Appears in OOPLSA'93 Proceedings.

Typing the Specialization Interface

John Lamping
Xerox PARC*

Abstract

Current typing systems for object oriented programming leave out crucial information about the specialization interface, the interface between a class and the subclasses that specialize it. In particular, the internal interdependencies among a class's methods, which profoundly affect the consequences of overriding, are not described. We show how typing systems can be extended to describe this information. This technique of extending typing is independent of any particular type system. It also suggests a new relationship among the client interface, the specialization interface, and inheritance.

1 Introduction

A class in object oriented programming exports two distinct interfaces: a client interface to users of objects of the class, and a specialization interface to specializers of the class, those who use the inheritance mechanism to build subclasses of it. While the basic operation at the client interface is message sending, the basic operations at the specialization interface are extending and overriding. Put differently, while the emphasis of the client interface is on invoking the features of objects, the emphasis of the specialization interface is on creating variant classes, which may have altered client interfaces and altered implementations. The availability of both kinds of interface in object oriented programming enhances the opportunities for software reuse.

Type systems for object oriented programming have focused on the needs of the client interface. The types say what functionality is available at the client interface, what messages can be sent, but they say nothing about the internal structure of the class's implementation. This is in accordance with the information hiding principles that motivate abstract data types; the client interface is supposed to hide the internal structure of the class, and clients of a class should not know or depend on what is none of their business.

But object oriented programming goes beyond abstract data types because it has specialization. An important difference is that while clients of a class see an atomic object that is accessed only via messages, specializers who override methods are necessarily interacting with the class's internal structure. Given this difference, it is not surprising that clients of the specialization interface are properly concerned with additional kinds of information.

For example, consider a set class that includes an add method. All that a client of set cares about add is what its arguments are and how it affects the set. A specializer of the set class who is considering using the specialization interface to override the add method needs to know more because the consequences of the change depend on the internal structure of the set class; other methods might call the add method, for example, so that changing it might change how the class responds to a number of messages. In short, the specialization interface should inform the specializer of a class about

*3333 Coyote Hill Rd., Palo Alto CA 94304;
Lamping@parc.xerox.com.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1993 ACM

how its methods combine to produce its behavior, so the specializer can understand the consequences of overriding methods. Kiczales and Lamping[8] present a more extensive discussion of the kinds of information that specializers need.

This paper points out that much of that information involves how the methods of a class depend on one another, as in the set example. We show that this dependency information can be captured by the type system if one technical step is taken: if coverage of typing is extended to include not only the argument and result types of each message, but also what other facilities of the class, what other messages and state, may be used in executing it. This description of what other features of a class are relied on to implement a given feature exposes aspects of the internal structure of a class that are of vital interest to specializers without exposing details that should properly remain hidden.

Most typing systems can be seen as serving two roles: as automatically verifiable documentation, and as checks to preclude type errors. This extension is primarily aimed at the documentation role; the extended type declarations can be automatically verified to guarantee that the program is as the typing advertises. But the extension can also have a checking role. We show that, with the extension in place, subclasses can be allowed to alter some aspects of their specialization interface while preserving safety; the extended typing can be used to check that the alterations are the safe kind, forbidding those that might lead to type errors.

The extension propsed here is not specific to any particular type system; it should be applicable to any type system for object oriented programming. In particular, it is applicable both to static and dynamic type checking, although the emphasis in this paper will be on static checking. Also, it is worth pointing out that this proposal is addressing a different issue from that addressed by the efforts to capture object oriented programming in typed calculi, such as [2, 3, 6, 10], although both are involved with inheritance. Those efforts investigate the fundamental features of object oriented programming, such as inheritance and encapsulation, by exploring how to express them in terms of elegant typed calculi; their emphasis is on the formal properties of object oriented languages. The focus of this paper is on how to provide more information about classes; it advocates and investigates incorporating an additional kind of information into type systems, information which happens to be particularly relevant to specializers.

This paper does a three-step between the specialization interface and extended typing. The next section looks in more detail at the specialization interface and at the kinds of information that specializers need to know. Section 3 shows how to provide some of this information by extending typing to include the resources utilized in executing a message. It then gives examples of the kind of information this can convey. This information is mostly relevant to the specialization interface, but some can also be relevant to the client interface. Section 4 returns to the interfaces to discuss the consequences of the extension on the client and specialization interfaces. It addresses an issue raised by the extended typing: how to control the visibility of the new information at each interface; and it notes an additional opportunity: the extension permits a new, more flexible, rule for what properties of a class can be safely altered by a subclass.

2 The Specialization Interface

As alluded to in the introduction, the client interface and the specialization interface have different purposes and support different kinds of interaction. The client interface is used to access the functionality of objects. The interface is accessed by sending messages to objects, and the internal structure of the objects shouldn't be evident. The specialization interface, on the other hand is used to extend and modify classes. That is, it operates on classes, not objects[1]. It is accessed by making a subclass and adding messages and methods, refining types, or overriding methods. The last of these, overriding methods, is the operation that can modify behavior and that can interact with the internal structure of the class. It is what motivates the desire to present the specializer with more type information about the specialization interface.

[1] Because we want to focus on the difference between the client interface and the specialization interface, we don't directly consider prototype based systems in this paper, since they don't draw as strong a distinction.

```
class set
feature
    add (new_object: any) is do ... end;
    add_all (new_objects: set) is do ... end;
    remove (old_object: any) is do ... end;
    remove_if_present (old_object: any): boolean is do ... end;
    for_each (action: action) is do ... end
end
```

Figure 1: Part of the class set

To survey some of the information about the specialization interface that the specializer might want to be informed of, consider a class, set, modeled on Smalltalk's[7]. Figure 1 presents a partial description of it, written in the syntax of Eiffel[9]. We use Eiffel in the spirit of ecumenicalism, and because we will use its syntax to express type information.[2] The message names are almost self-explanatory. The add_all message is used to add all the elements of one set into another. The difference between remove and remove_if_present is that the former generates an exception if the object is not present, while the latter tells whether or not it was already present. The last message, for_each, takes an action, and iterates over each element of the set, performing the action on the element.

The figure is replete with the kinds of information that clients of set need to know, such as what messages are supported and how to invoke them. But it leaves out much of what specializers of set would like to know before overriding methods to change the internal behavior of the class. Just as a client programmer needs to know what messages may be sent, a specializer needs to know what methods may be changed and how far-reaching the consequences of a change will be. For example, a crucial piece of information about the specialization interface of set, as implemented in the Smalltalk system, is that three of the methods: add, remove_if_present, and for_each, are more fundamental than the other two, which are defined in terms of the three. The definition of remove, for example looks something like

```
remove (old_object: any) is
    local removed: boolean
    do
        removed := remove_if_present(old_object);
        check removed
    end
```

The three core methods are the only ones that have access to the hidden representation that encodes the set; the methods for add_all and remove do not depend on that representation.

A specializer can deduce several consequences from this piece of information. For example, overriding either add_all or remove will not affect the response to any other message or the performance of any other message. Another consequence is that if the specializer wants to use a different hidden implementation, it is necessary and sufficient to override just the three methods add, remove_if_present, and for_each; that will cause the behavior of the rest of the methods to follow along.

While this piece of information reveals only a little about the implementation of the class, it is the crucial information about the specialization interface that a specializer needs in order to know what changes will preserve correctness and what their consequences will be. This information is not present in Figure 1, which deals only with what messages a set accepts, not with how the methods that implement them depend on each other. Other implementation organizations of the set class are possible, which would have the same client interface but would lead to different interdependencies among the methods and different requirements for the

[2] To make the presentation simpler, we will depart from Eiffel in small ways when it is convenient to do so.

specializer.

The moral of this example is that a well designed class has an organization to how its methods rely on each other and on properties of the class. This is what makes method overriding a useful tool; it is what allows overriding a method to have consistent, predictable, consequences. Our goal is for the library designer to be able to communicate these dependencies to specializers via the type system.

3 Typing Dependencies

3.1 Describing dependencies

From the perspective of an individual method, the moral means that many of a class's methods do not rely on all the facilities, all the messages and state, provided by the class. We can express this in terms of type checking: for many of the methods of a class, it would be type correct if we pretended during the type-checking of their bodies that their class had poorer facilities than it really does. Thus, the remove method of the class set will type-check even if we pretend that set supports only the add, remove_if_present, and for_each messages; the method does not depend on the hidden representation of a set or on the other messages. (In fact, the actual method only relies on the remove_if_present message.)

This suggests extending the application of the type system so that classes can optionally indicate how much of their state and functionality is relied on by each of their methods. By expressing this information via the type system, it can be a readily visible part of the specialization interface, and can be automatically verified.

To make this idea more precise, we need to first distinguish the notions of class, protocol, and type. Of the three, the idea of protocol is arguably the most central. A protocol is a description of what is available at an interface. A protocol need not be completely formal, but it must be complete enough to tell how to use the interface correctly. Thus a protocol not only indicates what messages are supported by an interface, but also information like possible interactions among messages or restrictions on when certain messages should be invoked. Part of the protocol for the client interface to set, for example, would indicate that remove must not be called unless the item to be removed is in the set; part of the protocol for the specialization interface would indicate that overriding the remove method will not affect the response to any other message.

The other two concepts can be described relative to protocols: Classes tell how to implement objects that support protocols. Types describe formally but incompletely the protocols supported by objects. The key point about classes is that they are involved with implementation; they address what methods to run upon receipt of messages and what state an object must maintain. Several classes may implement the same protocol, and several protocols may be supported by a single class, since if a class satisfies one protocol, it also satisfies all weaker protocols. The key point about types is that they are formal. In general, the formal limitations of any given type system will mean that it can't fully capture all the nuances of protocols. Thus types capture only an approximation of protocols. The goal of this paper is to show how type systems can be extended to capture more of the information in protocols of the specialization interface.

Now, we can state the idea a little more precisely: the method implementing a message of a class may rely on a weaker protocol than the full protocol supported by the class. We will make it possible to declare the type corresponding to this weaker protocol in the type system and have it be checked. This provides valuable information to specializers.

Figure 2 shows how this might look. There are a lot of classes in the figure, but the first two classes are used just to declare protocols and their corresponding types. Classes have to be used since Eiffel, like most object-oriented programming, provides no separate syntax for protocols or types. Two of the classes, set_core and set_rep, are only relevant to the specialization interface. The former describes the three core messages, and the latter describes the hidden information relied on by the methods for the core messages. The class set describes the full set protocol. The definition of the class default-set refers to these classes to indicate what each of its methods relies on. It uses an optional indication, [*type*], which can appear immediately after a method name to indicate that the method only relies on those facilities of

```
class set_core  -- Protocol
feature
    add (new_object: any) is deferred end;
    remove_if_present (old_object: any): boolean is deferred end;
    for_each (action: action) is deferred end
end

class set  -- Protocol
inherit set_core
feature
    add_all (new_objects: set) is deferred end;
    remove (old_object: any) is deferred end;
end

class set_rep  -- Implementation of set_core
feature
    hidden_representation: ...
end

class default_set  -- Default class to make sets from
inherit set; set_rep
feature
    add [set_rep](new_object: any) is do ... end;
    add_all [set_core](new_objects: set) is do ... end;
    remove [set_core](old_object: any) is do ... end;
    remove_if_present [set_rep](old_object: any): boolean is do ... end;
    for_each [set_rep](action: action) is do ... end
end
```

Figure 2: Typing the methods' dependencies

the object it is invoked on that are in the type[3] (named by a class, since there is no separate way of naming types). The specified type must be one that is supported by the class on which the method is defined. In a statically typed system, the method will be checked to make sure it only uses the declared facilities; in a dynamically typed system, the method would be restricted, at run time, to use only the declared facilities.[4].

The declarations in the figure say that the methods for add_all and remove depend only on the facilities of set_core. Since, in particular, they do not depend on any hidden representation information, the specializer can be assured that if the core methods are overridden in a subclass, the pre-existing methods for add_all and remove will work correctly with the new core. Similarly, the declarations say that the three core methods depend only on the hidden representation, not on each other or on any other methods of set. Among other things, this tells the specializer which methods will have to be overridden to change the hidden representation. Taken together, the declarations indicate that no methods depend on either add_all or remove, so a specializer knows that overriding either of those methods will affect only the corresponding messages.

Stepping back a bit, the reason to recognize that the protocol relied upon by a method may not be the full protocol supported by the class to which the method is attached is that the method might become attached to other classes as well. That is, the method might be inherited to various subclasses, through the inheritance facility that is the basis of specialization. But because specialization allows overriding, the other methods on the subclasses might be different. Those methods that are carried over from the superclass find themselves invoking different methods in response to the messages they send. To understand how they will function in the new class it is important to know how they depend on their class, that is, how they depend on the results of messages they send. While the exact details of how a class will affect a method may be very complex, the idea in this paper is that it is easy to formally describe which part of a method's class it is sensitive to.

3.2 Some related ideas

One language that comes close to this proposal is Modula-3[4], where methods are initially defined as ordinary procedures. Procedures are then attached to objects as message handlers, so that invoking a message on the object will call the attached procedure passing the object as the first argument. The first argument of a procedure can be typed to reflect just the protocol that the procedure relies on, making it clear when a procedure doesn't require the full protocol of the objects for which it serves as a method.

Modula-3 differs from this proposal, however, in that object types don't record information about what protocols their methods rely on. That is, when a procedure is attached to an object as a method in Modula-3, there is a check that the object supports the protocol relied on by the procedure. But the information about the exact protocol required by the procedure is not then captured in the type of the resulting object. It is not available to specializers. The type of the object hides the protocol required by its methods, just as the object hides the procedured that are its methods. The idea of this proposal, in contrast, is that a class should be able to record dependency information information in its type, because while method bodies should be hidden, specializers of the class will want to know what protocols the methods of the class rely on.

Another possibility to consider is whether information about method dependencies could be expressed just in terms of inheritance structure, without needing to introduce an extra declaration as was done in Figure 2. The obvious idea would be to just define a method directly on a class that has exactly the protocol that it depends on, which does make it clear that it depends

[3] This would be more complicated in a language that supported multi-methods. The obvious idea would be to have an optional dependency declaration for each argument the method dispatches on. That could capture a multi-method's dependency on other uni-methods. But a multi-method might rely on other multi-methods, as well, and the problem of expressing that gets into the more general problem of typing in the presence of multi-methods, which is well beyond the aspirations of this paper.

[4] For purposes of this check, it is acceptable to invoke completely hidden messages whose methods cannot be overridden (the private messages of C++) even if they are not listed in the dependencies, provided that the resources used by those messages are acceptable. Put differently, calls to hidden messages whose methods cannot be overridden are treated as if the methods were in-lined.

```
class set_core
feature
    add (new_object: any) is deferred end;
    remove_if_present (old_object: any): boolean is deferred end;
    for_each (action: action) is deferred end;
    add_all (new_objects: set) is do ... end;
    remove (old_object: any) is do ... end
end
```

Figure 3: Trying to avoid the declarations

```
class set_general_methods
inherit set_core
feature
    add_all (new_objects: set) is do ... end;
    remove (old_object: any) is do ... end
end
```

Figure 4: Trying to avoid the declarations, take II

only on that protocol. Then the method could be inherited onto other classes where it was wanted. This trick should be suspect however, because what protocol a method relies on and what classes a method belongs on are two different things, and trying to equate them should be expected to cause trouble.

To see what kind of trouble, imagine what the set example would look like. Under the trick, remove and add_all would be defined on set_core, as shown in figure 3, to indicate that they just depended on the core interface. But now set_core lists all the methods, it no longer indicates which constitute the core. So this idea doesn't work; set_core must be left alone. But perhaps we could define an immediate descendant of set_core to hold the two messages, as shown in figure 4. This still doesn't get it right, because it suggests that remove and add_all might depend on each other. The problem is that when a method is moved up the inheritance hierarchy it contaminates the class it is moved to, making it appear that the other methods of that class might depend on the moved method. The only way to avoid this would be to give every method its own class, which is grossly awkward, raises problems with method overriding, and turns out to be unable to express some of the more subtle uses of typed method dependencies, which will be seen later. In conclusion, the protocol required by a method is fundamentally different from the classes the method should be defined on, and they can't be conflated.

3.3 What can be expressed

Returning to explicit dependency declarations, one of the internal class organizations they can express is that of layered protocols, where the default behavior of one message is determined, in part, by the behavior of one or more hidden messages. Figure 5 presents an example, using part of the declarations for a hypothetical button library. A client of button calls draw to have the button display itself on the screen. A specializer of the default button class can intervene at either of two layers of the internal method hierarchy to affect draw's behavior: at the finer layer, overriding the label_font and border_width methods to alter the behavior of the default draw method, which calls them; or, at the coarser layer, overriding the draw method, itself, if that is the only way to achieve some desired behavior. The advantage of a layered protocol, from a specializer's point of view is that there is some range of behavior that can be obtained very simply by intervening at the finer layer,

```
class button
feature
    draw is deferred end;
    handle_click is deferred end
end

class draw_sub_protocol
feature
    label_font [any]: font is deferred end;
    border_width [any]: integer is deferred end
end

class default_button
inherit button; draw_sub_protocol
feature
    draw [draw_sub_protocol] is do ... end;
    handle_click [button] is do ... end;
    label_font [any]: font is do ... end;
    border_width [any]: integer is do ... end
end
```

Figure 5: Part of a button library

while a broader range of behavior is available at the cost of doing more work by intervening at the coarser layer.

Layering is easy to characterize in terms of typing dependencies, because the job of the finer layer is to expose some of the internal workings of the methods whose behavior make up the coarser layer; the finer layer is sort of a sub-protocol. This means that the methods of the coarser layer of a protocol depend on the finer layer, but no other methods, such as handle_click, are allowed to directly depend on the finer layer. This is manifested in the type declarations in the figure; only draw depends on draw_sub_protocol. A specializer knows that changes to label_font and border_width will only affect draw and other methods that depend on draw.

The figure also illustrates another important class organization tool, the functional protocol. A functional protocol is one where the result of sending a given message does not depend on any internal state of the object; it only depends on the class of the object and the arguments to the message. The advantage of a functional protocol, from the point of view of a caller, is that the caller knows that the answer will only change if the arguments do, and so computations based on the answer can be recorded and reused without worrying about their becoming invalid.

In the figure, both label_font and border_width are declared to follow a functional protocol; they depend only on any, the most general class, so they cannot depend on any internal state particular to a button, only its immutable class. This means, for example, that the default method for draw can build a bit image of the button the first time it is called, and reuse that image on subsequent calls, confident that the button's font and border width will not change. The checking for functional protocols isn't perfect. Since dependency declarations only constrain the use of the facilities of the object on which the method is invoked, it is possible to write a method for label_font or border_width that depends on some other, global, mutable state. Checking for that is beyond the scope of this proposal.

There is another, subtle, declaration in this example: the draw method declares that it depends on draw_sub_protocol, and thus depends on label_font and border_width following functional protocols. That is, the draw method depends on label_font and border_width not depending on anything. This can't be checked. It is saying that the code for draw in some way makes use of the fact that label_font and border_width follow functional protocols, perhaps by caching a bit image of the button. This kind of dependency is too subtle for type checkers. So the situation is that declarations that a method does not depend on certain messages can be checked easily, but a declaration that a method depends on another method not depending can't be checked; the type system must accept the user's word. This isn't a severe problem. The effect of declaring that draw depends on label_font and border_width following functional protocols is to insist that if either of them is overridden in any class that inherits the draw method, the new method for label_font or border_width must still follow a functional protocol. The consequence of failing to declare the draw method's dependency on the functional protocols would be loss of this checking; type safety is not in jeopardy. The next section will go into this in more detail.

So far, method dependency typing has been used to describe which facilities methods depend on, but not how they depend on them, just as the typing of a function tells what arguments it expects, but not what it does with them. In both cases, the type information is partial, but valuable, and can be easily verified. Some languages have gone beyond this by supporting assertions, which can give more precise information, even if they can't be verified easily. Typing methods' dependencies allows the specializer to take better advantage of any assertional information. For example, suppose an assertion were added to the remove method from figure 2 to say that after you remove an object from a set, it's not there any more:

```
remove [set_core](old_object: any) is
        do ...
        ensure not includes(old_object)
        end
```

This is certainly pertinent information to a client of the class set. The question is what does it tell someone specializing set. The answer comes from the [set_core] declaration, which says that the only facilities and assertions of set that the method's correctness depends only on are those of set_core. This tells the specializer that, even if other methods are overridden, the remove method will satisfy its assertion as long as the functionality and assertions of set_core are preserved.

On the other hand, if remove was defined, for example, in terms of the hidden representation, then the [set_core] in the declaration would have to be replaced by a [set_rep], and the specializer would know that the method for remove would have to be overridden in any subclass that changed the internal representation of sets, and the assertion re-established. This later case, where the specializer is charged with maintaining a relationship between messages, has been called "consistent generic functions"[8].

4 Distinguishing the Interfaces

While we have been talking in terms of two different interfaces, a client interface for users of instances of a class, and an specialization interface for specializers of the class, we have thus far skirted the question of which aspects of a class specification apply to which interface and how the information should behave under inheritance. It is time to address it. It will turn out that the dependency information will actually present an opportunity to losen up the inheritance rules a bit without sacrificing the ability to detect type errors.

Traditionally, the division of facilities among interfaces has been on a per-message basis. Some messages are exported to the client interface, others are available only to the specialization interface (the protected messages in C++), and others may only be available to methods on the same class (the private messages in C++).

But in the context of dependency information, some information about a message might belong in one interface and other information in another. For example, it is commonly the case that some message should appear in the client interface, but its dependency information should be reserved for the specialization interface.

This might suggest that the dependency information should be visible only in the specialization interface. That is one feasible design choice. But there are cases where it is advantagous to include some dependency information in the client interface. An example is when a client interface would like to specify that the behavior of some message will follow a functional protocol, so that clients can cache the results. The natural way to express this, as discussed earlier, is telling the client that a method implementing the message is not allowed to depend on anything besides its arguments.

This suggests the more flexible idea of allowing a class declaration the freedom to specify how much of its type information it wants to guarantee to its clients. Syntactically, this could be achieved by a range of mechanisms: a completely separate client interface specification, optional additional type declarations in export clauses, or export annotations interspersed throughout a class definition. The same choices are available for how much information a class wants to provide to its specialization interface.

Figure 6 shows one way some of the declarations for a button might look. Like a typical abstract class, button exports all its declarations to both interfaces. In particular, clients of button know that size obeys a functional protocol; the size of a button will never change. The draw_sub_protocol class is sort of an abstract class for just the specialization interface; it exports all its declarations to its descendants, but provides no client interface. Finally, default_button exports a client interface equivalent to button's, and exports almost all of its dependency information to the specialization interface. The only exception is that the implementation of handle_click is declared not to depend on any of the features of default_button, but that information is not exported to descendants.

As the example illustrates, there end up being three typings associated with a class: a most specific typing, which is what methods in the class can rely on; a less specific typing, which is available at the specialization interface; and a still less specific typing exported to the client interface. The notion of specificity used here is the usual one based on conformance[2, 5]: a more specific type conforms to a less specific type.[5] This is a natural extension of the usual notions of hidden and protected information into a context of richer type information.

Another important question is which, if any, of these three types are expected to be preserved under inheritance. Currently, statically typed systems tend to require subclasses to conform both to their superclass's client interfaces, and to their superclass's specialization interfaces.[6] Preserving the client interface guarantees substitutability: an instance of a subclass will always will be able to provide clients with the facilities they expect of an instance of the class. Preserving the specialization interface guarantees method type correctness; when a method is inherited it will always find itself in a class that has the facilities it expects.

But trying to preserve the full specialization interface under inheritance becomes awkward when method dependencies can be typed, because it would mean that it is impossible to document dependency information for a method without also imposing that requirement on any overriding methods. Recall, for example, the typing of add_all, which depended only on the facilities of set_core. That information would be exported to the specialization interface, but not to the client interface. Now, it might make sense to make a subclass of default_set that overrode add_all to make it more efficient by directly manipulating the hidden representation. That would mean that the new method for add_all would depend on more information, and thus the subclass would not conform to the specialization interface for default_set.

This suggests freeing the inheritance mechanism from strict adherence to preserving the specialization interface. Fortunately, this can be done without sacrificing method type correctness. This is possible because inheritance is done on classes, not on objects.[7] Thus, unlike the situation of invoking an object through the client interface, where the exact class of the object is not known until run time, when inheritance is done, the

[5] Since it corresponds, in a sense, to a typing for an extra self argument, the method dependency declaration acts contravariantly, that is, relying on less information conforms to (is at least as specific as) relying on more. Precisely, a type A is at least as specific as a type B if and only if A is at least as specific as B when dependency information is ignored, and for each dependency specified for a method for B, there is a dependency specified for that method for A that is the same or less specific.

[6] As pointed out by Cook[6] the situation is actually more subtle, but the subtleties Cook brings out won't matter for this discussion.

[7] For languages based on prototyping, the test presented here might have to be done dynamically at sub-object creation.

```
class button
export_clients all
export_descendants all
feature
    draw is deferred end;
    handle_click is deferred end;
    size [any]: integer is deferred end
end

class draw_sub_protocol
export_descendants all
feature
    label_font [any]: font is deferred end;
    border_width [any]: integer is deferred end
end

class default_button
inherit button; draw_sub_protocol
export_clients draw, handle_click, size[any]
export_descendants
    draw[draw_sub_protocol], handle_click, size[any],
    label_font[any], border_width[any]
feature
    draw [draw_sub_protocol] is do ... end;
    size [any]: integer is once result := 100 end;
    handle_click [any] is do ... end;
    label_font [any]: font is do ... end;
    border_width [any]: integer is do ... end
end
```

Figure 6: The different interfaces to a button

exact class being inherited from is known at the point of the inheritance, because inheritance is done on classes. This means that the type correctness of the methods inherited into the new class can simply be verified directly. Each inherited method is checked to make sure that the new class, which may not completely follow the specialization interface protocol of the super-class, still adheres to that protocol relied on by the inherited method. The net effect is that subclasses are only required to preserve that part of the specialization interface protocol of their superclasses that is relied on by inherited methods. This is particularly useful when making a subclass of a class with a layered protocol; the subclass is free to violate some layers of the protocol, as long as it overrides all methods that depend on those layers.

In principle, this test would not require the ability to type method dependencies, since the source of each inherited method could be made available and its type correctness in the new class checked directly. But the ability to type method dependencies makes the test practical, because it means that both humans and machines can check the legality of a subclass from just the type information. There is no need to resort to the source of inherited methods, since the type of the class being inherited from can contain, in its dependency declarations, exactly the necessary information. In terms of dependency typing, the rule is that a subclass must be at least as specific as each of the types relied on by methods inherited from a superclass.[8]. This rule does rely on the necessary dependency information actually being declared or deduced; if no dependency information is recorded for an inherited method, then the inherited method must be presumed to depend on the entire specialization interface protocol, which the subclass must then be forbidden to override.

[8]The rule also makes sense for languages where the client type hierarchy and the inheritance hierarchy are decoupled, as proposed by various authors [12, 1, 11]. The separate client type hierarchy guarantees substitutability to clients, while the rule guarantees type correctness for inherited methods.

5 Conclusion

Object oriented programming supports reuse two ways: a class can be reused by clients who make instances and by specializers who make subclasses. In both cases, it is vital to document the interfaces, so that the implementor of the class and the user of the class can agree on what commitments they can rely on and what freedoms they retain. Type systems have been used successfully to formalize some of this information for the client interface, but the specialization interface has not been as well served. In particular, typing has not told the specializer anything about how different methods depend on each other, which is what the specializer must know to understand the consequences of overriding a method. This can be addressed by extending the application of type systems so that they can record an indication of what parts of a class's protocol each method relies on. This documents to the specializer what a method depends on. This also allows checking to see that a subclass is a coherent extension of a superclass even if it may have altered some of the specialization interface. In short, it brings more of the advantages of type systems to the specialization interface.

6 Acknowledgements

This work was provoked by ideas arising out of several onging projects in the Embedded Computation Area at Xerox PARC. Martin Abadi, Jim des Rivières, Mike Dixon, David Espinosa, Gregor Kiczales, Chris Maeda, Dylan McNamee, Anurag Mendhekar, Gail Murphy, Andreas Paepke, and Eric Ruf helped clarify the concepts in this paper and provided comments on earlier drafts.

References

[1] AMERICA, P., AND VAN DER LINDEN, F. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications* (1990).

[2] CARDELLI, L. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds., no. 173 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[3] CARDELLI, L. Extensible records in a pure calculus of subtyping. Tech. Rep. 81, DEC System Research Center, January 1992.

[4] CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KASLOW, B., AND NELSON, G. Modula-3 language definition. *ACM SigPlan Notices 27*, 8 (August 1992).

[5] COOK, W. R. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications* (1992).

[6] COOK, W. R., HILL, W. L., AND CANNING, P. S. Inheritance is not subtyping. In *Seventeenth ACM Symposium on Principles of Programming Languages* (1990), pp. 125–135.

[7] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[8] KICZALES, G., AND LAMPING, J. Issues in the design and documentation of class libraries. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications* (1992).

[9] MEYER, B. *Object-oriented Software Construction*. Prentice Hall, 1988.

[10] PIERCE, B. C., AND TURNER, D. N. Object-oriented programming without recursive types. In *Proceedings of the 20th ACM Conference on the Principles of Programming Languages* (1993).

[11] PORTER, III, H. H. Separating the subtype hierarchy from the inheritance of implementation. *Journal of Object-Oriented Programming 4*, 9 (February 1992).

[12] SNYDER, A. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications* (1986).