Mariano Martinez Peck
marianopeck@gmail.com
http://marianopeck.wordpress.com/
What is a proxy?

A proxy object is a surrogate or placeholder that controls access to another target object.
**Glossary**

- **Target**: object to proxify.
- **Client**: user of the proxy.
- **Interceptor**: object that intercepts message sending.
- **Handler**: object that performs a desired action as a consequence of an interception.
Forwarder and Logging example

```
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxify: target.
sel self assert: aProxy username = 'Mariano'. |
```

```
handleInterception: anInterception

Transcript show: 'The method ', anInterception message, ' was intercepted'; cr.
sel self forwardInterceptionToTarget: anInterception.
Transcript show: 'The method ', anInterception message, ' was forwarded to target'; cr.
```

```
username
Transcript show: 'username method'; cr.
^ username
```

```
Transcript
The method #username was intercepted
username method
The method #username was forwarded to target
```
In proxies with object replacement (#become:), the target object is replaced by a proxy.

Proxies without object replacement are a kind of factory.
With or without Object replacement?

In proxies with object replacement (#become:), the target object is replaced by a proxy.

Proxies without object replacement are a kind of factory.
Usage of a minimal object together with an implementation of a custom `#doesNotUnderstand`

```
ProtoObject
identityHash
pointersTo
nextObject
...
```

```
<table>
<thead>
<tr>
<th>Proxy</th>
</tr>
</thead>
</table>
target |
| #doesNotUnderstand |
| executeBeforeHandling |
| executeAfterHandling |
```

```
<table>
<thead>
<tr>
<th>doesNotUnderstand: aMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>self executeBeforeHandling</td>
</tr>
<tr>
<td>result := aMessage sentTo: target.</td>
</tr>
<tr>
<td>self executeAfterHandling</td>
</tr>
<tr>
<td>^ result</td>
</tr>
</tbody>
</table>
```
We are going to play a little game...
Are both prints in Transcript the same or not?
Are both prints in Transcript the same or not?

Conclusion: methods understood are NOT intercepted. Is that bad?
Are both prints in Transcript the same or not?

Conclusion: methods understood are NOT intercepted. Is that bad?

Different execution paths Errors difficult to find
Do we want the regular `#doesNotUnderstand` or to intercept the message?
Do we want the regular `#doesNotUnderstand` or to intercept the message?
I wanted the normal #doesNotUnderstand!!!
executeAfterHandling
    self methodDoesNotExist

User Interrupt
Proxy>>doesNotUnderstand: #methodDoesNotExist
Proxy>>executeBeforeHandling
Proxy>>doesNotUnderstand: #methodDoesNotExist
Proxy>>executeBeforeHandling
Proxy>>doesNotUnderstand: #methodDoesNotExist
Proxy>>executeBeforeHandling
Proxy>>doesNotUnderstand: #methodDoesNotExist
Proxy>>executeBeforeHandling
Proxy>>doesNotUnderstand: #methodDoesNotExist
Proxyv>>doesNotUnderstand: #methodDoesNotExist

doesNotUnderstand: aMessage
    self executeBeforeHandling.
    ^aMessage sendTo: target

I wanted the normal #doesNotUnderstand!!!
Problems

- #doesNotUnderstand: cannot be trapped like a regular message.

- Mix of handling procedure and proxy interception.

- Only methods that are not understood are intercepted.

- No separation between proxies and handlers

This approach is not stratified
Subclassing from \textit{nil} does not solve the problem.
This solution is not uniform
A Uniform, Light-weight and Stratified Proxy Model and Implementation.
Used hooks

- Object replacement (#become:)
- Change an object’s class (#adoptInstance:)
- Objects as methods (#run:with:in:)
- Classes with no method dictionary (#cannotInterpret:)
Object Replacement

A - B - D - C

c become: d

A - B - D - C
Objects as methods

testRunWithin

<table>
<thead>
<tr>
<th>target aProxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>target := User named: 'Mariano'.</td>
</tr>
<tr>
<td>aProxy := Proxy proxyFor: (User methodDict at: #username).</td>
</tr>
<tr>
<td>User methodDict at: #username put: aProxy.</td>
</tr>
<tr>
<td>target username.</td>
</tr>
</tbody>
</table>

The VM sends #run: aSelector with: anArray in: aReceiver
The VM sends `#run: aSelector with: anArray in: aReceiver`

So.....We can implement in Proxy:

```smalltalk
run: aSelector with: anArray in: aReceiver
| result |
self executeBeforeMethodExecution.
result := aReceiver withArgs: anArray executeMethod: target.
sel self executeAfterMethodExecution.
^ result
```
CLASSES WITH NO METHOD DICTIONARY

3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass.

References
- instance of
- message send
- lookup
- subclass

Proxy
- cannotInterpret: aMessage

ProxyTrap
- methodDict := nil

2: #aProxy lookup

4: #cannotInterpret: lookup

1: #username send

aProxy

aProxy username

Object
Ghost model

Object

Proxy
handler
target
cannotInterpret:
proxyFor:

ProxyTrap
initialize
nilMethodDict

Interception
message
proxy
proxyState

ProxyHandler
handleInterception: anInterception

SimpleForwarderHandler
handleInterception: anInterception

Monday, August 22, 2011
**How it works?**

```plaintext
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.
```

Diagram:
- **Object**
  - **Proxy**
    - handler
    - target
    - cannotInterpret:
      - proxyFor:
  - **ProxyTrap**
    - initialize
    - nilMethodDict

Monday, August 22, 2011
How it works?

```
testRegularObject
| target aProxy |
  target := User named: 'Mariano'.
  aProxy := Proxy proxyFor: target.
  self assert: aProxy username equals: 'Mariano'.
```

Proxy class >>

```
proxyFor: anObject
| aProxy |
  aProxy := self new
    initializeWith: anObject
    handler: SimpleForwarderHandler new.
  ProxyTrap initialize.
  ProxyTrap adoptInstance: aProxy.
  ^ aProxy.
```
How it works?

```plaintext
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.
```

Proxy class >>

```plaintext
proxyFor: anObject
| aProxy |
aProxy := self new
  initializeWith: anObject
  handler: SimpleForwarderHandler new.
ProxyTrap initialize.
ProxyTrap adoptInstance: aProxy.
^ aProxy.
```

Diagram:
- **Object**
  - **Proxy**
    - **handler**
    - **target**
    - **proxyFor:**
  - **ProxyTrap**
    - **initialize**
    - **nilMethodDict**
- **aProxy**
- **aHandler**
- **‘Mariano’**
How it works?

testRegularObject
| target aProxy |
| target := User named: 'Mariano'.
| aProxy := Proxy proxyFor: target.
| self assert: aProxy username equals: 'Mariano'.

Proxy class >>

proxyFor: anObject
| aProxy |
| aProxy := self new
  initializeWith: anObject
  handler: SimpleForwarderHandler new.
| ProxyTrap initialize.
| Proxy Trap adoptInstance: aProxy.
| ^ aProxy.

ProxyTrap class >>

initialize
| superclass := Proxy.
| format := Proxy format.
| methodDict := nil.

Monday, August 22, 2011
**How it works?**

```plaintext
testRegularObject
| target aProxy |
  target := User named: 'Mariano'.
  aProxy := Proxy proxyFor: target.
  self assert: aProxy username equals: 'Mariano'.
```

**Proxy class >>**

```plaintext
proxyFor: anObject
| aProxy |
  aProxy := self new
  initializeWith: anObject
  handler: SimpleForwarderHandler new.
  ProxyTrap initialize.
  ProxyTrap adoptInstance: aProxy.
```

**ProxyTrap class >>**

```plaintext
initialize
  superclass := Proxy.
  format := Proxy format.
  methodDict := nil.
```

---

Monday, August 22, 2011
How it works?

```ruby
testRegularObject
<table>
<thead>
<tr>
<th>target aProxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>target := User named: 'Mariano'.</td>
</tr>
<tr>
<td>aProxy := Proxy proxyFor: target.</td>
</tr>
<tr>
<td>self assert: aProxy username equals: 'Mariano'.</td>
</tr>
</tbody>
</table>
```

Proxy class »

```ruby
proxyFor: anObject
<table>
<thead>
<tr>
<th>aProxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>aProxy := self new</td>
</tr>
<tr>
<td>initializeWith: anObject</td>
</tr>
<tr>
<td>handler: SimpleForwarderHandler new.</td>
</tr>
<tr>
<td>ProxyTrap initialize.</td>
</tr>
<tr>
<td>ProxyTrap adoptInstance: aProxy.</td>
</tr>
<tr>
<td>^ aProxy.</td>
</tr>
</tbody>
</table>
```

Object

- Proxy
  - handler
  - target
  - cannotInterpret:
    - proxyFor:
  - ProxyTrap
    - initialize
    - nilMethodDict

```
```
How it works?

```plaintext
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.
```

2: #aProxy lookup
3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass
4: #cannotInterpret: lookup

3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass

- References
  - instance of
  - message send
  - lookup
  - subclass

- aProxy username
  - 1: #username send
  - 2: #aProxy lookup
  - 4: #cannotInterpret: lookup

- Proxy
  - cannotInterpret: aMessage

- ProxyTrap
  - methodDict := nil
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.

1: #username send

References
instance of --> message send --> lookup --> subclass

3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass

4: #cannotInterpret: lookup

References
ProxyTrap
methodDict := nil

Proxy
cannotInterpret: aMessage

Object

aProxy

1: #username send

2: #aProxy lookup
How it works?

```plaintext
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
sel assert: aProxy username equals: 'Mariano'.
```

1: #username send

2: #aProxy lookup

3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass

4: #cannotInterpret: lookup

ProxyTrap
methodDict := nil

References
instance of message send lookup subclass

aProxy username

Object

cannotInterpret: aMessage

Proxy

How it works?
**How it works?**

```ruby
testRegularObject
| target aProxy |
| target := User named: 'Mariana'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.
```

1: `#username send`

3: Since the method dictionary was nil, the VM sends `#cannotInterpret` to the receiver but starting the lookup in the superclass.

4: `#cannotInterpret: lookup`

References:
- instance of
- message send
- lookup
- subclass

Object

Proxy
- `cannotInterpret: aMessage`

ProxyTrap
- `methodDict := nil`

aProxy

1: `#username send`

2: `#aProxy lookup`

3: Since the method dictionary was nil, the VM sends `#cannotInterpret` to the receiver but starting the lookup in the superclass.
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.

How it works?

1: #username send
2: #aProxy lookup
3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass
4: #cannotInterpret: lookup

References
instance of message send lookup subclass

Object
Proxy
cannotInterpret: aMessage
ProxyTrap
methodDict := nil

Monday, August 22, 2011
How it works?

```smalltalk
testRegularObject
| target aProxy |
target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.

self assert: aProxy username equals: 'Mariano'.
```

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

1: #username send
2: #aProxy lookup
3: Since the method dictionary was nil, the VM sends #cannotInterpret to the receiver but starting the lookup in the superclass
4: #cannotInterpret: lookup

References
- instance of
- message send
- lookup
- subclass
testRegularObject
| target aProxy |
| target := User named: 'Mariano'.
aProxy := Proxy proxyFor: target.
self assert: aProxy username equals: 'Mariano'.

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

SimpleForwarderHandler >>
handleInterception: anInterception
| message result |
| Transcript show: 'message , anInterception message selector, , intercepted'; cr.
result := anInterception message sendTo: anInterception target.
Transcript show: 'message , anInterception message selector, , was forwarded to target'; cr.
^ result
<table>
<thead>
<tr>
<th>Traditional</th>
<th>Ghost</th>
</tr>
</thead>
<tbody>
<tr>
<td>#doesNotUnderstand: cannot be trapped like a regular message.</td>
<td>#cannotInterpret: is trapped like a regular message.</td>
</tr>
<tr>
<td>Mix of handling procedure and proxy interception.</td>
<td>No mix of handling procedure and proxy interception.</td>
</tr>
<tr>
<td><strong>Only</strong> methods that are not understood are intercepted.</td>
<td>“All” methods are intercepted.</td>
</tr>
<tr>
<td>No separation between proxies and handlers.</td>
<td>Clear separation between proxies and handlers.</td>
</tr>
</tbody>
</table>
Ghost is stratified
Methods not intercepted

1) Optimizations done by the Compiler

2) Special shortcut bytecodes between Compiler and VM

2.1) Methods NEVER sent: == and class
2.2) Methods that may or may not be executed depending on the receiver and arguments: e.g. in ‘1+1’ #+ is not executed. But with ‘1+$C’ #+ is executed.
2.3) Always executed, they are just little optimizations. Examples #new, #next, #nextPut:, #size, etc.
Methods not intercepted

1) Optimizations done by the Compiler

2) Special shortcut bytecodes between Compiler and VM

2.1) Methods NEVER sent: #== and #class
2.2) Methods that may or may not be executed depending on the receiver and arguments: e.g. in ‘1+1’ #+ is not executed. But with ‘1+$C’ #+ is executed.
2.3) Always executed, they are just little optimizations. Examples #new, #next, #nextPut:, #size, etc.
Methods not intercepted

1) Optimizations done by the Compiler

   initialize "MessageNode initialize"
   MacroSelectors :=
   #( ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue: ifNil: ifNil:"ifNil:"
   and: or: whileFalse: whileTrue: whileFalse whileTrue to:by:do:
   caseOf: caseOf:otherwise:
   ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil:)

2) Special shortcut bytecodes between Compiler and VM

   2.1) Methods NEVER sent: #== and #class
   2.2) Methods that may or may not be executed depending on the receiver and arguments: e.g. in ‘1+1’ #+ is not executed. But with ‘1+$C’ #+ is executed.
   2.3) Always executed, they are just little optimizations. Examples #new, #next, #nextPut:, #size, etc.
testProxyForClass
| aProxy aUser |
| aUser := User named: 'Kurt'.
aProxy := ClassProxy createProxyAndReplace: User.
self assert: User name equals: #User.
self assert: aUser username equals: 'Kurt'.

Object

Proxy
handler
target
cannotInterpret:
proxyFor:

ClassProxy
superclass
methodDict
format
handler
target
cannotInterpret:
proxyFor:

ProxyTrap
initialize
nilMethodDict

ClassProxy Trap
initialize
nilMethodDict

User
username
age
logIn:
validate:
testProxyForClass
<table>
<thead>
<tr>
<th>aProxy aUser</th>
</tr>
</thead>
</table>
| aUser := User named: 'Kurt'.
| aProxy := ClassProxy createProxyAndReplace: User. |
| self assert: User name equals: #User. |
| self assert: aUser username equals: 'Kurt'. |
testProxyForClass

<table>
<thead>
<tr>
<th>aProxy aUser</th>
</tr>
</thead>
<tbody>
<tr>
<td>aUser := User named: 'Kurt'.</td>
</tr>
<tr>
<td>aProxy := ClassProxy createProxyAndReplace: User.</td>
</tr>
<tr>
<td>self assert: User name equals: #User.</td>
</tr>
<tr>
<td>self assert: aUser username equals: 'Kurt'.</td>
</tr>
</tbody>
</table>
testProxyForClass
| aProxy aUser |
aUser := User named: 'Kurt'.
aProxy := ClassProxy createProxyAndReplace: User.
self assert: User name equals: #User.
self assert: aUser username equals: 'Kurt'.

ClassProxy
superclass
methodDict
format
handler
target
cannotInterpret:
proxyFor:

ClassProxyTrap
initialize
nilMethodDict

Proxy
handler
target
cannotInterpret:
proxyFor:

ProxyTrap
initialize
nilMethodDict

User
username
age
logIn:
validate:
Proxy for classes

testProxyForClass
| aProxy aUser |
aUser := User named: 'Kurt'.
\textcolor{red}{aProxy} := ClassProxy \textcolor{red}{createProxyAndReplace:} User.
sel\textcolor{red}{self assert: User name equals: #User.}
sel\textcolor{red}{aUser username equals: 'Kurt'.}

createProxyAndReplace: \textcolor{red}{aClass}
| aProxy newProxyRef newClassRef|
aProxy := self new
\textcolor{red}{initializeWith: SimpleForwarderHandler new methodDict: nil}
superclass: ClassProxy
format: ClassProxy format.
aProxy become: \textcolor{red}{aClass}.
"After the \textcolor{red}{become} is done, aProxy now points to aClass and aClass points to aProxy. We create two new variables just to clarify the code"
newProxyRef := aClass.
newClassRef := aProxy.
newProxyRef \textcolor{red}{target} := newClassRef.
ClassProxyTrap \textcolor{red}{initialize}.
ClassProxyTrap \textcolor{red}{adoptInstance} := newProxyRef.
^ newProxyRef.

Proxy
handler
target
cannotInterpret: proxyFor:

ProxyTrap
initialize
nilMethodDict

User
username
age
logIn:
validate:

\textcolor{red}{aUser}
instance of

\textcolor{red}{aClassProxy}
methodDict = nil
superclass = ClassProxy
testProxyForClass
| aProxy aUser |
aUser := User named: 'Kurt'.
aProxy := ClassProxy createProxyAndReplace: User.
self assert: User name equals: #User.
self assert: aUser username equals: 'Kurt'.

createProxyAndReplace: aClass
| aProxy newProxyRef newClassRef |
aProxy := self new
   initializeWith: SimpleForwarderHandler new
   methodDict: nil
   superclass: ClassProxy
   format: ClassProxy format.
aProxy become: aClass.
"After the become is done, aProxy now points to aClass
and aClass points to aProxy. We create two new variables
just to clarify the code"
newProxyRef := aClass.
ewClassRef := aProxy.
ewProxyRef target: newClassRef.
ClassProxyTrap initialize.
ClassProxyTrap adoptInstance: newProxyRef.
^ newProxyRef.
testProxyForClass
| aProxy aUser |
aUser := User named: 'Kurt'.
aProxy := ClassProxy createProxyAndReplace: User.
self assert: User name equals: #User.
self assert: aUser username equals: 'Kurt'.

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

<table>
<thead>
<tr>
<th>aProxy aUser method</th>
</tr>
</thead>
</table>
aUser := User named: 'Kurt'.
method := User methodDict at: #username.
aProxy := Proxy createProxyAndReplace: method.
self assert: aProxy getSource equals: 'username ^ username'.
self assert: aUser username equals: 'Kurt'.
Proxy for methods

testProxyForMethod
| aProxy aUser method |
aUser := User named: 'Kurt'.
method := User methodDict at: #username.
aProxy := Proxy createProxyAndReplace: method.
self assert: aProxy getSource equals: 'username ^ username'.
self assert: aUser username equals: 'Kurt'.

Regular message
Proxy for methods

testProxyForMethod
| aProxy aUser method |
aUser := User named: 'Kurt'.
method := User methodDict at: #username.
aProxy := Proxy createProxyAndReplace: method.
self assert: aProxy getSource equals: 'username ^ username'.
self assert: aUser username equals: 'Kurt'.

Regular message
Method execution
Proxy for methods

Just handling `#run:with:in` correctly is enough to also intercept method execution.
is Uniform
Ghost

IS more UNIFORM

Monday, August 22, 2011
More features

- Low memory footprint.
  - Compact classes.
  - Store the minimal needed state.
- Easy debugging.
  - Custom list of messages.
Conclusion

With a little bit of special support from the VM (#cannotInterpret hook), we can have an image-side proxy solution much better than the classic #doesNotUnderstand:
Future work

- Experiment with immediate proxies (memory address tag) in VM side.
- Think how to correctly intercept non-executed methods.
Links

- [http://rmod.lille.inria.fr/web/pier/software/Marea/GhostProxies](http://rmod.lille.inria.fr/web/pier/software/Marea/GhostProxies)

- [http://www.squeaksource.com/Marea.html](http://www.squeaksource.com/Marea.html)
A Uniform, Light-weight and Stratified Proxy Model and Implementation

Mariano Martinez Peck
marianopeck@gmail.com
http://marianopeck.wordpress.com/