Controlled Experiment to Assess a Test-Coverage Visualization: Lesson Learnt

– Experience Report, work in progress –

Alexandre Bergel[1], Vanessa Peña-Araya[1], Tobias Kuhn[2]

[1]PLEIAD Lab, Department of Computer Science (DCC), University of Chile

[2]Department of Humanities, Social and Political Sciences, ETH Zurich

http://bergel.eu

vpena@dcc.uchile.cl

kuhntobias@gmail.com

June 29, 2015

Abstract

Evaluating a software visualization is a difficult and error-prone activity. In this short paper, we report our experience when conducting a controlled experiment to evaluate *test blueprint*, a visualization to assess the test coverage. Our experiment went through two iterations. The first iteration was unfortunately inconclusive, due to some decisions we took that we are now considering as mistakes. After revising our experiment, we obtained exploitable results, which are also matching our intuition.

1 Visually Assessing Test Coverage

Motivation. 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.

Visualization. *Test blueprint* is an innovative visualization for test coverage [1]. It employs an effective and intuitive graphical representation to visually assess the quality of the coverage. A combination of appropriate metrics and relations

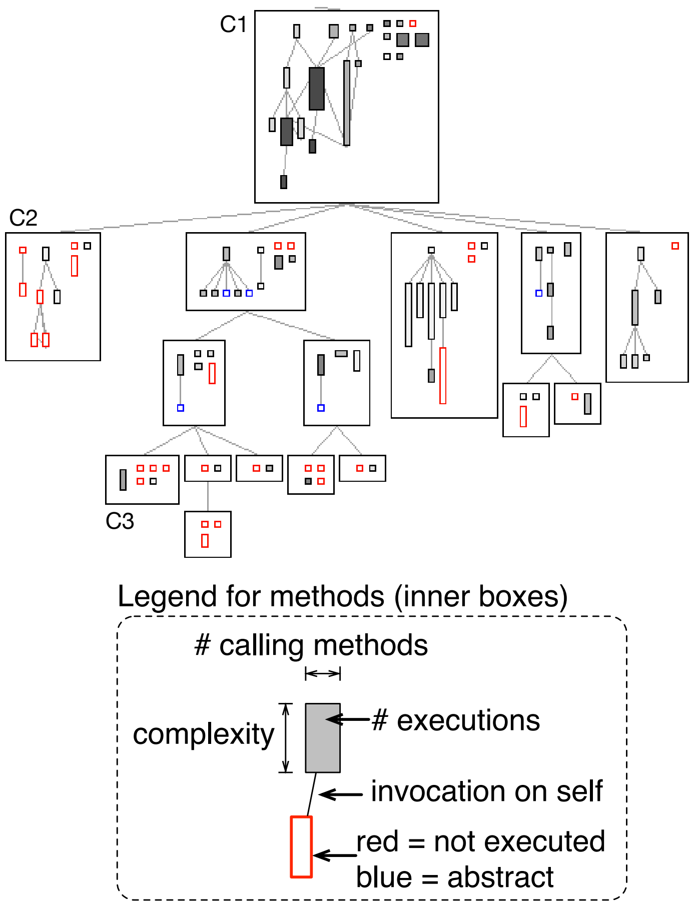


Figure 1: Test blueprint.

visually shape methods and classes, which indicates to the programmer whether more effort on testing is required.

Encapsulating boxes represents classes (*e.g.,* `C1`, `C2`). Inheritance is indicated with an edge between classes. Subclasses are below their superclass (`C1` is the superclass of `C2`). Inner boxes represent methods. `C3` defines six methods. Each method is represented as a small box, visually defined with five dimensions:

- height is the cyclomatic complexity of the method [2]. As the method may take different paths at execution time, the higher the box will be.
- 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 means the method has been executed a few times.

- gray intensity reflects the number of times the method has been executed. A dark method has been executed many times. A light-toned method 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 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 method `method1` contains the expression `self method2`, meaning that the message `method2` is sent to 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.

Evaluating test blueprint. Measuring the impact of the test blueprint on developers has been the topic of a long effort. The remaining of the paper describes the two attempts we carried out. The two hypotheses we are interested in are:

H1 - *Test blueprint helps identifying the method to test in order to maximize the coverage increase.*

H2 - *Test blueprint helps assessing the difficulty to test a class.*

2 First Attempt

Controlled experiment design. As a first attempt to evaluate test blueprint, we conducted a controlled experimented, designed as follows:

- As a base line, we took the visualization produced by EclEmma[1], which we consider as a standard test coverage tool for Java. Figure 2 illustrates this visualization.
- Instead of comparing the tools themselves, we solely focused on the visualization. Evaluating the tools would require more effort, especially since EclEmma has been developed and maintained for a long time period. In addition, evaluating the tools would introduce biases related to the programming language (Java vs Pharo) and to the IDE (Eclipse vs the Pharo IDE). Test blueprint is implemented in the Pharo programming language[2].

[1]`http://www.eclemma.org`
[2]`http://pharo.org`

- The Pharo open source community has multiple times expressed the need of a robust test coverage tool. Since we strongly felt test blueprint was aiming at addressing this need, we decided to directly survey the community. For that purpose, we designated a small web questionnaire for the community members to fill in.
- Participants were asked a number of questions on two sets of classes, one shown with test blueprint and the other with EclEmma. Questions were divided in two categories: (i) classes characterization regarding their easiness to test and (ii) methods having the highest potential to increase the coverage. These two questions are directly related to the two hypotheses. Category (i) contained 8 questions and category (ii) has 6 questions.

Results. The experiment has been carried out with care. Unfortunately, the experiment was not concluding. In particular, no tendencies could be drawn and the hypotheses have been left unverified. We believe this is the result of a number of suboptimal decisions we took:

- Classes that were given to the participants to assess were small. This was made on purpose to avoid measurement bias that would stem from large windows (*e.g.,* participants would need to scroll through the visualization). It is known that visualization helps facing scalability (especially when compared with textual listing). However we did not exploit this.
- Too many questions where asked to the participants. These questions were also imprecise and not directly linked to the hypothesis we wished to evaluate.
- Participants have a natural tendency to be resistant to proposals to improve their environment. Although we were not able to measure it, we had a clear impression that the participants' answers were indicating that they wish no changes.

3 Second Attempt

As a second attempt, we improved the experiment design as follow:

- 12 participants have been selected from the University of Chile and from a local company that is known to use unit tests in their production.
- participations were surveyed on paper and not via a website.
- Only two questions were asked to each participant.
- We have removed some variables in our design. In particular we do not make a distinction between small and large classes and between the inner structure of classes. The reason for this is that 12 participants was not enough to have a representative data set.

This revised experiment provided better results. The controlled experiment indicates that the hypothesis H1 is not verified, despite a better average score of test blueprint against EclEmma. The hypothesis H2 is verified: test blueprint significantly outperforms coverage report listing to indicate the difficulty to test a class.

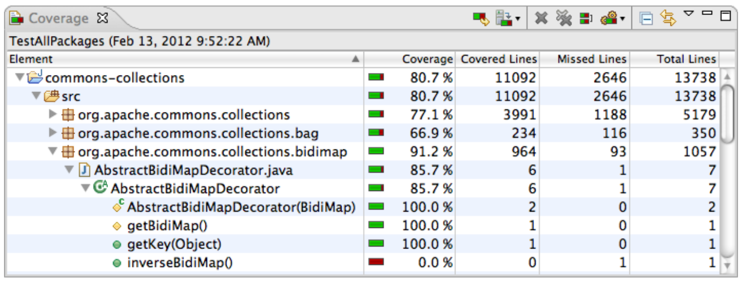
4 Conclusion

This short paper describes the two iterations we have carried out on evaluating a test coverage visualization. We hope the experience we are reporting will be beneficial to other researchers.

References

[1] A. Bergel, V. P. na, Increasing test coverage with hapao, Science of Computer Programming 79 (1) (2012) 86–100. `doi:10.1016/j.scico.2012.04.006`.

[2] T. J. McCabe, A measure of complexity, IEEE Transactions on Software Engineering 2 (4) (1976) 308–320.



TestAllPackages (Feb 13, 2012 9:52:22 AM)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
commons-collections	80.7 %	11092	2646	13738
src	80.7 %	11092	2646	13738
org.apache.commons.collections	77.1 %	3991	1188	5179
org.apache.commons.collections.bag	66.9 %	234	116	350
org.apache.commons.collections.bidimap	91.2 %	964	93	1057
AbstractBidiMapDecorator.java	85.7 %	6	1	7
AbstractBidiMapDecorator	85.7 %	6	1	7
AbstractBidiMapDecorator(BidiMap)	100.0 %	2	0	2
getBidiMap()	100.0 %	1	0	1
getKey(Object)	100.0 %	1	0	1
inverseBidiMap()	0.0 %	0	1	1

Figure 2: EclEmma's test coverage output