Test Coverage With Hapao

Vanessa Peña Araya, Alexandre Bergel

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

This paper uses colored figures. Though colors are not mandatory for full understanding, we recommend a colored printout.

ABSTRACT

Testing is an essential activity when developing software. It is widely acknowledged that a test coverage above 70% is associated with a decrease in reported failures. Coverage tools output after running the unit tests the list of classes and methods that are not executed. Simply tagging a software element as covered may convey an incorrect sense of necessity: executing a long and complex method just once is potentially enough to be 100% test-covered. As a result, a developer may have an incorrect judgement on where to focus testing effort.

We present test blueprint, a visual tool to help practitioners assess and increase test coverage by graphically relating execution and complexity metrics.

1. TEST COVERAGE

Any respectable software engineering book will argue that testing is an essential and central activity that has to be continuously exercised when producing software. Numerous frameworks are available for that purpose, including xUnit and TestTypes[1], just to name a few.

A kind of quality assurance comes from testing and the use of metrics gives a quantitative measure of quality. *Test coverage*, one popular metric, is concerned with determining what proportion of a defined piece of computer code is executed during a testing cycle. Test coverage is commonly reported as the proportion of packages, classes, methods and lines of code that are executed by the tests. A software that is well tested is commonly associated with a test coverage of 70%-80% [1].

[1]From Microsoft: `http://bit.ly/f2zzEl`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

The criteria that are commonly used are statement coverage, branch coverage, and path coverage [2]. Identifying uncovered statements, branches and paths indicate where the test effort has to focus on next.

A surprising fact from common test coverage criteria is that all the software elements considered when assessing a test coverage have the same relevance. A method, a statement or a branch is simply labelled as covered or not, and thus, independently if these syntactic elements are complex or not, dependent on other elements, or if they are exercised in many different situations. As a consequence, statement, branch and path coverage say that executing a method only once is enough to label it as 100%-covered. This is clearly inadequate in case of a complex or important method because indicating a full coverage gives the wrong signal to the developer. This paper is about fixing this situation by proposing an effective visual representation.

The intuition explored in this paper is that if a "complex and useful" piece of software is tested in many "different situations", then it is probably well tested. On the contrary, if a "complex code" is executed "too few times", then it is probably under tested. This takes on a fairly different stance from classical code coverage tools since we are not interested only on whether each instruction and branch of the code has been executed, but whether or not it has been *sufficiently* executed in different situations. We have identified five patterns to efficiently drive a test coverage assessment and increase effort.

Most approaches to testing use branch coverage to decide on the quality of a program test suite [2]. Test Blueprint takes a different stance by favoring visual patterns over coverage formal model.

Test blueprint is a polymetric view [**?**, **?**] has been implemented in Hapao[2], a test code coverage for Pharo. Pharo[3] is an emerging object-oriented programming language that is very similar to Smalltalk, syntactically simple, and has a minimal core. We have used Test Blueprint to increase the coverage of a number of applications; the following illustrates our points on two case studies.

2. TEST BLUEPRINT

Test Blueprint is a visual aid for practitioners to assess and increase the test coverage of their applications. Before showing Test Blueprint on a real world example, we first introduce the visualization on a small but representative example, given in Figure 1.

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

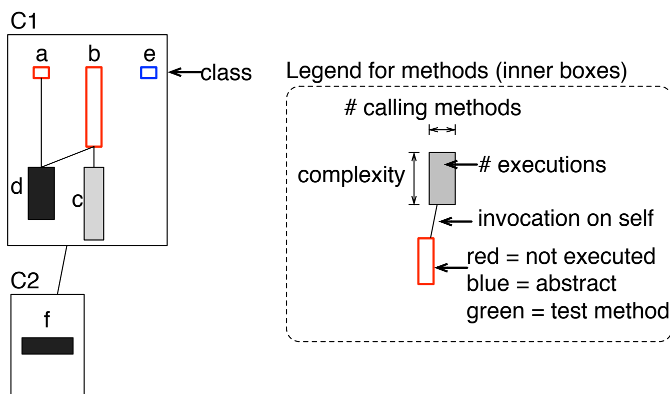

Figure 1: Test blueprint description.

Large boxes represents classes (`C1`, `C2` and `T`). Inheritance is indicated with a connecting line between classes. Subclasses are below their superclass. `C1` is the superclass of `C2`. The superclass of `C1` is not part of the analysis.

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 along fives dimensions:

- height is the cyclomatic complexity of the method. As the method may take different paths at execution time, the higher the box will be (*e.g.,* Method `b`). The minimal visual representation of a method is a 5 x 5 square. A method with an empty body is therefore represented as such.
- 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. A thin method (`a`, `b`, `c`) means the method has been executed zero or few times.
- 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.
- a red border color (light gray on a B&W printout) means the method has not been tested (*i.e.,* executed by the tests) (`a`, `b`). A blue border indicates abstract methods. A green border indicates that the method is a test method, defined in a unit test. Note that a unit test may contain methods that are not test methods; utility methods for example.
- 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 methods `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 call-flow is scoped to the class. Call-flow is statically determined from the abstract syntax tree of the method. Calling methods are located above the called methods (*e.g.,* `a` is above `d`).

Each choice made for the design of test blueprint is justified in the following sections.

2.1 Coverage evolution

We have undertaken a major effort to increase the coverage of the Moose test suite[4].

Figure 2 shows the evolution between the version 13 of the core test suite, before we started our coverage increase, and the version 48 of the test suite, after our effort. Version 13 comprises 15 unit tests and 176 test methods, which covered 63.54% of the package `Moose-Core`. Version 48 of the test suite raises the figures to 23 unit tests, 252 test methods, totaling a coverage of 86.07%.

The `MooseElement` class hierarchy has evolved during our effort. Producing new unit tests offers the opportunity to reconsider the relevance of each uncovered method: meaningless and obsolete code is removed. We started our effort with the version 313 of the package Moose-Core. This package comprises of 27 classes and 467 methods. The version 326 is cleaner, which comprises of 26 classes and 440 methods.

2.2 Complexity reduction

The internal representation of a class offered by Test Blueprint is effective at guiding a complexity reduction effort.

Figure 3 shows the evolution of a central class in the MetacelloBrowser application[5]. While we were increasing the coverage of the application, we exercised a number of code refactorings and dead code removal. Version 1.58.1 on the left-hand side contains 69 methods, where only 28 are covered (painted in gray), representing a coverage of 40.57%. This version contains a very tall uncovered method. This method is much more complex than others because of its size (it has a cyclomatic complexity of 15 whereas other methods have a complexity ranging from 2 to 7).

Version 1.58.9 on the right-hand side contains 66 methods, where 40 are covered, bringing the coverage to 60.60%. The complex method has been cut down into pieces, shorter in length and easier to test.

3. HAPAO

Test blueprint is implemented in Hapao, a test code coverage tool implemented in the Pharo programming language. Hapao is designed to consider each of the requirements given above.

Figure 4 is a screenshot of Hapao. The window title shows the version of the application being visualized. The tool bar contains a number of options for exporting; zooming; running the tests; getting statistics; opening a new window on the same software; getting help. Right-clicking on a class opens a menu with navigations options.

4. REFERENCES

[1] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of ESEM'09*, IEEE.

[2] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[4] `http://www.moosetechnology.org`

[5] `http://www.squeaksource.com/MetacelloBrowser.html`

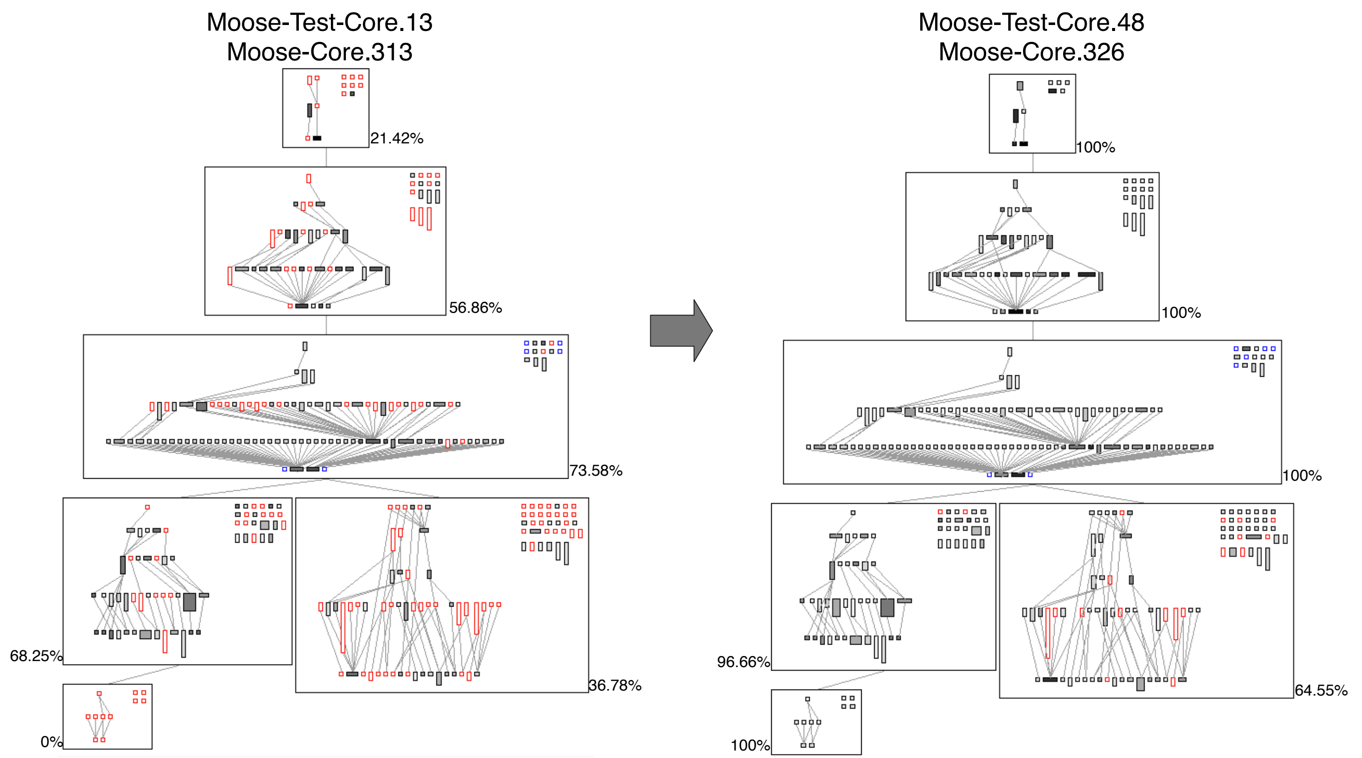

Figure 2: Evolution of the `MooseElement` class hierarchy.

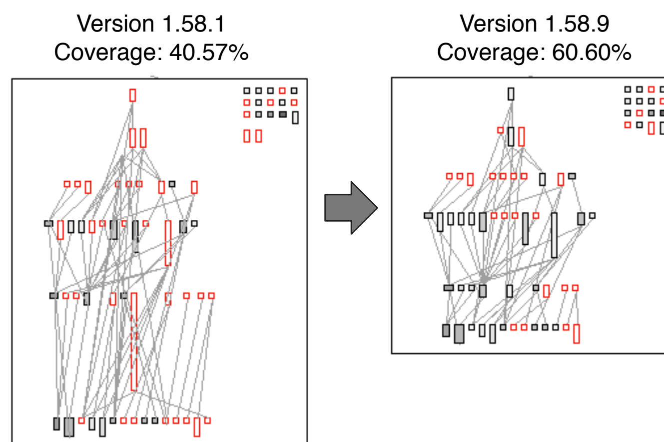

Figure 3: Complexity reduction in MetacelloBrowser.

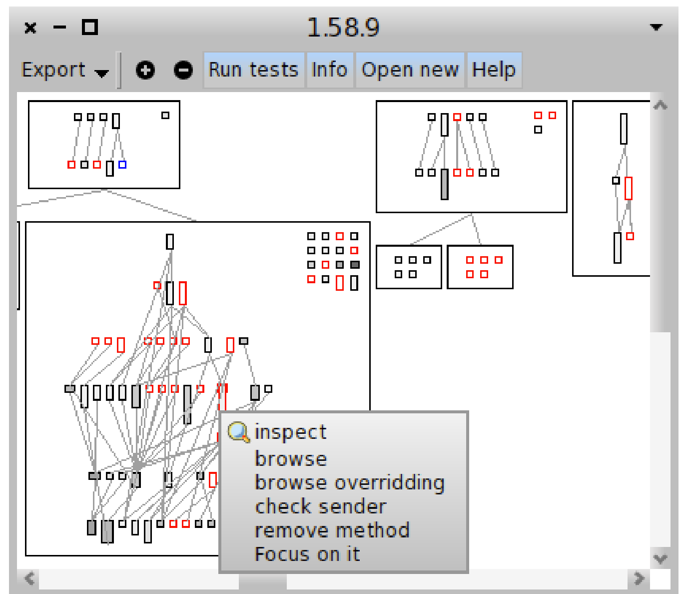

Figure 4: Hapao main window.