Determining What Code to Inspect and What Code Not to Inspect

Aaron A. Kagawa
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
kagawaa@hawaii.edu

Abstract

Imagine that your project manager has budgeted 200 person-hours for the next month to inspect newly created source code. Unfortunately, in order to inspect all of the documents, given your project's previous rates of preparation time, 400 person-hours are required. However, your manager refuses to increase the budgeted resources to the necessary 400 person-hours. How do you decide between what documents need to be inspected and the documents that have to be skipped?

The classical definition of inspections does not provide any advice on how to handle this situation. For example, the notion of entry criteria used in Software Inspections determines when documents are ready for inspection [4] rather than if inspection is needed at all.

This proposed research would investigate how to apply the available budgeted person-hours of inspection to areas of the system that need it most. It is commonly assumed that defects are not uniformly distributed across all documents in a system - that a relatively small subset of a system accounts for a relatively large proportion of defects [1]. If inspection resources were limited, then it would be most effective to identify and inspect the defect-prone areas.

To accomplish this, I propose a two-part thesis claim. Claim 1: it is possible to devise an evaluation framework based upon automated process and product measures to distinguish documents that are in "most need of inspection" from those in "least need of inspection". Some examples of the process and product measures that are being considered include: reported defects, unit tests, coverage, active time, and number of changes. Claim 2: code deemed in most need of inspection will generate more critical issues than code deemed least need of inspection.

Each measure affects the determination of "most and least" differently. For example, I hypothesize that coverage should be weighted more than active time. Therefore, weights of each measure will be calibrated based on my initial hypotheses. My research will employ a very simple evaluation strategy, which includes selecting code to inspect, analyzing the results, and adjusting the calibration of the measures.

There are three milestones that I propose will be completed in this research. Milestone 1: implementation of Hackystat Extension, January 2005. Milestone 2: completed 6-week evaluation, March 2005. Milestone 3: thesis submission and defense, April 2005.

1. Introduction

The use of software inspections has reported outstanding results in improved productivity and quality. In fact, one study has found that if the inspection process is followed correctly, then up to 95 percent of defects can be removed before entering the testing phase [3]. Inspections have been so successful that it is likely to be the closest thing we have to a "silver bullet" for improving software quality.

In another success story, the Jet Propulsion Laboratory adopted inspections to identify defects and experienced a savings of 7.5 million dollars by conducting

300 inspections [2]. This statistic is very impressive, however what is not usually emphasized is that each inspection had an average cost of 28 hours. Using that average cost, the total cost for JPL's inspection process was 8,400 hours or roughly 4 years of work. This illustrates a fundamental problem with inspections; better results come from greater investment [5].

Not all organizations have the time or the money to invest in full or complete inspections. In most cases, organizations have limited funds or resources that can be devoted to inspections. For example, a manager can devote 200 hours of a project schedule for inspections. These organizations must pick and choose what documents to use those precious resources on. This realistic management of inspections directly contradicts the classical inspection adage of "when a document is ready you should inspect it". The bottom line is that most organizations cannot inspect every document.

The correct inspection process begins with the initiation phase, or sometimes called the planning stage, in which authors volunteer their documents for inspection [5]. The inspection leader then checks the document against entry criteria to determine if the document is ready for inspection [4] [5]. Again this process works very well for organizations, like JPL, that have the resources to inspect every document that is ready. However, I believe that this phase of inspection is a major problem for organizations that do not the necessary resources, because the process does not consider that some documents are "better" to inspect than others. A simple illustration of this fact is that 80 percent of defects come from 20 percent of the modules [1]. Thus, volunteering a document from that 20 percent will likely be "more in need of inspection" than in any other module.

Furthermore, the current literature [4] [8] [5] on inspections does not provide any specific insights into the trade offs between inspecting some documents and not inspecting others. However, Tom Gilb provides two recommendations when resources are limited; sampling and emphasizing inspecting up-stream documents [5]. The use of sampling involves inspecting various areas of a system to identify areas of interest. Up-Stream documents are documents that define high-level requirements or designs. The idea is that at the very least one should ensure that high-level documents are of high quality. Although, these are very useful recommendations, they do not provide much specific guidance of how best to use limited resources. At the end of the day, an organization with limited inspection resources must select documents to inspect.

The goal of this research is to optimize the selection of documents for inspection. To do this I will create a Hackystat extension that will determine what packages are in "most need of inspection" versus packages that are in "least need of inspection". There are several research questions that I must answer in order to make that determination. The most important question is the operational definition of the general terms "most need" and "least need". What software attributes can quantifiably distinguish between "most" and "least" need of inspection? In order to create a definition we must understand the motivation for inspections.

Software inspection has two primary goals; increase quality and productivity. For this research I am primarily concerned with increasing quality. The successful inspection of a document has two main results: finding defects which, once removed, increases software quality or not finding defects thus indicating high software quality. Software quality is vaguely defined as "the degree to which software possesses a desired combination of attributes" [7]. Some of the possible attributes can include: portability, reliability, efficiency, usability, testability, understandability, and modifiability [6]. Some other widely accepted measures of quality include defect density and complexity. Whatever definition used for quality, inspections aim to increase or validate the level of quality in software. Therefore, I would claim that the same attributes defining software quality also provide good indications of what code to inspect. For example, finding code that has low portability, reliability, efficiency, usability, testability, understandability, and modifiability would be a good indication of code that would be beneficial to inspect.

My thesis claims are as follows:

1. The attributes that define software quality provide good indications of what code to inspect.

2. Code that represents "high" software quality will have a low number of defects found in inspection.

3. Code that represents "low" software quality will have a high number of defects found in inspection.

2. Evaluation Methodology

To evaluate my thesis claims, I will create a Hackystat Extension that will determine what packages are in "most need of inspection" from 'packages that are in "least need of inspection". When the system makes a determination of what package is in most need of inspection, I will recommend a package for inspection. The results of the inspection (i.e. number of critical issues) will determine if the package accurately reflects the "most" and "least" need of inspection determination. To do this, I first need to gain a basic understand of the attributes that affect quality.

It is important to note that I am not defining a set of attributes that represent quality for all software projects. Instead, by using the Hackystat Extension I will be able to go through a methodology to best calibrate the attributes to accurately reflect the quality for the project that I am studying.

For this evaluation, I will study the implementation of the Hackystat System developed in the Collaborative Software Development Laboratory, of the University of Hawaii at Manoa. Although, this is a project to which I also contribute, I will minimize any possible data contamination by doing two things. First, I will keep the results of the "most" and "least" need of inspection a secret both during and after conducting the inspection. Second, I will not participate in the inspections themselves.

To evaluate claim 1, I will conduct several weekly mini-studies to fine-tune my calculation of quality. I will begin my study with several basic attributes defined in the literature that are believed to affect quality. Some of these attributes include size and coverage. Based on the quality level, I will then recommend a specific package for inspection by the CSDL staff. After the inspection, I will analyze the number of valid issues generated and their severity. Thus, I will be able to conclude if the quality level actually reflected a package that was in most need of inspection. I will continue to fine-tune the attributes of quality on a weekly basis. Once I have verified the attributes of quality, I can begin to evaluate claim 2 and 3.

To evaluate claim 2 and 3, I will recommend and collect data on the inspections for a 6-week period. During this period, I will specifically choose three high quality and three low quality packages. If my selected attributes are correct then the high quality packages should have considerably less issues generated by inspection than the low quality packages.

3. Hackystat Quality Extension

This section provides a short description of the Hackystat Quality Extension system. This system extends the functionality of the Hackystat System to provide the "most" and "least" need of inspection determinations.

The Hackystat System provides several Sensor Data Types that represent quantitative data about both the product and development process of a software project. Using this data I will build attributes that represent quality. For example, some of the attributes that are currently possible are the following:

1. Active Time
2. Number of Changes (Commits)
3. Date of Last Change
4. Number of Inspections
5. Date of Last Inspection
6. Number of Defects
7. Date of Last Defect
8. Lines of Code, Number of Methods, and Number of Classes
9. Lines of Test Code, Number of Test Methods, and Number of Test Classes
10. Coverage
11. Number of Executed Unit Tests
12. Dependency Metrics

Currently, each of these attributes is collected for each package or workspace within a specified project. Figure 1 shows several example high quality (or "least need of inspection") workspaces with their respective attributes of quality.

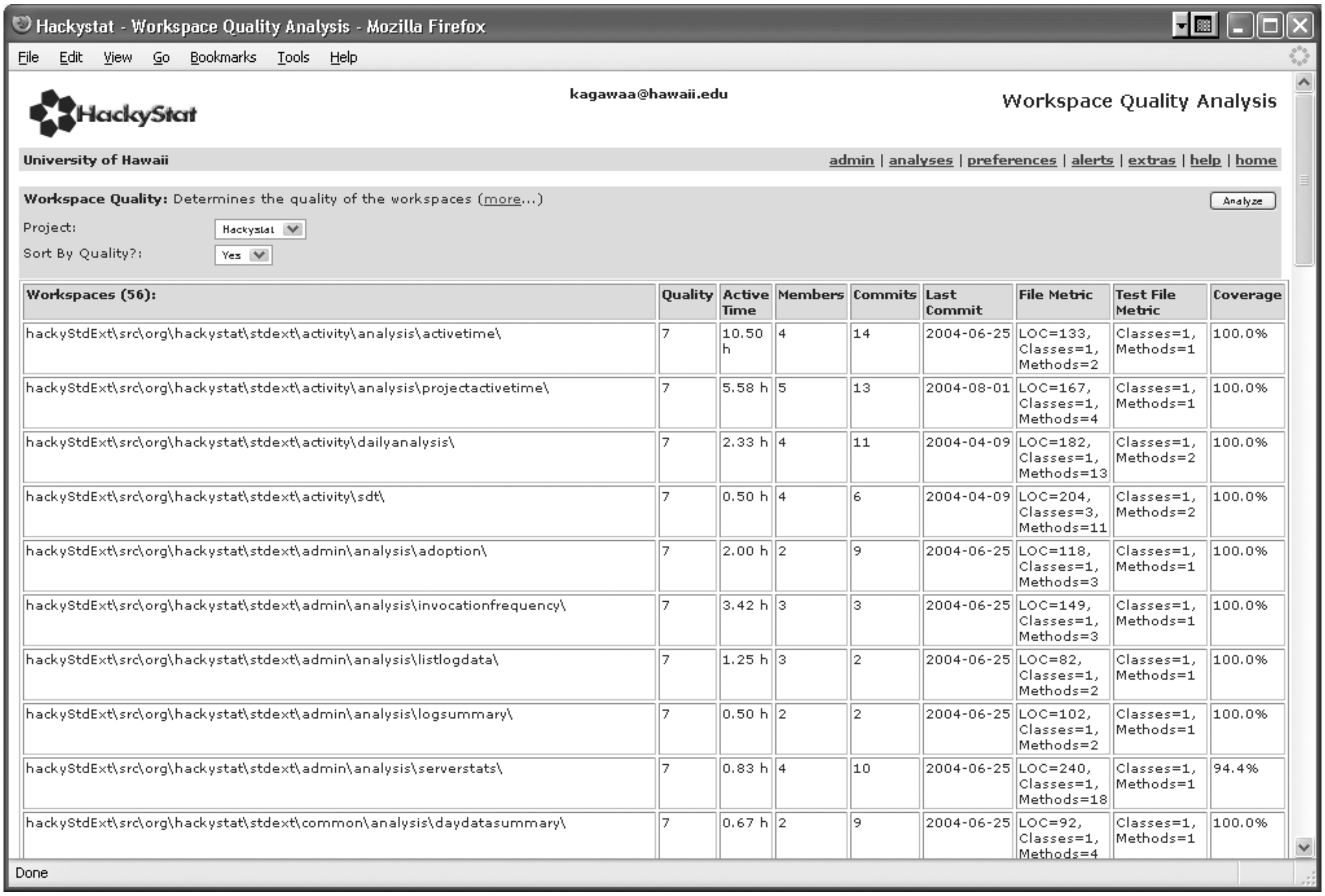

Figure 1. The Workspace Quality analysis. Workspaces are listed with its respective quality level and the attributes that make up its quality level.

To make the important determination of "most" and "least" need of inspection, I assign certain quality levels or numerical weights to the attributes. For example, if the coverage of a package is below 80 percent, I assign a "low" quality level for that attribute. Likewise, if the coverage of a package is a 100 percent, then I assign a "high" quality level. "Low" is operationalized by a 1, "high" is operationalized by a 3, and "middle ground" is operationalized by a 2. The system assigns each attribute a quality level and then assigns each package an aggregated quality level, which is the sum of the quality levels associated with its attributes. The packages are then sorted by the packages' aggregate quality level, sorting the "most need of inspection" to the bottom and "least need of inspection" to the top.

There are several issues with the assignment of numerical weights (or quality levels as I call them) that I still need to address. For example, I explicitly determine the quality levels using my own subjective measure of what is low versus high quality. I will need to explore if my subjective measure is sufficient, if some attributes should be weighted more than others, or if any other entirely different weighting methods provide more accurate results.

4. Initial Results

The use of the Hackystat Quality Extension system to provide the determination of "most" and "least" need of inspection has been promising. The initial implementation of the system has proven that it is technically possible to do what I have envisioned. In addition, I have already recommended the inspection of a package that was in "most need of a inspection" and the defects and issues identified have confirmed that the package had low quality.

Of course, I will continue to discover new attributes to define quality, fine tune the numerical weights associated with the attributes, and continue to recommend inspections until I believe my mechanism is ready for a thorough evaluation.

5. Contributions

If I find evidence that my thesis claims are true, then I believe the formal inspection process should address some sort of quantitative approach for initiating inspections.

In addition, I believe that the system's quantification of quality is valuable in of itself. Development teams can use the system's attributes of quality to guide the management of quality.

6. Timeline

References

[1] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 2001.

[2] M. Bush. Formal inspections–do they really help? In *Proceedings of the Sixth Annual Conference of the National Security Industrial Association*, Williamsburg, VA., April 1990.

[3] M. Bush and N. E. Fenton. Software measurement: A conceptual framework. *Journal of Systems and Software*, 12:223–231, 1990.

[4] R. G. Ebenau and S. H. Strauss. *Software inspection process*. McGraw Hill Inc, New York, 1994.

[5] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[6] R. L. Glass. *Facts and facilities of software engineering*. Pearson Education, Inc., Boston, MA, 2003.

[7] S. E. T. C. of the IEEE Compter Society. IEEE standard glossary of software engineering terminology. *IEEE-STD-729-1983 (New York; IEEE)*, 1983.

[8] K. E. Wiegers. *Peer reviews in software: A practical guide*. Addison-Wesley, Boston, MA, 2002.