Spying as an Object-Oriented Programming Paradigm

François Pachet
LAFORIA
Université Paris 6
4, place Jussieu
75252 Paris Cedex 05,
France
pachet@laforia.ibp.fr
http://www-laforia.ibp.fr/~fdp

Francis Wolinski
Informatique CDC
Caisse des Dépôts et
Consignations
113, rue Jean Marin Naudin
92220 Bagneux, France
wolinski@icdc.fr

Sylvain Giroux
LICEF
Télé-université,
1001 rue Sherbrooke est,
Montréal H2X 3M4 Canada
giroux@teluq.uquebec.ca

Abstract
We introduce spying, a novel way of programming with objects, based on capsule programming and reflective facilities. This programming style allows easy building of monitoring systems, such as tracers, debuggers. We point out three main problems related to this programming style, and propose practical solutions to some of them. We exemplify our claims with a system that performs master/slave communication across different Smalltalk images. We conclude by proposing a typology of applications where the spying paradigm may be productively used.

Key-words: Capsules, Proxies, Monitoring systems, Reflexion.

1. Introduction: a state of the art in capsule programming ?

Back in 1986, Pascoe introduced encapsulators [Pascoe 86], a paradigm for object-oriented programming that allowed better structuring of programs. Encapsulators, also called *capsules* or *interceptors* [Lalonde&Pugh 91], are objects that "wrap" around arbitrary objects, and redefine some of their behavior in a non intrusive way. The main idea behind capsules is their ability to redefine message send (or, rather, message reception). This has traditionally been implemented using a particularly popular mechanism of Smalltalk, the `doesNotUnderstand:`, which is considered the main reflective feature of Smalltalk [Foote & Johnson 89]. Thanks to this mechanism, capsules can easily intercept incoming messages to encapsulated objects, and redefine their semantics in various ways. Since then, capsules have steadily gained attention, and are now used intensively, although under different names, and with slight variations.

For instance, the VisualWorks environment makes intensive use of so-called *wrappers* [VisualWorks 94] in a effort to automate interface programming in MVC. Wrappers provide a useful programming framework, mainly because they automate dependency management, and handle the ubiquitous "changed" message of classical MVC. Following a similar trend, the use of *pluggable adaptors* in VisualWorks tends to systematize the use of intermediary objects that define generic "monitoring" behavior. However, wrappers and pluggable adaptors are "hard-wired" in the sense that they do not rely on the `doesNotUnderstand:` mechanism.

Similarly, Trevor Hopkins [Hopkins 94] proposes wrappers as the main ingredient for his "instance-oriented" programming style, and justifies his claims by non trivial frameworks such as the three-dimensional objects framework. Wrappers *à la* Hopkins make intensive use of the `doesNotUnderstand:` mechanism. On the actor scene, the Actalk system [Briot 89] proposes an implementation of actors as capsules. Basically, in Actalk, actors are capsules that intercept incoming message to interpret them using the mail box paradigm of actor languages. The Actalk system is not a toy system either, and now includes a complete programming environment [Briot 94].

However, this programming style is still considered a fashionable "hack" for experts, and has not yet acquired a first-class status among the object-oriented programming paradigms. Another observation about the current use of capsules and their derivatives is that capsules are still used statically. In particular the installation of capsules around arbitrary objects is most of the time under the programmer's responsibility, and requires a specific line of code to be inserted in the right place. Also, the use of capsules has not yet been entirely described. In particular, the drawbacks of this programming style have not yet been fully understood, or, at least, listed extensively.

In this paper, we propose an extension of the basic capsule programming technique, called *spying*, which relies on the systematic use of capsules, reflective capacities of the host language (the `doesNotUnderstand:`), as well as a scheme for the dynamic installation of capsules. We point out three main problems related to this programming style, and propose practical solutions for some of them, based on our experience with spying. We give a non trivial use of spying with a system that performs master/slave communication between Smalltalk images. We conclude by proposing a typology of applications where we think spying can be safely and productively used.

Requirements of the host language:

The architecture we present here is based on three requirements for the host object-oriented language:

1) possibility of redefining message execution.

2) possibility of accessing and modifying references to an arbitrary object

3) access to the stack of execution.

In practice, Smalltalk is a good candidate, thanks to the `doesNotUnderstand:`, `become:` operations, and the pseudo-variable `thisContext`. However, the proposed architecture is directly applicable in languages satisfying the three requirements above (such as some Lisp-based object-oriented languages, or Self). All the code described here is available on request.

2. Spying

We introduce an other terminology for capsules, called "spies". As we will see, spies are very similar to capsules, the only difference being the fact that they may be installed dynamically without the intervention of the programmer, and that they induce a particular philosophy of programming.

2.1. The spying philosophy

The spying architecture we present here is applicable to a wide range of systems, including tracers, debuggers, monitoring systems, advisor systems, etc. This class of system is defined by the two following hypothesis:

1 - Spying systems as **extensions** of host systems

We try to build a spying system as an extension of an existing system, called the *host system*. The strongest implication is that we do not want to modify the code of the host system.

2 - Only **external** events occurring to an object are interesting

We are interested to track all external events occurring to arbitrary objects of the host systems. Typically, in the case of advisor systems, we are interested in tracking the user actions with the host system, such as where and when he click, and which tools he manipulates. More generally we need to know what external actions a given object receives. As we will see, this notion is not straightforward, and does not simply boils down to the traditional public/private distinction.

[Böcker & Herczeg 90] proposed a toolkit with similar goals in mind. Their system (Trick), written in Smalltalk, may be seen as a *framework* to build tracers and debuggers. We do not here propose any library or system, but rather propose spying as a programming *paradigm*, useful to build the class of system described above.

2.2. Implementation

The basic spying mechanism is implemented by a combination of capsule programming and reflective facilities provided by Smalltalk-80. The idea is to intercept incoming messages by substituting special objects, called *spies* to the spied object. Using the reflective "trap-door" offered by the `doesNotUnderstand:` mechanism [Foote & Johnson 89], spies systematically reify all their incoming messages. We redefine message interpretation for spies so that they 1) execute some spying action and 2) redirect messages to the spied object, so that the system behave as if nothing happened.

The spying mechanism relies therefore on two steps:

1) an *installation mechanism* that installs a spy on a target object,

2) a *message interception* that relies on the reflexive capacity of the underlying language.

Practically this is realized by introducing a class called `Spy`, which defines a creation method for installing a spy on an arbitrary object, and an instance method for redefining message interpretation.

2.3. Installation of spies; the become: primitive

The capsule mechanism introduced by Pascoe consists in substituting capsule objects (here spies) to spied objects. In Pascoe's view, however, this substitution is left to the responsibility of the tracing mechanism, usually at creation time.

Practically, capsules require a *modification of methods* that actually create the objects to be spied. Encapsulation therefore may not be performed on existing code without modification. We propose to automate the creation of capsules, by a spying mechanism which automatically encapsulate objects. This mechanism is based on the systematic use of the Smalltalk primitive `become:`. This primitive message, defined in root class `Object`, swaps the internal addresses of two arbitrary objects. Although not documented, this method is used by the system in special cases, typically for growing collections. Here, the idea is to encapsulate objects by making them physically "become" spies, which in turn point to the original object. Note that the `become:` primitive does not modify the objects that are being swapped. It simply swaps the reference of the other objects in the environment to either of the two swapped objects (Cf. Fig. 1). Thanks to this mechanism, we can encapsulate objects "from the outside", without redefining existing code.

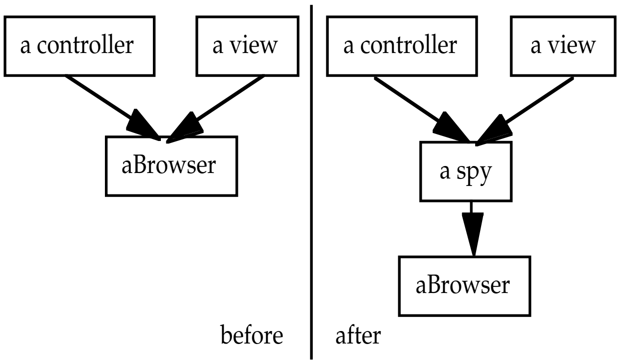

Figure 1. Installing a spy.

2.4. Objects that do not understand anything: `MinimalObject`

The hack for redefining message interpretation in Smalltalk is now well known: it consists in creating object that systematically raise an error, then redefining the `doesNotUnderstand:` message to implement the new message interpretation. We call the *MinimalObject problem*, the process involved in the definition of a class whose instances understand no message, and raise a `doesNotUnderstand:` message. This class is called *MinimalObject*, after the works of J.-P. Briot for actor languages [Briot 89].

As it turns out, creating objects that do not understand anything is not as simple as it seems. In Smalltalk, the idea is to build classes whose superclass is `nil`. From a practical point of view, there are lots of difficulties arising from that, such as problems with cross-references, or desynchronisations of change log files (Cf. [Pachet & al. 94] for more details). Spy classes are defined as subclasses of `MinimalObject`, with one instance variable pointing to the spied object, and redefine the `doesNotUnderstand:` message:

```
MinimalObject subclass: #Spy
  instanceVariableNames:
'spiedObject'
```

2.5. The script of the spy

In order to materialize the interception of messages, we introduce a special class that represent intercepted interactions. This class looks like the class `Message` (which represents reified, not understood messages), but adds time and sender information. Actually two different classes of interaction are created, to take into account the fact that once the message is executed, its *result* may be of some interest to the spy.

This script is defined as follows, to allow maximum flexibility. This interception behavior is the most general one, as it allows the insertion of a monitoring event both before and after the message is executed.

```
!Spy methodsFor: 'script'!

doesNotUnderstand:  aMessage
| r |
self scriptBefore:
      (InteractionBefore  message:
aMessage sender: self you time: Time
now).
r := self performMessage: aMessage.
self scriptAfter:
      (InteractionAfter  message:
aMessage sender: self you time: Time
now result: r).
^r
```

The spy, of course, redirects the message to the object it was sent in the first place, so that everything works as if the spy was not there:

```
performMessage:  aMessage
  ^spiedObject  perform:  aMessage
selector  withArguments:  aMessage
arguments
```

By default, the methods that actually perform some spying action do not do anything, and will be defined in concrete subclasses. Note also that the sender information is accessed via message `you`, that may be defined by introspecting in the stack (Cf. section 3.1.3 for the stack management).

2.6. Examples of subclasses of Spy

The most basic subclass of spy we can imagine is the `TranscriptSpy`, that systematically writes in the Transcript all intercepted messages. This is trivially realized by defining a subclass of `Spy` which redefines only one method, `scriptBefore:` as follows:

```
Spy subclass: #TranscriptSpy
```

```
scriptBefore:  anInteraction
  Transcript show: anInteraction
printString; cr
```

This `TranscriptSpy`, combined with an instance browser is already a very useful tool to understand the dynamic properties of objects in the Smalltalk environment.

A more elaborate kind of spy is the `SelectiveSpy`, which intercepts only message declared as "interesting". This is realized by defining a subclass of `Spy` holding a `testBlock` that evaluates to a Boolean, with the interaction as argument. The scenario is once again pretty straightforward:

```
Spy subclass:
#SelectiveTranscriptSpy
  instanceVariableNames: 'testBlock'
```

For instance, we can spy a browser and be interested only in tracing the instance/class switch. This amounts to creating a `SelectiveSpy` that handles only interactions whose selectors are `meta` or `meta:` (the messages sent to the browser when the user presses the switch) :

```
| b |
b := Browser new on: Smalltalk
organization.
DynamicSelectiveSpy newOn: b master:
self testBlock:
   [:i | #(meta meta:) includes: i
message selector].
Browser openOn: b withTextState: nil
```

The script for this spy consists simply in testing the `testBlock` prior to the writing in the Transcript :

```
scriptBefore:  anInteraction
(testBlock value: anInteraction)
  ifFalse: [^nil].
Transcript show: anInteraction
printString;cr.
```

An third interesting variation is the `RuleBasedSpy`. This kind of spy executes a particular action when it intercepts a particular message. Although we developed this idea by providing a fully-fledged rule-based mechanism, we show here the idea on a simplified version of `RuleBasedSpy` dedicated to our example.

Suppose that we want to redefine locally the behavior of the browser in the following fashion: each time the user selects the "hierarchy" option in the class menu, we want a dialog box to appear and ask the user if he wants to open a hierarchy browser. If yes, then open a hierarchy browser, if no, then proceed with the original action. This idea comes simply from the observation that users tend to be mixed up with the various options in the menu, and confuse the hierarchy option with the "open hierarchy" option.

The point we want to make here is that this refinement of the original browser may be realized without having to modify the class `Browser`, nor having to write a specialized subclass. This can be simply realized by writing a specialized spy as follows:

```
DynamicSelectiveSpy
          subclass: #RuleBasedSpy

scriptBefore:  anInteraction
anInteraction message selector =
#showHierarchy
  ifTrue:
    [(DialogView confirm: 'Do you
want to open a hierarchy Browser ?')
      ifTrue:
  [object spawnHierarchy]].
^super scriptBefore: anInteraction
```

In this example, the effect of the spy is the same as the effect of a subclass. The main difference is that the refined behavior may be associated with any existing instance of `Browser`, dynamically.

3. Properties and limit

Our architecture is based on a minimal extension to Smalltalk-80, which proved to work well on concrete and non trivial applications. However, there are three major problems related to this architecture: the self problem (itself divided in two sub problems), the instance creation problem, and the minimal message problem.

3.1. The self problem

There are two variant of the self problem for spies. The first one has no solution. The second one has one, which relies on the possibility to perform introspection in the stack.

3.1.1. The self problem itself

The self problem was already mentioned by [Lieberman 86], and is inherent to the very definition of capsules. Since only incoming messages are tracked, messages send by an object to itself cannot be intercepted, because `self` is not a real variable: objects may refer "directly" to themselves, without having to use to explicit pointers. As Foote and Johnson mentioned in [Foote & Jonhson 89], capsule correspond to *message forwarding*, and not *true delegation*. From a software engineering point of view, this amounts to saying that only "public" methods can be spied, and not private ones, i.e. only messages in which the sender and the receiver are different objects could be intercepted. Although most dialects of Smalltalk do not take this difference explicitly into account (the only exception so far is Smalltalk/ENVY [ENVY 94]), it is not hard to add a public-private facility to classify methods, and check that only public methods are spied. However, the problem is trickier than what Lieberman, and Foote & Jonhson suggested: not only private methods cannot be spied, but also some public methods as the following example shows. In fact the requirement for a message to be interceptable is that:

1) the message is not sent by the object itself,
2) the reference from the sender of the message to the receiver of the message (the spied object) was not installed via a reference to `self`.

We did not find a systematic solution to this problem. Experience showed that such cases are not frequent. We currently chose to design tools that dynamically detect such situations, so that spying system designers know exactly what is the status of the methods they try to spy.

3.1.2. The public / private problem revisited

Now this distinction between public and private methods is yet more subtle. This second problem is a dual version of the preceding one: there are external messages that *should not be* intercepted, because, although they are indeed public messages, they are indirectly the result of a message sent by the spied object itself!. This is the case for instance with the `changed` messages so frequent in MVC programming. For instance, when an object needs to notify its dependents that it has changed significantly, it sends itself a "`self changed`" message. This message in turns, warns the dependents that the object has changed, which results in the dependents asking the initial object for some information [Krasner & Pope 88]. These messages are usually "public" messages, so they will be intercepted, but, in our context, they should not, since they do not represent an actual external interaction!

For instance, Figure 2 shows the list of messages intercepted when a user clicks on a category in a spied browser (note that most of them are considered public messages).

This problem is tricky, and shows the importance of having a sound definition of what public/private exactly means. More precisely what we need is a definition of private/public that is dynamic. A solution to this problem is given by introspecting in the stack. This is particularly easy to do if the language provides an access to the stack as well as object structures to organize it. In Smalltalk-80, we propose the following method that looks up the stack until a particular condition (represented by a Smalltalk block) is satisfied:

```
youUntil:  aBlock
"looks up the stack until aBlock is
true, or the top is reached"
 | s |
 s := thisContext.
 [s := s sender.
 s isNil ifTrue: [^false].
   aBlock   value:   s   receiver]
whileFalse.
 ^true
```

This stack introspection is inserted in the script of spies as follows :

```
!Spy methodsFor: 'script'!
doesNotUnderstand:  aMessage
  | r |
(self youUntil: [:x | x == object])
ifTrue:
    [^self performMessage: aMessage].
  same as before ...
```

#category: withArguments: #(#'spying-essais') from: a SelectionInListView at 3:32:18 pm
#classList from: a SelectionInListView at 3:32:18 pm
#className from: a SelectionInListView at 3:32:18 pm
#className: withArguments: #(nil) from: a SelectionInListView at 3:32:18
#protocolList from: a SelectionInListView at 3:32:18 pm
#protocol: withArguments: #(nil) from: a SelectionInListView at 3:32:18 pm
#selectorList from: a SelectionInListView at 3:32:18 pm
#selector: withArguments: #(nil) from: a SelectionInListView at 3:32:18 pm
#text from: a TextView at 3:32:18 pm

Figure 2. The messages intercepted when the users clicks on a category, and when spies ***do not*** *look up the stack.*

Thank to that modification, only actual external events are intercepted. More than simply solving the bug, this notion of systematic stack introspection yields in fact a definition of an external event: an external event is a message sent to an object, such as there is no message sent by the object itself in the current stack. This approach is to be compared with the works of [Loia & Quaggetto 93], who propose *dynamic spying*, in the form of specific language constructs to access and modify the history of computation. Our scheme is less radical that Loia's, in that we do not intent to *modify* the stack, and only want to observe it to draw information on the behavior of the system.

3.2. The class problem

The second problem is spying of dynamically created objects. For instance, let us say we want to spy all hierarchy browsers created from a given browser. The natural solution would be to spy classes, and instance creation methods. This not feasible in practice, mainly because classes can't *become* non-classes objects (this is a limitation that we can forgive to the Smalltalk interpreter!).

All object creations and deletions are *eventually* caused by messages sent to instances. Instanciation is typically performed by a message sent to a class (such as : `HierarchyBrowser new`), but, as we saw, we cannot intercept class messages. To solve this problem, we intercept the creation of objects indirectly, by tracking calling instance messages. These messages are intercepted by a special kind of spy called `CreativeSpy`, which is in charge of installing a new spy on the newly created object.

In the case of hierarchy browser creations for example, the corresponding instance message is the message `spawnHierarchy` sent to the browser instance. The method `spawnHierarchy` in turn sends a message to the class `HierarchyBrowser` :

```
Browser methodsFor: 'class functions'
spawnHierarchy
  ^HierarchyBrowser
openHierarchyBrowserFrom: self
```

Since we cannot intercept message `openHierarchyBrowserFrom:`, we will intercept the message `spawnHierarchy` sent to the browser instance. More generally, we make the hypothesis that all instanciations are "triggered", somewhere in the system by an instance message which is significant enough to be intercepted in place of the actual instance creation message. This hypothesis has yet only been verified in practice. We work towards an automatization of "causal" instance messages to help the designer find out which instance messages cause new objects to be created. An other idea in progress consists in specifying in a declarative manner "access-paths" that link "root" objects to potential new objects, with a chain of instance messages, and have the system install specific creative spies to do the job.

3.3. The MinimalObject problem

The third problem raised by our architecture is inherent to the very concept of the `MinimalObject` class. In order to survive in the Smalltalk environment, without threatening it, objects have to understand a minimal set of messages. The problem arises when spies directly interpret a minimal message that was not directed at them, but at their spied object (such as the message `class`). The problem is the two-fold: 1) the original object does not receive the message, and 2) the action performed by the spy is the wrong one. The answer provided by the spy may be inconsistent with the rest of the host system.

The actual implementation of our architecture does not provide any reasonable answer to this problem. A good solution to this problem has been proposed by Pascoe [Pascoe 86], who introduces a wholly different hierarchy of classes and metaclasses, with specially prefixed selectors (E-selectors), to avoid ambiguities. Since this solution requires a lot more code than the actual implementation, we did not initially chose it, but we will probably switch to it if the current one really proves in-tractable. What we do in practice is to provide specific browsers that trace the invocation of minimal messages (such as `class`, `isKindOf:`, etc.), so that the user is aware of the potential dangers of using such primitives in a spied context.

4. A simple example: Replayer

A more sophisticated example of the use of spies is the construction of a mini-replayer. A replayer is an object that is able to record the flow of external events to an object, and replay them in the same order, to the original object, or to an other one. This is trivially realized by defining a class `Replayer` containing the list of recorded interactions, and by installing a spy on the object to be recorded, that simply forwards the intercepted messages to the replayer object.

A simple example of the use of a replayer is to provide an undo/redo facility for browsers. By plugging a replayer on a browser, we can simply "redo" a sequence of interactions, to put the browser back in a previous state.

5. A sophisticated example: Proxies

In the context of distributed applications where different parts of an application invoke remote objects, the problem set to the programmer is the transparency of access to these objects. This is typically done with a two layer mechanism : a *reference* layer and a *communication* layer. Reference layer may be implemented by *proxies* [Decouchant 86], [Bennett 87]. A proxy is an object available in a given machine that stands for an other object living in another one. Messages that are sent to the proxy are automatically routed to the remote object. Communication layer may be implemented by a RPC library, called RPC-Talk [Wolinski 94], that we developed at CDC.

In this section, we will show 1) that proxies may be implemented as special kinds of spies and 2) that spies and proxies may be simply combined together to provide the ability of spying a remote object. *Remote spying* will be seen as a natural extension of both spies and proxies.

5.1. RPC-Talk

The RPC (Remote Procedure Call) technique gives the ability to put any number of machines at the service of an application [Bloomer 86]. This technique supplies 1) a method for specifying all services available in the slave application (the server), 2) a multi-threading management of all contexts associated to each connected master (the clients) and 3) a normalized coding of requests arguments and results using XDR (eXternal Data Representation) for network transportation.

We have implemented an ObjectWorks\Smalltalk RPC library called RPC-Talk [Wolinski 94] and available as a Manchester goodies. This library allows 1) to specify a service so that a Smalltalk client can connect to any remote server, 2) to specify and to implement a Smalltalk server so that any remote client can connect to it, 3) for Smalltalk to Smalltalk communications only, an extended-XDR coding for basic Smalltalk objects and the use of Binary Object System Storage (BOSS) to transport arbitrary complex objects.

5.2. Proxies

5.2.1. General architecture

Let us consider an application running on machine M1. A remote object `o` is available on machine M2 and is represented by a proxy `p` in M1. When an object `x` in M1 wants to send a message to the object `o`, it sends it instead to the local proxy `p` which forwards the message through the network to `o` itself. For the sender the whole operation is transparent (Fig. 3).

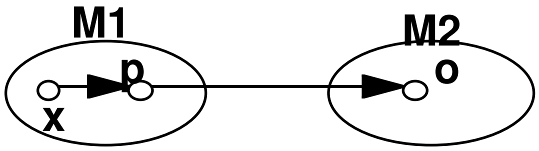

Figure 3. A message sent via a proxy.

This works fine when the argument of the message sent to `o` can be passed directly or by BOSS. A complex argument `a` (such as a model having dependent graphical objects) need to be sent as a proxy `q`. Finally, the result `r` of the message passing by object `x` to object `o` may be returned as a proxy `s` too (Fig 4).

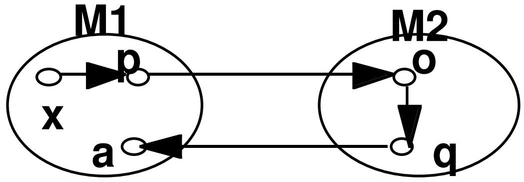

Figure 4. Treating the result as a proxy.

5.2.2. Proxies as spies

Proxies share with spies a common substitution principle : spying consists in replacing an object by a spy-object so that it receives all the messages that are sent to the original object. *Proxyfying* consists in replacing a remote object by a proxy-object so that it receives all the messages that are sent to the original one. This shared principle may be exploited to implement proxies as spies, i.e. class `Proxy` will be defined as a subclass of class `Spy`. Only the installation method is redefined for proxies to establish the connection with the remote image.

Spies and proxies have different aims, however. A spy is substituted to perform some specific actions before and/or after its performance by the main object. A proxy is substituted to allow the main object to receive its remote messages. This is trivially taken into account by overriding method `performMessage:` in class `Proxy`.

5.2.3. Implementation outlines

The common substitution principle leads naturally to similar implementations. Instead of holding directly the spied object, the proxy holds a communication-object which allows it to send actually the message to the remote object.

Our implementation of the reference layer is close to Benett's one [Benett 87]. The class `Proxy` (his `ProxyObject`) is defined as a subclass of `Spy`. Different class variables are defined to hold proxy/object correspondence table (his `RemoteObjectTable`). Note that his implementation was based on redefining the `doesNotUnderstand:` method.

Our implementation of the communication layer is based on a specific RPC service. It supplies two procedures : `send` which performs the remote message passing and `get` which returns a proxy given an known id.

Message interception is done by redefining the `performMessage:` method in the srcipt of `Proxy`. It asks the RPC client to invoke the `send` procedure with 3 arguments : proxy id, message selector and message arguments which may be proxified in the proxy image.

```
! Proxy methodsFor: 'scipt'!
performMessage:  aMessage
  ^ProxyClient
    exec: #send
    with: id
    with: aMessage selector
    with: aMessage arguments asProxy
```

5.3. Combining spies and proxies : remote spying

As we showed in the previous section, proxies are fully integrated in the spy hierarchy. More than a pure conceptual integration obtained by simple inheritance, we show in this section that spies and proxies may easily be combined to yield *remote spies*. Originally, spies were used to build non-perturbing extensions of existing systems. The opportunity to provide spies with an inter-machine communication capability allows them to delegate the actual processing of the spied information to remote objects. This is what we call *remote spying*.

5.3.1. Remote advising

A spy is plugged to an object in order to perform some specific actions before and/or after its execution by the host object. In the context of advisor systems, these actions are performed by an adviser which analyses the spied actions. These advisers are often called OSAS, i.e. "over the shoulder" adviser systems. If they are executed on the host machine, their activity may weight on the host application. In this configuration, the OSAS appellation should rather be interpreted as "on the shoulder" adviser systems ! Hence the need to have adviser running on a remote machine.

In this scheme, the spied application `x` and its spy `s` are on the machine M1 and the adviser `a` is on he machine M2. The spy is connected to the adviser through a proxy `p`. When the application holds an interaction, it is first sent to the spy `s`. The spy routes it to the proxy `p`. The proxy forwards the interaction to the adviser `a` through the communication layer. And the remote adviser can process at last the interaction. When the proxy has forwarded the interaction, the application executes it (Cf. Fig. 5).

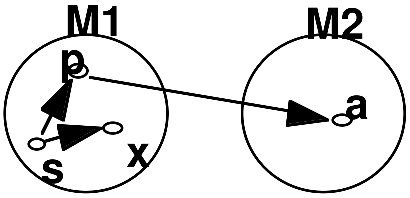

Figure 5. Remote advising.

5.3.2. Master-slave distributed applications

Some distributed applications may need a given program to run at the same time on different machines with communication facilities. Computed Aided Instruction systems may rely on such an architecture : actions of the different learners are sent back to the teacher supervisor. Another example is distributed simulations or games, in which all participants have a specific access to the same application and each action they perform has to be routed to the others.

The distributed application `x` is running on machines M1 and M2. In machine M2, the application is slaved by a slaver `v`. `Slaver` is a subclass of `Adviser` : a slaver holds an object (slave) and any interaction received is performed by the object. In M1, interactions to application `x` are intercepted by the spy `s`, routed to the slaver `v` tanks to its proxy and performed by the application in M2 (Cf. Fig. 6).

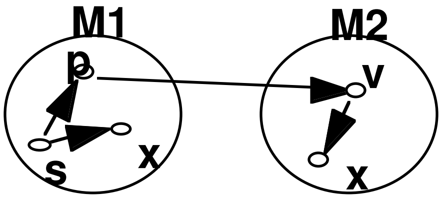

Figure 6. Proxies in a master/slave communication scheme.

6. Discussion

Spying is used in our lab as a tracing facility, mainly to understand Smalltalk programs, and find optimizations. As such, it may be seen as a companion tool to the profiler tools. RPC-Talk combined with spying is used in [Caetano 94] for building a simulator used in an integrated tutorial system. In this system, a double master/slave relationship is installed between the teacher's image and the student's one. The teacher defines scenarios in his image, which are automatically available in the student's image. Conversely, the student's actions are viewed "on-line" in the teacher's image.

Experience with spies encouraged us to think that performance is not an issue. Of course, the use of the doesNotUnderstand mechanism is not the most efficient way of intercepting messages. However, our spies are mostly used in prototyping mode. We did not find yet any occasion of complaining about the speed of spies. If such complain occurs, there are indeed other, more efficient ways of intercepting message sends, while retaining the general spying philosophy described here. [Böcker & Herczeg 90] propose to compile methods including notifiers in dynamically created subclasses, thereby avoiding the need for reifying messages. Similarly, the low-level primitives and set of flags proposed by SmalltalkAgents [Quasar 93] are very useful to customize message interpretation at the instance level. Other languages like Self allow the creation of completely dumb and deaf objects having absolutely no attributes and understanding no method (not terribly useful either), which would make the creation of `MinimalObject` much simpler. Another way of redefining interpretation of messages is to use the Meta-Object Protocol. A rewriting of the spying architecture using ClassTalk [Cointe & Briot 89] is considered using before and after metaclasses in the spirit of [Forman & al. 94] in SOM.

As we saw, spying is a means of collecting information on the dynamic behavior of a system. The next problem to solve is *what do to* with that information. For complex systems such as advisor systems, the main problem is indeed to organize information. This problem concerns a whole class of systems (that we call *epiphyte systems*, after a botanical metaphor). We designed EpiTalk [Pachet & al. 94], a framework and a system that proposes to organize spied information according to several viewpoints on the activity of the spied system. The viewpoints are materialized by lattices and are themselves used to generate the spying system automatically. Spies are then used to feed the spying system, from the interaction of a user with the host system. EpiTalk is being used in a number of tutorial systems such as DEW [Paquette & al. 94], as well as several advisor systems on top of tutorial systems written in Smalltalk. Other applications of EpiTalk include an environment for debugging actor-like languages [Giroux & Desbiens 94], and explanation-modules for expert systems. Extensions for dynamic typing of Smalltalk programs are also considered.

Based on our experience with the spying paradigm, we propose three basic categories of applications in which the spying paradigm may be relevant.

1) Spying as a means of *introspecting* programs

In this mode, spies are used to uncover hidden dynamic properties of programs. The spying system should then be *non intrusive,* i.e. the host system should behave exactly as if no spying occurred. The presence or the absence of a spy should not modify the program's behavior. This is most useful in a number of cases, such as building the replayer (section 4), but also finding optimizations of programs (a service partially offered by so-called "profilers"); understand complex programs, and so forth. Applications in dynamic typing of programs are also considered. When the analysis is executed on another machine, we call it *remote introspecting*.

2) Spying as a means of *extending* a system.

In this scheme, spies override the behavior of the spied object within the host system. This is the case of our small "extended browser" (see section 2.7). More generally, this is the case of advisor systems, seen as a module of the main application [Pachet & al. 94]. When the extended behavior is executed on another machine, we call it *remote advising* (Cf. section 5.3.1).

3) Spying as a means of *implementing* a *distributed* architecture

As we saw, spies may be specialized into proxies. Moreover, combining proxies with "normal" spies and remote advisers, yields an elegant implementation of a master/slave scheme (Cf. section 5.3.2).

7. Conclusion

We introduced spying as an object-oriented programming paradigm based on an extension of capsules with substitution operations. We pointed out three main problems related to this scheme, and proposed practical solutions to some them. We gave several examples of the application of spying, and proposed three main uses types of applications where spying is a useful programming paradigm.

References

[Bennett 87] Bennett, John K. The Design and Implementation of Distributed Smalltalk. *Proceedings of OOPSLA '87*, pp. 318-330 (1987).

[Bloomer 91] Bloomer J. Power Programming with RPC, *O'Reilly & Associates*, Inc (1991).

[Böcker & Herczeg 90]. Böcker H.-D, Herczeg J. What tracers are made of. *Proc. of OOPSLA/ECOOP '90*, pp. 89-99, Ottawa, Canada.

[Briot 89] Briot, J.-P. Actalk : A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 environment. *Proc. of ECOOP '89*, pp. 109-130.

[Briot 94] Briot, J.-P. Modélisation de classification de langages de programmation concurrente à objets. *Conference LMO (Langages et Modèles à Objets)*. Grenoble, pp. 153-166, October 1994.

[Caetano 94] H. Caetano. Représentation des connaissances dans le système informatisé de formation: Intempéries. *Seminar "Patrimoine culturel et formation"* 15-17 Sept. Ravello, Italy. Centre européen de protection du patrimoine culturel.

[Cointe & Briot 89] Cointe, P. Briot, J.-P. Programming with ObjVlisp Metaclasses in Smalltalk-80. *Proceedings of OOPSLA '89*, New Orleans, pp. 419-431, (1989).

[Decouchant 86] Decouchant, D. Design of a Distributed Object Manager for the Smalltalk-80 System. *Proceedings of OOPSLA'86*, pp. 444-452 (1986).

[ENVY 94] Smalltalk/ENVY, Reference Manual. *Object Technology International*, Otawa, 1994.

[Foote & Johnson 89] Foote, B. Johnson, R.-E. Reflective facilities in Smalltalk-80. *Proc. Of OOPSLA'89*, pp. 327-336, New Orleans, Louisiana.

[Forman & al. 94] Forman, Ira R. Danforth, Scott. Madduri, Hari. Composition of Before/After Metaclasses in SOM. *Proceedings of OOPSLA '94*, Portland, Oregon, pp. 427-439, (1994).

[Giroux & al. 94] S. Giroux, F. Pachet & J. Desbiens. Debugging multi-agent systems: a distributed approach to events collection and analysis. *Canadian Workshop on Distributed Artificial Intelligence* - CWDAI '94. Banff, Canada, May 1994.

[Hopkins 94]. Instance-Based Programming in Smalltalk. *Tutorial of the Second European Smalltalk User Group Summer school*, Cork, Sept. 94.

[Krasner & Pope 88]. Krasner, G. Pope, S. A Cookbook for using the model-view-controller paradigm in Smalltalk-80. *ParcPlace systems* (1988).

[Lalonde&Pugh 91] Lalonde, Wilf R. Pugh, John R. Inside Smalltalk, Volume I. *Prentice Hall*, 1991.

[Lieberman 86] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *Proc. of OOPSLA '86*, Portland, Oregon, pp. 214-223, 1986.

[Loia & Quaggetto 93] Loia, V. Quaggetto, M. High-Level Management of Computation History for the Design and Implementation of a Prolog System. Software Practice and Experience, Vol. 33 (2), pp. 119-150, Feb. (1993).

[Maes 87] Maes, P. Concepts and Experiments in Computational Reflection. *Proc. of OOPSLA '87*, pp. 147-155, Orlando, Florida (1987).

[Pachet & al. 95] Pachet, F. Wolinski, F. Giroux, S. From Capsules to Rule-Based Advisors, submitted.

[Pachet & al. 94] Pachet, F. Giroux, S. Paquette, G. (1994). Pluggable Advisors as Epiphyte Systems. *Calisce '94* (Computer Aided Learning in Science and Engineering), pp. 167-174, Paris, 31 Aug.-2 Sept. 1994.

[Paquette & al. 94] Paquette, G., Crevier, F. Aubin, C. Frasson, C. Design of a Knowledge-based Didactic and Generic Workbench. *Calisce '94* , pp. 303-312, Paris, 31 Aug.-2 Sept. 1994.

[Pascoe 86] Pascoe, G. Encapsulators: A New Software Paradigm in Smalltalk-80. *Proc. of OOPSLA '86*, pp. 341-346, Portland, Oregon.

[Quasar 93] Smalltalk Agents. Technical Overview. *Quasar Knowledge Systems*, Inc. (1993).

[VisualWorks 94]. VisualWorks User's Guide. *ParcPlace Systems*, 1992.

[Wolinski 94] Wolinski, F. (1994). RPC-Talk: une librairie RPC pour Smalltalk. *Laforia internal report*. 94/26.