



PACIFIC NW
28TH ANNUAL
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010



ACHIEVING
QUALITY
IN A COMPLEX
ENVIRONMENT

*Conference Paper Excerpt
from the*
**CONFERENCE
PROCEEDINGS**

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

Visualizing Software Quality

PNSQC 2010 conference paper

Mark Fink

August 14, 2010

Author contact: mark@mark-fink.de

Project website: <http://www.testing-software.org>

Abstract

When software projects reach a considerable size (multi-person, multi-man-year) they are difficult to maintain. Which results in high efforts for software maintenance, code duplication, bugs, and other quality related problems. The consequences include limited software life cycles, especially in areas where software evolves rapidly. For example in an industry setting, business applications are usually re-implemented every 3-5 years. This is commonly considered as the only foolproof strategy currently available. It is also considered as a huge waste of resources and a true sign of the software crisis.

Quality initiatives could significantly improve the software quality, but they usually suffer from tight budgets, lack of support and other neglectful factors. Also, due to the fact that software can be immensely complex, complete testing is impossible. So it is desired to drive testing efforts to parts of the software product that would benefit the most (risk considerations).

The root cause of the software crisis lies in software engineering itself. Software is the most complicated product we make and we can not see it, or touch software like we can see or touch the work products of other engineering disciplines, such as construction or mechanics. During visual inspections you instantly see whether a car misses the bumper, a door or the wind-shield. The quality of a software project is not as obviously apparent through visual inspections.

What if we could make the quality of software visible? This could be used to govern testing initiatives and could help to decide if the software is really ready for release. This paper provides in-depth insights and experience on interactively visualizing different aspects of software quality.

1 Introduction

I started the visualizer tool project in fall 2009 in order to create interactive visualizations of software quality for huge code bases, like Mozilla Firefox, Apache Webserver, and the Python code base. When you are working on huge code bases like these and you need to improve the quality, it is difficult to determine where to start. Most software quality initiatives have limited budgets these days. As a consequence you want to identify which parts of your code base would benefit the most from your initiative. It is also desirable to interactively visualize how the quality of your project evolved over time.

The times that one single developer could know the complete code of the whole system are long gone. For example the first Cisco router operating system had 30.000 lines of code and was written by one developer. Today's router operating system size is somewhere between 50 and 100 million lines of code. This is over 2.000 times as much code for maintenance, testing, etc. Today a whole team maintains the router OS. One of the reasons for this increase in size is that developers in the early days usually had to shrink their code to fit onto the hardware. The limited hardware resources gave them a strong incentive to clean up their code during maintenance.

When testing huge software systems an enormous amount of data today is generated, especially with test automation and continuous integration. Analysis of this data is a huge task. New methods are needed to analyze this data and to extract useful information from that data.

I think analysis of data is one of the key elements/ skills in software quality assurance. Since software systems dramatically increased in code size they usually require team work for development, testing and maintenance. As a consequence none of the team members has an overview about what needs testing/ retesting any more. The need for analysis tools arises that support the decision making process on what needs testing, and this is where visualization can add value.

The science of Data Visualization itself is an emerging discipline. Mankind has a long but erratic history with data visualization, shaped by the available technology and by the pressing needs of the time. Its modern roots date back for example in fighting the Cholera Epidemic in London 1854 [1]. The last decade have brought new and useful innovations in visualization techniques. Many innovations in this area were discovered in Genetics.

This paper gives an insight into the application of the discipline of software quality visualization. The focus is on discussing and documenting the requirements for an interactive software quality visualization tool with the aim to improve the software development process. A prototype of the tool has been implemented in order to demonstrate the interactive visualization of software quality. Some organizations currently are thinking about adapting software quality visualization, some have already managed to successfully apply them [2] and I give examples where possible.

2 The problem

The software crisis has roots in the nature of software itself [3]. Software-development is extremely powerful, so we can create complex application like Internet Browsers, Word Processors and massive simulations. But programming is also immensely difficult because software is the most complex product we make.

The management of a software project is very difficult, too. We still cannot agree on appropriate metrics to measure the quality or completeness of a software project. Here I believe that lines-of-code is absolutely inadequate for various reasons. First of all, effort estimation is not possible; it is impossible to determine at the beginning how many lines of code the system will have. Second, it is extremely difficult to say how much time it will take to get the lines done.

“Measuring programming progress by lines of code is like measuring aircraft building progress by weight”, Bill Gates.

When we look into testing as part of software construction, we already know that we can never fully test a software package of significant size because of limitations of resources like time and budget. Therefore we urge to put our sparse resources to effective use in order to maximize the benefits. We want to focus our testing resources on the parts of the software where we have the most risk.

In industry software projects we usually get a very different view. Let me explain. During my career in software testing I have been facing an ongoing phenomena which I call “The illusion

of testing resources well applied". A good example is during the maintenance phase of a software application: the test analysts come up with a sound test plan based on the design of the software change. The test plan contains test cases for the new features, plus regression test cases and an effort estimate for testing (in the sample 200TC, 3 testers allocated for a 3 weeks time frame). After the testing resources are allocated to the project the test manager assigns the test cases to these resources. Test progress is usually reported to management in the following form of a progress chart:



This almost scientific appearance of the process with the above diagram, which is used in management reporting, gives the impression that everything is in order. But on the contrary, testing resources are not applied efficiently! The fundamental flaw of this approach is that it does not contain any information about whether the test cases make sense, if they cover the software changes, or risk considerations. Unfortunately the review process for the test plan often does not reveal these problems. In practice the review process is often used to shift the responsibility away from the (outsourced) testing department.

This is really a planning/ analysis problem. In order to significantly improve the testing process and as a consequence the testers contribution to the project we have to apply better analysis and planning techniques.

A planning/ analysis tool like the visualizer should help you identify:

- Areas in the codebase that have been changed
- Areas in the codebase that require intensive testing (identify risks)
- Areas with low unit test coverage for test automation
- Performance critical spots in the codebase

3 Software Quality Visualization

Software Visualization is a topic that has been discussed in computer scientist cycles for many years. The broader topic of information visualization received a lot of attention e.g. through the breakthrough in the genome project made possible by visualization. The sheer amount of data we collect every day makes visual analysis skills a key competence for today's companies.

Some industries adapted visualization techniques a long time ago. For example air-traffic control could hardly be imagined without a graphical display. And Wired Magazine has long run a feature called "Info Porn", which attempts to showcase data visualization in interesting and unusual ways.

Discovery and analysis of data through visual exploration was made popular by Prof. Edward R. Tufte's books. The principle is to map the attributes of an abstract data structure to visual attributes such as Cartesian position, colour and size, and to display the mapping [4].

In the last two years some companies reported success in adapting visualization techniques to software testing [2].

3.1 Treemaps

The key to visualization principles is to understand the enormous capacity for human visual information processing. The human eye/ brain system is capable of processing thousands of information bits in a few milliseconds if presented visually. It does not take any effort to see whether there are one or several red dots and it can be done in less than 200 milliseconds [5].

The use of proximity coding, plus colour coding, size coding, animated presentations, and user-controlled selections enable users to explore large information spaces rapidly and reliably [5].

Treemaps are excellent for mapping arbitrary two dimensional data to hierarchies and for seeing relationships between the data and the tree [5]. Given primary graphical encodings of area, colour and enclosure, treemaps are best suited for the task of outlier detection, and cause-effect analysis. But without adequate computer support, they are difficult to generate by hand. Only recently have adequate support routines and libraries emerged that allow easy generation of treemaps.

Treemap implementations today appear as one of several different treemap layouts. Common Layouts are: Clustered, squarified, and slice & dice.

Treemap visualizations tend to scale up to support more items than other visualization techniques such as bar or radial graphs, in some cases they can be used to display more than one million items [5]. This is sufficient to handle the files or functions of the biggest software projects, like operating systems or applications like internet browsers.

I believe that treemaps beautifully implement the information visualization mantra coined by Ben Shneiderman: "overview first, zoom and filter, then details on demand" [5].

For more details on Prof. Tufte's Visualization principles and on Treemaps please refer to the excellent PNSQC 2009 paper and slides by Marlina Compton [6].

4 Firefox as a sample code basis

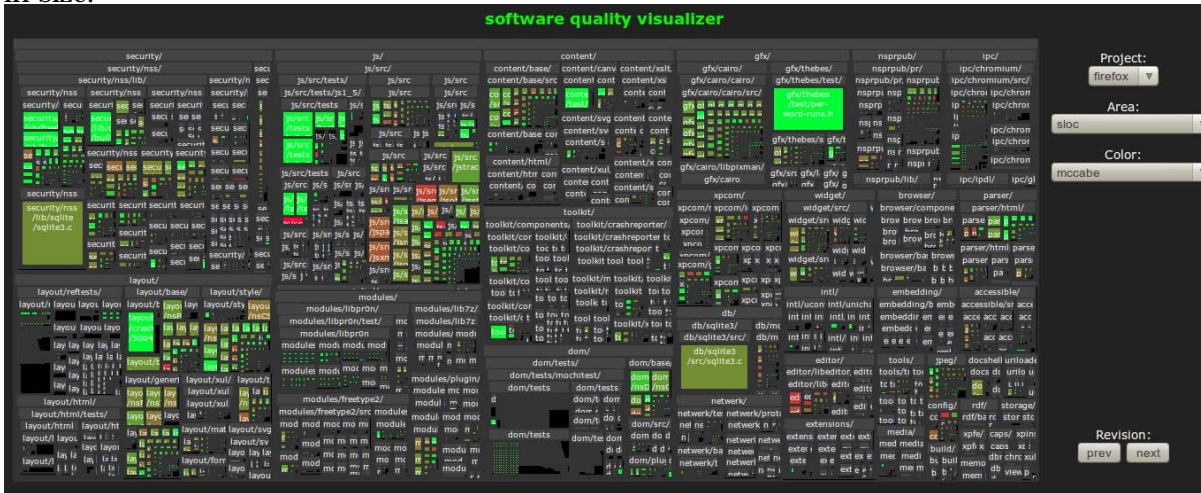
Many circumstances made the Mozilla Firefox browser the ideal reference project for the development and demonstration of the visualizer package.

- Open source project with code openly available
- Defect database openly available
- Significant size of the codebase (> 1 Mio. source files, > 50 Mio. lines of code)
- Proficiency in automation and mature quality assurance
- Technology affluence
- Existing contacts with people from Mozilla QA team; looking for feedback on the prototype

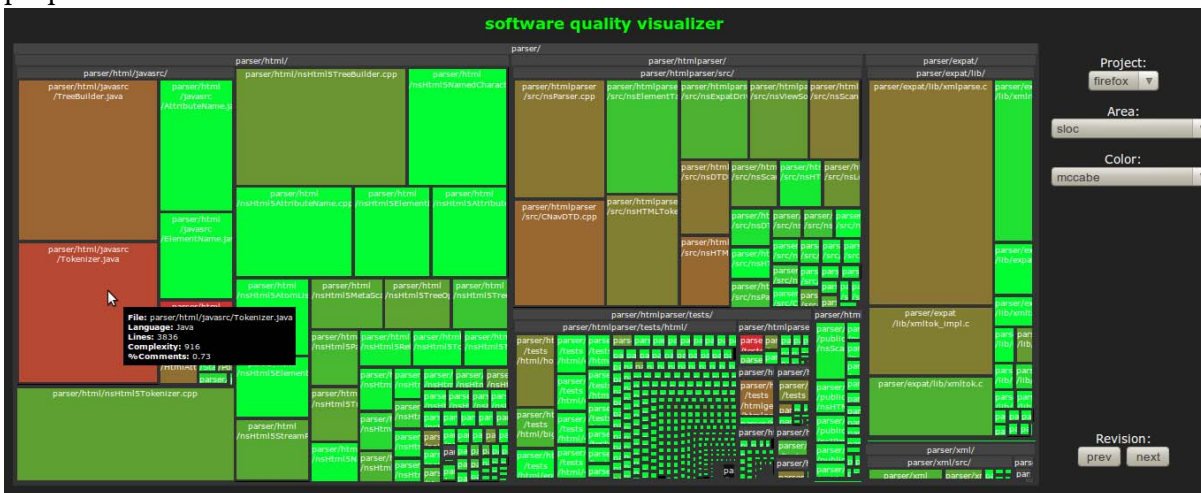
4.1 The visualizer demo

A significant part of the conference talk will be the demonstration of the features of the current visualizer implementation. The demonstration of the visualizer features will be performed along the lines of the visualization mantra (overview first, zoom, details on demand). Also the navigation of revisions will be demonstrated.

The following screen-shot shows the top level visualization of the Firefox source code. In this view we quickly familiarize about the structure of the project code and how the modules relate in size.



The next screen shows the visualizer after the user navigated one level down into the Firefox *parser* module. In this view the user can easily spot complex code like the *Tokenizer.java* which has over 3.800 lines-of-code and a McCabe complexity of over 900. Obviously the file contains a huge amount of switch statements. A comment-to-code ratio of 0.73 shows that the file maintains a sound level of documentation. But we have to make sure that we have proper tests for that one!



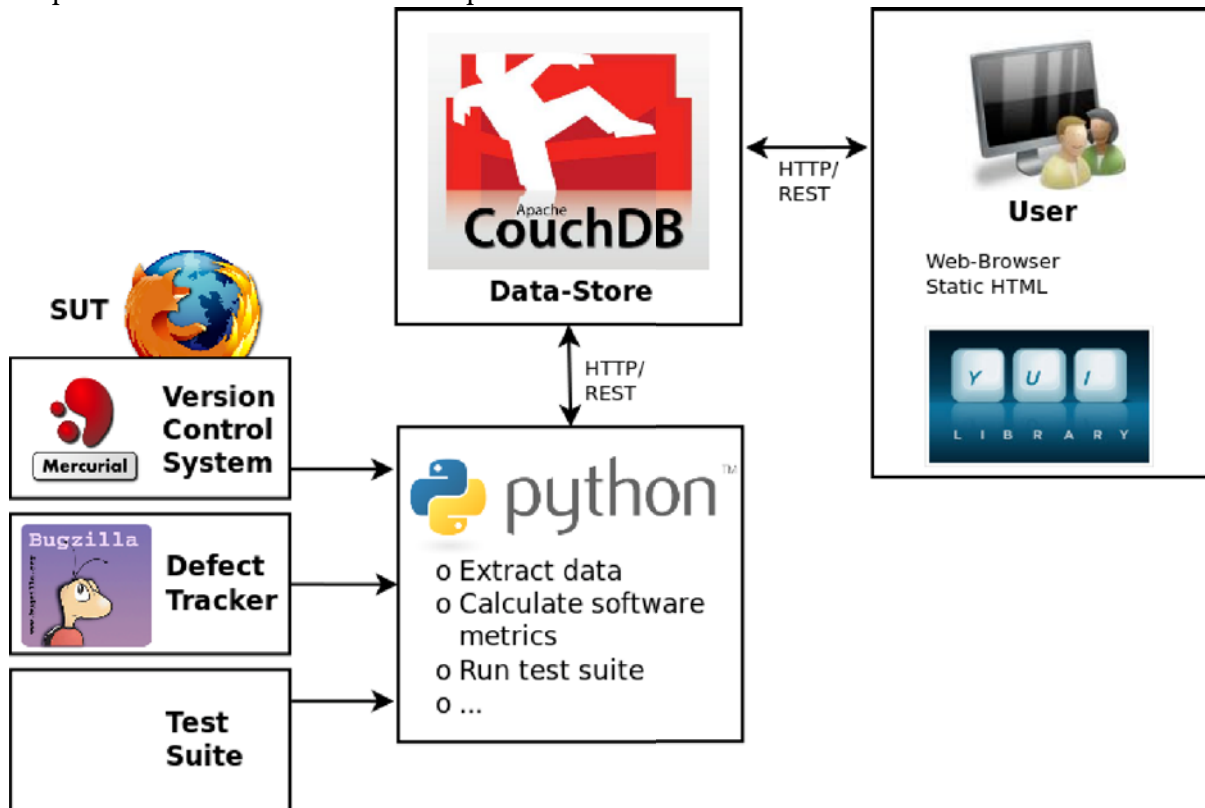
The screenshots above also show some of the features of the current visualizer implementation:

- **Select box for projects** - you can add another project to your client by providing a name and the URL of its data store which contains the metrics data
- **Select box for the metric used for area** - usually the lines-of-code metric but it is possible to visualize other metrics in relation
- **Select box for the metric used for colour** - select one of the metrics contained in your project data store. McCabe-complexity or comment-to-code ratio are good candidates to start your analysis.

- **Buttons to navigate revisions** - navigate project source code revisions in order to analyse how the project quality evolved over time
- **Context help** - provides the detailed metric information on demand

5 The visualizer architecture

The following diagram gives an overview of the architecture of visualizer tool implementation. All parts of that architecture are explained below.



5.1 System Under Test (SUT)

The SUT is the software project of which we want to analyse the software quality. Besides the source code of the software which is usually stored in a version control system, there are usually more development tools that provide important data for visualization of software quality. The basic set of tools is the version control system, the bug tracker and the test suite.

5.2 Data Collection

The metrics [8] package which is also maintained by the author is used to extract metrics from the code base and to walk the version control system (Subversion (SVN), Mercurial (HG)). Currently the metrics package provides metrics on lines-of-code, lines-of-comments, and McCabe-complexity consistently over multiple programming languages (for example C, C++, Java, Python, shell scripts, JavaScript, and many more). Existing metrics packages like SLOCcount, PyMetrics etc. could not be used because each metrics package has its own algorithm on how to count lines-of-code. Extraction of data from the defect-tracker and test-suite tools also belong to this part of the architecture but are not yet implemented.

The Data collection scripts have been implemented in the Python programming language because the available libraries make it particularly interesting as a glue language.

I assume that from your own experience it is clear to you that the data collection will be different for every project. That is also the reason why the visualizer tool will never work completely out of the box. The visualizer project aims to provide some common tools and infrastructure in order to ease the integration in your project environment. Data collection will still be highly dependant on your project environment and there will be at least some customizations necessary in order to access your project tools like version control system, defect tracker, continuous integration systems and test suite(s).

5.3 The data store

Nowadays successful companies like Google, Amazon, ebay, Yahoo, Facebook, etc. owning big applications, abandoned their RDBMS systems. Instead they are implementing their own flavour of document oriented or key-value oriented data stores. The main difference to RDBMS is that these new systems, which are commonly called NoSQL systems, scale better for big data stores. Meanwhile successful Open Source NoSQL implementations are also available.

Map/Reduce is a concept developed at University of California, Berkeley that applies functional programming concepts to the problem of big data analysis. CouchDB is an Erlang based open source implementation of the Map/Reduce concept by the Apache Software Foundation [9]. CouchDB has no Joins, no Relations, no Tables, and no SQL which are the RDBMS concepts that we all know from traditional databases. What makes CouchDB a particularly good fit for the visualizer implementation is that it uses JavaScript for writing the Map/Reduce functions and that it provides a REST API that plays nicely with the Ajax frontend of the visualizer. This means all the communication is plain HTTP. Additional middle-ware is not required.

The data store is one of the key elements of the visualizer architecture. As it turned out the CouchDB performance is not as good as it occurred to me at the beginning and it is also difficult to implement the advanced visualizer features like interactive queries in CouchDB. I currently re-evaluate CouchDB if it can be adapted enough to serve the needs of the visualizer tool or if it has to be replaced altogether by another data store.

5.4 Interactive visualization (frontend)

A key concept to the visualizer is that it has to be highly interactive, following Ben Shneidermans visualization mantra “overview first, zoom and filter, details on demand”. This is why the visualizer requires an implementation that supports the user’s interaction with the visualization data.

Another factor that has constrained the technologies being used for the visualizer implementation is that all Mozilla QA tools must be web based like for example Graph-Server, Tinderbox etc. The Firefox development is Open Source and therefore it is very rarely the case that two developers are co-located, except maybe for sprints. This requires web based tools that provide easy access for the developers.

For the prototype implementation the Yahoo User interface library YUI3 was used. YUI3 is a Javascript/ Ajax framework [10] which provides a sane module concept and proved to work very well during the implementation of the prototype.

6 Quality metrics

A metric is something that you can measure, and used as a method in order to reason or infer about something you can not (easily) measure. For example lines-of-code is a metric that is often used as a proxy to predict efforts/ cost.

Why measure? In everyday life we use experience, common sense, and intuition to come up with solutions for our problems. We measure because software development is so complex that intuition alone is not enough. Team work makes it even more difficult because the single expert with an overview does not exist any more in today's projects.

“Measurement is the key to all disciplines of science and technology, and the maturity of the discipline is marked by the extent to which it is supported by a sound and comprehensive system of measures, measurement standards, measurement tools and measuring procedures” [11].

Usually the software development projects have a lot of data that would be useful for quality analysis. The problem is that it often proves difficult to use this data for test planning because of incompatible formats, it is not captured, lack of tooling, etc.

One of the key ideas of the improvements of Visualizing Software Quality is to discover these data islands and to make the data available for the interactive visualization of software quality.

Important data sources for software quality visualization:

- Source code (Lines-of-code, Complexity metrics, test coverage by automated tests)
- Changes (code churn from the version control system)
- Defects (defect tracker)
- Test results (test suites in different test tools)

Another goal of the analysis is to gain knowledge from the history of the software project and apply it to new/ changed components. For example, correlating defects with code complexity can help to identify problems in new or maintained code. Also changed code is more likely to contain errors than unchanged.

I currently use the following criteria for metrics used in visualization:

- Interval, rational, and absolute scales (true/ false, enumeration, classifications do not work).
- Not necessarily comparable (metrics for different programming languages must be available, so that new relations can be uncovered. Even when comparing different programming languages).
- Same basis/ semantics. The metrics must be implemented in the same way for different programming languages. For example lines-of-code metric implementations come in many varieties which handle specifics differently (line breaks, statements, etc.).

6.1 Useful Metrics

Since there are so many metrics available, which ones are best? Opinions certainly differ on this topic! But we cannot do all, so I have limited the scope to a small set of what I believe are useful metrics for “Visualizing Software Quality”:

- Organization - number of engineers who touched the code (indicator for potential issues regarding communication paths or know-how transfer)
- Edit frequency (number of file edits per time-frame)
- McCabe-complexity (already implemented)
- Unit-Test-Coverage/ Failed tests
- Number of bugs fixed (defect tracker fixed with revision)

Most of the metrics are discussed in Nachiappan Nagappans paper [12] which provides scientific and practical evidence on the use of metrics to improve software quality.

The “Dependencies” metric discussed in the paper will probably not be implemented for the visualizer tool because it would be difficult to implement it transparently over multiple programming languages.

To the metrics discussed in the paper I added the following to the visualizer requirements:

- Lines-of-code (Needed for visualization; already implemented).
- Performance (In most industry settings performance is still a very important aspect of software quality. It is of enormous importance that performance problems are identified right when they are created because they are very expensive to analyse and fix later.).

Risk	Metric
Organization Development	Number of engineers who touched the code Lines-of-code Complexity
Testing	Edit frequency Unit-test-coverage Failed tests Continuous performance

One thing about the categorization above that I do not like at all is that it will easily lead to a discussion about responsibilities and who should have access to the data, etc. I also want to emphasise the point that the separation of development and testing teams is problematic. In my opinion it is the root cause of many problems in quality assurance.

The reason why I still included the above diagram is that I want to demonstrate a tool that can help you to verify if you have a shortage of metrics in one area of risk.

Obviously, what metrics should be implemented depends on the individual project environment. There is no golden rule that says implement all of these. My advice in this situation is to think about what you can easily extract from your version control system, defect-tracker, and unit test suite and you will get a clear picture very soon. At the very least, you now have more information available than before.

7 Conclusion and future work

The paper provides a brief overview of topic of data visualization, and reference to more detailed information. The paper does not provide an exhaustive documentation on the topic of visualization, interactive visualization of software quality, or technologies used for the implementation of the visualizer tool. One way to look at this paper is as a status report of the last 12 month working on the visualizer project.

At the time of this writing, the visualizer tool implementation is far from being complete. So why provide incomplete documentation and tools to a broader audience? My intention is to reach people interested in the visualization of software quality, in order to enter into fruitful discussions on the role of software quality visualization in improving the software development process and on how to improve the visualizer tool so it can be adapted into a variety of project environments. The engaged feedback I got from conference participants at the EuroPython 2010 was already very beneficial.

The following are what are planned as the next steps for the visualizer project.

Find a solution for multivariate metrics representation. For example colourization of complex modules with low unit test coverage.

Visualizer Implementation:

- Improve data store performance
- Reduced data transfer rate between backend and frontend

- Every time the user changes one of the view parameters a complete redraw is necessary. It seems like there is some room for improvement.
- Advanced interactive user queries
- Extract data from more sources, Unit/tests coverage, Code churn from version control system, Use defect tracker to locate error prone code, Performance tests (e.g. Firefox Talos module).
- More intuitive navigation.
- Best methods for displaying testing time, especially versus test coverage
- “Data export” as a way for the user to transfer information from the visualizer to other applications. I could think of lists, todo-lists and test-case-skeletons in the appropriate formats.
- Adjustable colouration for treemaps (probably an interactive slider)

8 Acknowledgements

Parts of this paper have been presented at the annual EuroPython conference in Birmingham, UK, in July 2010. During the presentation, I discovered a huge interest in the topic of visualization and received a lot of positive feedback on the visualizer tool and how to improve the presentation to make it clearer. Since that time I have continued to work on the system and the presentation will include the most updated software version and information.

Special thanks to Richard Vireday, Intel Cooperation for the insightful discussion about the visualizer implementation and his review comments on the paper over multiple stages from the early draft on.

-
- [1] Steven Johnson, *The Ghost Map*, 2006, Riverhead Books U.S.
- [2] James A. Whittaker, Presentation on “The Future of testing” at Google GTAC 2008, http://www.youtube.com/watch?v=Pug_5Tl2UxQ
- [3] Douglas Crockford, Presentation on *Quality* at the 2007 Frontend Engineering Summit
- [4] Edward R. Tufte, *The Visual Display of Quantitative Information*, 2001, Graphics Press
- [5] Benjamin B. Bederson, Ben Shneiderman, *The Craft of Information Visualization - Readings and Reflections*, 2003, Morgan Kaufmann Publishers
- [6] Marlena Compton, “Visualizing Software Quality”, paper and slides, 2009, PNSQC conference in Portland, <http://bit.ly/c75bbo>
- [7] Mark Fink, visualizer tool, <http://pypi.python.org/pypi/visualizer/>
- [8] Mark Fink, metrics tool, <http://pypi.python.org/pypi/metrics/>
- [9] Apache CouchDB, <http://couchdb.apache.org/>
- [10] YUI 3 — Yahoo! User Interface Library, <http://developer.yahoo.com/yui/3/>
- [11] Norman E. Fenton, *Software Metrics: A Rigorous and Practical Approach*, 1991, Chapman & Hall
- [12] Nachiappan Nagappan, *The Influence of Organizational Structure on Software Quality: An Empirical Case Study*, <http://research.microsoft.com/pubs/70535/tr-2008-11.pdf>