Wrappers to the Rescue

John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts

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

Abstract. Wrappers are mechanisms for introducing new behavior that is executed before and/or after, and perhaps even in lieu of, an existing method. This paper examines several ways to implement wrappers in Smalltalk, and compares their performance. Smalltalk programmers often use Smalltalk's lookup failure mechanism to customize method lookup. Our focus is different. Rather than changing the method lookup process, we modify the method objects that the lookup process returns. We call these objects *method wrappers*. We have used method wrappers to construct several program analysis tools: a coverage tool, a class collaboration tool, and an interaction diagramming tool. We also show how we used method wrappers to construct several extensions to Smalltalk: synchronized methods, assertions, and multimethods. Wrappers are relatively easy to build in Smalltalk because it was designed with reflective facilities that allow programmers to intervene in the lookup process. Other languages differ in the degree to which they can accommodate change. Our experience testifies to the value, power, and utility of openness.

1 Introduction

One benefit of building programming languages out of objects is that programmers are able to change the way a running program works. Languages like Smalltalk and CLOS, which represent program elements like Classes and Methods as objects that can be manipulated at runtime, to allow programmers to change the ways these objects work when the need arises.

This paper focuses on how to intercept and augment the behavior of existing methods in order to "wrap" new behavior around them. 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, they illustrate the power and importance of the reflective facilities that support them.

Before and after methods as we now know them first appeared in Flavors [30] and Loops [5]. The Common Lisp Object System (CLOS) [4] provides a powerful method standard combination facility that includes `:before`, `:after`, and `:around` 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 can wrap all of these, and has the option of completing the rest of the computation. The method combination mechanism built into CLOS also lets programmers build their own method qualifiers and combination schemes, and is very powerful.

Unfortunately, misusing method combination 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 understand and maintain. Using these facilities to solve application-level problems is often symptomatic of more serious design problems that should be addressed through refactoring instead. The result is that before and after methods have gained a bad reputation.

We use method wrappers mostly as a *reflective* facility, not a normal application programming technique. We think of them as a way to impose additional structure on the underlying reflective facilities. For example, we use them to dynamically determine who calls a method, and which methods are called. If methods wrappers are treated as a disciplined form of reflection, then they will be used more carefully and their complexity will be less of a problem.

Our experience with method wrappers has been with Smalltalk. Smalltalk has many reflective facilities. Indeed, Smalltalk-76 [17] was the first language to cast the elements of an object-oriented language itself, such as classes, as first-class objects. The ability to trap messages that are not understood has been used to implement encapsulators [26] and proxies in distributed systems [2, 23]. The ability to manipulate contexts has been used to implement debuggers, back-trackers [21], and exception handlers [15]. 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 [1] and can change the class of an object [16].

However, it is not possible to change every aspect of Smalltalk [10]. Smalltalk is built upon a virtual machine that defines how objects are laid out, how classes work, and how messages are handled. The virtual machine can only be changed 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 method wrappers to change how a method is executed. The most common reason for changing how a method is executed is to do something at every execution, and method wrappers work well for that purpose.

2 Compiled Methods

Many of the method wrapper implementations discussed in this paper are based on CompiledMethods, so it is helpful to understand how methods work to understand the different implementations. While this discussion focuses on VisualWorks, we have also implemented wrappers in VisualAge Smalltalk. They can be implemented in most other dialects of Smalltalk. However, the method names and structure of the objects are somewhat different. A complete discussion of how to implement wrappers in these other dialects of Smalltalk is beyond the scope of this paper.

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 other information about the method, such as the set of messages that it sends and the bytecodes that define the execution of the method.

Interestingly, CompiledMethods do not know the selector with which they are associated. Hence, they are oblivious as to which name they are invoked by, as well as to the names of their arguments. They are similar to Lisp lambda-expressions in this respect. Indeed, a compiled method can be invoked even if it does not reside in any MethodDictionary. We will use this fact to construct MethodWrappers.

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 different sources appear when the 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 a super-

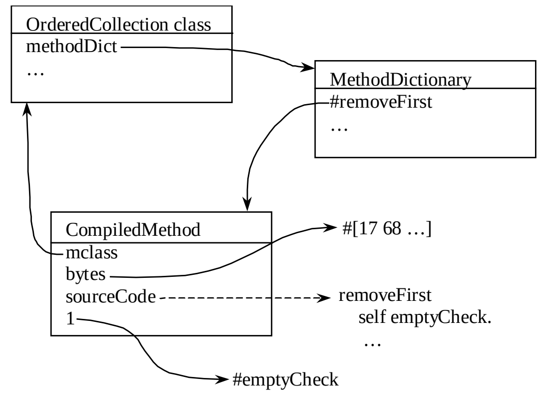

Fig. 1. `removeFirst` method in OrderedCollection

class. 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. 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 indicates the source code that the integer points to.

3 Implementing Wrappers

There are many different ways to implement method wrappers in Smalltalk, ranging from simple source code modification to complex byte code modification. In the next few sections we discuss seven possible implementations and some of their properties. Although many of the implementation details that we use are Smalltalk-specific, other languages provide similar facilities to varying degrees.

3.1 Source code modification

A common way to wrap methods is to modify the method directly. The wrapper 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 from 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 way to modify a method is to modify the CompiledMethod directly without recompiling [24]. 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. This makes installation 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 knowledge of the byte code instructions used by the virtual machine. These codes are not standardized and can change without warning.

3.3 New selector

Another way to wrap methods is to move the original method to a new selector and create a new method that executes the before code, sends the new selector, and then executes the after code. With this approach the new method is:

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

This approach was used by Böcker and Herczeg to build their Tracers [3].

This implementation has a couple of desirable properties. One is that the original methods do not need to be recompiled when they are moved to their new selectors.

Since CompiledMethods 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. Cloning these CompiledMethods objects (i.e. using the Prototype pattern [11]) is much faster than compiling new ones. 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 nor the new forwarding method needs to be compiled, so the installation is faster than the source code modification approach.

One problem with this approach is that the new selectors are visible to the user. Böcker and Herczeg addressed this problem by modifying the browsers. The 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 method wrappers since we must remember which of the selectors represent the original methods and which are the new selectors.

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 long been used for this purpose [26, 2, 10, 12, 14] This approach works well when some action must be taken regardless of which method is being called, such as coordinating synchronization information. Given some extra data structures, it can be used to implement wrapping on a per-method basis. For example, Classtalk [8] used `doesNotUnderstand:` to implement a CLOS-style before- and after- method combination mechanism.

A common way to do this is to introduce a class with no superclass to intercept the dispatching mechanism to allow per-instance changes to behavior. However, the `doesNotUnderstand:` mechanism is slow, and screening every message sent to an object just to change the behavior of a few methods seems wasteful and inelegant. The following sections examine how Smalltalk's meta-architecture lets us more precisely target the facilities we need.

3.5 Class Wrapper

The standard approach for specializing behavior in object-oriented programming is subclassing. We can use subclassing to specialize methods to add before and after code. In this case, the specialized subclass essentially wraps the original class by creating a new method that executes the before code, calls the original method using `super` mechanism, and then executes the after code. Like the methods in the new selector approach, the methods for the specialized subclass can also be copied, so the compiler is not needed.

Once the subclass has been created, it can be installed into the system. To install the subclass, the new class has to be inserted into the hierarchy so that subclasses will also use the wrapped methods. It can be inserted by using the `superclass:` method to change the superclass of all of the subclasses of the class being wrapped to be the wrapper. Next, the reference to the original class in the system dictionary must be replaced with a reference to the subclass. Finally, all existing instances of the original class have 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.

Like the new selector approach this only requires one additional message send. However, these sorts of wrappers take 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 have to iterate though them changing each of their classes.

3.6 Instance Wrapper

The class wrapper approach can also be used to wrap methods on a per instance basis, or a few at a time. Instead of replacing the class in the system dictionary, we can change only the objects that we want to wrap, by using the `changeClassToThatOf`: method on only those objects.

Instance wrappers can be used to change the way individual objects behave. This is the intent of the Decorator pattern [11]. However since these decorations are immediately visible though existing references to the original object, objects can be decorated dynamically.

3.7 Method Wrapper

A method wrapper is like a new selector in that the old method is replaced by a new one that invokes the old. However, a method wrapper does not add new entries to the method dictionary. Instead of invoking the old method by sending a message to the receiver, a method wrapper evaluates the original method directly. A method wrapper must know the original method, and must be able to execute it with the current arguments. Executing a CompiledMethod is easy, since a CompiledMethod responds to the `valueWithReceiver:arguments:` message by executing itself with the given a receiver and an array of arguments.

One way for a MethodWrapper to keep track of its original method is for MethodWrapper to be a subclass of CompiledMethod with one new instance variable, *clientMethod*, that stores the original method. MethodWrapper also defines `beforeMethod`, `afterMethod`, and `receiver:arguments:` methods as well as a few helper methods. The `beforeMethod` and `afterMethod` methods

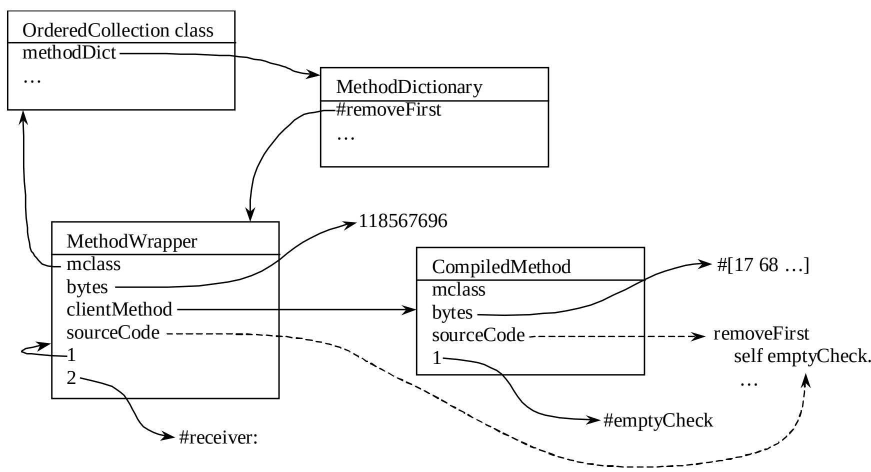

Fig. 2. MethodWrapper on `removeFirst` method

contain the before and after code. The `valueWithReceiver:arguments:` method executes the original method given the receiver and argument array.

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

The only remaining problem is how to send the `valueWithReceiver:-arguments:` message to a MethodWrapper. The method must be able to refer to itself when it is executing, but Smalltalk does not provide a standard way to refer to the currently executing method. When a CompiledMethod is executing, the receiver of the message, and not the CompiledMethod, is the "self" of the current computation. In VisualWorks Smalltalk, the code "`thisContext method`" evaluates to the currently executing method, but it is inefficient. We need some kind of "static" variable that we could initialize with the method, but Smalltalk does not have that feature. Instead, we make use of the fact that each Smalltalk method keeps track of the literals (i.e. constants) that it uses. Each MethodWrapper is compiled with a marked literal (we use #(), which is an array of size 0). After it has been created, the system replaces the reference to the literal with a reference to the MethodWrapper. 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 `valueWithReceiver:arguments:` method.

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

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

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.

Smalltalk's CompiledMethod objects and byte code were designed primarily to make Smalltalk portable. As with `doesNotUnderstand:`, Smalltalk's historic openness continues to pay unexpected dividends.

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.

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 more than ten times greater than the other techniques. Whenever a return occurs in a block, a context object is

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

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

created at runtime. Normally these context objects are not created so execution is much faster. 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.

Because wrappers are relatively fast, and because the overhead associated with them is predictable, they may be more suitable in time-critical applications than classical Smalltalk approaches based on `doesNotUnderstand:`.

4 Applications

Method wrappers can be used in many different areas. In this section we outline six different uses.

4.1 Coverage Tool (Image Stripper)

One application that can use method wrappers 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 of the application program is slow at first, but it rapidly increases once the base set of methods is 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. Removing methods in this manner can introduce errors into an otherwise correct program. 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. The best way to detect that a deleted method has been called is with the `doesNotUnderstand:` mechanism, though it is also possible to use method wrappers for this purpose.

4.2 Class Collaboration

Method wrappers can also be used to dynamically analyze 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.

Method wrappers can capture this information by getting the current context, just like the debugger does. Whenever a method is called, its wrapper needs 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). Meth-

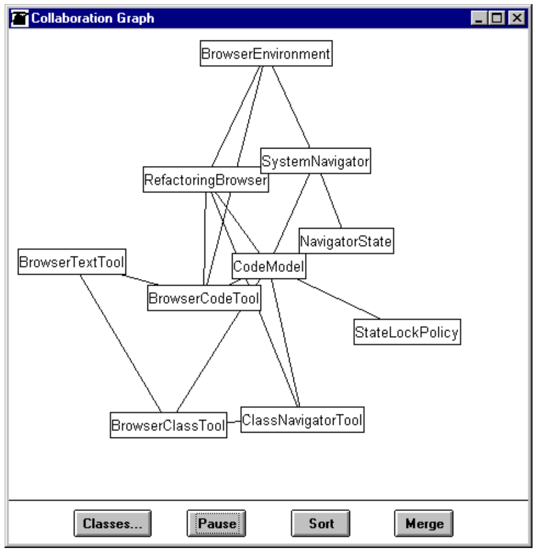

Fig. 3. Class collaboration graph of the Refactoring Browser

ods 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 such as the one 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 interaction diagram application allows users to select the set of methods that will be watched. These methods are wrapped, and the tool records traffic through them. 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:

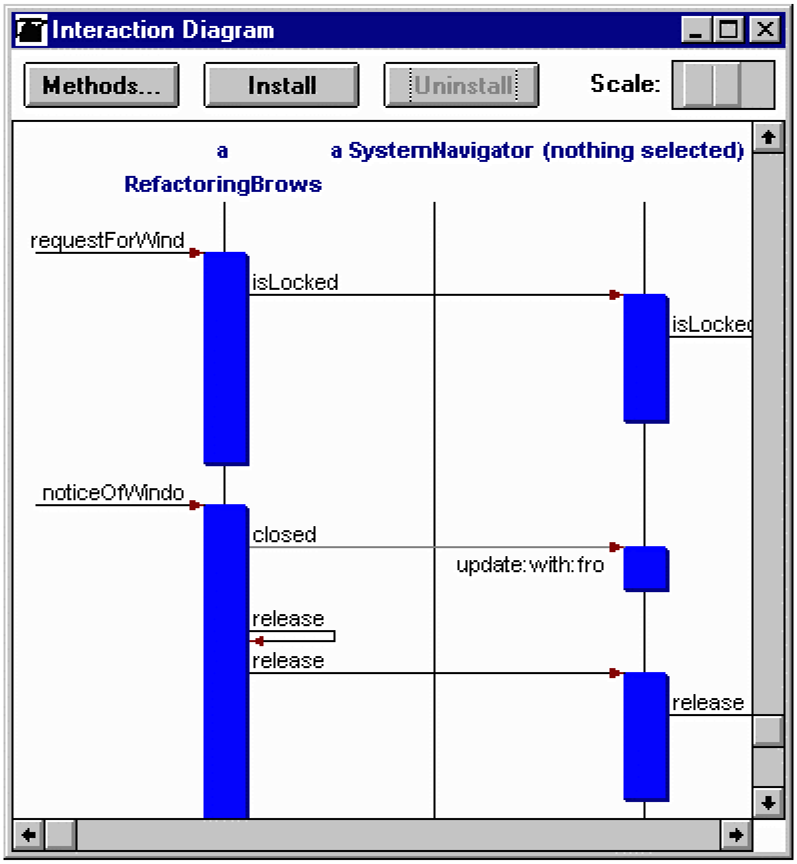

Fig. 4. Interaction Diagram on the Refactoring Browser

```
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 figure, 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. For instance Lange and Nakamura [22] modify source code to instrument programs for tracing. The relative absence of such tools in languages without support for wrappers testifies to the difficulty of intervening at these levels.

Both Probe from Arbor Intelligent Systems and the Object Visualizer in IBM's VisualAge for Smalltalk generate interaction diagrams using method wrappers. Probe uses method wrappers that are very similar to those described in this paper except that the before and after code is been inlined into the wrapper.

The Object Visualizer uses a combination of lightweight classes and method wrappers to capture the runtime interaction information. However, their method wrappers do not directly reference the wrapped method. Instead they look up the method for every send. Instance wrappers would have been a better choice given this approach.

4.4 Synchronized Methods

Method wrappers are also useful for 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 a language construct for synchronization. For example, Java takes this approach by defining a method tag that is used to specify that a method is synchronized [13]. The system ensures that only one method that is tagged with the `synchronized` keyword is running at any time for an instance and only one static method that is tagged is running at any time for a single class.

The Smalltalk compiler does not need to directly support synchronized methods 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 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. Method and class wrappers let us add this functionality in dynamically, whereas Java forces us to recompile to change the method's attribute.

4.5 Pre- and Post-conditions

Pre- and post-conditions help programmers produce quality software by describing a component and helping detect when it is being misused. The earlier an error is detected, the easier it is to fix. Eiffel supports pre- and post-conditions directly with the *require* and *ensure* keywords [25]. When conditions are enabled, invocations of the method are *required* to meet its conditions before executing and the method *ensures* its conditions after executing.

In systems like Smalltalk that do not directly support pre- and post-conditions, programmers sometimes write the checks 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, a preprocessing step replaces it 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.

Others have implemented pre- and post-conditions for Smalltalk [7, 27], but they modified the compiler to generate the conditions directly into the methods. Thus they require a complete recompile when (un)installing the conditions. [7] allowed conditions to be turned on and off, but they could only be completely eliminated by a complete recompile.

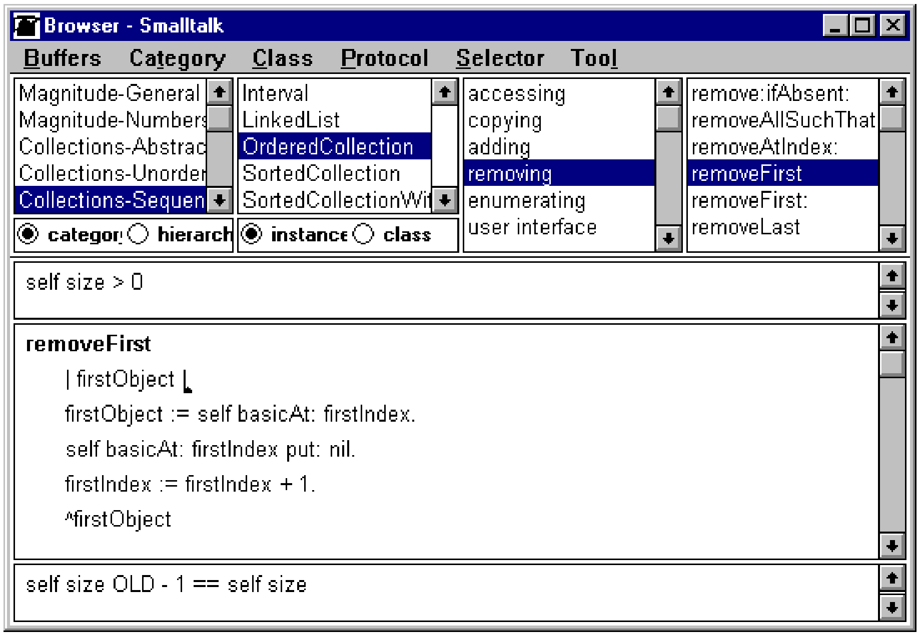

Fig. 5. Browser with pre and postconditions

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 and the CLOS Metaobject Protocol [18] provide 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 much simpler. They execute 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.

We have used method wrappers to construct mechanisms like those found in CLOS. Next, we will describe how to use them to build CLOS-style generic functions, method combination, and multimethods.

Multimethods [4] 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 distinguish methods based on the compile-time types of the arguments. Multimethods are more powerful because they choose a method at runtime.

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 determines which (if any) of its multimethods apply, and calls them in the appropriate order.

The way that multimethods are called is determined by a *method combination* object. Multimethods are not only *specialized* by 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 by taking these qualifiers into account. The CLOS Metaobject Protocol [18, 19] permits optimizations of this process by a sort of partial evaluation, using discriminating functions and effective methods.

Our Smalltalk multimethod implementation uses a dormant type syntax that is built into the VisualWorks Smalltalk compiler as its syntax for specializing multimethod arguments. This syntax lets us specify both ClassSpecializers, and EqualSpecializers for literal instances.

When the Smalltalk browser accepts a method with these specializations, it creates a MultiMethod object. MultiMethods are subclasses of CompiledMethod that are given selectors distinct from those of normal methods. MultiMethods also make sure there is an instance of GenericFunction for the selector for which they are being defined. GenericFunctions also keep track of one or more DiscriminatingMethods.

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 and arguments 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 [20].

Since a DiscriminatingMethod is called after a dispatch on its first argument has already been done, it can use that information to optimize its task.

To illustrate our syntax, as well as the power of multimethods, consider the impact of multimethods on the Visitor pattern [11]. First, consider a typical Smalltalk implementation of Visitor:

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

VariableNode>>acceptVistor: aVisitor
 ^aVisitor visitWithVariableNode: self

ConstantNode>>acceptVistor: aVisitor
 ^aVisitor visitWithConstantNode: self

OptimizingVisitor>>visitWithConstantNode: aNode
 ^aNode value optimized

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

However, when MultiMethods are available, the double-dispatching methods in the ParseNodes disappear, since the type information does not need to be hand-encoded in the selectors of the calls to the Visitor objects. Instead, the Visitor correctly dispatches calls on the `visitWithNode:` GenericFunction to the correct MultiMethod. Thus, adding a Visitor no longer requires changing the ParseNode classes.

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

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

The savings on the Visitor side may appear to be merely cosmetic. The `visitWithXxxNode:` methods are replaced by corresponding `visitWithNode: aNode <XxxNode>` methods, which are specialized according to the sort of node they service. Even here, though, savings are possible when a particular node's implementation can be shared with that of its superclass. For instance, if many of `OptimizingVisitor's` multimethods would have sent the optimized message to their node's value, they can share the implementation of this method defined for `OptimizingVisitor` and `ParseNode`. The hand coded double-dispatched implementations usually provide a stub implementation of the subclass `Node's` version of the method so as to avoid a breach of encapsulation.

5 Other Approaches

Many systems provide ways for programs to augment or preempt the behavior of existing functions. If the language 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 that 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++ has no standard 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" [9] to gain such access [29]. 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 [28] 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 Component Object Model (COM) [6] defines an explicit binary format that is similar to, and based upon, the customary implementation of simple C++ v-tables. Since any COM object must adhere to this format, it provides a potential basis for wrapping methods using v-table manipulation, since the rules by which v-tables must play are explicitly stated.

6 On the Importance of Being Open

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 add and remove wrappers at runtime makes tools like our interaction diagramming tool possible. In contrast, adding a feature like dynamic coverage analysis to an existing program is impossible for users of traditional systems, and difficult for the tool vendors.

While wrappers can, in principle, be used to solve problems at both the domain-level, or at the meta-, or language-level, the analysis tools and language extensions we present here are all reflective applications, in that they exploit our ability to manipulate program objects as pieces of the program itself, rather than as representations of application-level concerns.

In the case of *tools*, the fact that the program is built out of objects lets us inspect and alter these objects on-the-fly. Tools need an inside view of the program. For instance, when we wrap a CompiledMethod object using our interaction tool, we are exploiting its language level role as a program element and are indifferent to domain-specific behavior it engenders.

In the case of our *linguistic extensions*, the openness of the language's objects permitted us to construct these extensions, which were then used to write domain specific code. The use of raw reflective facilities to construct such extensions is a good way to harness the power of reflection.

None of the examples of method wrappers in this paper are domain specific. Domain specific uses of reflective facilities like before- and after- methods are frequently symptoms of problems with the application's structure that would be better addressed by refactoring its design. Reflective facilities are useful alternatives when a system becomes so overgrown that it can't be redesigned. Being able to change the language level gives programmers a big lever and can buy time until the resources to overhaul the system become available. However, metalevel tricks are no substitute for properly addressing a system's serious long-term structural problems.

A generation of Smalltalk programmers has turned to Smalltalk's lookup failure exception, `doesNotUnderstand:,` when the time came to extend the language. This paper has examined the strengths and weaknesses of several ways of intervening during the lookup process. Approaches based on `doesNotUnderstand:` have a brute force quality about them, since they must screen every message sent. Method wrappers let us intervene more precisely and selectively. When we needed a way to build power, efficient programming tools and language extensions, wrappers came to the rescue.

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.

Acknowledgments

Ian Chai, Dragos Manolescu, Joe Yoder, and Yahya Mirza provided valuable comments and insights on an earlier version of this paper. James Noble suggested a useful simplification in our Visitor pattern example. The ECOOP '98 program committee reviewers also suggested a number of valuable improvements.

The interaction diagramming tool was originally a project done by David Wheeler, Jeff Will, and Jinghu Xu for Ralph 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/Applications/MethodWrappers.html

1. Kent Beck. Using demand loading. *The Smalltalk Report*, 4(4):19-23, January 1995.
2. 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.
3. Heinz-Dieter Böcker and Jürgen Herczeg, *What Tracers are Made Of*, ECOOP/OOPSLA '90 Conference Proceedings, SIGPLAN Notices, Volume 25, Number 10, October 1990
4. 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.
5. Daniel G. Bobrow and Mark Stefik. *The LOOPS Manual*. Xerox PARC, 1983.
6. Kraig Brockschmidt. *Inside OLE*, second edition, Microsoft Press, Redmond, Washington, 1995.
7. Manuela Carrillo-Castellon, Jesus Garcia-Molina, and Ernesto Pimentel. Eiffel-like assertions and private methods in Smalltalk. In *TOOLS 13*, pages 467-478, 1994.
8. 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.
9. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.
10. 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.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
12. B. Garbinato, R. Guerraoui, and K. Mazoui. Implementation of the GARF Replicated Objects Platform. In *Distributed Systems Engineering Journal*, (2), 1995, 14–27.
13. James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, 1996.
14. R. Guerraoui, B. Garbinato, and K. Mazouni. The GARF System. In *IEEE Concurrency*, 5(4), 1997.
15. Bob Hinkle and Ralph E. Johnson. Taking exception to Smalltalk. *The Smalltalk Report*, 2(3), November 1992.
16. Bob Hinkle, Vicki Jones, and Ralph E. Johnson. Debugging objects. *The Smalltalk Report*, 2(9), July 1993.

17. Daniel H. H. Ingalls, The Evolution of the Smalltalk-80 Virtual Machine, in Smalltalk-80, Bits of History, Words of Advice, Glenn Krasner, editor, Addision-Wesley, Reading, MA, 1983
18. Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.
19. 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.
20. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard, *Object-Oriented Programming in the Beta Language*, 8 October, 1990.
21. 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.
22. Danny B. Lange and Yuichi Nakamura, Interactive Visualization of Design Patterns Can Help in Framework Understanding, In *Proceedings of OOPSLA ’95*, pages 342-357, October 1995, Published as ACM SIGPLAN Notices, volume 30, number 10
23. 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.
24. Steven L. Messick and Kent L. Beck. Active variables in Smalltalk-80. Technical Report CR-85-09, Computer Research Lab, Tektronix, Inc., 1985.
25. Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
26. 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.
27. Fred Rivard. Smalltalk: a reflective language. In *Proceedings Reflection ’96*.
28. Bjarne Stroustrop. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA 1994.
29. Michael D. Tiemann. Solving the RPC problem in GNU C++. In 1988 USENIX C++ Conference, pages 17-21, 1988.
30. D. Weinreb, and D. Moon. Lisp Machine Manual, Symbolics, 1981.