HAL
open science

Test Quality Feedback Improving Effectivity and Efficiency of Unit Testing

Michael Perscheid, Damien Cassou, Robert Hirschfeld

To cite this version:

Michael Perscheid, Damien Cassou, Robert Hirschfeld. Test Quality Feedback Improving Effectivity and Efficiency of Unit Testing. C5'12: Conference on Creating, Connecting and Collaborating through Computing, Jan 2012, Playa Vista, California, United States. 10.1109/C5.2012.7 . hal-01118969

HAL Id: hal-01118969

https://inria.hal.science/hal-01118969v1

Submitted on 3 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Test Quality Feedback
Improving Effectivity and Efficiency of Unit Testing

Michael Perscheid, Damien Cassou, and Robert Hirschfeld
Hasso–Plattner–Institute, University of Potsdam
Software Architecture Group
Potsdam, Germany
Email: firstname.lastname@hpi.uni-potsdam.de

***Abstract*—Writing unit tests for a software system enhances the confidence that a system works as expected. Since time pressure often prevents a complete testing of all application details developers need to know which new tests the system requires. Developers also need to know which existing tests take the most time and slow down the whole development process. Missing feedback about less tested functionality and reasons for long running test cases make it, however, harder to create a test suite that covers all important parts of a software system in a minimum of time. As a result a software system may be inadequately tested and developers may test less frequently.**

Our approach provides *test quality feedback* to guide developers in identifying missing tests and correcting low-quality tests. We provide developers with a tool that analyzes test suites with respect to their effectivity (*e.g.,* missing tests) and efficiency (*e.g.,* time and memory consumption). We implement our approach, named PathMap, as an extended test runner within the Squeak Smalltalk IDE and demonstrate its benefits by improving the test quality of representative software systems.

***Keywords*-Unit Tests, Dynamic Analysis, Test Quality Feedback**

I. Introduction

Software testing is an essential development activity to ensure that applications work as expected. Automated unit tests allow developers to specify the proper state and interaction of objects and provide early identification of erroneous behavior [1]. For these tests to be effective in identifying erroneous behavior it is necessary that test cases cover as much as possible of the system code [2]. Another requirement is that developers must execute these tests as frequently as possible and thus they should be fast [3].

Nevertheless, a complete coverage of a system is unfeasible and finding reasons for suspiciously long running tests in a large test suite is hard. Commonly, well tested applications own only a coverage rate around 70-80% [4] and increasing this coverage requires much more development effort [5]. Also optimizing test cases for run-time performance takes time and is often ignored or overlooked by developers to the benefit of other development tasks. To avoid loosing time on these issues feedback about coverage and performance is critical for developers.

Existing coverage and execution profiler tools provide feedback about the quality of test cases but they come with their own limitations. Code coverage tools analyze at the method, statement, or branch level which test cases execute which system parts. This fine granular view of covered code is important for creating new tests. Nevertheless, this view requires developers to manually correlate it with other information such as complexity or author ownership. For instance, few tools distinguish between complex code (requiring more tests) and trivial code, and even fewer identify potential experts most capable of writing new tests for less tested system parts. Moreover, collecting run-time information at statement or branch level comes along with a perceivable performance decrease [6]. Also, by focusing on a specific test case, execution profilers barely allow a comparison of multiple test cases for the purposes of identifying common run-time bottlenecks and memory leaks.

In this paper, we present *test quality feedback* that supports developers in the identification and correction of inadequately tested system parts. Our approach combines multiple high-level views of system, coverage, and profiling information through *effectivity* and *efficiency* feedback. *Effectivity feedback* reveals locations that are worthwhile increasing the testing effort, focusing the developers on important system parts. *Efficiency feedback* reports on run-time performance and memory consumption to speed up the execution of entire test suites with limited effort. To ensure automated, scalable, and fast responses to developers we base our approach on a lightweight and incremental coverage framework.

The contributions of this work are as follows:

- A test quality feedback technique that guides developers in identifying missing test cases and correcting low-quality tests with respect to their effectivity and efficiency.
- A coverage framework, named *Paths*, that provides immediate feedback at the method level and more detailed on-demand feedback at the statement level.
- A realization of our approach by providing integrated tool support for the Squeak/Smalltalk IDE and an assessment of the benefits and costs of our approach by applying it to several existing software systems.

The remainder of this paper is structured as follows: Section II introduces our motivating example and explains contemporary challenges in testing. Section III presents test quality feedback as a guide to improve effectivity and efficiency of test suites. Section IV describes our implementation.

Section V evaluates the practicability and applicability of our approach to arbitrary projects. Section VI discusses related work, and Section VII concludes.

II. Ensuring Adequately Tested Programs

In this section we introduce a motivating example for an incompletely tested application that serves as basis for our discussion of challenges in testing. By this example we demonstrate test quality feedback in Section III.

A. An Introduction to AweSOM

AweSOM is an implementation of a SOM (Simple Object Machine) virtual machine in Squeak/Smalltalk [7]. This research prototype realizes the high-level environment for running and interpreting SOM's file-based Smalltalk dialect.

AweSOM main components compile SOM Smalltalk files, interpret byte code, collect garbage from memory, bootstrap the virtual machine, and provide core functionality. AweSOM's implementation includes more than 4,000 lines of code in 750 methods and 69 classes and is the result of two years of work by four students and one post-doc researcher. The test base consists of 125 unit tests. The unit tests verify each component on its own and cover 76.66 % of all methods in about 20 seconds.

B. Challenges in Testing

Although developers have used test-driven development [1] during the implementation of AweSOM, they ran into challenges regarding their test base. The coverage rate of the system stagnates at about 80 % and developers often find defects that are not covered by test cases. An increase in coverage is difficult to achieve since most of the not covered parts require much more testing effort. Existing coverage tools list around 150 untested methods but their feedback neither includes what is important to invest development time in nor who is able to write missing tests with the least effort.

Furthermore, we observe that running all unit tests requires about 20 seconds. Thus, developers do not run them after each code change and so they get notified of mistakes later than necessary. Profiling specific test cases is insufficient to find common performance bottlenecks: a profiler can neither show similarities nor differences between unit tests.

Numerous studies [6], [8], [4] report similar observations during software testing. On the one hand test coverage motivates developers to write more tests and so it seems to increase the reliability of systems [9]. On the other hand developers complain that they do not know how much code is covered and how well the tested code is [6]. No underlying theory relates coverage with quality and so these studies can best suggest elementary guidelines [8]. For instance, a company might require developers to achieve 80 % statement coverage. Nevertheless, without details about the remaining 20 % it is hard to decide if all important system parts are sufficiently tested. From these studies and our own experience we argue that there is a need for developers to get more feedback about the quality of their tested systems.

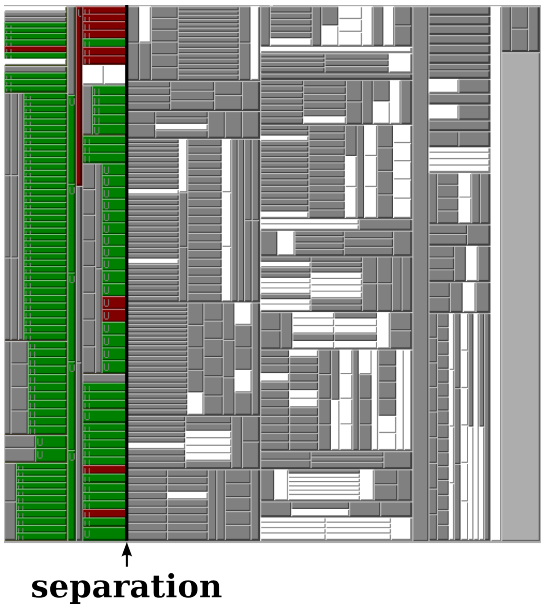

Figure 1. Method coverage of the AweSOM example. The tree map visualization is separated vertically with a black thick line. On the left the green and red boxes represent unit test methods while the other boxes represent helper methods for these unit tests. On the right the boxes represent non unit test methods of the system under observation. A box is dark if at least one test executes the method and bright otherwise.

III. Test Quality Feedback

Through the use of the AweSOM example we present test quality feedback as a guide to improve effectivity and efficiency of unit tests.

A. Effectivity Feedback

Effectivity feedback reveals locations that are worthwhile increasing the testing effort through the following stages:

1) a visualization presents an overview of the entire system, revealing inadequately tested parts;
2) then a developer can use additional static metrics (*e.g.,* method size or complexity) to emphasize suspicious parts that require immediate attention;
3) if necessary developers can request more detailed coverage information for each suspicious part;
4) finally the visualization proposes a list of experts most qualified for writing missing tests for suspicious parts.

We now detail each of these stages using AweSOM as a running example and show how our effectivity feedback leads us to improve AweSOM's reliability.

a) Finding Inadequately Tested System Parts: The first stage consists in presenting a high-level view of the system under observation and its test coverage information in form of a compact and scalable tree map [10]. Such a visualization allows for a high information density compared to a list or class diagram. Figure 1 presents the visualization for the AweSOM example. The visualization represents packages as columns and their classes as rows. Each class represents each of its methods as a box within the class. Packages, classes, and

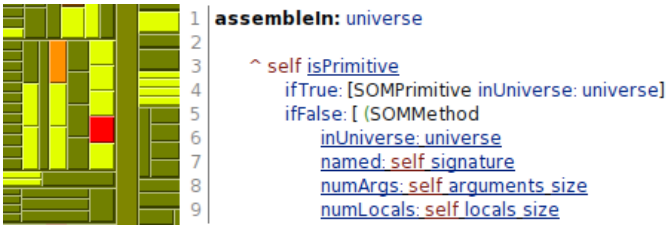

Figure 2. On the left-hand side an extract of a tree map where darkness represents coverage (as in Figure 1) and hue represents complexity. One of the boxes is both bright (inadequately covered) and hot (complex) revealing a suspicious method. On the right-hand side an editor shows the source code of this method. Unit tests cover only the underlined statements.

methods are sorted alphabetically and for a clear separation we distinguish between test classes on the left-hand side and application classes on the right-hand side. A developer can interactively explore the visualization to get more details about a specific method such as its name, the tests that cover it, and metric values (*e.g.,* complexity). The visualization colors a unit test method in green or red if the test respectively succeeds or fails. The visualization uses brightness to indicate system methods coverage (dark indicating coverage). As a result, a box that is either red (indicating failing test) or bright (indicating no coverage) requires attention from the developer. The tree map visualization can represent applications with thousands of methods on a standard screen. This visualization allows hiding some specific methods (*e.g.,* accessors) and summarizing large classes to cope with even larger systems. When a large class is summarized a developer can click its box to get a new and separate tree map visualization dedicated to the class and all its implementation details. The organization of packages, classes, and methods in the tree map makes finding a particular element simple, even for large systems.

Using Figure 1 a developer sees that most failing tests are grouped within a single class (red boxes, top left of the figure). Additionally, a developer sees that some classes are not covered by unit tests at all (bright boxes, bottom right of the figure). Hovering with a mouse on the tree map reveals the names of these classes: `SOMClassTest`, `SOMString` and `SOMSystem`.

b) Emphasizing Suspicious System Parts: The second stage consists in emphasizing suspicious system parts through the use of static source code metrics (*e.g.,* lines of code, complexity, or any other metrics the developer implements). The tree map uses the color hue to represent the method's result for the selected metric from green (for lowest results) to red (for highest results). Additionally, the method's box will be dark if any unit test covers the method. As a result, a system part with a bright and hot color requires more attention from the developer: such parts are more visible because sufficiently tested system parts are hidden by dark colors.

The left-hand side of Figure 2 shows an extract of AweSOM's tree map and some inadequately tested methods. In this figure, the hue describes the method's complexity. The developer sees that one method is both complex (red) and not covered by any test (bright). Hovering reveals that the method name is `assembleIn:` and is part of the `SOMMethodGenerationContext` class.

c) Refining Coverage Analysis: The third stage consists in refining the coverage analysis for a suspicious method emphasized in the previous stage. When a developer selects a particular method in the visualization a new editor pops up and shows the method source code. In the background, our coverage framework, named Paths, executes the tests that cover this method to collect statement-level coverage information. Upon completion Paths updates the editor with the covered statements underlined. As statement-level coverage is costly to compute, we restrict the performance decrease only to methods of interest and offer developers both fast access to method coverage and optionally refined statement coverage.

The right-hand side of Figure 2 shows a statement-level coverage analysis of the suspicious method emphasized in the previous stage. The developer sees that no test executes the branch for creating a primitive.

d) Identifying Experts: Once a developer finds an inadequately tested method the fourth stage allows for the identification of experts for implementing the missing tests. We argue that the required testing effort depends on individual skills and knowledge about the system under observation: Similarly to the debugging activity [11] more experienced developers invent better hypotheses than novices. To find the expert of a particular method our approach requires that developers use a version control system: Our Paths framework mines the version control system and finds the developer with the most commits for this particular method. Our approach can additionally determine an expert by analyzing who last changed the method, who wrote the initial implementation, and who changes the method the most frequently [12]. Depending on projects' needs, we allow developers to choose the proper metric for expert knowledge. When applied to the whole system the visualization assigns each expert a unique color by dividing the hue color wheel depending on the number of experts. As a result of this visualization a developer can find experts to increase coverage of suspicious methods, classes, or packages.

Figure 3 illustrates AweSOM's experts for all methods. Except for the green and red of the unit test methods on the left there are five colors for five experts. A developer sees that the pink expert is better suited to improve coverage of the `SOMSystem` class (bottom right of the figure).

B. Efficiency Feedback

Efficiency feedback identifies run-time and memory bottlenecks of unit tests so that developers are able to improve their test performance with limited efforts. Our approach presents efficiency feedback using the same tree map visualization as for effectivity feedback. The tree map reveals long running and frequently called methods as well as classes that instantiate a large number of objects.

In the following we detail how efficiency feedback works. We describe how efficiency feedback reveals the reasons for

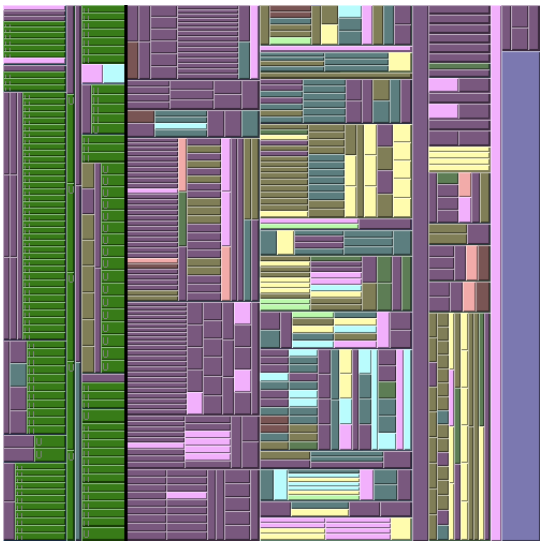

Figure 3. A tree map for the AweSOM example where darkness represents coverage (as in Figure 1) and hue represents authorship (except on the left where green and red indicates unit test status).

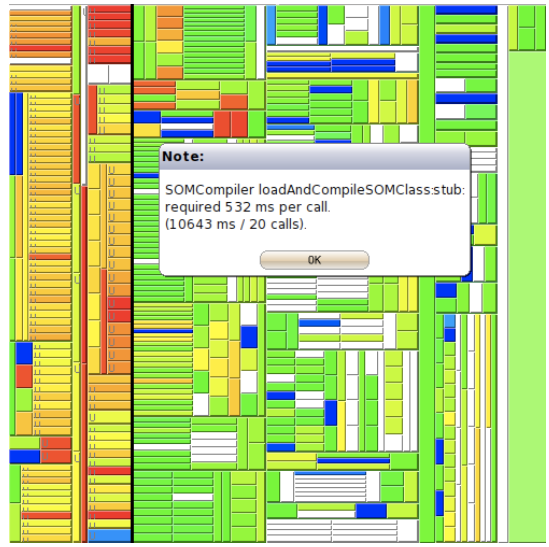

Figure 4. Required method run-time per call for all AweSOM tests. Red hot spots highlight long-running methods whereas blue methods are quite fast.

long running tests of AweSOM and permits their correction.

e) Run-time Bottlenecks during Testing: Our approach collects performance characteristics for each method during the execution of test suites. Then our approach summarizes these data inside the tree map. Our approach implements three different run-time measures:

- The *call* measure is a count of the number of times a particular method has been called. This measure helps in detecting methods that are called the most.
- The *tree* measure is a count of the total time required to execute a method from its call to its return. This measure helps in analyzing entry points into long running behavior such as expensive API calls or large loops.
- The *leaf* measure is a count of the time required to execute a method without including the methods it calls. This measure helps in finding methods requiring much execution time such as I/O operations.

The tree map uses the color hue to represent a method's run-time measure from blue (for lowest results) to red (for highest results).

To understand why running all unit tests of AweSOM takes so long (20 seconds for 125 unit tests) a developer starts by using the tree measure on the tree map as shown in Figure 4. The developer sees that the `loadAndCompileSOMClass-:stub:` method is more red than other methods: indeed, each call to this method requires about half a second and this method is called 20 times during testing. As a result, 10 seconds of the total execution time of the test suite are passed within this method. The developer decides to introduce a caching mechanism in this method which approximately divides the total execution time by two.

f) Memory Consumption of Test Objects: Efficiency feedback also reveals test cases that create a large number of system objects. For each test case our approach counts the number of instances of each system class and presents this data within the tree map. In this visualization, the tree map only presents packages and classes, while the color hue represents the number of instances of each system class from blue (for no instances) to red (highest number of instances).

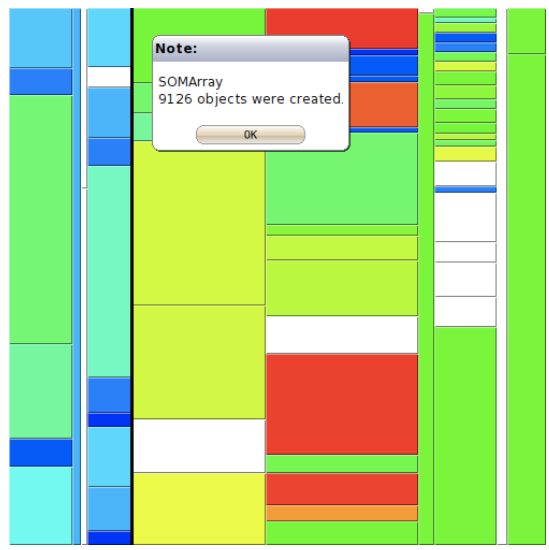

Figure 5. The tree map summarizes the number of created instances during testing. The color spectrum ranges from blue (only a few objects) to red (a large number of objects). This visualization only represents packages and their classes since methods are not important for memory consumption.

In the AweSOM example of Figure 5, efficiency feedback identifies numerous instances of the `SOMArray` and `SOM-Object` classes as these two classes are more red than the others. By analyzing in greater details the code that calls each of the constructors of these classes the developer decides that the code is correct.

IV. Implementation

For the realization of test quality feedback, we give an overview of our Paths coverage framework and our extended test runner named PathMap.

A. Paths Coverage Framework

Test quality feedback is based on a lightweight method coverage and profiling framework that provides on-demand refinements at the statement level. Doing this coverage on-demand is necessary as a complete statement-level coverage analysis of the whole system would slow down the execution by a factor of 100 [13]. At the level of methods, we collect run-time information with flexible method wrappers [14]: a wrapper introduces new behavior before and after the execution of a specific method without changing its behavior. Depending on the chosen test quality feedback, wrappers collect covering tests, method calls, or execution time for each system method. The framework stores the measurements and makes this data available to any interested tools including PathMap. To compute on-demand statement-level coverage for a specific method, the framework takes its covering tests and executes them in the background. A wrapper dedicated to this method records covered statements by executing method's byte code with a special Smalltalk interpreter.

B. PathMap Test Runner

Our Paths framework is the foundation for extending the test runner of the Squeak/Smalltalk IDE with our approach. Figure 6 is a screenshot representing our extended test runner assigned to the AweSOM software system. This test runner is composed of three main panes: the pane on the left lists all test classes of the software system; the middle pane presents the tree map of the software system, which is composed of a morphic hierarchy; the pane on the right allows for changing various options of the tree map visualization and presents a legend. The test runner also presents a status bar on the top displaying a summary of the test suites execution and a status bar on the bottom displaying a summary of metrics on the system. It is possible for a developer to interact with the tree map: hovering on a box results in the name of the attached method, its class and its package being displayed while clicking on a box results in a menu being displayed. This menu allows the developer to get additional information about the method such as its source code (as in Figure 2), the value of some metrics (*e.g.,* complexity and number of covering tests). The menu also lets the developer inspect the run-time behavior of the method and debug it [15].

C. Discussion

We argue that our approach can be adapted to other object-oriented programming languages. For implementing our Paths coverage framework, the language and its libraries have to support dynamic and static analysis techniques. While the dynamic analysis for method coverage can be implemented with aspect-oriented programming [16], statement-level coverage depends on the language features. For instance, in C++ many

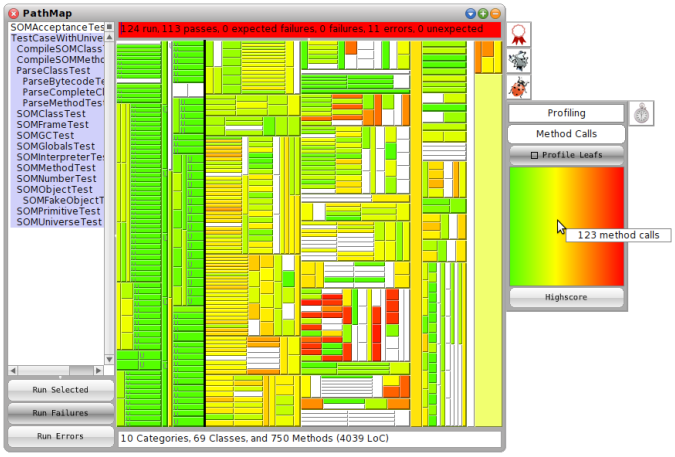

Figure 6. Extended test runner implementing our approach for Squeak.

coverage tools insert probes into the source code and in Python the interpreter offers a simple hook function for a fine-grained run-time analysis. Regarding static analysis, developers can rely on several external analysis tools or the reflection capabilities of the language. Finally, our PathMap tool is mostly a visualization concept whose implementation only depends on the underlying IDE user interface. For instance, Eclipse can be extended with a plug-in for rendering the tree map and Paths data.

V. Evaluation

This section evaluates the benefits and efficiency of our approach for different software systems.

A. Practicality

We evaluate the practicality of our approach and extended test runner through the study of two software systems: 4Conferences and Seaside.

1) 4Conferences: The first system, named 4Conferences, is an undergraduate student project. This project is a conference management web application permitting activities such as the registration of attendees, the organization of payments, the printing of badges, and the planning of talks. The project was developed in two phases by two distinct groups of students in the context of a software engineering course. The first phase resulted in a working system with basic features only and consisted of 5 packages, 77 classes, 1126 methods, and 21.58 % coverage by unit tests.

For the second phase, we asked a group of 16 bachelor students to add a specified set of features to the system. Writing unit tests was not mandatory but the students decided they needed some more to better understand the system and to prevent regression. We proposed them to use our extended test runner. This second phase lasted 3 months and resulted in 7 packages, 131 classes, and 1813 methods. As a result of their work the method coverage increased from 21.58 % in the first phase to 69.33 % at the end of the second. Figure 7 shows two tree maps. On the left-hand side the visualization shows

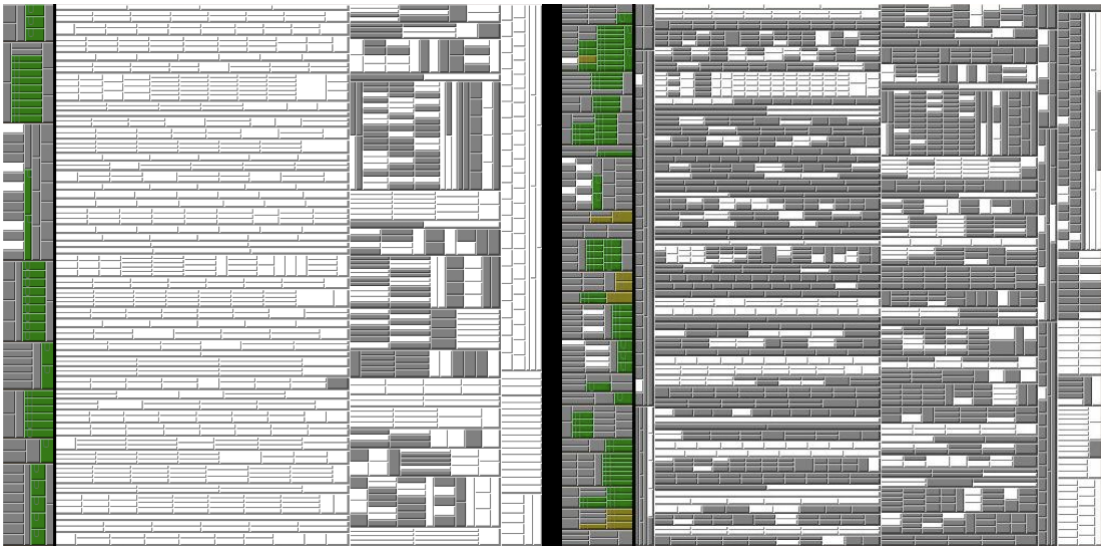

Figure 7. 4Conferences before and after the second phase of the implementation which used our test quality feedback

the system after the first phase: students have only tested the model package. On the right-hand side the tree map shows the system after the second phase.

The students told us they made extensive use of the test runner and especially the effectivity feedback. They told us the test runner helped them to find gaps in the coverage and made it clear which new tests they had to write. Even if social desirability biases this evaluation we argue that our student project illustrates at least the applicability of our approach to other systems and the improvements to the 4Conferences' test base.

2) Seaside: Seaside is an industrial web application framework[1] much larger than the previously described projects: In version 3.0, Seaside consists of 60 packages, 402 classes, and 3830 methods. Seaside is an open source project implemented by numerous developers not part of our research group.[2]

Figure 8 shows a tree map for the Seaside framework where darkness represents coverage and hue represents method size. This figure contains numerous large boxes summarizing classes with too many methods (as explained previously). The color of such class boxes is computed according to the average size of its methods or another strategy such as maximum or 90 percentile. Clicking on such a box reveals a new tree map dedicated to this class. Even for such a large project, our test runner is able to point out some gaps in the coverage of the system methods (*e.g.,* red boxes, bottom right of the figure).

B. Efficiency of PathMap

To evaluate the impact of using our test quality feedback on a day-to-day basis, we measure the performance overhead for two previously presented projects: AweSOM and Seaside.

We believe the chosen software systems are representative of the kind of system our approach can be used for: AweSOM

[1]http://www.seaside.st

[2]We acknowledge our participation in less than 1 % of the code base.

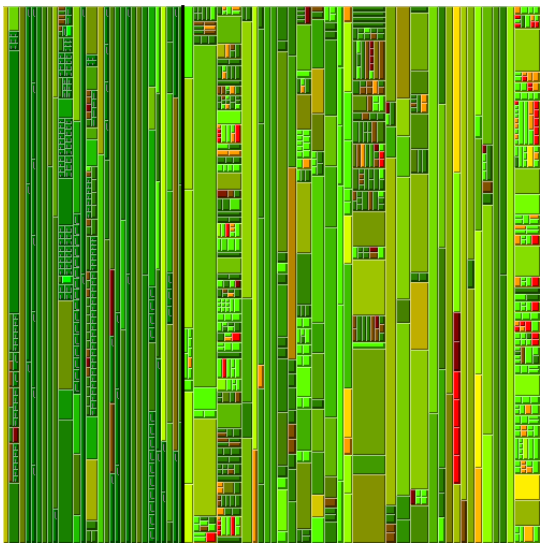

Figure 8. Test quality feedback for Seaside method coverage with lines of code

is a mid-sized research prototype with high coverage whereas Seaside is a much larger industrial programming framework with medium coverage. In the case of AweSOM the evaluation takes into account the changes discussed in Section III-B.

All experiments were run on a MacBook with a 2.4 GHz Intel Core 2 Duo and 4 GB RAM running Mac OS X 10.6.6, using Squeak version 4.1 on a 4.2.1b1 virtual machine. Table I presents the results of this evaluation in four parts:

1) The *system properties* part presents various information about the software system.
2) The *one-time cost* part presents the cost of analyzing the source code of the software system. Creating the tree map view can take a significant amount of time for a

		AweSOM	Seaside
System properties	Classes	69	402
	Methods	750	3830
	Tests	125	692
	Coverage	81.7 %	57.9 %
One-time cost	Creation of tree map (s)	1.0	7.1
	Complexity per method (s)	0.1	0.5
	Authorship per method (s)	3.3	11.4
Run-time overhead	Std execution time of all tests (s)	9.8	9.2
	With method coverage (s)	22.5 (2.3x)	26.6 (2.9x)
	With tree profiling (s)	54.9 (5.6x)	111.4 (12.1x)
Refine. cost	Avg. time for statement coverage (s)	3.7	1.0

Table I
PERFORMANCE CHARACTERISTICS OF TEST QUALITY FEEDBACK. MEASURES INDICATED WITH '(S)' ARE EXPRESSED IN SECONDS.

large project such as Seaside (7 seconds). However this creation is done only once, when the test runner opens. Our current implementation tends to be slow because it creates for each source code entity a separate morph object. We can improve this performance by drawing the tree map within a single morph. PathMap can calculate the complexity of all methods of the software systems in a very efficient way (half a second for Seaside). Calculating authorship of all methods is much more costly as this requires mining the repository with I/O operations. It is important to note that calculating such static properties is done only once for each method.

3) The *run-time overhead* part presents the overhead of executing the test suites with additional measures. For measuring the run-time overhead, PathMap executes the entire unit test suite with and without test quality feedback. For example, running all unit tests of AweSOM takes 9.8 seconds when no feedback is required and takes 22.5 seconds (2.3 times slower) when method coverage is required. Measuring performance characteristics for each method during the execution of test suites takes a significant amount of time. We plan on using a more advanced performance measurement technique (such as sampling) in future works.
4) The *refinement cost* part presents the cost of refining statement coverage for one particular method. For example, calculating the statement-level coverage of a method for the AweSOM system takes 3.7 seconds on average. This computation is slower for methods that are covered by a lot of unit tests as each of them must be executed to list the covered statements. Because AweSOM methods tend to be covered by more tests, the average cost is higher than for Seaside. An initial improvement would require stopping the execution of the unit tests as soon as all statements of a method are covered. Providing statement coverage by default instead of method coverage would result in an execution of all the test cases a magnitude slower. We argue on-demand analysis of statement-level coverage provides a good trade-off between performance and level of details.

We have used our coverage framework and extended test runner in several other projects not covered here such as Orca, a web application framework that translates Smalltalk code to JavaScript, and the Smalltalk Squeak compiler. In all projects we found that our extended test runner could be used in place of the standard Squeak Smalltalk test runner. Nevertheless, while evaluating our approach with Seaside we found that we had to improve the profiling overhead so that developers can more frequently rely on the proposed feedback.

VI. RELATED WORK

There exists numerous testing tools providing code coverage feedback. Yang *et al.* surveyed numerous of them [6]. In their work they emphasize the lack of support of these tools for "prioritization", *i.e.,* finding parts in the code that require more tests. Our work improves this situation.

Hapao is a test coverage tool which uses a graphical visualization to facilitate the discovery of not (completely) tested code [17]. To do so, classes and methods are represented as boxes with various heights, widths, colors and borders corresponding to various criteria such as complexity, size and coverage. With some experience with the visualization it is possible to rapidly detect graphical patterns within the visualization informing the developer of missing test cases. However, Hapao does not present any other information than coverage such as time and memory consumption. Moreover, our tree-map-based visualization is more scalable. For instance, Hapao's visualization of the Seaside core system is more than 12.000 pixels large, which hardly fits into a standard screen, whereas our visualization fills the space provided by the user and nothing more.

TestQ is a tool capable of statically analysing test code [18]. TestQ proposes various visualizations helping the developer to find problematic test suites, *e.g.,* test suites that are very long or that exercise too much. Their work is complementary to ours in that we focus more on dynamic information such as coverage and run-time performance.

Sonar[3] is a software quality platform which leverages various static code analyses tools. Sonar presents results of these tools within a unified and customizable web interface which makes it possible to navigate the source code and visualize analysis results. However, Sonar requires all analyses to be conduced before-hand, preventing immediate feedback to the developer, and does not propose to change the problematic code, slowing down the enhancement of the code. Sonar is then better integrated with a continuous integration system whereas our approach is better integrated within the IDE.

Jones *et al.* [19] proposed a seminal work that aims at finding faults within a software system by comparing unit test results. They offer a visualization where each statement is represented as a line of one-pixel height, each line with a color indicating how suspicious it is to contain a fault. This work and the numerous that follow propose to leverage existing unit test cases to find faults based on the passing/failing state of

[3] http://www.sonarsource.org/

each test case and a coverage analysis at the statement level. They however require a good unit test suite as input that our work aims to provide.

The visualization of our approach is based on tree maps [10] that visualize arbitrary hierarchical structures by subdividing a given area into small rectangles. In the course of time several other approaches have improved the standard layout algorithm: they prevent long rectangles that are difficult to see [20] or they highlight objects that are also neighbors in the hierarchical structure [21], [22]. We base our tree map on the standard layout and limit the hierarchy depth to four, make large rectangles zoomable on demand, and draw classes with a thicker border to emphasize their included methods.

VII. Conclusion

Developers require feedback to ensure good coverage of a software system and good quality of a test suite. Existing tools can list untested system parts but fail to help developers prioritize their testing effort. These tools lack effectivity feedback. Other tools can profile a specific test case to find inefficient code but fail to profile a complete test suite which prevents identification of common bottlenecks. These tools lack efficiency feedback.

In this paper, we have proposed an integrated approach where effectivity and efficiency feedback are combined to help developers prioritize their testing effort. We have shown how a single tree map visualization can be used to display information such as coverage analysis, complexity, and authorship of a complete system. We have also described a coverage framework, named Paths, that provides method-level coverage analysis by default and on-demand background analysis at the statement level. We have shown how we extended the Squeak Smalltalk test runner with an interactive tree map visualization. Our approach has been successfully applied to several projects, including by students unfamiliar with the approach.

We are currently expanding this work in several directions. We are working with a graphic design expert to help us choose the right colors for representing multiple information in a single visualization (such as coverage and complexity). We want to reduce the time that the developers have to wait before getting feedback. To do so we are applying continuous selective testing [23] which should result in an always up-to-date tree map visualization.

References

[1] K. Beck, *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[2] J. Lawrance, S. Clarke, M. Burnett, and G. Rothermel, "How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study," in *VL/HCC '05: Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 53–60.

[3] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley Longman, 2004.

[4] S. Berner, R. Weber, and R. K. Keller, "Enhancing Software Testing by Judicious Use of Code Coverage Information," in *ICSE '07: 29th International Conference on Software Engineering*, 2007, pp. 612–620.

[5] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage Measurement Experience during Function Test," in *ICSE '93: 15th International Conference on Software Engineering*, 1993, pp. 287–301.

[6] Q. Yang, J. J. Li, and D. M. Weiss, "A Survey of Coverage-Based Testing Tools," *Computer Journal*, vol. 52, pp. 589–597, August 2009.

[7] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn, "The SOM Family: Virtual Machines for Teaching and Research," in *ITiCSE '10: 15th Conference on Innovation and Technology in Computer Science Education*, 2010, pp. 18–22.

[8] X. Cai and M. R. Lyu, "The Effect of Code Coverage on Fault Detection under Different Testing Profiles," in *A-MOST '05: 1st International Workshop on Advances in Model-based Testing*, 2005, pp. 1–7.

[9] M. C. K. Yang and A. Chao, "Reliability-estimation & Stopping-rules for Software Testing, Based on Repeated Appearances of Bugs," *IEEE Transactions on Reliability*, vol. 44, no. 2, pp. 315–321, 1995.

[10] B. Johnson and B. Shneiderman, "Tree Maps: a Space-filling Approach to the Visualization of Hierarchical Information Structures," in *VIS '91: Proceedings of the 2nd Conference on Visualization*, 1991, pp. 284–291.

[11] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix this Bug?" in *ICSE '06: 28th International Conference on Software Engineering*, 2006, pp. 361–370.

[12] T. Fritz, G. C. Murphy, and E. Hill, "Does a Programmer's Activity Indicate Knowledge of Code?" in *ESEC-FSE '07: 6th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2007, pp. 341–350.

[13] M. Haupt, M. Perscheid, and R. Hirschfeld, "Type Harvesting - A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages," in *SAC '11: 26th Symposium on Applied Computing*, 2011, pp. 2169–2175.

[14] J. Brant, B. Foote, R. Johnson, and D. Roberts, "Wrappers to the Rescue," in *ECOOP '98: 12th European Conference on Object-Oriented Programming*, 1998, pp. 396–417.

[15] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt, "Immediacy through Interactivity: Online Analysis of Run-time Behavior," in *WCRE '10: 17th Working Conference on Reverse Engineering*, 2010, pp. 77–86.

[16] T. Gschwind and J. Oberleitner, "Improving Dynamic Data Analysis with Aspect-Oriented Programming," in *CSMR '03: 7th European Conference on Software Maintenance and Reengineering*, 2003, pp. 259–268.

[17] V. P. Araya, "Test Blueprint: An Effective Visual Support for Test Coverage," in *ICSE '11: 33rd International Conference on Software Engineering*, 2011, pp. 1140–1142.

[18] M. Breugelmans and B. van Rompaey, "TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites," in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[19] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *ICSE '02: 24th International Conference on Software Engineering*, 2002, pp. 467–477.

[20] B. Shneiderman and M. Wattenberg, "Ordered Treemap Layouts," in *INFOVIS '01: Symposium on Information Visualization*, 2001, pp. 73–78.

[21] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi Treemaps for the Visualization of Software Metrics," in *SoftVis '05: Symposium on Software Visualization*, 2005, pp. 165–172.

[22] J. J. Van Wijk and H. van de Wetering, "Cushion Treemaps: Visualization of Hierarchical Information," in *INFOVIS '01:Symposium on Information Visualization*, 1999, pp. 73–78.

[23] B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld, "Continuous Selective Testing," in *XP '10: Agile Processes in Software Engineering and Extreme Programming*, 2010, pp. 132–146.