Interactive Code Execution Profiling

Alexandre Bergel

PLEIAD Lab, Department of Computer Science (DCC), University of Chile
http://bergel.eu

1. INTRODUCTION

Programming environments have significantly improved over the last decade. Whereas abilities to efficiently edit and manage code source are now considered the minimum a modern IDE should provide, understanding and tracing program execution are still largely under-considered.

Code execution profilers are tools to extract information from a program execution and have a wide range of applications (*e.g.,* identifying execution hotspots[1], identifying test coverage[2], extracting program invariant[3]).

This paper sketches a demonstration of Spy [1], the code execution profiling framework we have designed for the Pharo programming language. Spy offers efficient means to easily record particular information about a program execution. A profile may then be graphically rendered using the Roassal visualization engine[4]. Such a visual representation of the program execution may exhibit relevant patterns indicating opportunities for improvement.

A video summarizing this demonstration is available online: `http://bit.ly/LiveWorkshop`.

2. DEMO OUTLINE

The duration of the Spy framework demonstration is approximately 15 minutes and consists in incrementally building a code profiler to obtain a test coverage tool. The demonstration is divided into two parts: enriching a minimal profiler and analyzing test coverage.

Part 1 - Creating a Minimal Profiler. The demonstration begins by creating a minimal code execution profiler. The profiler simply keeps track for each method of the profiled application the amount of times each method is executed. The profiler code is:

```
MyProfilerMethod>>beforeRun: name with: args in: receiver
  nbOfExecutions := nbOfExecutions + 1.
```

The method `beforeRun:with:in:` is similar to a before advice using the Aspect-Oriented Programming terminology. In our case, a counter is associated to each method of the base application. At each execution of a method, this counter is incremented.

A profile obtained from Spy is structured in terms of classes and methods (Figure 1): each large box is a class; edges represent inheritance: a superclass is located above its subclasses; inner boxes are methods.

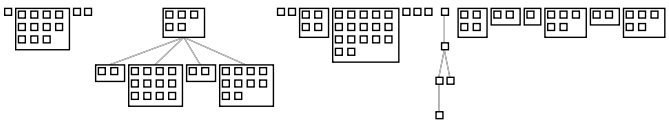

Figure 1: Default profile visualization

The default visualization of a profile uses a white small square to represent a method. Figure 1 is produced by the method `visualizeOn:`:

```
MyProfiler>>visualizeOn: view
  view nodes: self allClasses forEach: [ :each |
    view nodes: each methods.
    view gridLayout ].
  view edgesFrom: #superclass.
  view treeLayout
```

We refine the visualization to reflect the method counters and the structure of the application. We set all the test classes and test methods with a green border. We also relate the size of a method with the number of times the method is executed. The new definition of `visualizeOn:` is (**bold** indicates added code):

```
MyProfiler>>visualizeOn: view
  view shape rectangle
    if: #isTestClass borderColor: Color green.
  view nodes: self allClasses forEach: [ :each |
    view shape rectangle
      if: [ :m | m selector beginsWith: 'test' ]
     borderColor: Color green;
      size: [ :m | (m nbOfExecutions + 1) log * 8] .
    view nodes: each methods.
    view gridLayout ].
  view edgesFrom: #superclass.
  view treeLayout
```

[1] `http://www.ej-technologies.com`
[2] `http://emma.sourceforge.net`
[3] `http://groups.csail.mit.edu/pag/daikon`
[4] `http://bit.ly/roassal`

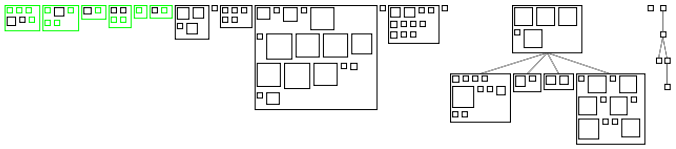

Figure 2: Augmented visualization

This profiler is used as a testbed for explorative experimentations. Figure 2 shows methods that are executed many times during the test execution.

Part 2 - Test Coverage. Test coverage is about relating test suites with the application base code. It is an essential tool to understand how well the base code is covered by the tests. Instead of relying on a textual list of covered and uncovered methods as most test coverage tools do (e.g., Emmaf[5], NCover[6]), we will use a visual support to indicate the coverage.

The profiler presented in the previous phase is incrementally augmented with new information extracting capabilities, such as dependencies between methods:

```
MyProfilerMethod>>beforeRun: name with: args in: receiver
  | v |
  nbOfExecutions := nbOfExecutions + 1.
  v := self getSpyOf: self callingMethod.
  self addIncomingCalls: v.
  v addOutgoingCalls: self
```

Methods are then positioned using a tree layout and colored to indicate non-covered methods. Methods that not executed are red. The visualization is then defined as:

```
MyProfiler>>visualizeOn: view
  ...
  view nodes: self allClasses forEach: [ :each |
    ...
    view nodes: each methods.
    view edges: each methods from:  #yourself toAll:
    #outgoingCalls.
    view treeLayout. ].
  ...
```

An example of a profile is given in Figure 3.

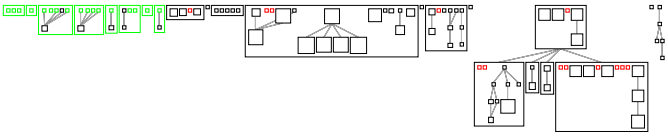

Figure 3: Augmented visualization

Visual representation may let some patterns emerge. Consider Figure 4, obtained from a large class.

The figure shows the importance of some methods in the call graph. It also indicates a large proportion of non covered methods. Contextual menus and tooltip (not shown in the figures) details which methods are visualized.

[5] http://emma.sourceforge.net/coverage_sample_a/index.html

[6] http://ncover.sourceforge.net/sample-output/NCover-report.html

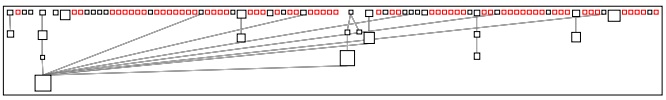

Figure 4: Augmented visualization

Technologies used. This demonstration is realized in Pharo, using the Spy and Roassal framework. Although these platforms will be used in the demo, the amount of actual code and specific techniques is minimal, meaning that no requirement is necessary from the audience.

3. VISION BEHIND SPY

Relating source code to an actual program execution is an essential step which remains tremendously difficult. Retrospectively looking at the evolution of program environments, it is striking to see that code execution profilers and debuggers have little evolved.

Spy is about easily scripting profilers and debuggers for punctual needs. Scripts may use Roassal to visually render a profile. Thanks to the Roassal domain specific language, interactive visualizations may be defined in an incremental fashion.

This demonstration present the steps we have taken when we designed Hapao[7], a full-fledged test coverage tool.

4. REFERENCES

[1] A. Bergel, F. Bañados, R. Robbes, D. Röthlisberger, Spy: A flexible code profiling framework, in: Smalltalks 2010, 2010.

[7] http://objectprofile.com/#/pages/products/hapao/overview.html