# PRIORITY RANKED INSPECTION:
# IMPROVING THE COST-BENEFITS OF SOFTWARE INSPECTION SUPPORTING EFFECTIVE INSPECTION IN RESOURCE LIMITED ORGANIZATIONS

A THESIS PROPOSAL SUBMITTED TO MY THESIS COMMITTEE

MASTERS

IN

INFORMATION AND COMPUTER SCIENCES

By
Aaron A. Kagawa

Thesis Committee:

Philip M. Johnson, Chairperson
Daniel D. Suthers, Martha E. Crosby

February 8, 2005
Version 1.1.0

# 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 adequately, you estimate that it will take 400 person-hours. However, your manager refuses to increase the budgeted resources for the inspections. How do you decide which documents to inspect and which documents to skip?

The classic definition of inspection does not provide any advice on how to handle this situation. For example, the notion of entry criteria used in Software Inspection [7] determines when documents are ready for inspection rather than if inspection is needed at all [4].

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

To accomplish this research, I will construct a framework based upon automated process and product measures to distinguish documents that are “more in need of inspection” (MINI) from those “less in need of inspection” (LINI). Some of the process and product measures include: reported defects, unit tests, test coverage, active time, and number of changes. Based on this framework, I hypothesize that the inspection of MINI documents will generate more critical defects than LINI documents.

My research will employ a very simple evaluation strategy, which includes inspecting MINI and LINI software code and checking to see if MINI code inspections generate more defects than LINI code inspections. There are three milestones that measure my progress in this research. Milestone 1: Implementation of Hackystat Extension, January 2005. Milestone 2: Completed evaluation, March 2005. Milestone 3: Thesis submission and defense, May 2005.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Software inspection is defined as: “A formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems...” [7]. Software inspection, or software review as it is sometimes called, can have fantastic results: “Rigorous inspection can remove up to 90 percent of errors from a software product before the fist test case is run” [8, 3].

Since Michael Fagan invented the inspection technique in 1976, there have been many variations on the general concept of inspection. We now have Fagan Inspection [5], Software Inspection [7], High-Impact Inspection, Phased Inspection, “regular” software inspection, software reviews, code walkthroughs, inspections without meetings, and many more different twists on the original concept. Each of these techniques claim to be the best inspection method for their certain circumstances. For example, some argue that the inspection meeting is a waste of time and resources [10, 11, 12]. Others argue that the inspection meeting is critical for supporting social and educational aspects of inspection [11].

My research is different from traditional inspection research. Instead of asking how to conduct the inspection process, I ask how to determine what to inspect, when to conduct inspections, and more importantly if inspection is really needed for a particular piece of code. In this proposal, I will describe how the selection of a document for inspection can create problems for organizations with limited inspection resources. I will then propose a new inspection document selection technique called Priority Ranked Inspection and evaluate its effectiveness.

## 1.1 The Problem of Limited Resources for Software Inspections

The use of software inspection has reported outstanding results in improving productivity and quality. One study has found that when the inspection process is followed correctly, up to 95 percent of defects can be removed before entering the testing phase [3]. In another success story, the Jet Propulsion Laboratory (JPL) adopted inspection 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 emphasized is each inspection had an average total 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.

The JPL experience illustrates a fundamental problem with inspections: better results come from substantial investment [7]. 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 may only have 200 hours of a project schedule to allocate towards quality assurance including inspections. Such organizations must decide how to best utilize their limited inspection resources. 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 traditional inspection process begins with the initiation phase, or sometimes called the planning stage, in which authors volunteer their documents for inspection [7]. A inspection leader checks the document against entry criteria to determine if the document is ready for inspection [4, 7]. Again this process works very well for organizations, like JPL, that have the resources to inspect every document after every significant change. However, I believe that this phase of inspection is a major problem for organizations that do not have 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 system [1]. Thus, volunteering a document from the defect-prone 20 percent will likely be "more in need of inspection" than any other part of the system.

Furthermore, the current literature [4, 14, 7] on inspections does not provide specific insights into the trade-offs between inspecting some documents and not inspecting others. However, Tom Gilb and Dorothy Graham provide two recommendations to use when inspection resources are limited; sampling and emphasizing up-stream documents [7]. 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. Inspecting up-stream documents ensures that

the requirements are correct before any implementation is started. Although, these are very useful recommendations, they do not provide much specific guidance of how best to use limited inspection resources.

## 1.2 The Priority Ranked Inspection Approach

To address some of the problems associated with conducting inspections with limited resources, I propose a new inspection process called "Priority Ranked Inspection", (PRI). The primary goal of PRI is to optimize the selection of documents for inspection by distingushing what documents are "more in need of inspection" (MINI) versus documents that are "less in need of inspection" (LINI). In addition, PRI will rank each document according to this determination in hopes of prioritizing the documents that need to be inspected. The converse is also true: PRI will identify documents that might not need to be inspected.

As I have shown in the previous section, it is extremely difficult for organizations with limited inspection resources to inspect every document before it exits the development process. Therefore, unlike traditional inspection process, PRI does not require that all documents be inspected. Instead, PRI is intended to help these organizations in two ways. First, PRI is intended to be able to identify documents in the current development process that need to be inspected. This will allow organizations to make an educated guess at what documents need to be inspected and what documents can be skipped. Second, it is unavoidable that some documents with critical defects will finish the development process without being inspected. Therefore, PRI is also intended to identify documents for inspection regardless if a document is currently in the development process or not.

There are four primary steps in the Priority Ranked Inspection (PRI) process. The following list is short description of each of the steps. The following sub-sections provide a summary description of each step.

1. The creation of the PRI weighting function, which distinguishes MINI documents from LINI documents. The weighting function design includes two steps:
   (a) Selection of product and process measures to use in the PRI weighting function.
   (b) Creation of a numerical weighting system that assigns a weight for each measure and the calibration of this weighting system.
2. The selection of a document for inspection based on the PRI ranking.

3. The actual inspection of the selected document.

4. Adjustment of product and process measure selection and calibration based on the results of the inspection.

### 1.2.1 Step 1a: Selection of Product and Process Measures

The PRI weighting function which distinguishes MINI documents from LINI documents will be generated automatically from various product and process measures. Product measures are usually obtainable from direct analysis of source code. For example, lines of code, complexity, and number of children are a few examples of product measures. On the other hand, process measures are collected from the actual software development process. The amount of developer 'effort' and the number of defects are examples of process measures. One might ask, what specific measures should PRI consider? The answer: it depends on the specific situation. Different projects and organizations could have a different set of measures in defining the optimum PRI weighting function and ranking. Therefore, a major component of the PRI process is the selection of the measures.

Software quality measures are one example of the type of product and process measures that could be used in PRI. 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" [13]. Some of the possible measures of quality include: portability, reliability, efficiency, usability, testability, understandability, and modifiability [8]. 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, the same measures of software quality will also provide good indications of what documents need inspection. For example, finding documents that have low portability, reliability, efficiency, usability, testability, understandability, and modifiability would provide a good indication that the documents are MINI.

Based on the previous example, the quality-specific product and process measures can be extracted and used in the PRI weighting function. Table 1.1 is an example of the PRI ranking of a software project that contains three documents. Presented in the table are the measures and values that are used in the PRI ranking. Due to constraints of the paper size, the table presents only a

couple of the measures discussed above. However, any number of measures can be used in the PRI weighting function.

Table 1.1. Step 1a - Example PRI ranking - After Measure Selection

| Document | PRI Ranking | Reliability | Efficiency | Testability | ... |
|---|---|---|---|---|---|
| Foo.java | MINI | 3 | 4 | 2 | ... |
| Bar.java | LINI | 4 | 2 | 6 | ... |
| Baz.java | LINI | 1 | 2 | 3 | ... |

Table 1.1 is an illustration of a fictitious PRI ranking. See Chapter 4: Hackystat PRI Extension for more details about the exact calculations necessary to create a PRI weighting function and ranking.

### 1.2.2 Step 1b: Calibration of the Product and Process Measures

Only selecting what measures to be used in the PRI weighting function is not adequate, because some measures are more important than others. For example, an organization may find that testability has a greater positive impact on the PRI weighting function than efficiency. Therefore, Step 1b of the PRI process is the calibration of the measures' importance. The calibration of the measures is based on a numerical weighting system. Each measure will be assigned a numerical weight and will be individually calibrated. The numerical weighting system and the calibration creates a PRI ranking, which provides a priority ranking of the documents.

Using the same example (Table 1.1) from the previous section, imagine that the organization has found testability to be a leading indicator in defect prevention. Therefore, the calibration is adjusted and the values of testability are given a higher weight than the other measures. This finding changes the PRI weighting function and ranking. The following table (Table 1.2) shows the new PRI ranking after the calibration.

Table 1.2. Step 1b - Example PRI ranking - After Measure Calibration

| Document | PRI Ranking | Reliability | Efficiency | Testability | ... |
|---|---|---|---|---|---|
| Bar.java | MINI | 4 | 2 | 6 | ... |
| Foo.java | MINI | 3 | 4 | 2 | ... |
| Baz.java | LINI | 1 | 2 | 3 | ... |

Notice that, as a result of the calibration, the PRI ranking for Bar.java has changed from LINI to MINI. This illustrates that the PRI weighting function and ranking can be flexible and that it has the pontential to reflect different development processes within different organizations.

### 1.2.3 Step 2: Selecting a Document for Inspection Based on the PRI Ranking

After the PRI weighting function and ranking is in place, an organization may use PRI to select documents for inspection. To select a document for inspection they simply consult the PRI ranking and find a MINI document (a document deemed more in need of inspection). This ranking will help constrain the number of possible documents that can be considered for inspection.

For example, in the example presented in Table 1.2, an organization should select Bar.java for the inspection, because it ranked the highest of the three documents. During the next inspection meeting, Foo.java should be inspected. However, Baz.java could be skipped, thus saving inspection resources.

Using the PRI ranking to select documents for inspection has three primary benefits. First, it can enhance the selection or the volunteering process of a document for inspection. Second, it can identify documents for inspection that a volunteering process typically does not. Third, inspecting a MINI document will generate more critical defects than inspecting a LINI document.

### 1.2.4 Step 3: Conducting an Inspection of the Selected Document

Once the document is selected, a traditional inspection process can begin. PRI does not have any special processes for this step. An organization can choose to use any traditional inspection process (i.e., Software Inspection, Fagan Inspection, In-Process Inspection). In other words, the PRI process is an outer layer that wraps around an already established inspection process to enhance the selection of documents. Therefore, in this research I will not discuss or evaluate traditional inspection concepts like; inspection leaders, preparation time, etc.

### 1.2.5 Step 4: Adjustment of the Measure Selection and Calibration

After the inspection of a document, the results can be used to further improve the PRI weighting function and ranking. For example, if the PRI weighting function appears to be incorrect, because it ranked a document as MINI but no major defects were found, then the PRI weighting function should be adjusted to classify this document as LINI. This adjustment can be achieved in two ways. First, one could add more product and process measures to make the PRI weighting

function more robust. Alternatively, one could calibrate the current measures to refine and correct the PRI weighting function and ranking. In either case, an incorrect PRI weighting function will provide data to help make future PRI weighting functions better. This process should be an ongoing evolving activity.

For example, consider the example presented with Foo.java, Bar.java, and Baz.java. If an organization has found results that suggest that efficiency is a leading indicator of defects, then it should be calibrated with a higher weight.

## 1.3 The Hackystat PRI Extension

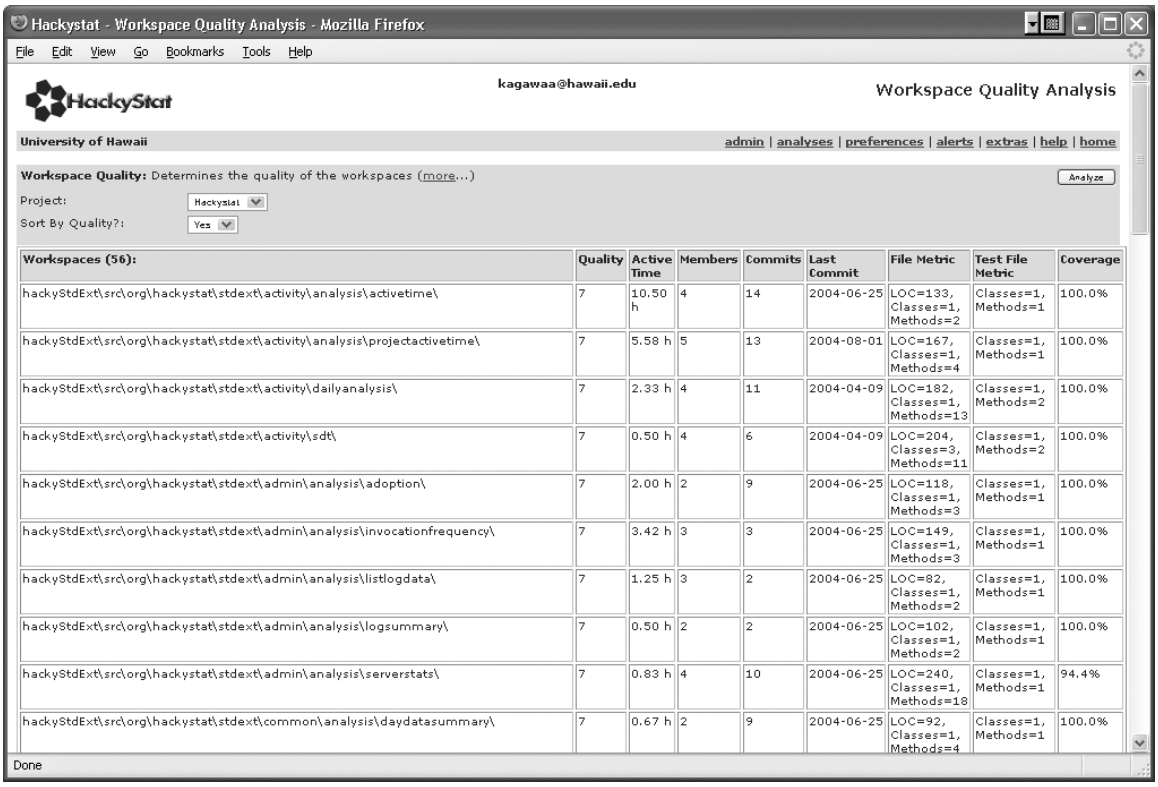
To successfully use PRI, the determination of MINI and LINI must be obtainable for a very low cost. In other words, if the weighting function takes three months to generate, a software project will have long past the need for those specific recommendations. Therefore, this determination must be obtained in real-time.

One way of obtaining weighting function values in real-time is through the use of the Hackystat system. Hackystat is a framework for collecting and analyzing software product and development process metrics in real-time. For more information about the Hackystat system see Chapter 3. For this proposed research, I have created an extension to the Hackystat system called the Hackystat PRI Extension (hackyPRI for short). hackyPRI provides a real-time PRI weighting function and ranking. Figure 1.1 demonstrates the PRI weighting function and its ranking for a software project that is obtainable from the Hackystat PRI Extension.

The Hackystat PRI Extension will be implemented to fully support all 4 steps of the PRI process. The next subsections demonstrate how hackyPRI supports each of the 4 steps. It is important to note that PRI is a proposed *process*, therefore many different tools can support it. Using the Hackystat PRI Extension is not required to conduct PRI inspections.

### 1.3.1 Step 1a: Selection of Product and Process Measures

The Hackystat system provides a standard set of product and process measures, these measures are called Sensor Data Types within the Hacksytat system, and the means of its collection. Initially, I have developed hackyPRI to use all of the measures obtainable from Hackystat. In addition, I have begun to implement a several new Hackystat Sensor Data Types (measures), that I feel will aid the PRI weighting function.




| Workspaces (56): | Quality | Active Time | Members | Commits | Last Commit | File Metric | Test File Metric | Coverage |
|---|---|---|---|---|---|---|---|---|
| hackyStdExt\src\org\hackystat\stdext\activity\analysis\activetime\ | 7 | 10.50 h | 4 | 14 | 2004-06-25 | LOC=133, Classes=1, Methods=2 | Classes=1, Methods=1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\activity\analysis\projectactivetime\ | 7 | 5.58 h | 5 | 13 | 2004-08-01 | LOC=167, Classes=1, Methods=4 | Classes=1, Methods=1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\activity\dailyanalysis\ | 7 | 2.33 h | 4 | 11 | 2004-04-09 | LOC=182, Classes=1, Methods=13 | Classes=1, Methods=2 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\activity\sdt\ | 7 | 0.50 h | 4 | 6 | 2004-04-09 | LOC=204, Classes=3, Methods=11 | Classes=1, Methods=2 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\adoption\ | 7 | 2.00 h | 2 | 9 | 2004-06-25 | LOC=118, Classes=1, Methods=3 | Classes=1, Methods=1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\invocationfrequency\ | 7 | 3.42 h | 3 | 3 | 2004-06-25 | LOC=149, Classes=1, Methods=3 | Classes=1, Methods=1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\listlogdata\ | 7 | 1.25 h | 3 | 2 | 2004-06-25 | LOC=82, Classes=1, Methods=2 | Classes=1, Methods=1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\logsummary\ | 7 | 0.50 h | 2 | 2 | 2004-06-25 | LOC=102, Classes=1, Methods=2 | Classes=1, Methods=1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\serverstats\ | 7 | 0.83 h | 4 | 10 | 2004-06-25 | LOC=240, Classes=1, Methods=18 | Classes=1, Methods=1 | 94.4% |
| hackyStdExt\src\org\hackystat\stdext\common\analysis\daydatasummary\ | 7 | 0.67 h | 2 | 9 | 2004-06-25 | LOC=92, Classes=1, Methods=4 | Classes=1, Methods=1 | 100.0% |

Figure 1.1. The Workspace PRI analysis. Workspaces are listed with its respective PRI ranking and measures.

Each column in Figure 1.1 is a measure that is obtainable from Hackystat. Each measure will be automatically and unobtrusively collected by Hackystat and directly fed in to the hackyPRI extension.

### 1.3.2 Step 1b: Calibration of Product and Process Measures

Each measure and its numerical weight will be stored within the hackyPRI extension. The numerical weights are not shown in Figure 1.1. However, the calibration and weighting system works behind the scenes to rank each document. For example, the coverage measure is assigned a numerical weight of 3 if the coverage value equals 100 percent, 2 if the coverage value is below 90 percent, 1 if the coverage value is below 80 percent, and 0 if the coverage value is below 70 percent. See Chapter 4 for a detailed description about the numerical weighting system.

### 1.3.3 Step 2: Selecting a Document for Inspection Based on the PRI Ranking

Using the PRI Hackystat analysis, an organization should select a document at the bottom of the PRI ranking table for inspection. The higher the document is in the table, the more it is likely to be LINI.

### 1.3.4 Step 3: Conducting an Inspection of the Selected Document

Once a document is selected it can be inspected. One interesting side effect of the PRI ranking is that specific statistics and measures can be presented during the inspection process. For example, if a document is selected because it has low coverage, then the inspection can focus on why the coverage is low.

### 1.3.5 Step 4: Adjustment of the Measure Selection and Calibration

If a document is shown to be incorrectly ranked, then an adjustment of the PRI weighting function is necessary. This can be accomplished by adding more Hackystat measures to the PRI weighting function or recalibrating the numerical weights associated with the measures. See Chapter 4 for a detailed description of the calibration process for the hackyPRI extension.

## 1.4 Thesis Statement

The thesis statement of this research is as follows; Priority Ranked Inspection can distinguish documents that are more in need of inspection (MINI) from others that need inspection less (LINI). This thesis statement can be decomposed into the following three main claims, which are based on the intended benefits of PRI.

1. PRI can enhance the volunteer-based document selection process.
2. PRI can identify documents that need to be inspected that are not typically identified by volunteering.
3. Documents that are deemed more in need of inspection (MINI) will generate more critical defects than documents deemed less in need of inspection (LINI).

My first claim states that PRI can enhance the selection process. In the traditional inspection process, this selection process is based on a developer selecting and volunteering a document

for inspection. This claim is an intended benefit of PRI because in the traditional inspection process the number of documents that a developer must select from can vary widely. If PRI can provide the MINI documents, then the developer can focus his selection on a smaller set of documents.

My second claim states that PRI can identify documents that have slipped through the cracks in the development process. For an organization with limited inspection resources it is not possible to inspect every document. Therefore, it is inevitable that some documents that need to be inspected have not been.

My third and last claim states that the inspection of MINI documents will generate more critical defects than LINI documents. This claim is very important to the PRI process because if this claim is proven to be false, then the PRI process cannot solve the limited inspection resource problem.

## 1.5 Evaluation

This section provides a short description of the methodologies used to evaluate my thesis claims. Chapter 5, Evaluation Methodology provides a detailed explanation of the methodologies and procedures that will be used in the evaluation of PRI.

I will evaluate the main thesis of this proposed research by testing each of my three claims. In this evaluation, I will be studying the inspection process of the Hackystat system developed by the Collaborative Software Development Laboratory. I will also be using the developers of Hackystat as subjects in my evaluation.

My first claim states that PRI enhances the volunteer-based document selection process. To evaluate this claim, I will conduct a qualitative and quantitative evaluation. First, I will assess the developers' current selection process by asking them to rank a few documents based on what documents they think are more in need of inspection. Then I will provide them with the PRI ranking of those same documents and ask them which rankings would they change. After this qualitative evaluation, I will ask CSDL to inspect a few documents to evaluate the validity of the developers' subjective ranking and the PRI ranking.

My second claim states that PRI can identify MINI documents that are not typically identified by the volunteering process. To evaluate this claim, I will ask CSDL to inspect a few documents that have not been identified in the previous evaluation.

My third and last claim states that the inspection of MINI documents will generate more critical defects than LINI documents. Throughout the previous two studies I will have collected

information about approximately 20 inspections. By quantitatively analyzing the results of these inspections I will be able to provide supporting evidence for this claim.

## 1.6 Structure of the Proposal

The remainder of this proposed research is as follows. Chapter 2 discusses previous studies that influenced this research. Chapter 3 and Chapter 4 contains a detailed description of the Hackystat system and the Priority Ranked Inspection (PRI) Hackystat extension. Chapter 5 discusses the evaluation methodology that will be implemented to evaluate the claims and benefits of PRI. Chapter 6 discusses the contributions and future directions of this research. Finally, Chapter 7 provides a detailed timeline of this proposed research.

# Chapter 2

# Related Work

This chapter presents the work related to Priority Ranked Inspection. The initial invention of PRI can be attributed the current traditional inspection literature's consistent lack of information on the selection of documents for inspection. Previous work on software inspection has focused on the process in which inspection is conducted. Instead, this proposed research focuses on the selection of documents for inspection.

## 2.1 Fagan Inspection

Michael E. Fagan invented inspections in 1976 while working at IBM. "Inspection", with a capital "I", or the term "Fagan Inspection" is used when referring to his technique. Using Fagan Inspection, Bell Labs reported 14 percent productivity increase, better tracking, early defect detection, and more importantly the employees credited Fagan Inspection with an "important influence on quality and productivity" [7].

## 2.2 Software Inspection

One of the most widely accepted type of inspection is "Software Inspection", which was developed by Tom Gilb and Dorothy Graham in the book of the same title. Software Inspection is based on the Fagan-style Inspection and is generally more robust and disciplined than other techniques.

Software Inspection is defined as a two part process, product Inspection and process improvement. According to the Software Inspection literature, product Inspection and process improvement cannot and should not exist without one another.

### 2.2.1 Project Inspection

There are ten lengthy steps in the product Inspection portion of the Software Inspection process. I have provided a short description of each of the steps in the following sections.

*Request: Initiating the Inspection Process*

This Inspection process begins with an author's voluntary request for an Inspection. The request is delegated to an Inspection leader. An Inspection leader is a trained-and-certified employee and is generally not a manager. It is the leader's responsibility to organize, plan and conduct the inspection.

*Entry: Making Sure 'Loser' Inspections don't Start*

The Inspection leader is required to check the volunteered document against an Entry Criteria. This criterion ensures that the document is worth inspecting. The leader conducts a quick look through the document to assess the initial quality of the document. For example, the author has spent an adequate amount of time working on the document, there are a minimal number of minor defects, etc. "The purpose of having entry criteria to the Inspection process is to ensure that the time spent in Inspecting the product and associated documents is not wasted, but is well spent" [7].

*Planning: Determining the Present Inspection's Objectives and Tactics*

If, and only, if the document has successfully passed the entry criteria, then the Inspection leader can begin to plan the Inspection. This includes many managerial tasks; inviting participants, scheduling an Inspection meeting, gathering supportive documentation, establishing average optimum checking rates and suggesting areas of possible improvement in the document.

*Kickoff Meeting: Training and Motivating the Team*

The purpose of a kickoff meeting is to ensure that Inspection process begins correctly. This includes dispensing required documents and explaining the expectations of the participants. This meeting saves time by dispensing the necessary information, which is needed to conduct the Inspection. This meeting is also an opportunity to introduce process changes in the Inspection process.

*Individual Checking: The Search for Potential Defects*

The participants, or " checkers", are required to work alone to find potential major defects in the documents provided. These defects are generally identified with the aid of rules, checklists, and other standards of the organization.

*Logging Meeting: Log Issues Found Earlier and Check for More Potential Defects*
This meeting has three purposes: log the issues generated in the individual checking phase, discover more major defects, and identify possible ways of improving the inspection process. This meeting is conducted and moderated by the Inspection leader.

*Edit: Improving the Product*
The overall goal of Inspection is to remove the defects that were found. During this phase, the author is given a list of the issues (issues become defects if they are deemed as valid defects) that were identified and is required to make the necessary improvements to remove any defects from the document.

*Follow up: Checking the Editing*
The purpose of this phase is to ensure that the Edit phase was correctly executed by the author. The Inspection leader must ensure that all issues are correctly classified, either as valid defects or invalid issues and that the author has corrected all known defects.

*Exit: Making Sure the Product is Economic to Release*
The Inspection leader consults the exit criteria to determine if the inspected document contains a certain level of quality as defined by the exit criteria. For example, the Exit criteria can contain rules that specify: successful Follow Up phase completion, certain metrics about this particular Inspection was recorded and within limits, and that the number of defects are below a certain threshold.

*Release: The Close of the Inspection Process*
This is the last phase of the Inspection process. At this point, the document can be officially released and the Inspection process is concluded. However, if it is determined that there are some acceptable/unavoidable defects remaining in the document, then such defects must be documented.

### 2.2.2 Process Improvement

Equally important to the product Inspection portion of Software Inspection is process improvement. Process improvement is the continuous improvement of the entire software development process. The idea is simple; Inspections can remove defects, but process improvement can prevent defects.

In Software Inspection, process improvement can be accomplished in many ways. A low-cost procedure could be as simple as discussing the cause of the defects. This discussion takes place

in a Process Brainstorming Meeting. “The purpose of the process brainstorming meeting is *not* to deal with the document and its defects. It is to deal with the *causes* of those defects” [7].

On the other hand, process improvement can be very expensive, Process Change Management Teams. Specialized teams can be formed to collect and analyze the metrics that are obtained from the conducted Inspections.

## 2.3 Inspection and Priority Ranked Inspection

The different types of traditional inspection processes explained above are quite different from the Priority Ranked Inspection (PRI) process that I am proposing. The biggest difference is traditional inspection processes provide guidelines on *how to conduct inspections* and PRI provides guidelines on *what to inspect*. In fact, the PRI process does not provide any guidance on how to inspect the documents once they are selected. Instead, PRI is a document selection process that wraps around any traditional inspection process.

### 2.3.1 Lack of Discussion about Selection of Documents

In the book, “Software Inspection”, Tom Gilb and Dorothy Graham provide very few paragraphs on the subject of document selection. The following is the entire paragraph that discusses document selection.

> *The starting point for any Inspection is the request from the author of a document that the document be Inspected. Inspection is always voluntary, and authors must not be coerced into ’volunteering’ documents against their will.*
> *Authors are motivated to request Inspection for two reasons:*
>
> 1. *they will get help to upgrade their document before official release;*
> 2. *they must achieve exit status in order to claim that they have met a deadline, and that the quality of their work is really good enough.*

In addition, several other books address document selection for inspection with even less words. The following is one of the few sentences about the initiation of the inspection process in Karl Wiegers’s book, “Peer Reviews in Software” [14].

> *The author initiates planning by announcing that a deliverable will soon be ready for inspection.*

The traditional inspection literature fails to address several key areas of selecting a document for inspection.

1. What happens when an organization does not have enough resources to inspect every document that is ready? Inspections are expensive. It can consume 15 percent of the projects budget [7]. What happens to the volunteering process when an organization can inspect one in every five documents?

2. What happens when two authors volunteer two different documents at the same time? Which document should be selected? Selecting what to inspect from two documents is not difficult. However, what if there are twenty or a hundred different documents that are waiting to be inspected.

3. Defects can occur in documents that already "exited" the development and inspection process, can these documents be inspected? The current literature suggests a linear development process, which documents that have been inspected and completed the development process are never to be inspected again.

I strongly believe that the selection of documents for inspection is a complicated process that warrants much more attention than the current inspection literature provides.

### 2.3.2 Cost Cutting

> *"The bottom line is that I [Tom Gilb] believe that it is more relevant to view Inspection as a way to control the economic aspects of software engineering rather than a way to get 'quality' by early defect removal" [6].*

Tom Gilb is correct. If inspections do not provide an economic benefit, then why do them at all? However, with an estimated 10-15 percent of a project's budget that is required to conduct successful inspections, it is difficult for organizations with limited inspection resources to correctly implement the suggested process. The bottom line seems to be that not all organizations can invest 10-15 percent of their budget to inspections.

In Software Inspection, there are three primary ways to reduce the resources; sampling, inspecting up-stream documents, and focusing on major defects. The practice of sampling suggest that instead of inspecting an entire document, pick one to four representative portions of the document. The practice of inspecting up-stream documents suggests that requirement and design documents need to be correct before programming can begin. Focusing on major defects suggests that minor defects, such as code comments, are irrelevant to the customer-performance of the system and should be ignored.

In my opinion, Software Inspection and other traditional inspection literature does not address the most obvious way to save resources, which is minimizing the number of documents that need to be inspected. The current literature suggests that inspections are a "gateway" to complete the document's development process. This process works well for organizations that have the resources to treat it as such. However, for organizations with limited inspection resources, inspecting every document is quite impossible. In contrast to traditional approaches, Priority Ranked Inspections embraces the notion of skipping the inspection of some documents.

### 2.3.3 Volunteering

Another problem associated with the selection of documents for inspection is the notion of volunteering. Notice in Section 2.3.1 the term 'voluntary' is emphasized in Gilb and Graham's selection process. Yet, in the very same book, "Software Inspections", contains a case study of Software Inspections used in a company where documents were required to be inspected rather than volunteered.

1. "Most who tried inspections responded with enthusiasm, but only four groups were continuing to do inspections - not surprisingly, those groups in which the manager *required* them. Most groups tried a few inspections, then interest waned as deadlines approached. A few managers ignored inspections altogether, citing schedule pressures as the reason."

2. "The vice president of marketing notified his department that inspections were *required* for approval of all mandatory documents produced."

3. "Each year, development groups are *required* to inspect more of their pre-code documents."

4. "Code inspections remain optional at least until *100 percent* of the pre-code documents are inspected."

# Chapter 3

# The Hackystat System

The Priority Ranked Inspection process is a theoretical process that can be implemented in many different ways. For this proposed research I will utilize the Hackystat system to aid the implementation of PRI. This chapter is an introduction to the Hackystat system which was invented by Dr. Philip M. Johnson, in the Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawaii at Manoa.

## 3.1 Overview of the Hackystat System

The Hackystat system is an open-source software framework for the automated collection and analysis of software product and process measures. Product measures can be defined as, measures that are obtainable from direct analysis of source code. For example, some product measures can include: lines of code, complexity, and the number of unit tests. Process measures can be defined as, measures that are obtainable from the actual development process which creates the source code. For example, the number of developers, the developers' "effort", the number of major releases and the number of defects, are examples of process measures.

The following list summarizes the features that Hackystat provides: [9]:

1. Hackystat utilizes custom "sensors" that are "attached" to various software development tools. Theses sensors unobtrusively collect data on various software product and development process measures.

2. Hackystat supports any and all software projects, development processes, software development environments, operating systems, and development tools.

3. Hackystat supports in-process project management by providing a set of extendible analyses of the product and process measures that are collect by the sensors.

4. Hackystat is well suited for empirical software development experimentation.

The Hackystat system is a mature and extendible software system. Currently, Hackystat is being utilized by NASA's Jet Propulsion Laboratory, Sun Microsystems, IBM, University of Torino, University of Maryland, and of course the University of Hawaii.

# Chapter 4

# Hackystat Priority Ranked Inspection Extension

This chapter provides a detailed description of the Hackystat PRI (hackyPRI) extension. hackyPRI extends the functionality of the Hackystat system to provide the PRI determination of more and less in need of inspection (MINI and LINI). I will discuss how hackyPRI supports the four steps of the Priority Ranked Inspection process.

## 4.1 The Four Steps of the Priority Ranked Inspection Process

Hackystat PRI Extension supports the four steps of the Priority Ranked Inspection process. The following list is the four steps of the PRI process.

1. The creation of the PRI weighting function, which distinguishes MINI documents from LINI documents. The weighting function design includes two steps:
   (a) Selection of product and process measures to use in the PRI weighting function.
   (b) Creation of a numerical weighting system that assigns a weight for each measure and the calibration of this weighting system.
2. The selection of a document for inspection based on the PRI weighting function and ranking.
3. The actual inspection of the selected document.
4. Adjustment of product and process measure selection and calibration based on the results of the inspection.

The following subsections detail how hackyPRI supports the steps.

### 4.1.1 Step 1a: Selection of Product and Process Measures

The Hackystat system provides a set of Sensor Data Types that represent various software product and process measures. I will use a subset of the available Sensor Data Types as the measures that make up the PRI weighting function. Table 4.1 contains a description of the measures that will be used in the hackyPRI extension.

Each measure is collected for each package or workspace within a specified project. Figure 4.1 shows several example LINI packages.

### 4.1.2 Step 1b: Calibration of Product and Process Measures

Each measure and its numerical weight will be stored within the hackyPRI extension. The numerical weights are not shown in Figure 4.1, however the calibration and weighting system works behind the scenes.

To make the important distinction of MINI and LINI, I assign certain numerical weights to the measures. For example, if the coverage of a package is below 80 percent, I assign a "low" weight for that measure. If the coverage of a package is a 100 percent, then I assign a "high" weight. "Low" is operationalized by a 1, "high" is operationalized by a 3, and "middle ground" is operationalized by a 2. The system assigns each measure a weight after analyzing its value. Table 4.2 contains a description of the weighting used in the hackyPRI extension.

After all measures are assigned a weight, the weights are aggregated to provide a combined weight for each package. The packages are then ranked by the packages' aggregate PRI level, sorting the MINI packages to the bottom and LINI packages to the top.

There are several issues with the assignment of numerical weights that I still need to address. For example, I explicitly determine the weights 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 measures should be weighted more than others, or if any other entirely different weighting methods provide more accurate results.

Table 4.1. Measures used in hackyPRI

| **Measure** | **Description** |
|---|---|
| Expert | The developer who has the most active time and commits. An expert represents the developer who is most familiar with the particular portion of the system. |
| Active Time | The total time developers spent editing a particular file. This measure is obtainable from attaching Hackystat sensors to the developers' Integrated Developer Environment (i.e., Emacs, JBuilder, Eclipse) |
| Last Active Time | The day of the last active time. |
| # of Developer Active Time Contributions | The number of unique developers who have contributed to the total active time. |
| Commits | The total number of commits to a particular file. This measure is obtainable from attaching Hackystat sensors to a Concurrent Version Control server (i.e., CVS). |
| Last Commit | The day of the last commit. |
| # of Developer Commit Contributions | The number of unique developers who have contributed to the total commits. |
| Review | The number of code reviews (inspections) that were conducted. This measure is obtainable from attaching Hackystat sensors to the Eclipse Jupiter Review plugin. |
| Last Review | The day of the last review or inspection. |
| Defects | The number of defects that were reported and stored in the Defect Tracking tool. This measure is obtainable from attaching Hackystat sensors to the Jira tool. |
| Last Defect | The day of the last defect. |
| Enhancements | The number of enhancements that are requested. This measure is obtainable from attaching Hackystat sensors to the Jira tool. |
| Last Enhancement | The day of the last enhancement. |
| File Metrics | The last known metrics of non-test code; lines of code, number of methods, and number of classes. This measure is obtainable from attaching Hackystat sensors to the LOCC tool. |
| Test File Metrics | The lines of test code, number of test methods, and the number of test classes. This measure is obtainable from the Hackystat sensors to the LOCC tool. |
| Dependency | The inbound and outbound references that indicate dependencies within the system. This measure is obtainable from attaching Hackystat sensors to the DependencyFinder tool. |
| Unit Tests | The number of unit tests that are executed during the last build of the system. This measure is obtainable from attaching Hackystat sensors to the JUnit tool. |
| Test Failures | The total number of test failures. This measure is obtainable from attaching Hackystat sensors to the JUnit tool. |
| Coverage | The last know coverage percentage of the system. This measure represents the number of methods that are executed during a test invocation over the number of total methods. This measure is obtainable from attaching Hackystat sensors to the JBlanket tool.. |

Table 4.2. The Weighting System used in hackyPRI

| **Measure** | **Weighting** | **Discussion** |
|---|---|---|
| Expert | johnson=3<br>anyone else=1 | Dr. Johnson is an active Hackystat developer. He is the most experienced programmer in CSDL. In addition, a lot of his development are technical over passes of the code to ensure that the code is of high quality. Therefore, code that he develops is weighted higher than others. |
| Active Time | Not Weighted | |
| Last Active Time | Not Weighted | |
| # of Developer Active Time Contributions | 3+ developers=2<br>2 developers=1<br>1 developer=0<br>none=0 | If code has been developed solely by one person, the code is more likely to contain defects. As more developers work on the code, the less likely defects will occur. |
| Commits | Not Weighted | |
| Last Commit | Not Weighted | |
| # of Developer Commit Contributions | 3+ developers=2<br>2 developers=1<br>1 developer=0 | Commit data is another way to determine if developers are working on a particular piece of code. If code has been developed solely by one person, the code is more likely to contain defects. As more developers work on the code, the less likely defects will occur. |
| Review | 2+ reviews=2<br>1 review=1<br>none=0 | More reviews (inspections) that are conducted equals higher quality code. |
| Last Review | today-last>30=2<br>today-last<31=1<br>none=0 | Code that has been reviewed recently tends to be higher quality code. |
| Defects | Not Weighted | In development |
| Last Defect | Not Weighted | In development |
| File Metrics | Not Weighted | |
| Test File Metrics | Not Weighted | |
| Dependency | 1=inbound>outbound | Inbound references represents the number of references that use a specific class. Outbound represents the number of references that the class uses. The more inbound references the more likely changes in a class will impact other classes. |
| Unit Test | >0=1 | Each day a set of unit tests are executed against the system. If there is at least one or more executions then we can be fairly certain that some portion of the system was tested. However, this does not represent the effectiveness and thoroughness of the tests. Effectiveness and thoroughness can be measured with a combination of Test Failure, Coverage, and Defects. |
| Test Failures | | |
| Coverage | 100%=2<br>99-90+%=1<br>89-0=%=0 | Higher coverage percentage, every thing else being equal, translates to higher quality code. |

### 4.1.3 Step 2: Selecting a Document for Inspection Based on the PRI Ranking

Using the PRI Hackystat analysis, an organization should select a document at the bottom of the PRI ranking table for inspection. The higher the document is in the table, the less it is in need of inspection.

In my initial studies, I have found that simply picking the highest priority document, or the document that is at the very bottom of the chart, will probably not be the "best" document to inspect. In most cases, I have found that the PRI ranking aids the selection of a document, but it does not select the document for you. In other words, it is more useful to consider a few documents from the bottom portion of the ranking and take an educated guess as to which document needs inspection more.

### 4.1.4 Step 3: Conducting an Inspection of the Selected Document

Once a document is selected it can be inspected. One interesting side effect of the PRI ranking is that specific statistics and measures can be presented during the inspection process. For example, if a document is selected because it has low coverage, then the inspection can focus on why the coverage is low.

The Hackystat PRI Extension or the PRI process does not support the actual inspection of the document. Instead, an organization should consult traditional inspection processes (i.e., Software Inspection, Fagan Inspection, In-Process Inspection, etc). In other words, the PRI process is an outer layer that wraps around an already established inspection process.

### 4.1.5 Step 4: Adjustment of the Measure Selection and Calibration

If a document is shown to be incorrectly ranked, then an adjustment of the PRI weighting function is necessary. In hackyPRI this can be accomplished by adding more Hackystat measures to hackyPRI or recalibrating the numerical weights associated with the measures. More specifically, a recalibration includes editing the Java source code in the hackyPRI system. Conceptually, recalibration would be as easy as editing the information presented in Table 4.2.

Hackystat - Workspace Quality Analysis

| Workspaces (212): | Quality | Expert | Active Time | Last Active Time | # of member active time contributions | Commits | Last Commit | # of member commit contributions | Review | Last Review | File Metric | Test File Metric | Dependency | Unit Test | Coverage |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| hackyKernel\src\org\hackystat\kernel\sensor\usermap\ | 9 | johnson@hawaii.edu (activeTime=14.83, commits=1) | 22.75 h | 18-Nov-2004 | 5 | 54 | 24-Nov-2004 | 4 | 62 | 08-Nov-2004 | LOC=294, Classes=6, Methods=22 | Classes=2, Methods=2 | Inbound:6, outbound: 10 | 2 | 40.9% |
| hackyReview\src\org\hackystat\app\review\activity\dailyanalysis\ | 9 | johnson@hawaii.edu (activeTime=1.25, commits=6) | 1.42 h | 11-Oct-2004 | 3 | 14 | 11-Oct-2004 | 3 | 0 | | LOC=368, Classes=2, Methods=18 | Classes=2, Methods=2 | Inbound:6, outbound: 25 | 2 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\coverage\sdt\ | 9 | johnson@hawaii.edu (activeTime=0.17, commits=3) | 0.33 h | 01-Nov-2004 | 3 | 8 | 04-Jun-2004 | 2 | 0 | | LOC=159, Classes=2, Methods=10 | Classes=1, Methods=2 | Inbound:4, outbound: 10 | 2 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\unittest\sdt\ | 9 | johnson@hawaii.edu (activeTime=0.25, commits=3) | 0.67 h | 28-Nov-2004 | 3 | 11 | 28-Nov-2004 | 3 | 0 | | LOC=162, Classes=2, Methods=12 | Classes=1, Methods=2 | Inbound:7, outbound: 10 | 2 | 100.0% |
| hackyKernel\src\org\hackystat\kernel\cache\ | 8 | johnson@hawaii.edu (activeTime=1.92, commits=9) | 2.33 h | 08-Oct-2004 | 3 | 12 | 11-Oct-2004 | 2 | 0 | | LOC=335, Classes=2, Methods=29 | Classes=1, Methods=3 | Inbound:11, outbound: 2 | 3 | 94.7% |
| hackyKernel\src\org\hackystat\kernel\user\ | 8 | johnson@hawaii.edu (activeTime=2.33, commits=24) | 2.75 h | 11-Nov-2004 | 3 | 27 | 11-Nov-2004 | 3 | 0 | | LOC=563, Classes=3, Methods=52 | Classes=2, Methods=3 | Inbound:271, outbound: 5 | 3 | 90.3% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\adoption\ | 8 | johnson@hawaii.edu (activeTime=1.83, commits=9) | 2.17 h | 26-Oct-2004 | 2 | 10 | 26-Oct-2004 | 2 | 0 | | LOC=120, Classes=1, Methods=3 | Classes=1, Methods=1 | Inbound:1, outbound: 12 | 1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\admin\analysis\serverstats\ | 8 | johnson@hawaii.edu (activeTime=0.58, commits=9) | 0.83 h | 21-Apr-2004 | 4 | 10 | 25-Jun-2004 | 2 | 0 | | LOC=240, Classes=1, Methods=18 | Classes=1, Methods=1 | Inbound:1, outbound: 19 | 1 | 93.8% |
| hackyStdExt\src\org\hackystat\stdext\build\dailyproject\ | 8 | johnson@hawaii.edu (activeTime=1.33, commits=5) | 2.58 h | 15-Oct-2004 | 2 | 7 | 15-Oct-2004 | 2 | 0 | | LOC=153, Classes=1, Methods=7 | Classes=1, Methods=1 | Inbound:2, outbound: 15 | 1 | 100.0% |
| hackyStdExt\src\org\hackystat\stdext\common\analysis\dailyprojectdetails\ | 8 | johnson@hawaii.edu (activeTime=1.17, commits=2) | 1.33 h | 22-Nov-2004 | 2 | 3 | 09-Nov-2004 | 2 | 0 | | LOC=75, Classes=1, Methods=1 | Classes=1, Methods=1 | Inbound:0, outbound: 0 | 1 | 100.0% |
| hackyCli\src\org\hackystat\app\cli\sdt\ | 7 | johnson@hawaii.edu (activeTime=0.42, commits=0) | 0.42 h | 31-Aug-2004 | 1 | 0 | | 0 | 0 | | LOC=110, Classes=2, Methods=7 | Classes=1, Methods=2 | Inbound:3, outbound: 10 | 2 | 100.0% |
| hackyCourse\src\org\hackystat\app\course\analysis\ | 7 | hongbing@hawaii.edu (activeTime=9.67, commits=31) | 25.33 h | 09-Nov-2004 | 5 | 76 | 09-Nov-2004 | 6 | 0 | | LOC=427, Classes=2, Methods=6 | Classes=1, Methods=8 | Inbound:0, outbound: 0 | 1 | 100.0% |
| hackyCourse\src\org\hackystat\app\course\preference\ | 7 | takuyay@hawaii.edu (activeTime=3.67, commits=11) | 9.58 h | 04-Jun-2004 | 5 | 61 | 25-Jun-2004 | 5 | 0 | | LOC=315, Classes=6, Methods=6 | Classes=2, Methods=6 | Inbound:0, outbound: 0 | 2 | 100.0% |
| hackyCourse\src\org\hackystat\app\course\selector\ | 7 | johnson@hawaii.edu (activeTime=0.75, commits=5) | 1.58 h | 04-May-2004 | 3 | 14 | 09-Mar-2004 | 4 | 0 | | LOC=74, Classes=2, Methods=8 | Classes=0, Methods=0 | Inbound:0, outbound: 0 | 0 | 100.0% |

file:///F|/csdl/techreports/04-14/figs/2004-12-01-all.html (1 of 14)12/7/2004 11:52:50 PM

Figure 4.1. The Workspace PRI analysis. Workspaces are listed with its respective PRI ranking and the measures.

# Chapter 5

# Evaluation Methodology

This chapter discusses the proposed evaluation methodology of this research. The main thesis of this proposed research is that Priority Ranked Inspection (PRI) can distinguish documents that are more in need of inspection (MINI) from those less in need of inspection (LINI).

One approach of implementing PRI is through Hackystat, thus I will create a Hackystat Extension called hackyPRI. This extension will provide a Hackystat analysis, which will distinguish what documents are MINI from documents that are LINI. This determination is based on a numerical weighting system of different process and product measures. Some measures include: reported defects, unit tests, test coverage, active time, and number of changes. Each measure will be assigned a numerical weight and will be individually calibrated. See Chapter 4: The Hackystat PRI Extension for more details.

It is important to note two limitations of this research. First, I am not defining a set of measures that represent the PRI weighting function for all software projects. Instead, by using hackyPRI I will be able to go through a methodology to best calibrate the measures to accurately reflect the determination for the project I am studying. Second, PRI is more beneficial for organizations that have limited inspection resources. PRI is of less use for organizations that have the necessary resources to thoroughly inspect every document, although this is yet to be studied.

## 5.1 Subjects Used in the Evaluation

In this evaluation, I will study the implementation and inspection process of the Hackystat System developed in the Collaborative Software Development Laboratory (CSDL), of the University of Hawaii at Manoa. Like most organizations, CSDL's inspection resources are limited and

therefore inspections are conducted, if at all, on a weekly basis regardless of the number of “ready” documents.

CSDL primarily inspects source code grouped by package; therefore, I will use the term ’packages’ when referring to CSDL’s use of PRI. I will use the term ’documents’ when referring to the general idea of inspections.

Although I am a member of CSDL and have been contributing to Hackystat, I will minimize any possible data contamination by doing two things. First, I will keep the PRI weighting function and ranking a secret both during and after conducting the inspection. Second, I will not participate in the inspections themselves.

The use of CSDL in my study indicates another limitation on this research. The most accurate and thorough evaluation of PRI should inspect *all* documents to evaluate if the PRI weighting function and ranking is correct. However, because I am using CSDL’s inspection resources, which are limited, this is not possible.

## 5.2 Evaluation of Thesis Claims

To evaluate this thesis, I will decompose it into three claims based upon the three intended benefits of PRI.

1. PRI can enhance the volunteer-based document selection process.
2. PRI can identify documents that need to be inspected that are not typically identified by volunteering.
3. MINI documents will generate more critical defects than LINI documents.

This section will detail the methodologies used to evaluate each of the three claims.

### 5.2.1 Claim 1: PRI Enhances the Volunteering Process

In the traditional inspection process, developers volunteer their documents for inspection. In most cases, developers select documents to volunteer to remove any critical defects before it is released. However, the current literature does not provide much guidance on which documents should be volunteered. One of the intended benefits of PRI is the enhancement of the volunteering process by providing suggestions on what should be inspected. PRI can do this in two ways. First,

it can minimize the number of documents that should be considered for inspection. Second, it provides a priority ranking of what documents would be most beneficial to inspect.

PRI can minimize the number of documents that should be considered for inspection. In Software Inspection [7], the number of possible inspection includes *all* the documents currently moving through the development cycle. (This technique tends to emphasize only current documents and not the highest priority documents for inspection. Claim 2 addresses this issue.) In PRI, the number of possible documents is minimized to MINI documents. This reduced number of possibilities seems to be advantageous for organizations that cannot inspect every document, because it will eliminate the need and more importantly it limits the possibility of inspecting LINI documents.

As an example of how PRI benefits an organization with limited resources consider the following fictitious scenario:

> *The FooBar organization has enough resources available to conduct inspections at least once a week. Because this organization produces more code than is possible to inspect, they use a round-robin approach by allowing a different developer to volunteer a piece of code to inspect. This developer must pick a small portion of the code he/she is currently working on and this decision is primarily based solely on his/her subjective opinions of the code.*

This method works well if the developer can be trusted to pick the right code to inspect. However, developers often do not know where every critical defect will appear. In other words, leaving this decision up to the subjective understanding of a developer maybe error prone. PRI provides an alternative solution to this limited resource problem. Instead of leaving the decision of what code to inspect entirely up to the developer, PRI can minimize the number of possibilities by providing a smaller area of selection. For this fictional organization, the developer can find the MINI documents and choose code from this smaller list.

PRI provides a priority ranking of what documents would be most beneficial to inspect. This advantage supports the volunteering process by allowing the developer to prioritize his/her selection of documents. The previous discussion showed how PRI can minimize the number of documents; and now that the number is reduced the developer still must select from this smaller number. To support this selection, PRI ranks the documents according to the calibration and numerical ranking. For example consider this scenario:

> *Developer John Doe is currently working on 10 different documents and he wants to volunteer one of them for inspection. He has a rough idea of what documents he thinks would be most beneficial for inspection but he isn't sure. He consults the PRI ranking and finds that 6 of his documents appear to be LINI. In addition, he is able*

*to use the rankings to select a MINI document that he believes would generate more critical defects.*

This scenario illustrates how PRI can enhance the volunteering process by first minimizing the number of documents that should be considered for inspection and then prioritizing them.

To evaluate this claim, I will ask the developers of Hackystat to provide a numerical ranking, based on their subjective feelings, of what packages they would volunteer for inspection. With these results I will be able to compare the developers' subjective rankings against the PRI ranking In addition, I will present each developer with the results of the hackyPRI analysis, which provides the PRI weighting fuction and ranking. Then I will ask the developers to re-rank the packages based on the new information. This evaluation will indicate whether PRI is really needed. The findings could indicate that developers can correctly distinguish, using their own subjective reasoning, what packages need to be inspected.

To conduct this evaluation, I will provide each developer with a list of Hackystat packages that they are currently working on. This will be determined by assessing the developers' active time and commits to a particular package. Given this listing I will ask each developer to provide a numerical ranking of each package.

The following steps will occur in this evaluation:

1. Obtain the rankings of packages from each individual developer. I will use the questionnaire presented in Appendix B to obtain these rankings.

2. Analyze the difference between the developers' ranking against the PRI MINI and LINI determination.

3. Conduct the following inspections:

   (a) Inspect 2 packages, where the developer and the PRI determinations agree, that are MINI.

   (b) Inspect 2 packages, where the developer and the PRI determinations agree, that are LINI.

   (c) Inspect 2 packages where the developer and the PRI determinations disagree. The developer provides a low ranking but the PRI claims that the package is MINI.

   (d) Inspect 2 packages where the developer and the PRI determinations disagree. The developer provides a high ranking but the PRI claims that the package is LINI.

4. Analyze the results of each inspection, which includes correlating the number of critical issues generated with both the developer ranking and the PRI determination. In addition, I may ask the developers for explanations of their rankings where applicable.

5. After each inspection I will adjust PRI calibration or add new product and process measures as necessary.

There are three possible results from this evaluation. First, I may find that developers automatically have a sense of what code is MINI and what code is LINI. This would indicate that PRI provides little added value. Second, developers have no idea what code needs to be inspected. The third possible result represents a middle ground between the two previous results, sometimes the developers are correct and sometimes they are wrong. The last two results will indicate that PRI provides some benefit.

In addition, this evaluation will provide more data to refine the calibration of the PRI weighting function. For example, if a developer rates a package very high, but PRI finds the package to be LINI and many critical issues are found, then this indicates that the PRI weighting function is flawed. Therefore, the PRI weighting function needs to be recalibrated to include this document. In addition to calibration, more process and product measures could be introduced. This event, although detrimental to the previous PRI weighting function, will provide more data for calibration and the addition of new measures will hopefully lead to a better and more accurate PRI weighting function.

### 5.2.2 Claim 2: PRI Identifies Documents that are Not Typically Identified by the Volunteering Process

Another intended benefit of PRI is it can find MIN documents that are not typically identified using a volunteer-based document selection process. If organizations with limited inspection resources blindly volunteer documents for inspection they could be missing some areas of the system that need inspection. A real example of this benefit is illustrated in the following scenario:

> *Not all Hackystat packages have experts. Instead there are some packages that I considered to be orphans. Orphaned-packages are usually packages that are considerably old code or code that has been written by developers who have left CSDL. In addition, these packages are usually never inspected and are considered to be in working order.*

This situation is quite dangerous, because as we all know a software system evolves and outdated packages may become error prone. Therefore, it is important to realize that old packages

can and should be MINI. Software Inspection [7] does not address this issue of outdated documents. The common adage of Software Inspection is to inspect documents as they move through the development cycle. This process tends to ignore documents that have already finished the development cycle. In addition, because organizations with limited resources can not inspect every document moving through the development cycle, it is very likely that some documents will finish the development cycle with major defects. Therefore, ensuring that "finished" documents are included as potential inspection candidates is very important.

To evaluate this claim, I will make several inspection recommendations of MINI packages and have not been investigated in the previous study. Again, developers cannot always identify areas of the system that they think is low quality and only using the volunteering method will likely miss some documents that need inspection. The following steps will occur in this evaluation:

1. Select a few packages that were not investigated in the previous study and has been classified as MINI and LINI.
2. Conduct inspections on the selected packages.
3. Analyze the results of each inspection, which includes correlating the number of critical defects generated with the PRI determination (either MINI or LINI).
4. After each inspection I will adjust PRI calibration or add new product and process measures as necessary.

There are two possible results of this evaluation. First, the packages that were selected were correctly categorized by PRI. This finding will support my claim. Second, the packages that were selected did not reflect the PRI weighting function.

### 5.2.3 Claim 3: More in need of inspection versus less in need of inspection

The last benefit of PRI is MINI documents will generate more critical defects than LINI documents. This claim is critically important for PRI's success. However, if a package is identified as LINI and yields many critical defects, then PRI weighting function is flawed. I will use this information to refine the PRI weighitng function. It is my hope that in the end of the study I will have been able to successfully calibrate the PRI weighting function for the Hackystat project.

During the evaluations of the previous two claims, CSDL will have conducted at least 12 inspections. In addition, I have and will collect information on past and future inspections on

Hackystat packages. In total, I believe I will have data on 20 inspections and information on the PRI weighting functions and rankings..

Currently, Hackystat and its extensions are comprised of 167 packages. As I previously stated, an accurate and thorough evaluation of PRI requires the inspection of all packages within the PRI rankings. However, because of CSDL's limited resources this is not possible. At best this will take 3 hours per inspection, totaling 501 hours of inspection. This is unrealistic. Therefore, my proposed evaluation will investigate a small percentage of the system, 20 of the 167 packages, in hopes that this cross-section will provide adequate and acceptable results.

To evaluate this claim, I will monitor the validity of the PRI weighting function, adjusting the calibration as necessary, throughout each inspection. To accomplish this, I will collect specific pieces of information when conducting inspections. The following is a specific list of the information collected:

- Inspection date
- Hackystat module, package, and inspection ID
- PRI determination (MINI or LINI)
- PRI measures and values
- Subjective discussion of the validity of the PRI weighting function before the inspection
- Number of issues generated and the categorization of these issues according to severity
- Retrospective discussion after the inspection was conducted to indicate possible areas of improvement.

This information will help me keep track of the progress of the inspections and the validity of the PRI weighting function. See Appendix C for a copy of the full log. As I previously stated, the calibration of the PRI weighting function is an ongoing and evolving process. This information will help keep track of that evolution. The end goal of this evaluation is to create a best practices recommendation of the types of process and product measures and their calibration that will provide the best PRI results for different projects.

## 5.3 Evaluation Timeline

The following timeline provides a timeline for the evaluation of this thesis:

Table 5.1. Evaluation Timeline

| **Timeline** | **Evaluation Activity** |
|---|---|
| February 9, 2005 | Pilot trial of the developer workspace rankings |
| February 16, 2005 | Request developer workspace rankings |
| February 16, 2005 | Process developer responses and create a plan of what will be inspected |
| February 23, 2005 | Start 4 weeks of inspection, inspecting 2 packages a week |
| March 30, 2005 | Hand pick 2 packages to inspect that was not volunteered |
| March April 6, 2005 | Finished analyzing the results. |

## 5.4 Initial Results of Evaluation

The use of PRI to provide the determination of MINI and LINI 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 few package that were "more in need of inspection" and the defects and issues identified have confirmed that the PRI ranking was correct. See the inspection log (C for a detailed description of the inspection results.

I will continue to discover new measures to add to the PRI weighting function, fine tune the numerical weights associated with the measures, and continue to recommend inspections.

# Chapter 6

# Future Directions

This proposed research contains many limitations. Most notably the evaluation of PRI and hackyPRI is constrained to only one specific software project. This fact raises many issues of adoption. For example, how hard would it be to implement PRI at another organization? How hard would it be to calibrate PRI for another set of product and process measures? This adoption issue can be addressed by future evaluations of PRI in other organization settings and other software projects. However, this issue will be left as a future direction, as I neither have the time or resources to conduct such a thorough evaluation. However, I believe the proposed evaluation taken in this thesis is necessary to prove or disprove that PRI is a worthy concept to try at other organizations. In addition, future work is needed to generalize the hackyPRI extension so that it it possible for other organizations and/or projects. Currently, hackyPRI is tailor-made for the CSDL organization and for the Hackystat project.

Priority Ranked Inspection was originally created for quality purposes that span other quality improvements techniques other than software inspections. Originally, I proposed a technique that could identify the lowest cost approach to increase quality of a particular piece of code. I envisioned a Hackystat extension that could identify the right “quality tool” that is needed to increase quality. For example, if the ranking showed that Unit Tests are a problem area, then the right “quality tool” could be increasing the number of Unit Tests for that particular piece of code. However, for this proposed research I have obviously decided to focus on one “quality tool”, namely software inspection. I choose inspections because the literature suggests that this process is the most effective way to increase quality. Another future direction for this research is to evaluate if Priority Ranked Inspection can also identify the right “quality tool” to use in specific situations.

# Chapter 7

# Timeline

There are three milestones that measure my progress in this research. Milestone 1: implementation of Hackystat Extension, January 2005. Milestone 2: completed evaluation, March 2005. Milestone 3: thesis submission and defense, April 2005.

Table 7.1. Proposal Timeline

| **Date** | **Evaluation Activity** |
|---|---|
| February 7, 2005 | Thesis Proposal submitted to Thesis Committee |
| February 9, 2005 | Pilot Trial Evaluation |
| February 16, 2005 | Evaluation - begin evaluation |
| March 30, 2005 | Write results chapter |
| April 6, 2005 | Write results chapter |
| May 2005 | Thesis Draft submitted to Thesis Committee |
| May 2005 | Thesis Defense |
| June 2005 | Final Thesis submitted to Graduate Division |
| August 2005 | Summer Graduation |

# Appendix A

# Consent Form

# Consent Form

I understand that I am voluntarily participating in a University of Hawaii research project with the Collaborative Software Development Laboratory (CSDL). This project will evaluate the Priority Ranked Inspection process and the Hackystat Priority Ranked Inspection Extension. My participation includes completing pre- and post-questionnaires. I am one of approximately 8 subjects involved in this research.

I understand that I can withdraw from this project at any time and have the information provided in my questionnaires be removed in entirety from the research project. This decision will not affect me in any manner academically.

I understand that all gathered data will be kept confidential to the extent provided by law, and that any and all references to information about me or my data will be kept anonymous to the extent provided by law.

For any questions relating to this research, I may contact:

Collaborative Software Development Laboratory (CSDL)
1680 East-West Road, POST 307B
Honolulu, HI. 96822

| | | |
|---|---|---|
| Aaron A. Kagawa | kagawaa@hawaii.edu | 956-6920 |
| Philip Johnson | johnson@hawaii.edu | 956-3489 |

For and questions about my participants rights, I may contact:

| | |
|---|---|
| Committee on Human Studies | 956-5007 |

The researchers foresee no risks to participating in this project.

---

Figure A.1. Consent Form

# Appendix B

# Questionnaires

# Questionnaire for the Priority Ranked Inspection Process

Thank you for your participation. As a reminder, your participation in this research is voluntary. All references to data gathered will be made anonymously.

Please provide a numerical ranking of the packages provided that represents what packages you believe should be inspected. For example, the following table shows that package com.example.bar is the highest priority package that should be inspected. And com.example.baz has the lowest priority ranking.

| Ranking | Package |
|---|---|
| 2 | com.example.foo |
| 1 | com.example.bar |
| 3 | com.example.baz |

Enter your own subjective rankings in this table. Please use values from 1 through 13 and do not use any number twice.

| Ranking | Package |
|---|---|
| | org.hackystat.app.telemetry.analysis |
| | org.hackystat.app.telemetry.analysis.selector |
| | org.hackystat.app.telemetry.config |
| | org.hackystat.app.telemetry.config.core |
| | org.hackystat.app.telemetry.processor.evaluator |
| | org.hackystat.app.telemetry.processor.parser |
| | org.hackystat.app.telemetry.processor.parser.impl |
| | org.hackystat.app.telemetry.processor.query |
| | org.hackystat.app.telemetry.processor.query.expression |
| | org.hackystat.app.telemetry.processor.reducer |
| | org.hackystat.app.telemetry.processor.reducer.impl |
| | org.hackystat.app.telemetry.processor.reducer.util |
| | org.hackystat.app.telemetry.processor.stream |

When finished, please wait for further instructions.



Figure B.1. Questionnaire - Part 1

| Workspaces (211): | Quality | Expert | Active Time | Last Active Time | # of member active time contributions | Commits | Last Commit | # of member commit contributions | Review | Last Review | File Metric | Test File Metric | Dependency | Unit Test | Coverage |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| hackyTelemetry\src\org\hackystat\app\telemetry\analysis\ | 4 | qzhang@hawaii.edu (activeTime=28.42,commits=119) | 28.58 h | 15-Nov-2004 | 2 | 120 | 15-Nov-2004 | 2 | 0 | | LOC=857, Classes=5, Methods=15 | Classes=4, Methods=7 | inbound:5, outbound: 66 | 7 | 86.7% |
| hackyTelemetry\src\org\hackystat\app\telemetry\analysis\selector\ | 4 | qzhang@hawaii.edu (activeTime=6.67,commits=42) | 6.67 h | 15-Nov-2004 | 1 | 42 | 15-Nov-2004 | 1 | 0 | | LOC=317, Classes=9, Methods=19 | Classes=1, Methods=1 | inbound:9, outbound: 18 | 1 | 92.0% |
| hackyTelemetry\src\org\hackystat\app\telemetry\config\ | 4 | qzhang@hawaii.edu (activeTime=24.33,commits=98) | 24.42 h | 01-Nov-2004 | 2 | 100 | 02-Nov-2004 | 2 | 0 | | LOC=440, Classes=6, Methods=30 | Classes=1, Methods=1 | inbound:13, outbound: 15 | 1 | 43.8% |
| hackyTelemetry\src\org\hackystat\app\telemetry\config\core\ | 4 | qzhang@hawaii.edu (activeTime=7.00,commits=22) | 7.00 h | 02-Nov-2004 | 1 | 22 | 02-Nov-2004 | 1 | 0 | | LOC=1051, Classes=8, Methods=63 | Classes=6, Methods=12 | inbound:24, outbound: 20 | 10 | 90.0% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\evaluator\ | 5 | qzhang@hawaii.edu (activeTime=8.75,commits=28) | 9.17 h | 31-Oct-2004 | 3 | 30 | 01-Nov-2004 | 3 | 0 | | LOC=687, Classes=3, Methods=11 | Classes=3, Methods=12 | inbound:18, outbound: 33 | 8 | 89.7% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\parser\ | 3 | qzhang@hawaii.edu (activeTime=6.92,commits=37) | 6.92 h | 02-Nov-2004 | 1 | 38 | 02-Nov-2004 | 2 | 0 | | LOC=369, Classes=2, Methods=10 | Classes=3, Methods=13 | inbound:12, outbound: 23 | 13 | 88.0% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\parser\impl\ | 1 | qzhang@hawaii.edu (activeTime=0.08,commits=32) | 0.08 h | 30-May-2004 | 1 | 33 | 01-Nov-2004 | 2 | 0 | | LOC=2641, Classes=7, Methods=131 | Classes=0, Methods=0 | inbound:7, outbound: 25 | 0 | 63.2% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\query\ | 4 | qzhang@hawaii.edu (activeTime=6.17,commits=55) | 6.25 h | 31-Oct-2004 | 2 | 56 | 02-Nov-2004 | 2 | 0 | | LOC=389, Classes=5, Methods=25 | Classes=3, Methods=7 | inbound:21, outbound: 12 | 7 | 88.2% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\query\expression\ | 4 | qzhang@hawaii.edu (activeTime=0.92,commits=51) | 0.92 h | 05-Jul-2004 | 1 | 52 | 07-Oct-2004 | 2 | 0 | | LOC=391, Classes=15, Methods=50 | Classes=4, Methods=7 | inbound:39, outbound: 15 | 6 | 94.1% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\reducer\ | 3 | qzhang@hawaii.edu (activeTime=1.75,commits=14) | 1.75 h | 01-Jul-2004 | 1 | 16 | 31-Aug-2004 | 3 | 0 | | LOC=183, Classes=4, Methods=18 | Classes=1, Methods=1 | inbound:32, outbound: 8 | 1 | 87.5% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\reducer\impl\ | 7 | qzhang@hawaii.edu (activeTime=18.08,commits=99) | 18.50 h | 15-Nov-2004 | 3 | 116 | 15-Nov-2004 | 5 | 0 | | LOC=1705, Classes=8, Methods=21 | Classes=8, Methods=39 | inbound:9, outbound: 36 | 25 | 100.0% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\reducer\util\ | 3 | qzhang@hawaii.edu (activeTime=2.75,commits=19) | 2.75 h | 27-Jul-2004 | 1 | 21 | 28-Jul-2004 | 2 | 0 | | LOC=852, Classes=5, Methods=29 | Classes=4, Methods=11 | inbound:21, outbound: 24 | 9 | 75.0% |
| hackyTelemetry\src\org\hackystat\app\telemetry\processor\stream\ | 5 | qzhang@hawaii.edu (activeTime=4.08,commits=18) | 4.08 h | 14-Oct-2004 | 1 | 18 | 18-Jun-2004 | 1 | 0 | | LOC=250, Classes=4, Methods=19 | Classes=3, Methods=5 | inbound:48, outbound: 12 | 4 | 100.0% |



Figure B.2. Questionnaire - Part 2.1

# Questionnaire for the Priority Ranked Inspection Process

Please take a few minutes to review the PRI determination table just handed to you. Listed in this table are the same packages with the rankings that you just provided. If there are any rankings that you would like to adjust based on the PRI determination, please do so now.

| Old Ranking | New Ranking | Package |
|---|---|---|
| | | org.hackystat.app.telemetry.analysis |
| | | org.hackystat.app.telemetry.analysis.selector |
| | | org.hackystat.app.telemetry.config |
| | | org.hackystat.app.telemetry.config.core |
| | | org.hackystat.app.telemetry.processor.evaluator |
| | | org.hackystat.app.telemetry.processor.parser |
| | | org.hackystat.app.telemetry.processor.parser.impl |
| | | org.hackystat.app.telemetry.processor.query |
| | | Org.hackystat.app.telemetry.processor.query.expression |
| | | org.hackystat.app.telemetry.processor.reducer |
| | | org.hackystat.app.telemetry.processor.reducer.impl |
| | | org.hackystat.app.telemetry.processor.reducer.util |
| | | org.hackystat.app.telemetry.processor.stream |

1. Using the PRI determination helps me rank packages for inspection?

(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

2. If I have to volunteer a package for future inspections, I will consult the PRI determination.

(1) Strongly Disagree (2) Disagree (3) Neutral (4) Agree (5) Strongly Agree

3. Please briefly describe two (if any) changes in rankings that occurred after consulting the PRI determination. Include, what helped you change your mind and why.



Figure B.3. Questionnaire - Part 2.2

# Appendix C

# Inpsection Log and Results

This appendix provides a journal of my processes throughout the evaluation of Priority Ranked Inspection (PRI). I include several details about each inspection that is conducted, which include: the PRI determination, some of the measures that PRI used to create the determination, my subjective opinions about the package to be inspected, the results in terms of the number of issues found and their severity, and my subjective opinions on the PRI determination given the results.

Using this journal, I hope to construct a general guidelines for the use of and calibration of PRI for other organizations and projects.

Hackystat Documentation

# Hackystat Quality Engineering Log

Aaron Kagawa
Collaborative Software Development Laboratory
University of Hawaii

$Id: README.html,v 1.1.1.1 2004/09/09 02:39:46 kagawaa Exp $

## Overview

This document is an engineering log, which provides detailed information about the inspections recommended using hackyQuality analyses. For more details, please see my thesis proposal.

## Engineering Log

This table represents the decisions that we made to recommend an inspection and the results of that inspection.

| Inspection Number: 1 | |
|---|---|
| Date: | 9.21.04 |
| hackystat-dev-l message: | [HACKYSTAT-DEV-L:82] Review Request 9/21/04 |
| Module and Review Id: | hackyStdExt - JavaMap |
| Package: | org.hackystat.stdext.workspace.mava.javamap |
| Stats: | Quality Analysis Info:<br>* active time: 0.5h<br>* last active time: July 27, 2004<br>* number of contributers (active time): 2 (hongbing and johnson)<br>* expert: hongbing (0.33h)<br>* review: 0<br>* last review: N/A<br>* file metric: loc=196, classes=1, methods=12<br>* test filemetric: classes=0, methods=0<br>* unit test: 0 |



Figure C.1. Inspection Log and Results - Part 1

| | |
|---|---|
| | * coverage: 90%<br><br>Other Info<br>* Code initially created on May 7, 2003 - fairly old code<br>* 7 other classes has least one reference to JavaClassWorkspaceMapManager |
| Quality Level: | Low |
| Discussion: | This package was not ranked with the lowest possible quality level. Yet, this package seemed like a very good package to pick because of the zero unit tests and 90% coverage. This value indicated that although we have no unit tests a lot of code must use this package for some sort of basic processing. This is evident by the 7 references to the JavaClassWorkspaceMapManager. The references metric would be a great metric to have. Currently, our static analysis tool LOCC does not collect this sort of information (and it is not known at this time whether this would be hard to do or not). In addition, the package has had a few developers contributing to it, therefore one would think that is a problem. Another issue is that the code is fairly old code, and because of the substantial changes to the project caches one would think that there should be some changes to the code.<br><br>The another critical factor here is that the code has never been reviewed. Yet, it is so vital to the way hackystat works. One would also think that because it is an underlying data structure that is used by many analyses that this makes it a good canidate for inspection.<br><br>The most critical factor for its selection for inspection was my own subjective feeling about the code. I can't seem to put into words why i thought this was a good package for inspection but, something made me feel that way. Perhaps it was because I knew that this code provided critical functionality to the hackystat project yet for some reason it was so low my quality determination. |
| Number of Issues: | 37 review issues:<br>- critical: 3<br>- major: 7 |



Figure C.2. Inspection Log and Results - Part 2

| | |
|---|---|
| | - normal: 12<br>- minor: 7<br>- trivial: 5<br>- unset: 3 |
| Retrospective: | This package was an excellent selection for inspection. Most of the issues that were generated where of upper severity (normal and above). The issues that were examined in the team phase proved that this was a good package to inspect. However, we didn't find any mission critical defects in the code that would cause system failures. However, most of the issues that were generated where valid issues that are severe enought to warrant change. |

| Inspection Number: 2 | |
|---|---|
| Date: | 9.29.04 |
| hackystat-dev-l message: | [HACKYSTAT-DEV-L:115] Review Request 9/29/04 |
| Module and Review Id: | hackyJira - IssueSdt |
| Package: | org.hackystat.stdext.issue.sdt |
| Stats: | * Burt is the primary developer<br>* Zero unit tests, 26% coverage<br>* Has never been reviewed |
| Quality Level: | Low |
| Discussion: | This package was created by a brand new Hackystat developer. In my subjective feeling, was that new developers will have more defects than more experienced developers. Hence, there should be a ranking of developers based on some quanitifiable measurement of experience. Maybe just a configurable ranking would suffice.<br><br>In addition to the developer issue, the package contained no unit tests which is also a major quality problem. It seems that any time a package has zero unit tests, the package should immediately inspected. Actually, there are a couple of alternatives; |



Figure C.3. Inspection Log and Results - Part 3

| | |
|---|---|
| | identify that a package should implement unit tests (1) if unit tests are implemented, then see how that affects its quailty ranking. If the package is still low in quailty then we should inspect it. Or identify that a package should implement unit tests, (2) if the developer responds that it is either too hard or impossible to do so, then most definitely we should conduct inspections.<br><br>However, it seems that the bottom line for this specific selection was the inexperience of the developer. Hopefully, this inspection will provide an educational value for the developer. |
| Number of Issues: | 19 review issues:<br>- critical: 1<br>- major: 2<br>- normal: 8<br>- minor: 5<br>- trivial: 1<br>- unset: 2 |
| Retrospective: | Although, this inspection was cancelled and only two members actually finished the individual phase many issues were generated. Many if not all of the issues were valid and required changes to the code at some level. I would claim that this package was definitely proven to be low quality and deserved the most in need of inspection destinction. I believe the primary indicator was the unexperience of the developer, in this case Burt.<br><br>Also, note that we didn't find any mission critical issues that would cause the code to fail under normal circumstances. |

| Inspection Number: 3 | |
|---|---|
| Date: | 10.6.04 |
| hackystat-dev-l message: | [HACKYSTAT-DEV-L:142] Review Request 10/06/04 |
| Module and Review Id: | hackyKernel - UserMap, hackyJira - IssueSensor |



Figure C.4. Inspection Log and Results - Part 4

| | |
|---|---|
| Package: | org.hackystat.kernel.sensor.usermap, cm.atlassian.jira.event.listener in the hackyJira/src_listener directory |
| Stats: | - It is extremely fresh code and is already on the public server.<br>- There were some junit failures in the latest build<br>- And we haven't had issues sent to the server yet |
| Quality Level: | Low |
| Discussion: | This inspection was not triggered by using the current quality levels. Rather, it was selected because the JIRA sensor was not working correctly.<br><br>However, there were some indicators on why this was low quality code. |
| Number of Issues: | Issues Sensor - 50 review issues:<br>- critical: 5<br>- major: 9<br>- normal: 27<br>- minor: 5<br>- trivial: 0<br>- unset: 2<br><br>UserMap - 20 review issues:<br>- critical: 0<br>- major: 5<br>- normal: 8<br>- minor: 5<br>- trivial: 1<br>- unset: 1 |
| Retrospective: | This inspection generated numerous issues. This time some of them were mission critical, meaning that the issue could cause system failure. This is obvious because the JIRA sensor wasn't working. Many issues were generated about coding style and formatting. Obvisously, the number of generated issues we consider formatting and coding style as very important. Again, unexperienced developers played a major factor. |

| |
|---|
| Inspection Number: 4 |



Figure C.5. Inspection Log and Results - Part 5

| | |
|---|---|
| Date: | 10.13.04 |
| hackystat-dev-l message: | [HACKYSTAT-DEV-L:179] Fwd: Review Request 10/06/04 (typo: should read 10/13/2004) |
| Module and Review Id: | hackyKernel - UserMap2, hackyJira - IssueSensor2 |
| Package: | org.hackystat.kernel.sensor.usermap, org.hackystat.stdext.sensor.jira |
| Stats: | - It is extremely fresh code and is already on the public server.<br>- And we haven't had issues sent to the server yet |
| Quality Level: | Low |
| Discussion: | Again, this inspection was not triggered on quality levels. Instead, it was triggered because there were so many issues generated in the last inspection session that we decided to put the same code up for re-inspection. |
| Number of Issues: | IssuesSensor2 - 55 review issues:<br>- critical: 1<br>- major: 11<br>- normal: 30<br>- minor: 8<br>- trivial: 2<br>- unset: 3<br><br>UserMap2 - 28 review issues:<br>- critical: 0<br>- major: 4<br>- normal: 11<br>- minor: 11<br>- trivial: 2<br>- unset: 0 |
| Retrospective: | Again, there were many issues generated. And again there were some mission critical issues generated that would cause system failure under normal conditions. The documentation that was provided lead to a lot of confusion and many issues were generated because of that. Again, unexperienced developers were the leading factor. In addition, I don't believe that the coverage or unit test attributes would have provided effective information in the level of quality. |



Figure C.6. Inspection Log and Results - Part 6

| Inspection Number: 5 | |
|---|---|
| Date: | 10.20.04 |
| hackystat-dev-l message: | [HACKYSTAT-DEV-L:205] Review Request 10/20/04 |
| Module and Review Id: | hackyKernel - UserMap3, hackyJira - IssueSensor3 |
| Package: | org.hackystat.kernel.sensor.usermap,<br>org.hackystat.stdext.sensor.jira |
| Stats: | |
| Quality Level: | Low |
| Discussion: | Again, this inspection was not triggered on quality levels. Instead, it was triggered because there were so many issues generated in the last inspection session that we decided to put the same code up for re-inspection. |
| Number of Issues: | IssuesSensor3 - 22 review issues:<br>- critical: 0<br>- major: 5<br>- normal: 12<br>- minor: 4<br>- trivial: 0<br>- unset: 1<br><br>UserMap3 - 20 review issues:<br>- critical: 1<br>- major: 1<br>- normal: 11<br>- minor: 5<br>- trivial: 2<br>- unset: 0 |
| Retrospective: | No mission critical issues were identified. However, the code was substantially improved since the last inspection such that we were able to identify some higher-level design issues that weren't apparent before. Again, unexperienced developers were the leading indicator. |



Figure C.7. Inspection Log and Results - Part 7

| Inspection Number: 6 | |
|---|---|
| Date: | |
| hackystat-dev-l message: | |
| Module and Review Id: | hackyTelemetry |
| Package: | |
| Stats: | |
| Quality Level: | Unsure |
| Discussion: | This inspection was proposed by Cedric with the knowledge that his code has never been inspected and the subjective claim that it is complicated code. This brings up an important point, the quality analysis that I provide is no better than ones own subjective feeling about the code. In other words, a inspection process should never give up the right to inspect code that is being volunteered and hackyQuality will never relace that. However, that does not mean that hackyQuality is inferior to the subjective feelings (volunteering), I would claim that the quality analysis will identify code that should be inspected that no one has ever thought about or have forgotten.<br><br>It seems that the quality analysis is just a tool that inspection managers can use to identify potential areas of interest. We can't inspect everything; volunteering code will work to a certain extent, but we are likely to miss some code that isn't voluteered. I think hackyQuality will help find the code that seeps through the cracks. |
| Number of Issues: | |
| Retrospective: | |

| Inspection Number: 7 |
|---|



Figure C.8. Inspection Log and Results - Part 8

| | |
|---|---|
| Date: | |
| hackystat-dev-l message: | |
| Module and Review Id: | hackyStdExt |
| Package: | org.hackystat.stdext.unittest.dailyproject, org.hackystat.stdext.coverage.dailyproject |
| Stats: | |
| Quality Level: | Unsure |
| Discussion: | These two packages should be similar in nature in that they process Sensor Data that contain only a Java class name and the associated data. In other words, the Sensor Data does not contain a workspace. Therefore, these two daily project data implementations must use the JavaClassWorkspaceMapManager to associate a classname with a workspace. This is where the similarities end. The two daily project data implementations are quite different and make design decisions that are probably functionally correct, but are hard to understand when looking at both of them. I believe that the two daily project data implementations should be similar; therefore it will be easier to understand. |
| Number of Issues: | |
| Retrospective: | |



Figure C.9. Inspection Log and Results - Part 9

# Bibliography


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

[2] Marilyn 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] Martin Bush and Norman E. Fenton. Software measurement: A conceptual framework. *Journal of Systems and Software*, 12:223–231, 1990.

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

[5] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[6] Tom Gilb. Software inspections are not for quality, but for engineering economics. 1999.

[7] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

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

[9] Philip Johnson. Hackystat framework. Technical report, Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawaii, January 2005.

[10] Philip M. Johnson and Danu Tjahjono. Assessing software review meetings: A controlled experimental study using CSRS. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 118–127, Boston, MA., May 1997.

[11] Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Journal of Empirical Software Engineering*, 4(1), January 1998.

[12] Lawrence G. Votta Jr. Does every inspection need a meeting? In *Proceedings of the ACM SIGSOFT 1993 Symposium on Foundations of Software Engineering*, volume 18(5) of *ACM Software Engineering Notes*, pages 107–114, December 1993.

[13] Software Engineering Technical Committee of the IEEE Compter Society. IEEE standard glossary of software engineering terminology. *IEEE-STD-729-1983 (New York; IEEE)*, 1983.

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