Subject-Oriented Programming: Supporting Decentralized Development of Objects

Harold Ossher and William Harrison
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
harrisn@watson, 8-863-7631; ossher@watson, 8-863-7975

Frank Budinsky and Ian Simmonds
AIX-CASE Architecture and Products, IBM Canada Ltd. Laboratory
1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada
frankb@torolab6, 8-778-3968; simmonds@torolab6, 8-778-3868

Overview

Object-oriented applications execute by performing *operations* on objects. In suites of inter-operating or integrated applications, such as are essential in an enterprise setting, the same objects must be manipulated by multiple applications. Each application generally requires its own particular operations to be supported. For example, a Shipping application might require *Truck* objects to support a “What is your capacity?” operation, whereas a Transportation application might require them to support a “Set your route” operation.

Typically, the responsibility for implementing all operations for a class of objects falls on a *class owner*. This leads to the need for centralized management of, or at least extensive and costly negotiation between, application developers and class owners. There are two aspects to this negotiation:

1. Reaching agreement as to the meaning of classes and operations.
2. Securing a commitment from class owners to implement applications' needed operations *in a timely fashion*.

In the example, the owner of the *Truck* class would be responsible for implementing “What is your capacity?,” “Set your route,” and any other operations required by any applications that use *Truck* objects, with agreed-upon meaning and in accordance with the applications' development schedules.

Subject-oriented programming is an enhancement of object-oriented programming that allows decentralized class definition. An application developer who needs new operations associated with classes can implement them him/herself, not by editing existing code for the classes, but as a separate collection of class definitions called a *subject*. Multiple subjects can be *composed* to yield a complete suite of applications; class definitions within the subjects will be combined so as to satisfy the needs of all the applications in the suite. A simple example is shown in Figure 1. Neither source code access nor recompilation are required to perform this composition, allowing extension and composition of object-code-only applications.

Without eliminating the advantages of encapsulation, this approach eliminates the need for class ownership, and hence for the second, more serious kind of negotiation noted above. An application developer can write all the code needed for the application, irrespective of which classes are involved, without levying development requirements on others. The cost of this flexibility is a small run-time overhead on operation calls.

Subject-oriented programming gets its name from the fact that each subject defines a *subjective view* of objects: the particular operations and internal data that that subject requires the objects to have. When different subjects are composed, the correspondence between the classes and operations in the various

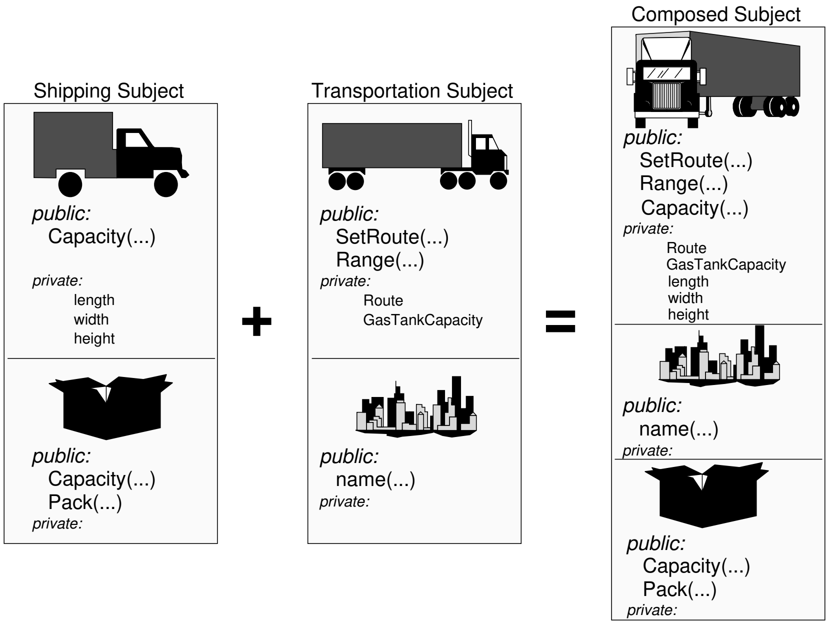

Figure 1. Application suite formed by composition of subjects

subjective views must be specified (or, when possible, be deduced by the system). Corresponding class definitions are then combined by the system.

This ability to reconcile different views of objects at composition time means that it is not essential to have advance agreement about the meaning of classes and operations, addressing the first aspect of negotiation mentioned above. The degree of disparity that can be handled effectively is still a matter of investigation. Advance negotiation about shared concepts remains valuable, but is considerably reduced in both quantity and importance.

Subject-oriented programming supports decentralization in time as well as in space. The developers of an application can program extensions to it as separate subjects to be composed with the base application, perhaps in multiple configurations. This leads to *requirement-based* development: the code that implements a new requirement is built as a coherent subject rather than being interleaved amongst other application code in a manner that makes it difficult to identify and maintain.

Customers would benefit directly from subject-oriented programming, in addition to benefitting indirectly from improved application development and maintenance. A software product bought as a composition of subjects could be reconfigured according to local requirements or preferences, could be extended by composition with other subjects (from the same supplier, different suppliers or written in-house), and could serve as a library of reusable parts. All configuration and composition could be done without access to source code. We believe that the technology has the potential to allow customers to compose independently-written applications into integrated suites of applications in ways not preplanned by the authors of the applications, and with much less effort or programming skill than integration endeavors currently require. Further research is required to realize the potential.

An overview of the issues involved in subject-oriented programming can be found in [4]. An understanding of the importance of decentralized development of objects was earlier obtained working with the RPDE[3] environment developed at Watson [2, 9], and the Object-Oriented Tool Integration Services (OOTIS) designed jointly by Watson and Toronto [3, 5, 6]. Though they do not support subjectivity directly, RPDE[3] and OOTIS validate many of the concepts and implementation details of subject-oriented programming.

We are building a prototype of the system support required to permit subject-oriented programming in C++. We are also actively looking for suitable development groups using or supporting C++ to whom to transfer the technology.

1 The Problem of Centralization

1.1 Tension between Application Developers and Class Owners

One of the essential aspects of object-oriented programming is *encapsulation:* objects can be manipulated only through operations. Each operation is implemented by means of a piece of code called a *method*. The methods are written in terms of the internal details of the object, such as the *instance variables* that contain the object's data. Clients can call the operations, but have no knowledge of or direct access to the internal details.

In most object-oriented languages, similar objects are grouped into *classes*. For example, a *Truck* class consists of all truck objects, which are called its *instances*. A single *class definition* specifies the internal details and methods for all instances of the class.

Encapsulation typically leads to a development methodology where each class has an *owner*. The class owner determines the internal details of the class, and is responsible for implementing all operations defined for the class. This has the advantage that the internal details are isolated, and the owner is free to change them provided the operations are still correctly supported.

The property of *polymorphism* allows an application to be written to use particular operations, rather than specific classes of objects. This provides important flexibility. For example, if a shipping application that determines how to package goods into shipments is written to use *Truck* objects specifically, extending it later to work with trains or ships is likely to require detailed changes. On the other hand, if it is written to use operations such as “What is your capacity?,” it will work equally well with all forms of transportation that support these operations.

Proper use of polymorphism in writing object-oriented applications therefore facilitates *plug-compatibility:* different objects can be interchanged freely, provided they all support the needed operations.

When a new application is being written, it is likely to require some new operations to be supported by the objects it manipulates. Implementation of these operations might require the objects to have additional state. If the benefits of plug-compatibility are to be realized fully, a large number of classes of objects might be involved. Each new application thus imposes requirements on many class owners to implement these new operations. This creates tension between application developers, who levy requirements, and class owners, who are responsible for satisfying them.

It is often thought that the mechanism of *subclassing* can be applied to resolve each application's need to add state and operations to those defined by class owners. By forming a new subclass, the application developer becomes the owner of the enhancements. Unfortunately, when applications cooperate over the same objects, the actual object being manipulated is of its class at time of creation, and not of the particular subclasses defined by each of the several application developers. This holds true whether the sharing takes place through persistent storage in a database, or through the shipment of objects and object references over communication channels.

1.2 Progressing from Isolated Applications to Application Suites

Object-oriented programming is still a relatively new technology. It has been most heavily used in the development of isolated applications or tightly-integrated systems written as units. In these settings, the tension between application developers and class owners is not prominent. It does surface, however, and developers often find themselves spending a good deal of time asking colleagues to extend their classes to meet new requirements. Since this all happens within a single development group, its impact is limited.

More recently, object-oriented programming has entered the world of inter-operating and integrated applications. Supporting an enterprise requires a vast and constantly evolving suite of applications that must work together effectively. The success of any software technology in the enterprise arena depends on its ability to support such application suites.

Object-oriented applications work together by manipulating shared objects. For example, the shipping application mentioned above and a transportation application that schedules trucks and determines their routes would both manipulate *Truck* objects. By sharing the same truck objects, they could coordinate packaging and actual transportation. Each application in a suite generally needs its own particular operations to be supported, and therefore imposes its own set of requirements on class owners.

Application suites, and the individual applications within them, must evolve constantly to improve service to the customer and to meet new customer requirements. At each step in the evolution, additional operations might be needed, leading to further requirements on class owners.

In the world of evolving suites of integrated applications, therefore, the requirements levied by application developers on class owners can easily become overwhelming.

1.3 Applications as Methods

We have written thus far of applications manipulating objects, such as the shipping application manipulating *Truck* objects. In many cases, the best way to write such applications is as methods on the objects they manipulate. For example, rather than writing the shipping application as a separate program that calls operations like “What is your capacity?” on *Truck* objects, one implements shipping operations like “Load” on the *Truck* objects, thereby implementing the shipping application itself. The application is now really a collection of methods provided by a collection of objects.

This approach is often followed in the development of isolated object-oriented applications. When it is used in the context of application suites, it presents a problem: only the class owners can write methods for their objects. This makes the class owners responsible for practically all application development that involves their classes, further increasing the tension between application developers and class owners, and requiring coordination among the owners of all the classes that participate in an application.

1.4 Negotiation and Management

The tension between application developers and class owners, described above, must be resolved by negotiation, probably arbitrated by centralized management.

There are two aspects to the negotiation needed in extending classes to support a new application:

1. Reaching agreement as to the meaning of classes and operations.
2. Securing a commitment from class owners to implement applications' needed operations *in a timely fashion*. This includes operations used by the applications and methods that constitute the very essence of the applications.

When classes are shared by multiple applications, the amount of negotiation can be extensive and costly. Application developers are concerned about short time-to-market for their applications. Class owners must juggle the requirements of all applications that are to use their objects, while maintaining integrity and correctness. Determining and enforcing priorities requires an overall vision and centralized control of all the applications and classes involved.

This negotiation and centralized management is a serious bottleneck. It tends to lead to frustration, development delays and lack of responsiveness to customer requirements. The effect can be so serious as

```
// Shipping Subject

class Truck {
public:
      float capacity() {...};
      // Other public operations
private:
      float length, width, height;
      // Other internal data and operations
};

class Box {
public:
      float capacity() {...};
      void pack(ItemList& items) {...};
// ...
};
```

```
// Transportation Subject

class Truck {
public:
      void setRoute(Route& r) {...};
      float range() {...};
      // Other public operations
private:
      Route& plannedRoute;
      float gasTankCapacity, mpg;
      // Other internal data and operations
};

class City {
public:
      char* name() {...};
// ...
};
```

Figure 2. C++ for separate Shipping and Transportation subjects

to cause abandonment of integration in favor of isolated applications, which do not inter-operate well but which can be developed and evolved more rapidly.

Subject-oriented programming addresses these problems by eliminating the notion of class ownership, and thereby decentralizing the development of classes.

2 Overview of Subject-Oriented Programming

A subject-oriented application, or suite of applications, is constructed by *composing* a collection of *subjects*. Each subject is itself an object-oriented program, though often an incomplete one. It consists of a collection of classes (arranged in a *classification hierarchy*, as is standard, though this is incidental). It is written in a standard object-oriented language.

The classes in a subject define a *subjective* view of a collection of objects, appropriate for a particular purpose. For example, consider the shipping and transportation systems mentioned above. The shipping application defines a particular view of trucks: objects into which items can be packed, with dimensions, capacity and similar properties. The application also defines other objects appropriate to shipping, such as boxes. All these definitions make up a "Shipping Subject." The transportation application, on the other hand, defines a rather different view of trucks: as objects that can travel according to a set route, with properties such as range that can be travelled on a single tank of gas. It also defines other objects appropriate to transportation, such as cities. These definitions make up a "Transportation Subject."

Illustrative skeletons of the two example subjects are shown in Figure 2. C++ syntax is used in this example, but subject-oriented programming applies to object-oriented languages in general. When these two subjects are composed, the details of corresponding classes are, in effect, combined to yield the composed subject shown in Figure 3. This combined subject clearly satisfies the needs of both applications.

Of course, each combined class definition, such as that for class *Truck* in Figure 3, could be written directly, rather than be derived in the course of composing two subjects. This leads to the problem of negotiation between application writers and class owners, however. The combined class definition is one piece of code, which in any reasonable approach must be programmed and maintained by one person, yet which contains details from both applications. The advantage of the subject-oriented approach is that the two subjects can be written and maintained separately, and then later be composed.

```
class Truck {
public:
      float capacity() {...};
      void setRoute(Route& r) {...};
      float range() {...};
      // Other public operations
private:
      float length, width, height;
      Route& plannedRoute;
      float gasTankCapacity, mpg;
      // Other internal data and operations
};

class Box {
public:
      float capacity() {...};
      void pack(ItemList& items) {...};
// ...
};

class City {
public:
      char* name() {...};
// ...
};
```

Figure 3. C++ for Shipping and Transportation subjects composed

Composition is performed on binary code.[1] The source-code combined subject shown in Figure 3 is therefore not produced as such. Instead, each subject is compiled separately to produce a *binary subject*. The binary subject consists of a *label* providing information about it, and binary code produced by the compiler. The subject-oriented *compositor* uses information in the labels to tie the subjects together. It does not examine or modify the individual subjects' binary code.

A binary subject label consists of three parts:

1. A *schema,* defining the classes provided and/or used by the subject, and their classification hierarchy and instance variables from the point of view of this subject.
2. *Interfaces,* defining the operations (generic functions) provided and/or used by the subject, and giving their signatures.
3. *Structure* specifications, defining details such as if and how this subject is composed from subsidiary subjects, and how operations are mapped to entry points (of methods) in the subject's binary code for the various classes defined in the schema. Method code itself does not appear anywhere in the label.

A subject developer can select the details to expose in the interface and schema parts of the label, thereby controlling what details are totally hidden within the subject and what details can be shared with other subjects. Figure 4 illustrates the labels for the Shipping and Transportation subjects.

This ability to perform composition of binary subjects means that subjects can be sold object-code-only and yet still be composable. The cost is a small run-time overhead on operation calls. Implementers of

1 The term “binary code” is used instead of the more usual “object code” to avoid confusion with the other meaning of “object.”

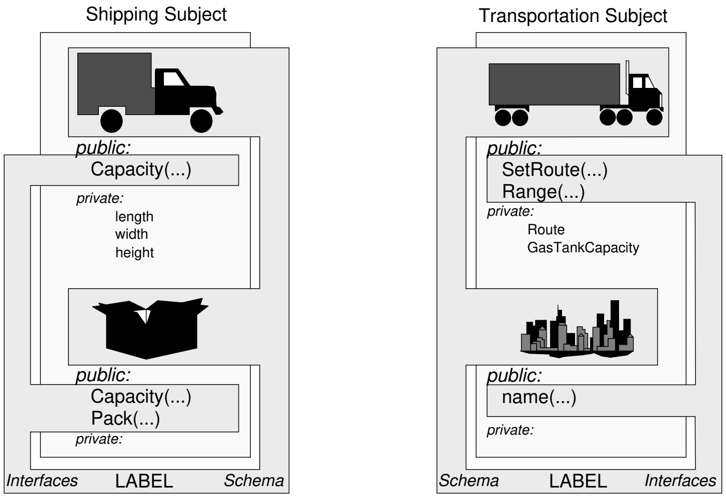

Figure 4. Labels for Shipping and Transportation subjects

run-time support for subject-oriented programming can reduce this overhead in various ways, especially for operation calls which, for a particular composition, involve only one subject.

The developer performing a composition of subjects can provide a *composition rule* specifying details of the composition desired. One option is distribution. The rule can specify whether all subjects are to be combined into a single executable, that executes as a single process, or whether some subjects are to run in separate processes, perhaps on different machines. The same binary subjects can be used without change in local or distributed compositions. This characteristic of moving the packaging decisions out of the method code is crucial to the reuse of class definitions in different compositions.

Unlike the simple example earlier, the subjects being composed might define a number of different classes, operations and instance variables, some of which might be shared by the subjects and some of which might not. The composition rule must specify how classes, operations and instance variables in different subjects correspond. For example, a vehicle-maintenance application might be written in terms of a *Vehicle* class. When it is composed with the transportation subject mentioned above, the composition rule would state that *Truck* in the transportation subject corresponds to *Vehicle* in the maintenance subject. This simple rule would give vehicle-maintenance functionality to *Truck* objects.

Composition rules can specify correspondence in simple, generic terms (such as the default “name-based matching”), or in detail. Detailed specifications of operation correspondence can include transformation of parameters. Detailed specifications of instance variable correspondence identify explicitly which instance variables are shared by multiple subjects. An interesting area of ongoing research is exploring high-level rules that are not fully explicit, with the compositor deducing the details (probably with user assistance obtained interactively).

Composition rules can also specify a variety of ways to combine corresponding classes. For example, if corresponding classes in two subjects both provide methods for corresponding operations, various options are available, including:

- Performing both methods whenever the operation is called. This is the default, because it effectively combines the functionality of the two subjects. There are various sub-options to do with how return values are handled.
- Performing a specific one of the two methods. This allows one subject to override the other.

- Regarding the composition as illegal.

Rules of this sort can be specified once for all classes, or on a class-by-class or even operation-by-operation basis.

Support for composition performed just before program execution begins has already been designed, and most key aspects of it have been prototyped. We are also exploring dynamic composition, in which actually running subjects can be composed without interrupting their execution. This would allow running applications to be extended or to begin cooperating with other applications as the user requires. We do not see major obstacles to extending the support to handle dynamic composition.

Further details of subject-oriented programming are given elsewhere [1, 4, 8].

3 Subject-Oriented Software Development

From the point of view of software engineering, a subject is a new kind of *module*. It is universally acknowledged that software systems should be built as collections of separate modules, and that the more changes that can be made by adding or reconfiguring modules, rather than by modifying code, the better the modularization.

When a subject-oriented approach is used in software development, many activities that previously required modification of existing programs can be performed by composition instead. This section briefly outlines some important examples.

3.1 Unplanned Extension

Any successful application must evolve during its lifetime to meet new customer needs. We firmly believe that no developer, however astute, can anticipate at original design time the nature of all future extensions. A good designer will anticipate some kinds of future extensions, and build in "open points" that allow those extensions to be added gracefully when they become necessary. The time will come, however, when an unanticipated extension is needed, for which no open point was built in. With conventional technology, such an extension can usually be made only by editing the internals of the system.

A subject-oriented application, on the other hand, can always be extended by composition down to the level of object granularity in the base application. As long as the application objects needing extension were initially designed as objects exposed in the subject label, the extension can be written as a separate subject, and then be applied by composition. This is possible because subject-oriented programming effectively makes all key points open: object creation and deletion, and operation call and return. The composition can be performed without access to the application's source code. Knowledge of the source code might be helpful to the programmer writing the extension, but it is potentially possible to write extensions with reference only to subject labels and documentation.

Application objects that have been "hidden" as data structures or as objects not exposed in the label cannot be flexed as easily. Instead, replacement behavior must be attached to the larger exposed objects of which they are parts. We call the degree to which application objects are exposed in a subject label the *granularity* of the application design.

While the emphasis here has been on unplanned extension, subject composition is convenient for pre-planned extension as well.

3.2 Unplanned Composition

There is a growing collection of applications available, and with it growing frustration on the part of users that the applications don't work together, even when the manner in which they should work together seems obvious at the end-user level. Current integration efforts involve a great deal of detailed programming and either modification of the applications, which requires agreement and commitment from the developers, or wrappering techniques, which seldom produce very satisfactory integration from the customer's point of view.

Subject-oriented applications hold the promise of being capable of composition in ways not anticipated by the original authors. For this to work, the authors must certainly have planned for the applications to be composed, but need not have anticipated what kinds of other applications would be involved nor what open points would be needed for inter-subject interaction, nor have reached up-front agreement with one another about application details. Differences between applications, inevitable in the world of separately-written applications without careful, centralized control, can potentially be resolved by the composition rule. The developer performing the composition thus studies the application labels and documentation and writes a composition rule to tie the applications together appropriately. The composition rule can be complex, though the hope is that it will often be simple. The rule can refer to custom-written code to perform needed transformations. There is potential for generating at least some of the composition rule and associated code automatically from analysis of subject labels; we are exploring this issue in conjunction with another research project, the Global Desktop [10].

The degree of disparity between subjects that can be handled automatically, or at all, is still a matter of investigation. Many relatively simple cases can certainly be handled, and these alone represent a significant advance over traditional technology.

3.3 Requirement-Based Development

Software is written to meet requirements, and modified to meet new requirements. Typically, the code written to meet a particular cluster of requirements is spread out all over the system, interleaved with code written to meet other requirements. Maintaining the correspondence between requirements and code is a classic problem in software engineering.

Subject-oriented programming supports a development methodology in which a subject is designed and coded for each cluster of requirements. The subjects are then composed to yield a system that meets all the requirements. The structure of the system as a composition of subjects therefore directly reflects the clusters of requirements. This is possible because, as illustrated by the simple composition example in section 2, composition of separate subjects can lead to interleaved code.

This approach is valuable during initial application development, and even more so during maintenance. As new requirements are identified, new subjects can be written to support them. The new subjects are composed with the original application, as described in the section on unplanned extension.

3.4 Feature-Based Configuration Management

Complex applications commonly provide multiple *features*. Different configurations of the application for different customers might require inclusion of different features, or choices from among alternative implementations of a feature.

Applications structured according to requirements as described above tend to have subjects corresponding to features. Different configurations can be obtained easily as different compositions of the feature subjects.

An important feature of an application is its user interface. Much work has been done in the past on separation of the user interface from the underlying application. Subject-oriented programming provides a natural way to do this: the user interface is a subject (or collection of subjects) that is composed with the subjects providing the processing. Different user interfaces can be provided, and the desired one used in the composition.

As such systems evolve, new features need to be added. This can be done by means of extension subjects, as described above, maintaining the correspondence between features and subjects as the system grows.

4 Encapsulation

The elimination of the concept of class ownership and the attendant decentralization of class development raises the concern that encapsulation has been broken. Encapsulation isolates developers from internal implementation decisions made by other developers. The important advantage of this is that each developer is free to change these decisions without impact on other developers. If encapsulation is broken, a change by one developer can lead to a cascade of required changes by others.

In the subject-oriented context, the concern arises from the ability of one developer to extend a class that another developer first wrote. Suppose that the first developer makes changes to the internal class details, which in standard object-oriented programming would be hidden from others. Would not the second developer now have to change the extension?

In truth, subject-oriented programming provides tighter encapsulation than standard object-oriented programming. Within limits, encapsulation is controlled by subject labels, which include all exposed details of the subject. When a developer produces a subject, s/he determines what to expose in the label and what not. Suitable security mechanisms could allow portions of labels to be exposed selectively (e.g. to holders of developer's licences as distinct from user's licences).

It may well be the case that the developers of both subjects are the same group, and that packaging as subjects has been adopted for reasons of requirement-based or feature-based design. No real loss of encapsulation has been incurred by this packaging choice, even if all possible details are exposed in the subjects' labels.

Material that is not included in a subject label is totally private to that subject. Even definitions of the same class in other subjects cannot access it. This is tighter encapsulation than is permitted by standard object-oriented languages, in which an entire class is the encapsulation unit.

Material that is included in the label can be used in compositions, and can be used by programmers writing extensions. If the label changes, the composition and extension code might have to be changed also. The label serves the role of an interface, and this corresponds to the normal situation that clients of an interface must change when the interface changes.

The internal details of a class exposed in the label of one subject can thus be used in other subjects, but only within definitions of the same class in the other subjects. Cross-class accesses (e.g. whether or not code in one class can access instance variables of another class) are still subject to the rules of the underlying language.

Being too restrictive in what is exposed in a subject label will limit the possible compositions in which that subject can participate. Being too open will limit the changes that can be made to the subject without affecting other subjects. Finding the optimal point on the spectrum is an instance of the challenging task of interface design. Subject-oriented programming provides the flexibility needed to move along this spectrum. Traditional object-oriented programming supports a few standard points on the spectrum, with the result that many extensions cannot be made without full access to source code.

5 Benefits to Customers

Customers would benefit from the use of subject-oriented programming within IBM in a number of ways.

Customers are greatly affected by the time it takes IBM to produce the software they need, and by our responsiveness to new requirements they identify. Any software development support or approach that reduces our development time or response time has immediate benefit to customers.

An application built as a composition of subjects would have additional value to the customer beyond its basic use. The customer could configure or reconfigure it according to local requirements or preferences, by forming his/her own compositions. The customer could extend it him/herself by composing it with other subjects. The other subjects might be obtained off the shelf from us or from other vendors, or be custom-written by or for the customer. The application could also be used as a library of reusable components for development of additional applications; it should be especially useful in this role for applications in the same domain. All configuration, composition and reuse could be done without access to source code.

We believe that the technology has the potential to allow customers to compose independently-written applications into integrated suites of applications in ways not preplanned by the authors of the applications, and with much less effort or programming skill than integration endeavors currently require. Further research is required to realize this potential.

Supporting subject-oriented software development in IBM's Computer-Aided Software Engineering (CASE) and programming environment offerings would give customers of these offerings the added advantage of being able to use the subject-oriented methodology for their own software development, passing the benefits on to their customers.

We believe that these would all be significant benefits to our customers, and would give IBM a significant competitive advantage in the software market.

6 Subject-Oriented Support for C++

As part of a cooperative effort between T. J. Watson Research and the IBM Canada Laboratory in Toronto, direct tool support for development of subjects in C++ is being prototyped. This support is embodied in two tools, called the “C++ Subjectifier” and the “Binary Subject Compositor”, as shown in Figure 5.

The C++ Subjectifier performs the language-dependent processing needed to create a binary subject (label + binary code) that can later be composed. It performs a series of source-to-source transformations on the C++ source programs to intercept:

- object creations
- object destructions
- instance variable references
- virtual function calls

These intercepts allow later binding of class definitions than is possible with the usual compilations of these constructs. The resulting source is compiled to obtain the binary code for the subject.

In addition to the binary code, the C++ Subjectifier incorporates selected information from the class hierarchy into the subject label:

- The selection of information to appear in the schema and interface portions of the label is under control of the developer. Not all of the C++ classes need appear as objects in the schema portion of the label. In addition, the developer selects which instance variables are to be exposed.
- The interfaces are abstracted from the classes included in the schema, but not all operations need to be exposed in the interfaces.
- The structure portion of the label is determined mechanically, using C++ inheritance rules.

The result is a binary, language independent, subject which can be combined with other subjects built with the same or different languages.

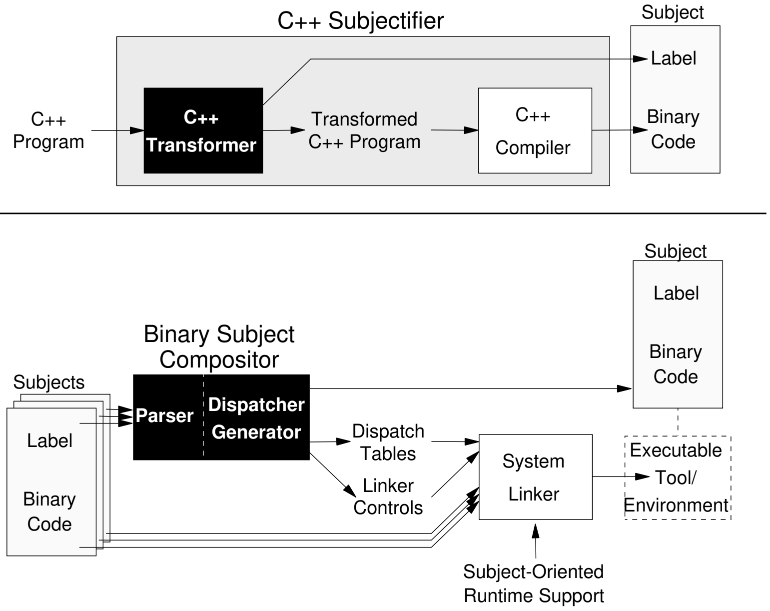

Figure 5. C++ Subjectifier and Binary Subject Compositor structure

The Binary Subject Compositor processes a collection of subjects. The labels from each are parsed and integrated using the composition rules. The resulting information is processed by the Research Dispatcher Generator [7] to create code and tables to control the dispatch process. These are combined with the binary code from the subjects and with run-time support routines to form a compound, executable subject.

7 Summary

Subject-oriented programming is an enhancement of object-oriented programming that permits decentralization of class definitions. Application developers can define their own views of shared classes of objects without centralized control. This reduces the need for costly negotiation between, and centralized management of, object-oriented application developers and class providers.

The technology is currently being prototyped and transferred. We believe that it has the potential to provide IBM customers with significant benefits: improved response to new customer requirements, increased ability for customers to tailor, extend and combine our software products to suit their needs, and CASE offerings that support the development of software with these advantages.

References

1. Frank Budinsky, William Harrison, Matthew Kaplan, Erica Lan, Harold Ossher and Ian Simmonds, “Subject-oriented programming for the C++ developer.” Research Report, IBM Thomas J. Watson Research Center, in preparation.
2. William Harrison, “RPDE^3: A framework for integrating tool fragments.” *IEEE Software*, vol. 4, pp. 46-56, November 1987.
3. William Harrison, Mansour Kavianpour and Harold Ossher, “Integrating coarse-grained and fine-grained tool integration.” In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE '92)*, pp. 23-35, July 1992.
4. William Harrison and Harold Ossher, “Subject-oriented programming (a critique of pure objects).” In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, (Washington, D.C.), ACM, September 1993.

5. William Harrison, Harold Ossher and Mansour Kavianpour, “Extending PCTE for transparent fine-grained object access in OOTIS.” In *Proceedings of the PCTE '93 Conference*, (Paris), November 1993.
6. William Harrison, Harold Ossher, Mansour Kavianpour and Eric Wong. “PCTE SDSs for modeling OOTIS control integration.” In *Proceedings of the PCTE '93 Conference*, (Paris), November 1993.
7. Matthew Kaplan, “A retargetable dispatcher generator for development by composition.” In *Proceedings of the 7th IBM Object-Oriented Technology Conference*, July 1994.
8. Harold Ossher and William Harrison, “Combination of inheritance hierarchies.” In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA '92)*, pp. 25-40, October 1992.
9. Harold Ossher and William Harrison, “Support for change in $RPDE^3$.” In *Proceedings of the Fourth Symposium on Software Development Environments (SDE4)*, ACM SIGSOFT, pp. 218-228, December 1990.
10. Tien Huynh, Charanjit Jutla, Andy Lowry, Robert Strom and Daniel Yellin, “The global desktop: A graphical composition environment for local and distributed applications.” Research Report RC 19342, IBM Thomas J. Watson Research Center, Janaury, 1994.