Wrappers to the Rescue

John Brant
Ralph E. Johnson
Donald Roberts
Brian Foote

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{brant, johnson, droberts, foote}@cs.uiuc.edu

Abstract

When an object-oriented language is itself built out of first-class objects, programmers may change and extend these objects as the need arises. One such language is Smalltalk. This paper focuses on how Smalltalk's reflective facilities can be used to "wrap" before- and after-behavior around calls to existing methods, and quantifies the relative performance of several ways of doing this. We then, in turn, show how wrappers have proven indispensable during the construction of a coverage tool, a class collaboration tool, and an interaction diagramming tool. We've also used wrappers to construct synchronized methods, assertions, and multimethods, where they proved to be equally valuable. The relative ease with which wrappers allowed us to build these analysis tools and linguistic extensions stands in contrast to the rather draconian measures one must take to achieve similar results in languages devoid of support for wrappers.

1. Introduction

One benefit of building programming languages out of objects is that programmers have a place where they can go when they want to change the way a running program works. Languages like Smalltalk and CLOS, which represent program objects like Classes and Methods as objects that can themselves be manipulated at runtime allow programmers to make permanent, or temporary, changes to the ways these objects work when the need arises.

This paper focuses on how to intercept and augment the behavior of existing methods in one such language: Smalltalk. Several approaches are examined and contrasted and their relative performances are compared. These are:

1. Source Code Modifications
2. Byte Code Modifications
3. New Selectors
4. Dispatching Wrappers
5. Class Wrappers
6. Instance Wrappers
7. Method Wrappers

We then examine several tools and extensions we've built using wrappers:

1. Coverage Tool
2. Class Collaboration Diagram Tool
3. Interaction Diagram Tool
4. Synchronized Methods
5. Assertions
6. Multimethods

Taken one at a time, it might be easy to dismiss these as Smalltalk specific minutiae, or as language specific hacks. However, taken together, we think they help illustrate the power and importance of the underlying reflective facilities that support them.

Before and after methods first appeared in Flavors [MW81] and Loops [BS83]. The Common Lisp Object System (CLOS) [BDG+88] provides a powerful method combination facility that includes before and after methods. In CLOS, a method with a `:before` qualifier that specializes a generic function, *g*, is executed before any of the primary methods on *g*. Thus, the before methods are called before the primary method is called, and the after methods are called afterwards. An `:around` method wraps a primary method and has the choice of calling it. The method combination mechanism built into CLOS lets programmers build their own method qualifiers and combination schemes, and are very powerful.

Unfortunately, when method combination is used badly, it can lead to programs that are complex and hard to understand. Application programmers use them to save a little code but end up with systems that are hard to maintain. The result is that before and after methods have gained a bad reputation, and few languages support them.

We use method wrappers primarily as a reflective facility, not a normal application programming technique. We think of them as a way of disciplining the underlying reflective facilities. For example, we use them for determining dynamically who calls a method, and which methods are called. If before and after methods are treated a disciplined form of reflection, then they will be used more carefully and their complexity will be less of a problem.

Our experience with before and after methods has been with Smalltalk. Smalltalk has many reflective facilities. The ability to trap messages that are not understood has been used to implement encapsulators [Pas86] and proxies in distributed systems [Ben87, McC87]. The ability to manipulate contexts has been used to implement debuggers, back-trackers [LG88], and exception handlers [HJ92]. The ability to compile code dynamically is used by the standard programming environments and makes it easy to define new code management tools. Smalltalk programmers can change what the system does when it accesses a global variable [Bec95] and can change the class of an object [HJJ93].

However, it is not possible to change every aspect of Smalltalk [FJ89]. Smalltalk is built upon a virtual machine that defines how objects are layed out, how classes work, and how messages are handled. The virtual machine can't be changed except by the Smalltalk vendors, so changes have to be made using the reflective facilities that the virtual machine provides. Thus, you can't change how message lookup works, though you can specify what happens when it fails. You can't change how a method returns, though you can use valueNowOrOnUnwindDo: to trap returns out of a method. You can't change how a method is executed, though you can change the method itself.

We use before and after methods to simulate changing how a method is executed. The most common reason for changing how a method is executed is to do something at every execution, and before and after methods work well for that purpose. For example, we have used them for determining which methods in a program get executed, for ensuring that only one process is executing a method at a time, and for checking the pre and post-conditions of a method.

This paper shows a variety of techniques for implementing before and after methods in Smalltalk-80 and describes their tradeoffs. It also describes several uses for them. It is another illustration of the merits of opening up a programming system.

2. Compiled Methods

Many of the before and after method implementations discussed in this paper are based on CompiledMethods, so it is helpful to understand how methods work to understand the different implementations.

Smalltalk represents the methods of a class using instances of CompiledMethod or one of its subclasses. A CompiledMethod knows its Smalltalk source, but it also provides a more efficient representation of a method. The virtual machine executes methods by translating them into machine code. Also, browsers use them to check senders of messages and references to variables as well as for inspecting source code.

CompiledMethod has three instance variables and a literal frame that is stored in its variable part (accessible through the at: and at:put: methods). The instance variables are *bytes*, *mclass*, and *sourceCode*. The *sourceCode* variable holds an index that is used to retrieve the source code for the method and can be changed so that different sources appear when the

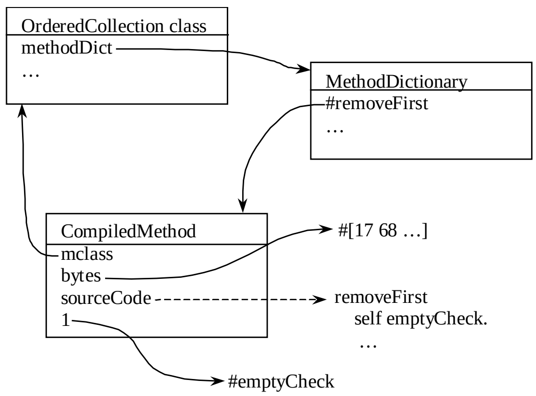


Figure 1: removeFirst method in OrderedCollection

method is browsed. Changing this variable does not affect the execution of the method, though. The *mclass* instance variable contains the class that compiled the method. One of its uses is to extract the selector for the method.

The bytes and literal frame are the most important parts of CompiledMethods. The *bytes* instance variable contains the byte codes for the method. These byte codes are stored either as a small integer (if the method is small enough) or a byte array, and contain references to items in the literal frame. The items in the literal frame include standard Smalltalk literal objects such as numbers (integers and floats), strings, arrays, symbols, and blocks (BlockClosures and CompiledBlocks for copying and full blocks). Symbols are in the literal frame to specify messages being sent. Classes are in the literal frame whenever a method sends a message to super. The class is placed into the literal frame so that the virtual machine knows where to begin method lookup. Associations are stored in the literal frame to represent global, class, and pool variables. As a result, every access of a global, class, or pool variable sends the value message to the association. Although the compiler will only store these types of objects in the literal frame, in principle any kind of object can be stored there.

Figure 1 shows the CompiledMethod for the removeFirst method in OrderedCollection. The method is stored under the #removeFirst key in OrderedCollection's method dictionary. Instead of showing the integer that is in the method's sourceCode variable, the dashed line shows the source code that the integer points to.

3. Implementing Wrappers

There are many different ways to implement before and after methods in Smalltalk, ranging from simple source code modification to complex byte code modification. In the next few sections we discuss six possible implementations and some of their properties.

3.1 Source code modification

A common way to implement before and after methods is to modify the method directly. The before and after code is directly inserted into the original method's source and the resulting code is compiled. This requires parsing the original method to determine where the before code is placed and all possible locations for the after code. Although the locations of return statements can be found by parsing, these are not the only locations where the method can be exited. Other ways to leave a method are by exceptions, non-local block returns, and process termination.

VisualWorks allows us to catch every exit of a method with the valueNowOrOnUnwindDo: method. This method evaluates the receiver block, and when this block exits, either normally or abnormally, evaluates the argument block. The new source for the method using valueNowOrOnUnwindDo: is

```
originalMethodName: argument
    "before code"
    ^["original method source"]
        valueNowOrOnUnwindDo:
            ["after code"]
```

To make the method appear unchanged, the source index of the new method can be set to the source index of the old method. Furthermore, the original method does not need to be saved since it can be recompiled from the source retrieved by the source index.

The biggest drawback of this approach is that it must compile each method that it changes. Moreover, it requires another compile to reinstall the original method. Not only is compiling slower than the other approaches listed here, it cannot be used in runtime images since they are not allowed to have the compiler.

3.2 Byte code modification

Another method modification approach is to modify the CompiledMethod directly without recompiling [MB85]. This technique inserts the byte codes and literals for the before code directly into the CompiledMethod so that the method does not need to be recompiled, thus installing faster. Unfortunately, this approach does not handle the after code well. To insert the after code, we must convert the byte codes for the original method into byte codes for a block that is executed by the valueNowOrOnUnwindDo: method. This conversion is non-trivial since the byte codes used by the method will be different than the byte codes used by the block. Furthermore, this type of transformation depends on knowing the byte code instructions used by the virtual machine. These codes are not standardized and can change without warning.

3.3 New selector

Another approach for before and after code moves the original method to a new selector and creates a new method that executes the before code, sends the new selector, and then executes the after code. Using this approach the new method is created as:

```
originalMethodName: argument
    "before code"
    ^[self newMethodName: argument]
        valueNowOrOnUnwindDo:
            ["after code"]
```

This implementation has a couple of nice properties. One is that the original methods do not need to be recompiled when they are moved to their new selectors. Since methods contain no direct reference to their selectors, they can be moved to any selector that has the same number of arguments. The other property is that the new forwarding methods with the same before and after code can be copied from another forwarding method that has the same number of arguments. The main difference between the two forwarding methods is that they send different selectors for their original methods. The symbol that is sent is easily changed by replacing it in the method's literal frame. The only other changes between the two methods are the sourceCode and the mclass variables. The mclass is set to the class that will own the method, and the sourceCode is set to the original method's sourceCode so that the source code changes aren't noticed. Since byte codes are not modified, neither the original method or the new forwarding method need to be compiled so the installation is faster than the source code modification approach.

The problem with this approach is that the new selectors are visible to the user. These new selectors cannot conflict with other selectors in the super or subclasses and should not conflict with users adding new methods. Furthermore, it is more difficult to compose two different before and after methods since we must remember which of the selectors represent the original methods and which are the new selectors. Another undesirable property is that inserting new selectors may cause the method dictionaries to grow, and since method dictionaries do not shrink, this space is effectively lost.

3.4 Dispatching Wrapper

One way to wrap new behavior around existing methods is to screen every message that is sent to an object as it is dispatched. In Smalltalk, the doesNotUnderstand: mechanism has often been recruited for this purpose [Pas86, Ben87, FJ89]. This approach has customarily been used where some action must be taken regardless of which method is being called, such as coordinating synchronization information. It could even be used, together with additional dictionaries, to orchestrate wrapping on a per-method basis. Together with lightweight classes, wrapping the dispatching mechanism can allow per-instance changes to behavior.

However, the doesNotUnderstand: mechanism is slow, and screening every message set to an object to change the behavior of a few methods has a blunderbuss quality about it. The following sections examine how Smalltalk's meta-architecture permits us to more precisely target the facilities we need.

3.5 Class Wrapper

The standard approach for specializing behavior in object-orient programming is subclassing. We can use this approach to specialize our methods to include the before and after conditions. In this case our specialized subclass would essentially wrap the original class by creating a method that would execute the before code, call the original method using super keyword, and then execute the after code. Like the methods in the new selector approach, the methods for the specialized subclass can also be copied so that the compiler is not needed.

Once the subclass has been created, it will need to be installed into the system. To install the subclass, the new class will need to be grafted into the hierarchy so that subclasses will also use the wrapped methods. It can be inserted into the hierarchy by using the superclass: method. Next, the reference to the original class in the system dictionary will need to be replaced with a reference to the subclass. Finally, all existing instances of the original class will need to be converted to use the new subclass. This can be accomplished by getting allInstances of the original class and using the changeClassToThatOf: method to change their class to the new subclass.

3.6 Instance Wrapper

This approach can also be used to give per instance changes. Instead of replacing the entry in the system dictionary, we can change the objects that we want, by using the changeClassToThatOf: only on those objects.

Like the new selector approach this only requires one additional message send, but unlike the new selector approach, it does not have the side effect of growing the method dictionary to install the before and after code. The biggest drawback of this approach is that it takes longer to install. Each class requires a scan of object memory to look for all instances of the original class. Once the instances have been found, we will need to iterate though them changing each of their classes.

3.7 Method Wrapper

A method wrapper is like a new selector, except that it does not add new entries to the method dictionary. Instead of sending a message to the new selector, this approach evaluates the original method directly by using the valueWithReceiver:arguments: method. The valueWithReceiver:arguments: method executes a method given a receiver and an array of arguments.

This approach uses a new subclass of CompiledMethod called MethodWrapper[1]. MethodWrapper adds one instance variable, *clientMethod*, that stores the original method. It also defines beforeMethod, afterMethod, and receiver:arguments: methods as well as a few helper methods. The beforeMethod and afterMethod methods contain the before and after code. The receiver:arguments: method executes the original method given the receiver and argument array.

```
receiver: anObject arguments: args
    self beforeMethod.
    ^[clientMethod
            valueWithReceiver: anObject
            arguments: args]
        valueNowOrOnUnwindDo:
            [self afterMethod]
```

[1] In VisualAge you cannot add temporaries to subclasses of CompiledMethod so MethodWrappers are a subclass of Object. A stub method is compiled which delegates to the MethodWrapper.

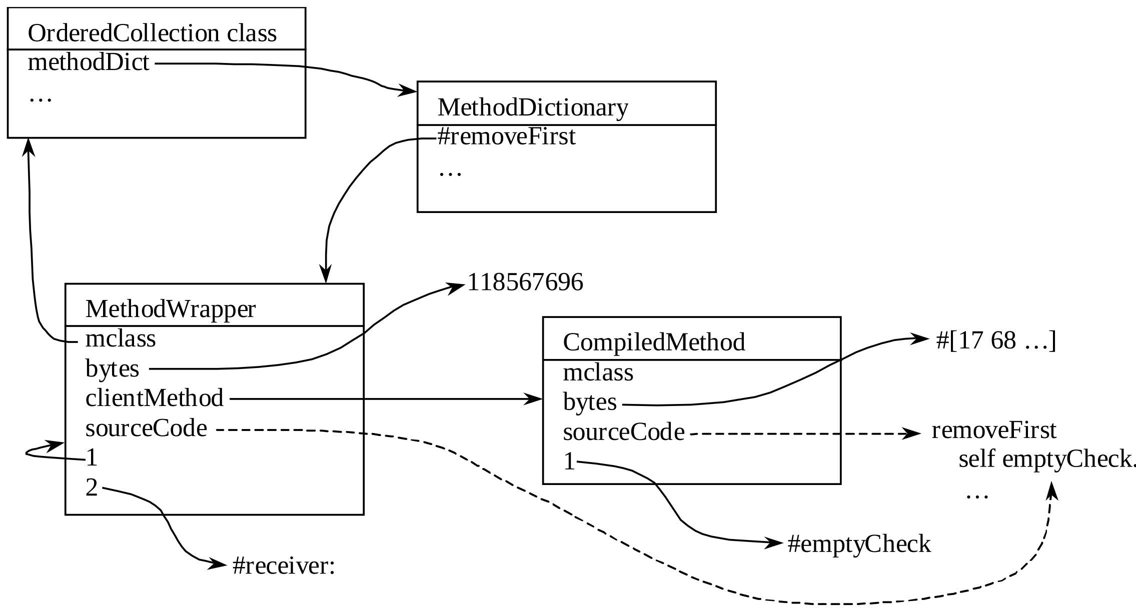


Figure 2: MethodWrapper on removeFirst method

When a MethodWrapper is executed it must pass control to its receiver:arguments: method, therefore the method must be able to refer to itself. The code "thisContext method" is one way for methods to refer to themselves, but this is inefficient since a context must be created every time the method is needed. Instead the method can be included in its own literal frame so that the code has direct access to it. Although one could modify the compiler to support a "thisMethod" keyword, a simpler approach is to compile another literal in place of the method, and then replace the literal with the method. Using this trick the receiver:value: message can be sent to the MethodWrapper by compiling

```
originalMethodName: argument
    ^#() receiver: self value: argument
```

and replacing the empty array (in the first position of the literal frame) with the method. The receiver:value: method is one of the MethodWrapper's helper methods. It is responsible for converting its value argument into an array and sending them to the receiver:arguments: method.

Figure 2 shows a MethodWrapper wrapping the removeFirst method of OrderedCollection. The CompiledMethod has been replaced by the MethodWrapper in the method dictionary. The MethodWrapper references the original method through its clientMethod variable. Also, the empty array that was initially compiled into the method has been replaced with a reference to the wrapper.

Like the new selector approach, MethodWrappers do not need to be compiled for each method. Instead they just need a prototype (with the same number of arguments) that can be copied. Once copied, the method sets its method literal, source index, mclass, and clientMethod. Since the method wrapper can directly execute the original method, no new entries are needed in the method dictionary for the original method.

Table 1 and Table 2 compare the different approaches for both runtime overhead and installation time. These tests were performed on an 486/66 with 16MB memory running Windows 95 and VisualWorks 2.0. The byte code modification approach was not implemented, thus it is not shown. The dispatching wrapper has been omitted from the installation times since it is only an instance based technique. Added to the listings is an inlined method wrapper. This new method wrapper inlines the before and after code into the wrapper without defining the additional methods. This saves four message sends over the default method wrapper. Although it helps runtime efficiency, it hurts installation times since the inlined wrappers are larger.

Approach	Number of arguments 0	1	2	3
Method modification (no returns)	5.2	5.2	9.2	9.7
Method modification (returns)	339.0	343.8	344.5	346.5
New selector	5.5	9.7	10.3	10.7
Dispatching wrapper	21.1	22.8	23.5	27.5
Class wrapper	5.9	9.8	10.5	10.9
Method wrapper	23.4	28.7	31.5	31.8
Inlined method wrapper	18.8	20.3	21.9	24.5

Table 1: Overhead per 1,000 method calls (ms)

Table 1 shows the overhead of each approach. The method modification approach has the lowest overhead if the method does not contain a return, but when it contains a return, the overhead for method modification jumps to be more than ten times greater than the other techniques. The new selector and class wrapper approaches have the best overall times. The two method wrapper approaches and the dispatching wrapper approaches have more than double the overhead as the new selector or class wrapper approaches since the method wrappers and dispatching wrappers must create arrays of their arguments.

Table 2 contains the installation times for installing the various approaches on all subclasses of Model and its metaclass (226 classes with 3,159 methods). The method wrapper techniques are the fastest since they only need to change one entry in the method dictionary. The new selector approach is slightly slower since it needs to change two entries in the method dictionary. Although the class wrapper only needs to add one entry, it must scan object memory for instances of each class to convert them to use the new subclass wrapper. Finally, the method modification approach is the slowest since it must compile every method.

Approach	Time
Method modification	262.6
New selector	25.5
Class wrapper	44.2
Method wrapper	17.0
Inlined method wrapper	19.9

Table 2: Installation times for 3,159 methods in 226 classes (sec)

4. Applications

Before and after methods can be used in many different areas. In this section we outline four different uses.

4.1 Coverage Tool (Image Stripper)

One application that can use before and after methods is an image stripper. Strippers remove unused objects (usually methods and classes) from the image to make it more memory efficient. The default stripper shipped with VisualWorks only removes the development environment (compilers, browsers, etc.) from the image.

A different approach to stripping is to see what methods are used while the program is running and remove the unused ones. Finding the used methods is a coverage problem and can be handled by method wrappers. Instead of counting how many times a method is called, the method wrapper only needs a flag to signify if its method has been called. Once the method has been called, the original method can be restored so that future calls occur at normal speeds.

We created a subclass of MethodWrapper that adds two new instance variables, *selector* and *called*. The *selector* variable contains the method's selector, and *called* is a flag that signifies if the method has been called. Since the method wrapper does not need to do anything after the method is executed, it only needs to redefine the beforeMethod method:

```
beforeMethod
    called ifFalse:
        [called := true.
        mclass addSelector: selector
            withMethod: clientMethod]
```

This method first sets its flag and then reinstalls its original method. The ifFalse: test avoids infinite recursion in case that the method is called while performing the addSelector:withMethod: operation. Execution is slow at first, but it rapidly increases once the base set of methods are reinstalled.

The method wrapper correctly reports whether it has been called. However, this stripping scheme requires 100% method coverage. Any method that is not used by the test suite will be removed, so if a test suite does not provide 100% method coverage (which they rarely do) then the stripper will remove a method that is needed later. Therefore, all methods should be saved to a file before they are removed. If one of the removed methods is called, it must be loaded, installed, and executed. Once again method wrappers can be used for the deleted methods. Instead of containing a direct reference to their wrapped methods, they contain an index which they use to load their method if it is called. In addition, instead of replacing every method with a method wrapper, additional space can be saved by creating only a few wrappers and handling most methods through the doesNotUnderstand: mechanism.

4.2 Class Collaboration

Another use of before and after methods is dynamically analyzing collaborating objects. For example, we might create call graphs that can help developers better understand how the software works. Furthermore, such information can help the developer visualize the coupling between objects. This can help the developer more quickly analyze when inappropriate objects are interacting.

This information can easily be captured using method wrappers together with some of Smalltalk other reflective facilities such as the ability

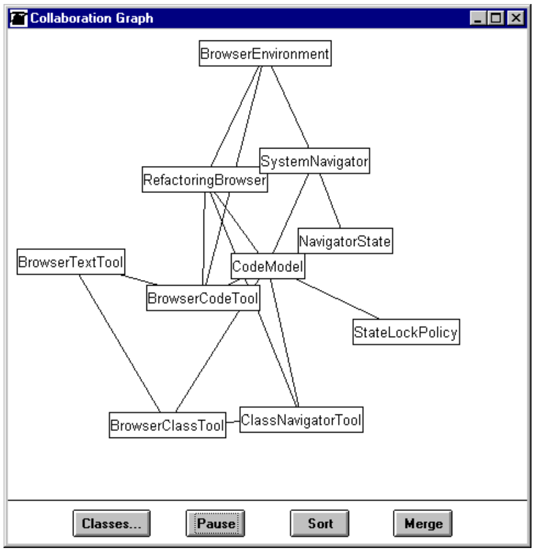


Figure 3: Class collaboration graph of the Refactoring Browser

to get the current context with the thisContext keyword. Whenever a method is called, we need to record who called the method, where the call occurred (which method and statement inside the method), the starting and ending times for the method, and finally how the method terminated (either normally with a return, or abnormally by a signal). Methods that return abnormally might be a problem since the programmers might not have programmed for such a case.

Using the information collected by the method wrappers, we can create a class collaboration graph as shown in Figure 3. Whenever one object of a class sends a message to another object in another class, a line is drawn between them. Classes whose objects collaborate a lot are attracted to each other. The collaboration graph can help the programmer see which objects are collaborating as well as how much they are collaborating.

4.3 Interaction Diagrams

Interaction diagrams illustrate the dynamic sequence of the message traffic among several objects at runtime. The InteractionDiagramApplication allows users to select the set of methods that will be watched. These methods are wrapped, and traffic through them is recorded by the tool. When the wrappers are removed, the interactions among the objects that sent and received these messages are depicted, as in Figure 4.

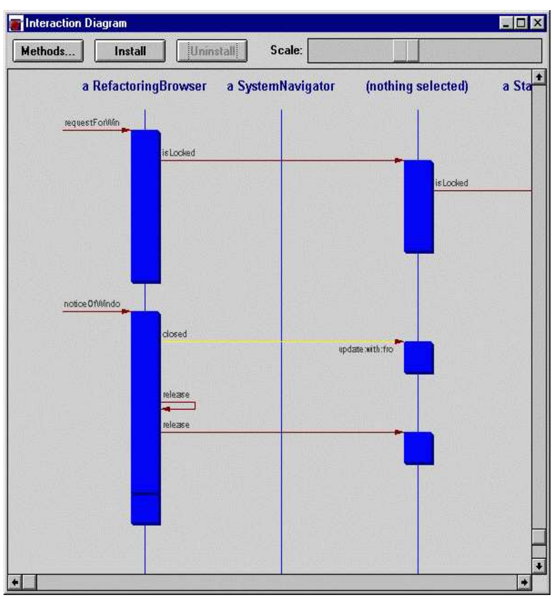
The diagrams generated by the tool are similar to the interaction diagrams seen in many books, with one notable exception. Since we only select a few methods to observe, we miss some messages. As a result, there are times when a message is received, but the last method entered did not send the message. For example, suppose you have:

```
Foo>>createBar
   ^Bar new

Bar>>initialize
   "do some initialization"

Bar class>>new
   ^super new initialize
```

and that you only wrap `Foo>>createBar` and `Bar>>initialize`. If you send a `Foo` the `createBar` message, that event will be recorded. It will send the `new` message to `Bar` class, but since that method is not wrapped, it is not observed. When the `new` method sends the `initialize` method to a `Bar`, it is observed, but the last observed method did not send it. Such events are called indirect message sends and are displayed as yellow lines. In the picture above, we can see that "a RefactoringBrowser" sent a `closed` message to some object that wasn't wrapped, which resulted in the `update:with:from:` method being called on "(nothing selected)" (a CodeModel).



Without a facility for wrapping the watched methods, tools would have to intervene at the source or binary code levels. The relative absence of such tools in languages without support for wrappers testifies to the difficulty of intervening at these levels

4.4 Synchronized Methods

Yet another example where before and after methods are useful is synchronizing methods. In a multithreaded environment, objects used concurrently by two different threads can become corrupt. A classic example in Smalltalk is the Transcript. The Transcript is a global variable that programs use to print output on. It is most often used to print debugging information. If two processes write to the Transcript at the same time, it can become corrupt and cause exceptions to be raised. To solve this problem we need to ensure that only one process accesses the Transcript at a time.

One solution would be to define specific language construct that explicitly support synchronization. Java takes this approach as it defines a special method tag that is used specify that a method is synchronized [GJS96]. All methods within a class that are tagged with the `synchronized` keyword cannot be run concurrently across an instance and all static methods that are tagged cannot be run concurrently across all instances of that class.

Although synchronized methods are directly supported by the Java compiler, they don't need to be directly supported in Smalltalk since Smalltalk exposes enough of its implementation to allow us to implement these features. For example, we can implement static synchronized methods by using method wrappers where each wrapper acquires its lock before executing the original method and releases it after the method executes. Similarly, the non-static synchronized methods can easily be implemented by using class wrappers where each instance would have its own class wrapper that would wrap each `super` message send with the lock. By using method and class wrappers, we can add this functionality in a dynamic and incremental fashion, whereas with Java, we would be forced to recompile to change the method's attribute.

4.5 Pre- and Post-conditions

Pre- and post-conditions can be used to aid the programmer in producing quality software. These conditions can help the programmer quickly detect when a component is being misused. Since the detection occurs sooner, it is more likely to be easier to fix. Eiffel supports pre- and post-conditons directly with the *require* and *ensure* keywords [Mey92]. When the conditions are enabled, invocations of the method are *required* to meet its conditions before executing and the method *ensures* its conditions after executing.

Sometimes in systems such as Smalltalk that do not directly support for pre- and post-conditions, the checks are written directly into the code. For example, the removeFirst method in OrderedCollection checks that it is non-empty. Other times these conditions are written as comments in code, or not written down at all.

While it is useful to have these checks in the code when developing the software, they are not as useful after releasing the software. To the user, an unhandled empty collection signal raised by the empty check in removeFirst is the same as an unhandled index out of bounds signal that would be raised if the error check was eliminated. Both cause the product to fail. Therefore, to be useful to developer, a system that implements pre- and post-conditions should be able to add and remove them quickly and easily.

Pre- and post-conditions can be implemented by using method wrappers. For each method, a method wrapper would be created that would test the pre-condition, evaluate the wrapped method, and finally test the post-condition on exit.

Post-conditions can also have *old* values. Old values are useful in comparing values that occur before executing a method to the values after execution. To support old values, we added a special selector, OLD, that when sent to an expression will refer to the value of the expression before the execution of the method. Although this selector appears to be a message send, we modified to the compiler to replace the message with a temporary. The receiver of the message is then assigned to the temporary before the method is executed.

As an example, consider the removeFirst method of OrderedCollection. It might have a pre-condition such as "self size > 0" and a post-condition of "self size OLD - 1 == self size" (i.e., the size of the collection after execution is one less than the size before). The method wrapper for this example would be:

```
| old1 |
old1 := self size.
[self size > 0] value ifFalse:
    [self preconditionErrorSignal raise].
^["code to evaluate wrapped method"]
    valueNowOrOnUnwindDo:
        [[old1 - 1 == self size] value
            ifFalse: [self
                postconditionErrorSignal
                    raise]]
```

Notice that the "self size OLD" from the post-condition has been replaced by a temporary and that the receiver, "self size", is assigned at the beginning of the wrapper.

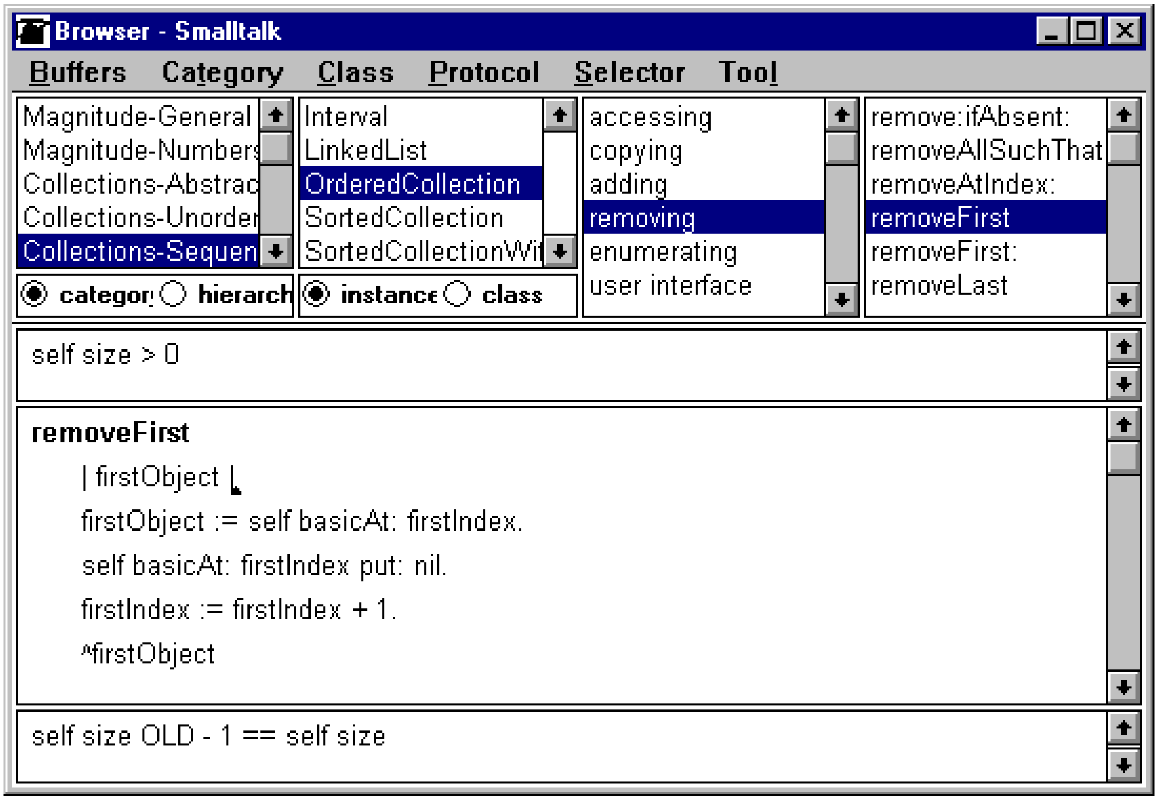


Figure 5: Browser with pre and postconditions

Others have implemented pre- and post-conditions for Smalltalk [CCGMP94, Riv96], but they modified the compiler to generate the conditions directly into the methods. Thus they require a complete recompile when (un)installing the conditions. [CCGMP94] allowed conditions to be turned on and off, but they could only be completely eliminated by a complete recompile.

Figure 5 shows a browser with pre- and post-conditions inspecting the removeFirst method. The three text panes at the bottom display the method's pre-condition, the source, and the post-condition. Both the pre-condition and the post-condition panes can be eliminated if the programmer does not wish to view them. Since the pre- and post-conditions are separated from the method, we don't need to augment the method definition with special keywords or special message sends as Eiffel and the other two Smalltalk implementations do.

4.6 MultiMethods

The Common Lisp Object System provides elaborate facilities for method wrapping. The CLOS method combination mechanism provides programmers with a great deal of control over how different kinds of methods interact with the inheritance hierarchy to determine how and when methods are executed. The CLOS standard method combination mechanism executes the `:around` and `:before` methods in outermost to innermost order. Next, the primary methods are executed, followed by the `:after` methods in innermost to outermost order. Finally, the `:around` methods are resumed.

Our basic wrappers are more modest in this respect. These wrappers executes the before code and primary code for each wrapper, before calling the wrapped method. If that method is wrapped, its before code and primary code is executed. Like CLOS `:around` methods, our wrappers may decide to not call their wrapped methods.

Method wrappers can be used to construct mechanisms like those found in CLOS. Indeed, we have used our method wrappers as the basis for a Smalltalk implementation of CLOS-style generic functions, method combination, and multimethods.

Multimethods [BDG+88] are methods that are dispatched at runtime by taking the identities of all the methods arguments into account, rather than just that of the message receiver, as is the case in languages like Smalltalk, Java, and C++. Java and C++ use static overloading to allow the compile-time types of the arguments to distinguish methods. Allowing full runtime polymorphism among all the arguments to a family of methods is a more powerful facility.

In CLOS, all the multimethods that share the same function name (selector) are members of a *generic function* by that name. When this function is called, it is the generic function's job to determine which (if any) of the multimethods defined for it apply, and to call them in the appropriate order.

The manner in which multimethods are called is determined by a *method combination* object. Multimethods are not only *specialized* by their the types of their arguments, they may also be *qualified*. For instance, the standard method combination object conducts the execution of `:around`, `:before`, `:after`, and `primary` methods in the matter described above by taking these qualifiers into account. The CLOS Metaobject Protocol is designed [KdRB91][KL92] to permit optimizations of this process via a sort of partial evaluation, using discriminating functions and effective methods.

Our Smalltalk Multimethod implementation uses a dormant, postfix bracketed type syntax that is built into the VisualWorks 2.5 Smalltalk compiler as its syntax for specializing multimethod arguments. Using this syntax, ClassSpecializers (for Classes or Metaclasses) and EqualSpecializers (for literal instances) may be specified.

When a method with these specializations is accepted in a Smalltalk browser, a MultiMethod object is created. MultiMethods are subclasses of CompiledMethod. MultiMethods are given selectors distinct from those of normal methods. MultiMethods also ensure the existence of a GenericFunction object for the selector for which they are being defined. GenericFunction objects, in turn, are entrusted with deploying any DiscriminatingMethods an object might need.

DiscriminatingMethods are subclasses of MethodWrapper that intercept calls that occupy the MethodDictionary slots where a normal method for their selector would go. When a DiscriminatingMethod gains control, it passes its receiver along to its GenericFunction, which can then determine which MultiMethods to execute in what order. It does so by passing control to its MethodCombination object.

Subclasses of our MethodCombinations, besides implementing the standard before/after/primary –style combinations, can be constructed to collect the values of their primary methods, as in CLOS, or to call methods in innermost to outermost order, as in Beta [KMMN90].

Of course, by virtue of being called in a context where a dispatch on its first argument has already been done, DiscriminatingMethods can, in conjunction with their MethodCombination objects, take advantage of such information to optimize their tasks.

Multimethods can considerably simplify the implementation of the Visitor pattern [GHJV95]. For instance, consider a typical Smalltalk implementation of Visitor:

```
ParseNode>>acceptVistor: aVisitor
    ^self subclassResponsibility

VariableNode>>acceptVistor: aVisitor
    ^aVisitor visitWithVariableNode: self

ConstantNode>>acceptVistor: aVisitor
    ^aVisitor visitWithConstantNode: self

AbstractVisitor>>
visitWithConstantNode: aNode
    ^self subclassResponsibility

OptiminzingVisitor>>
visitWithConstantNode: aNode
    ^aNode value optimized

OptiminzingVisitor>>
visitWithVariableNode: aNode
    ^aNode lookupIn: self symbolTable
```

When MultiMethods are available, the double dispatching methods in the ParseNodes disappear, since the type information need no longer be hand-encoded in the selectors of the calls to the Visitor objects. Instead, the Visitor is able to correctly dispatch calls on the visitNode: GenericFunction to the correct Multimethod:

```
ParseNode>>acceptVistor: aVisitor
   ^aVistor visitWithNode: self

OptimizingVisitor>>
visitWithNode: aNode <ParseNode>
   ^self value optimized

OptiminzingVisitor>>
visitWithNode: aNode <VariableNode>
   ^aNode lookupIn: self symbolTable
```

The savings on the ParseNode side of the hierarchy are dramatic. All the redispatching methods that encode the identity of the node from which they are called are gone.

The savings on the Visitor side may appear to be merely cosmetic. The visitWithXxxNode: methods are replaced by corresponding visitWithNode: aNode <XxxNode> methods. Even here, though, savings are possible where a particular node's implementation can be defined in a superclass of the leaf node. For instance, if many of an OptimizingVisitor's multimethods would have sent the optimized message to their Node's value, they can share the implementation of this method defined for OptimizingVistor and ParseNode. With the double dispatched implementation, a stub implementation of the subclass version of the method is usually provided so as to avoid a breach of encapsulation.

Classtalk [C90] provided an implementation of CLOS-style before and after method combination (without multimethods), that uses Smalltalk's doesNotUnderstand: mechanism to gain control.

5. Existing Approaches

Many systems provide ways for programs to augment or preempt the behavior of existing functions. Programmers invariably exploit these facilities wherever they are found. If the language itself does not permit such modifications, programmers will often resort to low-level, implementation specific schemes to achieve their ends.

Wrapping strategies are not limited to languages. For instance, all the routines in the Macintosh Toolbox can be wrapped. The architects of the Toolbox designed it so that calls to the ROM-based built-in Toolbox functions were accessed indirectly through a table in RAM. This indirection allowed Apple to ship patched copies of Toolbox entries to correct or supplement the existing routines. It also gave third-party software designers the opportunity to change the routines from which the system was built.

Over the years, Macintosh programmers have shown remarkable ingenuity in the ways they've exploited these hooks into the system. For instance, applications like wide-desktops and screen savers were built by wrapping the Toolbox. This shows the wisdom of designing systems with flexible foundations.

Programmers using Microsoft Windows have achieved similar results with the dynamic linking mechanism used to implement Dynamic Link Libraries (DLLs). A function can be wrapped by providing a wrapping implementation for it in a DLL that is referenced before the wrapped DLL.

C++ provides no explicit mechanisms for allowing programmers to intercept calls to C++ functions, virtual or otherwise. However, some programmers have exploited the most common implementation mechanism for dispatching C++ virtual functions, the "v-table" [ES90] to gain such access [Tie88]. By falling back on unsafe C code, and treating v-table entries as simple C function pointers, programmers can dynamically alter the contents of the v-table entry for a class of objects. By substituting another function with the same signature for a given v-table entry, that entry can be wrapped with code that can add before and after actions before calling (or not calling) the original method.

Since the v-table mechanisms are not a part of the C++ standard, and since more complex features of C++ such as multiple inheritance and virtual bases often employ more elaborate implementations, programmers cannot write portable code that depends on "v-table surgery".

Interestingly, C with Classes contained a mechanism [Str94] that allowed programmers to specify a function that would be called before every call to every member functions (except constructors) and another that would be called before every return from every member function. These `call` and `return` functions resemble dispatching wrappers.

In contrast to C++, the Microsoft Common Object Model (COM) [Bro95] defines an explicit binary format that is similar to, and based upon, the customary implementation of simple v-tables. Since any COM object must adhere to this format, it provides a potential basis for v-table manipulation.

6. Conclusion

The original Smalltalk designers did a wonderful job of building a language out of objects that users can change. We rarely run into the "keep out" signs that so often frustrate users of other languages. This lets us add new tools to the programming environment, keep up with the latest database and network technology, and maintain and enhance our own systems as they evolve.

Smalltalk's reflective facilities, together with our wrappers, allowed us to construct powerful program analysis tools and language extensions with relative ease. The ease with which we can and remove wrappers at runtime makes tools like our interaction diagramming tool possible.

By contrast, adding dynamic coverage analysis, for example, to an existing program at either the source or binary level is impossible for users of traditional systems, and next to impossible for the vendors of such tools.

We feel that if the next generation of object-oriented software is to fulfill its promise, that tools like the ones we've described, and flexibility of the sort we've demonstrated, will be instrumental in its construction. This in turn, argues that we should think twice before letting the next round of object-oriented languages go forward without facilities of the sort we've examined here.

When we needed a way to trap object interactions, or to build synchronized methods, wrappers came to the rescue. We must remain vigilant if programmers in the next century's languages are to be similarly fortunate. The design of the reflective facilities for these languages cannot be taken lightly.

Acknowledgements

The interaction diagramming tool was originally a project done by David Wheeler, Jeff Will, and Jinghu Xu for Prof. Johnson's CS497 class. Their report on this project can be found at:

http://radon.ece.uiuc.edu/~dwheeler/interaction.html.

References

The code referenced in this article can be found at:

http://st-www.cs.uiuc.edu/~brant/wrappers.html

[BDG+88] Dan 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, September 1988.

[Bec95] Kent Beck. Using demand loading. *The Smalltalk Report*, 4(4):19-23, January 1995.

[Ben87] John K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87*, pages 318-330, December 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.

[Bro95] Kraig Brockschmidt. *Inside OLE*, second edition, Microsoft Press, Redmond, Washington, 1995.

[BS83] Daniel G. Bobrow and Mark Stelik. *The LOOPS Manual*. Xerox PARC, 1983.

[C90] Pierre Cointe, *The Classtalk System: a Laboratory to Study Reflection in Smalltalk*, OOPSLA/ECOOP '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, Mamdouh Ibrahim, organizer.

[CCGMP94] Manuela Carrillo-Castellon, Jesus Garcia-Molina, and Ernesto Pimentel. Eiffel-like assertions and private methods in Smalltalk. In *TOOLS 13*, pages 467-478, 1994.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In Proceedings *OOPSLA '89*, pages 327-336, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[GJS96] James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, 1996

[HJ92] Bob Hinkle and Ralph E. Johnson. Taking exception to Smalltalk. *The Smalltalk Report*, 2(3), November 1992.

[HJJ93] Bob Hinkle, Vicki Jones, and Ralph E. Johnson. Debugging objects. *The Smalltalk Report*, 2(9), July 1993.

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

[KL92] Gregor Kiczales and John Lamping, *Issues in the Design and Implementation of Class Libraries*, OOPSLA '92, Vancouver, BC, SIGPLAN Notices Volume 27, Number 10, October 1992

[KMMN] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard, *Object-Oriented Programming in the Beta Language*, 8 October, 1990

[LG88] Wilf R. LaLonde and Mark Van Gulik. Building a backtracking facility in Smalltalk without kernel support. In *Proceedings OOPSLA '88*, pages 105-122, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[MB85] Steven L. Messick and Kent L. Beck. Active variables in Smalltalk-80. Technical Report CR-85-09, Computer Research Lab, Tektronix, Inc., 1985.

[McC87] Paul L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87*, pages 331-341, December 1987. Published as ACM SIGNPLAN Notices, volume 22, number 12.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[MW81] D. Weinreb, and D. Moon. Lisp Machine Manual, Symbolics, 1981

[Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86*, pages 341-346, November 1986. Published as ACM SIGPLAN Notices, volume21, number 11.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings Reflection '96*.

[Str94] Bjarne Stroustrop. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA 1994

[Tie88] Michael D. Tiemann. Solving the RPC problem in GNU C++. In 1988 USENIX C++ Conference, pages 17-21, 1988.