



# Object-Centric Instrumentation with Pharo

Steven Costiou

Square Bracket tutorials

February 26, 2019

Copyright 2017 by Steven Costiou.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:  
<http://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):  
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

# Contents

<b>Illustrations</b>	<b>iii</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Illustration example . . . . .	4
1.2 Evaluation criteria . . . . .	5
1.3 Performance overhead evaluation . . . . .	5
<b>2 Summary of the overall evaluations</b>	<b>7</b>
2.1 Performance overhead evaluation . . . . .	7
2.2 Property evaluation overview . . . . .	9
<b>3 Anonymous subclasses</b>	<b>11</b>
3.1 Example . . . . .	11
3.2 Evaluation . . . . .	12
3.3 Other documentation . . . . .	12
<b>4 Talents</b>	<b>13</b>
4.1 Example . . . . .	13
4.2 Evaluation . . . . .	14
4.3 Other documentation . . . . .	14
<b>5 Ghost</b>	<b>15</b>
5.1 Example . . . . .	15
5.2 Evaluation . . . . .	16
5.3 Other documentation . . . . .	17
<b>6 MetaLink</b>	<b>19</b>
6.1 Example . . . . .	19
6.2 Evaluation . . . . .	20
6.3 Other documentation . . . . .	21
<b>7 Change Detector on Read Only Objects</b>	<b>23</b>
7.1 Example . . . . .	23
7.2 Evaluation . . . . .	24
7.3 Other documentation . . . . .	25
<b>8 Towards an object-centric debugger in Pharo</b>	<b>27</b>

**Bibliography**

**29**

# Illustrations

2-1	Comparison of execution times for Anonymous Classes, MetaLink and Talents	8
2-2	Comparison of execution times for Ghost and Change Detector . . . . .	8



## 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.







# 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 so-

lution'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:

```
1 Metacello new
2   baseline: 'ObjectCentricEvaluationExamples';
3   repository:
4     'github://StevenCostiou/PharoObjectCentricEvaluationExamples';
5   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:`.

```
1 Person>>name: aName
2   name := aName
3
4 Person>>tag: aTag
5   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.

```
1 ObjectCentricInstrumentationTest>>assertObjectCentricInstrumentation
2   p1 name: 'Worf'.
3   p2 name: 'Dax'.
4   self assert: p1 tag isNil.
5   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.

```
1 [ ObjectCentricOverheadEvaluation>>evaluateOverheadFor: aPerson
2   ^[ 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

## Summary of the overall evaluations

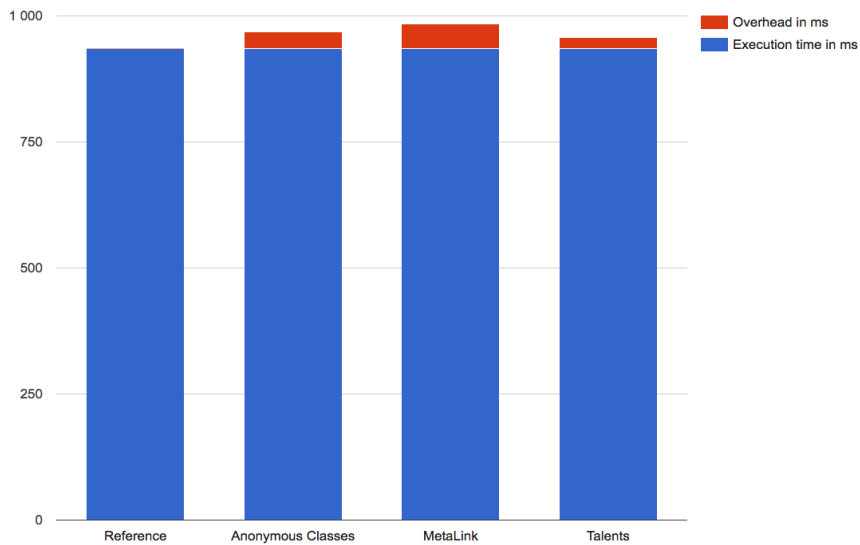
If you already know Pharo and (some of) the presented technique, this chapter is a global summary of the overall evaluation. It contains spoilers, obviously. Results are not commented nor detailed. The performance overhead evaluation, described in Chapter 1, speaks for itself, while the property evaluation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

### 2.1 Performance overhead evaluation

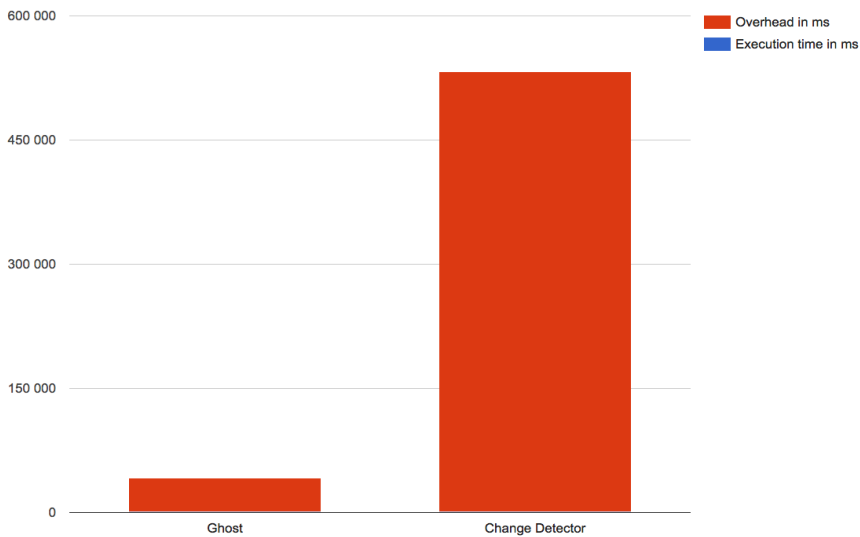
The following table reports execution times and a time factor for each solution against the reference code (see Chapter 1).

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

The following table reports the property evaluation described in Chapter 1. For detailed analysis, please report to chapters 3 to 7.

Prop.	Anon. Classes	MetaLink	Talents	Ghost	Chg Detec.
<b>Entity</b>	Classes	Metalinks	Traits	Proxies	Objects
<b>Reusability</b>	Partial	Yes	Yes	Yes	Yes
<b>Flexibility</b>	None	Limited	Limited	Limited	Limited
<b>Granularity</b>	Method	Sub-method	Method	Method	State access
<b>Integration</b>	Limited	Limited	Limited	Full	Full
<b>Self pb.</b>	Solved	Solved	Solved	Solved	N/A
<b>Super pb.</b>	Unsolved	Solved	Solved	Solved	N/A







## Anonymous subclasses

Anonymous classes are nameless classes that are inserted between an object and its original class [FJ89, HJJ93]. The object is migrated to that new class, which takes the original object's class as its superclass. Methods from the original class can be redefined and reimplemented in the anonymous class, having the effect to change the behavior of that single object. Original behavior that is not redefined in the anonymous subclass is preserved. It is one of the fastest implementation for object-centric instrumentation [Duc99].

### 3.1 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.
3   anonClass := anObject class newAnonymousSubclass.
4   anonClass
5     compile:
6       'name: aName
7         self tag: aName.
8         name := aName'.
9   "migrates the object to its new class"
10  anonClass adoptInstance: person.
11  "migrates back the object to its original class"
12  anonClass superclass adoptInstance: person.
```

## 3.2 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.

## 3.3 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](http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_78.html)



# Talents

Talents are originally behavioral units, that can be attached to an object to add, remove or alter behavior [RGN<sup>+</sup>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.

## 4.1 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.
3 |  talent := Trait new.
4 |  talent
5 |    compile:
6 |      'name: aName
7 |        self tag: aName.
8 |        name := aName'.
9 |  person addTalent: talent. "adds the talent to the object"
10 | person removeTalent: talent. "removes the talent from the object"
```

## 4.2 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`.

## 4.3 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/Traits.md>



# Ghost

Ghost is a general and uniform proxy implementation[PBF<sup>+</sup>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.

## 5.1 Example

In this example, we use the original implementation of Ghost [PBF<sup>+</sup>15]. In Ghost, intercepted messages are encapsulated into an instance of `GHInterception`. This instance holds the message, the proxy and its target (the original 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:

```
1 GHProxyHandler subclass: #MyProxyHandler
2   instanceVariableNames: ''
3   classVariableNames: ''
4   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 instrumenta-

tion. 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 |
3 | message := interception message.
4 | proxy := interception proxy.
5 | target := proxy proxyTarget.
6 | message selector == #name:
7 |   ifTrue: [ target tag: message arguments first ].
8 | result := message sendTo: target.
9 | ^ result == target
10 |   ifTrue: [ proxy ]
11 |   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.
3 |   GHTargetBasedProxy
4 |     createProxyAndReplace: person handler: MyProxyHandler new.
5 |   person uninstall

```

## 5.2 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. Therefore, 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.

## 5.3 Other documentation

Implementations and documentation based on the original Ghost paper [PBF<sup>+</sup>15]:

- [http://esug.org/data/ESUG2011/IWST/PRESENTATIONS/23.Mariano\\_Peck-Ghost-ESUG2011.pdf](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>





# 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 subnode 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 3). 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.

## 6.1 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.
3 link metaObject: #object.
4 link selector: #tag:.
5 link arguments: #(#value).
6
7 person := Person new.
8 person link: link toSlotNamed: #name option: #write.
9 link uninstall

```

## 6.2 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 metaling 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.

## 6.3 Other documentation

Description of the Reflectivity API:

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



# 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.

## 7.1 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 `notify-OnChange:` 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
3
4 Instrumenter>>objectChangeNotifiedBy: modification
5   | object |
6   object := modification object.
7   (object class instVarNameForIndex: modification fieldIndex) = #name
8   ifFalse: [ ^ self ].
9   object tag: modification newValue

```

## 7.2 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.

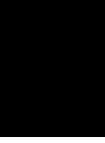
## 7.3 Other documentation

The change detector github repository:

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







# Towards an object-centric debugger in Pharo



# Bibliography

- [ABW98] Sherman R Alpert, Kyle Brown, and Bobby Woolf. *The design patterns Smalltalk companion*. Addison-Wesley Longman Publishing Co., Inc., 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<sup>+</sup>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.
- [RGN<sup>+</sup>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.

