


Supporting aspect-oriented software development with the Concern Manipulation Environment



W. Harrison
H. Ossher
S. Sutton
P. Tarr

In the past few years, the application of aspect-oriented software development (AOSD) technologies has helped improve the development, integration, deployment, evolution, and quality of object-oriented and other software for a growing community of software developers. The Concern Manipulation Environment (CME) is an open-source Eclipse project that targets aspect-oriented technologies. The CME contains task-oriented tools for usage approaches that apply aspect orientation in different development and deployment scenarios. The CME also provides component- and framework-level support for building aspect-oriented tools for a variety of types of software artifacts.

In the past few years, the application of aspect-oriented software development (AOSD) technologies has helped improve the development, integration, deployment, evolution, and quality of object-oriented and other software for a growing community of software developers, but, as with any new development approach, its adoption has been limited by the availability of supporting technology. AOSD still lacks tools to support a wide range of software development tasks, and there are no standard platforms or reusable and extensible components designed to facilitate the development of new AOSD approaches. The *Concern Manipulation Environment (CME)*¹ is an open-source Eclipse project that is intended to help the technology grow beyond these limitations. As such, it plays two roles. For developers applying aspect-oriented

technology, the CME contains *task-oriented tools*. Such tools generally help a particular community of developers to think about a problem they face and provide a way of performing specific tasks. However, it is seldom possible to reduce a large problem to a series of tasks without a *usage approach* to guide thinking in more focused terms. For researchers and developers of these tools, the CME also provides component- and framework-level support for building such tools in support of their usage approaches.

©Copyright 2005 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/05/\$5.00 © 2005 IBM

A usage approach is a *pattern* for development. It consists of (1) a characterization of a problem domain for which the approach is suitable, (2) a description of how a developer is expected to structure solutions for problems in that domain, and (3) tools or components that support problem solving with that solution structure. Aspect-oriented programming,² for example, is a usage approach that (1) applies to the problem domain of attaching additional behavior to an existing software base, (2) does so by defining *aspects* containing, for example, *advice* that can be attached to the existing base at *join points*, and (3) is supported by the programming language AspectJ**³ and its compiler and runtime library, which together are designed for expressing those constructs and their attachment. Subject-oriented programming,⁴ multidimensional separation of concerns,⁵ composition filters,⁶ adaptive programming,⁷ and mixin-layers⁸ also provide approaches to AOSD that can be described in similar terms, but address other parts of the problem space in different ways.

Many AOSD approaches emphasize the process of composition (discussed in more detail later in this paper), but composition is simply one of the tasks that make up a large-scale development activity. Other development tasks require different tools. We will introduce the notion of an *extraction/composition cycle*, during which software is separated and reintegrated according to *concerns*, and discuss the tools needed to support this cycle. (A concern is any issue of interest in a software system. When a concern is actualized as a development *artifact*, the term concern is also used to refer collectively to the various elements relating to an issue of interest. In this context, the term artifact refers to any item of material or information created or used in the development of a software system.)

This paper provides an overview of the CME, as well as references to more detailed material, and is organized as follows. The next section introduces aspect-oriented software technology, illustrating its application to the problem of separating an open-source base from additional, proprietary enhancements. We then describe the extraction/composition cycle and the role of the CME in supporting it. The following section sketches the open architecture of the CME, emphasizing the ways in which the extensible characteristics of the components can support a variety of AOSD approaches in an

integrated manner. We then discuss experience with the CME that illustrates how proprietary concerns can be separated from openly shared concerns and that shows how the open architecture of CME can be used to enhance support for additional artifact languages.

ASPECT-ORIENTED SOFTWARE TECHNOLOGY

In comparison to typical object-oriented software, aspect-oriented software provides a number of distinct advantages, which are embodied, in particular, in its use of aspects and concerns. In the following subsections we describe these concepts in more detail and provide an example of their application.

Advantages of aspect-oriented software

Even the best-written object-oriented code usually contains classes with fragments that address many different requirements. As in point-of-sale software that must also capture location-dependent tax information, or customer-service software that must produce service logs, the objects and classes created for most applications generally tangle together the handling of a wide variety of concerns. Conversely, the code or design for logging is not local to one class but is scattered across many. The scattering and tangling of code fragments addressing various concerns make it difficult to understand, maintain, evolve, or reuse such software in different tasks or contexts.⁵

Figure 1 illustrates the problems of scattering and tangling in the context of a real-world Java** 2 Enterprise Edition (J2EE**) application server. The application server originally included both basic capabilities and functionality supporting EJB** (Enterprise JavaBeans**) containers, as depicted on the left side of the figure. The code relating to the EJB container functionality was scattered throughout the application server and tangled with the other application server capabilities.

Aspect-oriented software is structured to emphasize the local organization of code, design, and other artifacts with respect to concerns of particular interest, such as the EJB container feature and the basic J2EE application server capabilities. Material localized within the artifacts of a concern can be logically or physically separated from related material that applies to other concerns. A complete application is produced by integrating separate

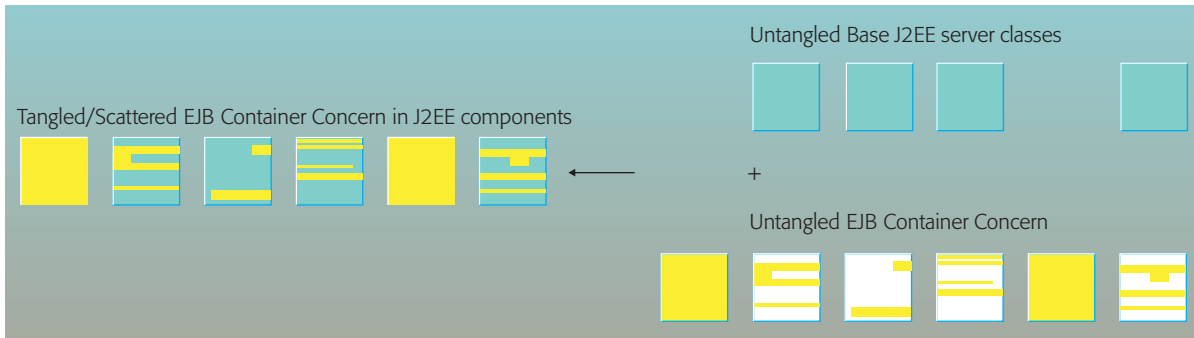


Figure 1
Tangling and scattering of EJB container concerns in an application server

concerns, a process known as *composition*. Composition is performed according to explicit statements about how the concerns should be put together.

Aspects and Concerns

Concerns provide a principled and meaningful way to organize software into manageable, comprehensible, and potentially reusable pieces. This allows all of the software relating to a given concern to be treated together and also isolates it from the software relating to other concerns. For example, sometimes it is useful to separate the basic functioning of an application, such as a portfolio-management system, from its systemic concerns, such as logging, recovery, transaction handling, and security. Concerns like these that cut across the concerns of the basic function are often called aspects, or *crosscutting concerns*.³ At other times it is useful to develop software products by using a common base with a suite of concerns, or *features*, which the developers can choose to combine in various ways to form different products in a product line. The software for such features may not only cut into the base software but also need to be integrated with that of other features as well.⁵ Explicit separation of software into concerns can be especially helpful when software to be integrated later is being developed by independent teams. Because multi-team development is characteristic of open-source software, we expect that the combination of AOSD and open source will prove much more powerful than either alone.

The concern structure of software is not static. Some concerns can be identified early in the software's lifetime, perhaps from the statement of require-

ments, and can be encapsulated at that time. Other concerns may emerge later. For example, a customer may request a new feature (representing a new concern), or an error may occur that highlights the need to treat all of the pieces of software pertaining to that error as a single concern. An important capability of AOSD is that it promotes the identification, encapsulation, and manipulation of concerns *on demand*, to accommodate the ever-changing set of concerns that are relevant to a given software system and its spectrum of stakeholders.

Although AOSD can separate software according to its concern structure, it cannot automatically resolve intrinsic conflicts among concerns. However, AOSD can allow decisions about the handling of conflicts to be expressed at a higher level than the coding or design of features. The CME Concern Manager, discussed later in this paper, does so by addressing relationships among concerns in a general way. The application of composition technology, like a previous generation's application of compiler technology, allows the expression of intent at a high level and employs software tools to carry out the detailed software changes implied by those intents in a mechanical, consistent, and complete manner.

Example: Using AOSD to separate proprietary and open-source concerns

As a way of structuring software, AOSD holds promise for simplifying the creation of software that is partly open-source and partly proprietary. For example, in the J2EE application server shown in Figure 1, the basic capabilities might be offered as open source; whereas, the EJB container support might be a proprietary, value-added capability, as shown on the right side of the figure. This

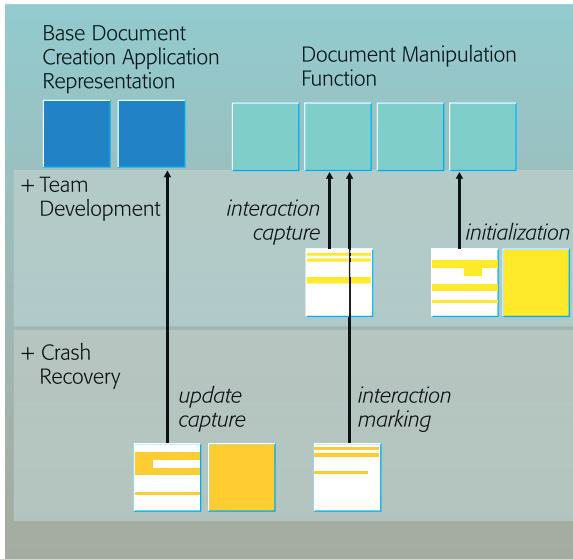


Figure 2
A concern-based approach to separating core functionality from proprietary team development and crash recovery features

combination strengthens the technical and business-case viability of open-source software. This is because it allows developers and researchers to share common frameworks and components but still integrate proprietary support for additional concerns into that software simply, and possibly at a later time. Code-level support for AOSD is already beginning to gain acceptance in practice, and

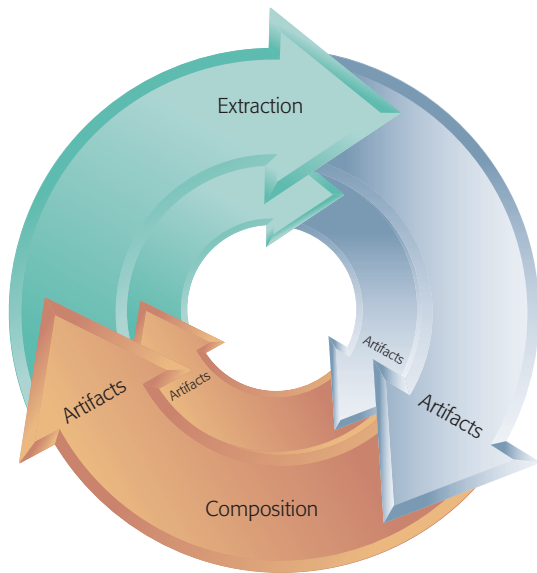


Figure 3
The extraction/composition cycle

component- and design-level support are currently being explored.

Consider, for example, the challenge of providing a common open source for the core functions of a document-creation tool while maintaining the ability to provide proprietary extensions for *team development* and *crash recovery* requirements. A concern-based solution might look like the one shown in *Figure 2*, wherein the open-source base contains objects representing the document itself and providing the document manipulation function. Two proprietary concerns are then developed. The concern for team development captures the developers' interactions and passes them through its own objects, providing connectivity and arbitration, to other team members' software, which injects them into their interaction streams in an appropriate manner. When the document creation application is initialized, the links to other team members must be established as well. The concern for the crash recovery feature intercepts and logs updates to the representation, providing logical recovery points at these intercepted interactions. In both cases, the proprietary features contain new classes as well as material that must be integrated with existing classes in the base application. Directions for how the new material is to be integrated with the base are provided by directives that contain both a query to identify the corresponding points in the new material and the base, and a model of how the new material and base are to be attached. For example, *update capture* may indicate that when any field in an object of some particular class is updated, a method is to be invoked that is defined for that class in the crash recovery concern. That method can log the old value of the field before doing the update.

SUPPORTING EXTRACTION AND COMPOSITION OF SOFTWARE CONCERNS

In the following sections, we introduce the notion of an *extraction/composition cycle*, shown schematically in *Figure 3*, during which software is separated and reintegrated according to concerns. We then introduce the CME and discuss its role in supporting this cycle.

The extraction/composition cycle

Although much discussion of aspect-oriented technology begins with scenarios like the one described previously, in which the different aspects or concerns are *composed*, the availability of compo-

sition engines for concerns also significantly supplements our ability to *decompose* software simultaneously according to many different kinds of concerns. We are then able to effectively *recompose* the separated concerns back into a variety of useful applications. This enables us to bring even more advanced capabilities to bear on the problem of restructuring existing software bodies, such as the J2EE application server (Figure 1), in which scattered, tangled concerns such as the EJB-container concern can be gathered together into a single, encapsulated concern, noninvasively (i.e., logically rather than physically). The ability to decompose existing software into concerns requires the ability to *identify* concerns and the software elements relating to those concerns in the existing software. It also requires the ability to *encapsulate* those elements, either physically or logically, within an additional concern. In the case where encapsulation is physical, identification and encapsulation together are termed *extraction*. Encapsulated concerns can themselves then be subject to further software development activities, including extraction and composition. These complementary capabilities of extraction and composition are the fundamental processes that AOSD adds to the software development repertoire. These activities can and should be applied to artifacts at all stages of the development process, including the requirements analysis, design, coding, and any other relevant stages. Indeed, research into the application of aspect-oriented concepts to requirements, design, and testing has already begun.^{9,10}

The Concern Manipulation Environment

The CME aims to support the extraction/composition cycle throughout the entire software life cycle by providing both a suite of tools for developers and a platform upon which tool builders and researchers can create such tools.

Different AOSD approaches, such as those noted in the introduction to this paper, have been applied to part or all of the extraction/composition cycle, and each has its advocates.³⁻⁸ Unfortunately, the development of tools to realize these different AOSD approaches represents a huge investment of time and effort, as each one must currently be built from scratch or from low-level abstractions. Consequently, the tools represent isolated point solutions and rarely have any ability to interoperate or be integrated. They fail, therefore, to provide devel-

opers and other stakeholders with appropriate and effective ways of thinking about their software's structure and the overall process by which it is developed, deployed, used, and evolved. This has hindered the development of full-life-cycle AOSD. It has also significantly hindered the use of existing tools and paradigms by application developers, who find themselves unable to use available tools and paradigms together. Thus we have dual motivations for developing the CME: to support economy of development of AOSD tools and methods, and to provide wide-spectrum accessibility through a common idiom. These requirements strongly suggest the need for a component-based environment within which research and development can progress and significant development efforts using aspect-oriented tools and methods can be undertaken.

The CME was first demonstrated at the Third International Conference on Aspect-Oriented Software Development (AOSD 2004). The initial software base was developed jointly by teams at the IBM Hursley Development Laboratory and the IBM Thomas J. Watson Research Center. Tools currently available include support for querying software, defining concerns based on queries, modeling concerns, browsing and visualizing software from multiple points of view based on concerns and relationships, and composing aspects and other concerns. The tools apply uniformly to different kinds of artifacts, allowing, for example, relationships across artifacts to be shown and navigated. Most have Eclipse user interfaces, although some (and in particular all the underlying components) can be used outside the context of Eclipse if desired.

A COMPONENT SUITE FOR BUILDING ASPECT-ORIENTED TOOLS

For tool builders and researchers, the CME provides a suite of open-source components upon which they can build AOSD tools. The term "open source" can simply mean making software source code available for open development. However, structuring open-source software as an open-component suite has the additional advantage of allowing individual contributors to focus efforts on enhancing smaller software elements while preserving the ability to connect and share these elements. Being able to leverage one another's work allows the community to avoid building from scratch. Instead, developers

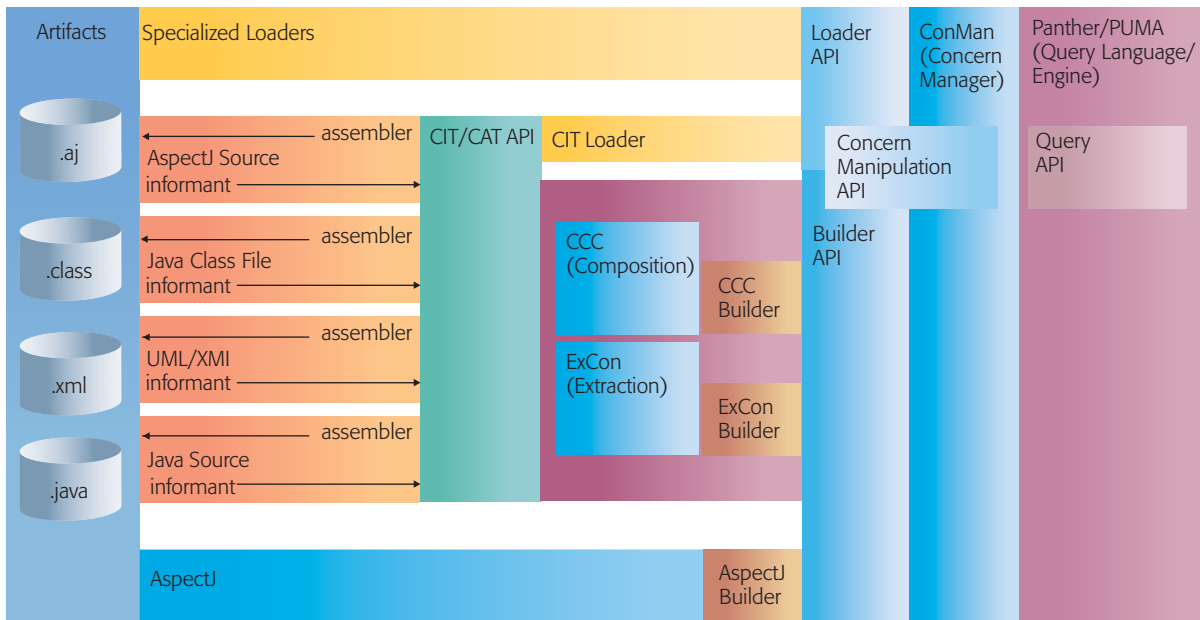


Figure 4
Partial CME component architecture

can share solutions to parts of the overall problem in order to realize larger solutions.

The CME exploits this open-source/open-architecture theme. For maximum flexibility, the initial CME implementation employs *open-points* at several levels of granularity, including both larger abstract components with supporting frameworks and smaller strategies and plug-ins. An open-point in a software system is a place where additional, optional, or externally provided software is expected to be incorporated, presumably in some planned-for manner. Although additions may be made at any point in open-source development, open-points generally provide locations at which the manner of extension is made simpler, clearer, and more robust.

Abstract components

The CME architecture is illustrated in *Figure 4*. **Abstract components**, introduced in bold italics in the text, are the first-order open-points of the CME architecture. Each plays a distinct role in realizing the extraction/composition cycle. These components are described by their APIs (application programming interfaces), and their implementations depend only on the APIs of the other components. Abstract factories are used to shield clients and other components from knowledge of the implementation class structure within a component. The CME also

includes concrete instances of these components to provide both basic functionality and examples of implementation. The initial set of concrete components in the Eclipse project are themselves generally designed to be extended with strategies and plug-ins, giving rise to a second-order set of open-points.

This paper can describe neither the concrete components nor the strategies and plug-in opportunities they present in great detail. However, such details can be found in the collection of research reports comprising References 11–14.

Support for manipulation of concern-structured software in the CME corresponds to the extraction/composition cycle discussed previously. The **Concern Manager** component, shown in Figure 4, provides a starting point for discussion. The Concern Manager is the (abstract) component used to hold and manipulate a model that identifies concerns and describes both the units of software they contain and their relationships. *Units* represent software artifacts, like classes, methods, or UML** (Unified Modeling Language) diagrams. *Concerns* here are groups of concern model elements, listed explicitly (extensionally) or specified as queries (intensionally). *Relationships* can be derived from software artifacts, such as references or dependencies, or inserted by tools, for example, in cases

where relationships have been asserted by the developer. ConMan¹¹ is a concrete component fulfilling the Concern Manager API.

In order to work with a body of software, the concern model must first be populated with units, concerns, and relationships modeling the artifacts themselves (blue) and their structure, according to the AOSD approach being used. This is done by means of abstract components called **Loaders** (yellow). One concrete implementation provided for the Loader API is a generic loader. Generic CME components obtain specific information about artifacts using other components called **Concern Informants** (orange) that are specific to particular kinds of artifacts. The Concern Informant Toolkit (CIT) API is similar in content to the Java Reflection API,¹⁵ but expressed as interfaces instead of abstract classes to permit greater breadth of implementation. In addition, the CIT API applies to a wider variety of software structures than just Java, including simpler artifacts having only a nested container model with interpretable material at the leaves. The CIT API can also express information like weaving directives that are unique to aspect-oriented software.

Once loaded, the concern model can be manipulated by the developer, who can form queries about the material in the various artifacts (evaluated by the **Query** component), and further structure the material into other concerns appropriate for the task at hand. Information in a concern model or accessible through the CIT API can in turn be accessed through the Query API, implemented by concrete components called Panther and Puma.¹² Panther is the query-language processor, and Puma is the query engine. These permit plug-in extension to address the need to add additional relationship types, entity types, data organizations, and query language constructs for various AOSD approaches.

The structuring of software elements into concerns represents a *logical* (non-invasive) encapsulation of those concerns. When a developer has a software model expressed as concerns, those concerns can be used to produce new artifacts, by extraction (*physically* separating concerns into new, disentangled artifacts) and composition (physically integrating concerns into new, composed artifacts). Contributing to CME's flexibility in integrating different AOSD approaches, **Builders** (tan) perform this role, often aided by other components. Some of these operate

on specific kinds of artifacts, as does AspectJ.³ Others, like CCC (the Concern Composition Component),¹³ provide more generic capabilities tailored by strategies to meet specific semantic requirements, as discussed later in this paper. Just as the creation of generic loaders is enabled by the existence of the concern informants, creation of generic builders is enabled by using specific, artifact-format-dependent components called **Concern Assemblers** (orange). The Concern Assembler Toolkit (CAT) API¹⁴ defines a low-level API for creating artifacts that result from the composition of other artifacts. Its interfaces support the copying of source artifacts, the remapping of references they may contain to other artifacts, and the creation of sequences to control the order of execution of appropriate material.

Frameworks for abstract components

It is common for some abstract components to have many implementations, in which case the implementations to be used are provided as plug-ins. Examples in CME include loaders, builders, concern informants, and concern assemblers, which provide access to artifacts across the whole software life cycle. In this context, artifacts are expected to include requirements, use cases, Cosmos concern models,¹⁶ UML designs, source and executable code, test suites, documentation, images, build scripts such as Ant¹⁷ files or make files, packaging constructs such as JAR (Java archive) files, deployment descriptors, and the like. CME contains frameworks to assist in creating concrete components for manipulating such artifacts.

Extension points for concrete components

Many of the CME concrete components contain extension points. This often allows new approaches to be developed as variants of existing approaches, a more economical process than the writing of entirely new concrete components. Thus, CCC provides generic composition capabilities, not based on the details of any specific kind of artifact. However, many artifacts have special composition semantics. For example, composing Java involves simple issues such as handling its particular set of modifiers (*public*, *static*, etc.), and much more complex issues such as ensuring that the composed class hierarchy has no multiple inheritance and handling the construction protocol correctly for composed constructors. These artifact-specific issues are handled by a *rectification strategy*. Other languages can also be composed by CCC, provided suitable rectification

strategies are supplied for them. Similarly, Puma accepts plug-ins that implement artifact-specific query operations, and ConMan accepts loaders and builders as plug-ins.

Supporting new kinds of artifacts on the CME

Assuming the existence of a base of software for manipulating a new kind of artifact, such as a Java bytecode toolkit or an XML (Extensible Markup Language) parser, a few “person weeks” of development are generally needed to implement a CIT API, producing a concern informant for this artifact and enabling the artifact to be loaded using the generic CIT loader. If new kinds of relationships or new attributes are needed to enhance the concern model and query capabilities, a few more person weeks are needed to create additional loader plug-ins and query extensions. If it is desired to compose new kinds of artifacts, either an existing composition engine can be introduced, as was done for AspectJ, or the CCC composition engine can be used by supplying an implementation of the CAT API. Using the existing frameworks, this task generally takes a few person months of effort and results in a new concern assembler. However, if the new kinds of artifacts have particular constraints that are not accommodated by general composition, a new rectification strategy must be implemented for CCC, as discussed previously. Depending on the complexity of the semantics, this can require development time ranging from one or two person weeks to a much more significant effort.

EXPERIENCE

Although the CME is an ongoing open-source effort, there have already been two specific efforts that indicate its value both for development of aspect-oriented software and for development and integration of tools and technologies that support AOSD. The first of these projects involved an application of some of the CME components to a large-scale application server, and the second involved additional tool and component software that has been added to CME since that time.

Separating EJB support from a J2EE application server

Although our Hursley partners have reported on this work in detail elsewhere,¹⁸ we briefly summarize the experience here to illustrate the claim made earlier about separation of open-source and proprietary software.

We and others^{19,20} believe that the application of AOSD technologies to middleware technology is an area of great importance. Thus, an experiment was conducted to separate the deeply tangled support for EJBs from the remainder of the application server shown earlier in Figure 1, a system comprising some 15,000 classes, so that

1. a properly functioning base application server without EJB support could be built that would fail gracefully if EJB capabilities were used, and
2. using tools that could be applied to software in binary form, the EJB support could be added to this base application to build a properly functioning application server with EJB support.

The experiment was completed successfully using a combination of the AspectJ compiler and the loader, concern manager, and query component prototypes from the CME concrete component suite. CME concern modeling was used to model the EJB concern (which evolved as the experiment progressed) and the other components and their relationships. The query capability was used to determine where other components were dependent on the EJB concern, and proved much more efficient than tools tried previously. Those dependencies were then removed, sometimes by object-oriented refactoring and sometimes by refactoring into AspectJ aspects. A side experiment used pure Java and the CCC composition engine (Figure 4) to achieve similar results on a small subset of the cases in which aspects were created.

This experiment illustrates the fact that, should it prove desirable, it is possible to construct software composed of an open-source component as complex as an application server, and then later add proprietary support for additional concerns as complex as the support of EJBs to the binary form of this software.

Extending the CME concrete component suite

Since the conclusion of the EJB-extraction experiment, the set of languages supported by CME has been extended from its pure Java base through the addition of Ant¹⁷ and AspectJ support. Ant is a language similar to that in the UNIX** *make* utility but intended for describing software builds in XML. Because Ant artifacts are an integral part of a body of software, it is desirable to include them in the concern model along with Java artifacts. This was

accomplished simply by building a small Ant loader component using the CME extensible loader architecture, a task which required two person weeks of development. This loader added the units, concerns, and relationships needed to represent Ant to the concern model. It was then immediately possible to use the CME tools mentioned previously to work with Ant artifacts, including navigating from Ant artifacts to the Java artifacts to which they referred.

Support for the AspectJ programming language was also added to the CME. Unlike the Ant loader, the AspectJ loader was implemented by providing a concern informant component for AspectJ that is used by the generic CIT loader and by other components needing information about the AspectJ programs.

CONCLUSIONS

In this paper we have demonstrated the fact that aspect-oriented software technology has a synergistic relationship with open-source development. Not only does AOSD simplify the loosely coordinated development of elements that fit within a broader architecture, but it also promotes the unbundling of software into open and proprietary components. We have highlighted the fact that the CME can materially assist with the separation and reintegration of concerns in software in general and is therefore of interest to anyone needing support for using AOSD in open-source development. Finally, we have outlined the open architecture of the CME and described and illustrated how it reinforces its own use of open source when providing new support for developers using AOSD approaches.

ACKNOWLEDGMENTS

We thank our Hursley partners, Matt Chapman, Andy Clement, Adrian Colyer, Helen Hawkins, and Sian January, for the important roles they have played in this joint development effort. This research was supported in part by the Defense Advanced Research Projects Agency under grant NBCHC020056.

** Trademark or registered trademark of Object Management Group, the Palo Alto Research Center, Inc., Sun Microsystems, Inc., or The Open Group.

CITED REFERENCES

1. Concern Manipulation Environment, Eclipse Foundation, <http://www.eclipse.org/cme/>.

2. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proceedings of the 11th European Conference on Object-Oriented Computing (ECOOP'97)*, Jyväskylä, Finland, June 9–13, 1997, *Lecture Notes on Computer Science*, Vol. 1241, Springer-Verlag, New York (1997), pp. 200–242.
3. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, Budapest, Hungary, June 18–22, 2001, *Lecture Notes on Computer Science*, Vol. 2072, Springer-Verlag, New York (2001), pp. 327–353.
4. W. Harrison and H. Ossher, "Subject-Oriented Programming: A Critique of Pure Objects," *Proceedings, of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, Washington, DC, September 26–October 1, 1993, ACM, New York (1993), pp. 411–428.
5. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 16–22, 1999, ACM, New York (1999), pp. 107–119.
6. M. Aksit, L. Bergmans, and S. Vural, "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach," *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP 1992)*, Utrecht, The Netherlands, June 29–July 3, 1992, *Lecture Notes on Computer Science*, Vol. 615, Springer-Verlag, New York (1992), pp. 372–395.
7. K. Lieberherr, D. Orleans, and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM* **44**, No. 10, 39–41 (2001).
8. D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, Portland, OR, May 3–10, 2003, ACM, New York (2003), pp. 187–197.
9. S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code," *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Denver, CO, November 1–5, 1999, ACM, New York (1999), pp. 325–339.
10. S. Clarke and R. Walker, "Towards a Standard Design Language for AOSD," *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, Enschede, The Netherlands, April 22–26, 2002, ACM, New York (2002), pp. 113–119.
11. W. Harrison, H. Ossher, S. Sutton, and P. Tarr, *Concern Modeling in the Concern Manipulation Environment*, Research Report RC-23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (October 2004).
12. P. Tarr, W. Harrison, and H. Ossher, *Pervasive Query Support in The Concern Manipulation Environment*, Research Report RC-23343, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (October 2004).
13. W. Harrison, H. Ossher, and P. Tarr, *Concepts for Describing Composition of Software Artifacts*, Research Report RC-23345, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (October 2004).
14. W. Harrison, H. Ossher, P. Tarr, V. Kruskal, and F. Tip, *CAT: A Toolkit for Assembling Concerns*, Research Report

- RC-23345, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (April 2002).
15. D. Green, *Trail: The Reflection API*, Sun Microsystems, Inc., <http://java.sun.com/docs/books/tutorial/reflect/>.
 16. S. Sutton Jr. and I. Rouvellou, "Modeling of Software Concerns in Cosmos," *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, Enschede, The Netherlands, April 22–26, 2002, ACM, New York (2002), pp. 127–133.
 17. The Apache Ant Project, The Apache Software Foundation, <http://ant.apache.org/>.
 18. A. Colyer and A. Clement, "Large-Scale AOSD for Middleware," *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 22–26, 2004, ACM, New York (2004), pp. 56–65.
 19. E. Wohlstadtler, S. Jackson, and P. Devanbu, "DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems," *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, Portland, OR, May 3–10, 2003, ACM, New York (2003), pp. 174–186.
 20. T. Cohen and J. Gil, "AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework," *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, June 14–18, 2004, *Lecture Notes on Computer Science*, Vol. 3086, Springer-Verlag, New York (2004), pp. 221–245.

Accepted for publication November 1, 2004.

Published online April 7, 2005.

William Harrison

IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (harrison@us.ibm.com). Mr. Harrison has been with IBM since 1966, and has been associated with the IBM Thomas J. Watson Research Center since 1970. In development, he worked on the design and implementation of IBM operating systems. He has been active first in research on and the design of languages, compilers, and optimization, and subsequently in the design of advanced integrated software development environments. He has been recognized with several Outstanding Contribution and Innovation Awards, the most recent of which was for the innovation and development of *subject-oriented programming*, an early formulation of what has come to be called aspect-oriented software development. He has been a member of the IBM Academy of Technology since 1995.

Harold Ossher

IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (ossheh@us.ibm.com). Mr. Ossher has been a researcher at the IBM Thomas J. Watson Research Center since 1986. His efforts on software environments and tool integration led in 1992 to early work in the area that has come to be called aspect-oriented software development. He is one of the originators of subject-oriented programming, multi-dimensional separation of concerns and Hyper/J, and the Concern Manipulation Environment. A spin-off of this latter research included a framework for performing matching and reconciliation of information models that evolved into EMF Edit and Mapping Frameworks, which has been released as open-source software by IBM. He was General Chair of the First International Conference on Aspect-Oriented Software Development in 2002 and is a member of the AOSD Steering Committee that oversees the conference series.

Stanley Sutton

IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (suttons@us.ibm.com). Dr. Sutton received his Ph.D. in computer science from the University of Colorado in 1990. He has worked in both academia and industry in the areas of middleware, software quality, software process, and aspect-oriented software development. He has been a visiting scientist and a consultant at the IBM Thomas J. Watson Research Center, where he currently works as a software engineer on the Concern Manipulation Environment project. He has served on program committees for the International Conference on Aspect-Oriented Software Development and numerous workshops relating to AOSD. He is one of the originators of the multi-dimensional separation of concerns approach to AOSD and is the principal author of the Cosmos concern-modeling schema.

Peri Tarr

IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (tarr@us.ibm.com). Dr. Tarr received her Ph.D. from the University of Massachusetts in 1996 and has been a researcher at the Thomas J. Watson Research Center since that time. She is the technical co-lead of the Concern Manipulation Environment open-source project, of which she was one of the inventors. Throughout her career, she has worked on many aspects (no pun intended) of the problem of reducing and managing software complexity. She has worked in the areas of software engineering environments, software consistency and inconsistency management, integration, interoperability, and AOSD. She was one of the originators of the multi-dimensional separation of concerns approach to AOSD—one of the seminal pieces of work in this area—and its first realization in the Hyper/J tool, which was later used in various forms in a number of research and industrial efforts. Her research focuses on AOSD throughout the software life cycle and on morphogenic software (software that remains malleable throughout its lifetime). She has served on numerous organizing and program committees for all of the major conferences in software engineering, and she is currently serving as Program Chair for AOSD 2005. ■