Implementing Scoped Method Tracing with ContextJS

Jens Lincke  Robert Krahn  Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam, Germany
{firstname.surname}@hpi.uni-potsdam.de

ABSTRACT
Customized method tracers can be a valuable tool for debugging and program comprehension. They allow to declaratively specify what parts of the call graph should be captured and are an alternative to tedious manual debugging techniques. Method tracers are easy to implement in dynamic languages but avoiding multiple method instrumentation and recursion in the client code can become complex. In this paper we show how Context-oriented Programming (COP) can be leveraged to address such issues. Our approach is based on ContextJS, a COP implementation for JavaScript, which provides scoping mechanisms and an infrastructure for method instrumentation. These abstractions allow to separate target and tracer logic so that self-referentiality is avoided.

1. INTRODUCTION
Tracing is a debugging and program comprehension technique programmers refer to when simple printf style logging and manually stepping through the debugger become too tedious.

Dynamic analysis tools may help gaining insights in the dynamic behavior of programs. But when such tools are not available or programmers encounter specific needs they might employ a custom tracer. Implementing tracers in dynamic languages like JavaScript is easier than implementing them with less reflective languages like C or Java. JavaScript’s reflective capabilities are expressive enough to implement a method call tracer directly by manipulating objects. Modifying source or binary code or instrumenting the virtual machine is not needed.

However, creating such dynamic and reflective meta programs can also be challenging. When the tracer is defined and executed in the same environment as the code under development it should, for example, not accidentally capture its own execution. Using Context-oriented programming (COP) [3,5] can help to avoid such issues by expressing such meta code as a separate concern.

Context-oriented Programming provides infrastructure and scoping mechanisms to directly describe such concerns in a tracer. Dynamic scoping by withLayers and withoutLayers in ContextJS [10] our COP implementation for JavaScript allows to separate target and tracer code so that the tracer does not accidentally trace itself.

Tracing behavior is a typical homogenous crosscutting concern [1] which means that one piece of code should be executed in several places. In contrast to Aspect-oriented Programming (AOP) [8], COP has no explicit support for expressing such concerns, but standard meta programming facilities in JavaScript allow to overcome such limitations.

In this paper, we make the following contributions:
- We describe how scoping of behavioral adaptations with the (de-)activation of COP layers makes the implementation of a method tracing easy
- We show how COP infrastructure can be leveraged to avoid multiple method instrumentations
- We demonstrate how in dynamic languages homogenous concerns can be represented by COP layers using simple meta programming

The remainder of the paper is structured as follows. Section 2 discusses the problems of implementing a simple method tracer in JavaScript. Section 3 gives a short introduction of COP and ContextJS and Section 4 demonstrates how ContextJS can be used to address these problems. Section 5 shows two examples of using customized tracing. Section 6 discusses related work and Section 7 concludes.

2. TRACING WITH JAVASCRIPT
In the following we demonstrate problems that occur when building a custom tracer from scratch in a dynamic language such as JavaScript. Listing 1 defines two JavaScript objects Target and Transcript. The object Target should be instrumented to capture its method invocations.\footnote{The examples be tested out online under \url{http://www.lively-kernel.org/repository/webwerkstatt/demos/contextjs/SimpleObjectTracing.xhtml} (visited 2011-04-20)}

Implementing method call tracing can be accomplished by JavaScript’s built-in meta programming facilities. In JavaScript everything is build out of objects and functions, so methods are defined by putting functions in object properties. The difference to normal function calls is that the this pseudo variable is bound to the object at call time (in Figure 1, m2 is defined in Line 3 and called in Line 20). Listing 2 shows how adding additional behavior to methods m1

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

COP’11, July 25, 2011, Lancaster, UK.
Copyright 2011 ACM 978-1-4503-0891-5/11/07 ...$5.00.
Target = new Object();
Target.m1 = function(a) { return a * 2};
Target.m2 = function() { return this.m1(this.p)};
Target.toString = function() {return "Target"};
Target.print = function() {
Transcript.show(this.toString() + ", p = " + this.p);
};

// Definition of simple tool object
Transcript = new Object();
Transcript.toString = function() { return "Transcript" }; Transcript.items = [];
Transcript.show = function(s) {
alert(s);
this.items.push(s);
};

// Usage
Target.p = 3;
Target.m2() // evaluates to 6
Target.print() // "p = 3" is displayed

Listing 1: Definition of an example object

var orgM1 = Target.m1;
var orgM2 = Target.m2;
Target.m1 = function(a) {
Transcript.show("m1 " + a);
return orgM1.apply(this, arguments)
};
Target.m2 = function() {
Transcript.show("m2");
return orgM2.apply(this, arguments)
};

Listing 2: Wrapping methods around other methods in JavaScript. This allows adding behavior without changing the original source.

and m2 can be accomplished without changing the source code. The original functions of that method are wrapped in a function that executes new behavior (Line 3) before evaluating the original function stored in orgM1. Abstracting this example further leads to the method trace as shown in Listing 3.

The method trace allows to trace arbitrary methods invocations in objects shown in Listing 4. The tracer works but includes some non-obvious problems. First, instrumenting method m1 twice results in capturing that method m1 two times in the Transcript. The tracer has no knowledge that it has instrumented that method already.

Furthermore, the Tracer might accidentally trace itself indirectly. Listing 5 shows how the method show of the Transcript object is instrumented. But the trace code itself makes use of the Transcript (see Line 6 in Figure 3). This leads to an unexpected endless recursion when the Tracer calls the Transcript which then calls the Tracer and so on.

The same problems occur when the tracer is further generalized to automatically trace all methods of an object. The traceObject method in Listing 6 implements this exemplary. Multiple objects with a common prototype may be traced at once by installing tracers in a prototype object. Since classes are implemented this way in JavaScript, discussing the tracing of objects is sufficient. The toString method is instrumented by the traceObject method, when as shown in Listing 6 each method of that object is wrapped with trac-

Listing 3: Definition of a simple tracer in JavaScript

Listing 4: Trace individual methods. (B) Transcript of calling m2 (C) after accidentally instrumenting m1 and m2 twice

Listing 5: Error when the "Transcript" should be traced
Listing 6: Usage of JavaScript introspection to trace all non-inherited methods of an object.

```javascript
for (var name in object) {
    if (object.hasOwnProperty(name) &&
        typeof object[name] == "function") {
        alert("trace " + name)
    }
}
Tracer.traceObject = function(object) {
    this.trace(object, name)

    if (object.hasOwnProperty(name) &&
        typeof object[name] == "function") {
        alert("trace " + name)
    }
}
Tracer.traceObject(Target)
```

3. CONTEXT-ORIENTED PROGRAMMING

Context-oriented programming [3, 5] (COP) extends object-oriented programming by providing dedicated language abstractions for defining and composing variations to basic program behavior. Behavioral variations are encapsulated by layers, modules that can crosscut classes or in the case of JavaScript objects. Layers can be dynamically de-/activated—and composed with other layers—for the dynamic extent of a code block. This mechanism allows for scoping behavioral variations to specific control flows.

ContextJS is a JavaScript language extension for Context-oriented programming. It is implemented as a library and allows for defining behavioral variations for JavaScript objects, that can be (de-)activated depending on context. ContextJS supports various approaches for scoping behavioral adaptations such as global, dynamic, instance-specific and structural scoping [10]. ContextJS changes only the execution of adapted methods in a system. This means objects and methods that no layer refines run totally unaffected.

Listing 7 shows how:

(A) a TranscriptLayer can be created.
(B) the behavior of the object Target is refined
(C) the behavior variation is activated for the execution of a code block
(D) the behavior variation is deactivated for the execution of a code block

The layer (de-)activations follows a stack like discipline [3]. Inner method deactivations cancel out outer layer activations and vice versa. Furthermore the whole layer composition can be customized by the objects themselves in ContextJS. We discussed the open implementation of such layer activation strategies like the scoping of behavioral variations to specific object structures in a previous paper [10].

```
TRacer.traceObject = function(object) {
    for (var name in object) {
        if (object.hasOwnProperty(name) &&
            typeof object[name] == "function") {
            alert("trace " + name)
        }
    }
}
```

Listing 7: ContextJS Syntax Example: A TranscriptLayer logs the method execution of m1 and m2

```
TranscriptLayer.refineObject(Target, {
    m1: function(a) {
        Transcript.show("m1 " + a)
    },
    m2: function(a) {
        Transcript.show("m2")
    }

    return cop.proceed()
});
```

Listing 8: Implementing the trace function with ContextJS

```
Tracer.trace = function(object, methodName) {
    var layeredMethodDef = {};
    layeredMethodDef[methodName] = function() {
        var args = arguments;
        var obj = this;
        TranscriptLayer.refineObject(object, layeredMethodDef);
    }
    return cop.proceed.apply(this, args)
};
```

4. CONTEXTJS TRACER

ContextJS can (de-)activate behavior variations for specific control flows. We can define tracing behavior in a layer and deactivate it when processing the method invocation in the Tracer itself. The definition of tracing as a separated concern is similar to the plain JavaScript version in Listing 3. Listing 8 shows how the trace method generates a function and stores it as partial method in the TranscriptLayer. The cop.proceed.apply(this, args) statement proceeds to the next layer (in most cases the original implementation) if there are no other active layers.

When using ContextJS-based trace, the instrumentation expression Tracer.trace(Target, "m1") can be called as often as needed. Since ContextJS handles method wrapping, the tracing code cannot accidentally be installed multiple times.

As a result of representing the tracing concern as a layer the tracing can be activated as needed as shown in Listing 9.

The second problem described in Section 2 is that the original tracer can accidentally use methods that are traced themselves. This can be taken care of by putting the instrumentation code inside a withoutLayers statement as shown
Tracer.traceObject(Target)

Target.m2() // without tracing

withLayers([TraceLayer], function()

Target.m2() // with active tracing

Listing 9: Installing the tracer and calling m2 with and without activated TraceLayer

Tracer.trace(Transcript, "show")

withLayers([TraceLayer], function()

Target.print()

// = Transcript =
// called Target's toString
// called Transcript's show
// Target, p = 3

Listing 10: Tracing the Transcript without running into an endless recursion

in Line 7 of Listing 8. This deactivates the TraceLayer in the dynamic extend of the Tracer.log execution, so the tracing code does not call itself any more. Listing 10 shows how Transcript's show method can now be instrumented without running into the endless recursion.

5. EXAMPLES

This section shows two example applications of the ContextJS tracer. The examples are taken from our work with the Web-based development environment Lively Kernel [7,9,11]. During our development we employ ContextJS on a regular basis and use the project as a source and testbed for new application scenarios of COP.

5.1 Finding a Font Size Bug in Lively Kernel

The first example demonstrates how customized tracing can be used to find bugs that are not easily debuggable with standard tools. The issue we wanted to debug was changing the font size of lists resulted in unexpected big list items as shown in Figure 1. To figure out the reason for the unexpected behavior we had to debug the Lively Kernel list abstractions. With the help of the customized tracer in Listing 11 we found out where the extent of the items changed and that the padding property was responsible as shown in Figure 2.

Using a debugger to step through the code would have been possible, however, using it to understand such complex behavior can be tedious. Traditional debuggers force the user to decide which methods to step over and which to step into. Without having an intimate knowledge of the code to debug this approach can require to restart time consuming debugging sessions several times. Using our customized tracing shown in Listing 11, we could declaratively adjust and refine the subject and location for what we were looking for. Repeating a run with the failing example was effortless because no manual user interaction was required.

Since the development environment Lively Kernel is self-sufficient it is crucial to separate tool logic from the source code under development. In the example above, TextMorph objects were instrumented. Since these objects were also used in the development environment, the ability to scope tracing to the actual execution of test snippets was important.

5.2 Understanding Layout Behavior

The second example shows how tracing was employed for understanding and profiling a new layout algorithm. We used tracing to visualize how graphical objects (Morphs) interact in the layouting process. We were only interested in methods of the class Morph, ignoring for example all methods in classes like Point, Rectangle, Array or String. The customized tracing and profiling did not only show only those methods, it further colored the lines in the trace so that morphs could be identified by their color as shown in Figure 3. Using a profiler in a meta level like the WebKit JavaScript tool suite, the profiling tools cannot be

Figure 1: Reproduction of setFontSize Bug in a Lively Kernel Web-based development environment. The list items got very big with a 14pt font compared with 12pt.
Figure 2: Dynamic analysis and source code of the setFontSize bug in Figure 1. The interesting places are highlighted by the developer in the debugging process for documenting the issue.

Figure 3: Tracing the method setExtent of Morphs (graphical objects) in the Lively Kernel. The tracing code is customized to produce colored text that matches color of the graphical object.

customized at runtime. Such profiles often show an overwhelming amount of information and developers cannot easily adapt them to mitigate those problems.

By using context specific scoping, tools can be separated from the domain objects like in external tool suites but can still be directly customizable.

6. RELATED WORK

A practical example how meta programming can be used in Web-based development environments [7] is a profiler implemented by wrapping JavaScript methods. The profiler is a Web-browser independent tool for analyzing JavaScript performance. Different to our approach, the presented profiler does not prevent multiple instrumentations and endless recursions when instrumenting code used in the profiler. But since in their approach the profiler code is fixed and cannot be customized, the profiler cannot accidentally instrument itself.

Tracing is a typical use case for Aspect-oriented Programming (AOP) [8]. AOP allows to declare pointcuts and write advice code that is executed in multiple places, which enables AOP to express homogenous crosscutting concerns. COP does not have this functionality, but in our approach we showed that this can be mitigated by using meta-programming to refine methods in several pages.

In the reverse engineering tool ARE [4] AspectJ is used to instrument method calls. These calls are later filtered out by a pluggable system. Programs are instrumented by linking the tool as a library into the program under development. Different to our approach ARE requires to instrument every method call because the scope of the tracing advice cannot be changed at runtime.

There are various approaches to deal with advice code triggering its own pointcuts directly or indirectly. In AspectJ the cflow pointcut designator allows for (de-)activating an aspect in a specific control. Another approach to avoid the endless regression caused by aspects directly or indirectly calling themselves, are execution levels for AOP [14]. In this approach, the problem of conflation [2] of base and advice code is solved by executing them in different levels. By default aspects only observe the execution of specific levels, thus avoiding endless regressions.

These execution levels are also implemented in As-
pectScript [15], an AOP implementation for JavaScript. As-
pectScript aims for expressibility and therefore has to rewrite
all JavaScript code, which makes the code several times slower even if no aspects are active. ContextJS does only instru-
ment methods that are refined by layers, which lets JavaScript code that is not refined by any layer run at full
speed.
Frameworks like Valgrind [13] and Pin [12] allow to in-
strument programs at runtime. This allows developers to
build customized dynamic analysis tools. These frameworks
instrument programs at the machine level and not at the
object level. These frameworks avoid reentrance and recur-
sion problems that might be caused from tools calling code
which is under instrumentation by executing the analysis
tools with a separate set of libraries [12].

7. CONCLUSION
In this paper we showed how a customized method tracer
can be implemented in a dynamic language like JavaScript.
As contributions we showed how COP technology can be
used to scope the tracing so that the tracer does not acci-
cidentally trace its own custom code. We further showed
how simple meta programming can be used to express ho-

genous concerns in our ContextJS COP implementation.
In two examples from our work with the Lively Kernel, we
showed how customizable tracers can be a valuable tool for

program comprehension, debugging, and profiling.
Tracing is an excellent use case for the withoutLayers con-
struct, which provides clean and reliable mechanisms to sepa-
rate instrumented code from the instrumentation itself. In
environments where the target system and the instrumen-
tation code are strongly separated this is not so much an
issue. In self-supporting-systems [6] like the Lively Kernel,
it is very important that tools like tracers are self aware
so that they do not measure themselves or produce endless
regressions. Putting tracing code into a layer that can be
(de-)activated depending on the context solves this problem
elegantly.

Tracing is a typical case of homogeneous cross-cutting con-
cerns, which normally cannot be expressed by COP. By us-
ing standard JavaScript meta programming this could be
overcome. COP implementations in less dynamic environ-
ments might not be used in this way.

In future work we will analyze how other cross-cutting
concerns like an Undo history in Lively Kernel are easy to
implement with COP.

8. REFERENCES
Structure of Crosscutting Concerns: Using Aspects or
Collaborations. In GPCE Workshop on
Aspect-Oriented Product Line Engineering (AOPLE),
2006.
confusion in metacircularity: The meta-helix. In
K. Futatsugi and S. Matsuoka, editors, Object
Technologies for Advanced Software, volume 1049 of
Lecture Notes in Computer Science, pages 157–172.
Springer Berlin / Heidelberg, 1996.
for Context-oriented Programming: An Overview of
ContextL. In DLS ’05: Proceedings of the 2005
symposium on Dynamic languages, pages 1–10, New
York, NY, USA, 2005. ACM.
Data Analysis with Aspect-Oriented Programming.
European Conference on Software Maintenance and
Systems, First Workshop, S3 2008, Potsdam,
Germany, May 15-16, 2008. Revised Selected Papers,
volume 5146 of Lecture Notes in Computer Science.
[7] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and
T. Mikkonen. The Lively Kernel A Self-Supporting
System on a Web Page. In R. Hirschfeld and K. Rose,
editors, S3 2008, LNCS 5146. Springer-Verlag Berlin
[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda,
Aspect-oriented Programming. In Eoocp 1997,
Proceedings 11th European Conference on
Object-Oriented Programming, volume 1241, pages
[9] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and
K. Palacz. Lively Wiki A Development Environment
for Creating and Sharing Active Web Content. In
[10] J. Lincke, M. Appeltauer, B. Steinert, and
R. Hirschfeld. An Open Implementation for
Context-oriented Layer Composition in ContextJS.
Science of Computer Programming, In Press,
Corrected Proof, 2011.
Lively Fabrik - A Web-based End-user Programming
Environment. In Proceedings of the Conference on
Creating, Connecting and Collaborating through
Computing (C5) 2009, Tokyo, Japan, 2009. IEEE.
G. Lowney, S. Wallace, V. J. Reddi, and
K. Hazelwood. Pin: building customized program
analysis tools with dynamic instrumentation. In
Proceedings of the 2005 ACM SIGPLAN conference
on Programming language design and implementation,
PLDI ’05, pages 190–200, New York, NY, USA, 2005.
ACM.
supervision framework. Electronic Notes in Theoretical
Computer Science, 89(2):44 – 66, 2003. RV ’03,
Run-time Verification (Satellite Workshop of CAV
’03).
Programming. In Proceedings of the 9th International
Conference on Aspect-Oriented Software Development,
ACM.
Expressive Aspects for the Web. In Proceedings of the
9th International Conference on Aspect-Oriented
Software Development, AOSD ’10, pages 13–24, New
York, NY, USA, 2010. ACM.