Designing Reuseable 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

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 reuseable.

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 reuseability. There is a set of design techniques that makes object-oriented software more reuseable. Many of these techniques are widely used within the object-oriented programming community, but few of them have ever been written down. This

[0]Authors' address: Department of Computer Science, 1304 West Springfield Ave., Urbana IL 61801
[0]Telephone: (217) 244-0093, (217) 333-3411
[0]e-mail: johnson@cs.uiuc.edu, foote@cs.uiuc.edu

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 second 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"[1]. 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.

Polymorphism is more powerful than the use of generic procedures and packages in Ada[Sei87]. A generic can be instantiated by macro substitution, and the resulting procedure or package is not at all polymorphic. On the other hand, a Smalltalk object can access an array in which each element is of a different class. As long as all the elements understand the same set of messages, the object can interact with the elements of the array without regard to their class. This is particularly useful in windowing systems, where the array could hold a list of windows to be displayed. This could be simulated in Ada using variant records and explicitely checking the tag of each window before displaying it, thus ensuring that the correct display procedure was called. However, this kind of programming is dangerous, because it is easy to forget a case. It leads to software that is hard to reuse, since minor modifications are likely to add more cases. Since the tag checks will be widely distributed through the program, adding a case will require wide-spread modifications before the program can be reused.

2.2 Protocol

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

[1] The object-oriented programming community does not have a standardized vocabulary. While "sending a message" is the most common term, and is used in the Smalltalk and Lisp communities, C++ programmers refer to this as "calling a virtual function".

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.[2] 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

[2]Booch shows how standard protocols might be used in Ada.[Boo87]

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.

An example of a Smalltalk class that needs to be reorganized is View, which defines a user-interface object that controls a region of the screen. View has 27 subclasses in the standard image, but is concrete. A careful examination reveals a number of assumptions made in View that most of its subclasses do not use. The most important is that each view will have subviews. In fact, most subclasses of View implement views that can never have subviews. Quite a bit of code in View deals with adding and positioning subviews, making it very difficult for the beginning programmer to understand the key abstractions that View represents. The solution is simple: split View into two classes, one (View) of which is the abstract superclass and the other (ViewWithSubviews) of which is a concrete subclass that implements the ability to have subviews. The result is much easier to understand and to reuse.

2.5 Inheritance vs. decomposition

Since inheritance is so powerful, it is often overused. Frequently a class is made a subclass of another when it should have had an instance variable of that class as a component. For example, some object-oriented user-interface systems make windows be a subclass of Rectangle, since they are rectangular in shape. However, it makes more sense to make the rectangle be an instance variable of the window. Windows are not necessarily rectangular, rectangles are better thought of as geometric values whose state cannot be changed, and operations like moving make more sense on a window than on a rectangle.

Behavior can be easier to reuse as a component than by inheriting it. There are at least two good examples of this in Smalltalk-80. The first is that a parser inherits the behavior of the lexical analyzer instead of having it as a component. This caused problems when we wanted to place a filter between the lexical analyzer and the parser without changing the standard compiler. The second example is that scrolling is an inherited characteristic, so it is difficult to convert a class with vertical scrolling into one with no scrolling or with both horizontal and vertical scrolling. While multiple inheritance might solve this problem, it has problems of its own. Moreover, this problem is easy to solve by making scrollbars be components of objects that need to be scrolled.

Most object-oriented applications have many kinds of hierarchies. In addition to class inheritance hierarchies, they usually have *instance hierarchies* made up of regular objects. For example, a user-interface in Smalltalk consists of a tree of views, with each subview being a child of its superview. Each component is an instance of a subclass of View, but the root of the tree of views is an instance of StandardSystemView. As another example, the Smalltalk compiler produces parse trees that are hierarchies of parse nodes. Although each node is an instance of a subclass of ParseNode, the root of the parse tree is an instance of MethodNode, which is a particular subclass. Thus, while View and ParseNode are the abstract classes at the top of the class hierarchy, the objects at the top of the instance hierarchy are instances of StandardSystemView and MethodNode.

This distinction seems to confuse many new Smalltalk programmers. There is often a phase when a student tries to make the class of the node at the top of the instance hierarchy be at the top of the class hierarchy. Once the disease is diagnosed, it can be easily cured by explaining the differences between the instance and class hierarchies.

3 Software reuse

One of the reasons that object-oriented programming is becoming more popular is that software reuse is becoming more important. Developing new systems is expensive, and maintaining them is even more expensive. A recent study by Wilma Osborne of the National Bureau of Standards suggests that 60 to 85 percent of the total cost of software is due to maintenance [LS80]. Clearly, one way to reuse a program is to enhance it, so maintainance is a special case of software reuse. Both require programmers to understand and modify software written by others. Both are difficult.

Evolutionary lifecycles are the rule rather than the exception. Software maintenance can be categorized as corrective, adaptive, and perfective. Corrective maintenance is the process of diagnosing and correcting errors. Adaptive maintenance consists of those activities that are needed to properly integrate a software product with new hardware, peripherals, etc. Perfective maintenance is required when a software product is successful. As such a product is used, pressure is brought to bear on the developers to enhance and extend the functionality of that product. Osborne reports that perfective maintenance accounts for 60 percent of all maintenance, while adaptive and corrective maintenance each account for about 20 percent of maintenance. Since 60% of maintenance activity is perfective, an evolutionary phase is an important part of the lifecycle of a successful software product.

We have already seen that object-oriented programming languages encourage software reuse in a number of ways. Class definitions provide modularity and information hiding. Late-binding of procedure calls means that objects require less information about each other, so objects need only to have the right protocol. A polymorphic procedure is easier to reuse than one that is not polymorphic, because it will work with a wider range of arguments. Class inheritance permits a class to be reused in a modified form by making subclasses from it. Class inheritance also helps form the families of standard protocols that are so important for reuse.

These features are also useful during maintenance. Modularity makes it easier to understand the effect of changes to a program. Polymorphism reduces the number of procedures, and thus the size of the program that has to be understood by the maintainer. Class inheritance permits a new version of a program to be built without affecting the old.

Many of the techniques for reusing software written in conventional languages are paralleled by object-oriented techniques. For example, program skeletons are entirely subsumed by abstract classes. Copying and editing a program is subsumed by inheriting a class and overriding some of its methods. The object-oriented techniques have the advantage of giving the new class only the differences between it and the old, making it easier to determine how a new program differs from the old. Thus, a set of subclasses preserves the history of changes made to the superclass by its subclasses. Conditionalizing a program by adding flag parameters or variant tag tests can almost always be replaced by making a subclass for each variant and having the subclasses override the methods making the tests.

Software reuse does not happen by accident, even with object-oriented programming languages. System designers must plan to reuse old components and must look for new reuseable components. The Smalltalk community practices reuse very successfully. The keys to successful software reuse are attitude, tools, and techniques.

Smalltalk programmers have a different attitude than other programmers. There is no shame in borrowing system classes or classes invented by other progammers. Rewriting an old class to make it easier to reuse is as important as inventing a new class[CB86]. A new class that is not compatible with old classes is looked down upon. Smalltalk programmers expect to spend as much time reading old code to see how to reuse it as writing new code. In fact, writing a Smalltalk program is very similar to maintaining programs written in other languages, in that it is just as important for the new software to fit in as it is for it to be efficient and easy to understand.

The most important attitude is the importance given to the creation of reuseable abstractions. Kent Beck describes the difficulty in finding reuseable abstractions and the importance placed on them by saying "Even our researchers who use Smalltalk every day do not often come up with generally useful abstractions from the code they use to solve problems. Useful abstractions are usually created by programmers with an obsession for simplicity, who are willing to rewrite code several times to produce easy-to-understand and easy-to-specialize classes." Later he states: "Decomposing problems and procedures is recognized as a difficult problem, and elaborate methodologies have been developed to help programmers in this process. Programmers who can go a step further and make their procedural solutions to a particular problem into a generic library are rare and valuable."[OBHS86]

The Smalltalk programming environment includes a number of tools that make it easier to reuse classes. There is a browser for examining and organizing classes, cross reference tools, and a tool for change management and detecting conflicts between versions[Gol84]. Although experience has shown the need for improvements to these tools and has generated ideas for new tools[Roc86, BC86b, BC86a, GB81], the existing tools greatly aid reuse in Smalltalk.

Techniques that improve reuse in Smalltalk can be divided into the coding rules and the design rules. Rochat discusses coding rules for Smalltalk that make programs easier to understand and reuse[Roc86]. The sixth section of this article describes design rules. These rules are based on the fact that useful abstractions are usually designed from the bottom up, i.e. they are discovered, not invented. We create new general components by solving specific problems, and then recognizing that our solutions have potentially broader applicability. The design rules in this paper are a way of converting specific solutions into reuseable abstractions, not a way of deducing abstractions from first principles.

4 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.[3] 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

[3] Apparently the name for frameworks at Xerox Information Systems is "teams".

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.

MacApp is a framework for Macintosh applications[Sch86]. An abstract MacApp application consists of one or more windows, one or more documents, and an application object. A window contains a set of views, each of which displays part of the state of a document. MacApp also contains commands, which automate the undo/redo mechanism, and printer handlers, which provide device independent printing. Most application classes do little besides define the class of their document. They inherit a command interpreter and menu options. Most document classes do little besides define their window and how to read and write documents to disk. They inherit menu options for saving the documents and tools for selecting which document to open next. An average programmer rarely makes new window classes, but usually has to define a view class that renders an image of a document. MacApp not only ensures that programs meet the Macintosh user-interface standard, but makes it much easier to write interactive programs.

Other frameworks include the Lisa Toolkit[AC84], which was used to build applications for the Lisa desktop environment, and the Smalltalk Model/View/Controller (MVC), which is a framework for constructing Smalltalk-80 user interfaces[Gol84]. Although these frameworks are concerned primarily with implementing a standard user interface, frameworks are by no means limited to the user interface. For example, the Battery Simulation[Foo88] is a framework for constructing realtime psychophysiological experiments.

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.

4.1 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 tailer 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.

One way of characterizing the difference between white-box and black-box frameworks is to observe that in white-box frameworks, the state of each instance is implicitly available to all the methods in the framework, much as the global variables in a Pascal program are. In a black-box framework, any information passed to constituents of the framework must be passed explicitly. Hence, a white-box framework relies on the intra-object scope rules to allow it to evolve without forcing it to subscribe to an explicit, rigid protocol that might constrain the design process prematurely.

A framework becomes more reuseable 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.

4.2 Toolkits

An object-oriented application construction environment, or toolkit, is a collection of high level tools that allow a user to interact with an application framework to configure and construct new applications. Examples of toolkits are Alexander's Glazier system for constructing Smalltalk-80 MVC applications[Ale87], and Smith's Alternate Reality Kit[Smi87]. All toolkits are based on one or more frameworks.

One of the advantages of black-box frameworks is that they are better at serving as the foundation of a toolkit. It is easy to build a tool that lets a user choose prebuilt components and connect them together, and a successful black-box framework permits most applications to be constructed that way. An example of such a tool is Glazier[Ale87], which builds an application within the Model/View/Controller framework using pluggable views.

Frameworks make it easier to define specialized programs for constructing classes. For example, a compiler might provide tools for building parsers, lexers, and code generators. It is easier to build a tool for constructing classes with well defined interfaces than it is to build a general purpose automatic programming system.

5 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 worthwhile 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.

One of the reasons that Smalltalk is good for prototyping is that the programmer can borrow code from anywhere in the system. However, this should never be mistaken for good design. It is almost always necessary at the end of a project to reorganize the class hierarchy. Unfortunately, few tools help this task. Section 6 will discuss how to recognize class hierarchies that need to be reorganized. Suggestions for tools to aid reorganization will appear in Section 7.

One sign that a good abstraction has been found is that code size decreases, indicating that code is being reused. Many Smalltalk projects have periods in which the size of the code increases at a steady rate, followed by periods in which little change occurs to the code, followed by a sharp decrease in the size of the code. Code size increases as the programmers add new classes and new methods to old classes. Eventually the programmers realize that they need to rearrange the class hierarchy. They spend a bit of time in debate and experimentation and then make the necessary changes, usually creating a new abstract class or two. Since Smalltalk programs tend to be compact, it is feasible to rewrite a system many times during its development. The result is much easier to understand and maintain than typical nonobject-oriented systems.

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 reuseable 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 waystation 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.

Barbara Liskov, in a keynote address given at the OOPSLA '87 conference in Orlando, distinguished between inheritance as an implementation aid (which she dismissed as unimportant) and inheritance for extending the abstract functionality of an object [Lis87]. Liskov claims that in the later case, only the abstract specification, and not the internal representation of the parent object, should be inherited. She was in effect advocating that only the black-box framework style should be employed. Such a perspective ignores the value of white-box frameworks. Prohibiting white-box frameworks ignores both their value in their own rights, and their value as the progenitors of mature components.

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.

6 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.

Object-oriented design starts with objects. Booch suggests that the designer start with a natural language description of the desired system and use the nouns as a starting point for the classes of objects to be designed[Boo86]. Each verb is an operation, either one implemented by a class (when the class is the direct object) or one used by the class (when the class is the subject). The resulting list of classes and operations can be used as the start of the design process.

Operations on an object are always thought about from the object's point of view. Thus, instead of displaying an object, an object is asked to display itself. Methods are receiver-centric—many of the comments in the standard Smalltalk-80 image use the word "I" to refer to the receiver. This is in stark contrast to other ways of programming, where “The use of anthropomorphic terminology when dealing with computer systems is a sign of professional immaturity”[Dij82].

Booch’s design methodology defines classes for objects in the problem domain. However, classes are often needed for operations in the problem domain. For example, compiling a program can be thought of as an operation on programs. However, because compilation is so complex, it is best to have separate compiler objects to represent compilation. The *compile* operation on programs would make a new compiler object and use it to make a compiled version of the program.

It can be difficult to decide whether an operation should be implemented as a method in a class or as a separate class. Halbert and O’Brien discuss this problem at length[HO87]. In general, there is no absolute way to decide, but positive answers to the following questions all indicate that a new class should be created.

1. Is the operation a meaningful abstraction?
2. Is the operation likely to be shared by several classes?
3. Is the operation complex?
4. Does the operation make little use of the representation of its operands?
5. Will relatively few users of the class want to use the operation?

The compiler example shows that it is possible to make an operation both a class and a method by having the method make an object of the class. This separates the implementation of the operation from that of the class and makes it more reuseable, but permits the user to continue to think of the operation as a method.

It can also be difficult to decide which class should implement an operation. Operations with several arguments can frequently be implemented as methods in the classes of any of its arguments. The rules listed above can also be used to make this decision. For example, if an operation does not send messages to an object or access its instance variables then it should not be in the object’s class.

We are not implying that classes can be reorganized mechanically. 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.

6.1 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,[4] 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

[4] Smalltalk message names indicate the number of arguments to the message.

```
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.

Case analysis of the values of variables is usually a bad idea, too. For example, a parse tree might contain nodes that represent instance variables, global variables, method arguments, and temporary variables. The Smalltalk-80 compiler uses one class to represent all these kinds of variables and differentiates between them on the value of an instance variable. It would be better to have a separate class for each kind of variable.

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, beaking 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.

6.2 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. A more concrete definition is given by the subtype compatibility rules of Trellis/Owl[SCB*86]. Of course, it is difficult to check type compatibility rules in an untyped language like Smalltalk, but they can be checked by hand.

In the middle of a project, it may be useful for a designer to make subclasses that are not specializations of their superclasses. If the superclasses are poorly designed, it might take a great deal of work to determine the proper design of the class hierarchy. It is better to forge ahead and then to later reorganize the classes. Thus, a subclass might be more general than its superclass or might have little relationship to its superclass other than borrowing code from it.

6.3 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 seperate 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.

7 Object-Oriented Programming Tools

There are at least two new kinds of tools needed for the style of programming we have just described. One kind helps the programmer reorganize class hierarchies and the other helps the programmer build applications from frameworks. Other tools would also be helpful, such as tools to help a programmer find components in libraries, but these will not be discussed.

It takes a great deal of inspiration to construct a good class hierarchy. However, it is possible to build tools that would let a programmer know that a class hierarchy had problems. These tools would be like the English style tools in the Unix writer's workbench. They would complain about perceived problems but would let the programmer decide whether the complaints were valid and how to fix them. Other tools could help reorganize the class hierarchy once a problem was diagnosed. For example, if a method of a superclass is ignored by its subclass then some abstractions in the superclass are not being inherited by the subclass. This is probably a case of subclassing for generalization or code sharing. It might be best to break the superclass into two classes, one a subclass of the other. The new subclass would have all the methods that are unused by the old subclasses. Similarly, a sign of inheritance for code sharing is that many of a superclass's methods are redefined. Perhaps some of these redefined methods should be in a concrete subclass, making the superclass abstract.

Reorganizing a class hierarchy is not difficult in Smalltalk, since it is easy to change the superclass of a class and to add and remove instance variables.

It is difficult to copy a class, but copying is rarely needed. A more important problem is that the lack of type checking in Smalltalk means that if a method inherited by two subclasses is moved into one of the subclasses then no warning will be given until runtime. It is virtually impossible to reorganize class hierarchies without creating a few missing methods, though these are fortunately easy to fix. However, it is very difficult to change core class hierarchies, since any mistake will crash the system. It would be good to have tools for building reorganization plans and for inspecting the results of applying these plans. Consistency checks could help ensure that the plans would result in class hierarchies with the same behavior as the original ones.

Much Smalltalk programming consists of combining existing components within a black-box framework. It seems unnecessarily complex to do this by writing a program—it would be much simpler to use graphical tools to select components and attach them to each other. This would make a system like Smalltalk much more useful to nonprogrammers. Such a system could also check to make sure that the components were compatible with each other. While special purpose tools like Glazier are valuable, ideally there could be general purpose tools that would work for any framework.

The interfaces between components in a framework are not fixed but depend on the classes of the components. A particular component may place more restrictions on the other components. For example, if a scroll controller is used in a MVC triad then the view will have to be able to respond to scrolling messages. This requirement is evidenced by the fact that the controller will send some specialized messages to the view and so not every view will be compatible with it.

The set of messages that an object can understand is essentially its type. In fact, determining which classes can be used in a framework is the same thing as determining type compatibility. Use of type constraint propagation can be used to determine the set of classes that can be used as components given that some components have already been chosen. If there were some way to describe an abstract framework and a database of classes that could serve as components, it should be possible to let the programmer pick components from a menu, only allowing consistent choices of components.

8 Conclusion

Smalltalk programmers often tell stories of how they built a complicated application in a few days. These experiences can occur only because the programmers are able to reuse so many software components and abstract designs. Building reuseable components and designs takes much more time. However, it is time that pays off handsomely in the long run.

A number of factors account for the high reusability of object-oriented components. Polymorphism increases the likelyhood 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 reexploit 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 129–131. 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, Object-Oriented Programming Systems, Languages and Applications*, 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, Object-Oriented Programming Systems, Languages and Applications*, 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.

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to trellis/owl. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, November 1986. printed as SIGPLAN Notices, 21(11).

[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, Object-Oriented Programming Systems, Languages and Applications*, 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.

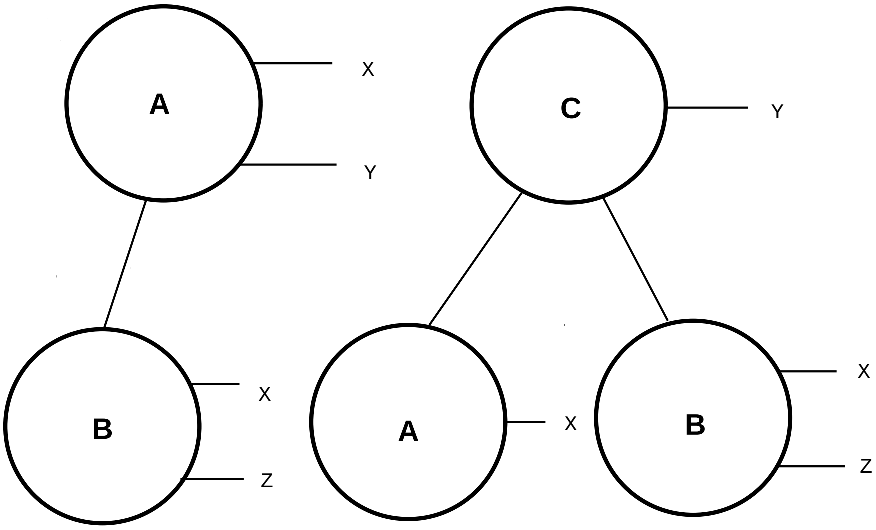

Figure 1