Appears in OOPLSA'92 Proceedings.

Issues in the Design and Specification of Class Libraries

Gregor Kiczales and John Lamping
Xerox Palo Alto Research Center*

Abstract

The design and specification of an extensible class library presents a difficult challenge: because extensibility comes from allowing the user to override parts of the implementation, more of the internal structure must be exposed to the user than in a typical procedure library. This raises issues in both how the library is designed and how its specification is written. Specification of the CLOS Metaobject Protocol required a combination of new and existing techniques to address these issues. We present those techniques, and discuss their relation to the underlying issues.

1 Introduction

Object-oriented programming has been praised for many virtues, of which we believe code reuse to be the most important. It is also the most subtle. The claim is that object-oriented techniques make it possible to build "class libraries" that are reusable in the sense that, when we later build systems that require their sort of functionality, we can reuse the library rather than having to code again from scratch. There are two important properties that cause class libraries to be reusable in this way: generality and extensibility. While the distinction between the two is not sharp, generality refers to the ability of one system, without modification, to serve in a large range of circumstances, while extensibility refers to the ability of a system to be easily modified to better meet a particular need.

One way to understand these properties is through an analogy to the hardware domain—Apple's Macintosh computers. The original Macintosh could be used for an incredibly wide range of purposes, ranging from spreadsheets, to paint programs, to educational software, to text editing and so on. This was the result of a concious effort on the part of the designers to build a general purpose machine, one that could meet the needs of a wide range of users. By making their design general, they also made it widely reusable. But the hardware design was not extensible—all machines were alike and there was no provision for user customization. If an application needed more memory than was provided, or a larger screen, or color, or anything else that, even in a relatively small way, exceeded the capabilities of the machine, the Mac could not be used. There was, in essence, no provision to reuse what was good about the design and only change those parts that were inadequate.

To address these problems, later designs allow the user to replace (or enhance) various internal modules: memory can be expanded and upgraded, disks can be upgraded, the monitor can be changed, the display card can be replaced. In addition, a variety of other components can be added, as long as they obey the appropriate bus/interface protocol: ethernet cards, MIDI interfaces, math coprocessors and the like. This extensibility essentially allows users to stretch the machine design in the direction of their needs. None of the resulting extensions

*3333 Coyote Hill Rd., Palo Alto, CA 94304; (415)812-4888; Gregor@parc.xerox.com, Lamping@parc.xerox.com.

changes the fundamental nature of the machine—it remains a Mac at heart—but they do make it better suit the particular user's needs. In this case, a combination of generality and extensibility is making the design more reusable than would be possible with generality alone.

Generality and extensibility play similar roles in class libraries. For example, the button library from a user interface toolkit might provide a collection of general purpose mouseable buttons that the user can create, put on the screen, click on, etc. Like the original Mac, this library derives its reusability from its generality. Also like the original Mac, the limits of this library begin to show when some user wants a particularly specialized kind of button, one that displays itself as a stop sign for example. But this can be addressed through extensibility. If the library design allows the user to step in and extend it—to make a button draw itself in a new way—then that library will be more reusable than one which is not extensible.

While generality is always an integral (and often a sufficient) means of developing a reusable library, our focus in this paper is on extensibility. In particular, we want to discuss the design and specification of extensible class libraries. Like all specifications, we want ours to meet two goals: (i) To be precise enough that users can use and extend the library. (ii) But not be so constrained that they overly restrict implementor freedom, thereby precluding efficient implementations, enhanced implementations, or further development of the library.

Our concern with these issues first arose while trying to specify the CLOS Metaobject Protocol [KdRB91], a medium-sized class library developed over the last several years.[1] Our goal was to provide significant extensibility in a programming language; because of the domain, we expected there would be multiple implementations, and could foresee various kinds of freedom implementors would want to have.

In attempting to do this, we ran into what was, at the time, surprising difficulty. We found that we had little trouble coming up with a general sense of how the protocol should work and what extensibility we wanted to provide, but that as we tried to actually write a specification, we invariably ended up overconstraining the implementor in unacceptable ways. We were fortunate that several prospective implementors and a number of prototype users were involved in the design; this helped us to see the problems more quickly and eventually work out a solution.

Using a combination of new and existing object-oriented design and specification techniques, we ended up producing an informal (in English) specification [KdRB91] and a prototype implementation [BKK+86, KR90]. The specification is currently being implemented by several vendors, on different hardware platforms. Early versions of those implementations and our prototype implementation are in use and have been extended by users in various anticipated and unanticipated ways.

This paper presents the techniques we used, with particular attention to how to balance the desire to provide the user with rich, extensible functionality against the desire to provide the implementor with appropriate leeway. We now believe that the library specification problem is a critical one for our community to address. By gathering together new and existing practice in this area, we hope to contribute to a dialogue in the community on this important topic.

1.1 An Example

Throughout the paper, we will continue to work with the button library example. Its simplicity makes it possible to capture, in a relatively concise form, many of the issues that come up in larger libraries.

This section sketches the button library somewhat more fully, with particular attention to the extensibility the library is intended to provide. More details will be filled in later, as we develop the specification technology required to express them. We will work with terminology and syntax from the Common Lisp Object System (CLOS) [BDG+88],

[1] There are approximately 25 classes, 75 operations or messages and 125 methods.

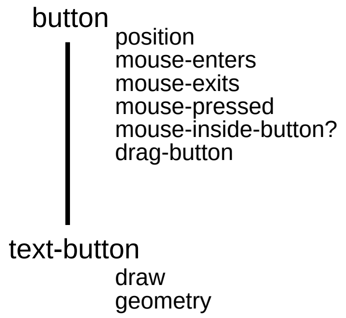

Figure 1: A simple button widget class library. Two classes, together with their methods are shown.

since that is the language in which this work was developed. But the framework we are presenting is applicable to other object-oriented languages; Section 5 discusses how it can be adapted to libraries written in C++[Str91]. To help make the paper accessible to those not familiar with CLOS, all CLOS-specific terms are explained in footnotes.

In the library, each button is an object. The window toolkit uses a well-defined protocol operating on the buttons to display them, move them, mouse-highlight them, invoke appropriate behavior when they are clicked on etc. A simplified sketch of the library is shown in Figure 1. There is one concrete class, `text-button`, and one abstract base class, `button`. All the generic functions save `geometry` and `draw` have methods provided by the class `button`.[2] These methods provide general-purpose implementations of these generic functions, which should be appropriate for most classes of button. No general purpose implementation of `draw` and `geometry` is possible; concrete subclasses are expected to provide their own. The class `text-button` provides methods that display buttons as a text string surrounded by a box.

The library presents the user with two interfaces, one through which buttons are *used*—that is created and placed on the screen; and another through which the library is *extended*—by defining new subclasses and appropriate methods. To create a button, the user calls `make-instance`:[3]

```
(make-instance 'text-button
  :position #(50 100)
  :text "Shutdown"
  :function #'shutdown-system)
```

To extend the library, the user defines a specialized subclass. A class of button that displays itself as a stop sign can be defined by: (i) Defining the new class as a subclass of `button`. (ii) Defining a method on `draw`, specialized to that class, that takes care of drawing the button. (iii) Defining a method on `geometry`, that returns a description of how much space, around its position, the button occupies. This would look something like:

```
(defclass stop-button (button) ())

(defmethod draw ((b stop-button))
  (draw-stop-sign (position b)))

(defmethod geometry ((b stop-button))
  *stop-sign-geometry*)
```

2 The Problem

This example shows how object-oriented techniques support extensibility. They make it possible for the user of a library to extend its functionality by writing new code that will be combined with the existing implementation to form a "new" implementation. As shown in Figure 2, defining a

[2] CLOS terminology distinguishes *generic functions*, which are the rough equivalent of what some other languages call messages, and *methods*, which are pieces of code, associated with a generic function and a class, that provide the class specific behavior for that generic function. We say that we *call* a generic function on an object, and that *method lookup* then determines which method is run. The term *specializer* refers to the class that a method is associated with. We also say that a method is *specialized to a class*. (In this paper, we ignore the CLOS multi-method functionality; all methods have only a single specializer. One consequence of this is that we can speak about method *inheritance*, a familiar term, rather than using CLOS's concept of method *applicability*.)

[3] In CLOS, `make-instance` is the general purpose mechanism for creating new instances of a class.

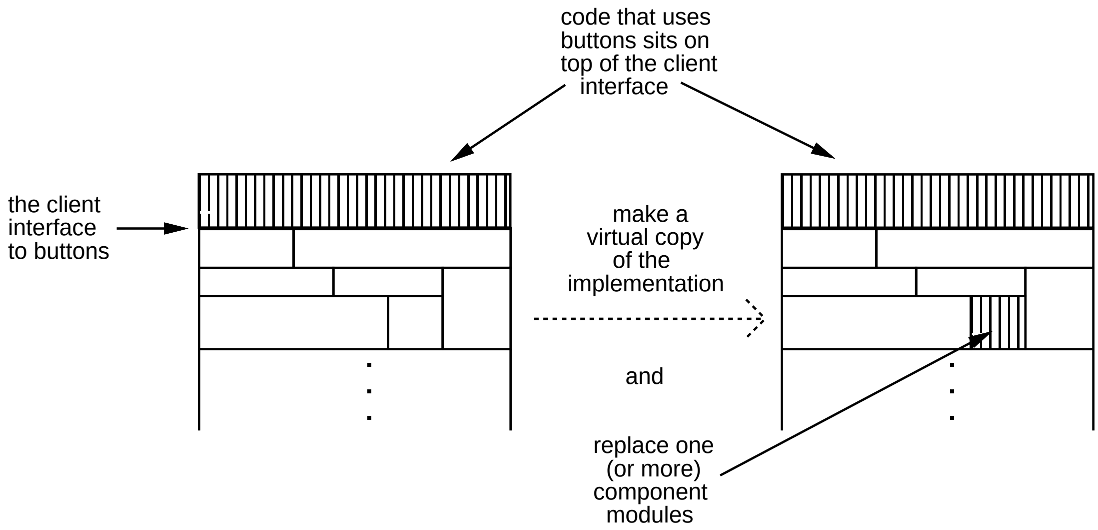

Figure 2: On the left: A class in the button library provides a client interface on top of which user code is written. Below that interface, the implementation has its own internal component structure. On the right: Object-oriented programming supports the incremental definition of a new class as an extension to the existing class. Defining the subclass makes a virtual copy of the original class's implementation; and specialized methods replace one (or more) component(s) in the copy. User code still uses the client interface to access the button functionality; but, there is now also user code running *inside* the button implementation.

subclass (i.e. `stop-button`) is like making a virtual copy of the existing implementation—it automatically inherits methods from its superclass. Defining specialized methods on the subclass (i.e. methods on `draw` and `geometry`) is like replacing, or, as in this example, filling in, internal module(s) in the copy.

The result of subclass specialization is that we end up with user-defined code running *inside* the library, where it can be called both by other user code (i.e. a direct call to `draw`) or by implementor code (i.e. when the window toolkit asks a button to display itself). This is in contrast to traditional procedure libraries where user code is never called by implementor code.[4] The problem then is how to say enough about the internal workings of the library that the user can write replacement modules, without saying so much that the implementor has no room to work.

Part of the specification problem has to do with the class graph and method inheritance. In our example, the user needs to know that if they define a subclass of `button`, they will inherit methods on `position`, `mouse-enters`, `mouse-exits` and the like. A second part of the problem has to do with protocol. The user needs to know, when they subclass `button`, what methods they must define, what behavior those methods can rely on and what those methods must and must not do.

The remainder of the paper is divided into six sections. In Section 3 we discuss the techniques we have developed for specifying class graph and inheritance relationships. Section 4 presents a combination of new and existing design and specification techniques for the protocol of a class library. Section 5 discusses the application of these techniques to libraries written in C++. In Section 6,

[4]Interfaces with callbacks are an exception, but callbacks can be seen as a "hand-built" object-oriented mechanism.

we step back and show that the issues that arise when specifying an extensible class library result not so much from object-oriented implementation technology as from the more fundamental goal of producing systems that derive their reusability, at least in part, from extensibility. The final sections discuss related work and summarize the paper.

3 Class Graph and Inheritance

In this section we focus on how to specify the class graph and inheritance structure of a library. The approach we have developed works in two parts: We first give a simple, but somewhat naive, specification of the inheritance relations, in a manner similar to Figure 1. Then, because that specification is overconstraining, we specify a set of rules that provide the implementor with desired leeway. The rules are general purpose, so they can be used with any class library.

The presentation in this section will follow the same structure: we start with the naive approach, and then show a variety of ways in which it is deficient, introducing the rules as we go along.

The naive approach is to specify: (i) the set of classes in the graph and the direct superclasses[5] of each; (ii) the set of generic functions; and (iii) the set of methods, by giving the generic function and the specializer of each.

This simplicity of this approach stems from the fact that we are basing the description on concepts in the CLOS language—direct superclasses, specializers etc. Unfortunately, it quickly becomes too restrictive. At the very least, the implementor wants the freedom to include implementation-specific enhancements. For example, the implementor might wish to provide an additional kind of button, as a subclass of `button`, which allows the button's image to be one of a family of icons.

Similarly, the implementor might want to have generic functions which are not mentioned in the specification. These might provide extra documented features, or they might simply be an internal part of the implementation. In our example, there might be a generic function, `save-underlying-bits`, which is called by the toolkit before a button is displayed.

As a first step to rectifying the situation, we found that it was necessary to introduce terminology that allows clear distinctions among the various class, generic function and method definitions.

- A *specified* definition is one which is listed in the specification of the library.

- An *implementation-specific* definition is one which is present in a given implementation, but which does not appear in the specification.

- A *portable* definition is one that is defined by the user, and that only depends on specified definitions. That is, it should be able to work with any implementation of the specified library.

- A *system-defined* definition is one that is part of the implementation, either specified or implementation specific.

- A *user-defined* definition is one defined by the user, either portable or not.

Given this terminology, we can begin to relax the simple specification to provide some of the freedom the implementor needs. (Throughout this section, the rules will be typeset this way to make them stand out.)

The implementor is allowed to: (i) Provide implementation-specific leaf classes as subclasses of specified classes. (This will be further relaxed shortly.) Such leaf classes can have methods on any system-defined generic function. (ii) Provide implementation-specific generic functions. These can have methods specialized to any system-defined class.

[5] In CLOS, the terms *direct superclass* and *direct subclass* mean that there are no intervening classes, while the terms *superclass* and *subclass* are used in cases where there may be intervening classes.

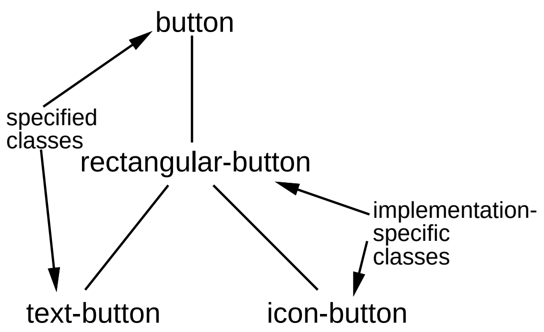

Figure 3: The class graph provided by a specific implementation has both specified classes (`button` and `text-button`) and implementation-specific classes (`rectangular-button` and `icon-button`).

> To prevent name conflicts with user definitions, the names of implementation-specific classes and generic functions must not appear in the default user package.[6]

3.1 Interposed Classes

In addition to providing implementation-specific leaf classes, the implementor may want to have implementation-specific classes that are "in amongst" the specified classes in the graph, that is, implementation specific classes which are superclasses of specified classes. This can happen when there is commonality in the implementation, perhaps among specified and implementation-specific classes, that is not reflected in the specified class graph.

In our button-library example, the implementor might want to define a class `rectangular-button` as a common superclass of both `text-button` and `icon-button`. This might be the class to which the method on `save-underlying-bits` is specialized. The resulting class graph is shown in Figure 3.

While we would like to permit the implementor to define these sorts of *interposed* classes, we need to be careful for the user. Specifically, the user needs to be assured that method inheritance will not be affected.[7] The CLOS concept of class precedence list provides a way to precisely specify the freedom we want to give the implementor.[8]

> Implementation-specific class interpositions are allowed as long as for any portable class C_P that is a subclass of one or more specified classes $C_0 \ldots C_i$, the following condition is met: In the class precedence list of C_P, the classes $C_0 \ldots C_i$ must appear in the same order as they would have if no implementation-specific modifications had been made.

3.2 Method Promotion

Given these interposed classes, situations may arise where the implementor would like to take a specified method and, rather than specializing it to its specified class, specialize it to an interposed superclass instead. We call this *method promotion*. The implementor may want to promote a method to reflect commonality among classes in the implementation that is not necessarily reflected in the specification. For example, in the class graph shown in Figure 3, and still assuming that the specification includes a method on `geometry` specialized to `text-button`, the implementor might want to promote that method to `rectangular-button`. This will make it applicable to both icon buttons and text buttons. Defining the method in this way, rather than defining two methods with identical bodies makes the implementation more clean and "object-oriented."

Again, we have to be careful that such method promotions don't affect the user's program.[9] In

[6] CLOS uses the Common Lisp package system to prevent name conflicts. While its precise workings are somewhat mysterious, it is sufficient to think of it as an idiosyncratic module system. (Also note that throughout the paper, the wording of package system rules is simplified. See pages 142 – 144 of [KdRB91] for details.)

[7] In CLOS, the resolution of multiple inheritance, together with the method combination mechanism, means that some class interpositions can affect method inheritance.

[8] A class's class precedence list is simply a linearization of its superclasses. This linearization is used to handle all inheritance and overriding decisions, so restrictions on it have a well-defined effect on all other inheritance behavior.

[9] In CLOS, a specified method can be combined with a user-defined method to get an effective method, and the po-

CLOS terminology, the precise definition of the additional freedom we allow the implementor is:

> Method promotion is permitted as long as the method inheritance of any specified generic function, at any specified or portable class, stays the same as it would have been had no implementation-specific promotions been made. We also allow implementations to merge specified methods as part of method promotion.

Effectively, this rule tells the implementor that a specified method can be "moved up" to an implementation-specific interposed class, but cannot be moved up as far as the next specified class.

3.3 Inheritance of Slots

As an additional aspect of the specification of classes, we must consider slots defined in those classes.[10] In general, we do not specify any slots, because good CLOS style dictates that methods on subclasses should (usually) not primitively access slots defined by superclasses.

But because of the CLOS rules for slot inheritance, we must go a little beyond this; we have to be sure to prevent inadvertent name conflicts between slots in user-defined and system-defined classes. As with classes and generic functions, this name conflict problem is solved using the package system. We add a restriction on implementations as follows:

> No portable class C_P may inherit, by virtue of being a direct or indirect subclass of a specified class, any slot for which the name is a symbol in the default user package.

sitioning of the specified method in the class hierarchy could effect how the methods were combined.

[10] *Slot* is the CLOS term for storage fields in an instance. (C++ calls these member data elements and Smalltalk calls them instance variables.) There are two ways to access a slot. Primitive access is intended for code that is familiar with the intimacies of an object, while access via a generic function is intended for client code. Unlike some other languages, there is no mechanism for enforcing this style, it is done by convention.

3.4 Redefining Reserved Words

The previous rules allow the implementor useful freedom in implementing the inheritance structure, but not so much freedom that it could affect user programs. In effect, they are the codification of a "you can cheat as long as you don't get caught" philosophy.[11] Before turning to the specification of protocol, there is also an important inheritance-related restriction we need to place on the user. This is to prevent user programs from damaging the implementation or otherwise inadvertently affecting other user programs they may be loaded with.

The first part of this restriction corresponds to the mandate in traditional languages against redefining reserved words—we must prevent the user from redefining specified classes, generic functions or methods.[12] That is, the user cannot change the behavior of the `draw` method specialized to the class `text-button`. If they want an alternate method, it must be on a subclass of `text-button`.

But we must go a little bit farther, to prevent the user from attempting to "fill in holes" in the method inheritance. To see how this could happen, consider our example and the `geometry` generic function. Also assume that the only specified method on `geometry` is specialized to `text-button`. Now, if the user defines two direct subclasses of `button`, they might want to define a method on `geometry`, specialized to `button` itself. This would make the method applicable to both their subclasses. They might assume that such a method definition was legal, since there is no specified method there. But this sort of method definition must be prohibited, because another application loaded into the same CLOS image could have

[11] Strictly speaking, because of the introspective facilities of the Metaobject Protocol, any deviation on the part of the implementor from the naive specification can be detected by the user. By "don't get caught" we mean that a user programming in the base CLOS language shouldn't be able to observe the deviations.

[12] In general, CLOS allows classes, generic functions and methods to be redefined. This is designed to support program development and is not intended as a mechanism for the user to alter a program someone else wrote.

the same idea, and the two would interfere.

We have found that a large number of users want to define these sorts of methods. It appears that the object-oriented story that you can "extend the functionality of the system" leads people in this direction, and not everyone anticipates the potential for trouble. For this reason, we have found that it is crucial to place the following explicit restriction on users:

> User programs must not redefine any specified classes, generic functions, or methods. User-defined methods on specified generic functions must be specialized to a user-defined class.[13] User-defined classes and generic functions must be named in a user defined package.

4 Protocols

The previous section discussed the specification of class graphs and method inheritance. Given that framework, it is possible to specify the inheritance structure of a library precisely enough that, for any portable class and generic function, a user will know what methods are applicable—while still allowing the implementor a great deal of leeway in structuring their implementation.

We now turn our attention to the protocol among objects, that is the behavior of specified generic functions and methods. The first observation is that unlike procedural libraries, it is not sufficient to simply specify the *behavior* of each generic function. We must step a least a little bit farther into the implementation and say how it relies on other generic functions to provide its behavior.

To see why this is so, consider the `draw` generic function. In order to know that defining a method on `draw` will have the desired effect, the user needs to know not just what their method should do, but must also be confident that all other parts of the system that want to draw a button will do so by actually calling `draw`. We have found that it is important to say this in two places: the callee and caller generic functions.

So, the specification of a generic function now plays two roles: (i) Saying what it does, which gives a general sense of when it is called and what its methods should do. (ii) Saying how it relies on other generic functions to provide its behavior, which provides the "backbone" of the protocol. The latter statements, because they govern all the methods of a caller generic function, are the user's guarantee that a callee generic function they specialize will actually be given the opportunity to fulfill its role.

So, the specification for `mouse-enters` would look something like:

`mouse-enters` (*button*) *[GFun]*
This generic function is called by the toolkit whenever the mouse enters the region occupied by the button. It arranges for `mouse-inside-button?` to return true, until the next call to `mouse-exits`. Then, `draw` is called to highlight the button.

4.1 Specified Methods

In addition to specifying generic functions, it is often important to say something about the behavior of the specified method(s). They are concrete implementations of the generic function abstraction, so there is generally something more complete to be said about them. Compare the specifications for the `draw` generic function and its specified method:

`draw` (*button*) *[GFun]*
Draws the button's image on the screen. The position for the image is determined by calling `position`. Calls `mouse-over-button?` to determine if the button should be highlighted.
`draw` ((*button* `text-button`))[14] *[Method]*
For a text button, the image is the button's text, in 14 point Helvetica. The image also includes a 2 pixel solid line around the text.

[13] The presence of **eql** specializers in CLOS actually requires that this restriction be somewhat more complicated. See page 144 of [KdRB91] for details.

[14] This CLOS-like syntax indicates that `draw` is a one argument generic function; that the argument is called *button*; and that this method is specialized to the class `text-button`.

The button is highlighted, if necessary, by underlining the text.

The specification of the generic function is as detailed as it can be, but it simply cannot, in and of itself, say how a particular kind of button will look. Only a method specialized to an instantiable class (i.e. `text-button`) can supply this level of detail, and until it is specified the user doesn't know enough about the behavior of the library to use the concrete classes.

4.2 Required Methods

Often, a library will provide incomplete classes that must be subclassed and extended before they can actually be used. These are generally called *abstract* classes. In the button library, `button` is an abstract class, since in order to use its functionality a subclass must be defined that provides methods for `draw` and `geometry`.

This is a common enough occurence that it is useful, in the specification, to explicitly point out methods that a subclass is required to provide. Some languages, such as Flavors, even provide mechanisms for declaring this that enable static checking of subclasses. Smalltalk programmers achieve a similar effect, only at runtime, using a default method that signals a standard error.

We did not, in the CLOS Metaobject Protocol specification, list required methods explicitly. This is in large part because the design of that library is such that users will not generally subclass abstract classes directly, but will instead subclass concrete classes. As a result, no methods will be *required*; the methods to be defined will depend on the user's goals.

In a more typical CLOS library, required methods should be listed explicitly. Because CLOS provides no built in mechanism for declaring these, a trick similar to Smalltalk's should be used to provide the user with reasonable error messages when they fail to provide a required method.

4.3 Efficiency Concerns

Fully specifying the intercalling relationships among generic functions, in the way we have been suggesting, can give the user a lot of power. But, like our initial naive approach to the specification of inheritance relationships, this approach can overburden the implementor. In particular it can lead to specifications which are difficult, if not impossible, to implement efficiently.

This is because what we are doing with object-oriented techniques is introducing a delayed binding of internal components of the system to their implementations. In our example, what code will implement `draw` isn't known until (roughly speaking) run time. This delayed binding has performance costs. First, there is the obvious overhead of the generic function invocation itself. In most cases, this is minor, and is not of concern to us here. But there is another, more subtle and generally more significant cost. The delayed binding means that the most fundamental of performance techniques—manually exploiting knowledge about other parts of the system—cannot be used.

For example, consider the `drag-button` and `draw` generic functions. Under our naive specification approach, the specification of `drag-button` would be in terms of `draw`: For each pixel the mouse drags the button, `drag-button` would first erase the button's old image (using the saved underlying bits) and then call `draw` to redraw the button. This simple protocol means that user-defined methods on `draw` also handle the case when the button is dragged.

But this protocol has the potential to be expensive, since it requires repeated redrawing of the button. Moreover, if the behavior of `draw` is constant (that is, it does not vary with the button's position on the screen) the expense is needless; `drag-button` could simply copy (bitblt) the button's image around on the screen rather than calling `draw` each time.

In a closed system, like a library of procedures, the implementor would be free to perform this optimization—that is to exploit the knowledge that comes from early binding—by folding into the implementation of `drag-button` what they know about how `draw` works. But in an extensible system, the delayed binding can preclude such optimizations. We use two techniques, functional protocols and consistent generic functions, to return to the implementor some of the knowledge that might help optimize performance.

4.4 Functional Protocols

Consider the `geometry` generic function. The toolkit uses its result every time the mouse moves to determine whether the mouse has entered or exited a button. Under the simple version of the specification, this would suggest that `geometry` is called, for every button, every time the mouse moves. This could be a significant performance load. If on the other hand, once a button has been initialized, its geometry is constant, we can allow the implementation to call `geometry` once per button and then memoize the result.

This is an example of a *functional protocol*—the generic function returns a constant function of its argument. The notion of functional protocols can be generalized somewhat. Rather than allowing the generic function to be called only once, the specification can describe the conditions under which a memoized result remains valid. The implementation is then free to memoize results as long as those conditions hold. For example, in a more elaborate version of the library, where the text associated with a button can be changed, results returned from `geometry` would be valid until such a change.

4.5 Consistent Generic Functions

Consistent generic functions are another way of relaxing the naive specification, to give the implementor more latitude to optimize the implementation. The idea is to explicitly identify sets of generic functions which should behave as if there are calls between them, but where the methods are actually free to in-line knowledge about one another. This means that when the user overrides[15] a method on the called generic function they usually need to override the corresponding method on the calling generic function.

In the `drag-button` example, we can allow the more efficient implementation, without actually having to talk about it in the specification, by specifying that `drag-button` and `draw` are *consistent* with each other. That is, methods on `drag-button` should behave as if they called `draw`, but are not required to actually call `draw`.

In our specifications, we explicitly indicate consistent generic functions; the specification of both the calling and the called generic function notes their relationship. The specification of methods is also affected since they become linked in a way that means they have to be overridden together or not at all. So, the specification of `draw` and its method now looks like:

> `draw (`*button*`)` *[GFun]*
> Draw the button's image on the screen. The position for the image is determined by calling `position`. Calls `mouse-over-button?` to determine if the button should be highlighted. The behavior of the `drag-button` generic function is consistent with a call to `draw`, but methods on `drag-button` are permitted to elide the actual call and fold in the behavior of the corresponding method on `draw`.
>
> `draw ((`*button* `text-button))` *[Method]*
> For a text button, the image is the button's text, in 14 point Helvetica. The image also includes a 2 pixel solid line around the text. The button is highlighted, if necessary, by underlining the text. The behavior of this method may be inlined by the `drag-button` `((`*button* `text-button))` method, so overriding this method requires overriding that one as well.

[15] In CLOS, a new method can override or extend a previous method. A method M_1 on a subclass, *extends* a method M_2 on a superclass if, when it is run, it also causes M_2 to run. (Using `call-next-method` which is CLOS's equivalent of Smalltalk's send-super. The nearest approximation in C++ requires qualifying the member function name with the superclass name.) M_1 *overrides* M_2 if it prevents M_2 from running.

4.6 Private Communication Among Methods

A similar situation happens when a set of specified methods have private communication amongst themselves. Consider the specified methods on `mouse-enters`, `mouse-exits` and `mouse-inside-button?`. There is some piece of hidden storage, the mouse over button bit, that all these methods have access to. But direct access to that storage is not specified in the protocol (there is no specified generic function and method for writing into that storage). This means that if a user wants to override the specified method on one of these generic functions, say `mouse-enters`, they must override the specified methods on the others as well.

While this sort of situation can be inferred from a specification (i.e. the absence of a specified generic function and method to write the storage clearly means that the three other methods have private communication), we believe it should be stated explicitly. In the specification of each method in such a set, we explicitly mention the others, and say that overriding one requires overriding them all.

4.7 Non-Overridable Methods

There are some cases where the implementor would like to define a method which they are certain will not be overridden. Initialization methods are the classic example of this.[16] For example, the user might want to define an initialization method which adds the newly-created button to a list of all the buttons in the world. While it is perfectly fine for the user to add additional initialization behavior, actually overriding the implementor's initialization behavior would have disastrous consequences.

CLOS allows us two mechanisms for providing the implementor this kind of guarantee. First, in the specification for a method, we can explicitly say that it cannot be overridden. (This requires the user, when they define a more specific method, to invoke `call-next-method` to make sure the specified method gets to run.) This technique works well in many cases, but it requires the library designer to anticipate the places where an implementor would want non-overridable methods.

The second approach is more general, and takes advantage of the CLOS method combination facility. We explicitly allow the implementor to define unspecified before and after methods on specified generic functions.[17] Since these before and after methods cannot be overridden, they are guaranteed to run.

4.8 Layered Protocols

In designing protocols for extensibility, there appears to be a basic tension between ease of use and degree of extensibility. That is, generic functions which have a lot of power tend to be harder to write methods for, and generic functions for which the methods are easier to write tend to have less power. This tension is a problem for designers, who want to build libraries that are both powerful and easy to use. Protocol layering is a technique that allows us to make typical extensions easy to write, without losing the power required for more substantial extensions.

We have previously said that the specification for a method tends to be more concrete than the specification of the generic function. In a layered protocol, the specification of a method goes so far as to mandate additional protocol: calls to generic functions not mentioned in the generic function specification. For example, consider the following alternative specification for `draw`:

`draw` (*button*) *[GFun]*
Draw the button's image on the screen. The

[16] CLOS provides a powerful mechanism for initializing objects when they are first created. This mechanism is under control of generic functions, in particular methods on `initialize-instance` are a primary place for such work to get done.

[17] In CLOS, the default method combination supports before and after methods. These are not overridden by normal methods defined on more specific classes. Instead, *all* the inherited before methods are run, then the most specific primary method is run, then *all* the inherited after methods are run.

position for the image is determined by calling `position`. Calls `mouse-over-button?` to determine if the button should be highlighted.

`draw ((`*button* `text-button))` *[Method]*
Text buttons are displayed as the text string, surrounded by a box. This method calls `font` to find out what font to use and `border-width` to find out how wide the border should be. The button is highlighted, if necessary, by underlining the text.

The generic function `draw` is the *coarse* protocol. Methods defined on it have a lot of power, but are correspondingly difficult to write. The class `text-button` now introduces a *finer* layer in the protocol: the generic functions `font` and `border-width`. These are easier to write methods for, but they have less power. By layering the protocol this way, we make the common specializations, to the font and border width of text buttons, easy to do, without losing the power of the more coarse-grained protocol, `draw`, for all kinds of buttons.

These two layers reflect a difference that is common in layered protocols. The coarse layer, `draw`, is a procedural protocol; it is called to do the actual work of drawing the button. The finer layer is more of a declarative protocol; `text` and `border-width` don't actually do any work, they return results in a "mini-language" which their caller will process.

Note that these extra protocol layers only apply to `text-buttons`, other subclasses of `button` are only governed by the `draw` protocol. Moreover, if the user defines a subclass of `text-button`, they are free to override the method on `draw` and eliminate (or change) the finer layer of the protocol. That is the advantage of layered protocols: we can provide finer layers for the user who finds them convenient, but they can be eliminated if they get in the way.

One additional point is that the provision of extra protocol layers is not restricted to the specifier of the library alone. A common scenario is for the user of a class library, as part of writing a particular application, to develop, for their own use, an extended library, that provides layered protocols more finely tuned to the needs of the application. Similarly, organizations that buy commercial class libraries might customize them with their own layers.

5 C++

The techniques of class library specification we have been discussing are not restricted to CLOS; they can be adapted to other languages as well. This section discusses their application to C++ in particular, beginning with the class graph and inheritance leeway rules from Section 3 and following with the protocol design and specification techniques from Section 4.

First off, to allow implementation leeway, we require that any portable code be recompiled for different implementations of the library.

Most name conflicts can be avoided by using C++'s information hiding facilities. There remains an issue, however, for implementation-specific members of specified classes. Since they are visible to portable subclasses, they can cause unexpected ambiguity in combination with multiple inheritance.[18] To avoid this problem, the specification must impose one of two restrictions: either implementation-specific members of specified classes must take names from a pool reserved for that purpose, or portable code must always use qualified names when using multiple inheritance.

The basic principle for class interposition is the same in CLOS and C++: classes can be interposed as long as user code is not affected by it. The exact conditions that translates to are, of course, different. Because C++ does not have method combination, the ordering of base classes does not matter.[19]

[18] Private members do not solve the problem. Privacy only forbids access, leaving the name visible. For example, suppose that a specified class, `spec`, has a private implementation-specific member, `mem`, and that a user makes a class, `user`, with a public member also named `mem`. Now if the user makes a class, `combined`, that has both `spec` and `user` as base classes, then `combined::mem` will be considered ambiguous and generate an error.

[19] We are assuming that the memory layout of a class and the order of its initialization and cleanup can be ignored.

On the other hand, the presence of different kinds of accessibility and the distinction between virtual base classes and nonvirtual base classes introduces some complexities. The rule imposed on the implementor would be: implementation dependent classes can be interposed in the class graph provided that every specified class inherits exactly the same specified virtual base classes with at least as great accessibilities, and inherits exactly the same specified non-virtual base classes with at least as great accessibilities at least as many times but without introducing additional ambiguities.[20]

The specification of a C++ library can be written in such a way that method promotion is not an issue. Since there is no method combination in C++, the specification can just say what specified member functions do at specified classes, without bring up where they are inherited from. The implementor is left free to do anything that gives the specified behavior.

Unlike the rules for the inheritance structure, the issues of protocol in Section 4 are mostly language independent and carry over to C++ largely intact. The significant exception is how to guarantee the implementor that the invocation of some virtual member function will always eventually invoke the implementation's definition. Since C++ has no analogue of before and after methods, the only technique is for the specification to require any user definition for the member function to include a call to the implementation's definition.

6 Stepping Back

In both the specification of inheritance and the specification of protocol, we followed the same pattern. We started with a naive specification, which gives the user power and extensibility but overconstrains the implementor, and then relaxed that specification, to strike a better balance. In the case of specification of inheritance, we have developed a general set of rules that allow the designer of any library to present the naive inheritance model and yet give the implementor appropriate freedom. In the case of protocol, we have presented a variety of techniques, new and old, for specifying parts of it in a more "relaxed" fashion than under the naive approach. This requires the designer to decide, for each part of the protocol, what additional leeway the implementor might need.

In each case, the naive specification looked a lot like actual CLOS code fragments. It should come as no surprise that using code as a specification leads to overspecification; in the case of procedural libraries, it is well established that specifications should be less specific (and more perspicuous) than a code listing.

Why then does this problem have to be solved anew? Why can't the techniques that have been developed for procedural libraries be adapted directly? The answer can be seen by turning back to Figure 2: in extensible object-oriented class libraries, we need to specify internal structure which, in a procedural library, would be considered hidden.

What this means is that as part of learning how to specify extensible class libraries we must learn a new sense of the distinction between "implementation details" and crucial parts of the specification. In a procedural button library, even the name of the function that draws buttons would be hidden, whereas we have seen that in an extensible class library we must not only expose the name `draw`, but also a fair amount about what it does, when, and how. We still want to hide things that truly are implementation specific—we didn't divulge the names of slots for example—it is just that we need to say more about the internal structure than we used to. The tension that results from needing to relearn this balance is evident throughout this paper. This paper is about the nature of that balance and particular techniques for striking and expressing it.

It is natural to ask how general is the problem we are trying to solve: is it unique to the object oriented programming technology or does it have deeper roots? Once again, Figure 2 suggests the

[20] This statement of the rule is not quite precise in those cases where a virtual base class itself has a nonvirtual base class, but we spare the reader the details.

answer: it is deeper, it is inherent in the goal of extensible software. What we are trying to specify is what the replaceable modules of such a system are and how they interact. That is, we must specify not only the client interface (i.e. how to make and use buttons), but also the interfaces and organization among the internal modules of the system. Object oriented programming is a way of getting the most benefit out of having designed such architectures, because subclassing and inheritance allow multiple variants to coexist, cooperate, and be easily constructed. It makes us more motivated to design them. But the challenge comes from designing extensible systems, not from using object oriented programming per se.

Thus, while the tecthniques we have presented are phrased in terms of object-oriented programming, they are actually addressing the more general issues of specifying extensible systems. The underlying ideas could be adapted to any technology being used to build such a system.

6.1 Specifying the Replaceable Units

The first task in specifying an extensible system is specifying what the replaceable units are: What are the pieces that can be changed to affect the behavior of the system as a whole?

In the case of object oriented programming, the problem is that while the language allows the user to subclass any class and to specialize any generic function, arbitrary replacement is almost certain to destroy the integrity of any actual system. An actual design will have a limited set of replaceable units. While the naive specification of an object oriented library would suggest that it was permissible to do any replacement that the object oriented mechanisms can carry out, an actual specification must be more restrictive.

The issues raised in Section 3 are involved with the class part of the problem. An implementation of a class library wants to "export" the documented classes, so that the user can make changes in terms of them, while keeping other classes for its own purposes. The problem is that all the classes are interrelated; the subclassing and inheritance relationships that allow the user to customize behavior are also an integral part of the implementation of the library. This problem is addressed by the rules presented in Section 3, which have the effect of distinguishing documented and internal classes.

Many of the other techniques we presented involve delimiting the units of replaceability in terms of methods. Saying that methods for a group of generic functions must be consistent, or that they may have private communication, tells the user that the replaceable unit is not the individual method, but rather the group of related methods. Similarly, non-overridable methods are a way of telling the user that a given module may not be displaced, although the user may be allowed to add functionality to it. By making the replaceable units larger, all of these techniques give the implementation more freedom.

There is a tension, when designing an extensible system, between having a few large replaceable units or many small ones. A design with many small replaceable units can make it easier for a user to make customizations, provided that only a few small, simple units need to be replaced. But, this finer control will not, in general, be able to generate as wide a range of behavior as a more coarse mechnism.

Layered protocols are a technique for relieving the tension by simultaneously having several granularities of replaceable pieces. The most coarse protocol is typically designed to accommodate as wide a range of behaviors as possible from each of its few pieces, while providing minimal support for modification. A sublayer, on the other hand, provides more convenient support for modification; but, since it mandates more internal structure, it limits the possible behaviors that can be generated. The finest subprotocol may just specify very simple controls that the user can adjust. If the user wants a behavior that is compatible with what the finer protocol can generate, they can just intervene at just a few small subpieces. But if the finer protocol can not generate the behavior the user needs, then the user is free to supersede the entire large scale piece; they can ignore its internal protocol, obeying only the rules of the coarser protocol.

6.2 Allowing for a Range of Behaviors

The important point of replaceable components is not just to allow the user to replace components with others that have the same behavior. We want them to be able to plug in a new component that differs, in a way that matters, in how it accomplishes its responsibilites. And we want this change to affect the behavior of the system as a whole in a predictable fashion.

Not surprisingly, the goal that replacing a component should be able to change the behavior of the system, rather than strictly adding to it, makes the specification problem more complex. It isn't sufficient just to specify what the replaceable components are; that doesn't say enough. And it isn't acceptable to specify the exact behavior replacements must have; that doesn't allow room for alternative components. We must specify, rather, the responsibility each module has to, and its effects on, the system as a whole; this is what we think of as protocol, it is what allows the user to actually plug in a new one and know what will happen.

Recall the `draw` method for `text-button`, whose specification says that it draws the button's text in 14 point Helvetica. This very complete specification is appropriate because it documents the behavior of an actual module rather than a role. The specification of a role, on the other hand, is more general, it gives the user the information they need to provide alternative modules for that role.

First, a role's specification must specify the range of acceptable behavior for the modules that can occupy it. A method for the `draw` generic function, for example, can display any image it wants, but shouldn't do something entirely different, like rearrange the windows on the screen. Furthermore, methods on `draw` and `geometry` must agree on the space taken up by the button's image.

The broader a range of behaviors is allowed, the more variation the system as a whole can exhibit, but the harder the implementation task becomes. Functional protocols are one restriction on the range of behaviors that attempt to give the implementation more options without overly constraining what can be accomplished with the system.

In addition, a role's specification must say how variations in the behavior of alternative modules will affect the behavior of the whole system, so that the user knows which modules to change to change system behavior. For example, the original protocol for the button system insists that the image of the button be completely determined by the behavior of the `draw` generic function, so that the user who wants to alter the image knows that all they have to do is define a new method on `draw`.

Specification at this level is describing the architecture of the system, rather than its particular behavior. It documents how the design functions as a framework for the interactions of the modules that make it up. Since it is addressing something more general than any particular concrete behavior, it is a correspondly more challenging problem, with a correspondingly higher payoff.

7 Related Work

Advice on object oriented design[Boo91, Str91, Mey88], tends to focus on what the objects are, and how are they are organized. This paper has focused on the subsequent problem of how to design and document the internal interfaces that support extensible systems.

Of course, class libraries have been documented before. MacApp[App] and NIHlib[GOP90] are two notable examples. But the norm has been to document a particular implementation, and the emphasis has been on reusability through generality rather than extensibility. There hasn't been an emphasis on the particular focus of this paper: how to design and specify an extensible library in a way that maintains the kind of leeway that allows multiple implementations.

Several systems, like Smalltalk[GR83] and Flavors[Moo86] have included tools to extract information such as which methods call which generic functions. Some have suggested that these facilities can be used to provide specification of protocols. But these are basically cross reference utilities; they only describe the code for a particular implementation. There is no way, in general, to determine what information reflects protocol principles intended for all implementations and what is idiosyncratic to the implementation at hand.

This paper has presented some precise, but informal, specification techniques. The literature includes formal mechanisms for some aspects of the task. The Eiffel language[Mey88], for example, provides facilities for asserting class invariants. There has been work on type systems[CHC90, AC91] that take account of the extensible nature of object oriented programs. But many of the issues we present, especially in section 4, aren't well addressed by the current formal techniques. The fact that we are specifying extensible, open systems rather than fixed, closed systems may mean that formal techniques will require significant new support from our formal foundations.

8 Summary

We have focused on the issues that arise when trying to design and specify an extensible class library. These specifications divide naturally into two pieces: inheritance relationships on the one hand, and protocol on the other. In each case, we have shown that a viable approach is to start with a simple specification, that provides the user with power at a cost in implementor freedom, and then relax that specification, as appropriate, to recover needed aspects of that freedom.

Specifying extensible class libraries requires us to say more about their internal structure than would be typical in a traditional library. The challenge is to find a way to say enough about the internal module structure—inheritance and protocol—to enable user extensibility, without saying so much that the implementor is overconstrained. Seen in this light, the problems that arise are not particular to object-oriented programming, they will come up in any scenario in which we attempt to provide the user with the ability to incrementally extend a default implementation.

Acknowledgements

We would like to thank the entire CLOS Metaobject Protocol community for their help and support in that project, and for their feedback which led to the development of the material presented in this paper. We would also like to thank Danny Bobrow, Jim des Rivières, Peter Deutsch, Mike Dixon, Roger Hayes, Luis Rodriguez, Erik Ruf and Amin Vahdat for their comments on earlier drafts of this paper.

References

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, 1991.

[App] Apple Computer. *MacApp 2.0 General Reference.*

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *Sigplan Notices*, 23(Special Issue), September 1988.

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[Boo91] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[GOP90] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

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

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KR90] Gregor J. Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 99–105, 1990.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Moo86] David A. Moon. Object-oriented programming with flavors. In *Object Oriented Programming Systems, Languages, and Applications*, pages 1–8, 1986.

[Str91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley, 1991.