Object-Centric Instrumentation with Pharo

Steven Costiou

Square Bracket tutorials
February 23, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

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 and concludes the book.

Acknowledgements

CHAPTER 1

Introduction

This booklet is about 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, which is a particular case of object-centric instrumentation. We will not go into deep technical usage description, nor into implementation details. Each chapter illustrates one solution with examples, and gives the necessary references if one wants to go deeper in the study of the solution. We study each technique following a three-fold evaluation. First, the studied technique is applied on a simple example of object-centric instrumentation. Second, the technique is evaluated against a set of desirable properties. Finally, performance overhead are evaluated. Only the raw solution is evaluated, without considering the possibility of enhancing the technique by building something on top.

This chapter presents the three-fold evaluation applied to each studied technique, based on the current stable 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 presented in this chapter is available on Github at the following address:

https://github.com/StevenCostiou/PharoObjectCentricEvaluationExamples

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 a `name` instance variable and a `name:` method. This method stores the parameter it is given into the instance variable. We would like that each time a value is stored in that instance variable, that value is printed on the `Transcript`.

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

The instrumentation can be defined as follows, if `aName` is the reference to the value being stored in the `name` instance variable of the `Person` instance:

```
aName logCr
```

The evaluation example is defined in the following script. Two instances `p1` and `p2` of class `Person` are created, and object-centric instrumentation must be applied to the `p2` instance. Then each of these instances is given a name through a call to the `name:` method. The result must be that `p2` prints its name in the `Transcript`, while nothing must happen for `p1`.

```
  |p1 p2|
  p1 := Person new.
  p2 := Person new.
  "Instrumentation must be applied to p2 here"
  p1 name: 'Worf'.
  p2 name: 'Dax'.
  "Only 'Dax' prints in the Transcript"
```

1.2 Evaluation criteria

Each solution is evaluated against the following desirable properties.

Property	Definition
Manipulated entity	The unit of instrumentation (*e.g.* a class, a Trait, an object...)
Reusability	The entity can be reused to instrument different objects
Flexibility	Instrumentation does not put constraint on the source code or in the coding style
Granularity	The level of at which behavior can be instrumented (*e.g.* method, AST...)
Integration	Instrumentation does not break system features

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 without instrumentation with the execution time of an instrumented block of code. The method `evaluateOverheadFor:` from the following script shows how the average execution time is computed. The parameter is an instance of `Person` that is either not instrumented (*i.e.* to compute the reference execution time used for comparison) or instrumented by one of the studied techniques. The `#name:` message is sent a thousand times to the `Person` instance and each time the execution time is recorded. An average of all the execution times is computed and returned by the method. This average time is used to compare execution time of an instrumented instance against the execution time of a non-instrumented instance.

```
1 evaluateOverheadFor: aPerson
2   |execTimes|
3   execTimes := OrderedCollection new.
4   1 to: 1000 do:[:i|
5     execTimes add: [aPerson name: i] timeToRun].
6   ^execTimes average
```

1.4 Structure of the book

The second chapter will provide an overview of the evaluation results of object-centric instrumentation techniques available in Pharo. 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 and concludes the book.

CHAPTER 2

Summary of the overall evaluations

If you already know Pharo and (some of) the presented technique, this chapter is a global summary with spoilers.

CHAPTER 3

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

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

Reusability: Partial. 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. Anonymous subclasses cannot be composed.

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: Partial. 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

CHAPTER 4

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.

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

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: Partial. 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: Partial. 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

CHAPTER 5

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.

5.1 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:

```
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

Manipulated entity: Classes. 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: Partial. 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.

5.3 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 6

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 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

Strenghts 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

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: Partial. 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: Partial. Object-centric instrumentation by means of metalinks can break tools relying on meta-information. Typically, tools or li-

braries 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

CHAPTER 7

Low-level techniques

7.1 Example

Installing Talents

aa

Example

bb

Listing 7-1 Installation from Github

```
Metacello new
  baseline: 'Talents';
  repository: 'github://tesonep/pharo-talents/src';
  load.
```

Listing 7-2 Installation from Github

```
talent := Trait named: 'MyTalent'.
talent compile: 'add: anObject
anObject logCr.
super add: anObject'.
col := OrderedCollection new.
col addTalent: talent.
col add: 'This is an added object.'
```

7.2 Evaluation

cc

Note this is a note annotation.

To do this is a todo annotation

Country	Capital
France	Paris
Belgium	Brussels
Country	**Capital**
France	Paris
Belgium	Brussels

CHAPTER 8

Conclusion

8.1 Example

Installing Talents

aa

Example

bb

Listing 8-1 Installation from Github

```
Metacello new
  baseline: 'Talents';
  repository: 'github://tesonep/pharo-talents/src';
  load.
```

Listing 8-2 Installation from Github

```
talent := Trait named: 'MyTalent'.
talent compile: 'add: anObject
anObject logCr.
super add: anObject'.
col := OrderedCollection new.
col addTalent: talent.
col add: 'This is an added object.'
```

8.2 Evaluation

cc

Note this is a note annotation.

To do this is a todo annotation

Country	Capital
France	Paris
Belgium	Brussels
Country	**Capital**
France	Paris
Belgium	Brussels

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+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+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.