Test Blueprint: An Effective Visual Support for Test Coverage

Vanessa Peña Araya[1]
vpena@dcc.uchile.cl

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

ABSTRACT

Test coverage is about assessing the relevance of unit tests against the tested application. It is widely acknowledged that a software with a "good" test coverage is more robust against unanticipated execution, thus lowering the maintenance cost. However, insuring a coverage of a good quality is challenging, especially since most of the available test coverage tools do not discriminate software components that require a "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 shapes methods and classes, which indicates to the programmer whether more effort on testing is required.

This paper presents the essence of Hapao using a real world case study.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Measurement, Verification

Keywords

Testing, Pharo, Coverage, Visualization

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 [5] and TestTypes[1], just to name a few.

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

A kind of quality assurance comes from testing and metrics, a quantitative measure of quality. *Test coverage*, one popular of such metrics, is concerned with determining what proportion of a defined piece of computer code has actually been executed during a testing cycle. Test coverage is commonly reported as the percentage of the packages, classes, methods and lines of code of the application that are executed by the tests. A software that is well tested is commonly associated with a test coverage in the 70%-80% range since increasing statement coverage becomes difficult and is not cost effective [7].

One important question in software testing is identifying the criterion that defines what constitute an adequate test. The criteria that are commonly used are statement coverage, branch coverage, path coverage and mutation adequacy [9]. *A stricken fact from common test coverage criteria is that all the considered software element have the same relevance in a test coverage report.* A method, a statement or a branch is covered or not, and thus, independently whether these syntactic elements belongs to the core of the application or in obsolete components. Moreover, the test coverage is determined from a binary information for each syntactical element: whether it has been executed or not.

The intuition exploited in my work is that if a "complex and useful" piece of software is executed in many "different situations", then it is probably well tested. On the opposite, if a complex code is executed too few times, then it is probably under tested. This takes a fairly different stance from classical code coverage tools since we are not interested only on whether each instruction and branches of the code has been executed, but whether it has been *sufficiently* executed in different situations.

2. TEST BLUEPRINTS

We propose *Test Blueprints*, a visual aid for practitioners to assess the test coverage of their applications. Prior to study a real world example, we first introduce the visualization on a contrived but representative example, given in Figure 2.

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 class border indicates a unit test.

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.

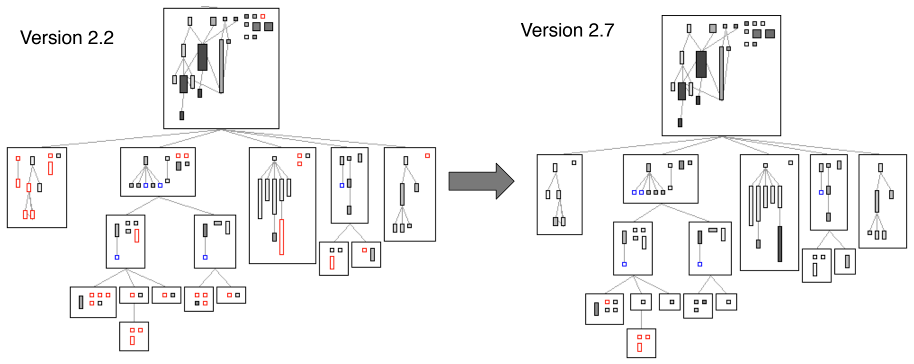

Figure 1: Test coverage evolution for Mondrian layout hierarchy.

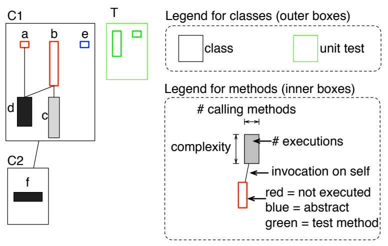

Figure 2: Test Blueprint description.

More the method may take different paths at execution time, higher 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. A thin method (`a`, `b`, `c`) means the method has been executed zero or a few amount of 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 executed (`a`, `b`). Blue border indicates abstract methods. 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.

The coverage of a method is determined by the relation between its cyclomatic complexity and the number of different contexts the method is tested on. The higher the number of different contexts the method have been exposed to, the lower is the number of unexpected behaviors and therefore the more robust its implementation is.

Cyclomatic complexity tells how many different independent paths a method has. Based on this, we can have an idea of how many different contexts a method can be exposed to be sufficiently tested. A complex method then, should be tested under many different contexts to try, at least once, all the different configurations it may be exposed to.

The contexts in which a method may be executed are given by the relation between the number of different methods that call it when running the tests and the number of times the method has been executed. This in under the assumption that a method can be called many times by the same method, but this does not necessarily give it different environments to be proved at.

Dependence by calls to or from other methods also determine the coverage of a method. If a method is linked by a `self` call to another, its behavior depends on it. So if a linked method has a low coverage, it may suggest that it has to be checked for the reliability of its caller or callers. This kind of relation also leads to identify patterns between methods: if a particular method has a similar structure to another, they may be linked in some way and this can be exploited replicating test case's structures for methods that are not covered.

The inner complexity of a class can be observed by the density of this edges between methods. An area dense in edges may be a candidate for a simplification.

Additionally, information like the green color for test class and methods are offered to give a global understanding of the software being analyzed and to help to focus on relevant areas. Methods like tests or abstracts are not entities for which their coverage is primordial when evaluating the test coverage of a software.

We have used Test Blueprint to increase the coverage of a number of software, including Mondrian[2], Merlin[3] and other tools of the Moose software analysis platform[4].

Our effort to increase the test coverage and remove dead code in the layout class hierarchy is summarizing in Figure 1. The left part shows the coverage of the hierarchy for Version 2.2 of Mondrian. Only 11.68% of the methods are executed when running the unit tests.

The right part is the blueprint obtained for Version 2.7 of Mondrian. This new version is the result of applying

[2]`http://www.moosetechnology.org/tools/mondrian`
[3]`http://www.squeaksource.com/Merlin.html`
[4]`http://www.moosetechnology.org`

enhancements using the metrics described in this section. 87.60% of the methods are executed by the unit tests.

3. CODE COVERAGE SCENARIOS AND PATTERNS

We have identified two essential steps when moving an application from being uncovered to covered: (i) increasing the coverage by testing untested software elements; (ii) strengthening weakly tested software elements. Each of these steps is described below in terms of a visual pattern from the Test Blueprint, including an illustration and a list of actions to increase the coverage.

3.1 Increasing the coverage

We have identified four patterns that represent situations a developer faces when uncovering software elements that have to be covered by unit tests.

Uncovered Callers

This pattern is characterized by one or more uncovered caller methods. It is graphically represented by red-bordered white boxes being above a dark one. This pattern is the most frequent one.

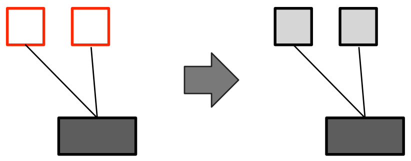
Figure 3: Uncovered callers pattern

For this pattern, two actions are available to increase the coverage:

- enhance the test that directly calls the black method, if it does not interfere with the intention of the test
- write new tests for the untested method

The right-hand side of Figure 3 shows the evolution of the blueprint, when one of this decisions are taken: the two methods will be covered.

Missing Branches

The missing branches pattern is characterized by uncovered called methods, graphically represented as boxes with a red border joined to boxes with a black border above them (*e.g.,* left-hand side of Figure 4). This situation stems from conditional statements or loops that have uncovered branches.

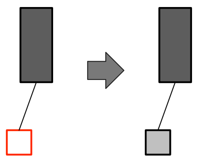
Figure 4: Missing branches pattern

Three actions may be taken to increase the test coverage:

- enhance the test that calls the dark method to invoke the uncovered methods, if the intention of the test is preserved
- write new tests that call the dark method with the adequate environment and parameters to invoke the untested branch
- write new tests that directly invoke the untested method. Note that this action may be taken only if a test shows that the red method can effectively be invoked by the dark method.

The right-hand of Figure 4 shows the evolution of the graphical representation of the pattern.

Uncovered Chain

This pattern is defined as a chain of self-calls with more than one uncovered method. It is depicted as a sequence of connected red boxes (*e.g.,* left-hand side of Figure 5).

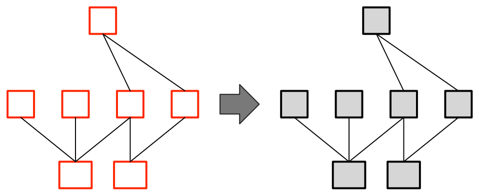
Figure 5: Uncovered Chain pattern

The test coverage is increased by writing new tests for the roots of the chain. This will have the benefit of covering the complete or a subset of the chain. We designate a root method as a method that is located above other connected methods. After covering all the roots, this patterns looks like the one on the right-hand side of Figure 5.

Isolated Methods

This fourth pattern represents uncovered and unconnected methods. Those methods are graphically represented as unconnected red-bordered boxes (*e.g.,* left-hand side of Figure 6). Methods are ordered along their cyclomatic complexity.

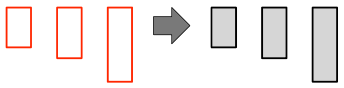
Figure 6: Isolated Methods pattern

Increasing the coverage is achieved by:

- adding extra calls of the uncovered methods to existing tests if the tests intention is preserved
- writing new tests that target these methods

Testing isolated methods puts the developer in a situation where a choice has to be made on which method to test first. The cyclomatic complexity of each method, represented as the box's height, is a criteria for the developer to select which methods to test first. Although we have not conducted an empirical study, apparently people often opt for simple methods to test first.

3.2 Strengthen the coverage

The key intuition we exploit to strengthen the coverage is: if a method is executed in "many" different contexts

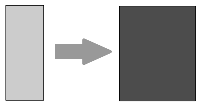

Figure 7: Weak coverage pattern

then it is probably "well" tested. We refer to "context" the message context in which the method is executed, which is the combination of the object receiver and the provided arguments. The larger the number of distinct objects calling the method, the larger the possible scenarios the method will be exposed to, making possible to discover more erroneous behavior.

This pattern is graphically represented with a light and horizontally narrow gray box, as explained in previous sections, the number of executions and the number of different callers are given by the color and the width respectively (*e.g.,* left-hand side of Figure 7). Note that this pattern is independent of the call flow of other methods. It may therefore be composed with other patterns.

It is up to the developer to decide whether or not a method deserves a better coverage based on its importance or/and complexity. Complex and few called methods are good candidates to receive attention.

After executing a new test, or improving o, the pattern appears as in the right-hand side of Figure 7.

4. RELEVANCE OF PATTERNS TO INCREASE THE COVERAGE

In the previous sections we identified four patterns to effectively increase the test coverage for a group of methods. Each of the patterns is associated to some actions that a software engineer may take to increase the coverage (Section 3.1).

The actions associated to a pattern have a different impact on the coverage than the actions of another pattern. As a concequence, not all the patterns have the same effectiveness in increasing the coverage. Since this is a costly activity, it is therefore crucial to prioritize the patterns to consider.

To measure the patterns effectiveness we performed a study where two evaluations were conducted: (i) we measure the test coverage increase for each pattern and (ii) we identify the optimal sequence to use the patterns which permits to do the testing process in the most efficient way [6].

We conducted simulations on a basket of six applications[5]. These applications are all written in Pharo and cover the most popular features of Pharo (data manipulation and algorithm, graphic rendering and network).

In the following sections a summary of our study it is presented for both evaluations.

4.1 Measuring pattern effectiveness

We measure the effectiveness of a pattern by increasing each application coverage by solely applying an action associated to the pattern. Instead of writing functional tests, we artificially increase the coverage by marking methods as tested. Other researchers, in a different field, have adopted a similar way to carry out their simulation [2].

We successively run four simulations for each of the six applications. Each simulation measures the increase of the test coverage after testing a picked method with a new test. The difference between the four simulations is the strategy to pick a method, where each one reflects each of the four patterns. The strategies are:

P1 - picking an uncovered method that calls a covered method (red method in the *Uncovered callers* pattern, Figure 3).

P2 - picking an uncovered method with a covered caller (red method in the *Missing branches* pattern, Figure 4).

P3 - picking an uncovered method root that has one or more uncovered branches (top red method in the *Uncovered Chain* pattern, Figure 5).

P4 - picking an isolated uncovered method (red method in the *Isolated Methods* pattern, Figure 6).

To compare the strategies, we define their effectiveness as the average number of newly covered methods per application of the strategy.

In the study we concluded that strategy P3 is the most effective for all the tested applications, with effectiveness values ranging from 1.72 to 1.85. On average, 1.78 new methods are covered by each new test using this strategy. Strategy P4 is, not surprisingly, always the least effective one with a constant effectiveness value of 1. P1 and P2 have effectiveness values between 1.00 and 1.42. In most cases, P1 was more effective than P2, but not always.

4.2 Sequencing the patterns

Our simulation empirically asserts that the picking strategy P3 is the most effective, for the applications we consider. However, the amount of methods that can be picked by P3 is in the average 8% of the total amount of methods. After testing all the methods picked by P3, another picking strategy has to be used for further testing.

The natural question that arises is identifying the optimal sequence of patterns to consider when increasing the coverage. From the set $\{P1, P2, P3, P4\}$, there is $4! = 24$ permutations. Sequencing the patterns along their effectiveness is likely to produce the optimal pattern sequence. We verify this by asserting that the sequence pattern P3, P1, P4, P2 is the most effective sequence to quickly increase the test coverage.

We designate the sequence P3, P1, P4, P2 as optimal. We empirically show the optimality of the sequence by testing our applications using different sequences.

For the 6 applications, we carried out 4 simulations. The first one called *optimal* in which we use the optimal pattern sequence. The 3 other simulations use a random combination, different than the optimal sequence. The 24 different sequence have been generated, thus covering all the permutations. We measure the test coverage against the amount of generated tests: the strategy that we designate as optimal indeed produce the higher coverage.

5. RELATED WORK

In this section we first revise some of the literature related with Test Blueprint and then some test coverage tools.

Horwitz [3] proposed three metrics to increase test coverage by reasoning on predicate and branch statements. However,

[5]The applications are Roassal, Mondrian, Seaside, Citezen, Pier, Zinc. These software are open source and have their source code available on `http://www.squeaksource.com`.

this metrics are meant to favor branch coverage. Instead, we adopted metrics to assess the complexity of method and the number of times it has been executed. Horwitz' innovation is estimating the difficulty to provide a missing input. We plan to research her metrics in future work.

There are approaches for prioritizing tests generation ([1, 4]), however they are mainly focused on automatic test generation and require cumbersome graph analysis. Without complex computation, the patterns presented previously and the optimal order in which they can be used, pretend to help testers to increase their coverage efficiently. Hapao visually highlight where a software engineer can start testing and all the methods that can be covered by a new test or the improvement of an old one.

Cobertura[6] is a widely used Java tool that calculates the percentage of code accessed by tests, the percentage of lines and branches covered for each class. Cyclomatic complexity is also provided for each class and package. Cobertura is useful to identify the code areas unreached by unit tests. Test Blueprint provides a more accurate situation of what actually needs to be covered first. Instead of reporting a coverage based on predicate (*e.g.,* whether a code statement has been executed or not), Test Blueprint relates three metrics in a concise visual support.

There is a large number of tools for test coverage. Q. Yang *et al.* [8] surveyed 17 coverage-based testing tools focusing in their characteristics: supported programming languages, program instrumentation overhead and additional features complementary to code coverage. They also mention the importance of assistance in conducting effective testing, prioritizing resources and emphasize the lack of effective tools to address this point. We consider Test Blueprint a solution to this situation.

6. CONCLUSION

Test Blueprint is an effective and intuitive visual representation of the test coverage. It provides a valuable help to practitioners to evaluate their software test coverage and, along with the patters, to choose efficiently their next target when increasing it. We presented a summary of our latest study in which we analyzed how Test Blueprint can be used to reduce time and effort when testing an application.

Test Blueprint is implemented in the Hapao test coverage tool[7]. At the beginning, the tool essentially produced a blueprint from a test execution. Intensively using Hapao and the feedback provided by our users have been key in shaping Hapao. New requirements have emerged, such as the need for additional specialized views to relate tests with tested elements and a full support for interactions in the programming environment.

Hapao was initially freely available only for the Pharo programming language. Due to the positive feedback from the Smalltalk community, in 2012 Hapao was ported to VisualWorks [8] and it is now available in the Cincom public repository. It will also be included in the next comming release (VisualWorks 7.9) as part of the contributed applications. Currently, Hapao is one of the two profiling tools forming the products foundation of Object Profile [9], a growing company with a lot of ideas to develop in the future.

7. REFERENCES

[1] Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 25–34, New York, NY, USA, 1994. ACM.

[2] Marco D'Ambros, Michele Lanza, Mircea Lungu, and Romain Robbes. On porting software visualization tools to the web. *In Journal on Software Tools for Technology Transfer*, 13:181 – 200, 2011.

[3] Susan Horwitz. Tool support for improving test coverage. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 162–177, London, UK, UK, 2002. Springer-Verlag.

[4] J.J. Li. Prioritize code for testing to improve code coverage of complex software. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp. –84, nov. 2005.

[5] Gerard Meszaros. *XUnit Test Patterns – Refactoring Test Code.* Addison Wesley, June 2007.

[6] Vanessa Peña Araya, Alexandre Bergel, and Tobias Kuhn. An effective visual support for test coverage. Submitted, 2012.

[7] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 287–301, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[8] Qian Yang, J. Jenny Li, and David M. Weiss. A Survey of Coverage-Based Testing Tools. *The Computer Journal*, 52(5):589–597, 2009.

[9] 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.

[6]`http://cobertura.sourceforge.net`

[7]`http://objectprofile.com/pier/Products/Hapao`

[8]`http://www.cincomsmalltalk.com/main/products/visualworks/`

[9]`http://objectprofile.com`