Computer Languages, Systems & Structures 38 (2012) 16–28

Contents lists available at SciVerse ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Spy: A flexible code profiling framework

Alexandre Bergel [a,*], Felipe Bañados [a], Romain Robbes [a], David Röthlisberger [b]

[a] Pleiad Lab, DCC, University of Chile Santiago, Chile
[b] University of Bern, Switzerland

ARTICLE INFO

Keywords:
Smalltalk
Profiling
Visualization

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.

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 [3–5], 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. Most professional programming environments include a code profiler. Pharo Smalltalk and Eclipse,[1] for instance, both ship a profiler.

A number of code profilers tracking various kinds of dynamic information are necessary to address the different facets of software quality [8]: method execution time, dynamic method call graphs, test coverage, tracking nil values, 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 present this information to the user, resulting in duplicated effort.

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

* Corresponding author.
E-mail address: alexandre.bergel@me.com (A. Bergel).
URLS: http://www.bergel.eu (A. Bergel), http://www.dcc.uchile.cl/~fbanados (F. Bañados), http://www.dcc.uchile.cl/~rrobbes (R. Robbes), http://www.droethlisberger.ch (D. Röthlisberger).

[1] http://www.eclipse.org

1477-8424/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2011.10.002

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—the necessary effort to implement various profiling tools is not always invested. A survey of several Smalltalk implementations—Squeak [9], Pharo [10], VisualWorks [11], and GemStone—reveals that none shines for its execution profiling capabilities: indented textual output is the norm (see Section 2).

This paper presents SPY, a framework to easily prototype a variety 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,[2] classes and methods. One principle of SPY is structural correspondence: the structure of meta-level facilities corresponds to the structure of the language manipulated. Once gathered, the dynamic information can easily be graphically rendered using the Mondrian visualization engine [12].[3]

SPY has been used to implement a number of code profilers. The SPY distribution offers a type feedback mechanism, an execution profiler [13], an evolutionary execution 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. 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 validation of the framework's flexibility, via the description of three additional framework instantiations, 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 (Sections 3.2–3.7). Implementation is then presented (Section 4). The practical applicability of SPY is then demonstrated by three additional applications and an IDE integration (Section 5) before concluding (Section 6).

2. Current profiler implementations

This section surveys the profiling capabilities of the Smalltalk dialects and implementations commonly available and briefly looks at profiling facilities available for other languages such as Java.

Squeak: Profiling in Squeak[4] 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 the millisecond, which is rather coarse.

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 MessageTallyspyOn : [MOViewRendererTest buildSuite run]. It simply profiles the execution of the tests contained in the class MOViewRendererTest. The call tree 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 (Fig. 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 time represented by each method.[5] 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: 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.

[2] In Pharo, the language used for the experiment, a package is simply a group of classes.
[3] http://www.moosetechnology.org/tools/mondrian
[4] http://wiki.squeak.org/squeak/4210
[5] Page 301 in http://www.gemstone.com/docs/GemStoneS/GemStone64Bit/2.4.3/GS64-ProgGuide-2.4.pdf.

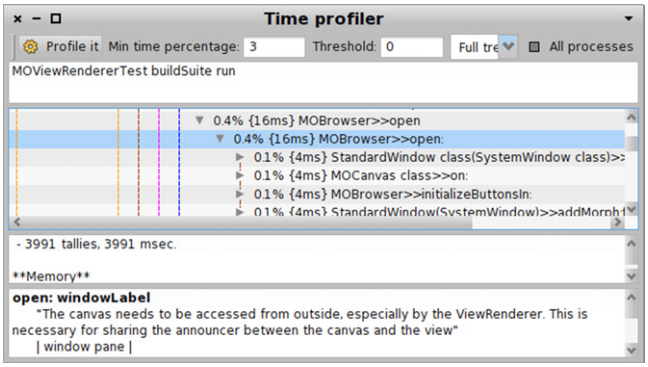

Fig. 1. TimeProfiler in Pharo.

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. Similar to TimeProfiler, branches of the call tree may be contracted and expanded.

Profiling in the Java World: After studying the profiling techniques of several Smalltalk dialects, we briefly discuss profiling and test coverage approaches applied in Java.

JProfiler[6] 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. Beyond post-mortem analyses, it also allows live profiling, displaying profiling information as the program runs. It also offers memory profiling in addition to runtime profiling, and thread and monitor profiling. JProfiler can use either sampling or instrumentation to collect the data it needs.

JFluid [14] exploits dynamic bytecode instrumentation and code hotswapping to collect dynamic information. The JFluid technology is integrated into the NetBeans Profiler.[7] JFluid only instruments and profiles those methods that are actually invoked by methods the user selected to profile; the rest of the code runs unchanged at full speed. To track memory allocations, JFluid does statistical sampling. Other sampling-based profiling techniques, which are often used for feedback-directed optimizations in dynamic compilers, are proposed by Arnold et al. [15] or Whaley [16]. JFluid is a profiler focused on identifying execution bottleneck. The amount of information obtained by JFluid is fixed and cannot be extended.

Emma[8] is a free test coverage tool for Java. Emma can instrument classes at runtime using a dedicated classloader. Emma acquires test coverage information on single code lines, even the partial execution of a line can be determined. Emma presents the results of a coverage analysis in HTML or XML reports. The reports contains information on whether each line is covered or not, and aggregate that information to the level of blocks, methods, and classes.

Cobertura[9] is a tool dedicated to measure test coverage. It uses instrumentation to gather data. Similar to Emma, test coverage information may be stored in an XML file which contains method call graph analysis and coverage. It can also be output in HTML files in an organization similar to the well-known Javadoc tool, showing coverage information up to the line and branch level. Both Emma and Cobertura aggregate information at the level of classes and methods in their reports; a source code correspondence mechanism such as the one offered by Spy would certainly streamline data collection and aggregation.

Conclusion. The Smalltalk code profilers available are very similar: all provide a textual list of methods annotated with their corresponding execution time share; some feature more advanced UI capabilities, but none stray from the familiar "tree of method with time share" data presentation. None 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. Code profilers available for other languages such as Java are usually more advanced—thanks to the greater manpower. Some feature more advanced data presentations, such as JProfiler's code graph, or the code coverage reports of Emma and Covertura that output annotated source code and aggregate the statistics at the method and class levels. However, none are as flexible and generally useable for our needs as we wish: they are rather limited to specific scenarios such as machine code profiling or test coverage analysis and often rely on dedicated virtual machines and/or specific source code instrumentation.

Hence, in all the cases we surveyed, capturing new kind of data—or devising novel presentations of said data—is not in the realm of the natural extensions the frameworks can handle; in contrast, Spy is explicitly designed to handle these specific issues.

[6] http://www.ej-technologies.com/products/jprofiler/screenshots.html
[7] http://profiler.netbeans.org
[8] http://emma.sourceforge.net/
[9] http://cobertura.sourceforge.net

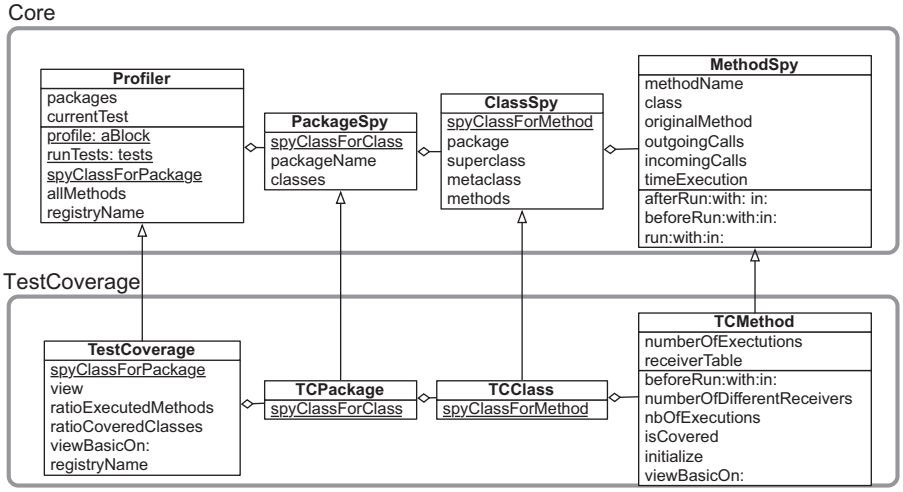

Fig. 2. Structure of Spy.

3. The Spy framework

3.1. Spy in a nutshell

The core classes of Spy are depicted in Fig. 2 and explained next. 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 : aBlockinPackagesNamed : 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 overridden to return a symbol name. Other IDE tools can then easily access the profiling data and analyze or display it as they see fit.

PackageSpy contains the profiling data for a package. Each instance has a name and contains a set of class spies, that is, for each class in the corresponding package, a class spy is created.

ClassSpy describes a Smalltalk class. Its attributes are: its name, a superclass spy, a metaclass spy, and a set of method spies. For each method in the corresponding class, the class spy instance creates a method spy.

MethodSpy wraps a plain Pharo method and accumulates information during the program execution.[10] It has a selector name and belongs to a class spy. MethodSpy is central to Spy, as it contains the hooks used to collect the actual 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 methods are by default empty; they should 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, which serves as a running example with practical uses. Identifying the coverage of the unit tests of an application may be considered as a code profiling activity. A simple coverage profiling tool 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*.

[10] MethodSpy is implemented as a method wrapper [17].

- 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.

Implementing coverage in 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.

The relation between the classes has to be set with the following class-side methods:

```
TestCoverage class >> spyClassForPackage
  ^TCPackage
TCPackage class >> spyClassForClass
  ^TCClass
ClassSpy class >> spyClassForMethod
  ^TCMethod
```

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 := BoundedIdentitySet maxSize : 100
```

Given the resources we can spend when profiling programs, 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 a large amount of times—say a million—then an equal amount of elements would be added to the collection. Allowing the ordered collection to grow up to a million elements significantly slows down the overall program execution, as the collection needs to grow frequently. In addition to this, identifying the number of different elements in a list with one million elements is also slow; moreover, keeping such a large amount of objects incurs a significant memory cost, severely hampering the scalability of the approach. The same schema applies for all the recursively called methods. Alternatively we could also store the elements in a set, but in that case the element addition is costly; further, the sets would still be unreasonably large.

We devise an alternative, as we are not interested in the exact number of receivers, but its order of magnitude. The class BoundedIdentitySet is a subclass of Set in which the number of different values is not 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 on how the related metric will be used: since we only want to differentiate between methods which are called on few and a larger number of receivers, we chose a threshold of 100 unique objects. We inserted the bounded set facility to minimize the time taken by the profiler to gather information. This requires for the developers to assess up to which upper bounds the gathered data remains relevant.

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

```
TCMethod >> beforeRun : methodSelector with : args in : receiver
  numberOfExecutions := numberOfExecutions + 1.
  receiverTable add : receiver.
```

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

The method allClasses is defined on Profiler, the superclass of TestCoverage; it simply returns the list of class spies instantiated during the profiling.

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. This code snippet creates and installs, for each affected package, class, and methods, one of the respective package, class and method spies, according to the principle of structural correspondence (i.e., we maintain a 1-to-1 relationship between each entity and its spy).

```
coverage : = TestCoverage
    profile : [MOViewRendererTest buildSuite run]
    inPackagesMatching : 'Mondrian-*'
```

Executing the code above returns an instance of TestCoverage.

3.4. Visualizing runtime information

The Mondrian framework [12] is integrated with Spy, in order to easily produce visualizations and explore novel ways to present the large amount of profiling data produced. 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 > > visualizeOn : 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 visualize
```

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

- 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.

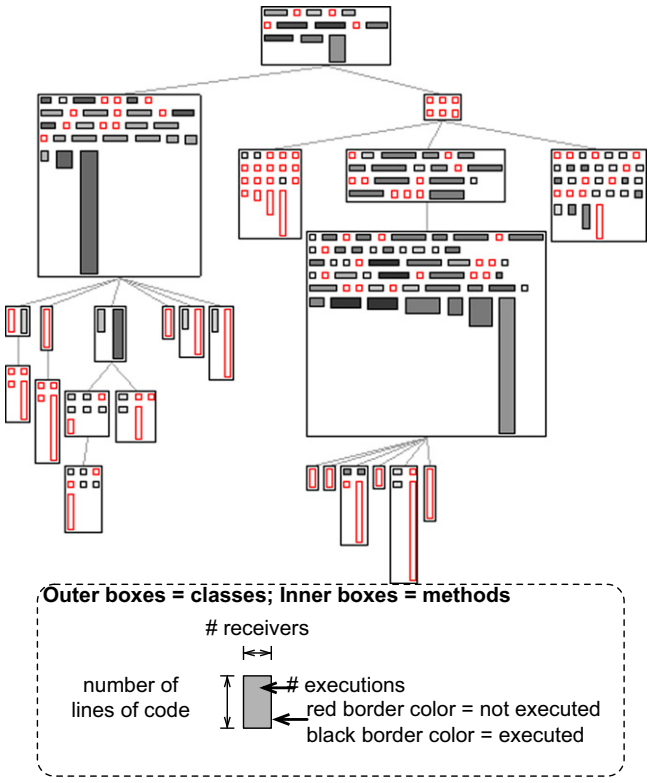

Fig. 3. Test coverage visualization.

- 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.

Visual patterns: From what is depicted in Fig. 3, a number of patterns can be visually identified in order to assist the programmer during his interpretation of the visualization of test coverage. We identified the following five recurring patterns:

- 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, 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 assists 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. These methods can be further inspected by the programmer, by increasing the level of details displayed.

3.5. Call graph and execution time

Additional information is needed, in the form of call graph and time data, in order for programmers to make informed choices to increase the coverage. 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 a second run of the profiled code.

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

The call graph and execution time is estimated by regularly sampling the method call stack: the executed process is interrupted every millisecond and counters associated to the methods that are currently on the stack are incremented. This gives the proportion of time spent in each method, from which the profiler estimates the execution time for each method. The ordering of the method call frames is used to recreate calling context [18], thus determining by which methods a particular method is called by. This is used by Spy to identifying the incoming and outgoing method for each method. On average, computing the execution time and incoming calls costs between 2% and 5% of the total execution time.

Getting the information necessary to build call graphs and compute the execution time is difficult when other runtime information is acquired, as collecting too much information slows down the program and distorts the time information. Because of this, TCMethod collects its information on a different execution of the base program. As a consequence, the code provided to profile : inPackage : must be executable twice. This may sound restrictive; in practice, we have not experienced any serious obstacles stemming from this limitation.

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, so that the additional information can be shown to the programmer on-demand. The script defined in TestCoverage > > visualizeOn : may be refined with a new menu item for visualizing the coverage at the method level:

```
    ...
view interaction action : #inspect;
   item : 'viewcall graph' action     :     visualize.
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 > > visualizeOn : 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.
  view nodes : methods.
  view shape arrowedLine width : 2.
  view edges : methods from : #yourself toAll : #outgoingCalls.
  view treeLayout
```

The script above (Fig. 4) shapes a method using its number of lines of code (height), the number of different receivers the method has been invoked on (width) and the number of different executions (color intensity). A logarithm scale softens the disparity for these metrics. Edges represent outgoing calls.

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.

```
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. Methods are ordered from top (callers) to bottom (callees). The arrowed edges represent the control flow between methods.

3.6. Structure of the profiling

One of the key features of Spy is the correspondence between the model classes in Spy and the meta-model of Smalltalk. Each code entity (Method, Class, Package) is monitored by exactly one entity (MethodSpy, ClassSpy, PackageSpy), which aggregates all the information required at that level. In contrast, most profilers gather the profiling information in the form of a run-time trace, that needs a consequent further processing to extract the higher-level information at the level of classes and packages. This feature is even useful at the level of methods, as some of the metrics that Spy gathers, such as the number of receivers a method is invoked on, need to be aggregated across all the executions of the methods. Adopting structural correspondence considerably simplifies the processing of the data as it is not spread out over an entire execution trace.

Aggregating information at the level of classes and packages also involves data processing at the class and package levels in conventional trace-based profilers. This is not the case in Spy which supports a natural aggregation of information

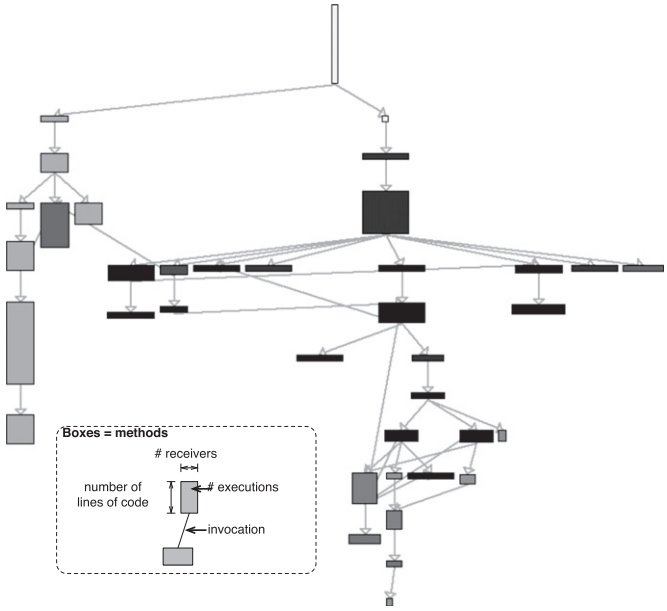

Fig. 4. Call graph of the method MOViewRenderer > > testTranslation.

for classes and packages. As an example, coverage information for classes and packages can be easily deduced from the information found at the method level, as demonstrated in the following code snippet:

```
TCClass > > isCovered
  ^self allMethods anySatisfy: #isCovered
TCClass > > allCoveredMethods
  ^self allMethods select: #isCovered
TCPackage > > allCoveredClasses
  ^self classes select: #isCovered
```

In case that the scope has to be reduced at runtime, a conditional may be inserted in the beforeRun : with : in : method of the coverage profiler; this is often necessary to reduce the amount of collected information. In case a filtering is only needed to display the collected information, Spy's structural correspondence allows us to simply filter out the class/package/method spies that we do not wish to be displayed, instead of traversing a potentially large execution trace. Hence structural correspondence is useful both for aggregating the information, and to filter it—in addition to being a natural and straightforward choice for the instrumentation part.

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

Even though we use test coverage as an illustrative scenario for Spy, the visualization of the coverage presents a number of innovations. First, it takes a different stance from other code profilers regarding the coverage assessment criterion. Most test coverage tools, including Cobertura,[11] JCoverage,[12] Parasoft Jtest,[13] consider structural elements in a binary fashion, either an element (method or statement) is executed or not by the test. Our coverage visualization correlates three metrics, namely number of different receivers, number of executions and number of lines of code. This unique combination gives a new perspective on whether an element is properly covered or not. Hapao[14] is a full-fledged test coverage tool that is based on the illustrative example used in this paper. Hapao has been used to increase the coverage of several medium-sized and large applications.

[11] http://cobertura.sourceforge.net
[12] http://www.jcoverage.com
[13] http://www.parasoft.com/jsp/smallbusiness/tool_description.jsp?product=Jtest&
[14] http://hapao.dcc.uchile.cl/

4. Implementation

Code instrumentation: Spy gathers dynamic information by wrapping compiled methods, which accumulate information at execution time. This first implementation of Spy was based on a technique called "object as compiled method" [19]. It uses a feature of the Pharo virtual machine to reify messages when a plain object is found instead of a compiled method in a method dictionary. Although easy to implement, this technique is rather slow with an overhead of approximatively 500% on macro-benchmarks.

The second approach was inspired by the original version of method wrapper [17]. The compiled method to instrument is replaced by a new compiled method that reifies the message being sent and invokes the associated instance of MethodSpy. This new compiled method is actually the copy of a template compiled method. In Smalltalk, a method can take up to 16 arguments. We therefore have 16 templates to cover all the methods that can be possibility instrumented. Consider the template for a 2-args method:

```
with2arg : v1 arg : v2
    ^ #metaObjectrun : #selector with : v1.v2} in : self
```

This method is not meant to be executed as it is: the method run : with : in : is not understood by the object #metaObject. For each method to instrument, a template is copied and adjusted with method attributes.

To illustrate the instrumentation, consider a method, let us say MOAnnounceritem : action :. Instrumenting item : action : is a four-steps process:

1. Copy the template with2arg : arg : given above. This copy will replace the method to be instrumented.
2. Adjust the copy the literals #metaObject by the corresponding MethodSpy instance and #selector by the selector #item : action :. The effect is to send the message run : with : in : to the MethodSpy instance with the selector #item : action : as second parameter. The template method is transformed into:

```
item : v1 action : v2
    ^ methodSpy run : #item : action : with : v1.v2} in : self
```

3. Putting the adjusted template copy into the method dictionary of MOAnnouncer
4. Adjust the copied method's literals corresponding to the superclass reference and the pragmas. Pragmas are a meta-information associated to a compiled method object.

The overhead incurred by solely wrapping methods is about 25%, which is reasonable in most of the situations we have to deal with.

Spy requirement: Spy does not rely on Smalltalk specificities. The instrumentation realized by Spy relies on the reflective capabilities of Pharo. The key point of the instrumentation is to associate an instance of the class MethodSpy to each method of the profiled application. The spy accumulate runtime information and contains a reference to a spy for a class. In principle, any bytecode instrumentation framework or aspect-oriented language extension is able to realize this instrumentation.

5. Validation

In this section, we present a validation of Spy by illustrating how we built three additional profiling tools on top of Spy. Since our framework is built with flexibility in mind, the best way to validate our claim is to exercise said flexibility by instantiating the framework in several instances. Additionally, we demonstrate the integration of profiling information into the Pharo IDE to highlight the interoperability of the Spy framework with existing tools and frameworks of Pharo Smalltalk.

5.1. 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.[15] For a given program written in Smalltalk, we can deduce the type information from executing the associated unit tests, as proposed by other researchers [20]. Comparing the deduced nominal types and the list of values effectively provided as argument helps in identifying anomalies and deficiencies. As an illustration, a method shape : takes instances of MOFilledShape and MOLabelShape, both subclasses of MOShape. MOLineShape is a sibling of MOLabelShape and MOFilledShape. The nominal type of the argument of shape : is MOShape. However, according to the definition of shape :, it does not make sense to provide instances of MOLineShape. This therefore suggests that a class MONodeShape was missing from the shape hierarchy.

The idea of this profiler 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

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

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

5.2. Time profiling blueprints

As a second application, we proposed a time execution profiler.[16] 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.

5.3. 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 [21]. 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"[17] (e.g., *how much has a metric increased*?).

Hip provides facilities to automatically profile a code snippet throughout a set of package versions available from a Monticello[18] repository by loading each version, profiling it, and adding the gathered profiling information to a Hip version collection structure.

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 the 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; yellow artifacts are also interesting, as from that version onwards, the developers should put focus on these newly created artifacts, and on how they evolve from now on.

5.4. 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 [22]. 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

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

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

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

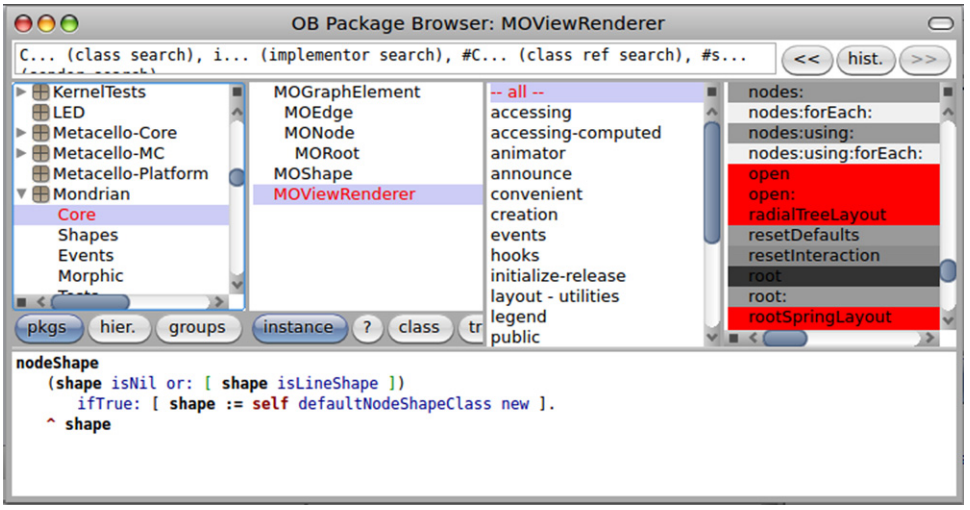

Fig. 5. Integration of profiling information into the Pharo IDE.

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. Fig. 5 illustrates how profiling information is visualized in the Pharo IDE.

5.5. *Flexibility of Spy*

In addition to our running example on code coverage, we described three instantiations of Spy in this section: extracting types, measuring execution time, and differencing profiling. All in all, Spy was successfully used in four occasions with very distinct goals, and integrated with other frameworks that easily made use of the information it produced.

These three instantiations —code coverage, type extraction, and execution time—demonstrate the flexibility of Spy for collecting a wide array of information, from binary coverage metrics up to numeric metrics such as execution time, number of executions, or number of receivers, which are aggregated upon several executions of the method; these metrics can in turn be seamlessly aggregated to produce class or package metrics. Our type extraction case studies show that an entirely distinct class of information—run-time types—was also collected using the same, simple, instrumentation mechanism.

Our last extension—profiling differentiation—highlights Spy's flexibility in another dimension, namely its ability to compare and cross-reference information collected in distinct runs of the application. Spy produces profiles that are simply structured, following the well-known Package/Class/Method division—thanks to structural correspondence—and hence easily processed by Spy itself and by third-parties.

In addition to profile differentiation, profiles are easily accessible by independent tools beyond the framework itself. This allowed us to easily integrate Spy with Mondrian, to the point that most of Spy's graphical user interface and visualization layer is now composed of Mondrian scripts. As we have also shown below, this ease of access by independent tools is not anecdotal, since we were able to integrate Spy's coverage information with Pharo's system browser.

To conclude, based on our experience instantiating Spy four times, and interfacing it with two distinct frameworks, we believe we have accumulated evidence that Spy is flexible both in terms of the kind of information it can collect—to devise novel profiling applications—and in terms of how that information can be accessed—to be used by third-parties in unforeseen situations.

6. 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. Mondrian offers a domain-specific language to define visualization. With all the applications of Spy we have realized, we have never felt the need to extend Mondrian's language to render profiling information. The core classes of Spy contain many navigation and information gathering methods that are intensively used in Mondrian scripts, which are often a few lines long.

As far as we know, Spy is the first Smalltalk framework that allows to easily collect information and display visual representations of program executions. 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 moving some of the primitives of Spy in the virtual machine (e.g., counting method execution, number of different receivers). Another restriction is about discriminating the information introduced by the framework itself from the one produced by the code to profile. No distinction is made so far. Even though this has not been a serious problem so far, this may constitute an obstacle when one wants to profile a profiler. Promising approaches have been proposed to tackle this very problem [23]. We plan to address this issue as a further future work.

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. The work presented in this article is partially funded by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile.

References

[1] Marré M, Bertolino A. Reducing and estimating the cost of test coverage criteria. In: ICSE '96: proceedings of the 18th international conference on software engineering. Washington, DC, USA: IEEE Computer Society; 1996. p. 486–94.
[2] Binder W. Portable and accurate sampling profiling for java. Software—Practice and Experience 2006;36(6):615–50, doi:10.1002/spe.v36:6.
[3] Agesen O, Holzle U. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. Technical Report. Santa Barbara, CA, USA: Department of Computer Science, University of California; 1995.
[4] Haupt M, Hirschfeld R, Denker M. 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. p. 17–22.
[5] Arnold M. Online profiling and feedback-directed optimization of java. PhD thesis. Rutgers University; October 2002.
[6] Röthlisberger D, Denker M, Tanter É. Unanticipated partial behavioral reflection: adapting applications at runtime. Journal of Computer Languages, Systems and Structures 2008;34(2–3):46–65, doi:10.1016/j.cl.2007.05.001.
[7] Holten D, Cornelissen B, van Wijk JJ. 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. p. 47–54, doi:10.1109/VISSOF.2007.4290699.
[8] Meyer B. Object-oriented software construction. 2nd ed, Prentice-Hall; 1997.
[9] Ingalls D, Kaehler T, Maloney J, Wallace S, Kay A. 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. p. 318–26, doi:10.1145/263700.263754.
[10] Black A, Ducasse S, Nierstrasz O, Pollet D, Cassou D, Denker M. Pharo by example. Square Bracket Associates; 2009. URL ⟨http://pharobyexample.org⟩.
[11] VisualWorks, Cincom Smalltalk ⟨http://www.cincomsmalltalk.com/⟩, archived at ⟨http://www.webcitation.org/5p1rRxls5⟩, 2010. URL ⟨http://www.cincomsmalltalk.com/⟩.
[12] Meyer M, Gîrba T, Lungu M. Mondrian: an agile visualization framework. In: ACM symposium on software visualization (SoftVis'06). New York, NY, USA: ACM Press; 2006. p. 135–44, doi:10.1145/1148493.1148513.
[13] Bergel A, Robbes R, Binder W. Visualizing dynamic metrics with profiling blueprints. In: Vitek J, editor. Objects, models, components, patterns. Lecture notes in computer science, vol. 6141. Berlin, Heidelberg: Springer; 2010. p. 291–309, doi:10.1007/978-3-642-13953-6_16.
[14] Dmitriev S. Language oriented programming: the next programming paradigm, November 2004 ⟨http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf⟩.
[15] Arnold M, Ryder BG. A framework for reducing the cost of instrumented code. In: Proceedings of the ACM SIGPLAN 2001 conference on programming language design and implementation, PLDI '01. New York, NY, USA: ACM; 2001. p. 168–79, doi:10.1145/378795.378832.
[16] Whaley J. A portable sampling-based profiler for java virtual machines. In: Proceedings of the ACM 2000 conference on Java Grande, JAVA '00. New York, NY, USA: ACM; 2000. p. 78–87, doi:10.1145/337449.337483.
[17] Brant J, Foote B, Johnson R, Roberts D. Wrappers to the rescue. In: Proceedings European conference on object oriented programming (ECOOP'98). Lecture notes in computer science, vol. 1445. Springer-Verlag; 1998. p. 396–417.
[18] Ammons G, Ball T, Larus JR. Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of the ACM SIGPLAN 1997 conference on programming language design and implementation, PLDI '97. New York, NY, USA: ACM; 1997. p. 85–96, doi:10.1145/258915.258924.
[19] Bergel A, Denker M. Prototyping languages, related constructs and tools with Squeak. In: Proceedings of the ECOOP'06 workshop on revival of dynamic languages, 2006.
[20] Röthlisberger D, Greevy O, Nierstrasz O. Exploiting runtime information in the IDE. In: Proceedings of the 16th international conference on program comprehension (ICPC 2008). Los Alamitos, CA, USA: IEEE Computer Society; 2008. p. 63–72, doi:10.1109/ICPC.2008.32.
[21] Gîrba T, Lanza M, Ducasse S. Characterizing the evolution of class hierarchies. In: Proceedings of 9th European conference on software maintenance and reengineering (CSMR'05). Los Alamitos, CA: IEEE Computer Society; 2005. p. 2–11, doi:10.1109/CSMR.2005.15.
[22] Bergel A, Ducasse S, Putney C, Wuyts R. Creating sophisticated development tools with OmniBrowser. Journal of Computer Languages, Systems and Structures 2008;34(2–3):109–29, doi:10.1016/j.cl.2007.05.005.
[23] Tanter É. Execution levels for aspect-oriented programming. In: Proceedings of the 9th ACM international conference on aspect-oriented software development (AOSD 2010). Rennes and Saint Malo, France: ACM Press; 2010. p. 37–48.