Technical note

From dynamic supertypes to subjects: A natural way to specify and develop systems

by W. H. Harrison
H. Kilov
H. L. Ossher
I. Simmonds

When we understand, specify, and develop systems, we use certain concepts and constructs to deal with complexity. Object-oriented (OO) approaches provide good ways for doing so. However, many existing OO approaches (perhaps based on object models used in existing OO languages) cannot solve important problems encountered in large and complex systems. For example, we often have to deal with properties of "things" that cannot be represented in a neat hierarchy. Some of these properties may significantly change with time. Moreover, many of these properties refer to collections of objects without identifying a single object as "owner" of each property. The authors of this technical note have separately proposed approaches for solving these problems, but at very different stages of the development life cycle. However, the underlying concepts of these approaches are so close that they can be successfully combined to provide a common solution that encompasses all stages of the life cycle.

Large systems are too complex to understand as a whole. Therefore, to understand a system we need to identify distinct concerns that we can address separately. The idea of "separation of concerns" in traditional programming has been known since the 1960s, thanks to E. W. Dijkstra.[1]

Separation of concerns can be applied more widely than in just programming—such as for the understanding of businesses and specification of large systems—to address only those concerns that are of interest, ignoring others that are unrelated.[2] To be of any real value, specifications should be precise and understandable by both software developers and business users. But businesses separate concerns along business, rather than technical, lines. It is natural, therefore, to systematically understand, specify, and implement systems from a set of business, implementation-independent, viewpoints.

The software infrastructure of an organization is typically developed incrementally. Businesses also evolve, and reconfigure themselves, incrementally. In either case, these increments are often best understood and handled as "viewpoints," and thinking in terms of viewpoints removes artificial ordering constraints associated with thinking in terms of increments. Each of these viewpoints describes properties of some relationships in the application domain and may be considered in isolation from other viewpoints (compare Zave and Jackson[3]). In other words, a business viewpoint is a partial specification, used by persons to understand a complicated system "in terms of a few ideas at a time."[4] However, until recently it has been extremely difficult, and prohibitively expensive, to seamlessly add viewpoints. As

0018-8670/96/$5.00 © 1996 IBM

a result, the software infrastructure of most organizations consists of fragmented sets of applications with frequent duplication of functionality, data, . . . (everything!), and general loss of integrity.

The authors have separately developed approaches to dealing with these problems, but at different stages of the development life cycle. "Dynamic supertyping"[5] and, more generally, multiple and dynamic typing, is a specification approach used during analysis. It allows types to be attached to or detached from specific objects dynamically, thereby modeling the manner in which they change their properties, both structural and behavioral, over time. Many have contended that this powerful specification approach is limited in its utility because of implementation difficulties (and, indeed, it is not supported by traditional object-oriented languages).

"Subject-oriented programming"[6,7] is a program composition technology that allows an object-oriented program to be written as separate modules, called "subjects," that are then composed. Each subject defines its own subjective view of the classes it provides and uses. These views are reconciled and combined during composition, as directed by a "composition expression."

On closer inspection, these approaches are highly compatible. This technical note discusses the synthesis of dynamic supertyping and subject-oriented programming to provide uniform support for separation of concerns and graceful evolution across the entire life cycle.[8]

The format of the technical note is as follows. The next section discusses the problem in more detail, motivating the kind of support that is needed to accomplish separation of concerns and graceful evolution in large systems. Next comes a brief discussion of the traditional approaches to the problem and why they are inadequate. The final section describes multiple and dynamic typing and subject-oriented programming in more detail, and shows how subject composition can be used as an implementation vehicle for multiple and dynamic typing.

Important contributions in the fields of object-oriented information modeling[9–12] and subjectivity in object-oriented systems[13,14] have been presented at OOPSLA (Object-Oriented Programming Systems, Languages, and Applications) conferences.

Motivation

Let us start by taking a closer and more precise look at these problems. First, we should explicitly recognize that we are treading in two territories: the territory of precisely formulating a business problem for potential and partial automation (also known as "business specification" or "analysis"), and the territory of providing a system solution that automates the desired part of the business problem (also known as "design" and "implementation").

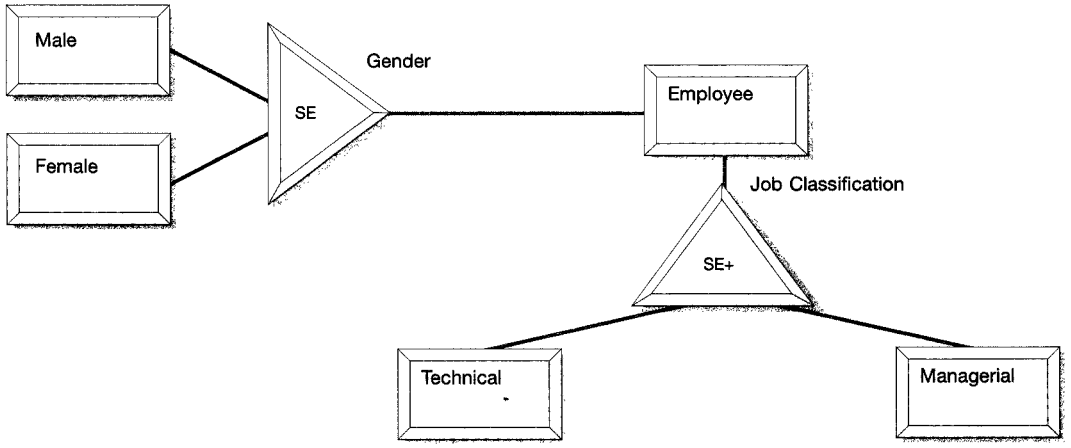
Many existing approaches to analysis have been substantially derived from, and consequently restricted to, preexisting system implementation constructs. This restriction is unwarranted, and has all too frequently led to unnatural, incomplete, and cumbersome business specifications. This need not be the case: a business specification should be built upon concepts and constructs appropriate for capturing the business semantics in the most natural, yet precise and explicit, way. It is then the task of the designer to make any necessary trade-offs to provide a system that meets these business needs in a selected technical environment. The challenge for the designer of technical infrastructural components (such as databases, programming languages, etc.) is to ease this process, and allow systems to be built in which the business specification is not only demonstrably satisfied but remains clearly visible.

In many cases, and especially for addressing the business problems that are used as examples in this paper, concepts used for specification and implementation are substantially the same. We will introduce them at the more abstract, specification level.

Types. Any one thing differs from any other thing. Even identical twins have different properties (names). However, it would be quite difficult to understand things and their relationships if each time we spoke of a thing we had to mention all of its relevant properties. Abstraction helps us with this task—fortunately, some things have common properties that can be studied independently of the individuality of each thing. We say that things with common properties belong to the same type, defined by a predicate that describes the properties held in common.[15,5]

In a sufficiently complicated system, we may be interested in subtypes (and supertypes)—in other words, in conjuncts of the predicates that define types. A new conjunct has to define interesting sub-

Figure 1 A simple example of multiple subtyping hierarchies



The graphical representation used here is described more fully by Kilov and Ross[5] and is also rather close to that of Wieringa et al.[20] (The terminology used in these publications is different; for example, role classes in Wieringa et al. seem to correspond to overlapping subtypes in Kilov and Ross.) Observe that the representation shows both the generic relationship name (e.g., SE for Subtyping Exhaustive or SE+ for Subtyping Exhaustive and Overlapping) and the application-specific relationship name (e.g., Gender). Subtyping Exhaustive means that any instance of the supertype is an instance of exactly one subtype in this subtyping hierarchy. Subtyping Exhaustive and Overlapping means that any instance of the supertype is an instance of at least one subtype in this subtyping hierarchy. Obviously, a linear representation of these relationships (e.g., in the language Z) will also include both the generic and application-specific relationship names.

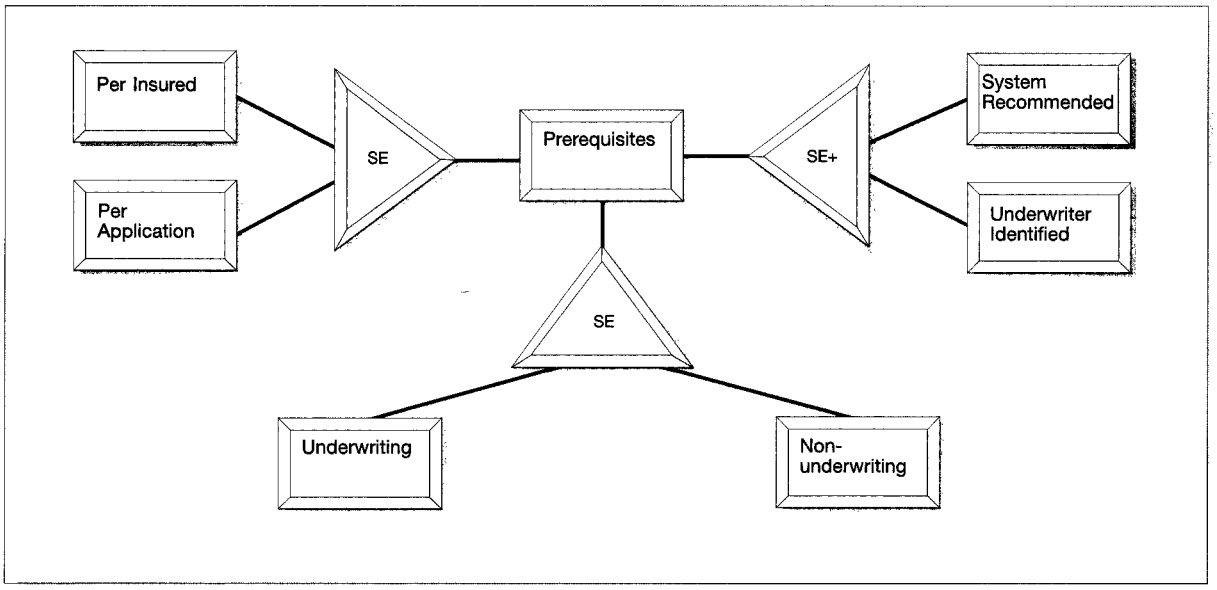
sets of things (belonging to a subclass, or—equivalently—satisfying a subtype). A thing "is of" a subtype (or belongs to a subclass) if it has all the common properties of its supertype (i.e., all the properties used to define the supertype, or, in other words, if it satisfies the predicate of the supertype), and some additional, subtype-specific, properties. This is the invariant of the subtyping relationship[5] that has to be satisfied all the time, as is usual for an invariant.

So far so good, and we leave aside important issues of property overriding, etc. However, the world is more complicated.

Viewpoints. The first complication is that a thing may satisfy several predicates, that is, have several (collections of) properties of interest. Each of these predicates may be considered as a partial specification of the thing. A *person* may have properties related to being a *teacher*, a *student*, a *homeowner*, a *Republican*, a *taxpayer*, a *party* in an insurance agreement, a *sole proprietor*, a *book author*, etc. In other words, a thing may have several types.[15,5] One of the types of a thing is the "most complete"—you can create instances using this type information. This may be considered as a complete specification of the thing. This type is called[15] a "template type." Note that not all of the thing's types are of interest in a particular context; in an insurance environment only a (small) subset of these types is of interest, for example.

To separate different concerns of the enterprise, we usually collect *semantically related* types together. Some of these types may have a common supertype. Several mutually orthogonal collections (*multiple subtyping* hierarchies) may exist for a given supertype. As a simple example, Figure 1 represents the supertype Employee subtyped into a gender subtyping hierarchy (consisting of two mutually exclusive types) and a job classification subtyping hierarchy (consisting of several, not necessarily mutually exclusive, types).

Figure 2 A real-life example of multiple subtyping hierarchies

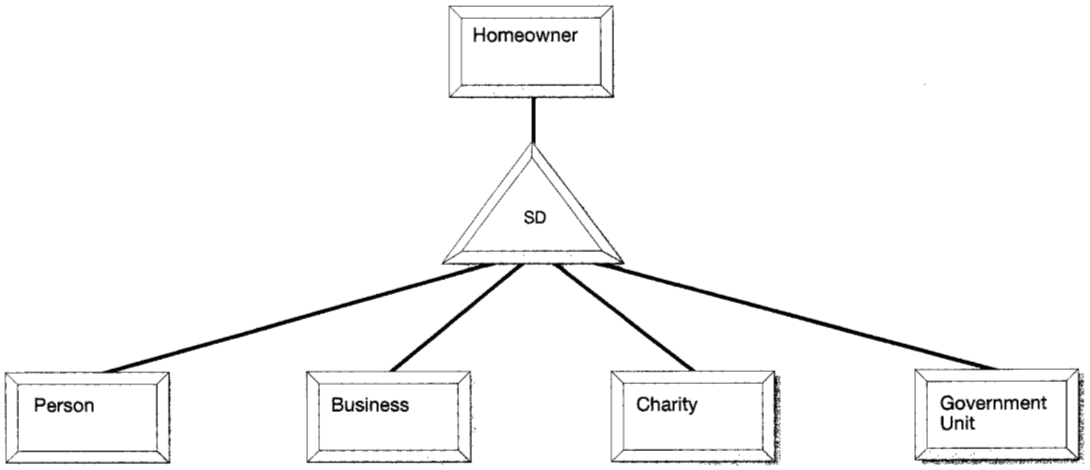


Moreover, we may require that an object that is an instance of the supertype is also an instance of a leaf subtype within each of these collections. In other words, the invariant of mutually orthogonal exhaustive subtyping relationships for the same supertype states that the existence of a supertype instance implies that it will have properties of one of the subtypes in each of its subtyping hierarchies; the properties of an instance have to satisfy the conjunction of the predicates for each of these subtypes[16] (again compare Zave and Jackson[3]). The subtypes need not be static, as we will see later. This invariant defines the behavior in which the instance may participate. To quote the Reference Model for Open Distributed Processing,[15] Clause 13.2.3: "an object may be in a number of contractual contexts simultaneously; the behavior is constrained to the intersection of the behaviors prescribed by each contractual context."

These considerations apply to any relationships in which a thing participates, not just to subtyping. A thing usually participates in several mutually orthogonal elementary relationships of different kinds (e.g., subtyping, composition, dependency, and so on[5,17]), and the thing's invariant (complete specification) is a conjunction of all the "primitive" invariants (partial specifications) defined by the thing's participation in each of these elementary relationships. All of the invariants have to be completely and explicitly specified; implicit assumptions ("unwritten rules") lead to very serious problems in conjoining the invariants which, for technological componentry, were termed "architectural mismatches" by Garlan, et al.[18]

Consider another, more interesting, example. After an application for an insurance policy is accepted by the insurance company (this is done by an underwriter of such a company), and before an insurance contract is issued, some prerequisites are to be satisfied. Certain of these prerequisites may be specified by an insurance system (i.e., they are either required by regulations or otherwise commonly known), whereas certain other prerequisites may be identified only by the underwriter, based on the underwriter's experience. Furthermore, certain prerequisites are, and certain are not, related to underwriting (e.g., the latter may refer to premium payment issues). And finally, an insurance application may include several insured, and therefore certain prerequisites refer to one insured (e.g., beneficiary amendment), whereas certain other ones refer to the application as a whole. Thus, there exist three different subtyping hierarchies for these prerequisites, as shown in Figure 2. (The specification represented

Figure 3 An example of a dynamic subtype



Note: Obviously, the actual specification is more complicated than the one shown in the representation; other objects and relationships between them are not shown. Here SD stands for Subtyping Dynamic.

here is a somewhat edited fragment of a "real-life" insurance application.)

These examples show that in most interesting cases we cannot represent the world as a set of nonintersecting subtype hierarchies (as some OO language authors may want us to do).

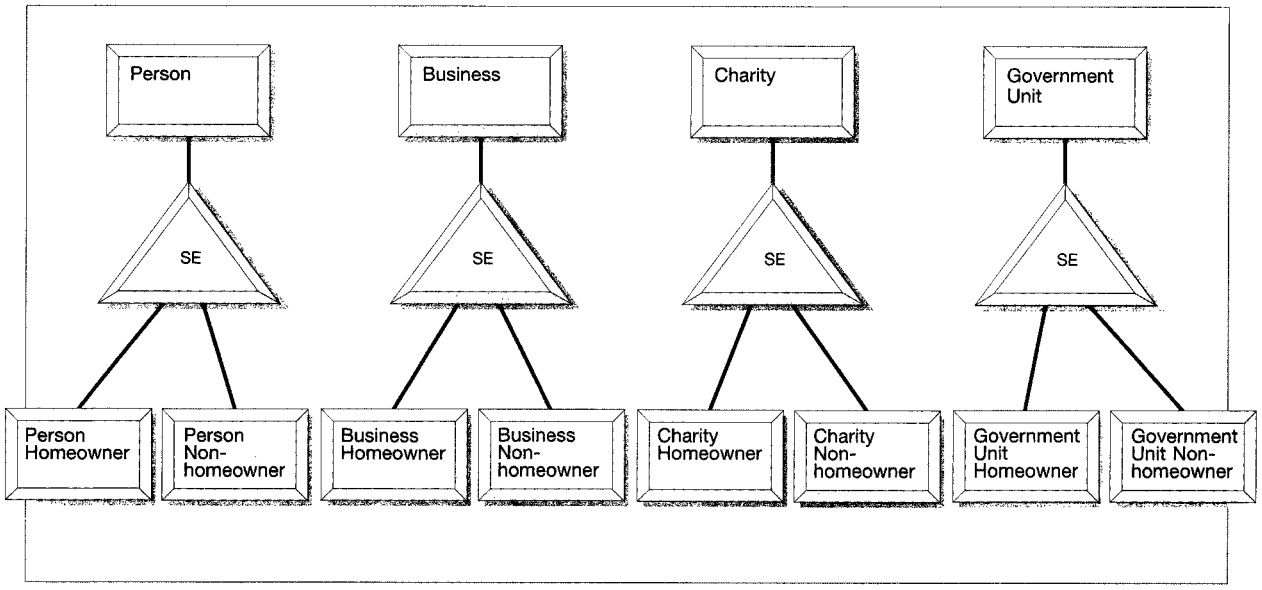
Changes. The second, and perhaps more important, complication is that things change. They acquire some properties, change the values of some other properties, and lose some properties. Therefore the type network (not a hierarchy, see Figure 2) is not fixed. An instance may dynamically change its types (in other words, start and stop satisfying appropriate predicates that define these types[19]). Rigorous specifications of this kind of subtyping relationship have been presented, e.g., in Kilov and Ross[5] and Wieringa et al.[20] Implementation mechanisms for dealing with such dynamic classification exist and have been presented, e.g., in Chambers[21] where predicates are used to define classes in essentially the same manner as in the Reference Model for Open Distributed Processing.[15]

Viewpoints and changes. We can see the third complication by means of an example. Consider the relationship between the types Person and Homeowner. Neither of them is a subtype of the other: a *person* need not be a *homeowner*, and a *homeowner* need not be a *person* (e.g., it may be a *business* or *charity*). However, in a reasonable system we will want to state that a *person* may acquire all the properties[22] of a *homeowner*, thus Person becomes a subtype of Homeowner. This supertyping relationship is not static; it does not satisfy the invariant for subtyping all the time (otherwise all people would be homeowners). In addition, not only *persons* may be *homeowners*; for example, *businesses*, *charities*, and *government units* also may be, and the types Person, Business, Charity, and Government Unit need not have a common (static) supertype. Figure 3 illustrates this relationship.

The identity of an instance does not change when a new supertype is attached or detached; a *person* can become a *homeowner* and therefore acquire properties of a *homeowner*, but the identity of this *person* will remain the same.

We have seen that, on the one hand, types that a particular instance satisfies may be dynamically attached or detached; and on the other hand, we may wish to describe (not necessarily changing) proper-

Figure 4 Modeling dynamic supertyping using static subtypes



ties of the different types of the same instance separately (separation of concerns). The same type may be attached to instances belonging to different, often quite unrelated, types. (Consider attaching the type Inventory Item to different kinds of *inventory items*, such as *computers*, *books*, *phones*, *desks*, *cars*, etc.).

Possible approaches

We need to deal with these issues in a precise and explicit manner, for both analysis (i.e., business specification) and design, and in a mutually consistent and natural way. Therefore, we need both a way to specify the situations described in the previous section, and a way to develop systems that, based on these specifications, will be able to handle these situations. Obviously it is very desirable for our solutions to be simple and elegant. We want to use essentially the same approaches for specification and implementation.

Most traditional approaches are derived from the constructs of legacy object-oriented languages. These languages usually merge the notions of class and type. A strong compiler constraint for a single class hierarchy and static instance properties leads to the imposition of a single, static subtyping hierarchy, which excludes an explicit notion of viewpoint. As a result, most traditional object-oriented approaches force developers to make early, and often artificial and contrived, choices for even simple problems, and thus many issues of large-scale systems, listed earlier, cannot even partially be resolved.

Traditional approaches often force business analysts to use programming constructs (e.g., messages) that are not easily understood by business users. This happens because there is a reluctance to write business specifications using constructs that are not immediately available in a popular programming language. As a result, simple concepts (e.g., that an object may have several types simultaneously) are not always easily and concisely expressed, leading to substantially larger and more complicated and confusing specifications. These considerations lead to skepticism, often expressed in the OO community with respect to important international standards.[15,17]

Traditional solutions to the problems of viewpoints and changes. During analysis, we may define, for example, a Homeowner static subtype for each of Person, Business, Charity, and Government Unit. We can do that, but this repetition of exactly the same static subtyping for several different types is not ad-

visable (at least, due to abstraction and therefore reuse considerations), as shown in Figure 4.

Static multiple inheritance (e.g., "mixins") seems to provide a reasonable solution for implementation. However, it does not solve the problem of property changes. In addition, most multiple inheritance environments specify semantics in quite a loose manner, often based on the intuition of a developer, and using a language-specific object model. Moreover, traditional implementations of multiple inheritance in programming languages do not support dynamic changes to the hierarchy. We can do better than that.

Design patterns. Design patterns[23] are a significant contribution toward providing a rigorous approach for incorporating flexibility into a system. Systems designed with judicious use of these patterns can be extended or changed in a variety of ways by replacement of parts with defined interfaces (that include signature and semantics). Most of these patterns allow substitution of alternative implementations (e.g., "abstract factory" or "strategy"), or dynamic selection of an appropriate implementation (e.g., "state"). The "wrapper" (or "decorator") pattern allows extension of interface, in a sense—each wrapper object can provide some additional behavior, which can be invoked by users who are aware of the wrapper. An important property of design patterns is that they do not require special language-level support, but this also leads to limitations.

The primary limitation of design patterns is that they do not support unanticipated kinds of extension or change. The designer must decide up-front which implementations are to be changeable, extensible, or reconfigurable. This is a valuable exercise, and making anticipated flexibility especially easy to exercise is clearly a hallmark of good design. However, it is impossible to anticipate all needed forms of flexibility, such as the need to add a new viewpoint at the specification level. When one encounters a need for unsupported flexibility, one must modify the original program, ideally introducing a design pattern that provides the new form of flexibility.

The primary goals of our approach are to encourage viewpoint-oriented modularity and ease the extension of systems. Viewpoint-oriented modularity is not addressed in Gamma et al.,[23] and the most closely related pattern for achieving interface extension is "decorator" (also called "wrapper"). This pattern is, rightly, primarily recommended as a solution for extending the interface of objects that are nodes within a hierarchy (a tree of objects in which "components" are almost exclusively accessed by invoking operations that perform a top-down traversal of the tree), since this avoids many limitations inherent in an extension mechanism without language-level support. Wrappers work properly only if all "relevant" calls to the wrapped object go through the wrapper, which requires that callers know of the existence and identity of the wrapper; the extended object has a truly "split identity" that must be carefully handled. Which calls must be passed through the wrapper depends on the nature of the wrapper. In general, it is difficult to ensure that this occurs; the "wrapping of nodes in a hierarchy" applicability of the decorator pattern limits the scope of relevant calls sufficiently to provide a solution for a limited, but important, category of interface extensions.

The relationship between design patterns and our approach will be revisited in more detail after we have presented our approach.

Other approaches. There exist quite a few good ideas for solving these problems (see, for example, the many references in Wieringa et al.[20]). However, they usually are presented as separate analysis-specific approaches[3,4] or, much more often, as implementation-specific approaches.[21] These approaches are often buried within prototype languages and systems or otherwise theoretically interesting treatises that do not immediately or directly address large-scale industrial applications.

Our approaches

We now show how to successfully combine existing—and highly compatible—analysis and implementation approaches earlier presented by the authors elsewhere. In addition, these approaches have a very important property leading to better understanding and therefore to industrial acceptance: they are simple.

Dynamic and multiple typing. In information modeling, the traditional supertyping relationship requires that all properties of a supertype are a subset of the properties of its subtype. We call such a relationship a "static supertyping." Almost invariably, traditional types were specialized using a single subtyping hierarchy.

However, as we have already seen, not all typing relationships are that simple. More often than not, several mutually orthogonal viewpoints exist and each of these viewpoints introduces its own subtyping hi-

erarchy for the same type. This is called "multiple subtyping." Each of these subtyping hierarchies has its invariant, and the conjunction of all these invariants must be satisfiable for the specification to make sense. These internal consistency considerations have to be valid because the invariants of these subtyping hierarchies "are all assertions over the same set of phenomena."[3]

We also need to define and work with those supertyping relationships for which the supertyping invariant is only satisfied *some* of the time. We call such a relationship a "dynamic supertyping." Its invariant is satisfied (for a particular instance) only after this instance acquires the properties of the supertype, but before this instance loses these properties. In other words, the complete supertyping invariant states that *if an instance of a subtype belongs to its supertype then all properties of the supertype are a subset of the properties of its subtype*.[5]

The properties of a subtype include the properties of its static supertype. For those instances of a subtype that are also instances of a dynamic supertype, the properties of a subtype also include the properties of this dynamic supertype. As mentioned above, a type can simultaneously participate in several different static or dynamic supertyping hierarchies.

Dynamic supertyping implies dynamic multiple inheritance. When an instance is created, it has (the properties of) some static type (i.e., it satisfies the predicate of this type). When another (super)type is attached to this instance, the existing predicate is conjoined with the new one, that is, with the predicate that defines the dynamically attached (super)type. As in static multiple inheritance, the conjunction of these predicates should not be false.

In addition to the invariants that define (dynamic) supertypes (and also exhaustive and overlapping subtypes), we can provide precise declarative specifications of generic operations applied to their elements.[5] These invariants imply, in particular, that we can attach a subtype to, or detach a subtype from, an instance of a supertype if and only if the subtypes are overlapping or nonexhaustive. These invariants also imply that we can change the subtype of a thing. We can attach a supertype to, or detach a supertype from, an instance of a subtype if and only if the supertyping is dynamic. Consider, for example, the specification of an operation "attach a supertype to a given instance." It consists of:

- The signature—the type and identity of the existing subtype instance and the supertype to be attached
- The semantics described by
  - The precondition—the subtyping hierarchy is dynamic; the subtype instance exists; the conjunction of the predicate for the new supertype and the predicate that the existing (subtype) instance satisfies is not false
  - The postcondition—the subtype instance acquires the properties of the new supertype

Observe that the semantics of dynamic supertyping constructs have been defined in a very explicit and precise manner. These definitions are declarative and do not prescribe any particular implementation mechanisms (sometimes leading to developer skepticism). Fortunately such mechanisms do exist.

Subject orientation. The subject-oriented programming paradigm[6,7] supports packaging of object-oriented systems into "subjects." Each subject is a (possibly incomplete) object-oriented program that views and represents its domain in its own, subjective way. This subjective model is defined by the collection of classes in the subject. Each class definition contains just those details implemented or used by the subject.

A composition designer can compose subjects to produce a larger subject by writing a "composition expression" made up of "composition rules" that specify how to reconcile the different points of view and how to combine the details from the various constituent subjects appropriately.[24]

A subject is written in an object-oriented source language, and compiled using a "subject compiler" for that language. Compiled subjects are composed by a language-independent "compositor," without modification, recompilation, or even examination of source code. Our current prototype compositor performs composition at link time, but there are no conceptual barriers to performing it dynamically at run time.

It is important to note that a subject does not, usually, define just a single class. It usually defines a whole collection of related classes from a single point of view. For example, a subject might correspond to a requirement or feature, containing those aspects of a number of different classes required for support of that feature. Composition then automatically introduces the entire feature, performing all the

needed class compositions as a unit. Different subjects might contain different levels of detail or be at different levels of abstraction. For example, one subject's view of a class might be as a black box, whereas another's view might include a whole hierarchy of objects that are instances of various other classes also defined in the subject.[25]

One difference between the specification level and the implementation level is that implementation must be concerned with "class" as well as "type." Type specifies what is expected of its instances: what interface they satisfy and what externally meaningful invariants they maintain. On the other hand, in subject orientation, class specifies the structure and code needed to support the type. Thus, when a type is enhanced, such as by dynamic supertyping, the class needs to be enhanced in order to provide the state and code needed to satisfy the enhanced type.

When subjects are composed, the individual classes defined within them are composed according to the specified composition expression. The composed class will satisfy a new type, which depends on the details of the composition. For example, the "merge" composition rule[6,24] is intended to combine classes in such a way that the composed class satisfies the conjunction of the original class's types. Other composition rules such as "override" are also possible. Formal specification of the semantics of subjects and subject composition are topics for future research.

Dynamic composition allows changes to classes during execution. If just a single instance is to acquire new properties, as in dynamic supertyping above, a two-step process is involved:

- The original class is dynamically composed with the class defining the new properties, if it has not already been done. This results in a new, dynamically created class.
- The instance is migrated from its original class to this new class. This migration will involve initialization of any new state, and is supported by subject-oriented run-time support.

This process is performed by the run-time support, and need not show through in this form to the programmer. Details of dynamic composition, especially its application to specific instances, remains a topic for future research.

The combined approach—an illustration. Since we are at the early stages of combining our approaches, we have chosen a simple, yet real, example. Consider the person and homeowner scenario, for which a business specification fragment was provided earlier. Subject-oriented development suggests specification and implementation of two subjects: *person* and *homeowner*.

The *person* subject deals with "intrinsic" details of people, such as name and gender. This subject contains no additional information about people, such as their addresses, family composition, employment, credit cards, or insurance policies. Any such additional information needed by applications would be defined in separate subjects (viewpoints). This fine-grained separation of concerns might seem excessive, leading to highly fragmented systems. It greatly facilitates maintenance and reuse, however, because each fragment deals with one, isolated issue, allowing multiple fragments to be composed in whatever combination is needed in a specific application. Also, the fragmentation can be encapsulated: a subject composed from many fragments, such as a more complete view of people including many common properties, can be used as a unit without regard for its substructure.

The *homeowner* subject deals with the issue of home ownership. It includes at least two types and associated classes: Homeowner and Home. The Homeowner class is not specific in any way to people; it defines just the state and contains just the code needed to represent home ownership and the various operations associated with it, such as paying property taxes and selling the home. The relationship to the actual home owned is an important property of a homeowner, so this subject must also model homes. Once again, however, it only includes basic details of homes needed in this context, such as location, appraised value, and purchase price. Other views of homes, defined in other subjects, might contain architectural plans and other details, history of ownership, and related ghost stories. (This last is included to make a point that is a primary motivation behind subject-oriented programming: it is never possible to predict all future extensions that someone will want to make to a system. Thus, a designer who builds a comprehensive model of a home, intended for use by all applications that have to do with homes, is bound to find that someone, some day, will need state or functions associated with homes that she or he never imagined. Subject-oriented programming allows these to be defined in separate subjects by the developer who needs them, and composed with the original definition. Obviously, dynamic mul-

tiple subtyping hierarchies described earlier solve this problem at the specification level.)

An application in which all people are homeowners and all homeowners are people would use a composition of the *person* and *homeowner* subjects with a composition expression stating that classes *person* and *homeowner* correspond. This would have the effect that whenever code in the *person* subject created a person, or code in the *homeowner* subject created a homeowner, the object actually created by the run-time system would be a synthesis of the two. This correspondence would automatically be inherited by all appropriate subclasses as well. Thus, for example, if Doctor is a subclass of Person, any *doctor* object created would also have homeowner functionality.

Dynamic composition is analogous to schema evolution in object-oriented databases.

In the more realistic application mentioned earlier, specific people and other objects can become and cease to be homeowners. In this case a *person* object is, on creation, just a *person* object as described in the *person* subject. When that person becomes a homeowner, the *person* object must become an instance of a new class defining the synthesis of Person and Homeowner. That class might have been defined statically during the composition that gave rise to the application, or might be produced on demand by dynamic composition. It is interesting to note that dynamic composition is analogous to schema evolution in object-oriented databases: both involve type enhancement in the presence of existing instances, and thus have many of the same conceptual and implementation issues.

Another way of looking at specific instances acquiring new behavior dynamically is as per-instance subject "activation." As in the simplified case mentioned first, the application's composition expression specifies composition of the *person* and *homeowner* subjects, with correspondence between the Person and Homeowner classes. However, it also indicates that the *homeowner* subject is not automatically activated for *person* objects. When a *person* object is created, therefore, it is just as described in the *person* subject, but it has the potential to become a homeowner. This potential is realized by explicitly activating the *homeowner* subject for a particular *person* object, and perhaps deactivating it later.

Design patterns and subjects. Many design patterns are concerned with issues other than flexibility, such as "singleton," "fly-weight," and "interpreter." These have no direct relationship to dynamic supertypes or subjects. Those that deal with flexibility of implementation are essentially complementary to our approach, and can be used to advantage in conjunction with it. However, in many cases similar flexibility can be accomplished directly with subject composition. For example, the "strategy" pattern encapsulates an algorithm applying to a "context" object within a separate strategy object that is attached to the context object. A subject-oriented approach might be to encapsulate it within a separate subject that adds the needed methods and state to the context object itself. Just as strategy objects can be switched dynamically, subjects can be recomposed dynamically. The primary difference between these approaches is that the subject approach allows the interface that the context object presents to the strategy object to be different from the interface it presents to its clients.

Subject-oriented programming supports unanticipated extension, with composition allowing classes to be extended without modification or recompilation of the original program. The composition technology essentially makes every object creation and every operation call a potential open point. Subjects also support interface extension by composition, without requiring callers to deal with unknown collections of wrapper objects, and without split-identity problems.

Flexibility of implementation toward future system extensions is naturally handled using subjects because they offer implementation support for viewpoints. Not only is it natural to address, for example, the billing algorithms for life insurance and automobile insurance from different viewpoints, these other viewpoints can also contain nonbilling aspects of life and automobile insurance. In this way analysis-level conceptual modularity is preserved and visible in system design and implementation.

In summary, many of the design patterns presented in Gamma, et al.[23] are complementary to subjects,

Table 1 Summary

Areas Compared	Multiple Inheritance	Dynamic Supertyping	Subjects
Life-cycle phase	Late	Early	Currently late
Complexity of use	Complex	Simple	Simple
Defined semantics	Loose	Yes	Can be done/in progress
Consistency throughout life cycle	Not applicable	YES	

as is the general idea of design patterns. However, many patterns attempt to provide important kinds of flexibility under the stringent constraint that no special language-level support can be added; as a result, they are limited in their success. Subjects do require significant language-level support, allowing them to support these kinds of flexibility, and especially unanticipated flexibility, more effectively and directly. It is a research issue to reformulate design patterns in subject terms.

Of equal importance, however, subjects allow and encourage a natural system modularity along the lines of business viewpoints, which is not achievable without language-level support.

Research issues

The existence of rigorous and complete business specifications, built upon analysis constructs such as ours, remains of sufficient novelty in its own right that general strategies for exploiting them as part of a rigorous transition to design are a subject for research (see Redberg[26] for an early example, and Hoare[27] for an excellent framework). It is natural, therefore, to build a detailed understanding of the roles of constructs and technologies such as, respectively, dynamic supertyping and subject composition, as part of this broader effort. At the time of writing, we are actively engaged in exploring these broader issues in consulting engagements with customers.

At the time of writing, support for subject-orientation in C++ is being built, with an expectation that it will be available for use in real software development within a matter of months. Prototype-level support for subjects in Smalltalk is also being developed. Nonetheless, many interesting issues in providing complete, high-quality support remain as research issues. Most notable among these is the implementation of dynamic composition.

Dynamic and multiple types, on the one hand, and subjects, on the other, appear to provide a suitable structure, allowing different views or features to be described separately at every level of information management. This two-dimensional structure can be viewed horizontally (all descriptions at a particular level) or vertically (all descriptions of the same feature, from highest-level specification to code) with equal convenience. Details of the contents of "specification subjects," of their composition semantics, and of the relationship between composition of specification subjects and composition of corresponding implementation subjects, are interesting and challenging areas for future research.

Formal, or at least rigorous, specifications of the semantics of viewpoint composition (including overriding and feature interaction) would be of considerable value. This includes a proper vocabulary and structuring techniques for amalgamation of these partial, viewpoint specifications.[4,18]

Design patterns that exploit language-level support offered by subjects is another area for research. Here we foresee an immediate simplification in many existing patterns. A search for patterns in the definition of subject interfaces, for placement of interfaces and implementations in separate subjects, and the definition of generic, reusable subjects in general should also be of considerable value.

A natural, yet precise, way to define and develop systems

We have proposed a combination of dynamic supertypes at the specification level and subjects at the implementation level. Both address the problems of viewpoints and dynamic type changes, and they correspond very well to each other. Subjects provide a natural means of implementing multiple and dynamic typing—specification constructs that some contended could not be implemented by most implementation mechanisms. These approaches thus provide a natural way of understanding, specifying, and developing important fragments of systems.

Other researchers have also examined formalization of dynamic typing,[20,21] and we can benefit from their work as we refine ours. We are not aware, however, of other work that deals with multiple and dynamic typing in the context of the full life cycle with any degree of formality. We also plan to explore the use of subjects directly for specification as well as implementation. Table 1 is our summary.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable and provocative suggestions for improving this paper.

Cited references and notes

1. E. W. Dijkstra, "Stepwise Program Construction," *EWD227* (February 1968). Reprinted in E. W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, Inc., Berlin (1982), pp. 1–14.
2. B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification," *IEEE Transactions on Software Engineering* **20**, No. 10, 760–773 (October 1994).
3. P. Zave and M. Jackson, "Techniques for Partial Specification and Specification of Switching Systems," *VDM '91: Formal Software Development Methods, Lecture Notes in Computer Science* **551**, Springer-Verlag, Inc., Berlin (1991), pp. 511–525.
4. M. Ainsworth, A. H. Cruikshank, L. G. Groves, and P. J. L. Wallis, "Viewpoint Specifications and Z," *Information and Software Technology* **36**, No. 1, 43–51 (January 1994).
5. H. Kilov and J. Ross, *Information Modeling: An Object-Oriented Approach*, Prentice Hall, Englewood Cliffs, NJ (1994).
6. W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)," *OOPSLA '93 Conference Proceedings*, Washington, D.C., ACM Press, New York (1993), pp. 411–428.
7. H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds, "Subject-Oriented Programming: Supporting Decentralized Development of Objects," *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, Santa Clara, CA (July 1994), pp. 337–349.
8. This is a substantially revised and expanded version of W. Harrison, H. Kilow, H. Ossher, and I. Simmonds, "From Dynamic Supertypes to Subjects: A Natural Way to Specify and Develop Systems," *Proceedings of the Workshop on Semantic Integration in Complex Systems: Collective Behavior in Business Rules and Software Transactions*, OOPSLA '95, Austin, TX (October 1995), pp. 17–24.
9. H. Kilov and W. Harvey, "Object-Oriented Reasoning in Information Modeling: Workshop Summary," *OOPSLA '92 Addendum to the Proceedings*, ACM Press, New York (1993), pp. 75–79.
10. H. Kilov, W. Harvey, and H. Mili, "Specification of Behavioral Semantics in Object-Oriented Information Modeling: Workshop Summary," *OOPSLA '93 Addendum to the Proceedings*, ACM Press, New York (1994), pp. 85–89.
11. H. Kilov and W. Harvey, "Precise Behavioral Specifications in Object-Oriented Information Modeling: Workshop Summary," *OOPSLA '94 Addendum to the Proceedings*, ACM Press, New York (1995), pp. 137–142.
12. H. Kilov, W. Harvey, and K. Tyson, "Semantic Integration in Complex Systems: Collective Behavior in Business Rules and Software Transactions: Workshop Summary," *OOPSLA '95 Addendum to the Proceedings*, to be published by ACM Press.
13. W. Harrison, H. Ossher, R. B. Smith, and D. Ungar, "Subjectivity in Object-Oriented Systems: Workshop Summary," *OOPSLA '94 Addendum to the Proceedings*, ACM Press, New York (1995), pp. 131–136.
14. W. Harrison, H. Ossher, and H. Mili, "Subjectivity in Object-Oriented Systems: Workshop Summary," *OOPSLA '95 Addendum to the Proceedings*, to be published by ACM Press.
15. ISO/IEC JTC1/SC21/WG7, *Open Distributed Processing—Reference Model: Part 2: Foundations* (ISO/IEC 10746-2: ITU-T Recommendation X.902, February 1995).
16. This leads us to the subject of feature interaction, which is complex and beyond the scope of this technical note. However, we have two observations. First, it is not clear that a conjunction of two invariants resulting in a "most restrictive invariant" (i.e., an invariant ensuring that each instance must satisfy both sets of constraints) is often really what is needed. Next, there are many times when you cannot conjoin invariants at all because they include property values that are quite simply mutually exclusive alternatives (e.g., a document cannot be formatted at the same time both as a portrait and as a landscape); in this case one property must be explicitly chosen at the expense of the other. However, our approach at least allows us to clearly and explicitly formulate the feature interaction problem. Moreover, as described later, subject-oriented programming allows such details of property composition to be specified explicitly in "composition expressions."
17. ISO/IEC JTC1/SC21, *Information Technology—Open Systems Interconnection—Management Information Systems—Structure of Management Information—Part 7: General Relationship Model*, ISO/IEC 10165-7 (1995).
18. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts," *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 1995, ACM Press, New York (1995), pp. 179–185.
19. Not every state change is important enough to define a new subtype. Other mechanisms, like reference associations in information modeling described by Kilov and Ross,[5] exist to specify not-so-important differences in predicates defining states. As suggested by Cusack,[28] it is inherently a matter of judgment when differences in predicates become important enough to be recognized as differences in subtypes.
20. R. Wieringa, W. de Jonge, and P. Spruit, "Using Dynamic Classes and Role Classes to Model Object Migration," *Theory and Practice of Object Systems* **1**, No. 1, 61–83 (1995).
21. C. Chambers, "Predicate Classes," *Proceedings of ECOOP'93 (European Conference on Object-Oriented Programming), Lecture Notes in Computer Science* **707**, Springer-Verlag, Inc., Berlin (1993), pp. 268–296.
22. We still need to specify what to do in cases of property clashes ("ambiguous overriding").
23. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
24. H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, "Subject-Oriented Composition Rules," *OOPSLA '95 Conference Proceedings*, Austin, TX, ACM Press, New York (1995), pp. 235–250.

25. The relationship between the information modeling notion of composition described by Kilov and Ross,[5] subject-oriented composition, and the (information modeling) composition that defines a collection of related classes included in a subject is a very interesting and important issue. It is a subject for future research.
26. D. Redberg, "Informational and Computational Models for Implementation," *Proceedings of the Workshop on Semantic Integration in Complex Systems: Collective Behavior in Business Rules and Software Transactions*, OOPSLA '95, Austin, TX, (October 1995), pp. 77–79.
27. C. A. R. Hoare, *Mathematical Models for Computing Science*, Oxford University, UK (1994).
28. E. Cusack, "Object-Oriented Modeling in Z for Open Distributed Processing," *Proceedings of the First International Workshop on Open Distributed Processing*, Berlin, 1991, North-Holland Publishing Co., New York (1992).

Accepted for publication January 12, 1996.

William H. Harrison *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: harrisn@watson.ibm.com).* Mr. Harrison is a member of the research staff at the Thomas J. Watson Research Center. He is currently managing a group concerned with technology for the design and implementation of applications that cooperatively provide the implementation of shared objects. He is also performing research on and design of repositories and object-oriented databases, and working with international standards organizations. His background includes work in compilers, program analysis, and operating systems. He was an IEEE Distinguished Lecturer on Software Environments for the years 1987 through 1990. Mr. Harrison received the Master of Philosophy degree in computer science in 1979 and the M.S. degree in 1973 from Syracuse University, and the B.S. degree in electrical engineering from Massachusetts Institute of Technology in 1968. He has been a member of the IEEE since 1973.

Haim Kilov *IBM Research Division, IBM Insurance Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: kilov@watson.ibm.com).* Mr. Kilov has been involved with all stages of information management system specification, design, and development, from initial conception to implementation and release. His approach to information modeling has contributed clarity and understandability to application and enterprise modeling, leading to models that are demonstrably better than "traditional" ones. His approach, widely used in areas such as telecommunications, financial, and document management, is described in *Information Modeling: An Object-Oriented Approach*, recently published by Prentice Hall. He is coeditor of *Object-Oriented Behavioral Specifications* (Kluwer, 1996). Mr. Kilov joined IBM in 1995, introducing and successfully using his business specification concepts and constructs in customer engagements. He is a member of, and active contributor to, several national standardization technical committees and a member of the editorial board of *Computer Standards and Interfaces*. His research interests are information modeling, business specification, and formal methods.

Harold L. Ossher *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: ossher@watson.ibm.com).* Dr. Ossher joined the Thomas J. Watson Research Center as a research staff member in 1986. He worked on the RPDE[3] (Research Program Development Environment 3) framework for building integrated, extensible environments and was one of the chief designers of OOTIS (Object-Oriented Tool Integration Services), a joint project with IBM Toronto. He is one of the originators of "subject-oriented programming." He is currently leading a team to develop support for subject-oriented programming in C++ and Smalltalk, and is engaged in a variety of other activities involving the exploration, use, and transfer of subject-oriented programming. Dr. Ossher received the B.Sc. and M.Sc. degrees in computer science from Rhodes University, Grahamstown, South Africa, and the Ph.D. degree from Stanford University. He is an associate editor of *TAPOS (Theory and Practice of Object Systems)*, published by John Wiley & Sons, Inc.

Ian Simmonds *IBM Research Division, IBM Insurance Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: isimmond@watson.ibm.com).* Mr. Simmonds received his B.A. and M.A. degrees in mathematics from the University of Cambridge in 1987 and 1990, respectively. From 1987 to 1993 he worked for General Electric Company Software, London, and Société Française de Genie Logicie, Paris, on the architecture of software engineering environments including the EAST Environment. In particular, he participated in the international (ISO) standardization of the Portable Common Tool Environment (PCTE) and its application for the development of software engineering tools and environments. Mr. Simmonds joined the IBM Software Solutions Division in Toronto in 1993 to do similar work related to IBM's implementation of the PCTE standard and the TeamConnection repository. Since joining the Insurance Research Center in February 1995, Mr. Simmonds has been studying the application of object-oriented techniques to the development of business systems for the insurance industry. His research interests include the rigorous specification of business requirements and the use of these as a basis for the systematic development of business systems, together with the transfer of these techniques for use by IBM's customers and consultants.

Reprint Order No. G321-5604.