crosscutting concerns. It behaves somewhat like a Java class, summarising the constructs described before, that is pointcuts, advices and inter-type declarations.

Aspect Oriented Software Development (AOSD) is about developing programs that rely on AOP principles and languages – or language extensions, respectively. Such programs get compiled slightly differently than usual. An *Aspect Weaver* generates a *Join Point Representation* of the program, by merging code and aspects. Only afterwards, the program is compiled into an executable.

Although AOP seems to successfully address the problem of crosscutting concerns, it also brings with yet another programming paradigm that application developers have to get familiar with. The new concepts and constructs further complicate software development. The source code gets further fractured because of AOP’s additional modules. It is harder to follow the program flow and one has less control on the behaviour of classes, so that the usage of development tools becomes inevitable. But besides this rather general criticism, there is more specific ones: The *Join Point Model* (JPM), for example, is reviewed in [145] which points out some unsolved issues in AOP, while investigating the following properties of join points:

- *Granularity*: only some locations in program code are suitable as join points
- *Encapsulation*: there is no effective way to protect join points from aspect-imposed modification
- *Semantics*: low-level join point identification leads to tight coupling between aspects and join points
- *Jumping Aspects*: context-sensitive join points execute different aspect code
- *Sharing*: dependencies and order of execution are unclear when having several pieces of advice at the same join point

At least some of the concerns that AOP addresses could be implemented with lifecycle-techniques as well. A logger, for example, could be created once at system startup. But

instead of accessing it across static methods, as suggested by the *Singleton* pattern (section 4.2.3), or executing an aspect-oriented *Advice* when a join point is reached, the reference to the logger instance could simply be forwarded from component to component, using a special *globalise* lifecycle method, so that each would be able to access the logger. However, also this solution becomes tedious with a growing number of objects to be forwarded. The new kind of programming introduced in part II of this work therefore suggests to put general functionality (concerns, aspects) into an interpreter program acting close to hardware and providing the general functionality to application systems executed by it.

4.3.7 Agent Oriented Programming

Components created after the principles of *Component Oriented Programming* (COP) are *passive*, because they follow the *Inversion of Control* (IoC) pattern. The functionality, or *Service*, they offer is called by a surrounding container, via a well-defined *Application Programming Interface* (API). *Active* components, on the other hand, act alone. An *Agent* is a self-acting component. It runs *autonomically* or *semi-autonomically*, is *proactive*, *reactive* and *social* [294, p. 330]. Many individual communicative software agents may form a *Multi Agent System* (MAS) [60]. Communication happens by some *Agent Communication Language* (ACL) (section 4.5.3). David Parks, who calls *Agent Oriented Programming* (AGOP) a *Language Paradigm*, writes [245]:

In AGOP, objects known as agents interact to achieve individual goals. Agents can exist in a structure as complex as a global internet or one as simple as a module of a common program. Agents can be autonomous entities, deciding their next step without the interference of a user, or they can be controllable, serving as a mediary between the user and another agent.

In search for a uniform definition of the term *Agent*, Ralf Kuehnel investigated numerous sources of literature but finally comes to the conclusion [183, p. 203] that the term is just a *Metaphor* standing for different properties, depending on the field it is used in. Typically mentioned means of agents, however, are [183, p. 11]: *Distribution*, common *Language* and *Ontology* (section 4.6), *Cooperation* and *Coordination*, *Security* and *Mobility*.

Comparing *Agents* of AGOP with *Objects* known from OOP, Parks [245] writes: *It is not clear, for example, what the concepts of inheritance and dynamic dispatch mean when discussing an agent.* He points out the following significant differences:

- The fields of an agent are restricted. The state of an agent is described in terms of *Beliefs*, *Capabilities* and *Decisions* (*Obligation* / *Commitment*). These ideas are built into the syntax of the language.
- Each message is also defined in terms of mental activities. An agent may engage another (or itself) with messaging activities from a restricted class of categories. In Shoham's formalism [287], the categories of messages are taken from *Speech-Act Theory*; they are: *Informing*, *Requesting*, *Offering*, *Accepting*, *Rejecting*, *Competing* and *Assisting*.

Yoav Shoham, who presented AGOP as a new way to describe intelligent agents [287], suggests that an AGOP system needs three elements to be complete, a:

- *Formal Language* with clear syntax for describing the mental state
- *Programming Language* in which to define agents
- *Method* for converting neutral applications into agents

To the *Mental State* of an agent belong information [183] about its:

- *Environment* (constraints)
- *Expertise* (capabilities) and *Motivations* (aims)
- *Actions* and *Plans*

Tim Finin et al. [87] classify the statements in a knowledge base into two categories: *Beliefs* and *Goals*. After them, an agent's beliefs encoded information it has about itself (capabilities) and its external environment (constraints), including the knowledge bases of other agents. An agent's goals encoded states of its external environment that the agent would act to achieve.

To a running *Agent* system belong the following modules [183]:

- *Knowledge Base*: mental state, as described above
- *Controller*: task controller, scheduler and option selection algorithm
- *Executor*: task runner and security
- *Interaction*: communication handler, sender and receiver
- *Management*: lifecycle manager, startup and shutdown

While early research in AGOP used special languages like Shoham's *Agent0* [287], agent-oriented systems created later were also built upon OOP- and other contemporary programming paradigms [183, p. 237]. Ralf Kuehnel [183] calls *Agent0* alone a *very limited programming language* and takes this as evidence for supporting both, the development of agents and the representation of knowledge with a framework based on OOP principles. For the implementation of this framework, his choice fell on Java as system programming language (section 4.1.7).

That is, although AGOP suggests the separation of a system's *Knowledge* (mental state) from its internal runtime processing and *Control* (agent) and sees them both as separate elements that should be implemented in *different* languages (*formal* vs. *programming*), as mentioned by Shoham (see above), many agent-oriented systems use just *one* language for implementing both. Even if they are kept in different modules, the conceptual differences between *high-level* application knowledge and *low-level* system control cannot be honoured sufficiently. This *Mix-up* puts them on the same level like traditional systems. *Cybernetics Oriented Programming* (CYBOP) as described in this work therefore defines a knowledge modelling language (chapter 9) which is independent from the implementation language of its underlying interpreter.

Furthermore, if OO concepts like *Composition* or *Inheritance* were present in knowledge models, the usage of an OOP language to implement the actual agent system could *not* be justified any longer. In such a case, lower-level *Structured and Procedural Programming* (SPP) languages would suffice, and work much more efficiently. Chapter 10 of this work introduces a knowledge interpreter that is written in the *C* programming language. The interpreter owns a knowledge base keeping all application knowledge, and it has modules for lifecycle management, signal (event) processing, communication etc., just like the definition of an agent (see above) suggests.

4.4 Domain Engineering

Undoubtedly, *Object Oriented Programming* (OOP) (section 4.1.15) is one of the most popular programming paradigms in use today. Its application within a *Software Engineering Process* (SEP) (chapter 2) requires two preliminary phases called *Object Oriented Analysis* (OOA) and *Object Oriented Design* (OOD).

The area of *System Family Engineering* applies a so-called *Six-Pack* approach (figure 4.42)

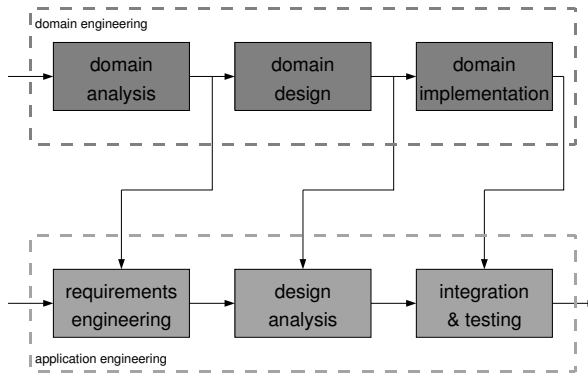


Figure 4.42: Six Pack Model of System Family Development [44, 79]

which is based on the separation of *Domain Engineering* (DE) and *Application Engineering* (AE). The focus of AE is a single system whereas the focus of DE is on multiple related systems within a domain, as [44] defines. Both of them consist of analysis-, design- and implementation phases. The results of each DE phase are *fed in* as foundation for the work to be done in AE. Most of the topics described in the previous sections (4.1, 4.2 and 4.3) turned around techniques which are applicable to both, DE as well as AE.

In Krzysztof Czarnecki's opinion [66], the only SEPs adequately addressing the issue of *Development for Reuse* were DE methodologies, most of which had essentially the same structure. He writes:

Domain Engineering is the activity of *collecting, organizing, and storing* past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable work-products), as well as providing an adequate means for *reusing* these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems.

After him, the main difference between traditional OOA/ OOD- and DE methods were that the former focus on developing *Single* systems, the latter though on developing *Families* of systems. Combined with the definition stated above ([44]), this means that OOA/ OOD methods may be used for application-, but not domain engineering. Different methods and

techniques exist for DE. Many are given in [338, 10, 84]. Just a few of them shall be mentioned here:

- *Feature Oriented Domain Analysis* (FODA)
- *Reuse driven Software Engineering Business* (RSEB)
- *Feature RSEB* (FeatuRSEB)

The RSEB methodology [164] places emphasis on purely *Object Oriented* (OO) techniques which it uses together with the *Unified Modelling Language* (UML). *Features* and *Feature Models* (section 4.4.4), as concept, were introduced by the FODA [46]. The combination of RSEB and FODA results in the FeatuRSEB approach [115]. It permits a separate treatment of domain knowledge and system functionality.

However, this work is less interested in the details of DE software development *Methods*, but rather in their *Knowledge Abstraction*- and *Implementation* techniques. Some of them are investigated in the following sections.

4.4.1 Tool & Material

In software engineering, the term *Domain* stands for a special field of business in which software systems are applied. Frequently, system development methods distinguish between data belonging to the *Domain* and functionality defining the actual *Application* working on the domain. The system family engineering mentioned before is one example.

This view is comparable to the well-known *Tools & Materials* approach [351] which is based on the distinction of *active* applications (tools) working on *passive* domain data (material). *Materials can never be accessed directly, but only by using appropriate tools*, as [351] writes. This simple idea is an important pre-condition for the separate treatment of *System* and *Knowledge*, as explained in chapter 6 of this work.

4.4.2 Generics

Generic Programming received its name from the *Generics* it uses. Wikipedia [60] writes: *Generics is a technique that allows one value to take different datatypes (so-called polymorphism) as long as certain contracts such as subtypes and signature are kept. Templates are*

one technique providing generics. They allow the writing of code without considering the data type that code will eventually be used with. Two kinds of templates exist [60]:

- *Function Template*: behaving like a function that can accept arguments of many different types
- *Class Template*: extending the same concept to classes; often used to make generic containers

Using templates of the C++ *Standard Template Library* (STL) [153], a list may be declared by writing `list<T>`, where *T* represents the type that may be substituted as needed. A linked list of integers, for example, would be created with `list<int>`. After [60], there are three primary drawbacks to the use of templates:

1. Less portable code due to the poor support for templates in compilers
2. Difficult development of templates due to unhelpful error messages produced by compilers
3. Bloated code due to the extra code (instantiated template) generated by compilers

Meanwhile, many other OOP languages like *Eiffel*, *Java*, *VB.NET* and *C#* provide generic facilities. Being used to improve the customisability of code at compile time, they retain the efficiency of statically configured code. However, in practice (own experience of the author) it is often hard for programmers to understand and handle generic techniques. Czarnecki [66], who summarises generic programming as *Reuse through Parameterisation*, criticises that it: *limits code generation to substituting generic type parameters with concrete types and welding together pre-existing fragments of code in a fixed pattern*. *Dynamic Typing* (section 4.1.8) is one possibility to circumvent the need for generic programming. The interpreter program introduced in chapter 10 uses dynamic typing; it references all knowledge via neutral pointers whose meaning gets determined only at runtime.

4.4.3 Domain Specific Language

While a *General Purpose Language* (GPL), no matter if in form of a scripting- or compiled programming language, can be used for performing a variety of different tasks, a (usually declarative) *Domain Specific Language* (DSL), though less comprehensive, is more expressive in a special domain context [323]. After [60], DSLs may: *enhance the productivity, reliability, maintainability, portability and reusability of software*. In Czarnecki's words [66], DSLs:

increase the abstraction level for a particular problem domain and, being highly intentional: allow users to work closely with domain concepts.

Several synonyms are used to label a DSL, for example: *Little Language*, *Application Language*, *Macro* or *Very High Level Language* [60]. To the numerous representatives belong simple spreadsheet *Macros* as well as graph definition languages like *GraphViz*'s *DOT* [186], languages for numerical and symbolic computation as used in *Mathematica* [155], or parser generator languages like *Yet Another Compiler Compiler* (YACC), found on *Universal Interactive Executive* (UNIX) systems. Even UNIX *Shell Scripts* can be considered a DSL, with emphasis on data organisation. Further DSLs exist, yet are the boundaries between the concepts of domain specific- and other languages quite blurry [60].

Martin Fowler [99] mentions that the *Lisp* [227] and *Smalltalk* [202] communities, rather than defining a new language, frequently morph the GPL into a DSL, in a *bottom-up* manner. Such *In-Language* DSLs, as he calls them, use constructs of the programming language itself. Wondering why, programming in *Smalltalk*, he never really felt the need to use a separate language, while, programming in C++/ Java/ C#, quite often he did, he concludes [99] that: *the more suitable languages (are) minimalist ones with a single basic idea that's deeper and simpler than traditional languages (function application for lisp, objects and messages for smalltalk)*, and finds that it is the *friendliness towards in-language DSLs* rather than *static versus dynamic typing* that let many software developers *enjoy programming in Smalltalk or Ruby so much more than in Java or C#*.

Besides their limited usability outside the special domain they were created for, to the problems that the usage of DSLs brings with belong after [211]:

- High cost of designing, implementing, and maintaining a DSL
- Difficult finding of the proper scope
- Difficult balancing between domain-specificity and GPL constructs
- Potential loss of efficiency when compared with hand-coded software

The language introduced in chapter 9 is simple and just because of that flexible enough to be applicable for modelling the knowledge of arbitrary domains. It might have the potential to replace some of the existing DSLs, the investigation of what is out of the scope of this work, though.

4.4.4 Specification Language

A *Specification Language*, after [60], were a formal language used during system analysis and design, as opposed to a *Programming Language*, which were a mostly directly executable formal language used to implement a system. As its name already indicates, a specification language describes systems at a much higher abstract level than a programming language does. But that also means that it: *must be subject to a process of refinement (the filling-in of implementation detail), before it can actually be implemented*, as [60] writes.

Many kinds of specification languages exist. Being a de facto standard, only the first two of those representatives listed following are introduced in slightly more detail below:

- *Unified Modeling Language* (UML) [235]
- *Feature Model* [46]
- *Z Specification Language* [33] and *B Specification Language* [7]
- *Vienna Development Method - Specification Language* (VDM-SL) [325]
- *Specification and Description Language* (SDL) [163]
- *Extended Meta Language* (Extended ML) [279]

Unified Modeling Language

Meanwhile, the probably most famous modelling- and specification language is the *Unified Modeling Language* (UML) [235, 32]. It uses a graphical notation defining a number of diagrams. UML 2.x specifications [235] extend the number of different diagram types from 9 (UML 1.x) to 13. A good overview is given by Ambler in [6], which table 4.1 reproduces in adapted form, showing only *some* diagram elements. The column *Importance* contains a certainly subjective recommendation of Ambler, indicating the *Learning Priority* the single diagram types have in his opinion (which the author of this work supports).

One extension to the UML that is now also part of the corresponding de-facto standard, is the *Object Constraint Language* (OCL). Being a declarative language, it describes rules applying to UML models, in a precise text format. This is because not all rules can be expressed by diagrammatic notation [60]. The range of possible rules comprises constraints like pre- and post-conditions or object query expressions. [312]

A common classification distinguishes UML diagrams as follows [6]:

Diagram	Elements	Importance
Class (CsD)	Class, Inheritance, Association	High
Activity (AD)	Activity, Flow, Fork/ Join, Condition, Decision/ Merge	High
Sequence (SD)	Object, Lifeline, Activation Box (Method-Invocation Box), Message	High
Use Case (UCD)	Use Case, Actor, Association	Medium
State Machine (SMD), formerly State Chart Diagram	State, Transition	Medium
Component (CmD)	Component, Interface, Dependency	Medium
Deployment (DD)	Node, Connection	Medium
Object (ObD), also referred to as Instance Diagram	Object, Relationship	Low
Package (PD)	Package, Dependency	Low
Communication (CoD), formerly Collaboration Diagram	Object, Association	Low
Composite Structure (CSD)	Collaboration, Object, Role	Low
Interaction Overview (IOD)	Interaction Frame, Interaction Occurrence Frame	Low
Timing (TiD)	Object, Lifeline, State, Timing Constraint	Low

Table 4.1: UML 2.x Diagram Types [6]

1. *Structure*: CsD, CmD, CSD, DD, ObD, PD
2. *Behaviour*: AD, SMD, UCD
3. *Interaction*: CoD, IOD, SD, TiD

Others share the information represented by the diagrams according to an underlying, independently existing model [60]:

- *Functional Model* (UCD): Functionality of the system from the user's point of view
- *Object Model* (CsD): Structure and substructure of the system using objects, attributes, operations, and associations
- *Dynamic Model* (AD, SD, SCD): Internal behaviour of the system

A program working with UML diagrams is called *UML Tool*, or more exactly *Computer Aided Software Engineering* (CASE) tool. Many of these programs have developed and

matured, over the past decade of years. Besides the standard UML diagram types, they offer source code parsing and -generation, documentation creation and more. Some tools introduced their own extensions to the UML de-facto standard, for example: *Object Process Diagram* (OPD) [40] and *Entity Relationship Diagram* (ERD) [152]. The description of a hypothetic design tool suggested for the language being introduced in chapter 9 will refer back to the UML diagrams as mentioned in this section, and suggest a different way to categorise them. Further, chapter 9 will try to define four diagram types to be used in conjunction with the language described in it.

Feature Model

Czarnecki [66] sees *Feature Modelling*, a technique for analysing and capturing *common* and *variable* features of a family of systems as well as their inter-dependencies in form of a *Feature Model*, as the main contribution of domain engineering to OOA/ OOD methods. The *System Families*, also called *Software Product Lines*, whose development feature models shall support, are described by Kai Boellert [30] as *group of software systems that are developed from a common set of reusable components*. Czarnecki writes:

Feature Models represent the configurability aspect of reusable software at an abstract level, i.e. without committing to any particular implementation technique such as inheritance, aggregation, or parameterized classes. Developers construct the initial models of the reusable software in the form of feature models and use them to *guide* the design and implementation (also called *Feature-driven Design*). To a reuser, on the other hand, feature models represent an *overview* of the functionality of the reusable software and a guide to *configuring* it for a specific usage context.

In other words, a feature model (figure 4.43) is an additional form of abstraction within a *Software Engineering Process* (SEP), placed between analysis- and design models. The properties contained in a feature model are structured hierarchically. In the *Feature Oriented Domain Analysis* (FODA) [46], a feature model distinguishes three kinds of features: *Context*, *Representation*, *Operational*. Detlef Streitferdt [299] defines five feature types:

1. *Functional*: used by customer to compose a system
2. *Interface*: describe required and provided component interfaces
3. *Parameter*: used to configure functional features

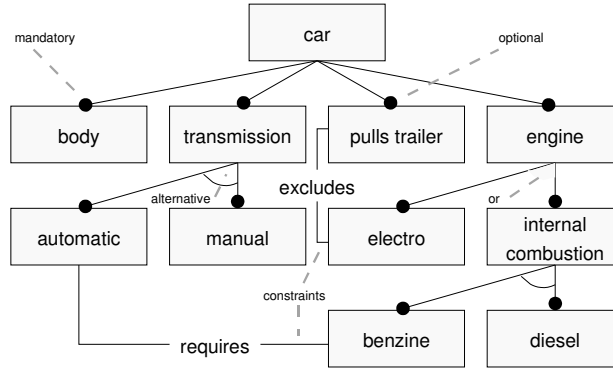


Figure 4.43: Classical Feature Model Diagram of a Car (based on [246])

4. *Structural*: relevant for an automated choice of components

5. *Conditional*: summarise sub-features to improve readability

Using feature models, the *Traceability* between concrete requirements and architecture components can be improved. Requirements can better be mapped to architecture elements, so that also the *Communication* between stakeholders in the development process can profit. The big abstraction gap (number 1 in figure 2.6 of section 2.6) gets split into two smaller (1a and 1b in figure 2.6) that do not close the gap conclusively, but make it easier to cross. The disadvantage of using feature models in a SEP, however, is that another abstraction gap causing additional effort is created through them.

The knowledge schema and language introduced in chapters 7 and 9 use a hierarchical structure comparable to the feature model. Their elements, though, do belong to just one of two possible kinds: *whole-part* model or *meta property* model. CYBOP knowledge models merge *some* of the information that would traditionally be found in feature models with that contained in the design diagrams and might thus be able to eliminate gap 1b. Non-functional requirements like *Performance*, *Scalability*, *Usability* or *Memory Efficiency* are *not* part of a CYBOP knowledge model, since they have nothing to do with the actual modelling of real-world items in form of abstract concepts and belong into a corresponding analysis- and specification document only.

4.4.5 Generative Programming

Generative Programming (GP), as proposed by Czarnecki [66], is a *comprehensive software development paradigm to achieving high intentionality, reusability, and adaptability without the need to compromise the runtime performance and computing resources of the produced software*. It encompasses techniques of the following, previously described paradigms:

- *Aspect Oriented Programming* (AOP) (section 4.3.6): used to achieve separation of concerns
- *Generic Programming* (section 4.4.2): used to parameterise over types
- *Domain Specific Language* (DSL) (section 4.4.3): used to improve intentionality, optimisation and error checking of program code
- *Feature Model* (section 4.4.4): used as configuration knowledge, to map between problem- and solution space

Czarnecki's work contributes to the formal specification and extension of the *Feature Model*, but does not itself deliver new forms of knowledge abstraction. GP, however, is mentioned here because of its idea of applying *Generators* (or generative techniques) producing implementation source code for a software system from the higher-level specifications defined in the design phase. Similar techniques are used in the *Model Driven Architecture* (see next section). GP is a trial to automate the process of crossing abstraction gap number 2 (with reference to figure 2.6), and it is often quite successful. However, the gap between architecture design models and program source code remains. Chapter 7 will introduce a *Knowledge Schema* serving as universal type, so that differing type-based architectures do not have to be designed anymore.

4.4.6 Model Driven Architecture

The *Model Driven Architecture* (MDA) [236] (figure 4.44), an approach to application design and implementation [35] specified by the *Object Management Group* (OMG), represents a suite of key standards including:

- *Unified Modeling Language* (UML): modelling, visualising and documenting the structure and behaviour of systems using graphical diagrams

- *Meta Object Facility (MOF)*: representing and manipulating meta models using CORBA and its *Interface Definition Language (IDL)*; UML can be expressed in terms of MOF, which is done to generate XMI
- *XML Metadata Interchange (XMI)*: interchanging UML metamodels and models using an *Extensible Markup Language (XML)*-based format
- *Common Warehouse Metamodel (CWM)*: enabling data mining across database boundaries at an enterprise using a complete, comprehensive metamodel; does for data modelling what UML does for application modelling
- *Common Object Request Broker Architecture (CORBA)*: communicating using a programming language-, operating system- and vendor-independent middleware platform

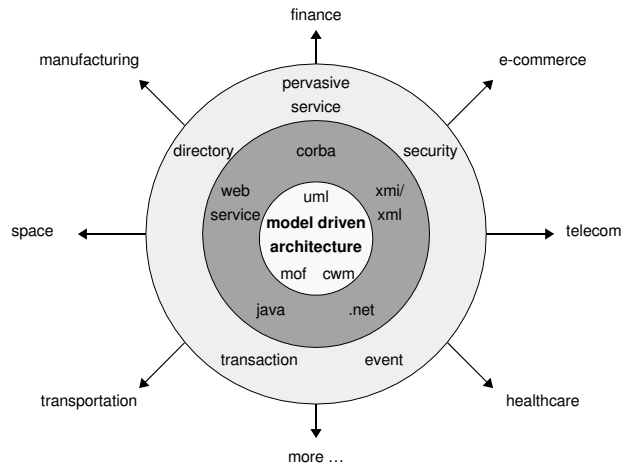


Figure 4.44: Model Driven Architecture [236]

Brown [35] writes: *MDA encourages efficient use of system models in the software development process, and it supports reuse of best practices when creating families of systems.* In the OMG's own words, MDA is a: *way to organise and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types.* It aims at providing an: *open, vendor-neutral approach to the challenge of business- and technology change.*

While traditional approaches like *System Family* or *Tools & Materials* (section 4.4.1) merely distinguish between *Domain* and *Application*, the conceptual framework provided by the

MDA takes another step in separating abstract knowledge: It treats *Platform Independent Models* (PIM), that is business- or application logic, different than the underlying *Platform Specific Models* (PSM), that is implementation technology. The translation between the two kinds of models is normally performed using automated tools for code generation (section 4.4.5).

The MDA claims to overcome the limitations of implementation technology-dependent *Computer Aided Software Engineering* (CASE) tools via standardised mappings and meta architectures à la MOF [102]. After Martin Fowler [99], one argument used in favor of MDA were that it makes it possible to use *Domain Specific Languages* (DSL) (section 4.4.3). However, he doubts a success of the MDA. And indeed, although some MDA standards like the UML are very sophisticated and widely used, it is still unclear whether the MDA, due to its complexity, will be able to infiltrate daily software business.

But the idea of separating *Application Knowledge* (PIM) from its hardware-close *Control and Processing* (PSM) clearly brings a new quality into software development and is important for later investigations in this work (chapter 6).

4.4.7 Model and Code

The knowledge abstraction- and implementation techniques considered in the previous sections belong to the current state-of-the-art in software design and -modelling, with focus on *Domain Engineering* (DE). Despite some exceptions like the *Feature Model* (section 4.4.4), which supports the mapping between abstractions of the analysis- and design phase, most described techniques deal with bridging the gap between design models and source code.

Alan Brown writes [35]: *One useful way to characterise current practice is to look at the different ways in which the models are synchronized with the source code they help describe.* Figure 4.45 shows the spectrum of modelling approaches in use today. The different categories [35] are:

Code Only

- almost entire reliance on the code
- informal and intuitive modelling of architectural designs
- models living on whiteboards, in presentations or the developers' heads

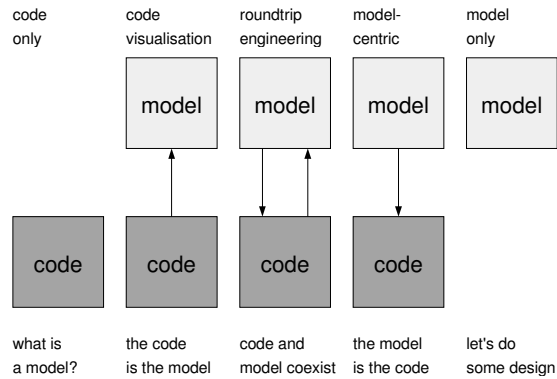


Figure 4.45: Model-Code Synchronisation [35, diagram by John Daniels]

- possible use of an *Integrated Development Environment* (IDE)

Code Visualisation

- alternative modelling using a graphical notation
- diagrams aid the understanding of the code's structure or behavior
- visual renderings become a direct representation of the code
- simultaneous display of code view and model view using *Computer Aided Software Engineering* (CASE) tools

Roundtrip Engineering

- bidirectional exchange between abstract design model and implementation code
- manual model-to-code transformation, and vice-versa
- frequent iterations as errors are detected
- considerable discipline necessary to keep models synchronized
- automated recognition of generated versus user-defined code by *Roundtrip Engineering* (RTE) tools (for example by placing markers in the code)

Model Centric

- sufficiently detailed system models enable the generation of full system implementations
- models include representations of persistent- and non-persistent data, business logic, presentation elements and more
- interfaces to legacy systems and various services
- specialized tools generate particular (constrained) styles of applications, in an automated process

Model Only

- models aid the understanding of a business domain, or the analysing of a proposed architecture
- models used as basis for discussion, communication and analysis among project teams within- or across organisations
- establishment of a shared vocabulary and set of concepts among disparate teams
- model-disconnected (outsourced) implementation of systems

Considering the developments of the last decades but especially recent years, the modelling trend clearly goes *left-to-right*, with respect to figure 4.45, that is from *Code only*- to more *Model-centric* approaches. The emerge of the *Model Driven Architecture* (MDA) (section 4.4.6) is one sign therefore. Yet although these efforts certainly contribute to easier and faster development, less inter-dependencies within systems, better documentation and clarity of source code, improved maintenance and more – the gap between *Design*- and *Implementation* phase, within a *Software Engineering Process* (SEP) (chapter 2), remains. By introducing a new knowledge schema (chapter 7), this work wants to conclusively close gap number 2 (with respect to figure 2.6 of section 2.6) and provide a model-only approach.

4.5 Knowledge Engineering

In order to correctly perform information input, memorising, processing and output, systems need to know about the structure of the data they are processing. While pure *Data*

abstract the real world in form of (mostly machine-readable) characters (quality) and numbers (quantity) (more on this in chapter 7), their interpretation in a semantic context may yield *Information*. What coincides across the many different definitions [60] of the term *Information*, are two statements: it has to be recognisable and: it has to contain something new. Organised information available in form of specific structures with inter-relations is often called *Knowledge*.

Quite often, knowledge gets shared into two kinds: *explicit* (codified) and *implicit* (tacit). While the former, after [221], referred to knowledge that is transmittable in formal, systematic language, the latter had a personal quality, which made it hard to formalise and communicate. This work is about explicit knowledge; it wants to provide concepts and schemata for expressing it as completely as possible (part II).

Over the years, many different *Knowledge Representation* models have been proposed. In software engineering, they represent different kinds of user interfaces (textual, graphical, web), workflows, persistent- or transient data. The biggest importance, however, models experience when representing a complete *Domain* (section 4.2.1), that is the special business field an application system was developed for. After John F. Sowa [294], *Knowledge Engineering* (KE) could be defined as: *the branch of engineering that analyses knowledge about some subject and transforms it to a computable form for some purpose*.

It has to be mentioned that the borders between *Domain Engineering* (DE) (section 4.4) and *Knowledge Engineering* are quite blurry. The *Feature Model* (section 4.4.4), although described as part of DE, can of course also be seen as one form of knowledge representation, just like many other models. Both, DE as well as KE specify formal languages and investigate how different models can be translated into each other. This work, however, makes a split between DE and KE, because:

- their topics are usually treated as different fields of science
- KE's focus is almost exclusively on representing (domain) knowledge in models
- DE also considers their relation to the applications working on them
- DE distinguishes models belonging to different phases in a *Software Engineering Process* (SEP)
- DE describes concrete implementation techniques

The following sections try to give a brief overview of the rather wide field of knowledge representation and -engineering, raising only a few topics. Their main intention is to show

the existence of two different kinds of knowledge: *Date and Rule*, both of which can be structured according to an ontology, what will be dealt with in the later chapters 7 and 8.

4.5.1 Representation Principles

Randall Davis, Howard Schrobe and Peter Szolovits (1993), cited by Sowa in [294, p. 134], summarise their review of the state-of-the-art in knowledge engineering in form of five *Basic Principles*. To them, a knowledge representation is a:

1. *Surrogate*: symbols and the links between them, which form a model that simulates a system, serve as surrogates of physical items → chapter 6 will distinguish between virtual models (symbols) and real world (physical) items
2. *Set of ontological commitments*: an ontology determines the categories of things that may exist in an application domain → chapter 7 will introduce a knowledge schema permitting to apply an ontological structure to data
3. *Fragmentary theory of intelligent reasoning*: a description of the behaviour and interactions of things in a domain supports reasoning about them → chapter 8 will separate state- and logic (behavioural) knowledge
4. *Medium of human expression*: a language facilitates communication between knowledge engineers and domain experts → chapter 9 will define a language for knowledge specification
5. *Medium for efficient computation*: encoded knowledge ensures efficient processing on computing equipment → chapter 10 will describe a low-level interpreter that processes high-level knowledge

4.5.2 Date and Rule

Two kinds of systems that gained greater popularity are *Expert Systems* and *Relational Databases*. After Sowa [294], both differed more in quantity than in quality: *Expert systems use repeated executions of rules on relatively small amounts of data, while database systems execute short chains of rules on large amounts of data*. Over time, their differences decreased and today, the *Structured Query Language* (SQL) for relational databases supports the same logical functions as early expert systems.

Both, expert systems and relational databases, have common logical foundations and store data in a subset of logic called *Existential Conjunctive* (EC) logic. EC is based on two logical operators: the *Existential Quantifier* \exists and the *Conjunction* \wedge ; the *Universal Quantifier* \forall and other operators (\sim , \supset , \vee) are never used. Sowa [294, p. 163] writes: *While variables in a query are governed by existential quantifiers, those in a rule are governed by universal quantifiers.*

The two primary inference rules of the above-mentioned systems are called *Modus Ponens* (putting) and *Modus Tollens* (taking away). Although being simple, the power of these rules comes from their combination and repeated execution. While repeated execution of modus ponens is called *Forward Chaining*, that of modus tollens is called *Backward Chaining*. In SQL, an implication used in backward chaining is called *View*, and that used in forward chaining is called *Trigger* [294].

Besides *Prolog* (section 4.1.10) and *SQL* (section 4.1.11), the *Microplanner* language [294, p. 157] uses the so-called *Backtracking* technique to answer a query: If one of a sequence of aims cannot be satisfied, the language tracks back to a previous aim and tries a different option. Although equivalent queries in Prolog and SQL differ in their syntax, the semantics is the same. *Logic determines the structure of a query*, as Sowa [294, p. 159] means.

To sum up, one can say that previous sections distinguished between *Domain-* and *Application Models*. What was shown in this section, however, is that many systems and their corresponding languages rely on a separation of *Data* (in state variables) and *Rules* (logic). This will be of importance in chapter 8.

4.5.3 Agent Communication Language

A whole palette of languages was suggested within the scientific field of *Artificial Intelligence* (AI). *Agent Oriented Programming* (AGOP) (section 4.3.7), for example, uses representation formats like the ones described following, for the knowledge bases and communication of its agent systems. That is why such formats are often labeled *Agent Communication Language* (ACL).

Knowledge Interchange Format

The *Knowledge Interchange Format* (KIF), as described in [111], is:

- a language designed for use in the interchange of knowledge among disparate computer systems
- not intended as a primary language for interaction with human users
- not intended as an internal representation for knowledge within computer systems
- in its purpose, roughly analogous to *PostScript* (PS) (section 4.1.13)
- not as efficient as a specialised representation for knowledge, but more general and programmer-readable

The idea behind KIF is that [111]: *a computer system reads a knowledge base in KIF, (and) converts the data into its own internal form (pointer structures, arrays, etc.). All computation is done using these internal forms. When the computer system needs to communicate with another computer system, it maps its internal data structures into KIF.* KIF's design is characterised by three features:

1. *Declarative Semantics*: independent from specific interpreters, as opposed to e.g. *Prolog*
2. *Logically Comprehensive*: may express arbitrary logical sentences, as opposed to *SQL* or *Prolog*
3. *Meta Knowledge*: permits the introduction of new knowledge representation constructs, without changing the language

The following syntax example [111] shows a logical term involving the *if* operator. If the object constant *a* denotes a number, then the term denotes the absolute value of that number:

```
(if (> a 0) a (- a))
```

The language introduced in chapter 9 may not only serve as interchange format between systems, but also for the definition of user interfaces, workflows and domain models, altogether. It treats state- and logic models as separate, composable concepts (chapter 8), which KIF does not. Further, it provides the means to express meta knowledge.

Knowledge Query and Manipulation Language

The *Knowledge Query and Manipulation Language* (KQML) [87] is a: *language and associated protocol by which intelligent software agents can communicate to share information and*

knowledge, as Tim Finin et al. [88] write. Its syntax were based on a balanced parenthesis list, because initial implementations had been done in Common Lisp (CL) [227]. After Finin et al., the initial element of the list were the *Performative* and the remaining elements were the performative's *Arguments* as keyword/ value pairs. The Free Wikipedia Encyclopedia [60] explains:

The KQML message format and protocol can be used to interact with an intelligent system, either by an application program, or by another intelligent system. KQML's *Performatives* are operations that agents perform on each other's *Knowledge* and *Goal* stores. Higher-level interactions such as *Contract Nets* and *Negotiation* are built using these. KQML's *Communication Facilitators* coordinate the interactions of other agents to support *Knowledge Sharing*.

An example message representing a query about the price of a share of IBM stock might be encoded as [88]:

```
(ask-one
:content (PRICE IBM ?price)
:receiver stock-server
:language LPROLOG
:ontology NYSE-TICKS)
```

System communication and its elements like *Sender*, *Receiver*, *Language* or *Message Content* will be further investigated in chapter 8. The new language introduced in chapter 9 defines communication operations (logic) accompanied by properties (meta information), much the same way performatives have arguments. Also, that new language may not only be used to encode knowledge for communication, but to represent knowledge of arbitrary domains. By combining pre-defined, primitive operations, it may be used to create more complex (higher-level) algorithms.

DARPA Agent Markup Language / Ontology Inference Layer

The DAML+OIL language resulted from a combination of the DAML and OIL languages. The *DARPA Agent Markup Language* (DAML) [114] was created in a project run by the *Defense Advanced Research Projects Agency* (DARPA) of the *United States of America* (USA); the *Ontology Inference Layer* (OIL) was created within the *Information Science Technologies* (IST) program of the *European Union* (EU) [332]. Both projects aimed at

developing a language and tools to facilitate the concept of the *Semantic Web* (section 4.5.4).

At the beginning of the project stood the realisation that: *The use of ontologies (section 4.6) provides a very powerful way to describe objects and their relationships to other objects.* The DAML+OIL language, being developed as an extension to the *Extensible Markup Language* (XML) (section 4.1.12) and the *Resource Description Framework* (RDF) (section 4.5.4), therefore provided a [348] rich set of constructs with which to create ontologies and to markup information so that it becomes machine-readable and understandable. Much of the work in DAML and OIL has now been incorporated into OWL (section 4.5.4).

Chapter 9 will introduce a language that is based on XML, too.

4.5.4 Semantic Web

As mentioned in section 4.1.12, the *Extensible Markup Language* (XML) [332] provides a: *set of rules for creating vocabularies that can bring structure to both documents and data on the Web* and it: *gives clear rules for syntax.* XML Schemas [295] served as: *a method for composing XML vocabularies.* Yet although XML were a powerful, flexible surface syntax for structured documents, it imposed no *Semantic Constraints* on the *Meaning* of these documents. Having investigated the usefulness of XML for a *meaningful* sharing of information units at the semantic level, Robin Cover writes [65]:

... the use of XML for *Data Interchange* may already outweigh its use for *Document Display*. For messaging and other transaction data, specifications approaching the level of formal semantics (e.g. KIF or KQML) are desirable, governing not just common (atomic) data types in business objects, but complex objects used by computer agents in large-scale business transactions. XML vocabularies supporting these applications will need to be defined in terms of precise object semantics.

He lists a number of efforts dealing with the support for generic XML semantics, that is *Semantic Transparency* of XML in a broader sense, to provide unambiguous semantic specification:

- *XML Data* [191]
- *Document Content Description* (DCD) for XML [34]

- *Schema for Object Oriented XML* (SOX) [69]
- *XML Metadata Interchange* (XMI) [236]
- *Resource Description Framework* (RDF) [348]
- *Web Ontology Language* (OWL) [349]

The RDF and OWL as well-known efforts are mentioned in the following two subsections. Both are often comprised under the umbrella term *Semantic Web*. Much of what is written about the semantic web sounds as if it was a replacement technology for the Web as known today. Yet Eric Miller, leader of W3C's semantic web activity, means [332]:

In reality, it's more Web Evolution than Revolution. The Semantic Web is made through incremental changes, by bringing machine-readable descriptions to the data and documents already on the Web. XML, RDF and OWL enable the Web to be a global infrastructure for sharing both, documents and data which make searching and reusing information easier and more reliable as well.

Resource Description Framework

The *Resource Description Framework* (RDF) [348] as part of the *Semantic Web* provides a standard way for simple descriptions to be made. It is: *a simple data model for referring to objects (resources) and how they are related. An RDF-based model can be represented in XML syntax.* [60]

RDF wants to achieve for *Semantics* what XML has achieved for *Syntax* – to provide a clear set of rules for creating descriptive information. Both follow a special schema, *RDF Schema* [348] and *XML Schema* [295], respectively, which defines the structure and vocabulary that may be used in the corresponding documents.

Many applications that use XML as syntax for data interchange, may apply the RDF specifications to better support the exchange of actual knowledge on the web. The RDF data framework is used [332] in: asset management, enterprise integration and the sharing and reuse of data on the web. Example applications combining information from multiple sources on the web [332] include: library catalogs, world-wide directories, news- and content aggregation, collections of music or photos.

In the words of Brian McBride [332], chair of the RDF core working group, his group had: *turned the RDF specifications into both a practical and mathematically precise foundation*

on which OWL and the rest of the semantic web can be built.

Chapter 9 will come back to RDF once more, and compare it with the new language then introduced.

Web Ontology Language

The *Web Ontology Language* (OWL) is [349]: *a semantic markup language for publishing and sharing ontologies on the world wide web . . . which delivers richer integration and interoperability of data among descriptive communities*. It uses *Uniform Resource Indicators* (URI) for naming and is an extension of the *Resource Description Framework* (RDF), adding more vocabulary for describing properties and classes, for example relations between classes, cardinality, richer typing of properties, or enumerated classes. OWL was originally derived from the *DARPA Agent Markup Language + Ontology Inference Layer* (DAML+OIL) web ontology language (section 4.5.3).

In the understanding of OWL, an ontology is a subject- or domain specific vocabulary which defines the terms used to describe and represent an area of knowledge [332]. However, there are other definitions of the term *Ontology* which are given in section 4.6. OWL aims to add to ontologies capabilities like [332]:

- Ability to be distributed across many systems
- Scalability to web needs
- Compatibility with web standards for accessibility and internationalisation
- Openness and extensibility

It introduces keywords for the use of *Classification*, *Subclassing* with *Inheritance* and further abstraction principles. RDF is neutral enough to permit such extensions. Also the language introduced in chapter 9 may be extended with meta properties, such as one for inheritance.

4.6 Conceptual Network

John F. Sowa [294] cites the computer science pioneer Alan Perlis who, being asked whether a computer could automatically write programs from informal specifications, replied: *It is not possible to translate informal specifications to formal specifications by any formal algorithm*. And Sowa writes on: *English syntax is not what makes the translation difficult*.

The difficulty results from the enormous amount of background knowledge that lies behind every word.

The structuring of such knowledge is what *Ontologies* shall support. They are the topic of the following sections and will be of importance for the knowledge schema introduced in chapter 7.

4.6.1 Ontos and Logos

The word *Ontology* stems from ancient Greek language, consisting of the two subterms *Ontos* and *Logos* which literally mean *Stone* (in the meaning of *Being*) and *Word* (in the meaning of *Study*). Thus, ontology designates *the study of the nature of reality*.

Manifold, more detailed definitions are given in literature. They mostly relate to one of the subjects, *Philosophy* or *Information Technology* (IT). A philosophical one that can be found in Smith and Welty [289] says: *Ontology is the science of what is, of the kinds and structures of objects, properties, events, processes and relations in every area of reality*. Since what it means for something *to be* or *to be real* were an issue beyond what is physically accessible, as Daniel [68] writes, ontological questions were *metaphysical*. *Metaphysics* included not only the study of being and reality but also *the study of specific kinds of beings*, such as minds. *Metaphysics* in general and ontology in particular were both interested in providing a *Logos*, a rational explanation for existence. The Dictionary of Philosophy of Mind [78], as further source, states:

Although the term terms *Ontology* and *Metaphysics* are far from being univocal and determinate in philosophical jargon, an important distinction seems often enough to be marked by them. What we may call ontology is the attempt to say what entities exist. *Metaphysics*, by contrast, is the attempt to say, of those entities, what they are. In effect, one's ontology is one's *List* of entities, while one's metaphysics is an explanatory theory about the *Nature* of those entities.

Besides rather philosophical descriptions, Eric Little [199] also quotes a more information science-like definition of Gruber [118] for whom an ontology is an: *explicit specification of a conceptualization* (of a domain), in other words a *formalisation of domain knowledge*. For the *Ontology Forum* [116], the key ingredients that made up an ontology were a *vocabulary of basic terms* and a *precise specification of what those terms mean*. The *Agent Communication Languages* (ACL) and *Semantic Web* technologies, introduced in sections 4.5.3 and 4.5.4,

respectively, use ontologies in the same meaning. The borders to *Terminology* (section 4.6.5) are often blurry.

The knowledge schema and new language of chapters 7 and 9 may represent entity information (an ontology) as well as meta information about these (metaphysical explanations). However, in order to avoid conflicts with philosophy, this work sticks to Gruber's definition of the term *Ontology*, for the time being, until it defines it in its own way, in chapter 7.

4.6.2 Applicability

The *Ontology Forum* [116] writes that ontologies find applicability in many areas of information systems engineering, for example in database design, in object systems, in knowledge based systems and within many application areas such as datawarehousing, knowledge management, computer supported collaborative working and enterprise integration. Depending on the nature of the knowledge they were concerned with, communities would differ:

- *Artificial Intelligence* (AI): ontologies capture domain knowledge, while problem-solving methods capture task knowledge
- *Natural Language*: ontologies characterise word meaning and sense
- *Database*: ontologies, as conceptual schema, provide semantic inter-operability of heterogeneous databases
- *Object Oriented Design Methods*: ontologies, as domain models, specify software systems that need not be knowledge-based

Sections 4.5.3 and 4.5.4 mentioned the use of ontologies for semantic-based information retrieval. What (conceptually) unites these communities, is the ability of ontologies to reduce semantic ambiguity for the purpose of sharing and reusing knowledge, to achieve inter-operability. In the context of this work, ontologies are mainly used to structure domain knowledge meaningfully, in levels of growing granularity, with unidirectional relations from higher-level layers to layers of lower granularity (chapter 7).

4.6.3 Two Level Separation

Although there appears to be no standard knowledge classification, a *Two Level Separation* of ontologies is often described, as for example in [118]:

At the *First Level*, one identifies the basic conceptualizations needed to talk about all instances of ... some kind of *Process*, *Entity* etc. For example, the first level ontology of *Causal Process* would include terms such as *Time Instants*, *System*, *System Properties*, *System States*, *Causes that change States*, *Effects* (also *States*) and *Causal Relations*.

At the *Second Level*, one would identify and name different types of (a process) and relate the *Typology* to additional constraints on or types of the concepts in the first-level ontology. For the causal process example, we may identify two types of causal processes, *Discrete Causal Processes* and *Continuous Causal Processes* and define them as the types of process when the time instants are *discrete* or continuous, respectively. These terms and the corresponding conceptualizations are also parts of the ontology of the phenomenon being analyzed. Second-level ontology is essentially open-ended: that is, new types may be identified any time.

The *Design Principles for the EHR* document [20] writes that a separation of this kind divided knowledge types into a *Foundation Level* (or what is called an *Ontology of Principles*) which could be numbered *Level 0* and *Everything else*. Knowledge in the latter category were more specific to particular uses and users. It could be divided into a number of sub-levels (according to various types of use) which could be numbered as *Level 1* to *Level N*. Concepts in levels 1 to N represented particular compositions of elements from the principles level into structures, similar to the way atoms are composed into molecules.

Knowledge encoded in the new language introduced in chapter 9 is based on state primitives (commonly known as *Primitive Types* in classical programming languages) and logic primitives (operations), both of which could be assigned to the first ontological level as mentioned above. Any knowledge template defined in that language is a composition consisting of these primitives and/ or other compound templates.

4.6.4 Building Blocks

As for the word *Ontology*, there are differing definitions for the meanings of the words used in the field of *Terminology*. The ones given in [162] differ only slightly from those of Jeremy Rogers, who has assembled a very useful website [276]. The following explanations are based on it. They are necessary background knowledge for the investigations on *Human Thinking* and the relations within the new knowledge schema introduced in chapter 7.

An elementary building block is the word *Term*, which is a word or phrase (many words) labelling some idea. Another word for idea is *Concept*. Commonly distinguished concepts are:

- *Primitive Concept* (Atomic): cannot be completely expressed in terms of other concepts
- *Composed Concept*: can be expressed in terms of other concepts
- *Pre-coordinated Concept* (Composed): has position in concept system that gets determined before the concept is supplied to end users
- *Post-coordinated Concept* (Composed): did not exist in the concept system as delivered to the user

Special kinds of terms are:

- *Synonym*: two different terms that mean the same thing
- *Homonym*: two terms that sound the same but are spelled differently
- *Eponym*: a term that includes a proper name (like *Murphy's Law*)

Concepts can be related to each other by a *Link*. Flavours of *Semantic Links* are:

- *IS-KIND-OF*: diabetes *is-a* disease
- *IS-PART-OF*: upper limb *has-a* hand
- *CAUSES*: smoking *causes* cancer

A *Code* is an abstract identifier for either a link, or a concept or a term. Rogers [276] writes on this:

If the concepts and the terms in a system are represented separately, then each concept and each term are *unique*. Therefore, each can have a unique code assigned to it. By this mechanism, a single concept may be associated with more than one term (e.g. synonyms or foreign language translations) and a given term might be associated with two quite different concepts (homonyms e.g. *cool* meaning *cold* and *cool* meaning *groovy*).

The language introduced in chapter 9 has primitive concepts (state- and logic primitives, as mentioned in the previous section) and it can express composed concepts. Knowledge

templates defined in that language represent pre-coordinated concepts which become post-coordinated knowledge models when instantiated and altered at runtime. Only *IS-PART-OF* relations (links) are of importance in the language. Knowledge templates written in it can hold many different codes, may they be part of various terminologies or translations into foreign natural languages.

4.6.5 Terminology

While a *Lexicon* is a list of pure words, a *Terminology* (sometimes called *Vocabulary*) can also contain phrases. Because it is a fixed list of lots of terms, a terminology should exclude any link to a separate list of concepts. When a terminology contains additional instructions describing how to interpret each term, or dictating when to choose one over another (prioritisation), it may be called a *Nomenclature*. The knowledge schema proposed in this work (chapter 7) shall be capable of storing codes of various terminology systems.

Lexicon and terminology stand for a *Set* of words or terms, respectively. To bring some structure into such a set, terms or concepts need to be ordered, that is organised through a system of links, into a *Hierarchy*, which Rogers [276] defines as a:

... tree-like structure, where things at the top of the tree are in some way more general or abstract than the things lower down. The nature of each link between each level in the tree may be explicit or only implied, and more than one flavour of semantic link can be used to build the tree (in which case it may be called a *Mixed Hierarchy*).

Kinds of hierarchies, as means of organisation, are:

- *Subsumption Hierarchy* (Classification, Taxonomy): only *is-a* relationships exist between parent-child pairs in the tree
- *Uniaxial Hierarchy*: each concept only ever has one parent, though it can have more than one child
- *Multiaxial Hierarchy*: each concept can have more than one parent as well as more than one child
- *Exhaustive Multiaxial Hierarchy*: all concepts have all the parents as well as all the children they should have

As organisation *Rules* count:

- *Formalism*: an explicitly expressed set of rules, like the specification for how to tell what should (not) be a parent of a concept
- *Concept System* (Model): a system of *Symbols* that stand in for concepts and/ or the links between them, and which may or may not be intended to be processed with reference to some formalism
- *Partonomy* (Mereology): a system of concepts and links intended to represent whole-part relationships specifically

On a yet higher abstract level, a *Data Structure* may hold organisations of concepts. Various types of data structures are:

- *Network*: a mesh-like structure that connects terms or concepts using links; a hierarchy can be thought of as simple case of a network
- *Graph*: a network
- *Directed Graph*: a network in which each link has a *Direction*
- *Directed Acyclic Graph* (DAG): a directed graph free of loops

A knowledge template expressed in the language that will be defined in chapter 9 describes an uniaxial hierarchy, that is its sub concepts have just one parent node. Its structure follows the partonomy (mereology) organisation rules and represents a DAG.

4.6.6 Schemes

One of the – if not the most complex domain in which terminologies are applied is *Healthcare*. As announced in section 1.6, it will serve as example domain for many ideas presented in this work – so for this section describing various organisation schemes for terminologies. The later chapter 11 will come back to this topic once more and briefly introduce a number of terminology systems for healthcare. Jeremy Rogers writes about health terminology [275]:

Health terminology is complex and multifaceted, more so than most language domains. It has been estimated that between 500,000 and 45 million different concepts are needed to adequately describe concepts like conditions of patients and populations, actions in healthcare and related concepts, such as biomedical molecules, genes, organisms, technical methods and social concepts.

The system itself can, for example, be called an ontology, medical entity dictionary, coding- and reference model or reference terminology. The differences in terminology are understandable – this kind of work is highly interdisciplinary and integrates knowledge from linguistics, philosophy, informatics and health sciences, and there is room for misunderstanding between disciplines.

After him [276], there were three broad families of technical approaches to terminologies: *Enumerative Scheme*, *Compositional Scheme* and *Lexical Scheme*. These are explained in the following subsections, mostly citing freely after Rogers [276]. The language defined in chapter 9 might possibly be suitable for creating terminologies following an enumerative- or, better yet, compositional scheme.

Enumerative Scheme

An *Enumerative Coding Scheme* lists, within the scheme, *all* phrases ever to be used, and gives each of them its own code for reference. The phrases can be very long and detailed. The list of phrases provided is *finite*, and it is *fixed*.

A very familiar example of an enumerated scheme is the traditional taxonomic classification of the animal kingdom (figure 4.2). Most of the existing medical terminologies, listing names of diseases, of surgical operations and the like, are also enumerative. One example is the *International Classification of Diseases* (ICD) (chapter 11).

After [276], attempting to enumerate in advance all useful phrases inevitably encounters two serious problems concerning the *Scale* and *Organisation*. Terminologies become:

1. *Scale*: too big to maintain, which results in inconsistent data that cannot be analysed anymore
2. *Organisation*: pre-categorised, which does not allow terms to be simultaneously placed under all different categories that are valid

A further limitation is caused by unfavourable technical choices. The code often serves two purposes. It is: the unique identifier of a concept and the means of representing the relative organisation of a concept. So the common practice of restricting the physical length of a code also restricts the levels of organisation.

level	name	characteristics
domain	eukarya	nucleus, organelles
kingdom	animalia	ingests food, multicellular, no cell wall
phylum	chordata	spinal cord
subphylum	vertebrata	segmented backbone
superclass	tetrapoda	four limbs
class	mammalia	nurse offspring
subclass	theria	live birth
order	primates	high level of intelligence
family	hominidae	walk upright
genus	homo	human
species	homo sapiens	modern human

Table 4.2: Taxonomic Classification of the Animal Kingdom

Compositional Scheme

A *Compositional Conceptual Scheme* typically contains a *controlled* and *fixed* list (*Dictionary*) of a relatively small number (a few ten-thousand) of *primitive* terms, each of which can have a unique code. These primitives may be combined together by users to form more complex terms, including those which might be found in an existing enumerative scheme but also other, sometimes trivial, variations and expansions [276].

Examples of compositional schemes include the *Generalised Architecture for Languages, Encyclopaedias and Nomenclatures in Medicine* (GALEN) and the *Systematized Nomenclature of Medicine* (SNOMED). Hybrid enumerative-compositional schemes are *Logical Observation Identifiers, Names and Codes* (LOINC) and the *International Classification of Nursing Procedures* (ICNP).

The sheer unlimited number of possible combinations, when seen as a problem, is called *Combinatorial Explosion*. Much worse problems, however, are the:

- *Nonsense* combinations that may be constructed (avoidable with a set of semantic links, a grammar and constraints)
- *Redundancy* which occurs when more than one combination of terms express the same concept (avoidable with formal algorithms helping to identify redundant compositions)
- *Post-hoc Classification* (unforeseeable addition of new, unknown concepts) that may prevent a meaningful data analysis (avoidable with a type hierarchy of primitives and of semantics links)
- *Intractability* of data due to *exploding* computer algorithms so that the computer will never find an answer

Lexical Scheme

A *Lexical Technique* is one that helps compare phrases based on what they appear to say – on which words appear, in which order, and in what grammatical constructs – rather than on what they might or might not actually mean. Such techniques can provide a powerful (but not 100% accurate) method for mapping between phrases in existing schemes, or between such phrases and the text found in papers, the *World Wide Web* (WWW) or other electronic resources.

One example of a lexical scheme is the *Unified Medical Language System* (UMLS).

Where lexical techniques break is when the language gets more *slippery*, that is ambiguities may occur. Humans might interpret such results correctly, but automated decision support systems would fail. Rogers [276] concludes that: *As an input for autonomous machine processing applications such as decision support, the outputs of natural language processing tools remain unsuitable.*

4.6.7 Ontology

The aforementioned building blocks (sections 4.6.4 and 4.6.5) can be combined to form new kinds of abstraction [276], as there are:

- *Coding Scheme*: a terminology in which each term also has a code
- *Classification*: a terminology and system of codes and hierarchy

- *Thesaurus*: a classification using a mixed hierarchy (*IS-KIND-OF* or *IS-PART-OF* links)
- *Ontology*: a system of concepts linked to a terminology

An *Ontology*, as system of concepts (section 4.6.1), provides a set of constructs that can be leveraged to build meaningful higher-level knowledge. The relationships between concepts are defined using formal techniques, and provide richer semantics than a classification. Thomas Beale writes [19]:

Ontologies are about representation of knowledge and in their most general form, they *may* have a definition of the atoms (basic constructs). But what they are really made of is semantic links, that is any atom is really defined by its relation to everything else – just like natural language ... real ontologies are more like a sponge or vast octopus-like network of links and concepts – not just atoms.

Chapter 7 will describe a knowledge schema with ontological structure, in other words a hierarchical one with unidirectional relations and additional meta information.

4.6.8 Archetype

With the aim of providing the means to build usable, maintainable, extensible *Electronic Health Records* (EHR), the *Archetype* as design concept was introduced in the *Design Principles* document of the *Good European/ EHR* (GEHR) project, which was later renamed into *Open EHR* [22]. Their website states: *An archetype is a re-usable, formal model of a domain concept*. Archetypes adhere to ontological principles; they can be composed of other archetypes or atomic elements. Their use is not limited to EHR building, despite OpenEHR's focus on the medical domain.

Comparing archetypes with terminologies, Beale [19] writes that a terminology like for example SNOMED Clinical Terms (SNOMED CT) had the form of a semantic network, i.e. ... with an ontological flavour. However, because rigorous design principles were not always applied, they tended to be internally inconsistent and had a lot of pre-coordination in them, while what was really needed was a generative/ compositional terminology. Further, SNOMED could tell what the meanings of the parts of e.g. a complete blood count test are, but it were not going to provide a model of an actual blood test. This is where archetypes ... would come in; they were about information *in use*, not definitions of reality

(as terminologies). ... So – even if SNOMED was perfect, it wouldn't do everything. It were a knowledge support part of the environment, and it could be used to name things and perform inferencing (*draw a conclusion/ deduction* [212]).

An *Archetype Definition Language* (ADL) [21] was created for the specification of archetypes. A corresponding ADL document has the following structure:

```

archetype_id = <"some.archetype.id">
adl_version = <"2.0">
is_controlled = <True>
parent_archetype_id = <"some.other.archetype.id">
concept = <[concept_code]>
original_language = <"lang">
translations = <
...
>
description = <
...
>
definition = <
cADL structural section
>
invariant = <
assertions
>
ontology = <
...
>
revision_history = <
...
>

```

Many separate sections can be identified in this archetype structure, and various syntaxes are used for them. Table 4.3 gives an overview of the structural elements of an ADL archetype. In addition to the single sections, it mentions two further syntaxes, for templates and constraints on data instances.

Chapter 9 will define a new language that is based on just one syntax: the *Extensible Markup Language* (XML) (section 4.1.12), an easy-to-grasp pure text format. Despite its limited vocabulary of just four tags and four attributes, that language may encode a rich set of knowledge constructs, including meta information and constraints.

Element	Syntax	Purpose
archetype structure	<i>Archetype Definition Language</i> (ADL)	glue syntax
definition section	<i>Constraint Form of ADL</i> (cADL)	constraints definition
description, ontology and other sections	<i>Data Definition Form of ADL</i> (dADL)	data definition
template	<i>Template Form of ADL</i> (tADL)	formalism to compose archetypes into larger constraint structures, used in particular contexts at run-time
data instances	<i>First Order Predicate Logic</i> (FOPL)	constraints on data which are instances of some information model (e.g. expressed in UML)

Table 4.3: Structural Elements of an ADL-defined Archetype [21]

4.6.9 Dual Model Approach

The idea of archetypes was inspired by Martin Fowler's *Analysis Patterns* [97] describing a kind of ad hoc two-level modelling, using a *Knowledge Level* and *Operational Level* – as described by the *Reflection* pattern (section 4.2.1), which calls the two levels *Meta Level* and *Base Level*, respectively. Fowler tried to put as much general domain knowledge as possible into the meta level, in order to make application systems more flexible, and to remove unnecessary dependencies. Archetypes represent what would traditionally have been put into the knowledge (meta) level.

Because an *Archetype Model* (AM) (defined in form of ADL documents) and its runtime instances constrain a *Reference Model* (RM) and its instances (figure 4.46), development with archetypes is called the *Dual Model Approach* [22]. Archetypes represent the knowledge belonging to the meta level; the reference model contains information belonging to the base level. *RMs are domain-invariant, i.e. the concepts expressed in the base models mean the same thing right across the domain.* [19, Beale]

The difference between the dual model approach and classical meta architectures is that the latter implement both, meta- and base level using the same technology (language). Archetypes, on the other hand, use the ADL for specifying knowledge. Further, the AM does not depend on the RM, as opposed to meta architectures, where both models depend

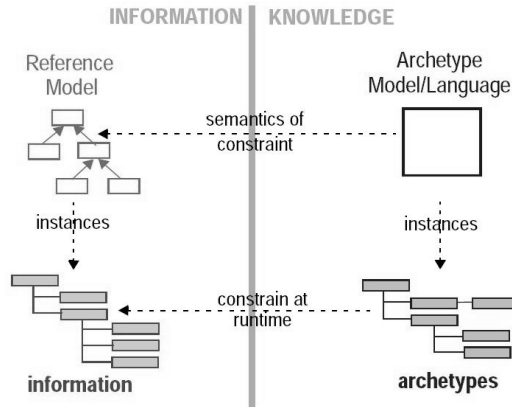


Figure 4.46: Dual Model Approach [18]

on each other bidirectionally. Ergo, the dual model approach is rather comparable to *Agents* (section 4.3.7) owning a mental state (knowledge base).

There are unclear views on what exactly should be constrained when. Sam Heard [19] writes about an *Archetype Kernel* – a tool that could help build (knowledge instances) and ensure that (they) comply to archetypes ... It could operate at a range of points, at:

- a write and edit time: allowing constraint of the data to be based on archetypes (meta-data) rather than application specific processes;
- b creation time of the application or schema: so that the application has read all information constraining the data as indicated – but not through interaction with the archetype at runtime;
- c persistence time: into a database or some other persistence means;
- d communication time: such as when creating a model extract.

Besides obvious benefits of these approaches in constraining domain knowledge, there are a number of potential problems, as Heard mentions:

a makes *b* and *c* redundant and allows the application to stay up to date with the archetype development process; *b* makes *c* redundant (potentially) but means

code has to be cut to incorporate new archetypes; *c* and *d* mean that data may be proved incompatible at a time later than data entry and this may lead to other problems; *d* means you can carry on regardless but there is a risk that data collected will not be compatible with models that are proposed for interoperability.

Further weaknesses of archetypes, the ADL and dual model approach are:

- mix of meta information (properties, constraints) and hierarchical whole-part structure in ADL
- incomplete domain knowledge in ADL lacking logic (algorithms/ workflows) and user interfaces
- inflexible structures due to runtime-dependency of RM instances from archetypes
- use of object-oriented concepts with all their limits, for RM as well as for AM instances

Although the *OpenEHR* project [22] claims archetypes to be both:

- *domain-empowered*: domain experts, rather than information technology people, become able to directly define and manage the knowledge definitions of their systems;
- *future-proof*: systems can be deployed prior to having created formal knowledge models of the (entire) domain

... the above-mentioned issues prevent them from being so. The dual model approach in conjunction with archetypes only partly fulfills the expectations of independent and complete knowledge structures. CYBOP sets out to solve these issues and to find a truly future-proof, long-life system architecture. Knowledge templates written in the language described later in this work (chapter 9) may not only contain meta information constraining application models (as archetypes do), they represent the application itself. The templates themselves are constrained at design time through one universal knowledge schema (chapter 7) dictating their structure. Runtime knowledge models do not reference the template they were instantiated with (contrary to RM instances referencing their corresponding AM instance); a CYBOP application holds all constraints directly in its knowledge models (runtime instances). Since these models follow the structure of the singular knowledge schema as well, they can be serialised easily what makes further constrain activities for persistence or communication unnecessary.

4.7 Modelling Mistakes

While the previous chapters elaborated on *Software Engineering Processes* (SEP) (chapter 2) and the *Physical Architecture* of an *Information Technology* (IT) environment (chapter 3), the sections of this chapter discussed state-of-the-art solutions for designing and implementing the *Logical Architecture* of software systems, that is their inner structure.

Computers can be controlled by *Software*. It contains the instructions after which a computer is run. Instructions can be grouped into levels of growing abstraction, starting from low-level *Digital Logic*, implemented in hardware, up to higher-level *Problem Oriented Languages* (POL). The borders between hardware and software are fluent. Initially, it is up to the computer constructor to decide whether functionality gets burned into hardware or coded into software.

A set of computer instructions is known as *Program*; the language a computer program is written in is known as *Programming Language*. While for early application systems, it was acceptable to write programs directly in *Machine-* or *Assembly Language*, later tasks required languages that were easier and faster to program. A palette of programming languages and -paradigms was introduced in section 4.1.

The more complex software requirements became, the better solutions had to be found to cope with them. Unfortunately, the complexity of the requirements is often targeted with equally complex design- and implementation techniques, leading to dependencies and high coupling within a system. A whole variety of such techniques, more or less complex, exists today and this chapter tried to investigate a rather big percentage of them, mentioning their advantages but also trying to identify disadvantages. To the investigated concepts belong:

- Structure and Procedure; Class and Inheritance (section 4.1)
- Pattern and Framework (section 4.2)
- Component and Concern; Agent with Knowledge Base (section 4.3)
- Application and Domain; Model generated into Code (section 4.4)
- Data and Rule (section 4.5)
- Terminology and Ontology; Archetype and Dual Model (section 4.6)

Some of the identified disadvantages are already addressed and improved by existing technologies. The inflexible *Static Typing* (section 4.1.7), for instance, can be avoided with

Typeless Programming (section 4.1.8). However, there remains a number of design problems to be solved, the main ones of which are listed following:

- Unpredictable behaviour due to container inheritance (section 4.1.15)
- Differing communication patterns due to wrong models (section 4.2.1)
- Bidirectional dependencies due to bad patterns (section 4.2.2)
- Global data access due to static methods (section 4.2.3)
- Redundant code or spread functionality due to concerns (section 4.3.5)
- Provision of a model-only approach to software development (section 4.4.7)
- Finding of a universal schema for knowledge representation (section 4.6.9)

Many of these are caused by *Modelling Mistakes* that will only turn out to be mistakes while being compared with inter-disciplinary ideas in later chapters (part II). One aim of this work is to improve software development by offering solutions to these problems. Yet instead of further complicating software design and -implementation, it tries to bethink on the elementary principle of programming, namely the: *Abstraction of states and logic in form of static knowledge, in order to dynamically control a computer system*. More on this in the following chapter.

5 Extended Motivation

*Those who don't have Courage to dream,
will not have Power to fight.*

AFRICAN SAYING

The previous chapters of part I of this document investigated state-of-the-art concepts for the development, physical- and logical architecture of software, and some of their good and bad sides. The *Suitability* of these concepts for solving a problem, of course, heavily depends upon the intended area of usage. This chapter introduces a new idea to software system design that is as simple as it is helpful. It suggests to:

**Inspect solutions of various other disciplines of science,
phenomenons of nature,
and apply them to software engineering
... in order to find out if existing weaknesses can be eliminated.**

Taken as *Extended Motivation* for this work, the idea leads to a new perspective, from which traditional concepts appear in a very different light. Former *Strengths* (like the bundling of attributes and methods in an OOP class) may suddenly be considered a *Weakness*. Additionally, completely *New Conceptual Solutions* (like the unification of system communication patterns) become possible. The merger of both, traditional and new concepts results in the *Cybernetics Oriented Programming* (CYBOP), as defined in chapter 1.

A description of the intended *Approach* for applying *inter-disciplinary* concepts to software system design finalises this chapter. Three topics that crystallise out here are *Statics and Dynamics*, *Knowledge Schema* and *State and Logic*. They are explained together with their parallels to science or nature in part II following afterwards, which represents the actual *Core* of this document.

5.1 Idea

Researchers quite often follow the approach of first looking into what nature offers and then trying to engineer a similar solution. All kinds of tools and machines were created this way, even (and most obviously, with respect to the human body and mind) robots and computers. Some scientists take the principles of human awareness as physical model to explain the *Universe* [274]. Some business people and consultants see analogies between processes in the human brain and *Organisational Structures of a Company* [281]. Researchers in human sciences systematise *International Public Law* by sharing it into the three parts *Society*, *Cooperation* and *Conflicts* which are chosen in analogy to biology, that is *Anatomy*, *Physiology* and *Pathology* of international relations [28].

Considering all that, one question is at hand: *Why not apply a similar approach to software engineering?* If computers are built after the model of the human being (information input, memorising, processing and output), why not structure the software that actually *runs* those computers after similar models? It seems logical and clear, yet the reality looks different. This work wants to change that, and thereby help to improve application programming.

In search for new concepts to structure software, other sciences are called in. The idea to marry systems sciences (notably general systems theory and cybernetics) for analysis with creative problem solving techniques of designers for synthesis is not new. Swift [73] for example tried to apply both in form of *Cyberpatterns* to complex systems problems, using a pattern language. Yet while Swift had turned his attention to what he calls *the extreme front end*, this work goes one step further. It applies the principles of nature (results of many different sciences) not only to the *User Interface* (frontend) of an application, but to whole software system architectures.

Figure 5.1 shows some of the sciences whose principles were considered in this work. The name of a field of science is shown on top of each box. Made observations are mentioned below, in the middle. The resulting design recommendations for software can be found at the bottom of each box. The recommendations are grouped into those that justify a separation of *Statics and Dynamics* (left-hand side), a new kind of *Knowledge Schema* (lower part of the figure) and a distinction between *State and Logic* models (right-hand side).

It has to be mentioned though, that only some of the principles underlying a specific field of science were considered in the figure and in more detail later in this work. The figure does by no means claim to be complete. The shown observations are only those that seemed promising in the context of software design. The existence of persistent and transient data,

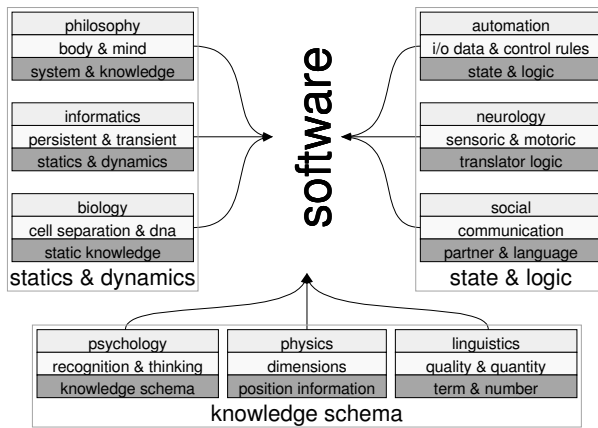


Figure 5.1: Mindmap of Sciences whose Principles influenced CYBOP

for example, is only one of many aspects of the science of informatics. Similarly is the existence of sensoric and motoric nerve system just one aspect of the field of neurology. And so on. Further details on the mentioned sciences and observations are not given here, since later chapters will elaborate on them.

5.2 Recapitulation

The concepts that were found by considering other scientific disciplines, reveal a number of state-of-the-art software design solutions that do not comply with their original in nature, for example the:

1. Mix of static application knowledge and instructions for dynamic system control (chapter 6)
2. False combination of information ignoring hierarchical structure and mixing in meta information (chapter 7)
3. Bundling of state- and logic knowledge (chapter 8)

These discrepancies are the major reason for the issues mentioned in section 1.3. They become clearer only later in this work (part II), where more background knowledge will be provided. Almost all problems they cause have their root in *Dependencies*. As a system grows, the inter-dependencies between its single parts grow with. Why does this happen? Simply because a clear architecture is missing. Even if developers really try to follow a such – on some point in the software’s lifetime, compromises have to be made due to unforeseen requirements and dependencies:

- *Meta Techniques* are used to provide basic functionality
- *Static Managers* accessible by any other parts in the system are introduced
- *Multiple Interfaces* are implemented to realise new properties (*Mix-In*)
- *Redundant Code* needs to be written to avoid too many unwanted inter-dependencies
- *Varying Mechanisms* are applied to plugin new software layers

It seems that today’s software models rarely abstract the real world correctly. This is *not* general critics on software development as it exists today, *nor* is it critics on the abilities of application developers who use current concepts and languages. It is just the neutral, unbiased realisation that there are a few concepts in use which cause unclear, unnecessary, wrong dependencies within software systems. The application of principles of other scientific disciplines might have the potential to solve that.

It was early that, in the style of *Bionics*, parallels between computing machines and the human brain were seen, yet unfortunately do both not function in exactly the same manner. Concepts like *Artificial Neural Networks* (ANN) that try to imitate the physical structure of the human brain exist, but are today’s computers with deterministic behaviour not built like that; they often have a *von Neumann Architecture* [250]. This forces human programmers to *adapt* their thinking to the machine concepts.

Traditional programming languages and design solutions try to ease application development by bridging the gap between concepts of human thinking and those of the machine. Software developers are given tools to design programs in a more abstract way, independently from the source code which gets generated later. But as long as the underlying concepts of abstraction are insufficient, design problems are to be expected. The kind and quality of abstractions is so important, because it influences – and *is* influenced by – all aspects of software development (part I) dealing with *Knowledge*:

- the *Software Engineering Process* specifies static knowledge models (abstractions resulting from process phases), to be later dynamically processed in a computer system
- the *Physical Architecture* requires the translation of knowledge models (communication) between systems
- the *Logical Architecture* provides the means to represent knowledge models (by languages and various techniques) within a system

They all, consciously or not, are trials to apply human patterns. The structure of knowledge models, for example, is based on concepts of *Human Thinking*, the logical *Mind* – as opposed to the above-mentioned neural networks that want to imitate the functioning of the physical *Brain*. Because of the central importance of knowledge, one aim of this work is to investigate new techniques for its abstraction, to thereby revise state-of-the-art software development. However, probably not all traditional concepts will be thrown away. Basic things like control structures (looping, branching etc.) abstracting logic knowledge in form of algorithms are still of importance but appear in a different form (as will be shown in chapter 9). It therefore seems to be more suitable to say that the new concepts will complement (and not revise completely) existing development techniques, as was planned at the beginning of this work (figure 1.3).

5.3 Approach

The *Software Gurus* like Gamma, Buschmann, Fowler et al. found their patterns by *observing* daily software development, architectures, projects and their standard solutions. This work observes other disciplines of science, phenomena of nature, and tries to find parallels to software engineering. It thus not just wants to provide solutions *using* state-of-the-art techniques but rather *question* existing techniques and try to find *alternative* concepts. Because of the steady comparison to principles of nature and other sciences, this approach is called *cybernetics-oriented*, as explained in section 1.4.

Although many of the ideas and solutions of this work, in a *bottom-up* manner, stem from writing source code in practice (following the *Constructive Development* method of research as announced in section 1.5), the overall approach and explanation of results follow a *top-down* path. High-level concepts are considered first, before moving on to an implementation and proof in practice.

Mind and Brain A first observation, when looking at human beings from a philosophical perspective, is the separation of *Mind* and *Brain* (Body) (figure 5.2). Accordingly, CYBOP treats computers as *Systems* owning and processing *Knowledge*. This is not unlike the idea of *Agent* systems owning a *Knowledge Base* (section 4.3.7). All abstract knowledge that humans make up belongs to their mind. The brain is merely a physical carrier of knowledge.

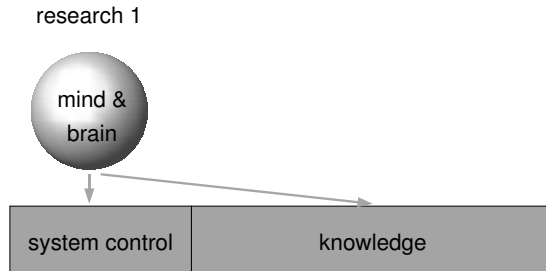


Figure 5.2: Separation of Mind/ Brain Leading to Knowledge/ System Control

Another conclusion resulting from this first observation is that there should actually be two kinds of software: one representing *passive* knowledge and the other *actively* controlling a system, close to hardware.

Chapter 6 deals with this topic.

Human Thinking Secondly, attention is paid to the concepts of *Human Thinking*, as investigated by psychology. Many of them are already considered in current programming languages, for example *Discrimination* and *Categorisation*. However, an essential one that has not been implemented yet is *Composition*. Its application would make every abstract model a *Hierarchy* by default.

Hierarchies are not new, they are present in many ways in today's programming. There are object hierarchies, process hierarchies, design patterns modelling a hierarchy and more. But: the hierarchy as concept is not *inherent* in the type system of current programming

languages. If it were, then *every* type would be a *Container* by default.

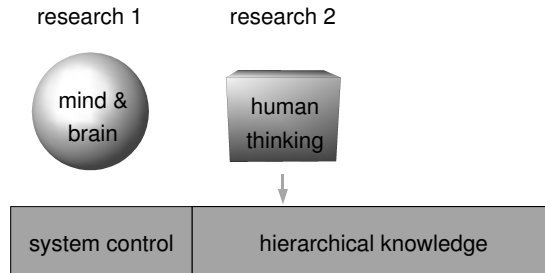


Figure 5.3: Concepts of Human Thinking Leading to Hierarchical Knowledge

Through the application of these thoughts, the knowledge becomes *Hierarchical Knowledge* (figure 5.3). Additionally, this work tries to embed knowledge models in an environment of *Dimensions*, as known from physics, and further properties. Every model keeps a number of *Meta Information* about its parts. *Positions* in space or time are one such example.

Chapter 7 further elaborates on these issues.

Data and Rules Thirdly, *State*- and *Logic* knowledge get distinguished (figure 5.4). It is known from neurological research that the human brain has special communication regions (optical, acoustical, motoric) responsible for information input and output. Simply spoken, these regions do nothing else than translating data, that is an input *State* into an output *State*, according to special rules which can be summarised by the term *Logic*.

This is where systems theory, that uses similar abstractions, comes in. Every system can be seen as *Black Box* with input/ output (i/o) states and a translation logic.

When talking about states, this work does *not* mean classical *State Models* which are often modelled by a *State Chart Diagram*. A CYBOP state model rather is a composed *Set* of states.

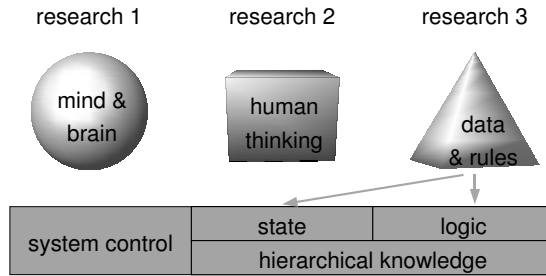


Figure 5.4: Translation of Data by Rules Leading to State-/ Logic Knowledge

Chapter 8 describes more details.

CYBOP In CYBOP, all knowledge, that is states as well as logic, belongs to the *Statics* of a system. It is described by fixed structures. The processing of knowledge at runtime, in order to control a system, is called *Dynamics*.

The complete modelling and storage of static knowledge requires a formal language, which gets introduced as *Cybernetics Oriented Language* (CYBOL) in this work. Its dynamic processing, close to hardware, is guaranteed by the *Cybernetics Oriented Interpreter* (CYBOI) which is needed to run systems defined in CYBOL (figure 5.5).

Chapters 9 and 10 explain CYBOL and CYBOI, respectively. Both are used in the *Res Medicinae* prototype application which gets introduced in chapter 11. Altogether, CYBOL, CYBOI and the theoretical concepts behind are called *Cybernetics Oriented Programming* (CYBOP).

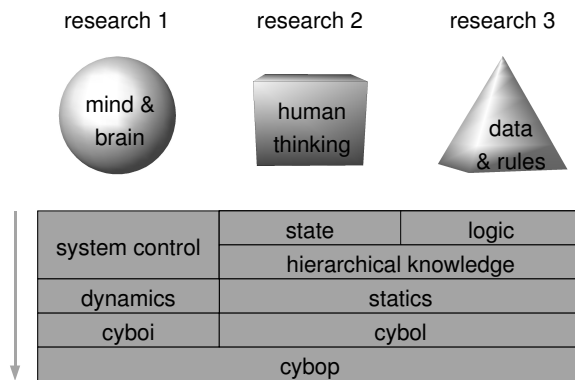


Figure 5.5: Overall CYBOP Approach Based on Statics and Dynamics

Part II

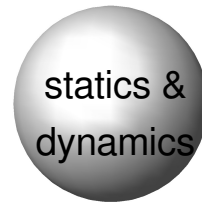
Contribution

6 Statics and Dynamics

I think therefore I am.

RENE DESCARTES

As first of the three main topics of part II of this work, this chapter investigates how a separation of *Static Knowledge* from its *Dynamic Processing* in a system can be justified. The later chapters 7 and 8 will deal with the structuring of knowledge.



6.1 Virtual- and Real World

A separate treatment of knowledge and system functionality can be observed in many fields of science. Some examples are given following.

6.1.1 Mind and Body

In *Philosophy*, it is common to distinguish between two traditions: Euro-American *Western Philosophy* and Asian *Eastern Philosophy* [60]. The former, also called *Western Academic Philosophy*, is often divided into: *Analytic-* and *Continental Philosophy*. While continental philosophy is predominant in continental Europe, analytic philosophy dominates Anglo-American philosophy. Western philosophy has its roots in ancient *Greek Philosophy*, which, among others, dealt with five broad types of analytical questions [60]:

- *metaphysical*: study of any of the most fundamental concepts and beliefs about the basic nature of *Reality*, such as *Ontology* as the science of *Being*
- *epistemological*: study of the nature, origin and scope of *Knowledge*
- *logical*: study of *Inference*, that is *Reasoning* used to reach a conclusion from a set of assumptions
- *ethical*: study of *Morality*, that is behaviour which is *good*
- *aesthetic*: study of the nature of *Beauty*

To the metaphysical questions belong:

- What is reality, and what things can be described as real?
- What is the nature of those things?
- Do some things exist independently of our perception?
- What is the nature of space and time?
- What is the nature of thought and thinking?
- What is it to be a person?

As already mentioned in section 4.6.1, ontology and metaphysics are closely related. The Skeptic's Dictionary [47] writes:

Ontology is a branch of *Metaphysics* which is concerned with being, including theories of the nature and kinds of being. *Monistic* ontologies hold that there is only one being, such as Spinoza's theory that God or Nature is the only substance. *Pluralistic* ontologies hold that there is no unity to being and that there are numerous kinds of being. *Dualism* is a kind of pluralistic ontology, maintaining that there are two fundamental kinds of being: *Mind* and *Body*.

The question how both are related is known as *Mind-Body-Problem*, and besides the above-mentioned pluralistic *Dualism*, there are two monistic views to it [60]:

- *Materialism* (*Physicalism*) is the view that mental events are nothing more than a special kind of physical event
- *Phenomenalism* (*Subjective Idealism*) is the view that physical events are nothing more than a special kind of mental event

The Wikipedia encyclopedia [60] writes:

Most neuroscientists believe in the identity of mind and brain, a position that may be considered related to materialism and physicalism, though there is a subtle difference; namely, that postulating an identity between mind and brain (or more specifically, particular types of neuronal interactions) does not necessarily imply that mental events are *nothing more* than physical events, but rather is more akin to saying that physical events and mental events are different aspects of a more fundamental mental-physical substratum which can be perceived as both mental and physical, depending on perspective.

The idea described hereafter follows this interpretation of materialism. Applied to human existence, this philosophical perspective means that the mind of human systems carries a *Virtual World* that is supposed to be formed by the activity of an underlying physical brain, which serves as representation within the *Real World*. Other questions like whether human systems also host something like a *Soul*, or if the mind actually is what makes up the soul are a topic of *Religion* and not further discussed here.

Transferring this philosophical view to information engineering, one might at first think that *Hardware* is what represents the body- and *Software* what represents the mind of a computer system. This is true in the first instance, but not thought through to the end. There are many kinds of software. *Operating Systems* (OS) with their *Device Drivers*, *Embedded Systems*, *Real Time Systems* or *Firmware* operate close to hardware. *Standard-* and *Business Applications*, on the other hand, contain a lot of logic and domain knowledge which is independent from the underlying hardware.

As result, one might as well treat hardware *together* with hardware-dependent system control software as the body-, and pure application knowledge as mind of a computer system. While system control requires some *active* software running a process (section 3.1) or threads and controlling devices, application knowledge may absolutely be *passive*.

One argument in favour of summarising hardware and system control software was mentioned in section 4.1 which cited Tanenbaum [305] who considers hardware and software to be *logically equivalent* because one could replace the other.

6.1.2 Brain Regions

Neurology as branch of *Medicine* deals with the *Central Nervous System* (CNS) and *Peripheral Nervous System* (PNS) of human beings. These can be further divided as shown in figure 6.1.

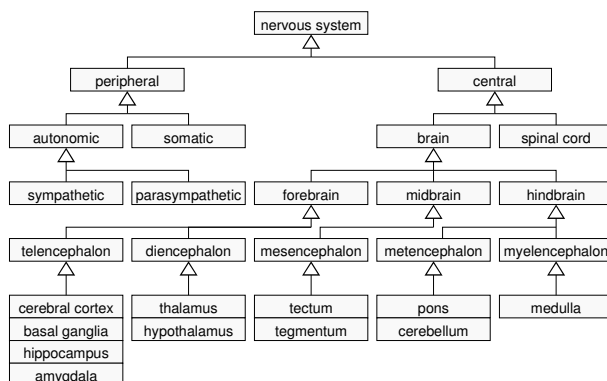


Figure 6.1: Divisions of the Nervous System [52]

Not all parts of this systematics shall be explained here; more details can be found at [270]. Chapter 8 will make some remarks on the PNS, whose *Neurons* (nerve cells) can be functionally divided into *sensory* (*afferent*) and *motor* (*efferent*) ones. The functions of some brain structures, as part of the CNS, are described in table 6.1. At the same time, this table (where *n/a* means *not applicable*) tries to give possible analogies to a standard computer system, whereby hardware as well as software are considered.

Of course, the analogies do not match exactly. Also, there are many functions – like *Thought* or *Emotions* – that a computer cannot perform. The important thing to notice, however, is that there are brain regions mainly *storing* (Hippocampus) and *applying* knowledge (Cerebral Cortex) and others *coordinating* the input/ output (i/o) of that knowledge (Midbrain, Basal Ganglia) in form of simplified information, through sensoric/ motoric organs of the human body.

That is, what philosophy calls *Mind* (section 6.1.1) is the *Knowledge* that is anatomically-physically mainly situated in the *Hippocampus* and *Cerebral Cortex*. And again, i/o control

Brain Structure	Function	Computer Analogon
Cerebral Cortex	Thought, Voluntary Movement, Language, Reasoning, Perception	Application/ Domain Knowledge
Cerebellum	Movement, Balance, Posture	n/a (only in Robots)
Brain Stem	Breathing, Heart Rate, Blood Pressure	Timer, Power Supply
Hypothalamus	Body Temperature, Hunger, Thirst, Emotions, Circadian Rhythms	Self-observing Sensors (Battery-/ CPU Status)
Thalamus	Sensory Processing, Information Forwarding, Movement	Event Mechanism, Signal Loop, IRQ Handler
Limbic System	Emotions	Signal Priorities
Hippocampus	Learning, Memory	Knowledge Storage, RAM
Basal Ganglia	Movement	n/a (only in Robots)
Midbrain	Vision, Audition, Eye- and Body Movement	I/O Device Drivers, Translation

Table 6.1: Brain Structures in Analogy to a Computer [52]

does not only rely on hardware devices but also on the corresponding driver software and signalling mechanism. To say it differently: The software that contains application/ domain knowledge is to be treated *separately* from system control software.

6.1.3 Cell Division

Among other topics, *Biology* – as the science of life – deals with the *Biological Cell*, as smallest structural and functional unit of all living organisms. All types of cells have a *Membrane*, which envelopes a substance called *Cytoplasm*, and *Desoxy Ribo Nucleic Acid* (DNA) as well as *Ribo Nucleic Acid* (RNA) molecules. A DNA molecule is, roughly said, a chain of *Chemical Bases*. The *Order* in which bases are placed determines the properties of (*Proteins* of) a biological creature. To the *Organelles* contained in cytoplasm belong [60]:

- *Cell Nucleus*: housing the genetic information
- *Ribosomes*: producing proteins
- *Mitochondria* and *Chloroplasts*: generating energy
- *Endoplasmic Reticulum* (ER) and *Golgi Apparatus*: transporting macromolecules
- *Lysosomes* and *Peroxisomes*: digesting

Multicellular organisms grow by a process called *Cell Division*, in which a *Mother Cell* divides into two *Daughter Cells*. The process differs slightly between cell types, but mostly, the genetic information (DNA) is replicated first, before the cell nucleus- and finally the whole cell divides, whereby the genetic information is distributed equally to both new-born cells. The new cells use the genetic information encoded in the DNA, to create new organelles and to function correctly.

In a comparative consideration, the cell corpus may be equated with a computer system, and the genetic information with the software which runs that system. Each cell represents a system with different hardware but all cells (in one-and-the-same biological creature) use the same configuration information. In other words, the configuration information can be forwarded and used in different hardware.

But, again, there is one thing to keep in mind: *System control software is not equal to application software*. The configuration information contained in a DNA may well represent the building plan for all kinds of cells in a biological organism – but it is not *controlling* those cells. Genetic information in a DNA is *passive*; in order to make use of it, some *active* mechanism must be employed. In a biological cell, it is the RNA molecules which transmit the genetic information from the DNA (via transcription) into proteins (by translation).

In a simplified view, one might say: *The cell is built according to the instructions read from the DNA*. The genetic information of the DNA may be compared to the domain knowledge of a software application, or generally to configuration information – also that of an *Operating System* (OS). The RNA activity and other cell control mechanisms, on the other hand, are comparable to signalling, control loops, or the device drivers of an OS.

6.1.4 Short- and Long-Term Memory

It was previously worked out that there are brain regions mainly storing and applying knowledge and others controlling the input/ output (i/o) and manipulation of that knowledge. *Learning, Storage and Recall* of knowledge are main tasks of the human brain, which are studied by the science of *Psychology*.

Besides the *persistent* storage in *Long Term Memory* (LTM), the brain is capable of storing *transient* information in *Short Term Memory* (STM), the latter also being called *primary* or *active* memory [60]. An additional *Sensory Memory* stores information arriving directly from the corresponding organs. Figure 6.2 tries to classify *some* types of memory, as de-

scribed by psychology. It is not more than a *trial* because psychology itself is not sure about memory classification and several theories exist.

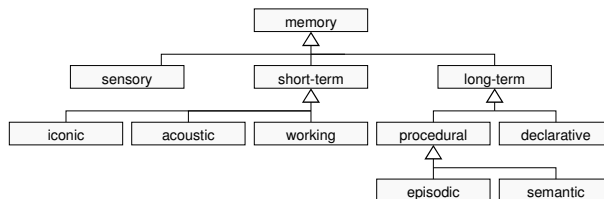


Figure 6.2: Types of Memory [187, 137]

The *Encyclopedia of Educational Technology* (EET) [137] writes:

The sensory information store has unlimited capacity, and reacts to both visual and auditory information. However, the duration of information in sensory memory is extremely brief, perhaps only 300 milliseconds, and is subject to rapid decay.

STM, in general, is characterized by a limited capacity of up to seven pieces of independent information, and in the brief duration of these items in STM, usually anywhere from three to 20 seconds. Additionally, decay appears to be the primary mechanism of memory loss in STM.

LTM efficiently stores our knowledge about the world. It is important to contrast LTM with other types of memory and understand how it is structured. The knowledge we store in LTM affects our perceptions of the world, and influences what information in the environment we attend to. LTM provides the framework to which we attach new knowledge, and its properties have important implications for instructional design.

In the words of the Free Wikipedia Encyclopedia [60], Information held in STM may be:

recently processed sensory input; items recently retrieved from LTM; or the result of recent mental processing. When doing mathematical calculations, for example, intermediary results stored in STM are available for only a short time and forgotten soon after.

The *declarative* LTM is conscious memory, a *film* of past contents [86]. The *procedural* – or *non-declarative* – LTM is unconscious memory which enables humans to carry out a task (like riding a bicycle), without having to consciously control it. In other words, procedures stored in non-declarative LTM are available- and may run as background program.

What effects do these reflections have on the design of software systems? The storage and dynamic processing of static knowledge may firstly rely on at least two different kinds of memory, one for *persistent*- and another one for *transient* storage of knowledge; secondly, they may rely not only on one main process controlling the system, but (as equivalent to procedural LTM) employ a number of background processes solving special tasks.

6.1.5 Information Processing Model

Information processing, from the view of *Cognitive Psychology*, follows the model shown in figure 6.3. General information has to pass several stages before it becomes meaningful knowledge. The results of processing stages are stored in different memories.

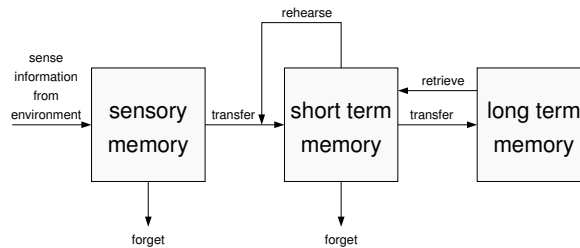


Figure 6.3: Information Processing Model [137]

The *Encyclopedia of Educational Technology* (EET) [137] writes on this:

After entering sensory memory, a limited amount of information is transferred into short-term memory. . . . The process of transferring information from STM into LTM involves the *Encoding* or *Consolidation* of information. . . . Recent research (focuses) on the necessity of the brain to organise complex information in STM *before* it can be encoded into LTM.

In this process of organization, the *Meaningfulness* or *Emotional Content* of an item may play a greater role in its retention into LTM. Also, on a more concrete level, the use of *Chunking* has been proven to be a significant aid to STM transfer to LTM. Because STM's capacity is limited to seven items, regardless of the complexity of those items, chunking allows the brain to automatically group certain items together.

Certainly, the emotional content of an item can be neglected for computer systems as of today. But chunking as a technique to divide information into discrete items is of great importance in human thinking, which gets investigated closer in chapter 7.

6.1.6 Persistent and Transient

In the science of *Informatics*, there are a few *Integrated Circuits* (IC) – so-called *Read Only Memories* (ROM) – containing unchangeable programs. The *Basic Input/ Output System* (BIOS) of a computer is one example for such programs, software of *Personal Digital Assistants* (PDA) or *Mobile Phones* are others. Often, the BIOS is stored in *Electrically Erasable Programmable ROM* (EEPROM) or its later form called *Flash ROM*. It thereby gets writable. When writing it, the complete old BIOS gets overwritten. Most software programs, however, reside on a *Hard Disk Drive* (HDD) as permanent storage medium.

In order to use them, such *persistent* programs often need to be loaded into an IC called *Random Access Memory* (RAM) which, other than a ROM, can be both read from and written to. Most RAMs are *volatile* which means that the data they store – to which also belong programs – are *transient*, that is get lost in case the computer is powered down. The ability to manipulate data in memory is a pre-requisite to usefully work with a computer.

One surely could, with some effort, rearchitect computers so to let the *Central Processing Unit* (CPU) communicate directly with persistent memory (like a HDD), instead of RAM. One reason for not doing this is the *Performance* of computer systems – RAM can be

accessed much faster than a HDD. Another reason is the independence from differing HDD designs.

What this section by its remarks tries to show is that *static* software becomes *dynamic*, that is changeable over time, when being loaded into a RAM whose data (represented by its state) can be processed (manipulated) through a CPU. Although not being so new, this statement has importance for some considerations later in this chapter.

While some kinds of software (like standard- or business applications) mainly *contain* static knowledge of a special domain, other kinds do mainly *use* that knowledge to dynamically control hardware. The point is that traditionally, both kinds of software are mixed up. A business application has to care about memory allocation, graphical input and output (even when using a framework for that), communication mechanisms and more, although these are not in its original interest. On the other hand, an operating system often contains configuration knowledge about its structure or available devices that does not actually belong into it. In order to achieve a clearer structure with less dependencies and more flexibility, it is necessary to treat both kinds of software differently.

6.2 System and Knowledge

Having explained why a strict separation of application knowledge and system control software is desirable, several state-of-the-art techniques can now be considered once again, in order to find out about possible new effects to software design or source code.

6.2.1 Configurable or Programmable

Knowledge (first defined in section 4.5 of this work) is present in all kinds of software systems. It is what characterises a system, because: it defines possible states and logic, their structure and relations, and thereby a complete application; it such determines the way a system process controls the computer hardware it runs on; it thereby has the capability to change the properties and behaviour of a whole computer system. One can therefore say that knowledge encoded in software represents the *Configuration* information necessary to run a computer system in the desired way.

In addition to the configuration information hard-coded in the program, most applications offer to alter special settings such as paths, language choice and spelling, editor and saving

options, colours, fonts and further properties. Usually, these are made persistent using some kind of external storage like a flat file or a database. One popular format for storing simple properties are so-called *Key-Value-Pairs*. Modern applications do also make use of hierarchical storage for more complex settings.

Yet if a software program already represents all knowledge needed to run an application on a computer, why storing extra settings externally? Obviously, a standard program is not *flexible* enough; it cannot be changed anymore after compilation. But program changes at runtime are often highly desirable.

So, if external storage of properties does make sense, why not storing *everything* outside the actual program? This seems to be a crazy but very useful idea, as it would result in absolutely flexible application systems. But it has limits. There *must* be some core program (*Kernel*) able to read and write (*interpret*), and to process (*handle*) external properties (*Signals*). The more complex, structured and inter-related these properties are, the more suitable it is to call them *Knowledge*.

A technical system may be able to understand external knowledge, just like human beings have the cognitive abilities to understand their environment by building a *Virtual World* of it. Yet is this not enough. Knowledge about the *Real World* environment is one thing; interacting with it another. A computer has hardware devices for interacting with the real world. The devices need to be operated correctly so that knowledge can be exchanged through them. This hardware driving functionality is normally provided by an *Operating System* (OS). But current OS have the deficiency of not being able to handle knowledge. What is needed, finally, is a system with *low-level* hardware control abilities like an operating system *plus* additional *high-level* knowledge handling abilities [125].

Other people reflected on this and have come to a similar conclusion. Thomas Beale writes in [168]:

The history of IT ... has taught us that the only kind of useful system that can be delivered to domain users ... is one which is not just configurable, but *programmable* – not by statements of source code, but by high-level *domain-user-oriented* tools.

Well, the user-oriented, domain-knowledge handling tools are desirable, but only in the second place. The important part of this statement is the realisation that systems need to become *programmable*, and this by *External Knowledge*. Whether this knowledge gets

created and maintained manually or by using graphical tools, is of minor importance. What is needed in any case is a formal knowledge specification language serving as basis for the people or tools to work on. Chapter 9 will introduce such a language.

6.2.2 Code Reduction

In his book *Programming Pearls* [26, page 128], Jon Bentley demonstrates *Code Reduction* on the following graphics program example:

```
for i = [17, 43] set(i, 68)
for i = [18, 42] set(i, 69)
for j = [81, 91] set(30, j)
for j = [82, 92] set(31, j)
```

He suggests to replace the *set* procedures that switch a *Picture Element* (Pixel) with suitable functions for drawing horizontal and vertical lines:

```
hor(17, 43, 68)
hor(18, 42, 69)
vert(81, 91, 30)
vert(82, 92, 31)
```

This code, finally, gets reduced to pure data stored in an array:

```
h 17 43 68
h 18 42 69
v 81 91 30
v 82 92 31
```

The data can be read by an interpreter program which knows about their meaning.

Bentley's example shows in a nice way how knowledge can be extracted from program source code. The graphic application's actual data are represented by the values in the array above. All other functionality accessing and manipulating Pixels directly does belong to system control and remains in the interpreter program. Chapter 10 will introduce an interpreter that is able to read and handle *general* knowledge, only on a much larger scale.

6.2.3 Base- and Meta Level

Reflective techniques as described in section 4.2.1 make use of one so-called *Base Level* and one or more *Meta Levels*. The reason for splitting a system's architecture in this way is the hope to be able to move rather general *System Functionality* into a meta level, while leaving domain-specific *Application Functionality* in the base level. (Well, in his book *Analysis Patterns – Reusable Object Models* [97], Fowler used meta levels to model general classes containing not exclusively system- but also domain-specific functionality.) The conflicts a design decision of that kind can bring with were described in section 4.2.1, which – above all – criticised the bidirectional dependencies.

However, what the proposition of reflective software patterns shows, is the existence of a wish among software developers, to separate general system- from more specific application functionality. And, as was shown in section 6.1, nature does exactly that. Yet while reflective mechanisms use the same implementation techniques for system- as well as for application-specific functionality, nature always treats passive knowledge strictly separate from active system control (section 6.1). Bidirectional dependencies do not exist between the both.

6.2.4 Reference- and Archetype Model

The *Archetype* concept as introduced in section 4.6.8 *does* provide an independent implementation technique (language) for the definition of application-specific domain knowledge: the *Archetype Definition Language* (ADL). The documents written in it, altogether, are referred to as *Archetype Model* (AM). They get parsed and instantiated at runtime. These instances are then used to constrain instances of a *Reference Model* (RM). Because of the existence of two models implemented with two independent techniques, this method of programming is called *Dual Model Approach* (section 4.6.9).

It wants to solve the dilemma of lacking domain semantics in classical information models. Archetypes are the corresponding knowledge documents carrying semantic information. They provide the structures and rules after which instances of an RM can be combined meaningfully. Despite its drawbacks mentioned in section 4.6.9, the dual model approach animated this work to pay attention to two things:

1. the usage of different implementation technologies for domain knowledge (AM) and underlying system-level functionality (RM)
2. the need to provide constraint information with knowledge models

The distinction between domain knowledge and system-level functionality is realised by providing a knowledge modelling language (chapter 9) and a corresponding interpreter (chapter 10). The language is capable of expressing structural- as well as meta information, to which also belong constraints.

6.2.5 Common- and Crosscutting Concerns

Section 4.3.6 argued that, after [254], *Aspect Oriented Programming* (AOP) were necessary because some concerns were not easily turned into *Classes* – the natural unit of modularity for *Object Oriented Programming* (OOP) – because they'd cut across classes. Much like OOP were a way of modularising *Common Concerns*, AOP were a way of modularising *Crosscutting Concerns*. Figure 6.4 is a trial to classify both kinds. (The distinction into *Development*- and *Production Concerns* is of minor importance here.)

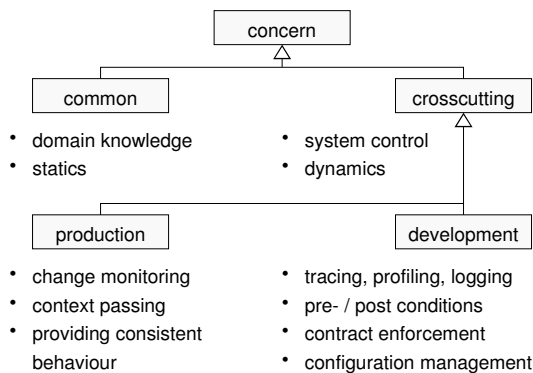


Figure 6.4: Classification of Concerns

Looking closer at these, it becomes obvious that crosscutting concerns represent general *System Control* functionality, while common concerns stand for properties specific to an *Application*. Comparing with nature (section 6.1), the separation of both kinds of concerns seems absolutely correct. It arises the question, however, if AOP with all its additional concepts is really the most suitable way for treating crosscutting concerns? This work

means *no* and suggests to simply put all general control functionality into a basic knowledge-interpreter system underlying all applications (chapter 10).

6.2.6 Application and Domain

Over the years, it has turned out to be helpful in software design, to separate *Domain Knowledge* from *Application Functionality*. In one-or-another form, the architectural patterns (section 4.2.1) *Layers*, *Domain Model* and *Model View Controller* (MVC) all suggest to apply this principle. Figure 4.1 showed a typical example of a system with logical layers, among them a domain layer containing business logic.

The *Tools & Materials* approach of section 4.4.1 talks of *active* applications (tools) working on *passive* domain data (material). And also *System Family Engineering*, as mentioned at the beginning of section 4.4, is based on a separate treatment of domain and application, in form of *Domain Engineering* (DE) and *Application Engineering* (AE).

An often neglected fact of these approaches is that not only the domain, but also the application contains important business knowledge. Figure 6.5 tries to demonstrate this by organising typical software patterns (section 4.2) and system functionality into two orthogonal pairs of layers. The *User Interface* (UI), for example, is mostly assigned to the application layer, yet since it is clearly tailored for a specific business domain, it be better assigned to a knowledge layer, together with the corresponding domain models. And the logic behind a UI, if not contained in the UI itself, is often put in a *Controller* which belongs to the application-, not the domain layer. But because a controller's task is the management of general functionality like the processing of signals (events) and communication, which are not application-specific, it be better sorted into a low-level system layer. It is true that controllers often contain some application logic; but that in turn belongs to the high-level knowledge layer above it.

Similarly, the domain often contains functionality which actually does belong into the application process: *Database* (DB) access is handled by help of patterns like the *Data Mapper* (section 4.2.1), in which the mapper objects contain *Structured Query Language* (SQL) code to connect to a *Database Management System* (DBMS); *Enterprise Java Beans* (EJB), which should better be pure domain objects, imitate a *Middleware* providing persistence- or other communication mechanisms, which originally have nothing to do with the business knowledge they contain.

	application	domain
knowledge	<ul style="list-style-type: none"> • MVC View • User Interface 	<ul style="list-style-type: none"> • MVC Model • Domain Model • DTO
system	<ul style="list-style-type: none"> • MVC Controller • Process/ Thread 	<ul style="list-style-type: none"> • Data Mapper • DB Access • Communication • EJB Persistence

Figure 6.5: Domain-Application- versus System-Knowledge Separation

It is precisely this *Mixup* of responsibilities between an application system and its domain knowledge, that leads to multiple inter-dependencies and hence unflexibility within a system. Instead, a separation should be made between active *System Control* and passive *Knowledge*. A UI's appearance would then be treated as domain knowledge, just as the logic of the functions called through it. A data mapper would be transformed into a simple *Translator* – similar to a *Data Transfer Object* (DTO) (section 4.2.1) – that knows how to convert data from one domain model into another; its DBMS access functionality, however, would be extracted and put into the system layer. More on that in chapter 8. Monstrosities like EJBs would likewise be opened up and parted into their actual domain knowledge, and all the other mechanisms around – the latter being moved into the low-level system layer.

To sum up this thought: The essential realisation here is that hardware-close mechanisms like the ones necessary for data input/ output (i/o), enabling inter-system communication, should be handled in an active system layer which was started as process on a computer, and *not* be merged with pure, passive domain knowledge. Application logic which is traditionally held in controller objects of the application layer, and other business data models should rather belong to a high-level knowledge layer.

In chapter 10, this work introduces an interpreter providing low-level functionality. High-level knowledge, on the other hand, may be modelled in the language defined in chapter 9.

6.2.7 Platform Specific and -Independent

The *Model Driven Architecture* (MDA) discussed in section 4.4.6 took a first step into the right direction, by distinguishing *Platform Independent Models* (PIM), that is domain- and application logic, and *Platform Specific Models* (PSM), that is implementation technology. It encourages the use of automated tools for defining and transforming these models.

While the definition, organisation and management of architectures (PIM) mostly happen in the analysis- and design phase of a *Software Engineering Process* (SEP), the generation of source code (PSM) can be assigned to the implementation phase. The approach still has weaknesses, and tools which can truly generate running systems are rare or not existent, at least to what concerns more complex software systems – not to talk of the so-called *Roundtrip Engineering* (section 4.4.7), which is managed by even less tools (experience of the author while developing on the *Object Technology Workbench* (OTW) UML tool [152] and working in several projects).

Nevertheless, the trend clearly goes towards more model-centric approaches, as section 4.4.7 pointed out. The aim of this work is to supply domain experts and application developers with a *Model Only* technology (in relation to figure 4.45), allowing to create application systems that do *not* have to be transformed into classical implementation code any longer, whereby the SEP abstraction gap number 2 (figure 2.6) could be closed conclusively. The knowledge schema introduced in chapter 7 is a necessary prerequisite therefor.

6.2.8 Agent with Mental State

One design paradigm that early recognised the advantages of splitting software into low-level system control and high-level knowledge, is *Agent Oriented Programming* (AGOP) (section 4.3.7). *Agents*, as active software components (which in this work means: *running in an own process*), have a *Mental State* representing their knowledge, which they are able to interpret and manipulate. This approach was copied in CYBOP.

Of the three elements *Formal Knowledge Representation Language*, *Agent Programming Language* and *Conversion Method*, which an AGOP system, after section 4.3.7, needs in order to be complete, this work provides the first two in form of the *Cybernetics Oriented Language* (CYBOL) and the *Cybernetics Oriented Interpreter* (CYBOI), described in sections 9 and 10, respectively. Thereby, CYBOI itself is not a language, but represents a ready system, written in the *C* programming language. A method for converting traditional

applications into agents is not provided, since methodologies are clearly *not* a topic of this work.

Of course, there are differences distinguishing *Cybernetics Oriented Programming* (CYBOP) from traditional AGOP systems. CYBOP needs an own interpreter, because of its new knowledge representation philosophy and -language, which are the topic of chapters 7 and 9. One reason for the difficult handling and intransparency of many traditional knowledge representation languages is that they mix two kinds of knowledge: *State*- and *Logic* descriptions. More on how this is avoided in CYBOP in chapter 8.

One may wonder why such a supposedly advantageous architecture is not used by all of today's systems? One reason may be that AGOP is still a rather young technology lacking the necessary popularity. Another reason may be the bad reputation of AGOP systems (and just about everything that has to do with knowledge representation) among average developers – partly because of their immaturity, but mostly because of their complicated knowledge models and -handling. Looking at the often quite cryptic appearance of the corresponding languages, one tends to understand the developers' dislike.

6.2.9 Data Garden

Now, if a separation of high-level knowledge from low-level system control software is considered to be useful, the next question must be: *How, that is in which form, best to store knowledge in a system?*

One possible structure called *Data Garden* [139] was proposed by Wau Holland, founder member and formerly chairman by seniority of the *Chaos Computer Club* (CCC) [48] – Europe's largest hacker group. (Note to the reader: A *Hacker* is just a computer freak; a *Cracker* is a criminal.) Although being a non-academic organisation, his ideas on knowledge modelling are very interesting to this work. He dreamt of whole *Forests*, *Parks* or – as the name says – *Gardens of Knowledge Trees* and *Data Bushes* (figure 6.6).

Many knowledge engineers share a similar view and consider knowledge to be a network of inter-related concepts. Philippe Ameline writes in [19]:

All hierarchical trees are inter-connected, and one does better replace (purely) *Hierarchical Traits* with *Named Traits* ...

A *Hierarchical Tree* is a set of traits between nodes. These traits are not labelled since they all mean *son of*. A *Semantic Network* is a set of *labelled* traits, with

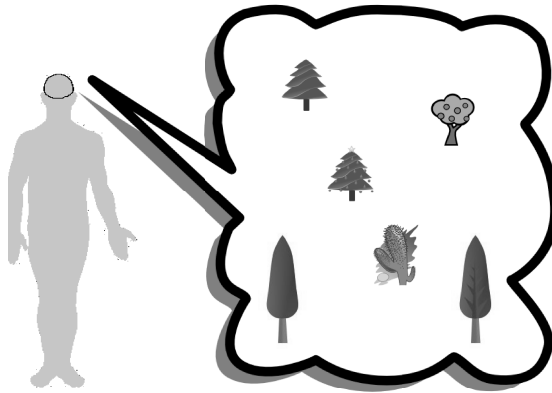


Figure 6.6: Data Garden

labels of the kind *is a*, *is part of*.

Thus, building a hierarchy and building a semantic network is the same kind of job, but the hierarchy demands two (huge) constraints: there is a *Single Root Node*, and nodes have a *Single Kind of Label*. These constraints usually can't be satisfied, so a migration from genuine hierarchy to semantic network usually occurs.

The system architecture proposed in this chapter will have *one single* root node, for all knowledge. Furthermore, chapter 7 will work out a hierarchical knowledge schema in which *all* parts of a whole have a unique *Name*.

As an aside: Holland's main interest was the question how information would best be represented *visually*. Knowledge having the form of plants could be recognised by their *Shape* or *Colour* which would ease orientation within a data garden. Old data models that lost in importance could visually loose their colour or move into the background, becoming smaller in perspective.

Visualisation techniques described in Lombardoni [201] and Barberena [16] are *Perspective Walls*, the *Benediktine Approach*, *Fractal Rooms*, *Cone Trees* and *Hyperbolic Trees*. In his work, Lombardoni mapped object-oriented models to three-dimensional graphics. The techniques are not elaborated further here since they belong to the visual side of knowl-

edge representation, whilst this work researches concepts for the structuring, storage and management of knowledge in general. However, the results of this work might simplify the visualisation of knowledge in future. Since CYBOP knowledge models are stored in tree-form, it should be rather easy to represent them graphically.

6.3 Knowledge Management System

Section 6.1 justified a separation of knowledge from system control software. Section 6.2 considered the effects of that separation to traditional software design. What remains to be investigated is how a system adhering to a separation of that kind would have to look like.

6.3.1 Hardware Connection

Knowledge is *passive*. What makes use of knowledge is the *active* parts of a system, in the case of computers a process like the *Operating System* (OS) or applications using external configuration settings. They are able to both, communicate with hardware and adopt knowledge, for it to be memorised and processed.

Traditionally, OS make use of a number of helper processes (*Daemons*, section 3.6), for services like printing or email delivery, which may also be used by applications. This work, however, wants to unify services in just one low-level system control process. Another issue are the varying communication paradigms a classical application has to consider. Persistence mechanisms, user interfaces, remote communication – they all have their specific requirements, whether supported by a special framework (section 4.2.4) or not. This work wants to simplify communication in a way that applications do not have to do more than issuing a simple *send* or *receive* instruction, adding the desired language of communication. By disburdening applications from low-level communication and signal (event) handling responsibilities, they become purely passive knowledge (statics) which cannot act itself, but needs to be read and interpreted by an active control process (dynamics).

Three main layers of information crystallise out: *Knowledge*, *Control Software* and *Hardware* (figure 6.7). Tanenbaum [305] calls the latter two *logically equivalent* (section 4.1), because one could replace the other. It is indeed up to the computer designer to decide how much control software should get burned into hardware. Hence, the important separation is between *Knowledge* on one side and *Hardware* together with *Control Software* on the other.

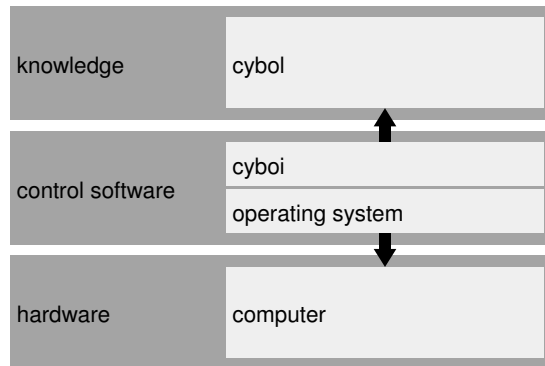


Figure 6.7: Knowledge – Hardware Connection

The previous sections 6.1.1, 6.1.2 and 6.1.3 tried to justify this separation by looking at nature. Knowledge is the equivalent of: *Mind* (philosophically) and the virtual information stored in a human brain's *Hippocampus* and *Cerebral Cortex*, as well as of the information encoded in a biological cell's *Desoxy Ribo Nucleic Acid* (DNA). Hardware and control software are the equivalent of: (philosophically) *Body*, (neurologically) parts of the human brain (*Midbrain*, *Basal Ganglia*) which coordinate the input/ output (i/o) of knowledge and (biologically) *Ribo Nucleic Acid* (RNA) molecules transmitting the genetic information from the DNA into proteins.

Chapters 9 and 10 will describe the *Cybernetics Oriented Language* (CYBOL) as knowledge specification format and the *Cybernetics Oriented Interpreter* (CYBOI) as system being able to handle such knowledge, as well as to serve as hardware interface. All hardware-controlling functionality needs to be present within either CYBOI or the underlying *Operating System* (OS) closely coupled with it. Together, they are the active entity allowing virtual and real world (knowledge and hardware) to communicate.

The remaining sections of this chapter describe important elements belonging to a control software's architecture. More detailed descriptions of the architecture and functionality will be given in chapter 10 devoted to CYBOI only.

6.3.2 Memory

The application/ domain knowledge a control software processes resides in a *Memory*. Two different kinds known from *Informatics* are the *persistent* and *transient (volatile)* memory (section 6.1.6). For the system architecture investigated in this section, the term *Memory* does not refer to hardware, but to a special data structure for knowledge storage.

Section 6.1.4 introduced the *Sensory Memory*, *Long Term Memory* (LTM) and *Short Term Memory* (STM), so labelled by the science of psychology. Sensory memory stores data arriving from input organs; LTM stores past contents; STM holds temporary information to be processed within the system. A knowledge-processing system with human archetype – such as the one proposed in this work – needs to have equivalents for all three of them. Figure 6.8 illustrates a system based on four kinds of memory:

- Knowledge Memory (equivalent of LTM)
- Signal Memory (equivalent of STM)
- Internal Memory (program-internal data)
- Input/ Output Memories (sensory data)

The *Knowledge Memory* is represented by one single root node that is able to keep knowledge hierarchies of arbitrary size. The *Signal Memory* is much the same as the *Event Queue* in classical systems. *Internal Memory* and *Input/ Output Memories* are helper memories for storing system-internal parameters.

6.3.3 Processing

While knowledge as such is static at a given time instant, its *Processing* and manipulation over time are dynamic. The processing is triggered by some *Signal* (also called *Event*), which is a state change known to the system. Such *signs with defined meaning*, as the Duden Encyclopedia [71] calls them, can be most different in their appearance and communication channel used.

Signals are commonly stored in a *Signal Memory* (also called *Event Queue*), as mentioned in the previous section. An endlessly running *Signal Loop* (also called *Waiting Loop*) as illustrated in figure 6.8 is constantly checking the signal memory for new signals. Once a signal is detected, it gets removed from the signal memory and handled by the system. The

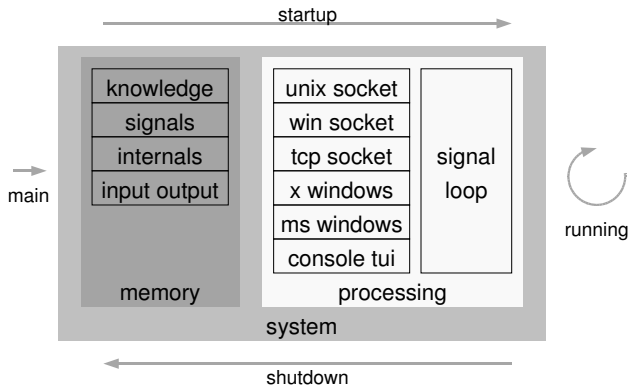


Figure 6.8: System with Memory Structures, Processing Loops and Lifecycle

signal with highest *Priority* is processed first. The later chapter 10 will explain further details and deliver a more functional illustration (figure 10.2).

Section 6.1.2 mentioned the *Hypothalamus* and *Limbic System* as parts of the human brain producing emotions. Section 6.1.5 wrote that the processing of a signal may be greatly influenced by the meaningfulness or *Emotional Content* of an item. Well, software systems do not work with emotions, but signals can be assigned a *Priority*, which is somewhat comparable. *Prioritising* as technique stems from *Operating System* (OS) research and can be well applied in the described knowledge-processing system: Signals can be filtered in a way that unimportant signals get discarded; urgent signals get processed right away; less important but meaningful signals get queued for later handling.

All *intra-system* and *inter-system* communication is based on the exchange of knowledge via signals. A signal can transport simple or more complex *Messages*, mostly in encoded form. The communication details, including encoding and decoding procedures for knowledge model translation, and the logic after which an input state gets transferred into an output state are the topic of chapter 8.

Besides the *declarative Long Term Memory* (LTM), section 6.1.4 mentioned the *procedural* (non-declarative) LTM, enabling humans to carry out a *Background Task*, without having to consciously control it. A similar principle is applied for input/ output (i/o) handling, in

the described knowledge processing system. Independent *Threads* running their own loops control a special i/o mechanism (like *UNIX socket etc.*), each (figure 6.8).

6.3.4 Lifecycle

Before the memories and control loops described before can fulfill their tasks of storing and processing knowledge, respectively, they have to be created and activated, which happens at system *Startup* (figure 6.8). The startup needs to be initiated by some *main* entry procedure. At system *Shutdown*, just the opposite needs to happen, that is control loops and memories have to be deactivated and destroyed. In between startup and shutdown, the system lives, it is *running*. The whole procedure of starting up, running and shutting down a system is called *System Lifecycle* – not to be mixed up with the lifecycle of software, which refers to its analysis, design, implementation and subsequent growing old. The term *System Lifecycle* in this work is used in relation to the component lifecycle of *Component Oriented Programming* (COP), described in section 4.3. It dictates the order in which creation and destruction of system parts (in memory) need to happen.

Once the startup phase has reached the endless signal waiting loop, it relies on the presence of an initial signal, to get some main application running, that is instantiate given knowledge templates. The signal has to contain a corresponding (logic) knowledge model describing an operation activity. It must be added to the signal memory during system startup. More details are given in chapter 10.

The difference between *Knowledge Templates* and *Knowledge Models* – as used in this work – is that templates contain static knowledge stored persistently in something like a file on a *Hard Disk Drive* (HDD); models, on the other hand, represent instantiated, dynamic knowledge that resides transiently in a computer's *Random Access Memory* (RAM). The process of *Instantiating* knowledge involves cloning a persistent template in order to receive a transient model, which can then be freely manipulated in RAM. Likewise, transient knowledge models can be made persistent, if so desired, by *Serialising* and writing them onto some persistent memory like a HDD.

Besides the knowledge that gets instantiated by the lifecycle at system startup, there has to be a possibility to create and destroy knowledge instances at a later point in system runtime. The decision whether to create a transient model at system startup or only later as the need arises, can have great effect. An application with *Graphical User Interface* (GUI), for example, may contain 100 dialogues. If there is enough RAM in the system,

all dialogues could get created at startup, so that one would only have to switch them visible or invisible, at runtime. In this case, high dialogue *Performance* (a non-functional requirement) would be guaranteed.

Not all systems will have enough RAM; not all developers will want to allocate (and thereby block) large parts of memory – especially not for those dialogues in the example, which are rarely used. Knowledge models must therefore be creatable anytime after system startup, using signals containing special logic. (More on that in chapter 8.) Creating a knowledge model only when it is needed is less well-performing but saves a lot of memory. The decision which instantiation paradigm to use finally falls to the developer. She or he has to consider concrete application needs, runtime requirements and the environment.

Again, parallels to biology can be drawn but differences are obvious, too. Persistent knowledge templates represent the *Configuration* after which a system gets created in form of transient knowledge models in memory. This is similar to a *Desoxy Ribo Nucleic Acid* (DNA) providing the building plan for a biological cell. Yet can one not simply create knowledge instances in memory and leave them on their own. Contrary to biological cells which develop and copy themselves, knowledge instances in a software system have to be *referenced*. This is necessary in order to be able to work with them and later to properly destroy them, since forgotten, unfreed memory areas are one main reason for system crashes. Traditional systems lacking a central management of instances had to invent special mechanisms like *Garbage Collectors* (GC), to find and destroy forgotten instances which are not referenced anymore. GCs are known from programming environments like *Smalltalk* [202] or *Java* [112]. The knowledge processing system proposed in this work manages all knowledge instances *centrally*, so that workarounds like GCs become superfluous.

The destruction of biological- and software systems differs, though. While biological systems molder in a diffuse manner over time, a knowledge instance tree representing a software system can be properly folded in the opposite way it was unfolded at creation time. Since, in the proposed system, all knowledge *hangs* on one single root node (section 6.3.2), forgotten instances can be easily identified and destroyed at system shutdown, to clean up the memory.

7 Knowledge Schema

Wise Man's Occupation is Organisation.

THOMAS AQUINAS

As first of the three main topics of part II of this work, chapter 6 investigated why a separation of static knowledge from its dynamic processing in a system is desirable. The sections of this chapter deal with the *Hierarchical Structuring* of knowledge, in order to find a general schema for its modelling. After that, chapter 8 will elaborate on different kinds of knowledge.



7.1 Human Thinking

Knowledge as created by the human mind is learned by associating information, by embedding it in a *Context*, as investigated by *Pragmatics*, a subfield of linguistics [60]. It is thus built of structured, inter-related data (definitions given in section 4.5). The interesting question explored in this section is what kind of structures and relations are used in knowledge models as known from nature, that is *Human Thinking*?

7.1.1 Basic Behaviour

Starting from a neutral view to understanding the universe – and science in general – Stephen Wolfram studied the abstract world of rules put into simple computer programs. *He took*

the lessons from what kinds of things occur there and had them in mind when investigating natural systems, as [60] writes. Wolfram's book *A new Kind of Science* [344] argues that the universe is made up of four basic types of behaviour (figure 7.1):

1. *Repetition*
2. *Nesting*
3. *Randomness*
4. *Localised Structures*

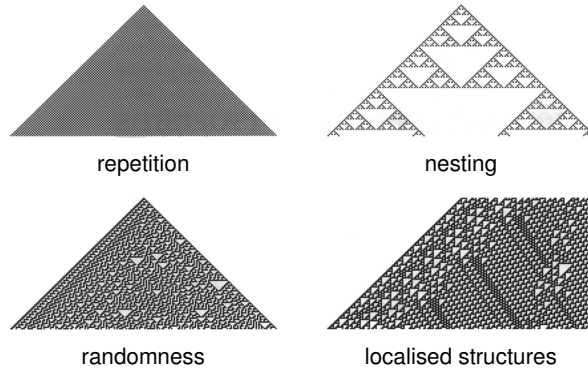


Figure 7.1: Wolfram's Four Basic Kinds of Behaviour [344]

These types of behaviour, after Wolfram, were present everywhere, in nature as in the whole universe. Just everything in existence contained at least one of these structures and all sciences were affected by them. Yet while Wolfram applies results of computing to the study of nature, this work follows the exact opposite way in that it observes phenomena of nature and concepts used in other sciences, and tries to apply them to the design of software systems. Without knowing a final answer – the unexpected surprise is that at least two of Wolfram's findings of basic behaviour match to abstractions as known from human thinking, and have a pendant in this work:

1. *Repetition* is the recurrence of equal structures. Yet before structures can be compared, they have to be demarcated. This is what section 7.1.3 will call *Discrimination* (later *Itemisation*), and also *Categorisation*.
2. *Nesting* creates the famous, beautiful *Fractals*. It is somewhat similar to *Repetition*, only that the repeated structures are not equal in size and do not occur along some chain. While the infinity of *Repetition* lies in its neverending *Continuation* along some line(s) on the same level, it lies in the *Diving* into a deeper level for *Nesting*. Section 7.1.3 will call this *Composition*.

Wolfram defines a *Principle of Universality* which states that in fact *all* kinds of systems, even very simple ones, are capable of showing complex behaviour in form of *Localised Structures*, once some threshold in the complexity of the underlying rules is passed. But where is this threshold and who defines what complex behaviour actually is? Does a threshold exist at all or are the four kinds of behaviour in fact not different? Indeed, one could argue that neither *Repetition*, nor *Nesting*, nor *Randomness* are anything special and that it is just the human mind interpreting them as something special, as assumed by this work. It claims that the human mind gives structure and meaning to the surrounding real world by building a virtual world. The principles of human thinking are therefore investigated in the next sections.

Software abstracts human thought which, in order to understand and act in the surrounding real world, needs to recognise and rely on *known* patterns. *Randomness* and *Localised Structures* in Wolfram's meaning, although existent in universe, are rather not used by the human mind to store information, nor do they seem useful for the design of deterministic software systems. Of course, information can arrive at- and influence a human mind in an arbitrary manner and be associated randomly, at will. But the concepts it forms need to be stable in order to build up knowledge. The research done in this work therefore relies on reproducible structures making sense to human minds, namely *Repetition* and *Nesting*.

7.1.2 Conglomerate

Universe is the most general word describing everything humans think exists. Whether it exists in reality or just as an illusion in their minds, is a fundamental question of philosophy and will not be discussed here. The author of this document assumes that a *Real World* exists and humans only *reflect* but not *construct* it in their minds.

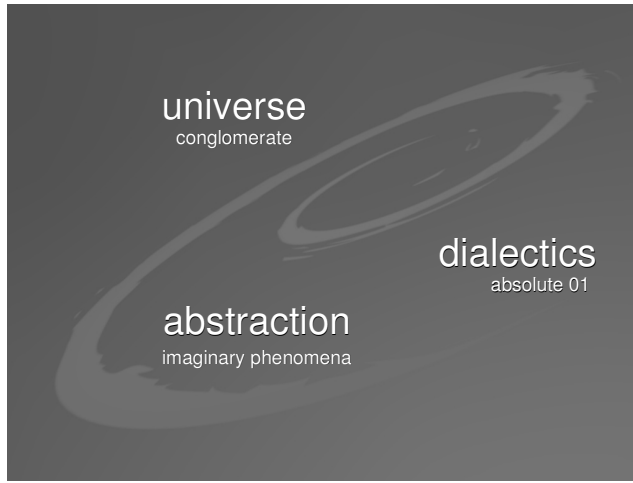


Figure 7.2: The Universe as to-be-abstracted Conglomerate (swirl from [258])

The whole universe can be seen as *Conglomerate* of everything in existence (figure 7.2). Computer systems enable humans to abstract things to just two states labelled *0* and *1* (section 4.1.3), on a very low level. Human systems use more high-level methods for abstraction. Common concepts to describe a real-world environment are *Particle*, *Dimension* or *Force*. Everybody will know more of them. But what is a particle?

7.1.3 Abstraction

Humans understand their environment by building simplified models (concepts) of it. These are based on fundamental *Abstractions* like *Item*, *Category* or *Compound* (figure 7.3), which are the topic of this section. Part of it was already published in [125].

Item

As first and most important abstraction, the human mind divides its real-world environment into discrete, countable *Items*. Physicists call smaller items *Particle*. Plenty of other synonyms exist. Software developers often talk of *Object*. This document preferably uses the more neutral name *Item*, since models are created not only of objects but also of *Subjects*.

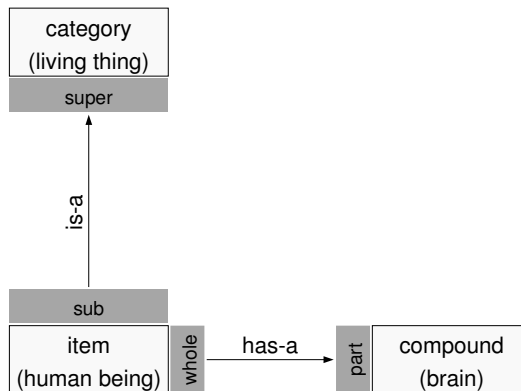


Figure 7.3: Abstractions of Human Thinking

Behavioural psychologists talk of this ability as *Discrimination*. It commonly focuses on a specific real world phenomenon, leaving out parameters which are not interesting in the given context. This is necessary because otherwise, a brain would have to model and capture the whole universe (with every single particle being duplicated), which is obviously impossible.

Not only human beings, but also some higher animal species (like apes) are able to *discriminate* their environment and to form terms to name it. (More on *Term* and *Language* in section 7.1.6.) Additionally, they have a primitive *Self Concept*, that is a term for their own personality. However, their cognitive abilities are limited in that concepts are only available in the presence of the corresponding object (item). Jaeger [165] calls that *Online Thinking*; cognition scientists speak of *Terms of first Order* or *Sensoric Type of Terms*.

Contrary to this, the more advanced *Offline Thinking* [165] allows humans to think about objects (items) they currently cannot sense. Cognition scientists here speak of *Terms of second Order*. They became possible by *associating* sensoric signals with terms of a language. The resulting *Net of Associations* brought a number of advantages [165]:

- *Decoupling* of thinking from immediate motoric reaction
- *Time Index* in scenes so that past memories can be recalled, the future be planned
- *Dual Representation* of online and offline contents
- *Self Awareness* thanks to online and offline thinking

- *Associations* increasing the expressiveness of terms

Self awareness is important for systems to know about their own capabilities, like those for information input/output (i/o). More on that in chapter 8.

Category

Offline thinking (in terms of second order) enables humans not only to discriminate items but also to *categorise* them into superior groups. Since it is impossible to exactly model the real world in complete, compromises have to be made: People do not model every single item in their minds but rather group them into *Types (Classes)* of common characteristics.

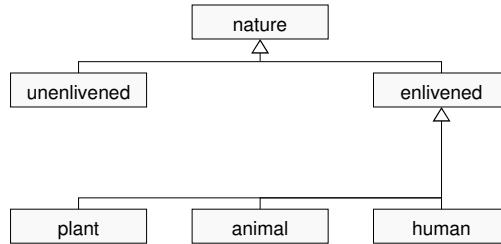


Figure 7.4: Systematics of Nature

This kind of classification stems from the earliest days of ancient science. *Plato's* (429-347 B.C.) pupil *Aristotle* (384-322 B.C.), being the teacher of *Alexander the Great*, was the first philosopher who logically captured and organised the world. It was him who sorted items into clear groups which he called *Categories*. And it was him who first distinguished between *enlivened* and *unenlivened* nature; who parted living forms into *Plants*, *Animals* and *Humans*. The science of biology calls this classification a *Systematics* (figure 7.4).

Categorisation (classification) can be seen from two sides, depending on what direction of that relationship one wants to emphasise. Taking *Aristotle's* examples, *Living Thing* would

be a *Generalisation* of *Plants*, *Animals* and *Humans*. *Animal* would be a *Specialisation* of *Living Thing*.

Software developers often call categorisation an *is-a* relationship and talk of *Super* and *Sub* categories (sometimes also *Parent* and *Child* categories). Section 4.1.15 described how *Object Oriented Programming* (OOP) uses categorisation to let a sub class inherit attributes and methods from its super class.

Compound

Composition is the third kind of abstraction that humans use to understand their environment. It is an important instrument for the human mind to associate information, that is to acquire, store and recall *Knowledge*. Every item can be recognised as a *Compound* of smaller items and can therefore also be called *Tree* or *Hierarchy*. The subject of *Artificial Intelligence* (AI) talks of *Concept* or *Schema* [294].

The great philosopher and mathematician Gottfried Wilhelm Leibnitz (1646-1716) made extensive use of the principle of hierarchy. His entire theory of *Monades* [192] is based on it.

In software design, the terms *Parent* and *Child* are often used to describe both, the items in a composition relation and the items in a categorisation (inheritance) relationship. To avoid misunderstandings, this work sticks to the terms *Super* and *Sub* for categorisation and to the terms *Whole* and *Part* for composition. Yet other terms to describe items of a composition would be *Container* and *Element*.

One obvious analogy comes from *Genealogy* where parents have children who become parents of their own children and so forth. Displaying these relations between family members in a graph, leads to a tree which may represent both, a category tree (for property inheritance) as well as a composition tree (for children ownership).

Taking the example of a *Human Being*, one could say that it is composed of organs such as *Eye*, *Ear*, *Heart*, *Brain*, *Arm* and further, also smaller parts. John F. Sowa [294, p. 109] writes on this:

From different viewpoints, the human body can be considered an aggregate of organs, an aggregate of cells, or an aggregate of molecules. Each viewpoint affects the terminology used to talk about the body, but not the body itself.

That is, one and the same real world object may be represented in many different ways. It is important to note the *unidirectional* kind of relations: A human being is composed of organs but an organ is never composed of a human being! Another example is that of a *Book*: physically, it may be composed of a *Paperback Cover* and *Paper Pages*; logically, however, it is usually separated into *Part*, *Chapter*, *Section*, *Paragraph*, *Sentence*, *Word* and *Character*. Obviously, knowledge representation always depends on what one wants to express in which context. Philippe Ameline who works in the *Nautilus-Odysee* project [215], writes in [168]:

Lets take a comparison with *Geography*: you can build an ontology (which consists of compound structures) in order to describe natural objects (mountains, rivers, ...). But if you build artificial frontiers and call them *Countries*, you cannot semantically include these concepts inside the geographical domain. – That is the very reason why human beings, very frustrated, had to invent the political domain ;-) ... I think that there (are similar differences) in medicine (as in) the geographical domain.

Not only *States* may be represented as compound; *Procedures* may be hierarchical as well. The process *Take Book from Library*, for example, may have the following structure:

- Check Catalogue
 - Investigate suitable Books
 - Note Registration Number
- Obtain Book
 - Look for Shelf
 - Take off Book
- Borrow Book

Returning to human thinking, one realises that in the end, everything in universe can be put into variable hierarchical models, that is consists of smaller items and belongs to a bigger item. From the physical point of view, nobody knows where this hierarchy really stops, towards *Microcosm* as well as towards *Macrocosm*. There is no *absolute*, *basic* item. A *Particle* as concept exists only in the human mind, placed somewhere between micro- and macrocosm, with hypothetical borders.

7.1.4 Interaction

As was explained before, most abstract models of the human mind have a composed nature, that is consist of smaller parts. If being compounds, they hold certain information about their parts, namely their name, model and abstraction. More on this in section 7.3. But isn't there other detailed knowledge a compound must have about its parts? What about the order or position, the size or colour of parts within their compound?

Meta Information

To find an answer, the science of *Psychology* needs to be called in. It distinguishes between various aspects of a (visual) impression of the human mind, as there are *Movement*, *Shape*, *Depth* or *Colour* [298]. Looking closer at these, one quickly realises that they contain representations of the classical physical dimensions that humans use to describe the world:

- *Movement* stands for changing the state of something over *Time*
- *Shape* is how items would appear in a two-dimensional world, as known from *Geometry*
- *Depth* (which is possible to recognise thanks to the human's ability for stereo vision) adds a third dimension to shapes, so that these become three-dimensional and form a *Space*
- *Colour*, not being considered a dimension, tells about how items reflect *Light*

Another physical value often used to abstract and describe the world is *Mass*. Again, it is not considered to be a dimension. If, according to modern physics, not all of the impressions listed above are dimensions, what else is common to them? – All are used to express a special *Interaction*. (Einstein [77] would probably prefer the term *Relation*, to better point out the relative nature of at least the space and time, in which a whole and its parts interact.) To avoid conflicts with other sciences, this document sticks to the term *Conceptual Interaction*.

The following paragraphs will describe some conceptual interactions in more detail and give examples for their understanding.

Space

To the common concept of an *Atom* belong a *Core* and *Electrons*. The atom provides the *Space* that the core and the electrons can fill with their extension. For core and electrons,

the atom represents the small universe they live in. Moreover, the atom *knows* about the *Position* (more correct *Trajectory*) of each electron. Thus, one can say that the atom as a *Whole* interacts with its *Parts* by means of space. Electrons, on the other hand, know nothing about their own position within the atom; they do not know about the existence of the atom at all.

A different example would be the *Graphical Frame* of a software application. It has an expansion that cannot be crossed by its children. Children may be a *Menu Bar*, *Tool Bar* and *Status Bar*. In order to be positioned correctly, the frame has to know about their coordinates or orientation. Again, this can be seen as an interaction over space.

A third and last example that was already stressed in previous sections would be the *Human Body* consisting of organs like *Heart*, *Brain* and *Arm*. Each organ has its special position within the body concept. However, it is always useful to keep in mind that models (concepts) are an abstraction, an *Illusion*. Taking the example of the human body, how is it constituted? Does belong to it the:

- Air in its lungs
- Sweat leaving its skin
- Radiation crossing it
- Food being eaten

The human body, in reality, is not stable; it changes permanently, in all dimensions. Human thinking only makes it stable by characterising it with arbitrary properties, picked out of millions. It actually exists (in the same state) for just an infinitesimal instant in time. The same counts for any other real world items. Some elementary or yet smaller particles have a lifetime of only a fraction of a second. But even within this minimal lifetime, they probably take on millions of different states.

Mass

A *Solar System*, as concept, has very much in common with the atom. It has a star, the *Sun*, as its core and it has *Planets* orbiting around that star. Besides the conceptual interaction over space that also exists here, there is another relation worth paying attention to: *Mass*. It has great influence on the *Gravitational Force*.

Conceptually, the solar system can be treated as a closed field of *Mass*, the sun representing the centre, the planets additions. The solar system as a *Whole* knows about the masses of its *Parts*, what can be considered a conceptual interaction.

Another example, taken from informatics, are *Artificial Neural Networks* (ANN) consisting of *Neurons* and *weighted* connections between them. Ideally, the ANN knows about its neurons – or, depending on its design, the layers that contain them – and the corresponding connections. This structural information needs to be complemented by *Weight* (*Mass*) information indicating the strength (importance) of an association between neurons.

Time

A third kind of conceptual interaction that humans use to place themselves and the environment into their very own model of the universe is *Time*. Section 7.1.3 showed on the example of *Take Book from Library* that any *Process* can be split into *Sub Processes* and thus represents a structure with *Hierarchical Character*.

In most cases, the *Order* in which sub processes are executed, is very important. Without it, no meaningful *Algorithm* could ever be created. A process thus needs to know about the *Occurrence* of its sub processes and this sequence information is usually stored in units of time.

Moreover, the *Whole* process sets a time frame that all *Part* processes, in sum, cannot exceed. Their *Duration* is limited. Again, process and sub processes have some kind of conceptual relation; in this case over time.

Constraint

The previous sections have discussed three kinds of conceptual interaction: *Space*, *Mass* and *Time*. They are used by a model (concept) to position parts within its area of validity.

Yet this meta knowledge is not enough. Frequently, parts have to be *constrained* to maintain the validity of the whole model. The concept of a *Table*, for example, may consist of a *Top* and one to four *Legs*. The additional meta information herein is the constraint of the number of legs to at least *one* and at most *four*.

Another example regards the area of valid values that parts can take on. The *Temperature* of an alive human body lies somewhere between a *Minimum* of +30°C and a *Maximum* of

+40°C (broad-minded estimation). A corresponding model has to remember these extrema in order to be able to limit numbers to the correct temperature range.

7.1.5 Intrinsic or Extrinsic Properties

Properties may not only represent the *Position* of a part in space/ mass/ time, but also its *Size* in the same dimensions. For space, it may be called *Expansion*, for mass *Massiness*, and for time *Duration*. While a size is always represented by the *Difference* of two values, a position is represented by a *Point*. No matter what the kind of property – all of them are stored as meta information in the compound model, that is external to the parts.

The abstraction principles used in this work thereby *differ* from the *Benediktine Approach* as introduced by Michael Benedikt in his article on the structure of cyberspace [25]. That distinguishes *extrinsic* and *intrinsic* spatial dimensions, to which the properties (attributes) of an item (object) may be mapped. While extrinsic dimensions in a graphical dialogue, for example, would be the positions of the buttons contained in it (location in space), intrinsic dimensions would be those that are contained directly in the buttons which they describe (shape, size, colour).

In other words, the *Benediktine Approach* proposes to keep some properties *outside* the described model (meta knowledge) and others *inside* the model (self-knowledge). Having reflected on the principles of human thinking, and speaking in Benediktine's terminology, this work, however, proposes a knowledge schema (section 7.3.2) which uses solely *extrinsic* properties.

7.1.6 Language

Having investigated the basic principles of knowledge modelling as applied by the human mind, this section now deals with the question how thoughts are put into language, in order to be communicated and stored.

Philosophers, evolution biologists, linguists and further scientists investigate human thinking and in particular the *Relationship* between language and thinking. Many of them hold the view that *Language*, *Cognition* and *Awareness* have developed hand-in-hand, during phylogeny of man.

After the paleoanthropologist Andre Leroi-Gourhan [165], the upright, two-legged walk caused a change in the geometry of the human skull which lead to the creation of new brain areas that today hosted important functionality for higher thinking. Two prominent areas situated in the *Cerebral Cortex* were the *Broca Area* (for language production) and the *Wernicke Area* (for language recognition). Jaeger [165] writes that it must have been in that time that the human brain had developed the fundamentally important ability to represent objects of the environment in a completely new, advanced kind of *Terms*.

A *Term* is an *Abstraction* which stands for or describes (a part of) the real world. It is terms (also called *Words*), and combinations of these, which form a *Language*. Combinations of terms are the *Phrase* or *Sentence*. All of these are also called *Unit*. Since forming words is much the same as building knowledge models, it is no surprise to find again the three abstraction principles of *Discrimination*, *Categorisation* and *Composition*, only that the second of these is called *Derivation* here [2].

The rules (*Patterns*) for combining terms are the *Syntax* (or *Grammar*) of a language. The meaning expressed by terms and sentences is their *Semantics* [71]. Collections of terms of a language are called *Vocabulary*, sorted collections a *Lexicon*.

For the philosopher Aristotle, reality existed completely independent from human cognition and language was not needed to understand things. Wilhelm von Humboldt (1767-1835) and contemporaries saw language as the *Organ forming Thoughts*. Today, philosophers distinguish three levels of language exerting influence [165]:

- The way its vocabulary divides the world (*Lexicon Structure*)
- Its physical appearance (*Materiality*)
- General properties (Is language just *reflecting* or *constructing* reality?)

Mapped to informatics, one might evaluate these three points as follows: For point one, some description was given in section 4.6.5 (*Terminology*). It is the actual arts of programming to structure and divide a particular domain into expressive parts, using terms and constructs of these. The second point is fundamentally important as it defines the final abstractions of terms of a language in software. As mentioned in section 4.1.3, in informatics, every piece of information (term) gets abstracted to only two states: *0* and *1*. Point three is left to the philosophers to further philosophise.

But what are the basic representations of a term, above the digital level?

In the first instance, one needs to distinguish between the *social* and *egocentric* form of language. The latter may exist as some kind of *inner language* (also called *internalised natural language*) of the human brain and is what is normally called *Thinking*. Scientists are not absolutely sure about its existence yet but the research works of the Russian psychologist Lew Demjonovitch Wygotski and the philosopher Peter Carruthers [165] show into that direction. Some even define thinking as *suppressed motoric action* [165, p. 41-42].

Materiality (Latin/ Greek)	Medium	Organ	Function
Visually/ Optically	Light	Eye	Seeing
Auditorily/ Acoustically	Air	Ear	Hearing
Odorously/ Osphrantly	Air	Nose	Smelling
Gustatorily/ Geustically	Substance	Mouth/ Tongue/ Palate	Tasting
Tactorily/ Haptically	Mass	Hand/ Skin	Fumbling/ Groping/ Feeling
		Inner Ear	Equilibrate/ Bal- ance
		Proprio Receptors	Perceive Motion/ Movement

Table 7.1: Materiality of Language, according to Five Human Senses [39]

The social (*communicative*) form of language is determined by the five *Human Senses* (table 7.1). Mankind has invented manifold kinds of abstracting and associating real world items, for example as *Gesture*, *Sound*, *Speech*, *Music*, *Image*, *Script*, *Video*, *Smell*, *Taste* or *Touch*. They all can serve as terms for communication, being part of a language.

It is important to notice that a term can not only be expressed by a *Word/ String*, what is frequently associated with it. As well, an image or sound can represent a term. In information science, the expressions (or *Materialities*) of language are the basic data blocks for information storage. Signs/ Characters, Texts, Images, Sounds and Videos are stored in special *Resource Files* or *Databases* (DB). Their data are not merged into the actual programming language code; they are just referenced from- and handled there.

7.1.7 Quality and Quantity

Languages, in general, do not only contain terms that associate some *Item*, also called a *Quality*; they do also offer terms representing a *Number*, also called a *Quantity*.

The science dealing with numbers is *Mathematics*. It uses different types of numbers, for example *Integer*, *Fraction* and *Complex*. A fraction is a combination of two integers (*Numerator* and *Denominator*). The two parts of a complex (*Real* and *Imaginary*) consist of one fraction each. Again, the composed nature even of numbers becomes obvious.

Many mathematical number types have their counterpart in programming languages. Two common ones that are mostly implemented as primitive types are *Integer* (Byte, Short, Long) and *Float* (Double). The variations given in parentheses differ from the basic type only in their range. Further, more complex number types need to be extra-coded, by combining primitive types.

Numbers can be organised in a *Numbering System* whose basic rules involve:

- Ordering items
- Grouping ordered items
- Expressing groups and items in a consistent way

Typical examples are the *Roman*- and the modern *Arabian* numbering system, the latter also being called *Algorism*. For historical reasons (10 fingers of human hands), most systems of that kind use a number base of 10. A *Number Base* value is implied by any use of numbers, as [293] annotates:

The simplest base value to use in a numbering scheme is 1. In this scheme, the number 2 is two things, or two groups of ones. The number 7 is seven things or seven groups of ones. Evidence of numbering in this fashion has been found in archaeological (excavation) pieces, dating as far back as 37,000 years.

Other examples of number base systems are:

- Binary (Base 2)
- Octal (Base 8)
- Decimal (Base 10)
- Duodecimal (Base 12)

- Hexadecimal (Base 16)
- Sexagesimal (Base 60)

In order to understand their environment, humans not only need quality terms, but also quantity terms (numbers) to count qualities. The primitive forms of both serve as final abstraction in the virtual models existing in the human mind.

7.2 Design Reflections

The previous sections investigated principles of human thinking, that is the structures and relations used by the human mind to build abstract models of its real world environment. The following sections focus on the impact of just these principles on software design and suggest a number of changes while rethinking state-of-the-art concepts.

7.2.1 Pattern Systematics

Software Patterns (section 4.2) are a popular architecture instrument of current systems and languages – in the first line, however, of *Object Oriented Programming* (OOP) (section 4.1.15). They describe design solutions that belong to a higher conceptual level, as opposed to the programming paradigms which are inherent to languages. A common criticism on the existence of patterns is put into words by the free *Wikipedia* encyclopedia [60] which writes:

Some feel that the need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should *not* be *copied*, but merely *referenced*. But if something is referenced instead of copied, then there is no pattern to label and catalog.

In other words, patterns would become superfluous, if they could be applied just *once* to a system, in a manner that allowed any other parts of that system to reference and reuse-, instead of copy them.

Cybernetics Oriented Programming (CYBOP) wants to eliminate the need for repeated pattern usage, and such enable application programmers, and possibly even domain experts, to faster create better application systems. On the way to reaching such sublime aims, a first step is to look at current pattern solutions and try to identify what their common characteristics are. This was already done in section 4.2, which used traditional proposals

[41, 108] to systematise patterns and divided them according to the first categorisation level shown in figure 4.14, into *Architectural*-, *Design*- and *Idiomatic* patterns.

This section proposes a *new* systematics to classify software patterns. It is based on the idea of classifying them after the principles of *Human Thinking*, as described in section 7.1 before. These fundamental principles are: *Discrimination*, *Categorisation* and *Composition*. Applied together, they may form an abstract *Schema* (introduced later, in section 7.3.2).

The latter two activities of abstraction – categorisation and composition – are based on special *Associations* (figure 7.3), between a *Super*- and a *Sub* model and between a *Whole*- and a *Part* model, respectively. Most patterns heavily rely on associations, too. This work therefore suggests to [129]: *Take the kind of association as criterion to sort patterns in a completely new way.*









Category	Equivalent	Representative	Advice
Itemisation	Discrimination	Command, Data Transfer Object, State, Memento, Envelope-Letter, Prototype	
1:1 Association	Composition	Delegator, Object Adapter, Proxy (Surrogat, Client-/ Server Stub), Wrapper, Handle-Body, Bridge	
1:n Association	Composition	Whole-Part, View Handler, Broker (Mediator), Master-Slave, Command Processor, Counted Pointer, Chain of Responsibility	
Recursion	Composition	Composite, Interpreter, Decorator, Linked Wrapper	
Bidirectionality	–	Observer (Callback, Publisher-Subscriber), Forwarder-Receiver, Chain of Responsibility, Visitor, Reflection	
Polymorphism	Categorisation	Template Method, Builder, Factory Method, Class Adapter, Abstract Factory (Kit), Strategy (Validator, Policy), Iterator (Cursor)	
Grouping	Categorisation	Layers, Domain Model, MVC	
Global Access	–	Singleton, Flyweight, Registry, Manager	

Table 7.2: Pattern Systematics

Table 7.2 shows a systematics of the new pattern categories with their equivalents in human thinking, some representative example patterns and a recommendation for their usage in software engineering. Patterns matching into more than one category are placed after the priority: *Recursion* over *Polymorphism*.

7.2.2 Recommendation

The first category *Itemisation* (objectification) is the base of any modelling activity and clearly necessary.

The next three categories *1:1 Association*, *1:n Association* and *Recursion* are special kinds of associations that rely exclusively on *unidirectional* relations and result in a clean architecture which is why their usage is strongly recommended.

Bidirectionalism, on the other hand, is an *ill* variant of the three aforementioned categories and should be avoided wherever possible. Patterns in this category are one reason for endless loops and unpredictable behaviour since it becomes very difficult to trace the effects that changes in one place of a system have on others (section 4.2.2).

Polymorphism is a good thing. It relies on categorisation and due to inheritance can avoid a tremendous amount of otherwise redundant source code. However, it also makes understanding a system more difficult, since the whole architecture must be understood before being able to manipulate code correctly. Unwanted source code changes caused by inheritance dependencies are often described with the term *Fragile Base Class Problem* (section 4.1.15).

Grouping models is essential to keep overview in a complex software system. A very promising technology to support this are *Ontologies* [127]. A lot of thought-work has to go into them but if they are well thought-out, they are clearly recommended.

The habit of *globally accessing* models is banned since OOP (section 4.1.15) became popular. However, it is not banned completely. Patterns like *Singleton* encapsulate and bundle global access but they still permit it. They disregard any dependencies and relations in a system, such are a security risk and reason for untraceable data changes. This work sees the whole category of *Global Access* as potentially dangerous and *cannot* recommend its patterns (section 4.2.3).

To sum this up: The different kinds of software patterns investigated in section 4.2 showed

various advantages, but also weaknesses (bidirectionality, global access, partly polymorphism), which became obvious through the new pattern systematics introduced in the previous section. It now turns out that the weaknesses show up in exactly those categories of patterns, which do not follow the principles of human thinking. The resulting recommendations of this section were considered in the design of the *Cybernetics Oriented Language* (CYBOL) and the *Cybernetics Oriented Interpreter* (CYBOI), described in the later chapters 9 and 10.

7.2.3 Model Metamorphosis

One way to recognise the importance of *Composition*, that is of models with hierarchical character, is to compare several traditional modelling approaches, as first suggested by Thomas Beale in [18, p. 11-18]. This *Metamorphosis of Models* is empathised in the following paragraphs.

Single Model

Today, the most common design approach for standard application software is to create a *Single Model* of types whose semantics is often described in form of an *Entity Relationship* (ER) or *Object Oriented* (OO) model, the latter sometimes illustrated using diagrams of the *Unified Modeling Language* (UML).

As example, figure 7.5 shows a UML *Class Diagram* (CsD). In its upper half, one can see a class *Person* associated with the classes *Name* and *Address*, as modelled at design time. They may be part of a much larger model. The lower half of the figure shows the objects (instances) at runtime, filled with concrete values. There are a number of problems with this approach:

Inflexible Architecture First and foremost, the static coupling of classes leads to an inflexible design. The names and number of attributes and methods as integral part of a class cannot be changed dynamically later-on; only their values can. The class structure represents a solution to a current problem. If it is static, then future requirements cannot be considered. Adaptation issues and workarounds, affecting stability and security, are thus to be expected.

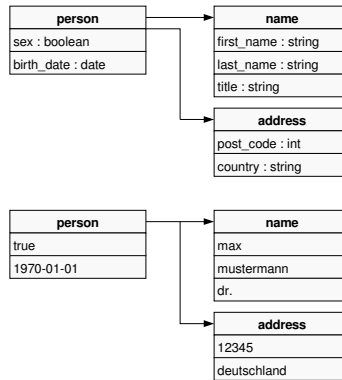


Figure 7.5: Single Model Approach (adapted from [18])

Concept Mix Further, specialised domain concepts identified during requirements analysis (such as a *Patient* being a kind of *Person*) are often mixed up with more general concepts as found during design (for example the application of a proper *Role* architecture instead of simple inheritance for the person-patient relation). The lack of a proper separation between pure domain knowledge (like a patient receiving a medication) and system control software (like logging facilities or persistence mechanisms) was already explained in detail in the previous chapter 6. It frequently leads to strong coupling between system layers and complicates software design.

Synchronisation Problems The mix of application knowledge with system control software also causes synchronisation (communication) problems within software development projects. Domain experts and software developers depend on each other: Developers need to first understand domain knowledge before being able to correctly implement it into software. Experts bring their knowledge into a more software-friendly form, during analysis.

Complicated Processing Due to the great variety of software architectures, it is pretty hard to capture and process data from different systems in a uniform way, for reasons of *Data Mining*, for example. Unpredictable architecture changes caused by new domain requirements hamper the creation of reliable rules for *Decision Support*.

Steady Upgrading Applications that were designed in a *Single Model* manner require steady upgrading. Whenever new domain knowledge gets worked into the system or existing knowledge gets adapted to new requirements, the software design may change – even in important parts that would better remain stable. Accordingly, systems of that kind cannot be labelled *future-proof*.

No Standardisation Finally, a true standardisation of single model systems is hardly reachable. Requirements are just too different between the systems, and they change much too often. A standard architecture of that kind won't remain stable for very long.

Semi Structured Model

A slightly improved version is the *Semi Structured Model*. It relies on the usage of *Named Values* (sometimes called *Tagged Values*), which are stored in a dynamically extensible structure such as a list. That way, future attributes can be added smoothly, without having to change the overall model.

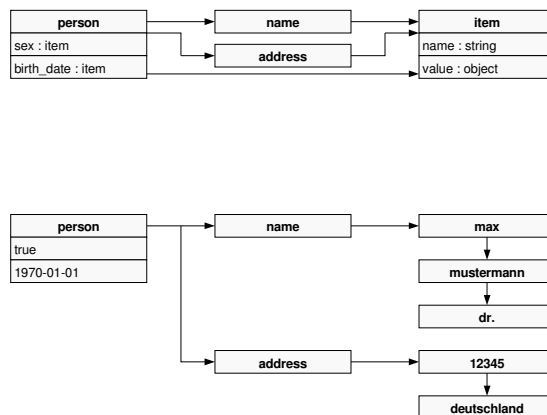


Figure 7.6: Semi Structured Model Approach (adapted from [18])

The example in figure 7.6 shows a class *Person* referencing two linked lists, one called *Name* and another called *Address*. The list elements are of type *Item*. The dynamically extensible

structure becomes more obvious in the lower half of the figure, showing how the single list elements reference each other. However, also here one can find disadvantages [18]:

- Not all fields are dynamically changeable. Some are concrete attributes.
- Only single lists of named values which do not allow for more complex internal structures are used.
- Variability in structure is not generally dealt with.
- Type information is lost for all list elements, since they are of one common type.
- The system does not know anymore which elements are required.

Hierarchical Model

The *Hierarchical Model* as yet more generalised form of data representation is based on *Composition* as one of the principles of human thinking (section 7.1.3). Its tree structure – ideally in form of a *Directed Acyclical Graph* (DAG) (section 4.6.5) – allows dynamic extensions of data types, by simply adding child nodes (parts) to a parent node (whole), in the knowledge tree.

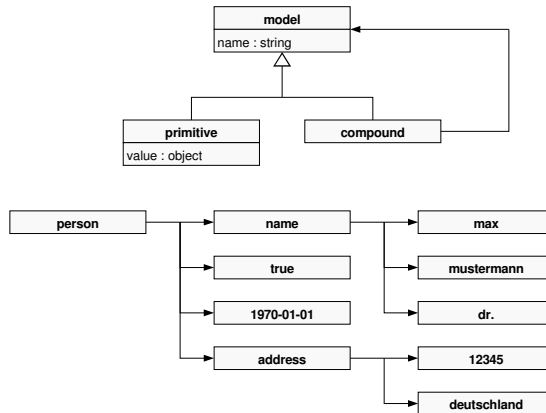


Figure 7.7: Hierarchical Model Approach (adapted from [18])

The upper half of figure 7.7 shows the *Composite* software pattern (section 4.2.2). Its classes do not contain hard-coded attributes; two exceptions are the *Primitive* class' attribute *value*

and the *Compound* class' dynamically extensible structure. The structure may be a list, and it stores all of the compound's parts in it. Parts may be primitives or compounds themselves. As already proposed by the *Semi Structured Model* before, each part is identified by a name that is unique within its compound. The diagram in the lower half of the figure clearly shows the hierarchical tree structure of runtime instances.

The only open issue when using purely hierarchical models is that the semantics – the actual domain concepts – is lost. Knowledge models, together with their parts and meta information about these (position, size, colour, constraints – as described in section 7.1), thus need to be defined somewhere else. The later section 7.3 proposes a generic knowledge schema for doing this.

7.2.4 Structure by Hierarchy

The principle of *Composition* not only allows the creation of highly flexible models, the *Hierarchies* it makes up allow humans to combine several concepts in a common, greater model. *Structure by Hierarchy* as idea has been applied to numerous *Knowledge-* and *Domain Models*, especially in the fields of *Artificial Intelligence* (AI) and *Knowledge Engineering* (section 4.5). But obviously, it has not been used for the design of *complete* systems yet, even though this seems quite logical.

To demonstrate the omnipresence of *Hierarchies* in a system, the parts of the *Model View Controller* (MVC) software pattern described in section 4.2.1 shall be considered once again. The MVC is a very representative example as it is used in one or another form by a majority of systems, today.

The MVC pattern (figure 7.8) consists of a *View* that is mostly implemented as *Graphical User Interface* (GUI) frame with panel, menubar, menu items and smaller components which, in this order, are all part of the frame's hierarchy. Then, there is the *Controller*. The *Hierarchical Model View Controller* (HMVC) pattern (section 4.2.1) suggested to use a controller hierarchy consisting of *MVC Triads*. Finally, there is the *Model* of domain concepts which not only knowledge systems can structure hierarchically. Reflecting these facts, one question is at hand:

If View, Controller and Model ideally have a hierarchical structure,
why not creating whole software systems after this paradigm?
Isn't every system, in essence, just a *Tree* of items?

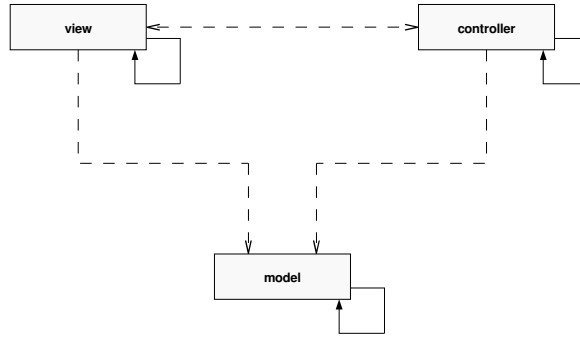


Figure 7.8: Adapted (H)MVC Pattern with Hierarchical Elements

7.2.5 Association Elimination

To pursue the idea of a purely hierarchical software system, it seems useful to investigate in which way domain data models can get simplified. This section therefore demonstrates how the principle of *Hierarchy* may be applied to obtain an *Ontology* [127].

An *Electronic Health Record* (EHR) will serve as example domain model, whose simplified structure is shown in figure 7.9. It consists of numerous parts whereof two may be *Address* and *Problem*. Following the *Episode-based EHR* recommendation [341], *Problem* may, besides others, consist of parts of type *Subjective* and *Objective*. All these associations between part models are needed to navigate through the overall domain model.

A frequent design decision in classical *Object Oriented Programming* (OOP) is to sum up common properties of *Sub* models by introducing a *Super* model (category). It should never be *Properties*, but rather the *Granularity* of objects leading to the creation of a super category, as the later section 7.2.8 will recommend. The *OpenEHR* project [22] suggests to let the above-mentioned sub models inherit from the more coarse-grained super categories *Record*, *Unit* and *Heading*.

Whichever reason – once the super categories are there, they should be associated similarly to their sub categories, that is in the same direction, using solely *unidirectional* depen-

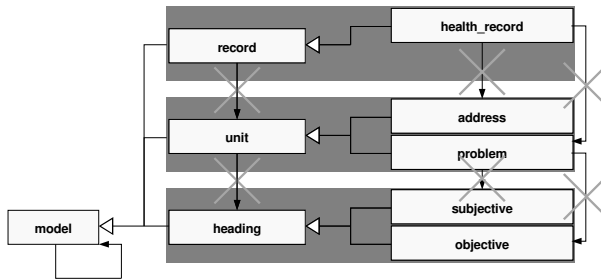


Figure 7.9: Association Elimination in an EHR

dencies. (The problematic nature of bidirectional dependencies was elaborated in section 4.2.) Afterwards, all associations between sub categories become superfluous as every sub category can reach its sibling across the super categories' association (figure 7.9). In other words: *Super- eliminate Sub Associations*.

Here a short Java code example for how the *HealthRecord* may retrieve a reference to *Address*:

```
Address a = (Address) get_element("address");
```

HealthRecord inherits the *get_element* method from its super category *Record*. *Record* holds differing sub models of category *Unit* and other instances. The *get_element* method delivers back a general *Model* that still needs to be down-casted to the expected sub category *Address*.

The definition of models, their dependencies (defined by associations) and granularities (defined by inheritance) in a software system results in several *Layers* of models of common granularity (figure 7.10). These layers are often called *Ontological Level* as they, together, form an *Ontology* (sections 4.6.7, 7.3.1).

An ontology of that kind can, of course, be created for every knowledge model. The financial sector – like an insurance company, for example – may use an *Insurance Record* with comparable structure.

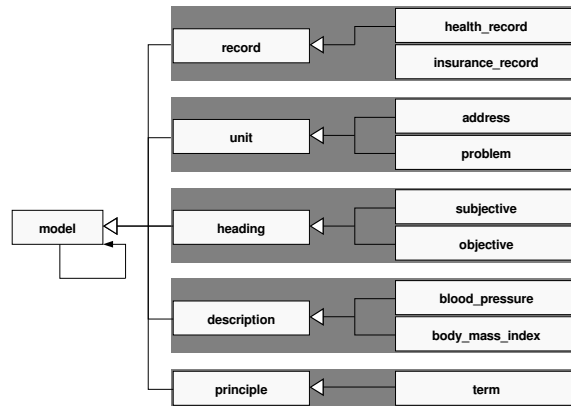


Figure 7.10: Model Container and Ontological Levels

The idea to structure software as a system of *Layers* was also suggested by the equally named pattern in section 4.2.1. The difference between the two is that the *Layers* pattern divides a system only by its *Functionality*, for example into *User Interface*, *Domain*, *Data Mapper* and *Data Source*. An *Ontology* additionally groups model items by their *Granularity*. By inheriting from a common superior category, sub categories indicate that they logically belong to the same *Layer*.

Continuing the structuring process of introducing more and more common super categories, for all equally-granular items, the development culminates in one top-most super category of all other categories in the system, which this paper calls *Model*. It is as general as the *java.lang.Object* class for the Java class library [112], only that it additionally represents a *Container* that can store models of any category, as explained in [126]. In other words, *Model* provides the meta functionality of a container behaviour to *all* other categories in a system.

7.2.6 Hierarchical Algorithm

Of course, algorithms, workflows and other activities over time can be structured hierarchically as well.

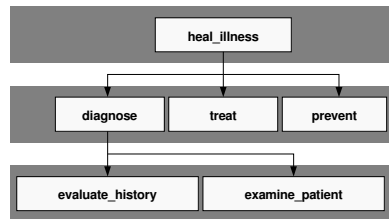


Figure 7.11: *Heal Illness* as Hierarchical Algorithm, taken from Medicine

To stick with the domain of medicine: A *Medical Doctor's* (MD) activity is to *Heal an Illness*. Part processes the MD has to carry out commonly are to *Diagnose*, to *Treat* and to *Prevent* an illness (figure 7.11). Further structurings are possible. To the process of diagnostics belong activities like *Evaluate History* or *Examine Patient*.

7.2.7 Framework Example

A much more complex example than the EHR structure demonstrated in section 7.2.5 is the *Reference Information Model* (RIM) framework of the *Health Level Seven* (HL7) standardisation organisation (chapter 11). It is a quite typical software model, developed in a *Single Model Approach* (section 7.2.3), as it may similarly exist in other business areas. The coloured legend in figure 7.12 helps distinguish the various parts of the RIM. The part to be picked out to have a closer look here is the several kinds of RIM *Entities* (figure 7.13).

Diving yet deeper into the framework, one will find the *Person* class, being a sub class of *Living Subject* (figure 7.14). Since the RIM is an *Object Oriented* (OO) framework, each class will probably have access methods for all of its attributes. For the *Person* class, the access methods for *birthdate* and *address* are shown in the figure. The right-hand side also shows the typical one-line contents of the *set/ get* methods, although more code may be put into them, for reasons of notification, update or others more.

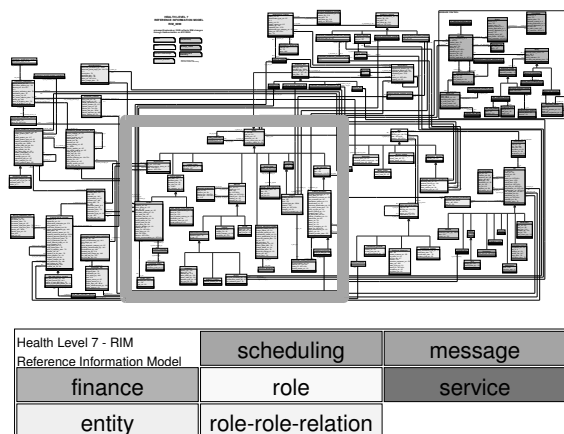


Figure 7.12: HL7 Reference Information Model Framework [150]

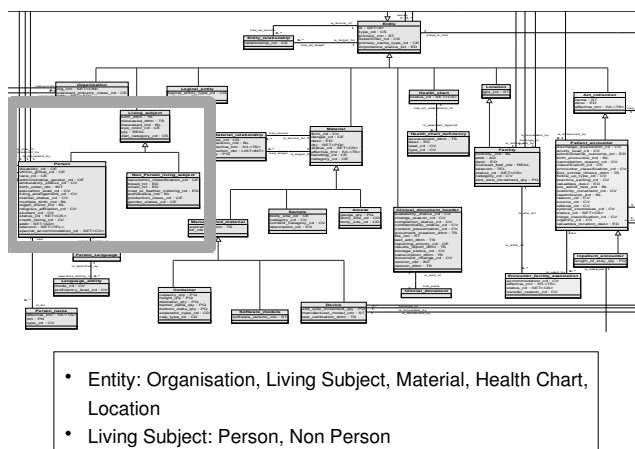


Figure 7.13: RIM Entities [150]

If the RIM got implemented in the Java programming language, all of its classes would inherit from the top-most super class *java.lang.Object*. For reasons of clarity, figure 7.14 omits several intermediary classes and lets *Living Subject* inherit directly from *java.lang.Object*.

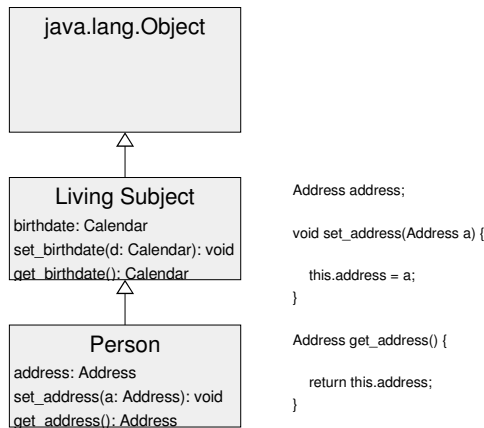


Figure 7.14: Accessing a Person's Attributes

Section 7.2.5 proposed to make the top-most model in a tree of models inheriting from each other a *Container*. In the case of Java that would mean to add a container attribute such as one of type *HashMap*, together with the necessary access methods *set*, *get* and *remove*, to the *java.lang.Object* class (figure 7.15).

That way, every abstract model (OO class) would, by default, become a container able to store a hierarchy of sub models (objects). This would be much closer to what section 7.1 worked out on the importance of the principle of *Composition* in human thinking, which perceives its environment in form of discrete, but composed items. Every item in universe can be seen as *Compound* of yet smaller items. The basis of every modelling effort must therefore be a *Container Structure*.

This would also be a solution to the criticism of chapter 5, meaning that: *the hierarchy as concept is not inherent in the type system of current programming languages*.

Additionally, access methods of classes inheriting from *java.lang.Object* (that is of *all* classes) would become superfluous. Because every class is a sub model of *java.lang.Object*, every class can use not only its container, but also the corresponding *set*, *get*, *remove* methods.

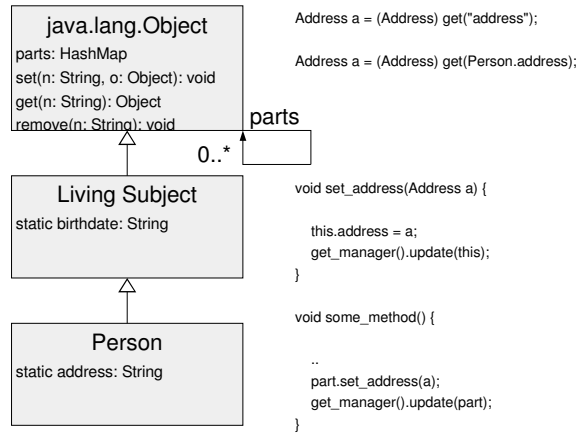


Figure 7.15: Access Method Elimination through Top-Level Container

On its top right-hand corner, figure 7.15 shows the program code that could be used to access an *Address* as a *Person*'s sub model. In order to correctly identify the various attributes of a person, each of them needs to carry a unique name. In the example of figure 7.15, the names are hold as static attributes of the classes they conceptually belong to. But they can as well be stored somewhere else – even in an external configuration file, better called *Knowledge Specification*.

One objection to the elimination of access methods in sub models could be that then, necessary updates cannot be initiated. Traditionally, such updates are often placed in the access methods directly. Figure 7.15 shows how an update manager is called in the *set_address* method, after the *address* attribute has been changed. However, the same update call can be made outside the access method. It would possibly have to be called at several places then, but judging from this work's author's experience with frameworks, the number of update calls won't be too high and is usually well manageable. The questionableness of the OO principle of *Encapsulation* in general was already mentioned in section 4.1.15. Finally, there is actually no access method-related code that cannot be handled alternatively.

7.2.8 Categorisation versus Composition

Since the beginnings of *Object Oriented Programming* (OOP), several of its paradigms were reflected critically and found to cause problems. One of them is the *Fragile Base Class Problem* explained in section 4.1.15.

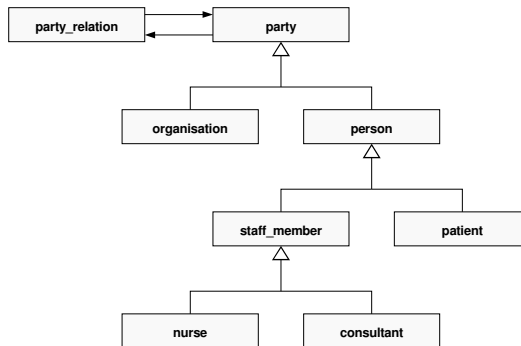


Figure 7.16: Categorisation versus Composition of Parties [18, p. 12]

Further problems may occur through the misuse of inheritance, leading to bad design solutions. A typical example is the modelling of demographic entities like *Patient* or *Nurse* as subtypes of *Person* (figure 7.16), whilst actually, they are *Party Relationships* [18, p. 12], [97].

As can be seen, inheritance can create more problems than are foreseeable. One way to circumvent unwanted dependencies is to sort abstract models according to their placement within a larger model surrounding them. Part models of a model may be grouped by their level of *Granularity*. It should never be only *Properties* leading to the creation of a super category.

A main reason for using inheritance is the reuse of functionality in form of methods. If methods as logic models were kept externally of state models, inheritance as way to reuse methods would not be necessary anymore (section 8).

7.3 Knowledge Representation

Section 7.1 identified the abstraction principles of human thinking, before section 7.2 reflected on their impact on software design. The final considerations of this chapter now deal with a possible architecture for knowledge representation (as first presented in [128]), which applies the principles of human thinking.

7.3.1 Knowledge Ontology

The previous sections tried to demonstrate the importance of *Composition* for knowledge modelling. One technique that was mentioned in this context are *Ontologies*. Section 4.6 introduced some of its numerous definitions. Section 7.2.5 demonstrated how the principle of *Hierarchy* may be applied to obtain an *Ontology*. The layers forming an ontology were called *Ontological Level*.

Basically, an ontology represents a systematic description of complex domain contexts. This work uses its own adapted definition, and considers an ontology to be *a strict hierarchy of abstract models, organised in levels of growing granularity, that are solely unidirectionally related*.

Terminologies as described in section 4.6.5 may be used to specify the basic elements of an ontology. Every term may be represented by an own abstract model (concept) containing a number of strings, one for each terminology system. Further strings may stand for language translations, which has importance for *Internationalisation*.

The following examples may seem simple, but want to strengthen the hierarchical thinking of the reader, under consideration of the granularity of models.

Biological Systems

One example showing the hierarchical structuring of biological systems is mentioned in [285]. Its models are listed in decreasing granularity, in table 7.3.

It is important not to mix ontological layers with parallel layers. In *Geology* or *Biology*, the latter (also called *Stratum*) may be layers of material arranged one on top of another (such as a layer of tissue or cells in an organism) [320]. However, these are not *composed* of each

Biological System
Ecosystem
Biocoenosis (Living Community)
Multiple Cell Organism
Single Cell Organism (Protozoa)
Organelle (Mitochondrie, Chloroplast)
Supra Molecular Complex (Ribosome, Chromosome, Membrane)
Small Molecule

Table 7.3: Hierarchical Structuring of Biological Systems

other. Ontological layers, on the other hand, have a different level of granularity, each so that higher-level abstractions are composed of lower-level abstractions.

Logical Book

The logical structure of a *Book* shall serve as second example. A *Chapter* may consist of *Paragraphs*. Yet it may become necessary to first subdivide *Chapter* into *Sections* which then consist of *Paragraphs*, as shown in table 7.4.

All ontologies can get extended *up-* or *downwards*, by adding further levels, at any later point in design time. But they can as well get extended by inserting *Intermediate Layers* between two already existing ones. However, additional levels should only get introduced if there really is a need for them.

Model Category
Library
Book
Part
Chapter
Section
Paragraph
Sentence
Word
Character

Table 7.4: Logical Book

In contrast to the division of a *logical* book, a *physical* book may be structured completely

differently, for example into *Binding*, *Cover* and *Pages*. Of course, the contents of an ontology heavily depends on the intended area of application (knowledge domain) of the software to be created.

Interdisciplinary Science

A third, certainly very subjective example tries to sort a number of known *Sciences* into one common system (table 7.5). *Arts*, *Linguistics*, *Mathematics* and *Informatics* have an extra status: They deal with already abstracted knowledge (paintings, music, language, numbers) and can be used as utility by any of the other sciences.

Scientific Subject	Example Model
Astronomy	Celestial Body (Big Bang, Cosmos)
Biology	Living Thing (Human, Animal, Plant, Virus)
Geography	Dead Thing (Air, Fire, Stone, Crystal)
Chemistry	Compounds (Water, DNA)
Physics	Particles (Elementary Particle, Atom, Matter, Energy)
Philosophy / Religion	Dialectic Dualism (Matter/Anti-Matter, +/-, 0/1)

Table 7.5: System of Sciences

The whole effort of finding new ways for representing knowledge, as done in this work, is an *inter-disciplinary* undertaking itself, touching various fields of science. The world (nature) needs to be understood in its basics so that humans are enabled to copy its concepts and put them into artificial models – exactly what *Cybernetics* is all about.

Car Model

A *Computer Aided Design* (CAD)/ *Computer Aided Manufacturing* (CAM) system of a car manufacturer will have a *Car* model like the one shown in table 7.6.

Model Category
Car
Body, Chassis, Engine, Transmission
Door, Axle, Wheel, Cylinder
Window, Suspension, Plunger

Table 7.6: Car Model

The ontology contains multiple categories of models which are composed of each other. An *Engine* consists of a *Cylinder* which consists of a *Plunger* and so on. That is why people speak of different *Layers* or *Levels* of abstract models. An *Axle* belongs to one level and a *Chassis* belongs to another, higher level. In a good ontology, the relations between models are always *unidirectional*, that is a chassis can link to axles but not the other way.

Macrocosm and Microcosm

Table 7.7 lists *Astronomical Particles* [85, 9]. It ends with the *Universe* and an undefinable *Macrocosm*.

Category	Example Model
Macrocosm	(Infinity)
Universe	Our Universe with its Laws of Nature
Heap of Galaxies	Local Group, Heap of Virgo
Galaxy	Milky Way (our), Andromeda, Magellan's Clouds
Planetary (Solar) System	Sol (Sun), 51 Pegasi
Star/ Planet	Beta Pictoris, Mercury, Venus, Earth, Mars

Table 7.7: Astronomical Particles

When trying to abstract things (in software), there has to be some limit, a kind of *Top Level Model*. It represents the *Concept* to be described. For a medical information system, one such top level model will be the *Electronic Health Record* (EHR); for an insurance application, it will be the *Electronic Insurance Record* (EIR); and so on.

Models do not only have to be limited *upwards*; the same holds true for modelling towards *Microcosm*. Table 7.8 organises particles, as used by natural sciences, into several categories.

Category	Example Model
Physical Compound	Air, Water, Fire, Ground
Chemical Compound/ Molecule	H ₂ , O ₂ , O ₃ , H ₂ O
Crystal	C (Diamond)
Atom (Chemical Element)	H, He, O
Elementary Particle	Quark, Lepton (Electron, Neutrino)
Urelement	(Primary Particle)
Microcosm	(Infinity)

Table 7.8: Physical Particles

Although the real world seems to be built like that (infinite, nobody knowing what comes beneath the *Quark* particles) – in software modelling it makes no sense (and is actually impossible) to neverendingly introduce lower and lower levels, towards *Microcosm*. On some point, the hierarchy has to be stopped, to be able to abstract it in software. The later chapter 8 gives an overview of common knowledge primitives.

7.3.2 Schema

A theoretical *Model* is an abstract clip of the real world, and exists in the human mind. Another common word for *Model* is *Concept*. It is the subsumption of *Item*, *Category* and *Compound*, resulting from three activities of abstraction: *Discrimination*, *Categorisation* and *Composition* (section 7.1.3). As such, each model *knows* about the parts it consists of.

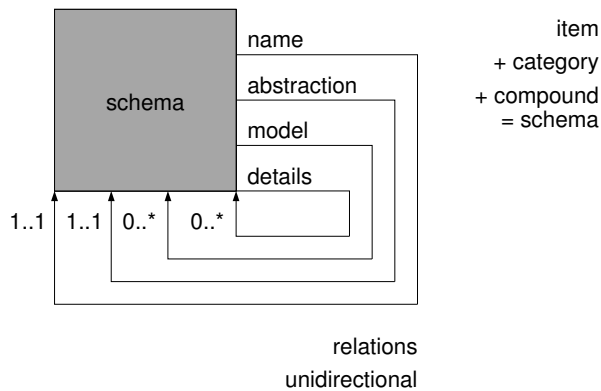


Figure 7.17: Knowledge Schema with Meta Information about Parts

Yet what does this knowledge of a compound model (whole) about its parts imply? Software developers call knowledge *about* something *Meta Information*. Figure 7.17 shows the four essential kinds of meta information in a whole-part relation. Software developers might want to call the illustration of these relations a *Schema* or *Meta Model*.

An obvious way is to give each part a unique *Name* for identification. The concept of a human body, for example, would have parts like heart, brain, left_arm and so on.

Secondly, a compound needs to know about the *Model* of each part since a part may itself be seen as compound that needs to know about its parts. Although all real world items can be modelled as compound, it does not make sense to do so in the virtual world of the human mind. As mentioned before, models have to be limited in their information contents, towards microcosm as well as towards macrocosm, in order to be comprehensible by the human mind. It is therefore necessary to introduce primitive models like a word or a number (compare *Quality and Quantity*, section 7.1.7), representing the final form of abstraction in a compound.

The distinction of the several kinds of models, in other words the kind of *Abstraction* (compound, term, number etc.) of a model is the third kind of information a compound needs to know about its parts. It is comparable to a *Type* in classical system programming languages (section 4.1.7).

All further kinds of meta information are summed up by a fourth relation which is called *Details* in this work. Just like the *Model*, it is a dynamically extensible structure. It will be explained in the following section.

The suggested knowledge model uses a simple *Tree* structure, capable of referencing parts of arbitrary type. It does not follow the *Composite* software pattern (section 4.2.2), because the meta information whether a part model is a compound (composite) or not (leaf) does not belong into the model structure. Section 7.2.8 explained this design mistake on the example of *Party* types. It is not good to fix some model as leaf, at design time. Who knows if at runtime (during program execution), that model would not have to have any parts? As an aside: A similar design (simple tree structure) is used by the *Java Swing* framework [112], for example. Its tree node class *DefaultMutableTreeNode* represents a *Tree Node* and *Tree Container*, at the same time.

7.3.3 Double Hierarchy

Finally, what makes up the character of a model (in the understanding of the human mind) is a combination of two hierarchies: the *Parts* it consists of, together with *Meta Information* about it.

Most properties of a molecule in *Chemistry*, for example, are determined by the number and arrangement of its atoms. *Hydrogen* (H_2) becomes *Water* (H_2O) (with a totally different character) when just one *Oxygen* (O) atom is added per hydrogen molecule. The Wikipedia

Encyclopedia [60] cites and writes about Richard Levins and Richard Lewontin who, in their book *The Dialectical Biologist* [194], sketch a *dialectical* approach to biology:

They focus on the (dialectical) relationship between the *Whole* (or *Totality*) and the *Parts*: *Part makes Whole, and Whole makes Part* [194, p. 272]. That is, a biological system of some kind consists of a collection of heterogeneous parts. All of these contribute to the character of the whole, as in reductionist thinking. On the other hand, the whole has an existence independent of the parts and feeds back to affect and determine the nature of the parts. This back-and-forth (dialectic) of causation implies a dynamic process. . . . Further, each species is part of the *Environment* of all of the others.

The kinds of meta information discussed in the previous sections were also called *Dimensions* or *Conceptual Interaction* between a *Whole* and its *Parts*. They may represent very different properties and each of them may be constrained to certain values- or areas of validity.

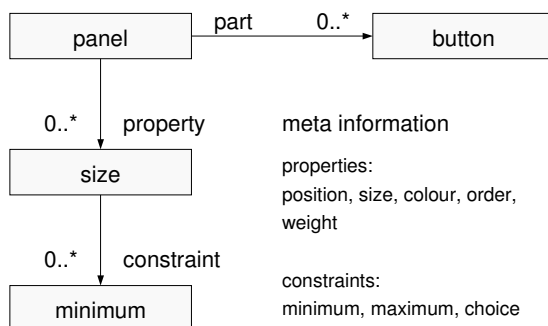


Figure 7.18: Double Hierarchy of Parts and Meta Information

Figure 7.18 illustrates the *Double Hierarchy* here spoken of. A graphical panel was chosen as example model. It may consist of smaller parts, among them being a number of buttons. Altogether, they form the *Part Hierarchy*. On the other hand, there are properties like the size, position or colour of the buttons, which are neither part of the panel, nor of the buttons themselves; they are information *about* the buttons and form an own *Meta*

Hierarchy. To the latter do also belong constraints like the minimum size of a button or a possible choice of colours for it. Constraints can be treated like meta information about properties. Once again: *Properties* are information about a *Part*; *Constraints* are information about a *Property*.

7.3.4 Modelling Example

Another example shall be given to substantiate the need to distinguish between the several kinds of information. How would one describe a *Horse*, unbiased as a child, by doing some brainstorming? Figure 7.19 shows a number of terms commonly used to create a model of a horse. Most importantly, there are structural observations describing the horse as concept consisting of parts like *Head*, *Legs* or *Hoofs*. Secondly, there are properties like the horse's *Colour*, *Shape* or *Size*. Thirdly, there are terms describing a horse's actions like its *Movement* or *Eating*, that change a horse's position and/ or state. Finally, there are a number of terms like *Hay* or *Saddle* associating concepts related to the horse.

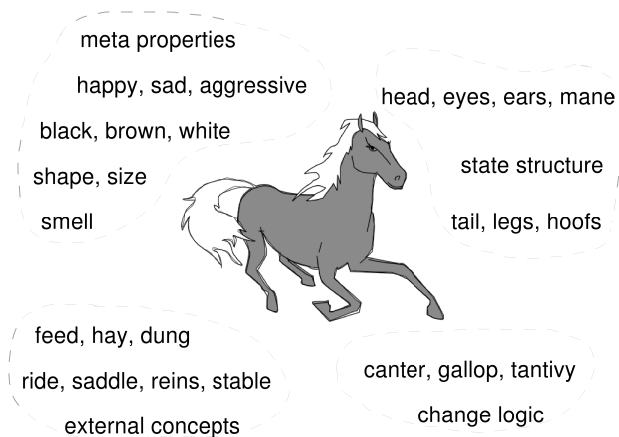


Figure 7.19: Concept of a Horse with Structure, Meta Properties and Logic

One might suggest to model properties like the position, size or colour of a horse's leg as *Part* of that leg. In fact, this is how classical programming approaches its solutions. *Structured- and Procedural Programming* (SPP) (section 4.1.6), for example, would probably use a structure called *struct* or *record* representing the leg and a field standing for the leg's

colour. Similarly, *Object Oriented Programming* (OOP) (section 4.1.15) would use a class representing the leg and an attribute standing for the leg's colour, which, in Java source code, would look as follows:

```
public class Leg {  
    private String knee;  
    private String hoof;  
    private String colour;  
}
```

However, when following the modelling principles of human thinking (section 7.1), this is *not* correct! It is true that in everyday language, one tends to say *A horse leg* has a *colour*. Unfortunately, this leads to the wrong assumption that a leg were made of a colour. But this is not the case. A leg does not *consist* of a colour in the hierarchical meaning of a whole consisting of parts. The colour is rather property information *about* the leg. It seems there is no correct expression in natural (English) language stating the property of something. The *IS-A* verbalisation is used to express that the leg belongs to a special category of items, for example: *A leg is a body element*. The *HAS-A* formulation is used to express that a leg as whole consists of smaller parts, for example: *A leg has a knee and it has a hoof*. But which formulation expresses a property? Well, perhaps it would be best to say: *A leg IS-OF a colour*.

It seems that scientists (including the author of this work) and adults in general have unlearned to think simple like children. Scientists sometimes tend to unnecessarily complicate things that can be described quite easy. Other times, they simplify things which better be distinguished. And looking back into the history of programming, one wonders who ever had this idea of mixing structural elements, properties with meta information and logic algorithms into just one structural entity as at least SPP (record, struct) and OOP (class) do.

The CYBOP knowledge schema introduced before takes care of these things and distinguishes whole-part- from meta information. Actions (like the gallop of a horse) causing some change in the model (horse) or its environment are called *Logic* in this work, since they follow certain rules. Chapter 8 will deal with these.

7.3.5 Container Unification

Section 4.1.15 demonstrated how container inheritance, due to polymorphism, may cause unpredictable behaviour leading to *falsified* container contents. The sections of this chapter introduced a knowledge schema which they claimed to be *general*. But that also means that all kinds of containers must be representable by the suggested schema. But why are there so many different kinds of containers? What actually is a container?

It is a concept expressing that some model *contains* some other model(s). Types of containers that were introduced in section 4.1.15 are *Collections* (Array, Vector, Stack, Set, List), *Maps* (Hash Map, Hash Table) and the *Tree*. They all are containers. What differs is just the meta information they store about their elements. A list, for example, holds position information about each of its elements. A map relates the name of an element to its model (1:1). A tree links one model to many others (1:n).

But does the different meta information a container holds about its elements justify the existence of different container models? If a knowledge schema was general enough to represent a container structure on one hand, and to express different kinds of meta information on the other, it might be able to behave like *any* of the known container types.

The schema proposed in this work claims to be this kind of knowledge schema. It has a container structure by default, and can thus hold many parts in a *Tree*-like manner. It holds standard meta information about its parts: their *Name*, *Model*, kind of *Abstraction* and further meta information called *Details* – and is therefore able to link the name of an element to its model, in a *Map*-like manner. To the additional meta information (details) may belong the *Position* of an element within its model, in a *List*-like manner. A *Table* structure can be represented as well, by splitting it into a hierarchical (tree-like) representation, as known from markup languages (section 4.1.12).

Chapter 9 will introduce a language capable of expressing all aspects of the knowledge schema as proposed in this chapter.

7.3.6 Universal Memory Structure

To better explain the differences between traditional- and cybernetics-oriented design models, a further example shall be given. Figure 7.20 illustrates design-time structures in the upper half, and runtime structures in the lower. Using *Structured- and Procedural Programming* (SPP) or *Object Oriented Programming* (OOP), a developer would design a model as

shown on the upper left-hand side in the figure. (The fact that OOP also offers inheritance relations and OOP classes do own methods in addition to attributes, while SPP structures do not, is of minor importance here.) At runtime, exactly that model would be applied to structure instances and their relations accordingly, as shown on the lower left-hand side in the figure.

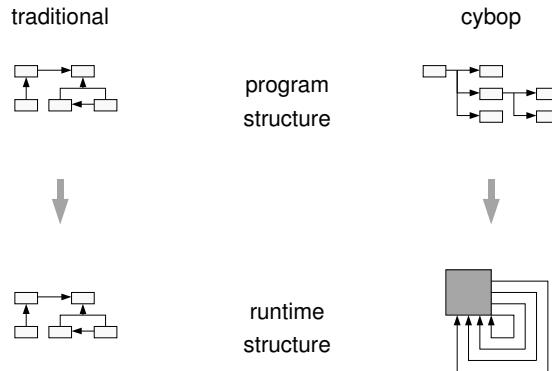


Figure 7.20: Universal Memory Structure

Not so in *Cybernetics Oriented Programming* (CYBOP). Knowledge templates as created at design time do always have a hierarchical structure, as shown on the upper right-hand side in the figure. They include *Whole-Part*- as well as *Meta Hierarchies*. At runtime, these templates get cloned by creating models that follow the structure of the CYBOP *Knowledge Schema*, as shown on the lower right-hand side in the figure. While SPP/ OOP rely on a variety of different structures to store knowledge in memory, CYBOP uses one *Universal Memory Structure* (knowledge schema) that, so to say, merges traditional structures like different kinds of *Containers*, *Class* and *Record/Struct*. Even algorithmic structures (logic) traditionally stored in a *Procedure* are covered by this knowledge schema. More on state and logic in the following chapter.

The advantages are obvious. Data available in a unified structure are easier to process. Dependencies of the knowledge schema are defined clearly and remain the same for all applications, so that domain/ application knowledge becomes independent from the underlying system control software. Global data access and bidirectional dependencies are not neces-

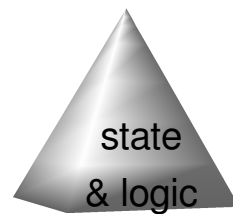
sary anymore, since every knowledge model can be accessed along well-defined paths within the knowledge hierarchy. Byte code manipulation and similar tricks and workarounds might finally belong to the past.

8 State and Logic

*Time comes from a non-existent Future,
into a non-durable Presence,
and goes into a Past that has ceased to exist.*

AURELIUS AUGUSTINUS

The previous two chapters justified a split of static knowledge from its dynamic processing in a system, as well as the modelling of knowledge in a double-hierarchy. This chapter, being the last of part II of this work, elaborates on why it is important to distinguish two different kinds of static models: *State-* and *Logic Knowledge*.



8.1 A Changing World

To start with, this section looks into nature and several sciences once more, to show how both kinds of knowledge (states and logic) appear in them.

8.1.1 Change follows Rules

What makes our world so interesting is not (only) its static structure, but its permanent *Change*. Already the ancient Greeks realised that *Motion* (*Flux*/ *Activity*/ *Change*) is central to existence and reality [123]. Change happens according to *Rules* following a *Logic*. John F. Sowa [294, p. 132] cites Immanuel Kant who writes about the omnipresence of rules:

Everything in nature, in the *inanimate* as well as the *animate* world, happens according to *Rules*, although we do not always know these rules. Water falls according to the laws of gravity, and the locomotion of animals also takes place according to rules. The fish in the water, the bird in the air move according to rules. All nature actually is nothing but a *Nexus* (inter-connection) of appearances according to rules; and there is nothing at all without rules. When we believe that we have come across an absence of rules, we can only say that the rules are unknown to us.

Some people may believe that there are *rule-less* things in universe, for example kinds of *Randomness* or *Chaos* (with the *Entropy* used as a measure of the disorder present in a system [60]). Stephen Wolfram, however, demonstrated that everything in existence shows at least one of the four following kinds of behaviour: *Repetition*, *Nesting*, *Randomness*, *Localised Structures* (*Universality*) (section 7.1.1). And, he reproduced these using simple rules encoded in abstract computer programs.

8.1.2 From Philosophy to Mathematics

Rules belong to a logic. Yet what is *Logic* and how best to describe it and the data it processes?

Syllogism

Many descriptions exist for the term *Logic*. *Webster's Revised Unabridged Dictionary* [212] defines it as:

The science or art of exact reasoning, or of pure and formal thought, or of the laws according to which the processes of pure thinking should be conducted; the science of the formation and application of general notions; the science of generalization, judgment, classification, reasoning, and systematic arrangement; correct reasoning.

The *WordNet Dictionary* [320] calls it: *A system of reasoning*. A third definition given by the *Free On-line Dictionary of Computing* (FOLDOC) [143] shall be mentioned: *Logic is a branch of philosophy and mathematics that deals with the formal principles, methods and criteria of validity of inference, reasoning and knowledge*. *The Devil's Dictionary* [27]

means that the basic of logic were the *Syllogism*, consisting of three propositions: a *major* and a *minor Premise* (assumption) and a *Conclusion*. An example:

- *Major Premise*: Sixty men can do a piece of work sixty times as quickly as one man.
- *Minor Premise*: One man can dig a post-hole in sixty seconds.
- *Conclusion*: Sixty men can dig a post-hole in one second.

The sense or nonsense (validity) of the results of reasoning is another issue. *Syllogism* means in short: *to conclude by deductive reasoning; to reckon all together; to bring at once before the mind; to infer* [212]. What is important to note here is that logic describes the laws after which one state (major and minor premise) is related to another state (conclusion). It associates two statements and defines the rules for deriving/ translating one from/ into the other.

As in all sciences, there is unsolved problems to *Logic*, like the *Aristotelian Problem of First Principles*. Kelley L. Ross [277] writes:

Logic is just the description of how (proposition) X implies (is a reason for) (proposition) Y and (proposition) Z, or that Y and Z are logical consequences of X. Logic can prove Y and Z on the basis of X, but it cannot prove X without further reasons (premises) ... If we continue to give reasons for reasons, from Z to Y, to X, to ..., this is called the *Regress of Reasons*. Aristotle's second point, then, was just that the regress of reasons cannot be an infinite regress. If there is no end to our reasons for reasons, then nothing would ever be proven. We would just get tired of giving reasons, with nothing established any more securely than when we started. If there is to be no infinite regress, Aristotle realized, there must be propositions that do not need, for whatever reason, to be proven. Such propositions he called the first principles (archai, principii) of demonstration. How we would know first principles to be true, how we can verify them, if they cannot be proven is the *Problem of First Principles*.

This work does not attempt to further consider or even solve logical- philosophical problems of that kind. Instead, it sticks with informatics which deals with processing given states according to well-defined rules of logic and focuses on their mathematical side, namely binary arithmetic and boolean logic, as described following.

Binary Arithmetic

One of the many great achievements of Gottfried Wilhelm Leibnitz (1646-1716) [224] was his development of the mathematical *Binary System of Arithmetic* (1679) [277]. Unlike the traditional number system that is based on the digits 0..9, the binary system uses only two digits: 0 and 1. Yet it is possible to express any number as sequence of Bits (section 4.1.3), called a *Binary*.

Many abstractions simplifying the real world are based on just *two* views (as investigated by the philosophical field of *Dialectic Dualism*), for example:

- Plus Infinity & Minus Infinity (Mathematics)
- Positive & Negative (Physics)
- Matter & Antimatter (Physics)
- Force & Counterforce (Physics)
- Masculine & Feminine (Biology)
- Active Neuron & Passive Neuron (Neurology)
- Black & White (Psychology)

The binary system is now the basis of all digital technology. It enabled scientists to construct simple electrical circuits and to combine them to greater, more complicated ones and, finally, complex chips which are used in every computer.

Boolean Logic

Almost twohundred years after Leibnitz completed his binary arithmetic, George Boole (1815-1864) took on those ideas and formed his *Boolean Algebra*, described in *An Investigation into the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities* (1854). The St. Andrew's website [224] states:

Boole approached logic in a new way reducing it to a simple algebra, incorporating logic into mathematics. He pointed out the analogy between algebraic symbols and those that represent logical forms. It began the algebra of logic called Boolean algebra which now finds application in computer construction, switching circuits etc.

The same site describes Boole's theory as: *a system of symbolic logic* and: *an algebra in which the binary operations are chosen to model the union and intersection operations in Set Theory. For any set A, the subsets of A form a Boolean algebra under the operations of union, intersection and complement.*

Boolean postulates and laws [24] are based on three operations: *AND*, *OR* and *NOT*. Every Boolean algebra can be built up by combining simple Boolean algebra, with its elements 0 and 1. These definitions make it possible to express and translate complex knowledge. Sowa [294] writes: *Yet logic is all there is: every programming language, specification language and requirements definition language can be defined in logic; and nothing less can meet the requirements for a complete definition system.*

8.1.3 System

Earth is a system, a biotope and its biological creatures, including human beings, are systems, our society, institutions, families and their individuals are systems, machines and computers are systems – and software applications are systems. Actually, almost everything in existence can be treated and simulated as system.

A rather general definition [60] describes a system as: *an assemblage of inter-related elements comprising a unified whole.* Systems are in a steady exchange with their environment. Information systems, for example, exchange data. Depending on their structure, relations and contents, these may be called *Knowledge*. From the view of a system, the data are called *Input* and *Output*. The output of one system can become the input for another.

A certain logic with special rules can be abstracted in form of a *System*. The system's logic causes its characteristic *Behaviour*, that is the way its *Input* gets manipulated to produce a specific *Output*.

Deterministic- and Stochastic Behaviour

Systems can be distinguished by their behaviour, which can be *deterministic* or *stochastic* (probabilistic). While the elements of the first work in a predictable way, probabilistic systems are not fully transparent and their results are only *likely*, but never *certain*. Living systems are entirely probabilistic, because firstly, not all of their elements are known and secondly, they always consist of sub systems on different functional levels. [285]

Two areas dealing with the simulation of stochastic behaviour are *Fuzzy Logic* and *Artificial Neural Networks* (ANN). Most software systems though, need *reliable* (deterministic) behaviour delivering *predictable* results. Deterministic systems are therefore in the focus of the research done in this work.

Black Box

An *Operation* can be well treated as system: it contains rules of logic after which its input gets transformed into its output. But not all systems are as easy as a simple operation; many are *composed* of yet smaller systems. Biological systems, for example, are extremely difficult to describe in their entirety, with a simple mathematical formula.

A system may be seen as a number of interacting *Functional Elements*. It is these elements and their interactions which determine the specific properties and behaviour of a system. However, for modelling the behaviour of a transmission system, its inner structure is not important. Systems theory focusses on the time-dependent progression of input- and output signals as well as their relation.

A common technique in systems engineering is to reduce complexity by *hiding* functionality which is unimportant in the given context, inside a system. One then talks of a system as *Black Box* since only its input, output and their relation, but not its inside, are considered (figure 8.1). The black box provides an encapsulation towards the infinite microcosm, and it knows nothing about its usage within a greater macrocosm (section 7.3.1).

The usual way to illustrate system elements and their relations is the *Block Diagram*. It is an important instrument for system analysis. Many structures and processes can be described in that manner. In software engineering, the *Unified Modeling Language* (UML) has become the de-facto modelling standard instead.

Open- and Closed Loop

The scientific discipline of *Automation- and Control Engineering* knows two kinds of control systems: *Open Loop* and *Closed Loop* (*Feedback*). German language better confines both by defining the terms *Steuerung* (steering) and *Regelung* (controlling). Figure 8.1 illustrates a closed loop system, received by adding a feedback loop to the black box mentioned before.

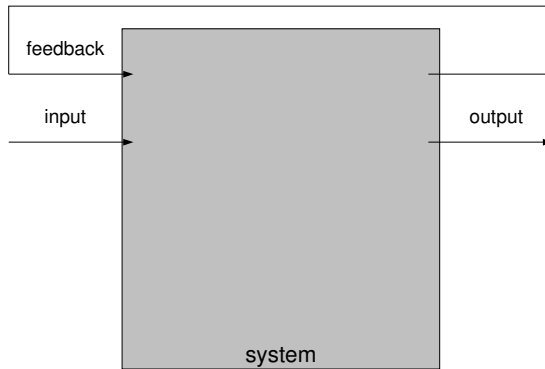


Figure 8.1: Closed Loop System with Feedback, modelled as Black Box

A device controlling the behaviour of a system is called a *Controller*. Automation engineering uses electronic components such as *Capacitor* and *Coil* to build controllers providing *linear* (proportional), *differential* or *integral* behaviour. In software engineering, things are simpler. A computer program containing special mathematical equations can simulate and control a system.

A software system's internal signal processing loop reads signals one by one, from a signal memory (section 6.3). While processing them, new signals may get created and placed in the signal memory. This is how output results may be fed back to become a new input to the software system.

Input/Output and Rules

In order to process data correctly, a system needs to know about their *Structure*. Software systems work with data belonging to some knowledge model. Chapter 7 demonstrated how knowledge can be modelled. Many applications keep their knowledge in special data files. Others, such as *Enterprise Resource Planning* (ERP) systems, retrieve their data from a database. Even systems claiming to do nothing than pure data processing, possibly using one operation only, rely on simple knowledge models, for at least their *Input/Output* (i/o) data.

Living systems rely on constantly exchanging information, energy, nutrients and excretion products with their environment; they are never in a stationary, but always in a *Steady State*. A biological cell, for example, has inputs and outputs and reacts in a certain manner which, after [285], is best modelled with a *Converter* containing *Rules*, and treated as black box. The cell's characteristic behaviour results from the way it relates inputs to outputs.

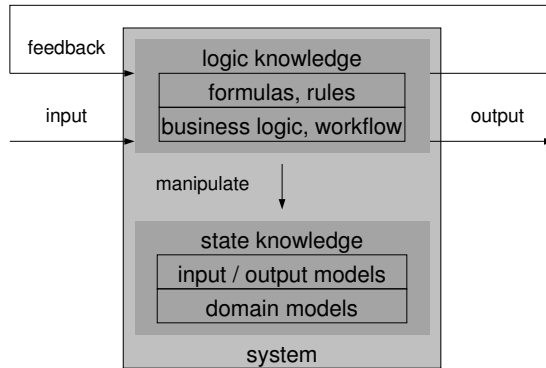


Figure 8.2: Logic translates between Input-, Domain- and Output States

Whilst figure 8.1 illustrated the i/o flow of a system from an *outside* view, figure 8.2 also considers the state- and logic knowledge situated *inside* a system. The arrow indicating the information flow is directed from *Logic*- towards *State* models, because the former manipulate the latter.

8.1.4 Self Awareness

One of the particular characteristics of human beings is their ability for *Self Awareness*. It contrasts the human- with an animal mind because it permits humans not only to understand what is going on in their environment, but also themselves and their being in this world. In other words, the *Mind* knows about itself and about its existence in form of the human organ called *Brain*, as its physical carrier and as the place of thinking.

Of course, the mind also knows about further organs and body parts. Some of the concepts stored in it contain the characteristics of a human being. This knowledge of the structure

of the human body is necessary for the mind to be able to steer it. While the functions for living are *passively* controlled by the *vegetative* (unconscious) nerve system, sensoric and motoric (input/ output) organs need to be controlled *actively*, by the *animalic* (conscious) nerve system.

A system also needs to know about its own abilities, in order to be able to communicate. It has to have a concept of its communication organs/ devices, spoken languages etc. This counts for a human- as for a computer system. The *Agent* systems used in *Agent Oriented Programming* (AGOP) (section 4.3.7) know about their *Capabilities*, which are stored together with other knowledge as their *Mental State*.

Human Body

Figure 8.3 shows parts of the animalic nerve system of a human being. *Sensoric* organs are used for information input; *motoric* organs for information output.

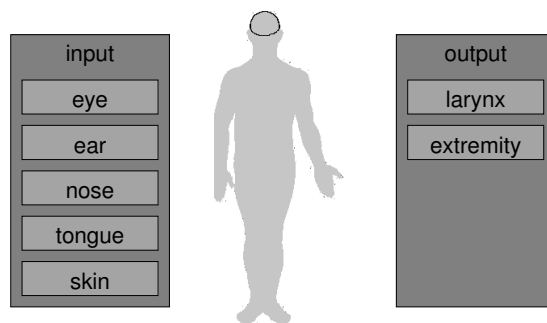


Figure 8.3: Human Body with Sensoric and Motoric Organs

The five (seven) human senses were already shown in table 7.1. They are able to receive signals which are transported by different mediums. The transport mechanisms rely on various physical and chemical *Effects*, as shown in table 8.1. Each sense is represented by an *Organ*. The optic cells of the retina of an *Eye* bundle light stimuli which the optic nerve

forwards as electrical signal to the brain. The inner *Ear* transforms oscillation frequencies of sound-waves into electrical signals to send on to the brain. And so forth.

Effect	Science	Sense
Oscillation, Wave	Physics, Mechanics	Seeing, Hearing
Density, Temperature	Physics, Mechanics, Thermodynamics	Tactile
Aroma	Chemistry	Smelling, Tasting

Table 8.1: Effects as Basis of Sensing

While the *Reception* of information is based on *Nerve Cells*, it is *Muscle Cells* which are responsible for information *Sending*. Humans communicate for example through visual *Gestures* using their *Extremities* or through acoustical *Talking* using their *Larynx/ Vocal Chords*. The latter, too, is based on muscle activity.

Computer Hardware

Gilbert Carl Herschberger II writes [132]: *A computer is a grossly oversimplified model of a human being. Humans can learn more about themselves by working with this model. And, they might learn more about what makes a good model by looking at themselves.* To the many analogies a computer has with a human being belong its input/ output (i/o) devices (figure 8.4), many of which have an equivalent organ in the human body:

- *Eye*: Camera, Scanner
- *Ear*: Microphone
- *Nose, Tongue*: Sensors
- *Skin*: Keyboard, Mouse, Joystick
- *Larynx*: Loudspeaker
- *Extremity*: Monitor, Printer, Braille Panel, Arm, Wheel

The difference to *Robotics* is that a robot has additional devices. Various *Sensors* are used for information input; many moveable parts like *Arms* or *Wheels* take motoric action and can be compared to the extremities of the human body. Table 8.5 gives an impression of how technical and biological environments can be compared.

To sum up: Among the abstract knowledge models stored in a system are some that describe the structure and capabilities of the system itself.

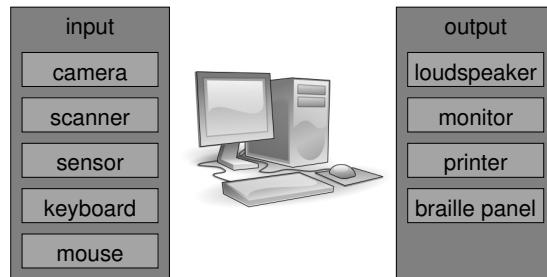


Figure 8.4: Computer Hardware with Input- and Output Devices

Ontological Level	Technical System	Biological Equivalent
Network	Internet, Wide Area Network (WAN)	Society, Biotope
Family	Local Area Network (LAN)	Family
System	Computer	Organism
Block	Component	Organ

Figure 8.5: Ontology comparing Technical- and Biological Environment

8.1.5 Communication

As described in the previous sections, humans have sensoric and motoric organs responsible for information input and output. In between input and output, the information is processed by the brain that contains a specific abstract model of the surrounding real world. The human brain consists of several regions (section 6.1.2), each being responsible for a special task, such as the optical region for seeing or the cerebral cortex for actual information processing which possibly leads to awareness. Depending on which medium, organ and language is used, systems may communicate across different channels.

Transient

Valentin Turchin [317] writes:

Sensations are produced by our sense organs. Perceptions are formed and used within the brain. Conceptions are created by ourselves while we create new, linguistic, models of the world. The triad: Sensation, Perception, Conception seems close in meaning to Kant's usage of these terms. We leave it to the reader, though, to judge on it.

The following example demonstrates a typical information processing procedure. Its sequential flow is illustrated in figure 8.6 which uses technical names, instead of biological ones. These can be found again in the explaining text, enclosed in parentheses. The terms *Assembler* and *Mapper* are converted and merged into the term *Translator*.

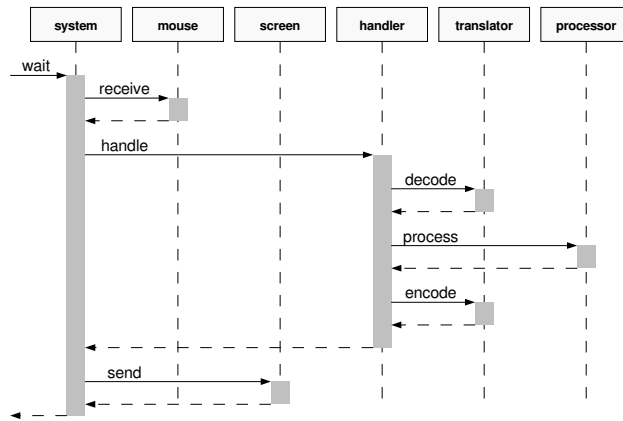


Figure 8.6: Signal Processing as UML Sequence Diagram

One human being (*System*) wants to send a message to another. It decides for an acoustical message (*Signal*), formulates a sentence and talks. The other human being, waiting for signals (*wait*), receives the message across its ear organ (*Microphone, Keyboard, Mouse*). The message is then forwarded to the receiver's brain (*Handler*), where a special region responsible for acoustics (*Translator*) translates (*decode*) the data (*Data Transfer Model*) contained in the message and sorts them into the human's abstract model of the surrounding real world (*Domain Model*). Processing of the message happens in the cerebral cortex of the brain (*Processor*). If the addressed listener wants to send an answer message (*Signal*), it may do so by triggering a muscle reaction. For this to happen, the motoric brain region (*Translator*) needs to translate (*encode*) model data (*Domain Model*) into a special transfer

model (*User Interface Model*), for the answer. Finally, the answer message (*Signal*) will be sent as muscle action (data display on *Monitor*).

If a communication partner does not, or only partly receives a message, the missing information is lost, unless the sender repeats it once more. The reason is that the transport mediums (light, air) do not steadily contain the information; the sent information is transient. Therefore, the whole process described above can be called *Transient Communication*.

Persistent

One great advantage of human beings is to be able to help each other, to cooperate in order to reach a common aim, to form a society which is to fill exactly these aims. Main tasks of a *State* as one form of organisation of society are: *Security, Education, Social Welfare*. While all of them depend upon *Politics*, there is an additional factor playing an increasingly important role: the *Availability of Knowledge*. Knowledge cannot only be exchanged between current citizens of earth; fortunately, it can be forwarded over *Generations*.

For this to become possible, mankind had to make use of different mediums for external storage, such as: *Rock Painting, Stone Tablet, Papyrus Roll, Paper Book, Chemical Film, Electronic File*. It also had to invent technologies for the dissemination of knowledge: *Monks' copying by hand, Library, Printing, Radio and TV, Internet*. The following example does therefore not deal with *direct* inter-systems communication, but rather its *indirect* counterpart – the interaction between a system and mediums in its environment. Of course, that environment could be treated as system, too; but for reasons of simplification it is not here.

One human being (*System*) wants to send a message to another, which is not near the same location, but at some remote place. The sender has to decide for a persistent message, and to choose a non-transient medium to store that message. He takes a piece of paper, writes down or paints some information and finally sends that paper as letter by (snail) mail. Paper and letter act as *Knowledge Carriers*. The receiver may then perceive the message optically and process it similarly to the transient communication explained in the previous section.

Because the information in this example is permanently available and reproducible from the external medium, communication processes of that kind may be called *Persistent Communication*. In computing, persistent information can be stored in files on a *Hard Disk Drive* (HDD), for example; data in *Random Access Memory* (RAM) or those sent over a network, on the other hand, are transient.

The interpreter system introduced in chapter 10 is capable of using transient- as well as persistent communication.

Models

Section 4.3.7 described a (software) *Agent* as *social* system communicating with other agents, including human beings [294, p. 330]. Various communication process models were contributed by the *Social Sciences*. Buesch [39] describes a *Mathematical Communication Model* by Shannon & Weaver (figure 8.7) that shows very well the presence of two translators: *Encoder* and *Decoder*.

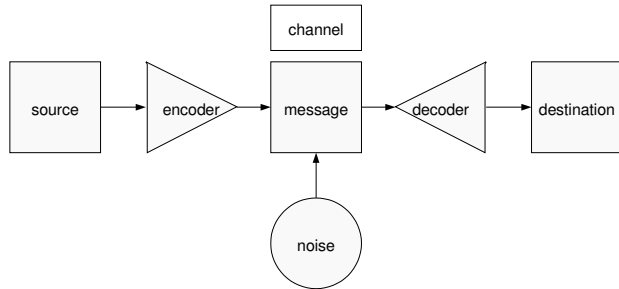


Figure 8.7: Mathematical Communication Model by Shannon & Weaver [286]

The *Conversation Model* of Osgood & Schramm (figure 8.8) extends the communication model to a circular process of *Question* and *Answer*, of *Sending* and *Receiving*. It shows more clearly, that *every* system, in order to communicate both ways, needs to own an *encoding* as well as a *decoding* translator. The interpreter system introduced in chapter 10 contains both kinds of translators.

The *Contents of Communication* is described by the *Lasswell Formula* (figure 8.9). After it, communication consists of the five elements: *Sender* (Who) and *Receiver* (Whom), *Message* (What), *Language* (Channel) and *Result* (Effect). The first four of these will be considered

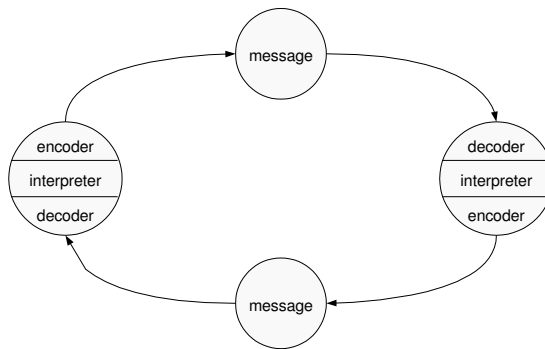


Figure 8.8: Conversation Model by Osgood & Schramm [240]

in the specification of the knowledge modelling language introduced in chapter 9. The effects a communication has (fifth element) are not a prerequisite for that communication to happen and thus not interesting in the context of its technical realisation, as investigated later in this work.

8.2 Translator Architecture

Section 8.1 emphasised the different roles of state- and logic knowledge within systems and communication processes. This section investigates how classical software system design handles both kinds of knowledge models.

8.2.1 Interacting Systems

Chapter 3 introduced an example *Information Technology* (IT) environment (*Physical Architecture*), containing many interacting systems: server and client, local and remote, human and artificial (figure 8.10). In (object oriented) software design, special patterns are used to architect a system such that it is able to communicate with other systems across various mechanisms (*Logical Architecture*). To these patterns count the *Data Mapper*, *Data*

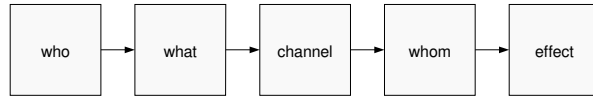


Figure 8.9: Contents of Communication (Lasswell Formula) [189]

Transfer Object (DTO) and *Model View Controller* (MVC) (section 4.2).

Although software development has become a lot easier in the last decades, it is still a big effort that should not be underestimated. One thing that application developers have to care about much of their time is the *Conversion* between various kinds of (communication) models that a system has:

- Frontend (Communication with Human User)
- Backend (Communication with Data Source)
- Remote (Communication with Server)
- Domain (Communication with own Knowledge)

The different mechanisms and patterns that have to be considered for such model conversion often need to be implemented repeatedly, for each new application. Some trials to unify all backend communication in a common *Persistence Layer* exist [5], but are remote- and frontend communication seldom considered in a comparable way. Obviously, no current effort treats the frontend as just another communication model that has to be *sent* to the human user as just another system.

The following sections will first reconsider three common communication patterns, before embedding them into the classical model of logical system layers (section 4.2.1). After

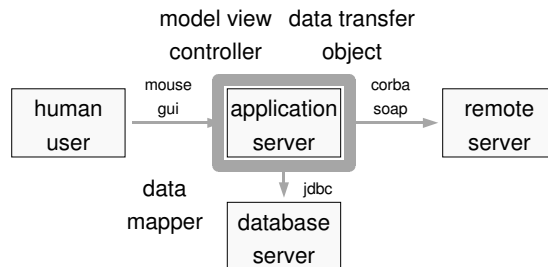


Figure 8.10: IT Environment with Server using Communication Patterns

that, a simplification is suggested which finally leads to a new *Translator Architecture* (first introduced in [127]).

8.2.2 Basic Patterns

Of the patterns described in section 4.2, some are of particular interest for communication. They are explained in Martin Fowler’s pattern collection called *Enterprise Application Architecture* (EAA) [101], and reinvestigated for commonalities here.

Data Mapper Reflection

The most important idea of the *Data Mapper* pattern is to abolish the interdependency between domain model and data source (persistence medium). All information about where a data source like a *File* or *Database* (DB) is located, how to talk to it (File Stream, JDBC with SQL etc.) and how to map *Domain-* to *Entity Relationship Model* (ERM) data is moved away from the domain, into the data mapper layer.

This separation contributes to a clear architecture; it is not enough, though. The data mapper layer often concentrates not only *Mapping-*, but also *Communication* functionality.

Database Management Systems (DBMS) such as PostgreSQL [122, 264, 265] or MySQL [214] are often treated different than normal servers. Frequently, they are assigned a logical *Data Source* layer (figure 4.1). But in fact, DBMS are *Systems*, as their name says, and as such need to be addressed using special communication mechanisms (like JDBC or ODBC).

It therefore seems useful to extract all communication functionality from the data mapper, and put it directly into the system control layer. Chapter 6 explained why it is favourable to have application knowledge separated from system control mechanisms. While persistence/ communication mechanisms as such do not contain any domain knowledge, mapper (translator) modules do. The remaining data mapper layer would hence contain application-related logic knowledge, for translating data from the domain model to the corresponding persistence model and vice-versa.

The *Cybernetics Oriented Interpreter* (CYBOI) that will be introduced in chapter 10 is a system able to handle local- as well as remote communication mechanisms. Applications written in the *Cybernetics Oriented Language* (CYBOL), introduced in chapter 9, will have to deliver the necessary logic for model translation, but application developers are freed from implementing the same low-level system communication functionality (like sockets) again and again, leading to clearer code with greatly reduced size. CYBOL application developers are offered a number of communication mechanisms to choose from.

Data Transfer Object Reflection

The *Data Transfer Object* (DTO) pattern proposes to bundle domain data before sending/ receiving them among systems. An *Assembler* packs/ unpacks needed domain data into/ from a flat data structure called DTO.

Comparing with the data mapper described before, the assembler's task of translating between data models seems quite similar. Wouldn't it be possible, hence, to use *Translator* models (logic knowledge) similar to those suggested for persistence, also for inter-system communication? Different types of translator models could be provided for different communication protocols. Again, communication mechanisms would be put into the *System Control* layer, and translator logic into application-related *Knowledge* models.

Model View Controller Reflection

After having had a closer look at two common software patterns for persistence and communication, this section finally considers the so-called *Frontend* of an application, which is often realised in form of a *Graphical User Interface* (GUI). Nowadays, the well-known *Model View Controller* (MVC) pattern is used by a majority of standard applications. Its principle is to have the *Model* holding domain data, the *View* accessing and displaying these data and the *Controller* providing the workflow of the application by handling any signals (events/ actions) appearing on the view.

Since the view serves as means of communication between a software system and its user (*Human Being* as system), it is in fact just another kind of communication model that should be assembled by a special *Translator*. Because there are many ways in which domain data can be displayed, different types of user interfaces exist (textual, graphical, web, vocal, Braille). Each of them has to have its very own translator that knows how to map data both ways, from the *Domain* model to the *User Interface* (UI) model and vice-versa.

Signalling and related mechanisms, as well as hardware-driving functionality such as graphics adapter access belong into *System Control* modules; UI translators, on the other hand, are application-specific models containing *Logic Knowledge*.

8.2.3 Placement

Many state-of-the-art software systems consist of a layered architecture similar to the one shown in figure 4.1. Yet how do the communication patterns explained before suit this classical architecture? In the traditional model of a layered software system, a startable process, best placed in the *Controller*, creates the whole application tree, to which belong the *Views* (as user interface), several *Models* of the *Domain* (providing data to the views and as facade to remote servers) and the *Data Mapper* (translating between domain- and database model).

It is not difficult to figure out where the communication patterns of section 8.2.2 fit in here (figure 8.11): The *Model View Controller* (*Presentation Layer*) determines the parts to interact with a human user via the *View*; the *Data Mapper* pattern with its inherent *Entity Relationship Model* (ERM) encapsulates mechanisms to connect to a persistence medium such as a *Database* (DB); the *Data Transfer Object* (DTO) and its corresponding assemblers, finally, serve as means of communication with remote servers.

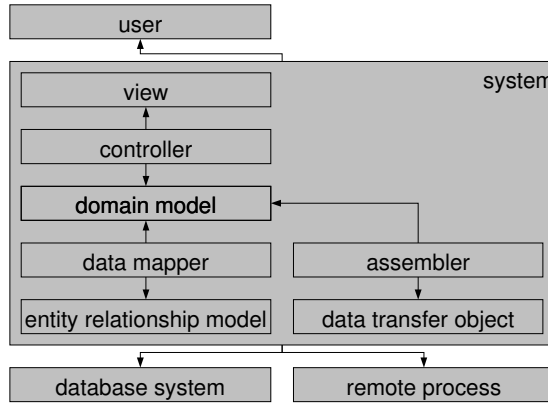


Figure 8.11: Communication Patterns placed in Layered Architecture

8.2.4 Simplification

For all three kinds of communication, there is a:

- *System* (Human User, Database, Remote Server)
- *Model* (View, ERM, DTO)
- *Translator* (Controller/ View Assembler, Data Mapper, DTO Assembler)

All models represent certain states; all translators contain logic for converting one state into another; all systems host their own, specific pool of state- and logic knowledge. Realising this, a much clearer view on software architectures can be retrieved (figure 8.12).

Existing communication patterns can be merged into this common architecture. Although these patterns suggest their very own communication paradigms, the basic principles of interaction, as investigated on the example of transient and persistent communication of humans in section 8.1.5, remain the same:

An active *System* (concrete process) has a mental state represented by passive *Knowledge*. In order to exchange information with another system, it translates parts of its domain- into a special communication model which it sends to the other system. This is done by accessing its hardware infrastructure with *input/*

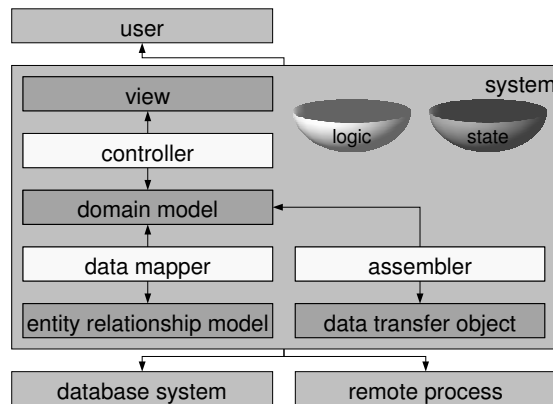


Figure 8.12: Simplified Layered Architecture with State-/ Logic Knowledge

output (i/o) abilities. The other system receives the communication model and translates it back to its own domain model.

Because domain models differ between systems, each system needs its own translator models. Only communication models need to be agreed upon between systems; they need to be understood by both communication partners.

8.2.5 Communication Model

As section 8.1.5 pointed out, systems (alive or not) never communicate directly, but always across the detour of an external (transient or persistent) *Medium*. This makes it necessary to use special *Communication Models*, since nearly always, only *parts* of a complete *Domain Model* want to be exchanged. The use of communication (transfer) models again, entails the use of model *Translators*. Sowa [294] writes in his book *Knowledge Representation*:

In computer science, there is no end to the number of specialized notations. Besides the hundreds of programming languages, there are diagrams for circuits, flowcharts, parse trees, game trees, Petri nets, PERT charts, neural networks, design languages, and novel notations that are invented whenever two programmers work out ideas at the blackboard. Musical notation ... is an example of

a complex language that is both precise and human factored. As long as the mapping rules are defined, all of these notations can be automatically translated to or from logic.

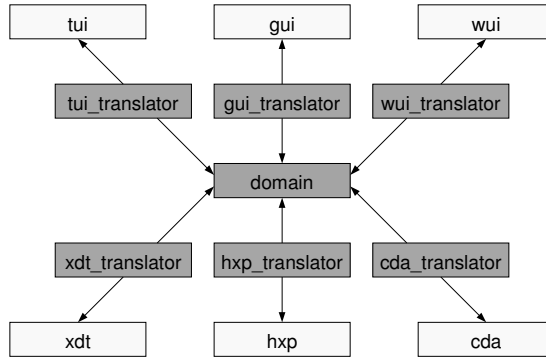


Figure 8.13: Different Kinds of Model Translators

Although he does not talk of *Domain-* and *Communication Models*, but of *Notations*, Sowa obviously means the same: Any kind of abstract model can be translated into any other kind, as long as the translation *Rules* are defined. Model *Translators* are able to map domain model data to transfer model data. Depending on which communication style is used, different translators with different rules need to be applied.

Figure 8.13 shows a number of possible model translators, for a: *Textual User Interface* (TUI), *Graphical User Interface* (GUI) and *Web User Interface* (WUI) as well as for the German standard file format for exchanging medical data called *x Datenträger* (xDT), the *Healthcare Xchange Protocol* (HXP) and HL7's exchange format called *Clinical Document Architecture* (CDA). More on these standards in chapter 11.

Many application systems have exactly one domain model but transfer models of arbitrary type should be addable anytime. Translators only know how to translate between the domain model and a special transfer model, of course in both directions. *Direct* translation between transfer models is an exception; it is possible but better done *via* the domain model.

The type of transfer model is independent from the communication mechanism used. The usage of a *Graphical User Interface* (GUI) model, for example, is not necessarily limited to human-computer interaction. It could very well be used for data transfer between remote computers, as long as both systems know how to translate that model.

8.3 Knowledge Abstraction and -Manipulation

Having shown the existence of state- and logic knowledge, not only in software systems, and having compared both with a classical layered software architecture, what is still missing is an overview of fundamental abstractions of state- as well as logic knowledge, and a solution showing how both can be manipulated, in a knowledge-based system.

8.3.1 Algorithm

For John F. Sowa [294], *Knowledge Engineering* (KE) is: *the application of logic and ontology to the task of building computable models of some domain for some purpose*. Section 7.3 showed how an *Ontology* can be applied to structure state knowledge of a domain, and introduced a new *Knowledge Schema*. This section investigates the universality of that knowledge schema, that is its applicability to *State-* as well as *Logic* models.

Not only input/ output (i/o) knowledge (states) can be structured hierarchically, using an ontology, the operations of a system (logic) can be cascaded and nested as well. The resulting logic models are *Sequences* of input-to-output mapping rules that can consist of yet finer-grained models. The theory of computing uses the word *Algorithm* to label a sequence of mapping rules. Banerjee [15] writes on this:

Each . . . mathematician had to precisely define the notion of an algorithm, and each defined it in a different way. Godel defined algorithm as a *Sequence of Rules* for forming complicated mathematical functions out of simple mathematical functions, Church used a formalism called the *Lambda Calculus*, while Turing used a mathematical object called the *Turing Machine* and defined an algorithm to be any set of instructions for his simple machine. All these seemingly different and independently contrived definitions turned out to be equivalent and they form the basics of the modern theory of computing. No modern programming

language can achieve more, in principle, than the Turing machine or the lambda-calculus.

Time plays an important role in data processing. It dictates the order in which steps of an algorithm are executed and thereby ensures a correct sequence of actions. Every element of an algorithm needs to be assigned an instant (position) in time, as meta information. Although the runtime-processing of data, according to an algorithm, is *dynamic*, the models of logic – just like i/o state models – are *static* (chapter 6).

8.3.2 Operations

In the end, all computer-implemented procedures go back to boolean operations and binary arithmetic (section 8.1.2), processed by digital logic circuits (section 4.1.3). A *Multiplication* can be expressed as sequence of additions. By representing the number to be subtracted in its negative form (*Two's Complement* [250]), a *Subtraction* can be mapped to an addition. An *Addition* itself is performed by linking Bits of the summands logically, using an *AND* operation. Fundamental operations for knowledge translation are:

- *Boolean*: and, or
- *Comparison*: equal, smaller, greater
- *Arithmetic*: add, subtract, multiply, divide

They all imply special rules after which one or more input operands (values) get transformed into one or more output operands. Both kinds representing static *State Models*, input and output can be placed as branches of one common knowledge tree. But also the rules as static *Logic Models* can be added to this tree.

8.3.3 Primitives

All software is based on the final two states called *one* & *zero*, or *on* & *off*, or *true* & *false*, or similarly. A *Binary Digit* (Bit) can take on the values *zero* or *one*; it represents the final abstraction of any software model and can be easily mapped to hardware. A second unit, the *Byte*, consists of eight Bits. One *Word* is made up of two Bytes, one *Double Word* of four Bytes and so forth. Data in a *File* on a *Harddisk Drive* (HDD) partition or another

Primitivum	Size [Byte]
Date, Time, Complex, Fraction, Term	many
Double, Float, Vector, String	8, 12, 16 or more
Integer, Pointer, Word, Short, Byte, Character	1, 2, 4, 7, 8 or more
Bit, Boolean	0 or 1 Bit

Figure 8.14: State Primitives sorted after their Granularity

storage medium are saved in form of Bit sequences, just like data in *Random Access Memory* (RAM). It is up to a program to interpret these data correctly, in the desired manner.

Many programming languages offer a number of basic types, also called *Primitives*, which are combinations of different numbers of Bits. Table 8.14 shows some of them, together with their possible memory usage. Besides the primitive types that are included in a programming language, there are other forms of storing data. Section 7.1.6 said that not only a *String*, but also an *Image* or a *Sound* can represent a *Quality*, that is a *Term* with special meaning. The format of such data sequences is often defined as *Multipurpose Internet Mail Extension* (MIME) type, for example:

- *text*: sgml, xml, html, rtf, tex, txt
- *image*: jpeg, png, gif, bmp
- *audio*: ogg, mp3, wav
- *video*: mpeg, qt, avi
- *application*: kword, sxw

8.3.4 Logic Manipulates State

Knowledge models of the form introduced in chapter 7 are stored as tree structure in *Random Access Memory* (RAM). They are the dynamic result of instantiating static knowledge templates (concepts), and hence changeable. Because knowledge models are passive, they need to be managed by an active system, responsible for their creation and destruction, as described in chapter 6.

As the previous sections of this chapter have shown, there are two kinds of knowledge: *States* and *Logic*. While the former may be placed in a spatial dimension, the latter are processed as sequence over time. Often, logic is labelled *dynamic* behaviour – but only the *execution* of a rule of logic is dynamic, *not* the rule itself. The rule is *static*.

Rules of logic manipulate input/ output (i/o) states, or more concrete: they translate input- into output states (section 8.1.3). What characterises a system is how it applies logic knowledge in order to translate state knowledge. Yet how to imagine a knowledge model consisting of state- as well as logic parts? The famous *Model View Controller* (MVC) pattern was introduced in section 4.2.1 and reviewed once more in section 8.2.2. The *Hierarchical MVC* (HMVC) of section 4.2.1 extended the MVC pattern towards a hierarchy of *MVC Triads*. On that basis, section 7.2.4 demonstrated the omnipresence of hierarchies in a system.

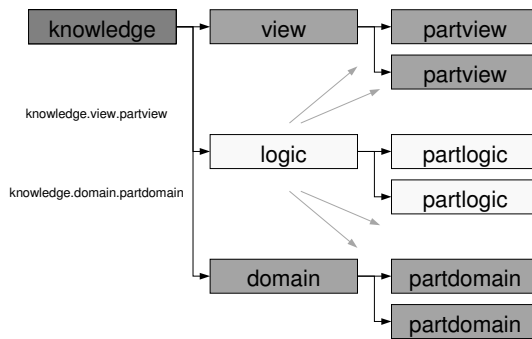


Figure 8.15: Runtime Model Hierarchy with Logic manipulating States

Figure 8.15 shows the three parts: *Domain* (Model), *View* and *Logic* (Controller) of the MVC pattern as independent branches of one common knowledge tree. Each of them represents a concept on its own. The logic model, however, is allowed to access and change the view- and domain model; it is able to link different knowledge models, to connect discrete concepts. But view- and domain model, representing states, are not allowed to access the logic model. In other words: The dependencies between logic- and state models are *unidirectional*. New state models such as textual- or graphical views could be added anytime. All that would be needed to make a system work with new state models, is the corresponding translation logic, given in form of logic models.

Further examples may be given. A rather easy one is the *Addition* of two numbers. The corresponding knowledge model looks pretty similar to the one shown in figure 8.15, only that there are three state models: *summand_1*, *summand_2* and *sum*. The logic model is

called *add*. While being processed, the *add* operation reads both summands and writes their sum to the equally named state model. More complex examples are the *Data Mapper* and *Data Transfer Object* (DTO) patterns reflected upon in section 8.2. Just like the MVC pattern, both want the same: translation for communication.

Overcoming the classical scheme of thinking in terms of *Frontend*, *Backend*, *Domain* and *Communication*, a translator-based architecture treats them all similar, as passive data models which can be converted into each other – as opposed to the traditional approach and patterns that unnecessarily complicate their handling. Translators simplify architectures by unifying the implementation and mapping of any kind of transfer model, thereby avoiding redundant parts. Resulting systems are highly flexible, easily extensible and better maintainable, because the interdependency of domain knowledge, user interface, persistence layer and (remote) communication layer is abolished.

The clear separation of states and logic into discrete, independent models avoids unwanted dependencies as caused by the bundling of attributes and methods in *Object Oriented Programming* (OOP) (section 4.1.15).

One major innovation of the kind of programming suggested in this work is that logic knowledge itself gets manipulatable. A logic model (algorithm, workflow) cannot only access and change state-, but also logic models, and even itself! Because models modified in that manner can be made persistent in form of CYBOL knowledge templates (chapter 10), and be reloaded the next time an application starts, this may be seen as a kind of *Meta Programming*, which [60] defines as: *the writing of programs that write or manipulate other programs (or themselves) as their data*.

8.3.5 Without Capsules?

Once again it has to be said that all this becomes possible only because all domain/ application knowledge is stored together in one single tree structure which is hold and managed by the *Cybernetics Oriented Interpreter* (CYBOI) (chapter 10). What was traditionally criticised as *Side Effect*, is now a *wanted* effect. Low-level system procedures within CYBOI forward just one pointer – the root of the knowledge tree, which they all may access and manipulate. Data values do *not* get copied among procedures; they exist just once in the knowledge tree and may be used by any procedure. Of course, this also means that any application has access to the knowledge of any other application. Ways ensuring sufficient security have to be found here (section 13.2).

Besides the *Encapsulation* of data through *Procedures*, there are other forms of encapsulating data, such as the *Class* (section 4.1.15). One of its purposes was to preserve transient data in memory, another to restrict access to certain data. In CYBOP, both tasks are taken care of by CYBOI. It holds the singular knowledge tree and manages access to it, through well-defined low-level procedures.

There are a number of advantages to this style of programming: An application developer has no chance of accessing memory areas directly, which prevents memory leaks and wrong pointers. Because all knowledge can be accessed through well-defined paths into the knowledge tree only, arbitrary security mechanisms may be applied and switched as needed, at runtime. Since all algorithms (logic knowledge) work with references to data in the knowledge tree (*Call by Reference*), no more data values need to be copied locally (*Call by Value*), which ensures efficient memory usage. Errors are not to be expected, because nonexistent knowledge references are simply ignored by CYBOI.

Part III

Proof

9 Cybernetics Oriented Language

The Whole is more than the Sum of its Parts.

ARISTOTLE

Chapter 7 introduced a new model for structuring knowledge, which chapter 8 separated into state- and logic knowledge. What still has to be given though, is a *Proof of Operability* for these concepts. The following sections will therefore define the *Cybernetics Oriented Language* (CYBOL), which contains all new principles and ideas, as first mentioned in [125].

9.1 Formality

Abstract models can be described in different ways, for example [250]:

- *informally* by natural language
- *semi-formally* by diagrams
- *formally* by a programming language

The use of a programming language eases model abstraction for human programmers. Special tools exist that break down models given in form of programming language code into their binary form, into sequences of *0* and *1*. These sequences are called *Machine Language* because they are understood by computers.

Classical programming languages have the linguistic means to express high-level *Knowledge* as well as low-level *System Control* operations, such as those for *input/ output* (i/o), necessary for communication. The use of such languages inevitably leads to a mess in program

code because knowledge and system control are mixed up. Inflexible, overly complex systems with numerous interdependencies are the result. Part I of this work criticised some of the weak points of traditional programming language concepts.

This work makes the necessary split: Knowledge gets *separated* from system control. Chapter 6 already discussed this separation giving manifold examples, taken from several sciences. The CYBOL language being described in the next sections is just another form of storing knowledge. It can therefore also be called a *Knowledge Modelling Language*. Any other low-level system control functionality belongs to the *Cybernetics Oriented Interpreter* (CYBOI), which gets introduced in the later chapter 10.

9.2 Definition

When defining a new language, several linguistic aspects have to be considered. One way to systematise *Language Analysis* as part of *Linguistics* is to stratify it [225] into the four fields:

- *Context*: topic described by the language, relationships between discussants, channel of communication
- *Semantics*: meaning of language symbols and character strings; includes what is usually called *Pragmatics* (*Use* of language)
- *Lexico-Grammar*: syntactic organisation of words into utterances
- *Phonology-Graphology*: study of the sound system and its phonemes [320]; judging a person's character from his handwriting [212]

Many more (sub) fields like *Orthography* and also other systematics [75] exist that will not be considered further in this document. *Phonology* and *Graphology* are not considered either, and the *Context* of CYBOL is very clear: It is to become a language for knowledge modelling. The remaining fields *Syntax* and *Semantics* are important for the definition of CYBOL and will get their own sections following. Another point receiving attention is the language *Vocabulary* (*Terms*) whose background in abstraction was discussed in sections 7.1.6 and 8.3.

The CYBOP knowledge schema (section 7.3) is based upon two kinds of hierarchies, one representing *Whole-Part* relations and the other the *Meta Information* which a whole keeps

about its parts. The syntax and semantics of CYBOL as new language must be rich enough to express abstract models using these kinds of hierarchies.

Yet before inventing a completely new language definition, it seems useful to make use of existing technologies and solutions. An interesting candidate is the *Extensible Markup Language* (XML), as introduced in section 4.1.12.

9.2.1 Syntax

Every language has a special *Syntax*, that is a *Grammar* with rules for combining terms and symbols [143]. CYBOL could define its own syntax or use an already existing one, of another language. Because of its popularity, clear text representation, flexibility, extensibility and ease of use, *XML* was chosen to deliver the syntax for CYBOL.

To mention just two of the syntactical elements of XML, *Tag* and *Attribute* are considered shortly here. Tags are special, arbitrary keywords that have to be defined by the system working with an XML document. Attributes keep additional information about the contents enclosed by two tags. Two examples:

```
<tag attribute="value">
  contents
</tag>

<tag attribute1="value" attribute2="contents"/>
```

An XML document carries a name and can such represent a *Discrete Item*, as suggested by the principles of human thinking (section 7.1). Being a *Compound*, it consists of parts – and, it can link to other documents treated as its parts. That way, a whole hierarchy can be formed. Tag attributes can keep additional information about the linked parts. Most importantly, XML documents have a hierarchical structure based on tags, which may be used to store meta information about a part.

Considering these properties of XML, it seems predestinated for formally representing abstract models using the CYBOP concepts. CYBOL, finally, is XML *plus* a defined set of tags, attributes and values, used to structure and link documents meaningfully.

9.2.2 Vocabulary

The *Vocabulary* is what fills a language with life. It delivers the *Terms* and *Symbols* that are combined after the rules of a syntax.

XML allows to define and exchange the whole vocabulary of a language. It offers two ways in which a list of legal elements can be defined: The traditional *Document Type Definition* (DTD) and the more modern *XML Schema Definition* (XSD). Besides the vocabulary, DTD and XSD define the structure of an XML document and allow to typify, constrain and validate items. In addition to DTD and XSD, the *Extended Backus Naur Form* (EBNF) of CYBOL is given following.

The language definitions were not added as appendix to this work, because firstly, they are not too long and secondly, an understanding of the CYBOL elements is necessary to be able to grasp the constructs and examples given in later sections.

Document Type Definition

A DTD represents the type definition of an SGML or XML document. It consists of a set of *Markup Tags* and their *Interpretation* [143]. DTDs can be declared inline, within a document, or as an external reference [272]. Figure 9.1 shows the DTD of the CYBOL language.

Following the pure hierarchical structure of the CYBOP knowledge schema (section 7.3), it would actually suffice to use a DTD as simple as the one shown in figure 9.2. Since the three elements *part*, *property* and *constraint* (compare figure 9.1) have the same list of required attributes, they could be summarised under the name *part*, for example. Because the structure of a CYBOL model is non-ambiguous, the meaning of its elements can be guessed from their position within the model.

For the purpose of expressing knowledge in accordance with the schema suggested by CYBOP, an XML document does not need to have a root element. The document's (file) name clearly identifies a model. For reasons of XML conformity, however, an extra root element called *model* was defined (figure 9.1). And for reasons of better readability and programmability, the three kinds of embedded elements were given distinct names.

```
<!ELEMENT model (part*)>
<!ELEMENT part (property*)>
<!ELEMENT property (constraint*)>
<!ELEMENT constraint EMPTY>

<!--
  ATTLIST part
  -->
<!--
  ATTLIST property
  -->
<!--
  ATTLIST constraint
  -->
```

name CDATA #REQUIRED
channel CDATA #REQUIRED
abstraction CDATA #REQUIRED
model CDATA #REQUIRED

name CDATA #REQUIRED
channel CDATA #REQUIRED
abstraction CDATA #REQUIRED
model CDATA #REQUIRED

name CDATA #REQUIRED
channel CDATA #REQUIRED
abstraction CDATA #REQUIRED
model CDATA #REQUIRED

Figure 9.1: Recommended CYBOL DTD

```
<!ELEMENT part (part*)>

<!--
  ATTLIST part
  -->
```

name CDATA #REQUIRED
channel CDATA #REQUIRED
abstraction CDATA #REQUIRED
model CDATA #REQUIRED

Figure 9.2: Simplified CYBOL DTD

XML Schema Definition

XML Schema is an XML-based alternative to DTD [272], and XSD is its definition language. There is a lot of discussion going on about the sense or *Myth* of XML Schema [38], that this document will not take part in. Figure 9.4 shows the XSD of the CYBOL language.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.cybop.net"
  xmlns="http://www.cybop.net" elementFormDefault="qualified">
  <xs:element name="part">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="part" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="channel" type="xs:string" use="required"/>
      <xs:attribute name="abstraction" type="xs:string" use="required"/>
      <xs:attribute name="model" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 9.3: Simplified CYBOL XSD

Again, a simplified version of that XSD could be created (figure 9.3). But for reasons explained before, the recommended XSD is the one shown in figure 9.4.

Extended Backus Naur Form

The EBNF adds regular expression syntax to the *Backus Naur Form* (BNF) notation [13], in order to allow very compact specifications [184]. Figure 9.5 shows the EBNF of the CYBOL language.

```

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.cybop.net"
  xmlns="http://www.cybop.net" elementFormDefault="qualified">
  <xs:element name="model">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="part" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="part">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="channel" type="xs:string" use="required"/>
      <xs:attribute name="abstraction" type="xs:string" use="required"/>
      <xs:attribute name="model" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="property">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="constraint" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="channel" type="xs:string" use="required"/>
      <xs:attribute name="abstraction" type="xs:string" use="required"/>
      <xs:attribute name="model" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="constraint">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="channel" type="xs:string" use="required"/>
      <xs:attribute name="abstraction" type="xs:string" use="required"/>
      <xs:attribute name="model" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Figure 9.4: Recommended CYBOL XSD

```

CYBOL      = '<model>'
              {part}
              '</model>';

part        = '<part ' attributes '\>' |
              '<part ' attributes '>'
              {property}
              '</part>';

property     = '<property ' attributes '\>' |
              '<property ' attributes '>'
              {constraint}
              '</property>';

constraint   = '<constraint ' attributes '\>';

attributes   = name_attribute channel_attribute abstraction_attribute model_attribute

name_attribute   = 'name=' name ' ';
channel_attribute = 'channel=' channel ' ';
abstraction_attribute = 'abstraction=' abstraction ' ';
model_attribute  = 'model=' model ' ';

name           = description_sign;
channel         = description_sign;
abstraction     = description_sign;
model           = value_sign;

description_sign = { ( letter | number ) };
value_sign       = { ( letter | number | other_sign ) };

letter          = small_letter | big_letter;
small_letter     = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                  'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                  'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                  'v' | 'w' | 'x' | 'y' | 'z';
big_letter       = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                  'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
                  'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
                  'V' | 'W' | 'X' | 'Y' | 'Z';
other_sign       = ',' | '.' | '/' | '+' | '-' | '*';
number           = '0' | '1' | '2' | '3' | '4' |
                  '5' | '6' | '7' | '8' | '9';

```

Figure 9.5: CYBOL in EBNF

9.2.3 Semantics

The meaning expressed by terms and sentences is their *Semantics* [71].

CYBOL files can be used to model either *State-* or *Logic Knowledge* (chapter 8). In both cases, the *same* syntax (document structure) with *identical* vocabulary (XML tags and -attributes) is applied. It is the attribute *Values* that make a difference in meaning.

The double hierarchy proposed by CYBOP's knowledge schema (section 7.3) is put into static CYBOL knowledge templates, by using XML *Attributes* for representing the whole-part hierarchy, and XML *Tags* for representing the additional meta information that a whole model keeps about its part models.

Attributes

Normally, an XML *Attribute* keeps meta information about the contents of an XML *Tag*. In CYBOL, however, three attributes keep meta information about a fourth attribute. The attributes, altogether, are:

- name
- channel
- abstraction
- model

The attribute of greatest interest is *model*. It contains a model either directly, or a path to one. The *channel* attribute indicates whether the *model* attribute's value is to be read from:

- inline
- file
- ftp
- http

The *abstraction* attribute specifies how to interpret the model pointed to by the *model* attribute's value. A model may be given in formats like for example:

- cybol (a state- or logic compound model encoded in CYBOL format)
- operation (a primitive logic model)
- string
- double
- integer
- boolean

The *name* attribute, finally, provides the referenced model with a unique identifier.

While the interpretation of the *model* attribute's value depends on the *channel*- and *abstraction* attributes, the other three attributes (*name*, *channel*, *abstraction*) themselves always get interpreted as character string.

Tags

There are many kinds of meta information besides the above-mentioned attributes, that may be known about a model. These are given in special XML tags called *property* and *constraint*. As defined in section 9.2.2, a CYBOL knowledge template may use four kinds of XML tags:

- model
- part
- property
- constraint

The *model* tag appears just once. It is the root node which makes a CYBOL knowledge template a valid XML document.

Of actual interest are the *part* tags. They identify the models that the *whole* model described by the CYBOL knowledge template consists of.

A *whole* model may know a lot more about its *part* models, than is given by a part model's XML attributes. A spatial state model may know about the *position* and *size* of its parts, in space. A temporal model (such as a workflow) may have to know about the *position* of its parts in time, in order to be able to execute them in the correct order. Further, the temporal model needs to know about the *input/output* (i/o) state models which are to be

manipulated by the corresponding logic operation (part model). The number of parts within a whole (compound) model may be limited. And so on. These additional information are provided by *property* tags whose number is conceptually unlimited.

Not only parts need additional meta information; properties may need such information, too. The position or size as properties of a part may have to be constrained to certain values, such as a *minimum* or *maximum*. The values of the *colour* property of a part model may have to be chosen out of a pre-defined set called *choice*. Information of that kind are stated in *constraint* tags.

Since the number of possible meta information implementable in CYBOL is already quite large and steadily growing, as the development continues, this section cannot list them all. At a future point in time, a more-or-less complete CYBOL specification document may be found at the CYBOP project's website [256].

9.2.4 Tag-Attribute Swapping

CYBOL swaps the meaning attributes and tags traditionally have in XML documents, where tags represent elements that may be nested infinitely and attributes hold additional (meta) information about a tag. Following an example of how CYBOL might have looked that way:

```
<model>
  <part>
    <name="title"/>
    <channel="inline"/>
    <abstraction="character"/>
    <model="Res Medicinae"/>
  </part>
  <part layout="compass" position="north">
    <name="menu_bar"/>
    <channel="file"/>
    <abstraction="cybol"/>
    <model="gui/menu_bar.cybol"/>
  </part>
</model>
```

The current final specification of CYBOL, on the contrary, uses attributes to define a nested element (part) and tags to give properties (meta information) about such a nested element, in the following way:

```

<model>
  <part name="title" channel="inline" abstraction="character" model="Res Medicinae"/>
  <part name="menu_bar" channel="file" abstraction="cybol" model="gui/menu_bar.cybol">
    <property name="layout" channel="inline" abstraction="character" model="compass"/>
    <property name="position" channel="inline" abstraction="character" model="north"/>
  </part>
</model>

```

This is because:

1. the number of attributes specifying a part in CYBOL is fixed, whereas the number of tags specifying a property of a part is not, and the number of XML tags is easier extensible than that of attributes;
2. that way it is also possible to specify a part without any properties in just one CYBOL code line, while otherwise four tags would always have to be given;
3. not only a part may be nested (consist of smaller parts), but also a property may be (for example a position consisting of three coordinates given as parts), which necessitates the four standard attributes to be given for properties and constraints as well.

9.3 Constructs

After having defined the CYBOL language in the previous section, the following examples can demonstrate how the language's constructs may be used in practice. Attention is also paid to how control structures of classical programming languages (compare section 4.1.6) may be implemented in CYBOL. Additionally, this section discusses how inheritance, containers and software patterns were considered in the design of CYBOL.

9.3.1 State Examples

The creation of composed state models is quite straightforward and clear, as the following CYBOL knowledge templates show.

Model-Part Relation

The DocBook DTD [336] is one of many well-known specifications for structuring documents. The Linux Documentation Project [197] makes heavy use of it. DocBook is based on numerous XML tags with defined meaning.

The following example shows how parts of a *Text Document* can be modelled differently, with at most four tags, using CYBOL:

```
<model>
  <part name="title" channel="inline" abstraction="string" model="Quo Vadis"/>
  <part name="author" channel="inline" abstraction="string" model="Henryk Sienkiewicz"/>
  <part name="date" channel="inline" abstraction="date" model="1896-01-01"/>
  <part name="contents" channel="file" abstraction="cybol" model="contents.cybol"/>
  <part name="chapter_1" channel="file" abstraction="cybol" model="chapter_1.cybol"/>
  <part name="chapter_2" channel="file" abstraction="cybol" model="chapter_2.cybol"/>
  <part name="chapter_3" channel="file" abstraction="cybol" model="chapter_3.cybol"/>
  <part name="appendix" channel="file" abstraction="cybol" model="appendix.cybol"/>
</model>
```

Meta Properties

When modelling *Graphical User Interfaces* (GUI), a speciality to take care about is the *Position* of GUI elements within their surrounding container. GUI components may have very different orderings and positions. The *Java Swing* framework [112], for example, offers *BorderLayout*, *BoxLayout*, *CardLayout*, *FlowLayout*, *GridBagLayout* etc.

The following example of a GUI *Dialogue* assumes that an interpreter knows how to handle *Compass* layouts, which are the pendant of the above-mentioned *BorderLayout*:

```
<model>
  <part name="title" channel="inline" abstraction="string" model="Prescription Dialogue"/>
  <part name="menu_bar" channel="file" abstraction="cybol" model="menu_bar.cybol">
    <property name="position" channel="inline" abstraction="string" model="north"/>
  </part>
  <part name="tool_bar" channel="file" abstraction="cybol" model="tool_bar.cybol">
    <property name="position" channel="inline" abstraction="string" model="west"/>
  </part>
  <part name="contents_panel" channel="file" abstraction="cybol" model="contents_panel.cybol">
    <property name="position" channel="inline" abstraction="string" model="centre"/>
  </part>
  <part name="status_bar" channel="file" abstraction="cybol" model="status_bar.cybol">
```

```

    <property name="position" channel="inline" abstraction="string" model="south"/>
  </part>
</model>

```

Further meta information such as the *Colour* or *Size* of a GUI component may be given. The following example shows how a GUI *Button* may be modelled as part of some GUI panel. Again, properties like *size* are not modelled as part, because the button does not *consist* of them, in a structural way of thinking:

```

<model>
  <part name="exit_button" channel="file" abstraction="cybol" model="exit_button.cybol">
    <property name="position" channel="inline" abstraction="integer" model="0"/>
    <property name="size" channel="inline" abstraction="vector" model="80,20,1"/>
    <property name="colour" channel="inline" abstraction="rgb" model="127,127,127"/>
    <property name="action" channel="inline" abstraction="string" model="exit.cybol"/>
  </part>
</model>

```

External Resources

A *Text Document* like the one shown in the example above often contains graphical illustrations called *Figures*, which it may include from external files. One common graphics format is *Encapsulated PostScript* (EPS), for example. *Graphical User Interfaces* (GUI) as modelled before do contain *Icons*; a GUI button may contain a *Glyph* and so forth.

CYBOL therefore offers ways for linking external resources, given in various formats, as shown in the following hypothetical knowledge template. The last of the template's parts retrieves its data not from a file in the local file system, but across a *Hyper Text Transfer Protocol* (HTTP) link instead:

```

<model>
  <part name="pdf_document" channel="file" abstraction="pdf" model="example.pdf"/>
  <part name="ogg_audio" channel="file" abstraction="ogg" model="example.ogg"/>
  <part name="mpeg_video" channel="file" abstraction="mpeg" model="example.mpeg"/>
  <part name="eps_image" channel="file" abstraction="eps" model="example.eps"/>
  <part name="jpeg_image" channel="http" abstraction="jpeg" model="host.domain.tld/example.jpeg"/>
</model>

```

Serialised Model

A possible (but not necessarily recommended) alternative to the linking of external resources is to store such information (as binary code) *inline* in the CYBOL knowledge template. One case in which it is necessary to store all information inline in the model is *Serialisation*.

A CYBOL address management application that does not rely on the existence of a *Database Management System* (DBMS) probably has to store addresses in form of serialised files, such as the one shown following. It contains two parts representing dynamically extensible lists, one for *phone_numbers* and another one for *addresses*:

```
<model>
  <part name="honorific_prefix" channel="inline" abstraction="string" model="Dr."/>
  <part name="given_name" channel="inline" abstraction="string" model="Tux"/>
  <part name="family_name" channel="inline" abstraction="string" model="Penguin"/>
  <part name="phone_numbers" channel="inline" abstraction="cybol" model="(
    <part name="home" channel="inline" abstraction="string" model="123"/>
    <part name="work" channel="inline" abstraction="string" model="456"/>
    <part name="mobile" channel="inline" abstraction="string" model="789"/>
  )"/>
  <part name="addresses" channel="inline" abstraction="cybol" model="(
    ...
  )"/>
</model>
```

The serialisation of CYBOL models causes one problem: Due to the double hierarchy (section 7.3.3) to which belong *Whole-Part* relations (stored in XML attributes) and *Meta Information* (stored in XML tags), it is not possible to store CYBOL models in an XML-conform manner. Instead of referencing external files containing the corresponding CYBOL *Part* models, a serialised *Whole* model has to contain these inline.

While XML tags were invented as pairs consisting of a *begin* and an *end* tag, XML attribute values are enclosed by simple quotation marks. Hence, the beginning markup of an attribute value does not look any different than its ending markup. This is a true problem, because serialised whole-part hierarchies of CYBOL models, with attribute values containing complete sub models with their own attributes, would get completely mixed up in pure XML notation.

It was therefore inevitable to break XML-conformity and introduce two additional markup tokens "(and)", indicating the beginning and end of an XML attribute value. The tokens are extensions of the quotation marks of standard XML attributes, with one left/ right

parenthesis, respectively. That way, the degree to which attributes are nested becomes countable and it is always clear to which tag an attribute belongs.

Meta Constraints

The example of this section shows a possible *Debian GNU/Linux* [258] *Package* definition, written in CYBOL:

```
<model>
  <part name="name" channel="inline" abstraction="string" model="resmedicinae"/>
  <part name="version" channel="inline" abstraction="string" model="0.1.0.0"/>
  <part name="section" channel="inline" abstraction="string" model="science"/>
  <part name="priority" channel="inline" abstraction="string" model="optional"/>
  <part name="architecture" channel="inline" abstraction="string" model="all"/>
  <part name="packages" channel="file" abstraction="cybol" model="resmedicinae-packages"/>
  <part name="files" channel="file" abstraction="cybol" model="resmedicinae-files"/>
  <part name="maintainer" channel="inline" abstraction="string" model="Happy Coder"/>
  <part name="description" channel="inline" abstraction="string" model="Medical System"/>
</model>
```

The part called *packages* in the example above references an external CYBOL knowledge template, which is displayed below. It represents a list of packages having different versions and varying strengths of dependency. The *strength* property of the last of these packages has the model value *suggests* and, it contains meta information about that property, namely a *constraint*. Constraints can be, for example: minima, maxima or a choice of possible values, as in this case.

```
<model>
  <part name="cyboi" channel="inline" abstraction="string" model="cyboi">
    <property name="strength" channel="inline" abstraction="string" model="depends"/>
    <property name="version" channel="inline" abstraction="string" model=">= 1.0.0.0"/>
    <property name="conflicts" channel="inline" abstraction="string" model="< 1.0.0.0"/>
  </part>
  <part name="cybol-healthcare" channel="inline" abstraction="string" model="cybol-healthcare">
    <property name="strength" channel="inline" abstraction="string" model="depends"/>
    <property name="version" channel="inline" abstraction="string" model=">= 0.1.0.0"/>
  </part>
  <part name="resadmin" channel="inline" abstraction="string" model="resadmin">
    <property name="strength" channel="inline" abstraction="string" model="recommends"/>
    <property name="version" channel="inline" abstraction="string" model=">= 0.8.0.0"/>
  </part>
  <part name="resmedicinae-doc" channel="inline" abstraction="string" model="resmedicinae-doc">
```



```

    <property name="strength" channel="inline" abstraction="string" model="suggests">
      <constraint name="choice" channel="inline" abstraction="set" model="suggests,recommends"/>
    </property>
    <property name="version" channel="inline" abstraction="string" model=">= 0.1.0.0"/>
  </part>
</model>

```

9.3.2 Logic Examples

The CYBOL implementation of logic models (chapter 8) needs more detailed explanation, in particular the use of special control structures as known from *Structured and Procedural Programming* (SPP) (section 4.1.6).

Operation Call

As stated previously, logic models may access and manipulate state models. The simplest form of a logic model is an operation with associated input/ output (i/o) state models. The following CYBOL knowledge template calls an *add* operation, handing over i/o parameters as *properties* of the corresponding *part*:

```

<model>
  <part name="addition" channel="inline" abstraction="operation" model="add">
    <property name="summand_1" channel="inline" abstraction="integer" model="1"/>
    <property name="summand_2" channel="inline" abstraction="knowledge" model="domain.summand"/>
    <property name="sum" channel="inline" abstraction="knowledge" model="domain.result"/>
  </part>
</model>

```

The example nicely shows how state models can be given in various formats. The *summand_1* is given as constant value, defined directly in the knowledge template. Its type of abstraction is *integer*. The *summand_2*- and *sum* parameters, on the other hand, are given as dot-separated references to the runtime tree of knowledge models. Their type of abstraction is therefore *knowledge*.

Algorithm Division

Compound logic models like *Algorithms*, which SPP languages implement using nested *Blocks*, can be expressed in CYBOL as well. It does not provide blocks in the classical sense, but its hierarchical structure allows to subdivide compound knowledge templates, and to cascade compound logic as well as primitive operations. The following example calls an addition operation, before a compound algorithm, situated in an external CYBOL file, gets executed:

```
<model>
  <part name="addition" channel="inline" abstraction="operation" model="add">
    <property name="summand_1" channel="inline" abstraction="knowledge" model="domain.number_1"/>
    <property name="summand_2" channel="inline" abstraction="knowledge" model="domain.number_2"/>
    <property name="sum" channel="inline" abstraction="knowledge" model="domain.number_3"/>
  </part>
  <part name="algorithm" channel="file" abstraction="cybol" model="logic/algorithm.cybol"/>
</model>
```

Simple Assignment

CYBOL does not know *Variables* as used in classical languages. All states a system may take on are represented by just one *Knowledge Tree* (compare chapter 6), which applications may access in a defined manner (dot-separated knowledge paths). Consequently, *Assignments* are done differently in CYBOL than in classical programming languages. All kinds of state changes go back to a manipulation of the one knowledge tree:

```
<model>
  <part name="copy_value" channel="inline" abstraction="operation" model="copy">
    <property name="source" channel="inline" abstraction="knowledge" model="domain.name"/>
    <property name="destination" channel="inline" abstraction="knowledge" model="gui.name"/>
  </part>
  <part name="move_branch" channel="inline" abstraction="operation" model="move">
    <property name="source" channel="inline" abstraction="knowledge" model="address_1.phone"/>
    <property name="destination" channel="inline" abstraction="knowledge" model="address_2"/>
  </part>
</model>
```

The first operation in the example above copies a value between two branches of the tree. Only primitive values can be copied. The second operation removes a whole tree branch

(referenced by the *source* property) from one parent node, and adds it to another (referenced by the *destination* property).

Loop as Operation

Looping is a major technique for the effective processing of whole stacks of data. As many other control structures, it is simplified to a logic operation, in CYBOL.

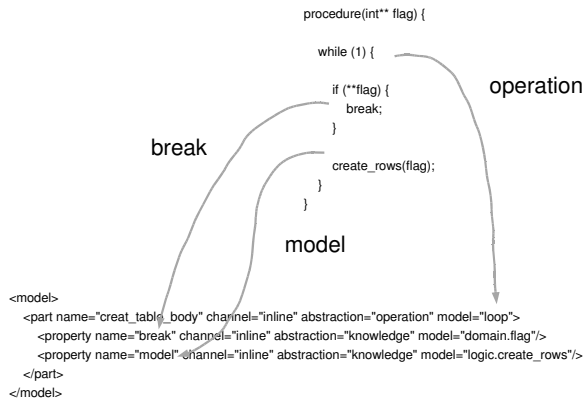


Figure 9.6: Loop Control Structure and Elements in C and CYBOL

The *loop* operation needs two parameters to be functional: a *break* flag as means of interruption and a logic *model* to be executed in each loop cycle (figure 9.6). An *index* counting loop cycles is not given, as it is in the responsibility of the logic *model* to manage that index, just like the setting of the *break* flag, internally. The following example dynamically creates a table consisting of a number of rows:

```

<model>
  <part name="creat_table_body" channel="inline" abstraction="operation" model="loop">
    <property name="break" channel="inline" abstraction="knowledge" model="domain.flag"/>
    <property name="model" channel="inline" abstraction="knowledge" model="logic.create_rows"/>
  </part>
</model>
  
```

Conditional Execution

An obviously presupposed part in the previous example is a logic setting the break *condition* (flag). If the break flag was not set, the loop would run endlessly. The following knowledge template therefore shows a *comparison* operation, as it could stand at the end of the loop's logic model, referenced by the *model* property in the previous example. After having compared the current loop index with a maximum loop count number, the break flag may or may not be set. When entering its next cycle, the loop operation checks whether the flag is set. If so, the loop is stopped:

```
<model>
  <part name="comparison" channel="inline" abstraction="operation" model="compare">
    <property name="operand_1" channel="inline" abstraction="knowledge" model="domain.index"/>
    <property name="operand_2" channel="inline" abstraction="knowledge" model="domain.count"/>
    <property name="operator" channel="inline" abstraction="string" model="greater_or_equal"/>
    <property name="result" channel="inline" abstraction="knowledge" model="domain.flag"/>
  </part>
</model>
```

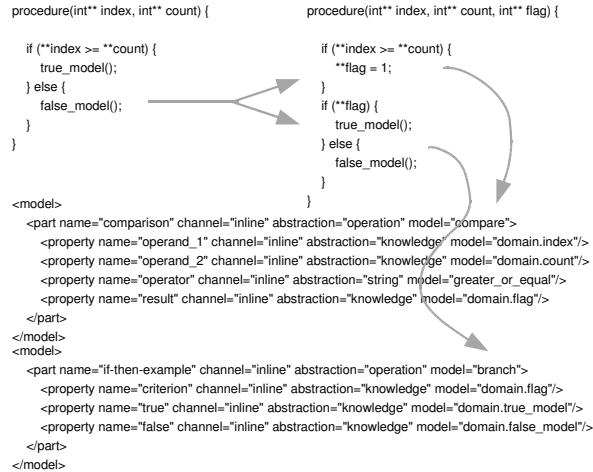


Figure 9.7: Condition Control Structure and Elements in C and CYBOL

Flags as one of the earliest techniques used in computing (in software as well as in hardware [180]) are the perfect means for controlling the execution of primitive logic models, namely operations. They represent a condition set as result of another logic model – the latter often

being some kind of comparison operation. In order to execute code upon activation of a flag, a conventional comparison control structure needs to be split up into two independent blocks (figure 9.7), with the flag being the linking element. The flag which was set by a comparison operation is used for branching the control flow.

The second example shows how a classical *if-then* statement would be written in CYBOL. The corresponding operation is called *branch* and it expects three properties: a *criterion* flag and two models, of which one is executed in case the flag is *true* and the other is executed otherwise.

```
<model>
  <part name="if-then-example" channel="inline" abstraction="operation" model="branch">
    <property name="criterion" channel="inline" abstraction="knowledge" model="domain.flag"/>
    <property name="true" channel="inline" abstraction="knowledge" model="domain.true_model"/>
    <property name="false" channel="inline" abstraction="knowledge" model="domain.false_model"/>
  </part>
</model>
```

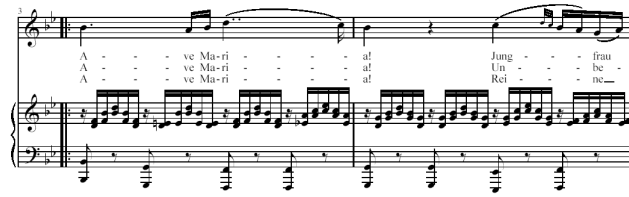
9.3.3 Special Examples

XML is used for representing data of very different domains, and a whole plethora of *XML* dialects exists. Two of them are mentioned following. The main purpose of the next examples, however, is to show how CYBOL can replace these.

Synchronous Execution

MusicXML [200] is a markup language *designed to represent musical scores, specifically common western musical notation from the 17th century onwards*. In principle, CYBOL could be used for this purpose as well. Of course, there are many details (additional properties) which would still have to be worked out in order to be able to correctly represent complete musical scores. As most models, the *Musical Work* displayed in figure 9.8 can be considered a hierarchy consisting of *Parts* (played/ sung by instruments/ voices). Parts in turn consist of *Measures*, which consist of *Notes*, which finally have a *Pitch* and sometimes *Lyric*.

The following knowledge templates deliver only short examples showing how music may be modelled in CYBOL. Their property names were taken over from *MusicXML*'s element tags,

Figure 9.8: Musical Score of Franz Schubert's *Ave Maria* [200]

as elaborated in [200]. Most are self-explanatory and shall not be further discussed here. The first example template represents an extract from a complete musical *Work*, consisting of the two parts *Voice* and *Piano*:

```
<model>
  <part name="number" channel="inline" abstraction="string" model="D. 839"/>
  <part name="title" channel="inline" abstraction="string" model="Ave Maria (Ellen's Gesang III)"/>
  <part name="composer" channel="inline" abstraction="string" model="Franz Schubert"/>
  <part name="poet" channel="inline" abstraction="string" model="Walter Scott"/>
  <part name="voice" channel="file" abstraction="cybol" model="voice.cybol">
    <property name="score_instrument" channel="inline" abstraction="string" model="P1-I14"/>
    <property name="instrument_name" channel="inline" abstraction="string" model="Choir Aahs"/>
    <property name="midi_instrument" channel="inline" abstraction="string" model="P1-I14"/>
    <property name="midi-channel" channel="inline" abstraction="integer" model="1"/>
    <property name="midi-program" channel="inline" abstraction="integer" model="53"/>
  </part>
  <part name="piano" channel="file" abstraction="cybol" model="piano.cybol">
    <property ...
  </part>
</model>
```

One of the *Parts* is shown in the next template. It consists of several measures:

```

<model>
  <part name="measure_1" channel="file" abstraction="cybol" model="measure_1.cybol">
    <property name="divisions" channel="inline" abstraction="integer" model="48"/>
    <property name="key_fifths" channel="inline" abstraction="integer" model="-2"/>
    <property name="key_mode" channel="inline" abstraction="string" model="major"/>
    <property name="beats" channel="inline" abstraction="integer" model="4"/>
    <property name="beat_type" channel="inline" abstraction="integer" model="4"/>
    <property name="staves" channel="inline" abstraction="integer" model="0"/>
    <property name="clef_sign" channel="inline" abstraction="string" model="G"/>
    <property name="clef_line" channel="inline" abstraction="integer" model="2"/>
  </part>
  <part name="measure_2" channel="file" abstraction="cybol" model="measure_2.cybol">
    <property ...
  </part>
</model>

```

A *Measure* again consists of *Notes*:

```

<model>
  <part name="note_1" channel="file" abstraction="cybol" model="note_1.cybol">
    <property name="duration" channel="inline" abstraction="integer" model="72"/>
    <property name="voice" channel="inline" abstraction="integer" model="1"/>
    <property name="type" channel="inline" abstraction="string" model="quarter"/>
    <property name="stem" channel="inline" abstraction="string" model="down"/>
    <property name="position" channel="inline" abstraction="integer" model="1"/>
  </part>
  <part name="note_2" channel="file" abstraction="cybol" model="note_2.cybol">
    <property name="duration" channel="inline" abstraction="integer" model="12"/>
    <property name="voice" channel="inline" abstraction="integer" model="1"/>
    <property name="type" channel="inline" abstraction="string" model="16th"/>
    <property name="stem" channel="inline" abstraction="string" model="up"/>
    <property name="position" channel="inline" abstraction="integer" model="2"/>
  </part>
  <part name="note_3" channel="file" abstraction="cybol" model="note_3.cybol">
    <property ...
    <property name="position" channel="inline" abstraction="integer" model="2"/>
  </part>
</model>

```

An important property to note here is the *position* value. It is common that two notes have to be played at the same time, the notes then being called a *Chord*. In contrast to MusicXML which provides an own tag to denote notes belonging to the same chord, CYBOL suggests to use a *position* property having identical values for all notes in a chord. An interpreter

program may thus not only read necessary sequence information, but can also figure out which of the notes have to be played *synchronously*.

A fourth example represents one *Note*, consisting of a *Pitch* and *Lyric* text, which are the final abstractions in this knowledge template:

```
<model>
  <part name="pitch" channel="inline" abstraction="string" model="B">
    <property name="alter" channel="inline" abstraction="integer" model="-1"/>
    <property name="octave" channel="inline" abstraction="integer" model="4"/>
  </part>
  <part name="lyric" channel="inline" abstraction="string" model="A">
    <property name="syllabic" channel="inline" abstraction="string" model="begin"/>
  </part>
</model>
```

Presentation and Content

The *Mathematical Markup Language* (MathML) [117] provides means for representing mathematical expressions. Its specification document states:

A fundamental challenge in defining a markup language for mathematics on the Web is reconciling the need to encode both the *Presentation* of a mathematical notation and the *Content* of the mathematical idea or object which it represents. The relationship between a mathematical notation and a mathematical idea is subtle and deep. On a formal level, the results of mathematical logic raise unsettling questions about the correspondence between systems of symbolic logic and the phenomena they model. At a more intuitive level, anyone who uses mathematical notation knows the difference that a good choice of notation can make; the symbolic structure of the notation suggests the logical structure.

This observation is very important because it helps avoid mixing *Content* and *Presentation* of data. Both are discrete models, comparable to the *Domain* and *User Interface* (UI) of a software application, which can be translated into each other (sections 8.2 and 8.3.4). The good side of MathML is that it [117]: *allows authors to encode both the notation which represents a mathematical object and the mathematical structure of the object itself*. Its bad side, however, is that: *authors can mix both kinds of encoding in order to specify both the presentation and content of a mathematical idea*. Another disadvantage is that different

tags are used for presentation and contents of a mathematical model.

CYBOL uses just four tags (section 9.2.3) but can represent mathematical expressions as well. What MathML calls *Content*, becomes a *Logic* knowledge template in CYBOL. The mathematical content of the formula $(a + b)^2$ would be modelled as follows:

```
<model>
  <part name="addition" channel="inline" abstraction="operation" model="add">
    <property name="summand_1" channel="inline" abstraction="knowledge" model="domain.a"/>
    <property name="summand_2" channel="inline" abstraction="knowledge" model="domain.b"/>
    <property name="sum" channel="inline" abstraction="knowledge" model="domain.c"/>
  </part>
  <part name="exponentiation" channel="inline" abstraction="operation" model="power">
    <property name="base" channel="inline" abstraction="knowledge" model="domain.c"/>
    <property name="power" channel="inline" abstraction="integer" model="2"/>
    <property name="result" channel="inline" abstraction="knowledge" model="domain.r"/>
  </part>
</model>
```

And the formula's *Presentation* would be defined by the following two CYBOL *State* knowledge templates, of which the second one represents the *Base* that is referenced by the first one:

```
<model>
  <part name="base" channel="inline" abstraction="file" model="domain/base.cybol">
    <property name="fence" channel="inline" abstraction="boolean" model="true"/>
  </part>
  <part name="power" channel="inline" abstraction="integer" model="2">
    <property name="superscript" channel="inline" abstraction="boolean" model="true"/>
  </part>
</model>

<model>
  <part name="summand_1" channel="inline" abstraction="string" model="a"/>
  <part name="operator" channel="inline" abstraction="string" model="+"/>
  <part name="summand_2" channel="inline" abstraction="string" model="b"/>
</model>
```

Hello World

A practice popularised by Brian Kernighan and Dennis Ritchie [174] is to write as first program one that prints the words *Hello, World!*, when teaching/ learning a new programming

language. Two possible CYBOL versions of that minimal program are given following. The first consists of only two operations: *send* and *exit*. The string message to be displayed on screen is handed over as *property* to the *send* operation, before the *exit* operation shuts down the system:

```
<model>
  <part name="send_model_to_output" channel="inline" abstraction="operation" model="send">
    <property name="language" channel="inline" abstraction="string" model="tui"/>
    <property name="receiver" channel="inline" abstraction="string" model="user"/>
    <property name="message" channel="inline" abstraction="string" model="Hello, World!"/>
  </part>
  <part name="exit_application" channel="inline" abstraction="operation" model="exit"/>
</model>
```

The second example template is slightly more complex. It starts with creating a domain model that consists of just one *greeting* string. That string is then sent as message to the human user via a *Textual User Interface* (TUI), just as in the first example. The difference is that now, the greeting is not handed over as hard-coded *string* value, but is read from the runtime knowledge model, which is indicated by its *abstraction* value:

```
<model>
  <part name="create_greeting" channel="inline" abstraction="operation" model="create_part">
    <property name="name" channel="inline" abstraction="string" model="greeting"/>
    <property name="channel" channel="inline" abstraction="string" model="inline"/>
    <property name="abstraction" channel="inline" abstraction="string" model="string"/>
    <property name="model" channel="inline" abstraction="string" model="Hello, World!"/>
  </part>
  <part name="send_model_to_output" channel="inline" abstraction="operation" model="send">
    <property name="language" channel="inline" abstraction="string" model="tui"/>
    <property name="receiver" channel="inline" abstraction="string" model="user"/>
    <property name="message" channel="inline" abstraction="knowledge" model="greeting"/>
  </part>
  <part name="destroy_greeting" channel="inline" abstraction="operation" model="destroy_part">
    <property name="name" channel="inline" abstraction="knowledge" model="greeting"/>
  </part>
  <part name="exit_application" channel="inline" abstraction="operation" model="exit"/>
</model>
```

The appearance of a *create_part*/ *destroy_part* pair in the second example already suggests how an application lifecycle with *startup*-, *runtime*- and *shutdown* phase could look like in CYBOL.

Any System?

While creating the CYBOP knowledge concepts and implementing them in the CYBOL language, one main aim was to make that language as flexible as possible, in order to be usable for the development of a variety of systems. It seems that CYBOL is indeed applicable for developing standard business applications in very different domains.

It might also be usable for creating desktop environments such as the *K Desktop Environment* (KDE) [81] – and even configuration parts of an *Operating System* (OS) (the information stored in the files of the */etc* directory and the */usr/src/linux/.config* file, when taking the Linux kernel as example) could possibly be encoded in CYBOL. The same counts for the configuration files of applications residing in the */etc* directory of systems that follow the *Filesystem Hierarchy Standard* (FHS). That is, both applications and their configuration files may be written in the same format: CYBOL.

Yet to limit the scope of this work, proof for these assumptions cannot be given here. As well, the applicability of CYBOL for programming *Real Time* (RT) systems was not investigated yet. A slightly more extensive example, however, is given in chapter 11, which describes the *Res Medicinae* prototype – a yet very incomplete *Electronic Health Record* (EHR) application.

9.3.4 Inheritance as Property

One fundamental concept of *Object Oriented Programming* (OOP) is *Inheritance* (section 4.1.15). It was also determined as a basic concept of human thinking, where it is called *Categorisation*.

In principle, there is no problem with implementing inheritance in CYBOL. If done, however, it would differ from traditional class architectures as known from OOP. Classical OOP systems resolve inheritance relationships at runtime; CYBOP systems, on the other hand, would resolve them just once when creating a knowledge model (instance) from a knowledge template. After instantiation, all inheritance relationships are lost since instances are stored as purely hierarchical *whole-part* models in memory, without any links to *super* models.

The following knowledge template shows how inheritance could be realised in CYBOL. Contrary to OOP classes which hold a link to their corresponding *super* class as *intrinsic* property, a CYBOL knowledge template does not know itself from which *super* template to

inherit from. That information is stored as *extrinsic* property outside the template instead, in other words in the *whole* template to which the inheriting template belongs.

```
<model>
  <part name="ok_button" channel="file" abstraction="cybol" model="gui/ok_button.cybol">
    <property name="super" channel="file" abstraction="cybol" model="button.cybol"/>
    <property name="size" channel="inline" abstraction="vector" model="90,30,1"/>
    <property name="colour" channel="inline" abstraction="rgb" model="127,127,127"/>
  </part>
</model>
```

One of the properties in the example template above carries the name *super*. Its model references another template which is treated as super template of the corresponding *part* the property belongs to. With slight modifications on the property name *super*, which has to be unique among all properties of a part, it would even be possible to implement *Multiple Inheritance*. Dependency complications are not to be expected because all inheritance relationships are forgotten in runtime models.

Although the described inheritance mechanism was tested successfully in an older prototype application, it has not been implemented in CYBOL. None of the created example applications showed a need for it, nor did any of them promise more effective programming. The reuse of CYBOL templates is realised through composition only, that is fine-granular templates make up more coarse-grained ones. This counts for both, state- as well as logic models, since they are not bundled like in OOP. And polymorphism as effect does not have to be considered.

9.3.5 Container Mapping

State-of-the-art programming languages offer a number of different container types, partly based on each other through inheritance. Section 4.1.15 of this work identified *Container Inheritance* as one reason for falsified program results.

Chapter 7 then introduced a *Knowledge Schema* which represents each item as *Hierarchy* by default, the result being that different types of containers are not needed any longer. A possible unification of container types was already discussed in section 7.3.5. But how are the different kinds of container behaviour implemented in CYBOL? Table 9.1 gives an answer. As can be seen, CYBOL is able to represent many container types.

Classical Container Type	Realisation in CYBOL Knowledge Template
Tree	Hierarchical <i>whole-part</i> structure
Table	Like a Tree, as hierarchy consisting of rows which consist of columns
Map	Parts have a <i>name</i> (key) and a <i>model</i> (value)
List	Parts may have a <i>position</i> property
Vector	A <i>model</i> attribute may hold comma-separated values; an extra template holds a dynamically changeable number of parts
Array	Like a Vector; characters are interpreted as <i>string</i>

Table 9.1: Mapping Classical Containers to CYBOL

9.3.6 Hidden Patterns

There are a number of software patterns (section 4.2) that may not be obvious (hidden) at first sight, but have been considered in the design of the CYBOL language.

Most obviously, CYBOL knowledge templates follow the *Composite* pattern, in a simplified form. All templates represent a compound consisting of part templates, which leads to a tree-like structure. But this also means that related patterns (see section 7.2.1) like *Whole-Part* and *Wrapper* are representable by CYBOL knowledge templates. A template as whole wraps its parts.

Knowledge templates with similar granularity can be collected in one directory, in other words one common ontological level. Templates with smaller granularity, that is those that the more coarse-grained templates consist of, can be placed in another common layer and so forth. What comes out of it is a system of levels – one variant of the *Layers* pattern.

9.4 Comparison

Other projects and efforts have tried to craft languages improving the representation and expressiveness (in particular the semantics) of knowledge documents. Some of them are based on XML, just like CYBOL. The following subsections will dwell on the differences between two other languages and CYBOL.

9.4.1 RDF

Using the *Resource Description Framework* (RDF) described in section 4.5.4, a catalogue of products available at a certain domain *www.example.com* might be encoded as in the following example [347]:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:exterms="http://www.example.com/terms/">
  <rdf:Description rdf:about="http://www.example.com/2002/04/products#item10245">
    <rdf:type rdf:resource="http://www.example.com/terms/Tent"/>
    <exterms:model rdf:datatype="xsd:string">Overnighter</exterms:model>
    <exterms:sleeps rdf:datatype="xsd:integer">2</exterms:sleeps>
    <exterms:weight rdf:datatype="xsd:decimal">2.4</exterms:weight>
    <exterms:packedSize rdf:datatype="xsd:integer">784</exterms:packedSize>
  </rdf:Description>
  ... other product descriptions ...
</rdf:RDF>
```

This piece of code contains just one product description: that of *item10245*, which is typed as a *Tent*. It owns properties like its *model*, *sleeps*, *weight* and *packed_size*. The XML namespace declaration in line four specifies that the namespace *Uniform Resource Indicator reference* (URIref) *http://www.example.org/terms/* is to be associated with the *exterms*: prefix. URIrefs beginning with the string *http://www.example.org/terms/* are used for terms from the vocabulary defined by the example organization, *example.org*. The *ENTITY* declaration specified as part of the *DOCTYPE* declaration in the second line defines the entity *xsd* to be the string representing the namespace URIref for XML Schema datatypes. This declaration allows the full namespace URIref to be abbreviated elsewhere in the XML document by the entity reference *&xsd;*, so that data types like *string* or *integer* may be written as *&xsd;string* and *&xsd;integer*, respectively.

The same product catalogue example written in CYBOL would look like this:

```
<?xml version="1.0"?>
<model>
  <part name="item10245" channel="http" abstraction="cybol"
    model="www.example.com/2002/04/products#item10245">
    <property name="type" channel="http" abstraction="rdf" model="www.example.com/terms/Tent"/>
    <property name="model" channel="inline" abstraction="string" model="overnighter"/>
    <property name="sleeps" channel="inline" abstraction="integer" model="2"/>
    <property name="weight" channel="inline" abstraction="decimal" model="2.4"/>
  </part>
</model>
```

```

    <property name="packed_size" channel="inline" abstraction="integer" model="784"/>
  </part>
  ... other products and their properties ...
</model>

```

One can find the product identifiable by the name *item10245*, as one part of the catalogue, as well as its properties representing meta (descriptive) information. The product's model has the abstraction *cybol* which means that the corresponding resource is available in CYBOL format. The resource's channel is *http* which means that it has to be read using that protocol and communication mechanism. Other abstractions are possible, of course. The *type* property in the example is available in RDF format, as indicated by its abstraction attribute.

The meaning of the single XML attributes was explained in previous sections. Up to now, there was no need to apply domains for building attribute- or tag names. CYBOL's tags, that is their number as well as their names, are fixed. The same counts for its attributes. What is changeable, are attribute *values* alone.

While RDF's main focus is on providing the means for making *descriptive* (meta) statements *about* a subject, CYBOL provides these meta information together with structural (whole-part) information, encoded in form of a double hierarchy (section 7.3.3).

9.4.2 OWL

Since the *Web Ontology Language* (OWL) described in section 4.5.4 is a vocabulary extension to RDF, the points explained in the context of RDF before do count for OWL as well. Further considerations are done using the following OWL code example representing an *incomplete* extract of a description of *Wine* (potable liquid) [346]:

```

<rdf:RDF ...
  <owl:Class rdf:ID="Wine">
    <rdfs:subClassOf rdf:resource="#food;PotableLiquid"/>
    <rdfs:label xml:lang="en">wine</rdfs:label>
    <rdfs:label xml:lang="fr">vin</rdfs:label>
    ...
  </owl:Class>
  ...
  <owl:Class rdf:ID="WineColour">
    <rdfs:subClassOf rdf:resource="#WineDescriptor"/>
    <owl:oneOf rdf:parseType="Collection">

```

```

        <owl:Thing rdf:about="#White"/>
        <owl:Thing rdf:about="#Rose"/>
        <owl:Thing rdf:about="#Red"/>
    </owl:oneOf>
</owl:Class>
...
<owl:Class rdf:ID="Vintage">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#vintageOf"/>
            <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
                1
            </owl:minCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
</rdf:RDF>

```

A class *Wine* inheriting from a super class *PotableLiquid* is defined first. Labels in several languages are given. Further, two classes *WineColour* and *Vintage* are defined. Properties referring to the *WineColour* class may take on one out of a collection of three colour values. The *Vintage* class has the restriction that at least one property *vintageOf* must exist. If the *Wine* class or one of its sub classes is connected to the *WineColour* or *Vintage* class (what is not shown here to keep the code example simple), it has to consider their possible restrictions.

An approximated CYBOL pendant to the OWL example above is shown following:

```

<model>
  <part name="wine" channel="http" abstraction="cybol" model="example">
    <property name="super" channel="file" abstraction="cybol" model="food/potable.cybol"/>
    <property name="label" channel="file" abstraction="cybol" model="terminology/wine.cybol"/>
    <property name="colour" channel="inline" abstraction="string" model="">
      <constraint name="choice" channel="inline" abstraction="string" model="red,white"/>
    </property>
    <property name="vintage" channel="inline" abstraction="integer" model="">
      <constraint name="requirement" channel="inline" abstraction="boolean" model="true"/>
    </property>
  </part>
</model>

```


Wine as one part of a greater model (such as a catalogue) is described. Its properties are equivalent to those in the OWL example above. The values of the *colour* property are constrained to a choice of two colours, and the *vintage* property is required to exist, i.e. it is not deletable.

9.5 Tool Support

When proposing a new theory of computing, or a new programming language, it is common to provide a suitable *Integrated Development Environment* (IDE) supporting the application of that theory or language. If not the tools themselves, a recommendation for how they may look like should be given at least. This is what the following sections try to achieve.

9.5.1 Template Editor

CYBOL applications can be written in an XML-conform way. The use of standard XML tools to edit and validate CYBOL knowledge templates, at design time, is therefore possible. An exception are serialised runtime CYBOL models, possibly made persistent in form of files or in a *Database* (DB), for which XML conformity has to be given up due to additional markup tokens, as explained in section 9.3.1.

Due to the fixed structure of CYBOL knowledge templates (four XML tags, four XML attributes), more convenient- than standard XML editors shall be providable. Figure 9.9 shows an editor proposal supporting both, the *Whole-Part*- as well as the *Meta Hierarchy* of CYBOL. There are three characters indicating the action that a click on a tree node would evoke:

- + Open the whole-part hierarchy
- & Open the meta hierarchy (properties and constraints)
- Close the hierarchy

The displayed template represents a graphical *dialogue* with its *title*, *menubar*, *toolbar*, *panel* and *status bar*. The opened panel node shows its parts, namely *button* and *label*. The opened status bar node, on the other hand, shows its properties *size* and *colour*, and additionally the *size*'s *minimum* constraint. The attribute values of a selected node would be editable in a table like the one shown on the right-hand side of the figure.

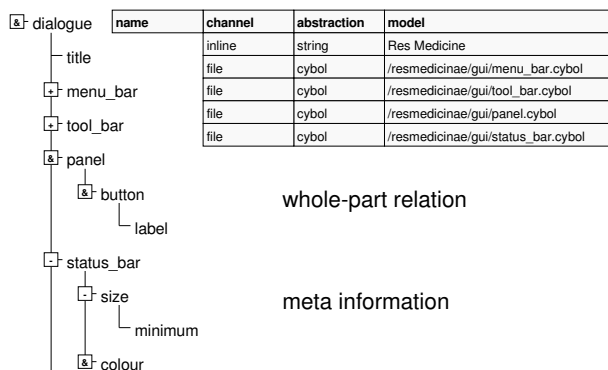


Figure 9.9: CYBOL Editor Supporting Double Hierarchies by Triple Choice

9.5.2 Knowledge Designer

Section 4.4.4 classified diagrams of the *Unified Modeling Language* (UML) notation into *Structure*, *Behaviour* and *Interaction*. With interaction- being a subset of behaviour diagrams, there are actually just *two* main categories for UML diagram classification: *Structure* and *Behaviour*. Since the idea underlying *this* work is to look at systems from two perspectives (section 5.3): *Statics* and *Dynamics*, whereby the former gets split up into two further perspectives: *States* and *Logic*, the question arises whether or not UML diagrams could be categorised accordingly? The answer is: *Not quite*. There are a number of aspects that have to be considered:

- UML classes bundle state- and logic aspects (attributes and methods). A CsD does not only express the relations between attributes, but also those between methods. This fact makes it impossible to sort that diagram into just one of the categories: state or logic. Likewise does an SD, to take a second example, not just display the order of message calls, but also their bundling with objects. CYBOL templates, on the other hand, strictly separate state- and logic knowledge.
- Classes in a CsD are linked network-like and may have bidirectional relations. Composition (recursion) as concept is missing in the class element of the UML meta model. The CYBOP knowledge schema is innately hierarchical and uses solely unidirectional

relations.

- UML objects (instances) know from which class (type) they stem from. Not at least, this is necessary for mechanisms like polymorphism (based on runtime inheritance) to work. CYBOL models know nothing about the original template they were initialised with; any links to it are lost.

However, an attempt will now be made to categorise the UML diagrams accordingly:

- *Statics (States)*: CsD
- *Statics (Logic)*: SMD, AD, SD, TiD, CoD, IOD, UCD
- *Dynamics*: ObD, CSD
- *Others*: CmD, PD, DD

All diagrams formerly belonging to either *Behaviour* or *Interaction*, are now summed up in the *Statics (Logic)* category. Former *Structure* diagrams are split up into the one describing *Statics (States)* and those illustrating *Dynamics* (runtime aspects). Some diagrams dealing with issues like packaging or distribution are put into an extra category called *Others*.

Because of the different programming philosophy behind CYBOP, standard UML diagrams cannot be used unalteredly for the design of CYBOL applications. Some of them, however, could be quite useful, when adapted a bit. The *Importance* column in table 4.1 indicated that not all diagram types are really needed to effectively design a system. For creating CYBOL applications, the following four can be considered sufficient. They model the structure of:

1. *Template Diagram* (TD): one design-time template (hierarchical, ontological concept), with purely unidirectional relations; does not illustrate relations between different concepts, as these are only established by logic models at runtime; could look like CsD or a tree, only that a template may not only represent states, but also logic (algorithms, workflows) (figure 9.10)
2. *Model Diagram* (MD): the runtime model tree; comparable to ObD, but a simple tree with named nodes would suffice; is important because input/ output parameters of operations are given as dot-separated paths to runtime knowledge tree models (figure 9.11)
3. *Organisation Diagram* (OD): template directories; could look like CmD or PD or a simple tree (figure 9.12)

4. *Communication Diagram (CD)*: a network of communicating systems, which may run on the same or on different physical machines (nodes); could look like DD; not to be mixed up with UML CoD (figure 9.13)

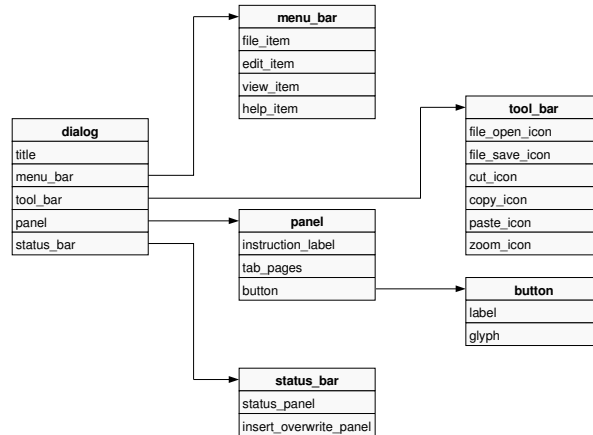


Figure 9.10: CYBOL Template Diagram (TD) Proposal

As said above, the four diagrams may look similar to their corresponding UML pendant. For demonstration reasons, one possible proposal is given for each diagram type. The TD (figure 9.10) illustrates the same graphical dialogue that was shown in the *Template Editor* (figure 9.9), in the previous section. The diagram looks pretty similar to a UML CsD. Attributes and methods are not bundled in one concept though, and inheritance does not exist. Associations are drawn if a concept links to an external concept which may reside in another file (like the *menu_bar*), for example. If a part (like the *title*) is hold inline in the concept, on the other hand, an association is not displayed. Upon clicking on a part in a concept box, a dialogue opens up that allows the entry of meta data like the part's channel, abstraction, model and further properties (details).

The MD (figure 9.11) displays the runtime models that were instantiated with knowledge templates providing the initial values. Again, the parts of the graphical dialogue of figure 9.9 are used in it.

The OD (figure 9.12) shows packages into which CYBOL knowledge templates may be organised. Packages do normally correspond to directories on file system level. The figure

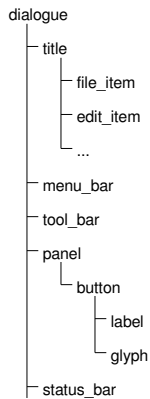


Figure 9.11: CYBOL Model Diagram (MD) Proposal

contains a *domain* package consisting of two sub packages, one containing knowledge templates for *administrative* patient data and the other holding templates for *clinical* data of a patient. Also, there is a *User Interface* (UI) package containing three sub packages, for: *Textual UI* (TUI), *Graphical UI* (GUI) and *Web UI* (WUI). Both, *domain*- as well as *user_interface* packages may be accessed from the operations residing in the *logic* package.

The CD (figure 9.13), finally, shows a number of independent systems communicating with each other. An *Electronic Health Record* (EHR) manager application may be found in the center of the figure. Patients communicate with it using a WUI; nurses using a GUI and doctors using a TUI (for better performance). A patient gets identified by asking a *person_identification* service. Documents may be exchanged with a *hospital_system* and images with a special *image_storage* system.

Besides these essential diagrams, additional ones may be used, of course. UML diagrams like the AD, SD or TiD assist in modelling the flow of actions, that is sequences of logic operations over time. Their ability to refine actions in another level of granularity is of special interest. When removing the objects bundled with method calls, they are well suitable for CYBOL. They use different graphical elements, but in the end would store their knowledge in the same templates. The transitions between different states of the runtime knowledge tree over time are what the SMD wants to display. It may well be used with CYBOL. But

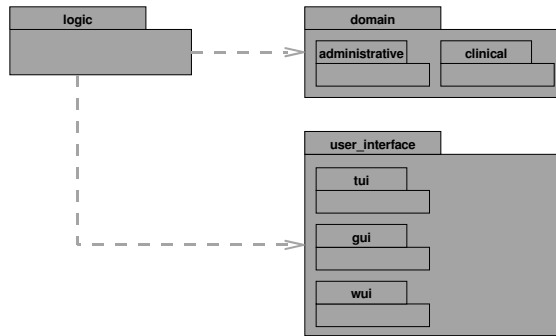


Figure 9.12: CYBOL Organisation Diagram (OD) Proposal

not only an application's behaviour over time is of interest, the positions and expansions of its elements in space are as well important. This mostly affects the design of user interfaces, graphical or textual. Designers for these are therefore added to the list of useful diagrams. The usefulness of feature model diagrams (section 4.4.4) for expressing constraints that branches of the knowledge tree impose on each other could not be investigated in this work, as this would break its frame.

Since CYBOL files contain all knowledge that is needed to define a complete application system, the generation or parsing of classical source code is not needed anymore, as chapter 12 will mention again. Therefore, CYBOL knowledge design tools do only have to provide scanner- and generator functionality for CYBOL (in order to formalise the knowledge designed as semi-formal diagrams), but not for implementation languages.

9.5.3 Model Viewer

Finally, it would be very helpful to have a tool displaying not just planned-, but *real* runtime statuses of the knowledge model tree, in other words a *live* memory snapshot displayed in a meaningful, hierarchical form. Such a *Model Viewer*, as it might be called, would not only help in debugging applications, but could be extended towards a runtime *Model Editor*, allowing to move whole knowledge tree branches from one parent node to another.

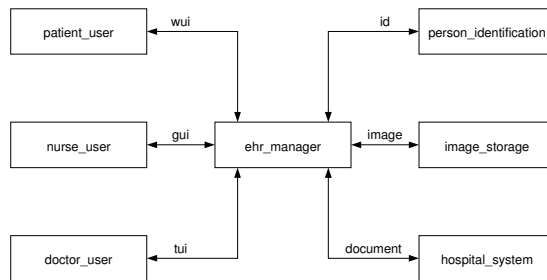


Figure 9.13: CYBOL Communication Diagram (CD) Proposal

To what concerns the usage of existing UML tools for modelling CYBOL applications, it has to be said that they are most likely not able to support CYBOP models, nor can they be easily adapted. Even though the diagram specifications may be similar, the underlying programming principles and model repositories are just too different. It might be possible, however, to create interfaces partly translating CYBOL-encoded (XML) models into UML models, possibly using the *XML Metadata Interchange* (XMI) standard or the like. It will presumably be easier to translate CYBOL- into UML models, than vice versa, because CYBOL is very expressive. While CYBOL knowledge templates distinguish between whole-part- and meta elements, for example, UML models do not. Another point is that CYBOL holds state- and logic knowledge in separate templates, that would have to be merged into common classes when being translated into UML. But the investigation and evaluation of further details concerning the tool support is left for future works.

10 Cybernetics Oriented Interpreter

*Two Things are needed to do our Work:
tireless Endurance and the Willingness to throw something,
that has cost much Time and Effort, away again.*

ALBERT EINSTEIN

The previous chapter described how the CYBOP concepts introduced in part II of this work can be implemented in a formal language carrying the abbreviated name CYBOL. The now still missing piece is a program able to dynamically handle static CYBOL sources, as mentioned in chapter 6. This chapter therefore describes the *Cybernetics Oriented Interpreter* (CYBOI) [256] which can do just that.

10.1 Architecture

The following sections dwell on how CYBOI fits into a general computer architecture, and then explain its inner structure and software patterns used.

10.1.1 Overall Placement

Considering an overall computer system architecture, *CYBOI* is situated between the application knowledge existing in form of *CYBOL* templates and the *Hardware* controlled by an *Operating System* (OS), as was shown in figure 6.7. *CYBOI* can thus also be called a *Knowledge-Hardware-Interface* (synonymous with *Mind-Brain-Interface*).

There are analogies to other systems run by language interpretation. Table 10.1 shows those between the *Java*- and *CYBOP* world. Both are based on a programming theory, have a

Criterion	Java World	CYBOP World
Theory	OOP in Java	CYBOP
Language	Java	CYBOL
Interpreter	Java VM	CYBOI

Table 10.1: Analogies between the Java- and CYBOP World

language and interpreter. A theoretical model of a computer hardware- or -software system may be called an *Abstract Computer* or *Abstract Machine* [60]. If being implemented as software simulation, or if containing an interpreter, it is called a *Virtual Machine* (VM). Kernighan and Pike write in their book *Practice of Programming* [173]:

Virtual machines are a wonderful, old idea, that latterly, through Java and the *Java Virtual Machine* (JVM), came into fashion again. They are a simple possibility to gain portable and efficient program code, which can be written in a higher programming language.

In that sense, CYBOI is certainly a VM. It provides low-level, platform-dependent system functionality, close to the OS, together with a unified knowledge schema (chapter 7) which allows CYBOL applications to be truly portable, well extensible and easier to program, because developers need to concentrate on domain knowledge only. Since CYBOI interprets CYBOL sources *live* at system runtime, without the need for previous compilation (as in Java), changes to CYBOL sources get into effect right away, without restarting the system.

The use of an OS, however, has to be seen as temporary workaround. One future aim is to remove all OS dependencies by stepwise integrating hardware device driving functionality and other OS concepts into CYBOI (chapter 13).

10.1.2 Inner Structure

To what concerns its inner architecture, there are two basic structures underlying CYBOI:

1. *Knowledge Container*: An array-based structure usable for storing static knowledge in form of primitive- and compound models, and capable of representing a map, collection, list and tree
2. *Signal Checker*: A loop-based structure usable for dynamically reading signals from a queue, and capable of processing them after their priority, in a special handler

All modules, into which CYBOI is subdivided, are built around these two core structures. Having read chapter 8 demonstrating the existence of state- and logic knowledge, one might argue that there should be two knowledge containers, one for each kind. But because knowledge models may be placed not only in space or time, but possibly other dimensions, too (like mass, for the weights in an artificial neural network), the prototype emerging from this work stores state- and logic-, as well as any other models in one and the same knowledge tree.

Not unlike John von Neumann's model of a computing machine [308], which distinguishes *Memory*, *Control Unit*, *Arithmetic Logic Unit* (ALU) and *Input/ Output* (i/o), CYBOI's modules are grouped into four architectural parts, as illustrated in figure 10.1. These have the following functionality:

- *Memoriser*: data creation, -destruction and -access (after Neumann, it contains not only data, but also the operations that are applied to them)
- *Controller*: lifecycle management, signal handling, i/o filters
- *Applicator*: operation application (comparison, logic, arithmetic and more)
- *Globals*: basic constants and variables, as well as a logger

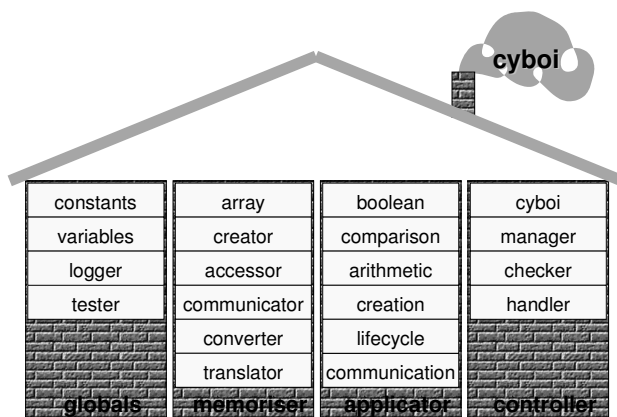


Figure 10.1: CYBOI Architecture consisting of Four Parts

The i/o data handling is not separated out here (as opposed to von Neumann's model); it is managed by the controller modules. The i/o data themselves, representing states, are

stored in memory. Global constants and variables are necessary additions. More details on the modules' functionality are given in section 10.2.

10.1.3 Pattern Merger

A variety of software patterns (section 4.2) can be found when inspecting CYBOI's architecture. Most of them, especially those relying on *Object Oriented* (OO) principles, are used in an *adapted* form, as described following.

Firstly, there are those that can be summed up under the umbrella term *Translator Patterns*: *Model View Controller* (MVC), *Data Mapper* and *Data Transfer Object* (DTO). As section 8.2 tried to show, they all contain two state models, one logic model and a controlling unit, which is why it is possible to unify- and place them into one common architecture. CYBOI represents the unit controlling all action, on a low system level. It stores state- as well as logic models in one common knowledge tree, and uses the rules encoded in logic models to translate state models into each other.

Secondly, there is the *Pipes and Filters* pattern which CYBOI uses not only to instantiate knowledge templates, but also for system communication. An input (i/p) state (like a persistent, serialised knowledge template) runs through a cascade of filters, namely *Creator*, *Communicator*, *Converter* and *Translator*, before it is processed (as transient knowledge model) inside the system, to be finally forwarded in opposite direction through the same filters, resulting in an output (o/p) state.

Thirdly, CYBOI acts as *Microkernel* and *Broker*, at the same time. It calls special threads (internal servers) managing data input/ output (i/o), and has the capability to communicate with remote systems (external servers), for data transfer. The actual impulse for communication comes from a passive knowledge model (adapter) that is actively processed by CYBOI. Since that impulse is *not* a direct method call, but either a *send*- or *receive* operation with varying parameters, special *Proxy* models are not needed anymore. CYBOI may act as client and server, at the same time, which enables the applications running within it to act as *Peer to Peer* (P2P) systems (section 3.8). It incorporates a signal (event) loop (like the broker) and handles low-level system (socket) communication (like the bridge).

The fact that future versions of CYBOI will be able to interpret CYBOL knowledge templates containing *Graphical User Interface* (GUI) descriptions, makes it a *GUI Renderer*. The task of a renderer is to translate GUI models into hardware-understandable function

calls and protocols. In the case of CYBOI, the graphical environment supported first will be the *X Window System* (X) *XFree86* [58] variant. The step towards rendering models given in *Hyper Text Markup Language* (HTML) format is not far then, so that CYBOI may act not only as web server, but also as web browser. All that, together with further additions, will make it virtually an *All-Rounder* system.

The usage of simplified forms of patterns like *Composite* (inheritance omitted), *Whole-Part*, *Wrapper* or *Layers*, for knowledge storage, was already mentioned in section 9.3.6.

Unfavourable patterns as mentioned in section 7.2.1 (those with *global access* or *bidirectional dependencies*) were avoided.

Finally, the merged appearance of patterns in CYBOI (and CYBOL for that matter) brings software development one step closer to *pattern-less* application programming. Application developers are freed from the burden of repeatedly figuring out suitable patterns, and enabled to concentrate on modelling pure domain knowledge, based on the concept of *Hierarchy*, instead.

10.1.4 Kernel Concepts

Although certainly not qualifying as *Operating System* (OS), CYBOI uses some similar concepts. However, it is not easy to assign CYBOI to one of the four broad categories of OS kernels, which the Wikipedia Encyclopedia [60] describes as follows:

1. *Monolithic Kernels*: providing rich and powerful abstractions of the underlying hardware
2. *Microkernels*: providing a small set of simple hardware abstractions and using applications called servers to provide more functionality
3. *Hybrid Kernels*: being much like pure microkernels, except that they include some additional code in kernelspace to increase performance
4. *Exokernels*: providing minimal abstractions but allowing the use of library operating systems to provide more functionality via direct or nearly direct access to hardware

Just like a *Monolithic Kernel*, CYBOI encapsulates low-level functionality like memory management and provides a high-level virtual interface (operations), over which CYBOL applications may access it. Although the processing of signals happens in its main process, CYBOI has to rely on a few communication services, running in their own threads, and

control their data exchange. This is (roughly) comparable to the *Microkernel* design which was mentioned as pattern in section 10.1.3 before. But these services sharing a common address space (*Internals Memory* and *Signal Memory*) with the CYBOI kernel, actually makes CYBOI a *Hybrid Kernel*. The only existing data in user address space (*Knowledge Memory*) are knowledge models that have been created from CYBOL knowledge templates. The kernel category coming closest to CYBOI's design, however, is the *Exokernel*. This is because CYBOI, though serving as *Hardware Abstraction Layer* (HAL) to CYBOL applications (what is actually known from classic monolithic- and microkernels), also has a central *Signal Checker* control loop calling subroutines managing a part of the hardware or software, which, after [60], were one of the simplest methods of creating an exokernel.

The above-mentioned services provide *Input/ Output* (i/o) functionality to CYBOI so that it can communicate (*virtual world*) ideas with other (human or technical) systems, across (*real world*) hardware. The services may be configured, started up, interrupted and shutdown via CYBOL operations. Similar to the *Self Awareness* of human systems (mentioned in section 8.1.4), a CYBOL application has to know about available i/o services, in order to actually use them. This configuration information may be stored in CYBOL files as well.

Another thought turns around the *Process* concept [304], which is used to share computing time of the *Central Processing Unit* (CPU) as well as resource space in *Random Access Memory* (RAM) between applications. One then says: *Every application runs in a separate process*. The example environment in chapter 3 contained many different kinds of intercommunicating systems, not all of which have to run on a separate physical machine, but surely in a separate process, if on one-and-the-same machine. That way, one machine may host multiple application systems.

However, the existence of many program processes running concurrently in an OS holds conflicts. It necessitates ways for *Inter-Process Communication* (IPC), that assure the integrity of data in memory and avoid deadlocks (blocking of the system). Common IPC methods include [310, 304]: *Pipes* and *Named Pipes*, *Message Queueing*, *Semaphores*, *Shared Memory* and *Sockets*.

A different approach was chosen in CYBOI: It is the only process running. CYBOL applications reside as separate knowledge models in the *Knowledge Memory* (user address space), and CYBOI controls them all. This is the exact opposite of running one process per application. However, it is unclear, at first, how multiple applications running in parallel receive their corresponding signals. CYBOI needs to evaluate incoming signals and then call the right logic of the right application. But how to do that?

The application to which a signal is sent can be identified, depending on the communication mechanism used. A *keyboard_pressed* or *mouse_clicked* event, for example, always references the top-most window in a GUI environment. Menu-, toolbar- and other buttons of the top-most window in turn reference a logic (algorithm) whose dot-separated name starts with that of the application. Within CYBOI, applications have to have a unique name, of course, so that signals can be addressed correctly, to the right receiver. Once a logic routine is identified, it can be sent as new signal to be processed by the signal loop. Another example would be signals arriving over network. Also here, applications can be identified, for example by a special *Port* that was assigned to them beforehand. A remote call references a specific logic by name so that it can be processed locally. Equally named local procedures are distinguished by the application name identified before. Within an application, logic names have to be unique, of course.

10.1.5 Security

Berin Loritsch of the former Apache Avalon Project [17] writes that system *Security* has three distinct concerns, of which *Encryption* is only a part:

1. *Authentication*: authoritative validation of the identity of a party, such as a software component
2. *Authorisation*: deciding what access a component has to system resources
3. *Architecture*: usage of a proper, secure architecture

Since this chapter is about CYBOI's architecture, what interests the most here is point number three. But how to ensure a secure architecture? The avoidance of global data access and bidirectional dependencies is clearly a requirement (section 7.2.2), which CYBOI accomplishes through disregard of the corresponding patterns (section 10.1.3). Any kind of application knowledge resides in its *Knowledge Memory*, whose tree structure may be navigated along well-defined, unidirectional paths.

One may wonder how address spaces of the different applications are protected, so that one application may not access another one's models? Traditionally, the process concept assures that separation, but with CYBOI being the only process, and all applications being part of it, another solution needs to be applied. The exact mechanism to solve this in CYBOI yet has to be determined. However, since all signals have to pass the same, central signal loop, they all can be checked for permissions, before being processed, or filtered out, if necessary.

It would be imaginable and not difficult, to attach an application name as a signal's origin, or a unique passphrase as additional meta information to a signal memory's signals, that own an *Identifier* (ID) anyway. The signal loop or signal handler could then decide whether or not to send a signal to a specific application.

Attached meta information of that kind would be comparable to a concept called *Capability*, which is known from *Secure Computing*, a subfield of *Security Engineering* [60]. It is the alternative to *Access Control Lists* (ACL), another means of enforcing firstly: *Mandatory Access Control*, and secondly: *Privilege Separation* (where an entity has only the privileges that are needed for its function). Wikipedia [60] writes on this:

Capabilities (also known as *Key*) achieve their objective of improving system security by being used in place of *Plain References*. A plain reference (for example, a path name) uniquely identifies an object, but does not specify which *Access Rights* are appropriate for that object and the user program which holds that reference. Consequently, any attempt to access the referenced object must be validated by the operating system, typically via the use of an *Access Control List* (ACL). In contrast, in a pure capability-based system, the mere fact that a user program possesses that capability entitles it to use the referenced object in accordance with the rights that are specified by that capability. In theory, a pure capability-based system removes the need for any ACL or similar mechanism, by giving all entities all and only the capabilities they will actually need.

With almost all important *Operating Systems* (OS) still using ACL, for various historical reasons, CYBOI could be seen as chance to implement a pure capability-based system. Since it concentrates all knowledge in one container realised as tree structure, and processes all signals in one central loop, security by design is given, and security checks of different shade can be easily applied to all knowledge models and all signals. These checks of dynamic runtime models are a necessary supplement to the checks of static CYBOL template files. Traditional OS do the latter via *File Descriptors* (also called *File Handles*), which are facilities very similar, but not equal to capabilities.

In the opinion of Wikipedia [60], one main reason why the benefits of a pure capability-based system could not be realised in a traditional OS environment, were the fact that entities which might hold capabilities (such as processes and files) cannot be made persistent in such a way that maintains the integrity of the secure information that a capability represents. The OS could not trust a user program to read back a capability and not tamper with the

object reference or the access rights. Consequently, when a program wished to regain access to an object that is referenced on disk, the OS had to have some way of validating that access request, and an ACL or similar mechanism were mandated.

Orthogonally Persistent OS like the *Flex Machine* and its successor *Ten15*, as [60] writes, were a novel approach to solving this problem. They maintained the integrity and security of the capabilities contained within all storage, both volatile and nonvolatile (dynamic and static), at all times. Further, such OS were responsible for storage allocation, deallocation and garbage collection, which immediately precluded a whole class of errors arising from the misuse (deliberate or accidental) of pointers. Two other features were:

- *Tagged, Write-Once Filestore*, which allows arbitrary code and data structures to be written and retrieved transparently, without recourse to external encodings; data could thus be passed safely from program to program
- *Remote Capabilities*, which allow data and procedures on other machines to be accessed over a network connection, again without the application program being involved in external encodings of data, parameters or result values

This reads like a description of CYBOI, which is able to parse/ serialise all knowledge from/ to CYBOL sources (e.g. files), and to handle low-level storage- and communication mechanisms. More in section 10.2.6. If, in this manner, user programs (CYBOL applications) were relieved of these responsibilities, there would be no need to trust them to reproduce only legal capabilities, nor to validate requests for access using an ACL-like mechanism. However, these were just some thoughts on how to bring yet more security into CYBOI's architecture. The details will have to be figured out in future works (chapter 13).

10.2 Functionality in Detail

CYBOI's architecture is based on three main parts, as introduced by figure 10.1 before: *Controller*, *Applicator* and *Memoriser*. (The *Globals* package is neglectable for the following explanations, since it contains static constants and variables that are *omnipresent*.) They appear again in figure 10.2 which shows the *Dependencies* between them. Additionally, the *Controller* modules and their *Control Flow* is illustrated. Starting from the *cyboi* module, the following subsections will demonstrate how CYBOI functions internally, along the flow of control touching the modules: *manager*, *checker* and *handler*. After that, the execution of

operations in the *Applicator* as well as the creation and transition of data in the *Memoriser* are described.

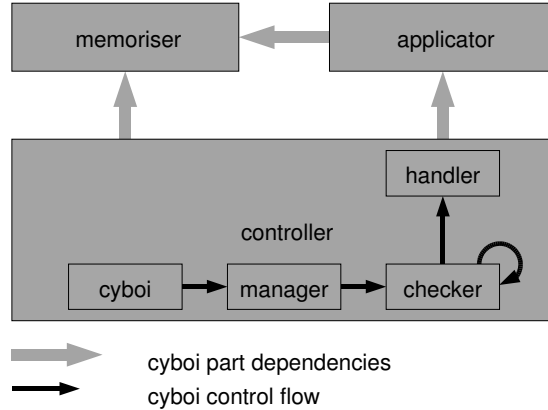


Figure 10.2: CYBOI Part Dependencies and Control Flow

10.2.1 Process Launching

As every other C-, C++- or Java program, CYBOI has a *main* procedure (*cyboi* module) serving as entry point for its process to run. It triggers the system lifecycle (in the meaning of startup and shutdown of a system process). After having initialised global variables and having read the command line parameter, the rest of the system is started up by the *manage* procedure (*manager* module).

10.2.2 Lifecycle Management

To the startup routine belong the creation of the three containers: knowledge memory, signal memory and internals memory, and the creation of a startup model which is placed as first signal into the signal memory. Additional meta information given are the signal's model, its kind of abstraction and priority.

Typical synonyms for *Signal* are *Event* or *Action* – and even an *Operating System* (OS) *Interrupt* is some form of signal, only on a lower system level, closer to hardware. In CYBOI,

a signal is simply a reference to a logic model, which may be either a composed algorithm, or a primitive operation.

With the startup signal being placed in the signal memory, the system enters the *check* procedure (*checker* module). On shutdown, the system runs through similar procedures in opposite direction, only that then startup signal, memories and global variables are destroyed.

10.2.3 Signal Checking

The *check* procedure consists of an endless loop continuously checking for signals residing in signal memory. It provides the dynamics and – so to say – keeps the system *alive*. Of all queued signals, the one with highest priority is retrieved first and forwarded to the *handle* procedure (*handler* module).

This principle can be observed not only in operating-, but many other kinds of systems. *Servers* run an endless loop waiting for (network) *Client* requests. Applications often use signalling mechanisms provided by a framework, that handles keyboard press- or mouse click signals stemming from a *Graphical User Interface* (GUI). However, as opposed to the event handling of such frameworks which relies on bidirectional dependencies since child components have to register as listener at their parent, a top-level signal checker loop forwards all events in a unidirectional manner to interested system parts. It is worth noting that signals may also be produced internally, as follow-ups, by other signals.

After having been processed, the signal gets removed from the signal memory. Once an *exit* signal occurs, the shutdown flag is set, so that the signal checking loop can be left – and the system be shutdown.

10.2.4 Signal Handling

Depending on the signal model's kind of abstraction, two different signal handling procedures may be called: *handle_compound* or *handle_operation* (both in the *handler* module). While the former breaks down composed signals (algorithms) into basic operations, the latter executes primitive signals (operations) directly, in form of low-level instructions, which may go down to direct calls of the instruction set of the *Central Processing Unit* (CPU).

Actual knowledge model changes, in other words the application of well-defined *Logic*- to *State* models, is done by primitive operations only.

10.2.5 Operation Execution

Each low-level operation has its own module, belonging to the *Applicator* part of CYBOI. An addition operation is executed in the *add* module, a comparison operation in the *compare* module, a creation operation in the *create* module, and so forth. Operations exist for several purposes, some of which are listed, together with an example operation, following:

- program flow (*loop*)
- boolean logic (*and*)
- comparison (*equals*)
- arithmetics (*add*)
- service control (*startup*)
- memory management (*create*)

10.2.6 Model Transition

The creation of transient knowledge models (to be kept in memory, at runtime) from persistent knowledge templates (given in form of CYBOL sources) is not a trivial thing. It is a mechanism consisting of a cascade of model transitions, comparable to the *Information Processing Model* of cognitive psychology (section 6.1.5). One may imagine this as a state changing its appearance, while *wandering* through the system. The same mechanism is applied when handling communication data (figure 10.3). Because CYBOI's architecture is easily extensible with various modules, such as *import/ export* (i/e) filters for different kinds of communication, it may act as universal data converter. All corresponding modules belong to the *Memoriser* part of CYBOI.

As opposed to the knowledge acquirement in *Artificial Neural Networks* (ANN), the knowledge in CYBOP systems is not learned, but *injected* by reading from external knowledge sources, which can be manipulated in a flexible manner any time. The difference to standard applications is that these hard-wire their knowledge within the system.

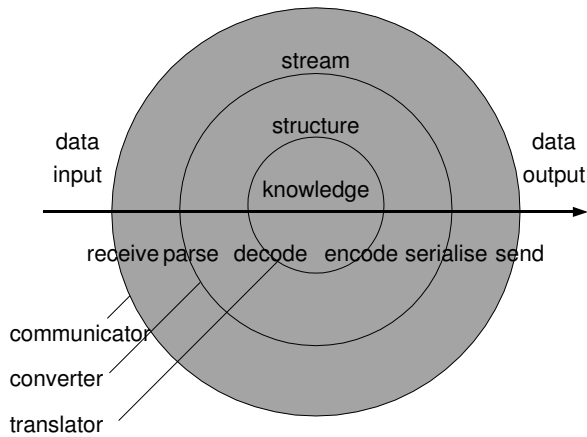


Figure 10.3: Input-to-Output Model Transition

A data input (i/p), after having been processed by a *receive* procedure (*communicator* modules), results in a stream – a data array in memory. A *parse* procedure (*converter* modules), depending on the kind of abstraction of the data, then builds a structure out of this simple array. The structure, finally, passes a *decode* procedure (*translator* modules) which creates a knowledge model.

Whenever an application running within CYBOI wants to send data, may it be to another system or for making them persistent on a local *Hard Disk Drive* (HDD), translator, converter and communicator have to be crossed in the opposite direction. Because a running application system is a tree of knowledge models allocating space in memory, parts of that tree can be *serialised* easily. It has to be mentioned, though, that slight changes of the XML format are necessary to achieve this: The usual quotation marks used to delimit XML attribute values have to be replaced with differing *begin* and *end* characters. This feature is an open issue which the current version of CYBOI does not provide yet.

10.2.7 Data Creation

The functionality of CYBOI as system is built around the manipulation of states in memory. The question how states, representing data, are stored is therefore of great importance. Besides containers like the *Internals Memory*, *Knowledge Memory* and *Signal Memory*,

belonging to its infrastructure, CYBOI uses special structures encapsulating primitive data. Not only types like *Character*, *Integer*, *Float* or *String* are wrapped this way, also *Operations* are. Any compositions of these are stored as *Compound*.

Encapsulated primitives have the advantage of being forwardable as reference (memory address pointer), instead of as copy. This ensures that redundant data are avoided and states of manipulated primitives are not lost. *Logic* operations are stored in form of a string indicating their name. Necessary references to input/ output (i/o) *State* models need to be provided as meta information, by the compound model surrounding the operation.

The *Compound*, as most important CYBOI structure capable of representing state- as well as logic knowledge, and capable of emulating a map, collection, list and tree, deserves closer inspection. Essentially, it is a container able to recursively reference instances of its own, thus spanning up a tree of parent nodes (*Whole* models) that may have child nodes (*Part* models). In fulfillment of the requirements of the knowledge schema introduced in section 7.3, a compound consists of a combination of many arrays containing a part's:

- *Name*: serving as unique *Identifier* (ID)
- *Model*: the actual contents (may be a part node)
- *Abstraction*: the kind of abstraction (type) of the model
- *Details*: further meta information (properties, constraints)

One may call the *Compound* a *multi-dimensional* container but it is probably easier described as large table with many columns, whereby the values of one row describe exactly one part model, and thus belong together.

The previously mentioned *Knowledge Memory* may be seen as huge tree consisting of compound- as well as primitive models. Its root is always a *Compound* – it, so to say, concentrates all knowledge in just one point, the single concepts being branches of it.

The essential procedures for managing data in memory are *create* and *destroy* (*creator* modules). Three additional kinds of procedures are provided for compound- and other container-like structures: *set*, *get* and *remove* (*accessor* modules). It should be noted at this point that CYBOL applications have no direct access to these procedures, so that *wild* memory allocation is not possible. A knowledge model can only be created using the corresponding CYBOL operation *create_part*.

10.3 Implementation

This section describes some issues that arose during implementation, and how they are addressed in CYBOI.

10.3.1 Simplified C

An early version of CYBOI was implemented in the *Java* programming language. Since, over time, its functionality was reduced to pure system control, by moving application-specific features to CYBOL, there was no longer a need for an *Object Oriented Programming* (OOP) language, which Java is. The OOP overhead caused by concepts like *Inheritance* inevitably results in lower performance, as compared to *Structured and Procedural Programming* (SPP) languages. Low-level system programming, close to hardware, focuses on fast data processing. OOP concepts would only disturb here. A later (the current) version of CYBOI was therefore rewritten in the slimmer *C* programming language.

It would, of course, be possible to implement CYBOI in other languages, too. Candidates could be *C++* or the increasingly popular *Python*. Both are OOP languages having similar dis-/advantages like *Java*. However, different CYBOI implementations are possible and as long as the CYBOL format gets interpreted correctly, different languages, frameworks and libraries can be used.

But also C (as other SPP languages) contains a number of unnecessary, redundant constructs (section 4.1.6) for one and the same concept (like three kinds of looping, for example) that deserve the name *Syntactic Sugar for the Programmer*. Some source code simplifications have therefore been issued as implementation guideline, and applied to CYBOI:

- Use only *procedures*, not *functions*! (A return value is just another parameter. There is no argument not to hand it over as such.)
- Use only *call by reference*, not *call by value*! (Handing over parameters as copy creates redundant data. It is better to use references instead.)
- Use only *if-else* conditions, not *case* statements! (Branching via simple conditions covers all necessary use cases.)
- Use only *while endless* loops, not *do-while-* or *for* loops! (Merging the loop concept with a *break* condition is not a good idea. The condition can be put into the loop's body, in order to realise *pre-* or *post-testing*.)

10.3.2 Corrected C

Moreover, there is at least one incorrect implementation solution used in the C standard libraries (*libc*) and various traditional systems. Arrays are all too often forwarded without necessary meta information like their *size* and *count* of elements. But a pointer without additional information about the size of the memory area (array) it points to is really worthless. The introduction of *Types* with defined size makes array *elements* countable, but does not solve the array size problem. A special workaround, to what concerns *string* arrays at least, was therefore thought out. It requires that a null character ('\0') be added as termination to every character array which is to be interpreted as string.

But this is not a clean solution. For large systems, it pollutes a computer's memory with thousands and thousands of termination suffixes, to compensate for the missing size information. Recalling a recommendation on knowledge modelling given in chapter 7, meta information is that *about* other information (like an array) and should thus *not* be stored inside, but *outside* the same (array). A further implementation guideline made out and considered in CYBOI, is therefore:

- Use (character) arrays only together with their *size* and *count* of elements, not as *null-terminated* strings! (Array references accompanied by necessary meta information make termination characters superfluous.)

10.3.3 Used Libraries

Before interpreting a stream of CYBOL data, a parser has to bring structure into it. Since CYBOL bases on the XML standard, one option was to use one of the many existing XML parser libraries [329, 209], for reading and writing CYBOL sources. The current CYBOI version uses *libxml2*, the *GNU Network Object Model Environment* (GNOME) XML library, written in C.

Existing libraries, however, bring with a lot of functionality, not all of which is actually used in CYBOI. A future version will therefore contain its own parsing procedures, tailored to CYBOI's needs. Another argument therefor are special requirements (differing XML attribute *begin* and *end* characters) when serialising knowledge models into CYBOL files.

Further libraries are needed. The *X Window System* (X) *X Libraries* (Xlibs), for example, contain routines to use *Graphical User Interfaces* (GUI) under UNIX-like *Operating Systems*

(OS); for a Windows OS, it would be the *Graphics Device Interface* (GDI). The *UNIX Socket*'s pendant in the Windows world is the *Windows Socket* (WINSOCK), both serving as communication mechanism. A *Textual User Interface* (TUI) for the UNIX console is programmed differently than one for a *Disk Operating System* (DOS) shell. And so on. Because of the steady changes in CYBOI's source code, and not to let this work's volume exceed the worst expectations, these shall not be elaborated on here.

10.3.4 Development Environment

Current CYBOI development happens under the *Linux* OS, using the *C Compiler* of the *GNUCompiler Collection* (gcc) [309]. CYBOI's code base has been kept simple and following the *C Standard* [156] of the *American National Standards Institute* (ANSI), which is why it should be compilable on other platforms, too. Exceptions to be considered are the above-mentioned, platform-dependent libraries with differences between UNIX or derivatives, Windows-, and other OS. As temporary workaround, the *CYGWIN* runtime environment [257] is used for running CYBOI under Windows.

The most important development tool is a simple text editor; it is used for writing program code. Compilation is started on a console or shell. Likewise, the compiled binary is run there, for testing purposes and error debugging.

10.3.5 Error Handling

One possibility to systematise errors frequently appearing during software development, is to distinguish between three kinds: *Syntax*, *Logical* and *Runtime*.

In typed programming languages like C – the language CYBOI is written in – *Syntax Errors* can be found by a compiler. Not so in CYBOL. It is a language whose models get interpreted at runtime, according to the abstraction assigned to each of them. This is comparable to scripting languages like *Python* 4.1.8 which do not request a type for variable declaration. Compilation of CYBOL files is therefore not needed and model abstractions (types) are not and cannot be checked before running a system. However, since CYBOL is based on XML, at least its correct XML syntax can be validated before execution.

Logical Errors are mostly more difficult to find than syntax errors. They may be a wrong initialisation, a false statement, a loop count mistake or comparable errors. They can better

be found in a running system, using a tool called *Debugger* that allows to check variable values at runtime. Although a special CYBOI debugger does not exist yet, it may not be too difficult to write one. CYBOI is slim; its transient models in memory are managed from one place (knowledge memory root); its signals are processed by just one single loop. A debugger would not have to jump through the actual application code, it could rely on the few containers and loop in CYBOI, and nevertheless show application model values at runtime.

Predictable *Runtime Errors* like crossing the limit of a number space, resulting from false user input, or similarly foreseeable activities can be notified to a user via an error message – on console, in a log file, or by popping up a graphical dialogue. Unpredictable runtime errors are tricky and quite hard to find. The longer CYBOI is used, the better it will be tested and the less likely will unpredictable runtime errors caused by wrong code occur.

10.3.6 Distribution and Installation

The current version of CYBOI, as well as already existing CYBOL applications, are distributed in form of *Debian GNU/Linux Packages* (DEB). Future versions may be provided in *RPM/ Red Hat Package Manager* (RPM) format as well. Also, installation files for other OS like Windows might be available then.

One question that had to be answered was where to put the CYBOI binary, but also CYBOL application files in UNIX *Filesystem Hierarchy Standard* (FHS) [278] directories. The */usr/bin* directory for CYBOI is obvious. CYBOL files, on the other hand, are the source + executable + configuration of an application, at the same time, all in one. A mailing list discussion [12, June 2005] finally suggested a practicable way, namely to put all CYBOL applications to the */usr/share/* directory.

11 Res Medicinae

*No Road can ever be too long,
side-by-side with a good Friend.*

UNKNOWN AUTHOR

The first two chapters (9 and 10) of part III of this work defined the CYBOL language and its corresponding interpreter CYBOI. Since a theory is worth more if it can be proven in practice, this chapter will describe an effort trying to apply both to create an application system named *Res Medicinae* [266] (Latin for *Matter of Medicine*).

11.1 Project

The – somewhat idealistic – aim was initially to create the prototype of a *Hospital Information System* (HIS). Due to the clearly too high-set aims, this was later revised so that the focus of the prototype became a standard *Practice Management System* (PMS) with an *Electronic Health Record* (EHR) as its core. Several technology changes during the progress of this work and the lack in time required to also revise this aim, so that now the final prototype consists of just the (rudimentary) address management module of the planned EHR application. It is written in CYBOL and executable by CYBOI.

The following sections describe the project background of *Res Medicinae*.

11.1.1 Free and Open Source Software

Just like CYBOP (including CYBOL and CYBOI) [256], *Res Medicinae* [266] is developed within a *Free/ Libre Open Source Software* (FLOSS) project. Its source code, resources and

documentation are placed under GNU's *General Public License* (GPL) (section 14.10.1) and *Free Documentation License* (FDL) (section 14.10.2), respectively. That means they can be freely redistributed and modified under the terms of these licences. Although distributed in the hope that they will be useful, the program and its resources come *without any warranty*, without even the implied warranty of *merchantability or fitness for a particular purpose*. See [104] for details.

More information on *Open Source Software* (OSS) in general can be found at [242]. There are plenty of resources for further background reading, a German one being the *Open Source Jahrbuch 2004* [110]. To what concerns FLOSS in the medical arena, many other projects exist. Comprehensive lists of these can be found at [296, 321, 8].

11.1.2 Portals and Services

As OSS became popular over the years, the number of its supporters rose. It is not long time that *Sourceforge* [217], the first *Development Portal* for FLOSS, was opened. Shortly after, others like *Freshmeat* [216] followed and meanwhile, there are also national initiatives like *BerliOS* [90] in Germany. Also the *Free Software Foundation* (FSF) offers an own portal called *Savannah* [95], hosting exclusively *free* [149] software projects. Figure 11.1 shows the four portals by their name, logo and *Uniform Resource Locator* (URL).



Figure 11.1: FLOSS Development Portals

As *BerliOS* states in its slogan, it is the aim of development portals of that kind to *foster open source development*. In *Savannah's* words, they are *central points for the development, distribution and maintenance of FLOSS*. Although very often supported by well-known sponsors, most portals are and want to stay independent. Using them, OSS projects and their developers are offered several free services (figure 11.2). Since not all of these are always useful, projects can configure their portal sites as needed.

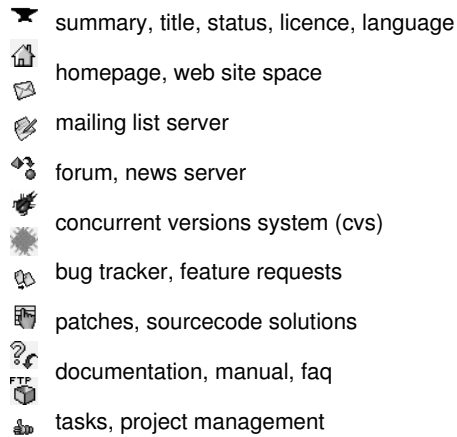


Figure 11.2: Portal Services

Res Medicinae was one of the first OSS projects registered at Sourceforge (number 4237 of now more than 100,000). CYBOP (CYBOL, CYBOI) is hosted at BerliOS.

11.1.3 Tools

Classical application development relies on tools like a *UML Designer*, for creating *Unified Modeling Language* (UML) diagrams, a *Text Editor*, *Compiler* and *Debugger*. Nowadays, these and other tools are offered in one package, as *Integrated Development Environment* (IDE).

Because the CYBOI interpreter is written in the system programming language *C*, its development requires a compiler. CYBOL applications, on the other hand, do not have to be compiled. They base on interpreted XML code which can be written in every text editor; nothing else is needed. An adapted editor was proposed in section 9.5.1.

Res Medicinae development could certainly be speeded up by using graphical diagrams in the style of the UML. But unfortunately, design tools that directly support CYBOP do not exist yet. As section 9.5.2 tried to show, some UML diagrams could be used with only minor adaptations for CYBOL modelling. For the time being, standard XML editors have to suffice.

For running and testing CYBOL applications, of course, the CYBOI interpreter is needed.

11.1.4 Contributors

OSS projects are not only *Hobby Activities* any longer. Many of them have long overtaken their commercial competitors, in functionality, stability, security and popularity. Due to the participation of sometimes hundreds of enthusiasts, they mostly have much greater momentum.

In the case of *Res Medicinae*, a number of *Medical Doctors* (MD) and *Software Engineers* have contributed with their work or expressed serious interest in collaboration. Many *Informatics Students* were (and are) involved and completed their diploma (master) works on a topic within the project. Finally, there are the OSS projects that follow similar aims, like:

- *GNUmed* [131]
- *Open Source Clinical Application Resource* (OSCAR) [263]
- *Care2002* (Care2x) [190]
- *Torch* [61]
- *Open Infrastructure for Outcomes* (OIO) [136]
- *Veterans Health Information Systems and Technology Architecture* (VistA) [328]
- *OpenEMed* [94]
- *Tcl/Tk Family Practice* (tkFP) [22]
- *Debian-Med* [259], as meta project for packaging

All of them want to provide software solutions for medicine. Being friendly concurrents, they use mailing lists such as [168] to exchange latest insights, offer help to each other and work towards a better integration. The technological decisions that have originally caused a division of forces and a multitude of projects to exist, may in the end turn out to be fruitful, with focus on the interoperability of systems. Additionally, organisations like the

Open Source Health Care Alliance (OSHCA) [241] bundle the projects' forces and regularly organise conferences.

11.2 Analysis

Abiding by the standard *Software Engineering Process* (SEP) (chapter 2), a *Requirements Analysis* stood as first activity for the development of *Res Medicinae*. The following sections will give a brief overview of some requirements and current modelling trends, concerning the *Electronic Health Record* (EHR). They do *not* try to replace more comprehensive works written on the subject.

11.2.1 Requirements Document

With the help of German medical doctors, a *Requirements Document* [135] was created and is meanwhile being updated and extended since about five years. It basically describes an EHR and the information it should include.

Since the document itself is just a hierarchical model consisting of parts, it can well be represented in CYBOL. Unfortunately, a document processor that can read and render CYBOL, in the style of *LaTeX* [188], has not been written to date (although CYBOI might integrate this functionality one day). It was therefore decided to write the requirements document in SGML/ XML, using the *DocBook* DTD [336] and tools described in *The Linux Documentation Project* (TLDP) [197].

11.2.2 EHR & Co.

Besides the now quite common term *Electronic Health Record* (EHR), some publications, experts or companies also talk of [207, 333]:

- *Personal Health Record* (PHR)
- *Virtual Health Record* (VHR)
- *Virtual Patient Record* (VPR)
- *Electronic Medical Record* (EMR)
- *Electronic Patient Record* (EPR)

- *Computer-based Patient Record* (CPR)
- *Computerised Patient Record* (CPR)
- *Computerised Medical Record* (CMR)
- *Automated Medical Record* (AMR)
- *Digital Medical Record* (DMR)
- *Patient Carried Record* (PCR)
- *Patient Medical Record* (PMR)
- *Integrated Care Record* (ICR)
- *Electronic Medical Infrastructure* (EMI)
- *Lifetime Data Repository* (LDR)

and state differences in their contents, access, maintainer, place of storage, technology or other aspects. David Kibbe, for example, as cited by Jennifer Bush [42], says:

There's recently been a subtle shift in terminology. EMR connotes a tool that's for doctors only and something that replaces the paper record with a database. EHR connotes more of a connectivity tool that not only includes the patient and may even be used by the patient, but also provides a set of tools to improve work-flow efficiency and quality of care in doctors' offices.

... An EHR should include a detailed clinical documentation function; prescription ordering and management capabilities; a secure messaging system; lab and test result reporting functions; evidence-based health guidelines; secure patient access to health records; a public health reporting- and tracking system; mapping to clinical- and standard code sets and the ability to interface with leading practice management software.

In essence, however, most of the above-listed terms are considered synonymous, since their definitions, if existent at all, differ just in nuances. Charlene Marietti, who investigated in this subject, writes [207]:

Meanwhile, most practical people don't see a big difference between the CPR and the EMR and the many other terms that exist.

Therefore, this work further on sticks to the term *EHR* and wants it understood as general description for either of the other terms mentioned above.

11.2.3 Episode Based

Historically, it took a long time until the concept of a modern EHR crystalised out. An early form of a time-oriented medical record stems from Hippocrates (5th century BC) who wanted to accurately reflect the course of a disease and indicate its possible causes. In 1907, the *Mayo Clinic* (formed by the American surgeon William Mayo) adopted one separate file for each patient, to be able to obtain a better overview of his complete disease history. This innovation was the origin of the *Patient Centered Medical Record* as known today, as [322] means.

The discussion on how to model an ideal EHR already lasts for decades and has not finished. Recent proposals brought in some new perspectives and ideas. One of them turns around the so-called *Episode-based* EHR [341]. In the centre of these considerations stands a structure that is described in a more pragmatic way by Karsten Hilbert of GNUmed [131]. He sees a complex EHR as hierarchical composition of the following items:

- Health Issue
- Clinical Episode
- Clinical Encounter
- Clinical Item

The additional concept of a *Partial Contact* as known from the Dutch *Episode Model* does not integrate into this hierarchy. But after Hilbert, *Partial Contacts* could be easily derived from existing EHR data by aggregating all *Clinical Items* that belong to the same *Clinical Encounter* and the same *Clinical Episode*.

Clinical Items are typically elements in the *SOAP* format of progress notes, as known from the *Problem Oriented Medical Record* (POMR) [339] that was introduced by Lawrence L. Weed in the 1960s. SOAP stands for:

- *Subjective*: Complaints as phrased by the patient
- *Objective*: Findings of physicians and nurses
- *Assessment*: Test results and conclusions, such as a diagnosis
- *Plan*: Medical plan, for example treatment or policy

11.2.4 Evidence Based

In an email to the *Open Health Mailing List* [168], David R. raised a number of unsolved issues concerning the *Evidence-based EHR*. In a first thought, he exposes the existence of two distinct views on an EHR: *clinical* and *evidential*. A medical record were not just a collection of clinical information, but also a *Legal Document* with financial importance. It were to give evidence of the healthcare services rendered by a particular provider for a particular organisation, and the reason why, mostly, patients do not own the record. Finally, an EHR were the result of the intersection of two major business processes: the *Clinical Process* and the *Records Management Process*.

This observation leads to the second important question whether records should be *accessed* remotely, leaving them in place at each of the organisations where the patient has been seen, or be *incorporated* as extract or full copy to each organisation's repository, as known from the paper-based world. Since the first method, promoted as trans-organisational *Virtual Record*, did not address an organisation's need for maintaining its evidential records, it had, in the opinion of David R., failed to gain widespread or long-term acceptance.

A third point turns around the authoring of an EHR. Record keeping were no longer simply a *personal* activity but rather an *inter-personal* action. David R. writes on:

Historically, providers have viewed the medical records they have created as though they were a personal journal kept by the provider to facilitate his or her process of delivering care to an individual patient. It was viewed as an aid to memory and extended the provider's thought across time. . . . In the setting of a highly mobile population of patients and providers, the record becomes a living document with multiple authors. Multiple individuals for multiple reasons consult it and . . . it is in this record that a shared understanding of the (health) problems and recommended solutions for . . . the individual occur.

Because the EHR could be seen as a space for collaboration, applications working with it had to support clinical process *Workflow* requirements. A new set of demands were also placed on health care providers, to document their activities with patients in a way that is mutually *intelligible* to those who have a stake in the information contained in the record.

11.2.5 Continuity of Care

A main result of the opinion stated in the previous section was the realisation that a major challenge for EHR design will be to overcome the difference between an organisation's evidential record management process with emphasis on *legal/ financial aspects* and the record keeping as *medical/ health documentation*, that an individual would do.

This is exactly the issue that Philippe Ameline and his French colleagues address in their *Nautilus/ Odysee* project [215]. It distinguishes between three levels of data:

- *Individual*: personal, various local
- *Group*: professional, 24 hour availability
- *Collective*: dedicated to continuity of care

The latter is called *Personal Health Project* (PHP). Its health management data can be shared between a *Patient* and his *Care Team*, with the EHR *passing by* institutions. Ameline writes in [19] that the management of these two referentials – health professional and patient – meant that applications now had to handle differently the *history data* with a time duration (which may get changed by someone else) and the data of the *instantaneous picture* kind (what one noticed and reported at a given time).

A similar effort with U.S. American roots is called *Continuity of Care Record* (CCR) [93]. Just like the PHP, it does not want to be a complete EHR, but rather: *organise and make transportable a set of basic patient information consisting of the most relevant and timely facts about a patient's condition*. Through specified XML code, the CCR becomes interoperable.

11.2.6 Core Model

Many kinds of application modules are needed in a healthcare-specific *Information Technology* (IT) environment. The tasks they fulfill, together with a proposed name within the *Res Medicinae* project, are listed following:

- *Revue*: Portal for module starting
- *Residenz*: Administrative data management
- *Record*: Clinical documentation

- *Rezept*: Prescription ordering and management
- *Reform*: Form printing
- *Report*: Public health reporting and tracking
- *Reagenz*: Laboratory- and test result retrieval
- *Rendezvous*: Scheduling
- *Roentgen*: Clinical imaging
- *Rechnung*: Billing
- *Richtig*: Statistics
- *Register*: Pharmaceutical reference
- *Ratlos*: Lexicon-, terminology- and code set query

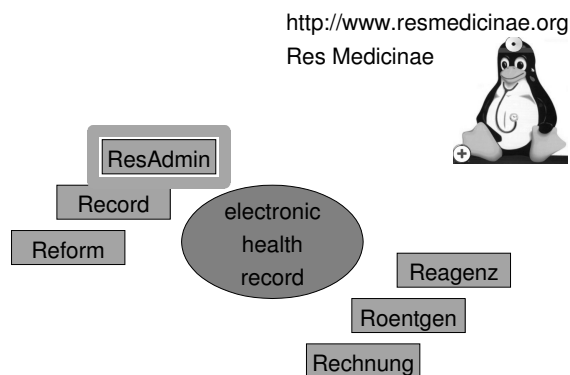


Figure 11.3: Applications Grouped around an Electronic Health Record Core

Figure 11.3 shows some of these modules, together with an EHR as their central data structure.

11.3 Standards

In a further thought, current standards of medical informatics had to be considered for the development of *Res Medicinae* application modules. There exists a whole plethora of (partly *de facto*) standards – far too many to discuss here. The following sections will give a brief overview of only a few standards which are potentially important for EHR development.

11.3.1 Overview

Figure 11.4 shows the medical informatics working groups of important standardisation organisations, namely the:

- *Deutsches Institut fuer Normung* (DIN)
- *Comite Europeen de Normalisation* (CEN)
- *International Organization for Standardization* (ISO)

ISO/ TC215 Health Informatics	CEN/ TC251 Health Informatics	DIN NAMed FB G Medizinische Informatik
WG1 Health Records and Modelling Coordination	WG I Information Models	AA G 1 Modellierung
WG2 Messaging and Communication	WG IV Technology for Interoperability	AA G 2 Kommunikation
WG3 Health Concept Representation	WG II Terminology and Knowledge Bases	AA G 3 Terminologie
WG4 Security	WG III Security, Safety and Quality	AA G 4 Sicherheit
WG5 Health Cards	Task Force Cards	AA G 5 Karten

Figure 11.4: Medical Informatics Working Groups of DIN/ CEN/ ISO [283]

The structure of the following sections is chosen after this systematics. Standards for *Health Record Modelling* will be described first, followed by those for *Messaging and Communication* and a section on *Terminology- and Coding Systems*. *Imaging-*, *Health Card-* and further

standards are mentioned afterwards. General remarks on current *Standards Development Processes* follow. Reflections on the *Implications* of standards on the development of *Res Medicinae* will conclude the topic.

11.3.2 Record Modelling

CEN/TC251

The *European Committee for Standardization* (CEN) as association of national *Standards Development Organisations* (SDO) is working on technical specifications for an *Electronic Health Record* (EHR). The *Technical Committee* (TC) dealing with that task and medical informatics in general carries the name *TC251* [50].

While its pre-standard *ENV 12265*, defined in 1995, focused on the EHR *Architecture*, the successor pre-standard *ENV 13606*, published in 1999, placed more emphasis on *Communication*. Among other things, it defined a set of reusable components (compositions) called *General Purpose Information Component* (GPIC) [208]. The *ENV 13606* pre-standard consisted of four parts:

1. Extended Architecture
2. Domain Termlist
3. Distribution Rules
4. Messages for the Exchange of Information

A third effort, the *EHR Communications* (EHRcom) task force, set up in December 2001, is to refine *ENV 13606* and to propose a revision that could be adopted by CEN as formal standard (EN), during 2004. Its current activities, as described in [76], happen in five areas:

1. *Reference Model*: generic information model for EHR communication
2. *Archetype Interchange Specification*: generic language for EHR representation and communication
3. *Reference Archetypes and Term Lists*: range of templates reflecting clinical requirements and settings
4. *Security Features*: concepts to enable interaction with security components

5. *Exchange Models*: set of models that can form the basis of message-based or service-based communication

EHRcom places special focus on the *Harmonisation* of different standardisation efforts [169] like CEN's *GPICs*, openEHR's *Archetypes* or HL7's *CDA*, described in a later section. But this is not an easy task. Bert Verhees, for example, reports [19] about name clashes between the reserved words of many programming languages and HL7 data types (Set, Array) which made it impossible to use the standard as is and necessitated a renaming of those types (into something like *HL7Set* and *HL7Array*). In his opinion, a standard should be platform independent (operating system- and programming language wise). Thomas Beale of openEHR writes to this topic [19]:

... almost all the issues ... are actually due to the HL7 data types, which CEN unfortunately decided to adopt/ adapt a long time ago. Tom Marley and others have struggled to find a version of them which a) remains faithful to the idea of HL7 but b) fixes some problems, like strange inheritance. Personally, ... I don't find the HL7 data types a good design at all ... and have made available the reasons in various standards discussions, along with many others who have pointed out the same problems. The result of this recently has been ... a new ISO work item called *Data Types for Clinical Informatics* ... which will recognise three layers:

1. Inbuilt types (like in ISO 11404)
2. General purpose clinical types (specified from requirements)
3. Bindings to particular model systems (such as HL7)

Open EHR

The *Open Electronic Health Record* (openEHR) [22] initiative, previously called the *Good European/ Electronic Health Record* (GEHR), arose from a European standardisation effort but is now based in Australia.

Pursuing an idea named the *Dual Model Approach* (section 4.6.9), which uses a *Meta-Level Architecture* as described by the *Reflection* pattern (section 4.2.1), it wants to specify so-called *Archetypes* – formally specified knowledge templates of requirements for representing and communicating EHR information.

The effort is based on the idea of no-cost open standards and free contribution.

11.3.3 Messaging and Communication

Health Level Seven

Health Level Seven (HL7), as it describes itself [150], is: *a not-for-profit, ANSI-accredited standards developing organization dedicated to providing a comprehensive framework and related standards for the exchange, integration, sharing, and retrieval of electronic health information that supports clinical practice and the management, delivery and evaluation of health services*. The more than 2000 individuals representing over 500 corporate members, world-wide, do for a great part belong to healthcare industry, implementing its interests. Accordingly, HL7's endeavors are sponsored, in part, by that industry. Its name *Level Seven*, after [276], refers to the highest level of the ISO OSI communication model (section 3.11).

Besides the mentioned framework called *Reference Information Model* (RIM), the organisation worked out a number of specifications for the exchange of messages and documents, newer formats being the *Clinical Document Architecture* (CDA), an XML-only specification, and the *Common Message Element Type* (CMET) [208], a reusable component.

Thomas Beale criticised in [168], that CDA did not have very strong semantics for some of its detailed parts, since they were derived from XHTML, and that it tended to mix presentation and representation concerns somewhat. Furthermore, CDA had recently incorporated some RIM classes into its content level, via the creation of a *Refined Message Information Model* (RMIM) which were a pity, because it reduced its genericness and made it depend on the RIM, which were essentially an analysis pattern of domain relationships, not a model of recording.

Instead, as Beale writes in a later message to [168], they (HL7) might start thinking about generic solutions, which incorporate clinical models, separated from their XML schemas, for a start. . . . Single level XML approaches didn't have much long term future in his opinion, because they didn't properly separate clinical models from information representation, which were required to allow compositional clinical models and specialisable clinical models to be built independently from the software.

HL7's recommendations found partly application in a greater number of *Hospital Information Systems* (HIS), but rarely in smaller or medium-sized *Practice Management Systems* (PMS).

Healthcare Domain Task Force

The *Object Management Group* (OMG) whose mission is: *to help computer users solve integration problems by supplying open, vendor-neutral interoperability specifications*, is the creator of widely-used de facto standards (section 4.4.6) like UML, MOF, XMI, CWM or CORBA, all now belonging to the *Model Driven Architecture* (MDA).

Published in the 1990s, the *Common Object Request Broker Architecture* (CORBA) was one of the first standards specifications created by the OMG. It tried to separate interfaces of programming objects (components) from their implementation, to improve communication between programs, independent of which programming language, operating system, computer architecture or network were used. The specification includes the neutral *Interface Definition Language* (IDL), the *Object Request Broker* (ORB) middleware functionality and a corresponding *Internet Inter ORB Protocol* (IIOP).

CORBA has been adopted by many applications, and been adapted for many domains, one of them being *Healthcare*. The OMG working group dealing with that field is the *Healthcare Domain Taskforce* (HDTF) [237] (formerly called *CORBAMED*). It defined IDL interfaces for a number of different healthcare services, for example:

- *Person (Patient) Identification Service* (PIDS)
- *Lexicon (Terminology) Query Service* (LQS)
- *Clinical Observations Access Service* (COAS)
- *Resource Access Decision Service* (RADS)
- *Clinical Image Access Service* (CIAS)

HDTF specifications are helpful in that they standardise certain functionality calls that all systems implementing the corresponding interfaces may rely on. However, they do not make any assertions about how medical knowledge should be structured.

EDIFACT

The *Electronic Data Interchange for Administration, Commerce and Transport* (EDIFACT) [318] is a standard maintained by committees of the *United Nations* (UN). It was defined to ease the electronic exchange of general business data, but is widely used for the transmission of healthcare information between organisations, too. [170]

The *European Board of EDI Standardisation* (EBES) participates in the development and distribution of EDIFACT standards. For the healthcare sector, this task falls to the *EBES Expert Group* (EEG) 9. It has specified many message formats, for instance for referral letters or electronic prescriptions. Although some countries like Denmark, Norway or Austria make use of these standards, they have not gained wider currency. [283]

Messages in form of the EDIFACT protocol are based on syntax elements which are described in another standard, the *ISO 9735*. Their data elements are contained in a well-defined collection of segments, in a well-defined sequence. [314]

x Data Carrier

Various national standards for medical software exist. In Germany, a widely used de facto standard for EDI in healthcare is the *x Datenträger* (xDT) [172], a family of formats specifying data packets. The German word *Datenträger* (DT) means something like *Data Container*. The *German College of Community Physicians*, called *Kassenärztliche Bundesvereinigung* (KBV), is initiator and maintainer of xDT, to which belong, for example:

- *Behandlungs DT* (BDT): EHR interchange between systems (was ADT and developed to the current BDT)
- *Labor DT* (LDT): laboratory data format (predecessor: *Bonner Modell*)
- *Abrechnungs DT* (ADT): reimbursement data format
- *Ambulant Operieren DT* (AODT): outpatient/ minor surgery data format
- *Geräte DT* (GDT): medical device interfacing (predecessor: *Münchener Protokoll*)
- *Kommunikations DT* (KDT): communications data format (referral letters)
- *Kassenärztliche Vereinigung DT* (KVDT): all sorts of reimbursement data (container format to wrap the others)

After Karsten Hilbert [168], xDT were mostly used in *Practice Management Systems* (PMS) of *General Practitioners* (GP). Most *Hospital Information Systems* (HIS) weren't using it. Some lab software being part of a HIS and servicing PMS apparently would use it.

Besides the KBV, institutions and groups like HL7, *KV Nordrhein*, *Zentralinstitut für die Kassenärztliche Versorgung* (ZI), *Deutsches Institut für Medizinische Dokumentation und Information* (DIMDI), *Bundesvereinigung Deutscher Apotheker Verbände* (ABDA), *Verband der Hersteller von IT Lösungen für das Gesundheitswesen* (VHitG), *Verband*

Deutscher Arztpraxis Softwarehersteller (VDAP) and *Qualitätsring Medizinische Software* (QMS) are currently working on converting xDT into an XML-based standard, called *Standardisation of Communication between Information Systems in Physician's Offices and Hospitals using XML* (SCIPHOX) [80], which originates in HL7's CDA. Accompanying projects [301] include the *Robert Koch Institut* (RKI), *Studienzentrum Göttingen* (Allgemeinmedizin) and others, as further partners.

Healthcare Xchange Protocol

Finally, there are standardisation activities in the *Open Source Software* (OSS) community. Just recently, the *Healthcare Xchange Protocol* (HXP) [269] was defined by a number of projects.

HXP is a data exchange protocol to be used by healthcare applications to communicate transparently with each other, regardless of their corresponding platform. The aim is to make data exchange simple to implement, easy to understand, flexible, reliable, secure, free, and more. HXP is based upon the XML *Remote Procedure Call* (RPC) open standards specifications, that is it uses messages in XML format for communication.

The specification and all documentation are open and everybody can contribute ideas.

11.3.4 Terminology Systems

Besides defining the differences between a *Lexicon* (list of pure words) and *Terminology* (also containing phrases), the latter sometimes called *Vocabulary*, section 4.6.5 introduced tree-like *Hierarchies* as one way to organise such sets of words or terms. Three concrete schemes for organising terminologies were described in section 4.6.6: *Enumerative*, *Compositional* and *Lexical*. Controversial opinions about terminologies exist. Thomas Beale wrote in [168, December 2003]:

... trying to standardise the whole of medicine ... is a fruitless enterprise. Sam Heard has said this many times in presentations in Australia, and when he first started saying it, was amazed not to be stoned publicly; in fact many people have come to this conclusion through their own hard work, but aren't comfortable with saying it, since it goes against current orthodoxy (embodied in things like SNOMED CT).

Nevertheless, terminologies *are* a topic of research and sometimes used in practice, as the example of ICD (see below) shows. This section therefore briefly describes some medical terminologies and, by referring to Jeremy Rogers [276], tries to assign them to one of the before-mentioned schemes.

ICD

The *International Classification of Diseases* (ICD): *has become the international standard diagnostic classification for all general epidemiological and many health management purposes ... It is used to classify diseases and other health problems recorded on many types of health and vital records including death certificates and hospital records.* [342]

Scheme: enumerative

Maintainer: World Health Organisation (WHO)

OPCS

The *Office of Population Censuses and Surveys Classification of Surgical Operations and Procedures* (OPCS) is a: *statistical classification of diseases and surgical procedures, respectively. It allows the: logical translation of clinical statements into codes in a way that facilitates the retrieval of data in a consistent manner and comparative analysis of aggregated datasets compiled from multiple sources.* [219]

Scheme: enumerative

Maintainer: *National Health Service Information Authority* (NHSIA)

READ

The *Read Codes* (READ), as their older name *Clinical Terms Version 3* (CTV3) says, are a: *list of terms describing the care and treatment of patients. They: cover a wide range of topics in categories such as signs and symptoms, treatments and therapies, investigations, occupations, diagnoses and drugs and appliances. Further, they: provide cross maps to both ICD-10 and OPCS-4 classification codes.* [220]

Scheme: enumerative

Maintainer: *United Kingdom* (UK) *National Health Service Information Authority* (NHSIA)

LOINC

The *Logical Observation Identifiers, Names and Codes* (LOINC) is a database whose purpose is: *to facilitate the exchange and pooling of results ... for clinical care, outcomes management, and research.* Its codes are: *universal identifiers for laboratory- and other clinical observations.* [157] After [276], it were now closely allied to SNOMED CT (see later section).

Scheme: hybrid enumerative-compositional

Maintainer: *United States (US) Regenstrief Institute*

ICNP

The *International Classification for Nursing Practice* (ICNP) is a: *combinatorial terminology for nursing practice that facilitates crossmapping of local terms and existing vocabularies and classifications.* It wants to: *establish a common language for describing nursing practice in order to improve communication among nurses, and between nurses and others.* [229]

Scheme: hybrid enumerative-compositional

Maintainer: *International Council of Nurses (ICN)*

SNOMED CT

The *Systematized Nomenclature of Medicine* (SNOMED) *Clinical Terms* (SNOMED CT) is: *a dynamic, scientifically validated clinical health care terminology and infrastructure that makes health care knowledge more usable and accessible.* The SNOMED CT core terminology: *contains over 364,400 health care concepts with unique meanings and formal logic-based definitions organized into hierarchies. As of January 2005, the fully populated table with unique descriptions for each concept contains more than 984,000 descriptions. Approximately 1.45 million semantic relationships exist to enable reliability and consistency of data retrieval.* [158]

SNOMED CT was created by combining the content and structure of the SNOMED *Reference Terminology* (SNOMED RT) with the United Kingdom's (UK) *Read Codes* (READ)

clinical terms. Meanwhile, mappings and integrations for further standards exist, e.g. for several ICD versions (ICD-9-CM, ICD-10, ICD-O3), OPCS-4 and LOINC.

Scheme: hybrid enumerative-compositional

Maintainer: *SNOMED International* and *College of American Pathologists* (CAP)

Odyssee

The *Odyssee* open source project [215] contains a terminology (*Lexique*) of more than 35,000 (French) terms, each with a code, at the core of its system. Additionally, it contains a *Semantic Network* of links between terms of the Lexique, to give sense. Links can be *is a*, *belongs to* or *has unit*. Philippe Ameline writes [19]:

In Odyssee, we describe all that we can with trees. If we compare the Lexique with medical vocabulary, trees are sentences made of its words. Each node of a tree is an object with fields like the Lexique's code, complement (to store numbers or external codes), degree of evidence (from 0=no to 100=certain). Trees can also contain free text sentences . . .

In Odyssee, each and every structured document is a tree; you just have to look at the Lexique term at its root to know what it is. The whole patient record can even be seen as a huge tree with (the) term *Patient* as root. Trees can be shown *as is* or, for report generation, be translated to natural language sentences.

Scheme: compositional

Maintainer: *Odyssee Non-Profit Organisation* (NPO), *Logiciel Nautilus*

OpenGALEN

The *Generalised Architecture for Languages, Encyclopaedias and Nomenclatures in Medicine* (GALEN) is trying to construct a: *semantically sound model of clinical terminology* – the *GALEN Common Reference Model* (CRM). The formal rules (representation scheme) for manipulating its concepts are provided by the *GALEN Representation and Integration Language* (GRAIL).

The original GALEN project was sponsored by the *European Union* (EU) and open-sourced and renamed into *OpenGALEN*, in 1999. It later continued as part of the *Synergy on the Ex-*

tranet (SynEx) project [59], which arose from the *Synapses* project aiming at implementing a federated healthcare record server.

Scheme: compositional

Maintainer: OpenGALEN *Non-Profit Organisation* (NPO)

UMLS

The *Unified Medical Language System* (UMLS) consists of knowledge sources (databases) and associated software tools (programs). By design, the knowledge sources are multi-purpose, that is: *they are not optimised for particular applications, but can be applied in systems that perform a range of functions involving one or more types of information, e.g. patient records, scientific literature, guidelines, public health data.* [228] There are three UMLS knowledge sources:

- *Metathesaurus*: a very large, multi-purpose, and multi-lingual vocabulary database containing information about biomedical and health-related concepts, their various names, and the relationships among them; its source vocabularies are many different thesauri, classifications, code sets, and lists of controlled terms, of which it cross-references over 79 [276], often by deriving from lexical analysis of the terms; a core thesaurus are the *Medical Subject Headings* (MeSH)
- *Semantic Network*: a consistent categorisation of all concepts represented in the UMLS Metathesaurus (currently 135 semantic types) and a set of useful relationships between these (currently 54 semantic links)
- *Specialist Lexicon*: a general English lexicon that includes many biomedical terms; records the syntactic, morphological, and orthographic information for each word or term

To its associated software belongs the *UMLS Knowledge Source Server* (UMLSKS), which is: *a set of Web-based interactive tools and a programmer interface to allow users and developers access to the UMLS knowledge sources, including the vocabularies within the Metathesaurus.* [228]

Scheme: lexical

Maintainer: *United States* (US) *National Library of Medicine* (NLM)

Others

Numerous other terminology systems exist, only some of which are listed below:

- *Oxford Medical Information System* (OXMIS) Dictionary
- *International Classification of Health Problems in Primary Care* (ICHPPC)
- *International Classification of Primary Care* (ICPC)
- *International Classification of Functioning, Disability and Health* (ICF)
- *Universal Medical Device Nomenclature System* (UMDNS)

11.3.5 Further Standards

As wide as the field of medicine – and therewith medical informatics – is the number of further standards that could be considered. Understandably, only a few more examples can be mentioned here.

DICOM

Digital Imaging and Communications in Medicine (DICOM) is a: *multi-part standard produced to facilitate the interchange of information between digital imaging computer systems in medical environments.* [230] Medical devices like *Computer Tomographs* (CT), manufactured by various vendors, produce a variety of digital image formats, which explains the need to standardise their transfer.

DICOM not only defines its own file format containing meta data and the actual image data (in compressed or uncompressed form), but also a transport protocol called *DICOM Message Service Element* (DIMSE), which is based on the *Transfer Control Protocol* (TCP). Just like the *Simple Object Access Protocol* (SOAP), DICOM uses the *Service Oriented Architecture* (SOA) – a simple principle describing a service as *Remote Procedure Call* (RPC). [177]

Maintainer: *American College of Radiology* (ACR), *National Electrical Manufacturers Association* (NEMA)

GMDN

Global Medical Device Nomenclature (GMDN) is a: *collection of internationally recognised terms used to accurately describe and catalogue medical devices ... in particular, the products used in the diagnosis, prevention, monitoring, treatment or alleviation of disease or injury in humans*. It is divided into 12 categories of devices and contains nearly 7,000 terms plus more than 10,000 synonyms to make the GMDN easier to use. [206]

Maintainer: *Maintenance Agency Policy Group* (MAPG)

NCPDP

Various electronic standards for the transmission of pharmacy data exist. They cover areas such as the: *identification of drugs and health related products*, the: *adoption of standard identifiers for pharmaceutical data transactions*, the development of: *standardised messages for prescribers, pharmacists, payers and/ or other interested parties to exchange multi-directional information* and, most importantly, the maintenance of: *standard forms and guidelines to accommodate electronic pharmacy claim information at the point-of-service*. [92]

Maintainer: *National Council for Prescription Drug Programs* (NCPDP)

CLSI

Globally applicable voluntary consensus documents and guidelines for healthcare testing are another area of standards development. In particular, clinical laboratory testing and in vitro diagnostic test systems are considered here. [54]

Maintainer: *Clinical and Laboratory Standards Institute* (CLSI), formerly called *National Committee for Clinical Laboratory Standards* (NCCLS)

ADA

Standards, specifications, technical reports and guidelines are also developed for: *components of a computerised dental clinical workstation and electronic technologies used in dental practice*. [238]

Maintainer: *Standards Committee on Dental Informatics (SCDI)* belonging to the *American Dental Association (ADA)*

CDISC

Standards with focus on the global, platform-independent data exchange between information systems are based on: *data models (that) will ultimately support the end-to-end data flow of clinical trials, from the source(s) into an operational database, through analysis to regulatory submission. The sources of data that are relevant to (these standards) vary among patient records – e.g. case report form data, clinical laboratory data, data from contract research organizations, shared data between companies with corporate mergers or development partners, and other sources.* [49] In this context, two kinds of data modelling are distinguished:

- *Operational Data Modeling (ODM)*: referring to standard data interchange models that are being developed to support data acquisition, interchange and archiving of operational data
- *Submission Data Modeling (SDM)*: referring to standard metadata models being developed to support the data flow from the operational database to regulatory submission

Maintainer: *Clinical Data Interchange Standards Consortium (CDISC)*

eHC

Finally, there are specifications defining *electronic Health Cards (eHC)* or *Health Professional Cards (HPC)*, like the ones to be introduced in Germany, in 2006. [105] They describe the *Card Operating Systems (COS)*, basic card applications and -functions, as well as many other issues like electronic prescriptions.

Maintainer: *Deutsches Institut fuer Medizinische Dokumentation und Information (DIMDI)*

11.3.6 Standards Development

Standards development in its today's form found a lot of critics, especially among developers of the OSS community [19]. Their complaints concern the:

- Nondisclosure and secrecy of specifications
- Lengthy update cycles
- Limited access to standardisation bodies
- High membership fees

Thomas Beale who argues that the current paradigm of development of technical/ information standards were broken at the core anyway [19], has some more arguments, which are summarised following.

Unproven Specifications The operativeness of technical specifications and designs produced by *Information Technology (IT) Standards Development Organisations* (SDO) were *never proven in practice*, yet the best way to test a design was to try to build it. Current standards specifications rather reminded on what software engineers would call *Requirements Analysis Models*.

Static Documents Modern developers understood the idea of a *Living Documentation* – one which were never finished and always under modification due to feedback from implementation and actual use. That is why systems got rebuilt two or three times before they were really good. Yet this didn't happen with standards. They were published as *static* documents, and the available feedback processes were so slow as to be nearly useless. But feedback had crucial value in validating and improving specifications. Many standards processes continued as talk-/ documentation fests for years, before anyone seriously tried to validate the models or designs.

Missing Methodology Standards specifications were not developed by any recognised engineering *Methodology*, often without any discipline whatsoever. Instead, they were developed by *ad hoc* argumentation in conference rooms, by whoever happens to turn up, with whatever skills (often many skills, but few relevant ones). Sometimes, whoever shouted the loudest would win.

Arbitrary Definitions The current results of many technical standards definition efforts were often arbitrary, contained bad modelling, and did not have proper statements of the problem or rigourously developed technical artifacts.

Beale concludes that any IT standards development not being a *live* process with *Implementation* and *Use* and *Feedback Loops*, were not worthwhile.

11.3.7 Implication

The number of standards for medical informatics is huge. The fields covered by these standards are manifold. Popular standardisation efforts dealing with the EHR structure are *Open EHR* and *CEN 13606*.

The borders to messaging and communication standards are blurred. Although *HL7*'s focus lies on message exchange, it created data structures in form of its *RIM* framework, too; a newer result for document exchange is their *CDA* specification. The former two standards (*CEN 13606* and *Open EHR*), on the other hand, focus on the EHR structure but offer a communication format as well; it is called *Transaction* or *Composition*, respectively. Beale concludes in [168]: ... *all efforts have converged independently on at least one solid concept – the unit of change and committal in the EHR.*

OMG's *HDTF* defines interfaces for the exchange of messages, which are grouped into special services. Some national efforts have defined their own data exchange formats, like the *xDT* standard (to become *SCIPHON*) in Germany. Yet other standards recommendations for electronic data interchange in medicine are *EDIFACT*, worked out by the UN, and *HXP*, defined by a number of medical OSS projects.

To what concerns the field of medical terminology, there exist longer-lasting efforts like *ICD*, *LOINC*, *SNOMED CT*, *OpenGALEN* or *UMLS*. Depending on their scheme of organisation, they may be grouped into the three categories: *enumerative*, *compositional* and *lexical*. A lot of time and money has been invested into them, yet only recently, their results have been adopted by increasingly more systems. Good acceptance and popularity was reached for the *ICD* codes classification system.

Other standards for related fields exist, among them being *DICOM* for clinical imaging and -device communication, *NCPDP* for the transmission of pharmacy data, *CLSI* for clinical laboratory testing, *ADA* delivering guidelines for dental informatics or *CDISC* for the exchange of large amounts of various data between information systems.

For the purpose of this work, with a minimalistic implementation of a prototype application, the considered (de facto) standards specifications mainly had a helper function, giving some architectural guidance. Concerning the record architecture, CYBOL applications follow the purely compositional principles of CYBOP anyway, so that record modelling advices had only few implications. CYBOP's architecture, however, is flexible enough to support many messaging standards in the future, by simply adding the corresponding translator modules. Existing terminologies can partly be used by associating terms appearing in CYBOL knowledge templates with their pendants in common terminology systems.

A promising trial, in this context, would be to use CYBOL for building up new, or structuring existing terminologies. CYBOL innately supports compositional structures, which makes it a perfect match for compositional schemes. Further, it allows to add meta information as well as to integrate constraints. The meta information, which is contained in so-called *property* tags of a term (chapter 9), at system runtime called *details*, may link to more than one superior (parent) category, thereby placing the term simultaneously under different categories that are valid. Thus, some problems of current terminologies (section 4.6.6) *might* get solved. But this remains to be figured out in future works (chapter 13).

Standards for imaging, pharmacy- or laboratory data transfer, guidelines for dental informatics, health card usage and related specifications will be considered closer as soon as more application modules are developed within *Res Medicinae*.

11.4 Realisation

Having analysed the domain of healthcare and having investigated corresponding standards, actual design solutions that have been tried out in the course of this work, by implementing them in software source code, can be described in the following sections.

11.4.1 Student Works

Some helpful contributions came from a number of students, collaborating within the *CYBOP* and/ or *Res Medicinae* projects. The works, completed at the *Technical University of Ilmenau* (TUI), are of the three types: *Seminar Paper*, *Research Project* or *Diploma Thesis*, and listed with their title and results in table 11.1.

The first six of these works were intended to become modules for the first-trial Java prototype of *Res Medicinae*, as described in the next section. Further works created tutorials for different base technologies, such as the *Xlibs* library of the *X Window System* or *Socket Communication* mechanisms. Finally, one diploma thesis helped in defining the CYBOL language, by creating a prototype in it.

Title	Type	Result
A flexible Software Architecture for Presentation Layers demonstrated on Medical Documentation with Episodes [31]	Diploma Thesis	Java application for topological documentation
A Technology-neutral Mapping Layer for Data Exchange demonstrated on Medical Form Printing as integrative part of an EHR [185]	Diploma Thesis	Java application with one form and persistent storage of data
Creating a Backup Module under Consideration of Common Design Patterns as provided by the ResMedLib Framework [23]	Research Project	Java application for file backup
Creating Web Frontends for Scheduling and Management of administrative Data, based on a Web-server with JSP Technologie [140]	Research Project	Apache webserver extension using Java and JSP
Creating Intuitive Frontends under Consideration of Internationalisation Aspects [171]	Research Project	Java application in English, German and Tamil (Latha)
Evaluating Component Technologies in the Domain of Medical Image Processing [177]	Diploma Thesis	ImageJ extension for image transfer via CORBA and SOAP
X11 Architecture and XLib Functionality [83]	Seminar Paper	Tutorial and prototype
Communication over Sockets [175]	Seminar Paper	Tutorial and prototype
XML Parser [311]	Seminar Paper	Code fragments
Implementation Possibilities for CYBOL Web Frontends, using Cybernetics Oriented Programming (CYBOP) Concepts [141]	Diploma Thesis	CYBOI extensions and a more detailed CYBOL specification

Table 11.1: Student Works [256]

11.4.2 First Trial

An early trial of a *Res Medicinae* module was *Record*, an application for EHR management (figure 11.5). It was a standard Java-based system and had a *Graphical User Interface* (GUI). Its classical architecture made use of many software patterns (section 4.2) and was shared into the parts *Domain Model*, *Graphical View* and *Controller*, as proposed by the equally named pattern, abbreviated *MVC*.

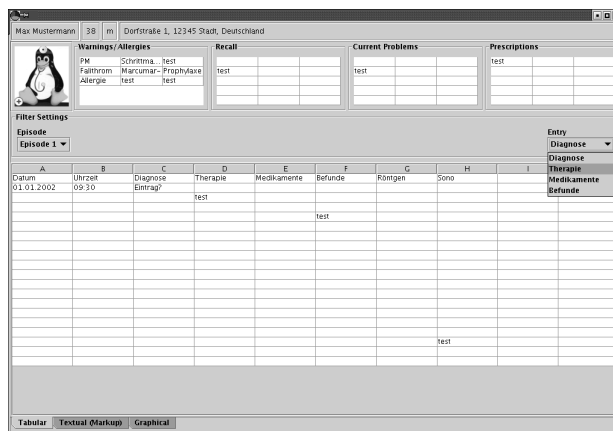


Figure 11.5: Early Record Module

Later prototypes extended that architecture by applying the CYBOP concept of *Composition*. In a first step, the *Hierarchical MVC* (HMVC) pattern was used to replace the MVC pattern, resulting in nested *Controllers* and *Views* (figure 11.6). Afterwards, the principle of *Hierarchy* was applied in general, also to *Domain Models* and to as many other parts as possible.

Classes as known from *Object Oriented Programming* (OOP) do not represent dynamically extensible containers but have a static structure with a fixed number of attributes. In other words, the *Hierarchy* as concept is not inherent in OOP types. Yet abstract models as humans build them in their minds are always based on hierarchies (section 7.1.3). A programming language which does not consider this, does not allow users to make full use of their modelling potential.

To eliminate this flaw and implement a hierarchical structure in the Java prototype, a top-

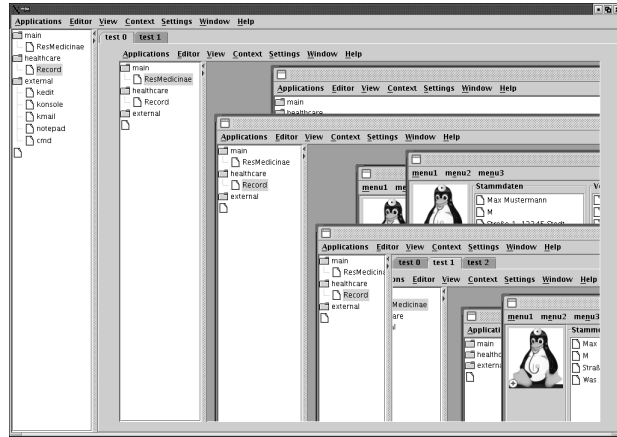


Figure 11.6: Nested Views of Module Frames

most super class named *cybop.Model* had to be introduced (compare also figure 7.15). It represented a container that had the capability to reference itself – in other words a *Tree Structure*. As such, it offered *set*, *get* and *remove* methods for its elements. Since these access methods were inherited, sub classes did not have to implement their own (for each attribute) anymore, which saved hundreds of lines of source code.

One of these advanced modules (*ReForm*) assisted medical form printing [185], another one (*ResAdmin*) managed administrative data of patients with emphasis on internationalisation [171], yet another one (*ResData*) served as appointment and scheduling module, accessible over web [141], a backup application (*Restore*) was implemented as well [23] and a last module (*Record*, in an extended version) was responsible for clinical documentation [31]. This documentation process was supported graphically, by *Record's* ability for *Topological Documentation* (figure 11.7) and, of course, it could also manage and store patient data, in XML files.

11.4.3 Knowledge Separation

In the case of the first prototypes, one could still speak of true *Implementation*, because design models had to be transferred into another form of abstract model: the Java programming language source code. Not so in later versions of *Res Medicinae*.

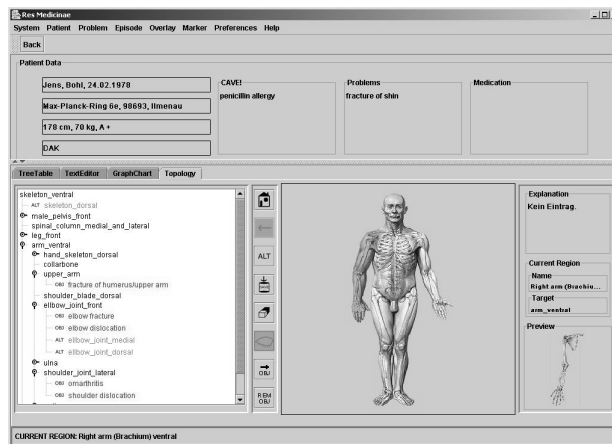


Figure 11.7: Topological Documentation in Record Module [31]

While the early prototypes represented the classical mix of domain knowledge and low-level system instructions, that was eliminated later. All knowledge got *extracted* and was put into special configuration files, in *CYBOL* format (section 9). Henceforth, these contained not only settings like font size or colour, as known from standard applications, but the *whole* domain knowledge, including user interface- and workflow structures.

Following the explanations of part II of this work, the *static* knowledge was shared into different models, some representing *state*-, and others *logic* knowledge. This was very much opposed to the earlier Java implementations whose classes bundled attributes and methods.

Without the knowledge, the remaining program code looked pretty much like a skeleton of basic system functionality. Serving as hardware interface, it concentrated memory- and signal handling in one place – exactly those things which part II of this work called *Dynamics*. Additionally, that remaining system had the ability to interpret knowledge, which is why it was called *CYBOI* (interpreter). One could, in some way, compare it with what the *Java Virtual Machine* (JVM) is for Java, only that CYBOI processed knowledge given in form of CYBOL templates, which look different than Java source code.

CYBOI needed an *XML Parser* in order to be able to read the knowledge contained in CYBOL files. The decision here fell on Apache's *Xerces* [268], because one of its versions is implemented in Java.

11.4.4 Reimplementation

The architecture-advanced prototype of the *Record* module had *much* less functionality than earlier ones, in fact not much more than starting a graphical frame with menu bar and exiting the application again. This was so, because yet before all domain knowledge could be extracted into CYBOL, another issue turned up:

CYBOP modelling concepts like *Itemisation* or *Composition* are an integral part of the CYBOL knowledge representation language. Other concepts like the *Bundling* of attributes and methods, property- and container *Inheritance*, as known from *Object Oriented Programming* (OOP), were considered unfavourable (section 4.1.15) and neither to be used in CYBOL, nor in the CYBOI interpreter. Consequently, OOP languages like Java or C++ were not suitable for CYBOI any longer. A slim and fast language, close to hardware and fast in processing CYBOL was needed.

Having such requirements, one of the first candidates coming to mind was the *C* programming language. It is *high-level* enough to permit fast programming and *low-level* enough to connect efficiently to hardware or an *Operating System* (OS). Many OS are written in C themselves, anyway. CYBOI was therefore reimplemented in C, which hasn't changed since. What has changed and is changing all the time is its functionality, an overview of which was given in chapter 10.

CYBOL sticks to the XML specification and standard XML parsers can be used to process and validate it. However, for reasons of performance and better integration, and due to the very limited vocabulary (set of possible tags and attributes), special parsing procedures were written and adapted to CYBOI.

Another problem that had to be solved was *Graphical User Interface* (GUI) handling. While the Java-implemented CYBOI could make use of the *Abstract Windowing Toolkit* (AWT)/Swing, the C-implemented CYBOI did not have such functionality at first. Toolkit candidates like *Qt* [315] or *wxWindows* [288], being implemented in C++, were out. Other GUI frameworks like the *Gimp Toolkit* (GTK) [307], written in C, were considered cumbersome to cope with so that finally, the decision was taken to use low-level graphics drawing routines. For CYBOI, being developed on a *GNU/Linux* OS [313], that meant using *XFree86's* [58] *X-Library* (Xlib) functionality directly. The necessary effort for transforming hierarchical CYBOL models into GTK- or other toolkit structures was estimated to be equal or even higher than translating them into Xlib functionality right away. At the time of writing this work, implementation is in progress but not completed.

Similar implementations are necessary for *Textual User Interfaces* (TUI), *Web User Interfaces* (WUI) and *Socket Communication Mechanisms*, the latter two being already finished in a first version. Further development activities may for instance enable CYBOI to run on other platforms and integrate more hardware-driving functionality, to get independent from underlying OS.

While the CYBOL specification can be considered quite mature, CYBOI, as could be seen, will need plenty of extensions and additions in future, in order to leave its prototype stage and become fully usable.

11.4.5 Module Modelling

When CYBOI had become more stable (besides the extensions that were – and are – frequently implemented), development could focus on the actual application again. From now on, *Res Medicinae* modules only had to be *modelled* in CYBOL, but no longer had to be *coded* in a programming language. The designed state- and logic knowledge, existing in form of CYBOL templates, already represented the complete application; no further implementation phase was needed.

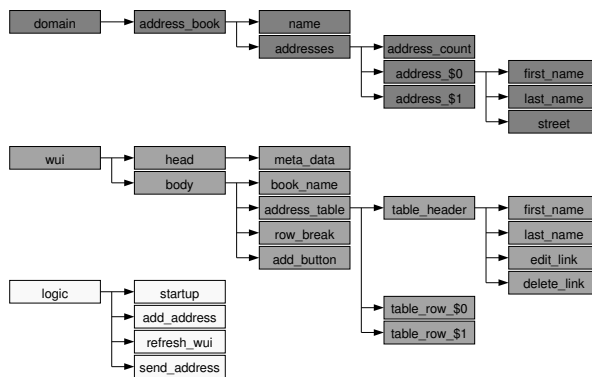


Figure 11.8: ResAdmin Knowledge Models (Extract)

Due to the tremendous complexity of an *Electronic Health Record* (EHR), only a very

small part of its data could be considered for the application prototype within this work. Administrative data like a person's name or address are standard information found in all EHRs. A corresponding module named *ResAdmin* [141] was therefore elected to be realised first (compare module list in section 11.2.6). Its models belong to three categories: *Domain*, *Web User Interface* (WUI) and *Logic* (figure 11.8).

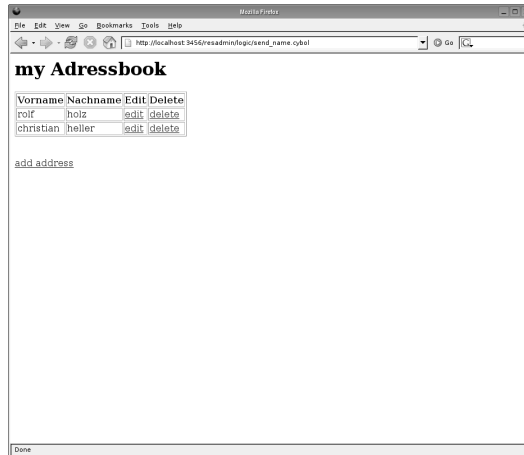


Figure 11.9: Simple Web User Interface of the ResAdmin Module [141]

The addresses contained in the *domain* branch of the knowledge tree are manipulated across *Hyper Text Markup Language* (HTML) *User Interface* (UI) models belonging to the *web* branch of that same tree. An example structure of a knowledge tree was shown in figure 8.15. Figure 11.9 shows the minimal WUI. Every action model that a user can trigger through the WUI exists as part of the *logic* branch of the knowledge tree.

Independently of what kind of knowledge model (state or logic) was created, the ontological principles (section 7.3) were strictly followed. Most importantly, relations within a hierarchical model were always *unidirectional*, that is from a *Whole*- to its *Part* models, but never the other way around. Additionally, however, logic models may reference and access runtime state models.

Some of the logic models represent *Translators* (compare section 8.2). They extract address information residing in the domain- and copy them to the web model, which is afterwards sent to the human user as communication partner. This principle holds true for the com-

munication between application systems, only that then other than web models are used as communication format. The vision to make all communication channels really *transparent* and easy to handle for the user now seems to be coming true. The following example shows an extract of a CYBOL logic knowledge template for ResAdmin:

```
<part name="set_loop_index" channel="inline" abstraction="operation" model="copy">
  <property name="source" channel="inline" abstraction="integer" model="0"/>
  <property name="destination" channel="inline" abstraction="knowledge"
    model="domain.index"/>
</part>
<part name="set_address_count" channel="inline" abstraction="operation" model="count_parts">
  <property name="basename" channel="inline" abstraction="string" model="address"/>
  <property name="model" channel="inline" abstraction="knowledge"
    model="domain.addressbook.addresses"/>
  <property name="result" channel="inline" abstraction="knowledge"
    model="domain.addressbook.addresses.address_count"/>
</part>
<part name="compare" channel="inline" abstraction="operation" model="compare">
  <property name="left_operand" channel="inline" abstraction="knowledge"
    model="domain.index"/>
  <property name="right_operand" channel="inline" abstraction="knowledge"
    model="domain.addressbook.addresses.address_count"/>
  <property name="operator" channel="inline" abstraction="string"
    model="greater_or_equal"/>
  <property name="result" channel="inline" abstraction="knowledge"
    model="domain.break_flag"/>
</part>
<part name="create_table_body" channel="inline" abstraction="operation" model="loop">
  <property name="break" channel="inline" abstraction="knowledge"
    model="domain.break_flag"/>
  <property name="index" channel="inline" abstraction="knowledge"
    model="domain.index"/>
  <property name="model" channel="inline" abstraction="knowledge"
    model="logic.create_table_rows"/>
</part>
<part name="send_wui" channel="inline" abstraction="operation" model="send">
  <property name="language" channel="inline" abstraction="string" model="tcp_socket"/>
  <property name="receiver" channel="inline" abstraction="string" model="user"/>
  <property name="message" channel="inline" abstraction="knowledge" model="web"/>
</part>
```

The loop *index* and *address_count* parameters are initialised first, before a comparison operation possibly sets the loop's *break_flag*. Then, the actual loop is executed and table rows

are created, until the break flag is really set. Finally, the complete *web* knowledge model is sent via *tcp-socket* to the human *user* as receiver watching the WUI graphically rendered by a web browser.

It can be concluded that the simple prototype module *ResAdmin*, realised within the *Res Medicinae* project, demonstrates how a CYBOL application including domain-, user interface- and logic models may look like.

Part IV

Completion

12 Review

Knowledge can create problems.

It is not through ignorance that we can solve them.

ISAAC ASIMOV

This chapter will review the whole work in brief. It begins with firstly, *validating* the achieved results in comparison to the aims set initially. Secondly, a short discussion will *evaluate* these results once more, before a third section mentions *limits* of the proposed solution.

12.1 Validation

The state-of-the-art chapters 2, 3 and 4, at the beginning of this work, dealt with the *Software Engineering Process* (SEP), the *Physical-* and *Logical Architecture* of information systems. A rather large number of existing software design concepts were investigated, and some of their aspects criticised, before chapter 5 suggested a new approach for their improvement. Many of its new concepts and ideas stem from nature or other disciplines of science, which is why that programming approach was given the attribute *cybernetics-oriented* (CYBOP). Part II then proposed a slightly different view on how to abstract knowledge in form of software, which part III tried to prove by introducing a language and interpreter, as well as an application prototype using both.

In order to validate the results of this work, the following sub sections explain once again in short why many of the problems identified in today's programming language concepts are solved when applying CYBOP principles.

12.1.1 Distinction of Statics and Dynamics

A first major mistake in current language concepts and design solutions is the mix-up of static and dynamic parts of software, that is pure application-domain knowledge and its processing, close to hardware. It is the reason for:

- a Abstraction gap between designed system architecture and implemented source code, in a SEP (section 2.6)
- b Global data access via static class methods being insecure (section 4.2.3)
- c Bidirectional dependencies caused by some software patterns (section 4.2.2)
- d Usage of reflective techniques which are based on bidirectional dependencies and often cause broken type systems with circular references between super- and sub platform (section 4.2.1)
- e Spread functionality through crosscutting concerns and complicated handling of aspects (section 4.3.6)
- f Memory leaks
- g Repeated implementation of the same platform-dependent functionality
- h Repeated usage and copying of the same software patterns (section 7.2.1)

Chapter 6 therefore recommended a strict distinction of high-level static knowledge and low-level dynamic system control functionality. The CYBOL language (chapter 9) was defined to express and specify knowledge in the most general sense; the CYBOI interpreter (chapter 10) was created to control a system based on CYBOL input.

a Since CYBOL knowledge templates are a complete formal description of an application's architecture, they represent its implementation at the same time. But that also means that the transfer of a system's design into a programming language, as known from classical application development, becomes superfluous. The *Design*- and *Implementation* phases are merged, so that a gap does not exist anymore.

b Since CYBOI holds all knowledge in one single instance tree whose nodes can be accessed along well-defined paths, data are not globally accessible anymore.

c Since parts of a knowledge model are accessed unidirectionally, bidirectional dependencies are not an issue any longer.

d Since CYBOI is based on one standardised knowledge schema providing a well-defined type structure which does not have to be changed at runtime, there is neither a need nor a possibility for workarounds like reflective mechanisms causing a broken type system. Because the knowledge schema's whole-part hierarchy already provides meta information such as a part model's name and kind of abstraction (comparable to an attribute's name and type hold by the meta class of a class), one main reason for using reflective techniques like meta classes thereby falls apart. A second reason that does not count any longer is the provision of basic features (like persistence or communication) through meta techniques; CYBOI already contains these features and may act as universal communicator.

e Since CYBOI does provide all necessary low-level mechanisms, crosscutting concerns become superfluous. These concerns usually want to achieve the same as reflection in that they provide basic functionality to all parts of a system. However, since CYBOL knowledge templates are free from low-level system control information and contain pure domain knowledge instead, crosscutting concerns and aspects are not a topic of interest any longer.

f Since CYBOI concentrates all knowledge models (instances) in one place, as branches of one single knowledge tree, forgotten models can get smoothly destroyed at application shutdown. Traditionally, special mechanisms like *Garbage Collectors* (GC) had to be applied to achieve this, because systems written in classical languages leave it up to the programmer to properly reference all instances. If a reference to one instance was lost, it could not get destroyed and resided as leak in memory. Often, more and more memory space got blocked that way, until all RAM space was taken and a computer hung (crashed). In CYBOI, a reference to any instance is always available, via the root of the knowledge tree.

g Since CYBOI contains all hardware-dependent functionality, the application knowledge encoded in CYBOL templates or serialised models is truly platform-neutral, easily switchable and exchangeable among systems. The low-level system gets uninteresting; high-level knowledge is what application developers can now concentrate on. Finally, the old dream of having knowledge engineers (domain experts) working independently from software system engineers might possibly be coming true.

h Since CYBOI already implements all necessary patterns, application developers and domain experts are freed from the burden to learn and apply the same software patterns

again and again; they can now develop application systems considering just one concept: that of hierarchical *Composition*.

12.1.2 Usage of a Double-Hierarchy Knowledge Schema

A further problem that was identified in this work is the missing concept of hierarchy, which is not inherent in types of the corresponding languages. Moreover, knowledge structures are mixed up with meta information leading to:

- a Inflexible static typing in system programming languages (section 4.1.7)
- b Fragile base class problem when using inheritance (section 4.1.15)
- c Overly large source code due to encapsulation without sense (section 4.1.15)
- d Unpredictable behaviour and falsified contents due to container inheritance (section 4.1.15)
- e Redundant code caused by concerns and difficult application of an ontological structure (section 4.3.5)
- f Complicated, partly impossible serialisation of knowledge models

Chapter 7 therefore proposed a new knowledge schema which considers structural- as well as meta information, in two different hierarchies.

a Since type information is not fixed statically, it gets dynamically configurable at runtime, leading to highly flexible application systems. There is only one static data structure – the standardised knowledge schema. It holds meta information about the kind of abstraction (type) of the data contained in it.

b Since runtime knowledge models in CYBOI rely on composition only, the fragile base class problem caused by runtime type inheritance in object-oriented systems does not occur.

c Since the CYBOI-internal knowledge structure is a container by default, it also provides all necessary access procedures. Thousands of useless access methods as known from object-oriented programming are avoided. The partial security they provided can be replaced with other mechanisms. Since all knowledge resides in just one instance tree, it is easy to apply any kind of security checks, whenever a part of the knowledge tree gets accessed.

- d** Since each CYBOP knowledge template or -model is constructed as hierarchy, so that containers of any kind can be emulated, problematic container inheritance belongs to the past.
- e** Since CYBOP knowledge models are purely hierarchical, it gets easier to apply ontological structures which bundle functionality, instead of spreading it in a concern-like manner.
- f** Since all knowledge is modelled hierarchically, it is easily serialisable and hence exchangeable.

12.1.3 Separation of State- and Logic Knowledge

A third aspect causing troubles in software system design is the bundling of state- and logic knowledge, known from object-oriented programming. It results in:

- a Difficult handling and repeated implementation of the same communication mechanisms (section 3.13)
- b Differing patterns complicating the handling of communication (section 4.2)
- c Bidirectional dependencies (circular references) between classes/ objects in object-oriented systems, due to attribute-method bundling (section 4.2.2)
- d Pre-defined logic concepts in structured/ procedural- as well as object-oriented programming

Chapter 8 therefore suggested a separation of state- and logic concepts, in order to eliminate unnecessary inter-dependencies and to be able to apply a unified translator architecture.

- a** Since low-level communication mechanisms are implemented in CYBOI, application developers writing CYBOL knowledge templates do not have to bother with these anymore.
- b** Since standard communication patterns are unified, the handling of communication is simplified. Thanks to this unification, an extensible translator architecture can be applied. Using it, any kind of abstract knowledge model can be translated into any other. Universal communication becomes possible.

c Since state- are split from logic concepts, many (partly bidirectional) dependencies between knowledge models disappear, which reduces the coupling between- and increases cohesion within models. Both kinds use exclusively unidirectional relations. Additionally, logic- may access state models and each other, but always unidirectionally.

d Since logic concepts (algorithms, workflows) are themselves modelled as CYBOL knowledge templates, they become configurable. Traditionally, only structures representing states are manipulatable at runtime; procedures representing logic are fixed and cannot be altered.

12.2 Evaluation

Having validated the results of this work, their impact on software design and -engineering can be discussed and evaluated.

12.2.1 Knowledge Triumvirate

Chapter 7 introduced a new *Schema* for knowledge representation; chapter 9 defined a language for knowledge specification, in form of *Templates*; chapter 10 described a system for knowledge processing, that uses *Models*.

All three of them are closely connected (figure 12.1): The CYBOP knowledge *Schema* provides a structure for both, knowledge templates and -models; CYBOI *Models* are the dynamic runtime instances of static design-time CYBOL *Templates*.

One reason for the mix-up of domain knowledge and system control in traditional applications is the lack of a comprising solution for knowledge representation. Software systems always have to fall back to using programming paradigms that introduce more and more dependencies, as the system grows. John F. Sowa [294] writes:

The enhanced productivity of the *Fourth Generation Languages* (4GL), the *Object Oriented Programming Systems* (OOPS) and the *Computer Aided Software Engineering* (CASE) tools is derived from a common strength: improved methods of representing application knowledge in a form that can be used and reused by multiple system components. Their limitations result from a common

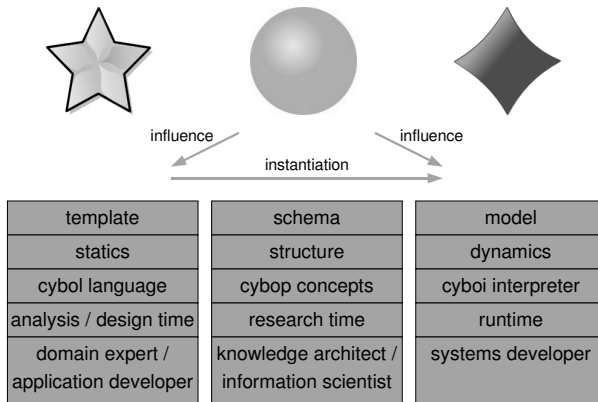


Figure 12.1: Knowledge Triumvirate with Schema, Template and Model

weakness: the inability to share that knowledge with systems that use a different representation. The potential for conflict is inevitable: sharing requires a common representation, but independently developed systems almost invariably use different representations. In order to support knowledge sharing among heterogeneous systems, the conceptual schema must be general enough to accommodate anything that can be represented in any current system, any legacy system inherited from the past, and any new system that may be developed in the future.

CYBOP claims to provide just that – a common schema for knowledge representation and -exchange. If not perfect, it is certainly a first step towards an all-general knowledge schema.

Section 4.1.9 contained a quote in which the *Association of Lisp Users* [227] states that, other than *Functional Programming*, *Procedural Languages* essentially performed everything as *Side Effects* (variable updates persisting after expression evaluation) to data structures. *A purely procedural language, after [227], would have no functions, but might have subroutines of no arguments that returned no values, and performed certain assignments and other operations based on the data it found stored in the system.* This is almost how CYBOI manipulates its knowledge – in the manner of one huge side effect. Its procedures do forward some parameters, but only one of these is really application-related: the *Knowledge Tree*.

State- and logic knowledge given in form of CYBOL templates are not bundled with low-level CYBOI procedures. Both are configurable knowledge. Logic knowledge models do not get input/ output (i/o) parameters handed over directly, but as dot-separated path to the corresponding value in the knowledge tree.

Because all knowledge is stored in tree-form, application systems become much more flexible than complex class networks as known from object-oriented programming. Tree structures are easy to edit, with or without supportive tools. They allow to better estimate necessary changes caused by new requirements, because dependencies are obvious. Software maintenance gets improved, because application developers can focus on pure domain knowledge; low-level system functionality is provided by CYBOI. CYBOL applications are therefore absolutely portable between platforms, as long as these were already considered in the underlying CYBOI. Due to the straight-forward possibility of accessing parts of a knowledge tree along well-defined paths, applications may win in performance.

12.2.2 Common Knowledge Abstraction

Although this work does not address the *Software Engineering Process* (SEP) directly, its results have great effect on it, which this section wants to shed some more light on. Classical SEP phases are: *Analysis*, *Design* and *Implementation* (chapter 2).

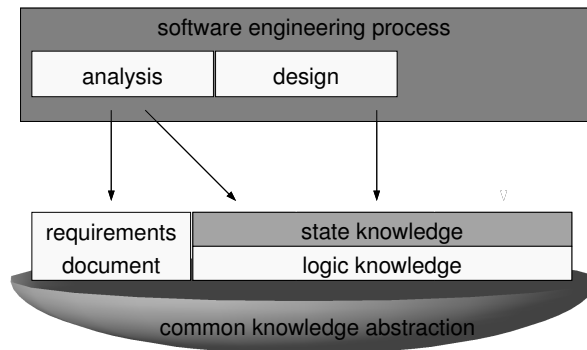


Figure 12.2: Common Knowledge Abstraction useable by many SEP Phases

Section 2.6 pointed out abstraction gaps and multiple development paradigm switches, happening during a software project's lifetime. It set out to find a *Common Knowledge Abstraction* for all phases. The results of this work overcome *Gap 2*, as shown in figure 2.6. Knowledge models as specified in this work can be used throughout most project phases (figure 12.2). Domain experts and application developers work with platform-neutral knowledge templates. Since these are interpreted, a translation into classical software source code is not necessary anymore. CYBOL knowledge templates represent the designed architecture *and* implemented application, at the same time. The formerly needed implementation phase thus disappears, which shortens the whole SEP. It is hard to estimate the amount of saved time and costs.

Using CYBOL, experts can hopefully yet more actively contribute to application development. Consequently, there may be less organisational problems in projects and companies, if experts and system developers can independently develop their parts (CYBOL vs. CYBOI) of the application system to be created. The whole SEP may become more *transparent* and *understandable*, and hopefully produce more *flexible*, *stable* and *secure* systems. However, the exact effort, especially to what concerns security issues cannot be estimated yet and has to be investigated further.

12.2.3 Long-Life Software System

The pure existence of proper knowledge does not suffice to create an improved kind of software system, within a slimmer software development process. The new systems need to know how to *handle* knowledge, at runtime. The criticism is twofold, because traditionally:

1. Operating systems do not have sufficient knowledge handling capabilities
2. Applications contain too much low-level system control functionality

This is changed when using CYBOP. The active CYBOI interpreter encapsulates memory allocation, persistence- and communication mechanisms, signal handling, logging facilities and more, which belong to the system level. While traditional programming philosophies try to make these *reusable*, CYBOI implements them just once, in a manner that *all* applications can access and use them. As a side-effect, the need for the study and repeated application of software patterns disappears. Additionally, and most importantly, CYBOI knows how to handle knowledge provided in form of passive CYBOL templates.

Although still dependent on an underlying operating system (for hardware device drivers and more), CYBOI is developing towards becoming one itself. Applications, on the other hand, do not have to care about communication paradigms and other low-level issues anymore; their focus is pure domain knowledge, encoded in CYBOL.

In classical type-based systems, no matter whether created in an SPP- or OOP language, the type of data needs to be known to find out about their structure and functionality. In CYBOP systems, all compound knowledge models have the same structure. Since they do not differ, they can be manipulated in the same manner. Only types (the kind of abstraction) of state primitives and logic operations need to be distinguished.

CYBOL knowledge templates, of which a *Clone* is made when a knowledge model (instance) gets created, are *not* treated as types. Once a knowledge model exists, its original knowledge template can never be accessed by it again, since a model holds no reference to its template. A template merely delivers the initial values for the model instantiated from it. After creation, a model exists on its own. It can be used and modified independently of any types, and is thus absolutely flexible.

But that also means that systems implementing the CYBOP knowledge schema are more future-proof. Unforeseeable requirements can be implemented anytime, without a static type model having to be changed, without fragile classes having to be considered, without dependencies causing existing functionality to break. CYBOP systems are therefore *Long-Life Systems* without architectural decay. Domain-/ application model changes do neither affect the structure of the knowledge schema, nor other parts of the static architecture of the underlying CYBOI interpreter.

The argument that systems developed in this manner were not safe because they lacked the constraints defined by a type does not hold, since also classical systems permit runtime objects to get manipulated, and to be assigned values not matching their type. In this case, of course, the system is alerted with an error, but in the end it is always the application developer who has to handle – or better prevent such errors.

12.3 Limits

Naturally, there are *Limits* to CYBOP. For instance, it:

- does not claim to be *the* approach for all kinds of programming problems, although

- it thinks to contribute suitable concepts for at least standard business application development. However, its usability for hardware-close systems with Real Time (RT) requirements, or for control engineering is questionable and yet to be investigated;
- depends on the existence of a system with knowledge-processing capabilities, which current *Operating Systems* (OS) are not. The CYBOI delivered with it is quite mature, but still lacks functionality like different *User Interfaces* (UI), various *import/ export* (i/e) filters/ translators, better error handling, prioritising and further OS features. Only functionality already implemented in CYBOI can also be used in CYBOL. But because CYBOI is free software, continuously developed in an open project, new features shall be implementable shortly;
 - has no type-checking features like classical compilers. This is the cost of flexibility. The knowledge schema is the only type structure provided by CYBOI. All domain- and application knowledge is held externally, in CYBOL knowledge templates, and interpreted only at runtime;
 - will have performance problems when using UI models, especially graphical ones, because these are sent in complete to the graphics adapter card, whenever a minor change is made. Techniques have to be found, that update only clips of a UI model, in graphics memory. The difficulty herein is that CYBOL application knowledge has *no* direct access to system-level functionality;
 - does not eliminate all abstraction gaps in a SEP. Requirements described informally by an analysis document have to be mapped to CYBOL knowledge templates, which then represent the application to be created. Although analysts and experts may create CYBOL models right from the project start, there will probably never be a true replacement for the written requirements analysis document, as one form of abstraction. However, if not the informal descriptions of its models, the document itself may be created in CYBOL, since it represents knowledge.

13 Summary and Outlook

To be Idealist means: Having Power for Others.

AFRICAN SAYING

The final remarks of this work sum up its content once more, before mentioning possible topics for future research.

13.1 Summary

As *Information Society* becomes reality and the collective knowledge of mankind grows, new ways for its storage, processing and communication have to be found. It is not that difficult to store simple data, pure information. It is much harder to store and reproduce structured data with meaningful associations, what makes up actual *Knowledge*.

This work reports about a new knowledge abstraction paradigm called *Cybernetics Oriented Programming* (CYBOP). It did not originally intend to create an all-new paradigm. Initially, its sole aim was to investigate and possibly improve existing concepts of software system design, by comparing them with principles found in other sciences besides informatics, and nature – as the name *Cybernetics* signifies.

Traditional programming concepts revealed a number of weak points, not all of which shall be mentioned here again. The previous chapter 12 reviewed them in detail. However, the most problematic ones, in short, are:

- Mix of static application (domain) knowledge and dynamic system control functionality
- Immaturity of schemas (types) ignoring hierarchical structure and mixing in meta information

- Bundling of states and logic, and inflexible logic (procedures) in current programming languages

It is them which cause unwanted, bidirectional inter-dependencies between layers of a software system, which make instance trees unnavigatable and static data access therefore necessary, which, together with a whole number of further issues, finally produce inflexible systems that are hard to maintain and thus prone to errors. Solving these frequently trouble-causing issues was a main motivation to write this work.

Following an interdisciplinary approach, several considerations of phenomenons as found in physics (dimensions), biology (human being as system), philosophy (structure of the universe), psychology (human thinking) and others delivered the ideas after which the theory behind CYBOP was conceived. Trusting these observations, CYBOP suggests three important points, namely: the distinction of *Statics and Dynamics*, the usage of a new *Knowledge Schema* with double hierarchy, and the separation of *State- and Logic* knowledge.

Since these are not consequently realised in today's programming environments, the new *Cybernetics Oriented Language* (CYBOL) and a corresponding *Cybernetics Oriented Interpreter* (CYBOI) had to be created, the latter actively managing and communicating knowledge formally specified in the former. Their general operability was proven through a minor, prototypic application called *Res Medicinae*. Certainly, plenty of extensions are still needed to make them all really useful.

Although not all consequences CYBOP has on software development and related fields could be considered, it surely affects the way application systems are created (abolished implementation phase). But CYBOP's extension and its embedding into a *Software Engineering Process* (SEP) are possible topics for future works.

First and last, CYBOP tries to deliver new ideas showing ways out of stagnation in software technology, what was called *Software Crisis* at the beginning of this work. It wants to improve some of the apparent deficiencies in current programming paradigms, and it may even have the potential to partly replace them. Software *can* become more flexible, clear, easy-to-understand and by this more reliable and better maintainable. Software development is not dead. On the contrary: it is further growing in importance and just starts to become interesting!

13.2 Future Works

The improvement of CYBOP's limits as mentioned in section 12.3, its extension and further issues not yet covered by it may form topics for future research. Some examples are given following.

Nature as Pool of Ideas for Software Design Further analogies between nature, sciences and software design/ informatics in general could be worth investigating. In particular the human brain and -mind seem to use interesting concepts which are not fully considered in software models to now. The *Hippocampus*, for example, is a part of the human brain that filters information by their importance and meaning [281]. In form of *Priorities*, a similar mechanism is already used by *Operating Systems* (OS). It is to be investigated in how far these filter mechanisms are suitable for handling general *Security* issues in software systems.

After all, software can only be as good as the models behind it. CYBOP's principles base on phenomenons of nature. Structures and their relations in space, time and further dimensions can only be implemented as well as they are currently known. It will be the task of natural sciences, philosophy, psychology and other fields to steadily improve their concepts so that software modelling can continue to learn from them.

The Applicability of CYBOP in Low-Level Systems The focus of CYBOP is standard business application development. Knowledge provided in form of CYBOL templates has a rather high abstraction level. It would be interesting to know in how far CYBOP concepts, the CYBOL language and the CYBOI interpreter are applicable in the development of *Real Time* (RT) systems, of control units in *Automation Engineering* (AE), and others more.

Surely, CYBOI's signal scheduling mechanism would have to be touched in such an investigation. In addition to the position property of a logic compound model's part, being used to place part signals in the correct order into the signal memory, a *Timestamp* (current time) could be used for this. It could determine the scheduled execution time. Possibly, such a timestamp could also serve as signal id.

In case CYBOI does not match RT requirements and -performance, at least CYBOL might be suitable for representing configuration knowledge appropriately. This may also count for the knowledge encoded by *Hardware Description Languages* (HDL).

Standardised Knowledge Templates for Various Domains Much effort has to go into the *Standardisation* of CYBOL templates for various domains such as *Transport*, *Telecommunication* or *Healthcare*. Actually, translator templates eliminate the need for a unification; every state- or logic model template can be translated into any other. Nevertheless, it seems very useful to provide internationally standardised model templates: They can serve as focal point, while developing a system. The more compliant different models are, the less translating is needed for data interchange between applications. For the greater part, this standardisation process has to be carried by domain experts; less by software specialists.

A closely related topic in this respect are *Terminologies*. Section 11.3.7 proposed to use CYBOL for terminology modelling, because it allowed to add meta information as well as to integrate constraints into its compositional structure. That way, problems like *Nonsense combinations* (section 4.6.6) or *Post-hoc Classification* (unforeseeable addition of new, unknown concepts that may prevent a meaningful data analysis) might be avoided. But this surely has to be figured out in future research works.

Besides for domain models like an *Electronic Health Record* (EHR) [22] for medicine, knowledge templates could also be defined for *User Interfaces* (UI), may they be textual, graphical or for the *World Wide Web* (WWW). Another kind of standardised template could be one for the creation of *Requirements Analysis Documents*. All templates would be exchangeable and reusable.

One point deserving special attention, is the handling of *Constraints* in CYBOL templates. Certainly, not all possible variants of constraints have been thought-out in this work. More details are needed.

Towards a CYBOI-based Operating System The CYBOI interpreter serves as connective link between CYBOL-encoded, passive application knowledge and its *input/ output* (i/o) via an underlying *Operating System* (OS) with communication facilities. The more low-level mechanisms CYBOI implements, the further it is developing towards becoming an OS itself. One topic of research could thus be to figure out which of the typical OS features [304, p. 80] are absolutely essential, and how these can be implemented into CYBOI.

Additionally, a number of questions are to be discussed. Is the *Process* concept- and are *Threads* really the ideal solution for running different applications on one computer? Or, are there ways to circumvent their usage and to base on just one signal-processing loop in the system? What are the possible algorithms for signal selection [183, p. 101]? What effects

would this have on security? If a system stores all knowledge of all applications running in it in just one memory, how can be made sure that an application accesses only the knowledge belonging to its scope? Apart from the *Process* concept, what other innovative ideas can be delivered for this? Would it make sense to assign special *Rights* to branches of the knowledge tree in memory? One possibility would be to always check the name of the application sending a signal and then allow it to access its own knowledge resources only. Another possibility could be to hold something like a resource access decision table in the low-level system, where each application is assigned the knowledge resources it may access. Further ideas are welcome and to be investigated.

In this context, how can concepts stemming from *Network Operating Systems* and *Distributed Operating Systems* [304, p. 16] be considered in CYBOI?

An OS does also contain configuration knowledge, such as the information whether or not a certain device driver should be loaded on a machine. This knowledge can be stored in form of CYBOL templates. Finally, the OS is nothing else than an application, only that it is the one started first in a system. A corresponding *Cybernetics Oriented Operating System* (CYBOS) could be created, whose structure and functionality are specified by CYBOL templates.

In a similar manner, OS in use today may be equipped with knowledge-processing capabilities. They would be the instance actively caring about low-level system control, as they should do exclusively, anyway, and factor out all configuration information into special files – or CYBOL templates. Further, hardware producers should be responsible for providing an OS suiting- and controlling their hardware. These systems would have to be able to handle well-specified knowledge, as in form of CYBOL templates, and hence be able to run applications accordingly.

The first version of CYBOI was written in *Java* and therefore carried some overhead. A second, much slimmer and more performant version (the current one) was written in the more low-level system programming language *C*. It will have to be investigated if and which parts of a third-version CYBOI could eventually be implemented in an *Assembler* language, to further increase performance. First preparations were already done in the current, second version: Only procedures, no functions were used; all parameters are handed over as neutral *void* pointers; only one kind of loop (endless, with break condition) was applied. Transformations into assembler code are thereby simplified. A fourth and final version of CYBOI would be one being burnt into *Hardware* directly, similar to the *Java Chips* [196] that people used to talk about some years ago.

A Development Process embedding CYBOP Principles It would be interesting to know how well the CYBOP principles fit into existing *Software Engineering Processes* (SEP), like the ones mentioned in chapter 2. Probably, an own SEP called something like *Cybernetics Oriented Methodology* (CYBOM) will have to be developed for CYBOP.

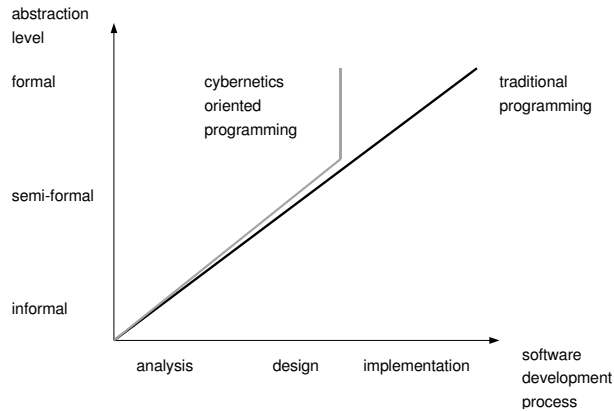


Figure 13.1: Comparison of a traditional SEP with CYBOM

Of special interest thereby is the missing implementation phase. Figure 13.1 shows a diagram comparing a traditional- with a cybernetics-oriented SEP. While the traditional SEP has to pass the three phases *Analysis*, *Design* and *Implementation*, CYBOM would stop at the end of the design phase, because its architected knowledge templates already represent the implementation, specified in a formal language – CYBOL.

However, the exact details are still *Speculation*. The figure is *not* based on research and just a *Guess* how a CYBOM might shorten the SEP time and effort. Further investigation and proof are needed. Also, it has to be figured out in how far missing CYBOI features have a delaying influence, since CYBOL can only make use of functionality that is already provided by CYBOI.

Strongly related with a CYBOM, is a plan for smoothly *migrating* systems that have been developed using *Structured- and Procedural Programming* (SPP) or *Object Oriented Programming* (OOP) techniques, to CYBOP. In conjunction with it, training methodologies, tutorials etc. are to be created.

CYBOP as Foundation for End User Development After [218], *End User Development* (EUD) is: *the collection of techniques and methodologies for the creation of non-trivial software applications by domain experts. End Users*, after the same source, are: *individuals who, although skilled in a task domain, lack the necessary computing skills or motivation to harness traditional programming techniques in support of their work*. Further on, [218] states that:

End User Development (EUD) has been the holy grail of software tool developers since James Martin launched *Fourth Generation Languages* (4GL) in the early eighties. Even though there has been considerable success in adaptive and programmable applications, EUD has yet to become a mainstream competitor in the software development market place. . . .

Today, there is increasing interest in the ability to rapidly evolve *Information Systems*, in response to changing opportunities and threats, but traditional development routes are prohibitively expensive. We believe that EUD techniques could be the source of a solution to this problem, by supporting the efficient development of flexible bespoke systems.

With CYBOL, this work claims to provide a language well-suitable for EUD. It is to be figured out how far end-user participation using CYBOP for application development can go. This could be done in form of case studies or the like.

13.3 Fiction

Lastly, there are some ideas which may seem a bit crazy or too fictionary for serious scientific work, which is why they were put into this extra section.

Dimensions *Time* is usually expressed as scalar product of *one* quantity and one unit. *Space* is represented by *three* such values. It would be interesting to know if *Mass* could possibly fill the gap in this order and be the one to be measured in *two* values. Latest research shows that *Gravitation* has probably a lot to do with *Geometry*. Why not so *Mass* as part of the definition of *Force*?

Are there other dimensions, besides time and space, in this order?

Structured DNA A *Desoxy Ribo Nucleic Acid* (DNA) molecule represents serialised knowledge. If it was possible to bring structure into the sequence of chemical bases that the DNA consists of, that is to recognise typical elements which all DNA strings contain (similar to the special tokens in a markup language), a corresponding knowledge schema might get created. One may assume that such a schema underlies each DNA molecule. It would help identify the meaning of the single DNA elements (*Genes*).

Teleportation One dream of research is *Beaming* (*Teleportation*), that is the transfer of matter in space. It seems very unlikely that big pieces of matter and their structures can be transferred. Looking at current transfer mode systems, it is not matter but abstractions of it which get transferred. Radio transfers acoustical signals; Television additional visual signals and so on. The visual signals consist of sequences of images which are abstract illustrations of some place or scene. These abstractions are transported in form of electromagnetic waves and later displayed on a screen by either electrons (cathod ray tube) or thin film transistors. Once again: not matter gets transferred but information.

Assuming that beaming really worked one day, it seems to become the easier part to build a new body after the model encrypted in a transmitted human *DNA*. (This would be the same as *Cloning* and in the same way, body damages could be repaired, too, which could lead to neverending life.)

A physical *Brain* might get copied or transferred, but how much more difficult would this be with the *Knowledge* contained in it? When building a body copy, billions of *Neurons* and their weighted *Connections* have to be duplicated identically. It is still not clear if the *Mind*/ *Character*/ *Personality* would then really be preserved.

For both problems (transfer of *Brain* and *Mind*), *Abstractions* are essential. They are a precondition for beaming to function correctly. The DNA represents the model of the body, abstracted to a chain of *Key-Value Pairs*. The Neurons and their connections as *Network of Weights* stand for the mind. If body and mind can be abstracted in complete, with no information-loss, *direct* knowledge exchange between human brains becomes possible. Also, body and mind can be stored in form of models, for a longer period of time.

14 Appendices

14.1 Abbreviations

3GL	Third Generation Language
4GL	Fourth Generation Language
AAFP	American Academy of Family Physicians
AAP	American Academy of Pediatrics
ABDA	Bundesvereinigung Deutscher Apotheker Verbände
ACID	Atomicity, Consistency, Isolation, Durability
ACL	Access Control List
ACL	Agent Communication Language
ACM	Association for Computing Machinery
ACR	American College of Radiology
AD	Activity Diagram
ADA	American Dental Association
ADL	Archetype Definition Language
ADL	Architecture Description Language
ADO	ActiveX Data Object
ADT	Abrechnungs DT
AE	Application Engineering
AE	Automation Engineering
AGOP	Agent Oriented Programming
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AM	Archetype Model
AMA	American Medical Association

AMR	Automated Medical Record
ANN	Artificial Neural Network
ANSI	American National Standards Institute
AODT	Ambulant Operieren DT
AOP	Aspect Oriented Programming
AOSD	Aspect Oriented SD
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASD	Adaptive Software Development
ASIC	Application Specific IC
ASTM	American Society for Testing and Materials
AWT	Abstract Window Toolkit
B2B	Business to Business
BASIC	Beginners All purpose Symbolic Instruction Code
B.C.	before Christ
BDT	Behandlungs DT
BeOS	Be, Inc. OS
BIOS	Basic I/O System
Bit	Binary Digit
BLOB	Binary LOB
BMC	BioMed Central
BNF	Backus Naur Form
BO	Business Object
BSD	Berkeley Software (System) Distribution (Design)
c/s	client/ server
CAD	Computer Aided Design
cADL	Constraint Form of ADL
CAM	Computer Aided Manufacturing
CAP	College of American Pathologists
CASE	Computer Aided Software Engineering
CBD	Component Based Design (Development)
CCR	Continuity of Care Record
CD	Communication Diagram
CDA	Clinical Document Architecture
CDISC	Clinical Data Interchange Standards Consortium

CEN	Comite Europeen de Normalisation (European Committee for Standardisation)
CENELEC	CEN Electrotechnique (European Committee for Electrotechnical Standardisation)
CERN	Conseil Europeen pour le Recherche Nucleaire
CGI	Common Gateway Interface
CIAS	Clinical Image Access Service
CICS	Customer Information Control System
CII	Computer Implemented Inventions
CISC	Complex Instruction Set Computing
CL	Common Lisp
CLOS	Common Lisp Object System
CLSI	Clinical and Laboratory Standards Institute
CmD	Component Diagram
CMET	Common Message Element Type
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
CMR	Computerised Medical Record
CNS	Central NS
COAS	Clinical Observations Access Service
COBOL	Common Business Oriented Language
CoD	Communication (Collaboration) Diagram
COM	Component Object Model
COP	Component Oriented Programming
CORBA	Common ORB Architecture
CP/M	Control Program for Microprocessors (Control Program/ Monitor)
CPR	Computer-based Patient Record (Computerised Patient Record)
CPU	Central Processing Unit
CR	CEN Report
CRC	Class, Responsibilities, Collaborations
CRM	Common Reference Model
CsD	Class Diagram
CSD	Composite Structure Diagram

CSS	Cascading Style Sheet
CSV	Comma Separated Variable
CT	Computer Tomograph
CTV3	Clinical Terms Version 3
CVS	Concurrent Versions System
CWM	Common Warehouse Metamodel
CYBOI	Cybernetics Oriented Interpreter
CYBOL	Cybernetics Oriented Language
CYBOM	Cybernetics Oriented Methodology
CYBOP	Cybernetics Oriented Programming
CYBORG	Cybernetic Organism
CYBOS	Cybernetics Oriented Operating System
dADL	Data Definition Form of ADL
DAG	Directed Acyclic Graph
DAML	DARPA Agent ML
DAO	Data Access Object
DARPA	Defense Advanced Research Projects Agency
DB	Database
DB2	DB 2
DBMS	DB Management System
DCC	Direct Client to Client Protocol
DCD	Document Content Description
DCE	Distributed Computing Environment
DCL	Data Control Language
DCOM	Distributed COM
DD	Deployment Diagram
DDE	Dynamic Data Exchange
DDL	Data Definition Language
DE	Domain Engineering
DEB	Debian GNU/Linux Package
DHTML	Dynamic HTML
DICOM	Digital Imaging and Communications in Medicine
DIF	Data Interchange Format
DIMDI	Deutsches Institut für Medizinische Dokumentation und Information

DIMSE	DICOM Message Service Element
DIN	Deutsches Institut für Normung
DLL	Dynamic Link Library
DML	Data Manipulation Language
DMP	Disease Management Programme
DMR	Digital Medical Record
DNA	Desoxy Ribo Nucleic Acid
DNS	Domain Name Service (Domain Name System)
DOM	Document Object Model
DOS	Disk OS
DPMI	DOS Protected Mode Interface
DSDM	Dynamic System Development Method
DSL	Domain Specific Language
DSOM	Distributed System Object Model
DSP	Digital Signal Processor
DSSSL	Document Style Semantics and Specification Language
DT	Datenträger
DTD	Document Type Definition
DTO	Data Transfer Object
DVI	Device Independent
e.g.	exempli gratia (for example)
EAA	Enterprise Application Architecture
EBES	European Board of EDI Standardisation
EBNF	Extended BNF
EC	Existential Conjunctive
ECC	Error Correction Code (Error Checking and Correction)
ECML	Electronic Commerce Modeling Language
EDI	Electronic Data Interchange
EDIF	EDI Format
EDIFACT	EDI for Administration, Commerce and Transport
EDP	Electronic Data Processing
EEG	EBES Expert Group
EEPROM	Electrically Erasable Programmable ROM

EET	Encyclopedia of Educational Technology
eHC	Electronic Health Card
EHCR	Electronic Health Care Record
EHR	Electronic Health Record
EHRcom	EHR Communications
EIA	Electronic Industries Alliance
EIR	Electronic Insurance Record
EJB	Enterprise Java Bean
EMI	Electronic Medical Infrastructure
EMR	Electronic Medical Record
EN	European Standard
ENV	European Prestandard
EPR	Electronic Patient Record
EPS	Encapsulated PS
ER	Endoplasmic Reticulum
ERD	Entity Relationship Diagram
ERM	Entity Relationship Model
ERP	Enterprise Resource Planning
et al.	et alii (and others)
etc.	et cetera (and so on)
EU	European Union
EUD	End User Development
Extended ML	Extended Meta Language
FAQ	Frequently Asked Question
FDD	Feature Driven Development
FDDI	Fiber Distributed Data Interface
FDL	Free Documentation License
FeatuRSEB	Feature RSEB
FHS	Filesystem Hierarchy Standard
FIFO	First In, First Out
FLOSS	Free/ Libre OSS
FODA	Feature Oriented Domain Analysis
FOLDOC	Free On-line Dictionary of Computing
FOPL	First Order Predicate Logic
FORE	Family Oriented Requirements Engineering

FOSS	Free and OSS
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FQN	Fully Qualified Name
FR	Frame Relay
FSF	Free Software Foundation
FTP	File Transfer Protocol
GALEN	Generalised Architecture for Languages, Encyclopaedias and Nomenclatures in Medicine
GC	Garbage Collector
GCC	GNUCompiler Collection (GNU CCompiler)
GDI	Graphics Device Interface
GDT	Geräte DT
GEHR	Good European/ EHR
GIF	Graphics Interchange Format
GIMP	General (GNU) Image Manipulation Program
GMDN	Global Medical Device Nomenclature
GNOME	GNU Network Object Model Environment
GNU	GNU is not UNIX
GoF	Gang of Four
GP	Generative Programming
GP	General Practitioner
GPF	General Protection Fault
GPIC	General Purpose Information Component
GPL	General Public License
GPL	General Purpose Language
GRAIL	GALEN Representation and Integration Language
GTK	GIMP Toolkit
GUI	Graphical UI
GUID	Globally Unique ID
h/w	Hardware
HAL	Hardware Abstraction Layer
HCI	Human-Computer Interaction
HD	Harmonisation Document

HDD	Hard Disk Drive
HDL	Hardware Description Language
HDTF	Healthcare Domain Task Force
HIMSS	Health Information Management and Systems Society
HIS	Hospital Information System
HL7	Health Level Seven
HMVC	Hierarchical MVC
HOWTO	How To? (Subject Specific Help)
HP	Hewlett Packard
HPC	Health Professional Card
HPTC	High Performance Technical Computing
HTML	Hypertext ML
HTTP	Hypertext Transfer Protocol
HTTPD	HTTP Daemon
HW	Hardware
HXP	Healthcare Xchange Protocol
i/e	import/ export
i/o	input/ output
i/p	input
IABG	Industrieanlagen-Betriebsgesellschaft
IBM	International Business Machines
IC	Integrated Circuit
ICANN	Internet Corporation for Assigned Names and Numbers
ICD	International Classification of Diseases
ICF	International Classification of Functioning, Disability and Health
ICHPPC	International Classification of Health Problems in Primary Care
ICN	International Council of Nurses
ICNP	International Classification for Nursing Practice (Procedures)
ICQ	I seek you
ICR	Integrated Care Record
ICPC	International Classification of Primary Care
ID	Identifier

IDE	Integrated Development Environment
IDL	Interface Definition Language
i.e.	id est (that is)
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IIM	Internet Interaction Management
IIOB	Internet Inter ORB Protocol
IIS	Internet Information Server
IMAP	Internet Message Access Protocol
Inc.	Incorporated
InterNIC	Internet Network Information Center
IoC	Inversion of Control
IOD	Interaction Overview Diagram
IP	Internet Protocol
IPC	Inter-Process Communication
IPv6	Internet Protocol (Version 6)
IPX	Internet Packet Exchange
IRC	Internet Relay Chat
IRQ	Interrupt Request
IS	International Standard
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
ISP	Internet Service Provider
IST	Information Science Technology
IT	Information Technology
ITU	International Telecommunication Union
J2EE	Java 2 Platform Enterprise Edition
JAR	Java Archive
JDBC	Java DB Connectivity
JDK	Java Development Kit
JEDEC	Joint Electron Device Engineering Council
JFC	Java Foundation Classes
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface

JOSMC	Journal of Free and Open Source Medical Computing
JPEG	Joint Photographic Experts Group
JPM	Join Point Model
JSP	Java Server Pages
JTS	Java Transaction Service
JVM	Java VM
KBV	Kassenärztliche Bundesvereinigung
KDE	K Desktop Environment
KDT	Kommunikations DT
KE	Knowledge Engineering
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
KV	Kassenärztliche Vereinigung
KVDT	KV DT
LAN	Local Area Network
LaTeX	Lamport TeX
LDR	Lifetime Data Repository
LDT	Labor DT
LGPL	Lesser GPL
LIFO	Last In, First Out
LILO	Linux Loader
LOB	Large Object
LOINC	Logical Observation Identifiers, Names and Codes
LQS	Lexicon (Terminology) Query Service
LTM	Long Term Memory
MAPI	Message Application Programming Interface
MAS	Multi Agent System
MathML	Mathematical ML
MATLAB	Matrix Laboratory
MBR	Master Boot Record
MD	Model Diagram
MD	Medical Doctor
MDA	Model Driven Architecture
MDI	Multiple Document Interface
MeSH	Medical Subject Headings

MFC	MS Foundation Classes
MIF	Management Information Format
MIME	Multipurpose Internet Mail Extension
MIS	Management Information System
MIT	Massachusetts Institute of Technology
ML	Markup Language
MMS	Massachusetts Medical Society
MMU	Memory Management Unit
MOF	Meta Object Facility
MOP	Meta Object Protocol
MPEG	Moving Picture Experts Group (Motion Picture Expert Group)
MPI	Message Passing Interface
MRPT	Management Resource Planning Tool
MS	Microsoft
MTU	Maximum Transmission Unit
Mutex	Mutual Exclusion
MVC	Model View Controller
MVS	Multiple Virtual Storage
n/a	not applicable
NC	Network Computer
NCCLS	National Committee for Clinical Laboratory Standards
NCPDP	National Council for Prescription Drug Programs
NEMA	National Electrical Manufacturers Association
NetBIOS	Network BIOS
NetDDE	Network DDE
NHS	National Health Service
NHSIA	NHS Information Authority
NIST	National Institute of Standards and Technology
NLM	National Library of Medicine
NNTP	Network News Transfer Protocol
NOS	Network OS
NPO	Non-Profit Organisation
NS	Nervous System
NSF	National Science Foundation

NSP	Network Service Provider
o/p	output
OASIS	Organization for the Advancement of Structured Information Standards
ObD	Object (Instance) Diagram
OCL	Object Constraint Language
OCX	OLE Custom Control
OD	Organisation Diagram
ODBC	Open DB Connectivity
ODI	Open Datalink Interface
ODL	Object Description Language
ODM	Operational Data Modeling
OFFIS	Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme
OGG	Ogg Vorbis Audio Encoding and Streaming Technology
OID	Object ID
OIM	Open Information Model
OIO	Open Infrastructure for Outcomes
OLE	Object Linking and Embedding
OM	Object Model
OMA	Object Management Architecture
OMG	Object Management Group
OMS	Object Model System
OO	Object Oriented (Object Orientation)
OOA	OO Analysis
OOD	OO Design
OODBMS	OO DBMS
OOM	OO Model
OOP	OO Programming
OOPS	OO Programming System
OPCS	Office of Population Censuses and Surveys Classification of Surgical Operations and Procedures
OPD	Object Process Diagram
OpenEHR	Open EHR

OPS	Official Production System
OQL	Object Query Language
ORB	Object Request Broker
ORDBMS	Object Relational DBMS
OS	Operating System
OSCAR	Open Source Clinical Application Resource
OSDN	Open Source Development Network
OSF	Open Software Foundation
OSHCA	Open Source Health Care Alliance
OSI	Open Source Initiative
OSI	Open Systems Interconnection
OSS	Open Source Software
OTW	Object Technology Workbench
OWiS	Objektorientierte und Wissensbasierte Systeme
OWL	Web Ontology Language
OXMIS	Oxford Medical Information System
P2P	Peer to Peer (Person-to-Person, Program-to-Program)
PAN	Personal Area Network
PAP	Password Authentication Protocol
PC	Personal Computer
PCL	Printer Control Language
PCMCIA	PC Memory Card International Association
PCR	Patient Carried Record
PD	Package Diagram
PDA	Personal Digital Assistant
PDF	Portable Document Format
PDL	Page Description Language
Perl	Practical Extraction and Report Language
PGP	Pretty Good Privacy
PhD	Philosophiae Doctor
PHP	PHP Hypertext Preprocessor (Personal Home Page)
PHP	Personal Health Project
PHR	Personal Health Record

PIC	Programmable Interrupt Controller
PIDS	Person (Patient) Identification Service
PIM	Platform Independent Model
PIM	Personal Information Manager
PIN	Personal Identification Number
PIO	Programmed Input Output
Pixel	Picture Element
PK	Public Key
PKI	PK Infrastructure
PL/1	Programming Language One
PLD	Programmable Logic Device
PMR	Patient Medical Record
PMS	Practice Management System
PNG	Portable Network Graphics
PnP	Plug and Play
PNS	Peripheral NS
POA	Portable Object Adapter
POL	Problem Oriented Language
POMR	Problem Oriented Medical Record
POP	Post Office Protocol
POSIX	Portable OS Interface for UNIX
PPC	Power PC
PPP	Point-to-Point Protocol
Prolog	Programmation en Logique
PS	PostScript
PSM	Platform Specific Model
QMS	Qualitätsring Medizinische Software
QoS	Quality of Service
Qt	Cute C++ Toolkit
RADS	Resource Access Decision Service
RAM	Random Access Memory
RAS	Remote Access Service
RAS	Reliability, Availability, Serviceability
RDBMS	Relational DBMS
RDF	Resource Description Framework

READ	Read Codes
RFC	Request for Comment
RFP	Request for Proposal
RIM	Reference Information Model
RISC	Reduced Instruction Set Computing
RKI	Robert Koch Institut
RM	Reference Model
RMI	Remote Method Invocation
RMIM	Refined Message Information Model
RNA	Ribo Nucleic Acid
ROM	Read Only Memory
RPC	Remote Procedure Call
RPM	RPM/ Red Hat Package Manager
RSEB	Reuse driven Software Engineering Business
RT	Real Time
RTE	Roundtrip Engineering
RTF	Rich Text Format
RTTI	Run Time Type Identification
RUP	Rational Unified Process
S/MIME	Secure MIME
s/w	software
SAS	Statistical Analysis System
SAX	Simple API for XML
SC	Subcommittee
SCD	State Chart Diagram
SCDI	Standards Committee on Dental Informatics
SCIPHOX	Standardisation of Communication between Information Systems in Physician's Offices and Hospitals using XML
SD	Software Development
SD	Sequence Diagram
SDK	System Development Kit
SDL	Specification and Description Language
SDM	Submission Data Modeling
SDO	Standards Development Organisation
SEP	Software Engineering Process

SEQUEL	Structured English Query Language
SET	Secure Electronic Transaction
SG	Secretary General
SGI	Silicon Graphics, Inc.
SGML	Standard Generalized ML
SHTTP	Secure HTTP
SID	Security ID
SIG	Special Interest Group
SL	Specification Language
SLIP	Serial Line Internet Protocol
SMB	Server Message Block
SMD	State Machine (Chart) Diagram
SME	Small- and Medium Sized Enterprise
SMIF	Stream based Model Interchange Format
SMP	Symmetric Multiprocessing (Shared Memory Processing)
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SNOMED	Systematized Nomenclature of Medicine
SNOMED CT	SNOMED Clinical Terms
SNOMED RT	SNOMED Reference Terminology
SNR	Signal to Noise Ratio
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOAP	Subjective, Objective, Assessment, Plan
SoC	Separation of Concerns
SOP	Script Oriented Programming
SOX	Schema for Object Oriented XML
SPID	Service Profile (Provider) ID
SPL	Special Purpose Language
SPP	Structured and Procedural Programming
SPX	Sequence Package Exchange
SQL	Structured Query Language
SSI	Server Side Include
SSL	Secure Socket Layer

STL	Standard Template Library
STM	Short Term Memory
SVG	Scalable Vector Graphics
SW	Software
SynEx	Synergy on the Extranet
SYSOP	System Operator
tADL	Template Form of ADL
TC	Technical Committee
Tcl	Tool command language
TCP	Transfer (Transport, Transmission) Control Protocol
TD	Template Diagram
TEI	Text Encoding Initiative
Telnet	Telephone Network
TiD	Timing Diagram
TIFF	Tagged Image File Format
Tk	Toolkit
tkFP	Tcl/Tk Family Practice
TLD	Top Level Domain
TLDP	The Linux Documentation Project
TP4	Transport Protocol Class 4
TPM	Third Party Maintenance
TR	Technical Report
TS	Technical Specification
TTF	True Type Font
TUG	TeX Users Group
TUI	Technical University of Ilmenau
TUI	Textual UI
TXT	Text
UCD	Use Case Diagram
UDP	User Datagram Protocol
UI	User Interface
UIML	UI ML
UIN	Universal Internet Number
UK	United Kingdom
UMDNS	Universal Medical Device Nomenclature System

UML	Unified Modeling Language
UMLS	Unified Medical Language System
UMLSKS	UMLS Knowledge Source Server
UN	United Nations
UNC	Universal Naming Convention
UNICODE	Universal, Unique, Uniform Character Set Standard
UNIX	Universal Interactive Executive (Uniplexed Information and Computing System)
UNO	Universal Network Objects
URI	Uniform Resource Indicator
URIref	URI reference
URL	Uniform Resource Locator
US	United States
USA	US of America
USB	Universal Serial Bus
USENET	User Network
USR	US Robotics
UUCP	UNIX to UNIX Copy
UUENCODE	UNIX to UNIX Encode
UUID	Universally Unique ID
VB	Visual Basic
VCL	Visual Component Library
VDAP	Verband Deutscher Arztpraxis Softwarehersteller
VDM	Vienna Development Method
VHDL	VHSIC HDL
VHitG	Verband der Hersteller von IT Lösungen für das Gesundheitswesen
VHR	Virtual Health Record
VHSIC	Very High Speed IC
VistA	Veterans Health Information Systems and Technology Architecture
VLAN	Virtual LAN
VM	Virtual Machine
VMM	Virtual Memory Manager
Voxel	Volume Element

VPN	Virtual Private Network
VPR	Virtual Patient Record
VR	Virtual Reality
vs.	versus (against)
VSA	Virtual Storage Architecture
VSAM	Virtual Storage Access Method
W3	WWW
W3C	W3 Consortium
WAIS	Wide Area Information Server
WAN	Wide Area Network
WAP	Wireless Application (Access) Protocol
WAR	Web Archive
WCS	World Coordinate System
WD	Working Document
WG	Working Group
WHO	World Health Organisation
WI	Work Item
WIMP	Windows, Icons, Menus, Pointing (Windows, Icons, Mouse, Pull-down Menus)
WINSOCK	Windows Socket
WINTEL	Windows/ Intel
WUI	Web UI
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
WYSIWYP	What You See Is What You Print
X	X Window System
XAML	Extensible Application ML
xDT	x DT
XHTML	Extensible HTML
Xlibs	X Libraries
XMI	XML Metadata Interchange
XML	Extensible ML
XP	Extreme Programming
XPath	XML Path Language
XSD	XML Schema Definition

XSL	Extensible Stylesheet Language
XSL-FO	XSL Formatting Objects
XSLT	XSL Transformations
XUL	XML UI Language (XUL)
YACC	Yet Another Compiler Compiler
ZI	Zentralinstitut für die Kassenärztliche Versorgung

14.2 References

- [1] Dragos Acostachioaie. Doc++ documentation system for c, c++, idl and java, December 2002. <http://docpp.sourceforge.net/>.
- [2] Canoo Engineering AG. Woerterbuecher und grammatik fuer deutsch. Internet Portal, 2004. <http://www.canoo.net>.
- [3] Christopher Alexander, Sara Ishikawa, Muray Silverstein, and et al. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977. <https://www.patternlanguage.com/cgi-bin/patternl/order.py>.
- [4] The Agile Alliance. To satisfy the customer through early and continuous delivery of valuable software. Website, 2004. <http://www.agilealliance.org>.
- [5] Scott W. Ambler. The design of a robust persistence layer for relational databases. Online White Paper, November 2000. <http://www.ambysoft.com/persistenceLayer.html>.
- [6] Scott W. Ambler. The diagrams of uml 2.0. Agile Modeling Essay, March 2005. <http://www.agilemodeling.com/essays/umlDiagrams.htm>.
- [7] APCB and the Z User Group. 4th international conference of b and z users. Website, November 2004. <http://www.zb2005.org/>.
- [8] Gerardo Arnaez. Linux medicine-howto. HOWTO Document, March 2004. <http://en.tldp.org/HOWTO/Medicine-HOWTO/>.
- [9] William (Bill) A. Arnett. The nine planets: A multimedia tour of the solar system, February 2005. <http://www.nineplanets.org/>.
- [10] G. Arrango. Domain analysis methods. In Schaefer, R. Prieto-Diaz, and M. Matsumoto, editors, *Software Reusability*, pages 17–49. Ellis Horwood, New York, 1994.
- [11] Tom Atwood. Vertical and horizontal computing architectures – trends and attributes. Whitepaper, Enterprise Systems Products, Sun Microsystems, Inc., SUPerG Berlin, May 2003. http://www.sun.com/datacenter/superg/docs/Atwood.SUPerG_Berlin.pdf.
- [12] Various Authors. Debian developers mailing list, 2004-2005. debian-devel@lists.debian.org.
- [13] John Backus, Peter Naur, and et al. Revised report on the algorithmic language algol 60. In Peter Naur, editor, *Communications of the ACM*, volume 3, no. 5, pages 299–314, May 1960. <http://www.masswerk.at/algol60/report.htm>.

- [14] Helmut Balzert. *Lehrbuch der Software-Technik*, volume Software-Entwicklung, no. 1 of *Lehrbuecher der Informatik*. Spektrum, Heidelberg, 2 edition, 2000. ISBN 3-8274-0480-2.
- [15] Subhashis Banerjee. Csl102: Introduction to computer science. Online Lecture Notes, 2004. <http://www.cse.iitd.ernet.in/suban/CSL102/history.html>.
- [16] Martha Argentina Barberena Najarro. *Visualisierung von Informationsraeumen*. PhD thesis, Technical University of Ilmenau, Ilmenau, June 2003. <http://www.bibliothek.tu-ilmenau.de/elektr.medien/dissertationen/2003/BarberenaNajarro.Martha/index.html>.
- [17] Federico Barbieri, Stefano Mazzocchi, and Pierpaolo Fumagalli. *Apache Jakarta Avalon Framework*. Apache Project, 2002. <http://avalon.apache.org/>.
- [18] Thomas Beale. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. Open Electronic Health Record (OpenEHR), 2.2.1 edition, August 2001. <http://www.deepthought.com.au/it/archetypes/archetypes.pdf>.
- [19] Thomas Beale and et al. Openehr technical mailing list, 2002-2004. openehr-technical@openehr.org.
- [20] Thomas Beale, A. Goodchild, and Sam Heard. *Design Principles for the EHR*. Open Electronic Health Record (OpenEHR), 2.4 edition, 2002. <http://www.openehr.org/Doc.html/Model/Principles/design-principles.htm>.
- [21] Thomas Beale and Sam Heard. *Archetype Definition Language (ADL)*. The openEHR Foundation, release 1.0 draft, revision 2.0rc1 edition, September 2005. <http://svn.openehr.org/specification/BRANCHES/Release-1.0-candidate/publishing/index.html>.
- [22] Thomas Beale, Sam Heard, and et al. Open electronic health record (openehr) project, formerly good european/ electronic health record (gehr), April 2005. <http://www.openehr.org>.
- [23] Dirk Behrendt. Erstellen eines backup-moduls unter verwendung gaengiger design patterns aus dem resmedlib framework. Student project (studienarbeit), Technical University of Ilmenau, Ilmenau, January 2003. <http://www.cybop.net>.
- [24] David Belton. Boolean algebra. Website, April 1998. <http://www.ee.surrey.ac.uk/Projects/Labview/boolalgebra/>.
- [25] Michael Benedikt. Cyberspace: Some proposals. In *Cyberspace: First Steps*, pages 119–224. MIT Press, Cambridge, MA, USA, 1991.

- [26] Jon Bentley. *Perlen der Programmierkunst. Programming Pearls*. Addison-Wesley, <http://www.aw.com>, Boston, Muenchen, 2000.
- [27] Ambrose Bierce. *The Devil's Dictionary*. Albert and Charles Boni, Inc., internet wiretap 1st online edition, April 1993. <http://wiretap.area.com/Gopher/Library/Classic/devils.txt>.
- [28] Remigiusz Bierzanek and Janusz Symonides. *Prawo Międzynarodowe Publiczne (citing S. E. Nahlik)*. Wydawnictwa Prawnicze (PWN), Warszawa, 5th edition, 1999. <http://www.wp-pwn.com.pl>.
- [29] *Java Blueprints*.
- [30] Kai Boellert. *Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen*. PhD thesis, Technical University of Ilmenau, Ilmenau, January 2002. http://www.bibliothek.tu-ilmenau.de/elektr_medien/dissertationen/2003/Boellert_Kai/index.html.
- [31] Jens Bohl. *Moeglichkeiten der gestaltung flexibler software-architekturen fuer praesentationsschichten, dargestellt anhand episodenbasierter, medizinischer dokumentation unter einbeziehung topologischer befundung*. Master's thesis (diplomarbeit), Technical University of Ilmenau, Ilmenau, January 2003. <http://www.cybop.net>.
- [32] G. Booch, J. Rumbaugh, and I. Jacobson. *Das UML-Benutzerhandbuch*. Addison-Wesley, Bonn, 1999. Original: The Unified Modeling User Guide.
- [33] Jonathan Bowen and et al. The z notation, November 2004. <http://www.afm.sbu.ac.uk/z/>.
- [34] Tim Bray, Charles Frankston, and Ashok Malhotra. *Document Content Description (DCD)*. World Wide Web Consortium (W3C), submission edition, July 1998. <http://www.w3.org/TR/NOTE-dcd>.
- [35] Alan Brown. An introduction to model driven architecture – part i: Mda and today's systems. Technical report, Inc., International Business Machines, Developerworks, January 2004. <http://www-106.ibm.com/developerworks/rational/library/3100.html>.
- [36] Alan W. Brown. *Large-Scale, Component-Based Development*. Object and Component Technology Series. Prentice Hall PTR, London, Sydney, 2000. <http://www.phptr.com>.
- [37] Christopher B. Browne. Alternatives to corba. Home Page, 2004. <http://cbbrowne.com/info/corbaalternatives.html>.
- [38] Eric Browne. The myth of self-describing xml, September 2003. <http://www.OceanInformatics.biz/publications/e2.pdf>.

- [39] Andreas Buesch. Vorlesung medienpaedagogik & kommunikationswissenschaft i. PDF Document, Katholische Fachhochschule Mainz, June 2003. <http://www.kfh-mainz.de/downloads/sasp/buesch/>.
- [40] Rainer S. Burkhardt. *UML – Unified Modeling Language: Objektorientierte Modellierung fuer die Praxis*. Programmer's Choice. Addison-Wesley, 2nd edition, March 1999. <http://www.addison-wesley.de>.
- [41] Frank Buschmann, Regine Meunier, Hans Rohnert, and et al. *Pattern-orientierte Softwarearchitektur. Ein Pattern-System*. Addison-Wesley, Bonn, Boston, Muenchen, 1. korr. nachdruck 2000 edition, 1998. <http://www.aw.com/>.
- [42] Jennifer Bush. Open source software: Just what the doctor ordered? *Family Practice Management*, 10(6):65–69, June 2003. <http://www.aafp.org/fpm/20030600/65open.html>.
- [43] Jason Cai, Ranjit Kapila, and Gaurav Pal. Hmvc: The layered pattern for developing strong client tiers. *Java World Online Magazine*, July 2000. <http://www.javaworld.com/javaworld/jw-07-2000/>.
- [44] Software Engineering Institute Carnegie Mellon University. Domain engineering: A model-based approach. Website containing technical Reports, 2003. <http://www.sei.cmu.edu/domain-engineering/domain.engineering.html>.
- [45] Software Engineering Institute Carnegie Mellon University. Capability maturity model for software (sw-cmm) and capability maturity model integration (cmmi). Website containing technical Reports, July 2004. <http://www.sei.cmu.edu/cmmi/>.
- [46] Software Engineering Institute Carnegie Mellon University. Feature-oriented domain analysis (foda). Website containing technical Reports, August 2004. <http://www.sei.cmu.edu/domain-engineering/FODA.html>.
- [47] Robert Todd Carroll, editor. *Skeptic's Dictionary*. John Wiley & Sons, August 2003. <http://skepdic.com/>.
- [48] Chaos Computer Club (CCC). Europe's largest hacker group, 2004. <http://www.wauland.de/datagarden.html>.
- [49] Clinical Data Interchange Standards Consortium (CDISC). Standards with focus on the global, platform-independent exchange of various data. Standardisation Effort, July 2005. <http://www.cdisc.org/>.
- [50] CEN Technical Committee 251 (CEN/TC251). European standardization of health informatics. TC Documents, 2004. <http://www.centc251.org/>.

- [51] Technische Universitaet Chemnitz. Dictionary. Free Online German-English Dictionary. <http://dict.tu-chemnitz.de/>.
- [52] Eric H. Chudler. Adventures in neuroanatomy: Parts of the nervous system. Neuroscience for Kids, December 2004. <http://faculty.washington.edu/chudler/introb.html>.
- [53] P. Clements. A survey of architecture description languages. In *8th International Workshop in Software Specification and Design*, Germany, 1996.
- [54] Clinical and Laboratory Standards Institute (CLSI) formerly National Committee for Clinical Laboratory Standards (NCCLS). Consensus documents and guidelines for clinical laboratory testing and -systems. Standardisation Effort, July 2005. <http://www.clsi.org/>.
- [55] Edgar F. (Ted) Codd. A relational model of data for large shared data banks. *Communications of the Association for Computing Machinery (ACM)*, 13(6):377–387, June 1970. <http://www.acm.org/classics/nov95/toc.html>.
- [56] Codehaus. Pico container, 2003-2004. <http://www.picocontainer.org/>.
- [57] OASIS UIML Technical Committee. User interface markup language (uiml). Specification, 1999-2004. <http://www.uiml.org/>.
- [58] The XFree86 Developers Community. Xfree86 – the open source x window system, June 2004. <http://www.xfree86.org/>.
- [59] SynEx Consortium. Synergy on the extranet (synex), successor of the synapses project. EU-sponsored Project, October 2000. <http://www.gesi.it/synex/suite.htm>.
- [60] Collaborating contributors from around the world. Wikipedia – the free encyclopedia. Web Encyclopedia, October 2004. <http://www.wikipedia.org>.
- [61] Tim Cook and et al. Torch project (formerly: Freepm), 1999-2004.
- [62] J. O. Coplien. *Advanced C++ – Programming Styles and Idioms*. Addison-Wesley, Bonn, Boston, Muenchen, 1992.
- [63] Borland Software Corporation. Delphi language and kylix/ delphi integrated development environment, November 2004. <http://www.borland.de/kylix/>.
- [64] Microsoft Corporation. Visual basic language and integrated development environment, November 2004. <http://msdn.microsoft.com/vbasic/>.
- [65] Robin Cover. Xml and semantic transparency. Technical report, Cover Pages hosted by OASIS, November 1998. <http://xml.coverpages.org/xmlAndSemantics.html>.

- [66] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Principles and Techniques of Software Engineering based on automated Configuration and Fragment-based Component Models*. Addison-Wesley, Boston, Muenchen, 2000. <http://www.aw.com>.
- [67] Bernd Daene. Personal talk, September 2004.
- [68] Stephen H. Daniel. Notes for ontology i: Dualism & behaviorism. Lecture Notes, 2003. <http://www-phil.tamu.edu/~sdaniel/Notes/dualism.html>.
- [69] Andrew Davidson, Matthew Fuchs, and et al. *Schema for Object Oriented XML (SOX)*. World Wide Web Consortium (W3C), 2.0 edition, July 1999. <http://www.w3.org/TR/NOTE-SOX/>.
- [70] L.L.C. dbXML Group. dbxml (native xml database), 2004. <http://www.dbxml.com/product.html>.
- [71] Wissenschaftlicher Rat der Dudenredaktion: Guenther Drosdowski ..., editor. *Der Duden: in 12 Baenden; das Standardwerk zur deutschen Sprache*, volume Duden, Rechtschreibung der deutschen Sprache. Dudenverlag, Mannheim; Leipzig; Wien; Zuerich, 21st edition, 1996. <http://www.duden.de/>.
- [72] Bereich Informatik der Technischen Universitaet Muenchen. Leo dictionary. Free Online German-English Dictionary. <http://dict.leo.org/>.
- [73] Design Matrix – Systems and Product Design, <http://www.designmatrix.com/bionics/>. *Design Matrix*.
- [74] Doulos Ltd. *Doulos Golden Reference Guides (GRG)*, 2004. http://www.doulos.com/de/grg_de/.
- [75] Gert Egle. Fachbereich deutsch / linguistik. teachSam Bildungsserver, 2000. http://www.teachsam.de/deutsch/d_lingu/lin0.htm.
- [76] Task Force EHRCOM. Ehrcom pren 13606-1 – 2nd working draft. Working Draft 1.2, Swedish Standards Institute (SIS), Stockholm, March 2004. <http://www.centc251.org/TCMeet/doclist/TCdoc04/N04-012prEN13606-1.2WD.pdf>.
- [77] Albert Einstein. *Ueber die spezielle und die allgemeine Relativitaetstheorie*. Vieweg, Wiesbaden, 1992. <http://www.einstein-website.de>.
- [78] Chris Eliasmith, editor. *Dictionary of Philosophy of Mind (PoM)*. Department of Philosophy, Washington University, St. Louis, May 2004. <http://www.artsci.wustl.edu/philos/MindDict/>.

- [79] Information Technology for European Advancement (ITEA) Project 99005 Eureka! 2023 Programme. Engineering software architectures, processes and platforms for system families (esaps), September 2001. <http://www.esi.es/en/Projects/esaps/overview.html>.
- [80] HL7 Deutschland e.V. and Qualitaetsring Medizinische Software (QMS). Standardisation of communication between information systems in physician's offices and hospitals using xml (sciphox). Standardisation Effort, July 2005. <http://www.sciphox.de/>.
- [81] KDE Project (e.V.). K desktop environment (kde). Open Source Project, August 2004. <http://www.kde.org>.
- [82] LinuxTag e.V. Linux tag linux expo and conference, June 2005. <http://www.linuxtag.org/>.
- [83] Martin Fache. Recherche und evaluierung eines konzeptes fr graphische benutzer-oberflaechen in cybol. Seminar work (hauptseminararbeit), Technical University of Ilmenau, Ilmenau, January 2004. <http://www.cybop.net>.
- [84] X. Fere and S. Vegas. An evaluation of domain analysis methods. In *4th CAiSE/IFIP8.1 International Workshop in Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD99)*, Heidelberg, Germany, 1999.
- [85] David Gomez Fernandez. The small astronomer's corner, February 2002. <http://usuarios.lycos.es/davidgomezfernandez/Ingles/principalpage.html>.
- [86] Guillen Fernandez and Bernd Weber. Hirnforschung: Gedaechnis – fische fangen im erinnerungsnetz. In *Gehirn & Geist*, volume 2, pages 68–73. Spektrum der Wissenschaft, <http://www.spektrum.de>, 2003. <http://www.gehirn-und-geist.de>.
- [87] Tim Finin and et al. Knowledge query and manipulation language (kqml), November 2004. <http://www.cs.umbc.edu/kqml/>.
- [88] Tim Finin, Yannis Labrou, and James Mayfield. Kqml as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, 1997. <http://www.cs.umbc.edu/kqml/papers/kqmlacl.pdf>.
- [89] Peter Flass. The pl/i language, 2002. <http://home.nycap.rr.com/pflass/pli.htm>.
- [90] Fraunhofer FOKUS. Berlios developer. Free and Open Source Software Development Portal, 2002. <http://developer.berlios.de/>.
- [91] Foundation for a Free Information Infrastructure. Call for action ii. Internet Page, May 2004. <http://swpat.ffii.org/>.

- [92] National Council for Prescription Drug Programs (NCPDP). Various electronic standards for the transmission of pharmacy data. Standardisation Effort, July 2005. <http://www.ncdp.org/>.
- [93] American Society for Testing and Materials (ASTM) International Subcommittee E31.28. *WK4363 Standard Specification for the Continuity of Care Record (CCR)*. ASTM International, Massachusetts Medical Society (MMS), Health Information Management and Systems Society (HIMSS), American Academy of Family Physicians (AAFP), American Academy of Pediatrics (AAP), American Medical Association (AMA), draft edition, March 2004. <http://www.astm.org/cgi-bin/SoftCart.exe/DATABASE.CART/WORKITEMS/WK4363.htm?L+mystore+lttx3162>.
- [94] David Forslund and et al. Openemed project, 2005. <http://www.openemed.org>.
- [95] Free Software Foundation. Savannah. Free and Open Source Software Development Portal, 2000-2003. <http://savannah.gnu.org/>.
- [96] The GNOME Foundation. Gnu network object model environment (gnome). Open Source Project, 2004. <http://www.gnome.org/>.
- [97] Martin Fowler. *Analysis Patterns – Reusable Object Models*. Addison-Wesley, Boston, Muenchen, 1997. <http://www.aw.com>.
- [98] Martin Fowler. The new methodology. Web Article, April 2003. <http://martinfowler.com/articles/newMethodology.html>.
- [99] Martin Fowler. Domain specific language (dsl). Web Article, February 2004. <http://martinfowler.com/bliki/DomainSpecificLanguage.html>.
- [100] Martin Fowler. Inversion of control containers and the dependency injection pattern. Web Article, January 2004. <http://www.martinfowler.com/articles/injection.html>.
- [101] Martin Fowler and et al. *Patterns of Enterprise Application Architecture (Information Systems Architecture)*. Addison-Wesley, Boston, Muenchen, 2001-2002. <http://www.aw.com>.
- [102] David S. Frankel. Model driven architecture (mda) – reality and implementation. Online Presentation, 2001. http://www.omg.org/mda/mda_files/DFrankel.MDA_v01-00_PDF.pdf.
- [103] Artists from around the world. Open clip art library. Archive of user contributed clip art that can be freely used, 2005. <http://www.openclipart.org/>.

- [104] Free Software Foundation (FSF). Gnu- and other licenses. Website, April 2005. <http://www.gnu.org/licenses/licenses.html>.
- [105] Deutsches Institut fuer Medizinische Dokumentation und Information (DIMDI). Various medical specification documents. Website, July 2005. <http://www.dimdi.de/>.
- [106] Regionales Rechenzentrum fuer Niedersachsen/ Universitaet Hannover, Zentralinstitut fuer Angewandte Mathematik, and Forschungszentrum Juelich GmbH. Die programmiersprache c, March 1991.
- [107] Anne-Kathrin Funkat and Gert Funkat. *Prozessbasiertes Knowledge Engineering in medizinischen Problemdomaenen*. PhD thesis, Technical University of Ilmenau, Ilmenau, 2003. http://www.bibliothek.tu-ilmenau.de/elektr_medien/dissertationen/2003/Funkat_Anne_Kathrin_Gert/index.html.
- [108] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang Of Four). *Design Patterns. Elements of reusable object oriented Software*. Addison-Wesley, Bonn, Boston, Muenchen, 1st edition, 1995. <http://www.aw.com>.
- [109] D. Garlan. Software architecture: A roadmap. In *The Future of Software Engineering*, pages 91–101. ACM Press, 2000.
- [110] Robert A. Gehring and Bernd Lutterbeck, editors. *Open Source Jahrbuch 2004 – Zwischen Softwareentwicklung und Gesellschaftsmodell*. Lehmanns Media – LOB.de, Berlin, 2004. <http://www.think-ahead.org>.
- [111] Michael R. Genesereth. Knowledge interchange format (kif). draft proposed American National Standard (dpANS); NCITS.T2/98-004. <http://logic.stanford.edu/kif/dpans.html>.
- [112] James Gosling, Bill Joy, Guy Steele, and et al. *The Java Programming Language Specification; The Java Development Kit (JDK)*. Sun Microsystems, Inc., Santa Clara, 2nd edition, 1996-2000. <http://java.sun.com>.
- [113] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. Wiley Publishing, Hoboken, Indianapolis, Weinheim, 2nd edition, October 2002. <http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471227293.html>.
- [114] Mark Greaves and et al. Darpa agent markup language (daml) and ontology inference layer (oil). Lecture Notes, 2004. <http://www.w3.org/TR/2001/NOTE-daml+oil-reference-20011218>.

- [115] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the rseb. *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, June 1998. <http://www.intecs.it>.
- [116] Ontology.org Group. Ontology.org independent industry and research forum. Internet Web Site, 2000. <http://www.ontology.org/>.
- [117] World Wide Web Consortium (W3C) Math Working Group. Mathematical markup language (mathml) 2.0 recommendation. Online Specification, October 2003. <http://www.w3.org/TR/2003/REC-MathML2-20031021/>.
- [118] T. R. Gruber. *A Translation Approach to Portable Ontology Specifications*. Academic Press, June 1993. <http://www-ksl.stanford.edu/kst/what-is-an-ontology.html>.
- [119] Volker Gruhn and Andreas Thiel. *Komponentenmodelle. DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Addison-Wesley, Boston, Muenchen, 2000. <http://www.aw.com>.
- [120] Markus Gumbel, Marcus Vetter, and Carlos Cardenas. *Java Standard Libraries: Java 2 Collections Framework und Generic Collection Library for Java*. Professionelle Programmierung. Addison-Wesley, Muenchen, Boston, San Francisco, an imprint of pearson education edition, 2000.
- [121] Graham Hamilton, Rick Cattell, and Maydene Fisher. *JDBC Database Access with Java*. Addison-Wesley, Reading/ Mass., Bonn, Boston, Muenchen, 1997.
- [122] Jens Hartwig. *PostgreSQL, Professionell und praxisnah*. Addison-Wesley, Muenchen, 2001.
- [123] Geoff Haselhurst. On truth and reality. Philosophical Website, March 2004. <http://www.spaceandmotion.com/>.
- [124] Christian Heller. Schlussbericht / ergebnisbericht. Vorhabensbezeichnung: Nutzung von Domänen-Engineering-Techniken zur Entwicklung objektorientierter Systeme mit Anbindung an hostbasierte Legacy-Systeme im Versicherungswesen, September 2003.
- [125] Christian Heller. Cybernetics oriented language (cybol). *IIIS Proceedings: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004)*, V:178–185, July 2004. <http://www.iiisci.org/sci2004> or <http://www.cybop.net>.
- [126] Christian Heller, Jens Bohl, Torsten Kunze, and Ilka Philippow. A flexible software architecture for presentation layers demonstrated on medical documentation with

- episodes and inclusion of topological report. *Journal of Free and Open Source Medical Computing (JOSMC)*, 1(26.06.2003):Article 1, June 2003. <http://www.josmc.net>.
- [127] Christian Heller, Torsten Kunze, Jens Bohl, and Ilka Philippow. A new concept for system communication. *Ontology Workshop at OOPSLA Conference*, October 2003. <http://swt-www.informatik.uni-hamburg.de/conferences/oopsla2003-workshop-position-papers.html>.
- [128] Christian Heller, Periklis Sochos, and Ilka Philippow. Reflexions on knowledge modelling. Paper on CYBOP Website, March 2006. <http://www.cybop.net>.
- [129] Christian Heller, Detlef Streitferdt, and Ilka Philippow. A new pattern systematics. Paper on CYBOP Website, March 2005. <http://www.cybop.net>.
- [130] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, Elsevier Science, Amsterdam, Boston, 3rd edition, 2003. <http://www.mkp.com>.
- [131] Horst Herb, Karsten Hilbert, and et al. Gnumed project, April 2005. <http://www.gnumed.org/>.
- [132] Gilbert Carl Herschberger II, Jonathon Tidswell, Stephen Crawley, and et al. The jos-general mailing list. jos-general@lists.sourceforge.net.
- [133] Hewett, Baecker, Card, and et al. Curricula for human-computer interaction (hci). Technical report, ACM SIGCHI, June 2004. <http://sigchi.org/cdg/index.html>.
- [134] Francis Heylighen. *Web Dictionary of Cybernetics and Systems. Principia Cybernetica Web*. Internet, 2002. <http://pespmc1.vub.ac.be/ASC/>.
- [135] Karsten Hilbert, Christian Heller, Roland Colberg, and et al. *Analysedokument zur Erstellung eines Informationssystems fuer den Einsatz in der Medizin*. Res Medicinae Free Software Project, 2001-2004. <http://resmedicinae.sourceforge.net/analysis/index.html>.
- [136] Andrew P. Ho and et al. Open infrastructure for outcomes (oio) project and reading material library, 2002. <http://www.txoutcome.org>.
- [137] Bob Hoffman and et al. Encyclopedia of educational technology (eet). Web Encyclopedia, January 2004. <http://coe.sdsu.edu/eet/>.
- [138] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, December 1999.
- [139] Herwart (Wau) Holland-Moritz. Der datengarten. Internet Website, 2003. <http://www.wauland.de/datagarden.html>.

- [140] Rolf Holzmueller. Erstellung intuitiver web frontends zur terminplanung und verwaltung administrativer daten, basierend auf einem webserver mit jsp technologie, sowie anbindung an ein medizinisches informationssystem. Student project (studienarbeit), Technical University of Ilmenau, Ilmenau, April 2003. <http://www.cybop.net>.
- [141] Rolf Holzmueller. Untersuchung der realisierungsmoeglichkeiten von cybol-webfrontends, unter verwendung von konzepten des cybernetics oriented programming (cybop). Master's thesis (diplomarbeit), Technical University of Ilmenau, Ilmenau, June 2005. <http://www.cybop.net>.
- [142] Cover Pages hosted by OASIS. Sgml and xml as (meta-) markup languages. Internet Article, July 2002. <http://xml.coverpages.org/sgml.html>.
- [143] Denis Howe. Free on-line dictionary of computing (foldoc). Internet Database, September 2003. <http://wombat.doc.ic.ac.uk/foldoc/Dictionary.gz>, <http://www.foldoc.org/>.
- [144] Peter Hrastnik. Comparison of distributed system technologies for e-business. *2nd International Interdisciplinary Conference on Electronic Commerce (ECOM-02)*, Electronic Commerce: Theory and Applications:49–56, November 2002. http://parlevink.cs.utwente.nl/ecom/02/proceedings/ecom02_07.pdf.
- [145] Stephan Huttenhuis and Nick Tinnemeier. The join point model (jpm) in aspect oriented programming (aop). <http://wwwhome.cs.utwente.nl/tinnemeiernam/JPM.pdf>, March 2004.
- [146] Graham Hutton. Frequently asked questions for comp.lang.functional. University of Nottingham, November 2002. <http://www.cs.nott.ac.uk/gmh/faq.html#functional-languages>.
- [147] Industrieanlagen-Betriebsgesellschaft (IABG). Das v-modell: Entwicklungsstandard fr it-systeme des bundes. Website, 2004. <http://www.v-modell.iabg.de/>.
- [148] Cunningham & Cunningham Inc. Portland pattern repository, 2004. <http://c2.com/cgi/wiki?PortlandPatternRepository>.
- [149] Free Software Foundation Inc. Gnu operating system, 2004. <http://www.gnu.org>.
- [150] Health Level Seven Inc. Reference information model (rim), clinical document architecture (cda) and more, 2004. <http://www.hl7.org/>.
- [151] International Business Machines Inc. <http://www.ibm.com>.
- [152] OTW Software Inc. and formerly Objektorientierte und Wissensbasierte Systeme (OWiS). Object technology workbench (otw). UML Tool, June 2000. <http://www.otwsoftware.com/>.

- [153] Silicon Graphics Inc. Standard template library (stl), 2004. <http://www.sgi.com/tech/stl>.
- [154] Sun Microsystems Inc. Javadoc source code documentation tool, February 2004. <http://java.sun.com/j2se/javadoc/>.
- [155] Wolfram Research Inc. Mathematica – computer algebra system, November 2004. <http://www.wolfram.com/products/mathematica/index.html>.
- [156] American National Standards Institute. Ansi c standard. <http://www.ansi.org/>.
- [157] Regenstrief Institute. Logical observation identifiers, names and codes (loinc). Standardisation Effort, June 2005. <http://www.regenstrief.org/loinc/>.
- [158] SNOMED International and College of American Pathologists (CAP). Systematized nomenclature of medicine (snomed). Standardisation Effort, July 2005. <http://www.snomed.org/>.
- [159] International Business Machines (IBM). *iSeries Information Center – POSIX Thread Application Programming Interfaces (API)*, November 2005. <http://publib.boulder.ibm.com/series/v5r2/ic2924/index.htm?info/apis/rzah4mst.htm>.
- [160] International Organization for Standardization (ISO). *ISO 8879:1986 – Standard Generalized Markup Language (SGML)*, 1st edition, 1986. <http://www.iso.ch/cate/d16387.html>.
- [161] Intershop Communications AG, Jena. *Enfinity 2 Pipeline Logic: Using the VPM*, 2003. <http://www.intershop.de/pdf/services/education/courses/EN2-220-EN.pdf>.
- [162] ISO/TC 215 Working Group 3 Health Informatics – Health Concept Representation. *Comments on Meta Terminology*. <http://www.tc215wg3.nhs.uk/pages/docs/cometate.rtf>.
- [163] International Telecommunication Union (ITU). Specification and description language (sdl) forum society, August 2004. <http://www.sdl-forum.org/SDL/index.htm>.
- [164] Ivar Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley-Longman, Boston, Muenchen, May 1997. <http://www.aw.com>.
- [165] Ludwig Jaeger. Linguistik: Ohne sprache undenkbar. In *Gehirn & Geist*, volume 2, pages 36–42. Spektrum der Wissenschaft, <http://www.spektrum.de>, 2003. <http://www.gehirn-und-geist.de>.
- [166] Gangolf Jobb. We need more freedom in science, more independence, more democracy. TreeFinder Website, 1997-2006. <http://www.treefinder.de/politics.html>.

- [167] Michael K. Johnson and Erik W. Troan. *Anwendungen entwickeln unter Linux*. Linux Specials. Addison-Wesley Longman, Bonn, Reading/ Mass., 1 edition, 1998. <http://www.aw.com>.
- [168] Bhaskar (formerly: Minoru Development) K. S. Open health mailing list, 1999-2005. openhealth@yahoogroups.com (formerly: openhealth-list@minoru-development.com).
- [169] Dipak Kalra. The use of international health it standards: Cen ehr communications task force – are we getting closer to an operational international ehr standard? Presentation, 2002. <http://www.epj-observatoriet.dk/konference2002/konferenceslides/DipakKalraB2.pdf>.
- [170] Dipak Kalra and et al. Headings for communicating information for the personal health record – headings within electronic healthcare records. Evaluation Report 1, Centre for Health Informatics and Multiprofessional Education (CHIME), London, May 1998. <http://www.nhsia.nhs.uk/headings/pdf/chime1.pdf>.
- [171] Kumanan Kanagasabapathi. Erstellung und anbindung intuitiver frontends an eine anwendung zur verwaltung administrativer personenaten unter beachtung von aspekten der internationalisierung. Student project (studienarbeit), Technical University of Ilmenau, Ilmenau, June 2003. <http://www.cybop.net>.
- [172] Kassenaerztliche Bundesvereinigung (KBV). x daten traeger (xdt), 2004. <http://www.kbv-it.de/it/xdtinfo.htm>.
- [173] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Boston, Muenchen, 1999.
- [174] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (ANSI-C)*. Prentice-Hall, Englewood Cliffs, 2nd edition, March 1988.
- [175] Marcel Kiesling. Das socketkommunikationsmodell. Seminar work (hauptseminararbeit), Technical University of Ilmenau, Ilmenau, July 2004. <http://www.cybop.net>.
- [176] Bill Kinnnersley. The language list. Website, September 2004. <http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>.
- [177] Jens Kleinschmidt. Ein beitrag zur evaluierung von komponententechnologien im umfeld von software zur medizinischen bildbearbeitung. Master's thesis (diplomarbeit), Technical University of Ilmenau, Ilmenau, July 2003. <http://www.cybop.net>.
- [178] Rainer Klute. *JDBC in der Praxis*. Addison-Wesley Longman, Bonn, Boston, Muenchen, 1998. <http://www.aw.com>.

- [179] Donald Ervin Knuth. Tex, 1978-2004. <http://www.tug.org/>.
- [180] Oswald Kowalski. Fachsprachen der prozessdatenverarbeitung. Script and Personal Discussions, 2004.
- [181] Philippe Kruchten, Ivar Jacobson, and et al. Rational unified process (rup). Website, 2004. <http://www-136.ibm.com/developerworks/rational/products/rup>.
- [182] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995. <http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf>.
- [183] Ralf Kuehnel. *Agentenbasierte Softwareentwicklung: Methode und Anwendungen*. Agenten Technologie. Addison-Wesley, Muenchen, 2001.
- [184] Markus Guenther Kuhn. A summary of the iso ebnf notation. Web Document, September 1998. <http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html>.
- [185] Torsten Kunze. Untersuchung zur realisierbarkeit einer technologieneutralen mapping-schicht fuer den datenaustausch am beispiel einer anwendung zum medizinischen formulardruck als integrativer bestandteil eines electronic health record (ehr). Master’s thesis (diplomarbeit), Technical University of Ilmenau, Ilmenau, January 2003. <http://www.cybop.net>.
- [186] AT&T Research Labs. Graphviz – graph visualization software and dot language, November 2004. <http://www.graphviz.org/>.
- [187] Gerald Lai, Scott Nisbitt, Ed Fong, and Phat Ha. Neural pathways to long term memory (ltm). Kin 356 Web Encyclopedia, January 2005. <http://ahsmail.uwaterloo.ca/kin356/ltm/ltm.htm>.
- [188] Leslie Lamport. Latex, 1984-2004. <http://www.latex-project.org/>.
- [189] Harold D. Lasswell. Describing the contents of communication. In Casey Ders., Smith, editor, *Propaganda, Communication and Public Opinion*. Princeton University Press, Princeton, 1946. <http://www.kfh-mainz.de/downloads/sasp/buesch/>.
- [190] Elpidio Latorilla and et al. Care2x, October 2004. <http://care2x.org/>.
- [191] Andrew Layman and Edward Jung. *XML-Data*. World Wide Web Consortium (W3C), note edition, January 1998. <http://www.w3.org/TR/1998/NOTE-XML-data-0105/>.
- [192] Gottfried Wilhelm Leibnitz. *La Monadologie*. E-Text.org, 2004. <http://www.e-text.org/text/Leibnitzogie.txt>.

- [193] Eric Levenez. Computer languages history – timeline. Website, September 2004. <http://www.levenez.com/lang/>.
- [194] Richard Levins and Richard Lewontin. *The Dialectical Biologist*. Harvard University Press, Harvard, reprint edition edition, March 1987. <http://en.wikipedia.org/wiki/Dialectics>.
- [195] C++ standard library (libstdc++), 2004. <http://gcc.gnu.org/libstdc++/>.
- [196] Clark S. Lindsey, Johnny S. Tolliver, and Thomas Lindblad. Java chips. Web Course, December 2004. <http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part3/Chapter24/chips.html>.
- [197] The linux documentation project. HOWTOs, Guides, FAQs, man pages, Linux Gazette, LinuxFocus, 2004. <http://www.tldp.org/>.
- [198] Barbara Liskov and et al. The clu programming language, 2004. <http://www.pmg.lcs.mit.edu/CLU.html>.
- [199] Eric Little. Initial taxonomical structure for upper-level military ontology categories of level 2 and 3 information fusion constructs. Presentation, 2003. [http://www.acsu.buffalo.edu/~eglittle/current research/MILO \(PWRPNT\).ppt](http://www.acsu.buffalo.edu/~eglittle/current%20research/MILO%20(PWRPNT).ppt).
- [200] Recordare LLC. Musicxml definition 1.0. Online Specification, March 2005. <http://www.musicxml.org>.
- [201] Andrea Lombardoni. VrmI interface for internet oMs. Master's thesis, Institut für Informationssysteme, Eidgenössische Technische Hochschule Zuerich (ETH), October 1999. <http://www2.inf.ethz.ch/personal/lombardo/archives/da/node5.html>.
- [202] Peter William Lount. Smalltalk.org, 2004. <http://www.smalltalk.org>.
- [203] RHA (Minisystems) Ltd. Dynamic data exchange (dde) and netdde faq. Frequently Asked Questions (FAQ), August 2005. <http://www.angelfire.com/biz/rhaminisys/ddeinfo.html>.
- [204] M. Lutz. *Programming Python*. O'Reilly and Associates, 1996.
- [205] Niels Malotaux. Evolutionary project management methods: How to deliver quality on time in software development and systems engineering projects. Booklet on Internet, February 2004. <http://www.malotaux.nl/nrm/pdf/MxEvo.pdf>.
- [206] Maintenance Agency Policy Group (MAPG). Global medical device nomenclature (gmdn). Standardisation Effort, July 2005. <http://www.gmdn.org/>.

- [207] Charlene Marietti. Will the real cpr / emr / ehr please stand up. *Healthcare Informatics Online*, May 1998. http://www.healthcare-informatics.com/issues/1998/05_98/cover.htm.
- [208] Tom Marley. Reusable information components in healthcare – a comparison of current methods and outputs. Discussion Paper 0.1, University of Salford, Shire, United Kingdom, December 2003. <http://www.centc251.org/TCMeet/doclist/TCdoc04/N04-001ReusableComponentDiscussion.pdf>.
- [209] David Megginson and et al. Simple api for xml (sax). Public Domain Software Project, 1997-2004. <http://www.saxproject.org/>.
- [210] Wolfgang Meier and et al. exist – open source native xml database, 2004. <http://exist.sourceforge.net/>.
- [211] Tim Menzies. Domain specific languages (dsl), 2004. <http://www.cs.pdx.edu/timm/dm/dsl.html>.
- [212] NJ MICRA, Inc. of Plainfield, editor. *Webster's Revised Unabridged Dictionary*. C. & G. Merriam Co., Springfield, Mass., february 1998 edition, 1913. <ftp://ftp.uga.edu/pub/misc/webster/>.
- [213] Leonid Mikhajlov and Emil Sekerinski. The fragile base class problem and its solution. Technical Report 117, Turku Centre for Computer Science (TUCS), May 1997. <http://www.tucs.abo.fi/publications/techreports/TR117.php>.
- [214] MySQL. Mysql database server, May 2005. <http://dev.mysql.com/>.
- [215] Logiciel Nautilus and Philippe Ameline. Odyssee project, 2003. <http://www.nautilus-info.com/>.
- [216] Open Source Development Network. Freshmeat. Free and Open Source Software Development Portal, 2004. <http://freshmeat.net/>.
- [217] Open Source Development Network. Sourceforge.net. Free and Open Source Software Development Portal, 2004. <http://sourceforge.net/>.
- [218] European network of excellence for End-User Development (EUD-net). Eud-net@umist. Website, April 2003. <http://www.co.umist.ac.uk/EUD-net/>.
- [219] National Health Service Information Authority (NHSIA). Office of population censuses and surveys classification of surgical operations and procedures. Standardisation Effort, April 2005. <http://www.connectingforhealth.nhs.uk/>.

- [220] United Kingdom (UK) National Health Service Information Authority (NHSIA). Read codes (read). Standardisation Effort, January 2002. <http://www.connectingforhealth.nhs.uk/>.
- [221] I. Nonaka. A dynamic theory of organizational knowledge creation. *Organization Science*, 5(1):14–35, February 1994.
- [222] Peter Norvig. The java iaq: Infrequently answered questions. <http://www.norvig.com/java-iaq.html>.
- [223] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.
- [224] John J O'Connor and Edmund F Robertson. The mactutor history of mathematics archive – indexes of biographies. Website of the School of Mathematics and Statistics, University of St Andrews, Scotland, January 2003. <http://www-history.mcs.st-andrews.ac.uk/history/index.html>.
- [225] Mick O'Donnell. What is systemic-functional linguistics? Internet Website, February 2004. <http://www.wagsoft.com/Systemics/Definition/definition.html>.
- [226] Journal of Free and Open Source Medical Computing (JOSMC). Free journal publishing web portal, September 2005. <http://www.josmc.org/>.
- [227] Association of Lisp Users. Object oriented and procedural lisp. <http://www.lisp.org/table/objects.htm>.
- [228] United States National Library of Medicine (NLM) and National Institutes of Health. Unified medical language system (umls). Standardisation Effort, June 2005. <http://www.nlm.nih.gov/research/umls/umlsdoc.html>.
- [229] International Council of Nurses (ICN). International classification for nursing practice (icnp). Standardisation Effort, June 2005. <http://www.icn.ch/icnp.htm>.
- [230] American College of Radiology (ACR) and National Electrical Manufacturers Association (NEMA). Digital imaging and communications in medicine (dicom). Standardisation Effort, April 2005. <http://medical.nema.org/>.
- [231] National Institute of Standards and Technology (NIST). Dictionary of algorithms and data structures. Online Dictionary, July 2004. <http://www.nist.gov/dads/>.
- [232] R. O'Hara and D. Gomberg. *Modern Programming Using REXX*. Prentice Hall, 1988.
- [233] Mike Olson and Uche Ogbuji. The python web services developer: Messaging technologies compared. IBM Developer Works Online Paper, July 2002. <http://www-106.ibm.com/developerworks/webservices/library/ws-pyth9/>.

- [234] Object Management Group (OMG). The common object request broker: Architecture and specification, 1992. <http://www.omg.org>.
- [235] Object Management Group (OMG). Unified modeling language (uml) specification, 2001. <http://www.uml.org>.
- [236] Object Management Group (OMG). Model driven architecture (mda), March 2002. <http://www.omg.org/mda/>.
- [237] Object Management Group (OMG). Healthcare domain task force (hdtf), formerly corbamed. Documents and Specifications, 2003. <http://healthcare.omg.org/>.
- [238] Standards Committee on Dental Informatics (SCDI) belonging to the American Dental Association (ADA). Standards and guidelines for dental practice. Standardisation Effort, July 2005. <http://www.ada.org/>.
- [239] OpenOffice.org Organization. Openoffice.org, 2004. <http://www.openoffice.org/>.
- [240] Charles E. Osgood, George E. Suici, and Percy H. Tannenbaum. *The Measurement of Meaning*. University of Illinois Press, Urbana, 1957. <http://www.kfh-mainz.de/downloads/sasp/buesch/>.
- [241] Open Source Health Care Alliance (OSHCA). Oshca homepage, 2000-2004. <http://www.oshca.net>.
- [242] Open Source Initiative (OSI). Open source definition, certification mark and program, approved licenses. Internet Site, 2004. <http://www.opensource.org/>.
- [243] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [244] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer magazine*, March 1998.
- [245] David Parks. Agent oriented programming: A practical evaluation. Web Article, May 1997. <http://www.cs.berkeley.edu/~davidp/cs263/>.
- [246] Ilian Pashov. *Feature-Based Methodology for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems*. PhD thesis, Technical University of Ilmenau, Ilmenau, 2004. http://www.bibliothek.tu-ilmenau.de/elektr_medien/dissertationen/2004/Pashov.Ilian/index.html.
- [247] Mark C. Paulk. Extreme programming from a cmm perspective. *IEEE Software*, 18(6):19–26, November/ December 2001. <http://www.sei.cmu.edu/publications/articles/paulk/xp-cmm.html>.

- [248] Mark C. Paulk, Charles V. Weber, and et al., editors. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, Carnegie Mellon University, Software Engineering Institute, 1995. ISBN 0-201-54664-7.
- [249] Margarete Payer and Alois Payer. Computervermittelte kommunikation / computer mediated communication (cmc). Lecture Notes on Website, November 2002. <http://www.payer.de/cmclink.htm>.
- [250] Ilka Philippow. Grundlagen der informatik. <http://www.theinf.tu-ilmenau.de/proinf/english/teaching/Lectures.html>, April 2003.
- [251] Joseph P. Pickett and et al., editors. *American Heritage Dictionary of the English Language*. Houghton Mifflin Company, Boston, 4th edition, 2000. <http://www.bartleby.com/61/>.
- [252] Wolfgang Pree. Meta patterns – a means for capturing the essentials of reusable object-oriented design. In *Proceedings of ECOOP '94*, pages 150–162, 1994.
- [253] Apache Project. Apache jakarta open source solutions on the java platform, 1999-2004. <http://jakarta.apache.org/>.
- [254] AspectJ Project. Aspectj: Aspect-oriented java extension, 2002. <http://aspectj.org>.
- [255] AspectWerkz Project. Aspectwerkz: Simple, dynamic, lightweight and powerful aop for java, 2002. <http://aspectwerkz.codehaus.org/>.
- [256] CYBOP Project. Cybernetics oriented programming (cybop), 2002-2004. <http://www.cybop.net>.
- [257] CYGWIN Project. Cygwin linux-like environment for windows. Internet Portal, 2006. <http://www.cygwin.com/>.
- [258] Debian Project. Debian gnu/linux, 1997-2004. <http://www.debian.org>.
- [259] Debian-Med Project. Debian-med, 2002-2004. <http://www.debian.org/devel/debian-med/>.
- [260] JDistro Project. Jdistro java distribution, 2002-2004. <http://www.jdistro.com/>.
- [261] JOS Project. Java operating system, 2000-2004. <http://cjos.sourceforge.net/>.
- [262] Mozilla Project. Xml user interface language (xul). Specification, 2004. <http://www.mozilla.org/projects/xul/>.
- [263] OSCAR/ McMaster Project. Open source clinical application and resource from mc-master university (oscar), 2002-2004. <http://www.oscarmcmaster.org/>.

- [264] PostgreSQL Project. Postgresql, 2000-2004. <http://www.postgresql.org>.
- [265] PostgreSQL Project. Postgresql – multiversion concurrency control, 2002. <http://www.postgresql.org/idsoc/index.php?mvcc.html>.
- [266] Res Medicinae Project. Res medicinae – medical information system, 1999-2004. <http://www.resmedicinae.org>.
- [267] Scope Project. Scope hmv java framework, 2001-2004. <http://scope.sourceforge.net/>.
- [268] The Apache XML Project. Xerces java parser, 2003. <http://xml.apache.org/xerces2-j/index.html>.
- [269] The HXP Project. Healthcare xchange protocol (hxp), 2004. <http://hxp.sourceforge.net/>.
- [270] Willibald Pschyrembel and Woerterbuch-Redaktion des Verlages. *Pschyrembel – Klinisches Woerterbuch*. Walter de Gruyter, Berlin, 260 edition, 2004. <http://www.pschyrembel.de/>.
- [271] Eric Steven Raymond. *The Cathedral and the Bazaar*. Internet Publication, 1.57 edition, 2000. <http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/>.
- [272] Refsnes Data. *W3Schools - Full Web Building Tutorials - All Free*, 1999-2004. <http://www.w3schools.com/>.
- [273] Fiach Reid. Xaml.net – a guide to xaml. Web Article, October 2004. <http://www.xaml.net/>.
- [274] Peter Ripota. Das universum hat ein bewusstsein! In *P.M. Magazin*, pages 21–25. Hans-Hermann Sprado, September 2003. <http://www.pm-magazin.de>.
- [275] Jeremy Rogers. *Health Informatics – Vocabulary for Terminological Systems*, May 2002. <http://www.tc215wg3.nhs.uk/pages/>.
- [276] Jeremy Rogers and University of Manchester Medical Informatics Group. Medical informatics standards / medical terminology. Website, January 2004. <http://www.cs.man.ac.uk/mig/links/RCSEd/standards.htm> — [terminology.htm](http://www.cs.man.ac.uk/mig/links/RCSEd/standards.htm).
- [277] Kelley L. Ross, editor. *The Proceedings of the Friesian School*, Fourth Series, Department of Philosophy, Los Angeles Valley College, Van Nuys, 2004. Ross, Kelley L. <http://www.friesian.com/>.
- [278] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. *Filesystem Hierarchy Standard (FHS)*. Filesystem Hierarchy Standard Group, 2.3 edition, January 2004. <http://www.pathname.com/fhs/>.

- [279] Don Sannella. Extended meta language (extended ml), June 2004. <http://homepages.inf.ed.ac.uk/dts/eml/>.
- [280] R. Schiedermeier. Programmieren i. <http://www.informatik.fh-muenchen.de/schieder/programmieren-1-ws96-97/struktpgm.html>, 1996.
- [281] Christoph Schoenhofer. Unternehmensberatung: Die neuro-manager. In *Gehirn & Geist*, volume 2, pages 74–75. Spektrum der Wissenschaft, <http://www.spektrum.de>, 2003. <http://www.gehirn-und-geist.de>.
- [282] W3 Schools. Structured query language (sql) tutorial, November 2004. <http://www.w3schools.com/sql/default.asp>.
- [283] Stephan H. Schug. Europaeische und internationale perspektiven von telematik im gesundheitswesen. Studie (Expert’s Report) Fassung 1.0, Gesellschaft fuer Versicherungswissenschaft und -gestaltung (GVG) e.V., Ausschuss Telematik im Gesundheitswesen (ATG), Koeln, December 2000. <http://www.iqmed.de>.
- [284] Viktor Schuppan and Winfried Ruszwurm. A cmm-based evaluation of the v-model 97. In R. Conradi, editor, *Proceedings of the 7th European Workshop on Software Process Technology*, pages 69–83, Kaprun, Austria, February 2000. Springer. <http://www2.inf.ethz.ch/personal/schuppan/VSchuppanWRuszwurm-EWSPT-2000.pdf>.
- [285] Peter v. Sengbusch. Kybernetik: Systeme, steuerung, regelung, information und redundanz, 2002. <http://www.biologie.uni-hamburg.de/b-online/d15/15.htm>.
- [286] Claude E. Shannon and Warren Weaver. Mathematical communication model. <http://www.kfh-mainz.de/downloads/sasp/buesch/>.
- [287] Yoav Shoham. Agent oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993. <http://portal.acm.org/citation.cfm?id=152188>.
- [288] Julian Smart, Anthemion Software Ltd., and et al. wxwidgets (formerly: wxwindows) cross-platform native ui framework, April 2005. <http://www.wxwidgets.org/>.
- [289] Barry Smith and Christopher Welty, editors. *Formal Ontology and Information Systems*. ACM Press, New York, 2001. <http://www.geog.buffalo.edu/ncgia/ontology/>.
- [290] IEEE Computer Society. Standard Specifications. <http://www.ieee.org>.
- [291] IEEE Computer Society. Recommended practice for architecture description of software intensive systems. Standard Specification IEEE Std 1471-2000, 2000.
- [292] The Internet Society. Electronic commerce modeling language (ecml). Field Specifications for E-Commerce, April 2001. <http://www.faqs.org/rfcs/rfc3106.html>.

- [293] Poseidon Software and Invention. Base valued numbers. Internet Page, November 1998. <http://www.psinvention.com/zoetic/basenumb.htm>.
- [294] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole, Thomson Learning, Pacific Grove, 2000.
- [295] C. M. Sperberg-McQueen, Henry Thompson, and et al. *XML Schema*. World Wide Web Consortium (W3C), recommendation part 0, 1, 2 edition, October 2004. <http://www.w3.org/XML/Schema>.
- [296] Sistema) Spirit Project Partners (minoru, ist. Eu spirit portal. Website, 2002. <http://www.euspirit.org/>.
- [297] Bernhard Steppan. *Java-Programmierung mit Borland JBuilder 4*. Addison-Wesley, Muenchen, Boston, 2001. <http://www.aw.com>.
- [298] P. Stoerig. Hirnforschung – visuelle wahrnehmung: Blindsehen. In *Gehirn & Geist*, volume 2, pages 76–80. Spektrum der Wissenschaft, <http://www.spektrum.de>, 2003. <http://www.gehirn-und-geist.de>.
- [299] Detlef Streitferdt. A component model for applications based on feature models. In *Workshop on Software Variability Management*, Groningen, December 2004. RUG.
- [300] Detlef Streitferdt. *Family-Oriented Requirements Engineering*. PhD thesis, Technical University of Ilmenau, Ilmenau, 2004. http://www.bibliothek.tu-ilmenau.de/elektr_medien/dissertationen/2004/Streitferdt_Detlef/index.html.
- [301] Abteilung Allgemeinmedizin Studienzentrum Goettingen. Medizinische versorgung in der praxis (medvip) projektantrag, teil e. <http://medvip.uni-goettingen.de/>, May 2005.
- [302] Z Sweedyk. Software development life-cycle models. Online Lecture Notes, 2003. <http://www.cs.hmc.edu/courses/mostRecent/cs121/lectures/02.lifeCycles.pdf>.
- [303] Andrew Stuart Tanenbaum. *Computernetzwerke*. Pearson Studium, Muenchen, 3rd edition, 2000. <http://www.pearson-studium.com>.
- [304] Andrew Stuart Tanenbaum. *Modern Operating Systems*. Prentice-Hall, New Jersey, London, Sydney, 2nd edition, 2001. <http://www.prentice-hall.com>.
- [305] Andrew Stuart Tanenbaum and James R. Goodman. *Structured Computer Organization / Computerarchitektur - Strukturen, Konzepte, Grundlagen*. Prentice-Hall, Muenchen, London, 4th edition, 1999. <http://www.prentice-hall.com>.
- [306] Andrew Stuart Tanenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms*. Prentice-Hall, New Jersey, 2002. <http://www.prentice-hall.com>.

- [307] GTK Team. Gimp toolkit (gtk), April 2005. <http://www.gtk.org/>.
- [308] SelfLinux Team. *SelfLinux – Linux-Hypertext-Tutorial*. PingoS e.V., Hamburg, 0.11.3 edition, June 2005. <http://www.selflinux.org/>.
- [309] The GCC Team. The gnu compiler collection, May 2004. <http://gcc.gnu.org/>.
- [310] TechTarget. Inter-process communication (ipc). Web Glossary, November 2004. http://searchsmb.techtarget.com/sDefinition/0,,sid44_gci214032,00.html.
- [311] Karsten Tellhelm. Xml parser. Code fragments, Technical University of Ilmenau, Ilmenau, February 2004. <http://www.cybop.net>.
- [312] Klasse Objecten Soest the Netherlands. Object constraint language (ocl) center. Web Tutorial, October 2004. <http://www.klasse.nl/ocl/index.html>.
- [313] Linus Torvalds, Alan Cox, and et al. The linux operating system kernel, 2004. <http://www.kernel.org/>.
- [314] Thomas Trepl. Edifactory. Website, 1999. <http://www.edifactory.de/edifact/edimain1.html>.
- [315] Trolltech. Cute toolkit (qt) c++ application development framework, April 2005. <http://www.trolltech.com/products/qt/index.html>.
- [316] Indian TeX Users Group (TUG). Online tutorials on latex, May 2003. <http://www.tug.org.in/tutorial/>.
- [317] Valentin Turchin. The cybernetic ontology of action. *Kybernetes*, 22(2):10–30, 1993. ftp://ftp.vub.ac.be/pub/projects/Principia_Cybernetica/Papers_Turchin/Cybernetic_Ontology.tex.
- [318] United Nations (UN). Electronic data interchange for administration, commerce and transport (edifact), 2004. <http://www.unece.org/trade/untdd/>.
- [319] Urban und Fischer Verlag Muenchen. Anatomical images from sobotta: Atlas der anatomie. Internet; CDROM, 2002. <http://www.urbanfischer.de>.
- [320] Princeton University. Wordnet 2.0 dictionary. Internet Web Database, September 2003. <ftp://ftp.cogsci.princeton.edu/pub/wordnet/2.0/WordNet-2.0.tar.gz>.
- [321] Ignacio Valdes. Linux med news. News Website, April 2005. <http://www.linuxmednews.org/>.
- [322] J.H. van Bommel and M.A. Musen, editors. *Handbook of Medical Informatics*. Erasmus University, Stanford University, Rotterdam, Stanford, website 3.3 edition, March 1999. http://www.mieur.nl/mihandbook/r_3.3/handbook/home.htm.

- [323] Arie van Deursen, Paul Klint, and Joost Visser. Domain specific languages (dsl): An annotated bibliography. Online Paper, February 2000. <http://homepages.cwi.nl/~arie/papers/dslbib/>.
- [324] Dimitri van Heesch. Doxygen documentation system for c++, c, java, objective-c, idl (corba and microsoft flavors) and to some extent php, c# and d, October 2004. <http://www.doxygen.org/>.
- [325] Vienna development method - specification language (vdm-sl), August 2004. ftp://gateway.dec.com/pub/vdmsl_standard.
- [326] BioMed Central (BMC) | The Open Access Publisher. Journal publishing web portal, September 2005. <http://www.biomedcentral.com/>.
- [327] Steve Vinoski. New features for corba 3.0. *Communications of the ACM*, October 1998. <http://www.ionas.com/hyplan/vinoski/cacm.pdf>.
- [328] World Vista. Open veterans health information systems and technology architecture (open vista), April 2005. <http://www.worldvista.org/openvista/index.html>.
- [329] World Wide Web Consortium (W3C). Document object model (dom), 2002. <http://www.w3.org/DOM/>.
- [330] World Wide Web Consortium (W3C). World wide web consortium homepage, 2002-2004. <http://www.w3c.org/>.
- [331] World Wide Web Consortium (W3C). Simple object access protocol (soap). Recommendation, 2004. <http://www.w3.org/2000/xp/Group/>.
- [332] World Wide Web Consortium (W3C). World wide web consortium issues rdf and owl recommendations: Semantic web emerges as commercial-grade infrastructure for sharing data on the web. Press Release, February 2004. <http://www.w3.org/2004/01/sws-pressrelease>.
- [333] C. Peter Waegemann. Ehr vs. cpr vs. emr. *Healthcare Informatics Online*, May 2003. http://www.healthcare-informatics.com/issues/2003/05_03/cover_ehr.htm.
- [334] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly and Associates, 2 edition, 1996.
- [335] Norman Walsh. A technical introduction to xml. Internet Tutorial, October 1998. <http://www.xml.com/pub/a/98/10/guide0.html>.
- [336] Norman Walsh and Leonard Mueller. *DocBook: The Definitive Guide*. O'REILLY, <http://www.oreilly.com/>, v4.3cr3 edition, January 2004. <http://docbook.org/>.

- [337] Elmar Warken. *Delphi 2 – Software-Entwicklung fuer 32-Bit-Windows*. Addison-Wesley, Bonn; Reading, Massachusetts, 1996. <http://www.aw.com>.
- [338] S. Wartik and R. Prieto-Diaz. Criteria for comparing domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):403–431, September 1992.
- [339] Lawrence L. Weed. *Medical Records, Medical Education, and Patient Care: The Problem-Oriented Record as a Basic Tool*. Year Book Medical Publishers, Inc, Chicago, 1971.
- [340] Don Wells. Extreme programming (xp). Website, January 2003. <http://www.extremeprogramming.org>.
- [341] Henk Westerhof and Dutch College of General Practitioners Utrecht. Episodes of care in the new dutch gp systems. *Primary Health Care Specialist Group Annual Conference Proceedings, Cambridge*, September 1998. <http://www.phcsg.org.uk/conferences/cambridge1998/westerhof.htm>.
- [342] World Health Organization (WHO). International classification of diseases (icd). Standardisation Effort, June 2003. <http://www.who.int/classifications/icd/en/>.
- [343] Ross N. Williams. Funnelweb tutorial manual, January 2000. <http://www.ross.net/funnelweb/tutorial/>.
- [344] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Inc., Champaign, 2002. <http://www.wolframscience.com/thebook.html>.
- [345] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*, 1.0 edition, February 1998. <http://www.w3.org/TR/1998/REC-xml-19980210>.
- [346] World Wide Web Consortium (W3C). *OWL Web Ontology Language Guide*, w3c recommendation edition, February 2004. <http://www.w3.org/TR/owl-guide/>.
- [347] World Wide Web Consortium (W3C). *RDF Primer*, <http://www.w3.org/tr/2004/rec-rdf-primer-20040210/> edition, February 2004. <http://www.w3.org/TR/rdf-primer/>.
- [348] World Wide Web Consortium (W3C). *Resource Description Framework (RDF)*, w3c recommendations edition, October 2004. <http://www.w3.org/RDF/>.
- [349] World Wide Web Consortium (W3C). *Web Ontology Language (OWL)*, w3c recommendation edition, February 2004. <http://www.w3.org/TR/owl-ref/>.
- [350] Juergen Zimmermann and Gerd Beneken. *Verteilte Komponenten und Datenbankankbindung. Mehrstufige Architekturen mit SQLJ und Enterprise JavaBeans™ 2.0*. Addison-Wesley, Boston, Muenchen, 2000. <http://www.aw.com>.

- [351] Heinz Zuellighoven and et al. Tools & materials approach to software-development. JWAM Open Source Project, 2004. http://www.jwam.de/engl/produkt/e_tmapproach.htm.

14.3 Figures

1.1	Scientific Inventions	2
1.2	Constructive Development	6
1.3	Merger of traditional and new Concepts	7
1.4	Document Structure	8
2.1	Waterfall Process with Back Flow	14
2.2	V-Model	15
2.3	Iterative Process	16
2.4	Agile Manifesto	17
2.5	Extreme Programming (strongly simplified)	18
2.6	Abstraction Gaps	21
2.7	Four Views Model [138]	22
2.8	The 4+1 View Model of Architecture [182]	23
3.1	Database Server (2 Tiers)	28
3.2	Presentation Client (3 Tiers)	30
3.3	Web Client and Server	31
3.4	Local Process	32
3.5	Human User	34
3.6	Peer-to-Peer Node Communication	35
3.7	Remote Server	36
3.8	Legacy Host	37
3.9	ISO OSI Reference Model	38
3.10	Vertical and Horizontal Scaling	39
3.11	Universal Communication between Humans and Computers	41
4.1	System with Logical Layers	44
4.2	Programming Language History	46
4.3	Programming Paradigm Systematics	47
4.4	Computer Structure (adapted from [305])	49
4.5	Statement as Program Flow Chart and Structure Chart	53
4.6	Condition as Program Flow Chart and Structure Chart	54
4.7	Loop as Program Flow Chart and Structure Chart	55
4.8	Classification as UML Diagram	69

4.9 Encapsulation as UML Diagram	71
4.10 Inheritance as UML Diagram	72
4.11 Polymorphism as UML Diagram	75
4.12 Java Container Framework Systematics	76
4.13 Falsified Contents with Container Inheritance	78
4.14 Software Pattern Classification	81
4.15 Layers Pattern	82
4.16 Layer Supertype Pattern	83
4.17 Domain Model Pattern	84
4.18 Data Mapper Pattern	85
4.19 Data Transfer Object Pattern	86
4.20 Model View Controller Pattern	88
4.21 Hierarchical Model View Controller Pattern	89
4.22 Microkernel Pattern	90
4.23 Broker Pattern	90
4.24 Pipes and Filters Pattern	91
4.25 Reflection Pattern	92
4.26 Java Type System	94
4.27 Command Pattern	97
4.28 Wrapper Pattern	98
4.29 Whole-Part Pattern	99
4.30 Composite Pattern	100
4.31 Chain of Responsibility Pattern	101
4.32 Observer Pattern	102
4.33 MVC- using Observer Pattern	103
4.34 Template Method Pattern	104
4.35 Counted Pointer Pattern	105
4.36 Singleton Pattern	106
4.37 Component Lifecycle Methods	111
4.38 Class Inheriting Loggable Concern Interface	114
4.39 Redundant Code through Usage of Concerns	115
4.40 Overlapping Code through Usage of Concerns	116
4.41 Concerns Spread Functionality, an Ontology Bunches it	117
4.42 Six Pack Model of System Family Development [44, 79]	123
4.43 Classical Feature Model Diagram of a Car (based on [246])	130

4.44 Model Driven Architecture [236]	132
4.45 Model-Code Synchronisation [35, diagram by John Daniels]	134
4.46 Dual Model Approach [18]	156
5.1 Mindmap of Sciences whose Principles influenced CYBOP	163
5.2 Separation of Mind/ Brain Leading to Knowledge/ System Control	166
5.3 Concepts of Human Thinking Leading to Hierarchical Knowledge	167
5.4 Translation of Data by Rules Leading to State-/ Logic Knowledge	168
5.5 Overall CYBOP Approach Based on Statics and Dynamics	169
6.1 Divisions of the Nervous System [52]	176
6.2 Types of Memory [187, 137]	179
6.3 Information Processing Model [137]	180
6.4 Classification of Concerns	186
6.5 Domain-Application- versus System-Knowledge Separation	188
6.6 Data Garden	191
6.7 Knowledge – Hardware Connection	193
6.8 System with Memory Structures, Processing Loops and Lifecycle	195
7.1 Wolfram's Four Basic Kinds of Behaviour [344]	200
7.2 The Universe as to-be-abstracted Conglomerate (swirl from [258])	202
7.3 Abstractions of Human Thinking	203
7.4 Systematics of Nature	204
7.5 Single Model Approach (adapted from [18])	218
7.6 Semi Structured Model Approach (adapted from [18])	219
7.7 Hierarchical Model Approach (adapted from [18])	220
7.8 Adapted (H)MVC Pattern with Hierarchical Elements	222
7.9 Association Elimination in an EHR	223
7.10 Model Container and Ontological Levels	224
7.11 <i>Heal Illness</i> as Hierarchical Algorithm, taken from Medicine	225
7.12 HL7 Reference Information Model Framework [150]	226
7.13 RIM Entities [150]	226
7.14 Accessing a Person's Attributes	227
7.15 Access Method Elimination through Top-Level Container	228
7.16 Categorisation versus Composition of Parties [18, p. 12]	229
7.17 Knowledge Schema with Meta Information about Parts	234

7.18 Double Hierarchy of Parts and Meta Information	236
7.19 Concept of a Horse with Structure, Meta Properties and Logic	237
7.20 Universal Memory Structure	240
8.1 Closed Loop System with Feedback, modelled as Black Box	249
8.2 Logic translates between Input-, Domain- and Output States	250
8.3 Human Body with Sensoric and Motoric Organs	251
8.4 Computer Hardware with Input- and Output Devices	253
8.5 Ontology comparing Technical- and Biological Environment	253
8.6 Signal Processing as UML Sequence Diagram	254
8.7 Mathematical Communication Model by Shannon & Weaver [286]	256
8.8 Conversation Model by Osgood & Schramm [240]	257
8.9 Contents of Communication (Lasswell Formula) [189]	258
8.10 IT Environment with Server using Communication Patterns	259
8.11 Communication Patterns placed in Layered Architecture	262
8.12 Simplified Layered Architecture with State-/ Logic Knowledge	263
8.13 Different Kinds of Model Translators	264
8.14 State Primitives sorted after their Granularity	267
8.15 Runtime Model Hierarchy with Logic manipulating States	268
9.1 Recommended CYBOL DTD	277
9.2 Simplified CYBOL DTD	277
9.3 Simplified CYBOL XSD	278
9.4 Recommended CYBOL XSD	279
9.5 CYBOL in EBNF	280
9.6 Loop Control Structure and Elements in C and CYBOL	291
9.7 Condition Control Structure and Elements in C and CYBOL	292
9.8 Musical Score of Franz Schubert's <i>Ave Maria</i> [200]	294
9.9 CYBOL Editor Supporting Double Hierarchies by Triple Choice	306
9.10 CYBOL Template Diagram (TD) Proposal	308
9.11 CYBOL Model Diagram (MD) Proposal	309
9.12 CYBOL Organisation Diagram (OD) Proposal	310
9.13 CYBOL Communication Diagram (CD) Proposal	311
10.1 CYBOI Architecture consisting of Four Parts	315
10.2 CYBOI Part Dependencies and Control Flow	322

10.3 Input-to-Output Model Transition	325
11.1 FLOSS Development Portals	332
11.2 Portal Services	333
11.3 Applications Grouped around an Electronic Health Record Core	340
11.4 Medical Informatics Working Groups of DIN/ CEN/ ISO [283]	341
11.5 Early Record Module	359
11.6 Nested Views of Module Frames	360
11.7 Topological Documentation in Record Module [31]	361
11.8 ResAdmin Knowledge Models (Extract)	363
11.9 Simple Web User Interface of the ResAdmin Module [141]	364
12.1 Knowledge Triumvirate with Schema, Template and Model	375
12.2 Common Knowledge Abstraction useable by many SEP Phases	376
13.1 Comparison of a traditional SEP with CYBOM	386

14.4 Tables

3.1	Systematics of Abstract System Concepts	26
3.2	Vertical and Horizontal Application Types [11]	40
4.1	UML 2.x Diagram Types [6]	128
4.2	Taxonomic Classification of the Animal Kingdom	151
4.3	Structural Elements of an ADL-defined Archetype [21]	155
6.1	Brain Structures in Analogy to a Computer [52]	177
7.1	Materiality of Language, according to Five Human Senses [39]	212
7.2	Pattern Systematics	215
7.3	Hierarchical Structuring of Biological Systems	231
7.4	Logical Book	231
7.5	System of Sciences	232
7.6	Car Model	232
7.7	Astronomical Particles	233
7.8	Physical Particles	233
8.1	Effects as Basis of Sensing	252
9.1	Mapping Classical Containers to CYBOL	301
10.1	Analogies between the Java- and CYBOP World	314
11.1	Student Works [256]	358

14.5 History

For those who are interested in how I came to develop the concepts that were introduced in this document, I have created this small history of events and enlightening ideas.

- 1986 Although having difficulties, my father manages to cross the border to West-Germany by visiting some relatives. He also manages to come back into the East :-) and a present I get is a brand-new *Commodore 64* home computer. I mostly play games on it but also do my first steps in programming with *BASIC*.
- 1988 In the late days of East Germany, new computer subjects are introduced at schools. So I take part in them and continue learning *BASIC* and some *dBASE*.
- 1990 At the *Technical University of Ilmenau*, one of the first subjects in my study is *Algorithms and Programming* where we learn structural programming in *Turbo Pascal*.
- 1993 During my year abroad at *Sussex University* in Brighton, England, I come in touch with the *Internet* (email) and *UNIX machines*.
- 1994 Back in Germany, I start *administrating* the information infrastructure in my parents' medical practice, learn about *Computer Hardware* and wonder about how doctors are cheated with overpriced products and services.
- 1995 In the German *c't* computer magazine, I develop an announcement of a new operating system called *Linux*. I order and install my first distribution, *SuSE November 1995*.
- 1995 We have a quite theoretical lecture on *Object Oriented Programming* in *Smalltalk*. However, that is the first time I hear about those (then still new) programming concepts.
- 1996 During my student's research project and diploma work, I implement parts of a *Neural Network* in *Object Pascal* (Delphi).
- 1998 First steps in *C++* at my first employer *HM Informatics*.
- 1999 At *OWiS Software*, I learn to change my thinking away from only the source code – towards the actual *Concepts* and *Architecture* behind a software. I also learn how to use the *Unified Modeling Language* (UML).
- 2000 Recognising the opening of the *SourceForge* developer portal in 1999, I set up the *Res Medicinae* project in April 2000. It aims at creating a *Medical Software* for physicians. Preparation, website, investigation on similar projects and so on take months.

- 2000 My *Java* knowledge can be manifested at *Intershop Communications* where we are building solutions for e-commerce (Web Client-Server Applications).
- 2001 I visit my first *Free and Open Source Software Developers' Meeting* (FOSDEM) in Brussels, Belgium, where Richard M. Stallman of the *Free Software Foundation* (GNU project) holds a talk.
- 2001 Returned to my former university, I start active coding on *Res Medicinae* in April 2001. It takes some time to handle the *Concurrent Versions System* (CVS).
- 2001 Following the standard approach and several Java Tutorials, my first application is nothing more than a *main* method which creates a Java Swing *JFrame*.
- 2001 Stepwise, I start moving out code into special classes, like for example all *GUI* code into a class *Frame*.
- 2001 Soon I loose overview and realise the need for some clearer structure. I remember the *Design Patterns* applied at my former employers and start using the *Model View Controller* (MVC) and further patterns.
- 2001 Since flexibility is one of the most important aims of my efforts, I realise the shortcomings of MVC and the *Observer* pattern (bidirectional dependencies), especially when it comes to web applications. A good solution I find is the *Hierarchical Model View Controller* (HMVC) design pattern, applied by the *Scope* free software project.
- 2002 My disposedness to style guides, clean and well-documented code lets me structure and order every method and attribute and check them all for *NULL* pointer errors and other exceptions. I find out that every attribute not only needs a *set* and *get* method, but also a *create* and *destroy* method. Class names as type information are handed over to the create method in form of a string.
- 2002 Following the idea of *create* and *destroy* methods, I come across the *Component Lifecycle*, described by the *Apache-Jakarta* project. I change all code by applying *Lifecycle Methods*.
- 2002 Having read the *OpenEHR Design Document*, I know that classes should be grouped in layers with clear dependencies, called an *Ontology*. Higher-layer objects consist of objects from lower-level layers, but not the other way. I restructure my code by moving all classes into new packages (directories), each representing an *Ontological Level*.
- 2002 Apache's lifecycle *Concerns* turn out to be useless. They only break the ontology rack and violate its dependency rules by connecting otherwise strictly separated system

parts. Moreover, concerns encourage the inheritance of redundant or overlapping properties. The same counts for variations like *Aspects* and *Interfaces* in general. I replace them all by pure *Classes*, following the ontology structure.

- 2002 I begin to realise that not only *View* (MVC) and *Controller* (HMVC) are *hierarchical*, but also the *Model* (Knowledge/ Domain) is. In many days and weeks of intensive thinking I find that – as in Universe – actually *every* other software component is hierarchical, too. Consequently, I introduce one top-most super (meta) class *Item* that represents a simple *Tree* (*Map* container). Inheriting classes do not need to implement *create*, *destroy*, *set* or *get* methods any longer since they inherit them from *Item*. This saves me hundreds of lines of code, at once, and improves clearness a lot.

And this is my personal **Break-Through**. From now on, I first think about nature and its concepts and then implement software source code. Suddenly, everything seems easier, I have an *Example*, a *Way-to-go*: *Nature*. There aren't less (rather more) tasks to solve now, but at least the direction is clear.

- 2002 By applying the new concepts, one difference gets very clear: *System Control-* and *Domain Model* code are both hierarchical, but different ontologies need to be defined for them. An *active* system works on a *passive* model, that is it depends on it. While a system provides the means for *input* and *output* and *controls* the *Action* (*Workflow*), a model just represents domain data. Searching for parallels in nature, I find that *Human Body* and *Human Brain* correspond to *System* and *Model*. What humans (possibly unconsciously) want to do is to imitate themselves, with robots as with computers as with other machines, tools or abstractions.

- 2002 The overall structure of the *Framework* seems clear now. There will be three major parts, *Basic*, *Model* and *System*. *Model* and *System* both depend on the fundamental abstractions of the *Basic* package which encapsulates programming language types. In addition, *System* depends on *Model*.

- 2003 Encountering difficulties in synchronising *Frontend* and *Backend*, I come to the conclusion that they are actually the same, passive data models that have to be translated into each other. It is not necessary – even badly wrong – to apply different design patterns for their implementation. I change inheritances and dependencies of many of my framework classes so that finally *Knowledge-/ Domain-*, *Backend-*, *Communication-* and *Frontend* models are of the same super type. This decision opens unforeseen new possibilities. All kinds of frontends, all communication and backend mechanisms can now be implemented *modular* and *flexible*.

- 2003 Java's event handling using *ActionListener* interfaces ignores the dependencies between ontological layers and is thus improper (just like the concern interfaces mentioned above). I implement a new *Signal Handling* mechanism which is oriented on parts of a sentence in human language like *Subject*, *Predicate*, *Object*.
- 2003 I realise how far I have moved away from my original aim of writing a small medical application. While implementing the new signal handling mechanism it gets clear that I move more and more towards the operating system and its hardware input/ output handling code which scares me a bit. But there seems to be no other way to go when aiming at the creation of clear, easy, modular, flexible, correct systems.
- 2003 *Exceptions* are system-internal *Signals*. It is not necessary to use an extra signalling mechanism for exception handling. And it is dangerous for security if an instance knows about its parent to bubble up an exception. For now, I remove all exceptions besides the standard one, which again saves me many lines of unnecessary code.
- 2003 An abstract model has multiple properties. Not only it keeps references to its parts (attributes) and procedures (methods), but also it knows about the *Model* and *Position* (and possibly more *Meta* properties) of each part.
- 2003 After difficulties with the item meta model, I realise that I actually want to replace the standard *Class Concept* offered by Java and other Object Oriented languages. It takes me some weeks of thinking to find the reason and way out: Traditional programming mixes *Knowledge* with the *Handling* of its instances. Both need to be separated. I move most classes to XML-based *Model* files and call their specification *Cybernetics Oriented Language* (CYBOL).

Here begins a *New Era*. From now on, business domain *Knowledge* and hardware-close *System Control* code are stored separately and treated differently. The concepts to store *static* knowledge are very different from those that are used to *dynamically* control a system's hardware.

- 2003 Later comparisons with biology support my theory. The genetic information is static knowledge that gets forwarded from one cell to another, during *Cell Separation*. The dynamic processing of that knowledge, that is the creation and functioning of organelles is a completely different issue. In a similar manner, application knowledge has to become transportable between different platforms and absolutely independent from hardware.

- 2003 Since the time when I first experimented with lifecycle methods à la *Apache Jakarta*, I had used a *configure* method in my code which could read external configuration files and configure internal items accordingly. Nothing different was needed to read CYBOL files. They represented the whole application configuration knowledge and only had to be interpreted correctly. The remaining Java code therefore not only had to contain system control functionality, but also had to be able to read and write knowledge in form of CYBOL files. For that reason, it was called *Cybernetics Oriented Interpreter* (CYBOI).
- 2003 In search for new concepts and ideas, I also read about a number of philosophical theories from Aristotle, Leibnitz and others and can identify a mistake in my thinking: the important separation is not between *Body* and *Brain* as presumed before, it is between *Body/Brain* and *Mind*! *Neural Networks* try to imitate the functioning of the physical brain, but what I wanted to do is to imitate concepts of the logical mind, of human thinking.
- 2003 Reading a magazine about psychology and neurology, I come to reflect the principles of *Human Thinking*. I see parallels to basic concepts of software modelling and try to connect them. As result, I can identify three fundamental kinds of abstraction that our brain uses to understand its real-world environment: *Discrimination*, *Categorisation* and *Composition*. Software developers apply these concepts all the time.
- 2003 From previous reflections I recall that a *Whole* item knows about the properties of its *Part* items. I try to identify such meta properties, read about *Shape*, *Depth*, *Movement* as well as *Colour* in my Psychology literature and finally stumble about *Dimensions* as known from physics. The position and extension that part items span up within their whole item do not only exist in *Space* and *Time*, but possibly also in other kinds of dimensions like *Mass* or *Force*.
- 2003 Since CYBOL contains all concepts that are necessary to model knowledge, including *Categorisation* (inheritance), a CYBOI written in *Java* causes unnecessary overhead. The interpreter's main tasks (input/ output- and memory handling as well as processing instructions) are anyway situated close to hardware so that I start to reimplement CYBOI in the *C* programming language.
- 2004 An open problem that had caused me many headaches was the handling of instructions. I remember the unification of communication models which represented *States* that could be transformed into each other by help of *Translators*. Journeys into *Systems Theory* and a consideration of the *Black Box* concept show the solution: *State*

and *Logic* knowledge need to be separated! The logic contains the rules (algorithms, operations) after which an input state is translated into an output state. It takes considerable thinking to figure out a common CYBOL structure capable of representing states as well as logic.

- 2004 The early CYBOL definition turns out to be insufficient. In hot but constructive discussions with Rolf Holzmüller, one of my students, the reason gets clearer and clearer to me: CYBOL must consider *two* different hierarchies in just *one* model. One *Model Hierarchy* represents the compound model as such. A second *Meta Hierarchy* holds meta information that a *Whole* knows about its *Parts*. The old CYBOL tried to put everything into XML attributes. The new CYBOL uses XML attributes to link to parts and XML tags to model meta information such as *Properties* and *Constraints*.
- 2004 A lot of time goes into the implementation of CYBOI. Techniques that have to be considered are, among others: signalling, threads, UNIX and TCP/IP sockets.
- 2005 While implementing a CYBOL prototype application, standard programming constructs such as for *Branching* and *Looping* are badly missing. It takes some reflexion to decompose these into their actual elements and to provide the corresponding CYBOL operations by using simple flags.
- 2005 A further problem is the automatic indexing of *Parts* belonging to a common list within a *Whole* model, for which a special *Name Structure* has to be defined and additional CYBOI routines have to be written.
- 2005 It turns out to be problematic to use existing *Graphical User Interface* (GUI) frameworks for input/ output (i/o). Many of them base on OOP principles; all require the adoption of special structures. Since the first version of CYBOI is developed under the *Linux Operating System* (OS), low-level *X Window System Libraries* (Xlibs) functions are used instead of a toolkit.
- 2005 The reception of signals (input) is moved into special threads, one for graphical-, one for textual user interfaces, one for sockets etc. But how can CYBOI handle these signals, if it has no application knowledge? The solution is to, as property of the *receive* operation, hand over a node of the knowledge tree containing possible commands to react to, which are mapped to their corresponding handler operations, in CYBOL.
- 2006 Within two months of being unemployed, I review the implementation of input/ output (i/o) threads, and can correct open conflicts by using *Mutual Exclusion* (Mutex) flags. Also, *Central Processing Unit* (CPU) busy states are now avoided.

- 2006 Branches of the runtime knowledge tree are pointed to by dot-separated names. Since sometimes, it is necessary to address meta information (such as the colour of a menu item) directly, the knowledge path gets extended by a second separation character, which allows to distinguish between whole-part and meta elements.
- 2006 At the time of writing this, open issues yet to be implemented are, for example: compiling CYBOI under *Windows* OS and making use of their GUI functionality; improving the GUI layout and providing a GUI theme infrastructure; specifying serialisation in more detail. For further tasks, see section 14.7!

14.6 Migration to CYBOL

Developers who have developed an interest in the *CYBOP* concepts may eventually want to switch their systems to the *CYBOL* programming language. This section tries to provide these developers with the necessary steps (something like a *Hands-on Guide*) that allow a (more or less) careful *Migration*. *Careful* means that after each step, one should have a running system again, before taking the next step.

The guidelines are especially suited for developers with experience in object oriented techniques, ideally programming in Java. Some knowledge in component oriented programming (such as lifecycle methods) will be helpful in understanding and implementing the following recommendations.

Caution! This migration guide is ad hoc and has not been tested. Use at your own risk!

- Use solely one single *Exception* class. Eliminate any other special exception classes.
- Make sure that instances are only created by special *create* methods, one for every attribute. Eliminate all wild calls of the *new* operator that are spread over the code.
- Create four methods for each attribute (here called *sample*): *createSample*, *destroySample*, *setSample* and *getSample*.
- Collect all *create* methods that are called during system startup and put them into a method *initialise* that is called at startup.
- Introduce a method *finalise* as counterpart to *initialise* and call the *destroy* method of every attribute there. This *finalise* method should be called at system shutdown.
- Insert a new *categorise* method which is called before *initialise*, at system startup. Move all configuration code there.
- Insert a method *decategorise* as counterpart to *categorise*. It should be called at system shutdown, after the *finalise* method. Save (write) any configuration settings there.
- Introduce your own top-most framework class *Item* that all other classes inherit from. This is somewhat difficult. Several classes in a Java application need to inherit from standard JDK classes. For now, only let those classes inherit from *Item* which currently have no (that is the *java.lang.Object*) or an own class as parent.

- Add the lifecycle methods *categorise*, *decategorise*, *initialise* and *finalise* to the *Item* class. All subclasses which use any of these methods should also call the superclass' implementation of the used method.
- Build a general *create* method capable of creating instances from a class name that was handed over as string parameter. This method should determine a class with `Class c = Class.forName(classnameAsString)`. The instance will then be created using `c.newInstance()`.
- Try to fix certain classes (ideally only one called *Controller*) whose task it will be to catch events. Let this class implement all event interfaces used by the system and also implement the necessary event handling methods, enforced by the interfaces.
- Add a method *handle* to the *Controller* class which is called by all other handling methods. It receives an event as parameter and contains simple *if-then* comparisons to filter out the single events and call a method which finally processes the event.
- Move as much method (*not* lifecycle method) functionality as possible into the *Controller* class.
- Structure all other (domain) model classes *hierarchically*. Make sure only *unidirectional* dependencies exist among them.
- Let the *Item* class encapsulate two additional attributes of type *java.lang.Object* and *javax.swing.MutableTreeNode*. Add the corresponding *set* and *get* methods.
- Implement or use an *XML Handler* for reading and writing CYBOL/ XML files.
- Move hierarchical domain model classes into *CYBOL* code. The java code is responsible for reading, altering and writing *CYBOL* files.
- Now structure methods hierarchically as well and move them into *CYBOL* code.
- It is now probably easier to use the *CYBOI* interpreter (written in C) than to further maintain your own interpreter (written in Java).
- That's it. Corrections, hints and improvements are very welcome!

14.7 Call for Developers

CYBOP's concepts want to ease application programming. From now on, developers can focus on pure domain knowledge, which they encode in form of CYBOL (XML) models. For this to become possible, a lot of standard functionality had (and has) to be integrated into the CYBOI interpreter. It does contain (or will so in future) firstly, all kinds of communication mechanisms and secondly, more and more hardware control functionality.

For our *CYBOP Free/ Open Source Software* (FOSS) project, we therefore steadily look for developers with interest in one of the following topics:

- *Graphical User Interface* (GUI) Design: Experience with toolkits like *Qt*, *GTK*, *wxWindows*, *Tcl/Tk* or the like may be helpful, but does CYBOI itself not use these. It integrates low-level graphics routines which currently base on the *Xlibs* libraries for UNIX' *X Window System* (XFree86). The corresponding functionality is still missing for other platforms like MS *Windows* or Apple Macintosh *OS X*.
- *Textual User Interface* (TUI) Design: Experience with libraries such as *ncurses* or *slang* may be helpful. For CYBOI, however, low-level *Console/ Terminal* programming is necessary.
- *Web User Interface* (WUI) Design: In CYBOI, pure CYBOL models get translated into pure HTML models. There is no mix-up of HTML with other code, as known from *JSP* or *PHP*. Although knowledge of the latter and related technologies like *JavaScript* may be helpful, these are not used in CYBOI.
- *Socket Communication*: Sockets are essential for system communication. They differ slightly between platforms and not all kinds have been implemented in CYBOI yet. Some experience with UNIX file-, Win- and TCP sockets is needed.
- *Database Communication*: Traditional mechanisms like *ODBC* or *JDBC* ease and standardise the communication with database systems. It still needs to be figured out whether to use these or better to write our own low-level SQL statements in CYBOI.
- *Data Conversion*: Different kinds of communication require different data transfer models. The same counts for persistently storing data in various file formats. Therefore, a huge number of parsers/ serialisers and CYBOL encoders/ decoders will be needed. Experience with import/ export filters and file format conversion is very welcome.

- OS Concepts: CYBOI is the active core system managing passive knowledge models and running directly on an *Operating System* (OS) or *Hardware*. It already provides signalling and prioritising itself and further OS concepts are planned to be integrated, wherever useful. Help in this would be appreciated.

If you think you might like to work on one of these topics, or just want to try out and learn by doing, or have further questions, just check out our website <http://www.cybop.net> or contact one of the mailing lists mentioned there!

14.8 Abstract

In today's society, information and knowledge increasingly gain in importance. Software as one form of knowledge abstraction plays an important role thereby. The main difficulty in creating software is to cross the abstraction gap between concepts of human thinking and the requirements of a machine-like representation.

Conventional paradigms of software design have managed to increase their level of abstraction, but still exhibit quite a few weaknesses. This work compares and improves traditional concepts of software development through ideas taken from other sciences and phenomenons of nature, respectively – therefore its name: *cybernetics-oriented*.

Three recommendations resulting from this inter-disciplinary approach are: (1) a strict separation of active system-control software from pure, passive knowledge; (2) the usage of a new schema for knowledge representation, which is based on a double-hierarchy modelling whole-part relationships and meta information in a combined manner; (3) a distinct treatment of knowledge models representing states from those containing logic.

For representing knowledge according to the proposed schema, an XML-based language named *CYBOL* was defined and a corresponding interpreter called *CYBOI* developed. Despite its simplicity, CYBOL is able to describe knowledge completely. A *Free-/ Open Source Software* project called *Res Medicinae* was founded to proof the general operativeness of the CYBOP approach.

CYBOP offers a new theory of programming which seems to be promising, since it not only eliminates deficiencies of existing paradigms, but prepares the way for more flexible, future-proof application systems. Because of its easily understandable concept of hierarchy, experts are put in a position to, themselves, actively contribute to application development. The implementation phase found in classical software engineering processes becomes superfluous.

Keywords

Cybernetics Oriented Programming (CYBOP), Knowledge Schema, Ontology, XML-based Programming, Free- and Open Source Software (FOSS), Res Medicinae, Electronic Health Record (EHR), Software Pattern, Programming Paradigm, Software Engineering Process (SEP)

Information

Author: Dipl.-Ing. Christian Heller, Technical University of Ilmenau, Germany

Supervisor 1: Prof. Dr.-Ing. habil. Ilka Philippow (Chair), Technical University of Ilmenau, Germany

Supervisor 2: Prof. Dr.-Ing. habil. Dietrich Reschke, Technical University of Ilmenau, Germany

Supervisor 3: Mark Lycett (PhD), Brunel University, Great Britain

Submission: 2005-12-12; Presentation: 2006-10-04

14.9 Kurzfassung

Informationen und Wissen gewinnen in der heutigen Gesellschaft zunehmend an Bedeutung. Software als eine Form der Abstraktion von Wissen spielt dabei eine entscheidende Rolle. Die Hauptschwierigkeit beim Erstellen von Software besteht in der Überbrückung der Diskrepanz zwischen menschlichen Denkkonzepten und den Erfordernissen einer maschinellen Darstellung.

Herkömmliche Paradigmen des Software-Designs haben ihr Abstraktionsniveau in der Vergangenheit erheblich steigern können, weisen allerdings immer noch etliche Schwächen auf. Diese Arbeit vergleicht und verbessert traditionelle Konzepte der Software-Entwicklung durch Denkansätze anderer Wissenschaftsgebiete bzw. Phänomene der Natur – daher ihre Bezeichnung: *Kybernetik-orientiert*.

Im Ergebnis dieser interdisziplinären Herangehensweise stehen dreierlei Empfehlungen: (1) eine strikte Trennung aktiver Systemkontroll-Software von purem, passivem Wissen; (2) die Verwendung eines Schemas zur Wissens-Repräsentation, welches auf einer Doppel-Hierarchie zur kombinierten Darstellung von Teil-Ganzes-Beziehungen und Meta-Informationen beruht; (3) eine getrennte Behandlung jener Wissens-Modelle, die einen Zustand verkörpern, von solchen, die Logik enthalten.

Zur Darstellung von Wissen gemäß dem vorgeschlagenen Schema wurde eine XML-basierende Sprache namens *CYBOL* definiert und ein dazugehöriger Interpreter genannt *CYBOI* entwickelt. Trotz ihrer Schlichtheit ist *CYBOL* in der Lage, Wissen komplett zu beschreiben. Als Prototyp zum Nachweis der prinzipiellen Funktionsfähigkeit des *CYBOP*-Ansatzes wurde *Res Medicinae*, ein *Free-/ Open Source Software* Projekt, ins Leben gerufen.

CYBOP bietet eine neue Theorie des Programmierens, die durchaus vielversprechend zu sein scheint, da sie nicht nur Mankos bestehender Paradigmen beseitigt, sondern vor allem flexiblere, zukunftssichere Anwendungen ermöglicht. Durch das leicht zu verstehende Hierarchiekonzept werden Fach-Experten in die Lage versetzt, selbst aktiv an der Anwendungs-Entwicklung mitzuwirken. Die in klassischen Software-Entwicklungs-Prozessen zu findende Implementierungsphase entfällt.

Schlagworte

Kybernetik-Orientierte Programmierung (CYBOP), Wissens-Schema, Ontologie, XML-basierende Programmierung, Freie und Quell-offene Software (FOSS), Res Medicinae, Elektronische Kranken-Akte (EHR), Software Muster, Programmier-Paradigma, Software Entwicklungs-Prozess (SEP)

Informationen

Autor: Dipl.-Ing. Christian Heller, Technische Universität Ilmenau

Gutachter 1: Prof. Dr.-Ing. habil. Ilka Philippow (Mentorin), Technische Universität Ilmenau

Gutachter 2: Prof. Dr.-Ing. habil. Dietrich Reschke, Technische Universität Ilmenau

Gutachter 3: Mark Lycett (PhD), Brunel University, Großbritannien

Einreichung: 2005-12-12; Verteidigung: 2006-10-04

14.10 Licences

14.10.1 GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991. Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

- 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
- 5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying,

distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH

ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

One line to give the program's name and an idea of what it does. Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other

than ‘show w’ and ‘show c’; they could even be mouse- clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

14.10.2 GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000, 2001, 2002. Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

TERMS AND CONDITIONS

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept

the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, \LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modifica-

tion. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section *Copying in Quantity*.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections *Verbatim Copying* and *Copying In Quantity* above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section *Modifications* above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents,

unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section *Copying In Quantity* is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the

Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section *Modifications*. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section *Modifications*) to Preserve its Title (section *Applicability and Definitions*) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

14.11 Index

- 1 Tier, 27
- 1:1 Association Patterns, 214, 216
- 1:n Association Patterns, 214, 216
- 2 Tiers, 28
- 3 Tiers, 30
- 4+1 View Model of Architecture, 22
- 4GL, 374

- A Changing World, 243
- A new Kind of Science, 199
- ABDA, 346
- Abrechnungs Datenträger, 346
- Abstract Class, 68
- Abstract Information, 49
- Abstract Levels of a Virtual Machine, 48
- Abstract Model, 1
- Abstract System Concepts, 26
- Abstract Window Toolkit, 109
- Abstract Windowing Toolkit, 362
- Abstraction, 388
- Abstraction Gaps, 4, 19, 376
- Abstractions, 13
- Abstractions of Human Thinking, 202
- Access Control List, 319
- Access Method, 70
- Access Method Elimination, 225
- ACL, 120, 138, 144, 319
- ACR, 352
- Action Pattern, 97
- Active Application (Tool), 187
- Active Component, 120
- ActiveX, 32
- Activity, 243
- Activity Diagram, 127
- Acylic Graph, 102

- ADA, 353
- Adaptability, 2
- Adapter Pattern, 98
- Adaptive Software Development, 16
- Address Space, 26
- ADL, 22, 153, 155, 185
- ADT, 346
- Advice, 117
- AE, 122, 187
- Agent, 109, 120, 189, 250
- Agent as Social System, 256
- Agent Communication Language, 120, 138, 144
- Agent Oriented Programming, 109, 120, 138, 189, 250
- Agent Programming Language of AGOP, 189
- Agent0 Language, 120
- Agile Alliance, 16
- Agile Manifesto, 16
- Agile Methodologies, 16
- Agile Software Development, 13
- AGOP, 109, 120, 138, 189, 250
- AI, 60, 138, 145, 205, 221
- Algol, 57
- Algorism, 213
- Algorithm, 209, 265
- Allocated Area, 76
- Allocation, 52
- Alternative, 53
- ALU, 50
- AM, 155, 185
- Ambulant Operieren Datenträger, 346
- Amelioration Pattern, 79
- American College of Radiology, 352
- American Dental Association, 353

- AMR, 335
- Analogue Electronic Component, 49
- Analysis, 14, 19, 376
- Analysis Pattern, 79, 92, 155
- Analysis Phase, 122
- AND Operation, 246, 266
- Animalic Nerve System, 250
- Animate World, 243
- ANN, 208, 247
- Anti Pattern, 79, 106
- AODT, 346
- AOP, 109, 117, 131, 186
- AOSD, 117
- API, 72, 110, 120
- Applets, 31
- Application, 26, 124
- Application and Domain, 187
- Application Engineering, 122, 187
- Application Functionality, 187
- Application Language, 125
- Application Layer, 40
- Application MIME Type, 266
- Application Programmer, 51
- Application Programming Interface, 72, 110, 120
- Application Server, 27, 39
- Application Specific Integrated Circuit, 67
- Application Types, 39
- Arabian Numbering, 213
- Archetype, 153, 185, 343
- Archetype Definition Language, 153, 185
- Archetype Model, 155, 185
- Architectural Pattern, 79, 81, 214
- Architecture Description Languages, 22
- Architecture Design Diagrams, 13
- Architecture Diagrams, 19
- Architecture Views, 22
- Arithmetic Logic Unit, 50
- Arithmetic Operation, 266
- Array, 76
- Array as Container, 239
- Array with Meta Information, 328
- Artificial Intelligence, 60, 138, 145, 205, 221
- Artificial Neural Network, 208, 247
- ASD, 16
- ASIC, 67
- Aspect, 109, 117
- Aspect Marker Interface, 117
- Aspect Oriented Programming, 109, 117, 131, 186
- Aspect Oriented Software Development, 117
- Aspect Weaver, 117
- Assembler, 51, 253, 260, 261
- Assembler Object, 86
- Assembly Language, 47, 51, 57
- Assignment, 52
- Association Elimination, 222
- Association of Lisp Users, 58
- Associations, 202
- Associations within Patterns, 214
- Associative Container, 75
- Astronomical Particles as Ontology, 233
- Atom with Electrons, 207
- Attribute, 68
- Audio MIME Type, 266
- Automated Medical Record, 335
- Automated Theorem Proving, 60
- Automatic Storage Management, 58
- Automation Engineering, 248
- Autonomous Systems, 25

- Awareness, 210
- AWT, 109, 362
- B Specification Language, 127
- Back Flow of Waterfall Process, 14
- Backend, 40
- Backend Communication Model, 257
- Backtracking, 137
- Backus Naur Form, 278
- Backward Chaining, 137
- Base Architecture, 107
- Base Class Access, 73
- Base Level, 155, 185
- Base Level in Reflection Pattern, 92
- Basic Behaviour in Nature, 199
- Basic Input/ Output System, 181
- Basic Patterns for Communication, 259
- Batch Language, 58
- BDT, 346
- Beaming, 388
- Bean, 109
- Behandlungs Datenträger, 346
- Behaviour, 247
- Behaviour Diagram, 127
- Behavioural Design Pattern, 79
- Being, 173
- Belief of an Agent, 120
- Benediktine Approach, 210
- BerliOS Development Portal, 332
- Bidirectional Dependencies in Data Mapper Pattern, 84
- Bidirectional Dependency, 100–102, 107, 110, 185
- Bidirectional Dependency in Java Class Framework, 93
- Bidirectionalism Patterns, 214, 216
- Big Bang Delivery, 14
- Binary, 246
- Binary Arithmetic, 246, 266
- Binary Digit, 49, 266
- Binary System of Arithmetic, 246
- Bio-Cybernetics, 5
- Biological Environment, 252
- Biological System as Ontology, 230
- Bionics, 5
- BIOS, 181
- Bit, 49, 246, 266
- BitTorrent, 34
- Black Box, 248, 249
- Black Box Reuse, 109
- Block, 52
- Block Diagram, 248
- BNF, 278
- BO, 28
- Body, 388
- Bonobo, 32
- Boolean Algebra, 246
- Boolean Logic, 246
- Boolean Operation, 266
- bottom-up, 19
- Brain, 250, 388
- Brain Regions, 176
- Branch, 53
- Branch Tree Node, 77
- Branching, 53
- Broca Area for Language Production, 210
- Broken Type System in Java, 93
- Broker Pattern, 91
- Broker Pattern in CYBOI, 316
- Bundesvereinigung Deutscher Apotheker Verbände, 346

- Bundling of Attributes and Methods, 68, 362
- Bureaucrat Pattern, 100
- Business Logic, 40
- Business Logic Layer, 82
- Business Objects, 28
- Byte, 266
- Byte Code, 51
- C, 57, 58, 68
- C Programming Language, 362
- C Programming Language Correction, 328
- C Programming Language Simplification, 327
- C++, 57, 68, 72, 125
- C++ Programming Language, 362
- C++ Standard Library, 93
- c/s, 34
- C#, 124
- CAD, 232
- cADL, 153
- Call by Reference, 269
- Call by Value, 269
- Callback Event Handling, 101
- Callback Mechanism, 107
- CAM, 232
- CAP, 349
- Capabilities of an Agent, 250
- Capability Based System, 319
- Capability Maturity Model for Software, 19
- Capability Maturity Model Integration, 19
- Capability of an Agent, 120
- Car Model as Ontology, 232
- Card Operating Systems, 354
- Cascade of Change, 73
- CASE, 374
- Case, 53
- CASE Tool, 127, 131
- Categorisation, 166, 204, 214, 234
- Categorisation versus Composition, 229
- Category, 202, 204, 234
- CBD, 109
- CCC, 190
- CCR, 339
- CD, 306
- CDA, 263, 344, 346
- CDISC, 354
- Cell Division, 177
- CEN, 341, 342
- CEN/TC251, 342
- Central Nervous System, 176
- Central Processing Unit, 26, 181, 323
- Central Processing Units, 39
- Chain of Responsibility Pattern, 88, 100, 110
- Change follows Rules, 243
- Chaos, 243
- Chaos Computer Club, 190
- Child Category, 204
- Child Item, 205
- Choice, 53
- Chunking, 180
- CIAS, 345
- CICS, 36
- Circular Reference, 101, 102
- Circular Reference in Java Class Framework, 93
- CL, 68, 139
- Class, 68, 112, 204
- Class Diagram, 127
- Class Hierarchy, 73, 115
- Class Method, 105
- Class Template, 124
- Classification, 68, 152, 204
- Classification of Concerns, 186

- Client, 25, 27, 30, 34
- Clinical and Laboratory Standards Institute, 353
- Clinical Data Interchange Standards Consortium, 354
- Clinical Document Architecture, 263, 344
- Clinical Encounter of an Episode Based EHR, 337
- Clinical Episode of an Episode Based EHR, 337
- Clinical Image Access Service, 345
- Clinical Item of an Episode Based EHR, 337
- Clinical Observations Access Service, 345
- Clinical Terms Version 3, 348
- CLOS, 68, 93
- Closed Loop Control System, 248
- CLSI, 353
- CLU, 93
- Clustering, 39
- CMET, 344
- CMMI, 19
- CMR, 335
- CNS, 176
- COAS, 345
- COBOL, 36
- Code of a Concept, 146
- Code Only Approach, 133
- Code Reduction, 184
- Code View, 22
- Code Visualisation Approach, 133
- Coding Scheme, 152
- Cognition, 210
- Collaboration Diagram, 127
- Collection, 75, 76
- Collection as Container, 239
- Collection Framework, 107
- Collective Code Ownership in XP, 17
- College of American Pathologists, 349
- Colour, 207
- COM, 32
- COM+, 35
- Combinatorial Explosion, 151
- Comite Europeen de Normalisation, 341
- Command Pattern, 97
- Commercial At , 62
- Common Business Oriented Language, 36
- Common Characteristics, 204
- Common Concern, 117, 186
- Common Knowledge Abstraction, 376
- Common Lisp, 68, 139
- Common Lisp Object System, 68, 93
- Common Message Element Type, 344
- Common Object Request Broker Architecture, 35, 109, 131, 345
- Common Warehouse Metamodel, 131
- Communication, 25, 253
- Communication Diagram, 127
- Communication Languages, 25
- Communication Model, 263
- Communication Model by Shannon & Weaver, 256
- Communication Models, 256
- Communication Partners, 25
- Communication Patterns, 257
- Communication Patterns placed in Layered Architecture, 261
- Comparison Operation, 266
- Compiler, 51
- Complement Operation, 246
- Complex Number, 213
- Complexity, 2
- Complexity Crisis, 2

- Complexity Hiding, 248
- Complicated Processing, 218
- Component, 43, 109–111, 113
- Component Based Design, 109
- Component Diagram, 127
- Component Isolation, 110
- Component Lifecycle, 110, 111, 196
- Component Object Model, 32
- Component Oriented Programming, 109, 113, 117, 120, 196
- Component Selector, 113
- Composite Pattern, 87, 88, 99, 100, 220
- Composite Structure Diagram, 127
- Composition, 166, 205, 214, 220, 221, 230, 234, 275, 359, 362
- Compositional Conceptual Scheme, 151
- Compositional Scheme, 151, 347
- Compound, 202, 205, 234
- Compound Model, 234
- Compound Statement, 52
- Compound Structure as Multi Dimensional Container, 325
- Computer, 1, 25
- Computer Aided Design, 232
- Computer Aided Manufacturing, 232
- Computer Aided Software Engineering, 374
- Computer Aided Software Engineering Tool, 127, 131
- Computer Hardware, 252, 313
- Computer Language, 43, 45
- Computer Languages Timeline, 45
- Computer Structure, 48
- Computer Tomograph, 352
- Computer-based Patient Record, 335
- Computer-Computer Communication, 40
- Computerised Medical Record, 335
- Computerised Patient Record, 335
- Concept, 146, 205, 233, 234, 267
- Concept Mix, 218
- Concept of a Horse, 237
- Conceptual Gaps, 19
- Conceptual Interaction, 207, 235
- Conceptual Network, 143
- Conceptual Ontology Representation, 19
- Conceptual View, 22
- Concern, 109, 113
- Concern Interface, 117
- Concern-less Development, 115
- Concurrency Pattern, 79
- Condition, 53
- Conditional Branching, 53
- Configurable System, 182
- Constraint, 209, 235
- Constraint Form of ADL, 153
- Container, 75, 110, 111, 113, 205, 239
- Container Inheritance, 77, 362
- Container Unification, 239
- Containers in CYBOL, 300
- Content of a Document, 61
- Contents of Communication, 256
- Context Driven Testing, 16
- Context of a Language, 274
- Continuity of Care Record, 339
- Contract, 107, 111, 113
- Control Engineering, 248
- Control Software, 192
- Control Structure, 51
- Controller, 248
- Conversation Model by Osgood & Schramm, 256

- Conversion between Communication Models, 257
- Conversion Method of AGOP, 189
- Converter containing Rules, 249
- COP, 109, 113, 117, 120, 196
- Copy Constructor, 77
- CORBA, 35, 109, 131, 345
- CORBAMed, 345
- COS, 354
- Counted Pointer Pattern, 103, 104
- Counting Loop, 55
- CPR, 335
- CPU, 26, 39, 181, 323
- Creational Design Pattern, 79
- Cross Referencing, 62
- Crosscutting Concern, 117, 186
- Crystal Family, 16
- CT, 352
- CTV3, 348
- Customer Information Control System, 36
- CWM, 131
- Cybernetics, 5, 232, 381
- Cybernetics Oriented Interpreter, 168, 313, 381
- Cybernetics Oriented Language, 168, 273, 381
- Cybernetics Oriented Programming, 5, 381
- Cybernetics Oriented Programming Approach, 168
- CYBOI, 168, 313, 360, 362, 374, 376, 377, 381
- CYBOI Architecture, 313
- CYBOI as Capability Based System, 319
- CYBOI as Exokernel, 317
- CYBOI as GUI Renderer, 316
- CYBOI as Hardware Abstraction Layer, 317
- CYBOI as Hybrid Kernel, 317
- CYBOI as Knowledge-Hardware Interface, 313
- CYBOI as Microkernel, 317
- CYBOI as Monolithic Kernel, 317
- CYBOI as Peer to Peer System, 316
- CYBOI as Secure Architecture, 319
- CYBOI as Universal Data Converter, 324
- CYBOI as Virtual Machine, 313
- CYBOI avoiding Access Control Lists, 319
- CYBOI avoiding Inter Process Communication, 317
- CYBOI Control Flow, 321
- CYBOI Data Creation, 325
- CYBOI Data Encapsulation, 325
- CYBOI Development Environment, 329
- CYBOI Distribution and Installation, 330
- CYBOI Error Handling, 329
- CYBOI Functionality, 321
- CYBOI Implementation, 327
- CYBOI Knowledge Container, 314
- CYBOI Lifecycle Management, 322
- CYBOI main Procedure, 322
- CYBOI Model Transition, 324
- CYBOI Operation Execution, 324
- CYBOI Part Dependencies, 321
- CYBOI Process Launching, 322
- CYBOI Signal Checker, 314
- CYBOI Signal Checking, 323
- CYBOI Signal Handling, 323
- CYBOI using External Libraries, 328
- CYBOI using Patterns, 316
- CYBOL, 168, 273, 313, 360, 362, 363, 374, 376, 377, 381
- CYBOL 'abstraction' Attribute, 281
- CYBOL 'channel' Attribute, 281
- CYBOL 'constraint' Tag, 282
- CYBOL 'Hello, World

- ' Example, 297
- CYBOL 'model' Attribute, 281
- CYBOL 'model' Tag, 282
- CYBOL 'name' Attribute, 281
- CYBOL 'part' Tag, 282
- CYBOL 'property' Tag, 282
- CYBOL Algorithm Division Example, 290
- CYBOL Attributes, 281
- CYBOL Communication Diagram, 306
- CYBOL Comparison to other Languages, 301
- CYBOL Conditional Execution Example, 292
- CYBOL Constructs, 284
- CYBOL Definition, 274
- CYBOL DTD, 276
- CYBOL EBNF, 278
- CYBOL External Resources Example, 286
- CYBOL for Any System, 299
- CYBOL Knowledge Designer, 306
- CYBOL Logic Example Constructs, 289
- CYBOL Loop as Operation Example, 291
- CYBOL Meta Constraints Example, 288
- CYBOL Meta Property Example, 285
- CYBOL Model Diagram, 306
- CYBOL Model Viewer, 310
- CYBOL Model-Part Relation Example, 285
- CYBOL Operation Call Example, 289
- CYBOL Organisation Diagram, 306
- CYBOL Presentation and Content Example, 296
- CYBOL Runtime Model Editor, 310
- CYBOL Semantics, 281
- CYBOL Serialised Model Example, 287
- CYBOL Simple Assignment Example, 290
- CYBOL Special Example Constructs, 293
- CYBOL State Example Constructs, 284
- CYBOL Synchronous Execution Example, 293
- CYBOL Syntax, 275
- CYBOL Tag-Attribute Swapping, 283
- CYBOL Tags, 282
- CYBOL Template Diagram, 306
- CYBOL Template Editor, 305
- CYBOL Tool Support, 305
- CYBOL Vocabulary, 276
- CYBOL XSD, 278
- CYBOL-OWL Comparison, 303
- CYBOL-RDF Comparison, 302
- CYBOP, 5, 362, 374, 381
- CYBOP 'abstraction' Attribute, 276
- CYBOP 'channel' Attribute, 276
- CYBOP 'constraint' Tag, 276
- CYBOP 'model' Attribute, 276
- CYBOP 'model' Tag, 276
- CYBOP 'name' Attribute, 276
- CYBOP 'part' Tag, 276
- CYBOP 'property' Tag, 276
- CYBOP Approach, 168
- CYBOP Distinction of Statics and Dynamics, 370
- CYBOP Evaluation, 374
- CYBOP Fiction, 387
- CYBOP Future Topics, 383
- CYBOP Knowledge Schema, 377
- CYBOP Knowledge Triumvirate, 374
- CYBOP Limits, 378
- CYBOP Long-Life Software System, 377
- CYBOP Separation of State- and Logic Knowledge, 373
- CYBOP Usage of a Double-Hierarchy Knowledge Schema, 372
- CYBOP Validation, 369
- CYBOP-Java Analogies, 313

- Cyclic Method Dependencies, 73
- dADL, 153
- Daemon, 32, 89, 192
- DAG, 99, 102, 148, 220
- DAML, 140
- DAML+OIL, 140, 143
- DARPA Agent Markup Language, 140, 143
- Data (Definition), 135
- Data and Rules, 167
- Data Bush, 190
- Data Control Language, 61
- Data Definition Form of ADL, 153
- Data Definition Language, 28, 61
- Data Garden, 190
- Data Hiding, 70
- Data Manipulation Language, 45, 61
- Data Mapper, 257
- Data Mapper Layer, 84
- Data Mapper Pattern, 84, 187, 259, 261, 267
- Data Mapper Pattern in CYBOI, 316
- Data Mapping, 40
- Data Mining, 39, 218
- Data Path, 50
- Data Source Layer, 43, 82, 84, 259
- Data Structure, 249
- Data Transfer, 64
- Data Transfer Object, 257, 261
- Data Transfer Object Pattern, 86, 187, 260, 267
- Data Transfer Object Pattern in CYBOI, 316
- Data Value, 52
- Data Warehouse, 39
- Database, 28, 187, 261
- Database Data Structure, 102
- Database Layer, 40, 84
- Database Management System, 28, 187, 259
- Database Server, 28
- Database Storage, 64
- Date and Rule, 135, 137
- Datenträger, 346
- DB, 28, 187, 261
- DBMS, 28, 187, 259
- DCD, 141
- DCE, 34
- DCL, 61
- DCOM, 35
- DCOP, 32
- DDE, 32
- DDL, 28, 61
- DE, 122, 131, 187
- Debian GNU/Linux Package Definition, 288
- Decision of an Agent, 120
- Decision Support, 218
- Declaration, 52
- Declarative Programming, 60
- Declarative Programming Language, 47
- Decoder, 256
- Decoupling, 202
- Decoupling of Components, 112
- Delegation Pattern, 98
- Delphi, 70, 109
- Dependency Injection, 110
- Deployment Diagram, 127
- Deployment View, 22
- Depth, 207
- Derivation, 210
- Design, 14, 19, 376
- Design Pattern, 79, 97, 214
- Design Phase, 122, 133
- Design Reflections, 214

- Design Technique, 79
- Design Time Structure, 239
- Design View, 22
- Desktop Communication Protocol, 32
- Desktop Publishing, 63
- Desoxy Ribo Nucleic Acid, 177, 388
- Deterministic Behaviour, 247
- Deutsches Institut für Medizinische Dokumenta-
tion und Information, 346
- Deutsches Institut fuer Medizinische Dokumen-
tation und Information, 354
- Deutsches Institut fuer Normung, 341
- Development Aspect, 117
- Development Concern, 186
- Development for Reuse, 122
- Development in XP, 17
- Development View, 22
- Device Independent Format, 63, 66
- Dialectic Dualism, 246
- Dialectical Relationship between Whole and Part,
235
- DICOM, 352
- DICOM Message Service Element, 352
- Dictionary, 77, 151
- Differential Behaviour, 248
- Difficult Standardisation of Software Models,
219
- Digital Imaging and Communications in Medicine,
352
- Digital Logic, 49
- Digital Logic Circuit, 266
- Digital Medical Record, 335
- Digital Technology, 246
- DIMDI, 346, 354
- Dimensions, 387
- DIMSE, 352
- DIN, 341
- Direct Communication, 253
- Directed Acyclic Graph, 102, 148
- Directed Acyclical Graph, 99, 220
- Directories, 39
- Discrimination, 166, 202, 214, 234, 275
- Distributed Application, 91
- Distributed Component Object Model, 35
- Distributed Computing Environment, 34
- Distributed System, 34
- DML, 45, 61
- DMR, 335
- DNA, 177, 388
- DNS, 37
- do-while, repeat-until, 55
- DOC++, 62
- DocBook, 61
- DocBook DTD, 285, 335
- Document Content Description, 141
- Document Publishing, 64
- Document Type Definition, 276
- Document View MVC Variant, 87
- Domain, 124
- Domain Communication Model, 257
- Domain Engineering, 122, 187
- Domain Knowledge, 187
- Domain Layer, 43
- Domain Logic Layer, 82
- Domain Model, 221, 259–261, 263
- Domain Model Layer, 84
- Domain Model Pattern, 83
- Domain Modelling, 64
- Domain Name Service, 37
- Domain Patterns, 40

- Domain Specific Language, 125, 131
- Domain-Application- versus System-Knowledge Separation, 187
- Double Hierarchy, 235
- Double Word, 266
- Doxygen, 62
- DSDM, 16
- DSL, 125, 131
- DT, 346
- DTD, 276
- DTO, 86, 187, 257, 260, 261, 267
- DTO in CYBOI, 316
- DTP, 63
- Dual Model Approach, 155, 185, 343
- Dual Representation, 202
- Dualism, 173
- Duplicate Element, 76
- Duplicate Key, 77
- Duration of Part Processes, 209
- DVI, 63, 66
- Dynamic Behaviour, 267
- Dynamic Data Exchange, 32
- Dynamic Model, 127
- Dynamic Parts of a Framework, 107
- Dynamic Processing, 173
- Dynamic Processing of Knowledge, 360
- Dynamic System Development Method, 16
- Dynamic Typing, 58, 124
- Dynamics, Terms and Synonyms, 43

- EBES, 345
- EBES Expert Group 9, 345
- EBNF, 276, 278
- EC Logic, 137
- EDI, 346
- EDIF, 67
- EDIFACT, 345
- eDonkey, 34
- EEG9, 345
- EEPROM, 181
- Egocentric Form of Language, 210
- eHC, 354
- EHR, 7, 113, 153, 222, 233, 331, 335, 342, 359, 363
- EHR Communications Task Force, 342
- EHRcom Task Force, 342
- Eiffel, 124
- EIR, 233
- EJB, 28, 187
- Electric Voltage, 49
- Electrically Erasable Programmable ROM, 181
- Electronic Circuit, 49, 67
- Electronic Data Interchange for Administration, Commerce and Transport, 345
- Electronic Data Interchange Format, 67
- electronic Health Cards, 354
- Electronic Health Record, 7, 113, 153, 222, 233, 331, 335, 342, 363
- Electronic Health Record as Core Model, 339
- Electronic Insurance Record, 233
- Electronic Medical Infrastructure, 335
- Electronic Medical Record, 335
- Electronic Patient Record, 335
- Element, 205
- Elementary Particle, 49
- Eliminated Sub Associations, 222
- EMI, 335
- empty, 362
- EMR, 335
- Encapsulated PostScript, 286
- Encapsulation, 70, 225, 269

- Encoder, 256
- Endless Loop, 102
- Endless Loop through Pattern, 216
- Enterprise Java Bean, 187
- Enterprise Java Beans, 28
- Enterprise Resource Planning System, 249
- Entity Relationship Diagram, 127
- Entity Relationship Model, 84, 217, 259, 261
- Entity-Relationship Model, 28
- Entrance and Exit of a Control Structure, 51
- Entropy, 243
- Enumeration, 76
- Enumerative Coding Scheme, 150
- Enumerative Scheme, 150, 347
- Enumerative-compositional Scheme, 151
- ENV 12265, 342
- ENV 13606, 342
- Envelope Letter Pattern, 103
- Episode Based EHR, 222, 337
- EPR, 335
- EPS, 286
- ERD, 127
- ERM, 28, 84, 217, 259, 261
- ERP, 249
- Error (of Syntax, Logical, at Runtime), 329
- Ethernet, 37
- European Board of EDI Standardisation, 345
- European Committee for Standardization, 342
- Event, 194
- Event Handler Pattern, 100
- Event Queue, 194
- Evidence Based EHR, 338
- Evolutionary Process, 14
- Example, 7
- Execution, 26
- Execution View, 22
- Existential Conjunctive Logic, 137
- Expert System, 60, 137
- Explicit Knowledge, 135
- Expression, 52
- Extended Backus Naur Form, 276, 278
- Extended Meta Language, 127
- Extended ML, 127
- Extensible Markup Language, 61, 64, 131, 140–142, 274–276, 278, 281
- Extensible Markup Language Processing, 39
- Extension of Ontologies, 231
- External Knowledge, 182
- External Server, 89
- Extreme Programming, 13, 16, 17, 19
- Extrinsic Property, 210
- Factory Method Pattern, 103
- Falsifying Polymorphism, 77
- FastTrack, 34
- Fat Client, 30
- FDD, 16
- FDDI, 37
- FDL, 331
- Feature Driven Design, 129
- Feature Driven Development, 16
- Feature Model, 19, 122, 127, 129, 131
- Feature Modelling, 19, 129
- Feature Oriented Domain Analysis, 122, 129
- Feature RSEB, 122
- FeatuRSEB, 122
- Feedback Control System, 248
- Feedback Loop, 14
- Fiber Distributed Data Interface, 37
- Field Programmable Gate Array, 67
- File, 266

- File System Structure, 102
- File Transfer Protocol, 37
- Firewall, 39
- First Order Predicate Logic, 153
- First Principles of Demonstration, 244
- Flash ROM, 181
- Flat Data Structure, 86
- Flex Machine, 319
- Flexibility, 2, 73
- FLOSS, 331
- Flux, 243
- FODA, 122, 129
- FOPL, 153
- for, for-next, 55
- Force, 208
- formal, 19
- Formal Knowledge Representation Language of
AGOP, 189
- Formal Programming Language, 273
- Formality, 273
- Fortran, 57
- Forward Chaining, 137
- FOSS, 17, 331
- Foundation Level of an Ontology, 145
- Four Views Model, 22
- Fourth Generation Languages, 374
- FPGA, 67
- FQN, 113
- FR, 37
- Fraction Number, 213
- Fragile Base Class (Problem), 73
- Fragile Base Class Problem, 216, 229
- Frame Relay, 37
- Framework, 43, 107, 110
- Framework Example, 225
- Free and Open Source Software, 17, 331
- Free Documentation License, 331
- Free Software Foundation, 332
- Free/ Libre Open Source Software, 331
- Freenet, 34
- Freshmeat Development Portal, 332
- Frontend, 40, 261
- Frontend Communication Model, 257
- Frozen Spot, 107
- FSF, 332
- FTP, 37
- Fully Qualified Name, 113
- Function Template, 124
- Functional Elements, 248
- Functional Language, 58
- Functional Model, 127
- Functional Programming, 47, 58, 374
- Fundamental Pattern, 79
- Fuzzy Logic, 247
- GALEN, 151, 350
- GALEN Common Reference Model, 350
- GALEN CRM, 350
- GALEN Representation and Integration Lan-
guage, 350
- Gang of Four, 79
- Garbage Collector, 104
- Gate, 49
- GDT, 346
- GEHR, 153, 343
- Gene, 388
- Genealogy, 205
- General Practitioners, 346
- General Public License, 331
- General Purpose Information Component, 342
- General Purpose Language, 58, 125

- Generalisation, 204
- Generalised Architecture for Languages, Encyclopedias and Nomenclatures in Med., 151, 350
- Generative Programming, 131
- Generator, 131
- Generic Programming, 124, 131
- Generics, 124
- Geraete Datenträger, 346
- German College of Community Physicians, 346
- Gimp Toolkit, 362
- Global Access, 111
- Global Access Patterns, 214, 216
- Global Data Access, 105, 106
- Global Medical Device Nomenclature, 353
- Global Variable, 68
- Glue Language, 58
- GMDN, 353
- GNU/Linux, 362
- Gnutella, 34
- GoF, 79
- Good European/ EHR, 153
- Good European/ Electronic Health Record, 343
- Goto (Jump) Command, 53
- GP, 131, 346
- GPIC, 342
- GPL, 58, 125, 331
- GRAIL, 350
- Grammar, 210
- Grammar of a Language, 275
- Granularity, 229, 230
- Granularity of Items, 222
- Graphical Frame consisting of Components, 207
- Graphical User Interface, 33, 87, 109, 196, 261, 263, 285, 359, 362
- Graphology of a Language, 274
- GraphViz DOT, 125
- Grouping Patterns, 214, 216
- GTK, 362
- GUI, 33, 87, 109, 196, 261, 263, 285, 359, 362
- GUI Design, 64
- GUI Layouts, 285
- Hard Disk Drive, 181, 196, 255, 266
- Hardware, 1, 173, 192
- Hardware Architecture, 48
- Hardware Description Language, 67
- has-a Relation, 237
- Hash Map, 77
- Hash Map as Container, 239
- Hash Table, 77
- Hash Table as Container, 239
- Hashtable, 77
- Haskell, 58
- HCI, 33
- HDD, 181, 196, 255, 266
- HDL, 67
- HDTF, 345
- Health Issue of an Episode Based EHR, 337
- Health Level Seven, 37, 225, 344
- Health Professional Cards, 354
- Healthcare Domain Taskforce, 345
- Healthcare Xchange Protocol, 263, 347
- Hidden Patterns in CYBOL, 301
- Hierarchical Algorithm, 224
- Hierarchical DBMS, 28
- Hierarchical Knowledge, 166
- Hierarchical Model Approach, 220
- Hierarchical Model View Controller, 267
- Hierarchical Model View Controller Pattern, 88, 100, 221

- Hierarchical Modularisation of Control Structures, 51
- Hierarchical MVC, 359
- Hierarchical Procedure, 205
- Hierarchical State, 205
- Hierarchy, 148, 166, 205
- Hierarchy as Principle, 222
- High Performance Technical Computing, 39
- High Voltage, 49
- Higher Level Languages, 51
- Higher Levels of a Computer Structure, 48
- HIS, 331, 344, 346
- HL7, 37, 225, 344, 346
- HL7 CDA, 342
- HMVC, 88, 100, 221, 267, 359
- Hook Method Pattern, 104
- Horizontal Market Framework, 107
- Horizontal Scaling, 39
- Horizontal System, 39
- Hospital Information System, 331, 344, 346
- Host, 36
- Hot Spot, 107
- HPC, 354
- HPTC, 39
- HTML, 61, 64, 363
- HTTP, 31, 37, 286
- Human Being, 252
- Human Body, 251
- Human Body having Organs, 207
- Human Senses, 251
- Human Thinking, 166, 199, 214
- Human User, 33
- Human-Computer Communication, 40
- Human-Computer Interaction, 33
- Human-Human Communication, 40
- HXP, 263, 347
- Hybrid Machine Language Level, 51
- Hyper Text Markup Language, 363
- Hyper Text Transfer Protocol, 286
- Hypertext Markup Language, 61, 64
- Hypertext Transfer Protocol, 31, 37
- i/o, 39
- IC, 50, 181
- ICD, 150, 348
- ICD-03, 349
- ICD-10, 348, 349
- ICD-9-CM, 349
- ICF, 352
- ICHPPC, 352
- ICN, 349
- ICNP, 151, 349
- ICPC, 352
- ICR, 335
- Identifier, 52
- Idiom, 103
- Idiomatic Pattern, 79, 103, 214
- IDL, 109, 131, 345
- IIOP, 345
- Image MIME Type, 266
- Imperative (Command Oriented) Programming Language, 47
- Imperative Language, 58
- Implementation, 14, 19, 360, 376
- Implementation Inheritance, 73
- Implementation Phase, 122, 133
- Implementation Source Code, 13
- Implementation View, 22
- Implicit Knowledge, 135
- In-Language DSL, 125
- Inanimate World, 243

- Incremental Process, 14
- Independent Loops Scenario for Data Forwarding, 91
- Indirect Communication, 255
- Inference, 244
- Inflexible Architecture, 217
- informal, 19
- Informal Language, 273
- Informatics, 1
- Information, 1
- Information (Definition), 135
- Information Age, 1
- Information Flow, 249
- Information Hiding, 70
- Information Processing, 253
- Information Processing Model, 180, 324
- Information Reception, 251
- Information Science, 1
- Information Sending, 251
- Information Society, 381
- Information Technology, 1
- Information Technology Environment, 25, 40, 257
- Inheritance, 72, 107, 204
- Inheritance as CYBOL Property, 299
- Initial Value, 52
- Initialisation, 52
- Inner Class, 68
- Inner Language, 210
- Inner Structure of Software Systems, 40
- Input Method Framework, 107
- Input of a System, 247
- Input/ Output, 39
- Input/ Output Memory, 194
- Inside of a Software System, 43
- Instance, 68
- Instance Diagram, 127
- Instance Tree, 106
- Instantiating Knowledge, 196
- Instantiation, 68
- Instruction Set Architecture, 50, 51
- Integer Number, 213
- Integral Behaviour, 248
- Integrated Care Record, 335
- Integrated Circuit, 50, 181
- Integrated Development Environment, 58
- Integration, 14
- Intentionality, 19
- Inter System Communication, 194
- Inter-Dependency, 102
- Inter-Disciplinary Effort, 4
- Inter-Process Communication, 32
- Inter-Type Declaration, 117
- Interacting Systems, 257
- Interaction, 207
- Interaction and Cooperation, 25
- Interaction Diagram, 127
- Interaction Overview Diagram, 127
- Interconnect, 39
- Interdisciplinary Science, 232
- Interface, 68, 72, 112
- Interface Definition Language, 109, 131, 345
- Interface Pattern, 112
- Internal Memory, 194
- Internal Meta Class in JVM, 93
- Internal Server, 89
- Internalised Natural Language, 210
- International Classification for Nursing Practice, 349
- International Classification of Diseases, 150, 348

- International Classification of Functioning, Disability and Health, 352
- International Classification of Health Problems in Primary Care, 352
- International Classification of Nursing Procedures, 151
- International Classification of Primary Care, 352
- International Council of Nurses, 349
- International Organization for Standardization, 37, 341
- Internet, 31
- Internet Inter ORB Protocol, 345
- Internet Packet Exchange, 37
- Internet Protocol, 31, 37
- Interpreter, 51
- Intersection Operation, 246
- Intra System Communication, 194
- Intrinsic Property, 210
- Introduction, 1
- Inversion of Control Pattern, 106, 110, 120
- IoC, 106, 110, 120
- IP, 31, 37
- IPC, 32
- IPX, 37
- is-a Relation, 237
- Is-a Relationship, 204
- is-of Relation, 237
- ISA, 50, 51
- ISO, 341
- ISO 9735, 345
- ISO OSI Model Layers, 82
- ISO OSI Reference Model, 37
- IT, 25
- IT Environment, 40, 257
- Item, 202, 213, 234
- Itemisation, 362
- Itemisation Patterns, 214, 216
- Iteration in XP, 17
- Iterative Process, 13, 14
- Iterative Structure, 17
- Iterator, 76
- Java, 57, 68, 70, 72, 92, 109, 112, 124, 359, 360
- Java Class Framework, 93
- Java Container Framework, 75
- Java Database Connectivity, 28
- Java Development Kit, 76, 107, 112
- Java Foundation Classes, 87
- Java Message Service, 32
- Java Native Interface, 93
- Java Programming Language, 362
- Java Server Pages Application, 39
- Java Swing Framework, 285
- Java Virtual Machine, 93, 360
- Java-CYBOP Analogies, 313
- JavaDoc, 62
- JDBC, 28
- JDK, 76, 107, 112
- JEDEC, 67
- JFC, 87
- JMS, 32
- JNI, 93
- Job, 26
- Job Control Language, 58
- Join Point, 117
- Join Point Model, 117
- Join Point Representation, 117
- Joint Electron Device Engineering Council, 67
- JPM, 117
- JSP, 39

- JVM, 93, 360
- Kassenärztliche Bundesvereinigung, 346
- Kassenaerztliche Vereinigung Datenträger, 346
- KBV, 346
- KDT, 346
- KE, 135, 265
- Kernel Concepts in CYBOI, 317
- Key-Value Pair, 388
- Key-Value-Pair, 77, 182
- KIF, 138
- Knowledge, 173, 182, 192, 244, 247, 360, 381
- Knowledge (Definition), 135
- Knowledge Abstraction, 265
- Knowledge and System Control, 166
- Knowledge Base of an Agent, 120
- Knowledge Carrier, 255
- Knowledge Engineering, 135, 221, 265
- Knowledge Engineering, Basic Principles, 137
- Knowledge Interchange Format, 138
- Knowledge Level, 155
- Knowledge Level in Reflection Pattern, 92
- Knowledge Management System, 192
- Knowledge Manipulation, 265
- Knowledge Memory, 194
- Knowledge Memory consisting of Compound-
and Primitive Models, 325
- Knowledge Model, 196, 221, 249, 267, 324,
377
- Knowledge Modelling Language, 273
- Knowledge Ontology, 230
- Knowledge Query and Manipulation Language,
139
- Knowledge Representation, 135, 230
- Knowledge Representation Principles, 137
- Knowledge Schema, 5, 199, 239, 265, 381, 388
- Knowledge Schema with Meta Information, 234
- Knowledge Specification, 225
- Knowledge Template, 196, 267, 324, 377
- Knowledge Tree, 190, 267, 269, 374
- Knowledge Triumvirate, 374
- Knowledge-Hardware Connection, 192
- Kommunikations Datenträger, 346
- KParts, 35
- KQML, 139
- KV Nordrhein, 346
- KVDT, 346
- Kybernetes, 5
- Labor Datenträger, 346
- Lambda Calculus, 265
- Lamport TeX, 61, 63
- Language, 210
- Language (Channel) as Communication Element,
256
- Language Analysis, 274
- Language History, 45
- Language List, 45
- Language Paradigm, 120
- Large Database, 39
- Lasswell Formula, 256
- Last-In-First-Out, 76
- LaTeX, 61, 63
- LaTeXe, 61, 63
- Layer Supertype Pattern, 82, 115
- Layered Architecture, 261
- Layers, 27, 43
- Layers of a System, 222
- Layers of an Abstract Model, 232
- Layers Pattern, 82, 88
- LDR, 335
- LDT, 346

- Leaf Tree Node, 77
- Learn and Communicate, 17
- Legacy Host, 36
- Legacy System, 109
- Legacy Systems, 36
- Levels of an Abstract Model, 232
- Lexical Scheme, 152, 347
- Lexical Tech, 152
- Lexico-Grammar of a Language, 274
- Lexicon, 148, 210
- Lexicon (Terminology) Query Service, 345
- Library, 51, 107
- libstdc++, 93
- Lifecycle, 111
- Lifecycle Method, 105, 115, 117
- Lifecycle Methods, 111
- Lifecycle of a System, 196
- Lifecycle of Software, 13
- Lifetime Data Repository, 335
- Lifetime Phase of a Component, 111
- LIFO, 76
- Lightweight Process, 26
- Linear Behaviour, 248
- Linguistics, 274
- LISP, 57
- Lisp, 58, 68, 125
- List, 76
- List as Container, 239
- Literate Programming, 62
- Little Language, 125
- Local Process, 32
- Local Variable, 52
- Localised Structures, 199, 243
- Locking, 28
- Logic, 243, 244
- Logic Functions (AND, OR), 49
- Logic Knowledge, 243, 262, 267
- Logic Knowledge Modelling, 281
- Logic Manipulating State, 267
- Logic Model, 249, 265
- Logic, Terms and Synonyms, 43
- Logical Architecture, 40, 43, 257
- Logical Book as Ontology, 231
- Logical Observation Identifiers, Names and Codes, 151, 349
- Logical Programming, 47, 60
- Logical View, 22
- Logiciel Nautilus, 350
- LOINC, 151, 349
- Long Term Memory, 194
- Long-Term Memory, 178
- Lookup Method identifying Components, 113
- Loop Control Structure, 55
- Looping, 55, 363
- Loosely-coupled external Interconnect, 39
- Low Voltage, 49
- Lower Level Instructions, 51
- Lower Levels of a Computer Structure, 48
- LQS, 345
- LTM, 178, 194
- Machine Language, 47, 51, 273
- Macro, 125
- Macrocosm, 205
- Macrocosm as Part of an Ontology, 233
- Main Entry Procedure, 196
- Mainframe, 36
- Maintenance Agency Policy Group, 353
- Manager Class, 107
- Manager Object Pattern, 105
- Map, 75, 77

- Map as Container, 239
- MAPG, 353
- Mapper, 253
- Mapping Containers to CYBOL, 300
- Mapping Rules, 263
- Markup Language, 45, 61
- Markup Tag, 276
- MAS, 120
- Mass, 207, 208
- Mass as Dimension, 387
- Materiality of Language, 210
- Mathematica, 125
- Mathematical Markup Language, 296
- MathML, 296
- MD, 306
- MDA, 131, 133, 189, 345
- Media Streaming, 39
- Mediator Pattern, 84
- Medical Informatics Standards, 341
- Medical Informatics Standards in Res Medicinæ, 356
- Medical Informatics Working Groups, 341
- Medical Information System, 7
- Medical Messaging and Communication Standards, 344
- Medical Record Modelling Standards, 342
- Medical Subject Headings, 351
- Medical Terminology Systems, 347
- Medium of Communication, 263
- Mediums for Knowledge Storage, 255
- Memory, 49, 194
- Memory Management in C++, 104
- Mental State of an Agent, 120, 189, 250
- Mercury, 60
- Merger of traditional and new Concepts, 5
- MeSH, 351
- Message, 194
- Message (What) as Communication Element, 256
- Meta Class, 93
- Meta Information, 92, 207, 234
- Meta Information Hierarchy, 274
- Meta Level, 155, 185
- Meta Level Architecture, 92
- Meta Level in Reflection Pattern, 92
- Meta Meta Level in Reflection Pattern, 92
- Meta Model, 234
- Meta Model Pattern, 79
- Meta Object Facility, 131
- Meta Object Protocol, 92, 117
- Meta Programming, 267
- Metamorphosis of Models, 217
- Metaphysics, 144, 173
- Method, 5, 68
- Method Maturity, 19
- Method Overloading, 74
- Method Overriding, 74
- MFC, 87
- Micro Architecture, 50
- Micro Architecture Hardware, 50
- Micro Architecture Level, 51
- Micro Program, 50
- Micro Program Level, 51
- Micro Program Software, 50
- Microcosm, 205
- Microcosm as Part of an Ontology, 233
- Microkernel Pattern, 89
- Microkernel Pattern in CYBOI, 316
- Microplanner, 137
- Microsoft Foundation Classes, 87

- Middleware, 28, 187
- MIME, 266
- Mind, 250, 388
- Mind and Body, 173
- Mind and Brain, 166
- Misleading Tiers, 4, 40
- Misuse of Inheritance, 229
- Mixed Push-Pull-Pipeline Scenario for Data Forwarding, 91
- Mixin Programming Concept, 117
- Mobility of an Agent, 120
- Model, 234, 374
- Model as Part of Communication, 262
- Model Centric Approach, 133, 189
- Model Driven Architecture, 131, 133, 189, 345
- Model Metamorphosis, 217
- Model Only Approach, 133
- Model Only Technology, 189
- Model Translator, 263
- Model View Controller, 257, 267
- Model View Controller Pattern, 87, 88, 101, 187, 221, 261
- Model View Controller Pattern in CYBOI, 316
- Model-Code Synchronisation, 133
- Modelling Example, 237
- Modelling Mistakes, 4, 158
- Modifier Invariant Function, 73
- Module, 51
- Module View, 22
- Modus Ponens, Inference Rule, 137
- Modus Tollens, Inference Rule, 137
- MOF, 131
- Monades Theory, 205
- Monkey and Banana Problem, 60
- MOP, 92, 117
- Motion, 243
- Motivation, 4
- Motoric Organs, 251
- Movement, 207
- Multi Agent System, 120
- Multi-directional Inter-Dependencies, 40
- Multiple Condition, 53
- Multiple Inheritance, 72
- Multipurpose Internet Mail Extension, 266
- Muscle Cell, 251
- MusicXML, 293
- MVC, 87, 88, 101, 187, 221, 257, 261, 267, 359
- MVC in CYBOI, 316
- MVC Triad, 88, 267
- n Tier, 27
- Named Values, 219
- Napster, 34
- National Committee for Clinical Laboratory Standards, 353
- National Council for Prescription Drug Programs, 353
- National Electrical Manufacturers Association, 352
- National Health Service Information Authority, 348
- National Library of Medicine, 351
- Native Method, 93
- NCCLS, 353
- NCPDP, 353
- NEMA, 352
- Nerve Cell, 251
- Nesting, 199, 243
- Net of Associations, 202
- NetBIOS, 37

- NetDDE, 35
- Network Basic Input/ Output System, 37
- Network DBMS, 28
- Network Dynamic Data Exchange, 35
- Neumann Model of a Computing Machine, 314
- nextElement Method, 76
- Nexus of Appearances, 243
- NHSIA, 348
- NLM, 351
- Nodes, 32
- Noise, 49
- Nomenclature, 148
- NOT Operation, 246
- Notation, 263
- Null Key, 77
- Null Value, 77
- Number, 213
- Number Base System, 213
- Numbering System, 213
- Numeric Languages, 51
- Object, 68, 202
- Object Constraint Language, 127
- Object Diagram, 127
- Object Finder Interface, 84
- Object Linking and Embedding, 32
- Object Management Group, 131, 345
- Object Mapper Implementation, 84
- Object Model, 127
- Object Orientation, 68
- Object Oriented Analysis, 122
- Object Oriented Design, 122
- Object Oriented Model, 217
- Object Oriented Programming, 43, 47, 68, 78, 79, 117, 120, 186, 204, 214, 222, 225, 229, 237, 239, 359, 374
- Object Oriented Programming Systems, 374
- Object Process Diagram, 127
- Object Query Language, 28
- Object Request Broker, 345
- Object Technology Workbench, 189
- Object Tree, 102
- Object-Oriented DBMS, 28
- Object-Oriented Model, 28
- Object-Relational DBMS, 28
- Objectification Patterns, 216
- Observer Pattern, 87, 101, 107, 110
- Occurence of Sub Processes, 209
- OCL, 127
- OD, 306
- ODBC, 28
- ODM, 354
- Odyssey, 350
- Office of Population Censuses and Surveys Classification of Surgical Operations and Procedures, 348
- Offline Thinking, 202, 204
- OIL, 140
- OLE, 32
- OMG, 131, 345
- Omnipresence of Hierarchy, 221
- Online Thinking, 202
- Ontological Layer, 230
- Ontological Level, 222, 230
- Ontology, 37, 43, 115, 120, 143, 144, 152, 173, 216, 222, 230, 265, 363
- Ontology Inference Layer, 140, 143
- Ontology of Principles, 145
- Ontos and Logos, 144
- OO, 68
- OOA, 122

- OOD, 122
- OODBMS, 28
- OOM, 28, 217
- OOP, 43, 47, 68, 78, 79, 117, 120, 186, 204, 214, 222, 225, 229, 237, 239, 359, 362, 374, 377
- OOP Innovations, 78
- OOPS, 374
- OPCS, 348
- OPCS-4, 348, 349
- OPD, 127
- Open Database Connectivity, 28
- Open EHR, 153
- Open Electronic Health Record, 343
- Open Implementation Pattern, 92
- Open Loop Control System, 248
- Open Source Development Portals and Services, 332
- Open Source Health Care Alliance, 334
- Open Source Software, 16, 109, 331, 347
- Open Systems Interconnection, 37
- OpenEHR, 155
- openEHR, 343
- OpenEHR Archetype, 342
- OpenGALEN, 350
- Operand, 52
- Operating System, 26, 27, 37, 39, 51, 89, 173, 182, 192, 194, 313, 362
- Operating System Concepts in CYBOI, 317
- Operation, 52, 248, 266
- Operational Data Modeling, 354
- Operational Level, 155
- Operational Level in Reflection Pattern, 92
- Operator, 52
- OQL, 28
- OR Operation, 246
- ORB, 345
- ORDBMS, 28
- Order of Sub Processes, 209
- Ordered Collection, 76
- Oriented Acyclic Graph, 102
- Orthogonally Persistent Operating System, 319
- Orthography of a Language, 274
- OS, 26, 27, 37, 39, 51, 89, 173, 182, 192, 194, 313, 362
- OS Concepts in CYBOI, 317
- OSHCA, 334
- OSI, 37
- OSS, 16, 109, 331, 347
- OTW, 189
- Output of a System, 247
- Overlapping Code through Concerns, 115
- OWL, 140–143, 303
- Oxford Medical Information System, 352
- OXMIS, 352
- P2P, 34
- PAC, 88
- PAC Agent, 88
- Package Diagram, 127
- Page Description Language, 45, 66
- Paradigm and Language, 45
- Paradigm Overview, 47
- Parallel Layer, 230
- Parameter Forwarding, 111
- Parent Category, 204
- Parent Class, 72
- Parent Item, 205
- Part, 205, 207, 235
- Partial Contact of an Episode Based EHR, 337
- Particle, 202, 205

- Pascal, 57, 58
- Passive Component, 120
- Passive Domain Data (Material), 187
- Patient Carried Record, 335
- Patient Centered Medical Record, 337
- Patient Medical Record, 335
- Pattern, 79, 107
- Pattern Language, 79
- Pattern System, 79
- Pattern Systematics, 214
- Pattern-less Application Development in CY-BOI, 316
- Patterns, 43
- Patterns in CYBOL, 301
- PCL, 66
- PCR, 335
- PDA, 181
- PDF, 63, 66
- PDFTeX, 63
- PDL, 66
- Peer Node, 34
- Peer-to-Peer, 34
- Performance, 2
- Performance of a System, 196
- Peripheral Nervous System, 176
- Perl, 58
- Persistence Layer, 257
- Persistence Model Layer, 84
- Persistent Communication, 255
- Persistent Data, 28, 181
- Persistent Memory, 194
- Persistent Storage, 178
- Person (Patient) Identification Service, 345
- Personal Digital Assistant, 181
- Personal Health Project, 339
- Personal Health Record, 335
- Philosophy, 173
- Philosophy and Mathematics, 244
- Phonology of a Language, 274
- PHP, 339
- PHR, 335
- Phrase as Combination of Terms, 210
- Physical Architecture, 25, 257
- Physical Book as Ontology, 231
- Physical Dimension, 207
- Physical Tiers, 43
- Physical View, 22
- Picture Element, 184
- PIDS, 345
- PIM, 131, 189
- Pipes, 32
- Pipes and Filters Pattern, 91
- Pipes and Filters Pattern in CYBOI, 316
- Pixel, 184
- PL/1, 57, 109
- PL/I, 36
- Platform Independent Model, 131, 189
- Platform Specific Model, 131, 189
- PLD, 67
- Plug & Play Environment, 89
- PMR, 335
- PMS, 331, 344, 346
- PNS, 176
- Point-to-Point Protocol, 37
- Pointcut, 117
- POL, 48
- Polymorphism, 74, 107
- Polymorphism Patterns, 214, 216
- POMR, 337
- Portable Document Format, 63, 66

- Position, 207
- Position as Point, 210
- Post Test Loop, 55
- PostScript, 66, 138
- PPP, 37
- Practice Management System, 331, 344, 346
- Pragmatics of a Language, 274
- Pre Test Loop, 55
- Pre-Conditions in JVM, 93
- Presentation Abstraction Control, 88
- Presentation Client, 30
- Presentation Clients, 27
- Presentation Layer, 40, 43, 82, 88, 261
- Primitive, 266
- Primitive Type, 75
- Primordial Class Loader in JVM, 93
- Principle of Universality, 199
- Printer Control Language, 66
- Prioritising, 194
- Priority of a Signal, 194
- private, 70
- Probabilistic Behaviour, 247
- Problem of First Principles, 244
- Problem Oriented Languages, 48
- Problem Oriented Medical Record, 337
- Procedural Language, 374
- Procedure, 51
- Process, 26, 209
- Process Group, 26
- Process Tree, 102
- Process View, 22
- Processing of Knowledge, 194
- Production Aspect, 117
- Production Concern, 186
- Program Counter, 26
- Program Flow Chart, 51
- Program Keywords, Symbols, Abbreviations, 51
- Programmable Logic Device, 67
- Programmable System, 182
- Programming Language, 45, 363
- Programming Language Generations, 45
- Programming Language One, 36
- Programming Paradigm, 45
- Programming Paradigm Systematics, 47
- Programming Paradigms, 43
- Programming Paradigms as Contrasting Pairs, 47
- Prolog, 58, 60, 137
- Property, 235
- Proportional Behaviour, 248
- protected, 70
- Prototype Software Project, 7
- Proxy Server, 39
- PS, 66, 138
- PSM, 131, 189
- Psychology, 178
- public, 70
- published, 70
- Publisher-Subscriber Pattern, 101
- Pull Scenario for Data Forwarding, 91
- Push Scenario for Data Forwarding, 91
- Python, 58, 68
- QMS, 346
- Qt, 362
- Qualitätsring Medizinische Software, 346
- Quality, 213
- Quantity, 213
- Quantum Computer, 49
- Quark, 49
- Qubit, 49

- Querying, 28
- RADS, 345
- RAM, 76, 106, 181, 196, 255, 266, 267
- Random Access Memory, 76, 106, 181, 196, 255, 266, 267
- Randomness, 199, 243
- RAS, 39
- Rational Unified Process, 14, 22
- RDBMS, 28, 61
- RDF, 140–143, 302
- RDF Schema, 142
- Re-ordering of Code, 62
- READ, 348, 349
- Read Codes, 348, 349
- Read Only Memory, 181
- Real Time Pattern, 79
- Reasoning, 244
- Receiver (Whom) as Communication Element, 256
- Recommendation for Pattern Usage, 216
- Record, 68
- Recursion, 51, 99, 100
- Recursion Patterns, 214, 216
- Redundant Code, 115
- Redundant Meta Information in Java Class Framework, 93
- Reentrant Structure, 14
- Reference Information Model, 225, 344
- Reference Model, 155, 185
- Refined Message Information Model, 344
- Reflection Pattern, 92, 155
- Reflective Mechanisms and their Negative Effects, 93
- Reflective Technique, 185
- Regenstrief Institute, 349
- Register, 49
- Registers, 26
- Registry Object Pattern, 105
- Regress of Reasons, 244
- Relation, 207
- Relation (is-a, has-a, is-of), 237
- Relational Database, 137
- Relational Database Management System, 61
- Relational DBMS, 28
- Relaxed Layered System, 82
- Release Planning, 17
- Reliability, Availability, Serviceability, 39
- Remote Communication Model, 257
- Remote Method Invocation, 30
- Remote Procedure Call, 30, 37, 347, 352
- Remote Server, 35
- Repetition, 199, 243
- Requirements, 14
- Requirements Analysis Document, 13
- Requirements Document, 19
- Res Medicinae, 7, 168, 381
- Res Medicinae Application Prototype, 331
- Res Medicinae Contributors, 334
- Res Medicinae Core Model, 339
- Res Medicinae Development Tools, 333
- Res Medicinae Knowledge Separation, 360
- Res Medicinae Project, 331
- Res Medicinae Requirements Analysis, 335
- Res Medicinae Requirements Document, 335
- Res Medicinae Steps of Realisation, 357
- Res Medicinae Student Works, 357
- Res Medicinae Topological Documentation, 359
- Res Medicinae with Nested Views, 359
- Resource Access Decision Service, 345
- Resource Description Framework, 140–143, 302

- Resource Grouping, 26
- Responder Pattern, 100
- Result (Effect) as Communication Element, 256
- Reusability, 73
- Reuse, 2
- Reuse driven Software Engineering Business, 122
- Reuse through Parameterisation, 124
- Rexx, 58
- Rich Client, 30
- RIM, 225, 344
- RKI, 346
- RM, 155, 185
- RMI, 30
- RMIM, 344
- Robert Koch Institut, 346
- Robot, 1
- Role of a Component, 113
- ROM, 181
- Roman Numbering, 213
- Root Tree Node, 77
- Roundtrip Engineering, 189
- Roundtrip Engineering Approach, 133
- RPC, 30, 37, 347, 352
- RSEB, 122
- RTTI, 92
- Ruby, 125
- Rules, 243
- Rules of Translation, 263
- Run Time Type Identification, 92
- Running of a System, 196
- Runtime Structure, 239
- RUP, 14, 22
- Savannah Development Portal, 332
- Scalability, 39
- SCDI, 353
- Scenarios, 22
- Schema, 205, 374
- Schema for Object Oriented XML, 141
- Schema with Meta Information, 234
- Scheme, 58
- Schemes of Terminology, 149
- Sciences as Ontology, 232
- Scientific Inventions, 1
- SCIPHOX, 346
- Scribe, 61
- Script Oriented Programming, 58
- Scripting Language, 47, 58
- Scrum, 16
- SDL, 127
- SDM, 354
- SDO, 342
- Secure Socket Layer, 39
- Security by Design, 110
- Security in CYBOI, 319
- Self Awareness, 202, 250
- Self-calling Assumptions of a Class Method, 73
- Semantic Gaps, 19
- Semantic Link, 146, 148
- Semantic Web, 140, 141, 144
- Semantics, 210
- Semantics of a Language, 274, 281
- Semi Structured Model Approach, 219
- semi-formal, 19
- Semi-Formal Diagrams, 273
- Sender (Who) as Communication Element, 256
- Sensoric Organs, 251
- Sensoric Type of Terms, 202
- Sensory Memory, 178, 194
- Sentence as Combination of Terms, 210

- SEP, 13, 19, 122, 129, 133, 189, 335, 376, 381
- Separation of Concerns, 113
- Separation of Contract and Implementation, 113
- Separation of Interface and Implementation, 112
- SEQUEL, 61
- Sequence, 75, 76
- Sequence Diagram, 127
- Sequence of (Mapping) Rules, 265
- Sequence Package Exchange, 37
- Sequenced Step, 51
- Serial Line Internet Protocol, 37
- Serialising Knowledge, 196
- Server, 25, 27, 34
- Server Process, 27
- Service Manager for Components, 113
- Service Oriented Architecture, 352
- Servlets, 31
- Session, 26
- Set, 76
- Set as Container, 239
- SGML, 61, 64, 276
- Shape, 207
- Short Term Memory, 194
- Short-Term Memory, 178
- Shutdown of a System, 196
- Shutdown Phase of a Component, 111
- Side Effect, 58, 269, 374
- Signal, 49, 194
- Signal Loop, 194
- Signal Memory, 194
- Signal Pattern, 97
- Signal Processing, 253
- Signal to Noise Ratio, 49
- Simple Mail Transfer Protocol, 37
- Simple Object Access Protocol, 35, 352
- Simplified Layered Architecture, 262
- Single Inheritance, 72
- Single Model Approach, 217, 225
- Singleton Pattern, 103, 105, 107, 117
- Six Pack Model, 122
- Size as Difference, 210
- SLIP, 37
- Small Servers, 32
- Smalltalk, 68, 92, 93, 125
- SML, 58
- SMP, 39
- SMTP, 37
- SNOMED, 151, 349
- SNOMED Clinical Terms, 349
- SNOMED CT, 153, 349
- SNOMED International, 349
- SNOMED Reference Terminology, 349
- SNOMED RT, 349
- SNR, 49
- SOA, 352
- SOAP, 35, 352
- SoC, 113
- Social Form of Language, 210
- Socket Communication Mechanisms, 362
- Software, 1, 173
- Software Architecture, 22
- Software as Passive Knowledge and Active Control, 166
- Software Component, 109
- Software Crisis, 2, 381
- Software Design, 2
- Software Engineering, 5, 19
- Software Engineering Process, 13, 19, 122, 129, 133, 189, 335, 376, 381
- Software Framework, 107

- Software Lifecycle, 13
- Software Pattern, 79, 214
- Software Pattern Classification, 79
- Software Product Line, 129
- Solar System consisting of Planets, 208
- Solid State Physics, 49
- SOP, 58
- Source Code, 19
- Source Code Documentation, 62
- Sourceforge Development Portal, 332
- SOX, 141
- Space, 207
- Space as Dimension, 387
- Spatial Dimension, 210
- Special Purpose Language, 58
- Specialisation, 204
- Specification and Description Language, 127
- Specification Language, 45, 127
- Speech Act Theory, 120
- Spin State, 49
- Spiral Process, 14
- SPL, 58
- SPP, 47, 51, 68, 78, 237, 239, 374, 377
- Spread Functionality, 115
- Spread Functionality through Concerns, 115
- SPX, 37
- SQL, 28, 61, 137, 187
- SSL, 39
- Stack, 26, 76
- Stack as Container, 239
- Staged Process, 14
- Stand Up Meeting, 17
- Standalone Systems, 27
- Standard Generalized Markup Language, 61, 64, 276
- Standard Template Library, 75, 124
- Standardisation of Communication between Information Systems in Physician's Offices and Hospitals using XML, 346
- Standards Committee on Dental Informatics, 353
- Standards Development Criticism, 355
- Standards Development Organisation, 342
- Startup of a System, 196
- Startup Phase of a Component, 111
- State and Logic, 5, 243, 374
- State Chart Diagram, 127
- State Knowledge, 243, 262, 267
- State Knowledge Modelling, 281
- State Machine Diagram, 127
- State Model, 249, 265
- State Primitive, 266
- State, Terms and Synonyms, 43
- State- and Logic, 381
- State- and Logic Knowledge, 167
- Statement, 52
- States 0 and 1, 49
- States High and Low, 49
- States On and Off, 49
- Static Constraints of a Framework, 107
- Static Knowledge, 173, 360
- Static Models, 243
- Static Rule, 267
- Static Typing, 57
- Statically Accessible Classes, 106
- Statics and Dynamics, 5, 173, 381
- Statics, Terms and Synonyms, 43
- Steady State, 249
- Steady Upgrading, 219
- STL, 75, 124

- STM, 178, 194
- Stochastic Behaviour, 247
- Strategy Pattern, 87
- Stratum, 230
- Streams, 35
- Strong Coupling, 110
- Strong Coupling between Layers, 100
- Strong Typing, 57
- Struct, 68
- Structural Design Pattern, 79
- Structure by Hierarchy, 221
- Structure Chart, 51
- Structure Diagram, 127
- Structure of a Document, 61
- Structure of this Book, 8
- Structure, Terms and Synonyms, 43
- Structured and Procedural Programming, 68, 78
- Structured Data Type, 68
- Structured DNA, 388
- Structured English Query Language, 61
- Structured Query Language, 28, 61, 137, 187
- Structured- and Procedural Programming, 47, 51, 237, 239, 374
- Studienzentrum Göttingen (Allgemeinmedizin), 346
- Style of a Document, 61
- Sub Category, 204
- Sub Class, 74
- Sub Model, 222
- Sub Platform, 93
- Sub Process, 209
- Subclass, 73
- Subject, 202
- Submission Data Modeling, 354
- Subroutine, 51
- super, 74
- Super Category, 204
- Super Class, 74
- Super Model, 222
- Super Platform, 93
- Superclass, 73
- Superior Class, 72
- SW-CMM, 19
- Swing, 362
- Switch, 53
- Syllogism, 244
- Symbols of a Language, 276
- Symmetric Multiprocessing, 39
- Synapses, 350
- Synchronisation Problems, 218
- Synergy on the Extranet, 350
- SynEx, 350
- Syntactic Sugar, 93
- Syntax, 210
- Syntax of a Language, 275, 276
- Syntax Tree, 102
- System, 26, 247
- System and Knowledge, 182
- System as Part of Communication, 262
- System Constellations, 25
- System Family, 129
- System Family Engineering, 19, 122, 187
- System Integration Language, 58
- System Lifecycle, 196
- System of Sciences, 232
- System Programmer, 51
- System Programming, 47, 57
- System Programming Language, 57
- System- and Application Functionality, 185

- Systematics, 204
- Systematics of Nature, 204
- Systematized Nomenclature of Medicine, 151, 349
- SystemC, 67
- Systems Engineering, 5
- Systems Interconnection, 37
- Table, 77
- Table with constrained Number of Parts, 209
- tADL, 153
- Tagged Values, 219
- Task, 26
- Task Bag, 26
- Task Farm, 26
- Taxonomic Classification, 150
- TC251, 342
- Tcl, 58
- TCP, 31, 37, 352
- TCP/IP, 37
- TD, 306
- Technical Committee 251, 342
- Technical Environment, 252
- Technical Systems, 25
- TEI, 61
- Telephone Network, 37
- Teleportation, 388
- Telnet, 37
- Temperature with Minimum and Maximum, 209
- Template, 374
- Template Form of ADL, 153
- Template Method Pattern, 103, 104
- Ten15, 319
- Term, 146, 202
- Term as Abstraction, 210
- Termination Character-less Arrays, 328
- Terminology, 144, 148, 152
- Terms of a Language, 274, 276
- Terms of First Order, 202
- Terms of Second Order, 202, 204
- Test, 14
- TeX, 61
- Tex, 63
- TeX User Group, 63
- Text Encoding Initiative, 61
- Text MIME Type, 266
- Textual User Interface, 33, 263
- Textual User Interfaces, 362
- The Cathedral and the Bazar, 17
- The Linux Documentation Project, 285, 335
- Thesaurus, 152
- Thin Client, 30
- Thinking, 210
- Third Party Maintenance, 36
- Thread of Execution, 26
- Tier, 27
- Tightly-coupled internal Interconnect, 39
- Time, 209
- Time as Dimension, 387
- Time Index, 202
- Timing Diagram, 127
- TLDLP, 285, 335
- Token Character, 62
- Token Ring, 37
- Tools & Materials Approach, 124, 187
- Top Level Container, 225
- Top Level Model, 233
- top-down, 19
- Top-most Super Category, 222
- Topological Documentation in Res Medicinae, 359

- TP4, 37
- TPM, 36
- Traceability, 19, 129
- Traditional and New Ideas, 5
- Transaction Handling, 28
- Transaction Pattern, 97
- Transactional Database, 39
- Transfer Control Protocol, 31, 37, 352
- Transfer Model, 263
- Transient Communication, 253
- Transient Data, 28, 181
- Transient Memory, 194
- Transient Storage, 178
- Transistor, 49
- Translation, 51
- Translator, 253
- Translator Architecture, 84, 86, 87, 257
- Translator as Part of Communication, 262
- Translator Model, 260
- Translator Pattern, 187, 363
- Transport Protocol Class 4, 37
- Tree, 75, 77, 99, 102, 205, 221
- Tree as Container, 239
- Tree Node, 77
- Tree Structure, 374
- Trigger, Implication, 137
- Triple-Choice CYBOL Template Editor, 305
- TUG, 63
- TUI, 33, 263, 362
- Turing Machine, 265
- Two Level Modelling, 155
- Two Level Separation, 145
- Two's Complement, 266
- Type, 52, 204
- Typeless Programming, 47, 58
- Typeset Code and Documentation, 62
- Typing, 57
- TyRuBa, 60
- UDP, 37
- UI, 40, 187, 363
- UI Framework, 40
- UI Model, 261
- UMDNS, 352
- UML, 14, 19, 43, 68, 92, 122, 127, 131, 217, 248, 306
- UML Diagram Type, 127
- UML Tool, 68, 127, 189
- UMLS, 152, 351
- UMLS Knowledge Source Server, 351
- UMLS Metathesaurus, 351
- UMLS Semantic Network, 351
- UMLS Specialist Lexicon, 351
- UMLSKS, 351
- UN Standard, 345
- Unconditional Branching, 53
- Unidirectional Dependency, 110, 222, 267
- Unidirectional Relation, 205, 216, 232
- Unidirectional Structure, 106
- Unification of Communication Paradigms, 40
- Unified Medical Language System, 152, 351
- Unified Modeling Language, 14, 19, 43, 68, 92, 127, 131, 248, 306
- Unified Modelling Language, 122, 217
- Uniform Resource Indicator, 143
- Uniform Resource Indicator reference, 302
- Union Operation, 246
- United Nations Standard, 345
- Universal Communication, 40
- Universal Interactive Executive, 37, 58, 125

- Universal Medical Device Nomenclature System, 352
- Universal Memory Structure, 239
- Universal Network Objects, 35
- Universality, 243
- Universe, 233
- Universe as Conglomerate, 201
- UNIX, 37, 58, 125
- UNIX Shell Script, 125
- UNO, 35
- Unpredictable Behaviour through Pattern, 216
- URI, 143
- URIref, 302
- Use Case +1 View, 22
- Use Case Diagram, 127
- User Datagram Protocol, 37
- User Interface, 40, 187, 363
- User Interface Model, 261
- User Stories, 17

- V-Model, 14, 19
- V-Modell 97, 14
- Variable, 52
- VB, 58, 109
- VB.NET, 124
- VCL, 109
- VDAP, 346
- VDM-SL, 127
- Vector, 76
- Vector as Container, 239
- Vegetative Nerve System, 250
- Verband der Hersteller von IT Lösungen für das Gesundheitswesen, 346
- Verband Deutscher Arztpraxis Softwarehersteller, 346
- Verilog HDL, 67
- Vertical Market Framework, 107
- Vertical Scaling, 39
- Vertical System, 39
- Very High Level Language, 125
- Very High Speed Integrated Circuit, 67
- VHDL, 67
- VHitG, 346
- VHR, 335
- VHSIC, 67
- Video MIME Type, 266
- Vienna Development Method - Specification Language, 127
- View, Implication, 137
- Virtual Health Record, 335
- Virtual Machine, 48
- Virtual Patient Record, 335
- Virtual Private Network, 39
- Virtual Record (EHR), 338
- Virtual Storage Access Method, 36
- Virtual- and Real World, 173
- Visual Basic, 58, 109
- Visual Component Library, 109
- Visual Impressions of the Human Mind, 207
- VM, 48
- Vocabulary, 148, 210
- Vocabulary of a Language, 274
- Volatile Data, 181
- Volatile Memory, 194
- VPN, 39
- VPR, 335
- VSAM, 36
- W3C, 64
- Waiting Loop, 194
- Waterfall Process, 13, 14
- Web Browser, 31

- Web Client, 31
- Web Client and Server, 31
- Web Ontology Language, 141, 143, 303
- Web Server, 31, 39
- Web User Interface, 263, 363
- Web User Interfaces, 362
- Weight, 208
- Well-Defined Knowledge Paths, 269
- Wernicke Area for Language Recognition, 210
- while, while-do, 55
- Whirlpool Process, 14
- WHO, 348
- Whole, 205
- Whole Part Hierarchy, 274
- Whole-Part Pattern, 99
- Whole-Part Relationship, 235
- Wild Jump, Goto, 51
- Without Capsules, 269
- Word, 266
- Word as Abstraction, 210
- Work Queue, 26
- Workflow, 267
- Workflow Composition, 64
- World Health Organisation, 348
- World Wide Web, 61, 64
- World Wide Web Consortium, 64
- Wrapper Classes in Java, 93
- Wrapper Pattern, 87, 98
- WUI, 263, 362, 363
- WWW, 61, 64
- wxWindows, 362
- x Data Carrier, 346
- x Datenträger, 263
- x Datenträger, 346
- X-Library, 362
- X.226, 37
- X.25, 37
- xDT, 263, 346
- XFree86, 362
- Xlib, 362
- XMI, 131, 141
- XML, 39, 61, 64, 131, 140–142, 274–276, 278, 281, 344, 346, 347, 360, 362
- XML Attribute, 275, 281
- XML Data, 141
- XML Metadata Interchange, 131, 141
- XML Schema, 141, 142, 278
- XML Schema Definition, 276, 278
- XML Tag, 275, 281
- XP, 16, 17, 19
- XSD, 276, 278
- YACC, 125
- Yet Another Compiler Compiler, 125
- Yo-Yo, 19
- Z Specification Language, 127
- Zentralinstitut für die Kassenärztliche Versorgung, 346
- ZI, 346