Designing Reusable Classes

Ralph E. Johnson & Brian Foote

Journal of Object-Oriented Programming
June/July 1988, Volume 1, Number 2, pages 22-35

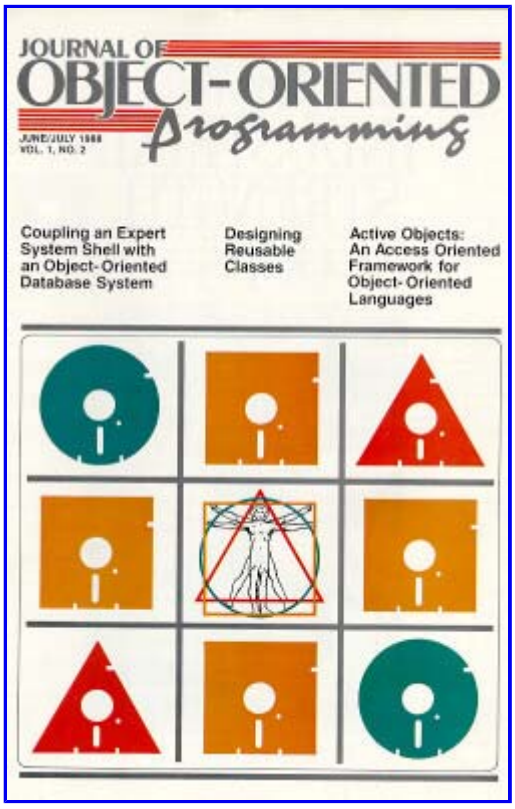
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Ave., Urbana IL 61801
(217) 244-0093, (217) 328-3523
johnson@cs.uiuc.edu, *foote@cs.uiuc.edu*

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.

Introduction



Object-oriented programming is often touted as promoting software reuse [Fischer 1987]. 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 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.

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.

Polymorphism

Operations are performed on objects by ``sending them a message'' (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''.) 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 [Seidewitz 1987]. 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 explicitly 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.

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 [Johnson 1986]. 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. (Booch shows how standard protocols might be used in Ada [Booch 1987].) 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.

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++ [Stroustrup 1986] 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.

Abstract Classes

Standard protocols are often represented by *abstract classes* [Goldberg & Robson 1983].

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.

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.

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 [Meyers 1988]. Clearly, one way to reuse a program is to enhance it, so maintenance 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 reusable 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 programmers. Rewriting an old class to make it easier to reuse is as important as inventing a new class [Cunningham & Beck 1986]. 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 reusable abstractions. Kent Beck describes the difficulty in finding reusable 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. [O' Shea et. al. 1986]

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 [Goldberg 1984]. Although experience has shown the need for improvements to these tools and has generated ideas for new tools [Rochat 1986][Beck & Cunningham 1986a] [Beck & Cunningham 1986b][Goldstein & Bobrow 1981], 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 [Rochat 1986]. 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 reusable abstractions, not a way of deducing abstractions from first principles.

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. (Apparently the name for frameworks at Xerox Information Systems is ``teams".) 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.

MacApp is a framework for Macintosh applications [Schmucker 1986]. 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 [Apple 1984], which was used to build applications for