Reflection as a Tool to Debug Objects

Steven Costiou
Univ. Lille, Inria, CNRS, Centrale Lille,
UMR 9189 CRIStAL
F-59000 Lille, France
steven.costiou@inria.fr

Vincent Aranega
Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 CRIStAL
F-59000 Lille, France
vincent.aranega@inria.fr

Marcus Denker
Univ. Lille, Inria, CNRS, Centrale Lille,
UMR 9189 CRIStAL
F-59000 Lille, France
marcus.denker@inria.fr

Abstract

In this paper, we share our experience with using reflection as a systematic tool to build advanced debuggers. We illustrate the usage and combination of reflection techniques for the implementation of object-centric debugging. Object-centric debugging is a technique for object-oriented systems that scopes debugging operations to specific objects. The implementation of this technique is not straightforward, as there are, to the best of our knowledge, no description in the literature about how to build such debugger.

We describe an implementation of object-centric breakpoints. We built these breakpoints with Pharo, a highly reflective system, based on the combination of different classical reflection techniques: proxy, anonymous subclasses, and sub-method partial behavioral reflection. Because this implementation is based on common reflective techniques, it is applicable to other reflective languages and systems for which a set of identified primitives are available.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Classes and objects**; **Object oriented development**; **Software testing and debugging**.

Keywords: Reflection, Meta-Objects, Proxies, Anonymous subclasses, Object-centric debugging

1 Introduction

Reflection is a language feature that enables a system to observe and modify itself at run time [14]. Reflection is useful to build debugging tools because it allows us to manipulate running executions. For instance, we can use reflection to acquire execution meta-data (*i.e.*, *reifications*) that is not available from the base program in a normal execution. We can, *e.g.*,, obtain the name of a variable being assigned, then trigger a debugging operation with that variable name as parameter (*e.g.*, logging, halting...).

In this paper, we share our experience with using Reflection as a systematic tool to implement advanced debugging features. We describe how we use and combine reflection techniques to implement object-centric breakpoints. These breakpoints are part of object-centric debugging [12, 13], which formulates the hypothesis that focusing the scope of debugging (views, interactions, operations) on singular objects would ease the debugging of object-oriented programs.

Scoping debugging operations (such as breakpoints) on specific objects is difficult. The most straightforward solution is to use conditions to check for the identity of objects. Most development environments provide conditional breakpoints and a way of identifying objects through a unique identifier, a pointer or a memory address. To implement object-centric breakpoints, we could compare the receiver of a method hitting a breakpoint with a list of target objects, and activate the breakpoint only if that receiver is among the targets.

However, this is tedious to implement and it does not scale. It requires a heavy machinery, with intelligence to dynamically add and remove objects from the target list during debugging, and global state to store that target list. It is also ineffective for implementing operations such as *break each time a target object executes any method.* This requires to place conditional breakpoints in all the class hierarchy of the target object to make sure all possible methods in the code are instrumented. All these methods would be instrumented, without the guarantee that they would even be executed at run time. This would induce a performance penalty for all objects using that code but not concerned by the object-centric breakpoint. Finally, this would not really be "object-centric", as it applies globally and filters objects with conditionals instead of really be specific to certain objects.

In the following, we summarize the specifications of an object-centric debugger (Section 2). We describe the reflection-based implementation of object-centric breakpoints that exists in Pharo[1] [2], and the limits of this implementation (Section 3). Then, we describe how we overcome these limits by combining various other reflective techniques (Section 4 & 5). For the sake of concision, we describe our implementation mostly from a conceptual level. We explain how we combined different reflective techniques to solve implementation problems inherent to object-centric instrumentation. We briefly cover the practical integration of our breakpoints into the Pharo debugging environment (Section 6). Finally, we discuss the applicability of our solution to other languages and systems (Section 7), so that a reader could reproduce our implementation.

2 Object-Centric Debugging in a Nutshell

In object-centric debugging, one must interrupt a first time an execution with conventional breakpoints, select an object to debug, then apply an object-centric operation on that object [12]. This requires modifying methods during run

[1]The Pharo syntax fits on a postcard and is available online: https://commons.wikimedia.org/wiki/File:Pharo_syntax_postcard.svg

time, *e.g.*, to install an object-centric breakpoint. This makes reflection a candidate of choice for implementation.

Debugging operations are activated upon different events related to an object's structure and behavior. For a target object O, we consider three kind of events:

- State access: variables of O are read or written.
- Message receive: an object tells O to execute a method.
- Message send: O tells an object to execute a method.

Originally, object-centric debugging [13] defines more events, *e.g.*, when O receives a message with a specific object as parameter. However, we do not consider them as the three aforementioned events are sufficient to detect finer-grained events by complementing debugging operations with conditionals over the events' data. The only event that cannot be detected from these three base events is when a class is instantiated. How to intercept this event strongly depends on languages and their implementation. Therefore, we let it out of the scope of this paper.

For each of these events, debugging operations must be applicable with a different *granularity*. We might be interested in debugging a single event, *e.g.,* a specific message send or a specific assignment, or we might be interested in anything that happens to O (all messages and state accesses). On another dimension, we want to be able to filter events by variable name (for state accesses) or by message name (for message sends), or filter by kind of events, *e.g.,* all readings or writings into one or all of the variables of O.

3 Object-Centric Breakpoints in Pharo

Pharo implements a limited subset of the breakpoints described by object-centric debugging. For that, Pharo relies on Reflectivity [3], a reflective library that provides fine-grained reification of run-time entities and a mechanism similar to AOP [7] pointcuts to target sub-elements of methods (message sends, assignments, ...). It provides a precise selection of the code to instrument, down to the AST level. We describe Reflectivity and how it is used to build breakpoints for state-access and message sends in Pharo.

3.1 Sub-Method Partial Behavioral Reflection

With Reflectivity, developers instrument their system at run time using *metalinks*. A metalink is an object that associates an AST node to a meta-object and parameters: (1) which behavior to call on the meta-object on the node execution, (2) when to call this behavior (before, instead or after the node execution) and (3) which run-time entities to reify and pass to the meta-object. A metalink can span many nodes (*i.e.,* cross-cutting instrumentation), or be dedicated to one specific node. Installing metalinks dynamically transforms, recompiles and reinstalls code.

3.2 Building Breakpoints with Reflectivity

Pharo breakpoints are all based on metalinks installed on AST nodes. These metalinks are configured to trigger the `now:`[2] method of the meta-object `Break` (*i.e.,* the breakpoint class) just before the AST nodes execution:

```
metalink := MetaLink new.
metalink metaObject: Break selector: #now: control: #before.
```

The breakpoint metalink is defined, and now needs to be installed on relevant AST nodes. Reflectivity provides a cross-cutting API to span metalinks through class hierarchies. In the following, we describe how this cross-cutting API is used in Pharo to install breakpoint metalinks across the class hierarchy sketched in Figure 1. We present the code[3] for the three object-centric debugging aspects: state access, message send and receive.

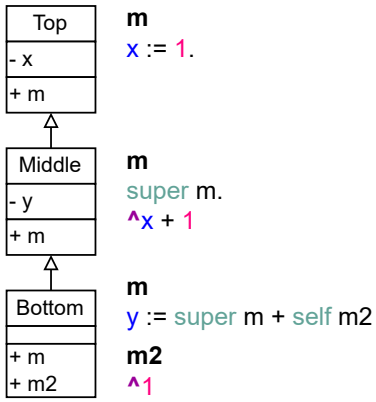

Figure 1. Example class hierarchy.

State-access breakpoints. The metalink is configured with arguments that reify variable access data from the execution, then installed on the instance variable `x` of class `Middle`:

```
metalink arguments: #(receiver variable value newValue).
Middle link: metalink toVarNamed: #x
```

Variable `x` is accessed 2 times in our class hierarchy: in methods from `Middle` (a read access) and from `Top` (a write access). Reflectivity automatically resolves the AST nodes corresponding to these accesses and installs the metalink on them. At run time, when these accesses happen, the metalink sends the `now:` message to the `Break` class with as arguments an array containing: the object assigning a value to `x` (the `receiver`), the variable binding `x` (the `variable`), the value of `x` (`value`) and the new value assigned to `x` in a write access context (`newValue`). Reflectivity has a broader API to select

[2] In Pharo, `#now:` is a message selector that takes one parameter. When sent to an object O, the Java equivalent of O `now: 0` would be O`.now(0)`.
[3] In Pharo, `:=` is the assignment operator and `^` is the return operator.

reads and writes accesses, and to span a metalink through all variables of a class hierachy.

Message receive breakpoints. A new metalink is instantiated as described above, configured with the *receiver* reification to obtain the instance receiving the message, then installed on the method `m` of class `Bottom`:

```
metalink arguments: #(receiver).
(Bottom>>#m) ast link: metalink
```

At run time, when the method `m` is executed on an instance of `Bottom`, the metalink sends the `now:` message to the meta-object with as parameter the receiver of the message `m`. This reification is required since the metalink applies to all instances of the instrumented class. Knowing the receiver is helpful to know which instance is executing the method `m`.

Message send breakpoints. A new breakpoint link is configured to reify the receiver on all message sends within the method `m` of `Bottom`. First, all nodes representing message sends to objects within the method `m` are recovered (*i.e.*, *send nodes*). Then, the link is installed on all these nodes:

```
((Bottom>>#m) ast sendNodes do: [:node| node link: metalink]
```

3.3 Reflectivity and Anonymous Subclasses

To scope instrumentation to single objects without conditionals, Reflectivity uses anonymous subclasses [6]. We illustrate the technique in Figure 2 where we instrument the method `m` of an instance `aBottom` of class `Bottom`. First, Reflectivity dynamically subclasses the class of the object with an anonymous class `anonBottom`. Second, the object is migrated to the new class `anonBottom` using the `adoptInstance` primitive that migrates an object from its class to another one. These operations happen on the first instrumentation of an object. The same anonymous class is reused for further instrumentation of that object. The third and last step is the instrumentation of the method. The method `m` is copied to `anonBottom`, and breakpoint metalinks are installed in that method. Consequently, only the copied method(s) are affected by the breakpoint metalink. As `anonBottom` is a subclass of `Bottom`, the object `aBottom` conserves its protocol and behavior.

3.4 Limits of Pharo's Object-Centric Breakpoints

The Pharo implementation we described is incomplete. First, Reflectivity's reflection capabilities are too limited to cover all events and granularity from Section 2. The combination of Reflectivity and anonymous subclasses works well for intercepting specific messages or state access, but not for intercepting all events occurring for a given object. Second, anonymous subclasses make it difficult to correctly intercept messages sent to the `super` special variable. Reflectivity only instruments methods in anonymous subclasses. Every message sent to `super` ends up executing a method in a super

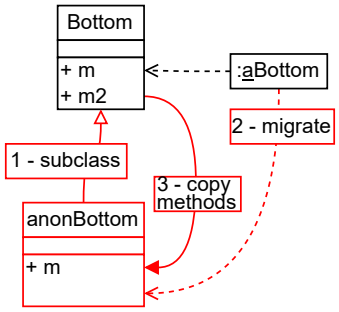

Figure 2. Message-passing control with Anonymous classes.

class, thus escapes from Reflectivity's control. This leads to erratic and undefined behavior. In the next sections, we complete Pharo's object-centric breakpoints implementation to overcome these limits.

4 The super Problem

This problem appears in anonymous subclasses, when one or more messages are sent to the special variable `super` in an instrumented method copied from the original class. We describe this problem using the class hierarchy example from Figure 1. We will instrument an hypothetical object *bottom* instance of `Bottom`. That instrumentation will result in the creation of an anonymous subclass of `Bottom`, named `anonBottom`.

Suppose that we want to intercept calls to `m` for *bottom*. That method is copied down in `anonBottom` and raises two issues:

1. When executing `m` from `anonBottom`, the system performs a super call. This call is statically resolved to class `Bottom` instead of `Middle`. The code of `m` will execute twice, the first time from `anonBottom` and the second time from `Bottom`.
2. Only `m` from `Bottom` was copied down and is, therefore, instrumented. It performs a super call that should execute `m` from `Middle`, which itself performs a super call that should execute `m` from `Top`. The method `m` should be sent three times to *bottom* but we intercept only one of these calls because we only instrument the method defined in class `Bottom`.

Suppose now that we want to intercept read accesses to the variable `x` for *bottom*. The only reading to `x` that is reachable by *bottom* is performed in `m` from class `Middle`. We cannot copy this method in `anonBottom` to instrument the reading. This would mask `m` from `Bottom`, which should normally execute first to perform a super call to `m` from `Middle` that performs this reading.

We solve these problems by replicating the *super call chain* and flattening it in the anonymous subclass as illustrated in Figure 3. First, when the Reflectivity API attains a node to instrument, we scan the class hierarchy of the method

defining this node. We collect and organize sequentially all methods part of a *super call chain* which include that method. We then copy all methods from that *super call chain* in the anonymous subclass with an alias to differentiate them since they all have the same selector. Finally, we have to replace each super-send by a self-send to the corresponding aliased method in all methods of the *super call chain*.

The following simplified code shows how we use Reflectivity to achieve such a replacement:

```
methodChain do:[:mc| |link| "link is a temporary variable"
  link := MetaLink new.
  link metaObject: self selector: mc aliasedSelector.
  link  control: #instead.
  m ast superSends do:[:node| node link: link]
```

For each method `mc` in the chain, we build a new metalink. This link is configured to send the *aliasedSelector* of each method to `self` instead of `super`. This *aliasedSelector* is the aliased name of the method copied from the super class of the class defining the method `mc`. In Figure 3, if the method being instrumented `mc` is the copy of `m` from `Bottom`, then its aliased selector is `_Middle_m`. For each method in the chain, we install one such link on all super-send nodes defined in the method ast. At run time, the link ensures that the aliased methods are called on `self` instead of performing super calls.

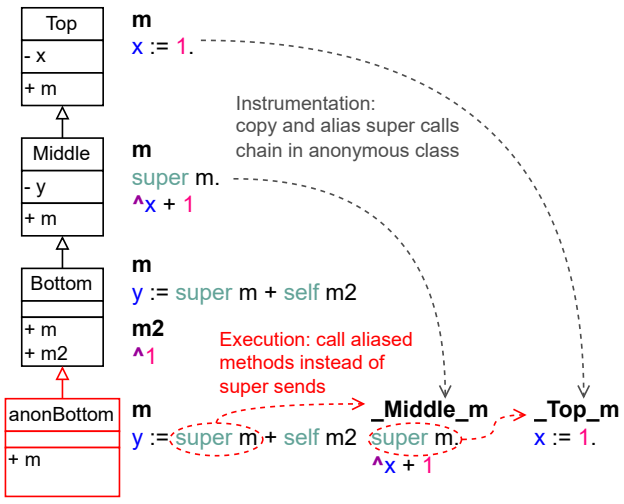

Figure 3. Solving the `super` problem.

5 Intercepting and Instrumenting On-The-Fly

The combination of Reflectivity and anonymous subclasses described in Section 3.3 and Section 4 is sufficient to obtain fine-grained object-centric breakpoints for *specific* methods and variable accesses. However, breaking *every time* an object receives or sends a message requires a coarse-grained solution. We could copy all methods from the class hierarchy of the instrumented object in the anonymous subclass, taking care of aliasing those concerned by super-sends. We argue this is not satisfying. It induces a potentially heavy compilation and instrumentation cost as we have to duplicate and instrument all methods up to `Object` for each object in which we install a breakpoint.

In this section, we introduce a meta-object to automatically propagate metalinks in instrumented methods, and a proxy to ensure the interception of all messages received by the target object.

5.1 A Meta-Object to Propagate Metalinks

We replace the meta-object of our metalinks, *i.e.,* the `Break` class, by a dedicated meta-object. Each time a metalink fires upon a message send, the link sends that message to the meta-object instead of directly breaking the execution. For that, we modify the configuration of the metalinks we install:

```
metalink metaObject: metaObject selector: #send:withArgs:
```

The meta-object is responsible to propagate the instrumentation in the method to call with new metalinks, and then to break the execution if necessary (thus realizing object-centric breakpoints). We illustrate this in Figure 4. In this illustration, the method `m` of the target object `aBottom` has already been instrumented. All message sends in `m` therefore have their own metalink. When `aBottom` receives `m`, the corresponding method is executed (1). When the message send `m2` is executed, it fires the metalink that sends `m2` with its arguments and contextual reifications to the meta-object (2). The meta-object first copies `m2` in `anonBottom` and instruments all its message sends. Then, it sends back the `m2` message to `aBottom` which executes the corresponding method, and so on.

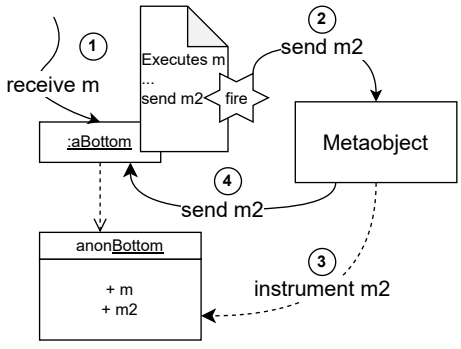

Figure 4. A meta-object to propagate metalinks.

The meta-object does not necessarily trigger a breakpoint each time it intercepts a message send. It only cares about implementing and propagating the instrumentation properly. This means the meta-object is now responsible of implementing the solution to the `super` problem, as well as defining and

configuring metalinks for each instrumented node. This also means that the meta-object implements an API and owns state to specify which nodes are going to trigger breakpoints. All data and instrumentation relative to an instrumented object is configured and held by the meta-object. For instance, the meta-object redefines the links installed on instance variable accesses to intercept these accesses and act accordingly:

```
metalink metaObject: metaObject selector: #iVarRW:withArgs:
```

This dynamic propagation of metalinks ensures that only methods that are actually called during an execution are instrumented. In effect, we use partial reflection to dynamically propagate an object-centric MOP that reifies all operations that could trigger breakpoints.

5.2 Proxies to Catch External Calls

We now need to intercept all messages received by our instrumented objects. We use a *forwarding proxy* [17] object (Figure 5) that wraps our target object `aBottom`. Upon wrapping, we use the `become` primitive [8] that swaps all references of `aBottom` with references to that proxy. Other objects now dialogue exclusively with our proxy, and not with the real object that is only known by the meta-object. Messages received through the proxy are systematically forwarded to the meta-object that acts as described in Section 5.1.

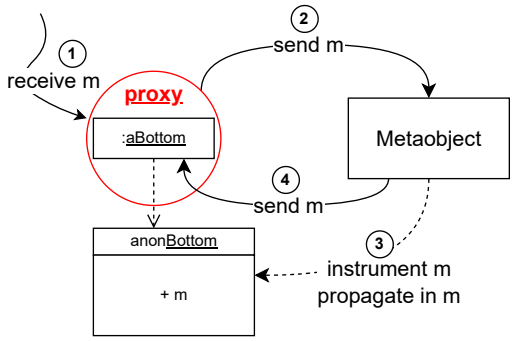

Figure 5. Proxies to catch external calls.

References to the target object can still escape. Any external object calling a method (intercepted by the proxy or the meta-object) either returning *self* or passing back `self` as parameter of a message send (*e.g.,* a visitor) will obtain a reference to the target object. The external object will then be able to bypass the proxy and the meta-object and send direct messages to the target object. To solve this problem, when instrumenting a method the meta-object adds a metalink on such escaping `self` nodes. It looks for all `self` nodes passed as parameters or directly returned by the instrumented method. At run time, the link replaces the `self` reference by a reference to the proxy. Therefore, we make sure that no reference to the target object escapes, and that any message send destined to the target object goes through the meta-object or the proxy first.

In this design, we separate message passing control (proxy and meta-object) and instrumentation (the anonymous class). This simplifies testing, debugging and the evolution of the tool.

6 Object-Centric Breakpoints in Practice

In practice, our object-centric breakpoints need to be transparently accessible by developers from the system tools they are accustomed to. Originally, object-centric debugging is meant to augment standard debugging tools [13]. In Pharo, we integrated our object-centric debugger in the reflective tools of the language, such as the Pharo object inspector. (Appendix A, Figure 7) In addition, we developed new tools to control object-centric breakpoints, *e.g.*, navigate them or remove them (Appendix A, Figure 6). Finally, when an object triggers an object-centric breakpoint, a debugger opens and shows a standard debugging view. Whenever a view tries to display information about that object, that view sends message to the object which might trigger an object-centric breakpoint again. This can recursively go on and freeze the system. To be able to use the debugger, we added a meta-level boolean [5, 15] in our meta-objects. Whenever an object-centric breakpoint hits, it triggers a reflective action which executes code at the meta-level [5] (*i.e.*, not in the program's functional code). Before executing the reflective action, the meta-object sets its meta-level to `true`, which blocks potential meta-recursions. When closed, the debugger sets the meta-level back to `false` which reactivates breakpoints.

7 Implementation Requirements

Implementing our solution requires three elements. It requires *Sub-method* and *Partial Behavioral Reflection* (Section 3.1). These features are available in Java with Reflex [16] or AOP [7], or in Python through run-time AST transformations [1, 4]. Then, it requires the `adoptInstance` primitive to dynamically change an object's class (Section 3.3). In Python the class of an object can be changed to another one at run time. This other class can be dynamically generated as a subclass of the original object's class to reproduce the anonymous subclass mechanism. This is possible, to some extent, in Java using the *unsafe* api [9]. Finally, it requires the `become` primitive [8] to swap all references to an instrumented object by a reference to a proxy (Section 5). Some Python libraries enable this feature [11]. The java *unsafe* API can be used to scan the heap and swap object references [10].

8 Conclusion

We described how we combined reflection techniques to build object-centric breakpoints in Pharo: *Sub-method Partial Behavioral Reflection*, *Anonymous Subclasses* and *Proxy*. These techniques complement each other when implementing object-centric breakpoints features. Our implementation is applicable to languages exposing the *adoptInstance* and

become primitives, and cross-cutting, sub-method reflection. This is, to the best of our knowledge, the first reproducible guide for object-centric debugging implementation.

Acknowledgments

This work was funded by the ANR JCJC OCRE Project (https://anr.fr/Project-ANR-21-CE25-0004).

References

[1] Vincent Aranega, Steven Costiou, and Marcus Denker. 2021. *Tool demo: fine-grained run-time reflection in Python with Reflectivipy.* Research Report. Inria. https://hal.inria.fr/hal-03463035

[2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland. 333 pages.

[3] Steven Costiou, Vincent Aranega, and Marcus Denker. 2020. Sub-method, partial behavioral reflection with Reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming* 4, 3 (2020). https://doi.org/10.22152/programming-journal.org/2020/4/5

[4] Steven Costiou, Vincent Aranega, and Marcus Denker. 2021. *Reflectivipy: building python debuggers with sub-method, partial behavioral reflection.* Poster. GPL 2021 - Génie de la Programmation et du Logiciel.

[5] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. 2008. The Meta in Meta-object Architectures. In *Proceedings of TOOLS EUROPE 2008 (LNBIP, Vol. 11).* Springer-Verlag, 218–237. https://doi.org/10.1007/978-3-540-69824-1_13

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

[7] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. 1997. *Aspect-oriented programming.* Technical Report. Xerox Palo Alto Reserach Center.

[8] Eliot Miranda and Clément Béra. 2015. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In *International Symposium on Memory Management (ISMM '15).* Portland, United States, 93–104. https://doi.org/10.1145/2754169.2754186

[9] Openjdk-jdk11. 2017. *unsafe.java (Accessed: 2022-07-18).* https://github.com/AdoptOpenJDK/openjdk-jdk11/blob/master/src/java.base/share/classes/jdk/internal/misc/Unsafe.java

[10] Luis Pina. 2014. *Unsafe Rubah (Accessed: 2022-07-18).* https://www.luispina.me/2014/12/01/Unsafe_Rubah.html

[11] pyhtonhosted.org. 2011. *pyjack 0.3.2 (Accessed: 2022-07-18).* https://pythonhosted.org/pyjack/

[12] Jorge Ressia. 2012. *Object-Centric Reflection.* Ph. D. Dissertation. Institut fur Informatik und angewandte Mathematik.

[13] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-Centric Debugging. In *Proceeding of the 34rd international conference on Software engineering* (Zurich, Switzerland) *(ICSE '12).* https://doi.org/10.1109/ICSE.2012.6227167

[14] Brian Cantwell Smith. 1984. Reflection and Semantics in Lisp. In *Proceedings of POPL '84.* 23–3. https://doi.org/10.1145/800017.800513

[15] Éric Tanter. 2010. Execution levels for aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development.* 37–48. https://doi.org/10.1016/j.scico.2013.09.002

[16] Éric Tanter, Noury Bouraqadi, and Jacques Noyé. 2001. Reflex — Towards an open reflective extension of Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Vol. 2192. Springer-Verlag, 25–43. https://doi.org/10.1007/3-540-45429-2_2

[17] Camille Teruel, Erwann Wernli, Stéphane Ducasse, and Oscar Nierstrasz. 2015. Propagation of Behavioral Variations with Delegation Proxies. *Transactions on Aspect-Oriented Software Development (TAOSD)* (2015). https://doi.org/10.1007/978-3-662-46734-3_2

Appendices

A Tool Integration in Pharo

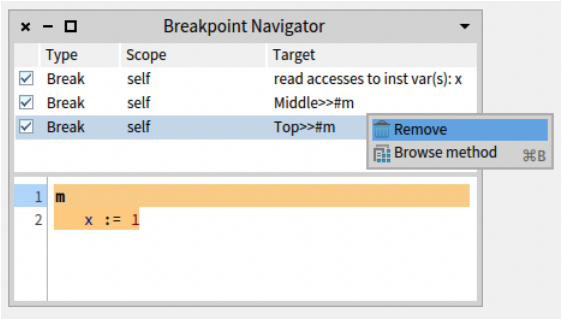

Figure 6. The breakpoint navigator controls all breakpoints installed in the system.

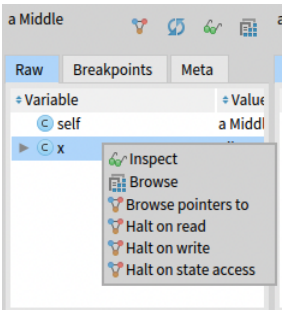

Figure 7. Menus added to the Pharo object inspector to install object-centric breakpoints.

Received 2022-08-08; accepted 2022-09-30