# Ghost: A Uniform and General-Purpose Proxy Implementation

Mariano Martinez Peck[1,2,*], Noury Bouraqadi[2,*], Luc Fabresse[2,*], Marcus Denker[1,*], Camille Teruel[1,*]

**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 are unable to create proxies for objects with an important role in the runtime infrastructure such as classes or methods. Proxies can be complex to install, they can have a significant overhead, they can be limited to certain kind of classes, etc. Moreover, proxy implementations are often not stratified and they do not have a clear separation between proxies (the objects intercepting messages) and handlers (the objects handling interceptions).

In this paper, we present Ghost: a uniform and general-purpose proxy implementation for the Pharo programming language. Ghost provides low memory consuming proxies for regular objects as well as for classes and methods.

When a proxy takes the place of a class, it intercepts both the messages received by the class and the lookup of methods for messages received by its 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, Smalltalk

## 1. Introduction

A proxy is an object that acts as a surrogate or placeholder that controls access to another target object. A large number of scenarios and applications have embraced the Proxy Design Pattern [GHVJ93]. Proxy objects are a widely used solution for different scenarios such as remote method invocation [Sha86, SMS02], distributed systems [Ben87, McC87], future objects [PSH04], behavioral reflection [Duc99, KdRB91, WS99], aspect-oriented programming [KLM+97], wrappers [BFJR98], object databases [Lip99], inter-languages communications and bindings, access control and read-only execution [ADD+10], lazy or parallel evaluation, middlewares like CORBA [WPSO01, KK00, HJC05], encapsulators [Pas86], security [VCM10], memory management and object swapping [MPBD+11b, MPBDF11], among others.

Most proxy implementations support proxies for instances of common classes only. Some of them, *e.g.,* Java Dynamic Proxies [jav, Eug06], even need that at creation time the user provides a list of *Java interfaces* for capturing the appropriate messages.

In the context of class-based dynamic object-oriented languages that reify those entities with an important role in the runtime infrastructure such as classes or methods with first-class objects, proxies can be created only for regular objects. Creating *uniform* proxies for other entities such as classes or methods has not been considered. In existing

☆This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

*Corresponding author

*Email addresses:* marianopeck@gmail.com (Mariano Martinez Peck), noury.bouraqadi@mines-douai.fr (Noury Bouraqadi), luc.fabresse@mines-douai.fr (Luc Fabresse), marcus.denker@inria.fr (Marcus Denker), camille.teruel@inria.fr (Camille Teruel)

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1.

[2]Mines-Telecom Institute, Mines Douai, http://car.mines-douai.fr.

work, it is impossible for a proxy to take the place of a class and a method and still be able to intercept messages and perform operations such as logging, security, remote class interaction, etc.

For example, imagine a virtual memory for dynamic languages which goal is to use less memory by only leaving in primary memory what is needed and used, swapping out the unused objects to secondary memory [Kae86, BM08, MPBD+11b, MPBDF11]. To achieve this, the system replaces the original (unused) objects with proxies. When one of the proxies intercepts a message, the original object is brought back into primary memory. In this system, the original objects can be instances of common classes but they can also be methods, classes, method context themselves, etc. Therefore, a proxy implementation must deal with all kind of objects including classes and methods because this weakness strongly limits the application of proxies.

Another property of proxy implementations is memory footprint. As any other object, proxies occupy memory and there are scenarios *e.g.,* the previously mentioned object graph swapper, where the number of proxies and their memory footprint becomes a problem.

Traditional implementations in dynamic languages such as Smalltalk are based on error handling [Pas86]. This results in non stratified proxies meaning that not all messages can be trapped leading to severe limits. Not being able to intercept messages is a problem because those messages will be directly executed by the proxy instead of being intercepted. This can lead to different execution paths in the code, errors or even a VM crash.

Traditionally, proxies not only intercept messages, but they also decide what to do with the interceptions. We argue that these are two different responsibilities that should be separated. Proxies should only intercept, which is a generic operation that can be reused in different contexts. Processing interceptions is application-dependent. It should be the responsibility of another object that we call *handler*.

In this paper, we present Ghost (an extension of our previous work on proxies published as a workshop paper [MPBD+11a]): a uniform and general-purpose proxy implementation for the Pharo programming language [BDN+09]. Ghost provides low memory consuming proxies for regular objects, classes and methods. It is possible to create a proxy that takes the place of a class or a method and that intercepts messages without breaking the system. When a proxy takes the place of a class, it intercepts both the messages received by the class and the lookup of methods for messages received by instances. Similarly, when a proxy takes the place of a method, then the method execution is intercepted too. Last, Ghost supports *controlled stratification*: developers decide which message should be understood by the proxy and which should be intercepted and transmitted for processing to the handler.

The contributions of this paper are:

- Describe and explain the common proxy implementation in dynamic languages and, specially, in Pharo.
- Define a set of criteria to evaluate and compare proxies implementations.
- Present Ghost: a new proxy implementation which solves most of the problems with uniform proxies.
- Validate our solution (Ghost) using the defined criteria.

The remainder of the paper is structured as follows: Section 2 defines and unifies the vocabulary and roles used throughout the paper and presents the list of criteria used to compare different proxy implementations. Section 3 describes the typical proxy implementation and it presents the problem by evaluating it against the previously defined criteria. An introduction to Pharo reflective model and its provided hooks is explained in Section 4. Section 5 introduces and discusses Ghost proxies and shows how the framework works. Section 6 explains how Ghost is able to proxify methods and classes. Certain messages and operations that need special care when using proxies is analyzed in Section 7. Section 8 provides an evaluation of Ghost based on the defined criteria and discusses Ghost model generality. Real case studies of Ghost are presented in Section 9. Finally, in Section 10, related work is presented before concluding in Section 11.

## 2. Proxy Evaluation Criteria

### *2.1. Vocabulary and Roles*

For sake of clarity, we define here the vocabulary used throughout this paper and we highlight the roles that objects are playing when using proxies (see Figure 1).

*Target.* It is the original object that we want to *proxify*.

*Client.* It is an object which uses or holds a reference to a target object. It is this reference that is replaced by one of a proxy.

*Interceptor.* It is the proxy object. But, its responsibility is restricted to the *interception* of 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. Depending on the application, the hander may hold a reference to the target.

One implementation can use the same object for taking the roles of interceptor and handler, *i.e.,* the proxy plays both roles. In another solution, such roles can be supported by different objects. With this approach, the proxy usually takes the role of interceptor.

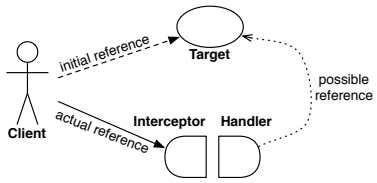


Figure 1: Roles in Proxy.

### *2.2. Proxies Implementation Criteria*

From the implementation point of view, there are criteria that have to be taken into account to compare and characterize different solutions [Duc99, VCM10]:

***Stratification***. Many solutions to implement proxies are based on dedicated messages such as doesNotUnderstand:. The problem with approaches that reserve a set of messages for the proxy implementation is that there is a clash between the API of the proxified object and the proxy implementation.

To address such problem, some solutions proposed stratification [VCM10]. Stratification means that there is a clear separation between the proxy support and application functionalities. In a fully stratified proxy, all messages received by a proxy should be intercepted and forwarded to a handler. The proxy API should not pollute the application's namespace. Besides, having this stratification is important to achieve security and to fully support transparency of proxified objects for the end-programmers [BU04].

Stratification highlights two responsibilities in a proxy toolbox: (1) intercepting messages (interceptor role) and (2) managing interceptions (handler role), *i.e.,* performing actions once messages are intercepted. In a stratified proxy framework, the first responsibility is covered by a proxy itself and the second one by a handler.

***Interception levels***. The interception granularity determines which messages are intercepted by the proxy. There are the following possibilities:

- Intercept *all* messages, even those not defined in the object API *e.g.,* inherited from superclasses.
- Intercept all messages excluding a list of messages defined by the user.
- Intercept all messages excluding some messages imposed by the proxy toolbox *e.g.,* inherited methods if we are using a solution based on error handling such as using the doesNotUnderstand: message.

With the last option, the developer has no control over messages that are not intercepted and hence performed by the proxy itself. This can be a problem because it is impossible to distinguish messages that are performed by the proxy from the ones that are intercepted. 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 consists in making client objects refer to the proxy instead of the target. Two cases exist:

1. Often, the target is an existing object with other objects referencing it. The target may need 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 unused objects and to replace them with proxies. We refer to this functionally as *object replacement*.

2. In the other case, the proxy is just created and it does not replace another already existing object. For example, when doing a query with a database driver, it can create proxies to perform lazy loading on some parts of the graphs. As soon as a proxy receives a message, the database driver loads the rest of the graph. Another example is remote method invocation where targets are located in a different memory space from the clients' one. This means that, in the client memory space, we have proxies that can forward messages and interact with the real objects in the other memory space.

Object replacement is not a feature implemented by the proxy toolbox itself. However, since it allows users to proxify objects, the proxy library must support the case where objects are replaced by proxies. This is a challenge because there are objects that in order to be correctly replaced they need special proxies.

***Uniformity***. We refer to the ability of creating a proxy for any kind of object (regular object, method, class, block, process, etc) and replacing the object with it. Most proxy implementations support proxies only for regular objects *i.e.,* proxies cannot replace a class, a method, a process, etc., without breaking the system. Certain particular objects like nil, true and false cannot be proxified either.

This is an important criterion since there are scenarios where being able to create proxies for any runtime entity is mandatory. As described in Section 9 an example is the mentioned virtual memory which replaces all type of unused objects with proxies

***Transparency***. A proxy is fully transparent if clients are unaffected whether they refer to a proxy or the target. No matter what message the client sends to the proxy, it should answer the same as if it were the target object.

One of the typical problems related to transparency is the identity issue when 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 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 with references to the proxy, clients will only see the proxy. However, this "illusion" will be broken as soon as the target provides its own reference (self) as an answer to a message. This is known as the "self problem" as coined by Henry Lieberman [Lie86].

Another common problem is asking a proxy the class or type since, most of the times, the proxy answers its own type or class instead of the one of the target. The same happens if there is special syntax or operators in the language such as "+", "/", "=", ">" [ADF11]. To have the most transparent proxy possible, these situations should be handled in a way which allows the proxy to behave like the target.

***Efficiency***. The proxy toolbox must be efficient from the performance and memory usage points of view. In addition, we can distinguish between installation performance and runtime performance. For example, for installation, it is commonly evaluated if a proxy installation involves extra overhead like recompilation.

Depending on the usage, the memory footprint of the proxies can be substantial. The space analysis should consider not only the size in memory of the proxies, but also how many objects are needed per target: it can be either only one proxy instance or a proxy instance and a handler instance.

***Ease of debugging***. It is difficult to test and debug in the presence of proxies because debuggers or test frameworks usually send messages to the objects present in the current stack. These messages include, for example, printing an object, accessing its instance variables, etc. When the proxy receives any of these messages it may intercept such message, making debugging more complicated.

*Proxy Toolbox and Implementation.* The proxy toolbox and its implementation can also raise some specific concerns:

***Constraints.*** The toolbox may require, for example, that the target implements certain interface or inherits from a specific class. It is important that the user of the proxy toolbox can easily extend or adapt it to his own needs.

***Portability.*** A proxy implementation can depend on specific entry points of the virtual machine or on certain features provided by the language.

## 3. Common Proxy Implementations

Although there are different proxy implementations and solutions, there is one that is the most common among dynamic programming languages. This implementation is based on error raising and the resulting error handling [Duc99, BDN+09]. 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 object receiving a message is resolved at runtime. When an unknown message is sent to an object, an error exception is thrown. A typical proxy implementation is to create objects that raise errors for all the possible messages (or a subset) and customize the error handling process.

In Pharo, for instance, when an object does not understand a message the Virtual Machine sends the message doesNotUnderstand: to that object with a reification of the message passed as an argument. To avoid infinite recursion, all objects must understand the message doesNotUnderstand:. That is the reason why such method is implemented in the class ProtoObject, the root of the class hierarchy. The default implementation throws a MessageNotUnderstood exception. Similar mechanisms exist in dynamic languages like Ruby, Python, Objective-C and Perl.

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 [McC87, Pas86] 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 10.

*Obtaining a minimal object.* A minimal object is one which understands none or only a few methods [Duc99]. In some programming languages, the root class of the hierarchy chain (usually called Object) already contains several methods. In Pharo, Object inherits from a superclass called ProtoObject which inherits from nil. ProtoObject understands a few messages[3]: the minimal messages that are needed by the system. Here is a simple Proxy implementation.

```
ProtoObject subclass: #Proxy
    instanceVariableNames: 'targetObject'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Proxies'
```

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

Figure 2: Naive proxy implementation based in minimal object and handling not understood methods in Pharo.

*Handling not understood methods.* Common behaviors of proxies include logging before and after the method, forwarding the message to a target object, validating some access control, etc. If needed, it is valid to issue a super send to access the default doesNotUnderstand: behavior.

To be able to forward a message to an object, the virtual machine usually reifies the message. In Pharo, the argument of the doesNotUnderstand: message is an instance of the class Message. It specifies the method selector, the list of arguments and the lookup class (in normal messages it is the class of the receiver and, for super sends, it is the superclass of the class of the method doing the super send). To forward a message to another object, the class Message provides the method sendTo: anotherObject.

This solution is independent of Pharo. For example, the Pharo's doesNotUnderstand: and sendTo: are in Ruby method_missing and send, in Python __getattr__ and getattr, in Perl autoload, in Objective-C forwardInvocation:. In Section 10, we explain some of these examples with more detail.

[3] ProtoObject has 25 methods in PharoCore 1.4.

### 3.2. Evaluation

We now evaluate the common proxy implementation (cf. Section 3) based on the criteria we described in Section 2.2.

*Stratification.* This solution is unstratified:

- 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 [VCM10].
- There is no separation between proxies and handlers even if it would be possible to separate them.

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

*Object replacement.* It is usually unsupported by most programming languages. Nevertheless, Smalltalk implementations do support it using pointer swapping operations such as the become: primitive. However, with such solution, target references may leak when the target remains in the same memory: the target might provide its own reference as a result of some message. This way the client gets a reference to the target so it can by-pass the proxy.

*Uniformity.* There is a severe limit to this implementation since it is not uniform: proxies can only be applied to regular objects. Section 4.1 explains why certain kinds of objects like classes, methods and other core objects need special handling to be correctly proxified.

*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[4] it answers the references to the proxy instead of intercepting the message and forwarding it to a target. The same happens with the identity comparison or when asking the class.

In Pharo, it is possible not only to subclass from ProtoObject but also from nil in which case the subclass do not inherit any method. This solves some of the problems, such as the one of method name collisions, but the solution is still not stratified and makes debugging more complicated.

*Efficiency.* From the speed point of view, this solution is reasonably fast (it is based on two lookups: one for the original message and one for the doesNotUnderstand: message) 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. Regarding the memory usage, there is no optimization.

*Ease of debugging.* The debugger sends messages to the proxy which are not understood and, therefore, intercepted. To be able to debug in presence of proxies, one has to implement all these methods directly in the proxy. The drawback is that the action of enabling or disabling the debugging facilities means adding or removing methods from the proxy. Instead of implementing the methods in the proxy, we could also have a trait (if the language provides traits or any other composable unit of behavior) with such methods. However, we still need to add the trait to the proxy class when debugging and remove it when we are not.

*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 extend or change the purpose of the proxy adapting it to his own needs by just reimplementing the doesNotUnderstand:.

*Portability.* This approach needs just a few requirements that have to be provided by the language and the VM. Moreover, almost all available dynamic languages support these needs by default: a message like doesNotUnderstand:, a minimal object and the possibility to forward a message to another object. Therefore, it is easy to implement this approach in different dynamic languages.

[4] pointersTo is a method implemented in ProtoObject.

## 4. Pharo Support for Proxies

Before presenting Ghost, we first explain the basis of the Pharo reflective model and the provided hooks that our solution uses. We show that Pharo provides, out of the box, all the support we need for Ghost's implementation *i.e.,* object replacement, interception of methods' execution and interception of all messages.

### *4.1. Pharo Reflective Model and VM Overview*

Being a Smalltalk dialect, Pharo inherits the reflective model of Smalltalk-80. There are two important rules [BDN+09]: 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. Figure 3 shows a *simplified* reflective model of Smalltalk.

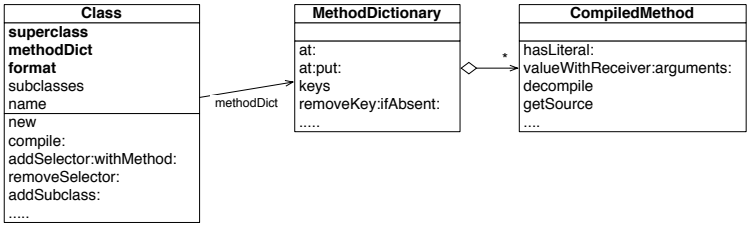


Figure 3: The basic Smalltalk reflective model. Bold instance variables are imposed by virtual machine logic.

Figure 3 shows that a class is defined by a superclass, a method dictionary, an instance format, subclasses, name and a couple of others. Pharo uses the Cog Smalltalk VM [Mir11] and the important point here is that the first two instance variables are imposed by the virtual machine[5]. The method dictionary is a hash table where keys are the methods' names (called selectors in Smalltalk) and the values are the compiled methods which are instances of CompiledMethod.

### *4.2. Hooks and Features Provided by Pharo*

The following is a list of the Pharo reflective facilities and hooks that Ghost uses for implementing proxies.

*Class with no method dictionary.* When an object receives a message, the VM starts the method lookup. During the method lookup, if the method dictionary of one class in the class hierarchy of the receiver is nil, the VM sends the message cannotInterpret: aMessage to the receiver. Contrary to normal messages, the lookup for the method cannotInterpret: starts in the *superclass* of the class whose method dictionary was nil. Otherwise, there would be an infinite loop. This hook is powerful for proxies because it let us intercept *all* messages that are sent to an object.

Figure 4 depicts the following situation: we get one object called myInstance, instance of the class MyClass whose method dictionary is nil. This class has a superclass called MyClassSuperclass. Figure 4 shows how the mechanism works when sending the message printString to myInstance. The message cannotInterpret: is sent to the receiver (myInstance) but starting the lookup in MyClassSuperclass.

*Objects as methods.* This facility allows us to intercept method executions. We can put an object that is not an instance of CompiledMethod in a method dictionary. Here is an example:

```
MyClass methodDict at: #printString put: Proxy 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. Since the object associated with the printString selector is not a compiled method, the VM sends a special message run: aSelector with: arguments in: aReceiver to that object, *i.e.,* the one that replaces the method in the method dictionary.

The VM does not impose any shape to objects acting as methods such as having certain amount of instance variables or certain format. The only requirement is to implement the method run:with:in:.

[5]The VM actually needs three instances variables, the third being the format. But, the format is accessed only by a few operations *e.g.,* instance creation. Since the proxy intercepts all messages including creational ones, the VM will never need to access the format while using a proxy.

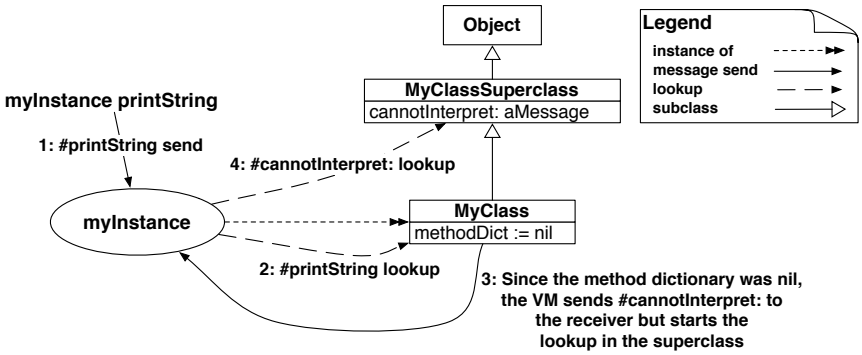


Figure 4: Message handling when a method dictionary is nil.

*Object replacement.* The primitive become: anotherObject is provided by the VM and it atomically swaps the 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 so that they point to the argument, *i.e.,* it is only one way.

This feature enables us to replace a target object with a proxy so that all variables that are pointing to the target object are updated to point to the proxy.

## 5. Ghost's Design and Implementation

Ghost is open-source and developed under the MIT license[6]. The website of the project with its documentation is in: http://rmod.lille.inria.fr/web/pier/software/Marea/GhostProxies. The source code is available in the SqueakSource3 server: http://ss3.gemstone.com/ss/Ghost.html.

### *5.1. Overview Through an Example*

Ghost distinguishes between *interceptors* and *handlers*. Proxies solely play the role of interceptors while handlers define the treatment of the trapped message. Data related to a trapped message is reified as an object we call *interception*. Figure 5 shows Ghost's basic design which is explained in this section. The most important features of Ghost are: (1) to be able to intercept all messages but also to exclude a user-defined list, (2) to be uniform (to be able to proxify any objects, even sensitive ones like classes or method), and (3) to be stratified (*i.e.,* clear separation between proxies and handlers) in a *controlled manner*.

The implementation uses the following reflective facilities: classes with no method dictionary, objects as methods and object replacement. The basic kernel is based on the hierarchies of AbstractProxy (whose role is to intercept messages) and AbstractProxyHandler (whose role is to handle intercepted messages) together with the communication from the former to the latter through Interception instances.

The handlers' responsibility is to manage message interceptions trapped by proxies. What the handler does with the interception, depends on what the user wants. To illustrate the implementation, we use a SimpleForwarderHandler which just forwards the intercepted message to a target object. In this example, each proxy instance uses a particular handler instance which is accessed by the proxy via an instance variable. Another user of the framework may want to use the same handler instance for all proxies. Consequently, different proxies can use the same or different handlers. How proxies are mapped to handlers depends on the user's needs and it is controlled by the method proxyHandler as explained later.

The information passed from a proxy to a handler is reified as an instance of the class Interception. It includes the message which reifies the selector and its arguments, the proxy and the receiver (as we see later sometimes the receiver is not the proxy but a different object).

In real-world scenarios (see Section 9), a proxy often needs to hold some specific information, for example, a target object, an address in secondary memory, a filename, an identifier or any important information. Thus, the proxy

[6] http://www.opensource.org/licenses/mit-license.php

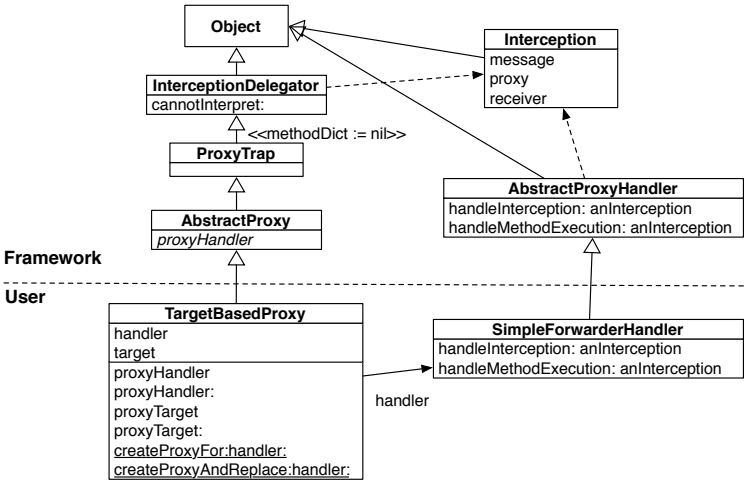


Figure 5: Part of the Ghost framework and an example of proxies for regular objects.

should provide at least an accessor to allow the handler to retrieve this information. The need for these application-specific messages understood by the proxy leads us to *controlled stratification*. In traditional proxy implementations the minimal object already understands some messages and therefore the developer cannot choose not to understand them. With our *controlled stratification*, the *developer controls* and decides the set (usually small) of messages understood by the proxy. By carefully choosing selectors for these messages (usually we use a specific prefix), one avoids collisions with applications messages and enhances proxy's transparency.

A typical reason for controlled stratification is saving memory by sharing a unique handler among all proxies. In the context of the simple forwarder example, we need to map each proxy to a target object that will eventually perform the messages trapped by the proxy. If one goes for full stratification, the proxy will intercept all messages and send them to the handler. But, the handler should hold a reference to the target. Then, for every target object we would have two placeholders: a proxy and a handler. If we use the same object for both responsibilities, then there is no clear division between proxies and handlers. Therefore, in this example, to have a smaller memory footprint, we introduced an instance variable in class TargetBasedProxy that stores the target object. A singleton handler, shared among all proxies, asks each proxy for its target before forwarding the intercepted message. To let the handler access the target object of a given proxy, TargetBasedProxy class implements the method proxyTarget. Section 5.3 explains that Ghost gives the user the flexibility to decide himself where and what data to store.

### *5.2. Proxies for Regular Objects*

This section shows Ghost's implementation for regular objects. Subclasses of AbstractProxy (such as TargetBasedProxy) provide proxies for regular objects, *i.e.,* objects that do not need any special management. Their responsibility is to intercept messages.

*Proxy creation.* The following code shows how to create a proxy for a point (3,4). Since the handler is a simple forwarder, the messages are forwarded to the proxy's target.

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

The class method createProxyFor:handler: creates a new instance of TargetBasedProxy and sets the handler (the user specifies which handler to use just by passing it as a parameter) and the target object.

*Message Trapping in Action.* ProxyTrap is a special class whose method dictionary is nilled out once created. When we send a message to an instance of TargetBasedProxy if the message is not implemented in that class, the method lookup continues in the hierarchy until ProxyTrap, whose method dictionary is nil. For all those messages (the ones not implemented in TargetBasedProxy and AbstractProxy), the VM will eventually send the message cannotInterpret: aMessage. Note that there are a few messages that are not executed but inlined by the compiler and the virtual machine (See Section 7). From now onwards, we consider that when we use cannotInterpret:, we intercept *all* messages except a specific list that we do not want to intercept. This is to distinguish it from doesNotUnderstand: where one can only intercept the messages not understood.

Coming back to the cannotInterpret:, remember that such message is sent to the receiver (in this case the aProxy instance) but the method lookup starts in the superclass of the class whose method dictionary is nil which, in this case, is InterceptionDelegator (see Figure 5). Because of this, InterceptionDelegator implements the mentioned method:

```
InterceptionDelegator >> cannotInterpret: aMessage
    | interception |
    interception := Interception for: aMessage proxy: self.
    ^ self proxyHandler handleInterception: interception.
```

An Interception instance is created and passed to the handler. In this case, for the interception we only need the proxy and the message. The receiver is unused here. InterceptionDelegator sends proxyHandler to get the handler. Therefore, proxyHandler is an abstract method which must be implemented by concrete proxy classes, *e.g.,* TargetBasedProxy and it must answer the handler to use.

Handler classes are user-defined and the example of the simple forwarder handler logs and forwards the received message to a target object as shown below.

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

Direct subclasses from AbstractProxy *e.g.,* TargetBasedProxy are only used for regular objects. We see in the following sections how Ghost handles objects that do require special management such as methods (see Section 6.1) or classes (see Section 6.2).

### *5.3. Extending and Adapting Proxies and Handlers*

To adapt the framework, users have to create their own subclass of AbstractProxyHandler and implement the method handleInterception:. They also need to subclass AbstractProxy and define which handler to use by implementing the method proxyHandler. It is up to the developer to store the handler in the proxy or to share a singleton handler instance among all proxies, or any other option. Other customizations are possible depending on the application's needs:

- Which state to store in the proxy. For example, rather than a simple target object, proxies for remote objects may require an IP, a port and an ID identifying the remote object. A database or object graph swapper may need to store a secondary memory address or ID.
- Which messages are implemented in the proxy and directly answered instead of being intercepted. The most common usage is implementing methods for accessing instance variables so that the handler can invoke them while managing an interception. Next section presents different examples.

### *5.4. Intercepting Everything or Not?*

One would imagine that the best proxy solution is one that intercepts *all* messages. However, this is not what the user of a proxy library needs most of the times. Usually, developers need to send messages to a proxy and get an answer instead of being intercepted. Here are a few examples:

- Storing proxies in hashed collections means that proxies need to answer their hash.

- With remote objects, it is likely that the system will need to ask a proxy its target in the remote system or information about it *e.g.,* URI or ID.
- Serializing proxies to a file or network means that the serializer will ask its class and its instance variables to be serialized as well.
- Debugging, inspecting and printing proxies only makes sense if a proxy answers its own information rather than intercepting the message.

The question "Intercepting Everything or Not?" is really a difficult one. On the one hand, to use a proxy as a placeholder, it is useful that it understands some basic messages such as identityHash, inspect, class, etc. Not only the user of the proxy framework usually needs to send messages to a proxy, but also the proxy toolbox itself *e.g.,*proxyHandler. On the other hand, it is a problem since those messages are not intercepted anymore.

To support these requirements, Ghost provides a flexible design so that proxies can understand and answer specific messages. The way to achieve this is simply by implementing methods in proxy classes. All methods implemented below ProxyTrap in the hierarchy are not intercepted. With our solution, we have the best scenario: *the user* controls stratification and decides *what* to exclude in the proxies interception and intercept *all* the rest. With solutions like doesNotUnderstand:, one can also implement methods in proxy classes to avoid being intercepted but proxies are forced by the system to understand (and hence do *not* intercept) even more messages like those methods that every object understands (e.g. identityHash, initialize, isNil, etc.). Such messages are not defined by the user but by the system.

### *5.5. Messages to be Answered by the Handler*

Apart from the possibility of adding methods to the proxy and avoiding interception of messages, Ghost supports special messages to which the *handler* must answer itself instead of managing them as regular interceptions. A handler keeps a dictionary that maps selectors of messages intercepted by the proxy to selectors of messages to be performed by the handler. This user-defined list of selectors is used with different objectives such as 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 eases the debugging in presence of proxies. The handler's dictionary of special messages for debugging can be defined as follows:

```
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 dictionary keys are selectors of messages received by the proxy and the values are selectors of messages that the handler must send to itself. For example, if the proxy receives the message printString, then the handler sends itself the message handlePrintString: and answers that. All the messages to be sent to the handler (*i.e.,* the dictionary values) take as parameter an instance of Interception which contains the message, the proxy and the receiver. Therefore, such messages have access to all the required information.

These special messages are pluggable *i.e.,* they are easily enabled and disabled. Moreover, they are not coupled with debugging so they can be used every time a user wants certain messages to be implemented and answered directly by the handler rather than performing the default action for an interception. As we explain in the next section, this feature is used, *e.g.,* to intercept method execution.

This feature is similar to the ability of defining methods in the proxy so that they are understood instead of intercepted. Nevertheless, there are some differences which help the user to decide which of the two ways to use in each situation:

- The mechanism of the handler is pluggable, while defining methods in a proxy is not.

- Some methods like those accessing instance variables of the proxy (such as the target object) *have* to be in the proxy. Another example is primitive methods. For example, if we want the proxy to understand proxyInstVarAt: so that it can be used for serialization purposes, we must define this method in the proxy itself because its implementation should use the original `instVarAt:` primitive that uses the current receiver. It is not possible to define the proxyInstVarAt: method in the handler without implementing a new primitive with an additional parameter for the object to instrospect.

- Handlers can be shared among several proxy instances and even different types of proxies. Therefore, we cannot put specific behavior to a shared handler that applies only to a specific type of proxy.

## 6. Proxies for Classes and Methods

In this section, we explain how Ghost also supports proxies for classes and methods in addition to regular objects. Given the possibility to proxify and replace the original object by a proxy for these three kind of entities (Object, Class, Method), Ghost provides all the basic mechanisms to be able to proxify any entity in Pharo. Nevertheless, some entities are directly accessed by the VM such as execution context, block closures or processes, requires special proxies and if we want to support replacement, their proxy must respect the shape expected by the VM (cf. Section 7 and Section 8).

### *6.1. Proxies for Methods*

In some dynamic languages, methods are first-class objects. This means that it is necessary to handle two typical and different scenarios when we want to proxify methods: (1) handling message sending to a proxified method object and (2) handling execution (from the VM) of a proxified method.

- *Sending a message to a method object.* In Pharo, for example, when a developer searches 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 sends 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.

- *Method execution.* This is when the compiled method is executed by the virtual machine. Suppose we want to create a proxy for the method username of User class. We need to intercept the method execution, for example, when doing User new username. Note that this scenario *only* exists if a method is replaced by a proxy.

*Proxy creation.* To clarify, imagine the following test:

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

The test creates an instance of User class and a proxy for method username. Using the method createProxyAndReplace:handler:, we create the proxy and we *replace* the original object (the method username in this case) with it. By replacing an object, we mean that all the pointers to the existing method then point to the proxy. Apart from createProxyAndReplace:handler:, the proxy also understands createProxyFor:handler: which creates the proxy but does replace objects, *i.e.,* object replacement is optional in Ghost.

Finally, we test both types of messages: sending a message to the proxy (in this case mProxy getSource) and executing a proxified method (kurt username in our example).

*Handling both cases.* Ghost solves both scenarios. In the first one, *i.e.,* mProxy getSource, Ghost has nothing special to do. It is just a message sent to a proxy and it behaves exactly the same way we have explained so far. In the second one, illustrated by kurt username, a proxified method is *executed*. In this case, Ghost uses the reflective capability "objects as methods", *i.e.,* when the VM looks for the method username, it notices that, in the method dictionary, there is not a CompiledMethod instance but instead an instance of another class. Consequently, it sends the message run:with:in to such object. Since such object is a proxy in this case, the message run:with:in: is intercepted and delegated to the handler just like any other message.

As already explained, the handler can have a list of messages that require special management rather than performing the default action. With that feature, we map run:with:in to handleMethodExecution:, meaning that if the handler receives an interception with the selector run:with:in it sends to itself handleMethodExecution: and answers that. Subclasses of AbstractProxyHandler that want to handle interceptions of methods' execution must implement handleMethodExecution: to fit their needs, for example:

```
SimpleForwarderHandler >> handleMethodExecution: anInterception
    | targetMethod receiver arguments |
    targetMethod := anInterception proxy proxyTarget.
     "Remember the message was run: aSelector with: arguments in: aReceiver"
    receiver := anInterception message arguments third.
    arguments := anInterception message arguments second.
    ^ targetMethod valueWithReceiver: receiver arguments: arguments
```

That method just gets the required data from the interception and executes the method with the correct receiver and arguments by using the method valueWithReceiver:arguments:.

Notice that the Pharo VM does not impose any shape to methods. Therefore, as we showed in the previous example, we can use the same proxy class (TargetBasedProxy) that we use for regular objects.

*Alternatives.* Another approach to manage interceptions of methods' execution is to implement run:with:in: in the proxy itself. In such situation, we can get the data from the parameters, create an Interception instance and pass it to the handler. However, we believe proxies should understand as little as possible from the proxy toolbox machinery and leave such responsibilities for their handlers.

### *6.2. Proxies for Classes*

Pharo represents classes as first-class objects and they play an important role in the runtime infrastructure. It is because of this that Ghost has to take them into account. Developers often need to be able to replace an existing class with a proxy. Instances hold a reference to their class and the VM uses this reference for the method lookup. Therefore, object replacement must not only update the references from other objects, but also the class references from instances.

AbstractClassProxy provides the basis for class proxies (See Figure 6). AbstractClassProxy is necessary because the VM imposes specific constraints on the memory layout of objects representing classes. The VM expects a class object to have the two instance variables superclass and methodDict in this specific order starting at index 1. We do not want to define TargetBasedClassProxy as a subclass of TargetBasedProxy because the two instance variables target and handler would get index 1 and 2, not respecting the imposed order. However, not being able to subclass is not a real problem in this case because there are only a few methods in common so we are not duplicating much code because of that.

*Requirements.* AbstractClassProxy has to be able to intercept the following kinds of messages:

- Messages that are sent directly to the class as a regular object.
- Messages that are sent to an instance of the proxified class, *i.e.,* an object whose class reference is pointing to the proxy (which happens as a consequence of the object replacement between the class and the proxy). This kind of message is only necessary when an object replacement takes place.

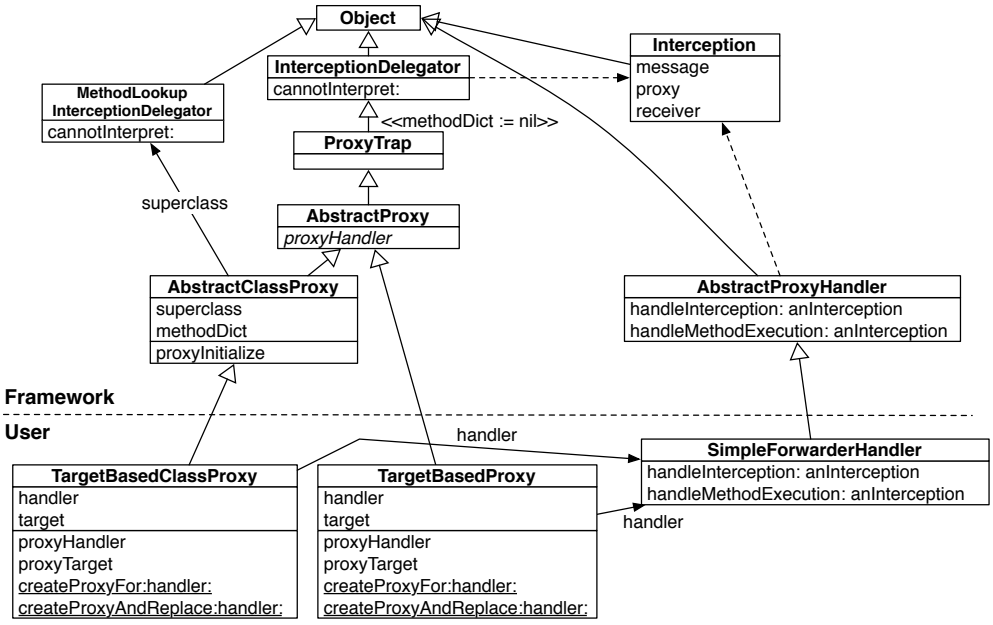


Figure 6: Part of the Ghost framework and an example of proxies for classes.

*Proxy creation.* To explain class proxies, consider the following test:

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

This test creates an instance of User and then, with the message createProxyAndReplace:handler:, it creates a proxy that replaces the User class. Finally, it tests that we can intercept both kind of situations: messages sent to the proxy (in this case User name) and messages sent to instances of the proxified class (kurt username in this case).

*Handling two cases.* The first message, User name, has nothing special and it is handled like any other message. The VM will end up sending cannotInterpret: to the receiver and starting the method lookup in the class which method dictionary was nil, *i.e.,* InterceptionDelegator. The second message is more complicated and needs certain explanation.

The method createProxyAndReplace:handler: is similar to the one of AbstractProxy except that after creating a new proxy it sends the message proxyInitialize to that proxy. In TargetBasedClassProxy we do not only set a handler and a target (as in the case of TargetBasedProxy), but also the minimal information required by the VM so that an *instance* of TargetBasedClassProxy can act as a *class*. These minimal information are: its method dictionary (initialized to nil) and its superclass (initialized to MethodLookupInterceptionDelegator).

Coming back to the example, when we evaluate kurt username, the class reference of kurt is pointing to the created TargetBasedClassProxy instance (as a result of object replacement). This proxy object acts as a class and it has its method dictionary instance variable to nil. Hence, the VM sends the message cannotInterpret: to the receiver (kurt in this case) but starting the method lookup in the superclass of the class with no method dictionary, in this case MethodLookupInterceptionDelegator.

A simplified definition (later in this section we see the real implementation) of the cannotInterpret: of class MethodLookupInterceptionDelegator is the following:

```
MethodLookupInterceptionDelegator >> cannotInterpret: aMessage
    | proxy |
```

```
proxy := aMessage lookupClass.
interception := Interception for: aMessage proxy: proxy receiver: self.
^ proxy proxyHandler handleInterceptionToInstance: interception.
```

It is important to notice the difference between this method and its counterpart in InterceptionDelegator. In both situations, User name and kurt username, we always need to get the proxy to perform the desired action.

User name case. The method cannotInterpret: is called on InterceptionDelegator and the receiver, *i.e.,* what self is pointing to, is the proxy itself.

kurt username case. The method cannotInterpret: is called on MethodLookupInterceptionDelegator and self points to kurt and not to the proxy. The proxy is the looked up class, *i.e.,* the receiver's class, which we can get from the Message instance. Then we send the message handleInterceptionToInstance: to the handler. We use that message instead of handleInterception: because the user may need to perform different actions. What the implementation of handleInterceptionToInstance: does in SimpleForwarderHandler is to execute the desired method with the receiver without sending a message to it[7] avoiding another interception and an infinite loop.

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.

*Discussion.* We could have reused cannotInterpret: implementation of InterceptionDelegator instead of using MethodLookupInterceptionDelegator and set it also in the method proxyInitialize of AbstractClassProxy. That way, InterceptionDelegator is taking care of both types of messages. The solution has to check which kind of message it is and, depending on that, perform a specific action. We think the solution with MethodLookupInterceptionDelegator is much cleaner.

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

*Problem with subclasses of proxified classes.* When we proxify a class but not its instances and one of the instances receives a message, Ghost intercepts the method lookup and finally uses the cannotInterpret: method from MethodLookupInterceptionDelegator. In that method, the proxy can be obtained using aMessage lookupClass, because the class of the receiver object is a proxy. However, this is not always possible. If we proxify a class but not its subclasses and a subclass' instance receives a message which does not match any method, the lookup eventually reaches the proxified class. Ghost intercepts the method lookup and executes the cannotinterpret: method from MethodLookupInterceptionDelegator. At this stage, we need to *find* the trapping class, i.e., the first class in the hierarchy with no method dictionary. In this scenario, message lookupClass does not return a proxy but an actual class: a subclass of the proxified class. To solve this problem, Ghost does the following implementation:

```
MethodLookupInterceptionDelegator >> cannotInterpret: aMessage
    | proxyOrClass proxy |
    proxyOrClass := aMessage lookupClass.
    proxy := proxyOrClass ghostFindClassWithNilMethodDictInHierarchy.
    interception := Interception for: aMessage proxy: proxy receiver: self.
    ^ proxy proxyHandler handleInterceptionToInstance: interception.
```

The method ghostFindClassWithNilMethodDictInHierarchy checks if the method dictionary of the current class is nil and, if it's not, it recurs to the superclass. This method is also implemented in AbstractClassProxy just answering self. Moreover, this method also works well with classes that have no subclass.

[7] Pharo provides the primitive method receiver:withArguments:executeMethod: which directly evaluates a compiled method on a receiver with a specific list of arguments without actually sending a message to the receiver.

## 7. Special Messages and Operations

Being unable to intercept messages is a problem because it means they will be directly executed by the proxy instead. This can lead to different execution paths in the code, errors or even make the VM to crash [ADF11].

One common problem when trying to intercept all messages is the existing optimizations for certain methods. In Pharo, as well as in other languages, there are two kinds of optimizations that affect proxies: (1) inlined methods and (2) special bytecodes.

### *7.1. Inlined Methods*

These are optimizations done by the compiler. For example, messages like ifTrue:, ifNil:, and:, to:do:, etc. are detected by the compiler and are not encoded with the regular bytecode of message send. Instead, such methods are directly encoded using different bytecodes such as jumps. As a result, these methods are never executed and cannot be intercepted by proxies. The second kind of optimization is between the compiler and the virtual machine.

Ideally, we would like to handle inlined messages the same way than regular ones. The easiest yet naive way is to disable the inlining. However, disabling all optimizations produces two important problems. First, the system gets significantly slower. Second, when optimizations are disabled, those methods are executed and there can be unexpected and random problems which are difficult to find. For instance, in Pharo, everything related to managing processes, threads, semaphore, etc., is implemented in Pharo itself without proper abstractions. The processes' scheduler can only switch processes between message sends. This means that there are some parts in the classes like Process, ProcessorScheduler, Semaphore, etc., that have to be atomic, *i.e.,* they cannot be interrupted and switched to another process. If we disable the optimizations, such code is not atomic anymore. Other examples are the methods used to enumerate objects or to get the list of objects pointing to another one. While iterating objects, each send to whileTrue: (or any other of the inlined methods) will create more objects like MethodContext generating an infinite loop.

The messages that are inlined in Pharo 1.4 are stored in the class variable MacroSelectors of the class MessageNode and they are:

1. ifTrue:, ifFalse:, ifTrue:ifFalse:, ifFalse:ifTrue:, and:, or:, implemented in True and False.
2. ifNil:, ifNotNil:, ifNil:ifNotNil: and ifNotNil:ifNil:, implemented in both classes: ProtoObject and UndefinedObject.
3. to:do: and to:by:do:, implemented in Number.
4. whileFalse:, whileTrue:, whileFalse, whileTrue and repeat, implemented in BlockClosure.

These messages involve special objects that can be split into two categories. In the first category, we have true, false, nil, and numbers which are related to the messages of items 1 to 3. These objects are so low-level that they cannot be replaced by proxies. Indeed, sending the become: message to one of them result into VM hang or crash. To our knowledge there is no use case where these objects should be proxified. Hence, inlining their messages is not an issue.

Block closures form the second category of objects which are involved in inlined messages. Actually, the inlining of the messages whileFalse:, whileTrue:, whileFalse, whileTrue and repeat is performed only when the receiver is a closure. In situations where the receiver is the result of a message or a variable like in the code below, the message is not inlined. The following code illustrates these two cases:

```
|welcomeBlock|
[Transcript cr; show: 'Hello'] repeat.  "Message inlined"
welcomeBlock := [Transcript cr; show: 'Welcome'].
welcomeBlock repeat.  "No inlining"
```

Since a proxy is not recognized as closure by the compiler, the message is not inlined. Therefore, messages sent to a proxy for a block closure are intercepted.

*7.2. Special Associated Bytecodes and VM optimization*

The second type of optimization correspond to a special list of selectors[8] that the compiler does not encode with the regular bytecode of message send. Instead, these selectors are associated with special bytecodes that the VM directly interprets. For these selectors, there are three groups:

- Methods that may be executed depending on the receiver or argument type. For example, the execution `1 + 2` never sends the message `+` to 1 because both are 32 bit integers but `1 + 'aString'` will do. Analyzing the implementation of the bytecode associated with each of those selectors shows us that all of them check the type of the receiver and arguments: the method is only executed when there is a type mismatch. For example, all arithmetic operations and bit manipulation expect small integers or floats, boolean operations expect booleans, `size` expects strings or arrays, etc. Whenever the receiver or arguments do not satisfy the conditions, the bytecode follows with the normal method execution, *i.e.,*, the message is sent. Since proxies never satisfy the conditions, then the messages are sent by the VM and trapped like normal messages.
- Methods that are always sent such as `new`, `next`, `nextPut:`, `do:`, etc. Here the only optimization done by the VM is just a quick and internal set of the selector to execute and the argument count. These methods are not a problem for proxies since they are always executed.
- Methods which are never executed but directly answered by the VM internal execution. In Pharo 1.4, there is only one single method of this type[9]: `==`. It answers whether the receiver and the argument are the same object.

The conclusion is that only `==` is not intercepted by proxies. Nevertheless, even if it were possible, object identity should never be handled as a regular message as demonstrated by Mark Miller [10]. However, object replacement conflicts with object identity. For example, given the following code:

```
(anObject == anotherObject)
    ifTrue: [ self doSomething]
    ifFalse: [self doSomethingDifferent]
```

Imagine that `anObject` is replaced by a proxy, *i.e.,* all objects in the system which were referring to the target (`anObject`), will now refer to the proxy. Since all references have been updated, `==` continues to answer correctly. For instance, if `anotherObject` was the same object as `anObject`, `==` answers true since both are referencing the proxy now. If they were not the same object, `==` answers false. Hence, checking identity is not a problem when there is object replacement. However, there is more to object identity than testing for equality. For example, object identity is changed with object replacement and this can affect security, *e.g.,* trusted objects can become untrusted or vice-versa. This is one of the reasons why object replacement is optional in Ghost.

## 8. Ghost Evaluation

This section evaluates Ghost using the criteria defined in Section 2.

*Stratification.* This solution is 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 and it will be intercepted like any other message. Thus, the proxy API does not pollute the application's namespace. Moreover, stratification is controlled: users can still select which messages they do not want to be intercepted.

*Interception levels.* It can intercept all messages while also providing a way to exclude user defined messages.

[8] In Pharo 1.4, we can get the list of those selectors by executing `Smalltalk specialSelectors`. Those are: +, -, <, >, <=, >=, =, ~=, *, /, \\, @, bitShift:, //, bitAnd:, bitOr:, at:, at:put:, size, next, nextPut:, atEnd, ==, class, blockCopy:, value, value:, do:, new, new:, x and y

[9] In earlier versions of Pharo `class` was also of this type, but the special bytecode associated it was removed in Pharo 1.4.

[10] http://erights.org/elib/equality/grant-matcher/index.html

*Object replacement.* Such feature is important since it allows one to seamlessly substitute an object with a proxy and this is provided by Ghost thanks to the become: primitive of Pharo. However, object replacement is optional meaning the user can decide whether to replace objects with proxies or not.

*Uniformity.* This implementation is quite uniform since proxies can be used for regular objects as well as for classes and methods. Ghost does not yet provide out-of-the-box proxies for processes, contexts and blocks. However, it does provide the infrastructure and flexibility to create special proxies for them when needed.

All proxies 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*[11] which contains 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 with proxies. The same happens with immediate objects, *i.e.,* objects that do not have object header and are directly encoded in the memory address such as SmallInteger.

The special objects 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 is Process and it is not possible to correctly replace a Process instance by a proxy. These limitations only occur when object replacement is desired. Otherwise, there is no problem and proxies can be created for those objects. Since classes and methods play an important role in the runtime infrastructure of Pharo, creating proxies for them is useful in several scenarios (as we see in the next section) and that is why Ghost provides special management for them. From our point of view, the mentioned limitations only exist in the presence of unusual needs. Nevertheless, if the user also needs special management for certain objects like Process instances, then he can create a particular proxy that respects the imposed shape.

*Transparency.* Ghost proxies are transparent even with the special messages inlined by the compiler and the VM. The only exception is object identity (message ==). Despite the fact that it is currently not possible to intercept == without modifying the VM, it is usually important to not intercept it because it enables to distinguish between proxies and targets for debugging purpose for example.

In Section 2.2 we mentioned the problem of "target leaking" in which a proxy returns a reference to the target object as an answer to a message. Ghost does not impose any particular state in the proxies and so they may not even have a target object. However, in the example of the SimpleForwarderHandler, which indeed has a target, the developer can solve this problem. When the handler processes an interception it forwards the message to the target object. Then the handler checks whether the answer from the target to that message was the target itself. If it was, then the handler answers the proxy, otherwise the original answer. Of course, this solution is generic (not specific to Ghost), manual (it relies on the developer) and incomplete because it does not apply to indirect reference leak. This simple solution just allows developers to easily solve part of the problem. One general solution to this"*self* leaking problem" is to use "membranes" such as in [VCM13]. But it still relies on the developer will to prevent these leaks so it is out of the proxies responsibilities.

*Efficiency.* Since Ghost interception is also based on two lookups (one for the original message and one for the cannotInterpret:) and the mechanism is similar, it has the same performance than the traditional proxies.

However, Ghost provides an efficient memory usage with the following optimizations:

- TargetBasedProxy and TargetBasedClassProxy are defined as compact classes. This means that in a 32 bits system, their instances' object header is only 4 bytes long instead of 8 bytes. For those instances whose body part is more than 255 bytes and whose class is compact, their header is 8 bytes instead of 12. The first word in the header of regular objects contains flags for the garbage 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 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 language[12] and accessible from the VM. With these 5 bits, there are 32 possible compact classes. This means that, from the language side, the developer can define up to 32 classes as compact. Declaring the proxy classes as compact, allows proxies to have smaller header and, consequently, smaller memory footprint.

[11] Check the method recreateSpecialObjectsArray in Pharo for more details.

[12] See methods SmalltalkImage»compactClassesArray and SmalltalkImage»recreateSpecialObjectsArray.

- Proxies only keep the minimal state they need. AbstractProxy defines no structure and its subclasses may introduce instance variables needed by applications.
- In the methods for creating proxies presented so far (createProxyFor:handler: and createProxyAndReplace:handler:), the last parameter is a handler. This is because, in our example, each proxy holds a reference to the handler. However, this is only necessary when the user's needs one handler instance per target object which is not often the case. The handler is sometimes stateless and can be shared and referenced through a class variable or a global one. In that scenario, proxyHandler must be implemented to answer a singleton. Therefore, we can avoid the memory cost of a handler instance and its reference from the proxy. If we consider that the handler has no instance variable, then it is 4 bytes saved 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.

*Ease of debugging.* Because Ghost provides a way to have messages answered directly by the handler, we can make debugging with proxies very easy. The handler can answer all methods related to debugging, inspecting, etc. In contrast with traditional proxy implementation based on doesNotUnderstand:, this mechanism is pluggable, *i.e.,* it can be enabled or disabled at runtime by just changing a dictionary. There is no need to remove or add methods to the proxy.

*Constraints.* The solution is flexible since the objects to proxify can inherit from any class and are free to implement or not all the methods they want. There is no kind of restriction imposed by Ghost. In addition, the user can easily extend or change the purpose of the toolbox adapting it to his own needs by just subclassing a handler and a proxy.

*Portability.* Ghost is not portable to other Smalltalk dialects because it is based on a VM hook (the cannotInterpret: message) present in the Pharo VM. In addition, it also needs object replacement (become: primitive) and objects as methods (run:with:in: primitive). The become: primitive is present in all Smalltalk dialects because it is used by the language itself. The hook method run:with:in: is not available in all dialects but we only need it if we need to intercept method execution. Furthermore, we rely on the primitive method receiver:withArguments:executeMethod: to be able execute a method on a receiver object without actually sending a message to it. We only use this method when we proxify classes and this method is present in some Smalltalk dialects.

Without these reflective facilities (described in Section 4), we cannot easily implement all the required features of a good proxy library. In the best scenario, we can implement these facilities but with substantial development effort such as modifying the VM or the compiler or even creating them from scratch. Pharo provides all those features by default and no changes are required for Ghost.

Nevertheless, the Ghost model is general. Even if the current implementation of Ghost relies on the cannotInterpret: mechanism to intercept messages, this primitive is only required by few features of Ghost, *i.e.,* stratification and the ability to intercept all messages. The rest of the features, contributions and problems solved by the Ghost model can also be implemented with different hooks to intercept messages *e.g.,* with the doesNotUnderstand:. Indeed, the clear division between proxies and handlers, the special messages in the handler that it can answer itself, the ease of debugging, the ability to proxify methods and classes, to name a few, are independent of the hook used to intercept messages. This means that the Ghost model could be implemented in another Smalltalk dialect and get most of the features provided right now by the Pharo implementation.

The ability to proxify methods and classes is also not mandatory in Ghost. The user can still use the framework for regular proxies and obtain all the mentioned advantages. However, if proxies for classes and methods are needed, then the implementation language must support a way to intercept method execution and a way to deal with objects whose classes are proxies. Pharo provides us both of them.

## 9. Case Study

As a matter of showing possible uses of Ghost proxies, we present in this section the Marea [PBD+13, Mar12] project that does an advanced usage of Ghost.

### 9.1. Marea in a nutshell

In OO software, some objects are only used in certain situations or conditions and remain not used for a long period of time. We qualify such objects as *unused*. These objects are reachable, and thus cannot be garbage collected. This is an issue because unused objects waste primary memory [Kae86].

Operating systems have been supporting virtual memory since a long time [Den70, CH81]. Virtual memory is transparent in the sense that it automatically swaps out unused memory organized in pages governed by some strategies such as the least-recently-used (LRU) [CO72]. As virtual memory is transparent, it does not know the application's memory structure, nor does the application have any way to influence the virtual memory manager.

Marea, is a virtual memory manager whose main goal is to offer the programmer a solution to handle application-level memory [PBD+13, Mar12]. Developers can instruct the system to release primary memory by swapping out *unused objects* to secondary memory [MPBD+11b, MPBDF11]. Marea is designed to: 1) save as much memory as possible *i.e.,* the memory used by its infrastructure is minimal compared to the amount of memory released by swapping out unused objects, 2) minimize the runtime overhead *i.e.,* the swapping process is fast enough to avoid slowing down primary computations of applications, and 3) allow the programmer to control or influence the objects to swap.

The input for Marea are object graphs. These graphs are serialized and moved to secondary memory. Swapped out graphs are swapped in (read from secondary memory and materialized) as soon as one of their elements is needed. Graphs to swap can have *any* shape and contain *any* kind of object. This includes classes, methods, closures and even the execution stack which are all first-class objects in Pharo, Marea's implementation language.

When Marea swaps a graph, it correctly handles all the references from outside and inside the graph. When one of the swapped objects is needed, its graph is *automatically* brought back into primary memory. To achieve this, Marea replaces original objects with proxies (object replacement). Whenever a proxy intercepts a message, it loads back the swapped graph from secondary memory. This process is completely transparent for the developer and the application, *i.e.,* any interaction with a swapped graph has the same results as if it was never swapped.

Marea proxifies and serializes regular objects, methods and classes. This is a challenge since it requires special handling. For the serialization, Marea uses Fuel [DMPDA12], a fast binary object graph serializer. For the proxies toolbox, Marea uses Ghost.

### 9.2. Marea Proxies and Handlers

Marea has its own subclasses of AbstractProxy which do not store a target object but instead, a proxy ID (composed by a graph ID and a position) which is needed by the algorithms to swap out and in. When a proxy intercepts a message, it means that the swapped-out object graph is needed again. Because of this, for every interception MareaProxyHandler (subclass of AbstractProxyHandler) reads the swapped-out object graph from secondary memory (the proxy has the needed information) loading it into primary memory and resending to it the original intercepted message. The following method illustrates this behavior:

```
MareaProxyHandler >> handleInterception: anInterception
    | originalObject |
    originalObject := self loadFromSecondaryMemory: anInterception proxy.
    ^ anInterception message sendTo: originalObject.
```

### 9.3. Requirements and Advantages of Using Ghost

*Proxifying methods and classes.* Typical unused objects are part of the applications' runtime. For example, classes and their methods are loaded on startup but most of them are useless regarding application functionalities. Consequently, applications usually occupy more memory than they actually need. Therefore, Marea needs Ghost to be able to proxify classes and methods.

*Reducing memory occupied by proxies.* Since for Marea it is important that proxies have the minimum memory footprint possible, it takes advantage of some features provided by Ghost:

- Proxies are instances of *compact classes*.
- Since MareaProxyHandler is stateless, it is shared among proxies.

- Marea encodes the proxy instance variables position and graphID in one unique proxyID. The proxyID is a SmallInteger which uses 15 bits for the graphID and 16 bits for the position[13]. Since SmallInteger are immediate objects in Pharo, there is no need for an object header for the proxyID.

*Avoiding unnecessary swap-ins.* Having classes and methods as first-class objects offers solid reflective capabilities. However, some system queries access *all* classes or methods in the system, that may cause the swap in of many of the swapped out graphs. These scenarios happen during application development. As Marea is intended to reduce memory for deployed applications, this is usually not a problem. Still, Marea provides a solution thanks to Ghost capabilities.

The solution requires to have certain messages handled by the proxy itself instead of forwarding it to the handler (that will swap in the graph). The proxy plays the role of a cache by keeping certain information of the proxified object. For example, Marea uses it for the case of class proxies. In Pharo, it is common that the system simply select classes by sending messages such as isBehavior, isClassSide, isInstanceSide, instanceSide or isMeta. Therefore, Marea defined such methods in ClassProxy to answer appropriate results, *i.e.,* isBehavior and isInstanceSide answers true, isClassSide and isMeta answers false and instanceSide answers self. This way, Marea avoids swapping in classes.

Marea also applied this solution to metaclasses and traits. But, it generalizes to any kind of object. An improvement of the previous solution, is to make proxies cache some values from the proxified objects. For instance, a class proxy can cache the class name, a method proxy can cache the literals of the proxified method. There is a trade-off here between sizes of proxies and proxified objects. It is often worth it to have larger proxies if they avoid swapping in large objects or objects that are roots of relatively large object subgraph.

*Interception of all messages.* In Marea, not being able to intercept messages is a problem because those messages will be directly executed by the proxy instead of being intercepted. Therefore, for Marea it is necessary that Ghost is able to intercept all messages.

### *9.4. Marea results*

Marea is not the focus of this paper but we demonstrate Ghost usage through Marea. Thanks to all the mentioned advantages of Ghost, among other reasons, Marea is able to significantly reduce the memory used by applications with a small execution overhead when swapping in objects graphs.

Benchmarks (deeply described in [PBD+13, Mar12]) demonstrate that the memory footprint of different representative real-world applications can be reduced between 25% and 40%. These benchmarks compare memory consumption of real applications loaded in a Pharo environment ready for production (already shrinked) with and without Marea. One of the reasons of Marea's good results is because Ghost proxies use a small memory footprint.

An application will have no execution time penalty at all until a swapping in is required. And we report in [Mar12] that in average, it takes about 40 milliseconds to swap in or swap out a graph of 50 to 777 objects and most of the time (60% in average) is spent to achieve object replacement (`become:`) in Pharo[14].

## 10. Related Work

### *10.1. Proxies in dynamic languages*

*Objective-C.* Objective-C provides a proxy implementation called NSProxy [15]. This solution consists of an abstract class NSProxy that implements the minimum number of methods needed to be a root class. Indeed, this class is not a subclass of NSObject (the root class of 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 proxy is of a subclass of NSProxy with a new instance variable to store a reference to the proxified object. Then, when a proxy receives a message, it forwards it to the proxified object using the stored reference.

[13] Marea also provides large proxies when the limits are exceeded.

[14] The `become:` implementation in Pharo is slow because it requires a memory traversal. Other Smalltalk implementations use a faster implementation based on object tables for example.

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

Since Objective-C is a dynamic language, it needs to provide a mechanism similar to the Smalltalk doesNotUnderstand: when an object receives a message that cannot understand. When a message is not understood, the Objective-C runtime sends 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. By implementing the method forwardInvocation:, a subclass can define how to process the invocation *e.g.,* forwarding it over the network. The method methodSignatureForSelector: is required to provide argument type information for a given message. A subclass' implementation should be able to determine the argument types for the messages it needs to forward and it should be able to build a NSMethodSignature object accordingly. Note that, from this point of view, Objective-C is not so dynamic.

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 intercepted. This solution is similar to the common solution in Smalltalk with doesNotUnderstand:.

*Ruby.* In Ruby, there is a proxy implementation which is called Delegator. It is just a class included in Ruby standard library but which can be easily modified or implemented from scratch. Similar to Objective-C and Smalltalk (and indeed, to most dynamic languages), Ruby provides a mechanism used when an object receives a message that it does not understand. This method is called method_missing(aSelector, *args). Moreover, from Ruby version 1.9, there is a new minimal class called BasicObject which understands a few methods and is similar to ProtoObject in Pharo.

Ruby's proxies are similar to Smalltalk's proxies using doesNotUnderstand: and to Objective-C' NSProxy as they have a minimal object (subclass from BasicObject) and implement method_missing(aSelector, *args) to intercept messages.

*Javascript.* Mozilla's Spidermonkey engine has long included a nonstandard way of intercepting method calls based on Smalltalk's doesNotUnderstand:. The equivalent method is named noSuchMethod. Such solution is not stratified and it only intercepts the messages that are not understood.

Nevertheless, Van Cutsem et al. give another implementation of proxies for Javascript [VCM10]. They provide a clear division between proxies and handlers. Similarly to our possibility of creating proxies for methods, they can create proxies for Javascript functions since they are objects too. As well as we do, they have several reasons to avoid intercepting === (object identity in Javascript).

Javascript is a prototype-based language which, besides function calls, has more language constructs. From what we can understand, their solution requires the VM to be changed so that each of these operators can check whether the receiver is a proxy or not, redirecting the invocation to the handler rather than following the normal steps when the answer is positive. They also provide a way for the user to specify a list of properties that are answered directly by the proxy instead of being intercepted. We believe their work takes a more reflective standpoint. They want to reify additional operations such as instance variable access for example. In Smalltalk, this is not needed for proxies since most of these operations are already reified.

The authors also claim that having only one trap message *e.g.,*doesNotUnderstand: does not scale if we were to introduce additional traps to intercept not only method invocation, but also property access, assignment, lookup, enumeration, etc. Contrary to them, we argue that having a small number of VM hooks makes the VM code simpler. For example, Ghost relies on a regular VM (non modified Cog Smalltalk VM [Mir11]) which have only one hook that enables trapping message sends and it covers most of those cases. This is possible because Ghost is based on the Pharo object model that uses message sending for almost all language features. For example, accessing the properties of an object is achieved by sending it a message. Enumerations are also just messages. Even the lookup can be intercepted with Ghost by using proxies for classes. The items that Ghost is currently unable to intercept are: assignment or direct instance variable access because these features are not reified and the current VM does not provide a way to intercept them.

Another difference is that Ghost enables to transparently replace an object, a class or a method by a proxy (eventually a special one to also dupe the VM). This powerful feature is only possible thanks to the `become:` primitive provided by the Cog Smalltalk VM.

### 10.2. Proxies in static languages

*Java.* Java, being a statically-typed language, supports quite limited proxies called *Dynamic Proxies*. It relies on the Proxy class from the java.lang.reflect package. 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 the declarations in the provided interfaces will be intercepted by a proxy instance of the generated class and delegated to a handler object.

Java proxies have the following limitations:

- The user *cannot* create a proxy for instances of a class which has not all its methods declared in interfaces. This means that, if the user wants to create a proxy for a domain class, he is forced to create an interface for it. Eugster [Eug06] proposed a solution which provides proxies for classes. There is also a third-party framework based on bytecode manipulation called CGLib [16] 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. Since Java proxies require interfaces, private methods cannot be intercepted either. Depending on the proxy usage, this can be a problem.
- Proxies are subclass from Object forcing them to understand several messages *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. Moreover, there are some specific exceptions: 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.

*.Net.* Microsoft's .NET platform [TL01] proposes a closely related concept of Java dynamic proxies with nearly the same limitations. There are other third-party libraries like *Castle DynamicProxy*[17] or *LinFu*[18]. DynamicProxy differs from the proxy implementation built into .NET which requires the proxified class to extend MarshalByRefObject. This is a too heavy constraint since instances of classes that do not subclass MashalByRefObject cannot be proxified. In LinFu, every generated proxy dynamically overrides all of its parent's virtual methods. Each of its respective overridden method implementations delegates each method call to the attached interceptor object. However, none of them can intercept non-virtual methods.

### 10.3. Comparison

Statically typed languages, such as Java or .NET, support quite limited proxies [Bar03]. Java and .Net suffer from the lack of replacement 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 meaning no handler sharing is possible which ends in a bigger memory footprint. Proxies are far more powerful, flexible, transparent and easy to implement in dynamic languages than in static ones.

Dynamic languages just need two features to implement a basic 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, none of them solve all the problems mentioned in this paper:

**Uniformity.** All the investigated solutions create proxies for specific objects but none of them are able to create proxies for classes or methods.

**Object replacement.** Some proxy solutions can create a proxy for a particular object X. The user can then use that proxy instead of the original object. The problem is that there may be other objects in the system referencing 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 limitation.

**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 to save memory.

[16] cglib Code Generation Library: http://cglib.sourceforge.net.
[17] Castle DynamicProxy library: http://www.castleproject.org/dynamicproxy/index.html
[18] Linfu proxies framework: http://www.codeproject.com/KB/cs/LinFuPart1.aspx

## 11. Conclusion

In this paper, we described the need for proxies, their different usages and common problems while trying to implement them. We introduced Ghost: a generic proxy implementation on top of Pharo Smalltalk.

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

Ghost's implementation takes advantages of Pharo VM reflective facilities and hooks. Nevertheless, we believe that such specific features, provided by Pharo and its VM, can also be ported to other dynamic programming language.


[ADD+10] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference Objects, Models, Components, Patterns (TOOLS'10)*, Malaga, Spain, June 2010.

[ADF11] Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on object oriented programming systems languages and applications*, OOPSLA '11, pages 921–938, New York, NY, USA, 2011. ACM.

[Bar03] Tom Barrett. Dynamic proxies in Java and .NET. *Dr. Dobb's Journal of Software Tools*, 28(7):18, 20, 22, 24, 26, July 2003.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Ben87] John K. Bennett. The design and implementation of distributed Smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 318–330, New York, NY, USA, 1987. ACM.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[BM08] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In Gail E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CH81] Richard W. Carr and John L. Hennessy. WSCLOCK a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 87–95, New York, NY, USA, 1981. ACM.

[CO72] Wesley W. Chu and Holger Opderbeck. The page fault frequency replacement algorithm. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, AFIPS '72 (Fall, part I), pages 597–609, New York, NY, USA, 1972. ACM.

[Den70] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[DMPDA12] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Fuel: A fast general-purpose object graph serializer. *Journal of Software: Practice and Experience*, 2012.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[Eug06] Patrick Eugster. Uniform proxies for Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM.

[GHVJ93] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[HJC05] Youssef Hassoun, Roger Johnson, and Steve Counsell. Applications of dynamic proxies in distributed environments. *Software Practice and Experience*, 35(1):75–99, January 2005.

[jav] 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.

[Kae86] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):87–106, November 1986.

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

[KK00] Rainer Koster and Thorsten Kramp. Loadable smart proxies and native-code shipping for CORBA. In Claudia Linnhoff-Popien and Heinz-Gerd Hegering, editors, *Trends in Distributed Systems: Towards a Universal Service Market, Third International IFIP/GI Working Conference, USM 2000, Munich, Germany, September 12-14, 2000, Proceedings*, volume 1890 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2000.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[Lie86] Henry Lieberman. Using protypical objects to implement shared behavior. In *Object-Oriented Systems (OOPSLA'86 Proceedings)*, 1986.

[Lip99] Paul Lipton. Java proxies for database objects. http://www.drdobbs.com/windows/184410934, 1999.

[Mar12] Martinez Peck Mariano. *Application-Level Virtual Memory for Object-Oriented Systems*. PhD thesis, Université de Lille, 2012.

[McC87] Paul L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 331–341, December 1987.

[Mir11] Eliot Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

[MPBD+11a] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Efficient proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*, Edinburgh, Scotland, 2011.

[MPBD+11b] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Problems and challenges when building a manager for unused objects. In *Proceedings of Smalltalks 2011 International Workshop*, Bernal, Buenos Aires, Argentina, 2011.

[MPBDF11] Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse, and Luc Fabresse. Object swapping challenges: an evaluation of ImageSegment. *Journal of Computer Languages, Systems and Structures*, 38(1):1–15, nov 2011.

[Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 341–346, November 1986.

[PBD+13] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Marea: An efficient application-level object graph swapper. *Journal of Object Technology*, 12(1):2:1–30, jan 2013.

[PSH04] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for Java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.

[Sha86] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 198–205, Washington, DC, 1986. IEEE Computer Society.

[SMS02] Nuno Santos, Paulo Marques, and Luis Silva. A framework for smart proxies and interceptors in RMI, 2002.

[TL01] Thuan Thai and Hoang Q. Lam. .NET framework essentials / T. thai, H.Q. lam., 2001.

[VCM10] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010.

[VCM13] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178. Springer Berlin Heidelberg, 2013.

[WPSO01] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt, and Ossama Othman. The design and performance of Meta-Programming mechanisms for object request broker middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-01)*, pages 103–118, Berkeley, California, 2001. USENIX Association.

[WS99] Ian Welch and Robert Stroud. Dalang - A reflective extension for Java, February 1999.