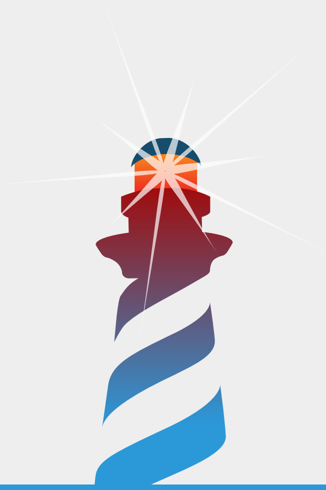

Advanced Reflection with Pharo

Steven Costiou and Damien Pollet

Square Bracket tutorials
April 18, 2020

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.
Cover art by Thomas Manchon.

Contents

Illustrations

Structure of the book

The first chapter introduces the contents of the book, *i.e.* an overview of object-centric instrumentation techniques available in Pharo 7. The second chapter is a summary of the evaluation results of the presented techniques. A reader may directly read this chapter if he is already familiar with the Pharo techniques presented in the book. Chapters 3 to 7 describe five solutions for object-centric instrumentation, and provide an evaluation of these solutions. Chapter 8 drafts the premises of an object-centric debugger based on the studied techniques and concludes the book.

> **Note** Each chapter illustrates an object-centric instrumentation technique based on an example depicted in Chapter 1. Having this example in mind is a prerequisite to the reading of those chapters.

Acknowledgements

People helping adding, rewriting or proof-reading materials will be thanked here.

CHAPTER 1
Introduction

This booklet enumerates and illustrates techniques for object-centric instrumentation in Pharo. An instrumentation is object-centric if it applies to one specific object (or a set of objects), without consideration of its class. It means the instrumentation can be applied on one object, leaving untouched all other instances of its class, or to an heterogeneous set of instances of different classes. This booklet gives an overview of available object-centric instrumentation techniques in Pharo, either present in the standard distribution or available on download. We only focus on object-centric state-access instrumentation, *i.e.* additional behavior triggered when an instance variable is read or written.

We will not go into deep technical usage description, nor into implementation details. Each chapter illustrates one solution, through the an example of object-centric state-access instrumentation depicted in Chapter 1. We give additional pointers to go deeper in the study of the solution.

We study five object-centric instrumentation solutions, namely *Anonymous Subclasses*, *Talents*, *Ghost*, *MetaLink* and *Read Only Objects*. To that end, we follow a three-fold evaluation. First, the studied technique is applied on a simple example of object-centric instrumentation. This example, depicted in this chapter, consists to the interception of the writing of a new value into an instance variable, and to capture of that value into another instance variable. Second, the technique is evaluated against a set of desirable properties. Finally, performance overhead is evaluated. We only evaluate raw solutions in the scope of the provided example, without considering the possibility of enhancing the technique by building something on top or with optimisations.

This chapter presents the three-fold evaluation that is applied to each studied technique, based on the current stable version of Pharo 7. Each time, a new Pharo image is created, the evaluation code is loaded as well as the studied solution's packages if needed. Then the evaluation is performed.

The evaluation code and example presented in this chapter is available on Github, along with unit tests illustrating the application of each technique. The whole code can be loaded into Pharo by executing the current code snippet:

```
Metacello new
    baseline: 'ObjectCentricEvaluationExamples';
    repository:
    'github://StevenCostiou/PharoObjectCentricEvaluationExamples';
    load.
```

1.1 Illustration example

Each studied solution is experimented on an example of object-centric behavior instrumentation. We use a class `Person` defined in the following script. This class has two instance variables: `name` and `tag`. It has two accessors: `=name:` and `tag:`.

```
Person>>name: aName
    name := aName

Person>>tag: aTag
    tag := aTag
```

We would like that each time a value is stored in that instance variable, that value is printed on the `Transcript`.

The instrumentation is defined as follows: each time a value is stored into the `name` instance variable, that value is captured and stored into the `tag` instance variable. We do not consider the moment of the capture, which can be before or after the *physical* write into `name`. As instances of `Person` only have one method accessing the `name` variable, we need to instrument that method and to scope the instrumentation of a specific object.

We evaluate this scenario for each studied technique through unit tests. In the following script, `p1` and `p2` are two different instances of the `Person` class. For each solution, the instrumentation is applied to the `p2` instance only, then the test from the script is executed. This test sends the `#name:` message to both instances. The `tag` instance variable must be `nil` for `p1` (*i.e.* no instrumentation). For `p2`, the instrumented instance, the value that was written in its `name` instance variable must have been captured during state access and stored into the `tag` variable.

```
ObjectCentricInstrumentationTest>>assertObjectCentricInstrumentation
  p1 name: 'Worf'.
  p2 name: 'Dax'.
  self assert: p1 tag isNil.
  self assert: p2 tag equals: 'Dax'
```

1.2 Evaluation criteria

Each solution is evaluated against the following desirable properties.

Manipulated entity. Classes, traits, objects, methods... The entity that is manipulated to express and install the instrumentation.

Reusability. Evaluates if the same instrumentation unit can be reused to instrument different objects with the same behavior.

Flexibility. We consider the solution to be flexible if it does not put any constraints on the program prior to the instrumentation, and if it does not requires the developer to perform annoying repetitive and error-prone tasks. For example, it must not requires to copy existing code, to use a specific coding-style, or if it cannot apply in certain cases.

Granularity. The level of at which behavior can be instrumented. We consider three granularities: method, sub-method and state access. The method granularity can only instrument a whole method, by controlling message passing. The sub-method level can instrument specific elements of a method. The state-access granularity enables the interception of direct instance variable access.

Integration. An instrumentation must not break system tools and features nor user libraries and programs.

`Self` problem. Object-specific instrumentation should be applicable to messages sent to the `self` pseudo-variable, *i.e.* the original receiver.

`Super` problem. Object-specific instrumentation should be applicable to messages sent to the `super` pseudo-variable, *i.e.* the original receiver. When those messages are not subject to instrumentation, their lookup must resolve in the super class of the original object's class.

1.3 Performance overhead evaluation

To provide a approximation of the performance overhead due to instrumentation, we compare the execution time of a block of code using an instrumented object with a reference execution time of a block of code using a non-instrumented object. The method `evaluateOverheadFor:` from the following script shows how execution time is computed. The parameter is an instance of `Person`. The `#name:` message is sent 100000000 times to the `Person` instance, and the overall execution time is returned. This returned time is used to compare execution time of an instrumented instance against the execution time of a non-instrumented instance.

```
ObjectCentricOverheadEvaluation>>evaluateOverheadFor: aPerson
  ^[ 100000000 timesRepeat: [ aPerson name: 'eval' ] ] timeToRun
```

The non-instrumented instance is an instance of a modified `Person` class in which the instrumentation is hardcoded. It is a means to obtain a reference execution time with instrumentation applied as if it were originally part of the program. Instrumented instances are instances of `Person` for which one of the studied solutions is used to apply the same instrumentation as the hardcoded one for the reference execution time. Performance overhead is evaluated for each solution following the aforementioned protocol.

Measurements are made using the following software and hardware:

- Pharo 7, macOS High Sierra
- MacBook Pro 2017, 2,2 GHz Intel Core i5 (2 cores), 16 GB 2133 MHz LPDDR3

CHAPTER 2

Summary of the overall evaluations

If you already know Pharo and (some of) the presented technique, this chapter is a global summary of the overall evaluation. It contains spoilers, obviously. Results are not commented nor detailed. The performance overhead evaluation, described in Chapter 1, speaks for itself, while the property evaluation is detailed in each chapter.

2.1 Performance overhead evaluation

The following table reports execution times and a time factor for each solution against the reference code (see Chapter 1).

Solution	Execution time (ms)	Time Factor
Reference execution	935	x1
Anonymous Classes	968	x1.035
MetaLink	984	x1.052
Talents	958	x1.025
Ghost	41500	x44.38
Change Detector	532496	x570

The two following figures visually illustrate the overhead. For the sake of readability, execution times with too much overhead have been separated from the evaluations with little overhead.

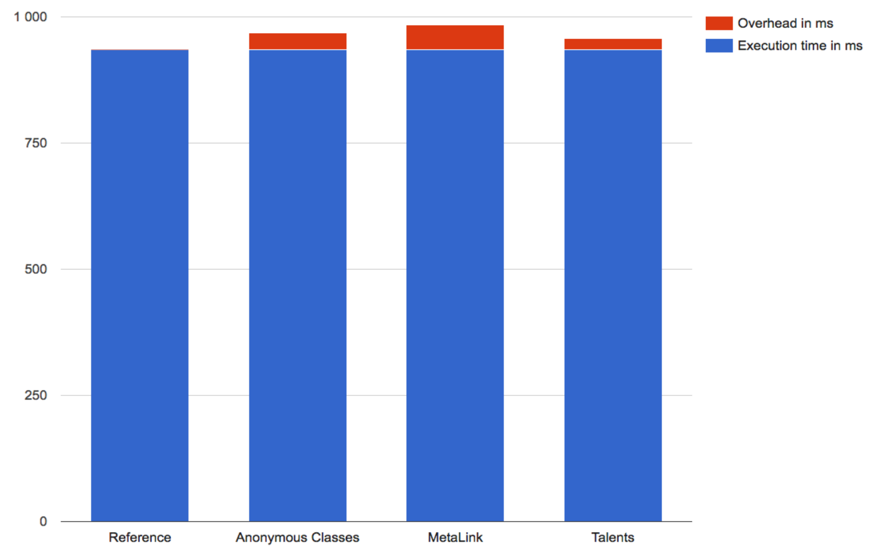

Figure 2-1 Comparison of execution times for Anonymous Classes, MetaLink and Talents

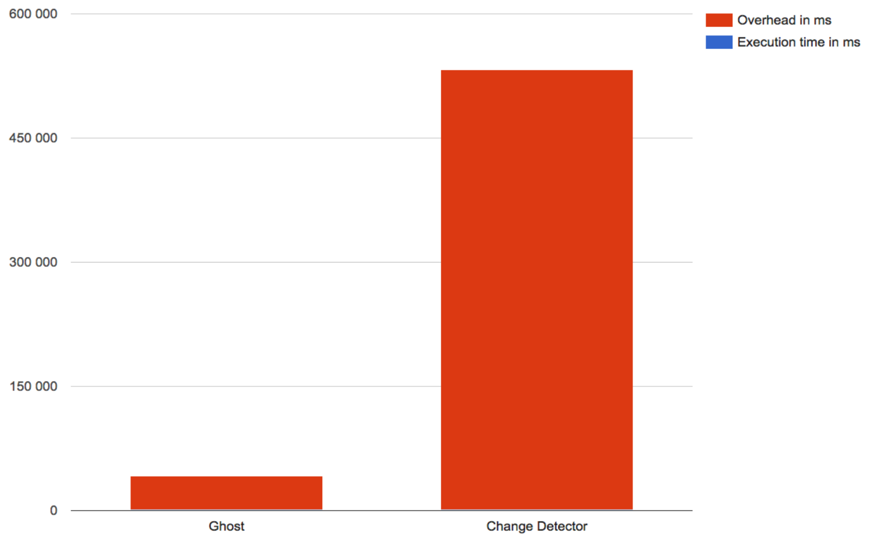

Figure 2-2 Comparison of execution times for Ghost and Change Detector

Note For performance overhead evaluation, we only consider object-centric instrumentation during the writing of an instance variable. Almost all techniques can be used for more general object-centric instrumentation, such as message interception, behavior replacement or behavioral variation. For instance, metalinks can be conditioned, and reify many information from the execution context. In that case the performance overhead may be more significant. The evaluation, for each solution, of general object-centric instrumentation execution cost, is not covered in this book.

2.2 Property evaluation overview

Each solution presented in this bookis evaluated against the following desirable properties for advanced reflection.

Granularity. It is the level at which a reflective operation is applicable: method, sub-method and slot. At the method granularity, we instrument a whole method by controlling message passing. At the sub-method level, we instrument specific elements of a method (*e.g., a specific message send). The slot granularity enables the interception of direct variable accesses.

Object-centric. A reflective operation is object-centric if it is applicable to only one specific object (or a set of selected objects). It means that for a given class, only a specific target instance is affected by reflection. All other instances of that class are not affected.

Scope. We consider two possible scopes for reflective operations: local and global. Local reflection only touches very restricted and selected part of the running code. Global reflection affects large parts or all of the running program.

Manipulated abstraction. It is the entity developers and tools manipulate to express, install and control reflective operations. For example: classes, traits, objects, methods, etc. Some solutions bring their own abstraction to deal with reflection.

The following table reports our evaluation of the presented solutions against the properties described above. In addition, each chapter in the rest of this book provides a detailed analysis for the solution it presents.

!Property	!Granularity	!Object-centric	!Scope	!Abstraction
Anonymous classes	methods	yes	local	class
Method Wrappers	methods	no	global	?
Reflectivity	sub-method	yes	local	Metalink
Talents	methods	yes	local	Trait
Proxies	methods	yes	local	objects
Change Detector	slot	no	local	slot
AOP	sub-method?	no	global	aspects

Handles? Mirrors? Membranes? (not sure if all those are reflection per se, I need to check more in details).

CHAPTER 3

Method Wrappers

A method wrapper inserts behavior before, instead or after a target method [BFJR98, Duc99]. A wrapper replaces the original target method. Each received message that should result in the invocation of the original method invokes the wrapper method instead.

3.1 Definition

The wrapper is a method object. It contains a reference to the original method, that is called the wrapped method. It defines a beforeMethod method and an afterMethod method. It may define an insteadMethod method. When invoked, the wrapper first executes beforeMethod. Then it executes the original method or, if defined, replaces it by insteadMethod. Finally, the wrappers executes the afterMethod.

Message passing is fully controlled:

- Every invocation of the wrapped method provokes the invocation of the wrapper,
- the returned value is either the one that is returned by the execution of the wrapped method, or is controlled by the afterMethod behavior.

Wrappers are installed in the classes where the methods they wrap are defined. When wrapped, a method M defined in a class C is replaced by a wrapper method W(M) in the method dictionary Cd of C.

In the original method dictionary Cd of class C, the original method M is associated to its selector

Cd: S -> M

When the method `M` is wrapped by a wrapper `W`, the original method `M` is replaced in the class dictionary by the wrapper method `W(M)`, that becomes associated with the selector `S`:

`Cd: S -> W(M)`

When an instance of `C` receives a message with selector `S`, the lookup resolves to the wrapper method `W(M)` in the method dictionary `Cd`. It is that wrapper method that is then executed.

3.2 Example

3.3 Evaluation

3.4 Other documentation

CHAPTER 4

Anonymous subclasses

Anonymous classes are nameless classes that are not registered in the system. Like standard classes, an anonymous class inherits from a parent class and specializes some of its parent structure and behavior. Anonymous classes are a convenient and powerful means for object-centric reflection, for example to control message passing for an individual object.

To provide us with object-centric reflection, an anonymous subclass is dynamically inserted between a target object and its original class [FJ89, HJJ93]. The target object is then migrated to that anonymous class. Object-centric reflective operations are implemented and applied by adding or redefining methods and structure in the anonymous class, that is now the active target object's class.

The granularity of reflective operations is the method. A typical example is to redefine a method from the target object's class in the anonymous subclass. In this redefined method, we control how the object responds to the corresponding message. The behavior of methods that are not redefined in the anonymous class is preserved, and message passing is not controlled in that case.

From a structural point of view, it is not transparent to the system. First, tools and users' code needing to access objects classes will be affected. For instance, if an object `o` instance of a class `C` is migrated to an anonymous class, then all users of that object requesting its class will now see the object as an instance of `anAnonymousClassOfC`.

Second, adding instance variables to the anonymous class will obviously change the structure of the object. The behavior of users checking for the equality of an instrumented object with other objects may be impacted.

Affected tools will be, for example, the inspector or the debugger.

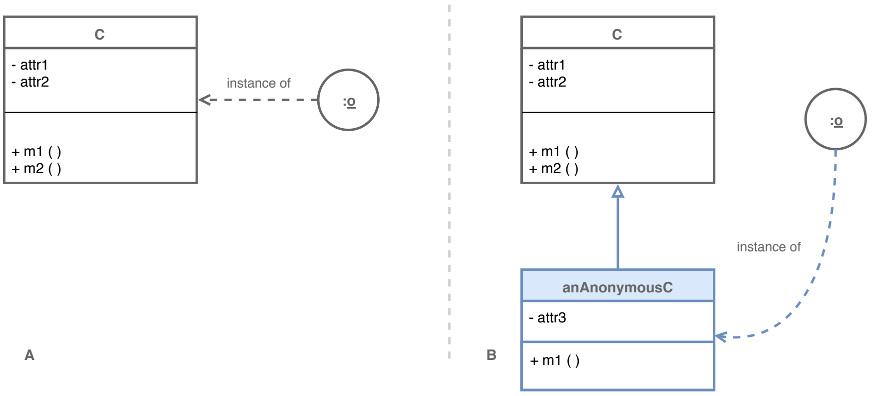

Figure 4-1 (A) An object *o* is instance of a class *C*. (B) An anonymous subclass of *C* implements object-centric behavior and structure, and *o* is migrated to that anonymous class.

It is one of the fastest known implementation for object-centric instrumentation [Duc99].

4.1 Example

Anonymous subclasses are derived from the original class of the object (line 3). Methods must be manually (re)written with instrumentation and compiled in the new class (line 4-8). Then the object has to be migrated to its new class (line 10). To rollback the instrumentation, the object must be manually migrated back to its original class (line 12). The migration is not *safe* if more than one process is using the instrumented object.

```
1  |person anonClass|
2  person := Person new.
   anonClass := anObject class newAnonymousSubclass.
```

```
1  anonClass
2    compile:
       'name: aName
4        self tag: aName.
         name := aName'.
```

```
1  anonClass adoptInstance: person.
```

```
1  anonClass superclass adoptInstance: person.
```

4.2 Evaluation

Strengths and weaknesses.

+ Fastest known technique for object-centric instrumentation in Pharo

+ Interesting starting point to build solutions on top

- No abstraction to control or to express object-centric instrumentation

- Very low flexibility regarding the modification of the instrumentation (manual copy and removal of methods to and from anonymous classes)

- Migration of objects to anonymous subclasses is not thread-safe

Manipulated entity: Classes. Behavioral variations are expressed in standard methods, compiled in anonymous classes.

Reusability: Limited. As an anonymous subclass is derived from the original class of an object, only instances of that same class can be migrated to the anonymous subclass. To apply the same instrumentation to an instance of another class, a new anonymous subclass must be created and the instrumented behavior must be recompiled in that subclass.

Flexibility: None. Instrumented methods must always be copied down to anonymous subclasses, and instrumentation must be inserted in the duplicated code. Without any tool built on top, that instrumentation is fully manual.

Granularity: Method. Instrumentation is implemented by recompiling modified copies of methods in anonymous subclasses. Sub-method level is achieved through manual rewriting of the method.

Integration: Limited. The object is migrated to an anonymous subclass, which does not break system tools. However, it is explicit that the object is now instance of an anonymous subclass. It may also break libraries and tools that use classes and class names as a discriminator.

`Self` problem: Solved. By design, as objects are migrated into anonymous subclasses, `self` always references the original object.

`Super` problem: Not solved. There are no means to express how to resolve the lookup when a message is sent to `super` from a method copied down in an anonymous subclass.

4.3 Other documentation

The Pharo Mooc provides materials on object-centric instrumentation based on object class migration and its flavours:

- http://rmod-pharo-mooc.lille.inria.fr/MOOC/Slides/Week7/C019-W7S04-OtherReflective.pdf
- http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_78.html

CHAPTER 5

Talents

Talents are originally behavioral units, that can be attached to an object to add, remove or alter behavior [RGN+14]. Only the object to which a talent is attached is affected by behavioral variations. The latest talent implementation relies on trait definition and anonymous subclasses. Talents can be considered as object-centric, stateful-traits.

5.1 Example

Talents are based on traits. Objects can answer to the `#addTalent:` messages (line 9), which takes a `Trait` as parameter. All behavior defined in the trait is flattened in the object. In the following illustration, we instantiate an anonymous trait (line 3), and we compile a method in this trait (line 4-8). That method is an instrumented version of the original `name` method of the class `Person`. This new method replaces the original one, until the talent is removed from the object (line 10). Talents now relies on anonymous subclasses, to which behavior is flattened before objects are migrated to the anonymous class.

```
1  |person talent|
2    person := Person new.
     talent := Trait new.
4    talent
       compile:
6        'name: aName
            self tag: aName.
8           name := aName'.
     person addTalent: talent. "adds the talent to the object"
10   person removeTalent: talent. "removes the talent from the object"
```

5.2 Evaluation

Strengths and weaknesses.

+ Can reuse and compose behavior from traits

- Inherits from traits limitations: glue code may be necessary and conflicts from composition must be solved manually

- Is Talent addition to objects is thread-safe?

Manipulated entity: Trait. Behavioral variations are expressed using traits. It can be Traits defined in the image or anonymous trait instances in which specific behavior is manually compiled by the developer.

Reusability: Yes. A trait can be added as a Talent to any number of objects.

Flexibility: Limited. Using anonymous traits forces the user to manually compile code in the method. This is however necessary to achieve a sub-method granularity. Conflicts must be resolved manually when Traits are composed.

Granularity: Method. Traits add, remove or alter (through aliasing) the behavior of a method. It can be done at a sub-method level (*e.g.* inserting a statement in the body of a method), but that requires manual rewriting of the method in the Trait.

Integration: Limited. The object is migrated to an anonymous subclass, which does not break system tools. However, it may break libraries that uses classes and class names as a discriminator.

`Self` problem: Solved. By design, as objects are migrated into anonymous subclasses, `self` always references the original object.

`Super` problem: Solved. By flattening all methods that should be found in the super class into the anonymous subclass, and by replacing message sends to `super` by message sends to `self`.

5.3 Other documentation

The new implementation of Talents is available and documented on Github:

• https://github.com/tesonep/pharo-talents

Documentation on Traits:

• https://github.com/pharo-open-documentation/pharo-wiki/blob/master/General/Trai

CHAPTER 6

Ghost

Ghost is a general and uniform proxy implementation[PBF+15]. A proxy replaces an object to control access to that object [ABW98]. Object-centric instrumentation by means of proxies is done by swapping an object (and all its references) with a proxy object (and references to that proxy object). A proxy object is instance of a proxy class, in which access control is defined. Access control is generally implemented through a single interface, which is called each time a message is intercepted by the proxy. Control behavior then decides what to do with the received message.

6.1 Example

In this example, we use the original implementation of Ghost [PBF+15]. In Ghost, intercepted messages are encapsulated into an instance of `GHInterception`. This instance holds the message, the proxy and its target (the oringal object). To handle an interception, we have to model a message handler which implements how the interception is processed. This is done by subclassing the `GHProxyHandler` class of Ghost:

```
GHProxyHandler subclass: #MyProxyHandler
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'MyProxyPackage'
```

In our new handler, we have to implement API methods that will be called by the proxy when a message is intercepted. The first method is the `manageMessage:` method, to which the proxy passes the interception object. We first extract from the interception the message, the proxy and its target, *i.e.* the original object (lines 3-5). Then we check if the intercepted message is the `#name:` message (line 6) and in that case, we perform our instrumentation. That instrumentation (line 7) goes over the proxy by sending a direct message to the original object. The intercepted message is then forwarded to the original object (line 8), and the proxy object is returned (lines 9-11). When the message forwarding returns the real object as a result (*i.e*, the method returned `self`), the proxy is returned instead. In that case, explicitly returning the proxy is important, to ensure that future messages sends to the real object will be intercepted.

```
 1 MyProxyHandler>>manageMessage: interception
 2   | message proxy target result |
     message := interception message.
 4   proxy := interception proxy.
     target := proxy proxyTarget.
 6   message selector == #name:
       ifTrue: [ target tag: message arguments first ].
 8   result := message sendTo: target.
     ^ result == target
10     ifTrue: [ proxy ]
       ifFalse: [ result ]
```

When the proxy is installed, the real object references will be exchanged with references to the proxy. Uninstalling the proxy must perform the opposite operation, and restore references to the original object. This is specified through the `handleUninstall:` API method in the proxy handler. That method is called by the proxy when the special message `#uninstall` is intercepted.

```
1 MyProxyHandler>>handleUninstall: anInterception
2   ^ anInterception proxy proxyTarget become: anInterception proxy.
```

To install the proxy, we use the `GHTargetBasedProxy` from the Ghost proxy model. The `createProxyAndReplace:handler:` interface (line 4) replaces all references to the real object by references to the proxy, that will intercept all messages meant to be sent to the original object. The parameters given to that method are the object to *proxify* and an instance of our handler class. The `uninstall` interface restores the original object's references (line 5).

```
1 |person|
2   person := Person new.
    GHTargetBasedProxy
4       createProxyAndReplace: person handler: MyProxyHandler new.
    person uninstall
```

6.2 Evaluation

Strengths and weaknesses.

+ Can intercept and control all messages sent to an object

+ Integration to the environment can be finely tuned through the explicit handling of meta-messages
- Dedicated proxy models must be built from the core Ghost proxies to cope with very specific cases, having a unique proxy model handling every possibility is at the cost of performance
- Slower than techniques based on anonymous classes
- Replacement of objects by proxies is not thread-safe

Manipulated entity: Proxies. Proxies are defined in classes which inherit from Ghost internal classes. Typically, developers subclass the base message handler from Ghost to create a proxy model that implements the wanted instrumentation. An API is provided to apply proxies to objects.

Reusability: Yes. The same proxy model can be reused to instrument any kind of object with the same instrumentation.

Flexibility: Limited. The responsibility to express how intercepted messages are handled falls to the developer. A proxy and a message handling model must be defined through classes. Specifically, as proxies intercept all messages sent to an object, which messages are handled (partial instrumentation) and which are not (*e.g* meta-messages) has to be manually defined or implemented through an *ad-hoc* solution.

Granularity: Method. A proxy intercepts messages sends to the object it *proxifies*. It can execute instrumentation behavior before, after or instead the intercepted message. Sub-method instrumentation cannot be achieved by means of proxies.

Integration: Full. Meta-messages can be explicitly configured as such, and handled as special messages. Therefor, instrumentation can be easily integrated into the environment, so that it does not interfere with tools.

`Self` problem: Solved. Messages sends to `self` can be intercepted, however it is implementation dependent. The original implementation of Ghost does not solve the `self` problem, but recent implementations do.

`Super` problem: Solved. Implementation dependent, see `Self` problem.

6.3 Other documentation

Implementations and documentation based on the original Ghost paper [PBF+15]:

- http://esug.org/data/ESUG2011/IWST/PRESENTATIONS/23.Mariano_Peck-Ghost-ESUG2011.pdf
- https://rmod.inria.fr/archives/papers/Mart14z-Ghost-Final.pdf
- https://gitlab.inria.fr/RMOD/Ghost

- https://github.com/guillep/avatar

Another implementation of Ghost:

- https://github.com/pharo-ide/Ghost
- http://dionisiydk.blogspot.com/2016/04/halt-next-object-call.html

CHAPTER 7

MetaLink

A metalink annotates the abstract syntax tree (AST) of a program with user-defined meta-behavior. At run-time, *i.e.* when the annotated node is executed, the metalink is triggered and executes the meta-behavior. To that end, a metalink is configured with a meta-object, a selector, and a list of arguments (reifications from the execution context). When the metalink is triggered, the message designated by the selector is sent to the meta-object, with the arguments list as parameter. Metalinks can annotate the AST of a method, or any sub-node of that AST, *e.g.* a message send or a variable read. By default, a metalink will trigger meta-behavior for all instances of the class from which a method's AST has been annotated. Metalinks can be scoped to specific objects, and an API eases the installation of metalinks on variable access (for instance and temporary variables). To scope a metalink to an object, the target AST is copied down in an anonymous subclass (Chapter 4). The metalink is then installed on the AST copy in the anonymous class and the object is migrated to that new class. Metalinks are part of the *Reflectivity* library[Den08], which is included in the reflection layer of Pharo.

7.1 Example

To annotate an AST, we must instantiate a `MetaLink` and configure this instance to implement our instrumentation (line 2). First we pass to the metalink a meta-object (line 3), here the `#object` reification. At run-time, it represents the object executing the current method in which the metalink is triggered. The selector defines the message that will be sent to the meta-object when the link is triggered. Here, it is the `#tag:` selector (line 4). The arguments is a list of reifications from the execution context. As we are trying to capture the value that is being written in the `name` instance variable, we ask for the `#value` reification (line 5). The metalink is then installed on an instance of class `Person` (line 8). The API that is used in this example will install the metalink on all write accesses to the `name` slot of the object. At run-time, the metalink will fire every time something is written in the `name` instance variable, *i.e.* send the `#tag:` message to the `person` object with the value being written as a parameter. The metalink can be uninstalled (line 9), removing all annotations from the AST and restoring the original behavior of the instrumented object.

```
1 |link person|
2 link := MetaLink new.
  link metaObject: #object.
4 link selector: #tag:.
  link arguments: #(#value).

  person := Person new.
8 person link: link toSlotNamed: #name option: #write.
  link uninstall
```

7.2 Evaluation

Strengths and weaknesses.

+ Provides access to fine reifications of the execution context

- Forces the user to manipulate the structure of the program (the AST)

- More suited for specific rather than system-wide instrumentation

- MetaLink installation is not thread-safe

Manipulated entity: Metalinks. Instances of `MetaLink` are the means to express and install meta-behavior on the AST of a program.

Reusability: Yes. The same metalink can be put on any AST node, as long as the reifications asked by the user can be provided by that node. A metalink can be installed on different classes as well as on any number of specific objects at the same time.

Flexibility: Limited. In some cases, object-centric instrumentation by means of metalinks is non-applicable. For example, an instance variable can be accessed both in a method defined in the object's class and in the method it redefines in the super class. Installing a metalink on all accesses to that instance variable may lead to un-predictable modification of the object's behavior. Both methods have the same signature, yet both are going to be copied down for instrumentation in the anonymous subclass, to which the object will be migrated.

Granularity: Sub-Method. Metalinks can be installed on any sub-node of a method's AST, achieving a very fine granularity for instrumentation.

Integration: Limited. Object-centric instrumentation by means of metalinks can break tools relying on meta-information. Typically, tools or libraries relying on structure (class) to discriminate objects will be affected, as instrumented objects are migrated to anonymous subclasses.

`Self` problem: Solved. By design, as objects are migrated into anonymous subclasses, `self` always references the original object.

`Super` problem: Solved. As instrumented objects are migrated to anonymous subclasses, the lookup for messages sent to `super` is altered. In instrumented methods, that lookup is instrumented so that it always resolves in the super class of the original object's class.

7.3 Other documentation

Description of the Reflectivity API:

- https://github.com/SquareBracketAssociates/Booklet-Reflectivity

CHAPTER 8

Change Detector on Read Only Objects

Objects can be put in read-only mode. When the read-only flag is set on an object, the VM cannot write into this object. Any attempt to change a value in an instance variable will fail, and raise an exception. The Change Detector is an experimental project which uses the read-only feature of the VM to monitor and instrument state-access at the level of objects. For every attempt to write into an instance variable of a monitored object, the change detector performs the following steps in order:

- The object is changed to be writeable,
- an exception is raised with contextual data about the write attempt,
- the exception is forwarded to a set of user-defined monitor objects, which can perform custom behavior,
- the state is actually written,
- and the object is put back to read-only mode.

8.1 Example

We implement an `Instrumenter` class, which aims at monitoring particular state-accesses of objects. We first implement the `instrumentWithReadOnly:` method which takes the monitored object as parameter (line 1). This method configures the object to notify `Instrumenter` instances when an attempt to write in one of its instance variable is made. This is done through the `notifyOnChange:` interface, which is available for every object. We then implement a callback (line 4-9), which is executed when attempting to write the state of a monitored object. A `modification` reification is passed to the callback, and contains contextual information about the state access attempt. We know the object that was accessed and the field to which a write access has been attempted. In our example, we only care to intercept write access to the `name` instance variable and do nothing specific for other state access (line 7-8). The instrumentation for our scenario is performed only for write attempts to the `name` instance variable (line 9).

```
1 Instrumenter>>instrumentWithReadOnly: anObject
2   anObject notifyOnChange: self

4 Instrumenter>>objectChangeNotifiedBy: modification
    | object |
6   object := modification object.
    (object class instVarNameForIndex: modification fieldIndex) = #name
8     ifFalse: [ ^ self ].
    object tag: modification newValue
```

8.2 Evaluation

Strengths and weaknesses.

+ Straightforward and simple instrumentation API
+ Direct interception of write accesses to instance variables
- Limited to the control of write access to instance variables
- Much slower than other techniques studied in this booklet
- Change Detector is still an experimental and exploratory project
- It appears to have, for now, problems with concurrency
- Some objects cannot be read-only, see the `setIsReadOnlyObject:` method

Manipulated entity: Objects. Monitored objects are the main entities being manipulated, in addition with user objects plugged on the Change Detector and implementing the instrumentation.

Reusability: N/A. Not applicable.

Flexibility: Limited. Only writing attempts into instance variables can be intercepted. The responsibility to express how intercepted state access are handled falls to the developer. We only implemented an *ad-hoc* solution to control state access within the scope of our example.

Granularity: State access. The Change Detector intercepts write attempts into an object's state.

Integration: Limited. There are integration problems with tools (*e.g.* SUnit), possibly involving concurrency and race conditions.

`Self` problem: `N/A`. Not applicable.

`Super` problem: `N/A`. Not applicable.

8.3 Other documentation

The change detector github repository:

- https://github.com/MarcusDenker/ChangeDetector

pwd!!Towards an object-centric debugger in Pharo

The idea is that any object has an API. Calling this API applies object-centric debugging operators to that object. This API is used to implement object-centric debuggers.

> **To do** confusing paragraph: why would the domain API apply debug operators and be also used to implement debuggers?

In this section, we illustrate how we can implement a simple object-centric debugger using Reflectivity as a backend for reflection.

8.4 Basic structure of the debugger

Our debugger will be composed of three superposed layers built on top of each other. The first and lower layer is the reflective framework that we'll use to instrument our objects. In our example, we'll use Reflectivity.

The second and middle layer is the object-centric debugging API built using the first layer. This second layer provides interface to instrument the object.

The third and top-level layer is the debugger API itself. It uses the second layer to control object-centric instrumentation of the object.

8.5 Implementing the object-centric debugging API

- instrument instance variable access (limitation)
- instrument methods

model: instrumenter that knows how to instrument, using the reflection layer. The instrumenter is an object that knows how to define and apply object-centric instrumentation to an object. This instrumenter relies on the reflection layer.

Let's define our API. The first interface we want is an object-centric breakpoint that halts an object before executing a particular method. We define this interface as follows:

We'll define our instrumenter object as we create new object-centric interfaces. In this chapter, we'll focus on two examples:

- Halting a given method before they execute in a target object,
- recording values written in a given instance variable of a target object.

Implementing an object-centric breakpoint API

Let's take a simple example: imagine two `Point` instances `p1` and `p2`.

```
p1 := 0@2.
p2 := 1@3.
```

Each of these points has two instance variables, `x` and `y`, that we can change by calling the `setX:setY:` method. Imagine that we have a bug related to point `p1`, and that we want to halt when this object executes the `setX:setY:` method. We definitely do *not* want to put a breakpoint directly in the `setX:setY:` method, because points are used all over the system and all of them would halt when that method is called.

We want to stop only when `setX:setY:` is called on `p1`.

This breakpoint is an operator called *halt-on-call*, defined by Jorge Ressia in his Object-Centric debugger [RBN12]; it halts whenever one specific object receives a given message. Let's define an interface for this operator in the `Object` class:

```
Object >> haltOnCallTo: methodSelector
  ^ ObjectCentricInstrumenter new
      halt: methodSelector for: self
```

This message installs a *halt-on-call* breakpoint on its receiver, and returns the object modeling that instrumentation. It is very important to keep a reference to that instrumenter object if we want to uninstall our breakpoint later. This would typically be handled by a higher level tool such as a real debugger.

This method is now our top-level API, available for all objects in the system. Using this API, we can now ask any object to halt when it receives a particular message.

Now, we have to create the `ObjectCentricInstrumenter` class and implement the `halt:for:` method. This class inherits from `Object` and has three instance variables:

- `targetObject`: the target object we instrument;
- `metalink`: the instrumentation per se, that is a metalink as we use Reflectivity in this example;
- `methodNode`: the AST node representing the method in the object we instrument.

```
Object subclass: #ObjectCentricInstrumenter
  instanceVariableNames: 'targetObject metalink methodNode'
  classVariableNames: ''
  package: 'Your-Pharo-Package'
```

In this class, we have to define how we install the *halt-on-call* breakpoint on our object. This is done through the `halt:for:` method. This method takes two parameters: the selector of the message and the target object that the breakpoint will affect. First, we store the `targetObject`. Then, we configure a metalink to send the message `#now` to the `Halt` class *before* the execution of the entity it will instrument. Finally, using Reflectivity's object-centric API, we install that metalink on the given selector in the target object.

When that is done, `targetObject` will halt whenever it receives the message `methodSelector`, but just before executing its corresponding method. All other objects from the system remain unaffected by the new breakpoint.

```
ObjectCentricInstrumenter >> halt: methodSelector for: anObject
  targetObject := anObject.
  metalink := MetaLink new
    metaObject: Halt;
    selector: #now;
    control: #before.
  targetObject link: metalink toMethodNamed: methodSelector
```

Our scenario is summarized by the following code. We instrument our point `p1` with an object-centric breakpoint on the `setX:setY:` method. We store the instrumenter object in the `instrumenter` variable so that we can reuse it later. Calling `setX:setY:` this method on `p1` will halt the system, while calling it on `p2` or any other point will not halt.

```
p1 := 0@2.
p2 := 1@3.
instrumenter := p1 haltOnCallTo: #setX:setY:.
p1 setX: 4 setY: 2. "<- halt!"
p2 setX: 5 setY: 3. "<- no halt"
```

After debugging, we will probably need to uninstall our breakpoint. As we kept a reference to the instrumenter object, we can use it to change or to remove the instrumentation it defines. Let's define an uninstall method in the `ObjectCentricInstrumenter` class. This method just calls the uninstall behavior of the metalink, removing all instrumentation from the target object.

```
ObjectCentricInstrumenter >> uninstall
  metalink uninstall
```

Our little example script becomes:

```
p1 := 0@2.
p2 := 1@3.
instrumenter := p1 haltOnCallTo: #setX:setY:.
p1 setX: 4 setY: 2. "<- halt!"
p2 setX: 5 setY: 3. "<- no halt"
instrumenter uninstall.
p1 setX: 4 setY: 2. "<- no halt"
p2 setX: 5 setY: 3. "<- no halt"
```

Recording all values written to the field of an object

In this version we will instrument an object to record all changes in one of its instance variables

```
Object >> recordValuesStoredIn: instVarName
  ^ ObjectCentricInstrumenter new
      recordValuesWrittenTo: instVarName for: self
```

```
ObjectCentricInstrumenter >> recordValuesWrittenTo: instVarName for:
     anObject
  targetObject := anObject.
  values := OrderedCollection new.
  metalink := MetaLink new
    metaObject: self;
    selector: #recordValue:;
    arguments: #(value);
    control: #after.
  targetObject
    link: metalink
    toSlotNamed: instVarName
    option: #write
```

```
accessors
```

8.6 Implementing the object-centric debugger API

Towards an object-centric debugger

Bibliography

[ABW98] Sherman R Alpert, Kyle Brown, and Bobby Woolf. *The design patterns Smalltalk companion*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[BFJR98] John Brant, Brian Foote, Ralph E Johnson, and Donald Roberts. Wrappers to the rescue. In *European Conference on Object-Oriented Programming*, pages 396–417. Springer, 1998.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. Lulu. com, 2008.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object Oriented Programming*, 12:39–50, 1999.

[FJ89] Brian Foote and Ralph E Johnson. Reflective facilities in smalltalk-80. In *ACM Sigplan Notices*, volume 24, pages 327–335. ACM, 1989.

[HJJ93] Bob Hinkle, Vicki Jones, and Ralph E Johnson. Debugging objects. In *The Smalltalk Report*. Citeseer, 1993.

[PBF$^+$15] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, and Camille Teruel. Ghost: A uniform and general-purpose proxy implementation. *Journal of Object Technology*, 98:339–359, 2015.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.

[RGN$^+$14] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 44(4):413–432, 2014.