Towards Efficient Object-Centric Debugging with Declarative Breakpoints

Claudio Corrodi

Software Composition Group
University of Bern, Switzerland
corrodi@inf.unibe.ch

Abstract

Debuggers are central tools in IDEs for inspecting and repairing software systems. However, they are often generic tools that operate at a low level of abstraction. Developers need to use simple breakpoint capabilities and interpret the raw data presented by the debugger. They are confronted with a large abstraction gap between application domain and debugger presentations. We propose an approach for debugging object-oriented programs, using expressive and flexible breakpoints that operate on sets of objects instead of a particular line of source code. This allows developers to adapt the debugger to their domain and work at a higher level of abstraction, which enables them to be more productive. We give an overview of the approach and demonstrate the idea with a simple use case, and we discuss how our approach differs from existing work.

1 Introduction

Debuggers are important tools for finding and correcting software defects, but are also used by programmers for code comprehension [10]. Conventional debuggers generally offer a similar interface to debugging, where IDEs like Eclipse[1] or IntelliJ IDEA[2] provide ways to set line breakpoints that will halt the execution when reached. The programmer is presented with a view of the source code where the program has halted, a view of the current call stack, and various means to access run-time information (e.g., a browser for variables and their values in the current scope). The operations *step into*, *step over*, and *step out* are commonly used to navigate through the stack and continue stepwise execution until a certain point of interest is reached (which again is visible by viewing the call stack and the corresponding source code).

When working with large, object-oriented projects that have a complex inheritance structure, the execution of a program can be hard to follow since there are many different locations in the

[1] https://www.eclipse.org/
[2] https://www.jetbrains.com/idea/

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016 (sattose.org), Bergen, Norway, 11-13 July 2016, published at http://ceur-ws.org

source code involved. Using the aforementioned debugger operations frequently results in a context switch, since using the *step* operations exclusively is too fine-grained. As a result, a developer often sees stack frames and source code as being irrelevant to the current task. To alleviate this problem, developers need to place multiple breakpoints (which can be tedious and error prone in larger projects) to obtain more meaningful (in the context of the current debugging task) *units of execution*. Furthermore, traditional debuggers do not provide a good interface for exploring the evolution of objects of interest, that is, they do not provide a way to browse the state of objects at an earlier point in the execution.

Conditional breakpoints, commonly available in mainstream IDEs, can be used to halt execution at a certain line of code when a certain condition is satisfied. For example, they can be used to only stop execution when a certain object is involved. However, they only limit whether execution is halted at that particular line and cannot result in halting at different locations in the source code.

Several alternative techniques and tools for debugging object-oriented software exist. Ressia et al. propose a debugger methodology called *Object-Centric Debugging* [9], which focuses on (live) objects instead of source code and the execution stack. During run time, one can specify dynamic breakpoints for individual objects with a set of operations (e.g., *halt on write*, *halt on call*, *halt on object in call*) in order to track its evolution with respect to certain properties of interest. These operations focus on individual objects, while we plan to extend the idea to sets of objects.

Lienhard, Fierz et al. present *Object-Oriented Back-in-Time Debugging* [8] and *Compass* [5], where they track object states at run time and provide tools for inspecting their evolution in the past. In their work, they depart from the traditional stack-based view during halted execution and provide views for analyzing side effects of method invocations (e.g., by presenting at which point in time a variable pointed to a certain value) and visualizing method executions. Data is collected for all objects that are reachable when halting the execution. In contrast, we plan to only track objects that are of interest in the debugging task by specifying, at run time, the objects that need to be tracked.

Chiș et al. make debuggers more dynamic by providing a framework that allows programmers to easily build custom debugger views for their specific domain [3]. They show how domain-specific debuggers can be created to improve comprehension and productivity. For example, they present a debugger for the SUnit testing framework that provides a diff view of the expected and actual values of an `assert: a equals: b` expression, which is automatically opened when the debugger interrupts the program. Apart from the custom views however, the debugger remains in the traditional form with stack trace and source code view.

Stateful Breakpoints [1], proposed by Bodden, are built on runtime monitors for creating complex, temporal breakpoints. For example, one could express a breakpoint that halts whenever a file is read after it has been closed already. This approach is similar to conditional breakpoints, since the possible places where the breakpoint can be hit are fixed at compile time. We will explore different possibilities and existing work for specifying our proposed breakpoints. This includes simple code blocks that return the objects, but also more specialized approaches like regular-expression like languages, the Object Constraint Language[3], and path expressions.

Phang et al. [7] present a trace-based debugging environment that provides a scripting language based on Python. In their approach, the authors provide ways to create execution traces for parts of the program execution. The traces can then be used to detect consistency errors in one's code. Their approach requires one to manually specify the points at which data (i.e., traces) are recorded, something that we aim to solve differently by specifying the data that need to be tracked at one single point only.

Guéhéneuc et al. implemented Caffeine [6], a tool that allows developers to write queries on events of a program execution, such as field reads and writes, or method invocations. These events can be composed to perform dynamic analyses on the execution by executing code when certain sequences of events occur. While the described events are similar to those in object-centric

[3] `http://www.omg.org/spec/OCL/`

debugging, the approach is not integrated into the debugger. Instead, queries are formulated and after the execution of the program an answer is obtained, which can then be interpreted.

We continue in the next section by presenting our ideas using a simple use case. We argue why there is a need for our proposed debugger and point out benefits over other approaches.

2 Efficiently Debugging Object-Oriented Programs

Let us consider a custom implementation (in Pharo Smalltalk) of a linked list, where `LinkedList` objects contain a method `first` that returns the first element. List elements are simple objects of type `Element` and have a method `value`, which returns the value the element represents, and a method `next`, which returns another `Element` instance (or `nil` if the receiver of the message is the last element in the list). Finally, a method `size` returns the number of elements that are currently in the list.

Suppose we are writing the following code, where we first add four elements, then insert one after the second, and finally access the fifth element and print it to a log (`Transcript`) window:

Listing 1: User code manifesting a bug.

```
list := LinkedList new.
list add: '1'; add: '2'; add: '3'; add: '4'.
list insertAfter: 2 value: '2.5'.
Transcript show: (list at: 5).
```

The methods `size` and `insertAfter:value:` are defined in the class `LinkedList` as follows:

Listing 2: Implementations of `size` and `insertAfter:value:` methods of the class `LinkedList`.

```
size
    "Count the elements by iterating through the list."
    | count element |
    count := 0.
    element := first.
    [ element isNotNil ] whileTrue: [
        count := count + 1.
        element := element next.
    ].
    ^ count

insertAfter: anIndex value: aValue
    "Insert a new element after the element at index anIndex."
    | count element newElement |
    count := 1.
    element := first.
    [ count < anIndex ] whileTrue: [
        count := count + 1.
        element := element next.
    ].
    newElement := Element new: aValue.
    newElement next: (element next next). "Bug! Should be '(element next)'."
    element next: newElement.
```

However, when we try to execute the code at the top, an error occurs, telling us that there are fewer than five elements in the list. When trying to find this bug with a traditional debugger, one would usually first find out what the actual `size` method invocation in the assertion returns. Here, the return value is still `4`, meaning that either the `size` implementation contains a bug, or one of the methods that modify the list. The debugger would now require us to step through the list operations and look at the code as it is executed. In particular, one would want to inspect the

methods `add:` and `insertAfter:value:`. In this example, there are only two methods that alter the structure of the list. However, there are other cases where objects are interconnected and have complex structures. An example is that of GUI frameworks, where elements can be nested within each other and children know about their parents and vice versa. The structure of these systems can often be modified in a large number of places in the source code, for example, when user input is handled or business logic is executed and the GUI is updated. Locating a bug in such a system with a traditional debugger can be very challenging, as a large number of possible sources of the errors have to be considered. Stepping through the source code line by line is time consuming, as is adding breakpoints at each location that modifies the structure.

To provide an efficient and high-level approach to finding bugs, we propose to develop a debugger that addresses the mentioned drawbacks of other approaches and allows developers to explore complex object structures (examples include data structures, design patterns, and GUI programs) in the context of their domain. The amount of code that needs to be inspected in order to locate the bug can be minimized and is only required at a late stage of the debugging process. Our envisioned debugger is rooted in the methodology of object-centric debuggers, that is, a debugger that allows a set of objects to be inspected and answer questions like *"When is this object used as a parameter in a message send?"* or *"When is this object modified?"*. We aim to allow developers to define domain-specific properties at which the execution should halt. We call these *declarative breakpoints* and propose to express them either by using arbitrary source code or by providing a domain-specific language. This way, a developer is able to express breakpoint conditions in his domain. In our example, we want to be able to specify expressions like: *"In this list instance, halt execution whenever the* `next` *instance variable of one of its elements is accessed or modified"*. (Note that this differs from simply adding a breakpoint at the setter and getter of `next`, since execution is only halted if the element is actually part of the list we are interested in. This should also apply to future list elements that may not exist yet when specifying the halting conditions.) What we aim for is a way to specify debugging operations in the context of live objects, with the possibility to express complex conditions that can be applied to one's domain. Our proposal differs from conditional breakpoints, since we do not have to specify the line of code where the execution should potentially halt; instead, we specify the operations (e.g., variable write) at which the execution should halt. It differs from object-centric debugging, since declarative breakpoints operate on groups of objects and are dynamically updated to adapt to changing structures at run time.

We revisit Listings 1 and 2 and propose a debugger that can support, for example, the following workflow to fix the presented bug.

1. Since we encounter the bug in the `testSize` method, we insert a conventional breakpoint after the `LinkedList` instance is created. Once the debugger opens, we need to set up the data that needs to be tracked. Here, we should be able to track the list itself, but also all elements that are reachable. We will explore different ways to do this. Initial ideas include a domain-specific language that can express reachable objects (here, we could write an expression like `list first next*` to select objects that are reachable through `list first`, followed by an arbitrary number of `next` messages). Alternatively, we could imagine that arbitrary code can be used to collect the objects. Either way, we aim to provide a simple, straightforward way to select these objects. We will take existing work into account, including path expressions and Object Constraint Language in general, and Expositor [7] and Stateful Breakpoints [1] in particular. By manually specifying the set of relevant objects, we can keep track of the data that is of interest and use the obtained data in visualizations.

2. We continue execution until we reach the (failing) assertion, where we open the debugger again. At this point, the recorded information is available and should be presented in a way similar to the approach of Ressia et al. are doing. Furthermore, we may be able to take advantage of Moldable Debuggers, in particular of custom renderers and presentations of objects in Chiș' work [3]. Figure 1 shows possible visualizations of the recorded data before and after the execution of `insertAfter:value:`.

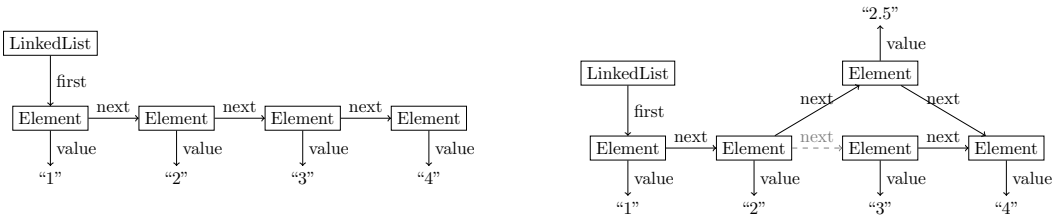

Figure 1: Tracked states before (left) and after (right) `insertAfter:value:` is executed. In the latter, the old `next` reference is shown as a dashed gray edge, akin to the side effect browser in Compass [5].

3. Since we can browse the evolution of the relevant objects, we now realize that one of the elements has become unreachable. We go back to the initial breakpoint and create a declarative breakpoint that halts *whenever a list element that belongs to the list instance we are inspecting updates the* ***`next`*** *reference*. Note that we can define this breakpoint even if the list is empty; execution will nevertheless be halted at some future point when list elements are involved. Finding a simple way to define the breakpoints is closely related to defining the objects that should be tracked in the previous step. Once we find a suitable way to do this, we intend to use it for both tasks.

4. We execute the test once again and halt at several points. Thanks to the flexibility of declarative breakpoints, we step through high-level units of execution, namely whenever a successor of a list element is updated. In the example, this happens in the `add:` and `insertAfter:value:` methods. Once the debugger interrupts program execution at a specified point, we want to present the obtained data to the programmer. As opposed to the generic views common in mainstream IDEs, we think that customized debugger views as seen in the Moldable Debugger Framework can be combined with the presentations that show the history of objects from the back-in-time debugger. Initially, we want this to be done manually, meaning that a programmer has to specify the particular views he wants to see either when interrupting the program execution or when specifying the data that is being tracked. In a later stage, we should consider techniques that can automatically select views and visualizations based on the available data. An example where this has been successfully achieved is the work of Cross et al. [4]. The authors present a classroom environment that can detect certain data structures and provide suitable visualizations for a large array of common structures with high precision.

   In the example, we can use the same visualization as in the previous steps. We can follow the evolution of the list and realize that the bug is inside the `insertAfter:value:` method. We now take a look at the source code of `insertAfter:value:` for the first time and fix the bug (e.g., by using the traditional debugger operations to step through the execution in a more fine-grained fashion).

3 Implementation

In this section, we discuss initial implementation details and outline how we proceed with prototyping the presented debugger.

We decided to implement our prototype in the Pharo[4] Smalltalk environment. Pharo is a dynamic programming environment that offers powerful mechanisms and libraries for reflective operations, making it especially suitable for prototyping debuggers and other IDE tools. Furthermore, the Moldable Tools approach, as introduced by Chiș [2], has been implemented in Pharo, so we can directly implement our approach on top of it. Due to the fast evolution of Pharo and recent changes to its reflective capabilities, we work with Pharo 6, which is currently in development.

[4] `http://pharo.org/`

What follows is a description of the main parts making up our envisioned debugger and how we intend to tackle this.

Ressia et al. implemented object-centric debugging [9] in an earlier version of Pharo using the Bifröst framework, which no longer works in the current Pharo 6 development image. As a consequence, we are reimplementing the object-centric debugging operations using the Reflectivity[5] and Ghost[6] libraries.

Once these object-centric debugging primitives are available, we can use them to build more complex breakpoints on top of them. Currently, we are using block closures to specify declarative breakpoints. For example, to express a declarative breakpoint for the example in Section 2, we can use the following code:

```
bp := DeclarativeBreakpoint new.
bp haltForElements: { list }
   onVariableWrite: #first.
bp haltForElements: [ "block code retrieving list elements omitted" ]
   onVariableWrite: #next.
```

In the second line, we express that for the elements in the block { `list` } (which returns a collection with the root list object), we halt when the `first` attribute is written. Similarly, we do the same for all list elements when writing to the `next` variable. Once the breakpoint is reached, we reevaluate all blocks in order to check for new elements. This way, when we add a new list element (using the `add:` or `insertAfter:value:` messages), we can install an object-specific breakpoint for it as well and thus cover the full list as intended.

Using block closures to discover the target objects and install breakpoints, while powerful, has its disadvantages. One has to be aware of the fact that any code can be passed, and thus side-effects can be an issue. While this approach is convenient, we intend to, as mentioned earlier, explore other ways to specify breakpoints (such as regular expression-like languages).

Since we use conventional breakpoint capabilities internally, debugging with declarative breakpoints results in the same debugger views as when using plain breakpoints. However, since Pharo uses the Moldable Debugger, we have the facilities to extend it with additional step operations and views. We plan to provide high-level operations to manage breakpoints from within the debugger. For example, one should be able to continue execution until the next time a particular declarative breakpoint is reached.

Techniques from back-in-time debugging approaches will be considered at a later stage. While Compass has been implemented in Pharo, it runs only on older images. Furthermore, there are many other approaches, in both industry and academia, that provide ways to step back in a program's execution.

4 Evaluation and future work

In this section, we outline how we plan to proceed with our work. While the current focus is on producing a usable prototype, we intend to evaluate the approach using surveys and user studies. The motivation provided in this section, along with the one in Section 1, will be used to guide the implementation.

Currently, we base our intuition for the need for declarative breakpoints in personal experience with existing debuggers. However, we intend to further motivate and establish the need for our work. In a first step, we intend to do a literature survey with the goal to determine how developers use debuggers and what challenges and difficulties they encounter. Currently, we motivate the need for declarative breakpoints with the work done by Sillito et al. The authors performed two studies to determine what kind of questions programmers pose during development [10]. They find that, once a developer locates an initial focus point (e.g., the location where an error message text is created), many questions are concerned with how objects relate to each other. These questions are

[5] Reflectivity is part of the standard Pharo image.

[6] `http://smalltalkhub.com/#!/~Pharo/Ghost`

described as questions for "Understanding a subgraph" or "Questions over groups of subgraphs" of objects. In the context of our work, we are mainly interested in the questions that are related to runtime data. We think that declarative breakpoints can help to answer some of these questions more effectively and thus provide an improvement over existing breakpoint techniques.

Depending on the findings in the literature survey, we could complement it by conducting a study, where we can identify debugging patterns and problems developers encounter when working with existing debuggers. We can then argue how declarative breakpoints help mitigate those problems and evaluate them in further surveys and studies.

To be able to obtain data from users working with declarative breakpoints, we need to make the implementation available to programmers. By promoting the tools (e.g., by announcing them on mailing lists and providing tutorials) and finding developers that are willing to use our implementation during their day-to-day activities, we will be able to conduct surveys and user studies at a later time. We aim to include the work in the Pharo development image which would mean that programmers can use declarative breakpoints without the additional obstacle of loading the code whenever they load a fresh image.

Currently, we envision two different directions for evaluating the approach. First, we want to conduct a survey with a number of users that have experience with declarative breakpoints. Here, we can focus on answering the following key questions:

- Do programmers use declarative breakpoints to answer domain-specific questions in their day-to-day activities?
- In which situations are declarative breakpoints used (e.g., code comprehension, debugging, testing)?
- Is the cost of using the current API low enough to be practical?

A survey will help us to determine more broad questions and as such may be useful in early stages to guide the further development and particularly the user-facing side of the implementation.

Another direction to evaluate the approach is to perform user studies. Sillito et al. [10] did investigate how developers approach software comprehension and evolution tasks. With a user study, we can investigate how declarative breakpoints change the way that questions—in particular, those related to understanding how sets of objects interact with each other—are answered. One way to do this is to let participants solve generic tasks (e.g., *explain how these components interact with each other*) and see how declarative breakpoints are used, if at all. Another set of tasks can focus on the usability of declarative breakpoints by requiring participants to formalize and answering given domain-specific questions using declarative breakpoints. In such a user study, we can address the following key questions:

- How quickly can users get familiar with the concept of declarative breakpoints and its implementation?
- How much time does it take users to formalize and answer given domain-specific questions using declarative breakpoints?

Addressing questions posed in this section using user studies and surveys will eventually allow us to gauge the practicability and usability of our approach to debugging complex systems.

5 Conclusion

We give an overview of existing approaches to object-oriented debugging and point out shortcomings of existing breakpoint facilities and presentation issues. We outline an idea and possible debugging workflow that allows developers to define expressive breakpoints—called *declarative breakpoints*—that can be used to define halting conditions on the structure within a set of related run-time objects. The proposed approach provides developers with a higher level of abstraction and minimizes the required interaction with traditional debugger facilities to navigate the call stack and program execution. We illustrate the workflow by discussing a simple example where

we locate and fix a bug in a linked list implementation. Declarative breakpoints are useful for programs where many methods modify the state of a set of objects, since we can avoid setting breakpoints in all places in the source code where the modifications take place.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018). We thank Oscar Nierstrasz, Mohammad Ghafari, and Andrei Chiş for their helpful inputs and all SCG members for interesting discussions and brainstorming sessions.

References

[1] Eric Bodden. "Stateful Breakpoints: A Practical Approach to Defining Parameterized Runtime Monitors". In: *ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Sept. 2011, pp. 492–495. URL: http://tubiblio.ulb.tu-darmstadt.de/59323/.

[2] Andrei Chiş. "Moldable Tools". PhD thesis. University of Bern, Sept. 2016.

[3] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. "Practical domain-specific debuggers using the Moldable Debugger framework". In: *Computer Languages, Systems & Structures* 44, Part A (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), pp. 89–113. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.08.005. URL: http://scg.unibe.ch/archive/papers/Chis15c-PracticalDomainSpecificDebuggers.pdf.

[4] James H. Cross II, T. Dean Hendrix, David A. Umphress, Larry A. Barowski, Jhilmil Jain, and Lacey N. Montgomery. "Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches". In: *Trans. Comput. Educ.* 9.2 (June 2009), 13:1–13:32. ISSN: 1946-6226. DOI: 10.1145/1538234.1538240. URL: http://doi.acm.org/10.1145/1538234.1538240.

[5] Julien Fierz. "Compass: Flow-Centric Back-In-Time Debugging". Master's Thesis. University of Bern, Jan. 2009. URL: http://scg.unibe.ch/archive/masters/Fier09a.pdf.

[6] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. "No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs". In: *ASE*. IEEE Computer Society, 2002, p. 117.

[7] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. "Expositor: scriptable time-travel debugging with first-class traces". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 352–361. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486835.

[8] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. "Flow-Centric, Back-In-Time Debugging". In: *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*. Vol. 33. LNBIP. Springer-Verlag, 2009, pp. 272–288. DOI: 10.1007/978-3-642-02571-6_16. URL: http://scg.unibe.ch/archive/papers/Lien09aCompass.pdf.

[9] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. "Object-Centric Debugging". In: *Proceedings of the 34rd international conference on Software engineering*. ICSE '12. Zurich, Switzerland, 2012. DOI: 10.1109/ICSE.2012.6227167. URL: http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf.

[10] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. "Questions programmers ask during software evolution tasks". In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: ACM, 2006, pp. 23–34. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181779. URL: http://people.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf.