SPY: A Flexible Code Profiling Framework

Alexandre Bergel, Felipe Bañados, Romain Robbes
DCC, University of Chile
Santiago, Chile

David Röthlisberger
University of Bern
Switzerland

www.bergel.eu
www.dcc.uchile.cl/∼fbanados
www.dcc.uchile.cl/∼rrobbes
www.droethlisberger.ch

Abstract

Code profiling is an essential activity to increase software quality. It is commonly employed in a wide variety of tasks, such as supporting program comprehension, determining execution bottlenecks, and assessing code coverage by unit tests.

SPY is an innovative framework to easily build profilers and visualize profiling information. The profiling information is obtained by inserting dedicated code before or after method execution. The gathered profiling information is structured in line with the application structure in terms of packages, classes, and methods. SPY has been instantiated on four occasions so far. We created profilers dedicated to test coverage, time execution, type feedback, and profiling evolution across version. We also integrated SPY in the Pharo IDE.

SPY has been implemented in the Pharo Smalltalk programming language and is available under the MIT license.

Keywords: Smalltalk, profiling, visualization

1. Introduction

Profiling an application commonly refers to obtaining dynamic information from a controlled program execution. Common usages of profiling techniques include test coverage [1], time execution monitoring [2], type feedback [3, 4, 5], or program comprehension [6, 7]. The analysis of gathered runtime information provides important hints on how to improve the program execution. Runtime information is usually presented as numerical measurements, such as number of method invocations or number of objects created in a method, making them easily comparable from one program execution to another.

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 method call graphs.

Most professional programming environments include a code profiler. Pharo Smalltalk and Eclipse, for instance, both ship a profiler [8, 9].

A number of code profilers are necessary to address the different facets of software quality [10]: method execution time and call graph, test coverage, tracking nil values, just to name a few. Providing a common platform for runtime analysis has not yet been part of a joint community effort. Each code profiler tool traditionally comes with its own engineering effort to both acquire runtime information and properly present this information to the user.

Most Smalltalk systems offer a flexible and advanced programming environment. Over the years different Smalltalk communities have been able to propose tools such as the system browser, the inspector or the debugger. These tools are the result of a community effort to produce better software engineering techniques and methodologies. However, code profilers have little evolved over the years, becoming more an outdated Smalltalk heritage than a spike for innovation. A survey of several Smalltalk implementations—Squeak [11], Pharo [8], VisualWorks [12], and GemStone—reveals that none shines for its execution profiling capabilities: indented textual output holds a royal position (see Section 2).

In the Java world, JProfiler[1] is an effective runtime execution profiler tool that, besides measuring method execution time, also offers numerous features including snapshot comparisons, saving a profiling trace in an XML file and estimating method call graphs. Cobertura[2] is a tool dedicated to measure test coverage. Similarly to JProfiler, test coverage information may be stored in an XML file which contains method call graph analysis and coverage. However, JProfiler and Cobertura do not share any library besides the standard Java libraries. There are multiple reasons why JProfiler and Cobertura are separated from each other even though both have to gather similar runtime information. One of them is certainly a lack of a common profiling framework.

This paper presents SPY, a framework for easily prototyping various types of code profilers in Smalltalk. The dynamic information returned by a profiler is structured along the static structure of the program, expressed in terms of packages[3], classes and methods. One principle of SPY is structural correspondence: the structure of meta-level facilities corresponds to the structure of the language manipulated[4]. Once gathered, the dynamic information can easily be graphically rendered using the Mondrian visualization engine [14][5].

SPY has been used to implement a number of code profilers. The SPY distribution offers a type feedback mechanism, an execution profiler [15], an execution evolution profiler, and a test coverage profiler. Creating a new profiler comes at a very light cost as SPY relieves the programmer from performing low-level monitoring.

To ease the description of the framework, SPY is presented in a tutorial like fashion: We document how we instantiated the framework in order to build a code coverage tool.

[1] http://www.ej-technologies.com/products/jprofiler/screenshots.html

[2] http://cobertura.sourceforge.net

[3] In Pharo, the language used for the experiment, a package is simply a group of classes.

[4] According to the terminology provided by Bracha and Ungar [13], ensuring structural correspondence makes SPY a mirror-based system.

[5] http://www.moosetechnology.org/tools/mondrian

The main contributions of this paper are summarized as follows:

- The presentation of a flexible and general code profiling framework.
- The construction of an expressive test coverage tool as an example of the framework's usage.
- The demonstration of the framework flexibility, via the description of three additional framework instantiation, and of its integration with Mondrian and Smalltalk code browsers.

The paper is structured as follows: first, a brief survey of Smalltalk profilers is provided (Section 2). The description of SPY (Section 3) begins with an enumeration of the different composing elements (Section 3.1) followed by an example (Section 3.2 – Section 3.6). The practical applicability of SPY is then demonstrated by means of three different situations (Section 4) before concluding (Section 5).

2. Current Profiler Implementations

This section surveys the profiling capabilities of the Smalltalk dialects and implementations commonly available.

Squeak. Profiling in Squeak[6] is achieved through the MessageTally class (MessageTally>> spyOn: aBlock). As most profilers, MessageTally employs a sampling technique, which means that a high-priority process regularly inspects the call stack of the process in which aBlock is evaluated. The time interval commonly employed is one millisecond.

MessageTally shows various profiling information. The method call graph triggered by the evaluation of the provided block is shown as a hierarchy which indicates how much time was spent, and where. Consider the expression MessageTally spyOn: [MOViewRendererTest buildSuite run]. It simply profiles the execution of the tests contained in the class MOViewRendererTest. The call graph is textually displayed as:

```
75.1% {10257ms} TestSuite>> run:
 75.1% {10257ms} MOViewRendererTest(TestCase)>> run:
  75.1% {10257ms} TestResult>> runCase:
   75.1% {10257ms} MOViewRendererTest(TestCase)>> runCase
   ...
```

This information is complemented by a list of leaf methods and memory statistics.

Pharo. Pharo is a fork of Squeak and its profiling capabilities are very close to those of Squeak. TimeProfiler is a graphical facade for MessageTally. It uses an expandable tree widget to comfortably show profiling information (Figure 1).

Gemstone. The class ProfMonitor allows developers to sample the methods that are executed in a given block of code and to estimate the percentage of total execution

[6]http://wiki.squeak.org/squeak/4210

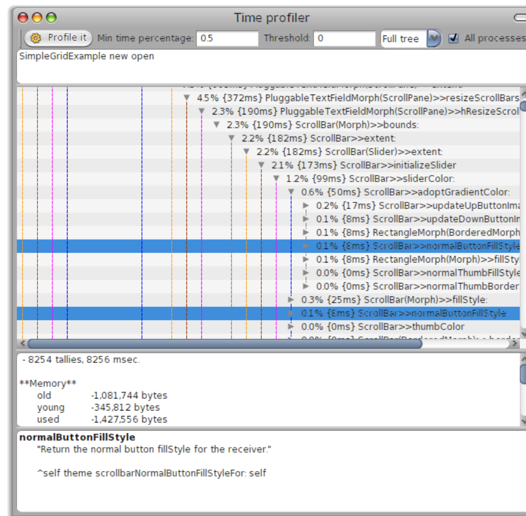

Figure 1: TimeProfiler in Pharo

time represented by each method[7]. It provides essentially the same ability as MessageTally in Squeak. One minor variation is offered: methods can be filtered from a report according to the number of times they were executed (ProfMonitor>> monitorBlock:downTo:interval:).

VisualWorks. The largest number of profiling tools are available in VisualWorks[8]. First, a *profiler window* offers a list of code templates to easily profile a Smalltalk block: profiling results may be directly displayed or stored in a file. Statistics may also be included.

VisualWorks uses sampling profiling. Repeating the code to be profiled, with timesRepeat: for example, increases the accuracy of the sampling. An additional mechanism to control accuracy is to graphically adjust the sampling size.

The profiling information obtained in VisualWorks is very similar to MessageTally's. It is textually rendered, indentations indicate invocations in a call graph, and execution times are provided in percentage and milliseconds. Methods may be filtered out based on their computation time. Similarly to TimeProfiler, branches of the call tree may be contracted and expanded.

Conclusion. The Smalltalk code profilers available are very similar. They provide a textual list of methods annotated with their corresponding execution time share. None

[7]Page 301 in http://www.gemstone.com/docs/GemStoneS/GemStone64Bit/2.4.3/GS64-ProgGuide-2.4.pdf

[8]Page 87 in http://www.cincomsmalltalk.com/documentation/current/ToolGuide.pdf

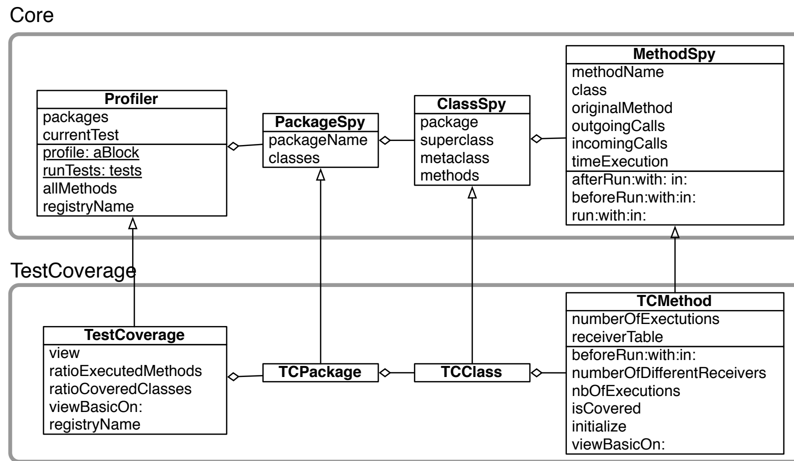

Figure 2: Structure of SPY

of these profilers is easily extensible to obtain a different profiling such as test coverage. The SPY framework described in the following addresses particularly this issue.

3. The SPY Framework

3.1. SPY *in a nutshell*

The essential classes of SPY are depicted in Figure 2 and explained in the following:

- The Profiler class contains the features necessary for obtaining runtime information by profiling the execution of a block of Smalltalk code. Profiler offers a number of public class methods to interface with the profiling. The profile: aBlock inPackagesNamed: packageNames method accepts as first parameter a block and as second parameter a collection of package names. The effect of calling this method is to (i) instrument the specified packages; (ii) to execute the provided block; (iii) to uninstrument the targeted packages; and (iv) to return the collected data in the form of an instance of the Profiler class which contains instances of the classes described below, essentially mirroring the structure of the program.

  Profiles are globally accessible by other development tools. The method registryName has to be be overridden to return a symbol name. Other IDE tools can then easily access the profiling.

- PackageSpy contains the profiling data for a package. Each instance has a name and contains a set of class spies.

- ClassSpy describes a Smalltalk class. It has a name, a superclass spy, a metaclass spy and a set of method spies.

- MethodSpy describes a method. It has a selector name and belongs to a class spy. MethodSpy is central to SPY. It contains the hooks used to collect runtime information. Three methods are provided for that purpose: beforeRun:with:in: and afterRun:with:in: are executed before and after the corresponding Smalltalk method. These empty methods may be overridden in subclasses of MethodSpy to collect relevant dynamic information, as we will see in the following subsections. The run:with:in method simply calls beforeRun:with:in:, followed by the execution of the represented Smalltalk method, and ultimately calls afterRun:with:in:. The parameters passed to these methods are: the method name (as a symbol), the list of arguments, and the object that receives the intercepted message.

The SPY framework is instantiated by creating subclasses of PackageSpy, ClassSpy, MethodSpy and Profiler, all specialized to gather the precise runtime information that is needed for a particular system and task.

3.2. Instantiating SPY

Test coverage. We motivate and demonstrate the usage of the SPY framework by building a test coverage code analyzer. Identifying the coverage of the unit tests of an application may be considered as a code profiling activity. A simple profiling reveals the number of covered methods and classes. This is what traditional test coverage tools produce as output (e.g., Cobertura).

We go one step further with our test coverage tool running example. In addition to raw metrics such as percentage of covered methods and classes, we retrieve and correlate a variety of dynamic and static metrics:

- number of method executions – *how many times a particular method has been executed.*
- number of different object receivers – *on how many different objects a particular method has been executed.*
- number of lines of code – *how complex the method is.* We use the method code source length as a simple proxy for complexity.

The intuition behind our test coverage tool is to indicate what are the "complex" parts of a system that are "lightly" tested, and what are the "apparently simple" components that are "extensively" tested. There is clearly no magic metric that will precisely identify such a complex or simple software component. However, correlating a complexity metric (i.e., number of lines of code in our case) with how much a component has been tested (i.e., number of executions and number of different receivers) provides a good indication about the quality of the test coverage.

Instantiating SPY**.** The very first step to build our test coverage tool is to subclass the relevant classes. TestCoverage, TCPackage, TCClass, and TCMethod, respectively, subclass Profiler, PackageSpy, ClassSpy and MethodSpy.

```
Profiler subclass: #TestCoverage

PackageSpy subclass: #TCPackage

ClassSpy subclass: #TCClass

MethodSpy subclass: #TCMethod
  instanceVariableNames: 'numberOfExecutions receiverTable'
```

TCMethod defines two variables, numberOfExecutions and receiverTable. The former variable is initialized as 0 and is incremented for each method invocation. The latter keeps track of the number of receiver objects on which the method has been executed. Recording the hash value of each receiver object can be easily implemented to provide a good approximation of the number of receivers in most cases.

```
TCMethod >> initialize
  super initialize.
  numberOfExecutions := 0.
  receiverTable := BoundedSet maxSize: 100
```

The class BoundedSet is a subclass of Set in which the number of different values is no greater than a limit. In our case, no more than 100 different elements may be inserted in a bounded set. This value is actually arbitrary and depends very much on how the related metric will be used. In our environment, for the types of programs we write, given the resources we can expend, we have not been able to devise a way to efficiently and easily keep track of all receiver objects of a method call. Using an ordered collection in which we insert the object receiver at each invocation is not practically exploitable. There is a number of reasons for this. As soon as a method is called many times, e.g., one million times, then one million elements have been added to the collection. Allowing the ordered collection to grow up to one million elements significantly slows down the overall program execution. In addition to this, identifying the number of different elements in a list with one million elements is also slow. The same schema applies for all the recursively called methods.

The method beforeRun:with:in: is executed before the original method. We simply increment the execution counter, and record the receiver.

```
TCMethod>> beforeRun: selector with: args in: receiver
  numberOfExecutions := numberOfExecutions + 1.
  receiverTable at: receiver hash put: true.
```

A number of utility methods are then necessary:

```
TCMethod>> isCovered
  ^ numberOfExecutions > 0

TCMethod>> numberOfExecutions
  ^ numberOfExecutions

TCMethod>> numberOfDifferentReceivers
  ^ (receiverTable select: #notNil) size
```

The ratio of executed methods and covered classes are defined on TestCoverage:

```
TestCoverage>> ratioExecutedMethods
  ˆ ((self allMethods select: #isCovered) size /
        self allMethods size) asFloat

TestCoverage>> ratioCoveredClasses
  ˆ ((self allClasses
        select: [ :cls | cls methods anySatisfy: #isCovered ]) size /
        self allClasses size) asFloat
```

The method allClasses is defined on Profiler, the superclass of TestCoverage.

3.3. Running Spy

Our TestCoverage tool can be run using the profile:inPackagesNamed: class method. In this example, we run it on the test cases of the Mondrian visualization framework.

```
coverage :=
  TestCoverage
        profile: [ MOViewRendererTest buildSuite run ]
        inPackage: 'Mondrian'
```

Executing the code above returns an instance of TestCoverage.

3.4. Visualizing Runtime Information

The Mondrian visualization engine framework [14] easily produces visualizations. Mondrian is a visualization engine that offers a rich domain specific language to define graph-based rendering. Each element of a graph (i.e., node and edge) has a shape that defines its visual aspect. Nodes may be ordered using a layout. Consider the method:

```
TestCoverage>> viewBasicOn: view
  view nodes: self allClasses forEach: [ :each |
        view shape rectangle
          height: #numberOfLinesOfCode;
          width: [ :m | (m numberOfDifferentReceivers + 1) log * 10 ];
          linearFillColor:
               [ :m | ((m numberOfExecutions + 1) log * 10) asInteger ]
          within: self allMethods;
          borderColor:
               [:m | m isCovered
                    ifTrue: [ Color black ] ifFalse: [ Color red ] ].
        view interaction action: #inspect.
        view nodes: (each methods
                         sortedAs: #numberOfLinesOfCode).
        view gridLayout gapSize: 2.
  ].
  view edgesFrom: #superclass.
  view treeLayout
```

The visualization is rendered by evaluating:

```
coverage viewBasic
```

An excerpt of the visualization obtained is depicted in Figure 3. The displayed class hierarchy represents Mondrian shapes. The root is MOShape. The visualization has the following characteristics:

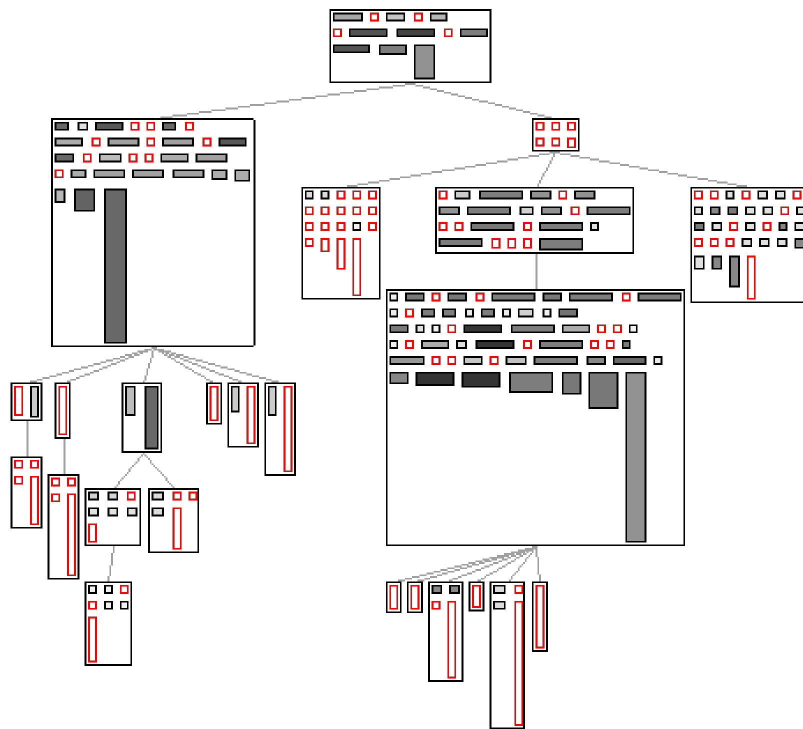

Figure 3: Test coverage visualization

- Outer boxes are classes.
- Edges between classes represent class inheritance relationships. A superclass appears above and a subclass below a particular class node. A tree layout is used to order classes which is adequate since Smalltalk uses single inheritance.
- Inner boxes are methods. Methods are sorted according to their source code length.
- White boxes with a red border are methods that have not been executed when running the coverage.
- The height of a method is the number of lines of code.
- The width of a method is the number of different receivers. We use a logarithmic scale to accommodate the variability of this metric.
- The color of a method is the number of method executions. We use a logarithmic scale also for this metric.

From what is depicted in Figure 3, a number of patterns can be visually identified:

- Some classes contain red methods only. This means that the class is absent from all the execution scenarios specified in the tests.
- Red methods that are tall and thin are long, untested methods. They are excellent targets for new test additions.
- Gray methods (few executions) and narrow methods (few receivers) are probably good candidates for further testing.
- Dark and large methods are extensively tested.
- Horizontally flat methods are very extensively tested more since they contain just a few lines of code and are still executed many times.

As it is the case for most software visualizations, the goal of our test coverage visualization is not to precisely locate software deficiency. Rather, it aims at assisting the programmer to identify candidates for software improvement. In this case, the visualization pinpoints red methods, and thin, gray methods, as likely candidates to consider in order to improve the coverage of the code by tests.

3.5. Call graph and execution time

Profiler defines an instance method getTimeAndCallGraph which simply returns false. By overriding this method in a subclass to make it return true, the execution time (in milliseconds and percentage) and the call graph for each method is computed during the block execution.

```
TestCoverage>> getTimeAndCallGraph
  "Each instance of TCMethod contains information about
   execution time and outgoing and incoming calls"
  ^ true
```

The call graph and execution time is estimated by regularly sampling the method call stack. For that very purpose, SPY contains a class called Sampling, which is a simplified version of MessageTally[9]. Each method spy will now store the execution time it took, as well as a list of outgoing calls and incoming calls.

By determining the method call graph from these incoming and outgoing calls, all packages involved during the block evaluation are easily identified. The profiling can now be realized using the profile: method. There is no need to provide a package name to extract the call graph of the execution.

```
coverage :=
  TestCoverage
        profile: [ MOViewRendererTest buildSuite run ]
```

Now that the method call graph is computed, we can add an entry point to a new visualization. The script defined in TestCoverage>> viewBasicOn: may be refined with a new menu item for methods:

```
        ...
        view interaction action: #inspect;
                item: 'view call graph' action: #viewBasic.
        view nodes: (each methods
                        sortedAs: #numberOfLinesOfCode).
        ...
```

For a user-selected method, the following script renders the method call graph, using the outgoingCalls method of MethodSpy:

```
TCMethod>> viewBasicOn: view
  | methods |
  methods := self withAllOutgoingCalls asSet.
  view shape rectangle
        height: #numberOfLinesOfCode;
        width: [:m | (m numberOfDifferentReceivers + 1) log * 10 ];
        linearFillColor: [ :m | ((m numberOfExecutions + 1) log * 10)
                                asInteger ]
        within: self package allMethods;
        borderColor: [ :m | m isCovered
                                ifTrue: [ Color black ]
                                ifFalse: [ Color red ] ].

  view nodes: methods.
  view shape arrowedLine width: 2.
  view edges: methods from: #yourself toAll: #outgoingCalls.
  view treeLayout
```

The visualization we provide may be enriched with information about the method execution time. Overriding the printOn: method will change the text that is displayed by Mondrian when hovering the mouse over a node.

[9] Sampling is not represented in Figure 2 since a user is not expected to use it directly.

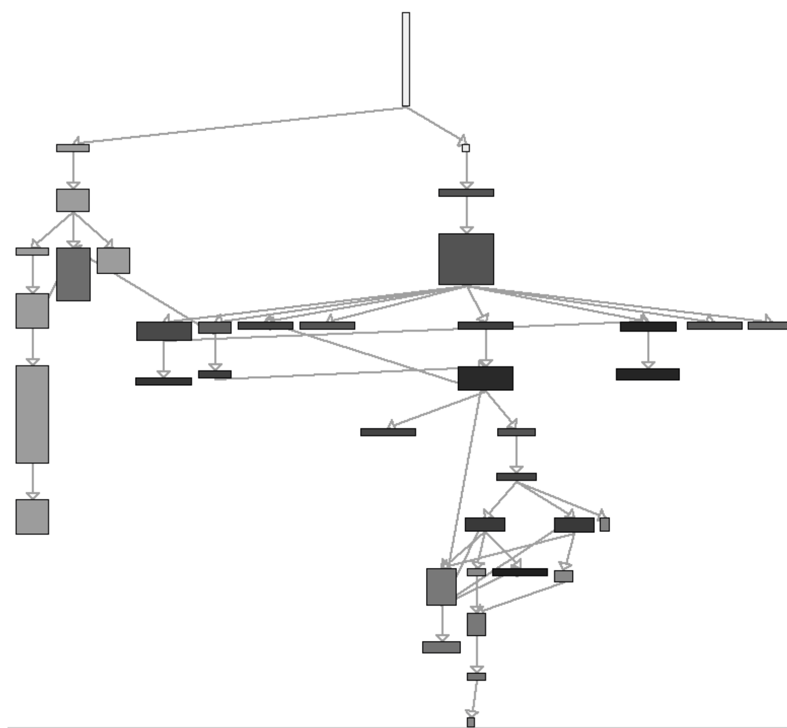

Figure 4: Call graph of the method MOViewRenderer>> testTranslation

```
TCMethod>> printOn: stream
   super printOn: stream.
   stream nextPutAll: self executionTime printString, ' ms'
```

By right-clicking on a method node, a menu item renders the call graph for the method (Figure 4). Methods are ordered from top to down. The arrowed edges represent the control flow between methods.

3.6. Summary

This section presented a simple application of SPY. It described the essential steps to create a code profiler: (i) recovering the required profiling information by instantiating the framework; (ii) visualizing this information with Mondrian; (iii) gathering further execution and call graph information; and (iv) visualizing this additional information.

Effective profiling visualizations may be produced using Mondrian. The fact that the profiling information follows the code structure leads to comprehensive and famil-

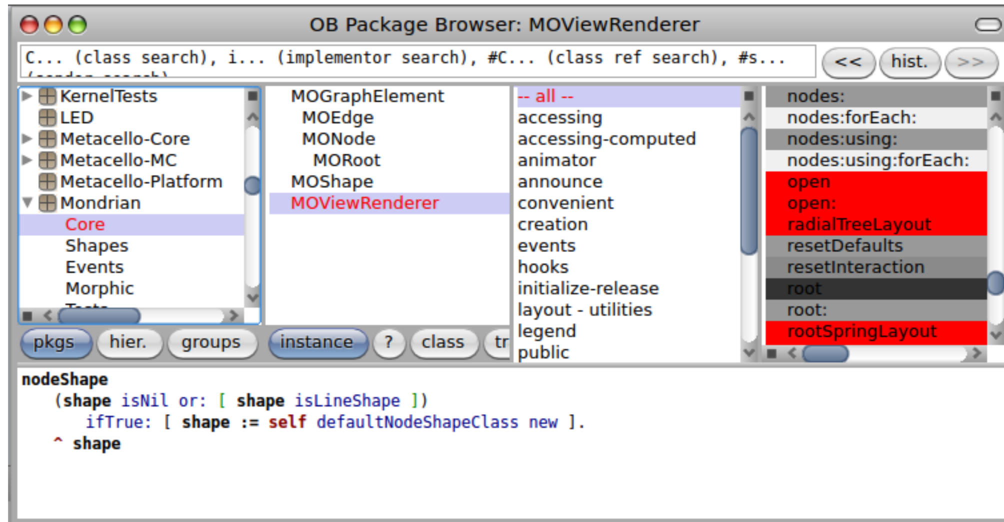

Figure 5: Integration of profiling information into the Pharo IDE

iar visualizations that are easy to implement as the profiling information's representation matches the one often used by Mondrian visualizations.

4. Applications

In this section, we present some of the profiling tools we built on top of SPY.

4.1. *Extracting types from unit tests*

As a first application of SPY, we proposed a mechanism for extracting type information from the execution of unit tests[10] [15]. For a given program written in Smalltalk, we can deduce the type information from executing the associated unit tests., as has been proposed by other researchers as well [16]. The idea is summarized as follows: (i) we instrument an application to record the runtime types of the arguments and return values of methods; (ii) we run the unit tests associated with the application; and (iii) we deduce the type information from what has been collected. The idea is to record the type of each message argument and return value to later deduce the most specialized types for each argument and return type. We refer to the most specialized type as the most direct supertype that is common for a set of classes. 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 exploit the extracted type information to find software faults. Type information combined with test coverage helps developers identifying

[10] http://www.moosetechnology.org/tools/Spy/Keri

methods that were not invoked with all possible type parameters. By covering these missing cases, we identified and fixed four anomalies in Mondrian.

4.2. Time profiling blueprints

As a second application, we proposed a time execution profiler[11]. Time profiling blueprints are graphical representations meant to help programmers (i) assess the program execution time distribution and (ii) identify and fix bottlenecks in 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 gives hints about its importance in the execution. When nodes represent methods, a large node means 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 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: the structural profiling blueprint visualizes the distribution of the CPU effort along the program structure and the behavioral profiling blueprint along the method call graph. These blueprints 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 paper help developers finding methods that are either unlikely to perform a side effect or always return the same result, good candidates for simple caching-based optimizations.

4.3. Profiling differentiation

The use of profiling information might be taken a step further by profiling different versions of an application. Spotting differences between them provides insights on the causes of slowdowns, and what should be improved next. Comparing, e.g., time profiling throughout a package's history allows one to confirm an optimization trial as an improvement and to find the potential bottlenecks that remain. The package Hip helps us in this task. Hip allows one to build a collection of history profiles, following a schema similar to the Hismo model [17]. Each method, class, and package profile can access the profiles of its previous and next version. Queries about metrics may be then formulated (e.g., *has a metric increased?*) as well as "differential measurements"[12] (e.g., *how much has a metric increased?*).

Hip provides facilities to automatically profile a block throughout a set of package versions available from a Monticello[13] repository by loading each version, profiling it, and adding the gathered profiling information to a Hip version collection structure.

[11] `http://www.moosetechnology.org/tools/Spy/Kai`

[12] This term is commonly employed in electronic and voltage measurement. We consider it to be descriptive in our context.

[13] Monticello is the version control mechanism commonly employed in Pharo.

Hip opens the door to a wide range of options to visualize the evolution of a program's runtime behavior. As an example, we propose a semaphore-like view that helps to identify bottlenecks. For a particular profiled object and version, Hip assigns one of five colors. In the case of a metric such as the execution time—where lower is better—source artifacts with a lower metric value compared to the previous version are colored green; those with a greater value red; unchanged artifacts are colored in white; removed ones black; and new ones yellow. The emphasis is on red and green artifacts for obvious reasons, and also on yellow artifacts, as from that version onward the developers should put focus on newly created artifacts, as they were not available before.

4.4. IDE integration

The primary tool developers use to develop and maintain software systems is the integrated development environment (IDE). For this reason we integrate profiling information gathered by SPY into Pharo's IDE which is implemented using the OmniBrowser framework [18]. As soon as a system's test suite has been executed with SPY, the IDE can access the test coverage information using the following statement:

```
Profiler profilerAt: #testCoverage
```

The Pharo IDE exploits the profiling information resulting from the execution of tests to highlight in the source code perspectives methods and classes that have been covered by the system's test suite. The same color scheme as introduced in Section 3.4 is used to highlight the source artifacts. A non-executed method is colored red to raise the awareness for untested code while methods colored dark (e.g., in a gradient from gray to black) have been executed often and are hence tested extensively. Gray methods, that is, methods that have not been executed often by the test suite, are good candidates to look at in detail in order to reveal whether they could benefit from more extensive testing. Visualizing profiling information directly in the IDE hence helps developers to easily locate methods that should be better covered with tests to improve a system's test coverage. Figure 5 illustrates how profiling information is visualized in the Pharo IDE.

5. Conclusion

SPY is a profiling framework for the Pharo Smalltalk environment designed to easily build application profilers. 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, Profiler, PackageSpy, ClassSpy and MethodSpy. These classes represent the profiler itself and profiling information for packages, classes and methods.

Once the data about a program's execution is gathered by SPY, one can explore the data by visualizing it using a dedicated visualization framework such as Mondrian.

However, SPY is not cost free. Mondrian tests are 3 times slower when the coverage is computed. Future effort of SPY will be dedicated to reducing information gathering

overhead based on bytecode transformation [19] and DTrace[14]. When method time execution matter, the user has always the option to rely on a second profiling "pass" triggered with the getTimeAndCallGraph option. The piece of code to profile is then executed a second time, using a sampling approach, less costly, but also less precise.

We have shown by a simple example how one can instantiate SPY for a given problem, such as building a code coverage tool. Furthermore, we have demonstrated the flexibility of SPY by presenting three additional applications we built on top of it, namely a type extraction profiler, a time profiling visualization tool, and an evolutionary time profiling visualization tool. Finally, we demonstrated that the information gathered via SPY is useful beyond visualization, as we integrated our code coverage profiler with the regular IDE, allowing a more direct interaction between the source code and its dynamic aspects.

Acknowledgment. We gracefully thank Dave Ungar for his comments and feedback of our paper.

References

[1] M. Marré, A. Bertolino, Reducing and estimating the cost of test coverage criteria, in: ICSE '96: Proceedings of the 18th international conference on Software engineering, IEEE Computer Society, Washington, DC, USA, 1996, pp. 486–494.

[2] W. Binder, Portable and accurate sampling profiling for java, Softw. Pract. Exper. 36 (6) (2006) 615–650.

[3] O. Agesen, U. Holzle, Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages, Tech. rep., Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA, USA (1995).

[4] M. Haupt, R. Hirschfeld, M. Denker, Type feedback for bytecode interpreters, in: Proceedings of the Second Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007), ECOOP Workshop, TU Berlin, 2007, pp. 17–22.
URL http://scg.unibe.ch/archive/papers/Haup07aPIC.pdf

[5] M. Arnold, Online profiling and feedback-directed optimization of java, Ph.D. thesis, Rutgers University (Oct. 2002).

[6] D. Röthlisberger, M. Denker, É. Tanter, Unanticipated partial behavioral reflection: Adapting applications at runtime, Journal of Computer Languages, Systems and Structures 34 (2-3) (2008) 46–65.

[14] http://www.adrian-lienhard.ch/blog?dialog=smalltak-meets-dtrace

[7] D. Holten, B. Cornelissen, J. J. van Wijk, Trace visualization using hierarchical edge bundles and massive sequence views, in: Proceedings of Visualizing Software for Understanding and Analysis, 2007 (VISSOFT'07), IEEE Computer Society, 2007, pp. 47 – 54.

[8] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.

[9] Eclipse, Eclipse platform: Technical overview, http://www.eclipse.org/white-papers/eclipse-overview.pdf (2003).

[10] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1997.

[11] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, a practical Smalltalk written in itself, in: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), ACM Press, 1997, pp. 318–326.

[12] VisualWorks, Cincom Smalltalk, http://www.cincomsmalltalk.com/, archived at http://www.webcitation.org/5p1rRxls5 (2010).

[13] G. Bracha, D. Ungar, Mirrors: design principles for meta-level facilities of object-oriented programming languages, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices, ACM Press, New York, NY, USA, 2004, pp. 331–344.

[14] M. Meyer, T. Gîrba, M. Lungu, Mondrian: An agile visualization framework, in: ACM Symposium on Software Visualization (SoftVis'06), ACM Press, New York, NY, USA, 2006, pp. 135–144.

[15] A. Bergel, R. Robbes, W. 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, 2010, to appear.

[16] D. Röthlisberger, O. Greevy, O. Nierstrasz, Exploiting runtime information in the IDE, in: Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008), IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp. 63–72.

[17] T. Gîrba, M. Lanza, S. Ducasse, Characterizing the evolution of class hierarchies, in: Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), IEEE Computer Society, Los Alamitos CA, 2005, pp. 2–11.

[18] A. Bergel, S. Ducasse, C. Putney, R. Wuyts, Creating sophisticated development tools with OmniBrowser, Journal of Computer Languages, Systems and Structures 34 (2-3) (2008) 109–129.

[19] M. Denker, S. Ducasse, É. Tanter, Runtime bytecode transformation for Smalltalk, Journal of Computer Languages, Systems and Structures 32 (2-3) (2006) 125–139.