# Limited Resource Software Inspections

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

## Abstract

*Imagine that your project manager has budgeted 200 person-hours for the next month to inspect newly created source code. Unfortunately, in order to inspect all of the documents adequately, you estimate that it will take 400 person-hours. However, your manager refuses to increase the budgeted resources for the inspections. How do you decide which documents to inspect and which documents to skip?*

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

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

*To accomplish this research, I will construct an evaluation framework based upon automated process and product measures to distinguish documents that are in "most need of inspection" from those in "least need of inspection". Some examples of the process and product measures that are being considered include: reported defects, unit tests, test coverage, active time, and number of changes. Based on this framework I hypothesize, that code deemed in most need of inspection will generate more critical issues than code deemed least need of inspection.*

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

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

## 1. Introduction

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

In another success story, the Jet Propulsion Laboratory adopted inspections to identify defects and experienced a savings of 7.5 million dollars by conducting 300 inspections [2]. This statistic is very impressive, however what is not usually emphasized is that each inspection had an average cost of 28 hours. Using that

average cost, the total cost for JPL's inspection process was 8,400 hours or roughly 4 years of work. This illustrates a fundamental problem with inspections; better results come from greater investment [5].

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

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

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

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

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

My thesis claims are as follows:

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

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

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

## 2. Related Work

## 3. Hackystat LRSI Extension

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

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

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

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

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

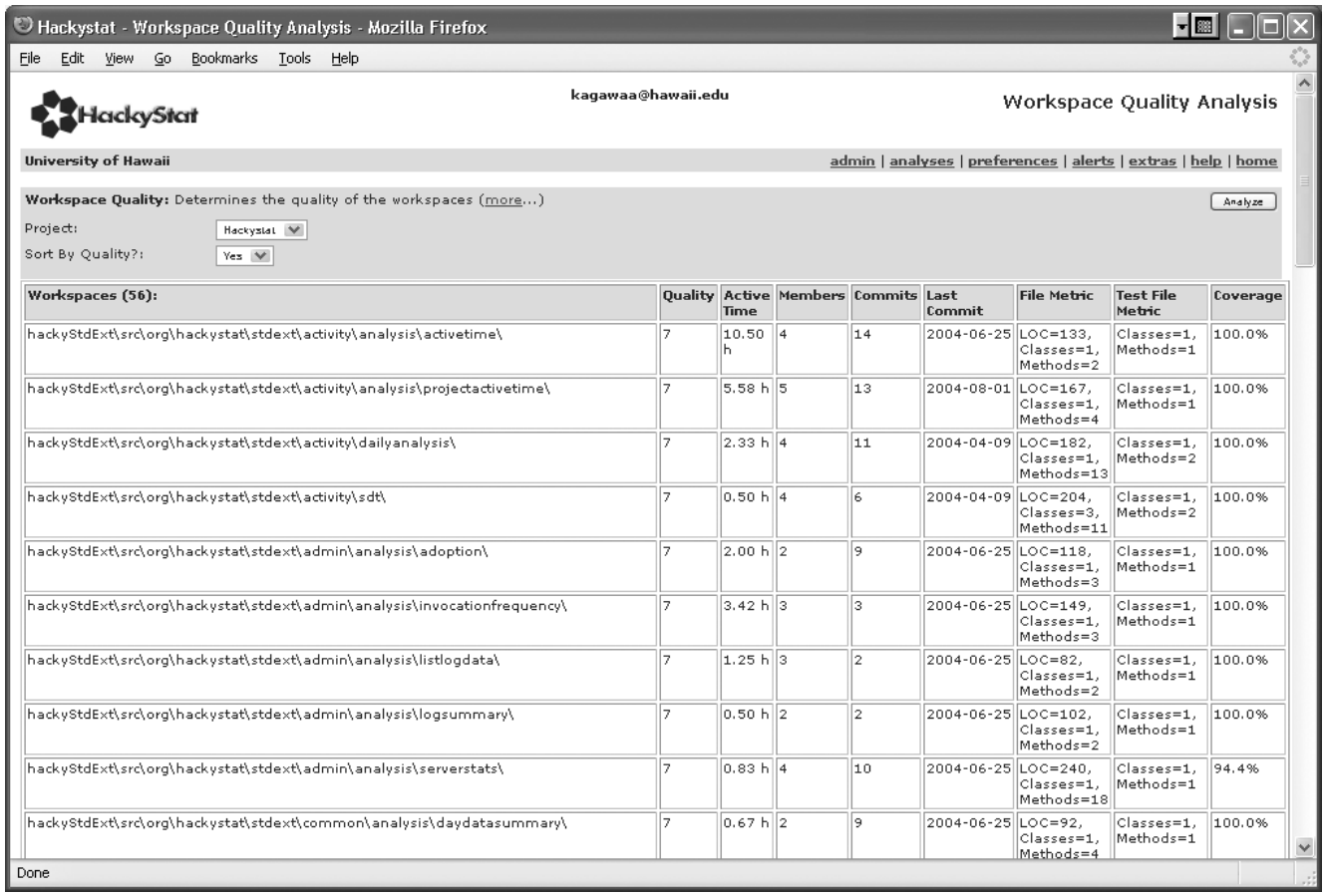
There are several issues with the assignment of numerical weights (or quality levels as I call them) that I still need to address. For example, I explicitly determine the quality levels using my own subjective measure of what is low versus high quality. I will need to explore if my subjective measure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

## 4. Evaluation Methodology

This section discusses the proposed evaluation methodology of this research. The main thesis of this work is that Limited Resource Software Inspection (LRSI) can distinguish documents that are in most need of inspection from those in least need of inspection.

One way of implementing LRSI is through Hackystat, thus I will create a Hackystat Extension called hackyLRSI. This extension will provide an analysis, which will determine what documents are in "most need of inspection" from documents that are in "least need of inspection". For this specific implementation, this determination is based on a numerical weighting system of different process and product measures. Some measures include: reported defects, unit tests, test coverage, active time, and number of changes. Each measure will be assigned a numerical weight and will be individually calibrated.

It is important to note two limitations of this research. First, I am not defining a set of attributes that represent the determination of most and least need of inspection for all software projects. Instead, by using hackyLRSI I will be able to go through a methodol-




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

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

ogy to best calibrate the attributes to accurately reflect the determination for the project that I am studying. Second, LRSI is only beneficial to organizations that have limited inspection resources. LRSI will not benefit an organization that has the necessary resources to thoroughly inspect every document. For these organizations it does not matter if one document is in more need of inspection over another, since they will inspect everything.

In this evaluation, I will study the implementation and inspection process of the Hackystat System developed in the Collaborative Software Development Laboratory (CSDL), of the University of Hawaii at Manoa. Like most organizations, CSDL's inspection resources are limited and therefore inspections are conducted on a weekly basis regardless of the number of "ready" documents. In addition, unlike most organizations who conduct Software Inspection and have limited resources, CSDL does not conduct sampling or inspections on up-stream documents to enhance the inspection process as recommended by Tom Gilb [5]. CSDL does not follow these recommendations for two reasons. First, CSDL does not have enough resources to conduct sampling. Second, Hackystat does not contain many up-stream requirement and design documents. Hackystat lacks these up-stream documents primarily because CSDL develops Hackystat on its own and not for a client.

A quick note: Hackystat does not contain documents per se and because CSDL primarily inspects source code grouped by package, I will use the term 'packages' when referring to CSDL's use of LRSI. The term 'documents' will still be used when referring to

the general idea of inspections.

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

The use of CSDL in my study indicates another limitation on this research. The most accurate and thorough evaluation of LRSI includes making a determination about most and least need of inspection and actually inspecting *all* the documents to test if that determination is correct. However, because I am using CSDL's inspection resources, which are limited, this is not possible.

Briefly talk about volunteering here.

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

1. LRSI can help constrain the area of volunteering.
2. LRSI can identify documents that need to be inspected that cannot be identified by volunteering.
3. Documents that is deemed in most need of inspection will generate more critical issues than documents deemed in least need of inspection.

The next sections will detail each of these claims and the methodologies used in their evaluation.

The following table provides a timeline for the Evaluation of this thesis:

| Timeline | Evaluation Activity |
|---|---|
| January 10, 2005 | Request developer workspace rankings |
| Januray 14, 2005 | Process developer responses and create a plan of what will be inspected |
| January 17, 2005 | Start 4 weeks of inspection, inspecting 2 packages a week |
| Feburary 14, 2005 | Hand pick 2 packages to inspect that was not volunteered |

### 4.1 Claim 1: Constrained area of volunteering

One of the benefits of LRSI is that it constrains the area in which developers can volunteer their code for inspection. In Software Inspection, the area of possible inspection includes *all* the documents currently moving through the development cycle. In LRSI, this area will be constrained to the documents that are in most need of inspection. This smaller LRSI area seems to be advantageous for organizations that cannot inspect every document, because it will eliminate the need and the more importantly it limits the possibility of inspecting code that is deemed in least need of inspection.

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

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

This method works well if the developer can be trusted to pick the right code to inspect. However, developers often do not know where every critical issue will appear. In other words, leaving this decision up to the subjective understanding of a developer is error prone [evidence?].

LRSI provides an alternative solution to this limited resource problem. Instead of leaving the decision of what code to inspect entirely up to the developer, LRSI can constrain the number of possibilities by providing a smaller area of selection. For this fictional organization, the developer can look up what code is in most need of inspection and choose code from this smaller list.

To evaluate this claim, I will ask the developers of Hacksytat to provide a numerical ranking, based on their subjective feelings, of what packages they would volunteer for inspection. The ranking will indirectly indicate the packages that are in most and least need of inspection. With these results I will be able to compare the developers' subjective rankings to the LRSI most and least need of inspection determination. This evaluation will indicate whether LRSI is really needed. For

example, the findings could indicate that developers can correctly distinguish, using their own subjective understandings, what packages need to be inspected and packages that do not need to be inspected.

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

The following steps will occur in this evaluation:

1. Obtain the rankings of packages from each individual developer.
2. Analyze the difference between the developers' ranking against the LRSI most and least need of inspection determination.
3. Conduct the following inspections:
   (a) Inspect 2 packages, where the developer and the LRSI determinations agree, that are in most need of inspection.
   (b) Inspect 2 packages, where the developer and the LRSI determinations agree, that are in least need of inspection.
   (c) Inspect 2 packages where the developer and the LRSI determinations disagree. The developer provides a low ranking but the LRSI claims that the package is in most need of inspection.
   (d) Inspect 2 packages where the developer and the LRSI determinations disagree. The developer provides a high ranking but the LRSI claims that the package is in least need of inspection.
4. Analyze the results of each inspection, which includes correlating the number of critical issues generated with both the developer ranking and the LRSI determination. In addition, I may ask the developers for explanations of their rankings where applicable.
5. After each inspection I will adjust LRSI calibration or add new product and process measures as necessary.

There are three possible results of this study. First, I may find that developers automatically have a sense of what code is in most need of inspection and in least need of inspection. This would indicate that LRSI provides little added value. Second, developers provide high rankings for both most and least need of inspection packages. Essentially, this will indicate that sometimes the developers are correct and sometimes they are wrong. And third, developers have no idea what code needs to be inspected. The last two results will indicate that LRSI provides some benefit.

In addition, this evaluation will provide more data to refine the calibrations of the measures that are used for the LRSI determination. For example, if a developer rates a package very highly, LRSI finds that package to be deemed least need of inspection, and many critical issues are found, then this indicates that the LRSI determination is flawed. Therefore, the LRSI determination needs to be recalibrated to include this document. In addition to calibration, more process and product measures could be introduced. This event, although detrimental to the previous LRSI determination, will provide more data for calibration and the addition of new measures that will hopefully lead to a better and more accurate LRSI determination.

### 4.2 Claim 2: LRSI is better than volunteering

Another benefit of LRSI is that it can find areas of the system that is in most need of inspection and has not been identified using the volunteering process. Organizations that have limited inspection resources simply cannot volunteer and inspect every single line of code. If these organizations blindly volunteer or pick and choose documents for inspection they could possibly be missing some areas of the system that need inspection the most.

A real example of this benefit is the following:

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

This situation is quite dangerous, because as we all know a software system evolves and outdated packages may become error prone. Therefore, it is important to realize that not only active packages need to be inspected but old packages can also be deemed in most need of inspection. Software Inspection [5] 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 this organization does not have the resources to inspect every document moving through the development cycle, it is very likely that some documents that make it through the cycle will have bugs in it. Therefore, ensuring that even these documents are included as potential inspection candidates is very important.

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

The following steps will occur in this evaluation:

1. Select a few packages that were not investigated in the previous study and has been deemed in most and least need of inspection.

2. Conduct inspections on those packages.

3. Analyze the results of each inspection, which includes correlating the number of critical issues generated with the LRSI determination.

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

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

### 4.3 Claim 3: Most need of inspection versus least need of inspection

The last benefit of LRSI is, documents that is deemed in most need of inspection will generated more critical issues than documents deemed in least need of inspection. This claim is critically important for LRSI's success.

However, if a package is identified as least need of inspection and yields many critical issues, then LRSI determination is flawed. I will use this information to refine the LRSI determination. It is my hope that in the end of the study I will have been able to successfully calibrate the LRSI determination for the Hackystat project.

During the evaluations of the previous two claims CSDL will have conducted at least 12 inspections. In addition, I have and will collect information on past and future inspections on Hackystat packages. In total, I believe I will have data on 20 inspections along with the information on the LRSI determinations.

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

To evaluation this claim, I will monitor the validity of the LRSI determination, and adjusting it as necessary, throughout each inspection. To accomplish this, I will collect specific pieces of information when conducting inspections. The following is a specific list of the information that is being collected:

- Inspection date
- Hacksytat module, package, and inspection ID
- LRSI determination (most need of inspection or least need of inspection)
- LRSI measures and values

- Subjective discussion of the validity of the LRSI determination before the inspection
- Number of issues generated and the categorization of these issues according to severity
- Retrospective discussion after the inspection was conducted to indicated possible areas of improvement.

This information will help me keep track of the progress of the inspections and the validity of the LRSI determination. As I previously stated, the calibration of the LRSI determination is an ongoing and evolving process. This information will help keep track of that evolution.

The end goal of this information collection is to create a best practices recommendation of the types of process and product measures and their calibration that will provide the best LRSI results for different projects.

### 4.4 Initial Results of Evaluation

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

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

## 5. Contributions

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

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

## 6. Timeline

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

## References


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

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

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

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

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

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

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

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