Agile Code Profiling Visualization

Alexandre Bergel

DCC, University of Chile, Santiago, Chile
http://bergel.eu

Abstract. This demonstration presents two code profiler tools. The first one is a concrete nominal type extractor which operates on unit tests. The second tool is a time profiler that offers a number of synthetic and expressive source code visualizations to easily identify execution bottlenecks. These two tools have been successfully employed to identify bugs, anomalies and bottlenecks in the Mondrian visualization engine.
These two tools are instantiations of Spy, a dedicated code profiling framework. Spy is briefly described and illustrated.
All this work has been implemented in the Pharo Smalltalk programming language and is available under the MIT license.

Even though computing resources are abundant, execution optimization and analysis through code profiling remains an important software development activity. Program profilers are crucial tools to identify execution bottlenecks and uncover interaction. Today, it is inconceivable to ship a programming environment without a code profiler included or provided by a third party.

When we retrospectively look at the history of code profiler tools, we see that tool usability and profiling overhead reduction have steadily improved, but that the set of offered abstractions has remained constant. For instance, gprof, which appeared in 1982, offers a number of textual output focussed on "how much time was spent executing directly in each function" and call graphs[1]. JProfiler essentially proposes the same output, using a graphical rendering instead of a textual one[2]. Most of the research conducted in the field of code profiling focus on reducing the overhead triggered by the code instrumentation and observation. On the other hand, the abstractions used to profile object-oriented applications are very close to the ones for procedural applications.

This demo proposal summarizes the result of two complementary research efforts. Implementation prototypes are first described from a user point of view. A succinct presentation of the Spy framework concludes this proposal.

Extracting types from unit tests. We propose a mechanism for extracting type information from the execution of unit tests. For a given program written in a dynamically typed language, we can deduce the type information from executing

[1] http://sourceware.org/binutils/docs/gprof/Output.html#Output
[2] http://www.ej-technologies.com/products/jprofiler/screenshots.html

the associated unit tests. The idea is summarized as follows: (i) we instrument an application to record runtime information; (ii) we run the unit tests associated to the application; (iii) we deduce the type information from what has been collected. Method signatures of the base program are then determined by the values provided to and returned by method calls while the tests are being executed.

As a concrete use case, we apply the extracted type information to find software faults. Type information combined with a test coverage helps identifying methods that were not invoked with all possible parameter types. By covering these missing cases, we identified and fixed a number of bugs and anomalies in Mondrian[3], a data visualization software.

Time profiling blueprints. Time profiling blueprints are graphical representations meant to help programmers (i) assess the time distribution and (ii) identify bottlenecks and give hints on how to remove them for a given program execution. The essence of profiling blueprints is to enable a better comparison of elements constituting the program structure and behavior. To render information, these blueprints use a graph metaphor, composed of nodes and edges.

The size of a node hints at its importance in the execution. In the case that nodes represent methods, a large node may say that the program execution spends "a lot of time" in this method. The expression "a lot of time" is then quantified by visually comparing the height and/or the width of the node against other nodes.

Color is used to either transmit a boolean property (*e.g.,* a gray node represents a method that always returns the same object value) or a metric (*e.g.,* a color gradient is mapped to the number of times a method has been invoked).

We propose two blueprints that help identify opportunities for code optimization. They provide hints to programmers to refactor their program along the following two principles: (i) make often-used methods faster and (ii) call slow methods less often. The metrics we adopted in this work help finding methods that are either unlikely to perform a side effect or return always the same result, good candidates for simple caching optimizations.

Figure 1 is an example of Structural Blueprint, one of the two visualizations we propose. Enclosing nodes are classes. A class and its superclass are linked each other with a line. A tree layout order the nodes by locating a superclass above its subclasses. Each class contains methods. Methods are rendered using three metrics: number of executions (width), total execution time (height), number of different object receivers. The figure shows that the method applyLayout, defined in the class MOGraphElement plays a major role in the total execution time for the code being profiled.

Our work on time profiling is described in a larger fashion in TOOLS'10 proceedings [BRB10]. TOOLS'10 is an event collocated with DYLA'10.

[3] `http://moose.unibe.ch/tools/mondrian`

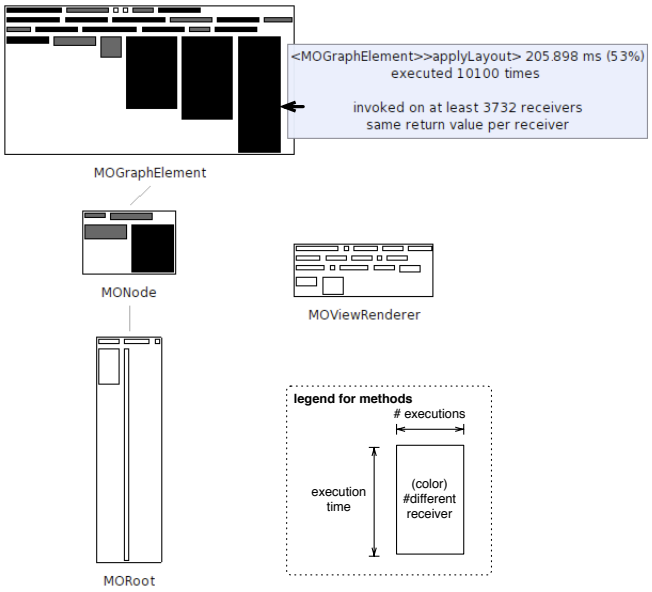

Fig. 1. Example of a structural time profiling blueprint

The Spy Framework. The tools presented above requires a support for runtime introspection. Unfortunately, not many libraries are available in the Smalltalk World. AspectS [Hir03] and Reflectivity [Den08] are two popular frameworks, however they are not available on Pharo, the Smalltalk dialect we adopted as the development platform.

Spy is a framework freely available[4] that offers the ability of model crosscutting concerns, in a fashion similar to what AspectJ [5] provide.

Spy is a framework designed to profile applications[6]. Profiling output is structured along the static structure of the analyzed program composed of packages, classes and methods. The core of Spy is composed of four classes, PackageSpy, ClassSpy, MethodSpy. The 3 first classes contains profiling information for packages, classes and methods. Profiling information may be stored and used later on for comparison.

We quality Spy as agile since it enables fast prototyping of profilers. By defining as much as 3 methods and 3 classes, profiling information may be visualized and inspected.

[4] `http://www.squeaksource.com/Spy.html`
[5] `http://eclipse.org/aspectj/`
[6] It is freely available on `http://www.squeaksource.com/Spy.html`

All the ideas presented in this demonstration were validated using the Pharo programming language[7], a Smalltalk like dynamically typed programming language. Our profiling tools are not particularly tied to the dynamicity of the language. Realizing this work in such a language however facilitates building and scripting our profilers while the profiled application is being executed.

References

[BRB10] Alexandre Bergel, Romain Robbes, and Walter Binder. Visualizing dynamic metrics with profiling blueprints. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'10)*. LNCS Springer Verlag, July 2010. to appear.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[Hir03] Robert Hirschfeld. AspectS — aspect-oriented programming with squeak. In *Proceedings NODe 2002*, volume 2591 of *LNCS*, pages 216–232. Springer-Verlag, 2003.

[7] `http://www.pharo-project.org`