Breakpoint support for Live Environments: The case of Pharo

Clara Allende
Inria RMoD - Universidad Tecnológica Nacional
clari.allende@gmail.com

Guille Polito
Inria RMoD - Institute Telecom Mines Douai
guillermopolito@gmail.com

Abstract

Interactive debugging is an important part of the software development process for live environments such as Pharo [2]. As such, widely used Integrated Development Environments (IDEs) and frameworks provide means to help the developer in this task, for example, the introduction of a *breakpoint i.e.,* a moment at runtime where the execution of a program will be halted. In Pharo Smalltalk the only means to add a breakpoint is to insert a message-send to the Halt class in the source code. Inserting this message-send poses the following problems: it mixes concerns by inserting debugging code within the application code; it generates noise in the versioning tools because it creates a new version of the source code; and it may cause infinite recursions if we try to debug system-level code such as system libraries, the IDE or the language kernel.

To overcome this problems, we developed a Pharo IDE plugin called *SmartBreakpoints*. *SmartBreakpoints* annotates the Abstract Syntax Trees (AST) of the method to debug with breakpoint code. Annotating AST introduces breakpoints in the source code transparently for the developer and versioning tools. Additionally, our plugin provides with a single point of control for breakpoints. This single point of control shows itself beneficial to avoid infinite recursions: we control whether a breakpoint is being triggered from application code or not.

Keywords debugging, breakpoints, live environments, pharo, smalltalk, reflection, metaprogramming

1. Introduction

In a living, dynamic environment such as Pharo Smalltalk a big part of the development process occurs during debugging sessions. The debugger is a very important tool, for it allows to inspect a program's execution in the same environment where it runs. It is also important to notice that by using the reflective capabilities of the language, no special debugging support must be included in the VM. During the history of Pharo, the debugger has changed and evolved: recent research shows that it can become a highly customisable tool [5].

In spite of this, debugging is not such a simple task within the Pharo IDE. There is not any simple means of halting the execution of a program, other than *manual breakpoints i.e.,* introducing a message-send to the Halt class. This leads to several problems: first, modifying the source code to add the breakpoint creates a new version of the source code in the versioning tools; second, given that breakpoint support in Pharo is hosted in the same environment as the debugged application, the code managing a breakpoint can trigger another breakpoint, leading to an infinite recursion. Third, *manual breakpoints* are global: they cannot be constrained to a particular context, and thus they can interfere between themselves.

To address these problems we propose *SmartBreakpoints*, a plugin adding breakpoint support to the Pharo IDE. *SmartBreakpoints* annotates the AST of methods and message sends to add the corresponding code interruptions. AST annotations do not modify the source code of the debugged application, which remains unchanged for the versioning tools and the developer. Our plugin avoids infinite recursions by controlling whether a breakpoint is being triggered from application code or not. This is achieved by accounting that a breakpoint does not trigger another breakpoint, which is possible as we have a single point of breakpoint management. Finally, global interference is leveraged with breakpoint reification.

This paper is structured as follows: In Section 2 we revisit and explain more in detail the problems generated by Halt usage. In section 3, we explain the ideas behind *SmartBreakpoints* and we detail its implementation in section 4 . Finally, we give a brief description of the related work in section 5 and we present our conclusions and further work in section 6.

2. Motivation

2.1 Background: Manual Breakpoints in Pharo

Since debugging is *per se* a non trivial task: developers need tools as helpful and smart as possible [12]. From this perspective, breakpoints are very useful: they allow to spot bugs, and set entry points to inspect a running program.

The current means to add a breakpoint in a program written in Pharo are *manual breakpoints*. A manual breakpoint is the insertion of a message send to the Halt class or any of its variants *(*e.g.,*self halt, Halt once, Halt now, etc)*. Figure 1 shows an example of a manual breakpoint in the method Task»timeToRun inserted as a self halt just before the return statement.

```
Task >> timeToRun
  self halt.
  ^ self fixedTime + self variableTime.
```

Figure 1. Adding a manual breakpoint in a method. We add a breakpoint as a self halt message send in the source code.

When a breakpoint is reached during execution, the Halt class throws an exception. If this exception is not caught by the application code but it is caught by the IDE code, the IDE launches a debugger on the point where that exception was thrown. It is important to notice that as we are in a living environment the application code, the breakpoint code (the exception throwing and its handling) and the debugger run altogether with no distinction *i.e.,* they share the infrastructural elements such as the compiler, the collection library and the language kernel.

The Halt class currently provides some breakpoint specialisation:

- Halt»haltOnce. If enabled, this message interrupts the execution on the next call to haltOnce *found in the system*. Once executed, it will disable the next calls to haltOnce until it is re-enabled manually by the developer.
- Halt»if:. This message triggers a breakpoint if the passed condition is met. The condition can be a block, an expression, or a selector.
- Halt»onCount:. This message halts the execution when a given number of halt calls is reached. However, the calls are accounted *globally* and not for each breakpoint.

2.2 Problems

The current breakpoint support in Pharo leads to several problems. The introduction of debugging code within the application code has an impact in the semantics of the method, but it also can have an impact in the tools and in the IDE itself. We identify three main problems, stated as follows:

Code Versioning. Modifying the source code to add a breakpoint generates a new version it. Versioning tools, unaware of breakpoints, will undesirably version the code. On one hand, this unwanted versioning generates noise for the developer. On the other, we could accidentally commit a manual breakpoint among the other application-specific changes. This can produce unexpected failures in test runs and automated builds.

Manual breakpoints are Global. Manual breakpoints cannot be constrained to a particular context. Because of this, multiple halts can interfere themselves: for instance, let's consider the example in *Figure 2*.

```
Workspace 1:              Workspace 2:
aTask := Task new.        aTask := Task new.
aTask timeToRun.          MyClass new example.
MyClass new example.      aTask timeToRun.

Task >> timeToRun
  self haltOnce.
  ^ self fixedTime + self variableTime

MyClass >> example
   self haltOnce.
   ^ 3+ 4
```

Figure 2. Manual breakpoints can interfere themselves. The message haltOnce halts at the next haltOnce found in the system, thus we can have an unexpected result if there are other halts that we are not aware of.

We can see that if we change the order in which we evaluate the statements in the workspace, we get the breakpoint at different moments. Let's suppose we want to break in the method Task»timeToRun but we are not aware that there is a haltOnce placed in MyClass»example. When we execute the code written in *Workspace 1* the execution will be interrupted where it was expected. However, the code in *Workspace 2* doesn't work in the same way: the breakpoint in MyClass»example is found before, and so it disables the following executions of haltOnce. Then, proceeding the execution after this breakpoint was found will not stop in Task»timeToRun as expected. This problem becomes more evident when debugging unrelated pieces of code within a larger code base.

Manual Breakpoints are not Scoped. Manual breakpoints require especial care when we want to debug infrastructural and debugging elements of the Pharo live environment (*e.g.,* the debugger, the compiler, the collection library, the language kernel.). A carelessly placed breakpoint may trigger, undesirably, another breakpoint in an unrelated context. Additionally, this chained breakpoint activation can be triggered recursively, possibly ad-infinitum [6]. Then, we can say that breakpoints may affect execution scopes that they were not meant to. Figure 3 shows an example of the activation of a badly placed

```
Set >> add: newObject
  "Include newObject as one of the receiver's elements, but only if
  not already present. Answer newObject."
  | index |
  self halt.
  index := self scanFor: newObject.
  (array at: index)
    ifNil: [self atNewIndex: index put: newObject asSetElement].
  ^ newObject

Stack trace when method is installed:
...
ClassOrganization>>classify:under:
Set>>add:
Halt
...
Set>>add:
Halt
...
```

Figure 3. Infinite recursion triggered by a breakpoint. We add a manual breakpoint in the method Set»add:. This method is used to categorise the method and triggers a second breakpoint, which in turn triggers a third breakpoint, and so on.

breakpoint in Set»add:. When we install this breakpoint, the modified method is compiled and installed in the Set class. Then, the IDE tries to classify it (see ClassOrganization»classify:under:) and the breakpoint is recursively triggered. This infinite recursion freezes the environment. Most likely it was not desired to halt the execution in *every* send of add:, but only when that message-send was invoked from application code. Nevertheless, there is no simple way to specify it by just manually adding the message send in the code.

3. SmartBreakpoints

SmartBreakpoints is a Pharo IDE plugin that allows developers to insert breakpoints transparently in the code. This is achieved by modifying the executable representation of a method without changing its source code. For this purpose, we adapt an Abstract Syntax Tree (AST) through annotations: we add meta-data to an AST node to dynamically modify its behaviour to add the breakpoint managing code. These AST annotations are taken into account during compilation to generate the corresponding executable code (cf. Section 3.1).

SmartBreakpoints manages our other two problems by using first class breakpoints. Instead of simply throwing an exception, as it was the case of *manual breakpoints*, we delegate the breakpoint management to a Breakpoint object. A Breakpoint object avoids to interfere with other Breakpoint instances by holding its own state (cf. Section 3.2). They also avoid infinite recursions by controlling whether they are being triggered from another breakpoint execution or not (cf. Section 3.3).

3.1 SmartBreakpoints in Action

We will show how SmartBreakpoints works through an example. Given the method Task»isComplete in Figure 4, we would like to insert a breakpoint at the start of the method, before the execution of the return statement. To insert a breakpoint using SmartBreakpoints, we create a *Breakpoint* instance using the node where we want to place it and its corresponding method. We then enable the breakpoint: it annotates the AST node with the code to trigger the breakpoint (breakpoint breakNow), expands the AST tree to contain the breakpoint message-send, compiles it and installs it into the corresponding class. This means that even though the static representation of the method (*i.e.,* the source code) remains unchanged, we can change the behaviour it describes in runtime. The fact that the addition of a breakpoint does not change the source code implies that there is no change on its version as well. Figure 5 shows the state of a method in its several representations (source code, AST, executable code) before and after breakpoint installation: the source code remains the same, the AST is annotated and new instructions are added to the executable code.

```
Task >> isComplete
    ^ self timeToRun isZero

theMethod := (Task >> #isComplete)
theFirstStatement := theMethod ast statements first.
breakpoint := Breakpoint
                    inMethod: theMethod
                    forNode: theFirstStatement.
breakpoint enable.
```

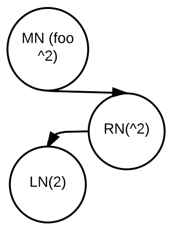
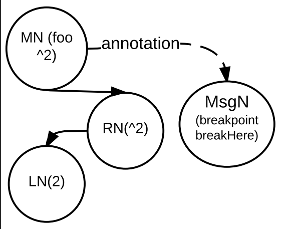
Figure 4. Breakpoint insertion example.

3.2 Scoping with Reified Breakpoints

The reification of the breakpoints solves the interference problem as we have dedicated objects for each inserted breakpoint. Each breakpoint object encapsulates its own state without interfering with other ones. This makes breakpoints independent of the order in where we find them.

Halt once. A halt once breakpoint in SmartBreakpoints knows whether it was already triggered or not. When it is triggered, it will disable itself without affecting other breakpoint instances. As a consequence, subsequent breakpoints remain active and work as expected.

Halt on count. A halt on count breakpoint in SmartBreakpoints accounts how many times the execution passed through him. When it is triggered it increments an internal counter, and if it reaches the given count it will interrupt the execution. Other breakpoint instances are as

	Before inserting breakpoint	After inserting breakpoint
Source Code	`Bar>>foo ^2`	`Bar>>foo ^2`
Simplified AST		 MN (foo ^2) annotation MsgN (breakpoint breakHere) RN(^2) LN(2)
Simplified Executable Code	pushLiteral: 2 return	**pushLiteral: Breakpoint** **send: #halt** pushLiteral: 2 return

References: MN = MethodNode, SN = SequenceNode, MsgN = MessageNode, RN = ReturnNode, LN = LiteralNode

Figure 5. Method state comparison before and after breakpoint insertion. The table shows that the annotated AST will generate different instructions (*i.e.,* IR) but it doesn't affect the source code.

well not affected and the count actually depends on the real times the method was executed.

3.3 Avoiding Infinite Recursion

To solve the possibly infinite recursions, we introduce a meta-object which accounts the meta-levels of execution[6]:

> *(...) we cannot apply behavioral reflection to any system library or to any other code that is executed as part of the meta-object. To enable reflection in mainstream languages, (there is a need of) a reflective architecture where meta-objects control the different aspects of reflection offered by the language..*

For a program run, we identify *levels* of execution. The first level, or *base-level*, is where application code is executed. Every time we use reflection, even for browsing or debugging code, we jump one level above: the *meta-level* of our base-level. If a reflective operation appears during an execution inside a meta-level, we jump to a second meta-level (the meta-level of the first meta-level), and so on. Additionally, when the reflective execution is finished, the jump down one level. Figure 6 sketches the code that checks the meta-level, and jumps one level and evaluates a block if corresponds.

```
RecursionHandler >> executeBlock: aBlock
  (currentContext isValid: self level) ifTrue: [
    currentContext jumpLevelUp.
    ^aBlock ensure: [currentContext jumpLevelDown] ]
```

Figure 6. Meta-level check. The code in the *meta-object* (enclosed in a block) is only executed in the base-level. This avoids infinite recursions.

In SmartBreakpoints, a breakpoint is a meta-object that accounts in which level of execution we are. Triggering a breakpoint, as it is debugging code, is considered a reflective invocation and provokes therefore a level jump. To avoid infinite recursions it checks if the current level of execution is the proper one and interrupts the execution if so.

4. Implementation

Pharo is a reflective language. The IDE, the compiler, and tools are implemented using its reflective features instead of depend on special VM support and external tools. That means that all these components are executed nowadays at the same level of execution than the application code. Any change on either one can affect the other, which may cause meta-level recursions. We want to provide a solution that keeps avoiding special VM support or external tools.

4.1 Our Cornerstone: Reflectivity

Reflectivity [6] is a framework to dynamically adapt methods through annotations on AST nodes. These annotations add an special object, a *meta-link*, to the adapted node's properties. These *meta-links* are used to insert message-sends to *meta-objects* and control the level of execution. Some examples of possible applications are profilers, test coverage, and several applications in debugging support. Figure 7 shows how reflectivity could be used to log a line every time we increment a counter. In this example, we create a meta-link that will be activated in level 0 (*i.e.,* the base-level) and execute the message send Transcript logCr: 'plus 1'. In this case, Transcript is our meta-object.

```
Counter >> inc
  count := count +1.

loggerLink := Metalink metaObject: Transcript selector: #logCr:
                        arguments: 'plus 1' level: 0.

theMethod := (Counter >> #inc).
theNode := theMethod sourceNode.
theNode link: loggerLink.

Wrapper adaptMethod: theMethod.
```

Figure 7. Simple logging with Reflectivity. We set the meta-object to be the Transcript, to which we send the message logCr: with a 'plus 1' as logging message.

After creating the meta-link, we get the AST node where we want to install the breakpoint and annotate that node with the meta-link. After annotating the AST, we expand it: a specialised visitor generates a new node for calling the action in the *meta-object* for those nodes that have *meta-links* installed. This expanded AST is used in the following steps of the compiler chain [1] to generate a new compiled method. Afterwards, the adapted method is installed in exchange of the original one. This adaptation process is managed by the Wrapper class.

When we create a *meta-link*, *Reflectivity* allows to refine where and when the added information is to be executed, as shown in our logger example:

- **execution level**: A number that denotes at which level (base-level or meta-level) is valid to execute the code from the link. This provides a solution for the *meta-level recursion* problem. Before executing the code in the *meta-object*, we check that we are in the correct level. If not, we skip the call.
- **position**: Whether the execution of the *meta-object* should be done *before, instead* or *after* the selected AST node. The default is *before*.
- **condition**: A BlockClosure or an expression with boolean value. It allows to set a particular condition to be met for the link to be executed.

4.2 SmartBreakpoints and Reflectivity

SmartBreakpoints relies on *Reflectivity* to add the breakpoint code transparently. We create Breakpoint instances which utilise *meta-links* to add the halt message-send, as shown in *Figure 8*. There is no explicit modification of the source code, since it is performed via the reflective mechanisms provided by the language [1, 6, 9], *i.e.,* AST manipulation and method recompilation. Moreover, there is no need of a dedicated view to debug a program, nor of special VM/ compiler support to insert a breakpoint.

```
Breakpoint >> haltLink
  ^ Metalink metaObject: self selector: #halt level: 0.
```

Figure 8. Breakpoint link creation. We use the Breakpoint instance itself as meta-object, to keep track of its activation state.

Since a breakpoint is installed in an specific node it is very easy to set breakpoints within nested blocks, cascaded messages, etc.

4.3 Specialized Breakpoints

Breakpoint specialisation options currently are as follows:

- **halt once**: once installed, it interrupts the execution **only** for the next call of the adapted method (in contrast to the functionality of Pharo's haltOnce). Then it is automatically uninstalled.
- **halt always**: stops the execution for **all** the following message sends of the method, until explicitly uninstalled.
- **halt on condition**: given a condition (a block closure or any statement with a boolean value), once installed, only stops the execution if and only if that condition is met.

We create a particular kind of breakpoint by specifying a particular argument during during its creation: #always, #once, and #when:. *Figure 9* shows examples of specialised breakpoint creation.

```
Breakpoint class >> break: aSymbol withArguments: anArgArray
    inMethod: aCompiledMethod inNode: aNode
        ^(self new node: aNode; method: aCompiledMethod; yourself)
            break: aSymbol withArguments: anArgArray.

Workspace
theMethod :=  (Task >> isComplete).
theNode := theMethod ast statements first.
condition := [Counter counter isZero].

aHaltAlways := Breakpoint
                        break: #always
                        inMethod: theMethod
                        inNode: theNode.
aHaltOnce :=  Breakpoint
                        break: #once
                        inMethod: theMethod
                        inNode: theNode.
aHaltIf := Breakpoint
                        break: #when:
                        withArguments: condition
                        inMethod: theMethod inNode: theNode.
```

Figure 9. Breakpoint instantiation. When creating breakpoint instances we define which specialisation we want to use.

5. Related Work

Debugging at full speed [10]. There is currently an implementation of AST based breakpoints, presented by Seaton *et al* from Oracle labs. They introduce a prototype debugger for *Ruby Truffle*. This implementation of Ruby includes an AST interpreter for the language that applies aggressive dynamic transformations. This AST Interpreter profiles AST execution to discover frequently executed trees, and inlines the methods involved into a single method.

The authors also propose that all nodes where a developer may want to perform a debugging action are adapted. This is an important difference with our approach: to bypass the performance cost of instrumenting all the nodes, *Ruby Truffle* requires a custom interpreter and a modified VM to perform several optimisations. In contrast, SmartBreakpoints takes advantage of Pharo's infrastructure without changing it (VM, AST Interpreter, and Compiler).

A Pointcut Language for setting Advanced Breakpoints [12]. In this work, the authors present a breakpoint implementation based on AspectJ's pointcuts. These breakpoints are named, and can be composed into higher level breakpoints. In addition, the language provides reifications on the execution context that can be used in the conditions. The debugging tool is completely separated from the debugged program, which allows debugging without changing the source code.

In our approach the debugging happens in the same environment where the program runs. Thus, we need to use Pharo's reflective mechanisms to avoid source code modification. This implies that we have to be careful when we adapt a method for debugging, because we can be affecting both the debugged program and the system itself.

One interesting characteristic of pointcut breakpoints is that they are composable. This is a feature that *SmartBreakpoints* lacks of, but we would like to add in the future.

Object-centric Debugger[8]. Ressia et al. presents a debugger whose abstractions are centered on object instances properties rather than in the classes, the stack, and other static concepts. This debugger is built in top of *Bifrost* [7], a reflection framework for Pharo based on *Reflectivity*. One of its key features is the ability to debug an *already running* program. *Object-centric debugger*'s breakpoints are placed in function of particular objects, instead of in function of source code locations. In addition, they can be scoped to a particular execution.

6. Conclusion and Future Work

In this paper, we acknowledge the need of better support for breakpoint insertion in Pharo Smalltalk. We identify the challenges of developing a tool to enable such support: source code must not be modified and it should be possible to insert a breakpoint in any method in the system.

We propose *SmartBreakpoints*, a tool which dynamically sets breakpoints by performing AST transformations. Our approach makes use of the language's reflective capabilities, thus it doesn't require external tools' support. Being the transformations done at AST level, the source code is not modified: there is no undesired versioning and the breakpoints are set transparently to the developer. In the future we would like to work on the following topics:

Slot integration. There is currently a simple version of slot-based *meta-links*, already integrated into Pharo 4.0 image. Slots [11] are first-class representations of instance variables and their fields, allowing to modify how they are read and written. In Pharo, instance variables, class variables and globals are represented with slots. Hence, having links installed in slots enables support for *state access based breakpoints*: we set a link that triggers a halt every time that the variable is accessed (both read and write).

Breakpoint sessions. It would be interesting to save a set of breakpoints installed during a debugging session. This could enable the ability to uninstall and re-install them on demand, and to version (to share the breakpoint "settings" that allow to reproduce a bug).

Line breakpoints. source-code based breakpoints, as a more common specialisation for the developers. Currently this is a cumbersome task because this kind of breakpoints work with the position in the code, and AST breakpoints don't. Therefore, we need a mapping between both.

Reifications. This is, providing easier mechanisms to reflect on the execution context, and refine breakpoint insertion. For example, reified senders, receiver and arguments could be used to constrain link activation to an specific object, or to an specific context state.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS (no. 295261).

References

[1] Clément Béra and Marcus Denker. Towards a flexible pharo compiler. In *Proceedings of IWST '13 Workshop*, 2013.

[2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, and Damien Pollet. *Pharo by Example*. Square Brackets Associates, 2009.

[3] Eric Bodden. Stateful breakpoints: a practical approach to defining paramterized runtime monitors. In *Proceedings of ESEC/FSE '11*, pages 492–495. ACM SIGSOFT, September 2011.

[4] Rick Chern and Kris De Volder. Debugging with control-flow breakpoints. In *Proceedings of AOSD '07*, pages 96–106, 2007.

[5] Andrei Chis, Oscar Nierstrasz, and Tudor Girba. Towards a moldable debugger. In *Proceedings of Dyla '13*, 2013.

[6] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, 2008.

[7] Jorge Ressia. *Object-Centric Reflection: Unifying Reflection and Bringing it back to Objects*. PhD thesis, University of Bern, October 2012.

[8] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of ICSE '12*, pages 485–495, June 2012.

[9] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[10] Chris Seaton, Michael L. Van de Vanter, and Michael Haupt. Debugging at full speed. In *Proceedings of Dyla '14*, pages 1–13, June 2014.

[11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nirstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of OOPSLA '11*, October 2011.

[12] Haihan Yin, Cristoph Bockish, and Mehmet Aksit. A pointcut language for setting advanced breakpoints. In *Proceedings of AOSD '13*, pages 146–156, March 2013.