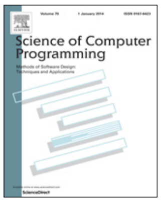
Science of Computer Programming 79 (2014) 86–100

Contents lists available at ScienceDirect

Science of Computer Programming

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



Increasing test coverage with Hapao

Alexandre Bergel [*], Vanessa Peña

PLEIAD Lab, University of Chile, Santiago, Chile

ARTICLE INFO

Article history:
Received 19 January 2011
Received in revised form 12 January 2012
Accepted 24 April 2012
Available online 7 May 2012

Keywords:
Testing
Pharo
Coverage
Visualization

ABSTRACT

Test coverage is about assessing the relevance of unit tests against the tested application. It is widely acknowledged that software with a "good" test coverage is more robust against unanticipated execution, thus lowering the maintenance cost. However, ensuring good quality coverage is challenging, especially since most of the available test coverage tools do not discriminate between software components that require "strong" coverage from the components that require less attention from the unit tests.

Hapao is an innovative test coverage tool, implemented in the Pharo Smalltalk programming language. It employs an effective and intuitive graphical representation to visually assess the quality of the coverage. A combination of appropriate metrics and relations visually shape methods and classes, which indicates to the programmer whether more effort on testing is required.

This paper presents the important features of Hapao by illustrating its application on an open source software.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Constructing solid software requires expressive software engineering tools. Writing and executing unit tests are two activities well supported by common programming environments. The panoply of tools commonly used comprises of an implementation of xUnit, a widely recognized testing framework. xUnit can be complemented with a test coverage tool [1] to assess the part of the program source that are executed by the associated unit tests. This provides useful indications on what has not been considered by unit tests.

Consider Cobertura,[1] a popular test coverage tool. Cobertura calculates the percentage of code accessed by tests. Results are reported in many different reports, which includes the percentage of lines and branches covered for each class and package. It even indicates the McCabe cyclomatic code complexity of each class to help developers identify software parts considered complex, suggesting to strengthen the tests related to that part. Whereas Cobertura produces an accurate status report of the software components that are executed by unit tests, it however offers little indication whether those components have been well tested and on how to efficiently increase the coverage.

The limitation of traditional test coverage tools stems from the fact that source code elements are tagged in a binary fashion: a method or a statement is either executed by unit tests or not. No indications are provided on *how well* the method or statement is covered. A complex method that is executed just once by unit tests is considered covered, and equally considered as a trivial method executed many times on many different receivers with diverse arguments. We argue that this way of assessing test coverage gives an inaccurate sense of what actually needs to be better tested. A software engineer

* Corresponding author. Tel.: +56 9 913 123 81.
E-mail addresses: alexandre.bergel@me.com (A. Bergel), vpena@dcc.uchile.cl (V. Peña).
URL: http://bergel.eu (A. Bergel).

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

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.04.006

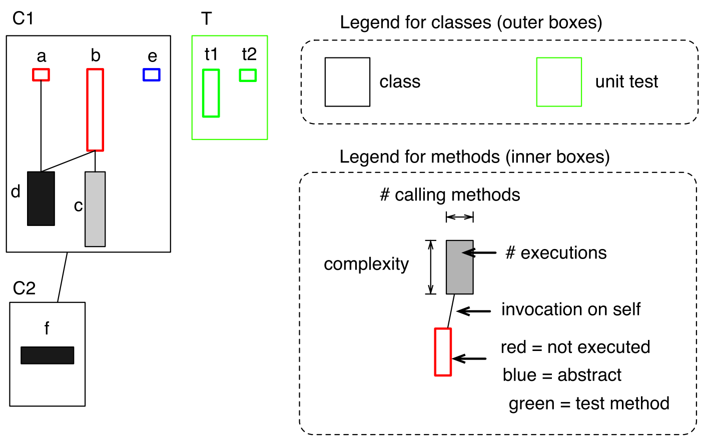


Fig. 1. Test blueprint description.

may deduce from the coverage given by traditional tools that the complex method does not need more testing, even if executed just once by the test.

This paper presents Hapao,[2] a new code coverage tool intended to accurately estimate the coverage and produce a valuable set of actions on how to increase it. Hapao reaches its goals by offering an appealing and intuitive graphical visualization of the coverage.

In case of a complex method executed just once by the unit tests, Hapao concisely indicates that the method is executed just once. The graph of call flow in which the complex method is part of is also provided. Those are essential ingredients to determine whether the complex method deserves further testing or not. None of the test coverage tools that we are aware of answers those questions.

Hapao is available for the Pharo programming language[3] under the MIT license and it has been used to increase the test coverage of several software systems.

This paper is organized as follows. The Hapao test coverage tool is first presented (Section 2), including a brief description of the visualization and a short tutorial on how to install and use it. The case study we conducted on Mondrian is subsequently presented (Section 3). Integration in the programming environment is then discussed (Section 4). The experience we gain during the construction of Hapao is then presented (Section 5). The related work is subsequently reviewed (Section 6) before concluding (Section 7).

2. Hapao

This section first introduces *test blueprint*, a visual aid to assess the coverage of a test. It then provides a tutorial-like presentation of Hapao, our test coverage tool.

2.1. Test blueprint

Test blueprint is a visual aid for practitioners to assess and increase the tests coverage of their applications. The blueprint shows the coverage of methods and class hierarchies.

Before applying test blueprint on a real world example, we first introduce the visualization on a contrived but representative example, given in Fig. 1.

Encapsulating boxes represents classes (`C1`, `C2` and `T`). Inheritance is indicated with an edge between classes. Subclasses are below their superclass (`C1` is the superclass of `C2`). The superclass of `T` is not part of the analysis. The green border indicates a unit test or a test method.

Inner boxes represent methods. `C1` defines five methods, `a`, `b`, `c`, `d` and `e`. `C2` defines one method, `f`. Each method is represented as a small box, visually defined with fives dimensions:

- Height is the cyclomatic complexity of the method. The more the method takes different paths at execution time, the taller the box will be (*e.g.*, Method `b`).
- Width is the number of different methods that call the method when running the tests. A wide method (`f`) means the method has been executed by many different methods, which could be a test method or not. A thin method (`a`, `b`, `c`) means the method has been executed zero or only a few times.

[2] http://hapao.dcc.uchile.cl.
[3] http://pharo-project.org.

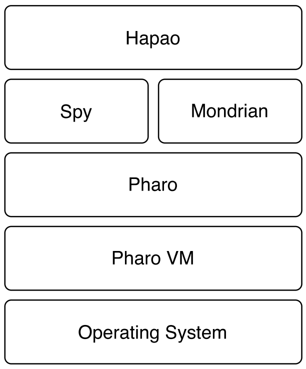


Fig. 2. Overall structure of Hapao.

- Gray intensity reflects the number of times the method has been executed. A dark method (`d`, `f`) has been executed many times. A light-toned method (`c`) has been executed a few times. The intensity is based on a relative proportion over all the methods. If all methods are executed just once, then they will all appear as black. If all but one methods are executed once and one is executed twice, then all methods will be gray, and the method executed twice will be black.
- A red border color (light gray on a B&W printout) means the method has not been executed (`a`, `b`). A blue border indicates abstract method (`e`). A green border indicates that the method is a test method (`t1` and `t2`), defined in a unit test (`T`). Note that a unit test may contain methods that are not test methods, such as utility methods.
- The call-flow on the `self` variable is indicated with edges between methods. This happens if the body of `a` contains the expression `self d`, meaning that the message `d` is sent to self. The method `a` calls `d` on `self`. The method `b` calls `d` and `c` on `self`. Note that we are focusing on the *call-flow* instead of the *control-flow*. The value of the visualization is to contrast what the path the execution could be (call-flow) against what methods is actually executed (non-red method). The call-flow is scoped to the class.

The test blueprint gives an overview of the coverage by using contrasted colors. Using white, gray and red has the effect of immediately calling attention to contrast differences [2].

We have used test blueprint to increase the coverage of a number of software, including Mondrian,[4] Merlin[5] and other tools of the Moose software analysis platform.[6]

Test blueprint is a polymetric view [3] and is a variant of the class blueprint proposed by Ducasse and Lanza [4]. Class blueprint has been produced for maintenance and reverse engineering activities. Test blueprint focuses on assessing test coverage.

2.2. *Loading Hapao*

The source code of Hapao is stored on the squeaksource forge. It is loadable in Pharo by executing the following code snippet:

```
Gofer new
 squeaksource: 'Spy';
 package: 'ConfigurationOfSpy';
 load.
(Smalltalk at: #ConfigurationOfSpy) project lastVersion load
```

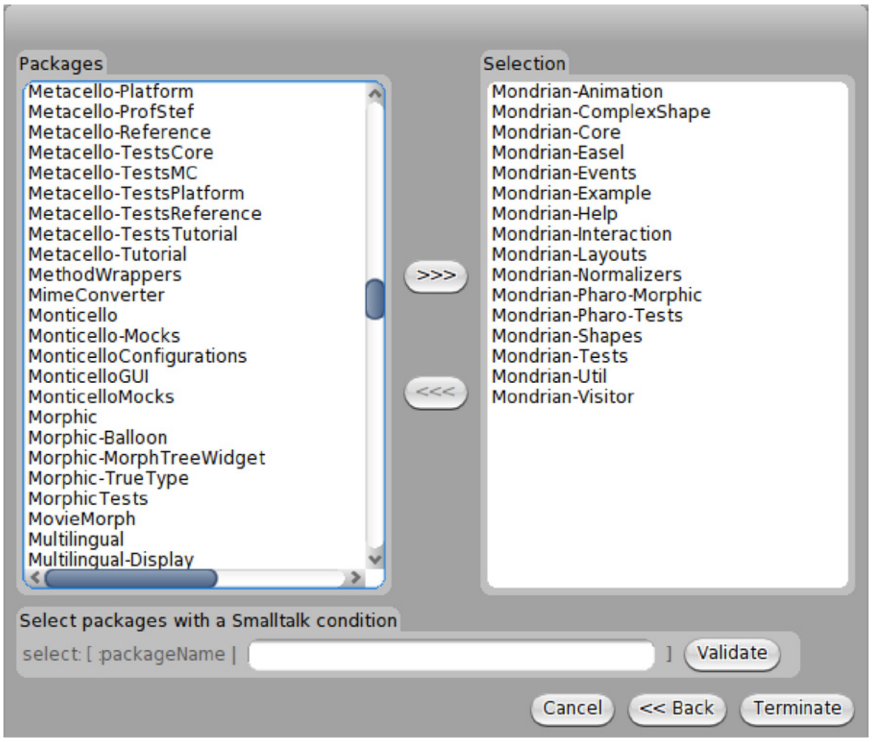
Hapao uses the Spy framework, a low level instrumentation framework [5], to profile the application. Hapao is packaged with Spy. A number of dependent libraries are loaded by Spy, including Mondrian, an agile visualization engine used to render the test blueprint (Fig. 2).

The official website of Hapao is http://hapao.dcc.uchile.cl, it contains a ready-to-use distribution and an end user manual.

2.3. *Hapao in a nutshell*

Once loaded, Hapao is accessible from the Pharo menu (obtained by clicking on the background). Hapao produces a test blueprint from a given set of packages. Opening Hapao will pop up a small wizard which lists the packages installed in the Pharo image (Fig. 3).

4 http://www.moosetechnology.org/tools/mondrian.
5 http://www.squeaksource.com/Merlin.html.
6 http://www.moosetechnology.org.


Packages
Metacello-Platform
Metacello-ProfStef
Metacello-Reference
Metacello-TestsCore
Metacello-TestsMC
Metacello-TestsPlatform
Metacello-TestsReference
Metacello-TestsTutorial
Metacello-Tutorial
MethodWrappers
MimeConverter
Monticello
Monticello-Mocks
MonticelloConfigurations
MonticelloGUI
MonticelloMocks
Morphic
Morphic-Balloon
Morphic-MorphTreeWidget
Morphic-TrueType
MorphicTests
MovieMorph
Multilingual
Multilingual-Display

Selection
Mondrian-Animation
Mondrian-ComplexShape
Mondrian-Core
Mondrian-Easel
Mondrian-Events
Mondrian-Example
Mondrian-Help
Mondrian-Interaction
Mondrian-Layouts
Mondrian-Normalizers
Mondrian-Pharo-Morphic
Mondrian-Pharo-Tests
Mondrian-Shapes
Mondrian-Tests
Mondrian-Util
Mondrian-Visitor

Select packages with a Smalltalk condition
select: [:packageName |]

Fig. 3. Selecting packages in Hapao.

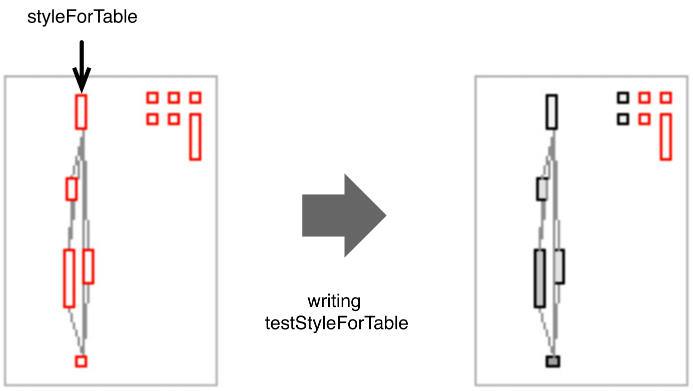


Fig. 4. The class `SHStyleElement` of Shout.

We first need to select the package we want to analyze. As a motivating example, we will consider the Shout application. Shout is a small framework to highlight and color the syntax of a source code. It is intensively used by the Pharo programming environment.

Select the packages `Shout` and `ShoutTests`. The following steps are then automatically taken: (i) Shout is instrumented with Spy; (ii) Shout's unit tests are run; (iii) Shout is reverted to its original definition (the instrumentation is removed); (iv) the test blueprint is generated and rendered.

Global numerical metrics and a legend are located next to the test blueprint. Shout is comprised of 11 classes and 182 methods. Consider the blueprint of the class `SHStyleElement`,[7] depicted on the left hand side of Fig. 4.

`SHStyleElement` is not tested at all since all its methods have a red border. We will now write a new unit test and test this class. However, `SHStyleElement` defines 11 methods (*i.e.*, the blueprint contains 11 small red boxes). The question Hapao helps answer is *"Which method should we test first?"*

We will concentrate on the higher method in the method call chain. The method `styleForTable` is the root of the call chain. The idea behind testing a root method is that it is likely that the methods located below it will also be tested.

[7] We use Pharo 1.4, version 14172.

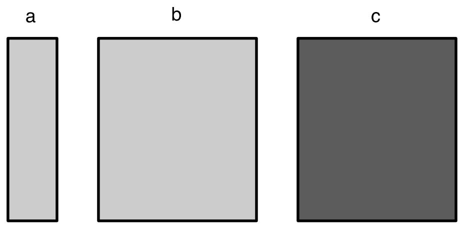


Fig. 5. Possible shapes for a particular method.

We create a new unit test called `SHStyleElementTest` and define the following method:

```
SHStyleElementTest>>testStyleForTable
 | element table |
 element := SHStyleElement withTokens: #(#self #super #true #false).
 table := element styleForTable.
 self assert: table size = 4.
```

This test creates a new element with a list of tokens provided in an array (Line 3) and calls `styleForTable` (Line 4). Pressing the button `Run tests` in the HAPAO window will generate a new blueprint that indicates the new test coverage of `SHStyleElement` (right-hand side of Fig. 4). We now see the simple call to `styleForTable` in the test covers 7 of the 11 methods of `SHStyleElement`.

Writing unit tests requires the developer to have knowledge of his or her software. HAPAO does not generate tests, they have to be manually written by the programmer. For this example, we are not particularly knowledgeable of Shout, but we were able to easily find an example in Pharo on how to use it.

Note that HAPAO does not give the result of the test execution. Pharo offers several tools to run and see the result of the tests. We did not see an immediate need to overload the test blueprint with the test results.

In comparison to other test coverage tools (Section 6), HAPAO indicates which methods are likely to significantly increase the coverage when tested. Testing a method located at the top of a class in the blueprint may have a ripple effect on the method below.

The next section details HAPAO using a larger example.

2.4. The shape of a method

Each method has a particular visual representation, which tells about its complexity, the amount of times it is executed, and the number of different methods by which represented method is invoked.

The way a method is shaped depends on the other methods that call it. Consider Fig. 5, which gives three different shapes for a same method. Only the way the method is called has an impact on its width and its color intensity. The height remains the same since it is the same method that is represented (the same complexity therefore).

Shape `a` indicates the method is invoked by few methods since it is narrow. Shape `b` tells that more methods invoke the represented method than when it is represented by `a`. Note that being invoked by more methods does not necessarily imply that is executed more often. With `a`, maybe the method is executed 5 times, by only 1 method.[8] With `b`, the method can still be executed 5 times, but by 3 different methods. The gray intensity of `a` and `b` is the same, meaning that both are executed the same amount of times. Shape `c` indicates that the method is executed by the same amount of different methods as `b`, but more often. The represented method is executed more often in `c` than in `a` and `b`.

The following section shows how the method shape drives the testing effort.

3. Assessing test coverage

This section presents a realistic case study we conducted on the Mondrian application. Mondrian is the visualization engine that we use to render test blueprint. As we use Mondrian as the focus of our analysis, we are therefore visualizing Mondrian using Mondrian.

3.1. Mondrian test blueprint

Fig. 6 shows a small portion of the test blueprint[9] obtained by running Mondrian's unit tests. The Mondrian layout hierarchy is composed of 16 classes.

[8] the actual values are given by tooltip, the visual aspect is just an indication.
[9] The complete rendering is available online: http://bergel.eu/download/HapaoOnMondrian/hapao.html.

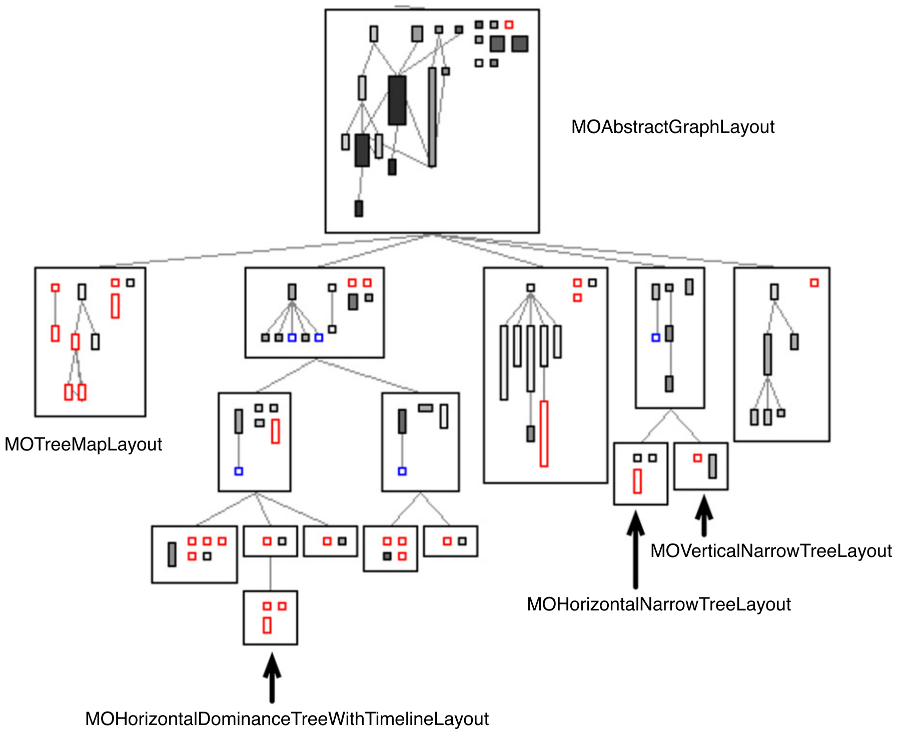

MOAbstractGraphLayout
MOTreeMapLayout
MOVerticalNarrowTreeLayout
MOHorizontalNarrowTreeLayout
MOHorizontalDominanceTreeWithTimelineLayout

Fig. 6. Mondrian layout hierarchy.

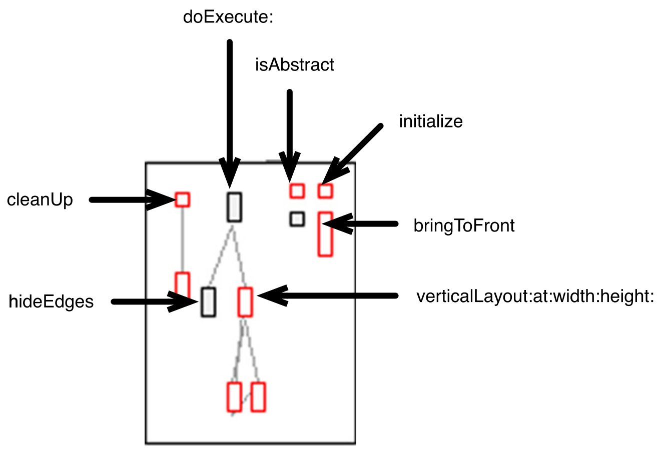


Fig. 7. Blueprint of `MOTreeMapLayout`.

The layout class hierarchy of Mondrian is heterogeneous in its coverage. For example, the class `MOAbstractGraphLayout` is relatively well covered since (i) only 1 of its 21 methods is not executed by Mondrian unit tests and (ii) all the tallest methods but one are wide, meaning that are called by several different methods. On the opposite, `MOTreeMapLayout` has only 3 of its 10 methods executed, which reflects a relatively poor coverage. `MOHorizontalDominanceTreeWithTimelineLayout` is not covered by the unit tests at all.

Fig. 7 zooms in on the class `MOTreeMapLayout`. Visually we see that the method `doExecute:` contains a call to `hideEdges` and `verticalLayout:at:width:height:`. The method `hideEdges` is covered by the tests since it is shaded in gray. This means that at least one call to `hideEdges` belongs to the control flow of the method `doExecute`. The other call, `verticalLayout:at:width:height:` has not been executed. This call is therefore probably contained in an unreached branch.

Right-clicking on the method gives an option to browse the method definition:

```
MOTreeMapLayout >>doExecute: aGraph
 | offset |
 graph := aGraph.
 self **hideEdges**.
 offset := self horizontalGap.
```

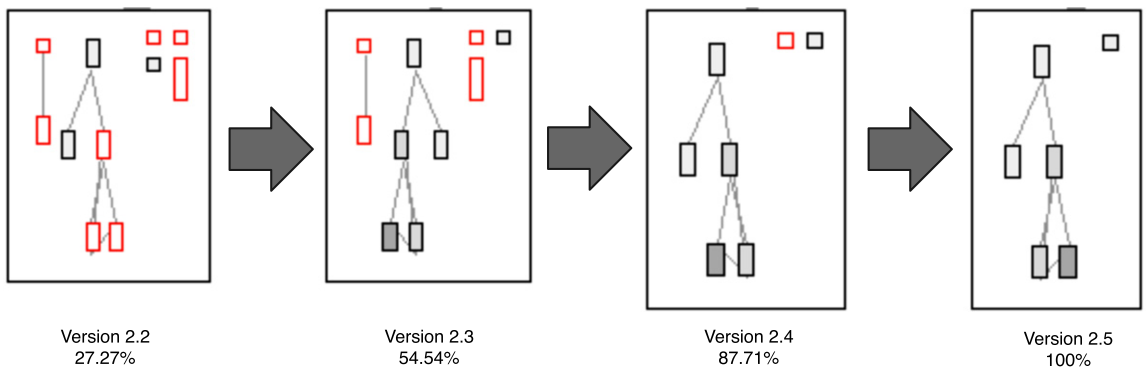


Fig. 8. Evolution of `MOTreeMapLayout`.

```
6   (self rootNodesFor: aGraph nodes) do:
7    [ :aRoot |
8    self
9     verticalLayout: aRoot
10    at: offset @ 5
11    width: aRoot width
12    height: aRoot height.
13   offset := offset + aRoot width + self horizontalGap ]
```

Since the method `verticalLayout:at:width:height:` is painted with a red border, Lines 8 to 13 are therefore never executed.

3.2. Increasing the coverage of `MOTreeMapLayout`

In the above, increasing the test coverage of Mondrian is achieved via two different ways: either by augmenting an already existing test with the missing method calls, or by creating a new test.

To illustrate the first way of increasing the coverage, we have to find the tests referencing our observed method. In the Version 2.2 of Mondrian, the only test that references the class `MOTreeMapLayout` is `MOLayoutTest>>testTransparentEdges`. The list of tests that reference a class is obtained by right-clicking on the class.

The test `MOLayoutTest>>testTransparentEdges` fails. The failure stems from `hideEdges`. This method is one of the remaining methods that has not been ported when in 2008 Mondrian was migrated to Pharo from its original implementation in VisualWorks. As a consequence, `testTransparentEdges` actually fails as `hideEdges` invokes non-existing and obsolete methods. Fixing the definition of `hideEdges` turns `testTransparentEdges` green, thus increasing the coverage. By fixing `testTransparentEdges`, the method `initialize` is now covered. This resulted in the Version 2.3 of Mondrian (Fig. 8). An updated visualization is obtained by pressing the *Run tests* button, at the top of the Hapao window.

Consider another example, the class `MOHorizontalNarrowTreeLayout`. This class has three methods. Only two of them are covered since they are painted in gray in Fig. 6. The most complex one (*i.e.*, the tallest, which has the highest cyclomatic complexity) is left uncovered. Only one test refers to this class, `testNarrowHorizontalTreeFromToPositions`. This test does not provide enough content to effectively test the untested method of `MOHorizontalNarrowTreeLayout`.[10] By looking at the tests that refer to `MOVerticalNarrowTreeLayout`, its sibling class, we found an opportunity to test `MOHorizontalNarrowTreeLayout`. The method `MOLayoutTest>>testChangingLayout` has been extended to:

```
MOLayoutTest>>testChangingLayout
 | view node layout |
 layout := MOVerticalNarrowTreeLayout new.
 ...
 view root applyLayout.
 self assert: (layout childrenFor: node) size = 2.
 self assert: (layout parentsFor: node) size = 1.
 ...
 layout := MOHorizontalNarrowTreeLayout new.
 view root layout: layout.
 view root applyLayout.
 self assert: (layout childrenFor: node) size = 2.
 self assert: (layout parentsFor: node) size = 1.
```

[10] `testNarrowHorizontalTreeFromToPositions` simply check that some instance variables are properly initialized upon instantiation.

Augmenting an existing test is the easiest way to augment the coverage since only a few additional calls may be enough. Right-clicking on a class or a method offers an option to browse unit tests that either refer to the class or the method, respectively.

We augmented `testChangingLayout` shown in **bold**. The assertions defined for `MOVerticalNarrowTreeLayout` are inspired for the test of `MOHorizontalNarrowTreeLayout`. This enhanced version of `testChangingLayout` fully covers the class `MOHorizontalNarrowTreeLayout` (Fig. 10).

3.3. Identifying dead code

The list of the methods that call a particular method are accessible by right-clicking on the method. Inspecting the callers of `cleanUp`, `bringToFront` and `isAbstract` reveals that these methods are called nowhere. No other methods in Mondrian call one of these three methods. As a consequence, either these methods belong to the public API of Mondrian and are meant to be called by an end user but no occurrence of it can be found, or they are simply dead code. By being close to the development of Mondrian, we immediately identify these three methods as dead code.

Version 2.4 of Mondrian removes `cleanUp` and `bringToFront` (Fig. 8). Version 2.5 removes `isAbstract`. At that stage, we consider `MOTreeMapLayout` as well covered.

Hapao identifies a potential dead method when the two following conditions are satisfied:

- the method appears with a red border: it means that the method is not executed when the unit tests are run.
- no other methods call it: right-clicking on the method gives the option `browse senders`, which opens a Pharo code browser with the list of methods that call the method of interest. If the list is empty, then the method is called nowhere.

A method can be removed by right-clicking on it. To prevent accidents, only non-covered methods can be removed.

3.4. Is my method well covered?

Executing a method in all the different possible settings cannot be realistically entirely conducted: the state of the executing environment is comprised of so many variables that all the different combinations based on its definition, the receiver, arguments and global environment cannot be covered by our unit tests.

Determining whether a method is well covered is not as simple as saying whether it has been executed or not, as traditional test coverage tools do. A number of factors, including its definition, the variety of its object receivers and arguments, need to be in order to assess a coverage.

For that reason, the quality of a method coverage has to be assessed from only a few combinations of these factors. We utilize three variables, the cyclomatic complexity, the number of different callers and the number of execution time. Each of these variables corresponds to a dimension of the visual representation, height, width, and color intensity, respectively.

Consider the class `MOLineShape` given in Fig. 9. This class contains a very tall, thin and gray method, `computeAttachPointsFor:and:`. This means that the method is complex and executed in relatively few different situations.

An inspector is accessible from the method by right-clicking on it. The inspector gives the numerical values associated with the visual dimensions.

The situation can be softened by creating new test methods that directly execute this method. This will have the effect of (i) widening the method since new methods will directly call the method and (ii) darkening it since it will be executed more often.

It is important to emphasize that having a long, thin and light-toned method does not necessarily identify a deficiency in the unit tests. The blueprint is not meant to replace the developer's knowledge about the accuracy and completeness of the test. However, it suggests potential improvements.

3.5. Increasing the coverage

Our effort to increase the test coverage and remove dead code in the layout class hierarchy is summarized in Fig. 10. The upper part shows the coverage of the hierarchy for Version 2.2 of Mondrian. Only a bit more than 11% of the methods are executed when running the unit tests.

The part below is the blueprint obtained for Version 2.7 of Mondrian. This new version is the result of applying the enhancements described in this section. More than 87% of the methods are executed by the unit tests.

3.6. Reducing complexity

The experiments we carried out on Mondrian and Hapao did not solely consist of reworking and writing new unit tests. It also has given an opportunity to rethink of crucial parts of the software. A class is visually represented as a set of interconnected methods. This representation conveys a realistic sense of the class inner complexity. Reducing this complexity has also been the topic of our experiments.

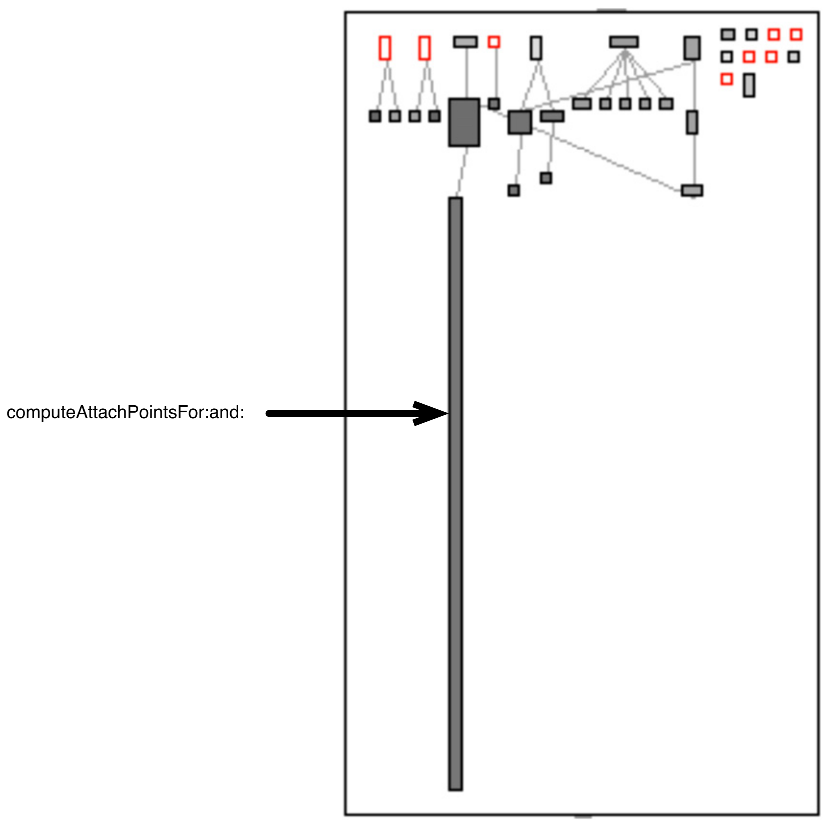


Fig. 9. Blueprint of `MOLineShape`.

Designing a graphical library is a difficult task in which the root of the graphical element class hierarchy is often bloated and complex.[11] Mondrian is not an exception. We have significantly reduced the complexity of the class `MOGraphElement`, the root of the graphic class hierarchy. Fig. 11 shows the evolution of `MOGraphElement` from Version 2.10 and 2.17 of Mondrian. In the version 2.10, `MOGraphElement` contains 143 methods and 18 variables (not represented in the test blueprint). In the last version of Mondrian, `MOGraphElement` contains 101 methods and 12 instance variables. Dead code has been removed and many methods have been relocated to other classes of the hierarchy according to their responsibility.

These operations were carried out via the numerous actions available from the test blueprint produced by Hapao and from Pharo tools, accessible by right clicking either on a class or a method.

3.7. Summary

Thanks to Hapao and its test blueprint, the Mondrian application has been improved in two complementary ways:

- the test coverage of the graph layout hierarchy has substantially increased since its last version. More than 87% of its methods are executed by Mondrian unit tests, and crucial methods defined in the top root class are executed on many different objects and by many different methods.
- the complexity of `MOGraphElement` and other classes has been significantly reduced (by applying code refactorings and removing dead code) and their coverage has increased.

The following section describes the way Hapao is integrated into the Pharo programming environment.

4. Programming environment integration

To be effective, developers have to rapidly and effectively switch from the test blueprint given by Hapao and the standard programming tools in order to take action without loosing the context. This context switch has to be started without loosing the focus of the current activity.

[11] This is the case for the `Morph` class hierarchy in Squeak (http://www.oldenbuettel.de/squeak-doku/Morphic-Kernel/Morph.html) and `JComponent` in Swing [6].

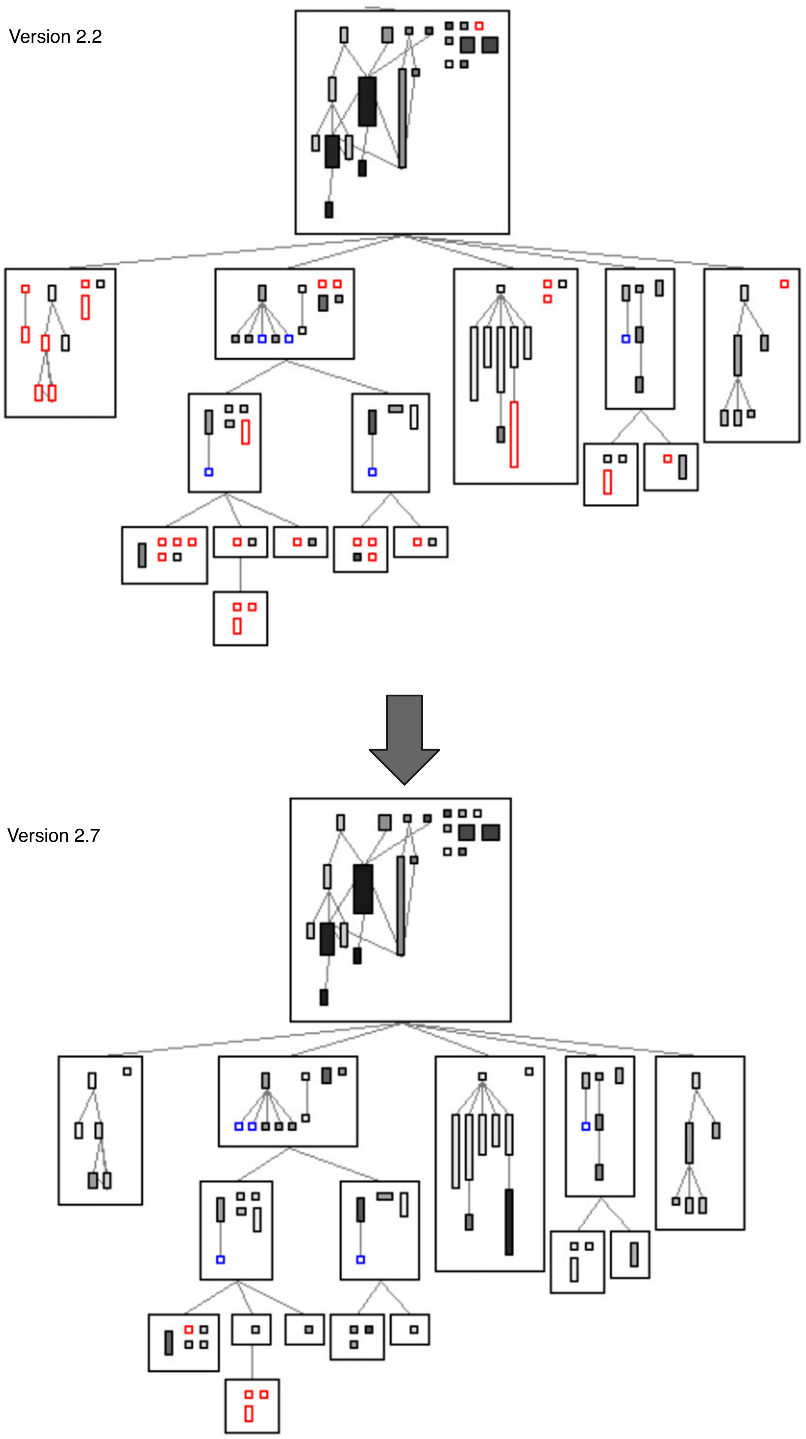


Fig. 10. Increase of the test coverage for the abstract layout class hierarchy.

4.1. Navigation

The graphical user interface of Hapao has been designed to support a close interaction loop between the test blueprint and the standard programming tools. Through our studies, we found that the following navigation actions are the most frequently used:

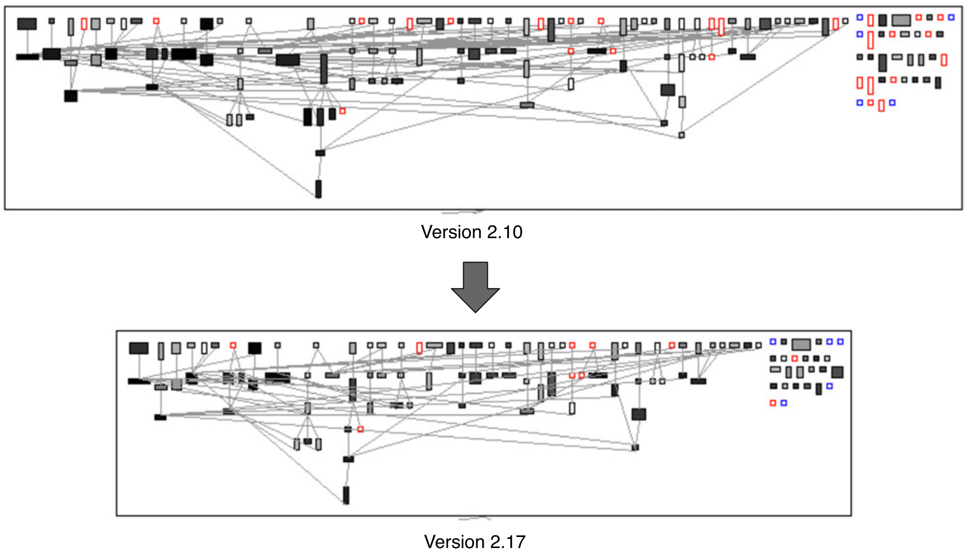


Fig. 11. Reduction of `MOGraphElement` complexity.

- Jump from a system browser to the test blueprint. An option to jump to HAPAO is accessible by right-clicking on a class definition in the system browser (the standard tool to write and browse code in Pharo). The class is selected in the test blueprint.
- Jump to a class or method definition from HAPAO.
- Jump to the definition of unit tests that reference the selected class from HAPAO.
- Unit tests may be executed within HAPAO to immediately see the difference in the blueprint.

All these navigation options are accessible via contextual menus offered when right-clicking on an element in HAPAO or in the system browser.

4.2. *Contextual information*

Waving the mouse above a method displays a contextual popup window and the class border color turns red (Fig. 12). This popup window shows the full name of the method (`MOFormsBuilder>>boundsOf:` in the figure), the invoking methods and invoked methods.

The popup window focuses on the method from which it has been produced. The method is located at the center of the popup window. Above are the methods that execute the method, below are the methods that are executed by it. Test methods are indicated with a green border.

The contextual popup window gives essential information about the context in which the method is used. For example, the method `boundsOf:` is invoked by three test methods and invokes many other non-complex methods (probably utility methods).

The window is displayed after a short delay to avoid image flickering when moving the mouse cursor over many methods.

5. Experience on building HAPAO

The core of HAPAO has been in development for more than one year. It has experienced several drastic changes including replacement of the instrumentation layer and multiple redesigns of the graphical layout used in the test blueprint. This section presents our experience on building HAPAO.

Instrumentation by wrapping methods. The Pharo programming language offers a mechanism to "wrap" methods [7]. The idea is to reflectively replace entries in a method dictionary to insert handcrafted methods. We use this mechanism to gather runtime information.

This mechanism is comparable to an around advice in aspect-oriented programming [8]. However, through Pharo this is achievable without using additional language, library or extending the programming environment. We were able to use standard development tools to develop and debug the instrumentation and profile the gathered information.

Prototyping the visualization. Mondrian enables a quick prototyping of a visualization. We were able to quickly sketch several versions of the test coverage visualization before agreeing on what the test blueprint should be.

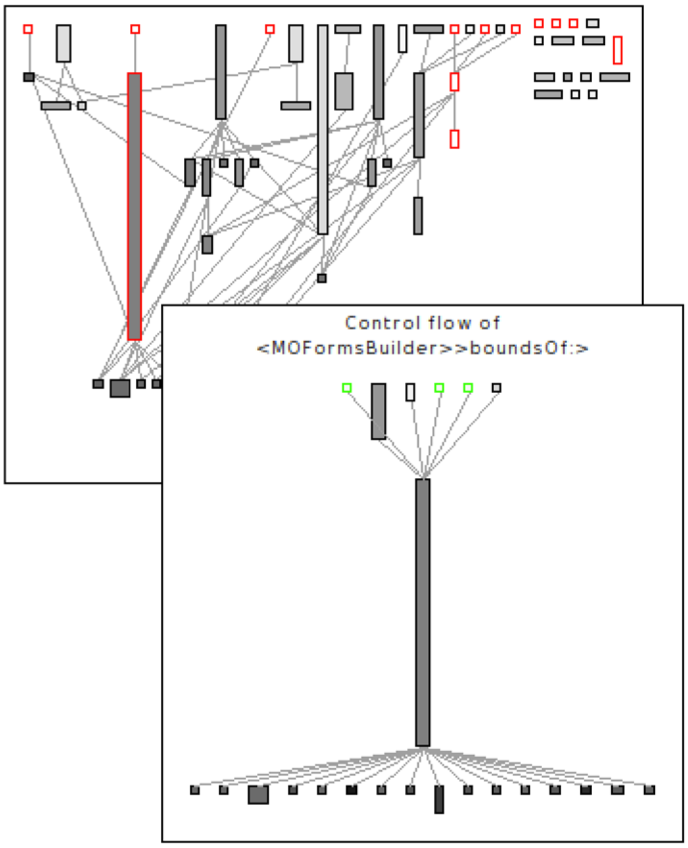


Fig. 12. Contextual popup window.

Integration in the programming environment. Mondrian supports a large amount of interacting facilities on which we rely to jump from the visualization to the source code. Going from the visualization tool to the code edition has to be carried out with as few clicks as possible. Going from the code edition to the coverage tool is subject to the same constraint.

Profiling the profiler and system classes. We experience several technical obstacles that prevent us from visualizing the test coverage of Hapao itself or system classes, such as the collection library for kernel classes. Doing so creates confusion between the profiled objects and the profiler itself, leading to an endless loop.

We tried to make the distinction between the base and meta-level explicit [9], however we could not solve this problem with the resources we were ready to allocate.

Source code coverage. Source code line coverage is a popular way to assess the test coverage, despite its shortcomings.[12] Hapao approximates the source code coverage by coloring in the source code the method calls that are covered. Unfortunately, this approximation is inaccurate with method calls made toward classes defined outside the instrumented application (*e.g.*, libraries) and in presence of several calls of the same method.

As future work, we plan to use Bifröst[13] [10] to attach a meta-object at each node of the abstract syntax tree. Allowing us to gather the necessary information.

Accidental vs. essential complexity. Building a tool to assess test coverage is a difficult task. The metrics we picked and the visual presentation we designed are the real value of Hapao. All this research work constitutes what we qualify as the *essential* complexity of Hapao. In addition, a number of annex problems have to be solved to make the profiler and the visualization work. This *accidental* complexity includes the instrumentation of the application, integration in the programming environment, associating the runtime information with the application source code, just to name a few.

Pharo reduces this accidental complexity by having an homogeneous programming environment and a simple virtual machine. Source code is easily instrumented and runtime information is accessible as any plain objects, living in a unique common memory space.

12 http://nedbatchelder.com/blog/200710/flaws_in_coverage_measurement.html.
13 http://scg.unibe.ch/research/bifrost.

6. Related work

Benefits of assessing the test coverage are widely recognized; it is therefore not a surprise to see a consequent amount of related work. We divide this section into two parts: the first one focuses on the visualization, more specifically how test blueprint compares with others. The second part speaks to the comparison of the tools.

6.1. Test coverage visualization

Tarantula. Similarly to Hapao, Tarantula[14] [11] is a tool to visualize the result of unit tests. It displays the relative success rate using coloring for each line of code. A statement that is executed by a failed test is colored in red. Similarly, a statement that is executed by a passed test is colored in green. Tarantula has 6 different modes to render the test execution. In its continuous mode, which displays a hue fading from red to green, parts of the application that are likely to be the cause of the test failures are red. One character in the source code is displayed as one pixel. This kind of microprint [12] representation is able to display a large amount of source code.

Tarantula has a different focus from that of Hapao since Tarantula is designed to help locate the cause of a test failure. Tarantula is helpful in assessing the coverage since the coloring indicates which portion of the application is covered; uncovered portions are left in gray. However, it does not indicate how well a system is covered, which is precisely the objective of Hapao. In addition, Tarantula is designed for visualizing source code in C, thus not suitable for analyzing object-oriented applications.

Gammatella. Gammatella [13] is a tool to visualize program-execution data, including coverage, exception-related execution and profiling. Three levels are used for the representation: a statement level that shows colored source code; a file level that uses a microprint to visualize source code files; a system level that employs a tree map to represent the data distribution on the whole system.

Gammatella offers three visually distinct representations to support micro and macro analyses. In contrast, Hapao offers a large amount of navigation options and contextual information to cope with a large amount of data. Hapao favors the switching of contexts that are similarly represented. Although we have not empirically compared the two approaches, we suspect that Hapao incurs a lower navigation effort.

6.2. Test coverage tools

The large number of available tools illustrates the importance of test coverage by practitioners. We have reviewed many of them, including EMMA,[15] JCover,[16] eCobertura,[17] GroboCodeCoverage,[18] JCoverage,[19] Parasoft Jtest,[20] PurifyPlus,[21] Semantic Designs,[22] TCAT,[23] Quilt,[24] NoUnit,[25] InsectJ,[26] Hansel,[27] Jester,[28] JVMDI Code Coverage,[29] JBlanket,[30] Coverlipse,[31] McCabe IQ,[32] Code Coverage,[33] JavaCov,[34] LDRA TestBed,[35] PiTest,[36] TestCocoon,[37] Testwell CTC++,[38] Scala Code Coverage

14 http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/.
15 http://emma.sourceforge.net.
16 http://www.mmsindia.com/JCover.html.
17 http://ecobertura.johoop.de.
18 http://java-source.net/open-source/code-coverage.
19 http://www.jcoverage.com.
20 http://www.parasoft.com/jsp/smallbusiness/tool_description.jsp?product=Jtest&.
21 http://www-01.ibm.com/software/awdtools/purifyplus/.
22 http://www.semdesigns.com/Products/TestCoverage/.
23 http://www.soft.com/TestWorks/Products/TCAT-Java/tcat.java.html.
24 http://quilt.sourceforge.net/quilt-0.4/.
25 http://nounit.sourceforge.net/.
26 http://insectj.sourceforge.net.
27 http://hansel.sourceforge.net – Note that most of the code is taken from Gretel, http://www.cs.uoregon.edu/research/perpetual/dasada/Software/Gretel.
28 http://jester.sourceforge.net.
29 http://jvmdicover.sourceforge.net.
30 http://csdl.ics.hawaii.edu/Plone/research/jblanket.
31 http://coverlipse.sourceforge.net.
32 http://www.mccabe.com/iq_testteam.htm.
33 http://www.dynamic-memory.com/products_CodeCover_htm.php.
34 http://www.alvicom.hu/eng/javacov.php.
35 http://www.ldra.com/softwaretesting.asp.
36 http://pitest.org.
37 http://www.testcocoon.org/index.html.
38 http://www.testwell.fi/ctcdesc.html.

Tool[39] and Koalog.[40] We have found these tools via web search engines, programming environment descriptions and scientific publications.

Due to space limitations, we report only on a few of them, those which we judge as the most representative: Cobertura, Agitator and Clover. Note that Agitator and Clover have proprietary licenses while Cobertura is considered open source (Apache Software License).

Cobertura. Cobertura[41] is a widely used Java tool that calculates the percentage of code accessed by tests. As well as xml reports, Cobertura produces javadoc-like html pages,[42] making them easily browsable from a web browser. Cobertura may be launched from the command-line, integrated into an Ant task[43] or into Maven.[44]

Reports produced by Cobertura are not directly accessible from the programming environment. Even though most Java programming environments allow for browsing html sites, Cobertura reports are disconnected from the source code and the code editing environment. For example, it is not possible to jump from one to the other, whereas Hapao supports such essential navigation actions.

As already mentioned in the introduction (Section 1), Cobertura considers a method/statement as covered or not. Hapao gives a context, expressed in terms of call-flows, cyclomatic complexity, number of invocations and number of different object receivers, intended to help a developer decide whether the method or statement is well covered.

As the unique indication of complexity, for each class Cobertura shows the average cyclomatic complexity of all its methods. Hapao provides a graphical representation of class interns, which offer a more intuitive and visual notion of complexity.

Cobertura has been integrated into an Eclipse Plugin. eCobertura enables you to "launch your applications or tests in Cobertura-covered mode directly from within Eclipse, view your source files colored according to the coverage results, browse through the detailed coverage results in a tree view". eCobertura indicates which exact portion of the source code is covered by the unit test. However, it does not indicate which unit tests and test methods test a particular method of the base code.

Agitator. Automatic test generation is a promising technique to automatically augment a test coverage. Agitator[45] automatically creates test cases by synthesizing sets of input. Agitator is distributed as a plugin eclipse, which means it is available to developers during their development.

Hapao takes a different stance from Agitator. Agitator produces an approximation of some missing piece of test to complete a coverage. Hapao offers suggestions on how to improve it. Similar to Cobertura, Agitator considers a method/statement as covered when it has been executed at least once by the unit test.

Clover. Clover[46] is another Java solution for assessing test coverage. Clover records which tests are responsible for a particular coverage. It associates each statement with the tests that executed it. A clickable tree map visualizes the coverage and allows for navigation. Clover produces HTML reports which contains numerous graphs about the coverage and its quality. For example, a scatter plot locates each class along its coverage and complexity; an histogram shows the distribution of the coverage; a name cloud highlights complex untested code.

Clover estimates the quality of coverage by relating its complexity and the associated tests for each class. This takes a different stance from Hapao since the call-flow and control-flow complement the cyclomatic complexity. Contrary to Clover, Hapao is closed integrated in the programming environment. Unfortunately, we have no indication on whether Hapao is more efficient at increasing the coverage than Clover. This is a point we intend to carefully address in future work.

7. Conclusion

Hapao is a tool that helps practitioners to assess and increase the test coverage of their applications. To achieve that goal, the test coverage is visually rendered based on test blueprint, an innovative graphical representation of test coverage. Hapao emerged from the need of the Moose and Pharo community to increase the coverage of their applications. Hapao's development is stable and it has been used to increase the test coverage of many essential applications. We are currently evaluating the effectiveness of the test blueprint to rapidly increase a test coverage.

We believe there is a significant gap between what happens at execution time, expressed in terms of a message sent between objects, and the output of code coverage tools, expressed in terms of covered lines and branches. Traditional test coverage activities essentially concentrate on low level infrastructure. We hope Hapao will raise the consciousness to make post-mortem execution profiles closer to what actually happens at execution time.

[39] http://mtkopone.github.com/scct.
[40] Sadly, no much information is available from its website: http://freshmeat.net/projects/kcc/.
[41] http://cobertura.sourceforge.net/.
[42] http://cobertura.sourceforge.net/sample.
[43] Ant is a project builder for Java (http://ant.apache.org).
[44] Maven is a software project management (http://maven.apache.org).
[45] http://www.agitar.com/solutions/products/software_agitation.html.
[46] http://www.atlassian.com/software/clover.

Acknowledgments

We warmly thank all the people who influenced and participated in the construction of Hapao. We thank Pablo Estefó and Juan Pablo Sandoval for reviewing an early draft of this article. We further thank Renato Cerro for his proofreading. This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile, and by FONDECYT number 1120094.

Appendix. Supplementary data

Supplementary data to this article can be found online at http://dx.doi.org/10.1016/j.scico.2012.04.006.

References

[1] Q. Yang, J.J. Li, D.M. Weiss, A Survey of Coverage-Based Testing Tools, The Computer Journal 52 (5) (2009) 589–597.
[2] E.R. Tufte, Envisioning Information, Graphics Press, 1990.
[3] M. Lanza, S. Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795.
[4] S. Ducasse, M. Lanza, The Class Blueprint: visually supporting the understanding of classes, Transactions on Software Engineering (TSE) 31 (1) (2005) 75–90.
[5] A. Bergel, F. Bañados, R. Robbes, D. Röthlisberger, Spy: A flexible code profiling framework, in: Smalltalks 2010, 2010.
[6] A. Bergel, S. Ducasse, O. Nierstrasz, Classbox/J: Controlling the scope of change in Java, in: Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'05, ACM Press, New York, NY, USA, 2005, pp. 177–189.
[7] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the rescue, in: Proceedings European Conference on Object Oriented Programming, ECOOP'98, in: LNCS, vol. 1445, Springer-Verlag, 1998, pp. 396–417.
[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Aksit, S. Matsuoka (Eds.), ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming, in: LNCS, vol. 1241, Springer-Verlag, Jyvaskyla, Finland, June 1997, pp. 220–242.
[9] É Tanter, Execution levels for aspect-oriented programming, in: Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development, AOSD 2010, ACM Press, Rennes and Saint Malo, France, March 2010, pp. 37–48. Best Paper Award.
[10] J. Ressia, L. Renggli, T. Gîrba, O. Nierstrasz, Run-time evolution through explicit meta-objects, in: Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010). (October 2010) 37–48 http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/.
[11] J.A. Jones, M.J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the 24th International Conference on Software Engineering, ICSE'02, ACM, New York, NY, USA, 2002, pp. 467–477.
[12] S. Ducasse, M. Lanza, R. Robbes, Multi-level method understanding using Microprints, in: Proceedings of VISSOFT 2005 (3th IEEE International Workshop on Visualizing Software for Understanding). (September 2005).
[13] J.A. Jones, A. Orso, M.J. Harrold, GAMMATELLA: visualizing program-execution data for deployed software, Information Visualization 3 (3) (2004) 173–188.