Efficient Proxies in Smalltalk

Marcus Denker, Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse

Efficient Proxies in Smalltalk

Mariano Martinez Peck[1 2] Noury Bouraqadi[2] Marcus Denker[1]
Stéphane Ducasse[1] Luc Fabresse[2]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1
[2]Université Lille Nord de France, Ecole des Mines de Douai

marianopeck@gmail.com, {stephane.ducasse,marcus.denker}@inria.fr,
{noury.bouraqadi,luc.fabresse}@mines-douai.fr

Abstract

A proxy object is a surrogate or placeholder that controls access to another target object. Proxy objects are a widely used solution for different scenarios such as remote method invocation, future objects, behavioral reflection, object databases, inter-languages communications and bindings, access control, lazy or parallel evaluation, security, among others.

Most proxy implementations support proxies for regular objects but they are unable to create proxies for classes or methods. Proxies can be complex to install, have a significant overhead, be limited to certain type of classes, etc. Moreover, most proxy implementations are not stratified at all and there is no separation between proxies and handlers.

In this paper, we present Ghost, a uniform, light-weight and stratified general purpose proxy model and its Smalltalk implementation. Ghost supports proxies for classes or methods. When a proxy takes the place of a class it intercepts both, messages received by the class and lookup of methods for messages received by instances. Similarly, if a proxy takes the place of a method, then the method execution is intercepted too.

Keywords Object-Oriented Programming and Design » Message passing control » Proxy » Interception » Object Swapping » Smalltalk

1. Introduction

A proxy object is a surrogate or placeholder that controls access to another target object. A large number of scenarios and applications [11] have embraced and used the Proxy Design Pattern [12].

Proxy objects are a widely used solution for different scenarios such as remote method invocation [24, 25], distributed systems [3, 20], future objects [23], behavioral reflection [10, 15, 29], aspect-oriented programming [16], wrappers [6], object databases [7, 19], inter-languages communications and bindings, access control and read-only execution [1], lazy or parallel evaluation, middlewares like CORBA [13, 17, 28], encapsulators [22], security [27], among others.

Most proxy implementations support proxies for regular objects (instances of common classes) only. Some of them, *e.g.,* Java Dynamic Proxies [11, 14] even requires that at creation time the user provides a list of *Java interfaces* for capturing the appropriate messages.

Creating uniform proxies for not only regular objects, but also for classes and methods has not been considered. In existing work, it is not possible for a proxy to take the place of a class and a method and still intercept messages, in order to perform operations such as logging, swapping or remote class interaction. This weakness strongly limits the applications of proxies.

In addition, traditional implementations (based on error handling [22]) result in non stratified proxies: not all the proxified API messages can be trapped leading to severe limits, and there is no clear division between trapping a message and handling it, *i.e.,* there is no separation between proxies and handlers. Trapping a message is intercepting it, and handle a message means to do something in particular with such interception. The handling actions depends on the user needs, hence they are defined by the user of the framework. Bracha et al. [5] defined stratification in the field of reflection as the following statement: “meta-level facilities must be separated from base-level functionality”. The same applies for proxies, where instead of meta-level facilities there are trapping or intercepting facilities [27].

Another interesting property of proxy implementations is memory footprint. As any other object, proxies occupy memory and there are cases in which the number of proxies and their memory footprint becomes a problem.

In this paper, we present Ghost, a uniform, light-weight and stratified general purpose proxy model and its implementation in Pharo Smalltalk [4]. In addition, Ghost supports proxies for classes or methods. This means that it is not only possible to create a proxy for a class or a method

but also that such proxy takes the place of the target original class or method, intercepts messages without crashing the system. If a proxy takes the place of a class it intercepts both, messages received by the class and lookup of methods for messages received by instances. Similarly, if a proxy takes the place of a method, then the method execution is intercepted too. Ghost provides low memory consuming proxies for regular objects as well as for classes and methods.

The contributions of this paper are:

- Describe and explain the common proxy implementation in dynamic languages and specially in Smalltalk.
- Define a set of criteria to evaluate and compare proxies implementations.
- Present Ghost, a new proxy model and implementation which solves most of the proxy's problems in a uniform, light-weight and stratified way.
- Evaluate our solution with the defined criteria.

The remainder of the paper is structured as follows: Section 2 defines and unifies the vocabulary and roles that are used throughout the paper, and then it presents the list of criteria used to compare different proxy implementations. Section 3 describes the typical proxy implementation and by evaluating it against the previously defined criteria, it presents the problem. Section 4 introduces and discusses the Ghost model, and then evaluates the needed language and VM support. An introduction to Smalltalk reflective model and its provided hooks is explained by Section 5. Ghost implementation is presented in Section 6, which also provides an evaluation of Ghost implementation based on the defined criteria. Finally, in Section 7 related work is presented, before concluding in Section 8.

2. Vocabulary and Proxy Evaluation Criteria

2.1 Vocabulary and Roles

For sake of clarity, we define here the vocabulary used throughout this paper. We hence make explicit entities in play and their respective roles.

Target. It is the original object that we want to *proxify*, i.e. the object that will be replaced by a proxy.

Client. This is an object which uses or holds a reference on the target object.

Interceptor. It is an object whose responsibility is to *intercept* messages that are sent to it. It may intercept some messages or all of them.

Handler. The handler is responsible of *handling* messages caught by the interceptor. By *handling* we refer to whatever the user of the framework wants to do with the interceptions, *e.g.,* logging, forwarding the messages to the target, control access, etc.

One implementation can use the same object for taking the roles of interceptor and handler. Hence, the proxy plays as interceptor and also as handler. In another solution such roles can be achieve by different object where the proxy usually takes the role of interceptor.

2.2 Proxies Implementation Criteria

From the implementation point of view, there are criteria that can be taken into account to compare and characterize a particular implementation [10]:

Stratification. Stratification means that there is a clear separation between the proxy support and application functionalities. With a stratified approach, all messages sent by the application's business objects to the proxy are intercepted.

The proxy API should not pollute the application's namespace. In a truly stratified proxy, all messages received by a proxy should be intercepted. This means that the handler itself cannot send messages to the proxy. Not only the handler cannot do that, but none other object in the system. Having this stratification is important to achieve security and to fully support transparency of proxified object for the end-programmers [5].

Stratification also covers the design of the proxy. There are two responsibilities in a proxy toolbox: 1) trapping or intercepting messages (interceptor role) and 2) managing the interception (handler role), *i.e.,* performing actions once the message is intercepted. In a stratified proxy framework the first responsibility can be covered by a proxy itself, and the second one by a handler. This means that proxies are just traps to intercept messages. When they intercept a message they just delegate to a handler, which does something in particular with it, *e.g.,* logging, access control, etc. Consequently, different proxies instances can use the same or different handler instance.

Interception granularity. There are the following possibilities:

- Intercept *all* messages sent to an object, even messages not defined in the object API.
- Intercept only user defined messages.
- Intercept only messages imposed by the system.

With the last two options, there are messages that are not intercepted and hence answered by the proxy itself. This can be a problem because it is not possible to distinguish messages sent to the proxy to ones that should be trapped. For example, when a proxy is asked its class it must answer not its own class but the class of the target object. Otherwise, this can cause errors difficult to manage.

Object replacement. Replacement is making client objects to reference the proxy instead of referencing the target. Two cases can be distinguished. On the one hand, there are scenarios where some objects become new clients. So, they will get a reference to a proxy instead of the reference to the target object. For example, for remote method invocation, targets are located in a memory space different from the clients one. Therefore, clients can only hold references on proxies to interact with targets. Messages sent by clients to proxies will be handled and forwarded to remote targets.

On the other hand, sometimes the target is an already existing object which is pointed to by other objects in the system and it needs to be *replaced* by a proxy, *i.e.,* all objects in the system which have a reference on the target should be updated so that they point to the proxy instead. For instance, for a virtual memory management we need to swap out objects and to replace them by proxies. In this case, we need to retrieve all objects which were pointing to the existing unused object to now point to the proxy. We refer to this functionally as *object replacement*.

Uniformity. We refer to the ability of creating a proxy for any object (regular object, method, class, block, process. . .) and replacing the object by the proxy.

Most proxy implementations support proxies only for regular objects and *without* object replacement, *i.e.,* proxies cannot replace a class, a method, a process, etc, without crashing the system. There can be not only more *classes* that require special management but also more special objects that require so. For example objects like nil, true, false, etc.

This is an important criteria since there are scenarios where being able to create proxies for living runtime entities is mandatory.

Transparency. A proxy is fully transparent if client objects have no mean to find out whether they reference the target or the proxy. .

One of the typical problems related to transparency is the identity issue in cases where the proxy and the target are located in the same memory space. Given that different objects have different identities, a proxy's identity is different from the target's identity. The expression proxy == target will answer false, revealing hence the existence of the proxy. This can be temporary hidden if there is object replacement between the target object and the proxy. When we replace all references to the target by references to the proxy, clients will only see the proxy. However, this "illusion" will be broken as soon as the target provides its reference as an answer to a message or a parameter.

Another common problem is asking the class or a type of an object since most of the times the proxy answers its own type or class instead of the target's one. The same happens if there is special syntax or operators in the language such Javascript's "+", "/", "=", ">", etc. In order to have the most transparent possible proxy, these situations should be handled in such a way that the proxy behaves like the target.

Now the question is whether the identity of an object should be controlled similarly to central messages such as class. We believe that most of the time it is important that the identity is treated similarly to messages, since code working based on object identity should work the same whether the object has been proxified or not. Now depending on the language or optimization in place, identity is not treated as a message but provided as a built in primitive, which means that it can be difficult to offer proper identity swapping.

Efficiency. Proxy handling must be efficient from both points of view: performance and memory usage. In addition, we can distinguish between installation performance and runtime performance. For example, for installation, it is commonly evaluated if a proxy installation requires extra overhead like recompiling.

Moreover, depending on the usage, the memory footprint of the proxies can be fundamental. It is not only important the size in memory of the proxies, but also the space analysis *i.e.,* how many objects are needed per target. Only a proxy instance? A proxy instance and a handler instance?

Implementation complexity. Since at constant functionality, a simpler implementation is better, this criteria evaluates the complexity of the implementation. For example, if the proposed solution is easy to implement or if it needs complex mechanisms.

Ease of debugging. It is difficult to test and debug proxies because the debugger or the test framework usually send messages to the objects that are present in the current stack. Those messages include, for example, printing an object, accessing its instance variables, etc. When the proxy receives any of those messages it may intercept it (depending whether the proxy understands that message or not). Hence, debugging is usually complicated in the presence of proxies.

Constraints. The toolbox may require, *e.g.,* that the target implements certain interface or inherits from a specific class. In addition, it is important that the user of the proxy toolbox can easily extent or change the purpose of the proxy adapting it to his own needs.

Portability. A proxy implementation can depend on the VM or the language where it is developed which can be different in other Virtual Machines or languages.

3. Common Proxy Implementations

Even if there are different proxy implementations and solutions, there is one that is the most common among dynamic programming languages: it is based on error raising and resulting error handling. We briefly describe it and show that it fails to fulfill important requirements.

3.1 Typical Proxy Implementation

In *dynamic languages*, the type of the message's receiver is resolved at runtime. When an unknown message is sent to an object, an error exception is thrown. The basic idea is then to create objects that raise errors for all the possible messages (or a subset) and customize the error handling process.

In Smalltalk, for instance, the Virtual Machine sends the message doesNotUnderstand: to the receiver object. To avoid infinite recursion, all objects must understand the message doesNotUnderstand:. That is the reason why such method is implemented in the class Object, the root of the hierarchy chain. In Smalltalk, the default implementation throws a MessageNotUnderstood exception. Similar mechanisms exist in dynamic languages like Ruby, Python, Objective-C, Perl, etc.

Since doesNotUnderstand: is a normal method, it can be overwritten in subclasses. Hence, if we can have a minimal object and we override the doesNotUnderstand: method to

do something special (like forwarding messages to a target object), then we have a possible proxy implementation. This technique has been used for a long time [20, 22] and it is the most common proxy implementation. Readers knowing this topic can directly jump to Section 3.2. Most dynamic languages provide a mechanism for handling messages that are not understood as shown in Section 7.

Obtaining a minimal object. A minimal object is that one which understands none or only a few methods. In some programming languages, the root class of the hierarchy chain (usually called Object) already contains several methods [1]. In Pharo Smalltalk, Object inherits from a superclass called ProtoObject which inherits from nil. ProtoObject understands a few messages[2]: the minimal amount of messages that are needed by the system. Here is a simple Proxy implementation in Pharo.

```
ProtoObject subclass: #Proxy
    instanceVariableNames: 'targetObject'
    classVariableNames: ''

Proxy >> doesNotUnderstand: aMessage
   |result|
   ..."Some application specific code"
   result := aMessage sendTo: targetObject.
   ..."Other application specific code"
^result
```

Handling not understood methods. This is the part of the code that is user-defined and not part of the Proxy framework itself. Common behavior include logging before and after the method, forwarding the message to a target object, validating some access control, etc. In case it is needed, it is perfectly valid to issue a super send to access the default doesNotUnderstand: behavior.

To forward a message to a target object, we need the message name and the list of parameters sent to it. The Smalltalk Virtual Machine invokes the doesNotUnderstand: aMessage with a message reification as argument. Such class specifies the method selector, the list of arguments and the lookup class (in normal messages it is the receiver's class and, for super sends, it is the superclass of the class where the method is implemented. To forward a message to another object, the class Message provides the method sendTo: anotherObject which sends such message to another object.

Notice that this solution is not limited to Smalltalk. For example, the Smalltalk's doesNotUnderstand: is in Ruby method_missing, in Python __getattr__, in Perl autoload, in Objective-C forwardInvocation:, etc. As we explain in Section 7, Objective-C provides a minimal object class called NSInvocation which understands the message invokeWithTarget:aTarget and forwards a message to another object. Example:

```
- (void)forwardInvocation:(NSInvocation *)invocation
    {
      [invocation invokeWithTarget:delegate];
    }
```

In Ruby we can do:

```
def method_missing(name, *args, &block)
    target.send(name, *args, &block)
end
```

In Python:

```
  def __getattr__(self, name):
  return getattr(self.realObject, name)
```

3.2 Evaluation

In this section we evaluate the common proxy implementation based on the criteria we provided above (see section 2.2).

Stratification. This solution is not stratified at all:

- The method doesNotUnderstand: cannot be trapped like a regular message. Moreover, when such message is sent to a proxy there is no efficient way to know whether it was because of the regular error handling procedure or because of a proxy trap that needs to be handled. In other words, the doesNotUnderstand: occupies the same namespace as application-level methods [27], hence this solution is not stratified.
- There is no separation between proxies and handlers.

Interception granularity. It cannot intercept all messages but instead *only* those that are not understood. As explained, this generates method name collisions.

Object replacement. In the common proxy implementation object replacement is usually not supported. Nevertheless, Smalltalk implementations do support it but suffer the problem of "reference leaks": the target might provide its own reference as a result of a message or a parameter. This way the client gets a reference to the target, and hence it can by-pass the proxy.

Transparency. This solution is not transparent. Proxies do understand some methods (those from its superclass) generating method name collisions. For instance, if we evaluate "Proxy new pointersTo" (pointersTo is a method implemented in ProtoObject) it answers the references to the proxy instead of intercepting the message and forward it to a target. The same happens with the identity comparison or asking the class.

Efficiency. From the CPU point of view, this solution is fast and it has low overhead. In contrast to other technologies, there is no need to recompile the application and the system libraries or to modify their bytecode, or to do other changes such as in Java modifying the environment variable CLASSPATH, the class loader. Regarding the memory usage, there is no optimization. Efficiency is not normally addressed in typical proxy implementations.

[1] Object has 338 methods in PharoCore 1.3

[2] ProtoObject has 40 methods in PharoCore 1.3

Implementation complexity. This solution is easy to implement: it just needs the doesNotUnderstand:, a minimal object, and be able to forward a message to another object.

Ease of debugging. It is not provided by this solution. The debugger sends messages to the proxy which may not be understood, and hence, delegated to a target object. This makes it hard to debug, inspect and print Proxy instances.

Constraints. This solution is flexible since target objects do not need to implement any interface or method, nor to inherit from specific classes. The user can easily extent or change the purpose of the proxy adapting it to his own needs by just reimplementing the doesNotUnderstand:.

Uniformity. This implementation is not uniform since proxies cannot be used as classes, methods, etc.

Portability. This approach impose few requirements for the language and the VM that are provided by almost all available dynamic languages. With the examples of the previous section we demonstrate that it is really easy to implement this approach in different dynamic languages.

4. The Ghost Model

This section describes and explains the Ghost proxy model. This model fits better for dynamic programming languages and it is intended to be a reference model, *i.e.,* developers from different dynamic languages can implement it. In addition, the model must clarify which are the expected requirements and hooks from the host language.

4.1 Proxies

Ghost model supports proxies for regular objects as well as for classes, methods, and any other class that requires special management. In addition, Ghost supports proxies for classes or methods. Furthermore, Ghost model distinguishes between *interceptors* and *handlers*. Proxies play solely the role of interceptors. Since we are describing the model, the design is abstract and general. The design of an implementation may look different from this model. Figure 1 shows the proxies hierarchy and the following is a quick overview of the responsibilities of each class:

ObjectProxy. This is the base class for all proxies of Ghost model and provides proxies for regular objects, *i.e.,* objects that do not need any special management. Its responsibility, as well as its subclasses, is to take care about the message interception, which is represented in Figure 1 as the method intercept(). In Ghost model, Proxies only play the role of interceptors. Proxies are instances of ObjectProxy or any of its subclasses and all they do is to forward intercepted messages to handlers. Each proxy must have an associated handler. Different proxies can use different handlers and vice versa.

Finally, note that since proxies just intercept messages and forward them to handlers, it is unlikely that the user of the framework needs to customize or subclass any of the proxy classes. What the user needs to define is what to do in the handler.

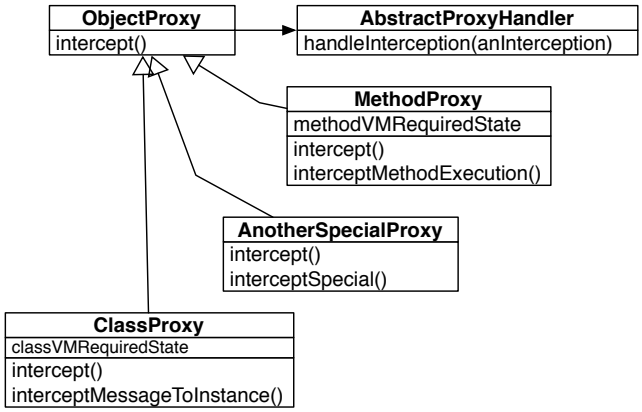

Figure 1. Proxies hierarchy in Ghost model.

ClassProxy. There are object-oriented programming languages that represent classes as first-class objects, *i.e.,* classes are not more than just instances from another class known as the Metaclass. ClassProxy provides proxies for class objects.

ClassProxy is needed as a special class in the model because the VM might impose specific constraints on the memory layout of object representing classes. For example, the Smalltalk VM expects the object to have three instance variables: format, methodDict, superclass. Since we are presenting Ghost *model*, that shape is generic. Different implementations may require different attributes or none. This is the reason why in Figure 1 the possible imposed memory layout for ClassProxy is represented by the attribute classVMRequiredState.

Frequently, the developer needs to be able to replace an existing class by a proxy. In that case, we need that the object replacement not only updates the references from other objects, but also the class pointer in the instances of the original class. For example, suppose there is an instance of User called bestUser. There is also a SecurityManager class that has a class variable called userClass which in this case points to User.

ClassProxy has to intercept the following type of messages:

- Messages that are sent directly to the class as a regular object. To continue with our example, imagine the method controlLogin in SecurityManager that sends the message maxLoggedUsers to its userClass instance variable. In Figure 1 this kind of interception is represented with the method intercept().
- Messages that are sent to an instance of the original class, *i.e.,* objects whose class references are pointing to the proxy (this happens as a consequence of replacing the class with the proxy). In our example, we can send the message username to the bestUser instance. In Figure 1 this kind of interception is represented with the method interceptMessageToInstance(). Notice that this kind of messages are only necessary when there is an object replace-

ment, *i.e.,* the instances' class pointers of the original class were updated to reference the proxy.

MethodProxy. In some dynamic languages, not only classes are first-class objects but also methods as well. In addition, similarly to the case of ClassProxy, there are two kinds of messages that MethodProxy needs to intercept:

- When sending messages to the method as a regular object. For example, in Smalltalk when you search for senders of a certain method, the system has to check in the literals of the compiled method if it is sending such message. To do this, the system searches all the literals of the compiled methods of all classes. This means it will send messages (sendsSelector: in this case) to the objects that are in the method dictionary. When creating a proxy for a method we need to intercept such messages. In Figure 1 this kind of interception is represented with the method intercept().
- When the compiled method is executed. Suppose we want to create a proxy for the method register of User class. We need to intercept the method execution, for example, when doing User new register. This kind of interception is represented in Figure 1 with the method interceptMethodExecution(). Note that this type of message exist *only* if there is object replacement, *i.e.,* when the original method is replaced by a proxy.

The same way that the VM imposes an object shape on classes, it may also do it on methods. This requirement is represented in Figure 1 with the instance variable methodVMRequiredState which may vary from one implementation to the other.

AnotherSpecialProxy. This class is just to document that the model must support different classes that need special management. In this paper, and in our implementation, we concentrate on classes and methods, but there can be more.

4.2 Handlers

Figure 2 shows the handler hierarchy of the Ghost model. Once again, note that this is an abstract model and a concrete implementation can vary significantly. Handler's responsibility is to handle the method interceptions that the proxies trap. It is not necessary to explain in details each handler, since we think it is self explanatory.

The information passed from a proxy to a handler can vary depending on the implementation. The typical passed information is:

- The name of the message received and its arguments.
- The proxy.
- The proxy's state. It can contain anything such as the target object, a filename or a number. This is necessary only if such state is in the proxy and not in the handler. Indeed, the proxy is supposed to intercept messages even if they are sent by the handler. So, the handler cannot send a message to the proxy to get its state. This is why it is the responsibility of the proxy to provide this state if any.

All that information is reified in the model as an instance of class Interception.

4.3 Discussions

Users can adapt and extend the Ghost framework according to their own needs via inheritance. In Figure 2 LoggerClassProxyHandler a user-defined class logs every intercepted messages and forwards them to the target object.

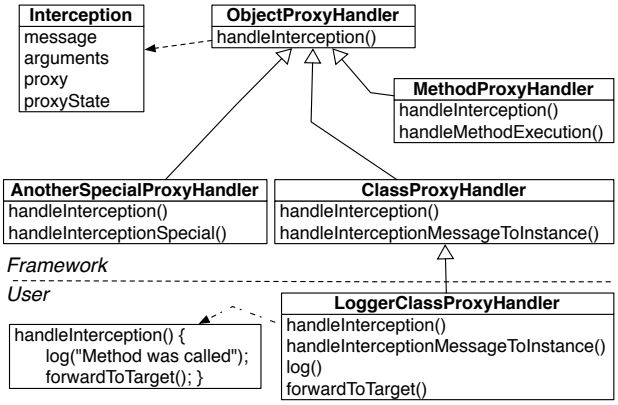

Figure 2. Handlers hierarchy in Ghost model.

Normally, some information is needed to accomplish the proxy process, for example, a target object, an address in secondary memory, a filename, an identifier, etc. This information can be stored in the proxies, in the handlers or elsewhere. However, as explained, if the state is kept in the proxy the handler cannot ask for it because such message sent will be intercepted. Hence, if the desire is to store the state in the proxy, such state must be included in the Interception object that is passed to the handler. This is represented as the instance variable proxyState in Figure 2. That instance variable can represent a target object, an address in secondary memory, a filename, an identifier, etc. Where to put this state is user's application dependent and a matter of design regarding the relationship between proxies and handlers.

Proxies delegates the interception to a handler. How the proxy gets the reference to the handler depends on the implementation. For example, in one case the handler can be an instance variable of the proxy that is provided when the proxy is created. In another case, all proxies can use the same handler, which in this case the previous instance variable may not be necessary and instead they reference directly to the handler class.

Notice that in the model we are modeling the interception of messages. However, some languages do not threat everything like a message sent, but instead they have special operators or syntax as part of the language. To implement Ghost, there must be a way to intercept such special syntax or otherwise pay the cost of not being able to intercept them.

5. Smalltalk Support for Proxies

Before presenting the Ghost implementation, we first explain the basis of the Pharo Smalltalk reflective model and some provided hooks. We show that Smalltalk provides all the necessary support for proxies *i.e.,* object replacement, interception of method execution and the reification of classes and methods as first-class objects.

5.1 Pharo Reflective Model and VM Overview

Readers familiar with the Pharo reflective model please feel free to skip this section. The reflective model of Smalltalk is easy and elegant. There are two important rules [4]: 1) *Everything is an object*; 2) *Every object is instance of a class*. Since classes are objects and every object is an instance of a class, it follows that classes must also be instances of classes. A class whose instances are classes is called a metaclass. Whenever you create a class, the system automatically creates a metaclass. The metaclass defines the structure and behavior of the class that is its instance. Figure 3 shows a *simplified* reflective model of Smalltalk.

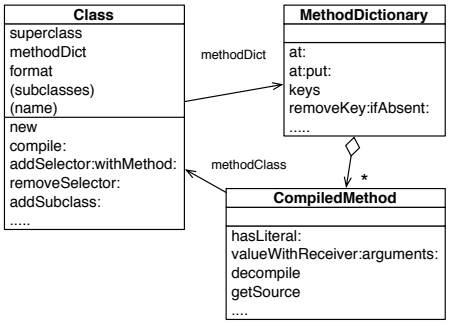

Figure 3. The basic Smalltalk reflective model.

Figure 3 shows that a class contains a name, a format, a method dictionary, its superclass, a list of instance variables, etc. The method dictionary is a map where keys are the methods names (called selectors in Smalltalk) and the values are the compiled methods which are instances of CompiledMethod.

5.2 Hooks and Features Provided by Pharo Smalltalk

Before explaining Ghost implementation on Pharo, we present some of the Smalltalk reflective facilities and hooks that can be used for implementing proxies.

Class with no method dictionary. The method dictionary is just an instance variable of a class, hence it can be changed. When an object receives a message and the VM does the method lookup, if the method dictionary of the receiver class (or of any other class in the hierarchy chain) is nil, then the VM directly sends the message cannotInterpret: aMessage to the receiver. But, the lookup for method cannotInterpret: starts in the *superclass* of the class whose method dictionary was nil.

Imagine the class MyClass which has its method dictionary in nil, and its superclass MyClassSuperclass. There is also an instance of MyClass called myInstance. Figure 4 shows how the hook works when sending the message printString to the object myInstance.

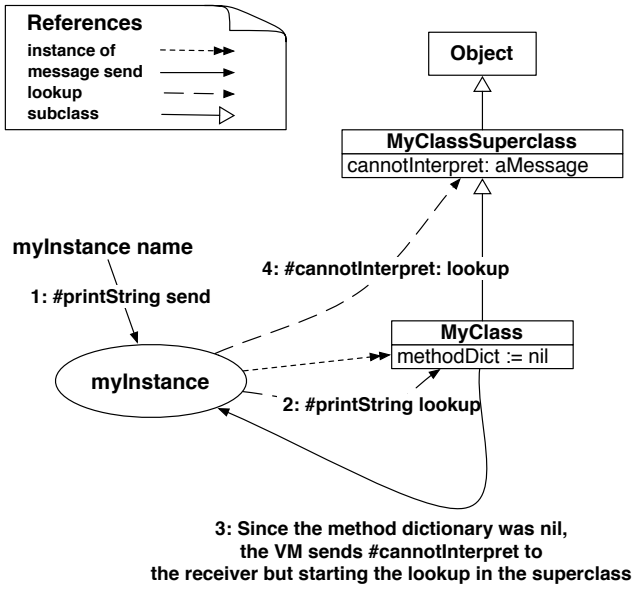

Figure 4. Message handling when a method dictionary is nil.

The cannotInterpret: is sent to the receiver but starting the method lookup from the *superclass*. Otherwise there will be an infinite loop. This hook is very powerful for proxies since it let us intercept all messages that are sent to an object.

Objects as methods. This facility allows intercepting method executions. It relies on replacing in a method dictionary a method by an object that is not an instance of CompiledMethod. Interception occurs if the object does understand the message run:with:in: as we explain below. Otherwise, we get a MessageNotUnderstood exception.

To illustrate interception consider the following code:

```
MyClass methodDict at: #printString put: MethodProxy new.
MyClass new printString.
```

When the printString message is sent the VM does the method lookup and finds an entry for #printString in the method dictionary. If the retrieved object is actually an instance of CompiledMethod (which is the case in the normal scenario), then the VM executes it. Otherwise, the VM sends a special message run: aSelector with: arguments in: aReceiver to that object, *i.e.,* the one that replaces a method in the method dictionary.

This technique is used when implementing MethodWrappers [6]. Using run:with:in is not the only possible technique to implement MethodWrappers in Smalltalk. In fact, the original implementation rely on subclassing CompiledMethod.

It is important to notice that the previous explanation means that the Pharo VM does not impose any shape to objects acting as methods such us having certain amount of instance variables or certain format. This is because the

VM checks whether the object in the MethodDictionary is a CompiledMethod or not and if it is not it sends the message run:with:in:. The only requirement is to implement that method. Therefore, MethodProxy does not need to fulfill any class shape in a Ghost implementation on Pharo Smalltalk.

Object replacement. The primitive become: anotherObject is provided by the Pharo VM and it swaps the object references of the receiver and the argument. All variables in the entire system that used to point to the receiver now point to the argument, and vice versa. In addition, there is also becomeForward: anotherObject which updates all variables in the entire system that used to point to the receiver now point to the argument, *i.e.,* it is only one way.

Change the class of an object. Smalltalk provides a primitive to change the class of an object. Although it has some limitations, *e.g.,* the object format and the class layout of both classes need to be the same. These primitives are Object»primitiveChangeClassTo: or Behavior»adoptInstance: .

6. Ghost Implementation

In this section, we present the Ghost implementation. Its most important features are: to be stratified (*i.e.,* clear separation between proxies and handlers), to be able to intercept all messages, and to be uniform. For this implementation we use the previously mentioned Pharo Smalltalk reflective facilities: classes with no method dictionary, objects as methods, object replacement and the ability to change the class of an object.

Regarding the discussions of Section 4.3, in this implementation we store the needed information, for example, the target object, an identifier, a filename, etc, in the proxies. Another possible implementation is to store the information in the handler for example. In addition, in the following implementation each proxy instance uses a particular handler instance, hence the handler is represented as an instance variable of the proxy.

To explain the implementation we use a SimpleForwarderHandler which just forwards the interceptions to a target object. Therefore, the state stored in the proxy is a target object.

6.1 Kernel

Figure 5 shows the basic design of Ghost.

To explain the implementation we start with the following simple test:

```
testSimpleForwarder
    | proxy |
     proxy := Proxy proxyFor: (Point x: 3 y: 4) handler: SimpleForwarderHandler new.
    self assert: proxy x equals: 3.
    self assert: proxy y equals: 4.
```

The class side method proxyFor:handler: creates a new instance of Proxy, sets the handler, and finally changes the class of the just created Proxy instance to ProxyTrap. The user of the toolbox can specify which handler to use just by sending it as a parameter of the proxy creational message proxyFor:handler:.

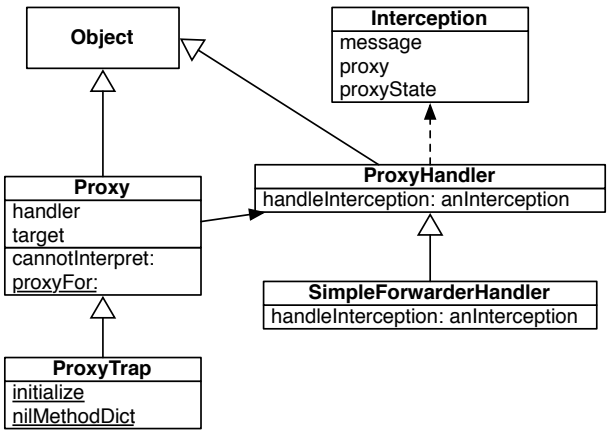

Figure 5. Ghost implementation's basic design.

```
Proxy class >> proxyFor: anObject handler: aHandler
    | aProxy |
    aProxy := self new
            initializeWith: anObject
            handler: aHandler.
    ProxyTrap adoptInstance: aProxy.
    ^ aProxy.
```

The class side method initialize is called right after loading ProxyTrap into the system and it sets the method dictionary of the class to nil. Notice that the system does not deal correctly with classes whose method dictionary is nil. Hence, we need to overwrite the method Behavior » methodDict to:

```
Behavior >> methodDict
  methodDict == nil ifTrue: [^ MethodDictionary new ].
  ^ methodDict
```

Since the system access the method dictionary with methodDict it looks like if the class has an empty method dictionary, but instead it has a nil. Since the VM access directly to the slow where the method dictionary is, *i.e.,* the VM does not use methodDict, it works for both things: the interception and the system.

With the line ProxyTrap adoptInstance: aProxy we change the class of aProxy to ProxyTrap, whose method dictionary is nil. This means that for *any* message sent to aProxy, the VM will finally send the message cannotInterpret: aMessage. Remember that such message is sent to the receiver (in this case aProxy) but starting the method lookup in the super class, which in this case is Proxy. Hence, Proxy implements the method cannotInterpret:

```
Proxy >> cannotInterpret: aMessage
    | interception |
       interception := Interception for: aMessage proxyState: target proxy: self.
    ^ handler handleInterception: interception.
```

An Interception instance is created and passed to the handler. In this example, the instance variable proxyState is the target object.

Handler classes are user-defined and in this example we use a simple forwarder handler, *i.e.,* it logs and forwards the received message to a target object. Users of the toolbox can create their own handlers that achieve their requirements.

```
SimpleForwarderHandler >> handleInterception: anInterception
    | answer |
     self log: 'Message ', anInterception message selector, ' intercepted'.
        answer := anInterception message sendTo: anInterception proxyState.
    self log: 'The message was forwarded to the target object'.
    ^ answer
```

For the moment, we can say that the class Proxy can only be used for regular objects (in the example we create a proxy for Point instance). We see in the following sections how Ghost handles objects that do require special management like classes or methods.

6.2 Proxies for Methods

As we have already explained in Section 4, for methods there are two kind of messages that we need to intercept:

- When the compiled method is executed.
- When sending messages to the compiled method object.

To clarify, imagine the following test:

```
testSimpleProxyForMethods
    | aProxy kurt method |
    kurt := User named: 'Kurt'.
    method := User compiledMethodAt: #username.
    aProxy := Proxy
        createProxyAndReplace: method
        handler: SimpleForwarderHandler new.
    self assert: aProxy getSource equals: 'username ^ name'.
    self assert: kurt username equals: 'Kurt'.
```

What the test does is to create an instance of a User and a proxy for method username. Then, we replace the original method username with the created proxy. Finally, we test both type of messages: when sending a message to the proxy (in this case aProxy getSource) and when sending message username that leads to the execution of the proxified method.

With Ghost implementation, both kind of messages are solved out of the box: the first case, *i.e.,*aProxy getSource has nothing special and it behaves exactly the same way we have explained so far. The second one, *i.e.,*kurt username, also works without any special management by using the explained hook of the method run:with:in:. However, this second type of message is only captured if the original method was replaced by the proxy. This is why in this test we use the method createProxyAndReplace:handler: instead of proxyFor:handler:, because we want to not only to create a proxy for the method but instead replace it with the proxy. The following is the implementation of such method:

```
Proxy class >> createProxyAndReplace: aClass handler: aHandler
    | aProxy newProxyRef newObjectRef|
    aProxy := self new
      initializeWith: anObject
      handler: aHandler.
    aProxy become: anObject.
    "After the become is done, variable aProxy points to anObject
    and variable anObject points to aProxy. We create two new
    variables just to clarify the code"
    newProxyRef := anObject.
    newObjectRef := aProxy.
    newProxyRef target: newObjectRef.
    ProxyTrap adoptInstance: newProxyRef.
    ^ newProxyRef.
```

Notice that createProxyAndReplace:handler: is useful for method proxies, as well as for regular objects. In the previous section where we used the method proxyOn: we could perfectly have used createProxyAndReplace:handler: instead.

Coming back to the test of kurt username, when the VM does the method lookup for the message username it notices that in the method dictionary is not a CompiledMethod instance but instead an instance from another class. Hence, it sends the message run:with:in to such object. Since such object is a proxy in this case, the message run:with:in: will be intercepted and forwarded just like any other message. In the base Pharo image, CompiledMethod does not implement such method, so Ghost implements it as a method extension in the following way:

```
CompiledMethod >> run: aSelector with: anArray in: aReceiver
     ^ self valueWithReceiver: aReceiver  arguments: anArray
```

That method just executes the method (the receiver). However, such change does not need to be necessary implemented in CompiledMethod. As we will see later, Ghost supports a way to define specific messages so that they are treated and answered by the handler instead of being managed as a normal interception. So we can tell the handler to perform something in particular if the message run:with:in: is intercepted (this information is available in the Message instance referenced by the Interception object). In this case we can directly use the method valueWithReceiver:arguments: to execute the CompiledMethod.

The previous explanation demonstrates how Ghost can create not only proxies for methods, but also how to replace them by proxies. In contrast to what we defined in the model, the Pharo Smalltalk VM does not impose any shape to methods. Therefore, we can use the same Proxy class that we use for regular objects, *i.e.,* the class MethodProxy defined in the Ghost model does not exist in this concrete implementation since we can directly use Proxy.

6.3 Proxies for Classes

Implementing proxies for classes and also to be able to replace and use a proxy as a class, has some important constraints:

- Class proxies must fulfill the expected object shape that the VM imposes in classes. In the case of Pharo Smalltalk, the minimum amount of instance variables that a class must have are: superclass, methodDict and format.
- Instances hold a reference to their class and the VM uses this reference for the method lookup.
- A class is involved with two kinds of messages that need to be intercepted as introduced in Section 4:
  - Messages that are sent directly to the class.
  - Messages that are sent to an instance of the class. Such messages are intercepted only if the original class was replaced by the proxy.

To explain class proxies, consider the following test:

```
testSimpleProxyForClasses
    | aProxy kurt |
    kurt := User named: 'Kurt'.
    aProxy := ClassProxy
        createProxyAndReplace: User
        handler: SimpleForwarderHandler new.
    self assert: User name equals: #User.
    self assert: kurt username equals: 'Kurt'.
```

The test creates an instance of a user, and then with the message createProxyAndReplace:handler: we create a proxy for the User class and we replace it by the created proxy. Finally, we test that we can intercept both messages: those which are sent to the proxy (in this case User name) and those which are sent to instances of the original class (kurt username in this case).

The first message, User name, has nothing special and it is handled the same way as any other message. The second one is more complicated and it requires certain explanation.

Figure 6 shows the design of ClassProxy. First, notice that we do not use the class Proxy but instead ClassProxy. This is because proxies for classes need to fulfill the expected object shape that the VM imposes in classes, *i.e.,* the instance variables superclass, methodDict and format. Second, in the model we showed that ClassProxy was a subclass of ObjectProxy but in this case it is not. The reason is that the VM does not only imposes the mentioned instance variables but also the *order*: superclass at position 1, methodDict at 2 and format at 3. If ClassProxy is a subclass of ObjectProxy it inherits the two instance variables target and handler and since they are defined in the superclass they are "first" in the array of instance variables of the object. So, superclass will be at position 3, methodDict at 4 and format at 5. Therefore, we are not respecting the expected shape.

The previous issue is not really a problem because ObjectProxy implements only two methods and they are even different in ClassProxy. Hence, even if the limitation is real, we are not duplicating code because of that.

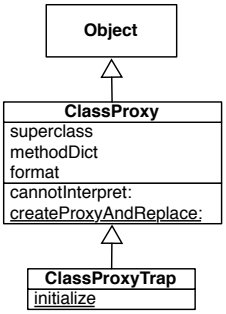

Figure 6. Class proxies in Ghost stratified implementation.

The method createProxyAndReplace:handler: is similar to the one used in Proxy:

```
ClassProxy class >> createProxyAndReplace: aClass handler: aHandler
  | aProxy newProxyRef newClassRef|
  aProxy := self new
      initializeWith: aHandler
        methodDict: nil
        superclass: ClassProxy
        format: aClass format.
  aProxy become: aClass.
  "After the become is done, aProxy now points to aClass
   and aClass points to aProxy. We create two new variables
   just to clarify the code"
  newProxyRef := aClass.
  newClassRef := aProxy.
  newProxyRef target: newClassRef.
  ClassProxyTrap adoptInstance: newProxyRef.
  ^ newProxyRef.
```

The difference is that in addition to setting the handler and the target, we also set the method dictionary, the superclass and the format. This is because an *instance* of ClassProxy must work as a class. Thus, we set its method dictionary in nil, ClassProxy as the superclass and finally the format (this is important so that the adoptInstance: does not fail).

Coming back to the example, when we evaluate kurt username this is what happens: the class reference of kurt is pointing to the created ClassProxy instance (as a result of the become:), and this proxy object that acts as a class, has the method dictionary instance variable in nil. Hence, the VM sends the message cannotInterpret: to the receiver (kurt in this case) but starting the method lookup in the superclass which is ClassProxy (as set in method ClassProxy class » createProxyAndReplace:handler: defined above). The definition of the cannotInterpret: of class ClassProxy is the following.

```
ClassProxy >> cannotInterpret: aMessage
  | interception |
  "The order of this expression is important
  because a proxy intercepts all messages including =="
  (ClassProxyTrap == aMessage lookupClass )
```

```
ifTrue: [ interception := Interception for: aMessage
                    proxyState: target proxy: self.
        ^ handler handleInterception: interception]
ifFalse: [ interception := Interception for: aMessage
                    proxyState: target proxy: aMessage lookupClass.
        ^ handler handleInterception: interception toInstance: self]
```

It is important to notice the difference in this method regarding the kind of message it is intercepting. On the one hand, when we evaluate User name and the cannotInterpret: is called, the receiver, *i.e.,* what self is pointing to, is the proxy itself. On the other hand, when we evaluate kurt username and cannotInterpret: is called, self points to kurt and not to the proxy.

The method Message lookupClass answers the class where lookup will start. If it is ClassProxyTrap it means the receiver was proxy, and not an instance of the original class.

A problem is that the CompiledMethod of cannotInterpret: cannot be correctly executed with a receiver like kurt. In fact, it can only be correctly executed with proxy instances. The reason is that the method ClassProxy » cannotInterpret: access the instance variable handler. Hence the first problem is that the class User does not define such instance variable. The second problem is that CompiledMethod do not store instance variable names but instead its offsets. So when the CompiledMethod of cannotInterpret: is executed the instructions (bytecodes) to access the instance variable handler is just something like "access instance variable at position 5", which is correct in the class where it was defined (ClassProxy). When evaluating the method with receivers of other classes *e.g.,*User then the VM can crash because it is accessing outside the object or just answer whatever is at that place. For example, if a class defines only two instance variables, the bytecode "accessing instance variable at position 5" means that the VM will access a memory address outside the object. Whether the VM crashes or not depends on the concrete VM implementation. In the case of Pharo Smalltalk, the VM crash in such scenario so we cannot use this solution.

Instead of directly accessing the instance variable handler one may think why not to send a message handler. This is not possible because since the proxy intercepts all messages, such message sent will finally call cannotInterpret: generating an infinite loop.

To that limitation, Ghost provides the following alternative. Instead of doing handler handleInterception: interception toInstance: self we send a special message to the proxy, which is accessible through aMessage lookupClass. Hence, we can do aMessage lookupClass handleInterception: interception toInstance: self. In the item *Ease of debugging* of the next section we explain that we can define a list of specific messages in the handler so that it does not manage such messages interceptions as it is done with the regular ones, but instead those messages are processed and answered by the handler itself. The message handleInterception:toInstance: is one of those messages and it is managed by the handler. At that point the handler has everything he needs *e.g.,*Interception object and receiver, so it can perform its task.

Coming back to the implementation, the last missing explanation is why we need ClassProxyTrap instead of reusing ProxyTrap. The reason is that the message adoptInstance: requires certain conditions, like having the same object format. Since ClassProxy and Proxy have different amount of instance variables and hence format, then we cannot reuse the same ProxyTrap.

```
ProxyTrap class >> initialize
  superclass := Proxy.
  methodDict := nil.
  format := Proxy format.

ClassProxyTrap class >> initialize
  superclass := ClassProxy.
  methodDict := nil.
  format := ClassProxy format.
```

The Ghost implementation uses ProxyClass and ClassProxyTrap not only because it is cleaner from the design point of view but also because of the memory footprint. Technically, we can use ProxyClass and ClassProxyTrap *also* for regular objects and methods. But that implies that for every target to proxify the size of the proxy can be unnecessary bigger in memory footprint, because of the additional instance variables needed by ClassProxy.

To conclude, with this implementation we can successfully create proxies for classes, *i.e.,* to be able to intercept the two mentioned kind of messages and replace classes by proxies.

6.4 Criteria Evaluation

Stratification. This solution is completely stratified. On the one hand, there is a clear separation between proxies and handlers. On the other hand, interception facilities are separated from application functionality. Indeed, the application can even send the cannotInterpret: message to the proxy. Since, proxies do not understand any message, cannotInterpert: would be intercepted like any other message. Thus, the proxy API does not pollute the application's namespace.

Object replacement. This is provided by Ghost thanks to the Smalltalk become: primitive.

Interception granularity. It intercepts all messages.

Transparency. The Pharo compiler associates special bytecodes for the messages class and == (identity), *i.e.,* even if there is an implementation of those methods, they are actually never executed and, therefore, they cannot be intercepted. Our solution is to modify the compiler so that it does not associate a special bytecode for both methods. Such modification is the following:

```
(ParseNode classVarNamed: 'StdSelectors') removeKey: #class.
(ParseNode classVarNamed: 'StdSelectors') removeKey: #==.
Compiler recompileAll.
```

We did a benchmark to estimate the overhead impact of such change. We run all the tests (8003 unit tests) present in a PharoCore 1.3 - 13204 image, twice: once with the class and == optimizations and once without them. The overhead of removing those optimizations was only about 4%, which means that it is only slightly perceptible in general system interactions.

In the discussion of Section 4 we talk about the possibility of some languages to have special syntax or operators in addition to messages sent. These special selectors class and == can be considered like that. However, Smalltalk allows us to convert them into messages so we have an easy way to deal with them. This way Ghost solution is fully transparent and both messages are intercepted and handled as any other message.

Efficiency. From the CPU point of view, this solution is fast and it has low overhead.

This solution provides an efficient memory usage with the following optimizations:

- Proxy and ClassProxy are "Compact Classes". This means that in a 32 bits system, their instances' object header are only 4 bytes long instead of 8 bytes for instances of regular classes. For instances whose "body" part is more than 255 bytes and whose class is compact, their header will be 8 bytes instead of 12. The first word in the header of regular objects contains flags for the Gargbage Collector, the header type, format, hash, etc. The second word is used to store a reference to the class. In compact classes, the reference to the class is encoded in 5 free bits in the first word of the header. These 5 bits represent the index of a class in the compact classes array set by the image[3] and accessible to the VM. With these 5 bits, there are 32 possible compact classes. This means that, from the language side, the developer can determinate up to 32 classes as compact. Their instances' object header are only 4 bytes long as we said. Hence, declaring the proxy classes as compact makes proxies to have smaller header and then smaller memory footprint.
- Proxies only keep the minimal state they need. For example, as we have already explained, we can use ClassProxy for every type of object. However, the size of the proxies would be unnecessary larger to store the additional needed instance variables of ClassProxy.
- In proxy creation methods presented so far (proxyFor:handler: and createProxyAndReplace:handler:) the last parameter is an instance of the handler. This is because in our examples, each proxy holds a reference to handler. However, this is only necessary when the user needs one handler instance per target object, which is not often the case. The handler is often stateless and can be shared and referenced through a class variable or a global one. Hence, we can avoid the memory cost of a handler instance variable in the proxy. Instead, one possible solution is to reference in the Proxy»cannotInterpret: method a handler class which has a class side method handleInterception:. For example:

```
Proxy >> cannotInterpret: aMessage
    | interception |
     interception := Interception for: aMessage proxyState: target proxy: self.
    ^ SimpleForwarderHandler handleInterception: interception.
```

An alternative is to use a handler class with a singleton or a default instance. For example:

```
Proxy >> cannotInterpret: aMessage
    | interception |
     interception := Interception for: aMessage proxyState: target proxy: self.
    ^ SimpleForwarderHandler uniqueInstance handleInterception: interception.
```

In both cases we save the memory corresponding to the instance variable to reference the handler plus the handler instance itself. If we consider that the handler has no instance variable, then it is 4 bytes for the instance variable in the proxy and 8 bytes for the handler instance. That gives a total of 12 bytes saved per proxy in a 32 bits system.

Implementation complexity. This solution is easy to implement: an approximation of 5 classes, with an average of 3.4 methods per class, and each method is of an average of 5 lines of code.

Ease of debugging. Ghost implementation supports special messages that the handler must answer itself instead of managing it as a regular interception. The handler can keep a dictionary that maps selector of messages intercepted by the proxy to selectors of messages to be performed by the handler itself. This user-defined list of selectors can be used for debugging purposes, *i.e.,* those messages that are sent by the debugger to the proxy are answered by the handler and they are not managed as a regular interception. This significantly ease the debugging of proxies. For example, the handler's dictionary of special messages for debugging can be defined as following:

```
SimpleForwarderHandler >> debuggingMessagesToHandle
| dict |
dict := Dictionary new.
dict at: #basicInspect put:#handleBasicInspect:.
dict at: #inspect put:#handleInspect:.
dict at: #inspectorClass put:#handleInspectorClass:.
dict at: #printStringLimitedTo: put: #handlePrintStringLimitedTo:.
dict at: #printString put: #handlePrintString:.
^ dict
```

The keys of the dictionary are selectors of messages received by the proxy and the values are selectors of messages that the handler must send to itself. All the selectors of messages to be sent to the handler (*i.e.,* the dictionary values)

[3] see methods SmalltalkImage»compactClassesArray and SmalltalkImage»recreateSpecialObjectsArray

have a parameter which is an instance of Interception, which contains the receiver, the message, the proxy and the target. Therefore, all those methods have access to all the information they need.

Moreover, these special messages are "pluggable" *i.e.,* they can be easily enabled *e.g.,* for debugging, and disabled for production.

Constraints. The solution is flexible since target objects can inherit from any class and they are free to implement or not implement all the methods they want. There is not any kind of restriction imposed by Ghost. In addition, the user can easily extent or change the purpose of the proxy adapting it to his own needs: he just needs to subclass a handler and implement the necessary methods like handleInterception:.

Uniformity. This implementation is uniform since proxies can be used for regular objects, as classes and as methods. Moreover they all provide the same API and can be used polymorphically. Nevertheless, there is still non-uniformity regarding some other special classes and objects. Most of them are those that are present in what is called the *special objects array* (check method recreateSpecialObjectsArray) in Pharo Smalltalk. Such array contain the list of special objects that are known by the VM. Examples are the objects nil, true, false, etc. It is not possible to do a correct object replacement of those objects by proxies. The same happens with immediate objects, *i.e.,* objects that do not have object header and are directly encoded in the memory address, like SmallInteger.

The special object array contains not only regular objects but also classes. Those classes are known and used by the VM so it may impose certain shape, format or responsibilities in their instances. For example, one of those classes in Process. Once again, it is not possible to correctly replace a Process instance by a proxy. The same limitation exists if we want to create a proxy not for instances of those special classes but for those classes.

The mentioned limitations occur only when object replacement is desired. Otherwise, there is no problem and proxies can be created for those objects. In addition, we believe that creating proxies for methods and classes is useful in several scenarios as we see in next section. The rest of the mentioned limitations is not a common need. Hence, those restrictions are not a real problem for Ghost users.

Portability. This is the bigger disadvantage of this approach. It requires the hook of setting nil to a method dictionary and the VM sending the message cannotInterpret:. In addition, it also requires object replacement (become: primitive) and to be able to change the class of an object (adoptInstance: primitive). However, without these reflective facilities we cannot easily implement all the required features of a good proxy library. In the best case, we can get everything but with substantial development effort such as modifying the VM or compiler, or even creating them from scratch. Smalltalk provides all those features by default.

7. Related Work

7.1 Proxies in dynamic languages

Objective-C provides an out-of-the-box Proxy implementation called NSProxy [21]. This solution consists of an abstract class NSProxy that implements the minimum number of methods to be a root class. Indeed, this class is not a subclass of NSObject (the Objective-C root class in the hierarchy chain) but a separate root class (like subclassing from nil in Smalltalk). The intention is to reduce method conflicts between the proxified object and the proxy. Subclasses of NSProxy can be used to implement distributed messaging, future objects or other proxies usage. Typically, a message to a proxy is forwarded to a profixied object which can be an instance variable in a NSProxy subclass.

Since Objective-C is a dynamic language, it needs to provide a mechanism like the Smalltalk doesNotUnderstand: for the cases where an object receives a message that cannot understand. When a message is not understood, the Objective-C runtime will send methodSignatureForSelector: to see what kind of argument and return types are present. If a method signature is returned, the runtime creates a NSInvocation object describing the message being sent and then sends forwardInvocation: to the object. If no method signature is found, the runtime sends doesNotRecognizeSelector:.

NSProxy subclasses must override the forwardInvocation: and methodSignatureForSelector: methods to handle messages that they do not implement themselves. A subclass's implementation of forwardInvocation: should do whatever is needed to process the invocation such as forwarding the invocation over the network or loading the real object and passing the invocation. methodSignatureForSelector: is required to provide argument type information for a given message. A subclass' implementation should be able to determine the argument types (note that ObjectiveC is not so dynamic from this regard) for the messages it needs to forward and should construct a NSMethodSignature object accordingly.

To sum up, the developer needs to subclass NSProxy and implement the forwardInvocation: to handle messages that are not understood by itself.

One of the drawbacks of this solution is that the developer does not have control over the methods that are implemented in NSProxy. For example, such class implements the methods isEqual:, hash, class, etc. This is a problem because those messages will be understood by the proxy instead of being forwarded to the wrapped object producing different paths in the code execution. This solution is similar to the common solution in Smalltalk with doesNotUnderstand:. A possible, yet tedious, solution may be to overwrite such methods in the NSProxy subclass so that they delegate to the wrapped object.

In Ruby, there is a proxy implementation which is called Delegator. This is just a class included with Ruby standard library but it can be easily modified or implemented from scratch. Similar to Objective-C and Smalltalk (indeed, similar to most dynamic languages), Ruby provides a mechanism to handle the situation when an object receives

a message that cannot understand. This method is called method_missing(aSelector, *args). Moreover, since Ruby 1.9 Object is not the root of the hierarchy chain and Object is a subclass of a new minimal class called BasicObject which understands a few methods and is similar to ProtoObject in Smalltalk.

The idea of Ruby proxies are similar to the Smalltalk solution using doesNotUnderstand: and to NSProxy: have a minimal object (subclass from BasicObject) and implement method_missing(aSelector, *args) to intercept messages. In Python, an analogous implementation can be done by overwriting the __getattr__ method in a proxy. Such method is called when an attribute lookup has not found the attribute in the usual places.

Arnaud et al. [1] took a much deeper approach: internally, an object X does not refer directly to another object Y, but instead X has a reference to a special Handler object that refers to Y. The handler object is fully invisible for the developer. The idea is that different references to an object can use different handlers. This can be used for several things, like defining read-only references to an object. But the solution is generic so for example a handler could be used as a proxy. For example, a simple handler could be implemented so that it does something in particular with the message interception *e.g.,* logging, and then forward it to the target object.

7.2 Proxies in static languages

Java, being a statically typed language, supports quite limited proxies called *Dynamic Proxy Classes* [14]. It relies on the Proxy class from the java.lang.reflect package. "Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods."[14]. The creation of a dynamic proxy class can only be done by providing a list of java interfaces that should be implemented by the generated class. All messages corresponding to declarations in the provided interfaces will be intercepted by a proxy instance of the generated class and forwarded to a handler object. "Each proxy instance has an associated invocation handler object, which implements the interface InvocationHandler. A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the invoke method of the instance's invocation handler, passing the proxy instance, a java.lang.reflect.Method object identifying the method that was invoked, and an array of type Object containing the arguments. The invocation handler processes the encoded method invocation as appropriate and the result that it returns will be returned as the result of the method invocation on the proxy instance. " [14].

Java proxies have the following limitations:

- You *cannot* create a proxy for instances of a class which methods aren't all declared in interfaces. This means that, if you want to create a proxy for a domain class, you are forced to create an interface for it. Eugster [11] proposed a solution which provides proxies for classes. There is also a third-party framework based on bytecode manipulation called CGLib [9] which provides proxies for classes.
- *Only* the methods defined in the interface will be intercepted which is a big limitation.
- Java interfaces do not support private methods. Hence since Java proxies require interfaces, private methods cannot be intercepted either. Depending of the proxy usage this can be a problem.
- Proxies are subclass from Object, forcing them to understand several messages. When the messages hashCode, equals or toString (declared in Object) are sent to a proxy instance they are encoded and dispatched to the invocation handler's invoke method, *i.e.,* they are intercepted. However, the same does not happen with the rest of the public methods, *e.g.,*getClass. So a proxy answers its own class instead of the target's one. Therefore, the proxy is not transparent and it is not fully stratified.

Microsoft's .NET platform [26] proposes a closely related concept of Java dynamic proxies with nearly the same limitations as in Java. There are others third-party libraries like *Castle DynamicProxy* [8] or *LinFu* [18]. DynamicProxy differs from the proxy implementation built into .NET which requires the proxified class to extend MarshalByRefObject. Extending MashalByRefObject to proxy an object can be too intrusive because it does not allow the class to extend another class and it does not allow transparent proxying of classes. In LinFu, every generated proxy, dynamically overrides all of its parent's virtual methods. Each one of its respective overridden method implementations delegates each method call to the attached interceptor object. However, non of them can intercept non-virtual methods.

7.3 Comparison

Statically typed languages, such as Java or .NET, support quite limited proxies [2]. In Java the problem is that types are bound to classes and in addition the lookup is done statically *i.e.,* at compile-time. There is also the replacement issue and transparency. Another problem in Java is that one cannot build a proxy with fields storing any specific data. Therefore, one has to put everything in the handler, hence no handler sharing is possible ending in a bigger memory footprint.

Proxies are far more powerful, flexible, transparent and easy to implement in dynamic languages than static ones.

In dynamic languages, just two features are enough to implement a naive Proxy solution: 1) a mechanism to handle messages that are not understood by the receiver object and 2) a minimal object that understands a few or no messages so that the rest are managed by the mentioned mechanism.

Objective-C NSProxy, Ruby Decorator, etc, all work that way. Nevertheless, non of them solves all the problems mentioned in this paper:

Memory footprint. None of the solutions take special care of the memory usage of proxies. This is a real limitation when proxies are being used, *e.g.,* to save memory.

Object replacement. Most proxy solutions can create a proxy for a particular object `X`. The user can then use that proxy as the original object. The problem is that there may be other objects in the system referencing to `X`. Without object replacement, those references will still be pointing to `X` instead of pointing to the proxy. Depending on the proxies usage, this can be a drawback.

Proxies for classes and methods All the investigated solutions create proxies for specific objects but none of them are able to create proxies for class objects or compiled methods.

8. Conclusion

In this paper, we described the Proxy pattern, its different usages and common problems while trying to implement them. We introduced Ghost, a generic, light-weight and stratified Proxy model and its implementation on top of Pharo Smalltalk.

Our solution provides uniform proxies not only for regular instances, but also for classes and methods. In addition, Ghost proxies can have a really small memory footprint. Proxies are powerful, easy to use and extend and its overhead is low.

Ghost was easy to implement on Pharo Smalltalk because the language and the VM provide unique reflective facilities and hooks. Nevertheless, we believe that such specific features, provided by Smalltalk and its VM, can also be ported to other dynamic programming language.

Acknowledgements

This work is supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the CPER 2007-2013.

References

[1] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *TOOLS-Europe'10*, June 2010.

[2] T. Barrett. Dynamic proxies in Java and .NET. *Dr. Dobb's Journal of Software Tools*, 28(7):18, 20, 22, 24, 26, July 2003.

[3] J. K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87*, volume 22, pp 318–330, Dec. 1987.

[4] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[5] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA'04*, pp 331–344, 2004. ACM Press.

[6] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In *ECOOP'98*, LNCS 1445, pp 396–417. Springer-Verlag, 1998.

[7] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[8] Castle dynamicproxy library. http://www.castleproject.org/dynamicproxy/index.html.

[9] cglib code generation library. http://cglib.sourceforge.net.

[10] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[11] P. Eugster. Uniform proxies for java. In *OOPSLA'06*, pp 139–152, 2006.

[12] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings ECOOP '93*, LNCS 707, pp 406–431, 1993.

[13] Y. Hassoun, R. Johnson, and S. Counsell. Applications of dynamic proxies in distributed environments. *Software Practice and Experience*, 35(1):75–99, Jan. 2005.

[14] Oracle. java dynamic proxies. the java platform 1.5 api specification. hhttp://download.oracle.com/javase/1.5.0/docs/api/java/lang/reflect/Proxy.html.

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

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings ECOOP '97*, LNCS 1241, pp 220–242, 1997.

[17] R. Koster and T. Kramp. Loadable smart proxies and native-code shipping for CORBA. In *USM*, LNCS 1890 , pp 202–213, 2000.

[18] Linfu proxies framework. http://www.codeproject.com/KB/cs/LinFuPart1.aspx.

[19] P. Lipton. Java proxies for database objects. http://www.drdobbs.com/windows/184410934, 1999.

[20] P. L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87*, volume 22, pp 331–341, Dec. 1987.

[21] Apple. developer library documentation. http://developer.apple.com/library/ios/#documentation/cocoa/reference/foundation/Classes/NSProxy_Class/Reference/Reference.html.

[22] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86* , volume 21, pp 341–346, Nov. 1986.

[23] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for java futures. In *OOPSLA '04*, pp 206–223, 2004.

[24] N. Santos, P. Marques, and L. Silva. A framework for smart proxies and interceptors in RMI, 2002.

[25] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *ICDCS'86*, pp 198–205, 1986. IEEE Computer Society.

[26] T. Thai and H. Q. Lam. .NET framework essentials / T. thai, H.Q. lam., 2001.

[27] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession apis. *Dynamic Language Symposium*, 45:59–72, 2010.

[28] N. Wang, K. Parameswaran, D. Schmidt, and O. Othman. The design and performance of Meta-Programming mechanisms for object request broker middleware. In *COOTS'01 (USENIX)*, pp 103–118, 2001.

[29] I. Welch and R. Stroud. Dalang - A reflective extension for java, OOPSLA99 Workshop on Reflection, 1999.