2009

PACIFIC NW

SOFTWARE QUALITY CONFERENCE

MOVING QUALITY FORWARD

October 27-28, 2009

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 commecial use.

Visualizing Software Quality

Marlena Compton
Equifax, Inc.
marlena.compton@equifax.com

Marlena Compton automates tests and performs testing in a distributed systems group at Equifax where she has worked for the past 5 years. She will be receiving her Software Engineering MS in December of 2009. This paper is derived from her thesis on visualizing software quality. She blogs about technical testing and data visualization at marlenacompton.com

Abstract
Moving software quality forward will require better methods of assessing quality quickly for large software systems. Assessing how much to test a software application is consistently a challenge for software testers especially when requirements are less than clear and deadlines are constrained. In assessing the quality of large-scale software systems, data visualization can be employed as an aid. Treemap visualizations can show complexity in a system, where tests are passing vs. failing and which areas of a system contain the most frequent and severe defects. This paper describes the principles of data visualization, how treemap visualizations use these principles and how treemaps can be employed for making decisions about software quality.

1. Introduction

For most software testers obtaining data is no longer a problematic task. Between test case management and bug tracking software, most testers can easily produce lists of highly critical bugs created from running mountains of test cases multiple times. Software testers have never before had so much access to so much data. But what does this data say about the health of a system?

Wading through bug reports and tests to produce a meaningful answer about the SUT's (System Under Test) overall health or health in specific areas can take more time than a project plan allows. Eventually, the task becomes boiling down all of the data into communication that is concise enough and clear enough to be understood in the limited review time available, yet still illustrates the problem with accuracy. As a means of communicating large sets of data with integrity, data visualization can help testers communicate their diagnosis of a system in a concise manner.

Let's begin with an historical example, albeit from a field far from software testing. During August of 1854, Dr. John Snow of London found himself in the center of just such a situation. Faced with a public fleeing the city's central district in sheer panic over a quickly spreading outbreak of cholera, Snow saw in the epidemic an opportunity. Despite the prevailing theory of the time that cholera was spread through contact with corpses or by breathing putrefied air, Snow suspected (and was determined to prove to a skeptical medical community) that the disease was spreading via a contaminated water source [1].

Beyond the curtailing of a cholera outbreak, there was an even more important reason for Snow to convince London authorities that cholera was spread through water. A few years earlier, in 1848, a law had been passed requiring that all waste in the city of London was to be deposited into a sewage system that flushed the waste directly into the River Thames which was also the source of drinking water for Londoners. Snow was convinced that London was on the verge of an even deadlier health epidemic. He used data collected from the 1854 epidemic and data visualization to prove this.

To effectively employ data visualization in making and communicating decisions about system quality, it is important to have an understanding of the process involved and the qualities in a good visualization that maintain the integrity of the data being presented. In this paper, we will begin with the above historical case study that demonstrates how a good map can communicate a complicated data set. We will then trace the path from understanding what makes a good visualization and the process involved in creating a good visualization to how a quality professional can apply these techniques for visualizing software quality, including tests and defects, in a system.

In section 2, we will summarize the characteristics of excellent visualizations with integrity, based on the work of one of the foremost data visualization experts, Prof. Edward Tufte. In section 3 an analogy will be drawn between John Snow's map of a cholera epidemic and maps that can help testers communicate system problems. In section 4, Treemap visualizations will be introduced. The way treemaps embody Tufte's principles of data visualization and how they are used to visualize source code metrics will be explained. Section 5 will show how treemaps can be applied to quality data such as tests and defects. Finally we will conclude with a discussion of what lies ahead for the visualization of software quality.

2. Data Visualization Principles

Before data visualization can be employed for *any* purpose, there must be an understanding of what constitutes a good visualization. In his pioneering book, The Visual Display of Quantitative Information [2], Edward Tufte explains the underlying principles of good data visualizations. At the highest level, visualizations should reveal *data*. High quality visualizations represent extremely large sets of numbers in a very small space. The data they represent is clear to the viewer and free of distortion. A good visualization allows the viewer to easily compare the information it is presenting. Data can be studied with the finest micro level detail, but also forms another layer of information at the macro level. Everything in an excellent visualization fits together like pieces of a puzzle in much the same way as a great painting or other work of art.

Tufte introduced several terms used in describing visualizations. ***Data ink*** refers to ink in a graphic that represents the statistics or other information in the graphic. It excludes marks such as gridlines. The ***data ink ratio*** is the amount of data ink divided by total ink used to print a graphic. The higher the data ink ratio, the greater the area of a graphic that is used to depict data. ***Chartjunk*** is extra decoration that appears in a graphic, but is not related to the information contained in the graphic.

Integrity in data visualization is very important. There are several principles that apply when assessing the integrity of a graphic. Labeling should be used extensively to dispel any ambiguities in the data. Explanations should be included for the data. Events happening within the data should be labeled. All graphics must contain a context for the data they represent.

Numbers represented graphically should be proportional in size to the numeric quantities they represent. The number of dimensions carrying information (size, color, location) should not exceed dimensions in the data. Variation should be shown for data only and not merely for the design of the graphic [2].

Section 3. Software Testing and the Visualization of an Epidemic

Communicating illness in a system to a business team eager for profits is a difficult task. All project managers have schedules, and if software quality threatens to derail the schedule, denial is often a popular fallback strategy. Thus, any visualization of quality must show problems in the system so explicitly as to be undeniable.

John Snow's map of a cholera outbreak accomplished this task and ultimately benefited the city of London with a much cleaner supply of drinking water. Figure 1 is taken from a reproduction of his map. Victims are marked with stacks of black lines (think black coffins piled high). The affected water pump is marked with a dot. Notice that there are no black marks at the neighborhood brewery, situated in the upper-right corner.

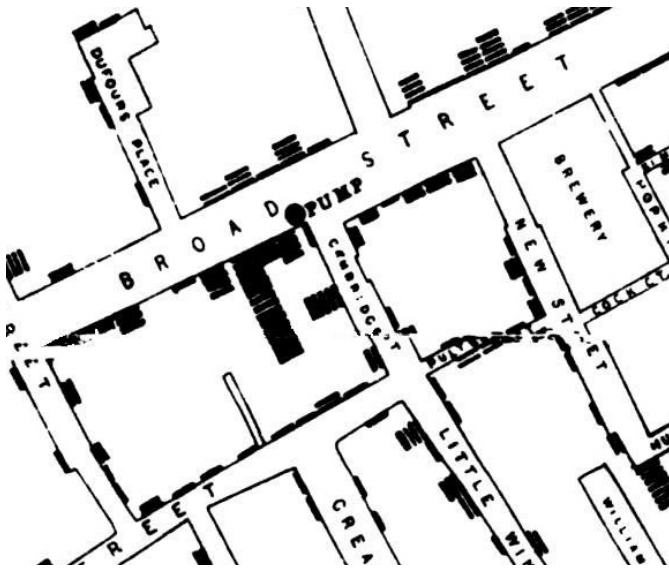


Figure 1. John Snow's Cholera Map

There are no marks at the brewery because the workers on-site were drinking their allotment of beer instead of drinking water from the contaminated Broad Street pump.

Snow's map is a bellwether not only because it proves, undeniably, a theory that the General Board of Health in London actively resisted, but because it is an example of compiling large amounts of data into a visualization depicting a dire problem. Snow had a list detailing the names and addresses of 83 people who had died from cholera. He also had an invaluable resource of detailed information on the neighborhood's residents in the form of local clergyman Henry Whitehead. While Whitehead tracked down and questioned everyone he possibly could about their drinking habits, Snow analyzed this data to form patterns of who had been *drinking* the water, who had *died* and, just as importantly, who had *not died*. Reverend Whitehead's authority and John Snow's data provided enough evidence for the parish board to remove the pump handle from Broad Street, which likely contributed to the end of the 1854 epidemic. Yet London's wider health community, along with the General Board of Health, remained unconvinced. Rather than attempting to share each and every story, Snow reduced the information into the map shown in Figure 1. If this type of visualization technique could improve the quality of a city's water, it can also be beneficial for the quality of software.

Moving forward approximately 150 years to the beginning of the 21st century, testers swim in a rising tide of tests and a sea of automated builds provided for us by the software testing tools of our highly connected, scripted and automated world. The number of tests for a system swells quickly. The amount of time between builds and releases continuously shrinks. In John Snow's time, the science of Epidemiology (the study of epidemics) was just beginning. The individual patient was the micro view of health whereas the health of a city or town was becoming the macro view of health. This required having access to data, such as a list of people who had died from cholera and where they had obtained their drinking water. The same sea change is now occurring in testing. One occurrence of a passing test is now a micro-level data point in a lengthy list of tests for components and sub-components executed several times. We now need a macro-level view of our tests. We need to see problems and complexities for our code, tests and bugs as they relate to a SUT's geography. Testers are in need of a *map*.

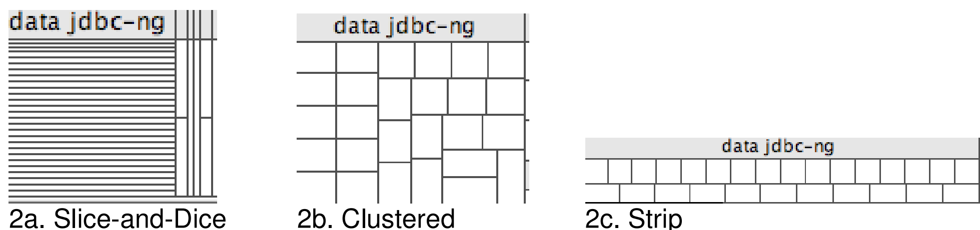
4. Using Treemaps to Visualize Source Code

4.1 Treemaps

The data measures, or graphic elements, on a map can perform multiple functions by showing both content and form. In the case of geography, for example, they show the shape and location of geographic units, whether these are a stream, road or topographical change in height. In an abstract sense, color and shading on a map can also indicate the level of a variable. Maps can thus contain very dense amounts of information in extremely small spaces. They can be examined at the macro and micro levels, revealing large, overall patterns across countries, states or lines of code. On the detail contained in maps, Edward Tufte writes, “The most extensive data maps… place millions of bits of information on a single page before our eyes. No other method for the display of statistical information is so powerful” [2].

A treemap is a visualization with an <u>extremely high data-ink ratio</u> used to represent hierarchically structured data. The viewer sees the entire structure of a tree on one screen, no matter how many branches and leaves occur within the tree's structure. Chartjunk is minimized. The first treemap display was created by Prof. Ben Shneiderman for displaying files on his computer [3].

Treemaps consist of nested rectangles representing the branches and leaves in a hierarchical structure. Branches form the borders and nesting within a treemap. The size and color of the boxes is based on attributes of the leaves. The size of any rectangle depends on some type of weighted factor. In the case of a treemap representing a file system, the directories and sub-directories form the borders, the sizes of the rectangles are based on file size and the color shows the file type. Larger directories take up more space in the visualization[3].



2a. Slice-and-Dice 2b. Clustered 2c. Strip

Figure 2. Treemap Layouts

The algorithm used to determine how the rectangles will be laid out on the screen is called the layout algorithm. The three most widely used and studied types of layouts are the original "slice-and-dice" algorithm and two variants, the "clustered " algorithm (sometimes referred to as "squarified") and the "strip." The slice-and-dice layout uses parallel lines to divide a rectangle representing a larger object into smaller rectangles representing sub-objects. At each level, the orientation of the lines are switched from vertical to horizontal. The clustered layout uses recursion to produce rectangles that have more of a uniform square shape than the slice-and-dice layout. The strip layout produces rectangles that are in order like the slice-and-dice layout but have a better aspect ratio for viewing like the clustered layout. Horizontal rows of rectangles are created in the strip layout [4].

Each layout has advantages and disadvantages. The slice-and-dice layout preserves the order of the data being represented and so produces treemaps in a consistent format. However, it creates long, skinny rectangles that are difficult to see, select, label and compare. While clustered layout treemaps are easier to view, if data is added to the dataset, order is not preserved and the entire layout changes. Maintaining the order within a dataset can be a key aspect when a viewer is looking for patterns in the data [4].

4.2 Metrics and Software Visualization

Bugs are often found by testing areas of code or features that are more complex. In using software metrics to assess code risk, also known as "code smell," testers often employ software metrics such as loc (lines of code) count, cyclomatic complexity, and CK (Chidamber and Kemerer) Metrics. Johnston et al point out that these metrics can sometimes be misleading or that they don't "tell the whole story. " Comparing metrics, however, can give testers a more complete picture of where the "smells" are [6].

Treemaps can visualize a comparison of metrics by using the size of a rectangle for one metric and the color of a rectangle for a second metric. For example, figure 3 shows a treemap visualizing java source code. The geography of the system is preserved because the hierarchy of the packages and classes is maintained. The size of the rectangles is determined by the loc count for each class and the color is determined by the weighted-methods per class (wmc).

WMC (Weighted Methods Per Class) is a metric proposed by Chidamber and Kemerer in their classic paper, "A Metrics Suite for Object Oriented Design." To calculate this metric, a complexity (specifically left by Chidamber and Kemerer to the reader) is applied to each method in a class. These complexities are then summed as weighted methods per class. If no complexity is applied to the methods, the metric simply becomes the number of methods [5].

By visualizing a comparison of software metrics for a system under test (SUT), testers can make better decisions about areas of code more likely to contain bugs or that would be more complicated to test. Figure 3 shows the component "build" in a SUT. Comparisons can be made about the sub-components. The largest class with a higher WMC complexity is JobModuleConfigurePanel. This class is larger and more complex than JobXMLContainer. JobXMLContainer is larger than APModuleConfig, but it has a lower WMC complexity.

Figure 3 Java Source Code and CK Metrics

Figure 3 is an illustration of areas where tradeoffs occur when assessing the need for testing resources. A component that is larger or more complex will require more test cases, more time and possibly more testers. If more resources are used for testing the larger components, this means that less resources will be available to test smaller components. Figure 3 shows that some of the smaller components in this SUT, despite their size, have a higher complexity than the larger components, and therefore, a higher risk of containing bugs. Minimizing the testing of JobBasicInfoPanel, APModuleConfig and JobOutputConfigPanel is less desirable once complexity is taken into account. It is possible to boil both the complexity and size down to one number, but this would create an oversimplification for a complex situation. Seeing loc and complexity together allows for more informed decisions about the amount of testing required.

5. Visualizing Quality with Treemaps

5.1 Visualizing Tests

Producing a treemap based on source code and source code metrics is fairly easy. Given the availability of data from a test case management system, it is also possible to create treemaps of tests and treemaps of defects. This begs the question, what would one want to discover about tests that would best be suited for the format of a treemap?

Mozilla uses the test case management software, Litmus, to track all of their testing in an online environment. This allows for volunteer participation and means that the vast majority of their test cases and test runs are available for viewing online. Currently, test cases are primarily shown 1 web page at a time. There are a few reports that show more tests, including most recent failures which shows failing test cases in groups of 50 at a time.

Figure 4 is an excerpt from a treemap visualizing 3859 Mozilla tests. This treemap shows the status of most of Mozilla's tests. The size of each rectangle is determined by the number of test steps in a test case. The color is determined by the number of failing test runs.

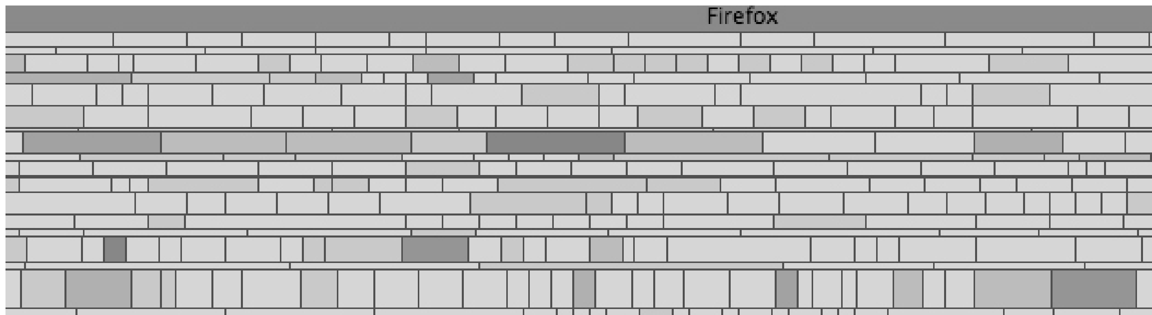


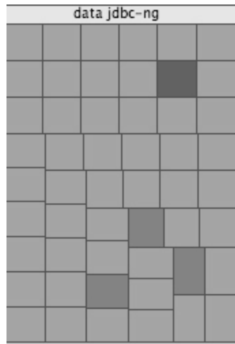
Figure 4. Approximately 150 of 3859 Firefox Tests in a Strip Treemap

Figure 4 uses the strip layout because this layout preserves the order of the data. This is in contrast to the clustered layout used to visualize source code. The clustered layout produces more uniform squares, but at the compromised loss of any order in the data represented. For viewing failed tests, preserving the order of the data has the benefit of showing clusters of failures. This provides a macro/micro view of where tests are failing at the higher product level or at the more finely grained level of individual tests.

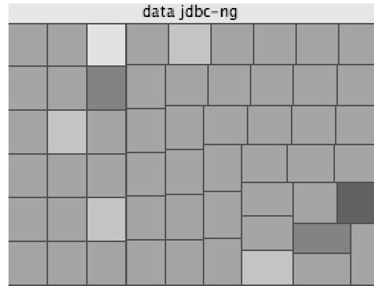
Figure 4 has a couple of issues. Data used in this treemap was parsed from html. Some tests had to be discarded because of badly formed html. This could have an effect on the accuracy of the failure clusters by de-emphasizing areas with failures that are not included. While the number of test steps can give a vague indication of test size, it is a very weak indication. This does not address how long a test may take to run or the complexity of the steps needed to run the steps. Plenty of research exists for source code complexity, but more research is needed for test case complexity.

5.2 Visualizing Defects

Defects visualized in a treemap can show areas of software that contain the most severe bugs and the most volume of bugs. When organized according to bug status, a treemap of bugs can show a bug's status in the bug lifecycle. If open and closed bugs are shown side by side, it is possible to see which areas of a system under test have had very severe bugs in the past and compare that with the severity of current bugs in the same area.



5a. Open Defects



5b. Closed Defects

Figures 5a and 5b show sections of a treemap created from the open source project, GeoTools. Figure 5a shows open defects for the component data jdbc-ng. Figure 5b shows closed defects for the same component. All of the defects are equally sized, and color is used to show severity. As the severity of a bug increases the color showing the severity becomes darker. Although 1 high severity defect and 2 medium severity defects have been closed, there is 1 severe defect remaining.

6. Conclusion and Future Directions

In this paper, I have introduced the concept of data visualization and how it can be useful for moving software quality forward. I have summarized Tufte's principles for data visualizing large data sets of multi-variate data. I have examined how the quality of source code is already visualized using CK metrics and applying Edward Tufte's principles of data visualization. Following that, I showed how this could also be applied to software quality data to help testers make decisions.

There are many opportunities for the future use of treemaps in testing. Some work has been to show antipatterns in treemaps [7], but this work is minimal. Future opportunities for visualizing quality branch out much further than treemaps. Performance testing was not included in this discussion but is ripe with opportunities particularly in the use of a stacked graph or a horizon graph. Stacked graphs have most recently been used to visualize music listening history on last.fm [8]. The horizon graph addresses a weakness of the treemap because it shows the time series for large numbers of components by using Edward Tufte’s concept of small multiples [9].

With some involvement from the field of artificial intelligence, not only could information such as test status and hierarchy be visualized, but the semantic content of tests could also be analyzed. The semantic data of tests and bugs could be mined and filtered to produce relationships among tests in structures such as a network graph.

Even though we are on the verge of using these newer types of visualizations, the basic principles of visualization should not be forgotten. They should be kept in mind and ready for use, even for charts as seemingly mundane as a test execution report. By practicing good visualization techniques with simpler charts and graphs we will have the ability "at the ready" to make the creative leap necessary for more complex forms of visualization that are rapidly becoming more necessary.

7. References

[1] Johnson, Steven. *The Ghost Map: The Story of London's Most Terrifying Epidemic - and How it Changed Science, Cities and the Modern World*. New York: Riverhead Books, 2006.

[2] Tufte, Edward R. *The Visual Display of Quantitative Information*, 2nd ed. Cheshire, Connecticut: Graphics Press, 2001.

[3] Johnson, Brian and Ben Shneiderman. “Treemaps: A Space Filling Approach to the Visualization of Hierarchical Information Structures.” *IEEE Conference on Visualization, San Diego, California, Oct. 22-25, 1991.* Ed. Gregory M. Nielson, Larry Rosenblum. Los Alamitos, CA: IEEE Press, 1991.

[4] Bederson, Benjamin B., Ben Shneiderman and Martin Wattenberg. “Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies.” *ACM Transactions on Graphics.* 21.4 (2002): 833-854.

[5] Chidamber, Shyam R. and Chris F. Kemerer. “A Metrics Suite for Object Oriented Design.” *IEEE Transactions on Software Engineering*. 20.6 (1994): 476 - 493.

[6] Johnston, Ken, Alan Page and B.J. Rollison. *How We Test Software at Microsoft*. Redmond, Wa: Microsoft Press, 2009.

[7] Langelier, Guillaume, Houari Sahraoui and Pierre Poulin. “Visualization-based Analysis of Quality for Large-scale Software Systems.” *IEEE/ACM international Conference on Automated Software Engineering, Long Beach, Ca, Nov. 7-11, 2005.* Ed. David Redmiles. New York: ACM Press, 2005. Pages: 214-223.

[8] Byron, Lee and Martin Wattenberg. “Stacked Graphs – Geometry & Aesthetics.” *IEEE Transactions on Visualization and Computer Graphics*. 14.6 (2008): 1245-1252.

[9] Agrawala, Maneesh, Jeffrey Heer and Nicholas Kong. “Sizing the Horizon: The Effects of Chart Size and Layering on the Graphical Perception of Time Series Visualizations.” *International Conference on Human Factors in Computing Systems, Boston, Ma, April 4-9, 2009*. Ed. Dan R. Olsen, Jr., Richard B. Arthur. New York: ACM Press, 2009. Pages 1303-1312.