Designing Reusable Classes

Ralph E. Johnson
Brian Foote

Department of Computer Science
University of Illinois, Urbana-Champaign
Journal of Object-Oriented Programming
June/July 1988

August 26, 1991

[Editado por la cátedra de POO - 2001]

Abstract

Object-oriented programming is as much a different way of designing programs as it is a different way of designing programming languages. This paper describes what it is like to design systems in Smalltalk. In particular, since a major motivation for object-oriented programming is software reuse, this paper describes how classes are developed so that they will be reusable.

1. Introduction

Object-oriented programming is often touted as promoting software reuse[Fis87].Languages like Smalltalk are claimed to reduce not only development time but also the cost of maintenance, simplifying the creation of new systems and of new versions of old systems. This is true, but object-oriented programming is not a panacea. Program components must be designed for reusability. There is a set of design techniques that makes object-oriented software more reusable. Many of these techniques are widely used within the object-oriented programming community, but few of them have ever been written down. This article describes and organizes these techniques. It uses Smalltalk vocabulary, but most of what it says applies to other object-oriented languages. It concentrates on single inheritance and says little about multiple inheritance.

The first section of the paper describes the attributes of object-oriented languages that promote reusable software. Data abstraction encourages modular systems that are easy to understand. Inheritance allows subclasses to share methods defined in superclasses, and permits programming-by-difference. Polymorphism makes it easier for a given component to work correctly in a wide range of new contexts. The combination of these features makes the design of object-oriented systems quite different from that of conventional systems.

The middle section of the paper discusses frameworks, toolkits, and the software lifecycle. A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes. During the early phases of a system's history, a framework makes heavier use of inheritance and the software engineer must know how a component is implemented in order to reuse it. As a framework becomes more refined, it leads to "black box" components that can be reused without knowing their implementations.

The last section of the paper gives a set of design rules for developing better, more reusable object-oriented programs. These rules can help the designer create standard protocols, abstract classes, and object-oriented frameworks.

As with any design task, designing reusable classes requires judgement, experience, and taste. However, this paper has organized many of the design techniques that are widely used within the object-oriented programming community so that new designers can acquire those skills more quickly.

2. Object-Oriented Programming

An object is similar to a value in an abstract data type: it encapsulates both data and operations on that data. Thus, object-oriented languages provide modularity and information-hiding, like other modern languages. Too much is made of the similarities of data abstraction languages and object-oriented languages. In our opinion, all modern languages should provide data abstraction facilities. It is therefore more important to see how object-oriented

languages differ from conventional data abstraction languages.

There are two features that distinguish an object-oriented language from one based on abstract data types: polymorphism caused by late-binding of procedure calls and inheritance. Polymorphism leads to the idea of using the set of messages that an object understands as its type, and inheritance leads to the idea of an abstract class. Both are important.

2.1 Polymorphism

Operations are performed on objects by “sending them a message". Messages in a language like Smalltalk should not be confused with those in distributed operating systems. Smalltalk messages are just late-bound procedure calls. A message send is implemented by finding the correct method (procedure) in the class of the receiver (the object to which the message is sent), and invoking that method. Thus, the expression a + b will invoke different methods depending upon the class of the object in variable a .

Message sending causes polymorphism. For example, a method that sums the elements in an array will work correctly whenever all the elements of the array understand the addition message, no matter what classes they are in. In fact, if array elements are accessed by sending messages to the array, the procedure will work whenever it is given an argument that understands the array accessing messages.

2.2 Protocol

[Nota: preste atención a la definición de protocolo e interface en esta sección, que difiere a la vista en la práctica 1]

The specification of an object is given by its protocol, i.e. the set of messages that can be sent to it. The type of the arguments of each message is also important, but “type" should be thought of as protocol and not as class. For a discussion of types in Smalltalk, see [Joh86]. Objects with identical protocol are interchangeable. Thus, the interface between objects is defined by the protocols that they expect each other to understand. If several classes define the same protocol then objects in those classes are “plug compatible". Complex objects can be created by interconnecting objects from a set of compatible components. This gives rise to a style of programming called building tool kits, of which more will be said later.

Although protocols are important for defining interfaces within programs, they are even more important as a way for programmers to communicate with other. Shared protocols create a shared vocabulary that programmers can reuse to ease the learning of new classes. Just as mathematicians reuse the names of arithmetic operations for matrices, polynomials, and other algebraic objects, so Smalltalk programmers use the same names for operations on many kinds of classes. Thus, a programmer will know the meaning of many of the components of a new program the first time it is read.

Standard protocols are given their power by polymorphism. Languages with no polymorphism at all, like Pascal, discourage giving different procedures the same name, since they then cannot be used in the same program. Thus, many Pascal programs use a large number of slightly different names, such as MatrixPlus, ComplexPlus, PolynomialPlus, etc. Languages that use generics and overloading to provide a limited form of polymorphism can benefit from the use of standard protocols, but the benefits do not seem large enough to have forced wide use of them. In Smalltalk, however, there are a wide number of well-known standard protocols, and all experienced programmers use them heavily.

Standard protocols form an important part of the Smalltalk culture. A new programmer finds it much easier to read Smalltalk programs once standard protocols are learned, and they form a standard vocabulary that ensures that new components will be compatible with old.

2.3 Inheritance

Most object-oriented programming languages have another feature that differentiates them from other data abstraction languages; class inheritance. Each class has a superclass from which it inherits operations and internal structure. A class can add to the operations it inherits or can redefine inherited operations. However, classes cannot delete inherited operations.

Class inheritance has a number of advantages. One is that it promotes code reuse, since code shared by several classes can be placed in their common superclass, and new classes can start off having code available by being given a superclass with that code. Class inheritance supports a style of programming called programming-by-difference, where the programmer defines a new class by picking a closely related class as its superclass and describing the differences between the old and new classes. Class inheritance also provides a way to organize and classify classes, since classes with the same superclass are usually closely related.

One of the important benefits of class inheritance is that it encourages the development of the standard protocols that were earlier described as making polymorphism so useful. All the subclasses of a particular class inherit its operations, so they all share its protocol. Thus, when a programmer uses programming-by-difference to rapidly build classes, a family of classes with a standard protocol results automatically. Thus, class inheritance not only supports software reuse by programming-by-difference, it also helps develop standard protocols.

Another benefit of class inheritance is that it allows extensions to be made to a class while leaving the original code intact. Thus, changes made by one programmer are less likely to affect another. The code in the subclass defines the differences between the classes, acting as a history of the editing operations.

Not all object-oriented programming languages allow protocol and inheritance to be separated. Languages like C++[Str86] that use classes as types require that an object have the right superclass to receive a message, not just that it have the right protocol. Of course, languages with multiple inheritance can solve this problem by associating a superclass with every protocol.

2.4 Abstract Classes

Standard protocols are often represented by abstract classes [GR83]. An abstract class never has instances, only its subclasses have instances. The roots of class hierarchies are usually abstract classes, while the leaf classes are never abstract. Abstract classes usually do not define any instance variables. However, they define methods in terms of a few undefined methods that must be implemented by the subclasses. For example, class `Collection` is abstract, and defines a number of methods, including `select:`, `collect:`, and `inject:into:`, in terms of an iteration method, `do:`. Subclasses of `Collection`, such as `Array`, `Set`, and `Dictionary`, define `do:` and are then able to use the methods that they inherited from `Collection`. Thus, abstract classes can be used much like program skeletons, where the user fills in certain options and reuses the code in the skeleton.

A class that is not abstract is concrete. In general, it is better to inherit from an abstract class than from a concrete class. A concrete class must provide a definition for its data representation, and some subclasses will need a different representation. Since an abstract class does not have to provide a data representation, future subclasses can use any representation without fear of conflicting with the one that they inherited.

Creating new abstract classes is very important, but is not easy. It is always easier to reuse a nicely packaged abstraction than to invent it. However, the process of programming in Smalltalk makes it easier to discover the important abstractions. A Smalltalk programmer always tries to create new classes by making them be subclasses of existing ones, since this is less work than creating a class from scratch. This often results in a class hierarchy whose top-most class is concrete. The top of a large class hierarchy should almost always be an abstract class, so the experienced programmer will then try to reorganize the class hierarchy and find the abstract class hidden in the concrete class. The result will be a new abstract class that can be reused many times in the future.

3. Toolkits and Frameworks

One of the most important kinds of reuse is reuse of designs. A collection of abstract classes can be used to express an abstract design. The design of a program is usually described in terms of the program's components and the way they interact. For example, a compiler can be described as consisting of a lexer, a parser, a symbol table, a type checker, and a code generator.

An object-oriented abstract design, also called a framework, consists of an abstract class for each major component. The interfaces between the components of the design are defined in terms of sets of messages. There will usually be a library of subclasses that can be used as components in the design. A compiler framework would probably have some concrete symbol table classes and some classes that generate code for common machines. In theory, code generators could be mixed with many different parsers. However, parsers and lexers would be closely matched. Thus, some parts of a framework place more constraints on each other than others.

Frameworks are useful for reusing more than just mainline application code. They can also describe the abstract designs of library components. The ability of frameworks to allow the extension of existing library components is one of their principal strengths.

Frameworks are more than well written class libraries. A good example of a set of library utility class definitions is the Smalltalk Collection hierarchy. These classes provide ways of manipulating collections of objects such as `Arrays`, `Dictionaries`, `Sets`, `Bags`, and the like. In a sense, these tools correspond to the sorts of tools one might find in the support library for a conventional programming system. Each component in such a library can serve as a discrete, stand-alone, context independent part of a solution to a large range of different problems. Such components are largely application independent.

A framework, on the other hand, is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific.

Frameworks can be built on top of other frameworks by sharing abstract classes. FOIBLE is a framework for building “device programming" systems in Smalltalk[Eri87]. It lets the user edit a picture consisting of a collection of interconnected devices. These devices have computational meaning, so editing the picture is a form of programming. FOIBLE uses the MVC framework to implement the editor, but adds Tools and Foibles to implement the semantics of the picture and the visual representation of components. Thus, FOIBLE is built on top of MVC.

Frameworks provide a way of reusing code that is resistant to more conventional reuse attempts. Application independent components can be reused rather easily, but reusing the edifice that ties the components together is usually possible only by copying and editing it. Unlike skeleton programs, which is the conventional approach to reusing this kind of code, frameworks make it easy to ensure the consistency of all components under changing requirements.

Since frameworks provide for reuse at the largest granularity, it is no surprise that a good framework is more difficult to design than a good abstract class. Frameworks tend to be application specific, to interlock with other frameworks by sharing abstract classes, and to contain some abstract classes that are specialized for the framework. Designing a framework requires a great deal of experience and experimentation, just like designing its component abstract classes.

3.1 White-box vs. Black-box Frameworks

One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.

A framework's application specific behavior is usually defined by adding methods to subclasses of one or more of its classes. Each method added to a subclass must abide by the internal conventions of its superclasses. We call these white-box frameworks because their implementation must be understood to use them.

A good example is the MVC `Controller` class, which maps user actions into messages to the application. When the mouse moves into the region of a controller, it is sent the `startUp` message, which causes the controller to be sent the `controlInitialize`, `controlLoop`, and `controlTerminate` messages, in that order. The behavior of a controller when it is selected and deselected is changed by redefining `controlInitialize` and `controlTerminate`. The default behavior of controlLoop is to repeatedly send the controller the `controlActivity` message until the mouse moves out of the region of the controller. Thus, the reaction of a controller to mouse movement, mouse button clicks, and keyboard events is determined by the definition of the `controlActivity`.

The major problem with such a framework is that every application requires the creation of many new subclasses. While most of these new subclasses are simple, their number can make it difficult for a new programmer to learn the design of an application well enough to change it.

A second problem is that a white-box framework can be difficult to learn to use, since learning to use it is the same as learning how it is constructed.

Another way to customize a framework is to supply it with a set of components that provide the application specific behavior. Each of these components will be required to understand a particular protocol. All or most of the components might be provided by a component library. The interface between components can be defined by protocol, so the user needs to understand only the external interface of the components. Thus, this kind of a framework is called a black-box framework.

There is a set of black-box components of MVC called the pluggable views. These components were designed with the realization that the majority of MVC classes that were created were controllers with a customized menu. The pluggable views let controllers take the menus as parameters, thus greatly reducing the need to create new controller classes. Most of the programming tools in the latest versions of Smalltalk-80, such as the browser, file tool, and debugger, use pluggable views and do not require any new user interface classes. The method that invokes a tool will create instances of the various components, send messages to them to customize them for the tool, and connect them together.

Black-box frameworks like the pluggable views are easier to learn to use than white-box frameworks, but are less flexible. Pluggable views are usually sufficient to describe user interfaces that display only text, but the user who wants a more graphical user interface will have to use the original MVC framework. Fortunately, pluggable views fit into the MVC framework well, so the user only has to create components to handle the graphical aspects of the interface.

A framework becomes more reusable as the relationship between its parts is defined in terms of a protocol, instead of using inheritance. In fact, as the design of a system becomes better understood, black-box relationships should replace white-box ones. Black-box relationships are an ideal towards which a system should evolve.

4. Lifecycle

The lifecycle of a Smalltalk application is not necessarily different from that of other programs developed using rapid prototyping. However, the lifecycle of classes differs markedly from that of program components in conventional languages, since classes may be reused in many applications.

Classes usually start out being application dependent. It is always worth-while to examine a nearly-complete project to

see if new abstract classes and frameworks can be discovered. They can probably be reused in later projects, and their presence in the current project will make later enhancements much easier. Thus, creating abstract classes and frameworks is both a way of scavenging components for later reuse and a way of cleaning up a design. The final class hierarchy is a description of how the system ought to have been designed, though it may bear little relation to the original design.

There are many ways that classes can be reorganized. Big, complex classes can be split into several smaller classes. A common superclass can be found for a set of related classes. Concrete superclasses can be made abstract. An white-box framework can be converted into a black-box framework. All these changes make classes more reusable and maintainable.

Every class hierarchy offers the possibility of becoming a framework. Since a white-box framework is just a set of conventions for overriding methods, there is no fine line between a white-box framework and a simple class hierarchy. In its simplest form, a white-box framework is a program skeleton, and the subclasses are the additions to the skeleton.

Ideally, each framework will evolve into a black-box framework. However, it is often hard to tell in advance how an white-box framework will evolve into a black-box framework, and many frameworks will not complete the journey from skeleton to black-box frameworks during their lifetimes.

White-box inheritance frameworks should be seen as a natural stage in the evolution of a system. Because they are a middle ground between a particular application and an abstract design, white-box inheritance frameworks provide an indispensable path along which applications may evolve. A white-box framework will sometime be a step in the evolution of a loose collection of methods into a discrete set of components. At other times, a white-box framework will be a finished product. A useful design strategy is to begin with a white-box approach. White-box frameworks, as a result of their internal informality, are usually relatively easy to design. As the system evolves, the designer can then see if additional internal structure emerges.

Finding new abstractions is difficult. In general, it seems that an abstraction is usually discovered by generalizing from a number of concrete examples. An experienced designer can sometimes invent an abstract class from scratch, but only after having implemented concrete versions for several other projects.

This is probably unavoidable. Humans think better about concrete examples then about abstractions. We can think well about abstractions such as integers or parsers only because we have a lot of experience with them. However, new abstractions are very important. A designer should be very happy whenever a good abstraction is found, no matter how it was found.

5. Design methodology

The product of an object-oriented design is a list of class definitions. Each class has a list of operations that it defines and a list of objects with which its instances communicate. In addition, each operation has a list of other operations that it will invoke. A design is complete when every object that is referenced has been defined and every operation is defined. The design process incrementally extends an incomplete design until it is complete.

A class should represent a well-defined abstraction, not just a bundle of methods And variable definitions. Human judgement is needed to decide when and how a class hierarchy is to be reorganized. Nevertheless, the following rules will frequently point out the need for a reorganization and suggest how it is to be accomplished.

5.1 Rules for Finding Standard Protocols

It is very important that the design process result in standard protocols. In other words, many of the classes should have nearly identical external interfaces and there should be sets of operations that many classes implement.

Standard protocols are developed by choosing names carefully. The need for standard protocols is one reason why it takes a long time to become an expert Smalltalk programmer. Many of the more important protocols are described in the Blue Book[GR83], but just as many are not documented anywhere except in the source code. Thus, the only way to learn these protocols is by experience.

There are a number of rules of thumb that will help develop standard protocols. A programmer practicing these rules is more likely to keep from giving different names to the same operation in different classes. These rules help minimize the number of different names and maximize the number of names shared by a set of classes.

Rule 1 *Recursion introduction.*

If one class communicates with a number of other classes, its interface to each of them should be the same. If an operation X is implemented by performing a similar operation on the components of the receiver, then that operation should also be named X. Even if the name of the operation has to be changed to add more arguments, it makes sense to make the names similar so that readers of the program will note the connection. The result is that a method for a message sends that same message to other objects. If the other objects are in the same class as the sender then the method is recursive. Even if no real recursion exists, the method appears recursive, so we call this rule recursion introduction.

Recursion introduction can help decide the class in which an operation should be a method. Consider the problem of converting a parse tree into machine language. In addition to an object representing the parse tree, there will be an object representing the final machine language procedure. The \generate code" message could be sent to either object. However, the best design is to implement the generate code message in the parse tree class, since a parse tree will consist of many parse nodes, and a parse node will generate machine code for itself by recursively asking its subtrees to generate code for themselves.

Rule 2 *Eliminate case analysis.*

It is almost always a mistake to explicitly check the class of an object. Code of the form

```
anObject class == ThisClass
      ifTrue: [anObject foo]
      ifFalse: [anObject fee]
```

should be replaced with a message to the object whose class is being checked. Methods will have to be created in the various possible classes of the object to respond to the message, and each method will contain one of the cases that is being replaced.

Eliminating case analysis is more difficult when the cases are accessing instance variables, but it is no less important. If instance variables are being accessed then self will need to be an argument to the message and more messages may need to be defined to access the instance variables.

Rule 3 *Reduce the number of arguments.*

Messages with half a dozen or more arguments are hard to read. Except for instance creation messages, a message with this many arguments should be redefined. When a message has a smaller number of arguments it is more likely to be similar to some other message, thus increasing the possibility of giving them the same name.

The number of arguments can be reduced by breaking a message into several smaller messages or by creating a new class that represents a group of arguments. Frequently there will be several kinds of messages that pass the same set of objects around. This set of objects is essentially a new object, and the design can be changed to reflect that fact by replacing the set of objects with an object that contains them.

Rule 4 *Reduce the size of methods.*

Well-designed Smalltalk methods are almost always small. It is easier to subclass a class with small methods, since its behavior can be changed by redefining a few small methods instead of modifying a few large methods. A thirty line method is large and probably needs to be broken into pieces. Often a method in a superclass is split when a subclass is made. Most of the inherited method is correct, but one part needs to be changed. Instead of rewriting the entire method, it is split into pieces and the one piece that has changed is redefined. This change leaves the superclass even easier to subclass.

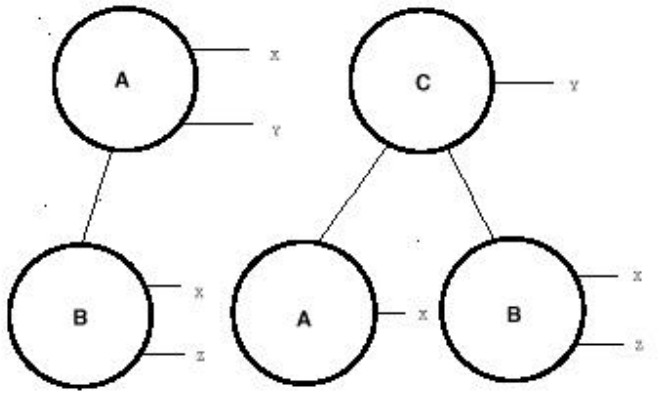

Figure 1

These design rules are all related, since eliminating cases reduces the size of methods, breaking a method into pieces is likely to reduce the number of arguments that any one method needs, and reducing the number of arguments is likely to create more methods with the same name.

5.2 Rules for Finding Abstract Classes

Rule 5 *Class hierarchies should be deep and narrow.*

A well developed class hierarchy should be several layers deep. A class hierarchy consisting of one superclass and 27 subclasses is much too shallow. A shallow class hierarchy is evidence that change is needed, but does not give any idea how to make that change.

An obvious way to make a new superclass is to find some sibling classes that implement the same message and try to migrate the method to a common superclass. Of course, the classes are likely to provide different methods for the message, but it is often possible to break a method into pieces and place some of the pieces in the superclass and some in the subclasses. For example, displaying a view consists of displaying its border, displaying its subviews, and displaying its contents. The last part must be implemented by each subclass, but the others are inherited from View.

Rule 6 *The top of the class hierarchy should be abstract.*

Inheritance for generalization or code sharing usually indicates the need for a new subclass. If class B overrides a method x that it inherits from class A then it might be better to move the methods in A that B does inherit to C, a new superclass of A, as shown in Figure 1. C will probably be

abstract. B can then become a subclass of C, and will not have to redefine any methods. Instance variables or methods defined in A that are used by B should be moved to C.

Rule 7 *Minimize accesses to variables.*

Since one of the main differences between abstract and concrete classes is the presence of data representation, classes can be made more abstract by eliminating their dependence on their data representation. One way this can be done is to access all variables by sending messages. The data representation can be changed by redefining the accessing messages.

Rule 8 *Subclasses should be specializations.*

There are several different ways that inheritance can be used[HO87]. Specialization is the ideal that is usually described, where the elements of the subclass can all be thought of as elements of the superclass. Usually the subclass will not redefine any of the inherited methods, but will add new methods. For example, a two dimensional array is a subclass of Array in which all the elements are arrays. It might have new messages that use two indexes, instead of just one.

An important special case of specialization is making concrete classes. Since an abstract class is not executable, making a subclass of an abstract class is different from making a subclass of a concrete class. The abstract class requires its subclasses to define certain operations, so making a concrete class is similar to filling in the blanks in a program template. An abstract class may define some operations in an overly general fashion, and the subclass may have to redefine them. For example, the size operation in class Collection is implemented by iterating over the collection and counting its elements. Most subclasses of Collection have an instance variable that contains the size, so size is redefined in those subclasses to return that instance variable. There are a couple of ways that a designer can tell whether a subclass is a specialization of a superclass. An abstract definition is that anywhere the superclass is used, the subclass can be used. Thus, a subclass has a superset of the behavior of its superclass.

5.3 Rules for Finding Frameworks

Large classes are frequently broken into several small classes as they grow, leading to a new framework. A collection of small classes can be easier to learn and will almost always be easier to reuse than a single large class. A collection of class hierarchies provides the ability to mix and match components while a single class hierarchy does not. Thus, breaking a compiler into a parsing phase and a code generation phase permits a new language to be implemented by building only a new parser, and a new machine to be supported by building only a new code generator.

Rule 9 *Split large classes.*

A class is supposed to represent an abstraction. If a class has 50 to 100 methods then it must represent a complicated abstraction. It is likely that such a class is not well defined and probably consists of several different abstractions. Large classes should be viewed with suspicion and held to be guilty of poor design until proven innocent.

Rule 10 *Factor implementation differences into subcomponents.*

If some subclasses implement a method one way and others implement it another way then the implementation of that method is independent of the superclass. It is likely that it is not an integral part of the subclasses and should be split off into the class of a component. Multiple inheritance can also be used to solve this problem. However, if an algorithm or set of methods is independent of the rest of the class then it is cleaner to encapsulate it in a separate component.

Rule 11 *Separate methods that do not communicate.*

A class should almost always be split when half of its methods access half of its instance variables and the other half of its methods access the other half of its variables. This sometimes occurs when there are several different ways to view objects in the class.

For example, a complex graphical object may cache its image as a bitmap, but the image is derived from the complex structure of the object, which consists of a number of simple graphical objects. When the object is asked to display itself, it displays its cached image if it is valid. If the image is not valid, the object recalculates the image and displays it. However, the graphical object can also be considered a collection of (graphical) objects that can be added or removed. Changing the collection invalidates the image.

This graphical object could be implemented as a subclass of bitmapped images, or it could be a subclass of Collection. A system with multiple inheritance might make both be superclasses. However, it is best to make both the bitmap and the collection of graphical objects be components, since each of them could be implemented in a number of different ways, and none of those ways are critical to the implementation of the graphical object. Separating the bitmap class will make it easier to port the graphical object to a system with different graphics primitives, and separating the collection class will make it easier to make the graphical object be efficient even when very large.

Rule 12 *Send messages to components instead of to self.*

An inheritance-based framework can be converted into a component-based framework black box structure by replacing overridden methods by message sends to components. Examples of such frameworks in conventional systems are

sorting routines that take procedural parameters. Programs should be factored in this fashion whenever possible. Reducing the coupling between framework components so that the framework works with any plug-compatible object increases its cohesion and generality.

Rule 13 *Reduce implicit parameter passing.*

Sometimes it is hard to split a class into two parts because methods that should go in different classes access the same instance variable. This can happen because the instance variable is being treated as a global variable when it should be passed as a parameter between methods. Changing the methods to explicitly pass the parameter will make it easier to split the class later.

6. Conclusion

A number of factors account for the high reusability of object-oriented components. Polymorphism increases the likelihood that a given component will be usable in new contexts. Inheritance promotes the emergence of standard protocols, and allows existing components to be customized. Inheritance also promotes the emergence of abstract classes. Frameworks allow a collection of objects to serve as a template solution to a class of problems. Using frameworks, algorithms and control code, as well as individual components, can be reused.

Object-oriented techniques offer us an alternative to writing the same programs over and over again. We may instead take the time to craft, hone, and perfect general components, with the knowledge that our programming environment gives us the ability to re-exploit them. If designing such components is a time consuming experience, it is also one that is aesthetically satisfying. If my alternatives are to roll the same rock up the same hill every day, or leave a legacy of polished, tested general components as the result of my toil, I know what my choice will be.

References

[AC84] Inc. Apple Computer. Lisa Toolkit 3.0. Apple Computer, Cupertino, CA, 1984.

[Ale87] James H. Alexander. Paneless panes for Smalltalk windows. In OOPSLA'87, 1987.

[BC86a] Kent Beck and Ward Cunningham. The Literate Program Browser. Technical Report, Tektronix, 1986.

[BC86b] Kent Beck and Ward Cunningham. Using the Diagramming Debugger. Technical Report, Tektronix, 1986.

[Boo86] Grady Booch. Software Engineering with Ada. Benjamin/Cummings, Menlo Park, CA, 1986.

[Boo87] Grady Booch. Software Components with Ada: Structures, Tools, and Subsystems. Benjamin/Cummings, Menlo Park, CA, 1987.

[CB86] Ward Cunningham and Kent Beck. ScrollController Explained: An Example of Literate Programming in Smalltalk. Technical Report,
Tektronix, 1986.

[Dij82] Edsger W. Dijkstra. How Do We Tell Truths that Might Hurt?, pages. Springer-Verlag, New York, NY, 1982.

[Eri87] Stewart Ericson. FOIBLE: A Framework for Object-Oriented Interactive Box and Line Environments. Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[Fis87] Gerhard Fischer. Cognitive view of reuse and redesign. IEEE Software, 4(4):60{72, 1987.

[Foo88] Brian Foote. Designing to Facilitate Change with Object-Oriented Frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[GB81] Ira P. Goldstein and Daniel G. Bobrow. PIE: An Experimental Personal Information Environment. Technical Report CSL-81-4, Xerox Palo Alto Research Center, 1981.

[Gol84] Adele Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, Massachusetts, 1984.

[GR83] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, Massachusetts,1983.

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programs. IEEE Software, to appear,1987.

[Joh86] Ralph E. Johnson. Type-checking Smalltalk. In Proceedings of OOPSLA `86, pages 315{321, November 1986. printed as SIGPLAN Notices, 21(11).

[Lis87] Barbara Liskov Keynote Address. Data Abstraction and Hierarchy. In OOPSLA `87 Addendum to the Proceedings, pp. 17-34 October 1987 (printed as SIGPLAN Notices 23(5)).

[LS80] Ware Meyers. Interview with Wilma Osborne. IEEE Software 5(3): 104-105, 1988

[OBHS86] Tim O'Shea, Kent Beck, Dan Halbert, and Kurt J. Schmucker. Panel on: the learnability of object-oriented programming systems. In Proceedings of OOPSLA `86, , pages 502{504, November 1986. Printed as SIGPLAN Notices, 21(11).

[Roc86] Roxanna Rochat. In Search of Good Smalltalk Programming Style. Technical Report CR-86-19, Tektronix, 1986.

[Sch86] Kurt J. Schmucker. Object-Oriented Programming for the Macintosh. Hayden Book Company, 1986.

[Sei87] Ed Seidewitz. Object-oriented programming in smalltalk and Ada. In Proceedings of OOPSLA `87, pages 202{213, December 1987.Printed as SIGPLAN Notices, 22(12).

[Smi87] Randall B. Smith. Experience with the alternate reality kit: an example of the tension between literalism and magic. In Proceedings of CHI 87, pages 61{68, April 1987.

[Str86] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley Publishing Co., Reading, MA, 1986.