Ghost

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

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE | INRIA

RMod

Ecole d'Ingénieurs Centre de Recherche Mines de Douai LILLE EUROREGION

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

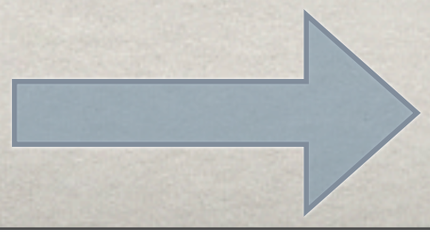
Workspace

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

```
handleInterception: anInterception

    Transcript show: 'The method ', anInterception message, ' was intercepted'; cr.
    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
```

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.

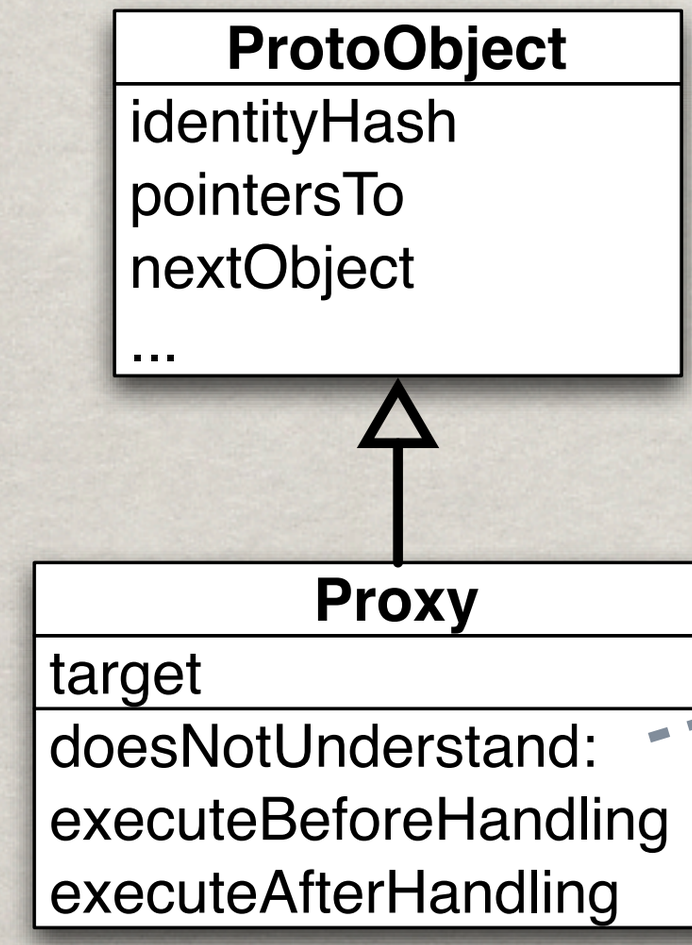
With or without Object replacement?

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

- Proxies without object replacement are a kind of factory.

Traditional proxy implementations

Usage of a minimal object together with an implementation of a custom #doesNotUnderstand




```
doesNotUnderstand: aMessage
    | result |
    self executeBeforeHandling.
    result := aMessage sentTo: target.
    self executeAfterHandling.
    ^ result
```

We are going to play a little game...

```
testMethodAlreadyUnderstood
    | target aProxy |
    target := User named: 'Mariano'.
    Transcript show: 'Target identityHash: ', target identityHash asString; cr.
    aProxy := Proxy proxyFor: target.
    Transcript show: 'Target identityHash: ', aProxy identityHash asString; cr.
```

Are both prints in Transcript the same or not?

```
testMethodAlreadyUnderstood
    | target aProxy |
    target := User named: 'Mariano'.
    Transcript show: 'Target identityHash: ', target identityHash asString; cr.
    aProxy := Proxy proxyFor: target.
    Transcript show: 'Target identityHash: ', aProxy identityHash asString; cr.
```

Are both prints in Transcript the same or not?

Transcript

```
Target identityHash: 533200896
Target identityHash: 894959616
```

Conclusion: methods understood are NOT intercepted.

Is that bad?

Workspace

```
| anObject anotherObject |
"  "
 .....
anObject identityHash = anotherObject identityHash
    ifTrue: [ self doSomethingWhenSameIdentityHash ]
    ifFalse: [ self doSomethingWhenDifferentIdentityHash ]
```

```
testMethodAlreadyUnderstood
    | target aProxy |
    target := User named: 'Mariano'.
    Transcript show: 'Target identityHash: ', target identityHash asString; cr.
    aProxy := Proxy proxyFor: target.
    Transcript show: 'Target identityHash: ', aProxy identityHash asString; cr.
```

Are both prints in Transcript the same or not?

Transcript

```
Target identityHash: 533200896
Target identityHash: 894959616
```

Conclusion: methods understood are NOT intercepted.

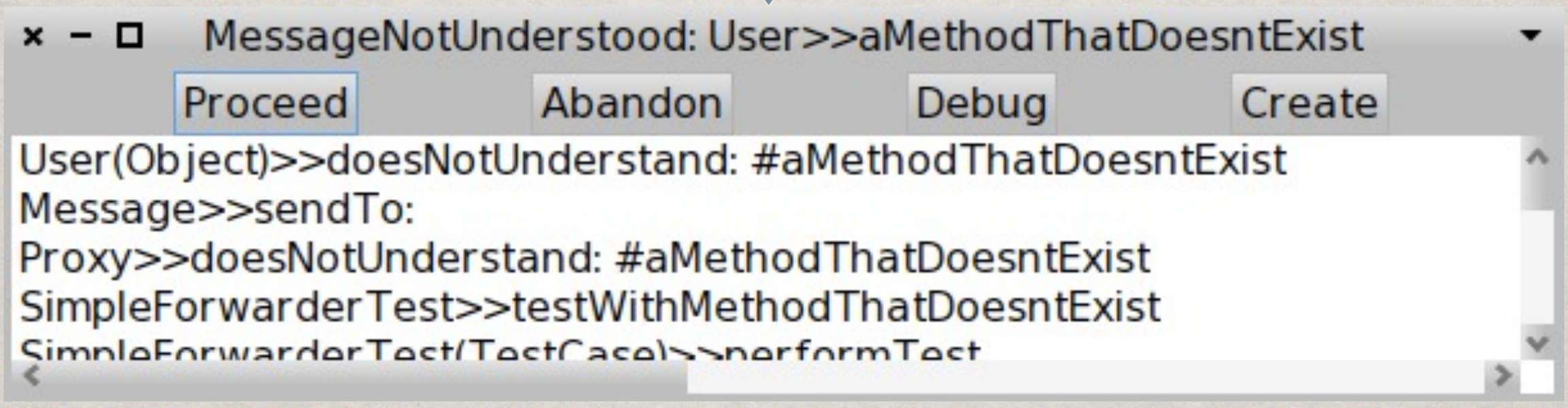
Is that bad?

Workspace

```
| anObject anotherObject |
"  "
 .....
anObject identityHash = anotherObject identityHash
    ifTrue: [ self doSomethingWhenSameIdentityHash ]
    ifFalse: [ self doSomethingWhenDifferentIdentityHash ]
```

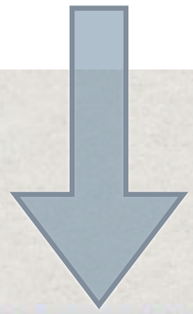
Different execution paths → Errors difficult to find

```
testWithMethodThatDoesntExist
    | target aProxy |
    target := User named: 'Mariano'.
    aProxy := Proxy proxyFor: target.
    aProxy aMethodThatDoesntExist.
```



Do we want the regular #doesNotUnderstand
or to intercept the message?

```
testWithMethodThatDoesntExist2
    Proxy  new aMethodThatDoesntExist.
```



MessageNotUnderstood: receiver of "aMethodThatDoesntExist" is nil

UndefinedObject(Object)>>doesNotUnderstand: #aMethodThatDoesntExist
Message>>sendTo:
Proxy>>doesNotUnderstand: #aMethodThatDoesntExist

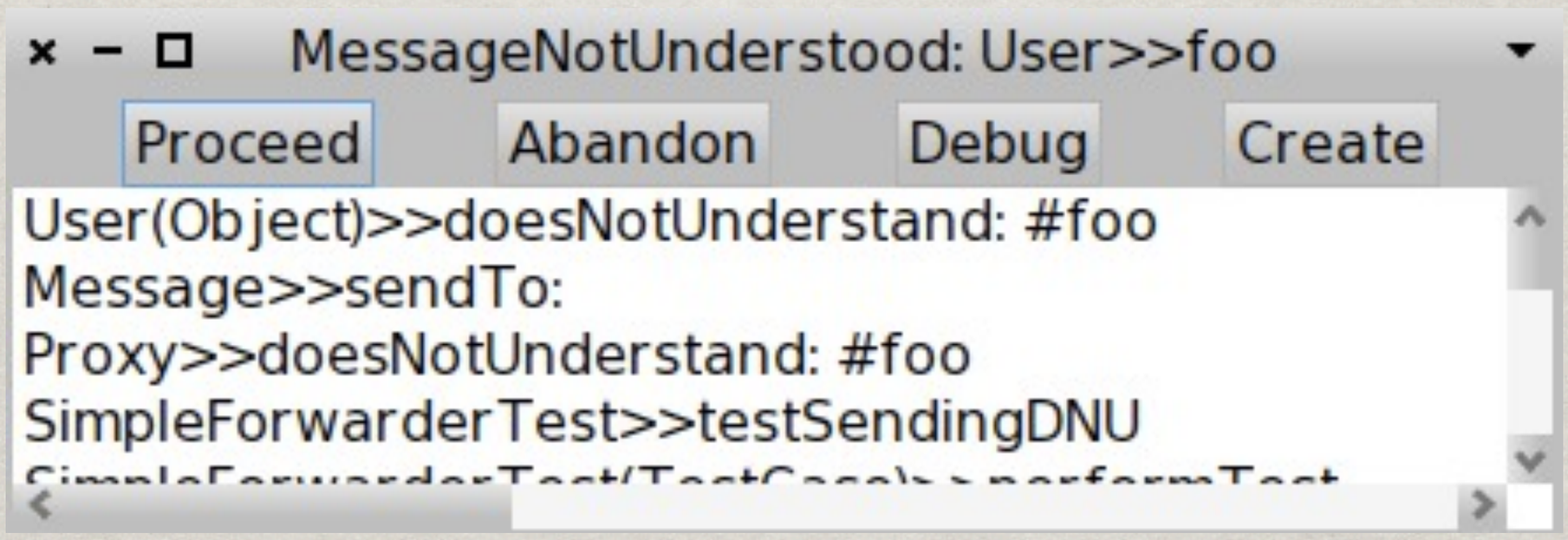
Proceed | Restart | Into | Over | Through | Full Stack | Run to Here | Where | Create

```
sendTo: receiver
    "answer the result of sending this message to receiver"

    ^ receiver perform: selector withArguments: args
```

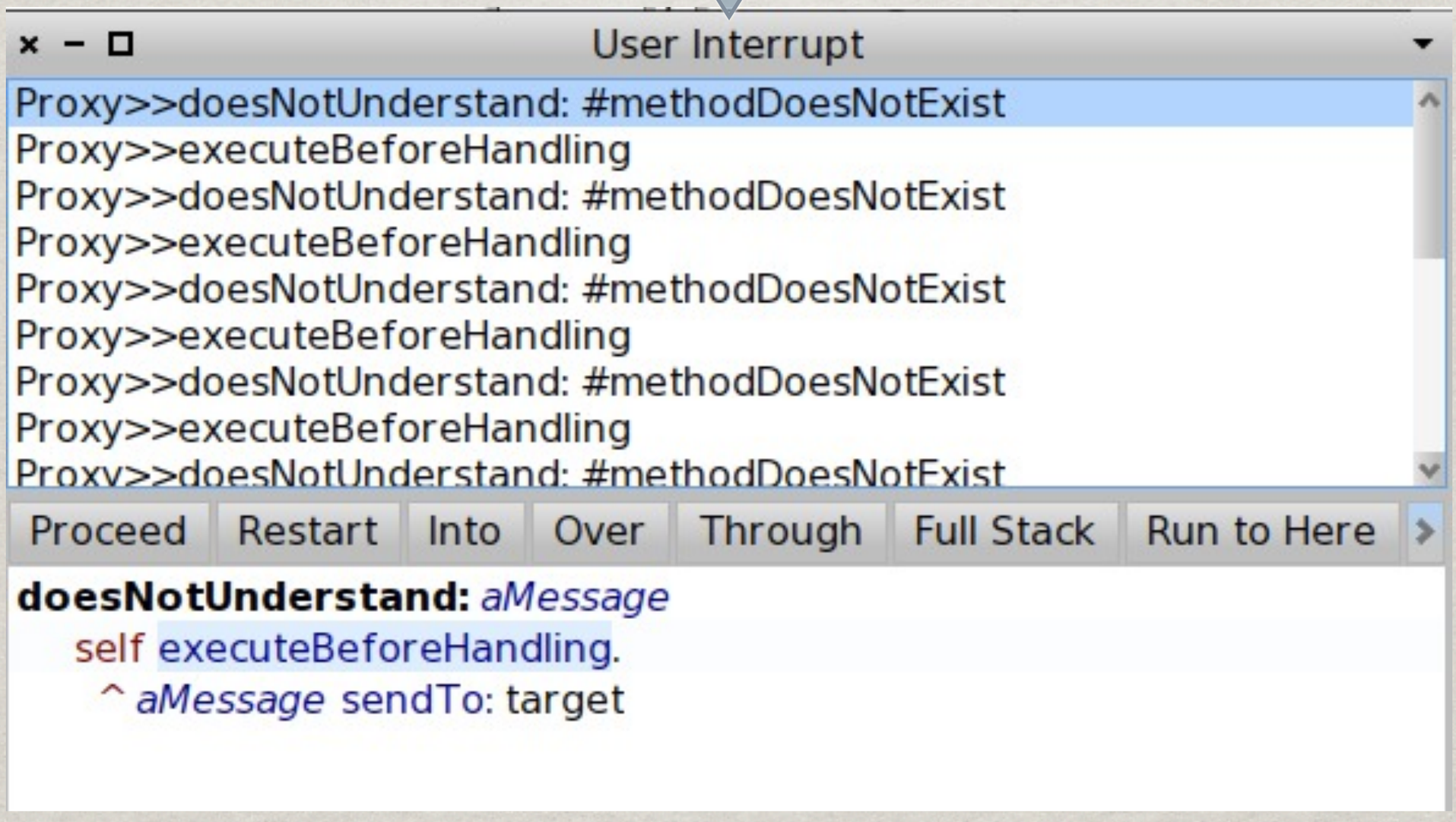
Do we want the regular #doesNotUnderstand
or to intercept the message?

```
testSendingDNU
    | target aProxy aMessage |
    target := User named: 'Mariano'.
    aMessage := Message selector: #foo argument: #().
    aProxy := Proxy proxyFor: target.
    aProxy doesNotUnderstand: aMessage
```



I wanted the normal #doesNotUnderstand!!!

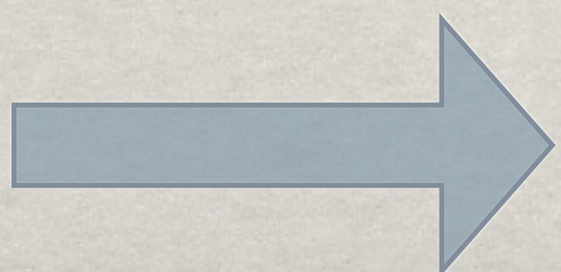
```
executeAfterHandling
    self methodDoesNotExist
```



I wanted the normal #doesNotUnderstand!!!

Problems

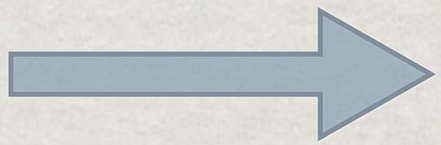
- #doesNotUnderstand: cannot be trapped like a regular message.
- Mix of handling procedure and proxy interception.
- **Only** methods that are not understood are intercepted.
- No separation between proxies and handlers



This approach is not **stratified**

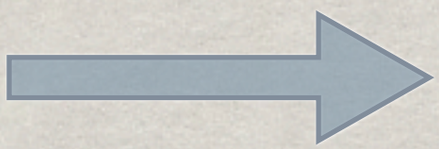
Subclassing from *nil* does not solve the problem.

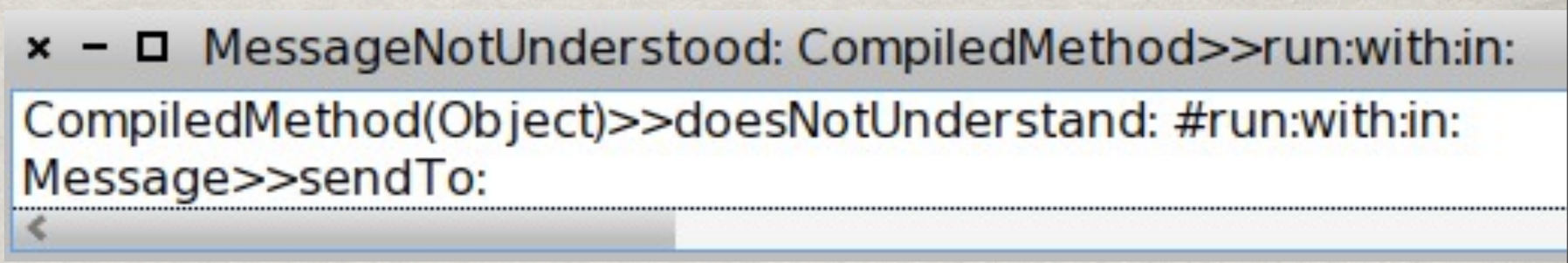
```
testWithClass
    | target aProxy |
    target := User named: 'Mariano'.
    aProxy := Proxy proxyFor: User.
    target username.
```



VM CRASH

```
testWithCompiledMethod
    | target aProxy |
    target := User named: 'Mariano'.
    aProxy := Proxy proxyFor: (User >> #username).
    target username.
```





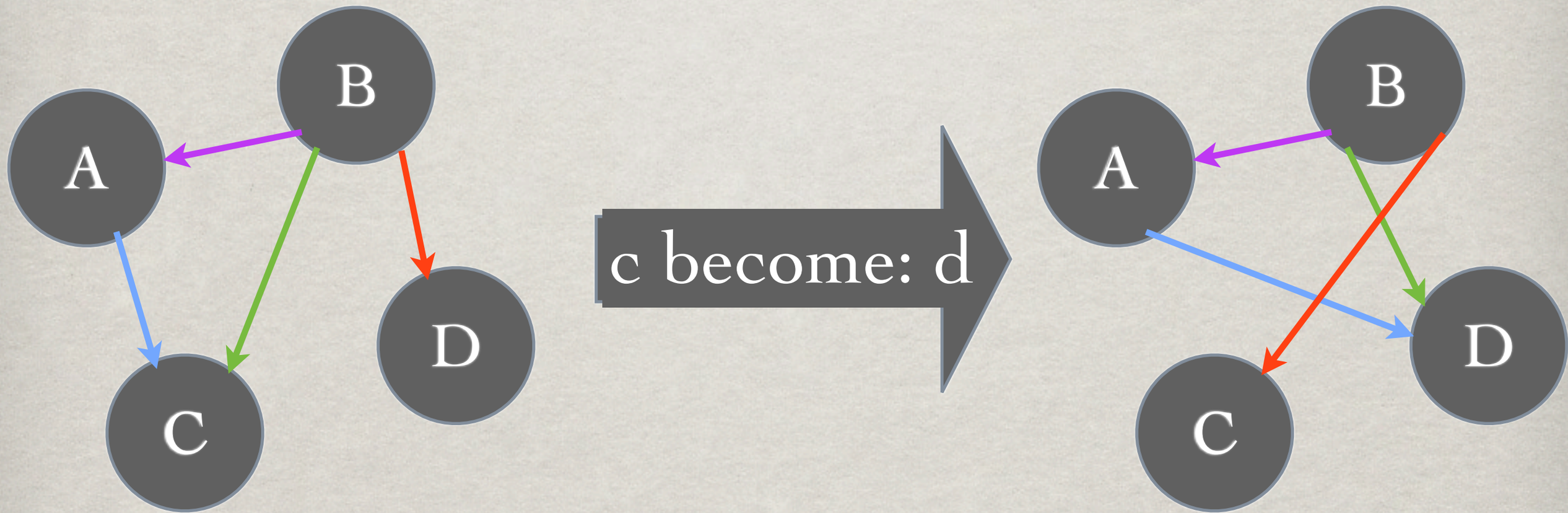
This solution is not **uniform**

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



Objects as methods

```
testRunWithIn
    | target aProxy |
    target := User named: 'Mariano'.
    aProxy := Proxy proxyFor: (User methodDict at: #username).
    User methodDict at: #username put: aProxy.
    target username.
```

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

Objects as methods

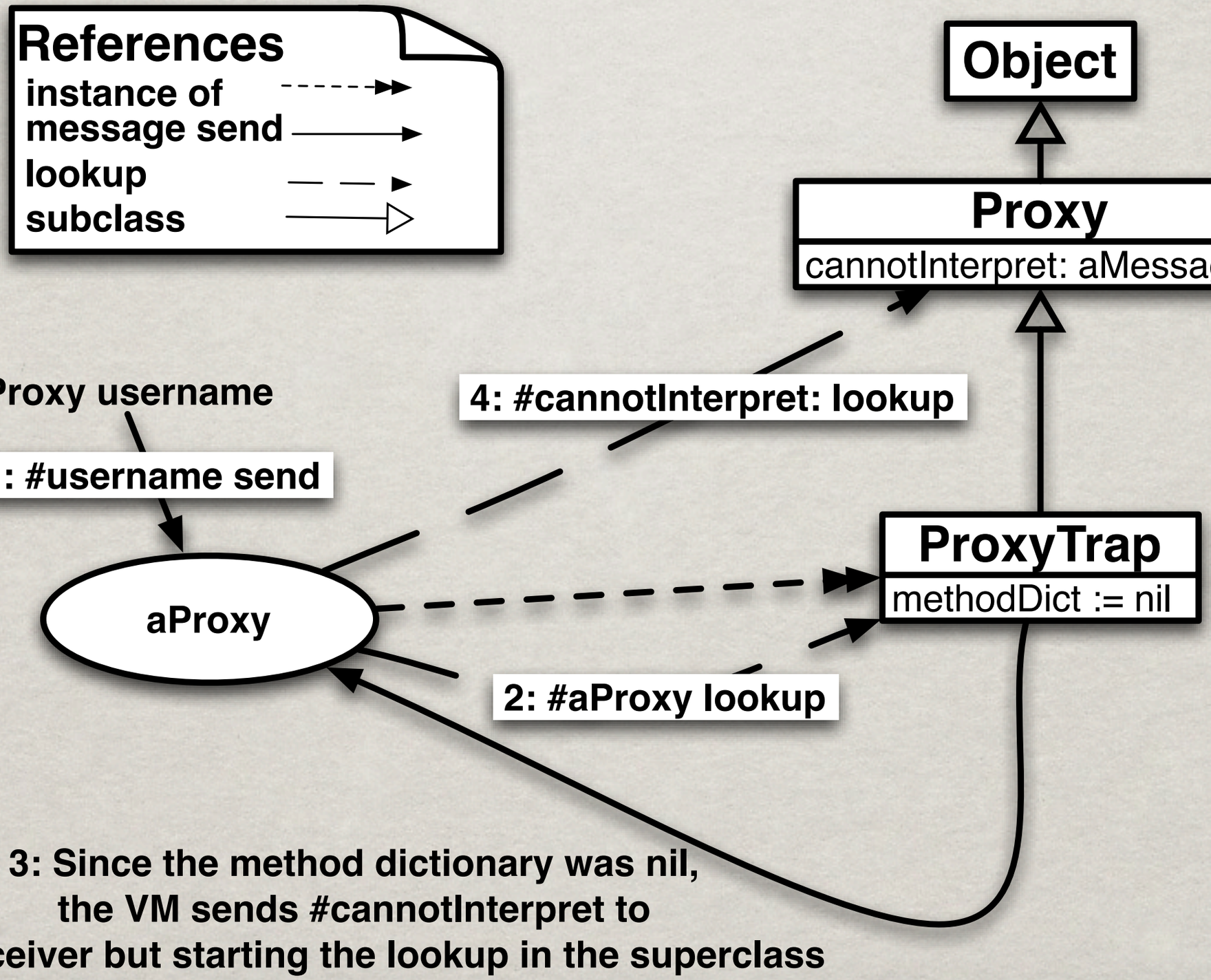
```
testRunWithIn
    | target aProxy |
    target := User named: 'Mariano'.
    aProxy := Proxy proxyFor: (User methodDict at: #username).
    User methodDict at: #username put: aProxy.
    target username.
```

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

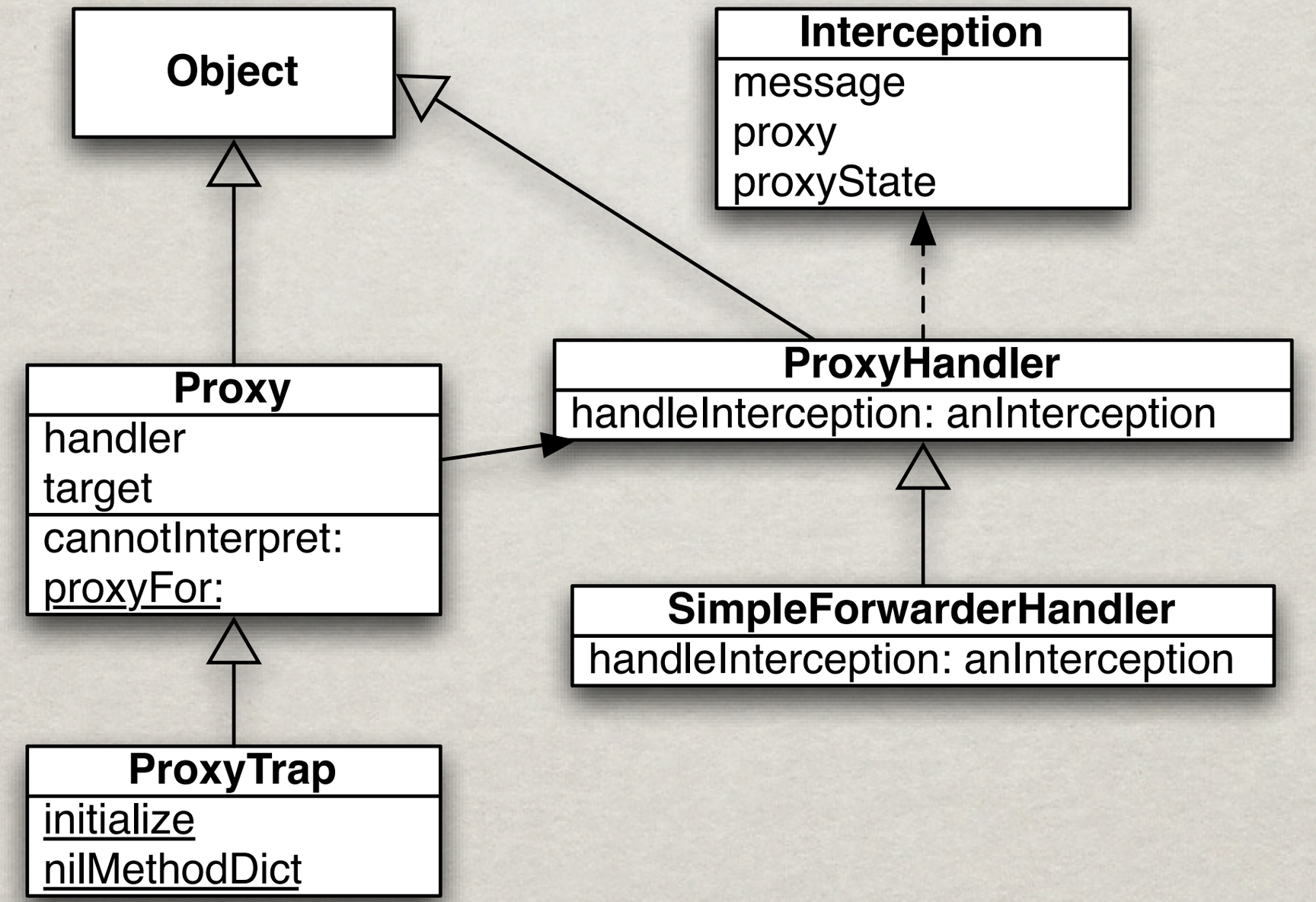
So.....We can implement in Proxy:

```
run: aSelector with: anArray in: aReceiver
    | result |
    self executeBeforeMethodExecution.
    result := aReceiver withArgs: anArray executeMethod: target.
    self executeAfterMethodExecution.
    ^ result
```

Classes with no method dictionary

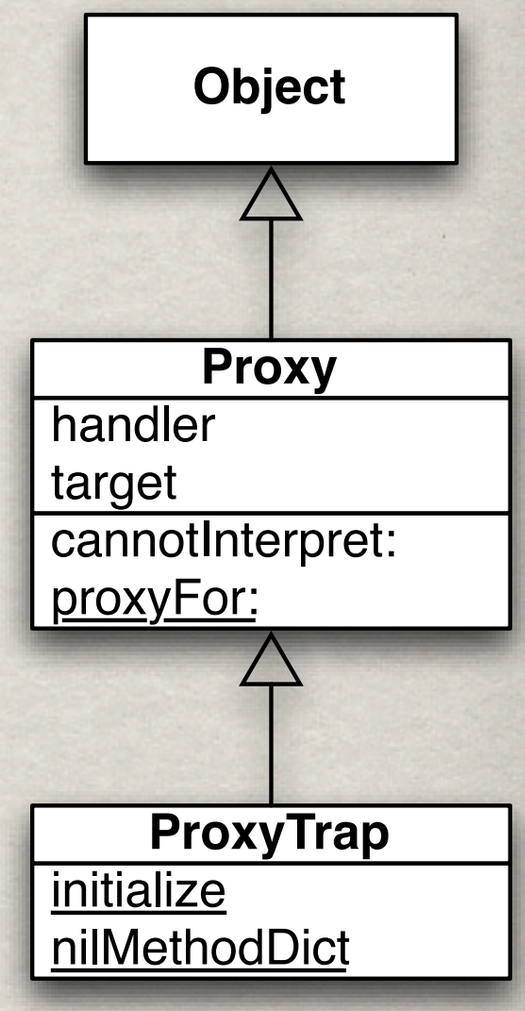


Ghost model


Object
Interception
message
proxy
proxyState
Proxy
handler
target
cannotInterpret:
proxyFor:
ProxyHandler
handleInterception: anInterception
SimpleForwarderHandler
handleInterception: anInterception
ProxyTrap
initialize
nilMethodDict

How it works?

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

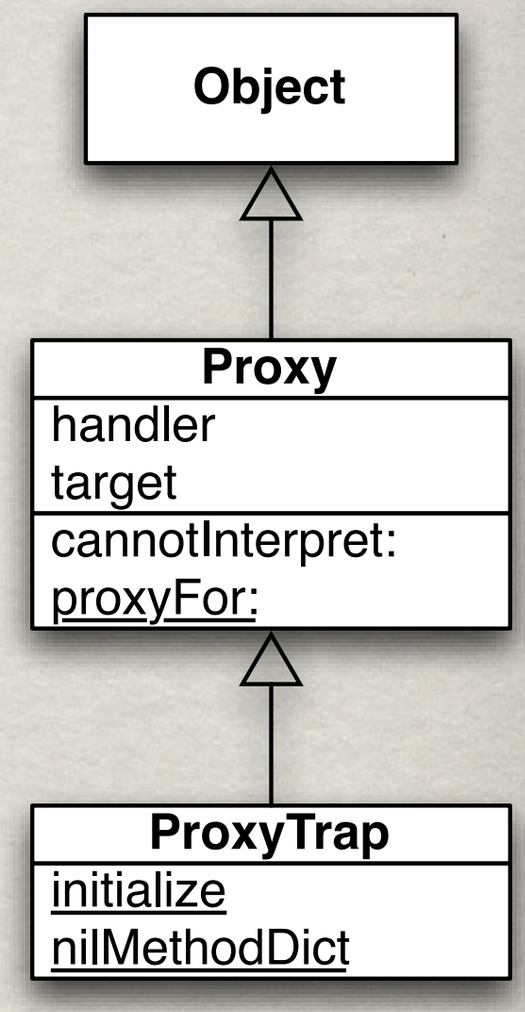


How it works?

```
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.
    ProxyTrap adoptInstance: aProxy.
    ^ aProxy.
```

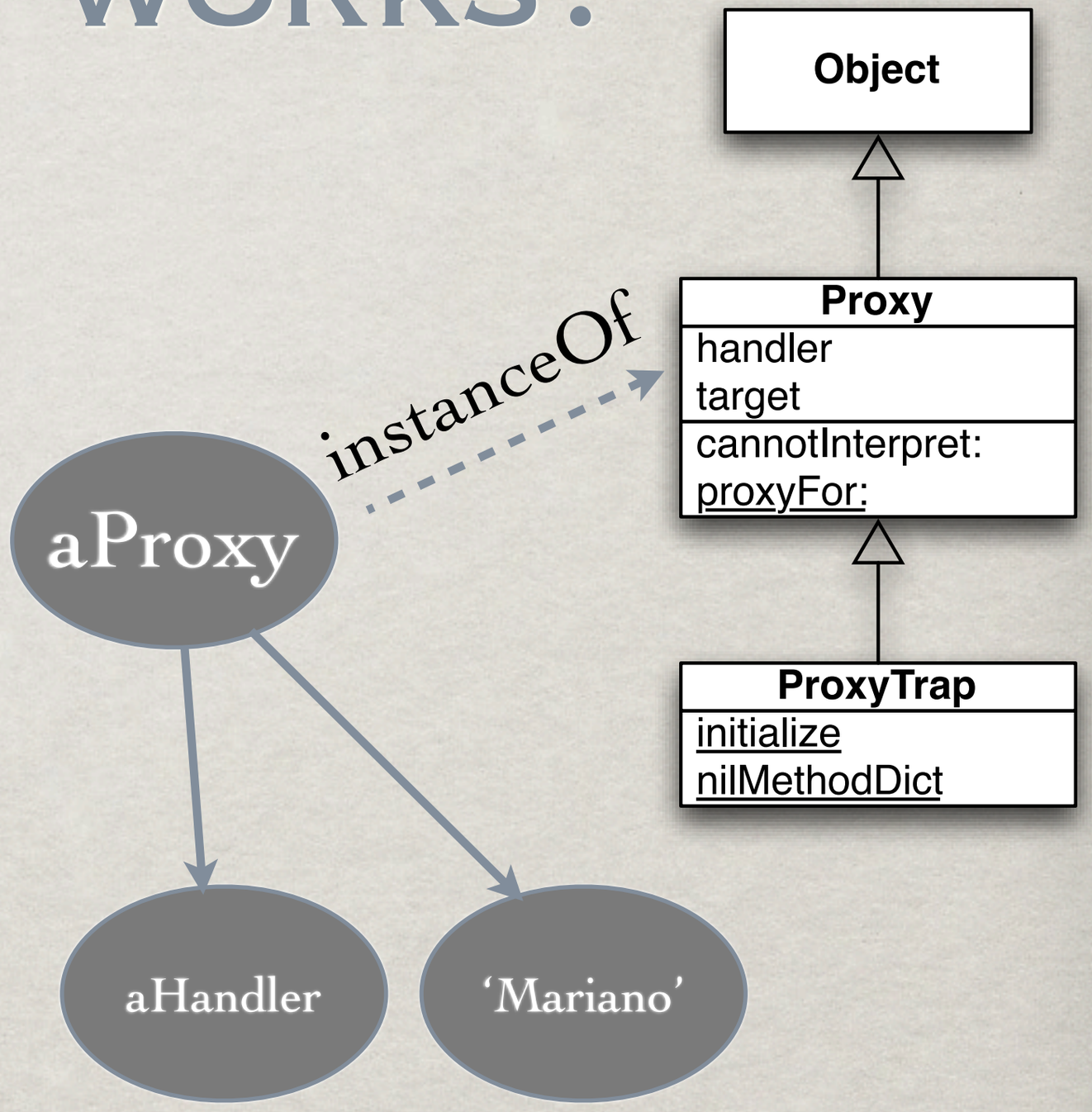


How it works?

```
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.
```

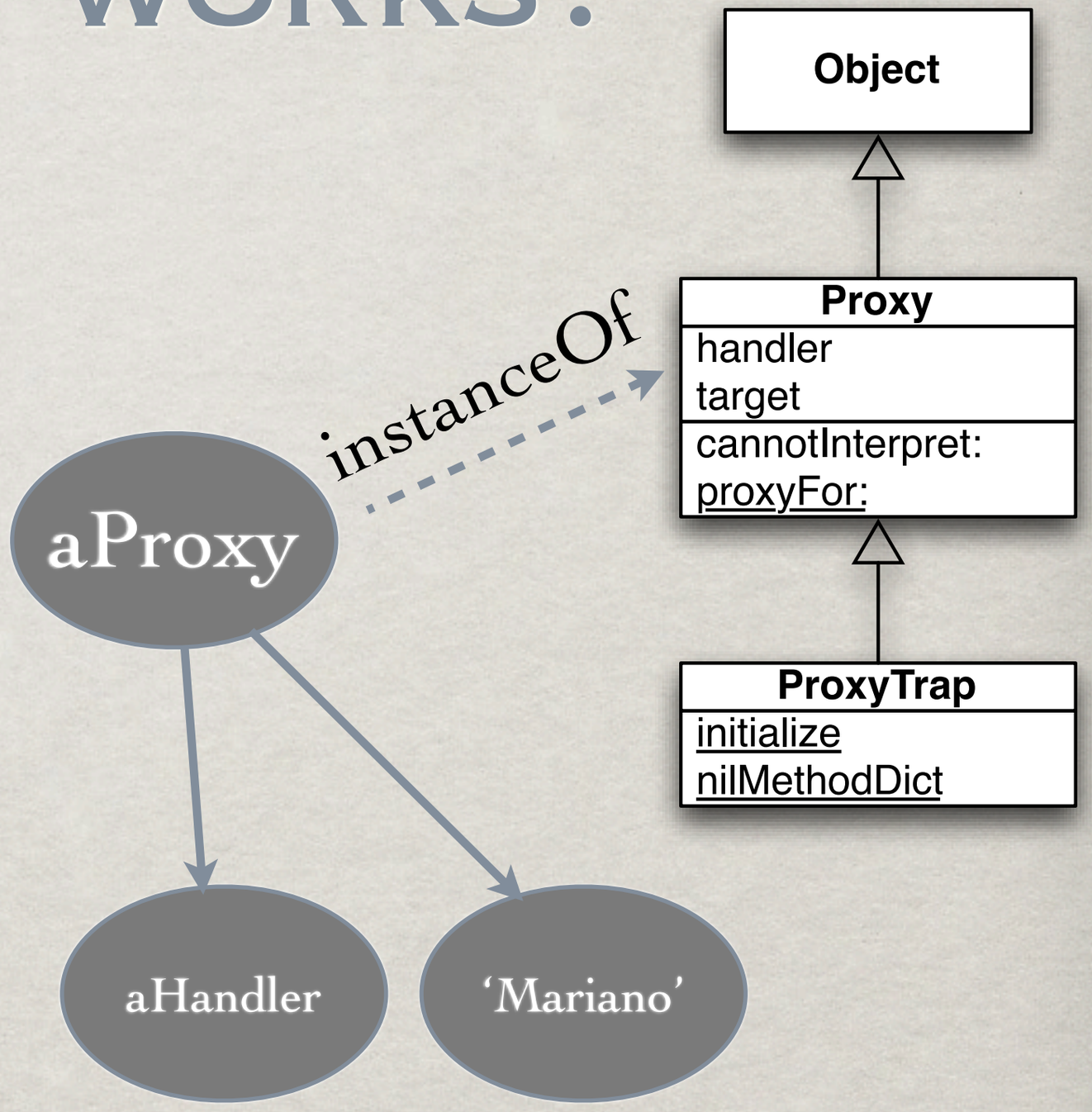


How it works?

```
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.
```



ProxyTrap class >>

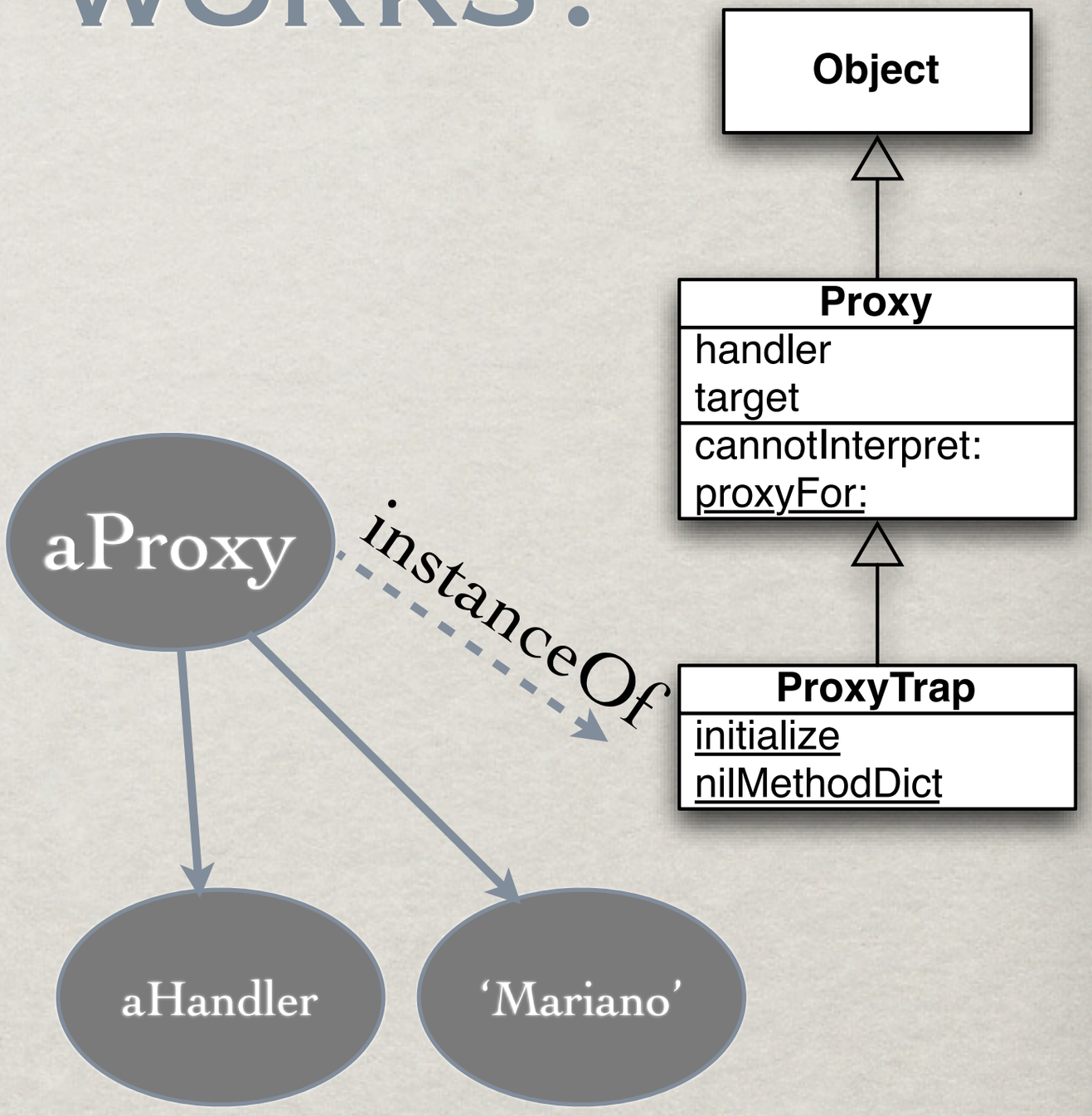
```
initialize
    superclass := Proxy.
    format := Proxy format.
    methodDict := nil.
```

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.
```



ProxyTrap class >>

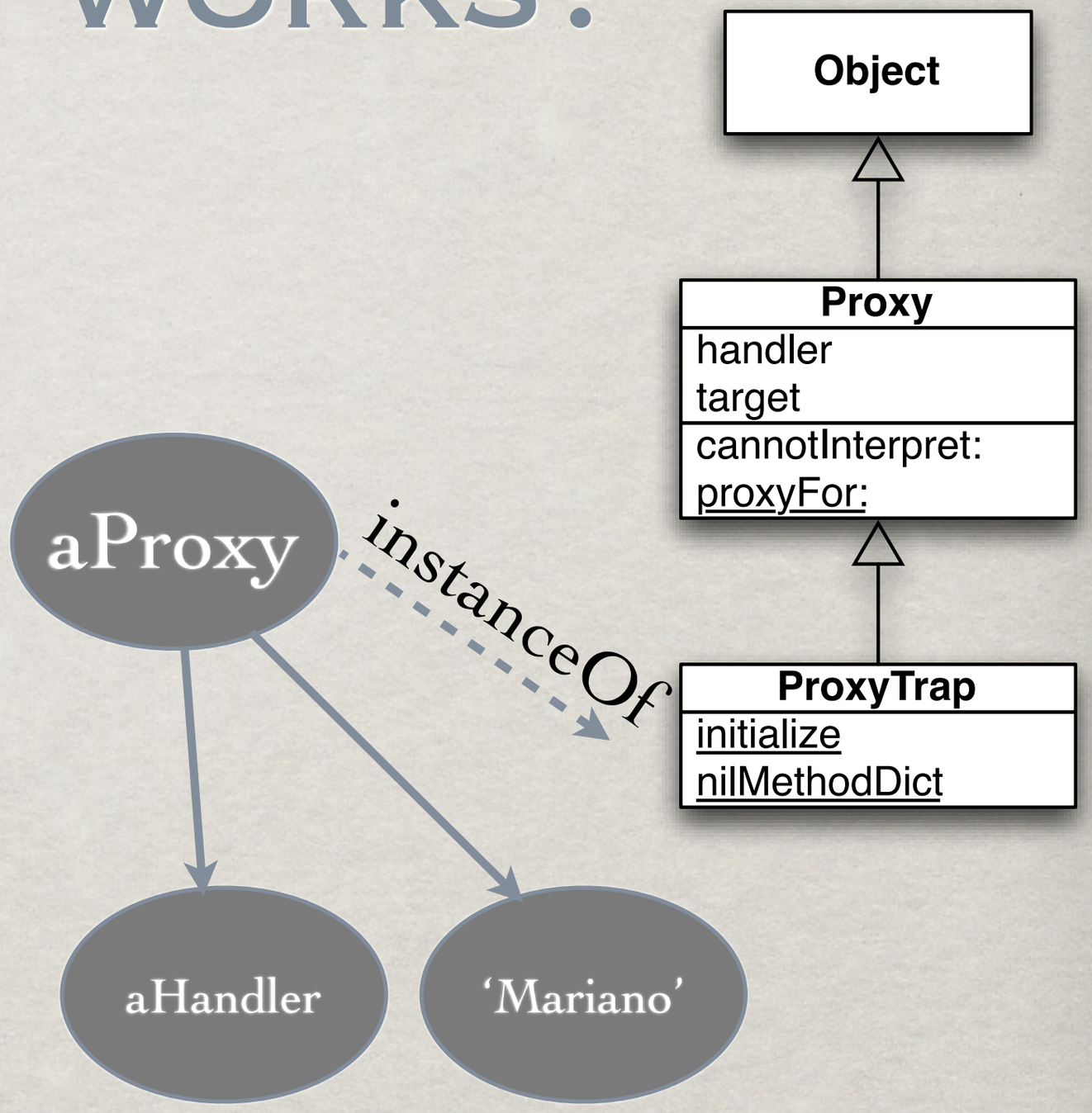
```
initialize
    superclass := Proxy.
    format := Proxy format.
    methodDict := nil.
```

How it works?

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

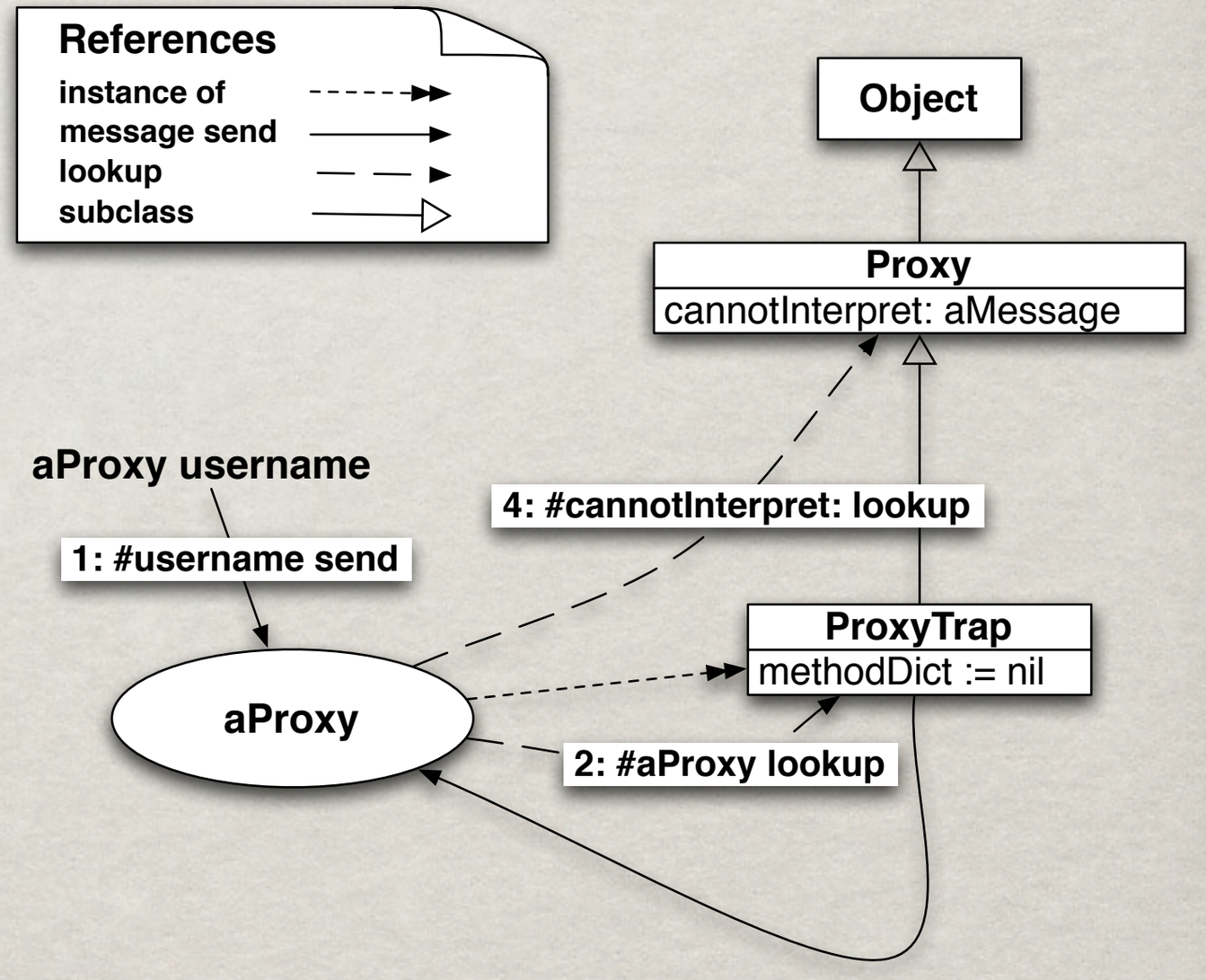
Proxy class >>

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



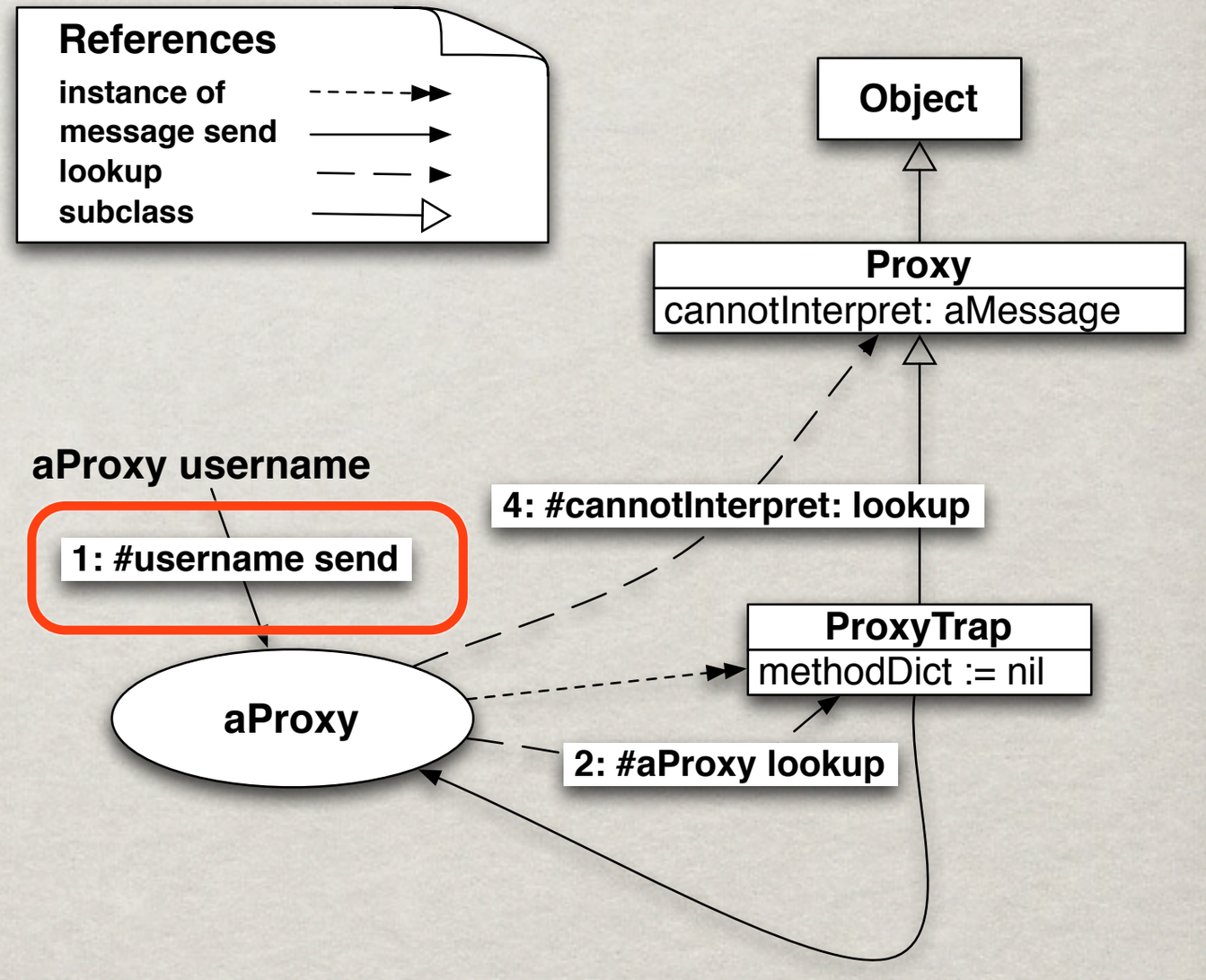
How it works?

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



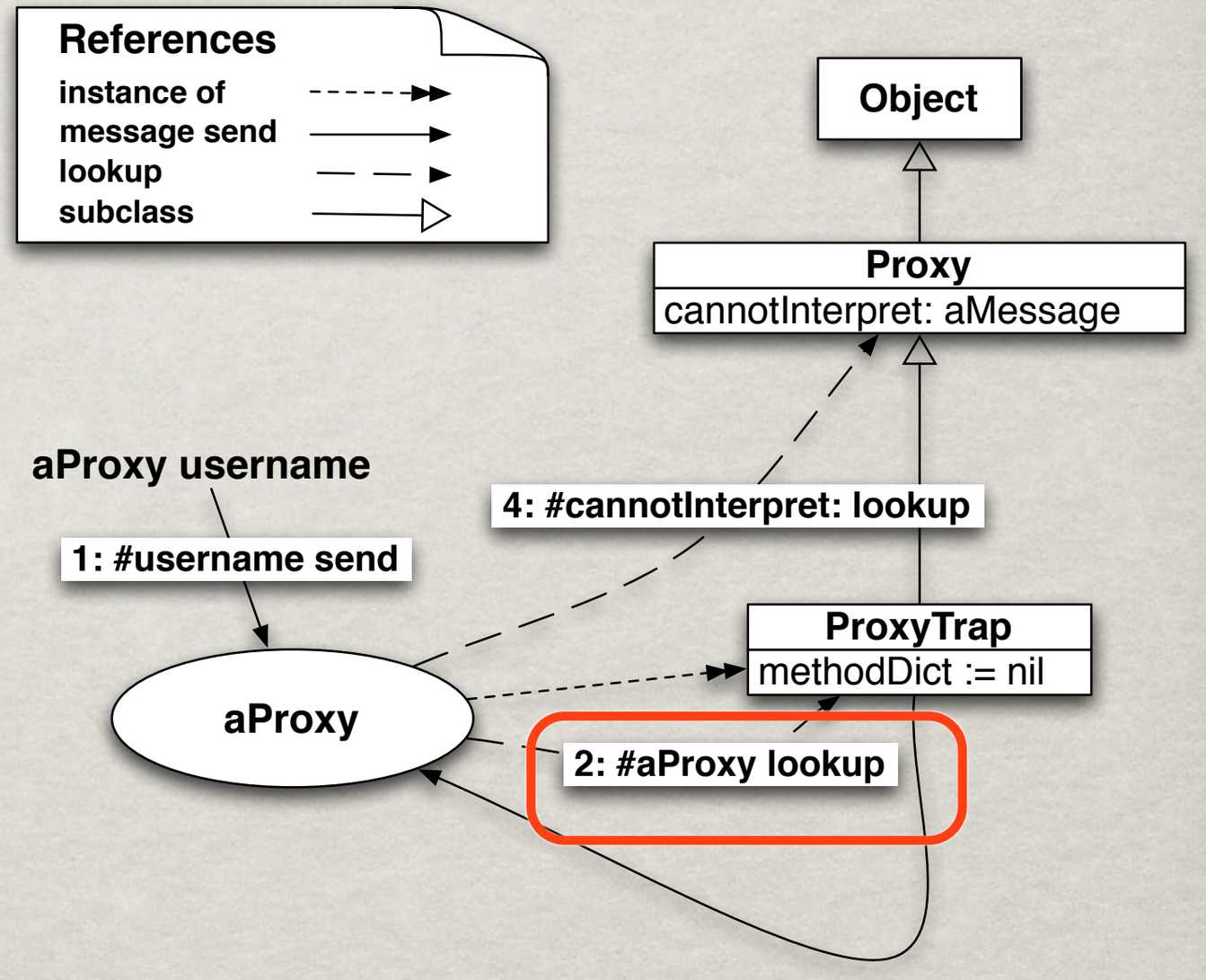
How it works?

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



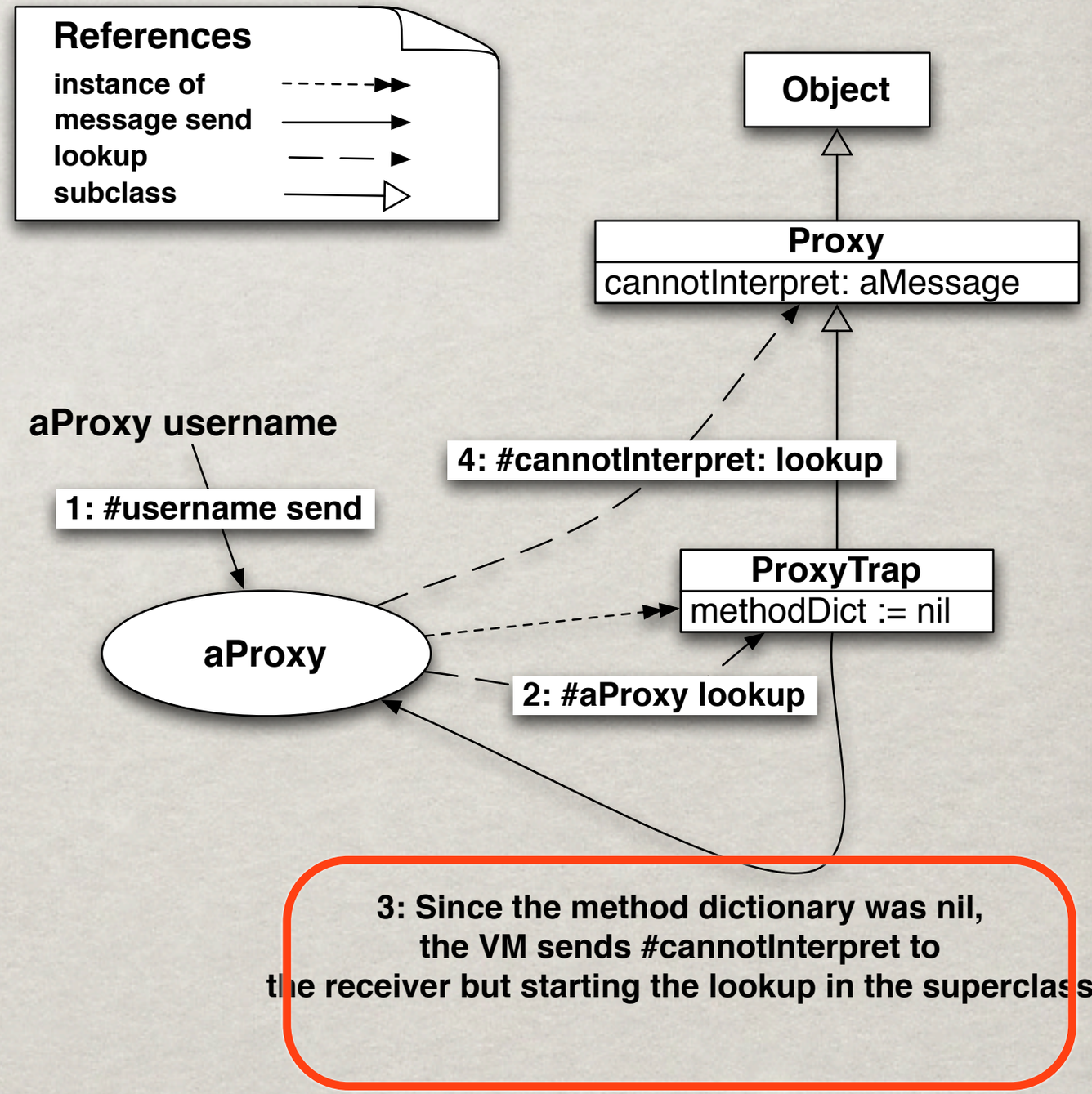
How it works?

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



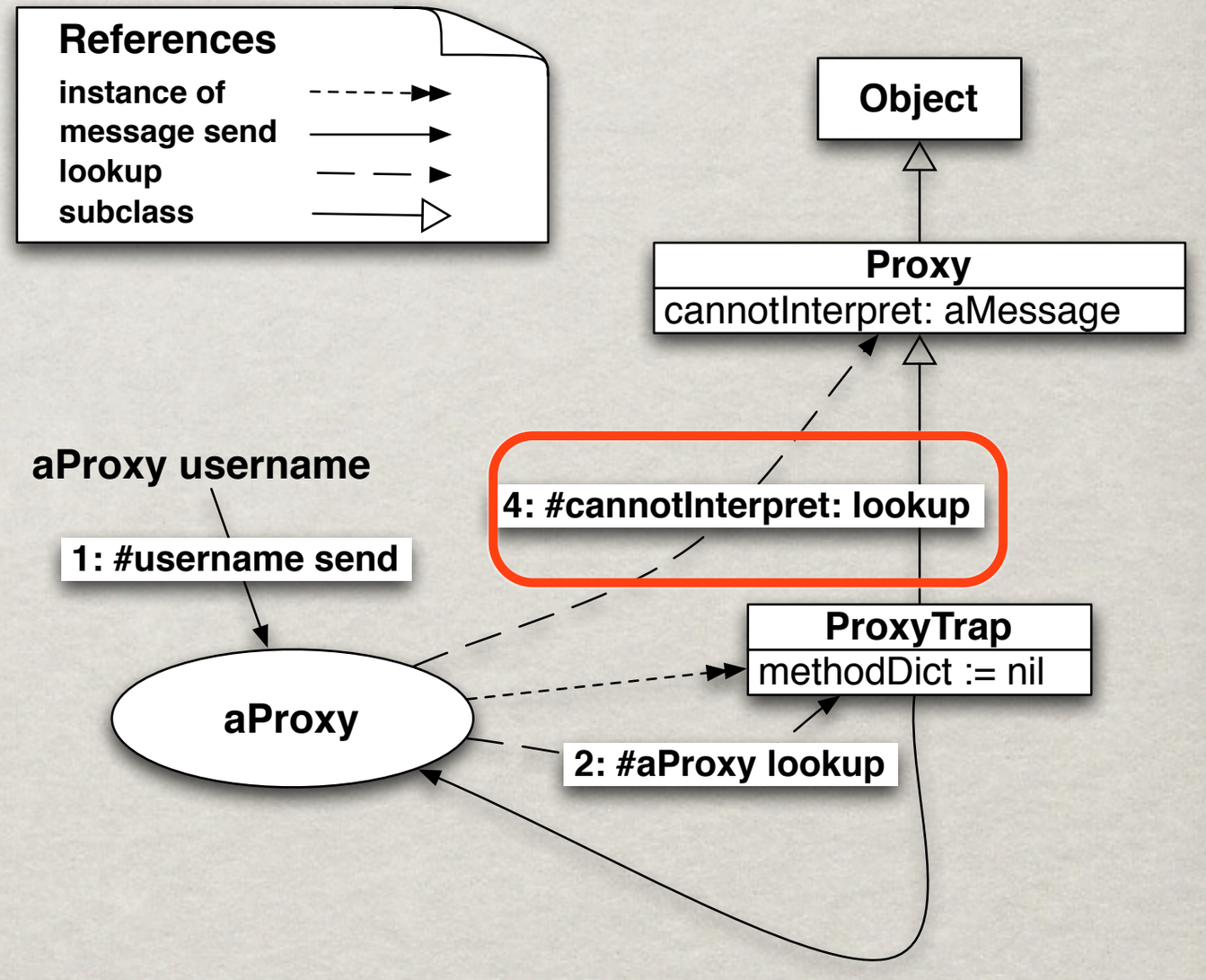
How it works?

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



How it works?

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

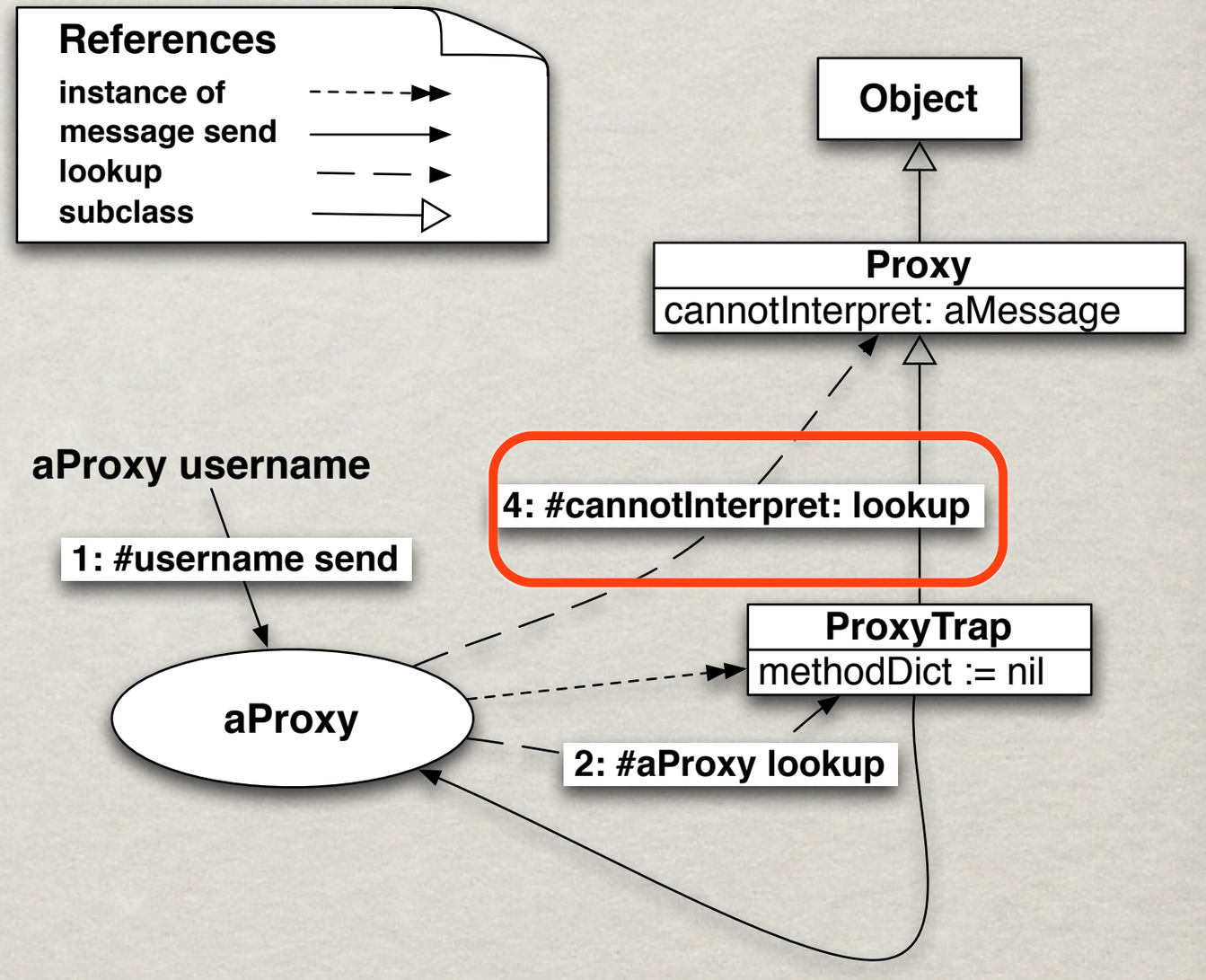


How it works?

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

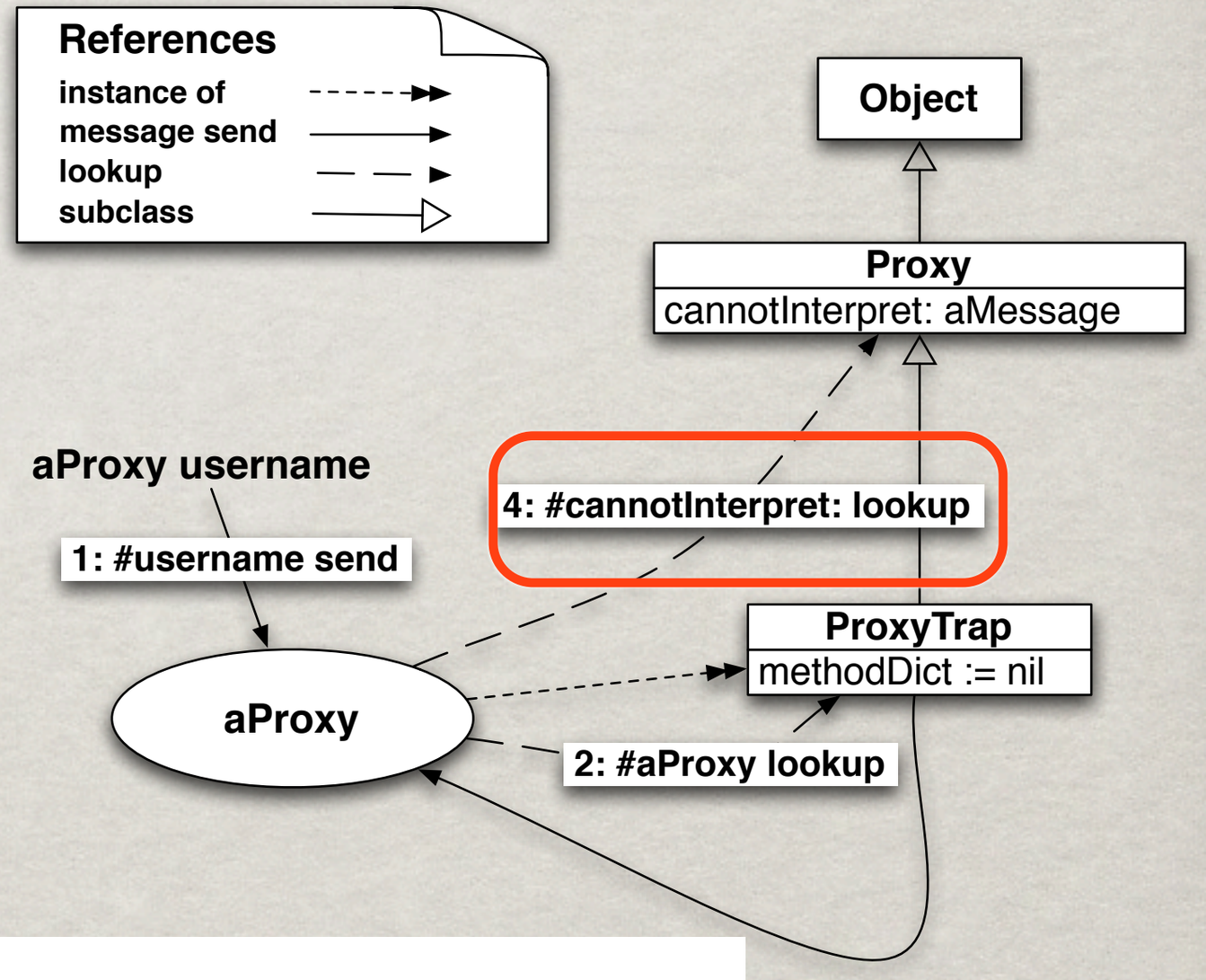
Proxy >>

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



How it works?

```
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
```

Traditional	Ghost
#doesNotUnderstand: cannot be trapped like a regular message.	#cannotInterpret: is trapped like a regular message.
Mix of handling procedure and proxy interception.	No mix of handling procedure and proxy interception.
Only methods that are not understood are intercepted.	"**All**" methods are intercepted.
No separation between proxies and handlers.	Clear separation between proxies and handlers.


Ghost
is stratified

Methods not intercepted

1) Optimizations done by the Compiler

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

2) Special shortcut bytecodes between Compiler and VM

Workspace

```
Smalltalk specialSelectors #(#+ 1 #- 1 #< 1 #> 1 #<= 1 #>= 1 #= 1 #~= 1 #* 1 #/ 1 #\\ 1 #@ 1
#bitShift: 1 #// 1 #bitAnd: 1 #bitOr: 1 #at: 1 #at:put: 2 #size 0 #next 0 #nextPut: 1 #atEnd 0 #==
1 #class 0 #blockCopy: 1 #value 0 #value: 1 #do: 1 #new 0 #new: 1 #x 0 #y 0)
```

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:
            and: or:
            whileFalse: whileTrue: whileFalse whileTrue
            to:do: to:by:do:
            caseOf: caseOf:otherwise:
            ifNil: ifNotNil:  ifNil:ifNotNil: ifNotNil:ifNil:).
```

2) Special shortcut bytecodes between Compiler and VM

Workspace

```
Smalltalk specialSelectors #(#+ 1 #- 1 #< 1 #> 1 #<= 1 #>= 1 #= 1 #~= 1 #* 1 #/ 1 #\\ 1 #@ 1
#bitShift: 1 #// 1 #bitAnd: 1 #bitOr: 1 #at: 1 #at:put: 2 #size 0 #next 0 #nextPut: 1 #atEnd 0 #==
1 #class 0 #blockCopy: 1 #value 0 #value: 1 #do: 1 #new 0 #new: 1 #x 0 #y 0)
```

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.

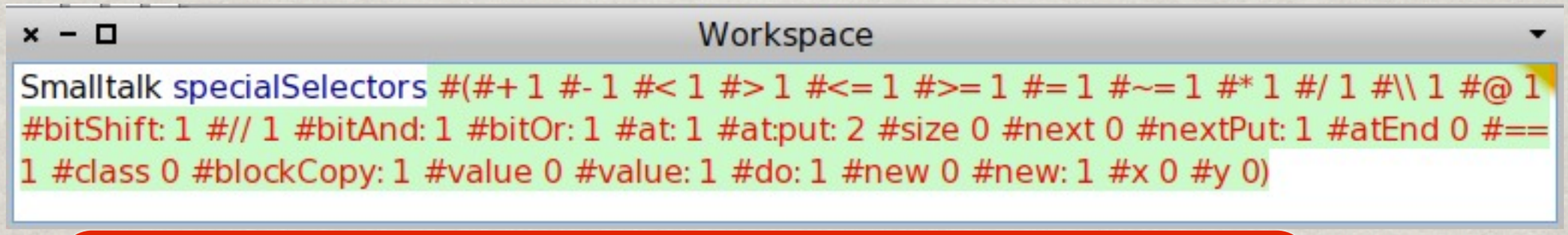
2.3) Always executed, they are just little optimizations. Examples #new, #next, #nextPut:, #size, etc.

Methods not intercepted

1) Optimizations done by the Compiler

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

2) Special shortcut bytecodes between Compiler and VM



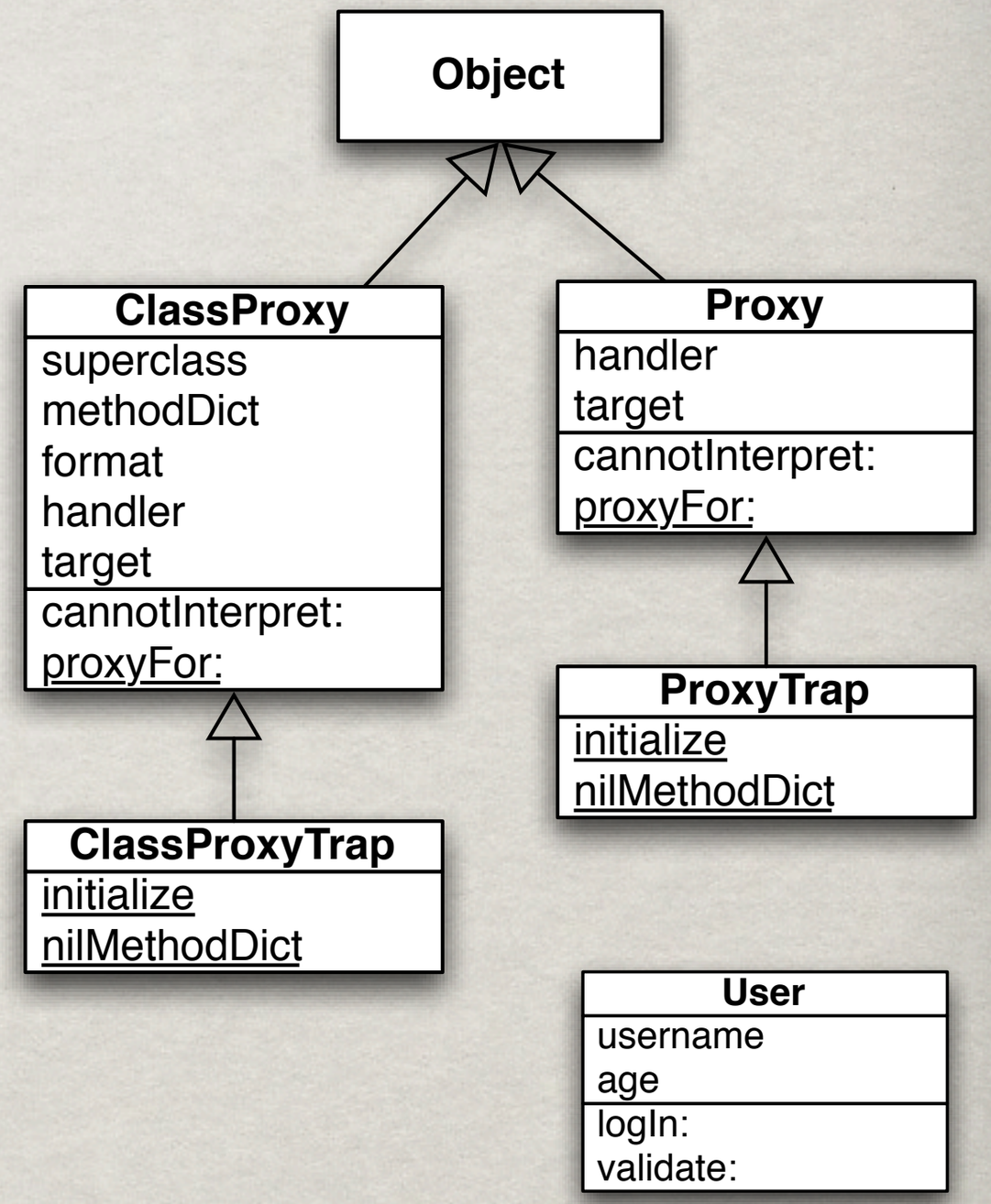
2.1) Methods NEVER sent: #== and #class

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.

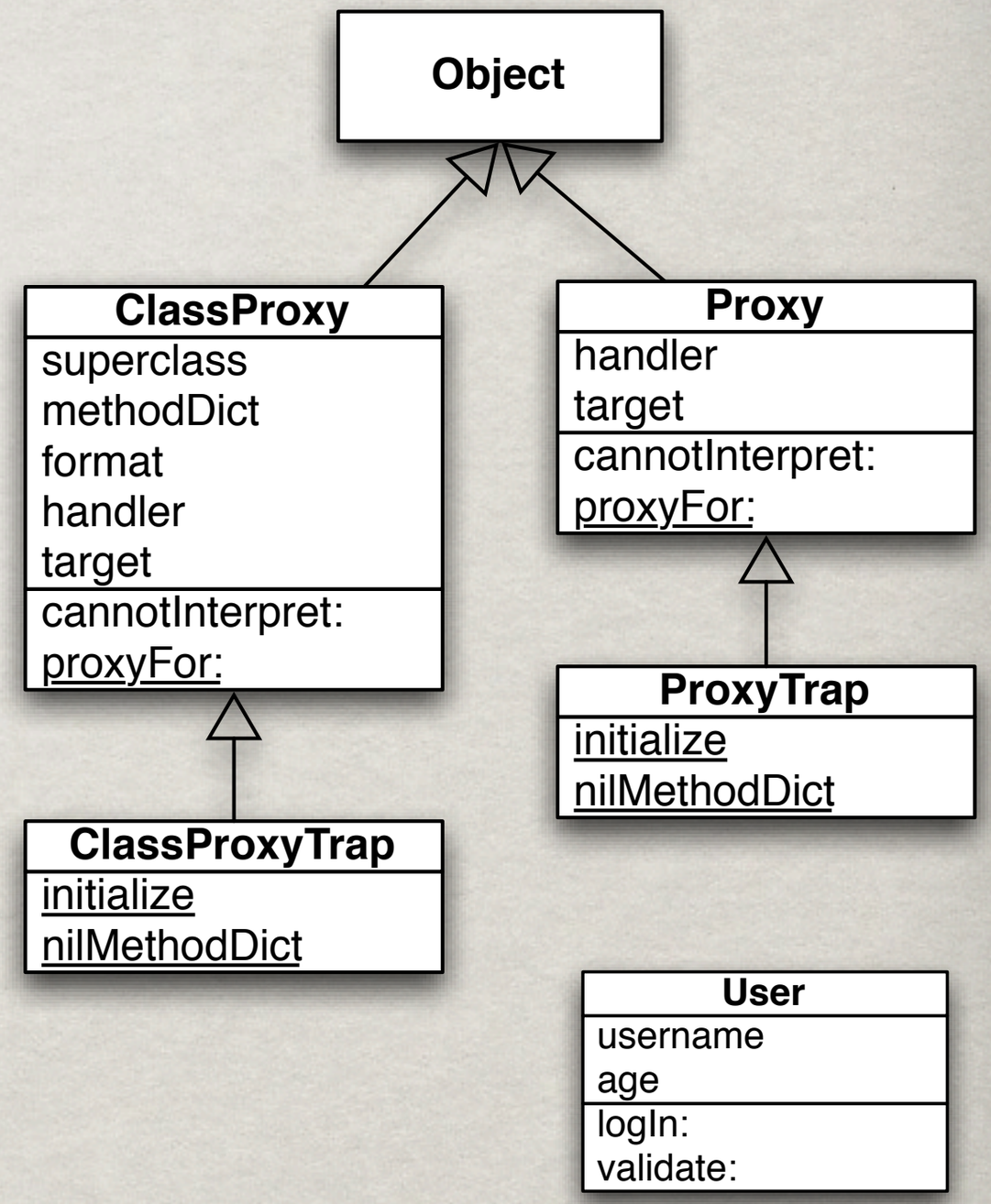
Proxy for classes

```
testProxyForClass
    | aProxy aUser |
    aUser := User named: 'Kurt'.
    aProxy := ClassProxy createProxyAndReplace: User.
    self assert: User name equals: #User.
    self assert: aUser username equals: 'Kurt'.
```



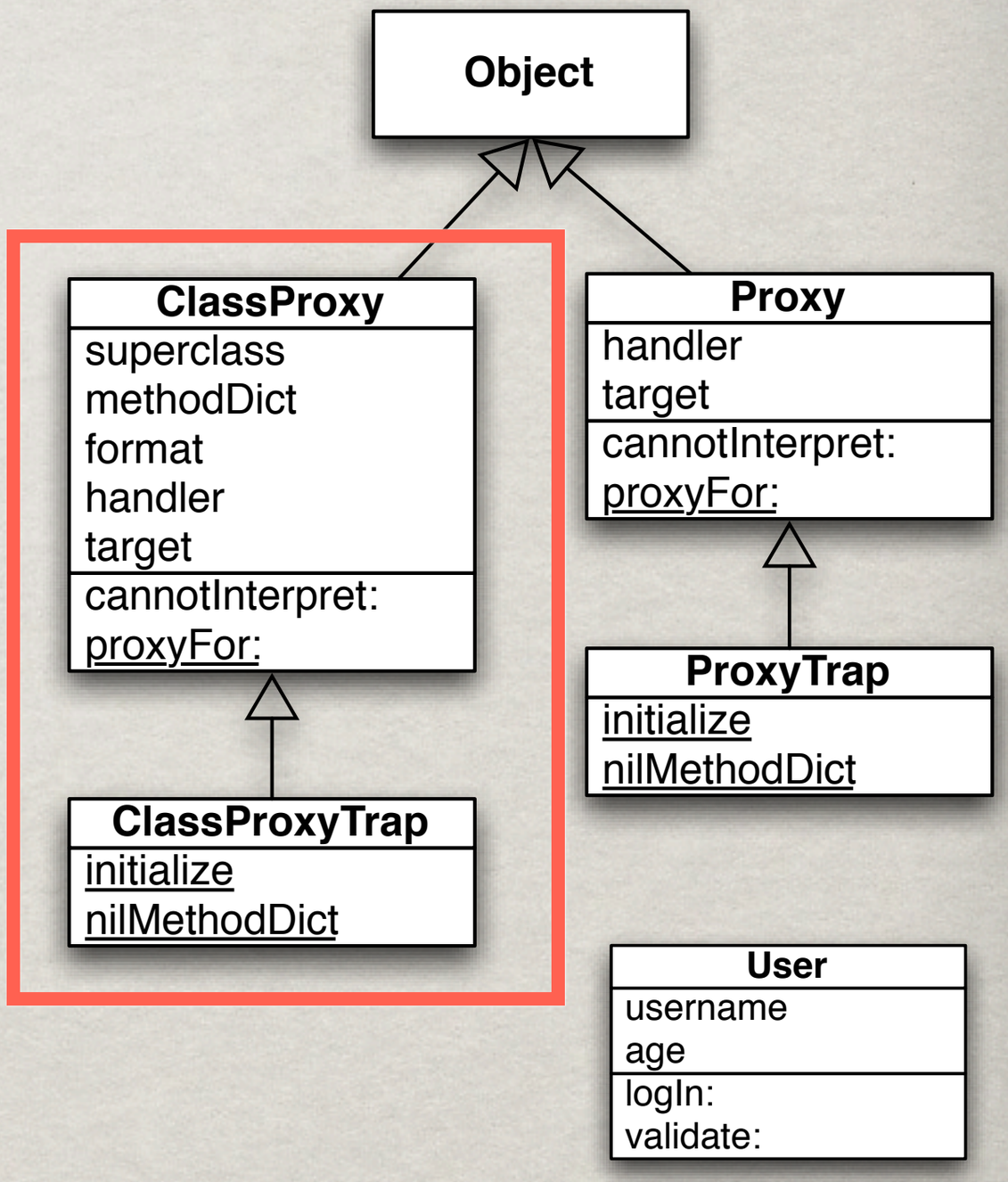
Proxy for classes

```
testProxyForClass
    | aProxy aUser |
    aUser := User named: 'Kurt'.
    aProxy := ClassProxy createProxyAndReplace: User.
    self assert: User name equals: #User.
    self assert: aUser username equals: 'Kurt'.
```



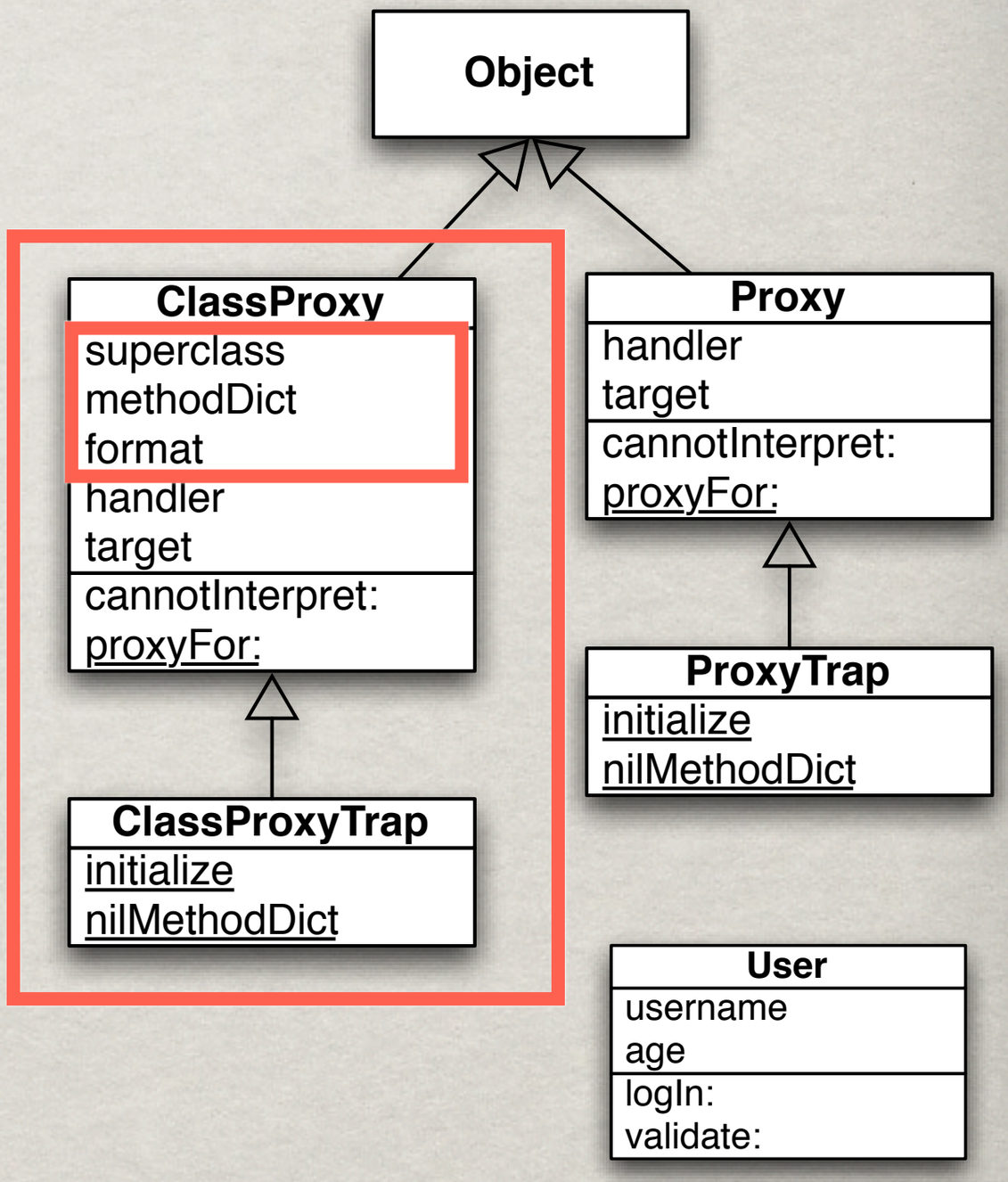
Proxy for classes

```
testProxyForClass
    | aProxy aUser |
    aUser := User named: 'Kurt'.
    aProxy := ClassProxy createProxyAndReplace: User.
    self assert: User name equals: #User.
    self assert: aUser username equals: 'Kurt'.
```



Proxy for classes

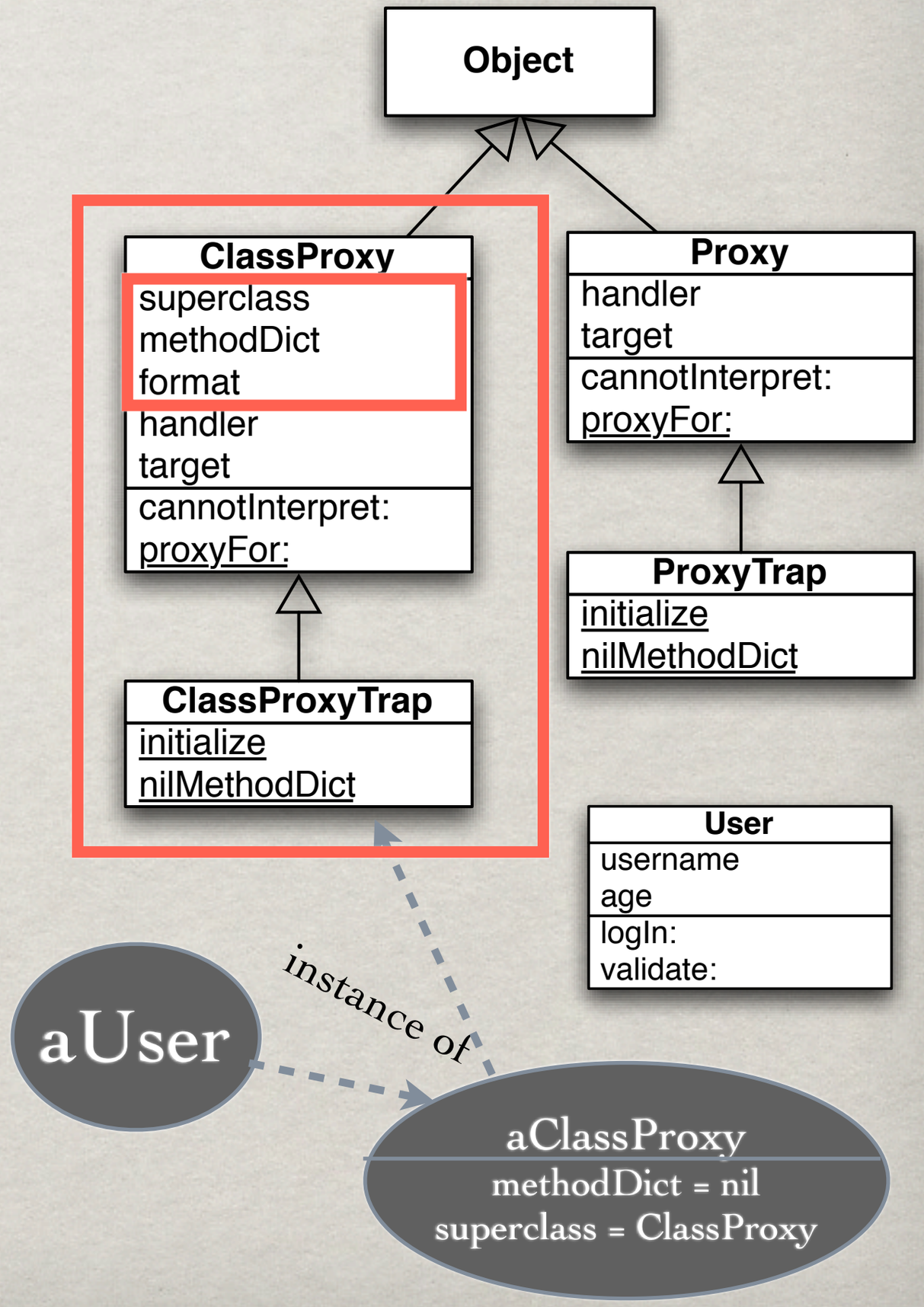
```
testProxyForClass
    | aProxy aUser |
    aUser := User named: 'Kurt'.
    aProxy := ClassProxy createProxyAndReplace: User.
    self assert: User name equals: #User.
    self assert: aUser username equals: 'Kurt'.
```



Proxy for classes

```
testProxyForClass
    | aProxy aUser |
    aUser := User named: 'Kurt'.
    aProxy := ClassProxy createProxyAndReplace: User.
    self assert: User name equals: #User.
    self assert: aUser username equals: 'Kurt'.
```

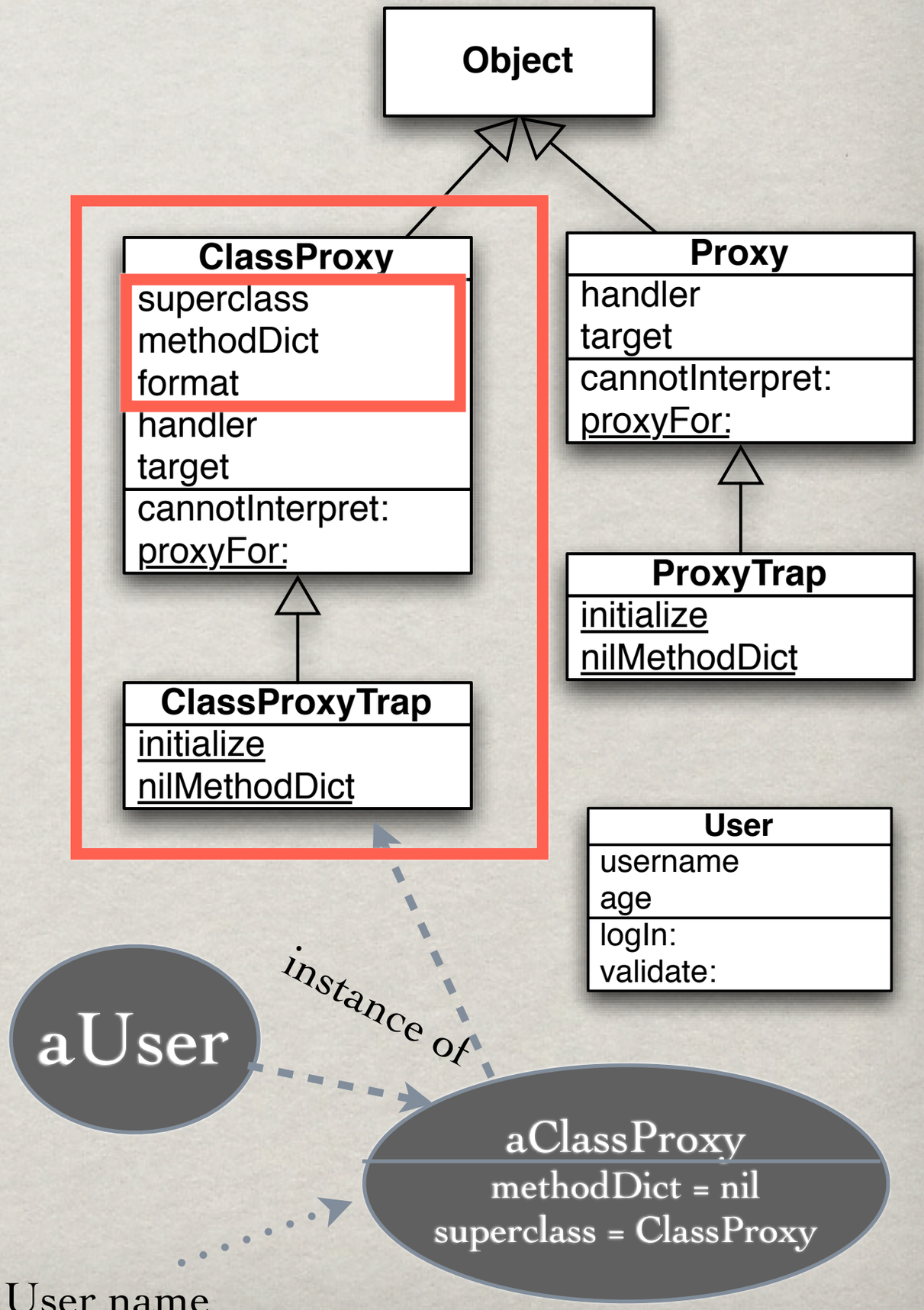
```
createProxyAndReplace: aClass
  | aProxy newProxyRef newClassRef|
    aProxy := self new
      initializeWith: SimpleForwarderHandler new
      methodDict: nil
      superclass: ClassProxy
      format: ClassProxy format.
   aProxy become: aClass.
    "After the become is done, aProxy now points 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.
```



Proxy for classes

```
testProxyForClass
    | aProxy aUser |
    aUser := User named: 'Kurt'.
    aProxy := ClassProxy createProxyAndReplace: User.
    self assert: User name equals: #User.
    self assert: aUser username equals: 'Kurt'.
```

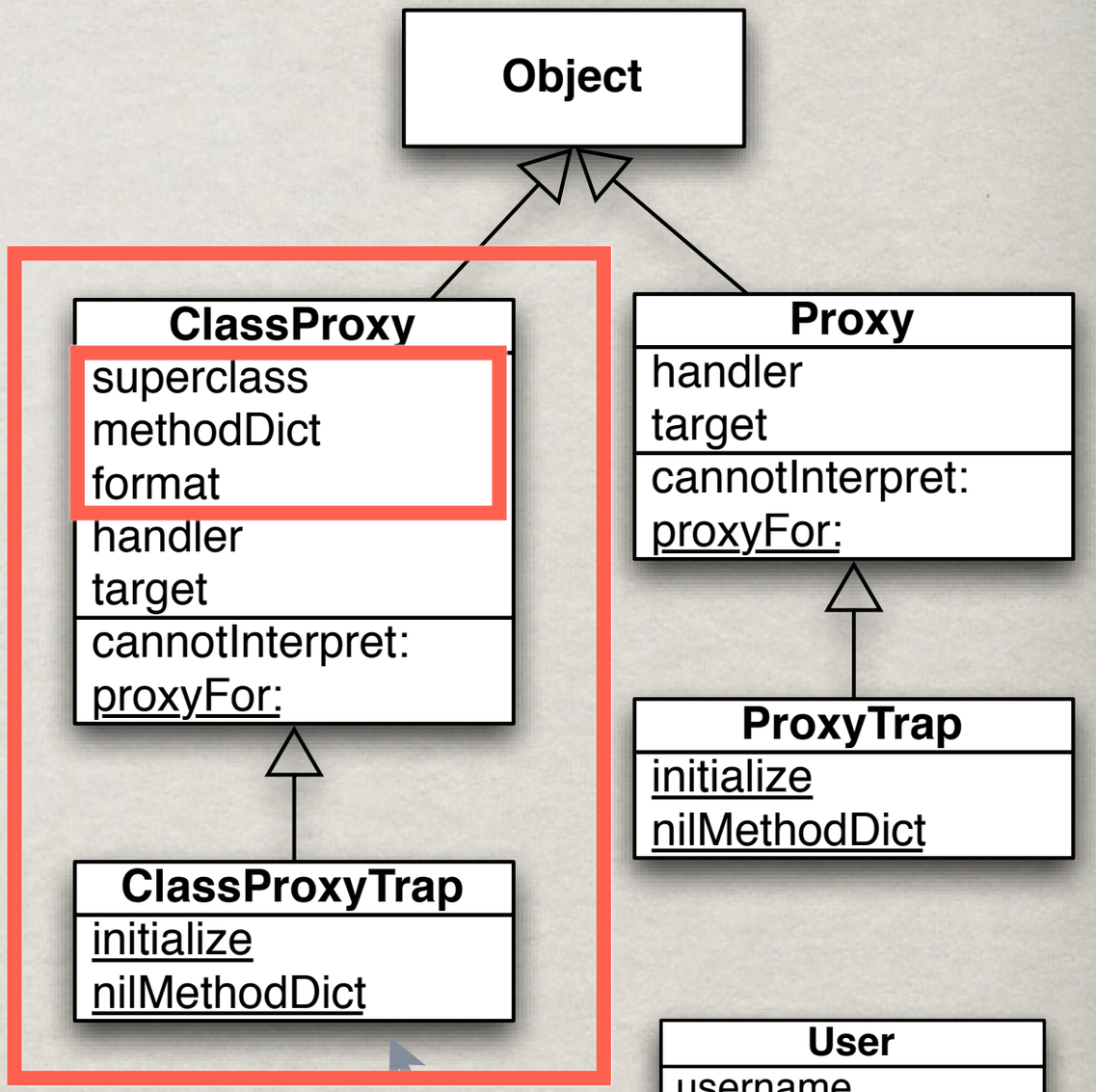
```
createProxyAndReplace: aClass
  | aProxy newProxyRef newClassRef|
    aProxy := self new
      initializeWith: SimpleForwarderHandler new
      methodDict: nil
      superclass: ClassProxy
      format: ClassProxy format.
    aProxy become: aClass.
    "After the become is done, aProxy now points 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.
```

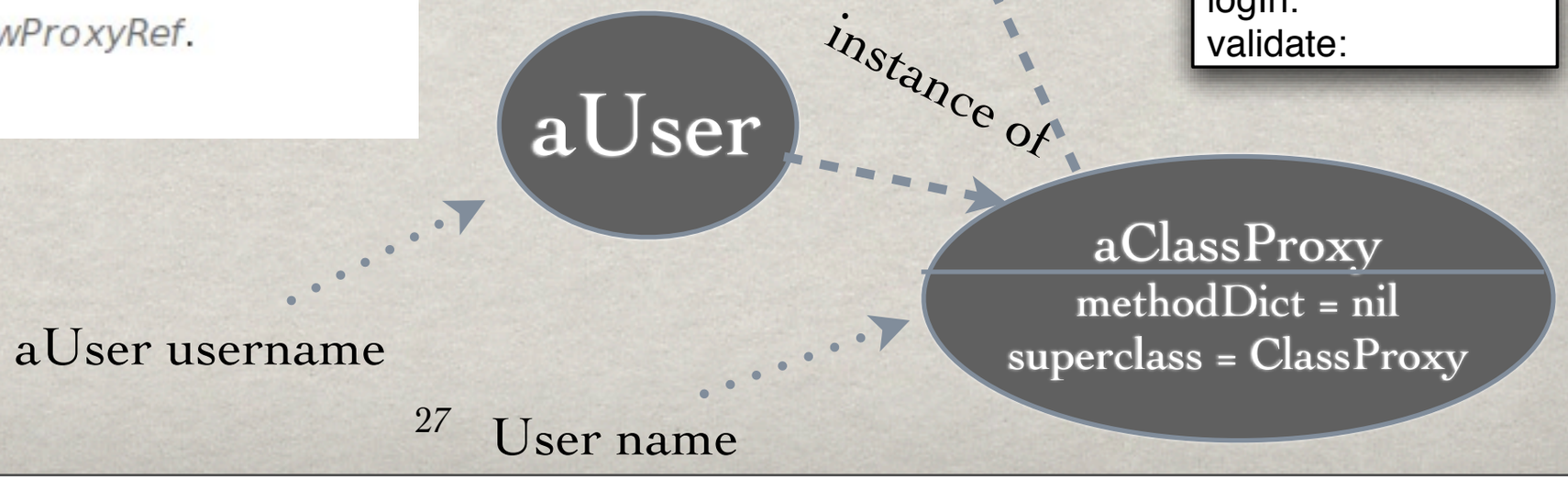


Proxy for classes

```
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.
```





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'.
```

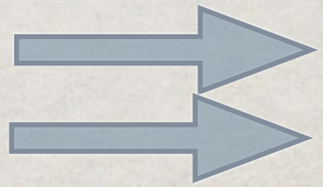
Proxy for methods

```
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'.
```

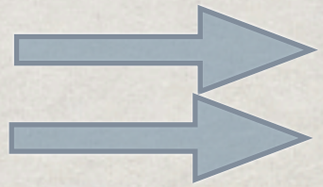


Regular message

Method execution

Proxy for methods

```
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

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

Ghost

is Uniform

Ghost

IS more UNIFORM

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://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/

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE | INRIA

RMod

Ecole d'Ingénieurs Centre de Recherche
Mines de Douai
LILLE EURORÉGION