

PRIORITY RANKED INSPECTION:
SUPPORTING EFFECTIVE INSPECTION IN RESOURCE LIMITED
ORGANIZATIONS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTERS

IN

INFORMATION AND COMPUTER SCIENCES

AUGUST 2005

By
Aaron A. Kagawa

Thesis Committee:

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

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Masters in Information and Computer Sciences.

THESIS COMMITTEE

Chairperson

©Copyright 2005

by

Aaron A. Kagawa

To my family.

Acknowledgments

I will always cherish the memories of my Graduate education. It has been a long and difficult journey, but at the same time it has also been a lot of fun and very educational. Any success and achievements that I have accomplished are a direct result of the support of many people. To these people, I say thank you from the bottom of my heart.

There is nothing more important to me than my family. I am eternally grateful for the strength, support, and above all, the love that all of you have given to me.

This Master's Thesis would have not been possible without the tutelage of Professor Philip Johnson. Thanks for your encouragement and guidance. There is no doubt that you have greatly influenced many of my skills in software engineering, Java programming, software design, decision making, and team leadership.

I would also like to send out many thanks to my fellow CSDL members. You have provided an excellent environment for all of us to achieve our educational goals with your brilliant minds and outstanding support. Most of all, thanks for all the fun times we've had together. One day we'll achieve our goal of being the only PGA tour members with Graduate degrees in Computer Science! Until then, I think we should stick to hacking Java.

I would also like to thank my committee members, Professor Philip Johnson, Professor Dan Suthers, and Professor Martha Crosby. Thank you for your guidance, criticisms, and most of all, praise.

Last of all, I would like to thank the whole Department of Information and Computer Sciences here at the University of Hawaii Manoa. My undergraduate and graduate studies have been educational, memorable and believe it or not, it has been fun.

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? Unfortunately, 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 [1] determines when documents are ready for inspection rather than if it is needed at all [2].

My research has investigated 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 [3]. If inspection resources are limited, then it will be more effective to identify and inspect the defect-prone areas.

To accomplish this research, I have created an inspection process called Priority Ranked Inspection (PRI). PRI uses software product and development process measures to distinguish documents that are “more in need of inspection” (MINI) from those “less in need of inspection” (LINI). Some of the product and process measures include: user-reported defects, unit test coverage, active time, and number of changes. I hypothesize that the inspection of MINI documents will generate more defects with a higher severity than inspecting LINI documents.

My research employed a very simple exploratory study, which includes inspecting MINI and LINI software code and checking to see if MINI code inspections generate more defects than LINI code inspections. The results of the study provide supporting evidence that MINI documents do contain more high-severity defects than LINI documents. In addition, there is some evidence that PRI can provide developers with more information to help determine what documents they should select for inspection.

Table of Contents

Acknowledgments	v
Abstract	vi
List of Tables	xii
List of Figures	xiv
1 Introduction	1
1.1 The Problem of Limited Resources for Software Inspections	1
1.2 The Priority Ranked Inspection Approach	4
1.2.1 Step 1a: Selection of Product and Process Measures	5
1.2.2 Step 1b: Calibration of Indicators	6
1.2.3 Step 1c: Declaring MINI and LINI documents	7
1.2.4 Step 2: Selecting a Document for Inspection Based on the PRI Ranking	8
1.2.5 Step 3: Conducting an Inspection of the Selected Document	8
1.2.6 Step 4: Adjustment of the Measure Selection and Calibration	9
1.3 Implementing PRI with Hackystat	9
1.4 Thesis Statement	12
1.4.1 Thesis Claim 1	12
1.4.2 Thesis Claim 2	12
1.4.3 Thesis Claim 3	14
1.5 Exploratory Study and Results	15
1.5.1 Thesis Claim 1	15
1.5.2 Thesis Claim 2	15
1.5.3 Thesis Claim 3	16
1.6 Structure of the Thesis	16
2 Related Work	17
2.1 Types of Inspection	18
2.1.1 Informal Inspection Processes	18
2.1.2 Formal Inspection Processes	19
2.1.3 Software Inspection and Priority Ranked Inspection	22
2.2 Alterations to the Inspection Process	25
2.2.1 Eliminating Steps in Inspection	25
2.2.2 Automated Software Inspection	26
2.3 Selection of Documents for Inspection	28
2.3.1 Code Smells	29
2.3.2 Crocodile	29

2.3.3	Risk Analysis	30
3	The Hackystat System	32
3.1	Overview of the Hackystat System	32
3.2	Hackystat’s Architecture	33
3.3	Hackystat Sensors	34
3.4	Some Screenshots of Hackystat	35
3.5	More Information about Hackystat	38
4	Implementing PRI with Hackystat	39
4.1	Limitations of Implementing PRI with Hackystat	41
4.2	PRI Ranking Function	41
4.2.1	PRI Measures	41
4.2.2	PRI Indicators	42
4.3	User Interface	44
4.3.1	PRI Analyses	45
4.3.2	PRI List Workspace Analysis	46
4.3.3	Project PRI Configuration Management	47
4.3.4	Create a Project PRI Configuration	48
4.3.5	Project PRI Configuration Management - After the Creation of a PRI Configuration	51
4.3.6	Project PRI Ranking Analysis	52
4.3.7	Project PRI Ranking Analysis Selectors	53
4.3.8	Project PRI Ranking Analysis - Detailed View	55
4.3.9	Project PRI Ranking Analysis - hackyKernel	56
4.3.10	Project PRI Module Ranking Analysis	57
4.4	The Four Steps of the Priority Ranked Inspection Process	59
4.4.1	Step 1a: Selection of Product and Process Measures	59
4.4.2	Step 1b: Calibration of PRI Indicators	62
4.4.3	Step 1c: Declaring MINI and LINI documents	64
4.4.4	Step 2: Selecting a Document for Inspection Based on the PRI Ranking	65
4.4.5	Step 3: Conducting an Inspection of the Selected Document	65
4.4.6	Step 4: Adjustment of the Measure Selection and Indicator Calibration	65
4.5	Hackystat Priority Ranked Inspection Extension	66
4.5.1	Design and Implementation	66
4.5.2	Design and Implementation Improvements	73
4.6	Future Implementation Enhancements	75
4.6.1	Threats to Data Validity	76
4.6.2	PRI Indicator Ranking	76
4.6.3	Other Levels of Ranking	76
4.6.4	Better Support for More Programming Languages	77
4.6.5	Link with Software Project Telemetry	77
4.6.6	Automatic Calibration Feedback Loop	77
4.7	Contributions to Hackystat	78
4.8	Using the Hackystat PRI Extension	79
5	Exploratory Study Procedure	80
5.1	Subjects Used in the Study	80

5.2	Study Limitations	83
5.3	Study of Thesis Claims	83
5.3.1	Part 1 - Pre-Selection Questionnaire	84
5.3.2	Part 2 - Package Selection	86
5.3.3	Part 3 - Request for Inspection	87
5.3.4	Part 4 - Inspection of Selected Package	87
5.3.5	Part 5 - Post-Inspection Questionnaire	87
5.3.6	Part 6 - Record Results of Inspection	87
5.4	Study Timeline	88
6	Results	90
6.1	Limitations Revisited	91
6.2	Inspections	91
6.3	Result Terminology	94
6.4	Thesis Claim 1	95
6.4.1	Inspection Results by Severity	95
6.4.2	Post-Inspection-Questionnaire Results	97
6.4.3	Inspection Results by Type	99
6.4.4	Inspection Results by Review Active Time	100
6.4.5	Inspection Results by Averages	102
6.5	Thesis Claim 2	103
6.5.1	Selection Trends	103
6.5.2	Validity of PRI's MINI and LINI ranking	107
6.5.3	Educational Value of Inspection	108
6.6	Thesis Claim 3	109
6.6.1	Selection Trends	110
7	Conclusions and Future Directions	111
7.1	Future Directions	111
7.1.1	More Evaluations	111
7.1.2	Implementation of the Hackystat PRI Extension	112
7.1.3	How to Determine MINI-Threshold	112
7.1.4	Comparison of PRI with Code Smells and Crocodile	112
7.1.5	Use of PRI in Other Quality Assurance Situations	113
7.2	Final Thoughts	113
A	Consent Form	114
B	Pre-Selection Questionnaire	116
C	Post-Inspection Questionnaire	121
D	Pre-Selection-Questionnaire Results	123
D.1	Question 1	124
D.2	Question 2	125
D.3	Question 3	126
D.4	Question 4	127
D.5	Question 5	128
D.6	Question 6	129
D.7	Question 7	130
D.7.1	hackyReview	130

D.7.2	hackyIssue	132
D.7.3	hackyCGQM	134
D.7.4	hackyZorro	137
D.7.5	hackyTelemetry	139
D.8	Question 8	141
D.9	Question 9	147
E	Inspection and Post-Inspection-Questionnaire Results	151
E.1	Inspection 8	151
E.2	Inspection 9	153
E.3	Inspection 11	155
E.4	Inspection 12	157
E.5	Inspection 13	159
E.6	Inspection 14	162
E.7	Inspection 15	164
E.8	Inspection 16	167
E.9	Inspection 17	170
	Bibliography	173

List of Tables

<u>Table</u>	<u>Page</u>
1.1 Step 1a - Example PRI ranking - After Measure Selection	6
1.2 Step 1b - Example PRI ranking - After Indicator Calibration	6
1.3 Step 1b - Example PRI ranking - After Indicator Weighting Calibration	7
1.4 Step 1c - Example PRI ranking - Declaring MINI and LINI	8
4.1 Java Packages in the hackyPRI system	67
4.2 WorkspacePriMeasure Interface	69
4.3 Description of the PRI Measures	70
4.4 The workspace package	73
4.5 ProjectWorkspaceRanking algorithm	74
5.1 Jupiter Properties and Values	82
5.2 Study Timeline	89
6.1 All Inspection - Results by Severity	92
6.2 All Inspection - Results by Type	93
6.3 All Inspections - Results by Type and Severity	93
D.1 Question 1 Responses	124
D.2 Question 2 Responses	125
D.3 Question 3 Responses	126
D.4 Question 4 Responses	127
D.5 Question 5 Responses	128
D.6 Question 6 Responses	129
D.7 hackyReview Developer Ranking	130
D.8 hackyIssue Developer Ranking	132
D.9 hackyCGQM Developer Ranking	134
D.10 hackyZorro Developer Ranking	137
D.11 hackyTelemetry Developer Ranking	139
D.12 Question 8 Responses - Participant 1	141
D.13 Question 8 Responses - Participant 2	142
D.14 Question 8 Responses - Participant 3	142
D.15 Question 8 Responses - Participant 4	143

D.16	Question 8 Responses - Participant 5	143
D.17	Question 8 Responses - Participant 6	144
D.18	Question 9 Responses - Participant 1	147
D.19	Question 9 Responses - Participant 2	148
D.20	Question 9 Responses - Participant 3	149
D.21	Question 9 Responses - Participant 4	149
D.22	Question 9 Responses - Participant 5	150
D.23	Question 9 Responses - Participant 6	150
E.1	Inspection 8 - Package Details	151
E.2	Inspection 8 - Results by Severity	152
E.3	Inspection 8 - Results by Type and Severity	152
E.4	Post Inspection 8 - Responses	152
E.5	Inspection 9 - Package Details	153
E.6	Inspection 9 - Results by Severity	153
E.7	Inspection 9 - Results by Type and Severity	153
E.8	Post Inspection 9 - Responses	154
E.9	Inspection 11 - Package Details	155
E.10	Inspection 11 - Results by Severity	155
E.11	Inspection 11 - Results by Type and Severity	155
E.12	Post Inspection 11 - Responses	156
E.13	Inspection 12 - Package Details	157
E.14	Inspection 12 - Results by Severity	157
E.15	Inspection 12 - Results by Type and Severity	157
E.16	Post Inspection 12 - Responses	158
E.17	Inspection 13 - Package Details	159
E.18	Inspection 13 - Results by Severity	159
E.19	Inspection 13 - Results by Type and Severity	159
E.20	Post Inspection 13 - Responses	160
E.21	Inspection 14 - Package Details	162
E.22	Inspection 14 - Results by Severity	162
E.23	Inspection 14 - Results by Type and Severity	162
E.24	Post Inspection 14 - Responses	163
E.25	Inspection 15 - Package Details	164
E.26	Inspection 15 - Results by Severity	164
E.27	Inspection 15 - Results by Type and Severity	164
E.28	Post Inspection 15 - Responses	165
E.29	Inspection 16 - Package Details	167
E.30	Inspection 16 - Results by Severity	167
E.31	Inspection 16 - Results by Type and Severity	167
E.32	Post Inspection 16 - Responses	168
E.33	Inspection 17 - Package Details	170
E.34	Inspection 17 - Results by Severity	170
E.35	Inspection 17 - Results by Type and Severity	170
E.36	Post Inspection 17 - Responses	171

List of Figures

<u>Figure</u>	<u>Page</u>
1.1 The PRI Ranking analysis	11
3.1 Hackystat Architecture	34
3.2 Hackystat Sensors and Sensor Data Types	35
3.3 The Hackystat Home Page	36
3.4 The Hackystat analysis webpage	36
3.5 The Hackystat preferences webpage	37
4.1 PRI Ranking analysis	40
4.2 PRI analyses	45
4.3 Execution of the List Workspace analysis	46
4.4 Project PRI Configuration Management preference page	47
4.5 Create Project PRI Configuration	48
4.6 Project PRI Configuration Management preference page	51
4.7 Execution of the Project PRI Ranking analysis	52
4.8 Project PRI Ranking analysis selectors	53
4.9 Execution of the Project PRI Ranking analysis	55
4.10 Execution of the Project PRI Ranking analysis	56
4.11 Execution of the Project PRI Module Ranking analysis	57
6.1 All Inspections - Results by Severity and Type	93
6.2 Inspection Results - Severity	95
6.3 Inspection Results - Average Severity	96
6.4 Inspection Results - Severity Percentage	97
6.5 Inspection Results - Average Severity Percentage	97
6.6 Post Inspection Questionnaire - Question 1	98
6.7 Post Inspection Questionnaire - Question 1 - Average Responses	98
6.8 Post Inspection Questionnaire - Question 3	99
6.9 Post Inspection Questionnaire - Question 3 - Average Responses	99
6.10 Inspection Results - Type	100
6.11 Inspection Results - Type	100
6.12 Inspection Results - Review Active Time	101
6.13 Inspection Results - Review Active Time	101

6.14	Inspection Results - Averages	102
6.15	Question 8 Part 1 Responses	105
6.16	Question 8 Part 2 Responses	106
6.17	Question 8 Comparison	107
6.18	Post Inspection Questionnaire - Question 2	109
A.1	Consent Form	115
B.1	Pre-Selection Questionnaire - Part 1	117
B.2	Pre-Selection Questionnaire - Part 2	118
B.3	Pre-Selection Questionnaire - Part 3	119
B.4	Pre-Selection Questionnaire - Part 4	120
C.1	Post-Inspection Questionnaire	122
D.1	Question 1 Responses	124
D.2	Question 2 Responses	125
D.3	Question 3 Responses	126
D.4	Question 4 Responses	127
D.5	Question 5 Responses	128
D.6	Question 6 Responses	129
D.7	hackyReview PRI Ranking - Inspection 8	131
D.8	hackyIssue PRI Ranking - Inspection 9	133
D.9	hackyCGQM PRI Ranking - Inspection 11	135
D.10	hackyCGQM PRI Ranking - Inspection 11	136
D.11	hackyZorro PRI Ranking - Inspection 12	138
D.12	hackyTelemetry PRI Ranking - Inspection 14	140
D.13	Question 8 Part 1 Responses	145
D.14	Question 8 Part 2 Responses	145
D.15	Question 8 Part 1 Responses	146
D.16	Question 8 Part 2 Responses	146
E.1	hackyIssue PRI Ranking - Inspection 13	161
E.2	hackyKernel PRI Ranking - Inspection 15	166
E.3	hackyStdExt PRI Ranking - Inspection 16	169
E.4	hackyStdExt PRI Ranking - Inspection 17	172

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...” [1]. 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 first test case is run” [4, 5].

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 [6], Software Inspection [1], In-Process Inspection [7], peer review [8], software reviews, code walkthroughs, inspections without meetings, and many more different twists on the original concept. Each of these techniques claims to be the best inspection method for certain circumstances. For example, some argue that the inspection meeting is a waste of time and resources [9, 10]. Others argue that the inspection meeting is critical for supporting social and educational aspects of inspection [11, 12, 13, 14].

My research is complementary to traditional inspection research. Instead of investigating how to conduct the inspection process, I investigate how to prioritize what documents need to be inspected. In this thesis, I describe how the selection of a document for inspection can create problems for organizations with limited inspection resources. To solve these problems, I have created a new inspection document selection technique called Priority Ranked Inspection.

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 [5]. In another success story, the Jet Propulsion Laboratory (JPL) adopted an inspection process to identify defects. After conducting 300 inspections they experienced a savings of 7.5 million dollars [15]. This statistic is very impressive. However, what is not emphasized is each inspection had an average cost of 28 hours. Using that average cost, the total cost for JPL's 300 inspections was 8,400 hours or roughly 4 years of work.

The study of JPL's inspection experience illustrates a fundamental problem with inspections: better results come from substantial investment [1]. Furthermore, some consider inspections to be a "gating point" in the development life cycle of a project. A point at which there is an "opportunity to control the transfer of the product from one development process stage to the next" [2, 16]. However, not all organizations have the time or the money to invest in full or complete inspections. In most cases, organizations have limited 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" [2]. The bottom line is that most organizations cannot inspect every document. As an example, consider the following scenario:

An organization has enough resources available to conduct two inspections a week. However, each week this organization creates and finishes 10 different documents. How do they select 2 documents to inspect from the possible 10? To be fair to all developers, they use a round-robin type approach by allowing a different developer to volunteer a document for inspection. This approach is fairly successful and at least they are conducting inspections. But, are they inspecting the right documents? Obviously, each week this organization is unavoidably letting 8 documents slip through the "inspection-gate" and could be releasing documents that have high-severity defects.

The traditional inspection process begins with the initiation phase, or sometimes called the planning stage, in which authors volunteer their documents for inspection [1]. In addition, an inspection leader checks the document against entry criteria to determine if the document is ready for inspection [7, 1]. Again this process works very well for organizations, like JPL, that have the resources to inspect every document after every significant change. However, this phase of inspection can be 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 [3].

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.

Unfortunately, the current literature [7, 17, 1] on inspections does not provide specific insights into the trade-offs between inspecting some documents and not inspecting others. However, there are many publications describing different ways to reduce the required inspection resources. For example, Tom Gilb and Dorothy Graham provide two recommendations to use when inspection resources are limited, sampling and emphasizing up-stream documents [1]. Lawrence Votta Jr. found that the inspection meeting, where the inspectors meet and discuss the validity of the issues found in the individual phase, is ineffective and can actually delay the inspection process days and even weeks [10, 18]. Therefore, he proposed that the inspection meeting be totally eliminated. An extreme process change is outsourcing the whole inspection process to a third-party company. Jasper Kamperman states that outsourced software inspections are cheaper, easier, and greatly beneficial [19]. These examples might do an acceptable job of decreasing the necessary inspection resources, but they still do not provide an answer about the selection of document for inspection.

Inspections are supposed to be the “silver bullet” for software quality assurance. Yet, its acceptance in the mainstream software development community has been slow. I’ll conclude this section with some quotes found in the inspection literature.

[Inspections] are extremely costly, all claims that they are cheaper than their alternatives notwithstanding. In other words, inspection is a very bad form of error removal-but all others are much worse. ... Most companies don’t do many inspections, and some do none at all. At best, the state of the practice is “we inspect our key components.” At worse, its “we don’t have the time to do inspections at all.” And that’s too bad, because the value of inspections is probably the topic on which computing research agrees more consistently than any other [18].

[Inspection] is a double pain. First, the documents to be inspected must be produced, and we know that documentation itself is a pain. Second, inspection is one of these unpopular activities that are the first to be scrubbed when the deadlines are looming. ... Every shred of evidence indicates that formal technical reviews (for example, inspections) result in fewer production bugs, lower maintenance costs, and higher software success rates. Yet we’re unwilling to plan the effort required to recognize bugs in their early states, even though bugs found in the field cost as much as 200 times more to correct [20].

Inspection is a systematic and disciplined process that is guided by well-defined rules. These strict requirements often backfire, resulting in code inspections that are not performed well or sometimes even not performed at all [21].

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 distinguishing documents that are “more in need of inspection” (MINI) versus documents that are “less in need of inspection” (LINI). In addition, PRI ranks each document according to this determination in hopes of prioritizing the documents that need to be inspected. The converse is also true: PRI identifies 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 processes, 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 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 high-severity defects will finish the development process without being inspected. Therefore, PRI is also intended to identify documents for inspection regardless of whether a document is currently under development 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 ranking function, which distinguishes MINI documents from LINI documents. The ranking function design includes three steps:
 - (a) Selection of product and process measures to use in the PRI ranking function.
 - (b) The calibration of indicators, which evaluates the values of the measures to generate a ranking for the document.
 - (c) The creation of a MINI-threshold, which declares all documents above the threshold as LINI and all below as MINI.
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 of indicators based on the results of ongoing inspections.

1.2.1 Step 1a: Selection of Product and Process Measures

The PRI ranking function, which distinguishes MINI documents from LINI documents, is generated from various product and process measures. Product measures are usually obtainable from direct analysis of source code. Lines of code, complexity, and number of children are a few examples of product measures. On the other hand, process measures are collected from an actual software development process. The amount of developer “effort” and the number of user-reported 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 ranking function. Therefore, a major component of the PRI process is the selection of these 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: (1) increase quality and (2) increase productivity. For this research I am primarily concerned with increasing quality. The successful inspection of a document has two main results: (1) finding defects which, once removed, increases software quality, or (2) 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” [22]. Some of the possible attributes of quality include: portability, reliability, efficiency, usability, testability, understandability, and modifiability [4]. Whatever definition and attributes is used for quality, inspections aim to increase or validate the level of quality in software. Therefore, the same attributes that define software quality also provide good indications of what documents need to be inspected. For example, finding documents that have unacceptable levels of portability, reliability, efficiency, usability, testability, understandability, and modifiability would provide a good indication that the documents are MINI.

Unfortunately, the software quality research community has not come to an agreement on what specific software measures accurately reflect the previous mentioned quality-attributes. Therefore, for the following example, let’s assume that the measures “lines of code,” “number of defects¹,” and “unit test coverage” are acceptable proxies of maintainability, reliability, and testability. Ta-

¹The defects used here are user-reported defects, which is usually stored in issue management systems, for example, Jira, Scarab, and Bugzilla. These should not be confused with the estimated number of defects that should be found after inspecting the document.

ble 1.1 is an example of the PRI ranking of a software project that contains three documents. The table presents the product and process measures ² associated with the three documents.

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

Document	PRI Ranking	Lines of Code	# of Defects	Coverage
Foo.java	missing	330 lines	1	49%
Bar.java	missing	400 lines	4	75%
Baz.java	missing	250 lines	0	96%

Table 1.1 contains a PRI ranking after finishing Step 1a of the PRI process. Notice that at this point in the PRI process, the PRI ranking for each workspace is missing, because we have only selected the measures that will be used. To actually generate a PRI ranking, we must configure indicators in Step 1b.

1.2.2 Step 1b: Calibration of Indicators

In Step 1a, we have selected the product and process measures that will be used in the PRI ranking function. In Step 1b, we must construct and calibrate indicators, which will provide a priority ranking of the documents. The main function of the indicators is to generate a PRI ranking for a document based on the measures values.

One way to accomplish this is to provide individual rankings for each of the measures. For example, an organization must agree on thresholds that evaluate the measure values and provide a score on a 0 to 100 scale; 0 indicating the worst possible score and 100 indicating the best possible score. Table 1.2 shows the PRI ranking after the creation of indicators, which provides a ranking for each measure and is aggregated to provide an overall ranking for each document. According the PRI ranking shown in the Table, Bar.java is the highest priority document and Baz.java is the lowest priority document.

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

Document	PRI Ranking	Lines of Code	# of Defects	Coverage
Bar.java	155 (=60+20+75)	400 lines (ranking=60)	4 (ranking=20)	75% (ranking=75)
Foo.java	169 (=70+50+49)	330 lines (ranking=70)	1 (ranking=50)	49% (ranking=49)
Baz.java	276 (=80+100+96)	250 lines (ranking=80)	0 (ranking=100)	96% (ranking=96)

²Note that, this table only shows three product and process measures, but in normal situations any number of measures may be used.

Some measures are more important than others when providing indicators for the PRI ranking. For example, an organization may find that coverage has a greater positive impact on the PRI ranking function than lines of code. Therefore, another consideration of Step 1b in the PRI process is the calibration of the indicators' importance. The calibration of the indicators is based on a numerical weighting system. Each indicator should be assigned a numerical weight and be individually calibrated.

Using the same example (Table 1.2), imagine that the organization has found coverage to be a leading indicator in defect prevention. Therefore, the calibration is adjusted and the values of coverage are given a higher weight than the other measures. This finding changes the PRI ranking. Table 1.3 shows the new PRI ranking after the adjustment of the indicators' calibration. Notice that the coverage rankings are multiplied by 2.

Table 1.3. Step 1b - Example PRI ranking - After Indicator Weighting Calibration

Document	PRI Ranking	Lines of Code	# of Defects	Coverage
Foo.java	218 (=70+50+98)	330 lines (ranking=70)	1 (ranking=50)	49% (ranking=49*2=98)
Bar.java	230 (=60+20+150)	400 lines (ranking=60)	4 (ranking=20)	75% (ranking=75*2=150)
Baz.java	372 (=80+100+192)	250 lines (ranking=80)	0 (ranking=100)	96% (ranking=96*2=192)

As a result of the calibration, the PRI ranking for Foo.java and Bar.java have switched places and now Foo.java is the highest priority document.

1.2.3 Step 1c: Declaring MINI and LINI documents

The final step of the PRI ranking function is the creation of a MINI-threshold, declares each document as MINI or a LINI, by evaluating the documents' PRI ranking. Table 1.4 shows one interpretation of the declaration of MINI and LINI based on the documents' rankings. In this example, the MINI-threshold was determined to be 371, which means all documents with a ranking lower than 371 are MINI and all rankings above are LINI. The major benefit of declaring a document, as a MINI or LINI, is that an organization can determine how many inspections PRI suggests are needed. Therefore, the organization can plan and reserve the necessary resources required to inspect all the MINI documents. In this example, PRI suggests that 2 inspections are needed on Foo.java and Bar.java.

Unfortunately, I have discovered that determining the MINI-threshold is harder than I first envisioned. Therefore, future research must be done to determine how to make an absolute declaration whether a document is a MINI or a LINI. However, I believe that Steps 1a and Step 1b

Table 1.4. Step 1c - Example PRI ranking - Declaring MINI and LINI

Document	PRI Ranking	Lines of Code	# of Defects	Coverage
Foo.java	MINI	330 lines	1	49%
Bar.java	MINI	400 lines	4	75%
Baz.java	LINI	250 lines	0	96%

are still greatly beneficial. In a software project with hundreds or even thousands of documents the MINI-threshold is less important, because the PRI rankings for each document provides a relative MINI and LINI declaration. In other words, the documents with the lowest numerical rankings will be more MINI than the documents with the highest numerical rankings and vice versa.

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

After the PRI ranking function 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, using the ranking presented in Table 1.4, an organization should select Foo.java for inspection, because it has the highest MINI ranking of the three documents. During the next inspection, Bar.java should be selected. However, according to PRI the inspection of Baz.java could be skipped, thus saving inspection resources. In this example, an organization will save resources required to inspect a single document. For a software project with thousands of documents, PRI could save the resources required to inspect hundreds of LINI documents. Of course, the organization can still choose to inspect all LINI documents. In this case, I would claim that PRI is still beneficial because it will allow them to prioritize their 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 defects with a higher severity than inspecting a LINI document.

1.2.5 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. However, it is best that the results of the inspection be recorded for use in the next step.

1.2.6 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 ranking function. For example, if the PRI ranking function appears to be incorrect, because it ranked a document as MINI but no high-severity defects were found, then the PRI ranking 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 ranking function more robust (Step 1a). Alternatively, one could calibrate the current indicators to refine and correct the PRI ranking function (Step 1b). In either case, an incorrect PRI ranking function provides data to help make future PRI ranking 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 the number of defects is a leading indicator of defects, then it should be calibrated with a higher weight.

1.3 Implementing PRI with Hackstat

To successfully use PRI, the determination of MINI and LINI must be obtainable for a very low cost. In other words, if the ranking function takes three months to generate, a software project may no longer need those specific recommendations. Therefore, the PRI rankings must be obtainable in real-time and without introducing new costs to the inspection process.

One way of obtaining ranking function values in real-time is through the use of the Hackstat system [23]. Therefore, I have decided to use the Hackstat system to implement and evaluate the PRI process in this research. Hackstat is a framework for collecting and analyzing software product and development process metrics in real-time³.

For this research, I have created an extension to the Hackstat system called the Hackstat PRI Extension (hackyPRI for short). hackyPRI provides a real-time PRI ranking by providing several automated functions. First, it provides Java implementations of PRI measures, which represent

³For more information about the Hackstat system see Chapter 3.

the product and process measures that are used in the PRI ranking function. Second, it provides Java implementations of PRI indicators, which are used in the PRI ranking function to interpret the values of the PRI measures to provide rankings for the PRI measures and the documents. Third, it automates and optimizes the generation of the PRI ranking function on years of Hackstat sensor data.

Figure 1.1 provides a portion of the PRI ranking for a software project that is obtainable from the Hackstat PRI Extension. Chapter 4 provides a detailed explanation of how to use the hackyPRI extension and how it generates a PRI ranking for a software project. The following is a short description of the data presented in Figure 1.1.

1. Each row in the table represents a workspace within a Hackstat project. The term, “workspaces”, generally means the directory where software code is located.
2. Each column to the right of the column labeled “Ranking,” represents the PRI measures used in the PRI ranking function. PRI measures provide the data that will be used in the PRI ranking function. The values of the PRI measures are presented to the user to help explain the overall ranking for the specified workspace.
3. PRI indicators work behind the scenes to help generate a PRI ranking. PRI indicators interpret the values of the PRI measures to calculate a ranking for a workspace. The resulting value of a PRI ranking function is presented in the column labeled “Ranking” for each workspace.
4. This figure presents some of the LINI workspaces within a Hackstat project. These workspaces, according to the hackyPRI extension, should not need inspection. Although not pictured, workspaces at the bottom of the table are MINI documents, which according to the hackyPRI extension, should be inspected by an organization.

The hackyPRI extension is implemented to fully support all 4 steps of the PRI process, except Step 1c. It is important to note that PRI is a proposed *process*; therefore many different tools can support it. Therefore, using the Hackstat PRI Extension is not required to conduct PRI inspections. However, there are many advantages of using the Hackstat system to automate the collection of product and process measures and the PRI ranking function [23].

1. Hackstat provides automated measure collection, therefore there is low to no overhead requirements for measure collection.
2. Hackstat can support any Software project’s development process, activities, and tools.

Workspaces (218):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont.	Com-mit	First Com-mit	Last Com-mit	Com-mit Cont.	Code Churn	Revi-ew	Last Revi-ew	Open Issue	Closed Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackystdext\src\org\hackystat\stdext\project	1179	developer1 (time=17.58, cont=52)	25.33 h	3.42 h	15-Mar-2003	15-Mar-2005	7	108	05-May-2003	17-Mar-2005	6	5202	0	0	0	1	LOC=1005, Class=7, Meth=70	LOC=96, Class=23, Meth=5	In=54, out=28	Pass=0, Fail=23, Error=21	78.00 %
hackyPRJ\src\org\hackystat\app\prj\mode\workspace\measures	1148	developer2 (time=25.42, cont=334)	25.42 h	3.25 h	27-Jan-2005	17-Apr-2005	1	334	28-Jan-2005	17-Apr-2005	1	13429	10	11-Mar-2005	0	0	LOC=4182, Class=48, Meth=312	LOC=890, Class=22, Meth=22	In=90, out=57	Pass=0, Fail=36, Error=10	93.12 %
hackyReport\src\org\hackystat\stdext\report	1126	developer3 (time=9.83, cont=873)	11.50 h	0.58 h	15-Jun-2003	10-Nov-2004	2	127	13-Jun-2003	27-Oct-2004	3	5882	0	0	0	0	LOC=190, Class=6, Meth=24	LOC=13, Class=1, Meth=1	In=26, out=4	Pass=0, Fail=4, Error=0	80.00 %
hackyKernel\src\org\hackystat\kernel\admin	1118	developer1 (time=24.17, cont=597)	39.58 h	4.42 h	04-May-2003	22-Apr-2005	6	145	04-May-2003	22-Apr-2005	5	6421	0	0	0	3	LOC=86, Class=11, Meth=20	LOC=933, Class=3, Meth=7	In=174, out=26	Pass=0, Fail=22, Error=16	78.79 %
hackystdext\src\org\hackystat\stdext\project\cache	1115	developer1 (time=24.17, cont=597)	28.00 h	4.83 h	25-Feb-2004	15-Mar-2005	6	185	25-Feb-2004	17-Mar-2005	6	9421	0	0	0	0	LOC=294, Class=11, Meth=32	LOC=32, Class=1, Meth=2	In=41, out=17	Pass=0, Fail=9, Error=123	66.67 %
hackystdext\src\org\hackystat\stdext\activity\analysis\project\activetime	1113	developer4 (time=2.00, cont=8)	5.58 h	2.83 h	09-May-2003	13-Jul-2004	5	25	10-May-2003	09-Nov-2004	6	1470	0	0	0	0	LOC=167, Class=2, Meth=5	LOC=22, Class=1, Meth=1	In=0, out=30	Pass=0, Fail=46, Error=20	100.00 %
hackyKernel\src\org\hackystat\kernel\cache	1100	developer1 (time=2.33, cont=12)	2.75 h	0.50 h	04-May-2003	10-Dec-2004	3	15	04-May-2003	10-Dec-2004	2	829	0	0	0	0	LOC=347, Class=3, Meth=4	LOC=85, Class=1, Meth=4	In=15, out=2	Pass=0, Fail=0, Error=2	91.30 %
hackyKernel\src\org\hackystat\kernel\command	1095	developer1 (time=2.09, cont=12)	2.42 h	0.33 h	04-May-2003	24-Sep-2004	2	17	04-May-2003	24-Sep-2004	2	1094	0	0	0	0	LOC=481, Class=4, Meth=44	LOC=10, Class=1, Meth=1	In=81, out=11	Pass=0, Fail=0, Error=2	88.24 %
hackyKernel\src\org\hackystat\kernel\util	1085	developer4 (time=6.56, cont=54)	17.33 h	7.50 h	04-May-2003	18-Mar-2005	5	112	04-May-2003	24-Mar-2005	5	4537	0	0	0	0	LOC=1317, Class=16, Meth=151	LOC=381, Class=10, Meth=22	In=456, out=16	Pass=0, Fail=0, Error=163	83.00 %
hackystdext\src\org\hackystat\stdext\activity\analysis\activetime	1072	developer4 (time=6.58, cont=13)	9.17 h	3.08 h	01-Jan-2003	05-Jun-2004	3	27	10-May-2003	25-Jun-2004	3	1681	0	0	0	0	LOC=133, Class=1, Meth=3	LOC=25, Class=1, Meth=1	In=0, out=26	Pass=0, Fail=106, Error=47	100.00 %
hackyPRJ\src\org\hackystat\stdext\ddview	1072	developer4 (time=0.67, cont=14)	0.67 h	0.00 h	24-Sep-2004	01-Nov-2004	1	14	24-Sep-2004	03-Nov-2004	1	876	44	26-Jan-2005	0	0	LOC=311, Class=8, Meth=21	LOC=101, Class=4, Meth=6	In=6, out=22	Pass=0, Fail=4, Error=20	83.33 %
hackystdext\src\org\hackystat\stdext\admin\analysis\serverstatus	1066	developer1 (time=0.59, cont=8)	0.83 h	0.42 h	08-May-2003	21-Apr-2004	4	10	08-May-2003	25-Jun-2004	2	537	0	0	0	0	LOC=240, Class=2, Meth=19	LOC=13, Class=1, Meth=1	In=1, out=19	Pass=0, Fail=8, Error=5	94.44 %
hackyKernel\src\org\hackystat\kernel\seno\usermap	1064	developer1 (time=1.25, cont=12)	21.17 h	1.25 h	08-Oct-2004	18-Nov-2004	5	48	10-Oct-2004	16-Feb-2005	5	1133	34	20-Oct-2004	0	0	LOC=293, Class=2, Meth=9	LOC=54, Class=2, Meth=9	In=7, out=10	Pass=0, Fail=21, Error=11	40.91 %
hackystdext\src\org\hackystat\stdext\daily\analysis\daystat	1064	developer4 (time=0.25, cont=11)	0.50 h	0.25 h	21-Aug-2003	18-Sep-2004	3	24	05-May-2003	19-Oct-2004	4	1111	0	0	0	0	LOC=149, Class=7, Meth=49	LOC=149, Class=9, Meth=9	In=45, out=4	Pass=0, Fail=0, Error=2	87.88 %
hackystdext\src\org\hackystat\stdext\admin\analysis\adoption	1054	developer1 (time=1.83, cont=9)	2.17 h	0.58 h	08-May-2003	26-Oct-2004	2	10	08-May-2003	26-Oct-2004	2	323	0	0	0	0	LOC=120, Class=2, Meth=4	LOC=10, Class=1, Meth=1	In=1, out=12	Pass=0, Fail=19, Error=6	100.00 %
hackystdext\src\org\hackystat\stdext\comon\alert\newdata	1052	developer1 (time=0.25, cont=12)	0.50 h	0.33 h	05-May-2003	06-Apr-2004	3	20	05-May-2003	30-Jul-2004	3	431	0	0	0	0	LOC=112, Class=4, Meth=11	LOC=16, Class=1, Meth=1	In=2, out=15	Pass=0, Fail=14, Error=23	80.00 %
hackyPRJ\src\org\hackystat\app\prj\mode\workspace	1048	developer2 (time=15.00, cont=67)	15.08 h	0.67 h	26-Jan-2005	17-Apr-2005	2	67	28-Jan-2005	17-Apr-2005	1	2485	19	24-Mar-2005	0	3	LOC=588, Class=8, Meth=40	LOC=93, Class=4, Meth=4	In=55, out=34	Pass=0, Fail=3, Error=0	91.43 %
hackystdext\src\org\hackystat\stdext\analysis\project\filetime	1048	developer1 (time=0.25, cont=2)	0.33 h	0.17 h	24-Jun-2004	24-Jun-2004	1	4	25-Jun-2004	09-Nov-2004	3	232	0	0	0	0	LOC=132, Class=5, Meth=5	LOC=20, Class=1, Meth=1	In=0, out=20	Pass=0, Fail=0, Error=3	100.00 %
hackyKernel\src\org\hackystat\kernel\soap	1045	developer1 (time=0.67, cont=11)	1.58 h	0.17 h	04-May-2003	15-Nov-2004	5	17	04-May-2003	08-Mar-2004	2	495	0	0	0	0	LOC=202, Class=4, Meth=8	LOC=33, Class=1, Meth=1	In=6, out=11	Pass=0, Fail=4, Error=0	63.64 %
hackystdext\src\org\hackystat\stdext\comon\analysis\lists\memorata	1044	developer1 (time=1.17, cont=5)	1.17 h	1.17 h	03-Apr-2004	06-Apr-2004	1	6	20-Jan-2004	30-Sep-2004	2	97	0	0	0	0	LOC=19, Class=2, Meth=2	LOC=19, Class=1, Meth=1	In=0, out=16	Pass=0, Fail=23, Error=36	100.00 %
hackystdext\src\org\hackystat\stdext\comon\analysis\daydatasummary	1040	developer1 (time=0.58, cont=9)	0.58 h	0.50 h	11-Jan-2004	09-Apr-2004	1	10	12-Jan-2004	30-Sep-2004	2	129	0	0	0	0	LOC=92, Class=2, Meth=5	LOC=18, Class=1, Meth=1	In=0, out=18	Pass=0, Fail=18, Error=50	100.00 %

Figure 1.1. Workspaces are listed with their respective PRJ ranking and measures.

3. Since Hackstat can collect product and process measures in real-time, the analysis of these measures to generate useful information that can help the project are also in real-time.

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 separated into the following three main claims, which are based on the intended benefits of PRI.

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

1.4.1 Thesis Claim 1

My first claim states that the inspection of MINI documents will generate defects with a higher severity 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.

However, if a document is identified as LINI and yields many high-severity defects, then the PRI ranking function is flawed. An organization can use this information to refine the PRI ranking function. It is my hope that at some point after future research is conducted on PRI, I will be able to successfully calibrate the PRI ranking function for a Hackstat project and be able to provide a description of best practices and how they are accomplished.

1.4.2 Thesis Claim 2

My second claim states that PRI can enhance the selection process. In the traditional inspection process, the selection process is based on a developer selecting and volunteering a document for inspection. In most cases, developers select documents to volunteer to find any high-severity defects before it is released. However, the current literature does not provide much guidance on which documents should be volunteered.

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 generate a list of MINI documents, then the developer can focus his selection on a smaller set of documents. Therefore, PRI enhances the volunteering process by suggesting 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.

Minimize the Number of Possible Documents

PRI can minimize the number of documents that should be considered for inspection. In PRI, the number of possible documents is limited to MINI documents. This reduces the number of possible inspections and can be advantageous for organizations that cannot inspect every document.

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

An 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 select the right code to inspect. However, developers often do not know where every high-severity defect will appear. In other words, leaving this decision up to the subjective understanding of a developer maybe error prone [16]. 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 use PRI to generate a list of MINI documents and choose code from this smaller list.

Priority Ranking of Documents

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 minimizes the number of documents; and now that the number is reduced the developer still must select from this smaller list. To

support this selection, PRI ranks the documents according to the calibration and numerical ranking. For example consider this scenario:

A developer 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 4 of his documents appear to be MINI. In addition, he is able to use the rankings to select a MINI document that he believes would generate the most high-severity 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.

1.4.3 Thesis Claim 3

My third claim states that PRI can identify documents that have completed 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 require inspection have not been. PRI can find MINI documents that are not typically identified using a volunteer-based document selection process. An example of this benefit is illustrated in the following scenario:

As we all know there are many problems that occur in any software project; requirements change, clients change or better clarify the software's functionality, and design problems are found late in the development process. These problems are commonly solved with new code, new patches, and quick fixes.

This situation is quite dangerous, because all software projects evolve and outdated documents may become error prone. Therefore, it is important to realize that old documents can also be MINI. Software Inspection [1] 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 cannot inspect every document moving through the development cycle, it is very likely that some documents will finish the development cycle with high-severity defects. Therefore, ensuring that "finished" documents are included as potential inspection candidates is very important.

1.5 Exploratory Study and Results

This section provides a short description of the methodologies used to study my thesis claims and the results. Chapter 5, provides a detailed explanation of the methodologies and procedures that were used in the exploratory study of PRI. Chapter 6, provides a detailed explanation of the results.

I have studied the main thesis of this research by testing each of my three claims. In addition, I studied the inspection process of the Hackystat system developed by the Collaborative Software Development Laboratory (CSDL). I also used the developers of Hackystat as subjects in my study.

1.5.1 Thesis Claim 1

My first claim states that the inspection of MINI documents will generate more high-severity defects than LINI documents. Throughout my study I have collected information about 9 inspections. By analyzing the results of these inspections, I was able to provide supporting evidence for this claim.

The results show evidence that inspections of MINI documents generated more overall defects, more high-severity defects, and more Program Logic defects than LINI documents. In addition, several other independent results validate this finding.

1.5.2 Thesis Claim 2

My second claim states that PRI enhances the volunteer-based document selection process. To evaluate this claim, I conducted a study to understand how developers select documents for inspection. First, I assessed 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 without the use of the PRI rankings. Second, I worked with the developers to select a document to be inspected by CSDL to evaluate the developers' skill at selecting documents and the ranking generated by hackyPRI.

The primary result of this study shows that the developers tend to select documents for inspection based on the "age" of the document, instead of more traditional quality measures like coverage and unit tests. This finding is surprising, because the developers have a large amount of software measures available to them. In addition, the rankings that they provided showed large variations when ranking code they did not author recently. Based on these results, I believe that PRI can provide the developers with more useful information, in the form of product and process measures

and the PRI rankings themselves, to select documents for inspection. This will hopefully, provide more consistent results when selecting documents for inspection in areas where their subjective knowledge is limited.

1.5.3 Thesis Claim 3

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

Unfortunately, there were two major problems in gathering a sufficient amount of supporting evidence for this claim. First, I believe the methodology of my study was flawed, because it was not designed well enough to test this claim. Second, with CSDL's limited inspection resources, it was difficult to schedule enough inspections to address this claim. On the other hand, other results associated with this claim show that developers tend to associate the document's age with the likelihood that it needs inspection, this may indicate that developers do not typically volunteer old documents for inspection. This is precisely the type of problem PRI is intended to solve. Future studies are needed to gain more insights into this claim.

1.6 Structure of the Thesis

The remainder of this 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 exploratory study methodology that has been implemented to evaluate the claims and benefits of PRI. Chapter 6 discusses the results of my study of PRI and hackyPRI. Finally, Chapter 7 discusses my conclusions and future directions of this research.

Chapter 2

Related Work

This chapter presents previous research that is 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 research on software inspection has focused on the process in which inspection is conducted. Instead, my research focuses on the selection of documents for inspection.

Inspections are one of the oldest formal software development processes. For more than 30 years, researchers have been studying many areas within the inspection domain. In this chapter, I will discuss three research areas that are most related to my research.

First, I will discuss the different types of inspection processes. Throughout this thesis, I use the term “inspection” to encompass processes defined as a static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems [24]. However, since Michael Fagan invented the inspection technique in 1976, there have been many variations on the general concept. We now have Fagan Inspection [6], Software Inspection [1], In-Process Inspection [7], peer review [8], software reviews, and code walkthroughs just to name a few. These processes range from formal and labor-intensive activities to informal and very cheap methods. Furthermore, each of these processes claims to be the best inspection method for certain circumstances. This area of research is very general and is almost entirely focused on how to conduct inspections.

Second, I will discuss some research ideas that alter the traditional inspection process in hopes of increasing its effectiveness. These ideas generally come from the understanding that inspections are too labor-intensive and finding a more effective method would greatly affect the results. For example, some argue that the inspection meeting is a waste of time and resources [9, 10].

Others argue that the inspection meeting is critical for supporting social and educational aspects of inspection [11, 12, 13, 14]. Unlike, the previous research area, alterations to an inspection process are more specific to an individual organization's development process. For example, deciding to alter an inspection process to remove meetings will work very well for an organization that is geographically dispersed. It would have less of an impact for an organization with three developers that work in the same office.

Third, I will discuss a very small research area focused on determining the optimum software document to select for inspection. There are very few publications that address this issue. Unlike the first two areas, this research area primarily focuses on finding a set of measures that help the selection process. Software measures vary tremendously from organization to organization; therefore acceptance of this research has been relatively low.

2.1 Types of Inspection

There are many different types of inspection processes. As previously stated, they range from very formal to very informal activities. This area of research focuses on the inspection process and how inspections should be conducted. In this section, I introduce a few of the most widely accepted inspection processes.

There are two major problems that account for the large variations in inspection research. First, there is no standard or accepted definition of the inspection process. Second, even when processes are fairly well designed they are extremely hard to follow. For example, a report in the IEEE Transactions on Software Engineering found that 84 percent of organizations performed inspections, but 0 percent performed them entirely correct [25]. Thus in practice there is many different executions of the same inspection process. These problems indicate that inspection processes have ambiguous or unclear definitions [26]. Another reason is that there is relatively little knowledge and theory about inspection effectiveness factors [2, 26]. Regardless of the problems listed above, it is generally accepted that adopting any variation of the inspection process is much better than not doing any.

2.1.1 Informal Inspection Processes

Informal inspections processes typically are loosely defined, not planned, not structured, and not recorded. Ad hoc Review, Peer Deskcheck, Pair Programming, and Walkthrough are all different types of informal inspection processes [17]. Most inspection literature suggests that even

though informal inspections are proven to find less defects than formal inspection, they are some times the best solutions for certain situations [1, 17]. For example, informal inspections can be used on low-risk documents and formal inspections can be used on high-risk documents [17]. The defining factor of informal inspection processes is they generally require much less resources than formal inspections.

Ad hoc review is by far the most informal form of inspection. It is a spur-of-the moment request of asking a fellow developer to help look at a piece of code to solve a problem. These reviews are not planned, not measured, and have no long-term impact on the software development process.

Pair programming is a practice from Extreme Programming whereby two programmers work side-by-side at one computer on the same design, algorithm, code or test. This practice is often considered to be a continuous informal inspection process, because the creation of documents is constantly under evaluation from the second programmer [17]. Pair programming is a relatively new practice and is not considered to be one of the traditional inspection processes.

Walkthroughs are informal presentations in which a developer, usually the author of the source code, describes various aspects of the document under review [17, 27]. Walkthroughs do not have a defined procedure and the results of the process are often not recorded.

2.1.2 Formal Inspection Processes

Formal inspection processes typically are defined in detail, carefully planned, very structured, multi-step, have assigned roles for participants, and recorded. Like informal inspections, formal inspections are most beneficial in certain situations. For example, formal inspections are generally quite expensive as they require extensive training and up to 15 percent of the projects resources [1].

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” [1].

One of the most widely accepted types of formal inspection is “Software Inspection”, which was developed by Tom Gilb and Dorothy Graham [1] 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. Since Software Inspection is one of the most widely accepted processes, I spend the next few paragraphs explaining the process in more detail.

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.

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.

(1) Request: Initiating the Inspection Process

The 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.

(2) 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" [1].

(3) 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.

(4) 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.

(5) Individual Checking: The Search for Potential Defects

The participants, or “inspectors”, 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.

(6) 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.

(7) 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 defects (issues become defects if they are deemed as valid) that were identified and is required to make the necessary improvements to remove any defects from the document.

(8) Follow up: Checking the Editing

The purpose of this phase is to ensure that the author correctly executed the Edit phase. 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.

(9) Exit: Making Sure the Product is Ready 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.

(10) 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 and unavoidable defects remaining in the document, then such defects must be documented.

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; Software 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” [1].

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

2.1.3 Software Inspection and Priority Ranked Inspection

The different types of traditional inspection processes explained above are quite different from the Priority Ranked Inspection (PRI) process. 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.

There are three main areas where Software Inspection and Priority Ranked Inspection differ. They are the selection of documents, cost cutting, and volunteering.

Lack of Discussion about Selection of Documents

In the book, “Software Inspection”, Tom Gilb and Dorothy Graham [1] 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.*

Like the Software Inspection process, most 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 [1]. What happens to the volunteering process when an organization can inspect one in every five volunteered 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; therefore 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 traditional inspection literature provides.

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” [12].

I believe 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 [2, 16]. 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.

Volunteering

Another problem associated with the selection of documents for inspection is the notion of volunteering. Notice 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.”

In my opinion, many of the problems associated with volunteering is that most developers do not have a clear understanding of what documents are best to volunteer [16].

2.2 Alterations to the Inspection Process

This section presents some examples of related research on alterations to traditional inspection processes to increase inspection effectiveness. Research in this inspection area focuses on specific process changes and the addition of tool support. I spend most of this section discussing an area of research aimed at using automated tools to support the inspection process, because this area is directly related to my PRI research. However, there are many other different research findings and suggestions that are not presented in this section.

2.2.1 Eliminating Steps in Inspection

One of the first things organizations do to save resources is alter the inspection process by changing the required steps within the process. I present two of the many publications in this research area. The first is a simple suggestion to eliminate the inspection meeting. The second suggests that outsourcing the entire inspection process is beneficial.

Inspections Without Meetings

Lawrence Votta Jr. found that the inspection meeting, where the inspectors meet and discuss the validity of the issues found in the individual phase, is ineffective and can actually delay the inspection process days and even weeks [10, 18]. Therefore, he proposed that the inspection meeting be totally eliminated. Of course, other publications and organizations disagree and would rather hold inspection meetings, because they value the benefits of synergy and education over the increased resources need to conduct inspections [11, 12, 13, 14].

Outsourcing Inspections

An extreme process change is outsourcing the whole inspection process to a third-party company. Jasper Kamperman states that outsourced software inspections are cheaper, easier, and greatly beneficial [19]. A company utilizing outsourced inspections would send out software code, documentation, and a survey and receive back a list of potential defects. However, Kamperman mentions that outsourced inspections will probably find more superficial programmatic defects than deep design problems. In addition, according to other research, Automated Software Inspection tools are also an effective way to detect superficial programmatic defects. I discuss these tools in the next section.

2.2.2 Automated Software Inspection

Due to the rigorous and labor-intensive requirements of formal inspection processes, a whole branch of research on inspection is aimed at automating some part of the inspection process. There are two different ways automation can be added to an inspection process. First, add tools that automate the inspection process, which makes it less labor-intensive to follow guidelines and record the results [21]. Second, add tools that inspect code automatically, which replace some of the work required of human inspectors [21]. Most of the current research focuses on the second type of automatic software inspection. This type of automation has a very low overhead with many potential benefits. For example, tools that support automated software inspection are less dependent on human factors [28]. PRI implemented with Hackystat is an example of a type of automated software inspection support.

Automated Software Inspection Tools

Automatic software inspection (ASI) tools are a relatively new way to identify defects early in a development process. In addition, it is very useful in identifying a subset of defects found in manual software inspection, but is not as labor-intensive. ASI tools are also known as static analysis tools that analyze source code and provide error and warning messages similar to a compiler.

Using ASI does not replace manual inspections. Labor-intensive manual inspections, for example Software Inspections, will find more complex, functional, algorithmic design problems [28]. However, ASI tools will make manual inspections more effective by allowing the inspectors to focus on these issues. In the related research field of pre-release defect density, Nagappan and Ball [29] had very successful findings in utilizing static analysis, which should also translate to inspection:

1. Static analysis defect density can be used as early indicators of pre-release defect density;
2. Static analysis defect density can be used to predict pre-release defect density at statistically significant levels;
3. Static analysis defect density can be used to discriminate between components of high and low quality.

Criticisms of ASI tools state that the defects found are generally superficial programmatic errors and cannot replace manual inspections [21]. In addition, a major problem with ASI tools is that they can generate up to 50 false positives for every valid defect [28].

Code Smells

A unique idea to improve the effectiveness of software inspection with automation is “code smells” [21]. Code smells are a metaphor to describe patterns that are generally associated with bad design and bad programming practices [21]. Code smells are different from other static analysis tools such as the ones mentioned in the previous section. Instead, it is inspired by the “code smells” defined in [30], which describe patterns of code that requires refactoring. Like the human sense of smell, any group of source code has a smell; the question is whether the smell is good or bad. The following are a few examples of code smells discussed in [21]:

1. Duplicate Code
2. Methods that are too long
3. Classes that contain too much functionality
4. Classes that violate data hiding of encapsulation
5. Classes that delegate the majority of their functionality to other classes

There are three defining characteristics of code smells; what smells are detected, how the smells are detected, and how are the smells are presented.

What smells are detected There are three general rules associated with code smells. First, like the human sense of smell, there is no static list of all possible code smells. Different projects and organizations can have a different set of code smells that works best for their quality assurance. Second, code smells are subjective measures, which are based on the organization and project’s previous experience. Code smells are parameterized to provide subjectivity on whether specific smells are good or bad. Third, code smells do not have to be precise. In other words, code smells do not give an absolute decision about the quality of a software document. Instead, it provides a possible indication of problems relative to the other documents in the same software system.

These three rules are very similar to the rules in PRI. First, PRI is fully extensible and provides the ability to add any PRI measure and PRI indicator to the ranking function. Second,

PRI supports subjectivity in the calibration of PRI indicators. Third, PRI does not give an absolute determination of a software document's MINI or LINI distinction. Instead, the MINI and LINI distinction is relative to the other PRI rankings in the same project.

How the smells are detected Code smells are detected with a software tool that statically analyzes source code. As I previously stated, the static analysis tool makes automated software inspection possible. Therefore, code smells provides the possibility of lowering inspection resources by automatically detecting code that has a bad smell.

Code smells are measures of a software product. PRI differs from this approach, because it allows measures of software product and development process activities. One interesting future enhancement to PRI is to investigate the addition of code smells to the PRI measures. One would think that if code smells are a useful measure, then the combination of code smells with other software product and development process measures would also be beneficial.

How are the smells are presented Code smells are presented with structural graphs. These graphs allow the user to interact with the graphs to find "smelly" code.

Although, I haven't used the presentation user interface, it seems that this particular choice of graphical presentation is a little difficult to use effectively. In my opinion, users will not want to search for the "smelly" code. Therefore, I believe the most useful presentation of code smells is a tabular ranking similar to the ranking provided in PRI.

To conclude code smells are a unique automatic software inspection technique and has many of the same problems and solutions that I have addressed in my PRI research.

2.3 Selection of Documents for Inspection

This section presents some inspection research on increasing the effectiveness of inspection by selecting the right documents to inspect. This is a very small area of inspection research relative to the two previous areas discussed in the proceeding sections. In fact, in some ways it is a much smaller branch of the "Alterations to the Inspection Process" domain. Research in this area is almost always focused on the utilization of tool support to help aid the selection process. Since tool support is very specific to an organization's development process, research in this area is not as general as the two previous areas.

“Few organizations have the time and commitment to inspect everything they create (unless contractually required to do so), so focus your inspection resources where they will do the most good” [17]. In this point in my thesis, I’ve said this many different ways. And each of the following sections contain research that try to find the “best code to inspect.” Once again, I believe PRI is an example of research in this area.

2.3.1 Code Smells

As I already explained in Section 2.2.2, code smells are a unique idea to improve the effectiveness by identifying software code that needs to be refactored. In addition, code smells can aid the inspectors’ assessment of quality software code. The research on code smells not only provides automated tool support, but it also provides a general process of which any tool can support the basic theory of code smells.

Although, authors of the publication [21] do not explicitly say, code smells could also be used to aid the document selection process for inspection. Instead, they leave us with a general statement that code smells can immediately show the maintainers if the system contains bad smells, what parts are affected, and where the concentration of smells is the highest [21]. In addition, they do not make any claims that the inspection of code with “really bad code smells” will detect more high-severity defects than the inspection of code with “really good code smells.”

2.3.2 Crocodile

Crocodile [31] is another automatic software inspection tool similar in nature to code smells. The research context of this tool is large object-oriented software systems. The publication [31] plainly states that a large software system is often too large to be entirely inspected by humans. Using Crocodile can concentrate inspection resources to “critical” areas of the system where inspection is most necessary. Therefore, unlike code smells, this research has explicitly claimed that Crocodile can help pre-select suspicious modules for manual inspection. In addition to building a tool, the publication [31] creates a process that utilizes the tool to support its intended design of auditing large software systems.

The Crocodile tool quantitatively measures structural properties of an object-oriented system. Like most automatic software inspection tools, this is done by static analysis of source code. The measurements are then fed into a database where meta-analysis is conducted to build a quality model of the system. The quality model is very similar to a Goal-Question-Metric graph and con-

sists of a goal (which is identifying software quality), factors (which are quality measures such as maintainability and reusability), criteria (which help define and measure the factors), and metrics (which are used to determine if the criteria is fulfilled). Each level of the quality model contains threshold values to determine the “good” or “bad” meaning of the measures. For example, the allowed threshold for the “Number of Parents” measure is 0 and 1. Although not explicitly stated, I assume these thresholds are configurable.

The Crocodile tool and the process of its use are, in some ways, very similar to PRI. First, they both have the same goal of trying to identify areas of a software system where inspection is most necessary (MINI). Second, the Crocodile’s threshold limits to determine meaning of the measures are very similar to the PRI indicators, which also uses threshold limits to provide a numerical ranking. Third, Crocodile’s process includes a phase in which the tool can be adjusted based on any anomalies found in the results. However, there are a few differences. First, Crocodile only supports one type of measurement, which is object-oriented metrics from static analysis. PRI supports any type of measurement, both product and process measures. Second, like code smells, they do not make the claim that the inspection of “critical areas” will detect more high-severity defects than the inspection of “non critical areas.”

2.3.3 Risk Analysis

Evaluating the risk of potential defects, usually called risk analysis, within documents is another way to select documents for inspection. Risk can be defined as the likelihood that a work product contains defects and the potential for damage if it does [17]. Therefore, one objective of using this approach is to reduce the risk associated with a specific document.

Karl Wieggers mentions a few high-level selection criteria in his book, “Peer Reviews in Software: A practical guide.” [17]. According to Wieggers, an organization should select documents to inspect that have the following properties.

1. Code that could potentially contain errors that could propagate throughout your product and lead to expensive rework or to execution failures.
2. Code that traces back to safety- or security-related requirements.
3. Modules that has been changed many times.
4. Modules that has a history of containing many defects.
5. Fundamental and early-stage documents, such as requirements and specifications.

6. Documents on which critical decisions are based, such as architectural models that define the interfaces between major system components.
7. The parts you aren't sure how to do, such as modules that implement unfamiliar or complex algorithms or enforce complicated business rules and other areas in which the developers lack experience or knowledge.
8. Components that will be used repeatedly.

To be able to repeatedly and reliably determine if work products contain any of these properties, there must be a way to accurately measure the properties that indicate documents in most need of inspection. PRI implemented with Hackstat provides the possibility to be able to “sense” some, if not all, of these properties.

Chapter 3

The Hackystat System

The Priority Ranked Inspection process is a theoretical process that can be implemented in many different ways. In this research, I utilized the Hackystat system to implement the PRI process for a specific organization. This chapter is a brief introduction to the Hackystat system, which was invented by Professor 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 include: lines of code, dependency, 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, some process measures include: the number of developers, the developers' "effort", the number of major releases, and the number of defects.

The following list summarizes the features that Hackystat provides [23]:

1. Hackystat utilizes custom "sensors" that are "attached" to various software development tools. These 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, Ikazyo.org, University of Torino, University of Maryland, and of course the University of Hawaii. A few of the many publications associated with Hackystat are the following: [32, 33, 34, 35, 36, 37, 38, 39, 40]

3.2 Hackystat's Architecture

Figure 3.1 presents one way of viewing Hackystat's architecture. Rather than paraphrase the information provided in Hackystat's User Guide, I provide the entire segment about Hackystat's architecture [23].

In this view, the "clients" are development environment tools, such as editors (Emacs, Eclipse, Vim), configuration management systems (CVS, Harvest), build tools (Ant, Make), Unit Testing tools (JUnit), and so forth. For each of these tools, a custom Hackystat sensor must be developed. It is "custom" in the sense that it must use the plug-in or extension point API for the tool, and "custom" in the sense that the type of product or process data that it collects is specific to the tool it supports.

Once data is collected by these client-side sensors, it is transmitted using SOAP to the "server", which is a custom web application running within a conventional servlet-supporting web server such as Apache Tomcat. Note that the client-side sensors have the ability to cache data in the event that a network connection cannot be made to the server and resend it later, allowing the developers to work offline.

Upon receipt of the "raw" sensor data by the server, various analyses can be run. Some of these analyses are run automatically by the server each day, others are run only when invoked by developers from a Web Browser interface. The goal of these analyses are typically to create abstractions of the raw sensor data stream that help developers and managers to interpret the current state and trajectory of the project and gain insight into possible problems or opportunities for improvement going forward.

In certain cases, these abstractions can be automatically emailed back to the developers on a daily basis, creating a feedback loop.

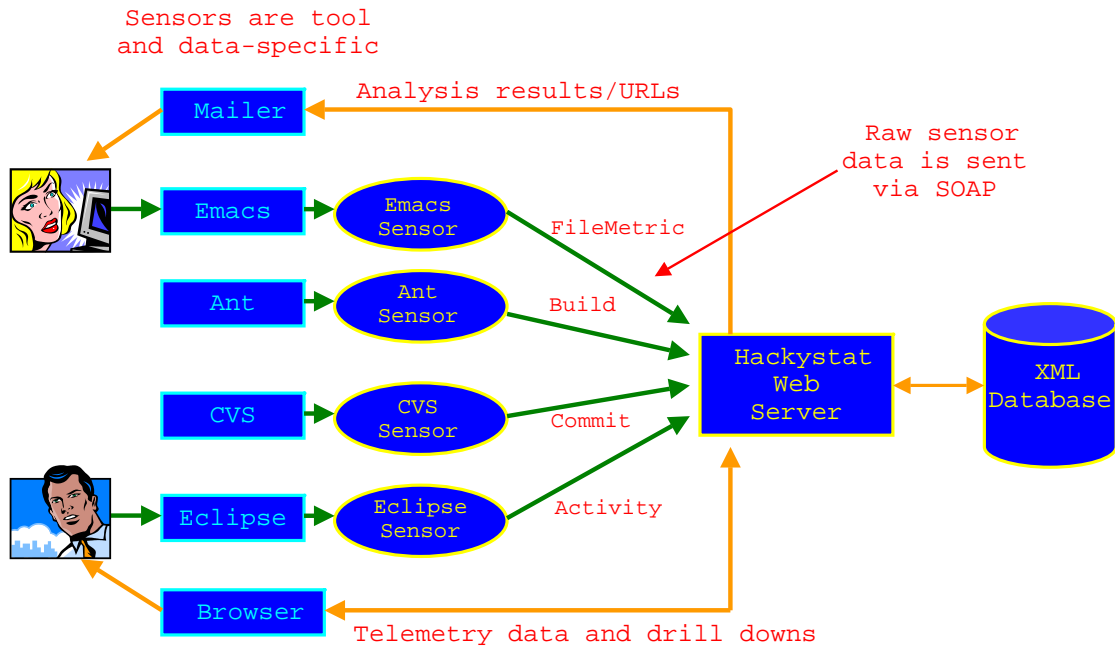


Figure 3.1. Displays a high level overview of the Hackystat architecture

3.3 Hackystat Sensors

Figure 3.2 presents a mapping between Hackystat’s sensors and sensor data types. Once again, rather than paraphrase the information provided in Hackystat’s User Guide, I provide the entire segment about Sensor Data Types [23].

A “sensor data type” represents the structure of a single kind of software product or process measurement. For example, the “UnitTest” sensor data type represents the outcome of invoking a single unit test. This data typically includes the time that the unit test was invoked, the class name of the unit test code, and the results of the invocation (pass, fail, exception). The “Commit” sensor data type represents the occurrence of committing a file to a configuration management repository, and thus has very different fields associated with it that are appropriate to this kind of event. A given sensor can collect data for one or more sensor data types.

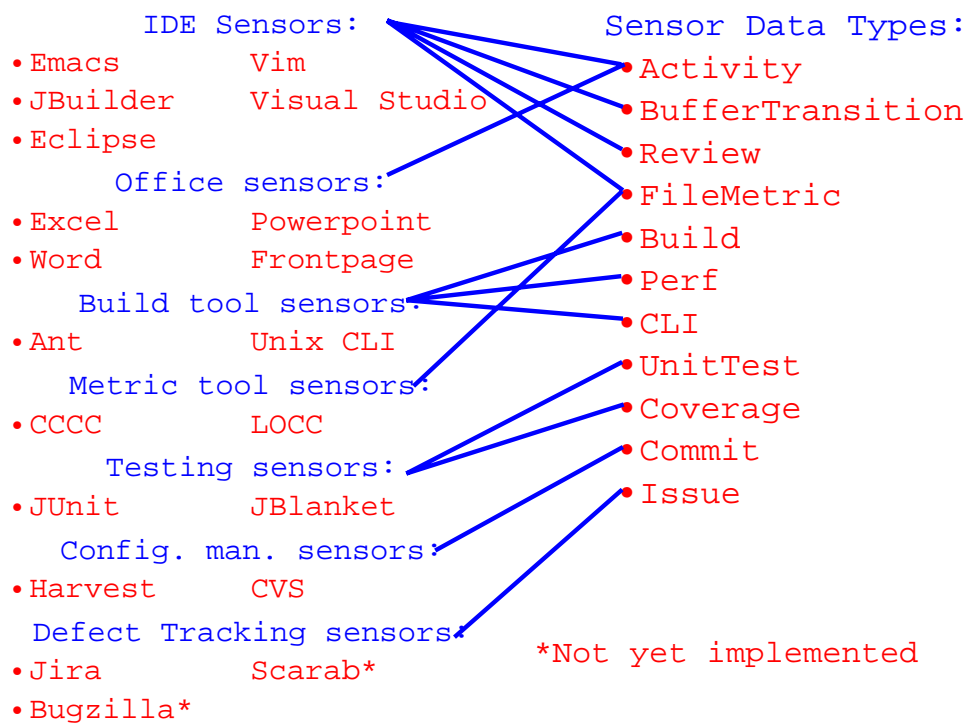


Figure 3.2. Displays the connections between the Sensors and the Sensor Data Types provided in Hackystat.

3.4 Some Screenshots of Hackystat

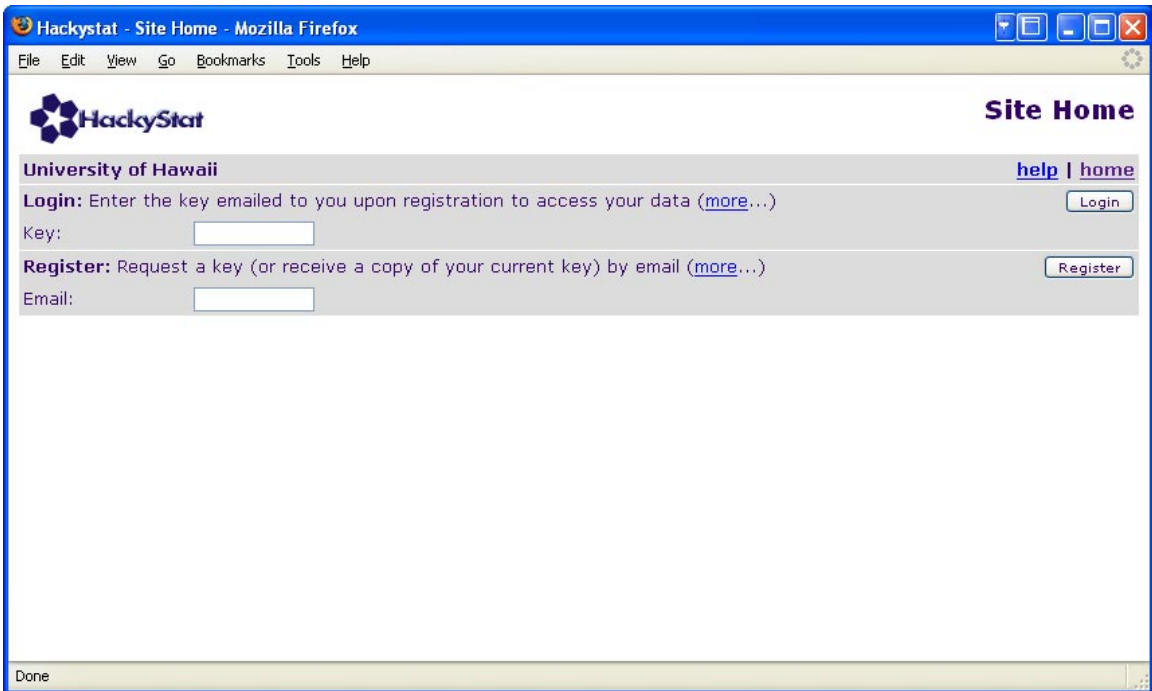


Figure 3.3. Displays the Hackystat home page. Hackystat is designed to protect the data of its users. Therefore, the system requires a registration and login before using Hackystat.

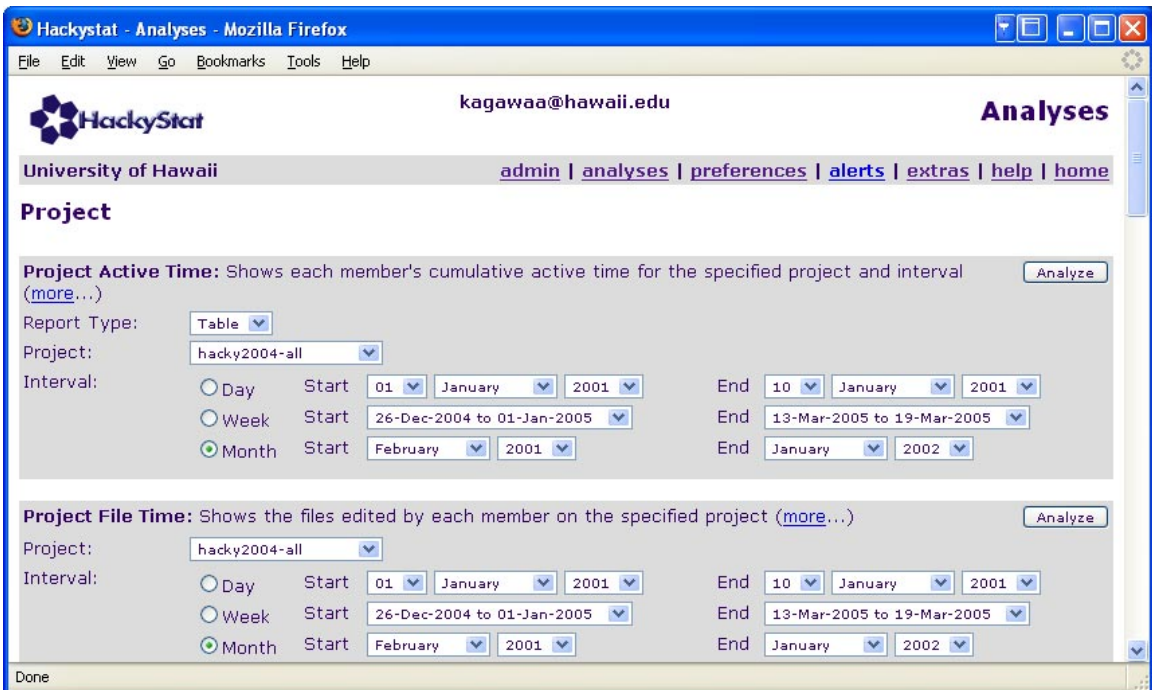


Figure 3.4. Provides a listing of all available Hackystat analyses. Each analysis displays its results with image charts, HTML tables, and downloadable files.

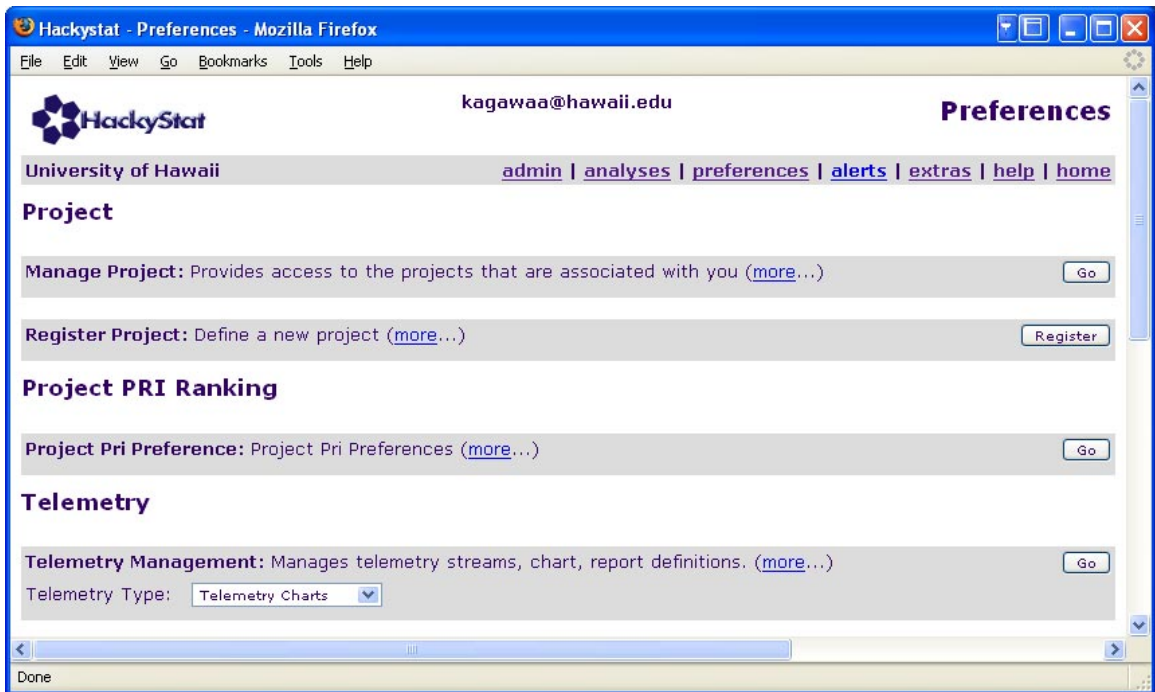


Figure 3.5. Provides a listing of all preferences, which allow users to set specific settings that are sometimes required and sometimes optional to run analyses on the Hackystat server.

3.5 More Information about Hackystat

The Hackystat system can be downloaded for use in other software organizations by visiting the Hackystat Developer Services website (<http://www.hackystat.org>). In addition, Hackystat's User Guide, full source code, Java documentation, and other useful information that are required to install Hackystat are obtainable at this website. Any questions and suggestions can be sent to Hackystat Users email mailing list (hackystat-users-1@hawaii.edu).

Chapter 4

Implementing PRI with Hackystat

To successfully use Priority Ranked Inspection, the determination of MINI and LINI must be obtainable for a very low cost. In other words, if the ranking function takes three months, or even 20 hours of management time to generate, then a software project may no longer need those specific recommendations. Therefore, this determination must be obtained in real-time and without introducing new costs to the inspection process.

One way of obtaining PRI rankings in real-time and without developer overhead is through the use of the Hackystat system [23]. Hackystat is a framework for collecting and analyzing software product and development process metrics in real-time ¹. For this research, I have created an extension to Hackystat called the Hackystat Priority Ranked Inspection Extension (hackyPRI for short). This extension provides a real-time PRI ranking. Figure 4.1 demonstrates the use of the PRI ranking function by ranking a software project's workspaces.

This chapter provides a detailed description of the Hackystat PRI (hackyPRI) extension. In Section 6.1, I identify the limitations of implementing the PRI process with Hackystat. Section 4.2 provides a general introduction to the PRI ranking function. Section 4.3 provides a user level description of how to use the hackyPRI extension. Section 4.4 discusses how hackyPRI supports the four steps of the Priority Ranked Inspection process. Section 4.5 provides a developer level explanation of the extension's design and implementation. Section 4.6 discusses the future enhancements that could be added to hackyPRI. Section 4.7 provides a list of contributions that I have added to the Hackystat system as a result of my implementation of hackyPRI. Finally, Section 4.8 provides some information for other organizations to install and use hackyPRI.

¹For more information about the Hackystat system see Chapter 3.

Workspaces (218):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont.	Com-mit	Com-mit	Code Churn	Revi-ew	Last Revi-ew	Open Issue	Closed Issue	File Metric	Test Metric	Test File	Depend	Unit Test Result	Coverage
hackystdx\src\org\hackystat\stdev\project	1179	developer1 (time=17.58, cont=52)	25.33 h	3.42 h	15-Mar-2003	15-Mar-2005	7	108	05-May-2003	17-Mar-2005	6	5202	0	1	LOC=1005, Class=7, Meth=70	LOC=95, Class=2, Meth=5	in=54, out=28	in=21, out=23	Pass=0, Fail=23	78.00 %
hackyPRI\src\org\hackystat\app\api\model\workspace\measures	1148	developer2 (time=25.42, cont=334)	25.42 h	3.25 h	27-Jan-2005	17-Apr-2005	1	334	28-Jan-2005	17-Apr-2005	1	13429	0	1	LOC=4182, Class=48, Meth=312	LOC=890, Class=22, Meth=22	in=90, out=57	in=0, out=10	Pass=0, Fail=36, Error=10	93.12 %
hackyReport\src\org\hackystat\stdev\report	1126	developer3 (time=9.83, cont=873)	11.50 h	0.58 h	15-Jun-2003	10-Nov-2004	2	127	13-Jun-2003	27-Oct-2004	3	5882	0	0	LOC=190, Class=6, Meth=24	LOC=13, Class=1, Meth=1	in=26, out=4	in=0, out=0	Pass=0, Fail=4, Error=0	80.00 %
hackyKernel\src\org\hackystat\kernel\admin	1118	developer1 (time=24.17, cont=597)	39.58 h	4.42 h	04-May-2003	22-Apr-2005	6	145	04-May-2003	22-Apr-2005	5	6421	0	3	LOC=933, Class=11, Meth=20	LOC=86, Class=3, Meth=7	in=174, out=26	in=0, out=16	Pass=0, Fail=22, Error=16	78.79 %
hackystdx\src\org\hackystat\stdev\project\cache	1115	developer4 (time=2.00, cont=8)	28.00 h	4.83 h	25-Feb-2004	15-Mar-2005	6	185	25-Feb-2004	17-Mar-2005	6	9421	0	0	LOC=294, Class=8, Meth=32	LOC=22, Class=1, Meth=2	in=41, out=17	in=0, out=17	Pass=0, Fail=19, Error=123	66.67 %
hackystdx\src\org\hackystat\stdev\activity\analysis\project\activetime	1113	developer4 (time=2.00, cont=8)	5.58 h	2.83 h	09-May-2003	13-Jul-2004	5	25	10-May-2003	09-Nov-2004	6	1470	0	0	LOC=167, Class=2, Meth=5	LOC=22, Class=1, Meth=1	in=0, out=30	in=0, out=30	Pass=0, Fail=46, Error=20	100.00 %
hackyKernel\src\org\hackystat\kernel\cache	1100	developer1 (time=2.33, cont=12)	2.75 h	0.50 h	04-May-2003	10-Dec-2004	3	15	04-May-2003	10-Dec-2004	2	829	0	0	LOC=347, Class=3, Meth=35	LOC=85, Class=1, Meth=4	in=15, out=2	in=15, out=2	Pass=0, Fail=0, Error=2	91.30 %
hackyKernel\src\org\hackystat\kernel\command	1095	developer1 (time=2.09, cont=12)	2.42 h	0.33 h	04-May-2003	24-Sep-2004	2	17	04-May-2003	24-Sep-2004	2	1094	0	0	LOC=481, Class=4, Meth=44	LOC=10, Class=1, Meth=1	in=81, out=11	in=81, out=11	Pass=0, Fail=0, Error=2	88.24 %
hackyKernel\src\org\hackystat\kernel\util	1085	developer4 (time=6.56, cont=54)	17.33 h	7.50 h	04-May-2003	18-Mar-2005	5	112	04-May-2003	24-Mar-2005	5	4537	0	0	LOC=1317, Class=16, Meth=151	LOC=381, Class=10, Meth=22	in=456, out=16	in=456, out=16	Pass=0, Fail=0, Error=163	83.00 %
hackystdx\src\org\hackystat\stdev\activity\analysis\stdev\ddv\view	1072	developer4 (time=6.58, cont=13)	9.17 h	3.08 h	01-Jan-2003	05-Jun-2004	3	27	10-May-2003	25-Jun-2004	3	1681	0	0	LOC=133, Class=2, Meth=3	LOC=25, Class=1, Meth=1	in=0, out=26	in=0, out=26	Pass=0, Fail=106, Error=47	100.00 %
hackystdx\src\org\hackystat\stdev\admin\analysis\serverstatus	1066	developer4 (time=0.67, cont=14)	0.67 h	0.00 h	24-Sep-2004	01-Nov-2004	1	14	24-Sep-2004	03-Nov-2004	1	876	44	0	LOC=311, Class=8, Meth=21	LOC=101, Class=4, Meth=6	in=6, out=22	in=6, out=22	Pass=0, Fail=4, Error=20	83.33 %
hackyKernel\src\org\hackystat\kernel\seno\usermap	1064	developer1 (time=0.59, cont=8)	0.83 h	0.42 h	08-May-2003	21-Apr-2004	4	10	08-May-2003	25-Jun-2004	2	537	0	0	LOC=240, Class=2, Meth=19	LOC=13, Class=1, Meth=1	in=1, out=19	in=1, out=19	Pass=0, Fail=8, Error=5	94.44 %
hackystdx\src\org\hackystat\stdev\daily\analysis\workspace	1064	developer4 (time=0.25, cont=11)	21.17 h	1.25 h	08-Oct-2004	18-Nov-2004	5	48	10-Oct-2004	16-Feb-2005	5	1133	34	0	LOC=293, Class=2, Meth=9	LOC=54, Class=2, Meth=9	in=7, out=10	in=7, out=10	Pass=0, Fail=21, Error=11	40.91 %
hackystdx\src\org\hackystat\stdev\admin\analysis\adoption	1054	developer1 (time=1.83, cont=9)	0.50 h	0.25 h	21-Aug-2003	18-Nov-2004	3	24	05-May-2003	19-Oct-2004	4	1111	0	0	LOC=149, Class=7, Meth=49	LOC=149, Class=7, Meth=49	in=45, out=4	in=45, out=4	Pass=0, Fail=0, Error=2	87.88 %
hackystdx\src\org\hackystat\stdev\common\alert\newdata	1052	developer1 (time=0.25, cont=12)	2.17 h	0.58 h	08-May-2003	26-Oct-2004	2	10	08-May-2003	26-Oct-2004	2	323	0	0	LOC=120, Class=2, Meth=4	LOC=10, Class=1, Meth=1	in=1, out=12	in=1, out=12	Pass=0, Fail=19, Error=6	100.00 %
hackyPRI\src\org\hackystat\app\api\model\workspace	1048	developer2 (time=15.00, cont=67)	15.08 h	0.67 h	26-Jan-2005	17-Apr-2005	2	67	28-Jan-2005	17-Apr-2005	1	2485	19	0	LOC=112, Class=4, Meth=11	LOC=16, Class=1, Meth=1	in=2, out=15	in=2, out=15	Pass=0, Fail=14, Error=23	80.00 %
hackystdx\src\org\hackystat\project\filetime	1048	developer1 (time=0.25, cont=2)	0.33 h	0.17 h	24-Jun-2004	24-Jun-2004	1	4	25-Jun-2004	09-Nov-2004	3	232	0	0	LOC=588, Class=8, Meth=6	LOC=93, Class=4, Meth=4	in=55, out=34	in=55, out=34	Pass=0, Fail=3, Error=0	91.43 %
hackyKernel\src\org\hackystat\kernel\soap	1045	developer1 (time=0.67, cont=11)	1.58 h	0.17 h	04-May-2003	15-Nov-2004	5	17	04-May-2003	08-Mar-2004	2	495	0	0	LOC=202, Class=4, Meth=8	LOC=33, Class=1, Meth=1	in=6, out=11	in=6, out=11	Pass=0, Fail=4, Error=0	63.64 %
hackystdx\src\org\hackystat\stdev\common\analysis\lists\memorata	1044	developer1 (time=1.17, cont=5)	1.17 h	1.17 h	03-Apr-2004	06-Apr-2004	1	6	20-Jan-2004	30-Sep-2004	2	97	0	0	LOC=89, Class=2, Meth=2	LOC=19, Class=1, Meth=1	in=0, out=16	in=0, out=16	Pass=0, Fail=23, Error=36	100.00 %
hackystdx\src\org\hackystat\stdev\common\analysis\daydatasummary	1040	developer1 (time=0.58, cont=9)	0.58 h	0.50 h	11-Jan-2004	09-Apr-2004	1	10	12-Jan-2004	30-Sep-2004	2	129	0	0	LOC=92, Class=2, Meth=5	LOC=18, Class=1, Meth=1	in=0, out=18	in=0, out=18	Pass=0, Fail=18, Error=50	100.00 %

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

4.1 Limitations of Implementing PRI with Hackstat

Certainly, utilizing the real-time and low-developer-overhead collection and analysis features of Hackstat is not perfect for every organization. There are many adoption barriers that come with a PRI process implemented with Hackstat. For example, the barriers could include the initial cost of setting up and using Hackstat to collect process and product measures, the configuration of the ranking function, the devotion required to ensure that Hackstat provides accurate information, privacy issues associated with collecting information on developers, and possibly many more. In addition, although it is yet to be studied, I hypothesize that PRI will not be instantaneously beneficial to an organization. In other words, if an organization installs, configures, and uses hackyPRI for only one week, then I believe the rankings will not be as beneficial as compared to another organization that has collected 3 years of Hackstat data.

I have designed the general theory of the Priority Ranked Inspection process to be independent of any specific means of collection, analysis, and ranking. However, I have chosen to use Hackstat to implement PRI in this research, because the organization that I am studying has already faced and found solutions for the barriers listed above.

4.2 PRI Ranking Function

The PRI ranking function is the most important and most complicated component of hackyPRI. There are two fundamental components in the PRI ranking function that must be understood by both users and developers of hackyPRI. They are the PRI measures and the PRI indicators. The following sections introduce these concepts.

4.2.1 PRI Measures

PRI measures represent the product and process measures that are the basis for determining whether a document is a MINI or a LINI. In addition, they are used to help generate the rankings of documents and help explain the rankings by displaying the measure values. To better illustrate this, see Figure 4.1. This figure presents a PRI ranking obtainable from hackyPRI. The figure contains a listing of the project's workspaces and its associated values collected from the PRI measures. In theory, PRI measures can be associated with any granularity level of a software artifact; for example, documents, packages, workspaces, and modules. Currently, I have only implemented a set of PRI measures that work for workspaces within a project.

Collection of PRI Measures

PRI measures provide the values of product and process measures associated with each document. In the general theory of the Priority Ranked Inspection process, PRI measures can be collected by any means. In hackyPRI, the Hackystat system is used to automate the collection of the PRI measures. In Hackystat, product and process measures are represented by three components. They are the Sensor Data Type, Sensor, and DailyProjectData components. A Sensor Data Type defines the attributes associated with a measure. A Sensor is used to collect the measures and send that information to a Hackystat server. The DailyProjectData provides a project-level representation of the low-level data that was collected by the Sensors. hackyPRI requires the use of these three components; therefore one must have the necessary Hackystat knowledge to successfully implement a PRI measure. In addition to the three Hackystat components, hackyPRI implements another component that represents a PRI measure, which is used within the PRI ranking. Section 4.5 explains the implementation of PRI measures in detail.

Aggregate and Snapshot PRI Measures

There are two different types of PRI measures. They are Aggregate and Snapshot measures. An Aggregate PRI measure is the result of the summation of calculated values obtained from processing one or more days of Hackystat Sensor Data. For example, the Active Time PRI measure is an Aggregate measure because it adds the values of active time over a specified time period. If a developer has generated 1.2, 1.0, and 2.0 hours of active time on three successive days, the Aggregate Active Time measure would return 4.2 hours. Snapshot measures are not aggregated. Instead, they represent the most recent value of a measure from one or more days. For example, the LOC (lines of code) measure does not make sense as an Aggregate measure. If the system size was 1000, 1100, and 1200 on three successive days, then the aggregate of those numbers (3300) is not useful. Therefore, LOC is a Snapshot PRI measure, and in this example 1200 is returned. In hackyPRI, Aggregate and Snapshot measures must be implemented differently. Therefore, the decision of the type of PRI measure (either Aggregate or Snapshot) must be made carefully.

4.2.2 PRI Indicators

PRI indicators use one to many PRI measures to provide a ranking for each document. PRI indicators provide indications of whether documents are MINI or LINI. For example, the Testing PRI indicator provides indications on whether a document has adequate testing. This indicator

accesses the Unit Test, Coverage, and Test Code Active Time PRI measures to evaluate the level of testing for the document.

PRI indicators have the following characteristics. First, PRI indicators determine whether a document is a MINI or a LINI. Second, each PRI indicator can use the data from one or many PRI measures. Third, each PRI indicator returns an integer value between 0 and 100. Fourth, different weights can be assigned to different PRI indicators.

The calibration of the PRI indicators' ranking and weights is a very important step when using hackyPRI. Each individual PRI indicator can and should be calibrated differently to provide a PRI ranking that best represents a MINI and LINI determination for a specific project. The calibration of a PRI indicator includes four steps.

1. **0 to 100 indicator ranking** - Once a PRI measure has gathered and calculated data from the product and process measures obtainable from Hackystat, its values are used within PRI indicators to return a ranking from 0 to 100. A zero ranking indicates the worst possible ranking and a hundred ranking indicates the best possible ranking. For example, in the Coverage (the percentage of code exercised by test cases) PRI indicator, one could imagine that 0 ranking would indicate 0 percent coverage and a 100 ranking would be reserved for 100 percent coverage. To calculate a ranking, certain thresholds must be identified that are specific for each PRI indicator. This is hard for some indicators and easy for others. In the Coverage PRI indicator, this is relatively straight forward, because the indicator relies on a single PRI measure and the coverage percentage is already on a 0-100 scale. On the other hand, the thresholds for the Testing PRI Indicator are less clear. The Testing PRI Indicator uses three different measures, the PRI Unit Test measure, the PRI Coverage measure, and the PRI Active Time measure. In addition, the values of these measures, when combined, do not fit nicely into a 0-100 scale. Currently, calculation of the indicator rankings is implemented with Java code within the hackyPRI system.
2. **Weights are assigned to each indicator** - It is possible that each PRI indicator affects the ranking differently. Therefore, each indicator can be given a different weight to reflect that difference. For example, an organization may find that coverage is the leading indicator of MINI documents and can weight coverage higher than any other PRI indicator. Weights can range from 0 to any integer. If an indicator has a weighting of 0, then this indicator will be "disabled" from the ranking function. These weighting are configurable through the hackyPRI

user interface for each project within Hackystat and all indicators are defaulted to a weighting of 1 in the initial configuration of a Project PRI Ranking.

3. **Compute aggregate ranking for all indicators** - Once the independent ranking and weights are in place, the system will automatically compute an aggregate ranking per document. A very simple example is the following: a document has these indicator rankings {92, 100, 30, 15} and these respective indicator weighting {1, 1, 1, 2}. The aggregate ranking of this document would be $(92*1) + (100*1) + (30*1) + (15*2) = 252$.
4. **MINI and LINI declaration** - The final step of this process is the declaration of MINI and LINI for each document based on the aggregate ranking. Currently, I have not implemented the facilities to make a declaration of MINI and LINI, because this has proven to be more difficult than I first envisioned. I will leave this as a future implementation and research task. However, the current system does provide a 'relative' ranking: a workspace with a higher PRI ranking is "more MINI" than all workspaces with a lower PRI ranking. In addition, Hackystat projects generally have hundreds, if not thousands, of different workspaces, therefore a relative ranking is sufficient for this current research.

4.3 User Interface

This section contains a description of important hackyPRI user interface components. It should be noted that, like most Hackystat user interface components, the interface components associated with hackyPRI have not been thoroughly tested for usability problems using traditional Human Computer Interaction evaluation techniques. However, the developers of Hackystat have informally reviewed the user interface.

Each description of the hackyPRI user interface components in this section will be accompanied by a screenshot.

4.3.1 PRI Analyses

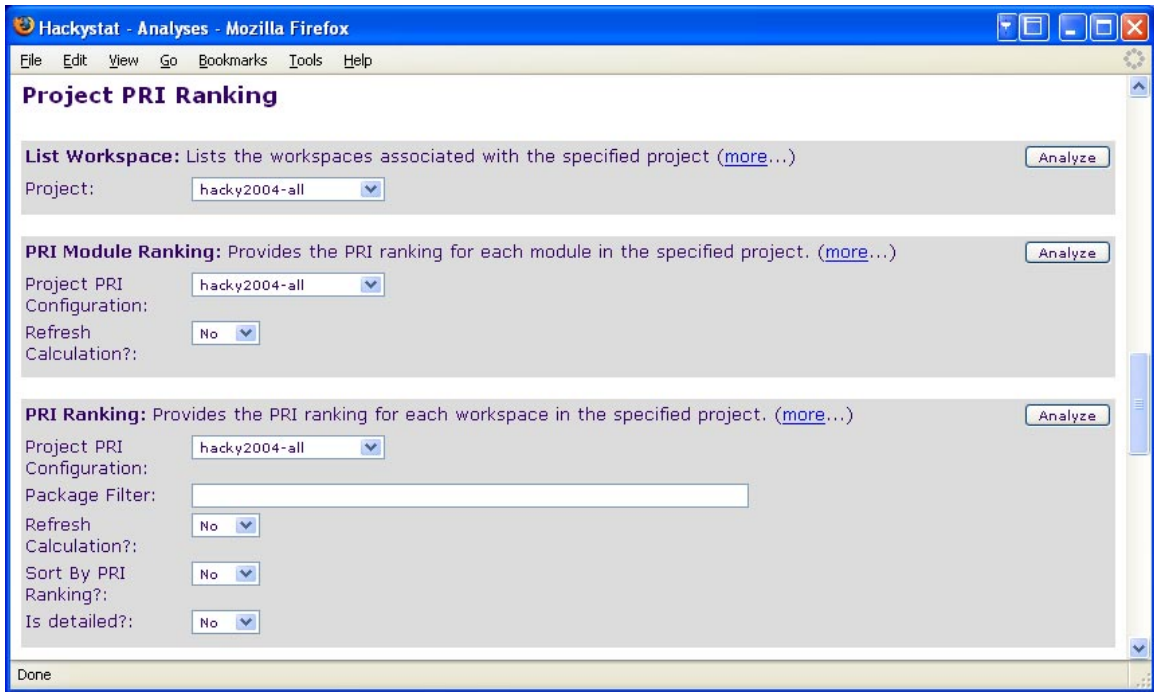


Figure 4.2. Presents the analyses that are provided by the hackyPRI system.

This screenshot shows the Hackystat analyses that the hackyPRI system provides. The List Workspace analysis, the first Hackystat analysis in the screenshot, allows users to validate the set of workspaces associated with a specific project. Because, the PRI ranking provides rankings for workspaces it is important to ensure that all workspaces within your project are used in the PRI ranking. Therefore, it is very important that all the project's workspaces, including child workspaces, are identified by this analysis.

The Project PRI Module Ranking analysis, the second Hackystat analysis in the screenshot, displays the PRI ranking for top-level workspaces, or modules. This analysis ranks the modules by the average PRI rankings for all workspaces within the modules.

The Project PRI Ranking analysis, the third Hackystat analysis in the screenshot, provides the PRI ranking for workspaces within the selected Project. The execution of this analysis results in a table that ranks workspaces within the project according to the PRI ranking function described in Section 4.2.2. Before using this analysis, you must configure the PRI ranking, see Section 4.3.3.

4.3.2 PRI List Workspace Analysis

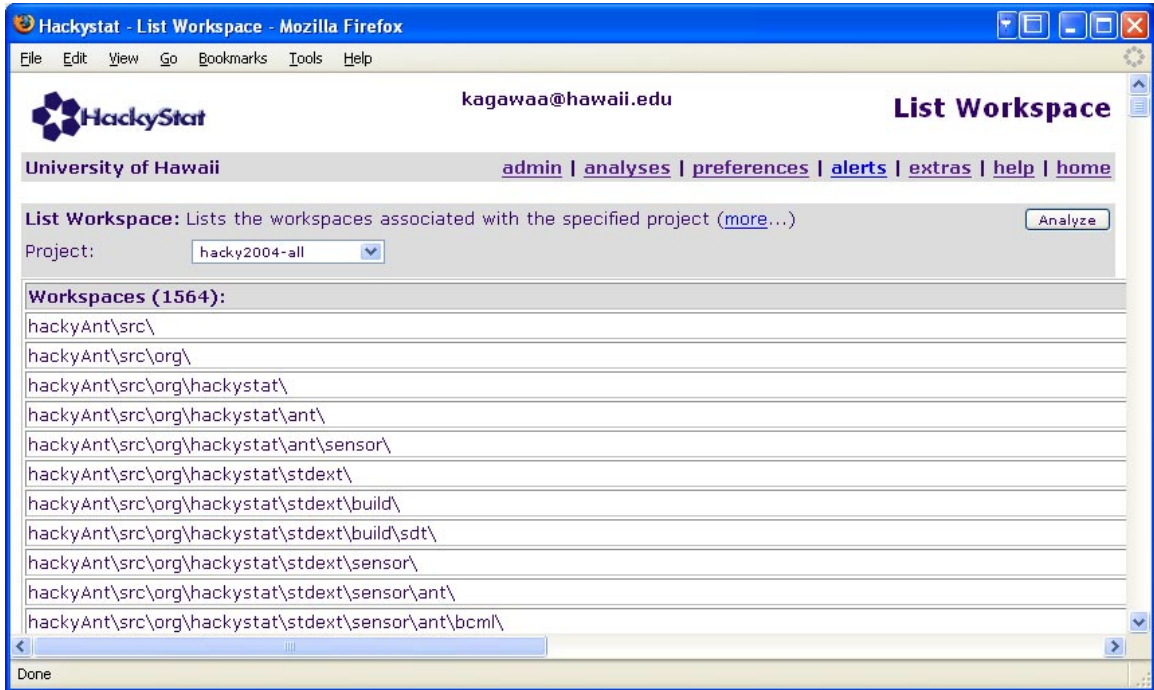


Figure 4.3. Presents an example execution of the List Workspace analysis.

This screenshot shows the result of the Hackystat List Workspace analysis. This analysis allows users to validate the set of workspaces associated with a specific project. Because, the PRI ranking provides rankings for workspaces it is important to ensure that all workspaces within your project are identified.

This screenshot provides a table of all workspaces within a project. In this particular execution, there are 1564 workspaces associated with the hacky2004-all Project. These are the workspaces that will be ranked by the PRI Ranking analysis. However, due to configurations that specify the programming language used in the hacky2004-all project, workspaces without Java software code are ignored from the ranking. Therefore, most of the 1,564 workspaces will not be ranked.

4.3.3 Project PRI Configuration Management

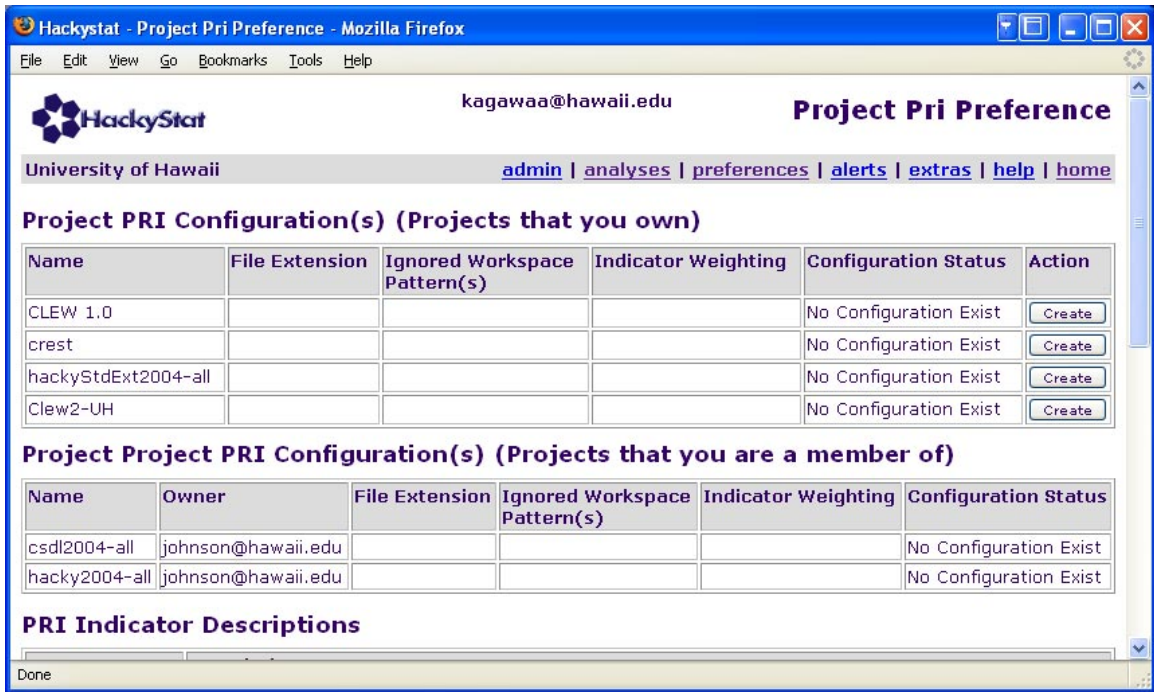


Figure 4.4. The Preference page that presents the Project PRI Configuration management.

This screenshot presents the Project PRI Configuration Management page. The configuration management is accessible on the Hackystat preferences page, shown in the screenshot in Figure 3.5. All software projects are different; therefore the purpose of the configuration management page is to configure the PRI ranking function for a specific project.

There are two different sets of configurations shown in this screenshot. The first table shows the PRI configurations associated with Hackystat Projects that you own. The second table shows PRI configurations associated with projects that you are a member of. The configuration management only allows users to create PRI configurations for the projects that they own. For example, I must contact johnson@hawaii.edu, the Project Owner for hacky2004-all, to create a PRI configuration for the hacky2004-all project. Once a Project PRI configuration is created, then the Project Owner can modify and delete the configuration.

4.3.4 Create a Project PRI Configuration

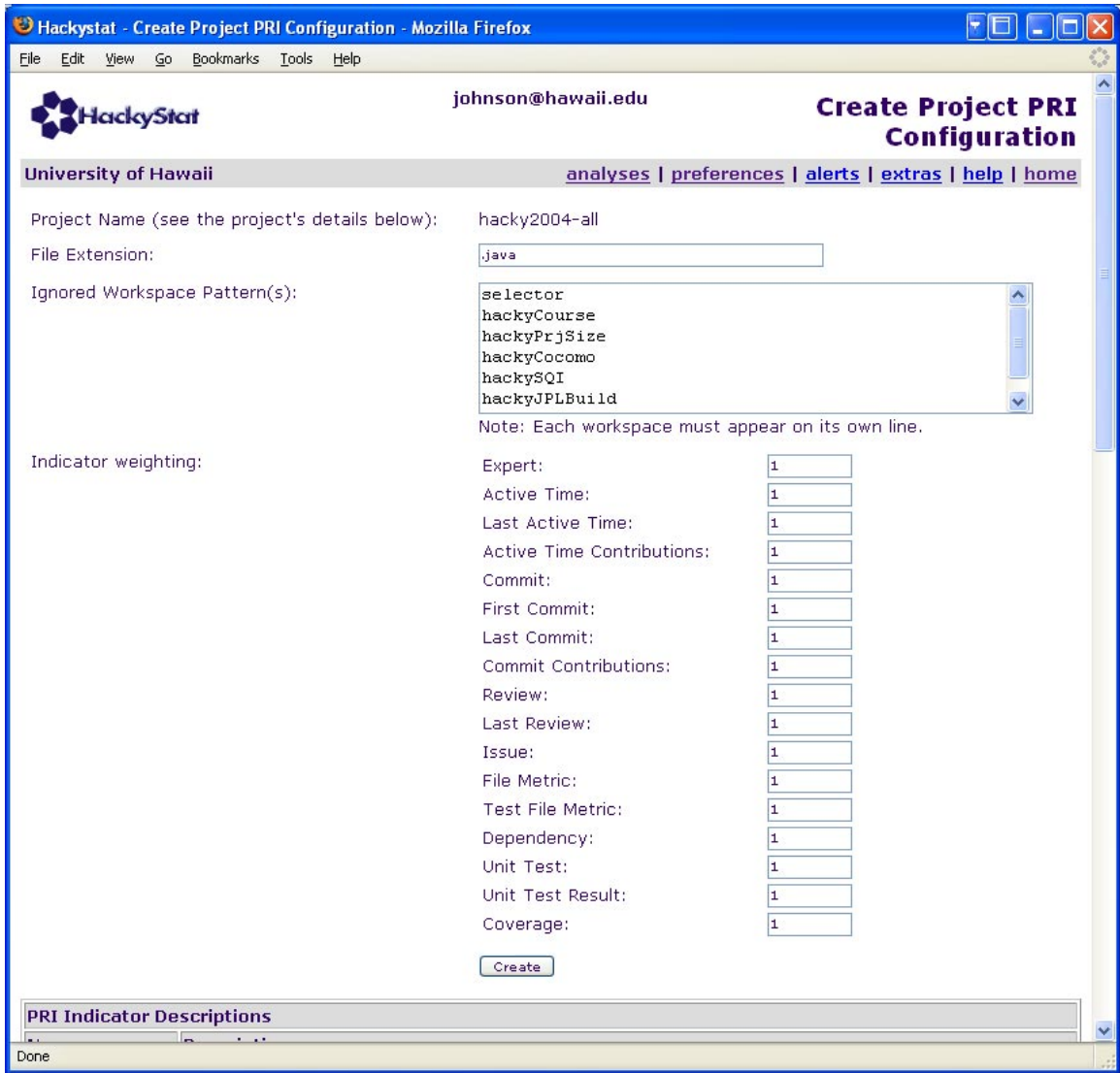


Figure 4.5. The Create a PRI Configuration page.

In the previous section, I explained that only a Project Owner has the authority to create a PRI configuration. Therefore, in this example, I've contacted the project owner and requested the creation of the PRI configuration for the hacky2004-all Project. This screenshot presents the Create Project PRI Configuration webpage. To create a configuration, we must provide the following information:

1. **File Extension** - this setting allows the specification of the type of programming language that is used in the PRI Ranking for this Project. Valid entries are file extensions that contain a “.” followed by any number of characters. For example, a valid entry could be “.java”, “.cpp”, “.html”, or whatever programming language file extension is associated with your project. There are some problems associated with this setting. First, this setting does not support Projects that are implemented with more than one programming language. Second, the Hackystat product and process measures implemented within Hackystat best support the Java Programming language. Other programming languages, like C and C++, are supported but not to the extent of Java. Furthermore, I did not test the generation of PRI racking for other programming languages other than Java. Of course, the hackyPRI and Hackystat systems can be fairly easy to extend to fully support any programming language. However, for this research I will leave these issues as a future enhancement.

2. **Ignored Workspace Pattern(s)** - this setting allows the specification of workspaces that should be ignored in the PRI ranking. Simply put, I have found that some workspaces in a Project do not need to be inspected. This can happen for a couple of reasons. First, some workspaces contain code that is no longer released. Second, some code, for example, automatically generated code, could be ignored. This is a debatable use of the ignored workspace pattern, but it is provided to the user as an option.

In this screenshot, the “hackyCourse”, “hackyPrjSize”, “hackyCocomo”, “hackySQI”, and “hackyJPLBuild” are parent workspaces that should be ignored. The configuration will also ignore any child workspace under those parent workspaces. The “selector” ignored workspace pattern, specifies that any workspaces that contains the string “selector” shall also be ignored. For the hacky2004-all project, selector code is an anomaly that generally does not need to be inspected.

3. **Indicator Weighting** - this setting allows the specification of the weights associated with the PRI indicators. By default, all PRI indicator weights are set to 1. Any positive integer, including zero, are valid weights. The indicator weights are used to calibrate the individual PRI indicator rankings, Section 4.2.2 explains this in detail. If a weight is set to zero, then the PRI indicator will not affect the PRI ranking. If an indicator’s weight is set to 2, and all other weighting remain at 1, then the indicator will have twice the significance as the other indicators. Indicator weighting is useful to correctly configure the indicators’ importance in

the PRI ranking. For example, if Coverage is a leading factor in the PRI ranking, then it should be weighted higher than the other indicators.

4.3.5 Project PRI Configuration Management - After the Creation of a PRI Configuration

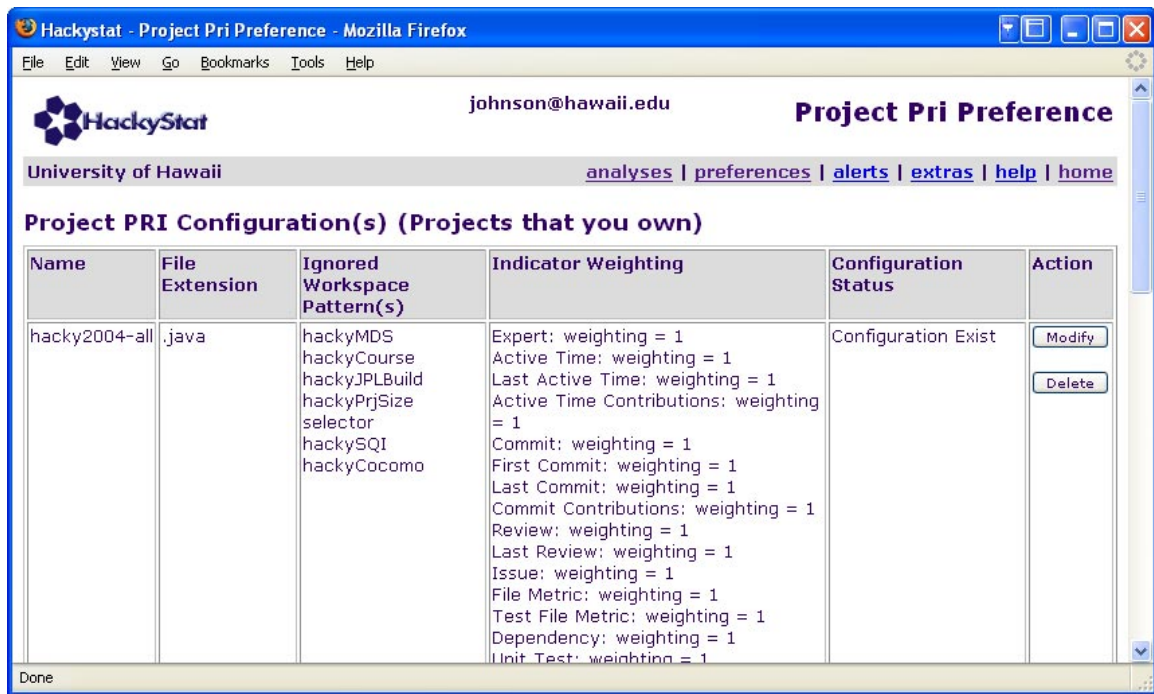


Figure 4.6. The Preference page that presents the Project PRI Configuration management.

This screenshot presents the Project PRI configuration management webpage after the creation of a PRI configuration. Provided in the webpage are the details of the configurations that have been created. In addition, the Project Owner can modify and delete the configuration.

4.3.6 Project PRI Ranking Analysis

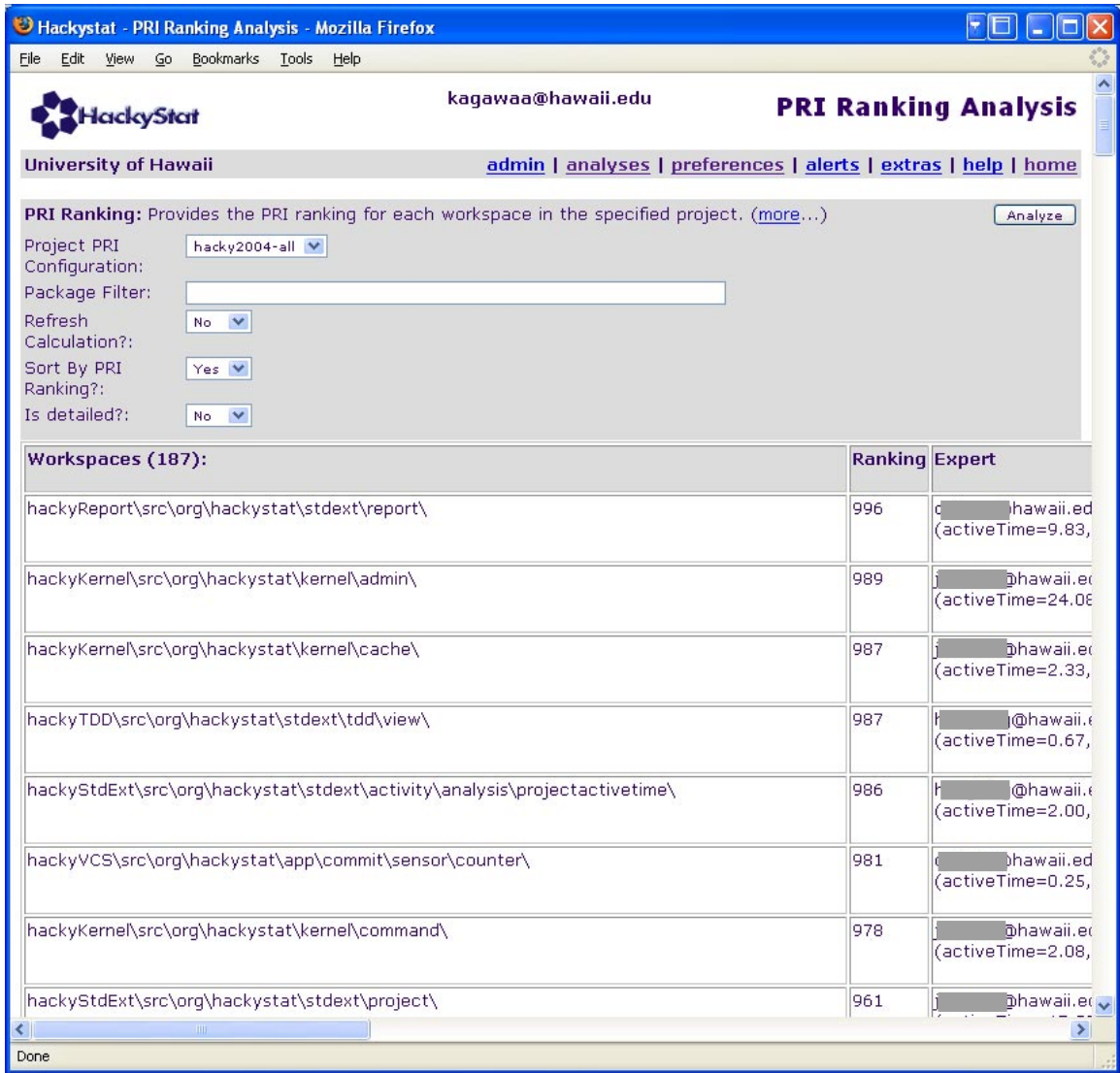


Figure 4.7. Presents an example execution of the Project PRI Ranking analysis.

After a Project PRI configuration has been successfully created and configured in the previous sections, then you can run the PRI Ranking analysis. The analysis is obtainable on the analyses page (Figure 3.4) in the Project PRI Ranking section (Figure 4.2).

This screenshot presents the rankings of the hacky2004-all project. This project contains the product and process measures that have been collect during the development of the Hackystat system. I will be using this Hackystat project to evaluate the hackyPRI system.

Due to page width constraints, the screenshot does not show the values of the numerous PRI measures and all the workspaces available. Normally, in this table you will be able to view the workspace, the aggregate ranking, and all the values of each of the PRI measures. This particular execution of the analysis ranks each workspace by its associated aggregate PRI ranking. It sorts the highest ranked workspaces (LINI) to the top of the table and the lowest ranked workspaces (MINI) to the bottom of the table.

4.3.7 Project PRI Ranking Analysis Selectors

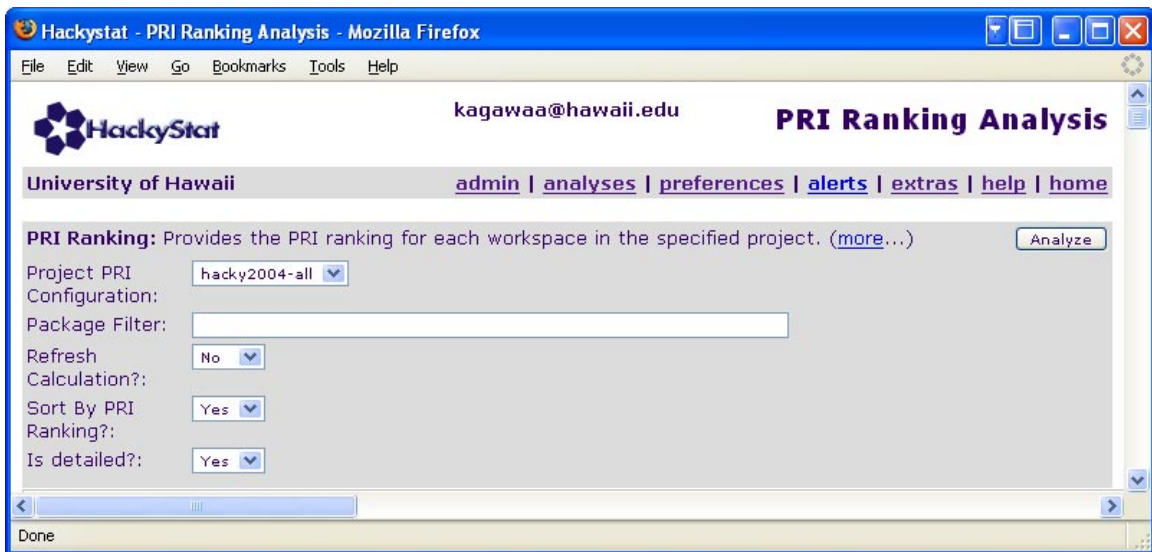


Figure 4.8. Presents the selectors that are available in the Project PRI Ranking analysis.

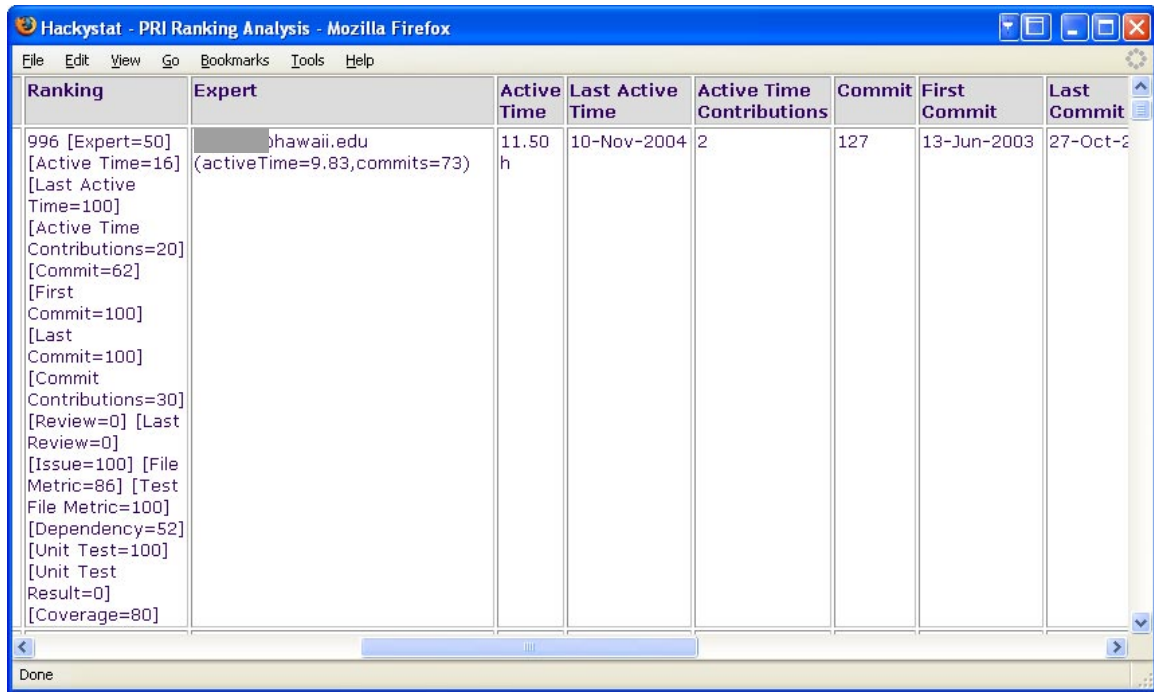
This screenshot presents the user interface for the PRI Ranking analysis. There are five different selectors that users can set to determine the PRI Ranking that they would like to generate.

1. **Project PRI Configuration** - users should use this selector to select what PRI configuration they would like to be used in the PRI ranking. A PRI configuration has a one-to-one correspondence with Hackystat Projects, thus when you select a PRI configuration you are also selecting what Project data to use in the PRI ranking. PRI configurations must be created using the PRI Configuration Management preference interface before a PRI ranking can be generated (See Section 4.3.3 and 4.3.4).
2. **Package Filter** - users should use this selector to show workspaces that contain a value of the string they enter into the textbox. For example, if a user wants to generate the PRI ranking

only for the hackyKernel module, then they should enter in “hackyKernel” into the Package Filter selector. Section 4.3.9 contains a screenshot of an example use of this selector. In addition, if a user wants to rank only analysis code, then the user can enter in “analysis” and all workspaces that contain that value will be ranked.

3. **Refresh Calculation?** - users should use this selector to either re-generate the PRI ranking or use the last PRI ranking stored in the system. This is an unusual and unique behavior for the standard set of analyses provided by Hackystat. However, since the PRI ranking takes an unusual amount of time to generate, in some instances two to five minutes, users will want to be able to play with the selectors without having to wait for the re-generation of the ranking. For example, users could sort by ranking, use the Package Filter, or see the detailed view, without wanting to re-generate the PRI ranking.
4. **Sort By PRI Ranking?** - users should use this selector to sort the workspaces by PRI ranking or by the workspaces. Selecting “Yes,” will sort the highest ranked (LINI) workspaces to the top of the table and sort the lowest ranked (MINI) workspaces to the bottom of the table. Selecting “No,” will sort the table by workspaces, which groups all similar workspaces together.
5. **Is Detailed?** - users should use this selector to view detailed information about the PRI ranking. Selecting “Yes,” will provide the weighted-rank for each PRI indicator. This will hopefully aid the indicator weighting configuration process. An example screenshot of the detailed view is shown in Section 4.3.8. Selecting “No,” will provide the standard output seen in Section 4.3.6.

4.3.8 Project PRI Ranking Analysis - Detailed View

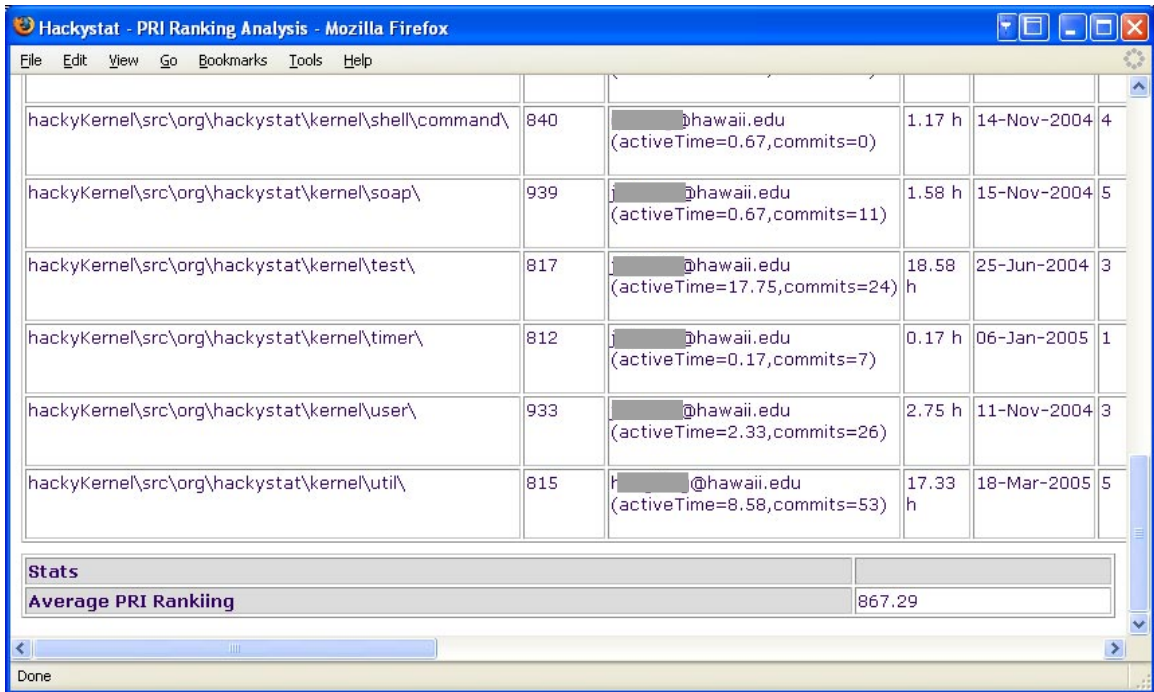


Ranking	Expert	Active Time	Last Active Time	Active Time Contributions	Commit	First Commit	Last Commit
996 [Expert=50] [Active Time=16] [Last Active Time=100] [Active Time Contributions=20] [Commit=62] [First Commit=100] [Last Commit=100] [Commit Contributions=30] [Review=0] [Last Review=0] [Issue=100] [File Metric=86] [Test File Metric=100] [Dependency=52] [Unit Test=100] [Unit Test Result=0] [Coverage=80]	hawaii.edu (activeTime=9.83,commits=73)	11.50 h	10-Nov-2004	2	127	13-Jun-2003	27-Oct-2

Figure 4.9. Presents an example execution of the Project PRI Ranking analysis.

This screenshot presents the results of a Project PRI Ranking with the “Is Detailed?” selector set to “Yes”. The detailed view shows information that is not available in the regular view and it is intended to help the user understand how the rankings are computed, what workspaces are ignored, and what workspaces do not have Java implementation. What is shown in this screenshot is the detailed information about the PRI indicator ranking (see Section 4.2.2 for a description of PRI indicators). This information will hopefully help the Project Owner in the configuration of the PRI indicator weighting (discussed in Section 4.3.3).

4.3.9 Project PRI Ranking Analysis - hackyKernel



Workspace Path	PRI Ranking	Active Time (h)	Commits	Date	Count
hackyKernel\src\org\hackystat\kernel\shell\command\	840	1.17 h	(activeTime=0.67,commits=0)	14-Nov-2004	4
hackyKernel\src\org\hackystat\kernel\soap\	939	1.58 h	(activeTime=0.67,commits=11)	15-Nov-2004	5
hackyKernel\src\org\hackystat\kernel\test\	817	18.58 h	(activeTime=17.75,commits=24)	25-Jun-2004	3
hackyKernel\src\org\hackystat\kernel\timer\	812	0.17 h	(activeTime=0.17,commits=7)	06-Jan-2005	1
hackyKernel\src\org\hackystat\kernel\user\	933	2.75 h	(activeTime=2.33,commits=26)	11-Nov-2004	3
hackyKernel\src\org\hackystat\kernel\util\	815	17.33 h	(activeTime=8.58,commits=53)	18-Mar-2005	5

Stats	
Average PRI Ranking	867.29

Figure 4.10. Presents an example execution of the Project PRI Ranking analysis.

This screenshot presents the results of the Project PRI Ranking analysis with the “Package Filter” selector set to “hackyKernel”. The most interesting use of the Package Filter is the Average PRI Ranking information obtainable at the bottom of the webpage. The Average PRI Ranking provides the average ranking of all workspaces shown in the table. For hackyKernel workspaces the average ranking is 867.29. The ranking of other workspaces will differ. In my research, I will be addressing the PRI Rankings of workspaces, however the PRI rankings can be attainable for whole modules as well.

4.3.10 Project PRI Module Ranking Analysis

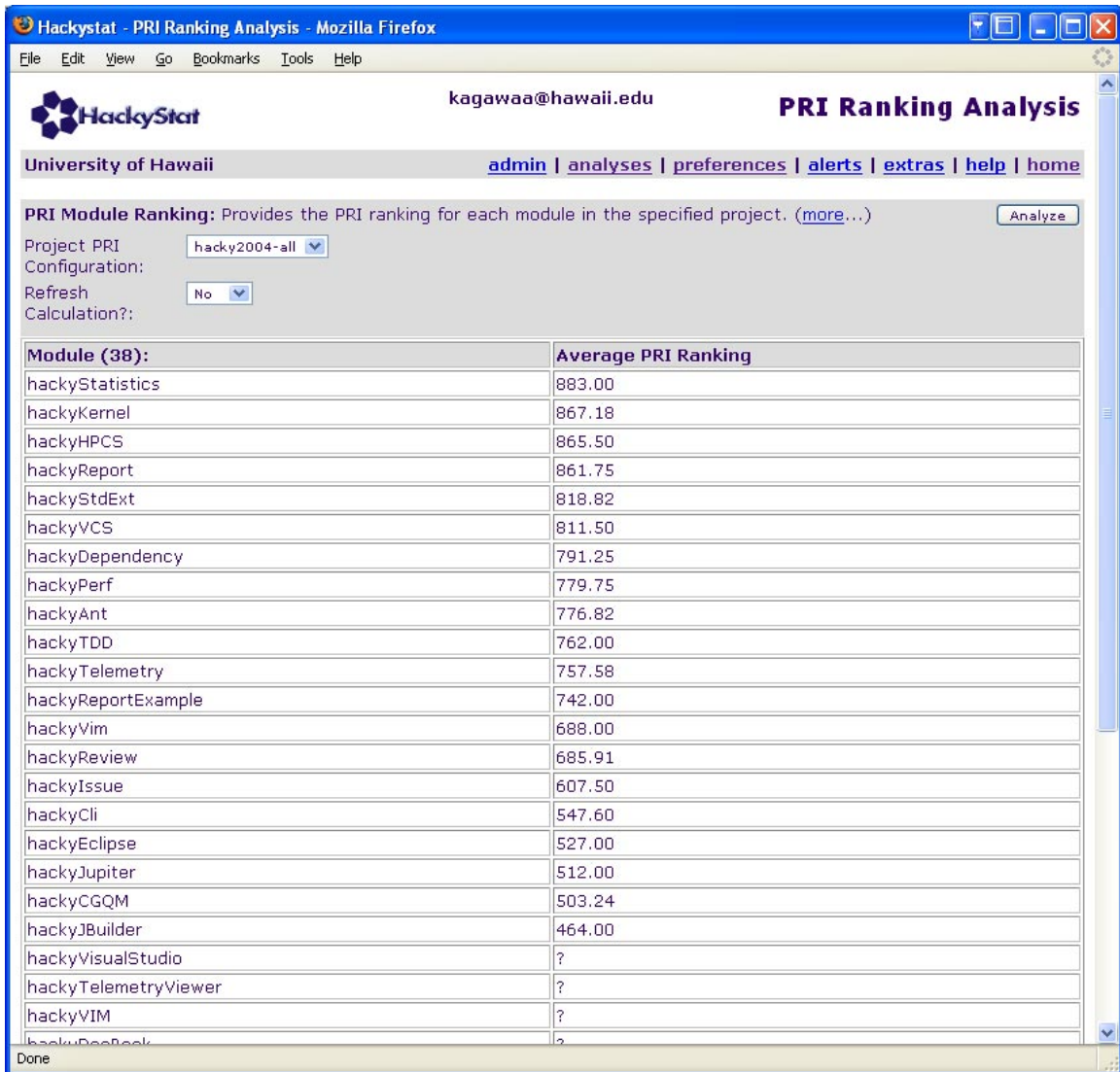


Figure 4.11. Presents an example execution of the Project PRI Module Ranking analysis.

The concept of average rankings, shown in Section 4.3.9, received a lot of interest within CSDL. Therefore, I implemented another Hackystat analysis that provides the average rankings for all top-level workspaces, also referred to as modules. This screenshot presents the results of the Project PRI Module Ranking analysis.

Currently, the hackyPRI system cannot generate a true PRI ranking for modules. Instead, the hackyPRI system can only rank workspaces. However, I was able to implement a facility

that uses the average ranking feature shown in Section 4.3.8. In the future, I hope to support all granularity-levels of PRI rankings, for example, modules, packages, single documents, and even methods.

The current results of this analysis are quite interesting. First, the ranking order seems to be congruent with the kernelized architecture of Hackystat. Hackystat is comprised of many different modules that are built in a kernelized fashion, meaning that there are base modules and modules that extend the functionality of the base modules. For example, `hackyKernel` is a base module that almost all other modules utilize. In this screenshot, the main base modules are ranked the highest: `hackyKernel`, `hackyStatistics`, `hackyReport`, and `hackyStdExt`. This is a positive finding. One would hope that the closer we move to the kernel of the system the higher the code quality becomes, because the base modules are the most used and most important modules.

There are a few missing rankings, indicated by a “?”. Some of these modules are ignored via the ignore workspace pattern, some of them have no FileMetric data so the system can’t tell if they no longer exist, and some of them contain no Java files. For example, the `hackyVisualStudio` and `hackyVIM` modules presented in the screenshot do not contain any Java files. Therefore, the current `hackyPRI` implementation does not know how to generate a PRI ranking for these modules. This is also interesting. Should Java code be ranked differently from other programming languages? Can they even be compared? I will address these questions in the future. For this current research, I have only focused on generating PRI rankings for Java code.

4.4 The Four Steps of the Priority Ranked Inspection Process

Hackstat 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 ranking function, which distinguishes MINI documents from LINI documents. The ranking function design includes three steps:
 - (a) Selection of product and process measures to use in the PRI ranking function.
 - (b) The calibration of PRI indicators, which evaluates the values of the measures to generate a ranking for the documents.
 - (c) The creation of a MINI-threshold, which declares all documents above the threshold as LINI and all below as MINI.
2. The selection of a document for inspection, based on the PRI ranking function.
3. The actual inspection of the selected document.
4. Adjustment of product and process measure selection and calibration of PRI indicators based on the results of the inspection.

The following subsections detail how hackyPRI supports the four steps of the Priority Ranked Inspection process.

4.4.1 Step 1a: Selection of Product and Process Measures

Step 1 of the Priority Ranked Inspection process states that the creation of a PRI ranking function will distinguish MINI documents from LINI documents. Step 1a concentrates on the selection of the PRI measures, which provide the data for the PRI ranking function. This selection process will not be the same for all software projects. Therefore, different software groups must be able to add new product and process measures to their own Hackstat installation.

A PRI measure is implemented with a `WorkspacePriMeasure`, explained in Section 4.5.1, and the following Hackstat-related components: a `Sensor Data Type`, a `Sensor`, and a `DailyProject-Data` representation. The Hackstat system provides a set of various product and process measures and I will utilize a subset of the available measures to create the PRI measures. Table 4.3 contains a description of the PRI measures that are implemented in hackyPRI. Each measure is collected for each workspace within a specified project.

The next two sections provide a detailed illustration of how to add and remove PRI measures to and from the system.

Adding a new PRI measure to the system

This section provides a detailed description of the required steps to add a new hypothetical Runtime Execution PRI measure to the system. This measure represents the total number of runtime executions for a specific piece of code during the span of 24 hours. Although this measure is currently not obtainable in the current set of Hackystat measures, it has many practical applications. For example, a Hackystat analysis can map out the areas of a project that are executed most often during normal usage. This would be a great measure to incorporate into PRI, because one would assume that if classes in package Foo are executed ten times more often than classes in package Bar, all other measures being equal, then package Foo could have a higher MINI ranking than package Bar.

Step 1 - Create a Runtime Execution Sensor Data Type All Hackystat measures are concretely defined in a Sensor Data Type. This representation specifies the exact information that is required to allow useful, interesting, and correct interpretations of the measure. Essentially, it is the schema that defines the data. Therefore, the first step is to define the attributes of a Runtime Execution Sensor Data Type.

Step 2 - Create a Hackystat Runtime Execution Sensor Like all Hackystat measures, there must be some way of collecting the Runtime Execution measure. Utilizing the Java Management Extension (JMX) is one of the many possibilities for creating a Runtime Execution sensor. In any case, imagine such a software tool exist such that a Hackystat sensor can extract the necessary information required by the Runtime Execution Sensor Data Type. Once the Sensor and Sensor Data Type have been implemented, Runtime Execution data can be sent to a Hackystat server.

Step 3 - Create a DailyProjectRuntimeExecution representation After the completion of steps 1 and 2, we should have Runtime Execution Sensor Data stored in Hackystat. This fine-grained data is meaningless if we are unable to associate the data to a specific Hackystat project. At this point, the creation of the DailyProjectRuntimeExecution representation is needed. The purpose of this representation is to provide coarse-grained information about Runtime Execution data at the project

level. For example, the number of executions during June 14, 2005 for the package Foo in project Baz.

Step 4 - Create a WorkspacePriRuntimeExecution class Up until this point we have not implemented a PRI measure. Instead, we have been implementing various Hackstat-related components that the PRI measure requires. Now we are ready to create the WorkspacePriRuntimeExecution class, which is the hackyPRI Java representation of the Runtime Execution PRI measure. During the implementation of this class, a critical question must be answered; should this measure be an Aggregate or Snapshot measure? In other words, should the executions be aggregated over time or should the number of executions be obtained from the last set of Runtime Execution Hackstat sensor data. This decision is debatable.

Step 5 - Add the WorkspacePriRuntimeExecution class to the WorkspacePriMeasureClass-Info class This step requires the addition of two lines of code. Simply add an instance of the WorkspacePriRuntimeExecution to the collection of PRI measures that are used in the PRI ranking. In addition, you must add the instance to the collection that creates the presentation of the PRI Ranking. This process can be streamlined in the future with the addition of configurable XML files that define PRI measures and its presentation components. However, I will leave this to a future implementation task.

We are done! After finishing these five steps you have successfully added a new product measure to the PRI ranking function. You should now move on to Step 1b in the four step Priority Ranked Inspection Process (Section 4.4.2) to create and calibrate a PRI indicator that uses this PRI measure within the PRI ranking.

Removing a PRI measure from the system

This section provides a detailed description of the required steps to remove a PRI measure from the system.

Step 1 - Remove the WorkspacePriRuntimeExecution class from the WorkspacePriMeasure-ClassInfo class In normal situations, removing a measure from the PRI ranking is as simple as not calculating the measure. To do this, simply comment out the lines added in Step 5 of the instructions to add a PRI measure to the system. In the future, this process could become much more

user friendly and robust by creating a user interface configuration of PRI measures. The goal of this enhancement would allow the addition and removal of PRI measures at runtime and not require any developer programming to make the change.

We are done! After finishing this one step you have successfully removed a PRI measure from the PRI ranking function. Of course, one could delete all Java implementation of the PRI measure you want to remove. This is quite simple but not suggested, because you might want that PRI measure in a future PRI ranking.

4.4.2 Step 1b: Calibration of PRI Indicators

The PRI ranking function, which is comprised of PRI measures and PRI indicators are implemented in the hackyPRI extension. The process of determining the rankings is not shown in Figure 4.1, however the calibration and ranking function works behind the scenes.

To make the important distinction of MINI and LINI involves a multi-step process. The first step is to identify thresholds to generate a 0 to 100 indicator ranking. The second step is assigning weights for each indicator, which is configurable for each Project through the hackyPRI user interface. See Section 4.2.2 for a detailed explanation of the PRI indicators.

The next three sections provide a detailed illustration of how to add, remove, and calibrate PRI indicators.

Adding a new PRI indicator to the system

This section provides a detailed description of the required steps to add a new hypothetical Runtime Execution PRI indicator to the system. In addition, I'll describe how to utilize a PRI measure within a PRI indicator to create a ranking and weighting. Recall that PRI indicators use the values of one or more PRI measures to calculate a ranking. However, in this particular example, lets assume that the Runtime Execution PRI measure is the only measure that will be used in the Runtime Execution PRI indicator.

Step 1 - Ensure that all PRI Measures Are Implemented PRI indicators are built on top of the PRI measures. Therefore, you must ensure that the PRI measures that you intend to use are completely implemented and working in the system. If not, you must revert to Step 1a in the Priority Ranked Inspection process. For this example, lets assume that you have already created the Runtime Execution PRI measure.

Step 2 - Create the RuntimeExecutionPriIndicator Since PRI indicators are built on top of PRI measures, the majority of the work is already provided by the measures. The job of a PRI indicator is to obtain the PRI measure values and provide a ranking according to certain thresholds. All indicators must return a ranking with values from 0 to a 100. The implementation of this ranking is the most critical and important part of a PRI indicator.

Step 3 - Add the RuntimeExecutionPriIndicator to the PriIndicatorClassInfo class This step requires the addition of two lines of code. Simply add an instance of the PriRuntimeExecutionPriIndicator to the collection of PRI indicators that are used in the PRI ranking. In addition, you must add the instance to the collection that creates the presentation of the PRI Ranking. This process can be streamlined in the future with the addition of configurable XML files that define PRI indicator and its presentation components. However, I will leave this to a future implementation task.

We are done! After finishing these three steps, you have successfully added a new PRI indicator to the system. You should now ensure the calibration of the PRI indicator is correct.

Removing a PRI Indicator from the system

This section provides a detailed description of the required steps to remove an indicator from the system.

Step 1 - Change the PRI Indicator Weighting in the Project PRI Configuration to Zero In normal situations, to remove a PRI indicator from the PRI Ranking you simply need to set that indicator's weighting to zero. See Section 4.3.3 and Section 4.3.4 for more information about changing the PRI configuration for a Project.

We are done! After finishing this one step, you have successfully removed a PRI indicator from the PRI ranking function. Of course, one could delete all Java implementation of the PRI indicator you want to remove. This is quite simple, but not suggested, because you might want that PRI indicator in a future PRI ranking.

Calibrating a PRI Indicator

This section provides a detailed description of the required steps to calibrate the Runtime Execution PRI indicator. This indicator has a one-to-one correspondence with the Runtime Exe-

cution PRI measure. However, other indicators could have a one-to-many relationship with PRI measures.

The Runtime Execution PRI indicator would be a great indicator to incorporate into PRI, because one would assume that if package foo is executed ten times more often than package bar, all other measures equal, then package foo could have a higher MINI ranking than package bar.

Step 1 - Take an initial guess In the previous paragraph, I hinted at an initial guess of one possible calibration of the Runtime Execution PRI indicator. This step requires that the creation of a calibration based on either an initial guess or even hard evidence. The calibration of the Runtime Execution PRI indicator requires the definition of certain thresholds. For example, 50 executions could represent the threshold for a high ranking and 10 executions could represent the threshold for a low ranking. Each PRI indicator returns a value of 0 to 100 to represent low and high rankings.

Step 2 - Run the PRI analysis and analyze the results Do not spend a great deal of time contemplating the definition of the thresholds in Step 1, because a calibration is useless unless you have concrete data. Therefore, Step 2 requires that you run the PRI ranking on a real software project to analyze the results of your initial calibration. The amount of effort that is put into validating the calibration of the indicators will directly affect the effectiveness of the PRI ranking. To thoroughly calibrate a PRI indicator, one must conduct inspections on a sample of the rankings to determine its validity.

Step 3 - Monitor inspection results over time Software products and development processes evolve over time; therefore the calibration of the PRI indicators that represent them must evolve as well. Monitoring the inspection results and comparing them to the calibration of PRI indicators is a continuous requirement. For example, if you find that Runtime Execution information does not have as much of an affect as previously determined, then the calibration must be adjusted.

We are done! After finishing these three steps you have successfully calibrated a PRI indicator.

4.4.3 Step 1c: Declaring MINI and LINI documents

As previously mentioned in Section 1.2, creating a mechanism to declare a document as MINI and LINI has not been solved. Therefore, hackyPRI implements Steps 1a and 1b to provide a priority ranking of the documents. Although, the declaration of MINI and LINI is very important, I

believe that my current research is not limited by my inability to solve this problem. Simply because in a Hackstat project of hundreds of documents, it is possible to claim that a few documents with the lowest rankings are more MINI than a few documents with the highest rankings.

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

Using the Hackstat PRI Ranking analysis (Figure 4.7), 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 MINI document, or the document that is at the very bottom of the table, 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 automatically. In other words, it is more useful to consider a few MINI documents and take an educated guess as to which document needs inspection more.

4.4.5 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. However, in my study of this research, I will keep all PRI information a secret.

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

4.4.6 Step 4: Adjustment of the Measure Selection and Indicator Calibration

If a document is shown to be incorrectly ranked, then an adjustment of the PRI ranking function is necessary. In hackyPRI, this can be accomplished by adding more PRI measures (Step 1a) or recalibrating the PRI indicator (Step 1b).

4.5 Hackstat Priority Ranked Inspection Extension

This section describes the design and implementation of the Hackstat PRI Extension (hackyPRI). My development of the hackyPRI system has been an ongoing process with many different revisions and enhancements. It has taken me three major evolutions to obtain the level of functionality described in the previous sections. The software has gone from a very simple and very slow-processing system, to an optimized and robust system, and then finally to configurable system that can support different projects and organizations.

This section provides a detailed description of the design and implementation of the system. Knowing this low level information provides relatively little advantage in respect to actually using hackyPRI. However, the design and implementation is presented in detail to provide future developers with the necessary information to continue the work that I have started.

The hackyPRI system is written entirely with Java technologies and is fully compatible with the Hackstat system. The system is currently installed and running on the Collaborative Software Development Laboratory's Public Hackstat Server (<http://hackstat.ics.hawaii.edu>). The source code, Javadocs and other useful information about the system are freely obtainable on the Hackstat Development Website (<http://www.hackstat.org>).

4.5.1 Design and Implementation

The Hackstat PRI Extension has a fairly complicated design. It consists of numerous classes organized by eleven different Java packages. See Table 4.1 for a listing and description of all the packages. In the next sections, I describe the design and implementation of some of the important hackyPRI packages.

Package `org.hackstat.app.pri.model.workspace.measure`

This package provides Java classes that represent PRI measures. Each measure implements the `WorkspacePriMeasure` Interface, shown in its entirety in Table 4.2. The purpose of this interface is to standardize the functionality of each and every measure. For example, each measure must be able to calculate its value and return the calculated value in a readable form. The standardization of the functionality of PRI measures greatly improved the configurability of the system. In addition, the interface defines four methods: `isAggregateMeasure`, `isCacheEnabled`, `writeCache`, and `readCache`, which provide the ability to save the results of a measure for future use. Once the measure is calculated it should not need to be re-calculated.

Table 4.1. Java Packages in the hackyPRI system

Package	Description
<i>org.hackystat.app.pri.admin.analysis.remove</i>	Provides administrative facilities for deleting the PRI measure persistent caches.
<i>org.hackystat.app.pri.analysis.listworkspace</i>	Provides the List Workspace analysis, which simply lists the workspaces within a Project.
<i>org.hackystat.app.pri.analysis.module</i>	Provides the Project PRI Module Ranking analysis, which provides the PRI ranking for top-level modules within a specified project.
<i>org.hackystat.app.pri.analysis.workspace</i>	Provides the Project PRI Ranking analysis, which provides the PRI ranking for workspaces within a specified project.
<i>org.hackystat.app.pri.analysis.workspace.selector</i>	Provides various selectors used in the Project Workspace Ranking analysis.
<i>org.hackystat.app.pri.model.configuration</i>	Provides the Project PRI Configuration representation, which models the configuration of PRI attributes for a specific Project's PRI ranking.
<i>org.hackystat.app.pri.model.configuration.selector</i>	Provides various selectors that allow the selection of Project PRI Configuration in the Hackystat analyses.
<i>org.hackystat.app.pri.model.workspace</i>	Provides the Project Ranking Workspace representation, which calculates, stores and ranks the PRI ranking for a specified project.
<i>org.hackystat.app.pri.model.workspace.measures</i>	Provides the implementation of various PRI measures, which are used in a Project's PRI ranking.
<i>org.hackystat.app.pri.model.workspace.measures.helper</i>	Provides classes that aid in the calculation of the PRI measures.
<i>org.hackystat.app.pri.model.workspace.measures.indicator</i>	Provides the implementation of various PRI indicators, which are used in a Project's PRI ranking.
<i>org.hackystat.app.pri.preference.configuration</i>	Provides a set of Project PRI Configuration preference commands, which allows the user to create, modify, and delete Project PRI Configurations.
<i>org.hackystat.app.pri.util</i>	Provides utility classes that aid the processing of PRI calculations.

The use of a persistent cache greatly reduces the processing time required to provide a real-time PRI ranking for a specified Project. PRI measure persistency is the first level of optimizations in the hackyPRI system. The second and third level is described in the workspace package. Optimization is an important aspect of hackyPRI, because the PRI rankings span across the lifetime of the Project, which includes all Hackystat Sensor data associated with the Project. As you can imagine, without some sort of persistency and optimization, generating a PRI ranking will be quite computationally expensive.

A specific example of one of the PRI measures is the Commit Contribution measure. Like all PRI measures provided by hackyPRI, the Commit Contribution measure implements the WorkspacePriMeasure Interface and therefore provides a standard set of functionality. However, each PRI measure has its own specific defining characteristics. First, the Commit Contribution measure is an Aggregate PRI measure. Therefore, its `isAggregateMeasure` and `isCacheEnabled` measure both return the boolean true. Second, it is obvious that each measure is calculated differently. Therefore, the Commit Contribution measure accesses specific Hackystat components to obtain the values of the product and process measures.

Table 4.3 presents a description of all the PRI measures in the hackyPRI system.

Table 4.2. The WorkspacePriMeasure Interface that defines the functionality of all PRI measures.

```

/**
 * Returns the label of this measure.
 * @return The label of this measure.
 */
public String getLabel();

/**
 * Determines if the measure is an aggregation of past data. If this method returns true,
 * then this indicates that each day since the start day of the project is used to calculate
 * the measure's value. If this method returns false, then the measure is calculated from
 * the last build day.
 * @return True if the measure is an aggregate calculation, false otherwise.
 */
public boolean isAggregateMeasure();

/**
 * Determines if the cache is enabled. Generally, measures that are calculated in an aggregation
 * of past data is cached.
 * @return True if the cache is enabled, false otherwise.
 */
public boolean isCacheEnabled();

/**
 * Calculates and returns the value of the PRI measure for the specified workspace and day.
 * @param workspace Specifies the workspace to calculate the measure for.
 * @param day Specifies the day to calculate the measure for.
 * @throws Exception If a problem occurs.
 */
public void calculate(String workspace, Day day) throws Exception;

/**
 * Returns the formatted String of the calculated value.
 * @param workspace The workspace to get the calculated value.
 * @return The formatted value.
 */
public String formatCalculatedValue(String workspace);

/**
 * Returns the Day of the earliest cached value. This day should be used
 * to start calculating the measures values.
 * @return The Day object of the earliest cached value for this measure.
 */
public Day getEarliestCacheDay();

/**
 * Writes out the cache objects associated with this instance of the measure
 * object. The collection of caches are stored in the Project owner's partitions.
 * @throws Exception If a problem occurs.
 */
public void writeCache() throws Exception;

/**
 * Reads in the cache objects associated with this instance of the measure
 * object and creates an internal representation of the cache. The collection of caches
 * are stored in the Project owner's partitions.
 * @throws Exception If a problem occurs.
 */
public void readCache() throws Exception;

```

Table 4.3. Summary description of all the PRI measures within the hackyPRI system

PRI Measure	Description
<i>WorkspacePriActiveTime</i>	Represents the total aggregate active time for each workspace in a project.
<i>WorkspacePriActiveTimeContribution</i>	Represents the total aggregate number of active time member contributions for each workspace in a project.
<i>WorkspacePriActiveTimeFirst</i>	Represents the first day active time was recorded for each workspace in a project.
<i>WorkspacePriActiveTimeLast</i>	Represents the last day active time was recorded for each workspace in a project.
<i>WorkspacePriActiveTimeTest</i>	Represents the total aggregate active time for test code in each workspace in a project.
<i>WorkspacePriCommit</i>	Represents the total aggregate number of commits for each workspace in a project.
<i>WorkspacePriCommitContribution</i>	Represents the total aggregate number of commit member contributions for each workspace in a project.
<i>WorkspacePriCommitFirst</i>	Represents the first day commit information was recorded for each workspace in a project.
<i>WorkspacePriCommitLast</i>	Represents the last day commit information was recorded for each workspace in a project.
<i>WorkspacePriCodeChurn</i>	Represents the total aggregate code churn for each workspace in a project.
<i>WorkspacePriCoverage</i>	Represents the latest snapshot of the method level coverage percentage for each workspace in a project.
<i>WorkspacePriDependency</i>	Represents the latest snapshot of the number of inbound and outbound dependency references for each workspace in a project.
<i>WorkspacePriExpert</i>	Represents the project member who has the most active time and commits for a each workspace in a project.
<i>WorkspacePriFileMetric</i>	Represents the latest snapshot of the number of lines of code, number of methods, number of classes for each workspace in a project.
<i>WorkspacePriFileMetricTest</i>	Represents the latest snapshot of the number of lines of test code, number of test methods, number of test classes for each workspace in a project.
<i>WorkspacePriIssueOpen</i>	Represents the latest snapshot of the total number of open issues for each workspace in a project.
<i>WorkspacePriIssueClosed</i>	Represents the latest snapshot of the total number of closed issues for each workspace in a project.
<i>WorkspacePriReview</i>	Represents the total aggregate number of review issues for each workspace in a project.
<i>WorkspacePriReviewLast</i>	Represents the last day review issues was recorded for each workspace in a project.
<i>WorkspacePriUnitTest</i>	Represents the latest snapshot of the number of executed unit tests for each workspace in a project.
<i>WorkspacePriUnitTestResult</i>	Represents the aggregate of the results of unit test invocations for each workspace in a project.

Package `org.hackystat.app.pri.model.workspace.indicator`

This package provides Java classes that represent PRI indicators. Each indicator implements the `PriIndicator` Interface. Much like the `WorkspacePriInterface`, the `PriIndicator` Interface standardizes the functionality of each and every PRI indicator within the system. The main functionality provided by a PRI indicator is the 0 to 100 ranking for each workspace. Although PRI indicators only provide a single function, creating the indicator ranking is quite complex. Essentially, a PRI indicator ranking is analogous to making a testable hypothesis about the workspace's MINI or LINI determination. For example, if you, the developer, believe that a certain PRI measure value is “bad”, then the Java implementation of the PRI indicator should be able to understand a “bad” value and return a low ranking.

As previously mentioned, PRI indicators are designed to be able to use one-to-many PRI measures to provide a ranking for a specific workspace. However, most of the current set of implemented PRI indicators within the hackyPRI system has a one-to-one relationship with the available PRI measures. Currently, each PRI measure explained in Table 4.3 has an associated PRI indicator. I made this design decision simply because a one-to-one implementation provides flexibility and simplicity for this research. However, I have not evaluated which technique provides the best results.

The following sections contain descriptions of a few PRI indicators in the hackyPRI system.

Expert PRI Indicator The calibration of this indicator ranking looks at the email address of the expert, determined by the Expert PRI measure. Currently, the Expert PRI indicator ranks a single developer as the highest ranked expert and any other expert is given a lower ranking. The justification for this calibration is as follows. This developer is the most experienced developer that is contributing to the project. In addition, most of the developer's implementation is technical passes over the code to ensure that the code is of high software quality. Therefore, code that this developer creates is ranked higher than code developed by other developers.

One major problem associated with this indicator ranking, is that the string that represents the email address of the highest ranked developer is hard coded into the Java code that implements the Expert PRI Indicator ranking. Currently, I have proposed several solutions to this problem, however, it seems that none of the solutions are very elegant. Therefore, a future solution could be the JESS tool [41], explained in Section 4.6.2.

Active Time PRI Indicator All other indicators being equal, the workspace with the highest active time would seem to be the highest ranked workspace. Therefore, this indicator calculates a ranking by accessing the Active Time PRI measure to determine the maximum active time for all workspaces and the active time associated with a specific workspace.

One major problem with this ranking is having an excessive amount of active time could indicate that the workspace is defect prone. This fact leads me to believe that some sort of statistics should be used in this ranking, to normalize the existence of a workspace with excessive amounts of active time (the outliers). I am currently searching for the right statistical method to use for this ranking

First Active Time PRI Indicator If a workspace has a recent first day of active time, then it can be determined that the code is relatively new. Therefore, all other indicators being equal, if a workspace has recent active time, then that indicates new code being added to the workspace. In most cases, one would assume that new code is more defect-prone than old code. Of course there seems to be the notion that very old code can become default prone as well.

Active Time Contribution PRI Indicator All other indicator being equal, the more developers that looks at the code and adds to its code base, then the code's quality will improve. This ranking suggests that if one developer is only developer to work on the code, then it could be defect prone. Of course, this ranking also could cause problems. For example, what if seven out of the eight developers in a project work on the code, then comes along a very inexperienced developer and adds defects, the ranking would actually improve.

Package `org.hackystat.app.pri.model.workspace`

This package is the workhorse of the system. It implements the PRI ranking function, which uses and manages PRI measures and PRI indicators. This package's most important tasks are to control the PRI ranking generation, PRI measure computation, PRI indicator rankings, and saving the PRI measures to a persistent cache at specific times during the processing of a PRI ranking. Table 4.4 provides a summary description of all classes in this package.

An important class in this package is the `ProjectWorkspaceRanking` class. This class implements the algorithm presented in Table 4.5. This algorithm is the second level of optimization in hackyPRI system. It is optimized to quickly calculate the PRI rankings for a Project. The algorithm

Table 4.4. Summary description of all classes in the org.hackystat.app.pri.model.workspace package

Class Name	Description
ProjectWorkspaceRankingManager	Provides the management of a collection of ProjectWorkspaceRanking objects.
ProjectWorkspaceRanking	Provides the facilities to calculate PRI measures and create a ranking based on the calculated results.
WorkspaceRanking	Provides a representation of the ranking for a specific workspace, which contains a collection of measure values and a collection of PRI indicator rankings. Used primarily for presentation purposes.
PriRankComparator	Provides a comparator that compares the PRI ranking value from two WorkspaceRanking objects.

determines when to process the PRI measures, when to persistently store them, when to gather the PRI indicator rankings, and when to process Aggregate or Snapshot measures.

In addition, the ProjectWorkspaceRanking class generates the aggregate PRI ranking for each workspace. For a single workspace, the ProjectWorkspaceRanking class generates an aggregate ranking by accessing each independent PRI indicator ranking and using the weights configured in the Project PRI configuration. A very simple example is the following: a workspace has these indicator rankings {92, 100, 30, 15} and these respective indicator weighting {1, 1, 1, 2}. The aggregate ranking of this workspace would be $(92*1) + (100*1) + (30*1) + (15*2) = 252$.

Another important class in this package is the ProjectWorkspaceRankingManager. This class manages ProjectWorkspaceRanking classes. It determines whether or not a PRI ranking was already calculated for a specified Project, whether or not to re-calculate the PRI ranking, and whether or not the Project and Configuration information have changed. These determinations are part of the last level of optimization in the hackyPRI system. The basic idea behind this optimization is that once a correct PRI ranking has been calculated, the system should not re-calculate the ranking. However, to ensure correct results the ProjectWorkspaceRankingManager is able to determine when a re-calculation is necessary.

4.5.2 Design and Implementation Improvements

As I previously stated, the Hackystat PRI Extension (hackyPRI), has evolved significantly as I discover new ways to optimize the calculations necessary to create a PRI ranking and to better support configuration for different projects and organizations.

Table 4.5. ProjectWorkspaceRanking algorithm

```

FOR each day starting from the project's end day to the project's start day
  IF the daily build is buildSuccessful
    successfulBuildDay = currentDay
    BREAK
  END IF
END FOR

Determine optimized startDay to start processing Sensor Data
Determine endDay to stop processing Sensor Data
Determine cacheDay to cache the aggregate PRI measures

FOR each day starting from the startDay to the endDay
  process aggregate WorkspacePriMeasure objects
  IF day is equal to successfulBuildDay
    process snapshot WorkspacePriMeasure objects
  END IF
  IF day is equal to cacheDay
    cache aggregate WorkspacePriMeasure objects
  END IF
END FOR

Create presentation WorkspaceRanking objects
Filter presentation WorkspaceRanking objects

```

Optimization is one of the major features that I have recently implemented. In previous designs, the system's execution time, the duration of time required to create a Project's PRI ranking, were unacceptable. In the first version of hackyPRI, the system required 60 minutes to execute a complete PRI ranking for the hacky2004-all Hackystat project, which contains 2 years of Sensor Data gathered from 9 different Hackystat users (approximately 14 thousand Hackystat XML sensor data files, roughly equal to 956 Megabytes of data). This result was collected on a computer with a 3.4 GHz Pentium 4 processor with Hyperthreading and 1.00 GB of RAM. Furthermore, the system required 60 minutes to process the same project's data for each and every execution even though the data and ranking did not change. In my opinion, this execution time violates the intended design of "real-time" PRI rankings. In addition, this slow execution time hampers the ability to properly calibrate the PRI indicators.

Therefore, in subsequent versions of the system, the computation time has been greatly improved. Under normal situations the execution time has been reduced from 60 minutes to execution times that range from a fraction of second to 5 minutes for the same Hackystat project and on the same 3.4 GHz computer. Using the longer duration of 5 minutes, I achieved a 92 percent decrease in processing time from the previous version. In my opinion, I have achieved the goal

of a “real-time” PRI ranking. A simple persistent caching of the calculated values accounted for the majority of the dramatic decrease in execution time. However, there are two situations where the execution time will become quite lengthy. First, when the persistent caches are non-existent. Second, when the persistent caches are deleted. In these two cases, the system must calculate and persistently store the caches. This action requires approximately 50 minutes on the same computer. However, once this action is executed, the system does not need to re-calculate the measure caches. Therefore, under normal situations the execution time ranges from a fraction of a second to 5 minutes. Of course, execution times will vary depending on the Hackystat server’s speed, memory, and on the amount of project data.

The second area of development focuses on the ability to configure hackyPRI for other organizations and projects. In previous versions of the system, it was quite impossible to extend and configure for other software projects without a significant amount of developer effort. For example, in a previous version, it definitely could be the case that another organization would have to redesign the entire system in order to create a PRI ranking that works best for them. Obviously, this was a major problem that needed to be resolved. Step 1 of the Priority Ranked Inspection process states that various product and process measures must be selected and calibrated to best distinguish MINI documents from LINI documents. This selection process will not be the same for all software projects. Therefore, it is quite obvious that a properly designed system will allow the configuration of different product and process measures without having to completely redesign the system. For hackyPRI to be successful, different software projects should be able to easily extend the current set of PRI measures.

The current version of hackyPRI implements smaller and more configurable pieces. Under this new design, when a new PRI measure is added to the system only a few lines of code must change. In addition, swapping different PRI measures and PRI indicators in and out of the PRI ranking is now very simple. See Sections 4.4.1 and 4.4.2 for a detailed description of the steps required to add and remove PRI measures and add, remove, and calibrate PRI indicators.

4.6 Future Implementation Enhancements

The development of the hackyPRI extension is by no means complete. Rather, it is just in its very beginning stages. There are many improvements that can be made to the system to better support the Priority Ranked Inspection process. The following is short list of some possible future implementation enhancements.

4.6.1 Threats to Data Validity

Currently, the PRI rankings are calculated for the entire life span of projects. Since projects can exist for years, this requires certain optimizations that make the calculations quicker. One of the optimizations that I have constructed is a persistent caching of PRI measures. This feature ensures that years of Hackystat Sensor Data will not be continually processed by the system. It also shortens the calculation time from hours to seconds.

However, there are a few tradeoffs that come with this optimization feature. First, changing or adding Hackystat data to a date that occurs before the date the cache was created will be problematic, because the new data will not be included in the cache. There are a couple of solutions. Solution one, a Hackystat administrator can rebuild the caches, which could take a long time (50 minutes for the hacky2004-all project). Solution two, we can choose to ignore the data, simply because at the grain size of years a couple of new data points would not affect the rankings that significantly. In any case, a solution needs to be found to ensure the validity of the PRI ranking.

The second problem occurs not only in hackyPRI but also in the whole Hackystat system. Refactoring is a major threat to data validity. For example, if a developer spends 50 hours working on a workspace, then decides to change its name; the 50 hours won't be attached to the new workspace name. This could be a major threat to data validity, because if every single workspace in the PRI rankings were refactored, then the workspaces' measures will be empty and no rankings will be generated. On the other hand, this action is very unlikely.

4.6.2 PRI Indicator Ranking

Currently, the PRI indicator ranking (the 0 to 100 ranking provided by each PRI indicator) is implemented with Java code in the hackyPRI system. I have chosen to do this at this time, simply because creating a configurable indicator ranking would be too time consuming. Furthermore, I'm not totally convinced that users of PRI should be able to change these low level rankings. In my current model, users are allowed to represent their own calibration by changing the indicators' weights. If I find that a configurable indicator ranking is needed, then one possible solution could be the JESS tool [41].

4.6.3 Other Levels of Ranking

Currently, hackyPRI only supports a PRI ranking for workspaces. Other levels of PRI rankings could be possible. For example, ranking modules, Java Classes, or even methods with in

a Java class. It is not known if there are any advantages of providing different levels of rankings at this time. Furthermore, I currently do not know the level of programming difficulty that would come with adding these different levels. However, I anticipate that this will not be difficult.

4.6.4 Better Support for More Programming Languages

Currently, the implementation of the PRI measures and indicators are best suited for the Java programming language. Although Hackystat and hackyPRI are both programming language agnostic, they both currently provide the more support for Java than any other programming language. However, because Hackystat and hackyPRI are easily extendible, adding more support for any another programming should be trivial.

4.6.5 Link with Software Project Telemetry

Currently, hackyPRI is not well designed for tracking changes over time. For example, if a developer suddenly adds 200 lines of code to a Java class that was previously 50 lines of code, then that could indicate a possible problem. A large and unusual addition such as this should affect the MINI and LINI determination for that Java class. Therefore, I believe that adding the variation of trends into the PRI ranking function will be a huge contribution to its overall robustness. In addition, I believe that tracking the trends of the determination of MINI and LINI and the validation of its correctness by conducting inspections over time will be a useful analysis, which could aid the calibration of the PRI ranking function.

Software Project Telemetry [34] is a new approach to software project management, which uses Hackystat to provide high-level development trends. This infrastructure is an excellent way to provide trends in the values of PRI measures, trends in PRI indicator ranking, and trends in the of the correctness of the MINI and LINI determination. I believe it would be possible to create the necessary Telemetry components to make this possible.

4.6.6 Automatic Calibration Feedback Loop

I believe it could be possible to automatically calibrate the PRI ranking function. This is a possibility, because Hackystat can collect validation data in the form of inspection data. If enough inspections are conducted then Hackystat could automatically determine the best possible calibration to identify the best possible inspection results. There are many human factors that can jeopardize this possibility. For example, if the amount of time spent on each inspection varies, then

it would be difficult for the automatic processor to determine how to calibrate the measures with inconsistent data.

4.7 Contributions to Hackystat

PRI ranking is a brand new way to represent software product and development process measures in Hackystat. PRI rankings are a novel idea and I believe it is a positive contribution to the possibilities of Hackystat. On the other hand, without the Hackystat framework, I am not sure it would be possible to create a usable PRI ranking implementation.

I have been a major contributor to Hackystat throughout my Master's studies and have implemented many components in the system. Throughout my contributions I have found that adding a new feature to Hackystat often requires a fix or implementation of another area of the system. Creating the hackyPRI extension was no different. Throughout my design and implementation of the hackyPRI extension, I have positively contributed to the Hackystat project in many ways. For example, during the design of the PRI ranking function, I have made contributions in the following areas:

1. Snapshot Sensor Data Type Enhancements [42] - I identified and fixed a major flaw in seven Sensor Data Types within the Hackystat system.
2. hackyDependency - I implemented Hackystat components to obtain dependency or coupling software product measures. This includes a Sensor Data Type, Sensor, and various other Hackystat components that allow this measure to be useful to all Hackystat analyses.
3. hackyIssue - Burt Leung and I implemented the Jira Issue sensor and Project-level infrastructures to support software issue measures.

The last two contributions are real examples of Step 1a, Selection of product and process measures to use in the PRI ranking function, of the Priority Ranked Inspection process (Section 4.4.1). Dr. Johnson and I both agreed that a previous PRI ranking function that I built, needed to consider Dependency and Issue measures, both of which were not implemented. Therefore, I implemented the necessary Hackystat components to be able to create the respective PRI measures.

4.8 Using the Hackystat PRI Extension

The Hackystat Priority Ranked Inspection extension can be downloaded for use in other software organizations by visiting the Hackystat Developer Services website (<http://www.hackystat.org>). In addition, Hackystat's User Guide, full source code, Java documentation, and other useful information that are required to install Hackystat and the hackyPRI system are obtainable at this website. Any questions and suggestions can be sent to Hackystat Users email mailing list (hackystat-users-1@hawaii.edu) or directly to me (kagawaa@hawaii.edu).

Chapter 5

Exploratory Study Procedure

This chapter discusses the exploratory study procedures conducted in this research. The main thesis of this 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). This chapter will describe how the main thesis was studied.

5.1 Subjects Used in the Study

I studied the implementation and inspection process of the Hackystat System ¹ developed in the Collaborative Software Development Laboratory (CSDL) ². CSDL is a research laboratory within the Department of Information and Computer Sciences at the University of Hawaii. Currently, CSDL is comprised of Professor Dr. Philip Johnson and seven graduate Computer Science students, including myself. Our mission is to provide a physical, organization, technological, and intellectual environment conducive to collaborative development of world-class software engineering skills. CSDL's current focus is on the development of Java software systems that support software development research. Hackystat [23], hackyPRI, Jupiter [43], LOCC [44] are examples of software we have developed. All eight members are highly experienced Java programmers and have been practicing high quality software development. We use tools such as Eclipse, Jakarta Tomcat, Apache Ant, and CVS. In addition, we practice several development techniques such as Extreme Programming and inspection.

In the exploratory study of the Priority Ranked Inspection process, I focused on studying PRI's effectiveness in aiding CSDL's inspection of Hackystat related software code. Like most

¹See <http://www.hackystat.org> for more information about the Hackystat System.

²See <http://csdl.ics.hawaii.edu> for more information about CSDL projects and members

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 Java packages; 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 minimized any possible data contamination by doing two things. First, I ensured that the inspection participants are "blind" to the document selection method. There are two methods of selection that were used in this study, selection with and without aid of PRI. I worked with individual authors to select documents based on their subjective selection or with the aid of PRI and kept that decision a secret from the rest of the participants. Second, although I participated in the inspections, the defects that I discovered will not be used in the study.

CSDL has been conducting and studying inspections since the early 1990's. CSDL's inspection process has gone through the use of many different tools and processes. Our current inspection guidelines are published in the Hackystat Developer Documentation: Software Review Guidelines [45]. The term "review" used in CSDL's process equates to the term "inspection". The primary goals of the current process includes creating an educational process that allows participants to learn new techniques and practices about developing high quality software design and implementation and to remove defects. The process is lightweight and includes 5 simple steps:

1. **Announcement (or Review Request)** - In this step, an author sends an email requesting that the group inspect the specified software. In addition, the author lists several questions to help direct the participants' attention to what the author thinks is most important. This announcement should be sent 24 hours before the meeting.
2. **Preparation** - In the hours between the announcement and the meeting, the review participants must individually examine the software listed in the announcement and log any issues that are found. Preparation time is limited to no more than one hour.
3. **Meeting** - At the scheduled time the group gathers to discuss the validity of the issues that have been discovered in the preparation step.
4. **Revision** - After the review meeting, the valid issues that were discovered must be fixed. In this step, the author or assigned developer must resolve these issues.
5. **Verification** - After the revision, a quick determination is required to ensure that all the issues have been resolved.

Note that the CSDL inspection process does not specify how to determine what software should be inspected. The process simply starts with an announcement. This missing step is evident in all traditional software inspection processes.

Currently, CSDL utilizes the Jupiter Eclipse Review Plugin ³ [43] to support our inspection process. Jupiter is a lightweight tool that supports, to a varying degree, all steps of the CSDL inspection process. For example, individual inspectors use Jupiter during the Preparation phase to log issues that he/she has found. In addition, Jupiter collects various properties of the review issues generated in an inspection process. The review issue properties that have been collected and used in this study are severity, type, and resolution. These properties allow the inspectors to specify additional information about the discovered issue to accurately discuss them in future phases of the inspection process. Table 5.1 provides a full listing of the properties and their values that were used in this study.

Table 5.1. Jupiter Properties and Values

Property	Meaning	Values
Severity	Allows the inspector to note the importance of the issue.	Critical, Major, Normal, Minor, Trivial
Type	Allows the inspector to note what type of issue he/she has found.	Coding Standards, Program Logic, Optimization, Usability, Clarity, Missing, Irrelevant, Suggestion, Other
Resolution	Allows the team to determine the validity and necessary actions required to resolve the issue.	Valid Needs Fixing, Valid Fix Later, Valid Duplicate, Valid Won't Fix, Invalid Won't Fix, Unsure Validity

I used the issues' property value information to analyze the validity of several claims. For example, I was able to count the number of high-severity issues that were discovered by the inspection. For this study, high-severity is defined as defects with a severity equal "Critical" or "Major" and a resolution not equal to "Invalid Won't Fix" and "Valid Duplicate". A major problem that I did not address in this research and study is the subjective opinion used when inspectors assign values to specific properties. For example, one inspector's subjective opinion of a Severity value could differ from another inspector's opinion.

³Takuya Yamashita, who is also a CSDL member, developed the Jupiter software.

5.2 Study Limitations

The use of CSDL resources in my study indicates a major limitation on this research. The most accurate and thorough evaluation of PRI should inspect *all documents* to evaluate PRI's classification of MINI and LINI documents. However, because I am using CSDL's inspection resources, which are limited, this was not possible.

Currently, Hackystat and its extensions are comprised of 218 packages. At best this will take 2 hours per inspection per member, therefore totaling 3,488 hours of inspection. Requiring the use of that many hours is an unrealistic demand on CSDL resources. Therefore, my exploratory study investigated a small percentage of the system in hopes that a cross-section provided adequate and acceptable results. Furthermore, CSDL conducts inspections to increase quality and spread knowledge. It would be detrimental to this development group, if I required the inspection of many packages that did not provide that return on investment.

It is important to note two other limitations of this research. First, I am not defining a set of PRI measures and PRI indicators that represent the PRI ranking function for all software projects. Instead, by using hackyPRI I will be able to go through a methodology to best calibrate the ranking functions 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.

5.3 Study of Thesis Claims

To study this thesis, I separated it into three claims based upon the three intended benefits of PRI.

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

To evaluate my thesis claims, I created a six-part study procedure. The study includes; questionnaires, working with authors to select documents for inspection, and the analysis of an

inspection log and results. The different portions of the study procedure do not necessarily correlate one-to-one with the three thesis claims. Instead, each procedure provides supporting evidence for all of my thesis claims. The following is a short summary of the steps of the study procedure. The following sections explain each of the steps in more detail.

1. **Pre-Selection Questionnaire:** I administered a questionnaire to obtain the developers feelings assessing the usefulness of inspection and the methods they use to select documents for inspection. In addition, I asked each developer to provide rankings, based on their current subjective opinions, for three different sets of workspaces. First, they ranked each top-level module within the Hackystat system (i.e., hackyKernel, hackyStdExt, hackyReview, etc). Second, they identified the top five packages throughout the whole Hackystat system that they thought were MINI and LINI. Last, I asked them to rank packages they have authored based on their opinions of what packages are MINI and LINI.
2. **Package Selection:** I worked with individual developers to select a package for inspection. The selection of packages can be made with or without the aid of PRI.
3. **Request for Inspection:** After a package was selected for inspection, I instructed the author to send an email-based request for inspection to our fellow Hackystat developers. I ensured that this email “blinded” the selection method.
4. **Inspection of the Selected Package:** Using the CSDL code review (inspection) process, the inspection participants inspected the package individually and met to discuss the issues that were discovered. The author of the package, who is also the developer I worked with in Step 2, did not inspect his/her own code.
5. **Post-Inspection Questionnaire:** Following the inspection I administered a questionnaire that asked the participants whether they believed the package was MINI or LINI.
6. **Record Results of Inspection:** I recorded the results of the inspection, the PRI ranking of the package before and after the inspection, and other PRI and inspection results.

5.3.1 Part 1 - Pre-Selection Questionnaire

The first study procedure that was used is a questionnaire. The goal of the questionnaire to obtain the authors’ opinions about inspections in general, their document selection process, and their subjective rankings of the modules, packages, and packages that they have authored.

Appendix B contains the Pre-Selection Questionnaire. This questionnaire contains three different sections; two general inspection questions about CSDL's inspection process, four questions assessing the developers' document selection method, and three tasks which gathered the developers' subjective rankings of various packages.

The first section contains general questions about the CSDL inspection process. These questions do not directly correlate to my thesis claims. However, I will use this information to help validate the data that I collect. For example, if the developers find an insignificant number of issues in both MINI and LINI documents, then I can correlate that finding with their enthusiasm towards inspection. In addition, one of the questions asks whether finding defects are the most important outcome of the CSDL inspection process and the data collected will aid future directions of this research. It is my future-hypothesis that PRI can also aid the selection of documents for purposes other than finding the most defects. For example, an inspection process can be used as training or education and PRI could aid the selection of documents that best suit that goal.

The second section contains questions about the developers' current document selection method. This section provides important information on the process in which developers select documents for inspection. In addition, it provides supporting qualitative data for the quantitative data provided in the last section.

The last section of the questionnaire asked the developers of Hackystat to provide a numerical ranking, based on their subjective opinions, of three different sets of packages. First, they were asked to rank the top-level workspaces, or modules, within Hackystat according to their subjective opinion of the quality of the modules. Hackystat contains many different modules that can interchange depending on the situation of use⁴. Second, they ranked the top five packages in the entire Hackystat project that they thought were MINI and LINI. Last, they ranked the packages that they would volunteer for inspection. The packages used in this last set were packages that the developer has authored.

When analyzing the results of the developers' subjective rankings, I will be able to compare the developers' subjective rankings against the PRI ranking. This comparison 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. There are three possible results from this study. 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. Alternatively, I might find that, developers have no idea what code needs to be inspected. The third possible result

⁴See the Hackystat Developer Website (<http://www.hackystat.org>) for a listing of the modules in the system.

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. Of course, these results will need to be validated with the actual inspection of the document to validate both the developers' subjective rankings and the PRI rankings.

5.3.2 Part 2 - Package Selection

The second study procedure that was used is the selection of packages for inspection. The goal of this procedure is to study the effectiveness of the MINI and LINI determinations and PRI's ability to help the selection process.

In this portion of the study, I worked closely with the various authors who contribute to the Hackystat project to select a package for inspection. To accomplish this, I created a weekly inspection schedule (See 5.2). Each week I worked closely with a different Hackystat developer to select one package for inspection. This selection process was designed with the following steps:

1. Explain to the developer that the goal of this collaboration is to find the document that is most in need of inspection. Therefore, using PRI is not required.
2. Ask the developer if they have a package they would like to be inspected. If so, record the package name and the MINI or LINI ranking and skip to step 6. If not, continue to the next step.
3. Present the developer with a list of MINI packages that he/she has authored. Work with the author to select a package from this listing. If a document is selected, then move to step 6. If not, continue to the next step.
4. Present the developer with a list of LINI packages that he/she has authored. Work with author to select a package from this listing. If a document is selected, then move to step 6. If not, continue to the next step.
5. If we have reached this step, then it can be determined that the author does not believe he/she has authored any packages that are more in need of inspection. Therefore, I will select a document from the MINI listing. According to the author this package should not generate many high-severity issues. However, according to PRI this package should generate high-severity issues. The results will be recorded.

6. Once a document has been selected, the author is required to send a request for inspection to the inspection participants (generally all current CSDL members).

This process was designed to “blind” the method used in selecting the package. The inspection participants, who include all CSDL members excluding the author and myself, did not know how the package was selected. This “blinding” of the selection method was created to ensure that the participants did not consciously or unconsciously persuade the results of the inspection.

5.3.3 Part 3 - Request for Inspection

After a package has been selected in Part 2, the author is required to send an email request for inspection. Again, this email “blinded” the selection method from the participants. It simply stated that the author requests the inspection of a particular package and provides the necessary information that is required to successfully inspect the package. This request announcement was congruent with CSDL’s inspection process [45].

5.3.4 Part 4 - Inspection of Selected Package

This part of the study procedure required little change from CSDL’s original inspection process defined in the Hackystat Software Review Guidelines [45]. The participants conducted the Preparation and Meeting phases of the inspection process.

The Jupiter review tool [43] was used to gather the issues generated in the Preparation phase. In addition, the Jupiter tool is used in the Meeting phase to record the validity and severity of the issues.

5.3.5 Part 5 - Post-Inspection Questionnaire

The fifth part of the study procedure is a quantitative questionnaire. Appendix C contains the Post-Inspection Questionnaire. The goal of this questionnaire was to obtain the developers’ opinions of the MINI or LINI determination. The results of the inspection and the developers’ opinions helped to determine if the PRI ranking for the package was correct.

5.3.6 Part 6 - Record Results of Inspection

There are two possible results of this portion of this study. First, the packages that were selected were correctly categorized by PRI. Second, the packages that were selected did not reflect the PRI ranking function. These findings will provide evidence for claim 3 of my thesis statement.

During this study, CSDL has conducted nine inspections. However, in addition to the inspections conducted under this study, I have recorded data about eleven other inspections. In total, I have data on twenty inspections and information on the PRI ranking functions.

Throughout my exploratory study of PRI, I monitored the validity of the PRI ranking function throughout each inspection. To accomplish this, I collected specific pieces of information when conducting inspections. The following is a specific list of the information collected:

- Inspection date
- Hackstat module, package, and inspection ID
- PRI determination (MINI or LINI)
- PRI measure values and PRI indicator ranking and weighting
- Subjective discussion of the validity of the PRI ranking 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 helped me keep track of the progress of the inspections and the validity of the PRI ranking function. As I previously stated, the calibration of the PRI ranking function is an ongoing and evolving process. Collecting these types of information will help an organization keep track of that evolution. The end goal of the continued study of PRI 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.

5.4 Study Timeline

The following table provides a timeline for the exploratory study of this thesis. The developer names used in this timeline are hidden to protect the developers' identity.

Table 5.2. Study Timeline

Timeline	Study Activity
April 6, 2005	Package Selection: Developer 5 Pre-Selection Questionnaire: Developer 5 Review Request: Developer 5 Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
April 13, 2005	Package Selection: Developer 6 Pre-Selection Questionnaire: Developer 6 Review Request: Developer 6 Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
April 20, 2005	Package Selection: Developer 9 Pre-Selection Questionnaire: Developer 9 Review Request: Developer 9 Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
April 27, 2005	Package Selection: Developer 7 Pre-Selection Questionnaire: Developer 7 Review Request: Developer 7 Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
May 4, 2005	Package Selection: Developer 4 Pre-Selection Questionnaire: Developer 4 Review Request: Developer 4 Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
May 11, 2005	Package Selection: Developer 3 Pre-Selection Questionnaire: Developer 3 Review Request: Developer 3 Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
June 1, 2005	Package Selection: Aaron Kagawa Review Request: Aaron Kagawa Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
June 8, 2005	Package Selection: Aaron Kagawa Review Request: Aaron Kagawa Review of Selected Code: CSDL Post-Inspection Questionnaire: CSDL
June 15, 2005	Finished analyzing the results.

Chapter 6

Results

This chapter presents the results of this research. The procedure I used in my exploratory study is explained in Chapter 5. In this study, I tested three of my thesis claims. The following list states my thesis claims and their associated results:

Thesis Claim 1: Inspecting MINI documents will generate more high-severity defects than inspecting LINI documents.

- The results show supporting evidence for Thesis Claim 1.

Thesis Claim 2: PRI can enhance the volunteer-based document selection process.

- The results show supporting evidence for Thesis Claim 2.

Thesis Claim 3: PRI can identify documents that need to be inspected that are not typically identified by volunteering.

- The results are inconclusive for Thesis Claim 3.

The sections of this chapter are organized in the following manner. In Section 6.1, I revisit some of the limitations that hinder the results of this study. In Section 6.2, I summarize the results of the conducted inspections. In Section 6.3, I explain a few terms that will help explain the results. In the next three sections, Sections 6.4, 6.5, and 6.6, I provide supporting evidence for my thesis claims. These three sections are organized by presenting the strongest evidence first. Therefore, the order is Thesis Claim 1 (Section 6.4), Thesis Claim 2 (Section 6.5), and finally Thesis Claim 3 (Section 6.6). In Appendix D, I present the raw data results from the Pre-Selection-Questionnaire. In Appendix E, I present the raw data results from the inspections and Post-Inspection-Questionnaire.

6.1 Limitations Revisited

As explained in the previous chapter, there are many limitations associated with the study of this research. Most notably is the absence of sufficient inspection resources within the CSDL organization, during the time period of my study, to thoroughly inspect a wide spectrum of MINI and LINI documents to determine the correctness of the PRI rankings. As expected, this limitation of the exploratory study also limits the results presented in this chapter. Unfortunately, this limitation has affected my ability to come to a definitive answer to my third thesis claim, which states that PRI can identify documents that need to be inspected that are not typically identified by volunteering.

6.2 Inspections

Throughout the duration of this research, starting in September 2004 till June 2005, I led CSDL in 20 different inspections. On average 5 to 6 CSDL members participated in each of the inspections. In addition, with an estimated cost of 2 hours¹ per inspection per member, CSDL spent about 220 hours dedicated to inspection. Although 220 hours seems substantial, I've estimated that we would require about 3,500 hours to inspect the entire Hackystat system, which would be a very unrealistic demand on CSDL's resources for this research. Therefore, as I've previously stated, my study investigated a small percentage of the system in hopes that a cross-section provides adequate and acceptable results.

Ten of the twenty inspections were designated to be part of the methodology used evaluate my thesis claims. Unfortunately, one of the 10 inspections that were designated for the study, namely Inspection 10, had to be excluded from the results, because the code that was inspected was written entirely in C++ and could not be ranked by PRI².

Table 6.1 displays a general overview of the inspection results for each inspection conducted under the study procedure. Taking a closer look at the inspection results shows that the inspections have found a considerable amount of valid defects. Furthermore, notice that a large percentage, 40.2 percent to be exact, of the defects are high-severity defects. High-severity defects are defined as defects with a severity equal to "Critical" or "Major" and with a resolution not equal to "Invalid Won't Fix" and "Valid Duplicate". See Section 5.1 for a discussion on how defects are collected using CSDL's inspection process and the Jupiter tool.

¹According to CSDL's inspection process, each member should spend about an hour individually inspecting the document. Another hour is spent in the inspection meeting.

²I allowed developers to select any code they were most concerned about, therefore in this instance the inspection could not be used in my study.

Table 6.1. All Inspection - Results by Severity

Inspection	Critical	Major	Normal	Minor	Trivial	Total
8	1	7	11	2	3	24
9		16	13	4	4	37
11	1	14	13	14	1	43
12		13	12	6		31
13	1	7	1	2		11
14	1	13	12			26
15		1	6	2	1	10
16		3	4	3		10
17		7	6	1	3	17
Total	4	81	78	34	12	209
Percentage	1.9 %	38.8 %	37.3 %	16.3 %	5.7 %	

Using the Type property associated with the valid defects provides a different view of the inspection results. Table 6.2 shows the results listed by different defects types provided by Jupiter. Note that the types “Missing”, “Irrelevant”, and “Other” were omitted from this table, because the use of these types were very limited, accounting for only 12 of the 209 defects. According to these results, Coding Standards and Program Logic account for more than half the valid defects discovered in CSDL’s inspections. Although not officially stated in CSDL’s inspection process and is largely based on the developers’ subjective opinion, Coding Standards defects are generally reserved for defects resulting in defects in code formatting, variable and method naming conventions, Javadoc documentation, and the alike. Program Logic defects are generally problems associated with Object-Oriented design, algorithmic, calculation errors, and other problems associated with the actual functionality of the software code.

Further analyzing the Type and Severity properties of the defects shows that Program Logic defects are of more concern than any other type. According to Table 6.3 and also shown in Figure 6.1, Program Logic defects account for the majority of the high-severity defects. On the other hand, Coding Standard defects account for the majority of the low-severity defects. This result indicates that Program Logic defects are generally high-severity problems. Therefore, an inspection that finds more Program Logic defects generally also has a larger number of high-severity defects.

Table 6.2. All Inspection - Results by Type

Inspection	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
8	10	6	1	2	5	
9	9	13	3	4	3	4
11	22	7	3	2		7
12	14	4	1	4	6	2
13	5	6				
14	9	4	1	8	1	2
15	6	2			1	
16			2	1	3	3
17	2	2	5		2	
Total	77	44	16	21	21	18
Total	36.8 %	21.1 %	7.7 %	10.0 %	10.0 %	8.6 %

Table 6.3. All Inspections - Results by Type and Severity

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical		3		1		
Major	13	28	11	8	6	10
Normal	31	12	4	12	13	4
Minor	24	1	1		1	4
Trivial	9				1	
Total	77	44	16	21	21	18

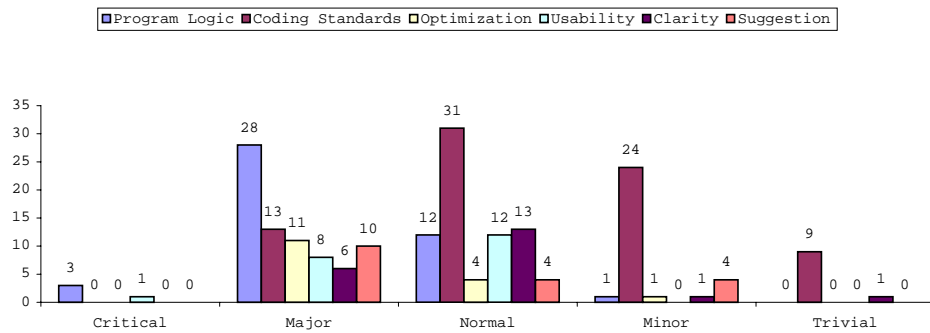


Figure 6.1. All Inspections - Results by Severity and Type

6.3 Result Terminology

Throughout the rest of this chapter, I will be using the terms *MINI-p*, *MINI-d*, and *LINI-p*. These terms represent three different groups of packages that were inspected in this study. I will use these three different groups of packages to explain the results and charts presented in the following sections.

- **MINI-p** represents inspections 8 and 9. According to the PRI rankings generated by the Hackystat PRI Extension, these inspections were conducted on MINI packages. In addition, according to the developers' rankings, these were also MINI packages. The developer rankings were obtained in the Pre-Selection-Questionnaire.
- **MINI-d** represents inspections 11, 12, and 14. According to the developers' rankings, these inspections were conducted on MINI packages. The developer rankings were obtained in the Pre-Selection-Questionnaire. However, according to the PRI rankings generated by the Hackystat PRI Extension, these packages were LINI. This group is interesting, because the developers and PRI ranking disagreed.
- **LINI-p** represents inspections 13, 15, 16, and 17. According to the PRI rankings generated by the Hackystat PRI Extension, these inspections were conducted on LINI packages. However, no developer rankings were obtained for these packages.

In addition, to distinguish between the two different methods used in this study to rank packages, I will use the terms *dMINI*, *dLINI*, *pMINI*, and *pLINI*. These terms distinguish between packages that were ranked by the Hackystat PRI Extension and packages that were ranked by the developers. The terms *MINI* and *LINI* will still be used with discussing the general meaning of *More In Need of Inspection* and *Less In Need of Inspection*.

- **dMINI and dLINI** represents the MINI and LINI packages according to the developers' opinion.
- **pMINI and pLINI** represents the MINI and LINI packages according to the Hackystat PRI Extension.

6.4 Thesis Claim 1

Claim: *MINI documents will generate more high-severity defects than LINI documents.*

The results presented in this section will provide evidence to support this claim. The supporting evidence for this claim is apparent in four separate results. The first supporting evidence is the results shown in Figure 6.2, which charts the total number of defects and the number of high-severity defects. Second, the results shown in Figure 6.6, which charts the responses from the Post-Inspection-Questionnaire. Third, the results shown in Figure 6.10, which charts the various defect types. Fourth, the results shown in Figure 6.12, which charts the average review active time from all participants. Each of these results is independent supporting evidence that provides varying levels of support for Thesis Claim 1.

6.4.1 Inspection Results by Severity

Figure 6.2 shows the results of the Severity property associated with the defects found in the inspections conducted on the three different groups, MINI-p, MINI-d, and LINI-p.

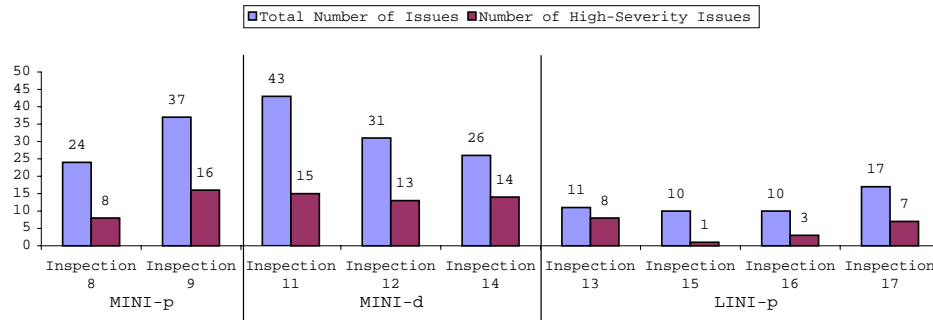


Figure 6.2. Inspection Results - Severity

Figure 6.3 provides the average number of defects and average number of high-severity defects in the MINI-p, MINI-d, and the LINI-p groups. The results in the Figure show that on average the MINI-p group generated about 18 more defects of any severity compared to the LINI-p group. Furthermore, on average the MINI-p group have generated about 8 more high-severity defects than the LINI-p group. This result provides supporting evidence for Thesis Claim 1. The fact that the MINI-p and MINI-d groups generated more defects than the LINI-p group provides supporting evidence that the Hackstat PRI Extension and developers can identify MINI documents.

However, in the case of the MINI-d group, where the developers indicated a MINI ranking but PRI indicated a LINI ranking, the results provide evidence that the developers were correct and the PRI ranking was incorrect. In other words, this data suggests the presence of possible 'false negatives' by PRI. More inspections and evaluations are needed to determine if PRI can correctly rank the packages in the MINI-d group.

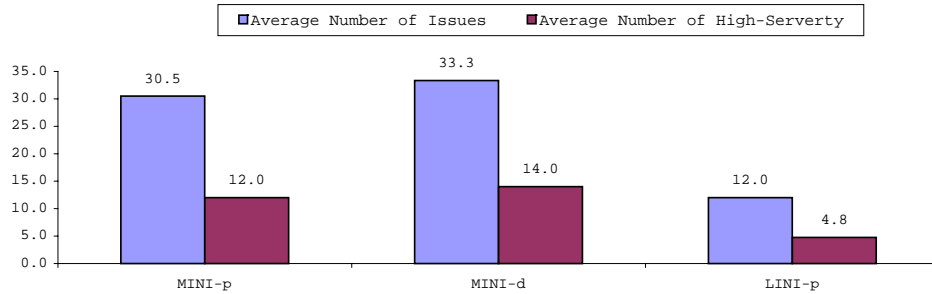


Figure 6.3. Inspection Results - Average Severity

Figure 6.4 shows the percentage of high-severity defects over total defects for each of the inspections. Figure 6.5 shows the average percentage for the MINI-p, MINI-d, and LINI-p groups. According to Figure 6.5, there does not appear to be a significant difference between the three groups in terms of the percentage of high-severity defects. However, it is my contention that this percentage is not useful in determining the correctness of the MINI and LINI determination, because the number of total defects does not stay constant throughout each inspection. For example, Inspection 13 has the highest percentage, however referring back to Figure 6.2 will show that this inspection contained the third lowest number of total defects and fourth lowest number of high-severity defects. Therefore, I believe that this percentage should not be used to compare the results of individual inspections.

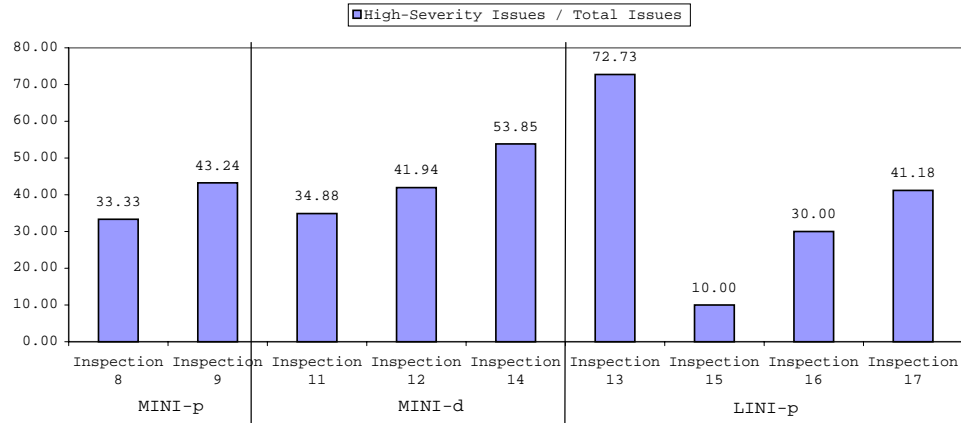


Figure 6.4. Inspection Results - Severity Percentage

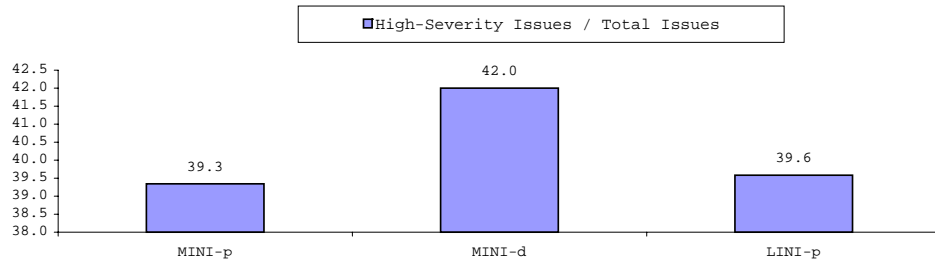


Figure 6.5. Inspection Results - Average Severity Percentage

6.4.2 Post-Inspection-Questionnaire Results

Figure 6.6 provides the results obtained by Question 1 in the Post-Inspection-Questionnaire. I administered this questionnaire after the meeting for each inspection and this question asks the participants whether the package, under inspection, needed to be inspected. Yes, indicates that the package needed to be inspected. No, indicates that the package did not need to be inspected. Figure 6.7 provides the average number of responses for the MINI-p, MINI-d, and LINI-p groups.

The results shown in Figures 6.6 and 6.7 continues the trend shown in Figure 6.2 and further confirms that the packages in the MINI-p and MINI-d groups were in fact MINI and the packages in the LINI-p group were in fact LINI. In addition, the results associated with the MINI-d group is interesting because it shows that the PRI ranking function can lead to false negatives. In other words, in the MINI-d group, PRI ranked the packages as LINI when the data indicates that they were actually MINI.

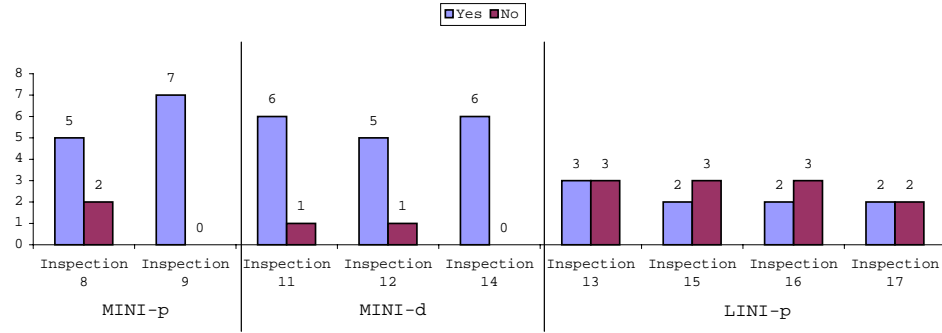


Figure 6.6. Post Inspection Questionnaire Question 1 - Provides responses to the question; *The package needed to be inspected?* Yes, indicates that the package needed to be inspected. Not, indicates that the package did not need to be inspected.

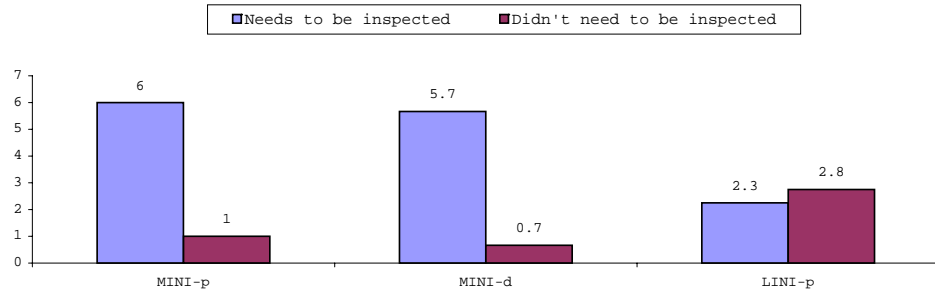


Figure 6.7. Post Inspection Questionnaire - Question 1 - Average Responses

Figure 6.8 provides the results obtained by Question 3 in the Post-Inspection-Questionnaire, which asks the participants whether the package’s software quality will increase after the defects that were found are fixed. Figure 6.9 provides the average number of responses for the MINI-p, MINI-d, and LINI-p groups. Although, the 25 percent of the participants felt that the inspection of LINI-p group packages would not increase the quality of the package, the majority felt that the MINI-p, MINI-d, and LINI-p groups would increase in quality. It is also interesting to note that regardless of the participants’ opinions about the package needing to be inspected (Figures 6.6 and 6.7), the majority of the participants felt that the package would improve in quality once the defects are resolved. This appears to mean that even LINI packages can improve in quality, which is something I had not considered previously.

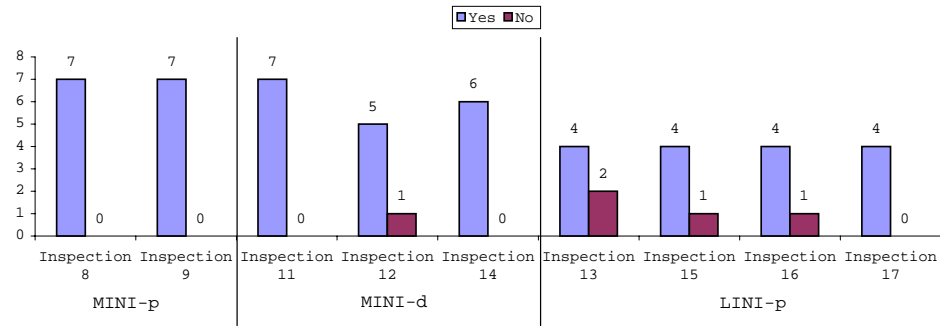


Figure 6.8. Post Inspection Questionnaire Question 3 - Provides responses to the question; *After the discovered defects are fixed, the package's software quality will increase?* Yes, indicates that the package's software quality will increase. No, indicates that the package's software quality will not increase.

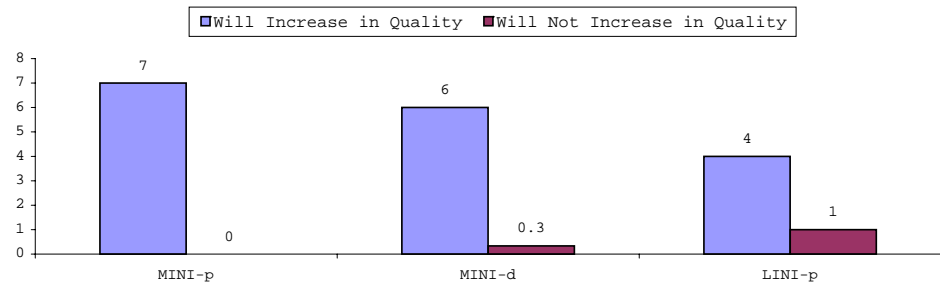


Figure 6.9. Post Inspection Questionnaire - Question 3 - Average Responses

6.4.3 Inspection Results by Type

Figure 6.10 provides the number of Program Logic and Coding Standards defects found in each of the inspections. Figure 6.11 provides the average number of Program Logic and Coding Standards defects for the MINI-p, MINI-d, and LINI-p groups. The results in Figure 6.11 show three interesting findings. First, the MINI-p group has about 7 more Program Logic and 6 more Coding Standards defects than the LINI-p group. Second, the MINI-d group has 2.5 more Program Logic and 12 more Coding Standards defects than the LINI-p group. Third, the MINI-p group has about 4.5 more Program Logic and about 6 less Coding Standard defects than the MINI-d group. The variation of the Program Logic and Coding Standard defects between the two MINI groups could indicate that the selection method, PRI or developer ranking, could identify documents that have different types of defects. Obviously, more inspections using both selection techniques will be needed to provide statistically viable results.

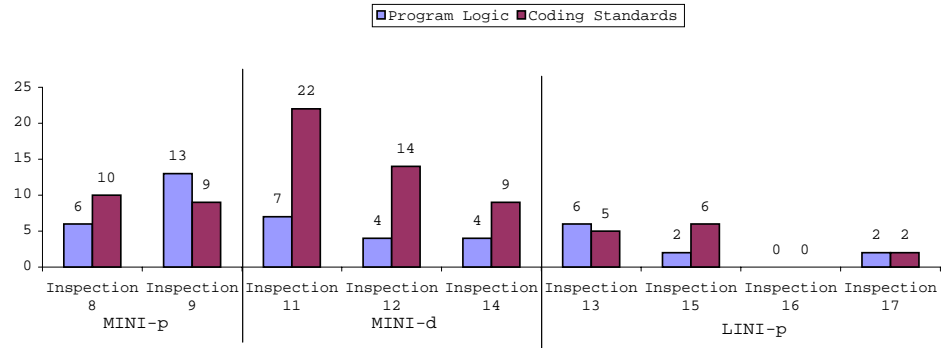


Figure 6.10. Inspection Results - Type

The main supporting evidence for Thesis Claim 1, provided by Figure 6.11, is that the MINI groups generated more Program Logic and Coding Standards defects than the LINI-p group. For example, Inspection 9, which is in the MINI-p group, has the highest number of Program Logic defects and one of the highest percentages of Program Logic to Total Defects than any other single inspection. As previously stated, Coding Standards defects are generally considered to be a violation of standard coding styles. For example, the CSDL organization strives to follow the rules defined in the book, “The Elements of Java Style.” Code Standards defects, although obviously very important to CSDL, generally do not cause runtime defects in software. On the other hand, Program Logic defects can cause possible runtime defects.

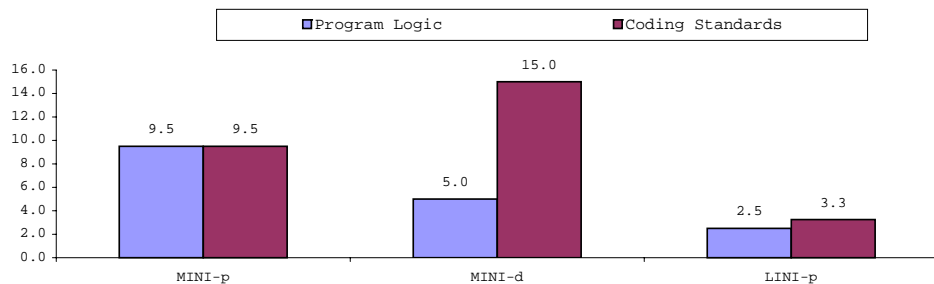


Figure 6.11. Inspection Results - Type

6.4.4 Inspection Results by Review Active Time

Hackstat provides various sensors for Interactive Development Environments (IDEs), like JBuilder, Eclipse, Emacs, and Visual Studio .Net. One of the measures collected by these

sensors is Active Time. The Active Time measure is a proxy of development effort spent interacting with a specific tool, in this case an IDE. A sensor for the Jupiter tool is also available, which provides an alternative to the traditional active time concept called Review Active Time. Review Active Time measures the time spent interacting with Jupiter during the preparation phase of CSDL’s inspection process. Figure 6.12 presents the average active time (in minutes) that inspectors spent during their individual preparation phase for each of the inspections.

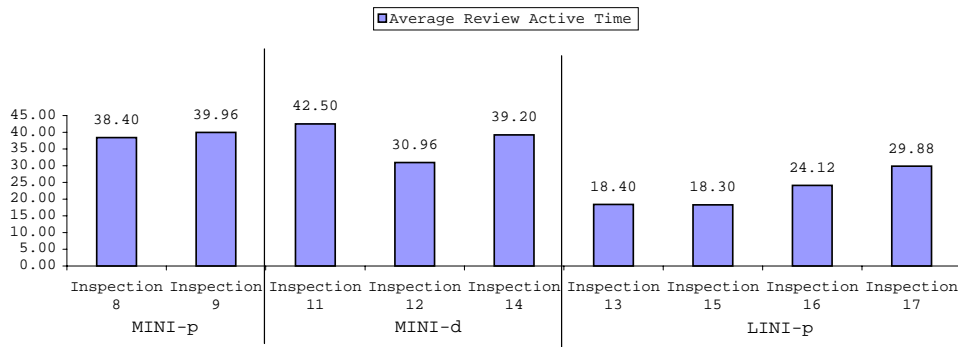


Figure 6.12. Inspection Results - Review Active Time

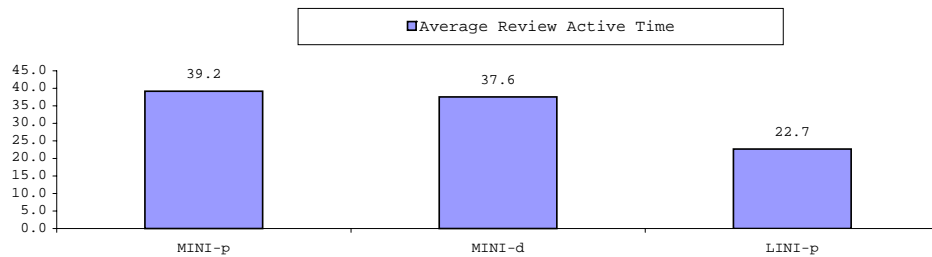


Figure 6.13. Inspection Results - Review Active Time

An interesting result is shown in Figure 6.13. According to the figure, the average review active time is about 15 minutes longer for the MINI-p and MINI-d groups than the LINI-p group. However, be aware that the review active time only provides the amount of time interacting with Jupiter. It does not account for the total time required to do an inspection. For example, an inspector might spend time reading code, documentation, searching for related code, and the alike without having to interact with the Jupiter tool. Therefore, this figure, in some ways, validates that the

LINI-p group contains the least amount of defects, because less time was required to interact with Jupiter to log these defects.

6.4.5 Inspection Results by Averages

Figure 6.14 provides a consolidated look at the previous four independent results to summarize the supporting evidence for Thesis Claim 1.

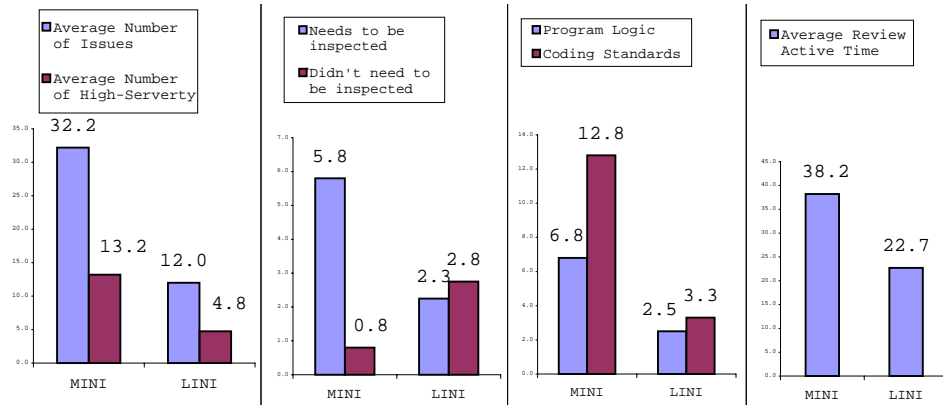


Figure 6.14. Inspection Average Results - Provides the averages of the previous inspection results. Each section represents an individual result obtained by analyzing inspection data.

Each section of the chart contains the average values obtained from the inspections. To summarize the findings and to look specifically at the difference between MINI and LINI results, I have combined the MINI-p and MINI-d groups. The first section shows that the average number of issues and the average number of high-severity issues are considerably higher for MINI packages than LINI packages. The second section shows that the developers thought (on average, 6 of 7 responses) that the MINI packages needed inspection. In addition, the developers thought (one average, 3 of 5 responses) that the LINI packages did not need inspection. The third section shows that the average numbers of Program Logic and Coding Standards defects are much higher for MINI packages than LINI packages. The fourth section shows that the average review active time spent by each inspector was 16 minutes longer for MINI packages than LINI packages.

6.5 Thesis Claim 2

Claim: *PRI can enhance the volunteer-based document selection process*

The results presented in this section will show supporting evidence for this claim. As my Thesis Committee pointed out during my proposal of this thesis research, I've carefully stated this claim with the word "can." I used the word "can" instead of "will," because I felt that CSDL's current volunteer-based selection process was entirely based on the developers' subjective opinion of code and therefore I had little knowledge on how PRI would affect that opinion. In addition, I believed that the factors that are used in deciding to select a document would have a large variation. In fact, the results of the study have shown strong evidence of this. Therefore, I did not have much confidence in saying *will* opposed to *can*. Furthermore, it turns out that my study procedure, which allowed developers to decide whether they wanted to use PRI or not was flawed. In all cases where the developers selected a package for inspection, they already had a good idea on what package they felt needed inspection and did not use PRI to help make that decision. Regardless of the procedure flaw, the goal of my study was to understand the developers' selection process. In my opinion, the portion of the study procedure to accomplish this goal was designed well.

The supporting evidence that PRI can aid the selection process is apparent in the selection trends of the participants, PRI's ability to correctly rank package as MINI and LINI, and educational value of inspections.

6.5.1 Selection Trends

Traditional inspection processes state that developers have to volunteer their code for inspection. Yet, little is known about the mental decisions required to select and volunteer a particular piece of code. The results presented in this section, show evidence that developers have extremely varying ideas of how to select documents for inspection. And one can only conclude that an inconsistent approach will cause inconsistent results. Therefore, I believe PRI can aid the selection process by providing a priority ranking of workspaces, product and process measures, and a more consistent approach to selecting documents for inspection.

The Use of Hackystat to Select Documents for Inspection

Ironically, even though the CSDL developers have an immense amount of software product and process data about their software in our Hackystat system, not one developer used it in the

past to gain any insights about their code before volunteering a package for inspection (See Question 5 in the Pre-Selection Questionnaire, Section D.5). This result leads me to believe that most developers have a very good understanding of their own code, or at least they think they do.

Factors that Influence their Selection

Question 3 and 4 in the Pre-Selection Questionnaire (See Sections D.3 and D.4), also provide evidence that developers use their own subjective opinions over actual software product measures. In these questions, there were only 3 of 12 responses that suggest software code that has low coverage, no unit tests, or high dependencies have any consequences on the likelihood that the code needs to be inspected. In addition, at least one participant selected code to inspect based on their knowledge of build failures ³.

The most consistent factor used by developers to select code for inspection is the “age” of the code. According the results, 9 of 12 responses indicate that newly created code, code that was developed by a new developer, and code that no one has seen before are the most frequent deciding factors when selecting code for inspection. This trend is also present in Question 8 in the Pre-Selection Questionnaire (See Section D.8). 11 of the 25 responses to Question 8, indicates “New Code” as the primary reason why they ranked a particular package as a *dMINI*⁴. On the other hand, old code and code that no one uses are the most frequent deciding factors for not select a document for inspection. 12 of the 19 responses indicated these two factors when ranking a particular package as a *dLINI*⁵.

Based on these results, I believe that PRI can provide the developers with more useful information, in the form of product and process measures and PRI rankings, to select documents for inspection.

Ranking Modules and Workspaces

The Pre-Selection-Questionnaire contained three questions that asked the participants to provide their subjective *dMINI* and *dLINI* rankings for top-level modules, all workspaces in the system, and workspaces they have recently authored. Not surprisingly, the results of these rankings vary from participant to participant. However, there were a few instances where the results were consistent.

³Part of Hackstat’s development process includes a continuous build occurring every night to ensure that the system compiles and passes various levels of testing.

⁴*dMINI* represents *MINI* packages ranked by developers.

⁵*dLINI* represents *LINI* packages ranked by developers.

Question 7 in the Pre-Selection-Questionnaire (See Section D.7), asks the participants to rank all workspaces in the system that they have recently authored. The results of this question do not have any variation because the responses were specific to each participant. A common result is that participants were able to rank their own code without much trouble. This supports my previous finding that the developers have an understanding of their own code. However, the question remains as to whether their understanding is correct.

Question 8 in the Pre-Selection-Questionnaire (See Section D.8), asks the participants to rank the top 5 dMINI modules ⁶ and the top 5 dLINI modules. Figure 6.15 shows the number of dMINI responses for each module. The modules with the most dMINI responses are hackyCGQM, hackyIssue, and hackyZorro. The results for these modules are very consistent, accounting for 15 out of 25 responses. However, the remaining responses were spread across of 9 different modules. In addition, there were 5 blanks (indicated by “??”), which means that the participants could not identify a module as a dMINI. According to these results, the participants can agree only on a couple of dMINI modules. Furthermore, the rest of the responses show too much variation to be able to identify the remaining top 5 dMINI modules.

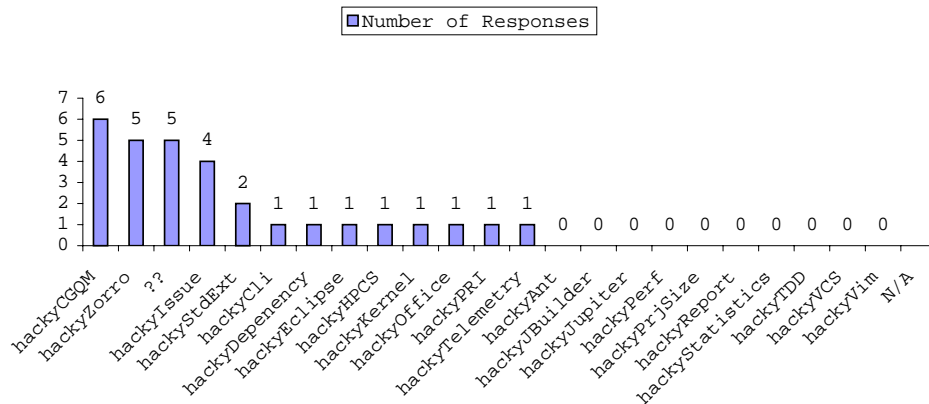


Figure 6.15. Question 8 Part 1 Responses - Provides the total number of responses that the participants felt were dMINI modules. ?? indicates that the participant did not know which module were dMINI.

Rankings of dLINI modules showed a larger variation. Figure 6.16 shows the number of dLINI responses for each module. The modules with the most responses are hackyStatistics and hackyReport. The rest of the responses are spread across 8 different modules. Furthermore, 11

⁶Modules are top-level parent workspaces that divide the Hackstat system into different portions. Usually, a single developer is responsible for a module.

responses were either blank (indicated by “??”) or N/A (which is interpreted as no module should be declared as a dLINI). These results are similar to the top 5 dMINI presented in the previous paragraph, but it appears that the developers have even less agreement in ranking dLINI modules.

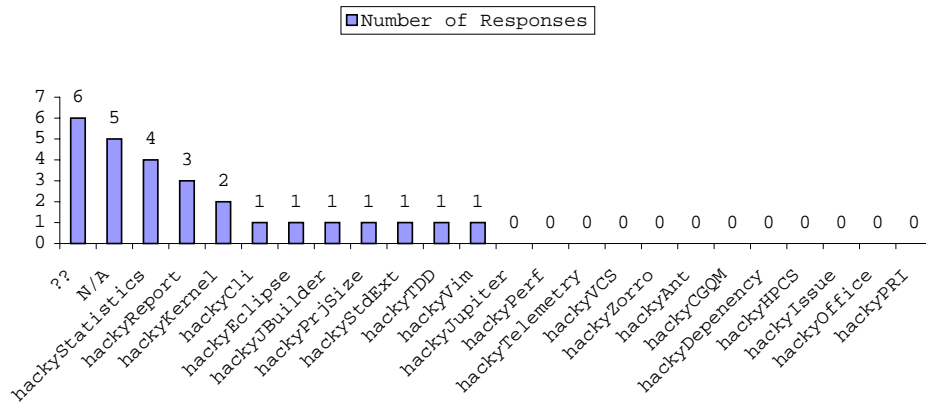


Figure 6.16. Question 8 Part 2 Responses - Provides the total number of responses that the participants felt were dLINI modules. ?? indicates that the participants did not know which modules were dLINI. N/A indicates that the participants felt no module should be declared dLINI.

Comparing Figures 6.15 and 6.16 yields the results shown in Figure 6.17. This figure helps illustrate two results. First, the highest level of agreement between the developers occurred in ranking the dMINI modules. The hackyCGQM module received a 100 percent (6 of 6 responses) agreement that it is a dMINI. And the hackyZorro module received an 83 percent (5 of 6 responses) agreement that it is a dMINI. On the other hand, highest agreement for the dLINI modules was 66 percent (4 of 6 responses). Second, there were 4 modules where the developers disagreed on the declaration of dMINI and dLINI. For example, hackyStdExt, hackyCli, hackyEclipse, and hackyKernel all had responses that they were both dMINI and dLINI. This result shows another way in which the developers ranked workspaces inconsistently.

Question 9 in the Pre-Selection-Questionnaire (See Section D.9), asks the participants to rank the top 5 dMINI and top 5 dLINI workspaces in the entire system. Of the three questions, this is by far the hardest task, because there are about 218 different workspaces in the system. It should be noted that almost all developers have authored code in other areas of the system that were not ranked in Question 7. For example, one developer has the most commits and active time for 22 workspaces in the system other than the workspaces used in Question 7.

To help the participants, I provided them with a full listing of all the workspaces. Even with that list, half of the participants either could not identify a fully qualified workspace and or just

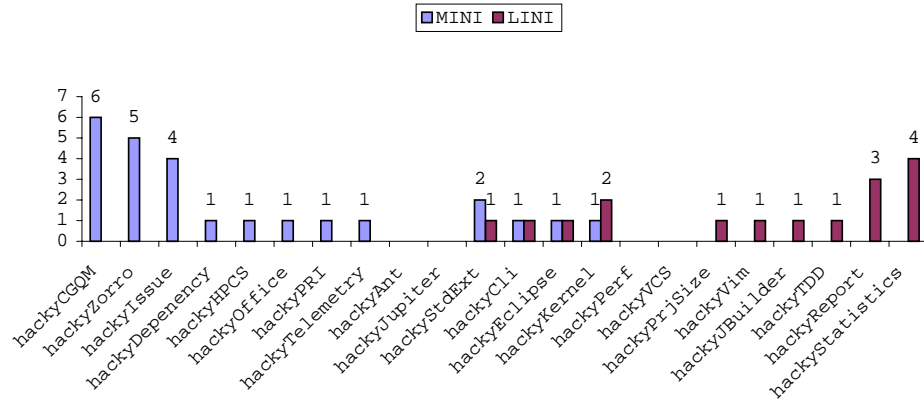


Figure 6.17. Question 8 Comparison of Responses - Provides the total number of responses that the participants felt were dMINI and dLINI modules.

simply provided a module name. In fact, several participants complained that they had no idea how to rank workspaces that they did not author. Furthermore, for an unknown reason, the factors of their decisions changed from using the “age” of the documents in Questions 3, 4, and 8, to guessing at the documents level of quality. Needless to say, the results of the rankings for both dMINI and dLINI workspaces were extremely varied. Only one workspace out of the 60 possible responses occurred more than once. This result indicates that while developers can rank workspaces that they have authored, they cannot provide much consistency for workspaces where their understanding is limited.

As previously stated, if PRI can provide more useful information, in the form of the PRI measures and PRI rankings, then that might lead to more consistent results when selecting documents for inspection in areas where their subjective knowledge is limited.

6.5.2 Validity of PRI’s MINI and LINI ranking

Although, none of the participants used PRI to select packages for inspection, I’ve designed the study procedures to be able to validate whether PRI could have helped the selection process. I’ve done this by doing two things. First, by obtaining the developers ranking of packages that they have authored according to their subjective opinion of the likelihood that the packages needed inspection (dMINI and dLINI). Second, by comparing the developer ranking with the PRI rankings generated by hackyPRI (pMINI and pLINI⁷). The results of this study show that two rankings agreed and three rankings disagreed (See Section D.7). The two rankings that agreed with one

⁷pMINI and pLINI represents MINI and LINI packages that were ranked by the Hackystat PRI Extension.

another were for the hackyReview and hackyIssue modules. By agreed, I mean both the developer rankings and PRI rankings were very similar, although not exactly the same. The three rankings that disagreed with one another were for the hackyCGQM, hackyZorro, and hackyTelemetry modules. By disagreed, I mean that the developer rankings and PRI rankings did not have any significant similarities.

It is interesting to note a couple of things about these modules. First, the hackyReview and hackyIssue modules implement very similar functionalities compared to each other. Second, the hackyCGQM, hackyZorro, and hackyTelemetry modules implement very different functionalities compared to any other modules in Hackystat. Obviously, these results do not have any statistically verifiable meaning at this point. But, this result shows that the PRI ranking function can be calibrated correctly for some modules and not for others.

As previously explained, the major limitation of this research is the lack of resources to thoroughly inspect a large percentage of the Hackystat system to validate MINI and LINI determinations. Therefore, because of this limitation I cannot conclude whether the developer rankings or PRI rankings were correct. More inspection resources would have been useful in studying the following situation. In some cases, the MINI and LINI rankings were flipped. For example, in the hackyZorro module, the developer ranked org.hackystat.stdext.zorro.control as a very high priority dMINI package and PRI ranked the same package as a low priority pLINI package. This package was inspected by CSDL (Inspection 12) and the results shown in the previous sections indicate that this package was a MINI. However, according to the PRI rankings there are many other packages in hackyZorro, which need inspection more. It would have been greatly beneficial to inspect a PRI-declared pMINI package to determine the correctness of the PRI rankings.

6.5.3 Educational Value of Inspection

Like traditional inspection processes, CSDL's current inspection process is based on a volunteering process. However, unlike most organizations, CSDL's members do not feel that the main goal of their inspection process is to remove defects, as evidence by their responses to Question 2 in the Pre-Selection-Questionnaire (See Section D.2 and Figure D.2). According to these results, most CSDL members do not agree with the statement: *The most important goal of the CSDL inspection process is to remove defects.* In fact, educational aspects of inspections are very important to CSDL. This is very apparent in Figure 6.18, which shows the results from Question 2 in the Post-Inspection Questionnaire that asks the inspectors if they have learned something from participating in the inspections.

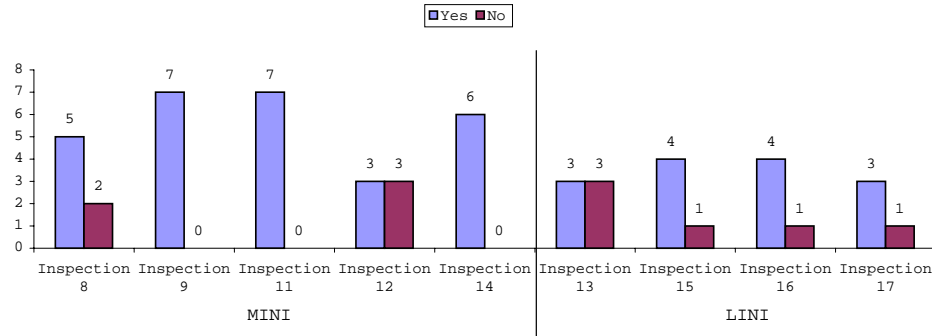


Figure 6.18. Post Inspection Questionnaire Question 2 - Provides responses to the question; *I learned something by participating in this inspection?* Yes, indicates that the developer learned something. Not, indicates that the the developer did not learn something.

In almost all cases, the majority of the inspectors learned something regardless of whether many defects were found. Furthermore, it seems that the educational value does not correlate to the number of defects found, the type of the defects, or the severity of defects. Therefore, one can conclude that inspecting high quality documents, for example packages in the LINI-p group, can provide as much educational value as inspecting packages in the MINI-p and MINI-d groups. This educational value could result in fewer defects in future code development, which is greatly beneficial in lowering the resources needed to inspect future documents [1]. Inspecting a document with very elegant coding, excellent documentation, and totally defect free could provide as much or even much more educational benefits than inspecting a MINI document.

This result indicates a couple of things. First, it might be the case that PRI is less beneficial for CSDL than other organizations that do not stress educational aspects in their inspection process. Second, the PRI ranking function that was created for this study, and specifically for CSDL, is incorrect. Instead of trying to identify MINI and LINI packages in the context of finding high-severity defects, maybe I should have focused more on the educational benefits of inspection.

6.6 Thesis Claim 3

Claim: *PRI can identify documents that need to be inspected that are not typically identified by volunteering.*

Unfortunately, after analyzing the results, I have realized that I have failed to create a viable study procedure to thoroughly evaluate this claim. It appears that the study procedures were

too centered on determining whether MINI documents contain more high-severity defects than LINI documents to leave much room for investigating evidence for this claim. Once again, I believe if I had conducted more inspections, then I would have more inspection results and be better able to address this issue.

6.6.1 Selection Trends

As previously discussed, developers have a much easier time selecting documents that they have authored. In addition, they tend to select packages based on the age of the code, regardless of the quality level indicated by various product and process measures available to the in their Hackystat server (excluding hackyPRI rankings). In fact, a large majority of developers did not mention anything about software quality when selecting packages. In addition, developers seem to struggle when selecting documents in which their subjective understanding of code was limited.

The only supporting evidence for this claim is that the developers view new code as MINI and old code as LINI. Based on the results, it seems that they most of the developers believe old code should never be inspected, which directly contradicts my thesis claim. Therefore, I still believe that there is some hope of validating my claim that PRI can identify MINI documents that are not typically identified by volunteering.

Chapter 7

Conclusions and Future Directions

The research conducted on the Priority Ranked Inspection process has shown supporting evidence that it can be beneficial for organizations with limited inspection resources. However, the conducted evaluation is very preliminary. In addition, there are a number of other future directions that are required to further the research of the PRI process.

7.1 Future Directions

The research presented on PRI in this thesis is the first of many possible steps that are needed to validate the potential of the Priority Ranked Inspection approach. Although PRI is centered on inspections, many other research fields, like software quality, defect prevention and prediction, software metrics, and the alike, play an equally important role. For example, the when calibrating PRI indicators, one can consult various software metrics and defect prediction literature to determine the threshold values that produce the best rankings. This section presents the many different future directions of this research.

7.1.1 More Evaluations

This 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. This issue will be left as a future direction. However, I believe the evaluation that was conducted

during this research was necessary to provide evidence that PRI is a worthy concept to try at other organizations.

7.1.2 Implementation of the Hackystat PRI Extension

In addition, future work is needed to generalize the hackyPRI extension so that it is possible for other organizations and projects. Currently, hackyPRI probably best supports the CSDL organization and the Hackystat project. Also, there were many future tasks associated with the implementation of the Hackystat PRI Extension that I've mentioned at the end of Chapter 4 that can still be addressed in future research. For example:

1. Solving the threats to data validity in hackyPRI.
2. Providing a configurable PRI indicator ranking with the JESS tool.
3. Providing other levels (i.e., modules, Java classes, methods) of rankings.
4. Linking PRI with Software Project Telemetry to track changes over time.
5. Implementing an automatic feedback loop of inspection results to help automatically calibrate indicators.

7.1.3 How to Determine MINI-Threshold

A major part of the PRI process that was not solved in this research is Step 1c. Step 1c is the third step in creating a PRI ranking function and states that the PRI ranking function should create a MINI-threshold, which declares all documents below the threshold as MINI and all above as LINI. The solution to this problem will be a major benefit to the PRI process, because it will provide an organization with the exact number of documents that should be inspected. They can use this information to schedule and plan inspections. For example, an organization can find that they must inspect 100 of their 500 documents and request the necessary inspection resources to do so.

7.1.4 Comparison of PRI with Code Smells and Crocodile

In Chapter 2, I explained that the PRI process is very similar to two software tools, Code Smells and Crocodile, which help identify the right areas of a system to inspect. A future direction of this research is the evaluation of the results obtained by all three approaches. For example, one

could generate the “rankings” from all three approaches on the same software system and pick different areas to conduct inspections on to determine the validity of each approach. Another possible evaluation could be studying the cost-effectiveness of each approach.

On the other hand, the PRI process is more robust than the two approaches, because it can include any type of product measures into the ranking function. Therefore, another possible future direction is the incorporation of the Code Smell and Crocodile measures into PRI.

7.1.5 Use of PRI in Other Quality Assurance Situations

Priority Ranked Inspection was originally created for purposes that span a number of quality assurance 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 research I have obviously decided to focus on one “quality tool”, namely software inspection. I chose inspections because the quality assurance 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.

7.2 Final Thoughts

The results of this research shows that it is very challenging to thoroughly evaluate PRI. It is also a little ironic that the sole purpose of PRI is to aid the inspection processes of organizations with limited resources, but at the same time, to evaluate PRI a thorough inspection of all ranked documents would provide the best results.

I firmly believe that the concept of PRI, which tries to identify the right documents to inspect, has promise, although future research is needed to provide more supporting evidence for that belief. Hopefully, one day Priority Ranked Inspection will be as well known and evaluated as much as other inspection processes like Software Inspections.

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

Pre-Selection Questionnaire

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 indicate your level of agreement to the following statements the best of your ability. In the following statements, the terms Inspections and Code Reviews are used interchangeably.

1. Inspections are an important part of the CSDL development process. (circle one)

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

2. The most important goal of the CSDL inspection process is to remove defects. (circle one)

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

3. Which of the following would most likely contain the most critical defects: (circle all that apply)

- (1) Newly created code
- (2) Code that has no (or very few) unit tests
- (3) Code that has low coverage
- (4) Code that was developed by a new developer
- (5) Code that only one developer has worked on
- (6) Code that has a large number of dependencies
- (7) Old code
- (8) Other: _____

4. When I request an inspection, I generally volunteer code that is: (circle all that apply)

- (1) Newly created code
- (2) Code that has low coverage
- (3) Code that no one has seen before
- (4) Code that has no unit test
- (5) Old code
- (6) Other: _____

5. When I request an inspection, I use Hackystat to help me pick a piece of code to inspect? (circle one)

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

6. Inspection should only occur on newly created code. Old code that has already been released does not need to be inspected. (circle one)

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

7. 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.bax` has the lowest priority ranking.

Ranking	Package
2	<code>com.example.foo</code>
1	<code>com.example.bar</code>
3	<code>com.example.baz</code>

Enter your own subjective rankings in this table. Please use values from 1 through 13 and do not use one number twice. After finishing the rankings we will discuss you justifications for those rankings.

Ranking	Package
	<code>org.hackystat.app.telemetry.analysis</code>
	<code>org.hackystat.app.telemetry.analysis.selector</code>
	<code>org.hackystat.app.telemetry.config</code>
	<code>org.hackystat.app.telemetry.config.core</code>
	<code>org.hackystat.app.telemetry.processor.evaluator</code>
	<code>org.hackystat.app.telemetry.processor.parser</code>
	<code>org.hackystat.app.telemetry.processor.parser.impl</code>
	<code>org.hackystat.app.telemetry.processor.query</code>
	<code>org.hackystat.app.telemetry.processor.query.expression</code>
	<code>org.hackystat.app.telemetry.processor.reducer</code>
	<code>org.hackystat.app.telemetry.processor.reducer.impl</code>
	<code>org.hackystat.app.telemetry.processor.reducer.util</code>
	<code>org.hackystat.app.telemetry.processor.stream</code>

When finished wait for further instructions. After finishing the rankings we will discuss you justifications for those rankings.

Ranking	Justification
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	118
12	
13	

8. To the best of your knowledge, please provide the top 5 modules that you think need to be inspected and the top 5 modules that you think do NOT need to be inspected. Below is a list of all Hackystat modules.

Modules that need to be inspected		
Ranking	Module	Justification
1		
2		
3		
4		
5		

Modules that do NOT need to be inspected		
Ranking	Module	Justification
1		
2		
3		
4		
5		

Hackystat Modules

- o hackyAnt
- o hackyCGQM
- o hackyCli
- o hackyDependency
- o hackyEclipse
- o hackyHPCS
- o hackyIssue
- o hackyJBuilder
- o hackyJupiter
- o hackyKernel
- o hackyPerf
- o hackyPRI
- o hackyPrjSize
- o hackyReport
- o hackyReview
- o hackyStatistics
- o hackyStdExt
- o hackyTDD
- o hackyTelemetry
- o hackyVCS
- o hackyVim
- o hackyZorro

9. To the best of your knowledge, please provide the top 5 workspaces that you think need to be inspected and the top 5 workspaces that you think do NOT need to be inspected. Attached is a list of all Hackystat workspaces.

Workspaces that need to be inspected		
Ranking	Workspace	Justification
1		
2		
3		
4		
5		

Workspaces that do NOT need to be inspected		
Ranking	Workspace	Justification
1		
2		
3		
4		
5		

Appendix C

Post-Inspection Questionnaire

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 indicate your level of agreement to the following statements the best of your ability. In the following statements, the terms Inspections and Code Reviews are used interchangeably.

1. This package needed to be inspected

- (1) Yes
- (2) No

2. I learned something from this inspection

- (1) Yes
- (2) No

3. The inspection of this package increased its quality (once all the issues are resolved/fixed).

- (1) Yes
- (2) No

Appendix D

Pre-Selection-Questionnaire Results

This section contains the results from the Pre-Selection-Questionnaire. Sections D.1 and D.2 contains the results from the general questions about CSDL's inspection process. Section D.3 to Section D.6 contains the results from the questions about the developers' document selection method. Section D.7 to Section D.9 contains the results from the developers' subjective rankings of three separate sets of packages.

D.1 Question 1

Question 1. Inspections are an important part of the CSDL development process. (Choose One)

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

Table D.1. Question 1 Responses

Participant	Response
1	Strongly Agree
2	Strongly Agree
3	Strongly Agree
4	Agree
5	Strongly Agree
6	Agree

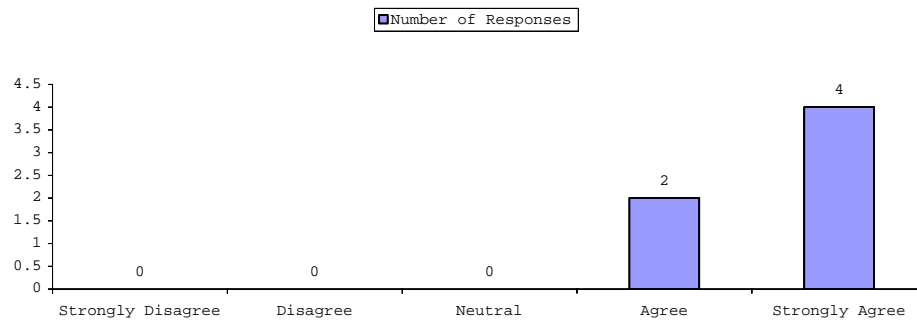


Figure D.1. Question 1 Responses

D.2 Question 2

Question 2. The most important goal of the CSDL inspection process is to remove defects. (Choose One)

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

Table D.2. Question 2 Responses

Participant	Response
1	Neutral
2	Agree
3	Neutral
4	Disagree
5	Agree
6	Neutral

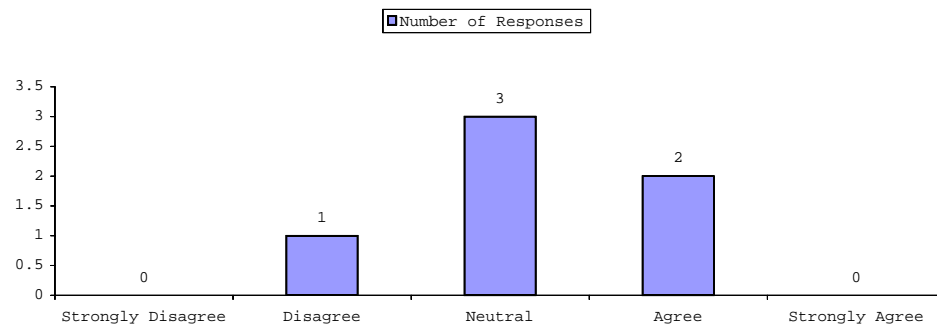


Figure D.2. Question 2 Responses

D.3 Question 3

Question 3. Which of the following would most likely contain the most critical defects (Choose all that apply)

- (1) Newly created code
- (2) Code that has no (or very few) unit tests
- (3) Code that has low coverage
- (4) Code that was developed by a new developer
- (5) Code that only one developer has worked on
- (6) Code that has a large number of dependencies
- (7) Old code
- (8) Other:

Table D.3. Question 3 Responses

Participant	Response
1	(1), (2), (3), (4), (5)
2	(1), (2), (3), (4)
3	(1), (4), (6)
4	Other: Strongly Refactored Code
5	(4)
6	(4)

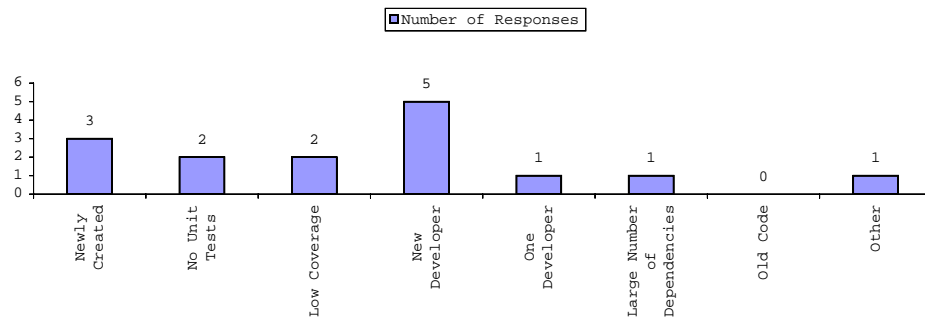


Figure D.3. Question 3 Responses

D.4 Question 4

Question 4. When I request an inspection, I generally volunteer code that is: (Choose all that apply)

- (1) Newly created code
- (2) Code that has low coverage
- (3) Code that no has seen before
- (4) Code that has no unit tests
- (5) Old code
- (6) Other:

Table D.4. Question 4 Responses

Participant	Response
1	(1), (3), (5)
2	(1), (3)
3	(1), (3)
4	Other: Fulfills a critical mission; is to be used by others
5	(3)
6	Other: Code I have no confidence in. Code that I feel is badly designed

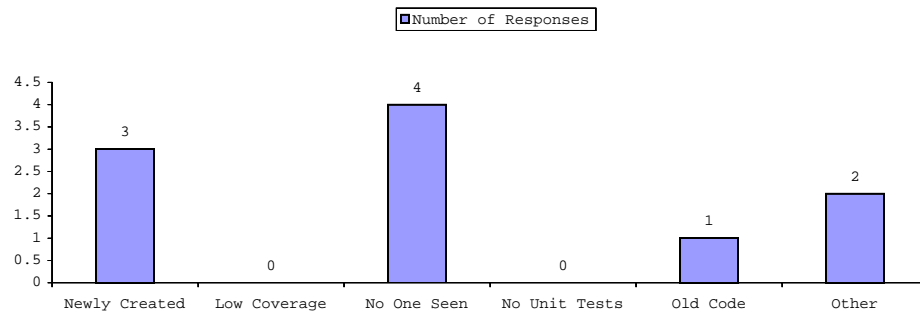


Figure D.4. Question 4 Responses

D.5 Question 5

Question 5. When I request an inspection, I use Hackystat to help me pick a piece of code to inspect? (Choose one)

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

Table D.5. Question 5 Responses

Participant	Response
1	Disagree
2	Disagree
3	Disagree
4	Disagree
5	Disagree
6	Disagree

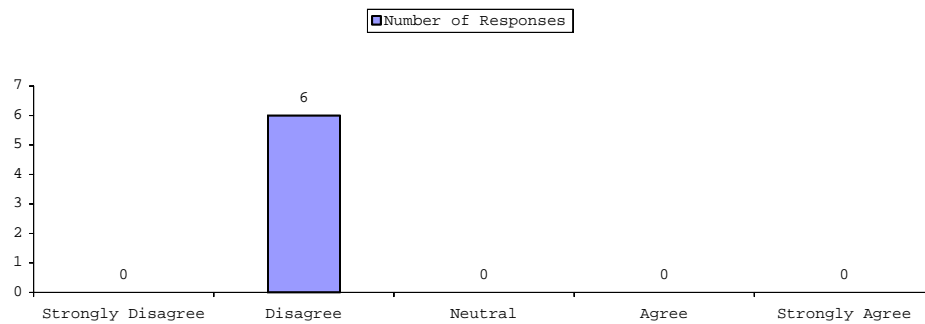


Figure D.5. Question 5 Responses

D.6 Question 6

Question 6. Inspection should only occur on newly created code. Old code that has already been released does not need to be inspected. (Choose one)

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

Table D.6. Question 6 Responses

Participant	Response
1	Strongly Disagree
2	Agree
3	Disagree
4	Disagree
5	Neutral
6	Neutral

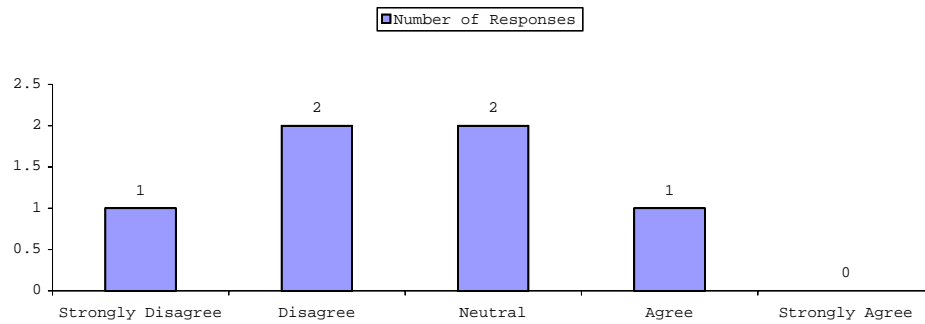


Figure D.6. Question 6 Responses

D.7 Question 7

Question 7. Please provide a numerical ranking of the packages provided that represent what packages you believe should be inspected. Enter your own subjective rankings. Please use values from 1 through N (N denotes the number of packages in the developers list of packages) and do not use one number twice.

D.7.1 hackyReview

The response to this question contains two results. First, a developer ranking, which is provided in Table D.7. Second, a PRI ranking, which is provided in the Figure D.7. Furthermore, the results of the developer rankings were used to help aid the developer in selecting a package for inspection. Therefore, for Inspection 8, the packages `org.hackystat.stdext.analysis.stream` and `org.hackystat.stdext.analysis.cache` were selected.

In this particular case, the developer ranking and PRI ranking agreed that those two packages were MINI packages, relative to other packages in the same module.

Table D.7. hackyReview Developer Ranking

Ranking	Package
1	<code>org.hackystat.app.review.analysis.stream</code>
2	<code>org.hackystat.app.review.analysis.cache</code>
3	<code>org.hackystat.app.review.analysis</code>
4	<code>org.hackystat.app.review.analysis.comparison</code>
5	<code>org.hackystat.app.review.analysis.summary</code>
6	<code>org.hackystat.app.review.issue.reducer</code>
7	<code>org.hackystat.app.review.activity.reducer</code>
8	<code>org.hackystat.app.review.issue.dailyproject</code>
9	<code>org.hackystat.app.review.activity.dailyproject</code>
10	<code>org.hackystat.app.review.issue.dailyanalysis</code>
11	<code>org.hackystat.app.review.activity.dailyanalysis</code>
12	<code>org.hackystat.app.review.issue.std</code>
13	<code>org.hackystat.app.review.activity.std</code>

Figure D.7. hackyReview PRI Ranking - Inspection 8

Workspaces (2011):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Commit First	Last Commit	Commit Cont	Re View	Last Re View	Open Issue	Closed Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyReview\src\org\hackyestat\app\review\issue\sd\	733	developer5 (time=0.42, cont=7)	0.42 h	0.00 h	21-Mar-2005	29-Mar-2005	1	11-Sep-2004	30-Mar-2005	1	0	0	0	0	LOC=221, Class=3, Method=19	LOC=72, Class=1, Method=2	in=1, out=10	Pass=0, Fail=32, Error=12	100.00 %
hackyReview\src\org\hackyestat\app\review\issue\sd\	732	developer5 (time=0.83, cont=19)	1.50 h	0.50 h	11-Sep-2004	30-Mar-2005	3	11-Sep-2004	30-Mar-2005	3	0	0	0	0	LOC=527, Class=4, Method=19	LOC=241, Class=2, Method=15	in=2, out=22	Pass=0, Fail=224, Error=149	100.00 %
hackyReview\src\org\hackyestat\app\review\activity\sd\	718	developer5 (time=1.08, cont=6)	1.08 h	0.50 h	07-Sep-2004	09-Mar-2005	1	11-Sep-2004	28-Mar-2005	2	0	0	0	0	LOC=231, Class=1, Method=14	LOC=113, Class=1, Method=4	in=9, out=11	Pass=0, Fail=32, Error=29	100.00 %
hackyReview\src\org\hackyestat\app\review\analysis\	716	developer5 (time=0.67, cont=95)	60.67 h	3.83 h	14-Feb-2005	05-Apr-2005	1	18-Feb-2005	04-Apr-2005	1	0	0	0	0	LOC=1082, Class=1, Method=71	LOC=196, Class=1, Method=8	in=20, out=35	Pass=0, Fail=14, Error=42	94.12 %
hackyReview\src\org\hackyestat\app\review\activity\dailyproject\	711	developer5 (time=7.00, cont=18)	8.58 h	0.50 h	24-Sep-2004	28-Mar-2005	2	11-Sep-2004	02-Apr-2005	2	0	0	0	0	LOC=544, Class=2, Method=26	LOC=121, Class=1, Method=6	in=2, out=21	Pass=0, Fail=37, Error=66	91.67 %
hackyReview\src\org\hackyestat\app\review\issue\dailyproject\	709	developer5 (time=10.58, cont=18)	16.00 h	1.83 h	24-Sep-2004	27-Mar-2005	2	11-Sep-2004	30-Mar-2005	2	0	0	0	0	LOC=898, Class=4, Method=45	LOC=106, Class=2, Method=7	in=4, out=27	Pass=0, Fail=137, Error=93	81.40 %
hackyReview\src\org\hackyestat\app\review\activity\reducer\	677		0.25 h	0.17 h	08-Oct-2004	24-Mar-2005	2	11-Oct-2004	30-Mar-2005	3	0	0	0	0	LOC=252, Class=2, Method=9	LOC=124, Class=1, Method=7	in=2, out=23	Pass=0, Fail=47, Error=68	100.00 %
hackyReview\src\org\hackyestat\app\review\activity\dailyanalysis\	658		1.83 h	0.08 h	05-Oct-2004	30-Mar-2005	2	11-Sep-2004	02-Apr-2005	2	0	0	0	0	LOC=411, Class=4, Method=20	LOC=83, Class=2, Method=2	in=6, out=28	Pass=0, Fail=29, Error=11	100.00 %
hackyReview\src\org\hackyestat\app\review\analysis\summary\	647	developer5 (time=3.85, cont=36)	6.83 h	3.58 h	14-Feb-2005	04-Apr-2005	1	18-Feb-2005	04-Apr-2005	2	0	0	0	0	LOC=441, Class=1, Method=21	LOC=23, Class=1, Method=3	in=2, out=25	Pass=0, Fail=17, Error=175	94.74 %
hackyReview\src\org\hackyestat\app\review\issue\dailyanalysis\	604	developer5 (time=0.08, cont=1)	0.08 h	0.00 h	27-Mar-2005	27-Mar-2005	1	27-Mar-2005	27-Mar-2005	1	0	0	0	0	LOC=135, Class=2, Method=10	LOC=43, Class=1, Method=1	in=1, out=18	Pass=0, Fail=16, Error=5	100.00 %
hackyReview\src\org\hackyestat\app\review\analysis\stream\	587	developer5 (time=2.42, cont=8)	2.42 h	0.58 h	29-Mar-2005	05-Apr-2005	1	30-Mar-2005	02-Apr-2005	1	0	0	0	0	LOC=178, Class=4, Method=10	LOC=57, Class=2, Method=4	in=4, out=14	Pass=0, Fail=20, Error=0	100.00 %
hackyReview\src\org\hackyestat\app\review\analysis\cache\	586	developer5 (time=10.33, cont=14)	10.33 h	1.33 h	29-Mar-2005	05-Apr-2005	1	30-Mar-2005	04-Apr-2005	1	0	0	0	0	LOC=476, Class=5, Method=35	LOC=128, Class=4, Method=10	in=10, out=29	Pass=0, Fail=98, Error=9	96.43 %
hackyReview\src\org\hackyestat\app\review\analysis\comparison\	565	developer5 (time=8.17, cont=17)	8.17 h	1.58 h	23-Feb-2005	03-Apr-2005	1	24-Feb-2005	04-Apr-2005	1	0	0	0	0	LOC=748, Class=1, Method=49	LOC=136, Class=1, Method=3	in=21, out=39	Pass=0, Fail=2, Error=2	40.91 %
Stats																			
Average PRI Ranking																			
664.85																			

D.7.2 hackyIssue

The response to this question contains two results. First, a developer ranking, which is provided in Table D.8. Second, a PRI ranking, which is provided in the Figure D.8. Furthermore, the results of the developer rankings were used to help aid the developer in selecting a package for inspection. Therefore, for Inspection 9, the package `org.hackystat.stdext.issue.reducer` was selected.

In this particular case, the developer ranking and PRI ranking agreed that the package was a MINI package, relative to other packages in the same module.

Table D.8. hackyIssue Developer Ranking

Ranking	Package
1	<code>org.hackystat.stdext.issue.reducer</code>
2	<code>org.hackystat.stdext.issue.dailyproject</code>
3	<code>org.hackystat.stdext.issue.analysis.issueprojectdetails</code>
4	<code>org.hackystat.stdext.issue.sdt</code>

Figure D.8. hackyIssue PRI Ranking - Inspection 9

Workspaces (204):	Ranking Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Commit: First Commit	Last Commit	Commit Cont	Last Review	Open Issue	Closed Issue	File Metric	Test File Metric	Depend	Unit Test	Unit Test Result	Coverage
hackyIssue\src\org\hackystat\stdectx\issue\analysis\issueprojectdetails\	684 developer2 (time=3.17, comts=9)	3.17 h	0.17 h	19-Feb-2005	04-Apr-2005	1	20-Feb-2005	22-Feb-2005	1	0	0	1	LOC=228, Class=2, Method=5	LOC=32, Class=1, Method=1	in=1, out=21	10	Pass=0, Fail=0, Error=4	100.00 %
hackyIssue\src\org\hackystat\stdectx\issue\gallyproject\	659	15.75 h	0.58 h	13-Feb-2005	05-Apr-2005	2	14-Feb-2005	10-Mar-2005	2	0	0	0	LOC=535, Class=2, Method=4	LOC=66, Class=1, Method=1	in=3, out=23	40	Pass=0, Fail=3, Error=7	100.00 %
hackyIssue\src\org\hackystat\stdectx\issue\prod\	591	3.83 h	0.42 h	10-Feb-2005	05-Apr-2005	2	08-Feb-2005	10-Mar-2005	2	0	0	1	LOC=364, Class=1, Method=3	LOC=63, Class=1, Method=2	in=5, out=11	20	Pass=0, Fail=0, Error=4	52.17 %
hackyIssue\src\org\hackystat\stdectx\issue\reducer\	381 developer6 (time=18.42, comts=2)	18.42 h	0.00 h	28-Feb-2005	05-Apr-2005	1	09-Mar-2005	10-Mar-2005	1	0	2	0	LOC=239, Class=1, Method=8	LOC=0, Class=0, Method=0	in=0, out=20	0	Pass=0, Fail=0, Error=0	11.11 %
Stats																		
Average PRI Ranking																		
578.75																		

D.7.3 hackyCGQM

The response to this question contains two results. First, a developer ranking, which is provided in Table D.9. Second, a PRI ranking, which is provided in the Figure D.9. Furthermore, the results of the developer rankings were used to help aid the developer in selecting packages for inspection. Therefore, for Inspection 11, the packages `org.hackystat.app.cgqm.interfaces.executables`, `org.hackystat.app.cgqm.interfaces.results`, and `org.hackystat.app.cgqm.implementations.executables` were selected.

In this particular case, the developer ranking and PRI ranking disagreed that the packages were MINI packages, relative to other packages in the same module. However, because only one inspection was conducted in this module, it is not known whether the developer rankings or PRI rankings were incorrect.

Table D.9. hackyCGQM Developer Ranking

Ranking	Package
1	<code>org.hackystat.app.cgqm.interfaces.executables</code>
2	<code>org.hackystat.app.cgqm.interfaces.results</code>
3	<code>org.hackystat.app.cgqm.implementation.executables</code>
4	<code>org.hackystat.app.cgqm.implementation.results</code>
5	<code>org.hackystat.app.cgqm.common.classloaders</code>
6	<code>org.hackystat.app.cgqm.utils</code>
6	<code>org.hackystat.app.cgqm.datamodel.cgqm</code>
8	<code>org.hackystat.app.cgqm.datamodels.cgqm.goals</code>
9	<code>org.hackystat.app.cgqm.datamodels.cgqm.metric</code>
10	<code>org.hackystat.app.cgqm.datamodels.cgqm.question</code>
11	<code>org.hackystat.app.cgqm.manager</code>
12	<code>org.hackystat.app.cgqm.telemetry.reducer</code>
13	<code>org.hackystat.app.cgqm.testbase</code>
14	<code>org.hackystat.app.cgqm.utils.freemarker</code>
15	<code>org.hackystat.app.cgqm.webinterface</code>
16	<code>org.hackystat.app.cgqm.webinterface.selector</code>
17	<code>org.hackystat.app.cgqm.datamodels.cgqm.common</code>
18	<code>org.hackystat.app.cgqm.datamodel.cgqm.goal.goalDimension</code>
19	<code>org.hackystat.app.cgqm.datamodel.cgqm.goal.sheetComponents</code>
20	<code>org.hackystat.app.cgqm.common.jiBx</code>
21	<code>org.hackystat.app.cgqm.common.exceptions</code>
22	<code>org.hackystat.app.cgqm.datasources</code>
23	<code>org.hackystat.app.cgqm.telemetry.webHookDataSource</code>
24	<code>org.hackystat.app.cgqm.telemetry.webHookDataSource.describer</code>

Figure D.9. hackyCGQM PRI Ranking - Inspection 11

Workspaces (218):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Commit First Commit	Last Commit	Commit Cont	Code Re Churn	Re view	Last Re view	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyCGQM\src\java\main\org\hackystat\app\cgqm\common\exceptions	930	developer7 (time=0.67, conts=16)	0.67 h	0.17 h	22-Jan-2005	20-Mar-2005	1	16	18-Apr-2005	1	564	0		0	0	LOC=95, Class=16, Method=2	LOC=28, Class=1, Method=2	In=58, out=5	Pass=0, Fail=0, Error=0	70.59 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\interfaces\results	818	developer7 (time=2.50, conts=42)	2.50 h	0.08 h	04-Apr-2005	15-Apr-2005	1	42	25-Apr-2005	1	2890	0		0	0	LOC=382, Class=14, Method=47	LOC=21, Class=2, Method=2	In=33, out=22	Pass=0, Fail=0, Error=0	93.10 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\interfaces\results	714	developer7 (time=3.58, conts=42)	3.58 h	0.33 h	20-Mar-2005	25-Apr-2005	1	42	25-Apr-2005	1	3411	0		0	0	LOC=420, Class=2, Method=54	LOC=59, Class=2, Method=2	In=32, out=12	Pass=0, Fail=5, Error=0	82.76 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\interfaces\results	714	developer7 (time=0.08, conts=10)	0.08 h	0.00 h	15-Mar-2005	15-Mar-2005	1	10	15-Mar-2005	1	797	0		0	0	LOC=167, Class=5, Method=5	LOC=30, Class=2, Method=2	In=1, out=7	Pass=0, Fail=0, Error=0	0.00 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\utils	710	developer7 (time=13.17, conts=56)	13.17 h	3.92 h	19-Mar-2005	21-Apr-2005	1	56	25-Apr-2005	1	5881	0		0	0	LOC=1489, Class=16, Method=149	LOC=301, Class=6, Method=17	In=31, out=34	Pass=0, Fail=56, Error=20	76.92 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	699	developer7 (time=7.83, conts=37)	7.83 h	1.17 h	16-Mar-2005	14-Apr-2005	1	37	25-Apr-2005	1	2682	0		0	0	LOC=651, Class=6, Method=43	LOC=45, Class=2, Method=3	In=15, out=25	Pass=0, Fail=21, Error=31	90.91 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	687	developer7 (time=2.83, conts=10)	2.83 h	0.83 h	15-Mar-2005	18-Apr-2005	1	18	25-Apr-2005	1	603	0		0	0	LOC=113, Class=4, Method=4	LOC=25, Class=2, Method=2	In=0, out=20	Pass=0, Fail=0, Error=27	100.00 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	667	developer7 (time=5.00, conts=62)	5.00 h	1.17 h	09-Mar-2005	14-Apr-2005	1	43	25-Apr-2005	1	2292	0		0	0	LOC=604, Class=5, Method=24	LOC=121, Class=4, Method=4	In=0, out=0	Pass=0, Fail=0, Error=4	100.00 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	652	developer7 (time=7.25, conts=81)	7.25 h	1.17 h	01-Apr-2005	22-Apr-2005	1	81	05-Apr-2005	1	4010	0		0	0	LOC=1048, Class=11, Method=37	LOC=203, Class=5, Method=6	In=0, out=0	Pass=0, Fail=48, Error=3	78.57 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	646	developer7 (time=1.00, conts=21)	1.00 h	0.08 h	01-Apr-2005	02-Apr-2005	1	21	18-Mar-2005	1	947	0		0	0	LOC=172, Class=4, Method=11	LOC=40, Class=2, Method=2	In=6, out=6	Pass=0, Fail=0, Error=2	75.00 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	631	developer7 (time=2.17, conts=9)	2.17 h	0.00 h	04-Feb-2005	04-Feb-2005	1	9	11-Mar-2005	1	305	0		0	0	LOC=6, Class=1, Method=1	LOC=0, Class=0, Method=0	In=0, out=1	Pass=0, Fail=47, Error=17	100.00 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	617	developer7 (time=9.50, conts=24)	9.50 h	0.32 h	09-Apr-2005	18-Apr-2005	1	24	25-Apr-2005	1	1699	0		0	0	LOC=434, Class=5, Method=19	LOC=84, Class=2, Method=3	In=0, out=0	Pass=0, Fail=36, Error=18	75.00 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	614	developer7 (time=0.83, conts=6)	0.83 h	0.00 h	21-Mar-2005	21-Mar-2005	1	6	18-Mar-2005	1	811	0		0	0	LOC=312, Class=4, Method=30	LOC=0, Class=0, Method=0	In=6, out=2	Pass=0, Fail=0, Error=0	53.33 %
hackyCGQM\src\java\main\org\hackystat\app\cgqm\manager	608	developer7 (time=0.42, conts=27)	0.42 h	0.00 h	15-Mar-2005	16-Mar-2005	1	27	11-Mar-2005	1	2290	0		0	0	LOC=258, Class=5, Method=37	LOC=0, Class=0, Method=0	In=11, out=3	Pass=0, Fail=9, Error=0	0.00 %

Figure D.10. hackyCGQM PRI Ranking - Inspection 11

604	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.08, comets=21)	0.08 h	11-Mar-2005	11-Mar-2005	1	21	11-Mar-2005	16-Mar-2005	1	401	0	0	LOC=88, Class=7, Method=17	LOC=0, Class=0, Method=0	in=14, out=6	Pass=0, Fail=0, Error=0	0.00 \$
604	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.25, comets=19)	0.25 h	11-Mar-2005	11-Mar-2005	1	19	11-Mar-2005	16-Mar-2005	1	326	0	0	LOC=72, Class=6, Method=8	LOC=0, Class=0, Method=0	in=12, out=7	Pass=0, Fail=0, Error=0	0.00 \$
566	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\SheetComponents\plugin\overview\top\	developer7 (time=3.92, comets=8)	3.92 h	11-Apr-2005	18-Apr-2005	1	18	12-Apr-2005	25-Apr-2005	1	2483	0	0	LOC=985, Class=9, Method=113	LOC=18, Class=1, Method=2	in=0, out=0	Pass=0, Fail=1, Error=3	53.85 \$
554	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=1.75, comets=10)	1.75 h	14-Apr-2005	18-Apr-2005	1	10	11-Mar-2005	25-Apr-2005	1	460	0	0	LOC=152, Class=9, Method=5	LOC=24, Class=1, Method=1	in=0, out=30	Pass=0, Fail=0, Error=7	0.00 \$
541	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.58, comets=5)	0.58 h	18-Apr-2005	20-Apr-2005	1	5	18-Apr-2005	22-Apr-2005	1	431	0	0	LOC=167, Class=3, Method=16	LOC=0, Class=0, Method=0	in=24, out=11	Pass=0, Fail=0, Error=0	90.91 \$
537	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.25, comets=5)	0.25 h	04-Apr-2005	04-Apr-2005	1	5	05-Apr-2005	25-Apr-2005	1	108	0	0	LOC=9, Class=2, Method=0	LOC=0, Class=0, Method=0	in=2, out=3	Pass=0, Fail=0, Error=0	100.00 \$
513	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.50, comets=4)	0.50 h	10-Apr-2005	13-Apr-2005	1	4	12-Apr-2005	13-Apr-2005	1	371	0	0	LOC=136, Class=3, Method=14	LOC=0, Class=0, Method=0	in=3, out=4	Pass=0, Fail=0, Error=0	87.50 \$
511	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.75, comets=10)	0.75 h	09-Mar-2005	21-Mar-2005	1	14	11-Mar-2005	25-Apr-2005	1	514	0	0	LOC=37, Class=1, Method=5	LOC=0, Class=0, Method=0	in=7, out=7	Pass=0, Fail=0, Error=0	0.00 \$
491	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\goal\GoalDimension\	developer7 (time=0.25, comets=8)	0.25 h	11-Mar-2005	07-Apr-2005	1	8	11-Mar-2005	26-Apr-2005	1	324	0	0	LOC=69, Class=1, Method=7	LOC=0, Class=0, Method=0	in=3, out=14	Pass=0, Fail=0, Error=0	0.00 \$
466	hackyCGQM\src\java\main\org\cgqm\common\	0.00 h	0.00 h	11-Mar-2005	11-Mar-2005	0	10	11-Mar-2005	18-Apr-2005	1	216	0	0	LOC=43, Class=3, Method=5	LOC=15, Class=1, Method=1	in=2, out=3	Pass=0, Fail=0, Error=5	0.00 \$
435	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\metric\	0.00 h	0.00 h	11-Mar-2005	11-Mar-2005	0	5	11-Mar-2005	29-Mar-2005	1	92	0	0	LOC=7, Class=1, Method=1	LOC=0, Class=0, Method=0	in=6, out=1	Pass=0, Fail=0, Error=0	0.00 \$
399	hackyCGQM\src\java\main\org\hackystat\app\cgqm\data\models\cgqm\metric\	0.00 h	0.00 h	20-Mar-2005	25-Apr-2005	1	4	20-Mar-2005	25-Apr-2005	1	96	0	0	LOC=4, Class=1, Method=0	LOC=0, Class=0, Method=0	in=9, out=1	Pass=0, Fail=0, Error=0	0.00 \$
Stats													Average PRI Ranking		614.96			

D.7.4 hackyZorro

The response to this question contains two results. First, a developer ranking, which is provided in Table D.10. Second, a PRI ranking, which is provided in the Figure D.11. Furthermore, the results of the developer rankings were used to help aid the developer in selecting packages for inspection. Therefore, for Inspection 12, the packages `org.hackystat.stdext.zorro.control`, `org.hackystat.stdext.zorror.control.stream`, and `org.hackystat.stdext.zorro.model.action` were selected. It appears that the developer selected the `control.stream` and `model.action` packages to provide examples of the use of control package.

In this particular case, the developer ranking and PRI ranking disagreed that the package was a MINI package, relative to other packages in the same module. However, because only one inspection was conducted in this module, it is not known whether the developer rankings or PRI rankings were incorrect.

Table D.10. hackyZorro Developer Ranking

Ranking	Package
1	<code>org.hackystat.stdext.zorro.analysis</code>
2	<code>org.hackystat.stdext.zorro.control</code>
3	<code>org.hackystat.stdext.zorro.control.tokenizer</code>
4	<code>org.hackystat.stdext.zorro.jess</code>
5	<code>org.hackystat.stdext.zorro.action.file.refactoring</code>
6	<code>org.hackystat.stdext.zorro.model.episode</code>
7	<code>org.hackystat.stdext.zorro.model.action.command</code>
8	<code>org.hackystat.stdext.zorro.model.action.file.edit</code>
9	<code>org.hackystat.stdext.zorro.control.stream</code>
10	<code>org.hackystat.stdext.zorro.model.action.file</code>
11	<code>org.hackystat.stdext.zorro.model.action</code>
12	<code>org.hackystat.stdext.zorro.common</code>
13	<code>org.hackystat.stdext.zorro.control.tokenizer.selector</code>
14	<code>org.hackystat.stdext.zorro</code>

Figure D.11. hackyZorro PRI Ranking - Inspection 12

Workspaces (218):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont.	Commit	First Commit	Last Commit	Commit Cont.	Code Churn	Re View	Last Re View	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyZorro\src\org\hackystat\stedxt\zorro\model\action	966	developer4 (times=17,33, counts=139)	17.33 h	6.83 h	22-Feb-2005	06-Apr-2005	1	139	24-Feb-2005	06-Apr-2005	1	5096	0		0	0	LOC=132, Class=4, Method=22	LOC=41, Class=1, Method=2	in=54, out=3	Pass=0, Fail=81, Error=38	100.00 %
hackyZorro\src\org\hackystat\stedxt\zorro\common	849	developer4 (times=0,33, counts=6)	0.33 h	0.00 h	28-Feb-2005	28-Feb-2005	1	6	01-Mar-2005	06-Mar-2005	1	183	0		0	0	LOC=30, Class=2, Method=2	LOC=16, Class=1, Method=1	in=5, out=1	Pass=0, Fail=1, Error=0	75.00 %
hackyZorro\src\org\hackystat\stedxt\zorro\model\episode	816	developer4 (times=2,50, counts=15)	2.50 h	1.42 h	26-Feb-2005	06-Apr-2005	1	15	01-Mar-2005	06-Apr-2005	1	458	0		0	0	LOC=168, Class=5, Method=21	LOC=88, Class=2, Method=5	in=18, out=9	Pass=0, Fail=4, Error=12	92.86 %
hackyZorro\src\org\hackystat\stedxt\zorro\model\action\command	813	developer4 (times=0,58, counts=24)	0.58 h	0.08 h	04-Apr-2005	07-Apr-2005	1	12	04-Apr-2005	07-Apr-2005	1	572	0		0	0	LOC=206, Class=3, Method=9	LOC=59, Class=2, Method=3	in=8, out=6	Pass=0, Fail=0, Error=0	78.57 %
hackyZorro\src\org\hackystat\stedxt\zorro\control	802	developer4 (times=2,67, counts=37)	21.67 h	2.58 h	28-Feb-2005	14-Apr-2005	1	37	01-Mar-2005	14-Apr-2005	1	2809	0		0	0	LOC=243, Class=8, Method=25	LOC=123, Class=3, Method=6	in=8, out=39	Pass=0, Fail=32, Error=28	81.48 %
hackyZorro\src\org\hackystat\stedxt\zorro\control\strteam	778	developer4 (times=11,17, counts=57)	11.17 h	4.00 h	24-Mar-2005	07-Apr-2005	1	57	23-Mar-2005	07-Apr-2005	1	2344	0		0	0	LOC=195, Class=21, Method=52	LOC=304, Class=10, Method=24	in=27, out=55	Pass=0, Fail=69, Error=73	100.00 %
hackyZorro\src\org\hackystat\stedxt\zorro\control\tokenizer	778	developer4 (times=5,58, counts=36)	5.58 h	2.17 h	02-Mar-2005	08-Apr-2005	1	36	03-Mar-2005	08-Apr-2005	1	1179	0		0	0	LOC=516, Class=9, Method=27	LOC=317, Class=4, Method=8	in=11, out=19	Pass=0, Fail=28, Error=7	95.65 %
hackyZorro\src\org\hackystat\stedxt\zorro\test	778	developer4 (times=3,08, counts=8)	3.08 h	3.08 h	07-Mar-2005	08-Apr-2005	1	8	07-Mar-2005	08-Apr-2005	1	1245	0		0	0	LOC=206, Class=1, Method=12	LOC=206, Class=1, Method=12	in=0, out=15	Pass=0, Fail=90, Error=32	100.00 %
hackyZorro\src\org\hackystat\edit\zorro\model\action\file	730	developer4 (times=0,50, counts=7)	0.42 h	0.08 h	02-Apr-2005	03-Apr-2005	1	7	04-Apr-2005	04-Apr-2005	1	824	0		0	0	LOC=382, Class=1, Method=46	LOC=99, Class=1, Method=6	in=15, out=9	Pass=0, Fail=0, Error=6	86.36 %
hackyZorro\src\org\hackystat\stedxt\zorro\model\action\file	727	developer4 (times=0,50, counts=13)	0.50 h	0.00 h	01-Apr-2005	04-Apr-2005	1	13	04-Apr-2005	04-Apr-2005	1	1044	0		0	0	LOC=477, Class=13, Method=61	LOC=143, Class=5, Method=10	in=33, out=17	Pass=0, Fail=0, Error=13	83.33 %
hackyZorro\src\org\hackystat\stedxt\zorro\analysis	712	developer4 (times=6,50, counts=8)	6.50 h	0.17 h	12-Apr-2005	15-Apr-2005	1	8	12-Apr-2005	15-Apr-2005	1	437	0		0	0	LOC=213, Class=3, Method=7	LOC=12, Class=2, Method=2	in=0, out=30	Pass=0, Fail=0, Error=0	37.50 %
hackyZorro\src\org\hackystat\file\refactoring	680	developer4 (times=0,08, counts=8)	0.08 h	0.00 h	03-Apr-2005	03-Apr-2005	1	8	04-Apr-2005	04-Apr-2005	1	602	0		0	0	LOC=277, Class=8, Method=33	LOC=100, Class=3, Method=5	in=9, out=10	Pass=0, Fail=0, Error=2	89.47 %
hackyZorro\src\org\hackystat\stedxt\zorro	648	developer4 (times=1,08, counts=1)	0.08 h	0.00 h	08-Mar-2005	08-Mar-2005	1	1	08-Mar-2005	08-Mar-2005	1	30	0		0	0	LOC=9, Class=1, Method=2	LOC=0, Class=0, Method=0	in=17, out=0	Pass=0, Fail=0, Error=0	0.00 %
Stats																					
Average PRI Ranking																					
775.15																					

D.7.5 hackyTelemetry

The response to this question contains two results. First, a developer ranking, which is provided in Table D.11. Second, a PRI ranking, which is provided in the Figure D.12. Furthermore, the results of the developer rankings were used to help aid the developer in selecting a package for inspection. Therefore, for Inspection 14, the package `org.hackystat.app.telemetry.config` was selected.

In this particular case, the developer ranking and PRI ranking disagreed that the package was a MINI package, relative to other packages in the same module. However, because only one inspection was conducted in this module, it is not known whether the developer rankings or PRI rankings were incorrect.

Table D.11. hackyTelemetry Developer Ranking

Ranking	Package
1	<code>org.hackystat.app.telemetry.config</code>
2	<code>org.hackystat.app.telemetry.analysis</code>
3	<code>org.hackystat.app.telemetry.config.core</code>
4	<code>org.hackystat.app.telemetry.processor.reducer.impl</code>
5	<code>org.hackystat.app.telemetry.processor.parser.impl</code>
6	<code>org.hackystat.app.telemetry.processor.evaluator</code>
7	<code>org.hackystat.app.telemetry.processor.reducer.impl</code>
8	<code>org.hackystat.app.telemetry.processor.query</code>
9	<code>org.hackystat.app.telemetry.processor.parser</code>
10	<code>org.hackystat.app.telemetry.processor.query.expression</code>
11	<code>org.hackystat.app.telemetry.processor.reducer</code>
12	<code>org.hackystat.app.telemetry.processor.stream</code>

Figure D.12. hackyTelemetry PRI Ranking - Inspection 14

Workspaces (217):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Com mit Cont	Last Commit	Code Churn	Re view	Last Re view	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyTelemetry\src\org\hackystat\app\telemetry\processor\reducer\ impl	1051	developer3 (time=23.56, count=138)	27.08 h	9.33 h	31-May-2004	29-Mar-2005	4	162	01-Jun-2004	24-Mar-2005	10968	0	4	0	LOC=2080, Class=20, Method=75	LOC=1213, Class=10, Method=49	in=13, out=43	Pass=0, Fail=207, Error=105	100.00 %
hackyTelemetry\src\org\hackystat\app\telemetry\analysis	1046	developer3 (time=6.17, count=130)	29.56 h	4.58 h	26-Apr-2004	23-Mar-2005	2	132	26-Apr-2004	23-Mar-2005	7221	0	0	0	LOC=854, Class=7, Method=22	LOC=199, Method=7	in=5, out=67	Pass=0, Fail=26, Error=26	86.67 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\query	1029	developer3 (time=6.17, count=61)	6.25 h	1.50 h	21-May-2004	31-Oct-2004	2	62	21-May-2004	19-Mar-2005	2799	0	0	0	LOC=389, Class=8, Method=32	LOC=157, Class=3, Method=7	in=23, out=12	Pass=0, Fail=1, Error=1	88.24 %
hackyTelemetry\src\org\hackystat\app\telemetry\config	1021	developer3 (time=26.00, count=108)	26.08 h	3.50 h	10-Jun-2004	19-Mar-2005	2	110	13-Jun-2004	22-Mar-2005	7040	0	0	0	LOC=493, Class=7, Method=33	LOC=9, Class=1, Method=1	in=17, out=16	Pass=0, Fail=24, Error=11	43.75 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\evaluator	976	developer3 (time=9.00, count=50)	9.67 h	3.50 h	06-Jun-2004	24-Mar-2005	4	36	07-Jun-2004	24-Mar-2005	2344	0	0	0	LOC=712, Class=6, Method=23	LOC=303, Class=3, Method=12	in=33, out=33	Pass=0, Fail=30, Error=15	89.66 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\stream	973	developer3 (time=18.00, count=18)	4.17 h	1.00 h	30-May-2004	14-Mar-2005	2	20	01-Jun-2004	14-Mar-2005	910	0	0	0	LOC=259, Class=5, Method=25	LOC=106, Method=5	in=83, out=12	Pass=0, Error=0	100.00 %
hackyTelemetry\src\org\hackystat\app\telemetry\config\core	964	developer3 (time=12.00, count=42)	12.00 h	3.75 h	29-Oct-2004	22-Mar-2005	1	42	01-Nov-2004	22-Mar-2005	3636	0	0	0	LOC=1464, Class=17, Method=90	LOC=563, Class=6, Method=17	in=77, out=24	Pass=0, Fail=12, Error=12	89.04 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\query\oppression	949	developer3 (time=0.92, count=53)	1.08 h	0.67 h	21-May-2004	31-Mar-2005	3	54	21-May-2004	19-Mar-2005	1566	0	0	0	LOC=391, Class=19, Method=57	LOC=107, Class=4, Method=7	in=43, out=15	Pass=0, Fail=2, Error=1	94.12 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\reducer\util	921	developer3 (time=3.17, count=22)	3.17 h	1.33 h	31-May-2004	24-Mar-2005	1	24	01-Jun-2004	24-Mar-2005	1752	0	0	0	LOC=892, Class=10, Method=43	LOC=245, Class=4, Method=11	in=17, out=25	Pass=0, Fail=26, Error=16	75.00 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\reducer	890	developer3 (time=1.15, count=15)	1.75 h	0.42 h	05-Jun-2004	01-Jul-2004	1	17	01-Jun-2004	18-Mar-2005	580	0	4	0	LOC=183, Class=1, Method=19	LOC=17, Method=1	in=44, out=8	Pass=0, Fail=0, Error=0	87.50 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\parser	889	developer3 (time=9.42, count=43)	9.42 h	6.67 h	21-May-2004	21-Apr-2005	1	44	21-May-2004	21-Apr-2005	1570	0	0	0	LOC=433, Class=5, Method=27	LOC=316, Class=5, Method=17	in=13, out=23	Pass=0, Fail=118, Error=299	89.66 %
hackyTelemetry\src\org\hackystat\app\telemetry\processor\parser\ impl	804	developer3 (time=0.08, count=39)	0.08 h	0.00 h	30-May-2004	30-May-2004	1	40	07-Jun-2004	21-Apr-2005	7065	0	0	0	LOC=1938, Class=7, Method=106	LOC=0, Method=0	in=7, out=25	Pass=0, Fail=0, Error=0	65.52 %
Stats																			
Average PRI Ranking																			
959.42																			

D.8 Question 8

Question 8. To the best of your knowledge, please provide the top 5 modules that you think need to be inspected and the top 5 modules that you think do NOT need to be inspected.

The Tables D.12, D.13, D.14, D.15, D.16, and D.17 present the results of this question. Each table is one participants response. When available, I provide the participants justification for giving a module a specific ranking. “??” means that the participant could not provide an module. “N/A” means that the participant does not believe there is an appropriate module.

Figures D.13, D.14, D.15, and D.16 provide a graphical view of the responses.

Table D.12. Question 8 Responses - Participant 1

Ranking	Module	Explanation
Modules that need to be inspected		
1	hackyCGQM	New code
2	hackyZorro	New code
3	hackyIssue	New code
4	hackyDependency	New code
5	??	
Modules that do not need to be inspected		
1	hackyKernel	Old code
2	hackyStdExt	Old code
3	hackyStatistics	Old code
4	hackyReport	Old code
5	??	

Table D.13. Question 8 Responses - Participant 2

Ranking	Module	Explanation
Modules that need to be inspected		
1	hackyCGQM	Causes many build failures
2	hackyIssue	Knows there are defects in this code
3	hackyZorro	New code
4	??	
5	??	
Modules that do not need to be inspected		
1	hackyJupiter	Fairly old code. No new development. Worked last year.
2	hackyKernel	Core module. Used a lot. If there are problems, then it would show up somewhere fast.
3	hackyStdExt	Same as previous explanation
4	??	
5	??	

Table D.14. Question 8 Responses - Participant 3

Ranking	Module	Explanation
Modules that need to be inspected		
1	hackyCGQM	Frequently fails build. New code.
2	hackyZorro	Code has not been reviewed.
3	hackyHPCS	Code has not been reviewed.
4	hackyKernel	Important code
4	hackyOffice	None of the office sensors work properly
Modules that do not need to be inspected		
1	N/A	No code should be excluded from inspection
2	N/A	
3	N/A	
4	N/A	
5	N/A	

Table D.15. Question 8 Responses - Participant 4

Ranking	Module	Explanation
Modules that need to be inspected		
1	hackyEclipse	I don't trust the code.
2	hackyCGQM	New code
3	hackyZorro	New code
4	hackyIssue	New code
5	hackyPRI	New code
Modules that do not need to be inspected		
1	hackyStatistics	Works fine.
2	hackyVIM	No one uses it. Who cares?
3	hackyTDD	No one uses it (after the new hackyZorro replaced it). Who cares?
4	hackyCLI	No one uses it. Who Cares?
5	hackyJBuilder	No one uses it. Who Cares?

Table D.16. Question 8 Responses - Participant 5

Ranking	Module	Explanation
Modules that need to be inspected		
1	hackyStdExt	Large module and has many dependencies.
2	hackyCGQM	New code and new developer.
3	hackyIssue	I want to learn about the code.
4	hackyTelemetry	Important code. Used a lot.
5	hackyCLI	Could be useful, but we haven't paid any attention to it.
Modules that do not need to be inspected		
1	hackyKernel	Important code, but we can detect errors quickly.
2	hackyStatistics	Small module and its not used a lot.
3	hackyReport	Been stable for a while. No new development.
4	hackyEclipse	Been refactored. Has high use, so defects will be found quickly.
5	hackyPrjSize	No one uses it. Who Cares?

Table D.17. Question 8 Responses - Participant 6

Ranking	Module	Explanation
Modules that need to be inspected		
1	hackyCGQM	Always fails the build.
2	hackyZorro	Always fails the build.
3	hackyStdExt	Always fails the build.
4	??	
5	??	
Modules that do not need to be inspected		
1	hackyStatistics	Never fails the build.
2	hackyReport	Never fails the build.
3	??	
4	??	
5	??	

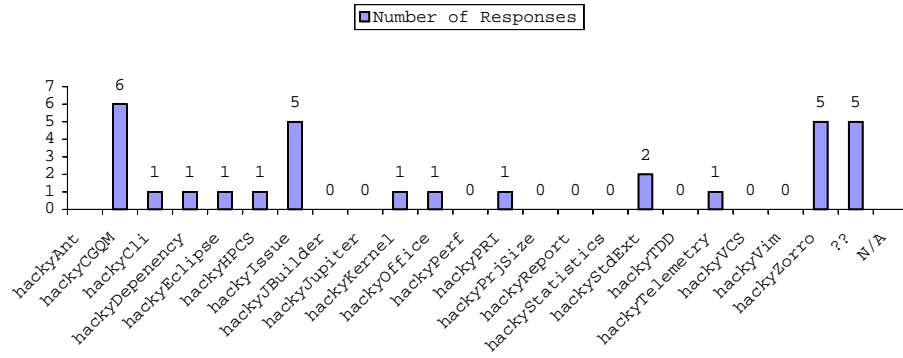


Figure D.13. Question 8 Part 1 Responses - Provides the total number of responses that the participants felt were MINI modules. ?? indicates that the participant did not know which module were MINI.

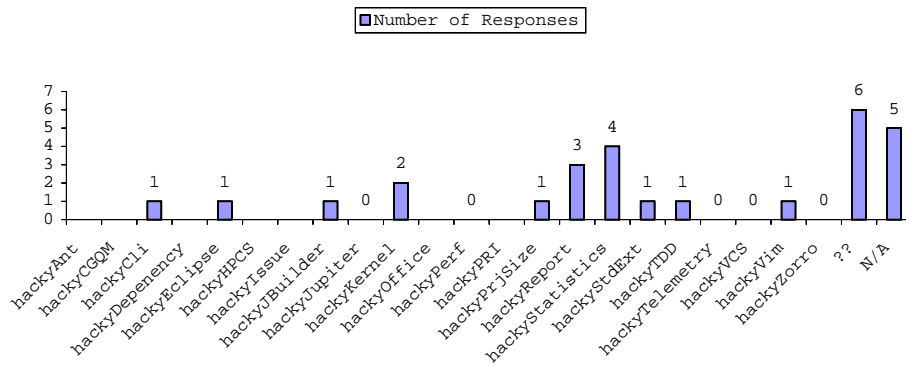


Figure D.14. Question 8 Part 2 Responses - Provides the total number of responses that the participants felt were LINI modules. ?? indicates that the participants did not know which modules were LINI. N/A indicates that the participants felt no module should be declared LINI.

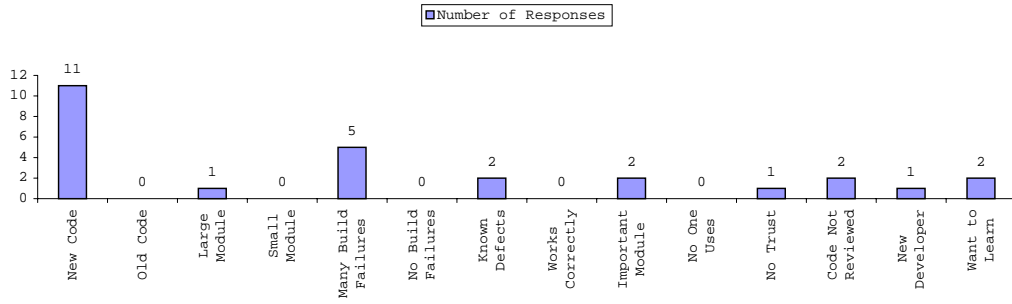


Figure D.15. Question 8 Part 1 Responses - Provides the total number of similar explanations used when ranking the top 5 MINI modules.

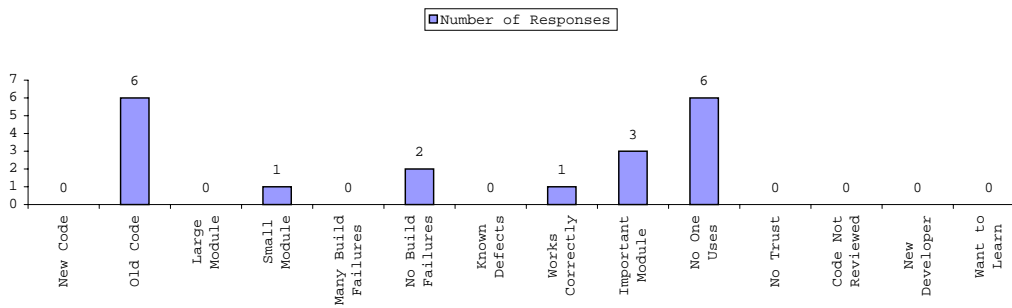


Figure D.16. Question 8 Part 2 Responses - Provides the total number of similar explanations used when ranking the top 5 LINI modules.

D.9 Question 9

Question 9. To the best of your knowledge, please provide the top 5 workspaces that you think need to be inspected and the top 5 workspaces that you think do NOT need to be inspected.

The Tables D.18, D.19, D.20, D.21, D.22, and D.23 present the results of this question. Each table is one participants response. When available, I provide the participants justification for giving a workspace a specific ranking. “??” means that the participant could not provide an workspace. “N/A” means that the participant does not believe there is an appropriate workspace.

Table D.18. Question 9 Responses - Participant 1

Ranking	Module	Explanation
Workspaces that need to be inspected		
1	hackyCGQM/src/java.main/cGQM/ plu- gin/common	High Coverage
2	hackyCGQM/src/org/hackystat/ app/cgqm/interfaces/executables/goals	High Coverage
3	hackyCGQM/src/org/hackystat/ app/cgqm/interfaces/executables/questions	High Coverage
4	hackyCli/src/org/hackystat/ app/cli/dailyanalysis	High Coverage
5	??	
Workspaces that do not need to be inspected		
1	hackyVCS/src/org/hackystat/ app/commit/analysis/projectchurn	Low Coverage
2	hackyVCS/src/org/hackystat/ app/commit/dailyanalysis	Low Coverage
3	hackyStdExt/src/org/hackystat/ stdext/bufftran/dailyanalysis	Low Coverage
4	??	
5	??	

Table D.19. Question 9 Responses - Participant 2

Ranking	Module	Explanation
Workspaces that need to be inspected		
1	hackyCGQM/src/java.main/org/hackystat/app/cgqm/manager	Package name seems important
2	hackyCGQM/src/java.main/org/hackystat/app/cgqm/telemetry/webHookDataSource	Package name seems important
3	hackyCGQM/src/java.main/org/hackystat/app/cgqm/telemetry/webHookDataSource/describer	Package name seems important
4	hackyIssue/src/org/hackystat/stdext/issue/reducer	New code. Known issues.
5	hackyAnt/src/org/hackystat/stdext/sensor/ant/jira	New code. Important code.
Workspaces that do not need to be inspected		
1	hackyKernel/src/org/hackystat/kernel/shell	Important code.
2	hackyKernel/src/org/hackystat/kernel/shell/command/	Works correctly.
3	hackyKernel/src/org/hackystat/kernel/soap	Works correctly.
4	hackyKernel/src/org/hackystat/kernel/util	Works correctly.
5	hackyKernel/src/org/hackystat/kernel/timer	Works correctly.

Table D.20. Question 9 Responses - Participant 3

Ranking	Module	Explanation
Workspaces that need to be inspected		
1	hackyCQGM (any workspace)	No idea which workspace, but we should inspect something in this module.
2	hackyZorro (any workspace)	No idea which workspace, but we should inspect something in this module.
3	hackyHPCS/src/org/hackystat/app/hpcs/dailyproject	Developed quickly.
4	hackyKernel/src/org/hackystat/kenrel/sdt	Before new improvements are implemented
5	hackyOffice (activity package)	Known issues.
Workspaces that do not need to be inspected		
1	N/A	No code should be excluded from inspection
2	N/A	
3	N/A	
4	N/A	
5	N/A	

Table D.21. Question 9 Responses - Participant 4

Ranking	Module	Explanation
Workspaces that need to be inspected		
1	??	I have no idea.
2	??	
3	??	
4	??	
5	??	
Workspaces that do not need to be inspected		
1	??	I have no idea.
2	??	
3	??	
4	??	
5	??	

Table D.22. Question 9 Responses - Participant 5

Ranking	Module	Explanation
Workspaces that need to be inspected		
1	hackyAnt/src/org/hackystat/stdext/sensor/ant/junit	Known issues.
2	hackyCGQM (any workspace)	Code standards.
3	hackyIssue/src/org/hackystat/sd-text/issue/reducer	Known issues.
4	hackyReview/src/org/hackystat/app/review/analysis/cache	Improvements from last inspection.
5	hackyZorro/src/org/hackystat/sdtext/zorro/jess	Known issues.
Workspaces that do not need to be inspected		
1	hackyKernel/src/org/hackystat/kernel/util	Used widely.
2	hackyKernel/src/org/hackystat/kernel/user	Used widely.
3	hackyStatistics (any workspace)	Not being used. Simple code.
4	hackyReport (any workspace)	Used widely.
5	hackyEclipse (any workspace)	Used widely.

Table D.23. Question 9 Responses - Participant 6

Ranking	Module	Explanation
Workspaces that need to be inspected		
1	hackyCGQM (any workspace)	Many build failures. No idea which workspace, but we should inspect something in this module.
2	hackyZorro (any workspace)	Many build failures. No idea which workspace, but we should inspect something in this module.
3	??	
4	??	
5	??	
Workspaces that do not need to be inspected		
1	hackyStatistics (any workspace)	
2	hackyStatistics (any workspace)	
3	hackyStatistics (any workspace)	
4	hackyStatistics (any workspace)	
5	hackyStatistics (any workspace)	

Appendix E

Inspection and Post-Inspection-Questionnaire Results

E.1 Inspection 8

Table E.1. Inspection 8 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Package	org.hackystat.stdext.review.analysis.cache
Module	hackyReview
Developer Ranking	MINI (See Table D.7)
PRI Ranking	MINI (See Figure D.7)
Product and Process Measures	See Figure D.7
Inspection Date	April 6, 2005
Jupiter Review ID	ReviewAnalysisCache
Number of Inspectors	5
Meeting Attendance	7

Table E.2. Inspection 8 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1			1		1	2
2		2	2			4
3			2	1	2	5
4	1	3	4			8
5		2	2	1		5
Total	1	7	11	2	3	24

Table E.3. Inspection 8 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical		1				
Major	2	3	1		1	
Normal	5	2		2	2	
Minor	1				1	
Trivial	2				1	
Total	10	6	1	2	5	0

Table E.4. Inspection 8 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	5	2
Did you learned something from this inspection?	5	2
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	7	0

E.2 Inspection 9

Table E.5. Inspection 9 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Package	org.hackystat.stdext.issue.reducer
Module	hackyIssue
Developer Ranking	MINI (See Table D.8)
PRI Ranking	MINI (See Figure D.8)
Product and Process Measures	See Figure D.8
Inspection Date	April 13, 2005
Jupiter Review ID	IssueReducer
Number of Inspectors	6
Meeting Attendance	7

Table E.6. Inspection 9 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1		7				7
2			1	3		4
3					2	2
4		2	7			9
5		5				5
6		2	5	1	2	10
Total	0	16	13	4	4	37

Table E.7. Inspection 9 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical						
Major	2	7	3		1	2
Normal	1	5		4	2	1
Minor	2	1				1
Trivial	4					
Total	9	13	3	4	3	4

Table E.8. Inspection 9 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	7	0
Did you learned something from this inspection?	7	0
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	7	0

E.3 Inspection 11

Table E.9. Inspection 11 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.app.cgqm.interfaces.executables, org.hackystat.app.cgqm.interfaces.results, org.hackystat.app.cgqm.implementations.executables
Module	hackyCGQM
Developer Ranking	MINI (See Table D.9)
PRI Ranking	LINI (See Figure D.9)
Product and Process Measures	See Figure D.9
Inspection Date	April 27, 2005
Jupiter Review ID	cGQMInterfaces
Number of Inspectors	6
Meeting Attendance	7

Table E.10. Inspection 11 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1	1	1	2	1		5
2		5	2			7
3				10		10
4			6			6
5		5	3		1	9
6		2		3		5
Total	1	13	14	14	1	44

Table E.11. Inspection 11 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical				1		
Major	2	6	2	1		3
Normal	8	1	1			3
Minor	11					1
Trivial	1					
Total	22	7	3	2	0	7

Table E.12. Inspection 11 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	6	1
Did you learned something from this inspection?	7	0
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	7	0

E.4 Inspection 12

Table E.13. Inspection 12 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.stdext.zorro.control org.hackystat.stdext.zorro.control.stream, org.hackystat.stdext.zorro..model.action
Module	hackyZorro
Developer Ranking	MINI (See Table D.10)
PRI Ranking	LINI (See Figure D.11)
Product and Process Measures	See Figure D.11
Inspection Date	May 04, 2005
Jupiter Review ID	DevelopmentStream
Number of Inspectors	5
Meeting Attendance	6

Table E.14. Inspection 12 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1		3	4	1		8
2		2	3	1		6
3		3	3	1		7
4		2	2	3		7
5		3				3
Total	0	13	12	6	0	31

Table E.15. Inspection 12 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical						
Major	1	4	1	2	3	2
Normal	7			2	3	
Minor	6					
Trivial						
Total	14	4	1	4	6	2

Table E.16. Inspection 12 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	5	1
Did you learned something from this inspection?	3	3
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	5	1

E.5 Inspection 13

Table E.17. Inspection 13 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.stdext.issue.reducer
Module	hackyIssue
Developer Ranking	Second review of this package
PRI Ranking	LINI (See Figure E.1)
Product and Process Measures	See Figure E.1
Inspection Date	May 4, 2005
Jupiter Review ID	IssueReducer2
Number of Inspectors	4
Meeting Attendance	6

Table E.18. Inspection 13 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1		1				1
2		5	1			6
3	1					0
4		1		2		3
Total	1	7	1	2	0	11

Table E.19. Inspection 13 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical		1				
Major	3	4				
Normal		1				
Minor	2					
Trivial						
Total	5	6	0	0	0	0

Table E.20. Inspection 13 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	3	3
Did you learned something from this inspection?	3	3
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	4	2

Figure E.1. hackyIssue PRI Ranking - Inspection 13

Workspaces (218):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Commit	First Commit	Last Commit	Commit Cont	Code Churn	Re view	Last Re view	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyissue\src\org\hackystat\stdext\issue\edt	988	developer6 (time=3.42, coms=16)	5.00 h	0.92 h	10-Feb-2005	27-Apr-2005	2	33	08-Feb-2005	11-Apr-2005	2	1338	58	29-Apr-2005	0	1	LOC=392, Class=8, Method=92	LOC=91, Class=3, Method=4	in=6, out=13	Pass=0, Fail=13, Error=12	81.48 %
hackyissue\src\org\hackystat\stdext\issue\reducer	901	developer6 (time=2.75, coms=6)	28.00 h	3.42 h	26-Feb-2005	29-Apr-2005	2	6	09-Mar-2005	11-Apr-2005	1	1177	38	27-Apr-2005	2	0	LOC=227, Class=9, Method=7	LOC=64, Class=1, Method=3	in=1, out=24	Pass=0, Fail=9, Error=20	100.00 %
hackyissue\src\org\hackystat\stdext\issue\analysis\issueprojectdetails	793	developer2 (time=3.17, coms=11)	3.17 h	0.17 h	19-Feb-2005	04-Apr-2005	1	11	20-Feb-2005	05-Apr-2005	1	619	0		0	1	LOC=228, Class=2, Method=5	LOC=32, Class=1, Method=1	in=1, out=21	Pass=0, Fail=0, Error=9	100.00 %
hackyissue\src\org\hackystat\stdext\issue\dailyproject	778		16.83 h	0.58 h	13-Feb-2005	29-Apr-2005	2	18	14-Feb-2005	05-Apr-2005	2	1377	0		0	0	LOC=536, Class=2, Method=18	LOC=66, Class=1, Method=4	in=5, out=23	Pass=0, Fail=31, Error=7	100.00 %
Stats																					
Average PRI Ranking																	865.00				

E.6 Inspection 14

Table E.21. Inspection 14 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.app.telemetry.config
Module	hackyTelemetry
Developer Ranking	MINI (See Table D.11)
PRI Ranking	LINI (See Figure D.12)
Product and Process Measures	See Figure D.12
Inspection Date	May 11, 2005
Jupiter Review ID	TelemetryWebConfig
Number of Inspectors	6
Meeting Attendance	6

Table E.22. Inspection 14 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1		1	4			5
2	1	2				3
3		1	2			3
4		3	4			7
5		4	1			5
6		2	1			3
Total	1	13	12	0	0	26

Table E.23. Inspection 14 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical		1				
Major	3	1	1	5		2
Normal	6	2		3	1	
Minor						
Trivial						
Total	9	4	1	8	1	2

Table E.24. Inspection 14 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	6	0
Did you learned something from this inspection?	6	0
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	6	0

E.7 Inspection 15

Table E.25. Inspection 15 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.kernel.cache
Module	hackyKernel
Developer Ranking	N/A - hand selected via PRI ranking
PRI Ranking	LINI (See Figure E.2)
Product and Process Measures	See Figure E.2
Inspection Date	June 1, 2005
Jupiter Review ID	KernelCache
Number of Inspectors	6
Meeting Attendance	5

Table E.26. Inspection 15 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1						0
2		1				1
3			2			2
4			3	2		5
5						0
6			1		1	2
Total	0	1	6	2	1	10

Table E.27. Inspection 15 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical						
Major		1				
Normal	3	1			1	
Minor	2					
Trivial	1					
Total	6	2	0	0	1	0

Table E.28. Inspection 15 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	2	3
Did you learned something from this inspection?	4	1
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	4	1

Figure E.2. hackyKernel PRI Ranking - Inspection 15

Workspaces (221):	Ranking Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Com mit Cont	Last Commit	Com mit Cont	Code Churn	Re View	Last Re View	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackykernel\src\org\hackystat\Kernel\admin	developer1 (time=25.50, conts=98)	42.25 h	5.25 h	04-May-2003	12-May-2005	6	152	04-May-2003	12-May-2005	5	6533	1	27-Apr-2005	4	LOC=988, Class=11, Method=10	LOC=115, Class=3, Method=10	ins=175, out=26	Pass=0, Fail=26, Error=17	80.28 %
hackykernel\src\org\hackystat\Kernel\util	developer4 (time=8.58, conts=54)	17.33 h	7.50 h	04-May-2003	18-Mar-2005	5	112	04-May-2003	24-Mar-2005	5	4537	0	0	0	LOC=1317, Class=26, Method=22	LOC=381, Class=10, Method=22	ins=456, out=16	Pass=0, Fail=9, Error=163	86.00 %
hackykernel\src\org\hackystat\Kernel\cache	developer1 (time=13, conts=12)	2.75 h	0.50 h	04-May-2003	10-Dec-2004	3	15	04-May-2003	10-Dec-2004	2	829	0	0	0	LOC=347, Class=8, Method=4	LOC=85, Class=3, Method=4	ins=15, out=2	Pass=0, Fail=0, Error=2	91.30 %
hackykernel\src\org\hackystat\Kernel\command	developer1 (time=2.08, conts=12)	2.42 h	0.33 h	04-May-2003	24-Sep-2004	2	17	04-May-2003	24-Sep-2004	2	1094	0	0	0	LOC=483, Class=4, Method=1	LOC=10, Class=1, Method=1	ins=82, out=11	Pass=0, Fail=0, Error=2	88.24 %
hackykernel\src\org\hackystat\Kernel\sensor\huermap	developer1 (time=4.67, conts=41)	21.17 h	1.25 h	08-Oct-2004	18-Nov-2004	5	48	10-Oct-2004	16-Feb-2005	5	1133	34	20-Oct-2004	0	LOC=293, Class=8, Method=2	LOC=54, Class=2, Method=2	ins=7, out=10	Pass=0, Fail=2, Error=11	40.91 %
hackykernel\src\org\hackystat\Kernel\hw	developer1 (time=8.58, conts=54)	6.08 h	0.25 h	04-May-2003	29-Mar-2005	4	57	04-May-2003	01-Apr-2005	4	2209	0	0	0	LOC=569, Class=9, Method=27	LOC=9, Class=1, Method=1	ins=195, out=15	Pass=0, Fail=39, Error=25	79.17 %
hackykernel\src\org\hackystat\Kernel\user	developer1 (time=2.33, conts=11)	3.00 h	0.08 h	20-Jan-2004	19-Apr-2005	3	29	20-Jan-2004	07-Mar-2005	3	1609	0	0	1	LOC=564, Class=5, Method=5	LOC=70, Class=2, Method=2	ins=494, out=5	Pass=0, Fail=5, Error=2	93.94 %
hackykernel\src\org\hackystat\Kernel\test	developer1 (time=0.67, conts=11)	1.58 h	0.17 h	04-May-2003	15-Nov-2004	5	17	04-May-2003	08-Mar-2004	2	495	0	0	0	LOC=202, Class=3, Method=8	LOC=33, Class=1, Method=1	ins=6, out=11	Pass=0, Fail=0, Error=0	63.64 %
hackykernel\src\org\hackystat\Kernel\shell	developer1 (time=3.92, conts=9)	8.58 h	0.00 h	21-May-2003	02-Nov-2004	5	21	04-May-2003	15-Nov-2004	5	1880	1	19-Oct-2004	2	LOC=684, Class=3, Method=43	LOC=0, Class=0, Method=0	ins=60, out=12	Pass=0, Fail=0, Error=0	62.86 %
hackykernel\src\org\hackystat\Kernel\time	developer1 (time=0.17, conts=7)	0.17 h	0.00 h	21-May-2003	06-Jan-2005	1	9	04-May-2003	06-Jan-2005	2	163	0	0	0	LOC=62, Class=2, Method=7	LOC=0, Class=0, Method=0	ins=4, out=2	Pass=0, Fail=0, Error=0	83.33 %
hackykernel\src\org\hackystat\Kernel\sensor\data	developer1 (time=4.25, conts=52)	5.42 h	0.00 h	07-Jun-2003	19-Apr-2005	7	45	04-May-2003	05-Apr-2005	3	1845	0	0	0	LOC=610, Class=9, Method=70	LOC=0, Class=0, Method=0	ins=133, out=21	Pass=0, Fail=0, Error=0	72.22 %
hackykernel\src\org\hackystat\Kernel\hw\log	developer1 (time=0.85, conts=27)	6.92 h	0.00 h	26-Jan-2004	25-Oct-2004	4	36	26-Jan-2004	25-Oct-2004	3	1118	0	0	0	LOC=230, Class=9, Method=24	LOC=0, Class=0, Method=0	ins=10, out=9	Pass=0, Fail=0, Error=0	38.46 %
hackykernel\src\org\hackystat\Kernel\sensor	developer1 (time=0.17, conts=9)	0.17 h	0.00 h	02-Oct-2003	10-Feb-2005	1	12	04-May-2003	10-Feb-2005	2	555	0	0	0	LOC=190, Class=3, Method=22	LOC=0, Class=0, Method=0	ins=3, out=6	Pass=0, Fail=0, Error=0	77.78 %
hackykernel\src\org\hackystat\Kernel\sd	developer1 (time=1.67, conts=12)	2.33 h	0.00 h	06-May-2003	05-Apr-2005	3	20	04-May-2003	05-Apr-2005	3	1144	0	0	0	LOC=412, Class=4, Method=49	LOC=0, Class=0, Method=0	ins=111, out=5	Pass=0, Fail=0, Error=0	73.68 %
hackykernel\src\org\hackystat\Kernel\shell\command	developer1 (time=1.67, conts=9)	1.17 h	0.00 h	05-May-2003	14-Nov-2004	4	17	04-May-2003	15-Nov-2004	5	633	0	0	0	LOC=287, Class=6, Method=20	LOC=0, Class=0, Method=0	ins=21, out=7	Pass=0, Fail=0, Error=0	52.94 %
hackykernel\src\org\hackystat\Kernel\user	developer1 (time=1.67, conts=9)	0.83 h	0.00 h	30-May-2003	21-Apr-2004	3	12	04-May-2003	21-Apr-2004	2	217	0	0	0	LOC=66, Class=3, Method=3	LOC=0, Class=0, Method=0	ins=8, out=9	Pass=0, Fail=0, Error=0	25.00 %
hackykernel\src\org\hackystat\Kernel\test	developer1 (time=17.75, conts=24)	19.58 h	0.67 h	19-Mar-2004	12-May-2005	4	26	06-Apr-2004	20-May-2005	3	2679	0	0	0	LOC=566, Class=4, Method=38	LOC=0, Class=0, Method=0	ins=88, out=4	Pass=0, Fail=14, Error=4	93.33 %

Stats																			
Average PRI Ranking	1.004_41																		

E.8 Inspection 16

Table E.29. Inspection 16 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.stdext.project
Module	hackyStdExt
Developer Ranking	N/A - hand selected via PRI ranking
PRI Ranking	LINI (See Figure E.3)
Product and Process Measures	See Figure E.3
Inspection Date	June 1, 2005
Jupiter Review ID	Project
Number of Inspectors	5
Meeting Attendance	5

Table E.30. Inspection 16 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1		1	2	1		4
2						0
3					1	1
4		1			1	2
5		1	2			3
Total	0	3	4	3	0	10

Table E.31. Inspection 16 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical						
Major			1		1	1
Normal				1	2	
Minor			1			2
Trivial						
Total	0	0	2	1	3	3

Table E.32. Inspection 16 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	3	2
Did you learned something from this inspection?	4	1
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	4	1

Figure E.3. hackyStdExt PRI Ranking - Inspection 16

Workspaces (221):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Com mit	Com mit	Last Com mit	Code Churn	Re view	Last Re view	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyStdExt\src\org\hackystat\stdext\project	1239	developer1 (time=17.58, com=52)	25.33 h	3.42 h	12-May-2003	16-Mar-2005	7	108	06-May-2003	17-Mar-2005	6	5202	0	0	1	LOC=1005, Class=7, Method=5	LOC=96, Class=2, Method=5	in=32, out=28	Pass=0, Fail=23, Error=21	78.00 %
hackyStdExt\src\org\hackystat\stdext\project\cache	1175		28.00 h	4.83 h	25-Feb-2004	16-Mar-2005	6	185	25-Feb-2004	17-Mar-2005	6	9421	0	0	0	LOC=294, Class=6, Method=32	LOC=32, Class=1, Method=2	in=41, out=17	Pass=0, Fail=78, Error=123	66.67 %
hackyStdExt\src\org\hackystat\stdext\activity\analysis\projecttime	1112	developer4 (time=2.00, com=8)	5.58 h	2.83 h	09-May-2003	13-Jul-2004	5	25	10-May-2004	09-Nov-2004	6	1470	0	0	0	LOC=167, Class=2, Method=5	LOC=22, Class=1, Method=1	in=0, out=30	Pass=0, Fail=46, Error=20	100.00 %
hackyStdExt\src\org\hackystat\stdext\activity\analysis\time	1072	developer4 (time=0.86, com=13)	9.17 h	3.08 h	01-Jan-2003	06-Jun-2004	3	27	10-May-2003	25-Jun-2004	3	1661	0	0	0	LOC=133, Class=3, Method=3	LOC=25, Class=1, Method=1	in=0, out=26	Pass=0, Fail=16, Error=47	100.00 %
hackyStdExt\src\org\hackystat\stdext\admin\analysis\serverdata	1065	developer1 (time=0.58, com=9)	0.83 h	0.42 h	08-May-2003	21-Apr-2004	4	10	08-May-2003	25-Jun-2004	2	537	0	0	0	LOC=240, Class=2, Method=19	LOC=13, Class=1, Method=1	in=1, out=19	Pass=0, Fail=8, Error=5	94.44 %
hackyStdExt\src\org\hackystat\stdext\dailyanalysis\dayarray	1064	developer4 (time=0.25, com=11)	0.50 h	0.25 h	21-Aug-2003	18-Oct-2004	3	24	05-May-2003	19-Oct-2004	4	1111	0	0	0	LOC=431, Class=7, Method=49	LOC=149, Class=2, Method=9	in=45, out=4	Pass=0, Fail=0, Error=2	87.88 %
hackyStdExt\src\org\hackystat\stdext\admin\analysis\adoption	1054	developer1 (time=1.83, com=9)	2.17 h	0.58 h	08-May-2003	26-Oct-2004	2	10	08-May-2003	26-Oct-2004	2	323	0	0	0	LOC=120, Class=2, Method=4	LOC=10, Class=1, Method=1	in=1, out=12	Pass=0, Fail=19, Error=6	100.00 %
hackyStdExt\src\org\hackystat\stdext\common\alert\metadata	1052	developer1 (time=0.25, com=12)	0.50 h	0.33 h	05-May-2003	06-Apr-2004	3	20	05-May-2003	30-Jul-2004	3	431	0	0	0	LOC=112, Class=2, Method=11	LOC=16, Class=1, Method=1	in=2, out=15	Pass=0, Fail=23, Error=23	80.00 %
hackyStdExt\src\org\hackystat\stdext\activity\analysis\project\filetime	1048	developer1 (time=0.33, com=2)	0.33 h	0.17 h	24-Jun-2004	24-Jun-2004	1	4	25-Jun-2004	09-Nov-2004	3	232	0	0	0	LOC=132, Class=2, Method=5	LOC=20, Class=1, Method=1	in=0, out=20	Pass=0, Fail=14, Error=3	100.00 %
hackyStdExt\src\org\hackystat\stdext\workspace	1048	developer4 (time=8.75, com=55)	11.75 h	2.50 h	05-May-2003	09-Mar-2005	6	97	06-May-2003	22-Feb-2005	4	3717	0	2	5	LOC=1063, Class=12, Method=75	LOC=293, Class=5, Method=8	in=42, out=23	Pass=0, Fail=26, Error=57	95.45 %
hackyStdExt\src\org\hackystat\stdext\project\dailyanalysis	1047	developer3 (time=6.25, com=17)	7.08 h	1.50 h	17-Aug-2003	22-Jun-2004	3	24	17-Aug-2003	22-Jun-2004	4	1265	0	0	0	LOC=139, Class=2, Method=9	LOC=22, Class=1, Method=1	in=1, out=14	Pass=0, Fail=13, Error=12	87.50 %
hackyStdExt\src\org\hackystat\stdext\common\analysis\listsensordata	1044	developer1 (time=1.17, com=5)	1.17 h	1.17 h	03-Apr-2004	06-Apr-2004	1	6	20-Jan-2004	30-Sep-2004	2	97	0	0	0	LOC=80, Class=2, Method=2	LOC=19, Class=1, Method=1	in=0, out=16	Pass=0, Fail=43, Error=26	100.00 %
hackyStdExt\src\org\hackystat\stdext\common\analysis\objdatasummary	1040	developer1 (time=0.56, com=9)	0.56 h	0.50 h	11-Jan-2004	09-Apr-2004	1	10	15-Jan-2004	30-Sep-2004	2	129	0	0	0	LOC=22, Class=2, Method=5	LOC=10, Class=1, Method=1	in=0, out=18	Pass=0, Fail=18, Error=50	100.00 %
hackyStdExt\src\org\hackystat\stdext\common\analysis\sensordata\links	1035	developer1 (time=0.17, com=6)	0.17 h	0.17 h	03-Apr-2004	03-Apr-2004	1	7	20-Jan-2004	30-Sep-2004	2	65	0	0	0	LOC=75, Class=2, Method=2	LOC=10, Class=1, Method=1	in=0, out=10	Pass=0, Fail=13, Error=21	100.00 %
hackyStdExt\src\org\hackystat\stdext\dailyanalysis	1034	developer1 (time=0.42, com=11)	1.17 h	0.00 h	05-May-2003	26-May-2004	4	26	05-May-2003	31-Aug-2004	4	718	0	0	0	LOC=232, Class=4, Method=29	LOC=0, Class=0, Method=0	in=52, out=14	Pass=0, Fail=0, Error=0	90.00 %
hackyStdExt\src\org\hackystat\stdext\admin\analysis\configdisplay	1028	developer1 (time=2.42, com=10)	2.42 h	0.25 h	13-Oct-2003	29-May-2004	1	10	14-Oct-2003	25-Jun-2004	1	211	0	0	0	LOC=86, Class=3, Method=5	LOC=10, Class=1, Method=1	in=1, out=10	Pass=0, Fail=9, Error=5	100.00 %

E.9 Inspection 17

Table E.33. Inspection 17 - Package Details - Provides various information about the package that was inspected. See other Tables and Figures in this chapter for more detailed information about the package.

Packages	org.hackystat.stdext.project.cache
Module	hackyStdExt
Developer Ranking	N/A - hand selected via PRI ranking
PRI Ranking	LINI (See Figure E.4)
Product and Process Measures	See Figure E.4
Inspection Date	June 8, 2005
Jupiter Review ID	ProjectCache
Number of Inspectors	5
Meeting Attendance	4

Table E.34. Inspection 17 - Provides the valid defects found by the participants grouped by Severity.

Participant	Critical	Major	Normal	Minor	Trivial	Total
1		1	3			4
2		1				1
3		3	1	1	2	7
4		1	2		1	4
5		1				1
Total	0	7	6	1	3	17

Table E.35. Inspection 17 - Provides the valid defects found by the participants grouped by Type and Severity.

	Coding Standards	Program Logic	Optimization	Usability	Clarity	Suggestion
Critical						
Major		2	2			
Normal	1		3		2	
Minor						
Trivial	1					
Total	2	2	5	0	2	0

Table E.36. Inspection 17 - Provides the responses from the Post-Inspection-Questionnaire

Question	Yes	No
Did this package needed to be inspected?	2	2
Did you learned something from this inspection?	3	1
Did the inspection of this package increase its level of quality (once all the issues are resolved)?	4	0

Figure E.4. hackyStdExt PRI Ranking - Inspection 17

Workspaces (222):	Ranking	Expert	Active Time	Test Active Time	First Active Time	Last Active Time	Active Time Cont	Com mit Cont	Last Com mit	Com mit Cont	Code Churn	Re view	Last Re view	Open Issue	Close Issue	File Metric	Test File Metric	Depend	Unit Test Result	Coverage
hackyStdExt\src\org\backystat\stdext\project\	1376	developer1 (time=17.58, conts=52)	25.92 h	3.42 h	12-May-2003	01-Jun-2005	7	108	06-May-2003	17-Mar-2005	6	5202	18	0	1	LOC=1005, Class=7, Method=70	LOC=96, Class=2, Method=5	in=352, out=28	Pass=0, Fail=23, Error=21	78.00 %
hackyStdExt\src\org\backystat\stdext\project\cache\	1191		28.00 h	4.83 h	25-Feb-2004	16-Mar-2005	6	185	25-Feb-2004	17-Mar-2005	6	9421	0	0	0	LOC=294, Class=6, Method=32	LOC=32, Class=1, Method=2	in=41, out=17	Pass=0, Fail=78, Error=123	66.67 %
hackyStdExt\src\org\backystat\stdext\activity\analysis\	1112	developer4 (time=2.00, conts=8)	5.58 h	2.83 h	09-May-2003	13-Jul-2004	5	25	10-May-2003	09-Nov-2004	6	1470	0	0	0	LOC=167, Class=2, Method=5	LOC=22, Class=1, Method=1	in=0, out=30	Pass=0, Fail=46, Error=20	100.00 %
hackyStdExt\src\org\backystat\stdext\activity\analysis\time\	1072	developer4 (time=0.58, conts=13)	9.17 h	3.08 h	01-Jan-2003	06-Jun-2004	3	27	10-May-2003	25-Jun-2004	3	1681	0	0	0	LOC=133, Class=2, Method=3	LOC=25, Class=1, Method=1	in=0, out=26	Pass=0, Fail=0, Error=47	100.00 %
hackyStdExt\src\org\backystat\stdext\admin\analysis\serverstats\	1065	developer1 (time=0.58, conts=9)	0.83 h	0.42 h	08-May-2003	21-Apr-2004	4	10	08-May-2003	25-Jun-2004	2	537	0	0	0	LOC=240, Class=2, Method=19	LOC=13, Class=1, Method=1	in=1, out=19	Pass=0, Fail=18, Error=5	94.44 %
hackyStdExt\src\org\backystat\stdext\dailyanalysis\dataarray\	1064	developer4 (time=0.25, conts=11)	0.50 h	0.25 h	21-Aug-2003	18-Oct-2004	3	24	05-May-2003	19-Oct-2004	4	1111	0	0	0	LOC=431, Class=7, Method=49	LOC=149, Class=2, Method=9	in=45, out=4	Pass=0, Fail=0, Error=2	87.88 %
hackyStdExt\src\org\backystat\stdext\workspace\	1061	developer4 (time=8.75, conts=55)	11.75 h	2.50 h	05-May-2003	09-Mar-2005	6	97	06-May-2003	22-Feb-2005	4	3717	0	1	5	LOC=1063, Class=12, Method=75	LOC=293, Class=5, Method=8	in=142, out=23	Pass=0, Fail=26, Error=57	93.94 %
hackyStdExt\src\org\backystat\stdext\admin\analysis\adoption\	1054	developer1 (time=0.93, conts=9)	2.17 h	0.58 h	08-May-2003	26-Oct-2004	2	10	08-May-2003	26-Oct-2004	2	323	0	0	0	LOC=120, Class=4, Method=4	LOC=10, Class=1, Method=1	in=1, out=12	Pass=0, Fail=0, Error=6	100.00 %
hackyStdExt\src\org\backystat\stdext\common\alerts\pnedata\	1052	developer1 (time=0.25, conts=12)	0.50 h	0.33 h	05-May-2003	06-Apr-2004	3	20	05-May-2003	30-Jul-2004	3	431	0	0	0	LOC=112, Class=4, Method=11	LOC=16, Class=1, Method=1	in=2, out=15	Pass=0, Fail=14, Error=23	80.00 %
hackyStdExt\src\org\backystat\stdext\activity\analysis\project\lifetime\	1048	developer1 (time=0.33, conts=2)	0.33 h	0.17 h	24-Jun-2004	24-Jun-2004	1	4	25-Jun-2004	09-Nov-2004	3	232	0	0	0	LOC=132, Class=2, Method=5	LOC=20, Class=1, Method=1	in=0, out=20	Pass=0, Fail=14, Error=3	100.00 %
hackyStdExt\src\org\backystat\stdext\common\analysis\listsensordata\	1044	developer1 (time=1.17, conts=5)	1.17 h	1.17 h	03-Apr-2004	06-Apr-2004	1	6	20-Jan-2004	30-Sep-2004	2	97	0	0	0	LOC=80, Class=2, Method=2	LOC=19, Class=1, Method=1	in=0, out=16	Pass=0, Fail=23, Error=36	100.00 %
hackyStdExt\src\org\backystat\stdext\common\analysis\adoption\	1040	developer1 (time=0.58, conts=8)	0.58 h	0.50 h	11-Jan-2004	09-Apr-2004	1	10	12-Jan-2004	30-Sep-2004	2	129	0	0	0	LOC=92, Class=2, Method=5	LOC=18, Class=1, Method=1	in=0, out=18	Pass=0, Fail=0, Error=50	100.00 %
hackyStdExt\src\org\backystat\stdext\common\analysis\sensordata\links\	1035	developer1 (time=0.17, conts=6)	0.17 h	0.17 h	03-Apr-2004	03-Apr-2004	1	7	20-Jan-2004	30-Sep-2004	2	65	0	0	0	LOC=75, Class=2, Method=2	LOC=10, Class=1, Method=1	in=0, out=10	Pass=0, Fail=13, Error=21	100.00 %
hackyStdExt\src\org\backystat\stdext\project\dailyanalysis\	1035	developer3 (time=6.25, conts=17)	7.08 h	1.50 h	17-Aug-2003	22-Jun-2004	3	24	17-Aug-2003	22-Jun-2004	4	1265	0	0	0	LOC=139, Class=2, Method=9	LOC=22, Class=1, Method=1	in=1, out=14	Pass=0, Fail=13, Error=12	75.00 %
hackyStdExt\src\org\backystat\stdext\workspace\map\javanap\	1035	developer2 (time=12.58, conts=41)	14.42 h	3.33 h	06-May-2003	17-Mar-2005	3	52	06-May-2003	17-Mar-2005	3	2119	0	4	4	LOC=772, Class=4, Method=44	LOC=221, Class=2, Method=12	in=14, out=29	Pass=0, Fail=60, Error=70	97.50 %
hackyStdExt\src\org\backystat\stdext\workspace\batern\	1029	developer3 (time=2.25, conts=12)	2.42 h	1.33 h	21-Jul-2004	09-Mar-2005	3	13	20-Jul-2004	22-Nov-2004	2	1305	0	0	0	LOC=700, Class=5, Method=23	LOC=211, Class=2, Method=7	in=74, out=10	Pass=0, Fail=12, Error=1	74.07 %

Bibliography

- [1] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.
- [2] Robert G. Ebenau. Predictive quality control with software inspections. *Cross Talk: The Journal of Defense Software Engineering*, June 1994.
- [3] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 2001.
- [4] R. L. Glass. *Facts and facilities of software engineering*. Pearson Education, Inc., Boston, MA, 2003.
- [5] Martin Bush and Norman E. Fenton. Software measurement: A conceptual framework. *Journal of Systems and Software*, 12:223–231, 1990.
- [6] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [7] Susan H. Strauss and Robert G. Ebenau. *Software Inspection Process*. McGraw-Hill, 1994.
- [8] Karl E. Wieggers. When two eyes aren't enough. *Software Development*, 9(10), March/April 2001.
- [9] 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.
- [10] 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.
- [11] Tom Gilb. Optimizing software inspections. *Crosstalk*, (3), March 1998.

- [12] Tom Gilb. Software inspections are not for quality, but for engineering economics. *IEEE Software on Inspection*, 1999.
- [13] Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Journal of Empirical Software Engineering*, 4(1), January 1998.
- [14] Philip M. Johnson. Reengineering inspection: The future of formal technical review. *Communications of the ACM*, 41(2), February 1998.
- [15] 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.
- [16] Karl E. Wieggers. Seven deadly sins of software reviews. *Software Development*, 6(3), March 1998.
- [17] Karl E. Wieggers. *Peer reviews in software: A practical guide*. Addison-Wesley, Boston, MA, 2002.
- [18] Robert L. Glass. Inspections-some surprising findings. *Communications of the ACM*, 42(4), 1999.
- [19] Jasper Kamperman. How to jump-start inspection by outsourcing. *StickyMinds.com*, 2005.
- [20] Daniel M. Berry. The inevitable pain of software development: Why there is not silver bullet. In *RISSEF*, pages 50–74. Springer, 2004.
- [21] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Oct 2002.
- [22] Software Engineering Technical Committee of the IEEE Computer Society. IEEE standard glossary of software engineering terminology. *IEEE-STD-729-1983 (New York; IEEE)*, 1983.
- [23] Philip Johnson. Hackystat framework. Technical report, Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawaii, January 2005.
- [24] The IEEE. IEEE standard for software reviews and audits. ANSI/IEEE STD 1028-1988, IEEE Computer Society, 1988.

- [25] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, October 1989.
- [26] Danu Tjahjono. *Exploring the effectiveness of formal technical review factors with CSRS, a collaborative software review system*. Ph.D. thesis, Department of Information and Computer Sciences, University of Hawaii, August 1996.
- [27] D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews*. Little, Brown, 4th edition, 1990.
- [28] Nachiappan Nagappan, Laurie A. Williams, John P. Hudepohl, Will Snipes, and Mladen Vouk. Preliminary results on using static analysis tools for software inspection. In *ISSRE*, pages 429–439, 2004.
- [29] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *The International Conference on Software Engineering*, pages 580–586, 2005.
- [30] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [31] Gerd Kohler, Heinrich Rust, and Frank Simon. An assessment of large object oriented software systems: A metrics based approach. In *Object-Oriented Product Metrics for Software Quality Assessment Workshop on 12th European Conference on Object-Oriented Programming*, pages 16–23, 1998.
- [32] Philip M. Johnson and Michael G. Paulding. Understanding HPCS development through automated process and product measurement with hackystat. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.
- [33] Michael G. Paulding. Measuring the processes and products of HPCS development: Initial results for the optimal truss purpose-based benchmark. Technical Report CSDL-04-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, September 2004.
- [34] Philip M. Johnson, Hongbing Kou, Michael G. Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Improving software development management through software project telemetry. *IEEE Software*, August 2005.

- [35] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from hackystat-uh. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, Los Angeles, California, August 2004.
- [36] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Christopher Chan, Carleton A. Moore, Jitender Miglani, Shenyan Zhen, and William E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.
- [37] Philip M. Johnson. The Hackystat-JPL configuration: Overview and initial results. Technical Report CSDL-03-07, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, October 2003.
- [38] Aaron Kagawa and Philip M. Johnson. The Hackystat-JPL configuration: Round 2 results. Technical Report CSDL-03-07, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, May 2004.
- [39] Stuart Faulk, Philip M. Johnson, John Gustafson, Adam A. Porter, Walter Tichy, and Larry Votta. Measuring HPC productivity. *International Journal of High Performance Computing Applications*, December 2004.
- [40] Aaron Kagawa. Hackystat MDS supporting MSL MMR. Technical Report CSDL-04-06, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, June 2004.
- [41] JESS, The Rule Engine for the Java Platform. <<http://http://herzberg.ca.sandia.gov/jess/>>.
- [42] Aaron A. Kagawa. Snapshot Sensor Data Type Enhancements. <<http://hackystat.org/hackyDevSite/doc/SnapshotEnhancements.html>>.
- [43] The Jupiter Code Review Eclipse Plugin. <<http://csdl.ics.hawaii.edu/Tools/Jupiter/>>.
- [44] LOCC code counter. <<http://csdl.ics.hawaii.edu/Tools/LOCC/>>.
- [45] Philip Johnson. Software Review Guidelines. <<http://hackystat.org/hackyDevSite/doc/Review.html>>.