Evaluating Message Passing Control Techniques in Smalltalk

Stéphane Ducasse
Software Composition Group, Universität Bern
ducasse@iam.unibe.ch
http://www.iam.unibe.ch/∼ducasse/

Appeared in JOOP (Journal of Object-Oriented Programming) June 1999

Abstract

In a language like Smalltalk in which objects communicate only via message passing, message passing control is a fundamental tool for the analysis of object behavior (trace, spying) or for the definition of new semantics (asynchronous messages, proxy,...). Different techniques exist, from the well known approach based on the specialization of the doesNotUnderstand: method to the exploitation the method lookup algorithm done by the virtual machine. Until now no comparison between these techniques has been made. In this article we compare the different techniques taking into account the reflective aspects used, the scope, the limit and the cost of the control.
Keywords: message passing control, instance specialization, doesNotUnderstand:, error handling, method compilation, anonymous class, minimal object

1 Message Passing Control: A need

Message passing control is the corner stone of a broad range of applications from application analysis (trace[BH90, PWG93], interaction diagrams, class affinity graphs) to the introduction of new language features (multiple inheritance[BI82], interfaces [Sch96], distributed systems[GGM95, Ben87, McC87], active objects [Bri89]...). CLOS is one of the rare languages that made the effort to *explicitly* provide message passing control at the meta-level via its MOP [KdRB91, Bec95]. In Smalltalk, message passing control is not explicitly provided. However, its reflective capabilities allows one to define message passing control using various techniques: The best-known is based on the definition of so called *minimal objects* and the specialization of the doesNotUnderstand: method [Pas86, Lal90, PWG93]. Some other techniques exist like the definition of method wrappers [Bra96] or anonymous classes[McA95].

Up to now, no comparison between these techniques has been made that evaluates their applicability, benefits and drawbacks. This is a problem because each solution possesses good and bad points and often people apply a technique without checking all the consequences of their choice.

In this article we compare these techniques taking into account the reflective aspects used, the controlled objects, the integration of the control into the programming environment, the limit and the cost of the control. We start by giving an overview of the different applications of message passing control in section 1.1. We define the criteria to compare the different techniques. For the sake of understanding, we summarize the reflective facilities of Smalltalk on which such techniques are based. We then present each main technique in detail: error handling specialization in section 2, exploiting the VM method lookup in section 3, and modification of the compiled method in section 4. Finally we conclude with a discussion of message passing control in other languages.

1.1 Message Passing Control Applications in Smalltalk

Applications[1] which use message passing control can be roughly sorted into three main categories. The first is *application analysis and introspection* that is based on the development of tools that display interaction diagrams, class affinity graphs, graphic traces [BH90, PWG93, Bra96, Mic96]. The second category is *Smalltalk language*

[1]Due to the space limitation we limited this short overview to the use of message passing control in Smalltalk.

extension. In such a case message passing control allows one to define new features from within the language itself: Garf [GGM95], Distributed Smalltalk [Ben87] or [McC87] introduce object distribution in a transparent manner. Language features like multiple inheritance [BI82], backtracking facilities [LG88], instance-based programming [Bec93b, Bec93a, Hop94], Java interfaces [Sch96] or inter-objects connections [DBFP95] have been introduced. Futures [Pas86, Lal90] or atomic messages [FJ89, McA95] are also based on message passing control capabilities. The third category is the *definition of new object models*, introducing concurrent aspects such as active objects (Actalk [Bri89]) and synchronization between asynchronous messages (Concurrent Smalltalk [YT87]). Other work proposes new object models like the composition filter model [ABV92] or CodA that is a meta-object protocol that controls all the activities of distributed objects [McA95].

1.2 Selected Reflective Features of Smalltalk

Even if Smalltalk is a reflective language [GR89, FJ89, Riv96], it is not possible to change all its aspects. Indeed, the virtual machine (VM) defines the way the objects are represented in memory, and how messages are handled. As message passing control implementations have to use the reflective facilities offered by the VM, we now summarize them.

The Smalltalk dialects referenced are: VisualWorks (previously named ObjectWorks from ParcPlace newly ObjectShare), IBM Smalltalk (integrated into the VisualAge environment of IBM) and VisualSmalltalk (previously Smalltalk/V then Parts of Digitalk). Note that the examples will be presented using VisualWorks and that we will discuss the other solutions when there are significant differences.

Reification and Dynamic Creation. In Smalltalk, classes and methods are objects and are described by classes. It is not only possible, as in Java [Fla97], to access to the information that represents such entities but also to modify and dynamically create instances of these classes.

In VisualWorks, classes are dynamically created by invoking the method subclass:instanceVariableNames:classVariableNames:poolDictionaries:category: of the class Class. It is possible to access and modify the inheritance link, the method dictionary and the methods defined in method dictionary of a class (methods superclass, superclass:, methodDictionary:, compiledMethodAt: of the class Behavior in VisualWorks).

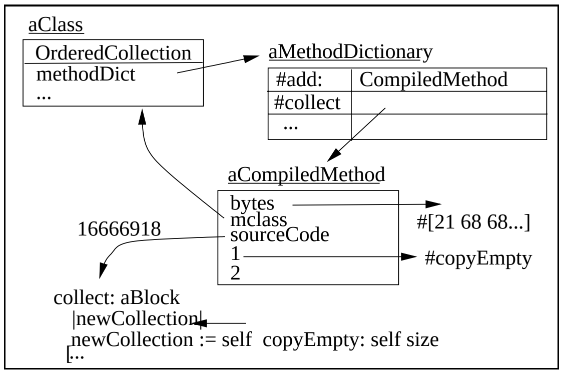

Figure 1: Relationship between class, method dictionary and compiled method in VisualWorks. The collect: method of the OrderedCollection class. The instance variable sourceCode holds an index that is used by the source manager to retrieve the source code for the method.

In VisualWorks, methods are instances of the CompiledMethod class. They can be created by invoking the method compile:notifying: of class Behavior. As shown in figure 1, they are stored in the class method dictionary. A compiled method defines information to access its source code (sourceCode), its compiled byte codes (bytes), the class that compiled it (mclass) and a variable part called the *literal frame* of the method that contains Smalltalk literal objects, such as the symbols, arrays, numbers, byte-arrays and blocks defined in the method.

Note that the source code of a method is stored separately from its byte codes and that a method only needs its byte codes to be executed. The method source can be changed without changing the executable byte codes of the method. Moreover, a compiled method is similar to a Lisp lambda-expression because it does not know its selector. To know the name of a method (its selector) the class for which it was compiled is asked. A compiled method can be executed without being defined in a method dictionary.

Finally, it is possible to invoke a given method without first doing dynamic dispatch (methods valueWithReceiver:arguments: of class CompiledMethod in VisualWorks and executeWithReceiver:andArguments in IBM Smalltalk). Note that this last functionality did not exist in the first implementations of Smalltalk. This recent addition explains why only a few implementations are based on this possibility.

Moreover, the method perform:with: defined on the class Object allows one to explicitly send a message to any object in the system. anObject perform: #zork with:12 sends to anObject the message whose selector is zork and argument 12.

Changing Reference. The become: primitive allows one to change object references. After invoking a become: b all the pointers that pointed on a point to b and conversely. Note that the semantics of this primitive depends on the Smalltalk implementations: it is symmetric in VisualWorks and asymmetric in IBM Smalltalk.

Changing of Class. An object can dynamically change its class, from a *source* class to a *target* class. This change can be perceived as pointer swap when the two classes possess the same instance structure. In VisualWorks the method changeClassToThatOf: takes as argument an object whereas in IBM Smalltalk the method fixClassTo: takes a class. The implementors of VisualWorks are then sure that the target class is an instantiable class without having to test this at the VM level. The change of class is only possible if the format of the source and target classes are compatible. The format of one class describes the memory layout of its instances (methods format, setFormat: defined on the Behavior class in VisualWorks, and instanceShape, instanceShape: defined on the class Class in IBM Smalltalk).

Message Reification and Error Handling Specialization. When an object receives an unknown message, the Smalltalk virtual machine sends the doesNotUnderstand: message to this object with the reification of the message leading to this error. On the class Object the method doesNotUnderstand: raises an exception which, if it is not trapped (unhandled exception), opens the debugger. This method can be specialized to support message passing control as will be shown in 2.

The reification of the message is done by the VM by creating an instance of the class Message. For example, the message 3 zork: 4 leads to the invocation of 3 doesNotUnderstand: aMessage for which aMessage possesses the following information:

```
aMessage selector     -> #zork:
aMessage arguments    ->  #(4)
```

A deontological remark. Some of the functionalities presented above and used in the techniques to be described are qualified as *private* in the Smalltalk versions and therefore are subject to change. It is common use and good style not to use such private methods. However, the internal aspects of the presented techniques imply their use. We stress that if such methods would had been really private some interesting techniques would have been simply impossible.

1.3 Three Main Techniques

First of all message passing control is not limited to the definition of auxiliary methods executed before and after the controlled method. Indeed, a full message control should be able to modify the original arguments, to change the semantics of the message as in remote-calls or even to refuse the execution of a method [DBFP95].

We identified 6 different techniques to implement message passing control. However, some of them are difficult to reproduce or lead to unportable code. That's why we briefly present and sort these techniques before describing the selected ones.

1. Source code modification. One way to control message passing is to instrument the code via source code modification and recompilation. In case of implementing a control similating CLOS-like before and after methods, a controlled method setX:setY: could look as follows after source code modification. As the object responsible for the message passing control is not necessarily the receiver itself, we use an ellipsis to represent it. For example, in the case of meta-object approaches [McA95], the receiver is not its own controller.

```
setX: t1 setY: t2
 ...before
 Original source code
 ...after
```

Note that one might try to use the method aBlock valueNowOrOnUnwindDo: anotherBlock that allows one to trap the return out of a method. This method evaluates aBlock (the receiver) and when this block exits, it evaluates anotherBlock. However, this is not appropriate, because, as we stated earlier, the execution of the controlled method can be delegated to the message passing control and not limited to additional actions like before and after method executions. Note that to simplify the presentation we will present controlling method body in case of a control simulating before and after CLOS-like methods and we will discuss how this can extended to full control.

The main drawbacks of this technique are: All controlled methods have to be reparsed and recompiled. Moreover, another recompilation is needed to reinstall the original method. This technique is not applicable in deployed or stripped images in which scanners and compilers have been removed.

2. Byte code extension. Smalltalk is based on a byte-code interpreter [GR89, IKM$^+$97], so it is possible to add new byte-code in order to introduce new message passing semantics, like in the Concurrent Smalltalk approach [YT87]. However as the resulting interpreter is no longer standard and the applications are no longer portable, we do not discuss this technique.

3. Byte code modification. Another way to control message passing is to directly insert new byte-code representing the control into the compiled method byte-codes [MB85]. However, implementing this technique is far from simple. More important, it heavily relies on knowledge of the byte code instructions used by the virtual machines. These codes are not standardized and can change.

4. Specialization of error handling. The idea is to encapsulate controlled objects into so called *minimal objects* that do not understand messages and to specialize the doesNotUnderstand: method [Pas86, Bri89, PWG93] (see section 2).

5. Exploiting the VM method lookup implementation. This is realized by explicit subclassing or by the introduction of anonymous classes in the instantiation chain [FJ89, McA95, Mic96, Riv97], or by the definition of a method dictionary array in VisualSmalltalk [Bec93b, Bec93a, Pel96] (see section 3).

6. Method substitution. The idea is to change the compiled method associated to the selector in class method dictionary [BH90, Bra96, Riv97] (see section 4).

The three last techniques can be implemented from within the language itself at a reasonable level of abstraction and, they are portable. That's why we only will present and compare them in detail in the following sections.

Note that we take into account only those techniques that introduce a control of *standard* message passing from the language itself. The key point here is that we want to control objects already defined in the Smalltalk language. Therefore we exclude approaches based on meta-interpreters that define their own explicit message sending [Coi90].

Remark. Message reification allows a particular interpretation of the message semantics such as asynchronous messages [Fer89]. However, message reification on its own does not allow one to control specific objects [Fer89, DBFP95]. Moreover, as mentioned by Adele Goldberg in [GR89] message reification has only been introduced in Smalltalk for error handling due to efficiency reasons. Nevertheless, the combination of message reification and instance-based control techniques offers a wide range of possibilities. For example, in CodA message passing control is implemented using the technique 5, but the message reification provided by the technique 4 is also used for the various message semantics offered in CodA [McA95].

1.4 Some Comparison Criteria

To compare the techniques on a common basis we propose the following comparison criteria.

Control granularity. Sometimes it is necessary to only control one specific message sent to one specific object. In other cases, all the messages sent to a set of objects should be controlled (note that objects can share the same message passing control definition without belonging to the same class).

So a control can be applied to all the instances of one class in a similar manner, or only to certain instances, or only one instance. We call the first possibility a *class-based control*, the second a *group-based control* and the third one an *instance-based control*.

Moreover, we qualify a control as *global* if *all* the messages sent to an instance are controlled, as *class-based* if all the methods of the class of the object are controlled and as *selective* if it is possible to only control certain specific messages.

Environment Integration. Since Smalltalk implementations offer rich programming environments, we also consider the impact of the techniques on the proposed tools. It is important to know if the browsers and their functionality (senders, implementors, messages, class references, instance variable references,...) continue to work after applying the message passing technique.

Efficiency. To compare the execution costs we consider that the code executed during the control, such as a display in a trace, to be constant for all the techniques. The cost takes into account only the mechanism used to control the invoked method. Moreover, we evaluate if the process requires methods to be recompiled during the installation of and during the reinstallation of the original methods.

Definition Cost. Finally we should mention if the proposed solution is easy to implement or if it needs quite complex mechanisms.

Glossary. *Controlling* entities (classes and methods) are those that implement the message passing control. *Original* entities are those that are normally executed in absence of control.

2 Error Handling Specialization

As presented in 1.2, when an object receives an unknown message the method doesNotUnderstand: is invoked. The technique consists of defining *minimal objects* that will encapsulate the object being controlled. A minimal object is an object for which *ideally* each message provokes an error. Note that to be viable in the Smalltalk environment such an object should possess a minimal set of methods that do not lead to an error. We use the become: primitive to substitute the object to be controlled by a minimal object that encapsulates it.

The figure 2 illustrates the message passing control: (1) the original message is sent, (2) the VM invokes the method doesNotUnderstand: and (3) the original method is executed.

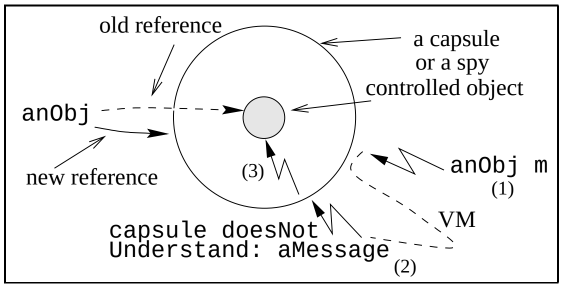

Figure 2: Installation of minimal objects and message passing control by generation and control of errors.

Note that the use of the become: primitive is only necessary when one needs to control *existing* objects of the Smalltalk library [Pas86, Lal90, PWG93, GGM95]. In [Ben87, McC87], the goal is not to control predefined objects but to define controllable objects, so the reference exchange is not necessary: messages are controlled because they are simply unknown for the object. Note that for this particular case the methods inherited from Object class should be recompiled to include control and substitute primitives calls by controllable methods [McC87].

2.1 Minimal Object

The creation of a *minimal object* [Bri89, PWG93], also named *capsule* or *encapsulator*, is based on the creation of a class that does not inherit from Object class. Doing so all the messages sent to an instance of such class invoke the doesNotUnderstand: method and then are controlled. The code to invoke the original method can be the following one:

```
MinimalObject>>doesNotUnderstand: aMessage
 ...”control specific actions”
 originalObject perform: aMessage selector
              withArguments: aMessage arguments
 ...
```

The creation of classes that inherit from nil (the unique instance of the UndefinedObject class whose value means referring to nowhere) does not lead to the desired solution. Indeed Smalltalk allows the creation of new root inheritance classes. To do so, the class creation protocol is redefined on the class UndefinedObject to permit the creation of class that does not inherit from any other class. However, to integrate such classes in the Smalltalk environment, Smalltalk defines a specialized version of the doesNotUnderstand: method that automatically and lazily copies the methods from the Object class. We then obtain an incremental copy of Object class.

The right technique to create a minimal object is the following: (1) creation of a subclass of Object, (2) assignment of the superclass link to nil and (3) definition of the minimal behavior by copying the needed methods from Object. Here follows the code taken from Actalk [Bri89].

```
MinimalObject class>>initialize
 superclass := nil.
 #(doesNotUnderstand: error: ~~ isNil =
   == printString printOn: class inspect basicInspect
   basicAt: basicSize instVarAt: instVarAt:put:)
          do: [:selector | self recompile: selec-
tor from: Object]
```

2.2 Problems

This approach implies three main problems identified by [PWG93].

The self problem. The variable self is a pseudo-variable with which objects refers to themselves without using explicit pointers. Messages that an object sends to itself are not redirected to the minimal object and thus not controlled. Moreover, this problem appears not only when an object sends messages to itself. In fact a message can only be controlled if: (1) the message is not sent by the object itself and (2) the reference from the sender of the message to the receiver of the message (the original object) was not installed via a reference to self [PWG93]. The authors of Spies [PWG93] proposed a delicate and costly solution based on the dynamic analysis of the execution stack to detect if the messages sent should or should not be controlled.

Class Control. Control of classes is impossible because classes can not be swapped by objects of different nature. The ClassBuilder uses become: when a class is incrementally defined but the swap is done between two classes.

Minimal Object. As already mentioned a minimal object should define a minimal set of methods such as class, isKindOf,=, ==, instanceVarAt:, myDependents... This leads to the problem of the interpretation by the minimal object of messages that were initially destined for the controlled object. The problem is double because not only is the message executed by the minimal object but the controlled object does not receive the message.

Pascoe proposed a heavy solution that consists in fully duplicating the inheritance hierarchy and to prefix all the methods destined for minimal objects with an E [Pas86]. Even if such a solution works well, it is heavy to set up and uses lots of memory.

2.3 Discussion

This approach proposes an *instance-based control* with a *global* granularity: all the methods are controlled. Contrary to other approaches that presuppose the knowledge of the messages that should be controlled, this approach is the only one to offer the ability to control *all* the sent messages. It is not mandatory to know in advance the potentially controllable messages.

In addition to the above mentioned problems this approach is not efficient as shown in 5.1. Indeed, the control is based on the error of the lookup of the method associated with the message. Thus each control needs one additional lookup and a double traversal of the execution stack due to exception handling. Moreover, each control implies a message instance creation.

This approach is simple to implement when one does not attempt to solve all its inherent problems such as those linked to the identity of the object.

3 Exploiting of the VM Method Lookup Algorithm

In object-oriented programming, the standard approach for specializing behavior is subclassing. In Smalltalk, when an object receives a message, the lookup of the method starts in object class and follows the inheritance link. The super variable allows one to invoke overridden methods. Its semantics is to start the lookup in the superclass of the class in which the method was found.

Controlling sent messages is possible by interposing between the object and its original class a new class that specializes the looked up methods. This can be achieved by an explicit traditional subclassing (see figure 3) or an implicit subclassing based on anonymous classes associated to each instances and a class change (see figure 4).

Common Principle. This approach is composed by three aspects: (1) creation of the controlling class that will be interposed between the object and its original class, (2) definition of controlling methods in that class and (3) class change (see in 1.2). Controlling methods should have the same selectors as the original methods.

3.1 Explicit Subclassing

The interposed class is created by invoking the class creation definition method. Moreover, an original method can be invoked by the controlling method by use of the super variable.

The newly created class can be inserted using superclass: into the class hierarchy, so the subclasses can benefit from the control of the methods. To support a control of all the instances of the class, the reference to the original class in the system dictionary class should be changed to refer the subclass.

Discussion. The control offered by this approach is a *group-based* or *class-based* control and possesses a *selective* granularity. Note that it could be possible to create as many classes as controlled instances but this would result in a proliferation of explicit classes.

The control is removed by another class change (see in 1.2). The execution cost is equal to the cost of a method execution. The main drawback of this solution is the creation of an explicit class, so this solution is not transparent from the point of view of the controlled objects.

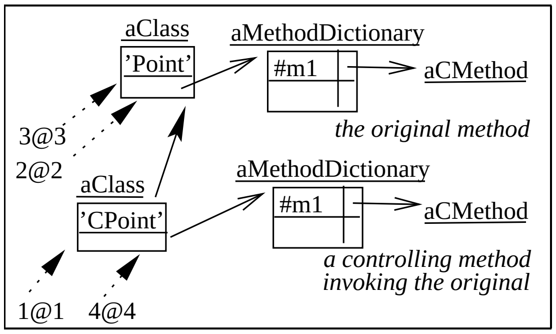

Figure 3: Explicit subclassing to control message passing. The CPoint class defines its own method dictionary containing controlled methods. Thus, 1@1 and 4@4 are controlled whereas 3@3 and 2@3 are not controlled.

3.2 Implicit Subclassing

Another solution is to interpose an anonymous class between the object and its class and to define controlling methods local to this specific object as shown by Fig. 4.

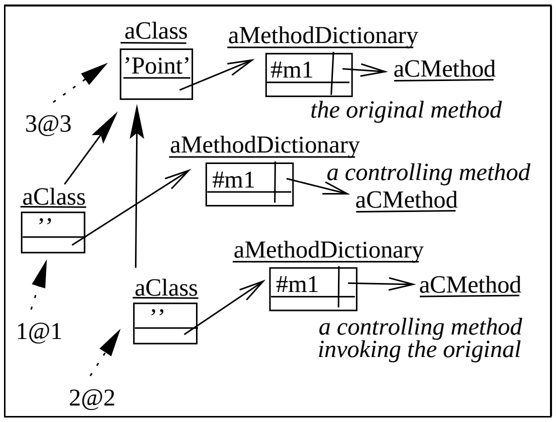

Figure 4: Implicit subclassing using anonymous classes to provide instance-based control message passing in VisualWorks.

The following steps define the control installation:

1. Create an anonymous class, nCl, instance of Behavior[2] in VisualWorks or instance of Class in IBM Smalltalk.
2. Copy the class instance description (format) from the class to nCl and assign the inheritance link of nCl to the original class of the object.
3. Change the class of the instance to refer to nCl.
4. Compile in nCl the methods that should be controlled.

[2]According to McAffer, Peter Deutsch mentioned that the class Behavior had been originally designed to allow such implementations [McA95] p. 68.

VisualWorks Implementation. A possible installation of the control is illustrated in the following example method. The line number corresponds to the previous mentioned steps.

```
Object>>specialize
    |nCl|
(1)  nCl := Behavior new
(2)          setInstanceFormat: self class format;
(2)          superclass: self class;
             methodDictionary: MethodDictionary new.
(3)  self changeClassToThatOf: nCl basicNew
```

The fourth step is implemented by invoking the method compile:notifying: of the class Behavior with a string representing the controlling method. Such a method source code can be automatically generated. In the case of a control implementing before and after CLOS-like methods, the controlling method for the method named setX:setY: could look like:

```
anAnonymousClass>>setX: t1 setY: t2
... before
   super setX: t1 setY: t2
... after
```

IBM Smalltalk Implementation. Joseph Pelrine in [Pel96] describes a similar implementation:

```
Object>>specialize
    |nCl|
(1)  nCl := Class new
(2)        superclass: self class;
(2)        instanceShape: self class instanceShape
(2)            instVarNames: self class instVarNames;
         setMethodDictionary: MethodDictionary new.
(3)  self fixClassTo: class
```

Integration and semantics of class. A good integration into the programming environment redefines locally in the anonymous class the class method. Without that the control cannot be transparent: a user could ask for the *original* class and obtain the *anonymous* class. This method can be compiled on the anonymous class as shown in the following method. Note that an access to the anonymous class is also compiled. basicCompile: is a method that invokes in a protected manner the compile:notifying: method defined in superclasses of the original class.

```
AnonymousClass>>installEssentialMethods
  self basicCompile: 'class ^ super class superclass'.
  self basicCompile: 'isControlled ^ true'.
  self basicCompile: 'anonymousClass ^ super class'
```

Invocation of the original method. The original method could be invoked from within the controlling method defined in the anonymous class. An obvious solution is to directly invoke the method using the super variable. However such a solution is only possible if the control is done by the receiver via the anonymous class implementation and not by another object like in CodA [McA95] or in FLO [DBFP95].

A possible solution in that case is to define the call to the original method via a block ([super selector args]) that will be activated later by a value method. This solution is costly because this kind of block closure cannot be optimized by the compiler. Another solution is to refer to the compiled method instance in the controlling method using the same trick as in MethodWrapper (see 4.2) and invoke directly the method (valueWithReceiver:arguments:).

When the control is done by another object (like a meta-object), the following code can be automatically generated for the original method with selector setX:setY:. Here the meta-object defines a method control:call:withArgs: that effectively does the control.

```
anAnonymousClass>>setX: t1 setY: t2
  ^ self meta control: #setX:setY:
          call: [super setX: t1 setY: t2]
          withArgs: (Array with: t1 with: t2)
```

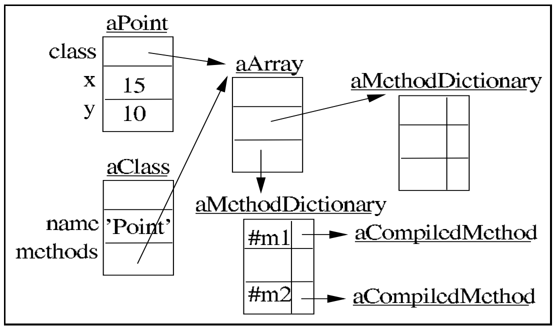

Figure 5: Relationship between instances, classes, method dictionaries and compiled methods in VisualSmalltalk: 15@10 an instance of Point does not refer to its class directly. It refers to an array of method dictionaries to which the class Point also refers to as method dictionary.

3.3 The VisualSmalltalk Solution

Contrary to VisualWorks and IBM Smalltalk, in which each object refers to its class that has a method dictionary, in VisualSmalltalk, each object refers to an array of method dictionaries. Such an array can be shared amongst all the instances of a class. Each method dictionary possesses an instance variable called class referring to the class to which it belongs as shown in 5. The method dictionaries are sorted from the class to its superclasses. This different implementation allows one to control message passing by using the VM method lookup [Bec93b, Bec93a, Pel96] as shown in fig. 6.

In VisualSmalltalk controlling a message sent to a specific instance is done by the following steps: (1) creation of a copy of method dictionary array of the object, (2) in the first place of this array addition of a new method dictionary and (3) definition of the controlling methods in this method dictionary.

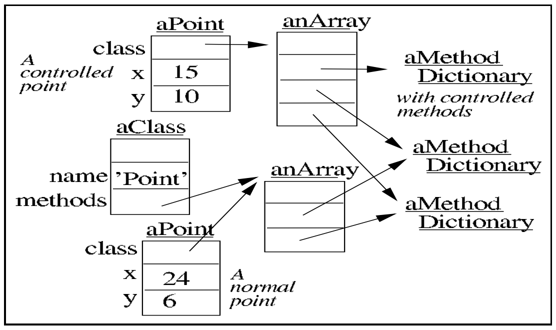

Figure 6: Instance specialization in VisualSmalltalk: 15@10 is controlled whereas 24@6 is not.

```
Object>>isSpecialized
 ^self methodDictionaryArray
  == self class methodDictionaries

Object>>specialize
 self isSpecialized ifTrue:[^self].
 self addBehavior: MethodDictionary new.

Object>>specialize: aString
 |assoc|
 self specialize.
 assoc := Compiler compile: aString in: self class.
 self methodDictionaryArray first add: assoc
```

The argument aString represents the source of a controlling method.

3.4 General Discussion

The technique based on anonymous classes is briefly mentioned in [FJ89], that qualified such classes as *lightweight* classes, and in CodA [McA95]. McAffer uses this technique to implement meta-objects and to control message passing. Ernest Micklei proposed a similar approach [Mic96]. However the meta-class is also controlled and his approach is more complex. NeoClasstalk uses this technique coupled with a method code change to implement dynamic specialization [Riv97] (see in 4.3).

These approaches support both *instance-based* control and *selective* control. Note that they can also support *class-based*, or *group-based* control by sharing the anonymous class amongst the controlled objects. Moreover, when all the instances of a given class have to share the same control, the method allInstances can be used to access to the instances of the original class.

These approaches are at the same time flexible and efficient as shown in 5.1. The lookup and execution of methods defined by the VM are used at their optimum. As the control is not based on method lookup failure, the cost is only one additional method execution. However, these techniques can only control methods that are known in advance to be controlled.

The implementation of these approaches is relatively simple and adaptable in the various dialects. However, an error during the installation can irreparably break the system. Indeed method dictionaries and format of the instances are crucial information for the VM. Moreover, method compilation is not necessary to install the control because the controlling methods can be copied and installed from predefined method skeletons (see 4.2). Therefore these techniques have a good installation speed and can be applied on deployed applications.

Finally, as a last important point, these methods do not raise the problem of object identity because the receiver of a controlled message is the object itself (see in 2.2).

4 Method Substitution

In Smalltalk, the methods defined in a class are stored in a method dictionary associated with the class. Such a dictionary associates each method selector (a symbol) with an instance of class CompiledMethod as shown in fig. 1.

As shown in figure 7, changing the compiled method associated with a selector supports message passing control. TRACER [BH90] and MethodWrappers [Bra96] use this technique. NeoClasstalk [Riv97] generalizes it. The original method can be simply stocked in the method dictionary associated with another symbol as in TRACER or it can be encapsulated in the controlling method like in MethodWrappers.

4.1 Hidden Methods

Another technique to control message passing is to associate a new selector (Xm1 in Fig. 7) with the original method and to associate a controlled method with the original method selector (m1) in the method dictionary. In case of before and after CLOS-like methods a controlling method could be schematically as:

```
aClass>>setX: t1 setY: t2
  ...before...
  self XsetX: t1 setY: t2
  ... after....
```

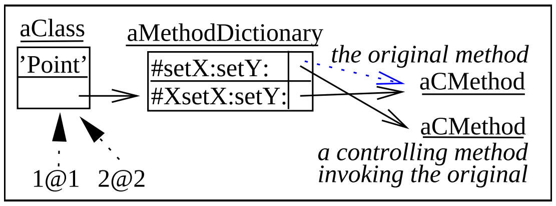

Figure 7: Addition of a new selector that refers to the controlled method and association.

As compiled methods do not refer explicitly to their selector, it is not necessary to recompile the methods when they are associated with different selectors. Moreover, the installation of the controlled methods can be done by copying method skeletons and changing some method information: if we compare two controlling methods, the only difference is that they send different selectors to invoke their original methods. The selector that is used for such an invocation can be easily changed by replacing it in the method's literal frame. Therefore, to install a controlling method from a skeleton one only needs to change the selector, to set up the mclass instance variable to refer to the class (see 1.2) and to change the source code to refer to the source code of the original method.

4.2 MethodWrappers

The previous solution has the serious drawback of introducing new selector-method associations in the method dictionary and to polluting the interface of the controlled object class. Although it is unlikely that a user will invoke a hidden method, this solution is not good when inspecting the system. MethodWrappers is a clever approach that does not stock the original methods in the method dictionary of the controlled objects class but in the compiled methods themselves [Bra96]. Instead of creating a new association selector-compiled method, the original method is substituted by a method that encapsulates the original one – the wrapper has a reference to the original method as shown in Fig. 8.

4.2.1 Definition.

The following code describes the class MethodWrapper subclass of CompiledMethod. The instance variable clientMethod refers to the original method and selector represents the original method selector.

```
CompiledMethod variableSubclass #MethodWrapper
  instanceVariableNames: 'clientMethod selector'
  classVariableNames: ''
  poolDictionaries:''
  category: 'Method Wrappers'
```

As shown by the control of the method color of the class Point below, the class method on:inClass: returns a wrapped method that can further be installed on a compiled method by invoking the method install.

```
(MethodWrapper on: #color inClass: Point) install
```

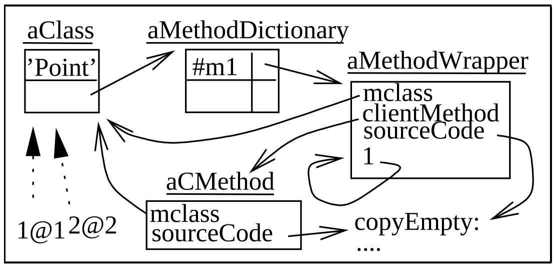

Figure 8: After installation: the original method is encapsulated into a method wrapper.

MethodWrapper class also specializes the method valueWithReceiver:arguments to introduce message passing control as follows. Note that in such a case the control is limited to before and after method executions implemented by helper method beforeMethod and afterMethod.

```
WrapperMethod>>valueWithReceiver: anObject arguments: args
  self beforeMethod.
  ˆ [clientMethod valueWithReceiver: object
              arguments: args]
    valueNowOrOnUnwindDo: [self afterMethod]
```

A controlling method definition ensures that the method valueWithReceiver:arguments: is called. The following method source, that is automatically generated, shows how the arguments are managed.

```
aClass>>originalSelector: t1
|t2|
(t2 := Array new: 1) at: 1 put: t1.
ˆ#() valueWithReceiver: self arguments: t2
```

When a message is sent to an object, it is necessary to invoke certain methods on the method wrapper itself (like valueWithReceiver:arguments in the previous code). But Smalltalk does not offer a pseudo-variable to refer to the current invoked method. Instead of using the thisContext pseudo-variable that costly reifies the method execution context, the author of MethodWrappers modifies the literals of the method wrapper. He uses the #() literal object in the previous code to reserve place to put a reference during the installation to the method wrapper itself. Note that using the self pseudo-variable in the source code of the prototype shown above was not the right solution because self represents the object on which the method was invoked and not the method itself.

As in the hidden method approach, MethodWrappers do not need to be compiled to be installed. The controlling method can be copied from a method skeleton having the same number of arguments. Then, the mclass instance variable, the literal and the clientMethod should be set. Moreover, to be fully and transparently integrated in the Smalltalk environment, the source code of the controlling method references the source code of controlled one as shown in Fig. 8.

4.3 NeoClasstalk

NeoClasstalk is a new implementation of Smalltalk that introduces explicit meta-classes [Riv97]. NeoClasstalk allows the definition of class properties such as method trace, instance variable access trace and pre- and post- conditions. These properties are based on a *controlled* modification of method source code. It proposes a framework for the composition of the different control policies. A meta-programmer can specify a part of the method source code that will be automatically compiled in the controlled methods.

The NeoClasstalk implementation uses similar techniques to MethodWrapper (prototype and literal modification) but gives the control to the class. Moreover, NeoClasstalk uses a dynamic change of class based on the definition of anonymous classes (as shown in 3.2).

Control Definition. In NeoClasstalk the execution of a method is invoked by the method execute:receiver:arguments: defined on the class AbstractClass. The *definition* (source code) of this method is defined by the method generateBodyOn: of the class TemporalComposition.

Let us suppose that we want to define a message passing control that realizes a trace of the invoked methods. To do so, we define a new class TraceAllMessages (subclass of TemporalComposition) and we specialize the method generateApplyBodyOn: that controls a part of the method source code generation of execute:receiver:arguments:. The following code shows the addition of the textual definition (source code part) of a trace to the normal method definition. The last line ensures that the normal behavior of the method will be added in this definition.

```
TraceAllMessages>>generateApplyBodyOn: aStream
  aStream nextPutAll: ' | window |
     window := self transcript.
                 cm printNameOn: window.
                 window cr; endEntry.'.
  super generatedApplyBodyOn: aStream
```

To control the class Point one should invoke the temporalComposition: method as follows:.

```
TraceAllMessages new temporalComposition: Point.
```

TraceAllMessages new creates an implicit class with method wrappers. temporalComposition: Point changes the class of the class Point so that it will be instance of the class TraceAllMessages.

A part of the Framework. As shown below, the method applyMethod defined on the class TemporalComposition specifies the definition of the source code of the method execute:receiver:arguments:. A part of this definition is under the responsibility of the method generateApplyBodyOn:. The method applyMethod ensures a semantic context of the generated method such as the insurance that the original method will be invoked (as shown by the message super execute:... below).

```
TemporalComposition>>applyMethod
"rec is the receiver, args are the arguments
 of the method, cm is the currently reified method"

 | ws |
ws := (String new: 100) writeStream.
ws nextPutAll: 'execute: cm receiver: rec arguments: args';crtab;
   nextPutAll: "system generated method";cr;crtab.
self    generateBodyOn:ws.               "<- the method to override"
^ ws contents
```

```
TemporalComposition>>generateApplyBodyOn: aStr

  aStr crtab;
       nextPutAll: '^ super execute: cm receiver: rec arguments: args'
```

Note that by changing the method generateApplyBodyOn: it is also possible to change the complete semantics of the control.

4.4 Discussion

These techniques possess a *class-based* control and a *selective* granularity. Indeed all the instances of a class are controlled without the ability to select them. The control execution cost is the cost of a method execution.

The first solution based on the definition of new association selector/method in the method dictionary polluted the interface of the objects. This problem does not appear with the other approaches. NeoClasstalk takes in charge the recompilation of the methods and proposes a well defined context for the definition and the composition of the method control. However, its solution is complex, and this complexity is not due to the concepts used as the automatic recompilation, but by the framework definition based on explicit meta-classes[3]. Contrary to the other approaches the reproduction of the mechanism is difficult.

Finally, contrary to the approach based on identity change, the main advantage of message passing control by means of anonymous classes (see in 3.2) or method wrappers (MethodWrappers and NeoClasstalk) is that the tools defined in the browsers such as (implementors, senders...) continue to fully function.

[3]Note that NeoClasstalk proposes tools for selecting class properties that simplifies the life of the lambda programmer.

5 Summary and Conclusion

Before presenting how other object-oriented languages support message passing control, we summarize and compare the techniques.

5.1 Overview

The following table gives a quick overview of the presented techniques in terms of the criteria defined in 1.4. We present here only the main or default characterics. For a deeper analysis, the reader should refer to the previous discussions. The *entity* column refers to the granularity of the control that states which entities can be controlled, the *message* criteria shows if all or some messages can be controlled, the last criteria establishes if the solution is well integrated in the Smalltalk environment in terms of browser functionality (senders and implementors) and transparency from the user point of view.

Technique	*entity*	*message*	*integration*
Error handling	instance-based	global	average
Explicit Subclassing	group-based	selective	average
Anonymous Class	instance-based	selective	good
Hidden Methods	class-based	selective	bad
Method Wrapper	class-based	selective	good
NeoClasstalk	class-based	class	good

The next table compares the different approaches for the runtime overhead. These tests were performed on a Power Mac 7100/166 with 24MB memory using Visualworks2.5. The results are the mean over five series of 10000 calls with 0,1,2 and 3 arguments. Moreover, during our numerous tests such results show some variability, therefore we consider that a difference up to 10 milliseconds is not really significant.

Technique	0	1	2	3
Explicit Subclassing	40.0	40.0	46.6	39.8
Anonymous Class	40.0	40.2	43.2	43.2
Hidden Methods	40.0	43.2	43.2	43.4
Method Wrapper	200	233	243.4	250
Inlined Method Wrapper	100	126	140	153
Error handling	213.4	229	233.4	240

As we can expect, the comparison shows that the techniques based on the explicit and implicit subclassing (anonymous classes) have the same overhead. Moreover, these two techniques have the same overhead than the technique based on hidden methods. It shows that the lookup of the method via super in the two first approaches is equivalent to the lookup via self in the hidden method approach. This is not surprising in presence of method cache mechanisms performed by the Virtual Machine. This comparison shows that the technique based on error handling is five times slower. The method wrapper approach has the same cost. This situation comes from the fact that method wrappers must create arrays for their arguments and that in our tests we do not remove the call of the valueNowOrOnUnwindDo: method.

As an experiment, we change the Method Wrapper's implementation, the controlling method continued to call the method valueWithReceiver:arguments: but we remove the call to the method valueNowOrOnUnwindDo:. The results, named Inlined Method Wrapper are two times faster than the normal Method Wrapper. Moreover, this approach could be optimized by inlining in the call inside the controlling method body instead of calling the method valueWithReceiver:arguments: defined on the class MethodWrapper.

5.2 Message Passing Control in Other Languages.

CLOS is the object system integrated into Common Lisp. It is one of the few class based languages to offer the ability to define instance specific methods using the *eql specializer*[Kee89]. Moreover CLOS is also one of the rare languages to provide a meta object protocol (MOP) in which message passing control is an entry point [KdRB91].

In CLOS the message passing concept is replaced by the generic function[4] The CLOS MOP allows one to control all the aspects of the generic function application: the application of the generic function (compute-discriminatingfunction), the application of the effective method (compute-effective-method-function) or the application of a single method composing the effective method (compute-method-function).

In the prototype based languages, Moostrap allows a message passing control based on the definition of a reflective protocol: object meta-object is responsible for the method lookup and application

[4]A generic function is a group of methods. During the application of a generic function, methods from that group are selected to constitute an effective method application. This is the effective method that is executed.

[MC93].

In the realm of less flexible languages, the definition of OpenC++ -that can be perceived in its last version as an open compiler [Chi95] - shows the interest for a control of message passing. More recently, the definition of MetaJava offers the ability to control message passing in Java [Gol97]. In this implementation anonymous classes called *shadow classes* are interposed between the instance and its original class (see in 3.2). However, in the new version called MetaXa, the interpreter is extended by the introduction of new byte-codes. As a direct consequence MetaXa's applications are no longer portable.

Java in its newest version 1.1 reified certain aspects of the language such as the classes, the methods and the instance variables (see Core Reflection API [Fla97]). However, this reification is *only* introspective reflection. Indeed, the classes Field, Method and Constructor are declared as final. This implies that they cannot be specialized. Moreover, only the Java VM can create new instances of these classes. Only the value of the instance variables can be modified and the methods can be invoked using the handle() method. Such an approach was necessary to offer tools comparable to the Smalltalk browsers in Java. However, this reification is not causally connected to the language. There is no possibility to modify the methods or the classes from within the language itself. This means the reflective facilities are not really adapted to extend or modify the language.

5.3 Conclusion

This comparison highlights that the most commonly used technique based on the specialization of the doesNotUnderstand: method is not the best one. As a first explanation of this situation, one should note that the ability to directly execute a method has only lately been introduced in the interpreters (methods valueWithReceiver:arguments: on CompiledMethod class in VisualWorks and executeWithReceiver:andArguments: in IBM Smalltalk). Moreover, this comparison shows that the techniques based on VM lookup method or method wrappers should be considered by more programmers than it is currently the case.

The reflective aspects of Smalltalk and their causal connection to the language itself offer strong advantages for the language extensions or modifications [5]. We illustrate them by showing how message passing control is possible by different approaches. This study shows the power offered by languages like CLOS or Smalltalk that provide reflective facilities that are not limited to introspective reflection like in the new version of Java (1.1).

[5] A reflective aspect of a language is said causal if any change in the reified aspect immediately influences the represented aspect and conversely.

Acknowledgments. The author would like to thank J. Brant, P. Cointe, M. Fornarino, J. McAffer, E. Micklei, O. Nierstrasz, F. Pachet, J. Pelrine, M. Rieger and F. Rivard.

References

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *ECOOP'92, LNCS 615*, pp 372–395, 1992.

[Bec93a] K. Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[Bec93b] K. Beck. Instance specific behavior: How and why. *Smalltalk Report*, 2(6), Mar 1993.

[Bec95] K. Beck. A modest meta proposal. *Smalltalk Report*, July/August 1995.

[Ben87] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. In *OOPSLA'87*, pp 318–330, 1987.

[BH90] H.-D. Böcker and J. Herczeg. What tracers are made of. In *OOPSLA/ECOOP'90*, pp 89–99, 1990.

[BI82] A. H. Borning and D. H. Ingalls. Mutiple Inheritance in Smalltalk-80. In *Proc. of NCAI AAAI*, pp 234–237, 1982.

[Bra96] J. Brant. Method Wrappers. http://st-www.cs.uiuc.-edu/users/brant/Applications/MethodWrappers.ht-ml, 1996.

[Bri89] J. Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *ECOOP'89*, pp 109–129, 1989.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, pp 285–299, 1995.

[Coi90] P. Cointe. The ClassTalk System: A Laboratory to Study Reflection in Smalltalk. In *OOPSLA/ECOOP'90 Workshop on Reflection and Metalevel Architectures*, 1990.

[DBFP95] S. Ducasse, M. Blay-Fornarino, and A. Pinna. A Reflective Model for First Class Dependencies. In *OOPSLA'95*, pp 265–280, 1995.

[Duc97] S. Ducasse. Des techniques de contrôle de l'envoi de message en smalltalk. *L'Objet*, 3(4), 1997. Numero Special Smalltalk.

[Fer89] J. Ferber. Computational reflection in class based object oriented languages. In *OOPSLA'89*, pp 317–326, 1989.

[FJ89] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *OOPSLA'89*, pp 327–336, 1989.

[Fla97] D. Flanagan. *Java in a Nutshell*. O'Reilly, 2nd edition, 1997.

[GGM95] B. Garbinato, R. Guerraoui, and K. Mazouni. Implementation of the GARF replicated objects plateform. *Distributed Systems Engineering Journal*, Mar. 1995.

[Gol97] M. Golm. Design and Implementation of a Meta Architecture for Java. Master's thesis, IMMD at F.A. University, Erlangen-Nuernberg, 1997.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, 1989. ISBN: 0-201-13688-0.

[Hop94] T. Hopkins. Instance-Based Programming in Smalltalk. Esug Tutorial, 1994.

[IKM+97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *OOPSLA '97*, 1997.

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

[Kee89] S. E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison-Wesley, 1989.

[Lal90] W. Lalonde. *Inside Smalltalk (volume two)*. Prentice Hall, 1990.

[LG88] W. R. LaLonde and M. V. Gulik. Building a Backtracking Facility in Smalltalk Without Kernel Support. In *Proceedings of OOPSLA'88*, pp 105–122, 1988.

[MB85] S. L. Messick and K. Beck. Active Variables in Smalltalk-80. Cr-85-09, Tektronix, Computer Research Lab., 1985.

[MC93] P. Mulet and P. Cointe. Definition of a reflective kernel for a prototype-based langage. In *ISOTAS'93, LNCS 742*, pp 128–144, 1993.

[McA95] J. McAffer. *A Meta-Level Architecture for Prototyping Object Systems*. PhD thesis, University of Tokyo, 1995.

[McC87] P. L. McCullough. Transparent Forwarding: First steps. In *OOPSLA'87*, pp 331–341, 1987.

[Mic96] E. Micklei. Spying messages to objects. Esug Tutorial, 1996.

[Pas86] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *OOPSLA'86*, pp 341–346, 1986.

[Pel96] J. Pelrine. Meta-level programming in smalltalk. Esug Tutorial, 1996.

[PWG93] F. Pachet, F. Wolinski, and S. Giroux. Spying as an Object-Oriented Programming Paradigm. In *TOOLS EUROPE'93*, pp 109–118, 1993.

[Riv96] F. Rivard. Smalltalk : a Reflective Language. In *REFLECTION'96*, pp 21–38, 1996.

[Riv97] F. Rivard. Evolution du comportement des objets dans les langages à classes réflexifs, 1997. Ecole des Mines de Nantes, Thèse de l'Université de Nantes.

[Sch96] B. Schaeffer. Smalltalk: Elegance and Efficiency. Ecoop Tutorial, 1996.

[YT87] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. In *OOPSLA'87*, pp 406–415, 1987.