Understanding the Sources of Variation in Software Inspections

Adam Porter*, Harvey Siy Audris Mockus, Lawrence Votta
University of Maryland Bell Laboratories†

April 30, 1997

Abstract

In a previous experiment, we determined how various changes in three structural elements of the software inspection process (team size, and number and sequencing of sessions), altered effectiveness and interval. Our results showed that such changes did not significantly influence the defect detection rate, but that certain combinations of changes dramatically increased the inspection interval.

We also observed a large amount of unexplained variance in the data, indicating that other factors must be affecting inspection performance. The nature and extent of these other factors now have to be determined to ensure that they had not biased our earlier results. Also, identifying these other factors might suggest additional ways to improve the efficiency of inspections.

Acting on the hypothesis that the "inputs" into the inspection process (reviewers, authors, and code units) were significant sources of variation, we modeled their effects on inspection performance. We found that they were responsible for much more variation in defect detection than was process structure. This leads us to conclude that better defect detection techniques, not better process structures, are the key to improving inspection effectiveness.

The combined effects of process inputs and process structure on the inspection interval accounted for only a small percentage of the variance in inspection interval. Therefore, there must be other factors which need to be identified.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging–*Code inspections and walk-throughs*

General Terms: Experimentation

Additional Key Words and Phrases: Statistical models, empirical studies, software inspection, software process

1 Introduction

Software inspection has long been regarded as a simple, effective, and inexpensive way of detecting and removing defects from software artifacts. Most organizations follow a three-step procedure

*This work is supported in part by a National Science Foundation Faculty Early Career Development Award, CCR-9501354. Dr. Siy was also partly supported by AT&T 's Summer Employment Program.

†Authors' addresses: A. Porter and H. Siy, Department of Computer Science, University of Maryland, College Park, MD 20742, {aporter,harvey}@cs.umd.edu; A. Mockus and L. Votta, Software Production Research Department, Bell Laboratories, Lucent Technologies, Naperville, IL 60566, {audris,votta}@research.bell-labs.com

of Preparation, Collection, and Repair. First, each member of a team of reviewers reads the artifact, detecting as many defects as possible (Preparation). Next, the review team meets, looks for additional defects, and compiles a list of all discovered defects (Collection). Finally, these defects are corrected by the artifact's author (Repair).

Several variants of this approach have been proposed in order to improve inspection performance. These include Fagan Inspections[12], Active Design Reviews[21], N-Fold Inspections[26], Phased Inspections[15], and Two-Person Inspections[4]. Most involve restructuring the process, e.g., rearranging the steps, changing the number of people working on each step, or the number of times each step is executed [25]. Some of these variants have been evaluated empirically. However, focus has been on their overall performance. Very few investigations attempted to isolate the effects due specifically to structural changes. However, we must know which effect are caused by which changes in order to determine the factors that drive inspection performance, to understand why one method may be better than another, and to focus future research on high-payoff areas.

Therefore, we conducted a controlled experiment in which we manipulated the structure of the inspection process[25]. We adjusted the size of the team and the number of sessions. Defects were sometimes repaired in between multiple sessions and sometimes not. Comparing the effects of different structures on inspection effectiveness and interval[1] indicated that none of the structural changes we investigated had a significant impact on effectiveness, but some changes dramatically increased the inspection interval.

Regardless of the treatment used, both the effectiveness and interval data seemed to vary widely. To strengthen the credibility of our previous study and to deepen our understanding of the inspection process, we must now study this variation.

1.1 Problem Statement

We are asking two questions: (1) Are the effects of process structure obscured by other sources of variation, i.e., is the "signal" swamped by "noise"? (2) Are the effects of other factors more influ-

[1] Inspections have many different costs and benefits. In this study we restricted our discussion of benefits to the number of defects found, and costs to inspection interval (the time from the start of the inspection to its completion) and person effort.

ential than the effects of process structure, i.e., are researchers focusing on the wrong mechanisms?

To answer the first question, we will attempt to separate the effects of some external sources of variation from the effects due to changes in the process structure. By eliminating the effects of external variation we will have a more accurate picture of the effects of our experimental treatments. Also, by understanding the external variation we may be able to evaluate how well our experimental design controlled for it, which will aid the design of future experiments.

To answer the second question, we will compare the variation due to process structure with that due to other sources. If the other sources are more influential than process structure, then it may be possible to significantly improve inspections by properly manipulating them. We expect that identifying and understanding these sources will aid the development of better inspection methods.

Therefore, we have extended the results of our experiment by identifying some sources of variation and modeling their influence on inspection effectiveness and interval. We will show that our previous results do not change even after these sources of variation are accounted for. This analysis also suggests some improvements for the inspection process and raises some implications about past research and future studies.

1.2 Analysis Philosophy

We hope to identify mechanisms that drive the costs and benefits of inspections so that we can engineer better inspections. To do this we will rely heavily on statistical modeling techniques. However, these techniques are not completely automated. Therefore, we must make judgments about which variables or combinations of variables to allow in the models. These choices are guided by our desire to create models that are robust and interpretable.

To improve robustness we avoided fitting the data with too many factors. Doing so could result in a model that explains much of the variation in the current data, but has no predictive power when used on a different set of data.

To improve interpretability we omitted factors for which we have no readily available measure. We also omitted factors whose effects were known to be confounded with other factors in the model. Finally, we rejected models for which, based on our experience, we could not argue that their variables were causal agents of inspection performance. Specifically, there are four conditions that must be satisfied before factor A can be said to cause response B[14]:

1. A must occur before B.

2. A and B must be correlated.

3. There is no other factor C that accounts for the correlation between A and B.

4. A mechanism exists that explains how A affects B.

One implication of all these is that the "best" model for our purpose is not necessarily the one that explains the largest amount of variation. Throughout this research we have chosen certain models over others. Some were rejected because a smaller, but equally effective model could be found, or because one variable was strongly confounded with another, or because a variable failed to show a causal relationship with inspection performance. We will point out these cases as they arise.

2 Summary of Experiment

With the cooperation of professional developers working on an actual software project at Lucent Technologies (formerly AT&T Bell Labs), we conducted a controlled experiment to compare the costs and benefits of making several structural changes to the software inspection process. (See Porter, et al.[25] for details.) The project was to create a compiler and environment to support developers of Lucent Technologies' 5ESS(TM) telephone switching system. The finished compiler contains over 55K new lines of C++ code, plus 10K which was reused from a prototype. (See Appendix A for a description of the project.)

The inspector pool consisted of the 6 developers building the compiler plus 5 developers working on other projects.[2] They had all been with the organization for at least 5 years and had similar development backgrounds. In particular, all had received inspection training at some point in their

[2] In addition, 6 more developers were called in at one time or another to help inspect 1 or 2 pieces of code, mostly to relieve the regular pool during the peak development periods. It is common practice to get non-project developers to inspect code during peak periods.

careers. Data was collected over a period of 18 months (June 1994 to December 1995), during which 88 code inspections were performed.

2.1 Experimental Design

We hypothesized that (1) inspections with large teams have longer intervals, but find no more defects than smaller teams; (2) multiple-session inspections[3] are more effective than single-session inspections, but at the cost of a significantly longer interval; and (3) although repairing the defects found in each session of a multiple-session inspection before starting the next session will catch even more defects, it will also take significantly longer than multiple sessions meeting in parallel.

We manipulated these independent variables: the number of reviewers (1, 2, or 4); the number of sessions (1 or 2); and, for multiple sessions, whether to conduct the sessions in parallel or in sequence. The treatments were arrived at by selecting various combinations of these (e.g., 1 session/4 reviewers, 2 sessions/2 reviewers without repair, etc.).

Among the dependent variables measured were inspection effectiveness—in terms of observed number of defects, as explained in Appendix B—and inspection interval—in terms of working days from the time the code was made available for inspection up to the collection meeting.[4]

2.2 Conducting the Experiment

To support the experiment, one of us joined the development team in the role of inspection quality engineer (IQE). He was responsible for tracking the experiment's progress, capturing and validating data, and observing all inspections. He also attended the development team's meetings, but had no development responsibilities. The inspection process used in this study is outlined in Appendix D.

When a code unit was ready for inspection, the IQE randomly assigned a treatment and randomly drew the review team from the inspector pool. In this way, we attempted to control for differences in natural ability, learning rate, and code unit quality.

The names of the reviewers were then given to the author, who scheduled the collection meeting.

[3] In this experiment, we used the term "session" to mean one cycle of the preparation-collection-repair process. In multiple-session inspections, different teams inspect the same code unit.

[4] For 2-session inspections, the longer interval of the two is selected.

If the treatment called for 2 sessions, the author scheduled 2 separate collection meetings. If repair was required between the 2 sessions, then the second collection meeting was not scheduled until the author had repaired all defects found in the first session.

The reviewers were expected to prepare sufficiently before the meeting. During preparation, reviewers did not merely acquaint themselves with the code, but carefully examined it for defects. Their instructions were to verify that the code correctly implemented the associated design. They were also asked to ensure that the code met applicable coding standards, and was, in their opinion, readable and maintainable. They were not given any specific technical roles (e.g., tester or end-user) nor any checklists. On an individual preparation form, they recorded the time spent on preparation, and the page and line number and the description of each issue (each "suspected" defect).[5] The experiment placed no limit on preparation time.

For the collection meeting one reviewer was selected as the moderator and another as the reader. The moderator ran the meeting and recorded administrative data on a moderator report form. This comprised the name of the author, lines of code inspected, hours spent testing the code before inspection, and inspection team members. The reader reads through the code and orally summarizes each section (paraphrasing). During this activity, reviewers brought up any issues found during preparation or briefly discussed newly discovered issues. On a collection form, the code unit's author recorded the page and line number and description of each issue regarded as valid, as well as the start and end time of the collection meeting. Each valid issue was tagged with a unique Issue ID. If a reviewer had found that particular issue during preparation, he or she recorded that ID next to the issue on his or her preparation form. This enabled us to trace issues back to the reviewers who found them. No limit was placed on meeting duration, although most lasted less than 2 hours.

After the collection meeting, the author kept the collection form and resolved all issues. In the process he or she recorded on a repair form the disposition (no change, fixed, deferred), nature (non-issue, optional, requires change not affecting execution, requires change affecting execution), locality (whether repair is isolated to the inspected code), and effort spent ($\leq 1hr, \leq 4hr, \leq 8hr, > 8hr$) on

[5] A sample of this, and all other forms we used may be found at `http://www.cs.umd.edu/users/harvey/variance.html`.

each issue. Afterwards, the author returned all paperwork to us. We used the information from the repair form and interviews with the author to classify each issue as a true defect (if the author was required to make an execution affecting change to resolve it), soft maintenance issue (any other issue which the author fixed), or false positive (any issue which required no action).

In the course of the experiment, several treatments were discontinued because they were either not performing effectively, or were taking too long to complete. These were the 1-session, 1-person treatment and all 2-session treatments which required repair between sessions.

After 18 months, we managed to collect data from 88 inspections, with a combined total of 130 collection meetings and 233 individual preparation reports. The entire data set may be examined online at `http://www.cs.umd.edu/users/harvey/variance.html`.

2.3 Self-Reported Data

Self-reported data tend to contain systematic errors. Therefore we minimized the amount of self-reported data by employing direct observation[23] and interviews[2]. The IQE attended 125 of the 130 collection meetings[6] to make sure the meeting data was reported accurately and that reviewers do not mistakenly add to their preparation forms any issues that were not found until collection. We also made detailed field notes to corroborate and supplement some of the data in the meeting forms. The repair information was verified through interviews with the author, who completed the form. Our defect classification was not made available to the reviewers or the authors to avoid biasing them.

Among the data that remained self-reported were the amount of preparation time and pre-inspection testing time expended. We had two concerns in dealing with these data: a participant might deliberately fail to tell the truth (e.g., reporting 2 hours preparation time when he or she really did not prepare at all); participants might make errors in recording data (e.g., reporting 2 hours of preparation time when the correct figure was 1.9 hours).

During the experiment, the IQE had an office next to those of the compiler development team, and after working with the team for 18 months, a great deal of trust was built up. Also, the

[6] The unattended ones are due to schedule conflicts and illness.

development environment routinely collects self-reported data, which is unavailable to management at the individual level. Thus developers are conditioned to answer as reliably as they can. We therefore see no reason to suspect that participants ever deliberately misrepresented their data.

As for the element of error, previous observational studies on time usage conducted in this environment have shown that although there are always inaccuracies in self-reported data, the self-reported data is generally within 20% of the observed data[22].

2.4 Results of the Experiment

Our experiment produced three general results:

1. Inspection interval and effectiveness of defect detection were not significantly affected by team size (large vs. small).

2. Inspection interval and effectiveness of defect detection were not significantly affected by number of sessions (single vs. multiple).

3. Effectiveness of defect detection was not improved by performing repairs between sessions of two-session inspections. However, inspection interval was significantly increased.

From this we concluded that single-session inspections by small teams were the most efficient, since their defect detection rate was as good as that of other formats, and inspection interval was the same or less.

The observed number of defects and the intervals per treatment are shown as boxplots[7] in Figures 1 and 2, respectively. The treatments are denoted [**1**,or **2**] **s**essions **X** [**1**,**2**, or **4**] **p**ersons [**N**o-repair,**R**epair]. (For example, the label 2sX1pN indicates a two-session, one-person, without-repair inspection.) It can be seen that most of the treatment distributions are similar but that they vary widely within themselves.

[7] We have made extensive use of boxplots to represent data distributions. Each data set is represented by a box whose height spans the central 50% of the data. The upper and lower ends of the box marks the upper and lower quartiles. The data's median is denoted by a bold line within the box. The dashed vertical lines attached to the box indicate the tails of the distribution; they extend to the standard range of the data (1.5 times the inter-quartile range). All other detached points are "outliers."[6]

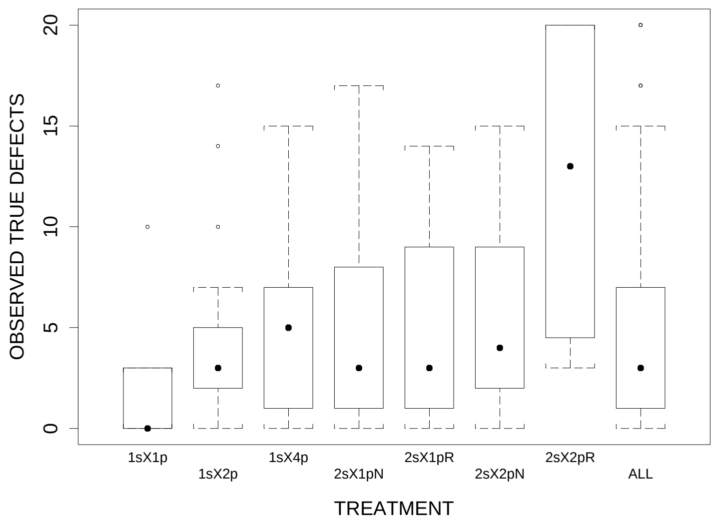

Figure 1: **Observed Number of Defects by Treatment.** The treatment labels are interpreted as follows: the first digit stands for the number of sessions, the second digit stands for the number of reviewers per session, and, for 2-session inspections, the 'R' or 'N' suffix indicates "with repair" or "no repair". As seen here, the distributions all seem to be similar except for 1sX1p and 2sX2pR, which were discontinued after 7 and 4 data points, respectively.

3 Sources of Variation

3.1 Process Inputs as Sources of Variation

In addition to the process structure, we see that differences in process inputs (e.g., code unit and reviewers) also affects inspection outcomes. Therefore, we will attempt to separate the effects of process inputs from the effects of the process structure. To do this we will estimate the amount of variation contributed by these process inputs.

Thus, our first question from Section 1.1 may be refined as, How will our previous results change when we eliminate the contributions due to variability in the process inputs?

Our second question becomes, (1) Are the differences due to process inputs significantly larger than the differences in the treatments? (2) If so, what factors or attributes affecting the variability of these process inputs have the greatest influence?

Figure 3 is a diagram of the inspection process and associated inputs, e.g., the code unit, the reviewers, and the author. It shows how these inputs interact with each process step. This is an

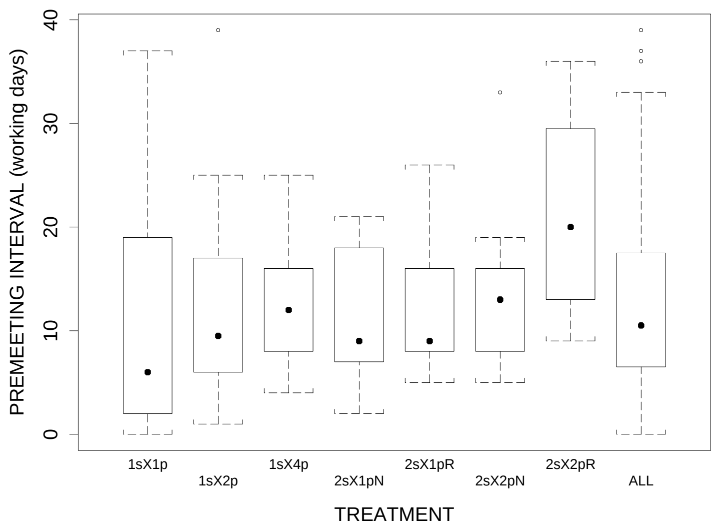

Figure 2: **Pre-meeting Interval by Treatment.** As seen here, the distributions all seem to be similar except for 2sX2pR, which was significantly higher.

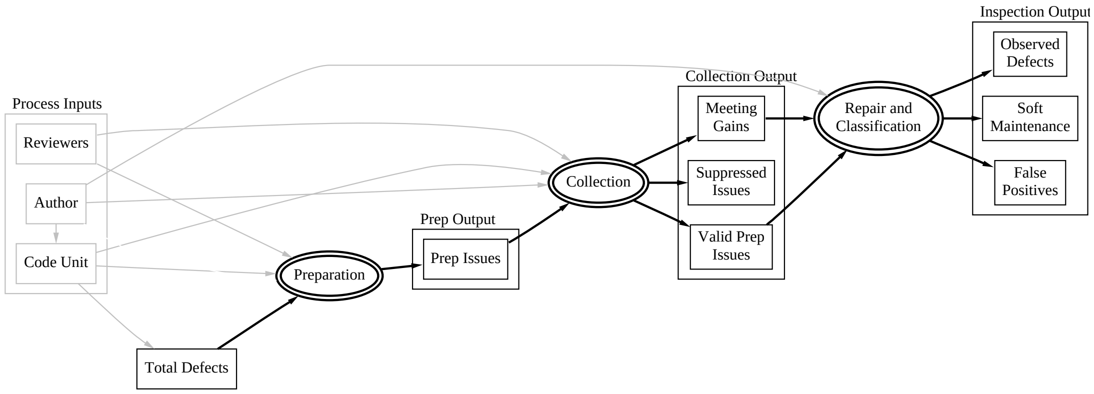

Figure 3: **A Cause and Effect Diagram of the Inspection Process.** The inputs to the process (reviewers, author, and code unit) are shown in grey rectangles on the left, the solid ovals represent process steps, the grouped boxes in between steps show the intermediate outputs of the process. Time flows left to right.

example of a cause-and-effect diagram, similar to the ones used in practice[16], but customized here for our use.

The number and types of issues raised in the preparation step are influenced by the reviewers selected and by the number of defects originally in the code unit (which in turn may be affected by the author of the code unit). The number and types of issues recorded in the collection step are influenced by the reviewers on the inspection team and the author (who joins the collection

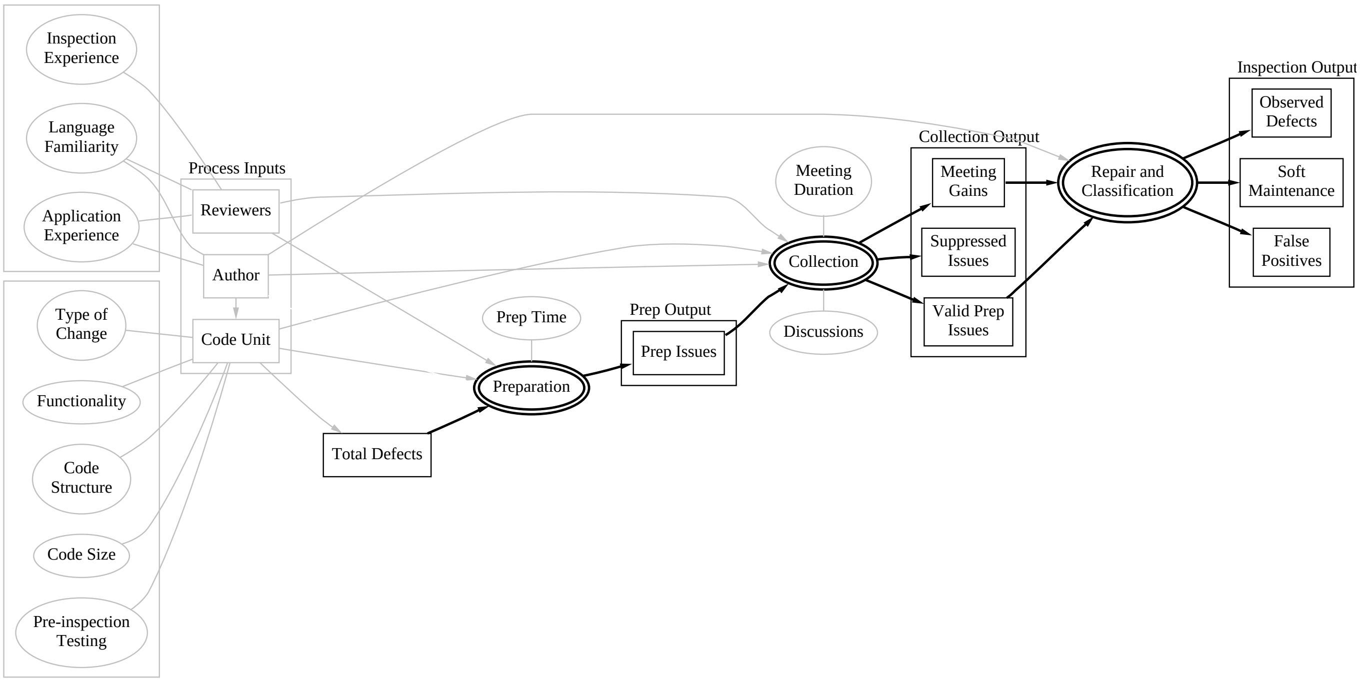

Figure 4: **The Refined Cause and Effect Diagram.** This figure extends the inspection model with some of the factors which we believe to affect reviewer and author performance and code unit quality.

meeting), the number of issues raised in preparation, and the number remaining undetected in the code unit.

3.2 Factors Affecting Inspections

We considered the factors affecting reviewer and author performance and code unit quality that might systematically influence the outcome of the inspection. (Some of these are shown in Figure 4.) In 3.2.1 through 3.2.3, we examine these factors, explain how they might influence the number of defects, and discuss confounding issues. [8] As we examine them, we caution the reader from making conclusions about the significance of any factor as a source of variation. The goal here is to establish possible mechanisms, not to test significance of correlation. Each plot is meant to be descriptive, showing the relationship of a factor against the number of defects, without eliminating the influence of potentially confounding factors. The actual test of a factor's significance will be carried out when

[8] We did not have any readily available measure of experience nor code complexity, so we did not include them in our analysis.

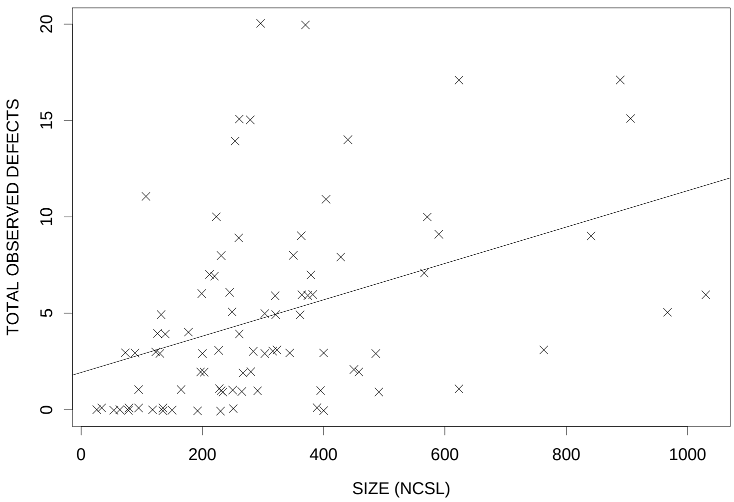

Figure 5: **Size vs. Defects Found.** This is a scatter plot showing the relation between the size of the code and the number of defects found (cor = 0.40). The line indicates the trend of the data. Note that the plot was "jittered" – a small, random offset was added to each point – to expose overlapping points. (In fact, every scatter plot in this paper that may have overlapping points was jittered.)

the model is built. (See Section 4.)

3.2.1 Code Unit Factors

Some of the possible variables affecting the number of defects in the code unit include: size, author, time period when it was written, and functionality.

Code Size. The size of a code unit is given in terms of non-commentary source lines (NCSL). It is natural to think that, as the size of the code increases, the more defects it will contain. From Figure 5 we see that there is some correlation between size and number of defects found (cor = 0.4).[9]

Author. The author of the code may inadvertently inject defects into the code unit. There were 6 authors in the project. Figure 6 is a boxplot showing the number of defects found, grouped according to the code unit's author. The number of defects could depend on the author's level of understanding and implementation experience.

[9]Correlations calculated in this paper are Pearson correlation coefficients.

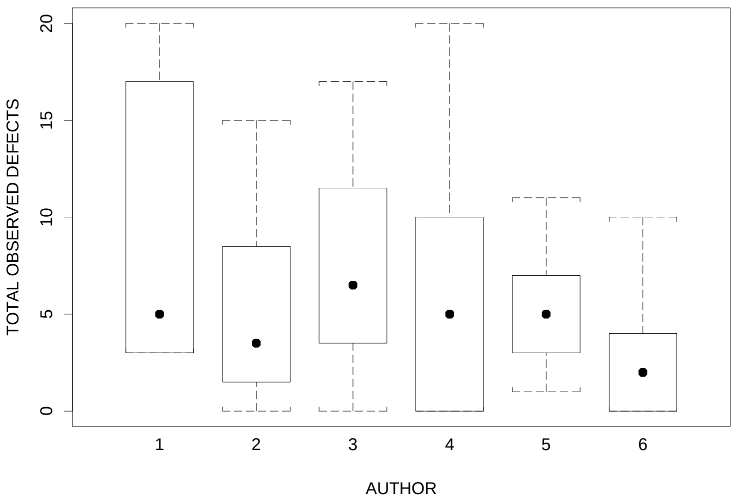

Figure 6: **Defects Found In Authors' Code Units.** These boxplots show the total defects found in each inspection, grouped according to the code unit's author.

Development Phase. The performance of the reviewers and the number of defects in the code unit at the time of inspection might well depend also on the state of the project when the inspection was held. Figure 7 is a plot of the total defects found in each inspection, in chronological order. Each point was plotted in the order the code unit became available for inspection. There are two distinct distributions in the data. The first calendar quarter of the project (July - September 1994) – which has about a third of the inspections – has a significantly higher mean than the remaining period. This coincided with the project's first integration build. With the front end in place, the development team could incrementally add new code units to the system, possibly with a more precise idea of how the new code is supposed to interact with the integrated system, resulting in fewer misunderstandings and defects. In our data, we tagged each code unit as being from "Phase 1" if they were written in the first quarter and "Phase 2" otherwise.

At the end of Phase 1, we met with the developers to evaluate the impact of the experiment on their quality and schedule goals. We decided to discontinue the 2-session treatments with repair because they effectively have twice the inspection interval of 1-session inspections of the same team size. We also dropped the 1-session, 1-person treatment because inspections using it found the lowest number of defects.

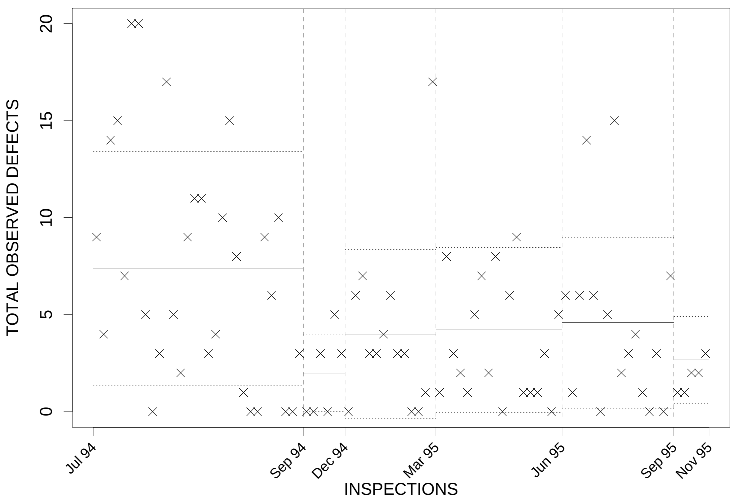

Figure 7: **Defects Detected Over Time.** This is a plot of the inspection results in chronological order showing the trends in number of defects found over time. The vertical lines partition the plot into calendar quarters. Within each quarter, the solid horizontal line marks the mean of that quarter's distribution. The dashed lines mark one standard deviation above and below the mean.

Figure 8 shows a time series plot of the number of issues raised for each code unit inspection. While the number of true defects being raised dropped as time went by, the total number of issues did not. This might indicate that either the reviewers' defect detection performance were deteriorating in time, or the authors were learning to prevent the true defects but not the other kinds of issues being raised.

Functionality. Functionality refers to the compiler component to which the code unit belongs, e.g., parser, symbol table, code generator, etc. Some functionalities may be more straightforward to implement than others, and, hence, will have code units with lower number of defects. Figure 9 is a boxplot showing the number of defects found, grouped according to functionality.

Table 1 shows the number of code units each author implemented within each functional area. Because of the way the coding assignments were partitioned among the development team, the effects of functionality are confounded with the author effect. For example, we see in Figure 9 that `SymTab` has the lowest number of defects found. However, Table 1 shows that almost all the code units in `SymbTab` were written by author 6, who has the lowest number of reported defects. Nevertheless, we may still be able to speculate about the relative impact of the two factors by

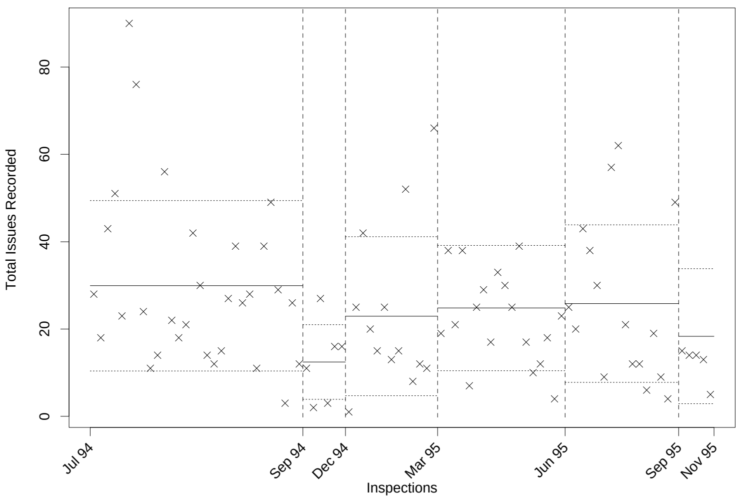

Figure 8: **Number of Issues Recorded Over Time.** This is a time series plot showing the trends in number of issues being recorded over time. The vertical lines partition the plot into quarters. Within each quarter, the solid horizontal line marks the mean of that quarter's distribution. The dashed lines mark one standard deviation above and below the mean. Note that the scale of the y-axis is different from the previous figure.

examining those functionalities with more than one author (`CodeGen`) and authors implementing more than one functionality (author 6).

In addition, functionality is also confounded with development phase as Phase 1 had most of the code for the front end functionalities (input-output, parser, symbol table) while Phase 2 had the back end functionalities (code generation, report generation, libraries).

Because author, phase, and functionality are related, they cannot all be considered in the model as they account for much of the same variation. In the end, we selected functionality as it is the easiest to explain.

Pre-inspection Testing. The code development process employed by the developers allowed them to perform some unit testing before the inspection. Performing this would remove some of the defects prior to the inspection. Figure 10 is a scatter plot of pre-inspection testing effort against observed defects in inspection (cor = 0.15). One would suspect that the number of observed defects would go down as the amount of pre-inspection testing goes up, but this pattern is not observed in Figure 10.

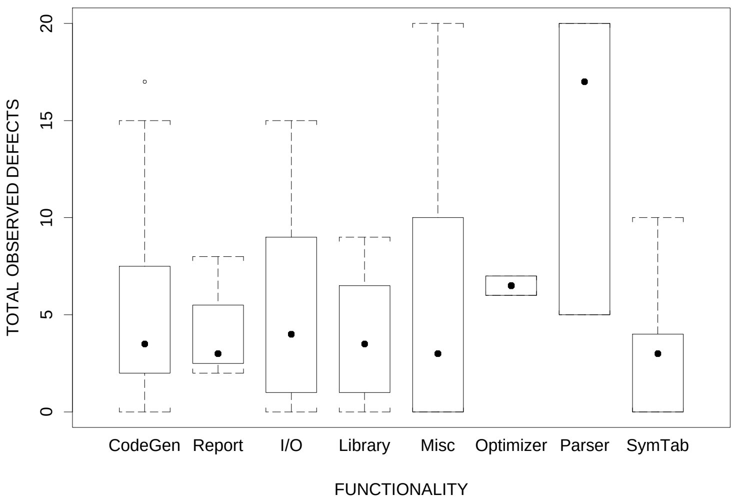

Figure 9: **Defects Found In Code Units Classified by Functionality.** These boxplots show the total defects found in each inspection, grouped according to the code unit's functionality. Note that specific authors were assigned to implement specific portions of the project's functionalities, so the effects of functionality is usually not separable from that of authors – the independent factors of author and functionality are confounded. For example, `SymTab`, which has the lowest number of defects found, was implemented by author 6, who has the lowest number of reported defects.

A possible explanation to this is that testing patterns during code development may have changed across time. As the project progressed and a framework for the rest of the code was set up, it may have become easier to test the code incrementally during coding. This may result in code which has different defect characteristics compared to code that was written straight through. It would be interesting to do a longitudinal study to see if these areas had high maintenance cost.

3.2.2 Reviewer Factors

Here we examine how different reviewers affect the number of defects detected. Note that we only look at their effect on the number of defects found in preparation, because their effect as a group is different in the collection meeting's setting.

Reviewer. Reviewers differ in their ability to detect defects. Figure 11 shows that some reviewers find more defects than others.[10] Even for the same code unit, different reviewers may

[10]In addition to the 11 reviewers, 6 more developers were called in at one time or another to help inspect 1 or 2 pieces of code, mostly to relieve the regular pool during the peak development periods. We did not include them in

	Author						
Functionality	1	2	3	4	5	6	Subtotals
CodeGen			8	6	8	6	28
Report		3					3
I/O		9					9
Library						12	12
Misc				11			11
Optimizer					2		2
Parser	4						4
SymTab	2					17	19
Subtotals	6	12	8	17	10	35	88

Table 1: **Assignment of Authors to Functionality.** Each cell gives the number of code units implemented by an author for a functionality.

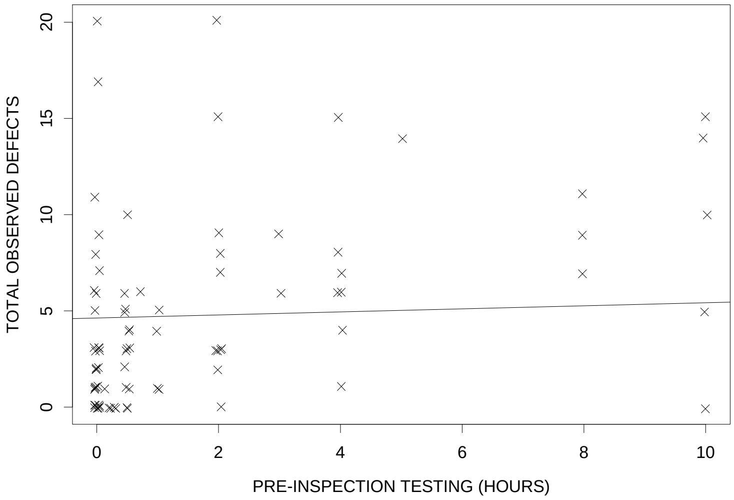

Figure 10: **Pre-inspection Testing Effort vs. Defects Found.** This is a scatter plot showing how the amount of pre-inspection testing related to the number of defects found in inspection (cor = 0.15). Note that the pre-inspection testing data was self-reported by the author. Points cluster at the quarter hours because we asked the authors to only record to that precision.

find different numbers of defects (Figure 12). This may be because they were looking for different kinds of issues. Reviewers may raise several kinds of issues, which may either be suppressed at the meeting, or classified as true defects, soft maintenance issues (issues which required some non-behavior-affecting changes in the code, like adding comments, enforcing coding standards, etc.),

this analysis because they each had too few data points.

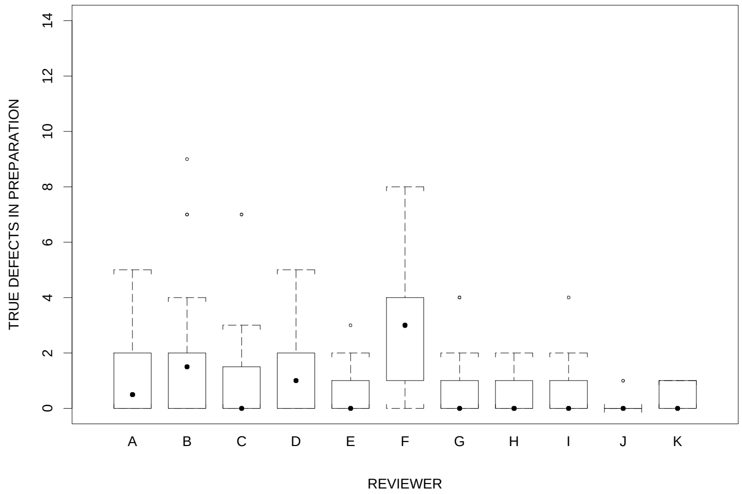

Figure 11: **Number of Defects in Preparation per Reviewer.** This plot shows the number of true defects found in preparation by each reviewer.

or false positives (issues which were not suppressed at the meeting, but which the author later regarded as non-issues). Figure 13 shows the mean number of issues raised by each reviewer as well as the percentage breakdown per classification. We see that some of the reviewers with low numbers of true defects (see Figure 11), like Reviewers H and I, simply do not raise many issues in total. Others, like Reviewers J and K, raise many issues but most of them are suppressed. Still others, like Reviewers E and G, raise many issues but most turn out to be soft maintenance issues. The members of the development team (Reviewers A to F) raise on average more total issues (see left plot in Figure 13), though a very high percentage turn out to be soft maintenance issues (see right plot in Figure 13), possibly because, as authors of the project, they have a higher concern for its long-term maintainability than the rest of the reviewers. An exception is Reviewer F, who found almost as many true defects as soft maintenance issues.

Preparation Time. The amount of preparation time is a measure of the amount of effort the reviewer put into studying the code unit. For this experiment, the reviewers were not instructed to follow any prescribed range of preparation time, but to study the code in as much time as they think they need. Figure 14 plots preparation time against defects found, showing a positive trend but little correlation (cor = 0.26).

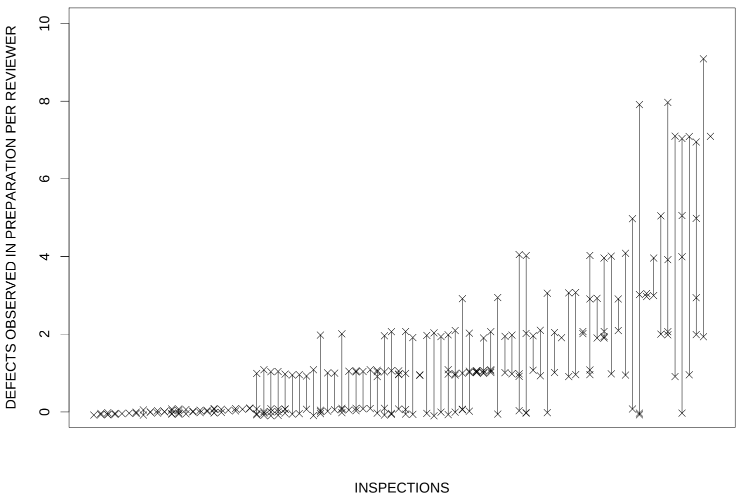

Figure 12: **Reviewer Performance Per Inspection.** This shows the number of defects found in preparation by each reviewer, grouped according to inspection. Each column represents one inspection. The points in that column represent the number of true defects reported during preparation by each reviewer in that inspection. The columns were ordered according to increasing means.

Even if preparation time is found to be a significant contributor, it must be noted that preparation time depends not only on the amount of effort the reviewer is planning to put into the preparation, but also on factors related to the code unit itself. In particular, it is influenced by the number of defects existing in the code, i.e., the more defects he finds, the more time he spends in preparation. Hence, high preparation time may be considered a consequence, as well as a cause, of detecting a large number of defects. Further investigation is needed to quantify the effect of preparation time on defects found as well as the effect of defects found on preparation time. Because there is no way to tell how much of the preparation time was due to reviewer effort or number of defects, we decided not to include it in the model. This is also in keeping with our analysis philosophy to only consider factors that occur strictly before the response. (See Section 1.2.)

3.2.3 Team Factors

Team-specific variables also add to the variance in the number of meeting gains.

Team Composition. Since different reviewers have different abilities and experiences, and possibly interact differently with each other, different teams also differ in combined abilities and

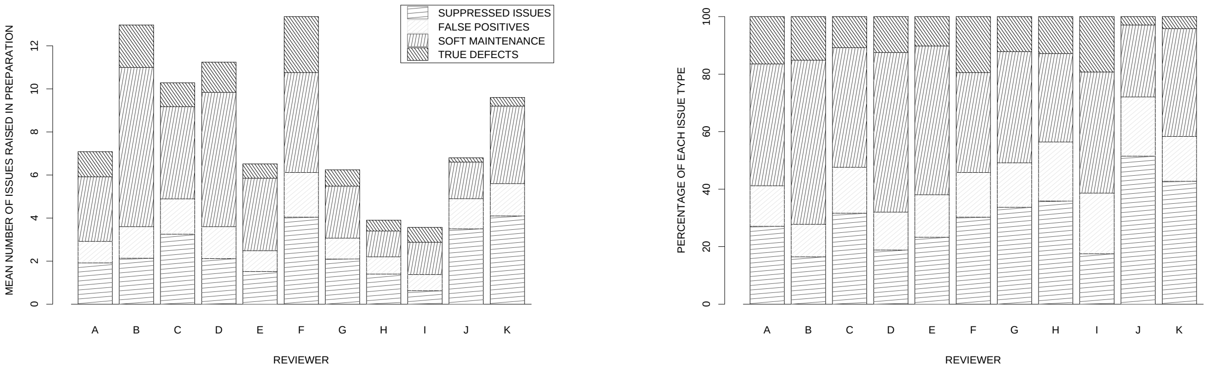

Figure 13: **Classification of Issues Found in Preparation.** The bar graph on the left shows the mean number of issues found in preparation by each reviewer, broken down according to issue classification. The bar graph on the right shows the percentage breakdown.

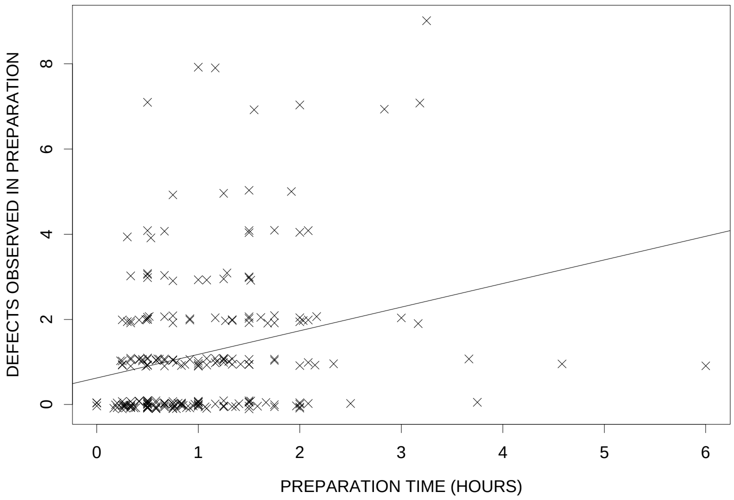

Figure 14: **Preparation Time vs. Defects Found In Preparation.** This is a scatter plot showing how the amount of preparation time related to the number of defects found in preparation (cor = 0.26).

experiences.

Apparently, this mix tended to form teams with nearly the same performance. This is illustrated in Figure 15 which shows number of defects found by different 2-person teams in each 2sX2pN inspection. Most of the time, the two teams found nearly the same number of defects. This may be due to some interactions going on between team members. However, because teams are formed

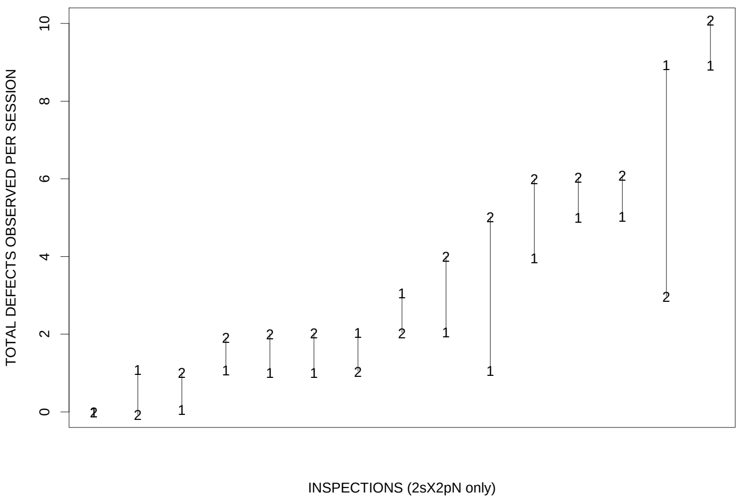

Figure 15: **Team Performance Per Inspection (2sX2pN only).** This shows the total number of defects found per session in each 2sX2pN inspection. Each column represents one inspection. The points in that column represent the total number of true defects reported in preparation and meeting by each team. "1" and "2" plot the number of defects found by the first and second teams, respectively. The columns are ordered by mean defects found.

randomly, there are only a few instances where teams composed of the same people were formed more than once, not enough to study the interactions.

We incorporated the team composition into the model by representing it as a vector of boolean variables, one variable per reviewer in the reviewer pool. When a particular reviewer is in that collection meeting, his corresponding variable is set to "True".

Meeting Duration. The meeting duration is the number of hours spent in the meeting. In the meeting, one person is appointed the reader, and he reads out the code unit, paraphrasing each chunk of code. The meeting revolves around him. At any time, reviewers may raise issues related to the particular chunk being read and a discussion may ensue. All these contribute toward the pace of the meeting. The meeting duration is positively correlated with the number of meeting gains, as shown in Figure 16 (cor = 0.57). As with the case of preparation time, the meeting duration is partly dependent on the number of defects found, as detection of more defects may trigger more discussions, thus lengthening the duration. It is also dependent on the complexity or readability of the code. Further investigation is needed to determine how much of the meeting duration is

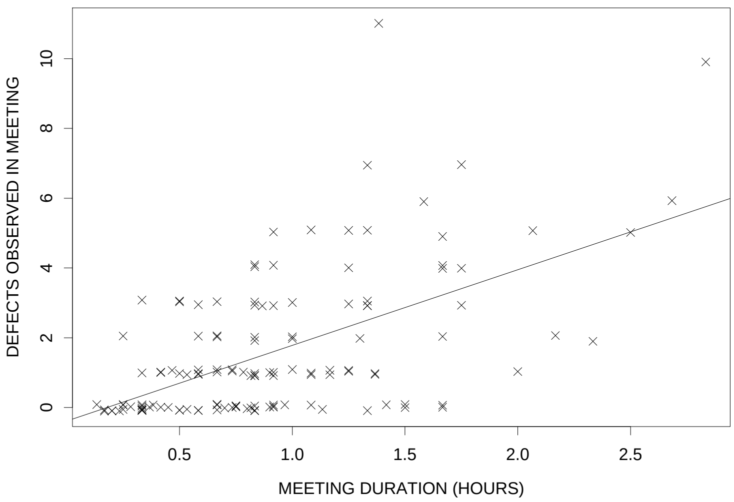

Figure 16: **Meeting Duration vs. Defects Found in Meeting.** This is a scatter plot showing how the amount of time spent in the meeting related to the number of defects found in the meeting (cor = 0.57).

due to the team effort independent of the complexity and quality of the code being inspected. For similar reasons as with preparation time (see the previous discussion on preparation time), we did not include this in the model.

Combined Number of Defects Found in Preparation. The number of defects already found going into the meeting may also affect the number of defects found at the meeting. Each reviewer gets a chance to raise each issue he found in preparation as a point of discussion, possibly resulting in the detection of more defects. Figure 17 shows some correlation between number of defects found in the preparation and in the meeting (cor = 0.4).

4 A Model of Inspection Effectiveness

4.1 Building the Model

To explain the variance in the defect data, we built statistical models of the inspection process, guided by what we knew about it. Model building involves formulating the model, fitting the model, and checking that the model adequately characterizes the process. We built the models in

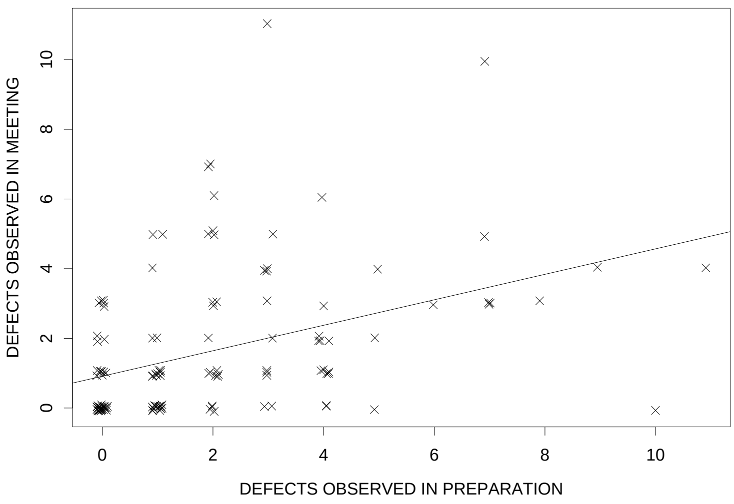

Figure 17: **Defects Found in Preparation vs. Defects Found in Meeting.** This is a scatter plot showing how the combined amount of defects found in the preparation related to the number of defects found in the meeting (cor = 0.4).

the S programming language[3, 7].

Using the factors described in the previous section, we modeled the number of defects found with a generalized linear model (GLM) from the Poisson family.[11] We started with a model which had all code unit factors, all reviewers, and the original treatment factors, represented by the following formula:[12]

$$\begin{aligned} Defects \quad \sim \quad & TeamSize + Sessions + Repairs + Phase + Author + Func + \log(Size) + \\ & R_A + R_B + R_C + R_D + R_E + R_F + R_G + R_H + R_I + R_J + R_K \end{aligned} \tag{1}$$

In this model, Functionality and Author are categorical variables represented in S as sets of dummy variables[7, pp. 20-22,32-36]. They have 7 and 5 degrees of freedom, respectively.

Stepwise model selection heuristic[13] selected the following model.

[11] The generalized linear model and the rationale for using it are explained in Appendix C.

[12] We used S language notation to represent our models[7, pp. 24-31]. For example, the model formula $y \sim a + b$ is read as, "y is modeled by a, and b." Furthermore, in this section, y is modeled by a and b with Poisson errors and logarithmic link function. In other words, y has Poisson distribution with mean $\exp(\beta_0 + \beta_1 a + \beta_2 b)$.

[13] Stepwise model selection techniques are a heuristic to find the best-fitting models using as few parameters as possible. To avoid overfitting the data, the number of parameters must always be kept small or the residual degrees of freedom high. To perform stepwise model selection we used the `step()` function in S[7, pp. 233-238].

$$Defects \sim TeamSize + Sessions + Repairs + Phase + Author + Func + \log(Size) + R_B + R_C + R_F + R_G + R_H + R_I$$

This resulting model is not satisfactory because it retained many factors, making it difficult to interpret. Also, even though these factors were considered important by the stepwise selection criteria, some of them do not explain a lot of the variance. So we increased the selection threshold to produce a smaller model.[14] Increasing the selection threshold did not simplify the model initially, until, at one point, a large number of factors were suddenly dropped. The resulting model then was:

$$Defects \sim Phase + \log(Size) + R_B + R_F$$

It must be noted that the factors left out of the model are not necessarily unimportant. We believe that there are other possible models for our data. In particular, Phase was considered important. Phase is a surrogate variable representing the change in defects being found over time. Figure 7 clearly showed that something had changed over time but it is not clear what caused it. The reason why this change over time explains a significant part of the variability may be attributable to other factors. It is not clear which mechanism explains why Phase affects the number of defects. We also knew that Phase was confounded with Functionality (e.g., parser was implemented before code generator). Since we knew also that some parts of the compiler are harder to implement than others, the effects due to Functionality are easier to interpret than the effects due to Phase. Thus we replaced Phase by Functionality in our final model:

$$Defects \sim Func + \log(Size) + R_B + R_F$$

The analysis of variance for this model is in Table 2. For comparison, the treatment factors were added to the model. See Appendix C for details on calculating the significance values. The resulting model explains $\sim 50\%$ of the variance using just 10 degrees of freedom.

[14] In S, increase the `scale` parameter of the `step()` function.

	Factor	Degrees of Freedom	Sum of Squares	F Value	Pr(F)	Effect
Treatment factors	Team Size	2	2.65	0.50	0.6062	
	Sessions	1	1.12	0.43	0.5146	
	Repair	1	4.01	1.53	0.2207	
Input factors	log(Code Size)	1	59.63	22.66	0.0000	+
	Functionality	7	43.76	2.38	0.0303	
	R_B	1	32.60	12.39	0.0007	+
	R_F	1	34.67	13.17	0.0005	+
	Residuals	73	192.11			

Table 2: **Factors Affecting Inspection Effectiveness.** The sum of squares give an indication of the relative amount each factor contributes to the variance of the defect data. (For each factor, we first account for the variance explained by all the other factors and then calculate the amount this this factor contributes. Note: these values are dependent on the ordering of the factors and thus, are not additive, i.e., adding up the sum of squares of all the significant factors will not necessarily give us the 50% variance explained by the model.) The probabilities indicate the significance of the contribution. The last column for each significant scalar factor indicates whether the factor was a positive or negative contributor to the number of defects. (Functionality had 7 degrees of freedom and different functionalities had different effects.)

In this model, Defects is the number of defects found in each of the 88 inspections. Note that the presence of certain reviewers (Reviewers B and F) in the inspection team strongly affects the outcome of the inspection. (See Table 2.) Note also the log transformation on the Code Size factor. We do not really know what the actual underlying functional relationship is between Code Size and Defects and so we applied square root, logarithmic, and linear transformations. Code Size explained more variance under the log transformation than under other transformations.

Figure 18 gives diagnostic plots of the model's goodness-of-fit. The left plot shows the values estimated by the model compared to the original values. It shows that the model reasonably estimates the number of defects. The right plot shows the values estimated by the model compared to the residuals. The residuals appear to be independent of the fitted values, suggesting that the residuals are random.

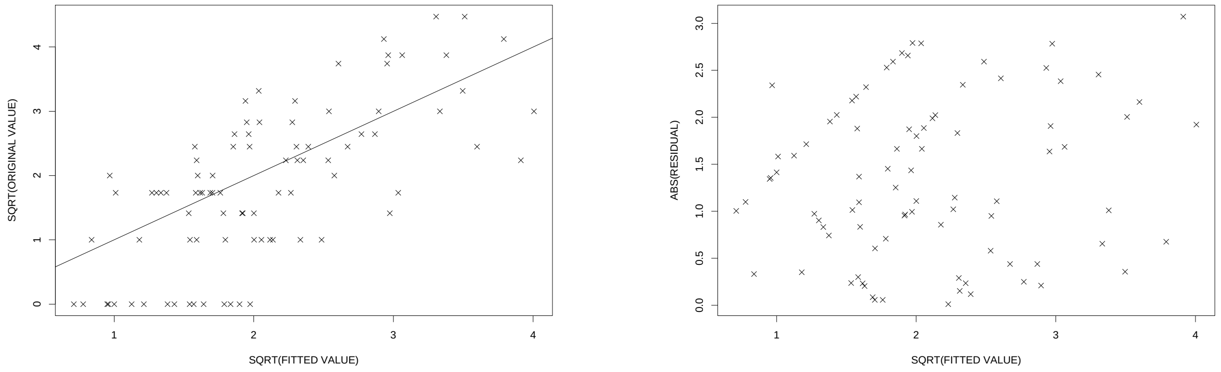

Figure 18: **Examining the fit of the model.** The left plot compares the values estimated by the model with the original values (a perfect fit would imply that everything is on the line y = x). There is a substantial correlation between the two (cor = 0.69). The right plot shows the relation of the fitted values to the residuals. The residuals appear to be independent of the fitted values.

4.2 Lower Level Models

The inspection model is a high level description of the inspection defect detection process. The effects of the process input and of the process structure can be compared using this model. But we also know that defect detection in inspections is performed in two steps: preparation and collection. These two steps may be considered as independent processes which can be modeled separately. Doing so has several advantages. We can understand the resulting models of the simpler separate processes better than the model for the composite inspection process. In addition, there are more data points to fit – 233 individual preparations and 130 collection meetings, as opposed to 88 inspections.

4.2.1 A Model for Defect Detection During Preparation

To build the preparation model, we started with the same variables as in inspection model 1. Since the same code unit was inspected several times, we added a categorical variable, CodeUnit, to the regression model. CodeUnit is a unique ID for each code unit inspected.

Using stepwise model selection, we selected the variables that significantly affect the variance in the preparation data. These were Functionality, Size, and Reviewers B, E, F, and J. This is represented by the model formula:

$$PrepDefects \sim Func + \log(Size) + R_B + R_E + R_F + R_J$$

In this model, *PrepDefects* is the number of defects found in each of the 233 preparation reports. The presence of all the significant factors from the overall model at this level gives us more confidence on the validity of the overall model.

4.2.2 A Model for Defect Detection During Collection

We started with the same variables as in preparation model. (See previous section.) Using stepwise model selection to select the variables that significantly affect the meeting data we ended up with Functionality, Size, and the presence of Reviewers B, F, H, J, and K. This is represented by the model formula:

$$MeetingGains \sim Func + \log(Size) + R_B + R_F + R_H + R_J + R_K$$

In this model, *MeetingGains* is the number of defects found in each of the 130 collection meetings. This is again consistent with the previous two models.

4.3 Answering the Questions

We are now in a position to answer the questions raised in Section 3.1 with respect to inspection effectiveness.

4.3.1 Will previous results change when process inputs are accounted for?

In this analysis, we build a GLM composed of the significant process input factors plus the treatment factors and check if their contributions to the model would be significant.

The effect of increasing team size is suggested by plotting the residuals of the overall inspection model, grouped according to Team Size (Figure 19(a)). We observe no significant difference in the distributions. When we included the Team Size factor into the model, we saw that its contribution

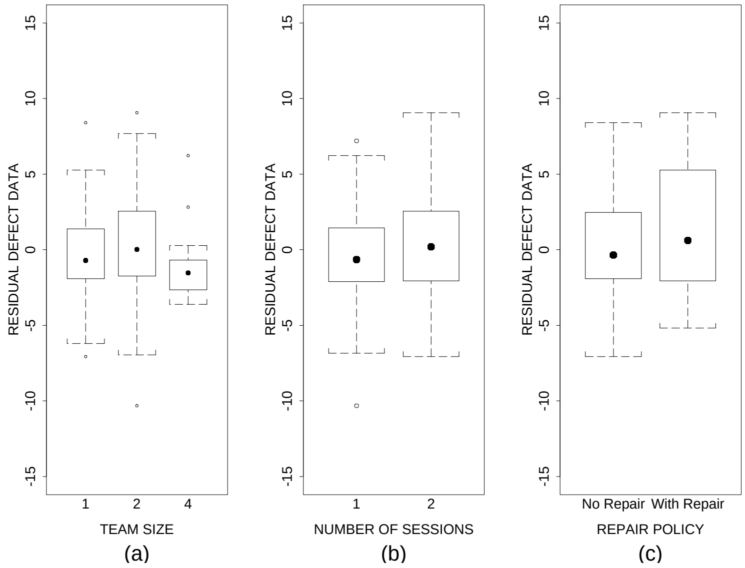

Figure 19: **Examining the Significance of the Experimental Treatment Factors.** These three panels depict the distribution of the residual data grouped according to Team Size, Sessions, and Repair.

was not significant ($p = 0.6$, see Table 2).[15]

The effect of increasing sessions is suggested by plotting the residuals of the overall inspection model, grouped according to Session (Figure 19(b)). We observe no significant difference in the distributions. When we included the Session factor into the model, we saw that its contribution was not significant ($p = 0.5$).

The effect of adding repair is suggested by plotting the residuals of the overall inspection model (for those inspections that had 2 sessions), grouped according to Repair policy (Figure 19(c)). We observe no significant difference in the distributions. When we included the Repair factor into the model, we saw that its contribution was not significant ($p = 0.2$).

4.3.2 Did the design spread process inputs uniformly across treatments?

We want to determine if the factors of the process inputs which significantly affect the variance are spread uniformly across treatments. This is useful in evaluating our experimental design. Although randomization guarantees that the long run distribution of the factors will be independent of the

[15] Appendix C Section C.3.1 describes how Tables 2 and 3 were constructed.

treatments, we had a single set of 88 data points. Thus we felt it is important to know of any imbalances in this particular randomization.

A visual inspection shows that the distribution of

Our examination of how reviewers were assigned to treatments shows that Reviewer F did not participate in any 1sX1p inspections. A similar analysis of how treatments were assigned to functionality shows 22 (out of 56) zero entries indicating uneven assignment of treatments to functionalities. Fifteen of the zeros are for the three treatments (1sX1p, 2sX1pR, 2sX2pR) that were discontinued.

4.3.3 Are differences due to process inputs larger than differences due to process structure?

Table 2 shows the analysis of variance for our model. The significance of the treatment factors' contribution were included for comparison.

The table shows that differences in code units and reviewers drive inspection performance more than differences in any of our treatment variables. This suggests that relatively little improvement in effectiveness can be expected of additional work on manipulating the process structure.

4.3.4 What factors affecting process inputs have the greatest influence?

The dominance of process inputs over process structure in explaining the variance also suggests that more improvements in effectiveness can be expected by studying the factors associated with reviewers and code units that drive inspection effectiveness.

Differences in code units strongly affect defect detection effectiveness. Therefore, it is important to study the attributes that influence the number of defects in the code unit. Of the code unit factors we studied, code size was the most important in all the models. This is consistent with the accepted practice of normalizing the defects found by the size of the code. The next most important factor is functionality. This may indicate that code functionalities have different levels of implementation difficulty, i.e., some functionalities are more complex than others. Because functionality is confounded with authors, it may also be explained by differences in authors. And because it is also confounded with development phase, another possible explanation is that code functionalities implemented later in the project may have less defects due to improved understanding of requirements and familiarity with implementation environment.

The choice of people to use as reviewers strongly affects the defect detection effectiveness of the inspection. The presence of certain reviewers (in particular, Reviewer F) is a major factor in all the models. It suggests that improvements in effectiveness may be expected by selecting the right reviewers or by studying the characteristics and background of the best reviewers and the implicit techniques by which they study code and detect defects.

5 A Model of Inspection Interval

Using the same set of factors, we also built a statistical model for the interval data. We measured the interval from submission of the code unit for inspection up to the holding of the collection meeting. Unlike defect detection, we do not see any further decomposition of the inspection process that drives the interval. The author schedules the collection meeting with the reviewers and the reviewers spend some time before the meeting to do their preparation. So instead of splitting the inspection process into preparation and collection, we just modeled the interval from submission to meeting.

A linear model was constructed from the factors described in the previous section.[16] We started by modeling interval with the same initial set of factors as in the previous section. Using stepwise model selection heuristic we arrived at the following model.

$$Interval \sim Phase + Func + R_E + R_H + R_K$$

Even though we ended up with a small set of factors, the model was hard to interpret. It did not make sense for Functionality to be an important factor influencing the length of the inspection interval. In addition Functionality and Phase were confounded so they may be explaining part

[16] The linear model was used here rather than the generalized linear model because the distribution of the interval data is approximately normal.

	Factor	Degrees of Freedom	Sum of Squares	F Value	Pr(F)	Effect
Treatment factors	Team Size	2	206.6	0.85	0.4308	
	Sessions	1	161.6	1.28	0.2619	
	Repair	1	532.0	4.39	0.0395	+
Input factors	Author	5	2195.0	3.62	0.0054	
	R_I	1	242.1	2.00	0.1618	+
	Residuals	77	9340.86			

Table 3: **Factors Affecting Interval.** The sum of squares measure the deviation contributed by each factor to the mean of the interval data. The probabilities indicate the significance of the contribution. The last column for each scalar factor in the model indicates whether the factor was a positive or negative contributor to the interval. (Author had 5 degrees of freedom and different authors had different effects.)

of the same variance. Our belief was that they were masking the effect of the other confounded factor, Author. It makes more sense for Author to be in the model since he is the central person coordinating the inspection. So we re-ran the stepwise model selection heuristic, instructing it to always retain the Author factor. The result was:

$$Interval \sim Author + R_I + Repair$$

In this model, *Interval* is the number of days from availability of code unit for inspection up to the last collection meeting.

The analysis of variance for this model is in Table 3. For comparison, all the treatment factors were added to the model. The model explains $\sim 25\%$ of the variance using just 7 degrees of freedom. The low explanatory power of the model indicates the limited extent to which structure and inputs affect interval and suggests that other factors (that were not observed in this study) are more important in determining the interval. The presence of Repair confirms our earlier experimental result stating that adding repair in between inspections increases the interval.

5.1 Model Checking

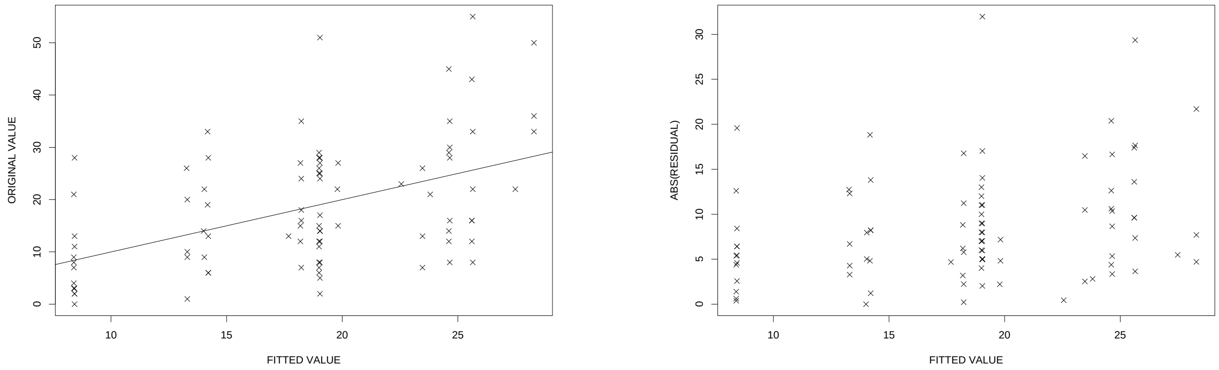

Figure 20: **Examining the fit of the model.** The left plot compares values estimated by the model with the original values (a perfect fit would imply that everything is on the line y = x). There is some correlation between the two (cor = 0.48). The right plot shows the relation of the fitted values to the residuals. The residuals appear to be independent of the fitted values.

Figure 20 gives diagnostic plots of the model's goodness-of-fit. The left plot shows the values estimated by the model compared to the original values. Because the model only explains 25% of the variance, it has limited predictive capabilities. The right plot shows the values estimated by the model compared to the residuals. The residuals appear to be independent of the fitted values.

5.2 Answering the Questions

We are now in a position to answer the questions raised in Section 3.1, with respect to inspection interval.

5.2.1 Will previous results change when process inputs are accounted for?

In this analysis, we build a linear model, composed of the significant process input factors plus the treatment factors and check if their contributions to the model are significant.

The effect of increasing team size is suggested by plotting the residuals of the interval model consisting only of input factors, grouping them according to Team Size (Figure 21(a)). We observe no significant difference in the distributions. When we included the Team Size factor into the model, we saw that its contribution was not significant ($p = 0.4$, see Table 3).

The effect of increasing sessions is suggested by plotting the residuals of the interval model

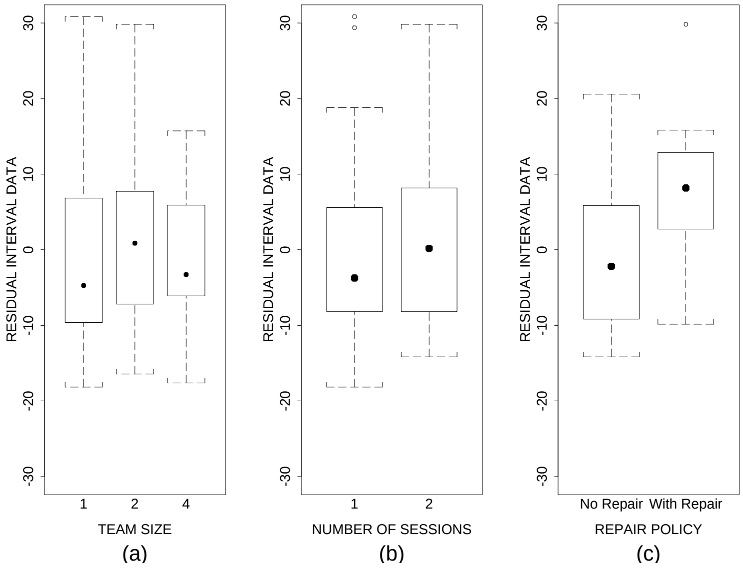

Figure 21: **Examining the Significance of the Experimental Treatment Factors.** These three panels depict the distribution of the residual data grouped according to Team Size, Sessions, and Repair.

consisting only of input factors, grouping them according to Session (Figure 21(b)). We observe no significant difference in the distributions. When we included the Session factor into the model, we saw that its contribution was not significant ($p = 0.3$).

The effect of adding repair is suggested by plotting the residuals of the interval model consisting only of input factors (for those inspections that had 2 sessions), grouping them according to Repair policy (Figure 21(c)). We have already seen that Repair has a significant contribution ($p = 0.04$) to the model in the previous section and this is supported by the plot.

5.2.2 Are differences due to process inputs larger than differences due to process structure?

Table 3 shows the factors affecting inspection interval and the amount of variance in the interval that they explain. We can see that some treatment factors and some process input factors contribute significantly to the interval. Among treatment factors Repair contributes significantly to the interval. This shows that while changes in process structure do not seem to affect defect detection, it does affect interval.

5.2.3 What factors affecting process inputs have the greatest influence?

The results of modeling interval show that process inputs explain only $\sim 25\%$ of the variance in inspection interval even after accounting for process structure factors. Clearly, other factors, apart from the process structure and inputs affect the inspection interval. Some of these factors may stem from interactions between multiple inspections, developer and reviewer calendars, and project schedule and may reveal a whole new class of external variation which we will call the process environment. These are beyond the scope of the data we observed for this study but they deserve further investigation.

6 Conclusions

6.1 Intentions and Cautions

Our intention has been to empirically determine the influence upon defect detection effectiveness and inspection interval resulting from changes in the structure of the software inspection process (team size, number of sessions, and repair between multiple sessions). We have extended the analysis to study as well the influence of process inputs.

All our results were obtained from one project, in one application domain, using one language and environment, within one software organization. Therefore we cannot claim that our conclusions have general applicability until our work has been replicated. We encourage anyone interested to do so, and to facilitate their efforts we have described the experimental conditions as carefully and thoroughly as possible and have provided the instrumentation online. (See `http://www.cs.umd.edu/users/harvey/variance.html`.)

6.2 The Ratio of Signal to Noise in the Experimental Data

Our proposed models of the inspection process proved useful in explaining the variance in the data gathered from our previous experiment. From them we could show that the variance was caused mainly by factors other than the treatment variables. When the effects of these other factors were removed, the result was a data set with significantly reduced variance across all of the treatments, which improved the resolution of our experiment. After accounting for the variance (noise) caused by the process inputs, we showed that the results of our previous experiment do not change (we see the same signal).

This has several implications for the design and analysis of industrial experiments. Past studies have cautioned that wide variation in the abilities of individual developers may mask effects due to experimental treatments[9]. However, even with our relatively crude models, we managed to devise a suitable means of accounting for individual variation when analyzing the experimental results. But ultimately, we will get better results only if we can identify and control for factors affecting reviewer and author performance.

Note also that the overall drop in defect data over time (see Figure 7) underscores the fact that researchers doing long term studies must be aware that some characteristics of the processes they are examining may change during the study.

6.3 The Need to Question Fundamental Assumptions

When process inputs are accounted for, the results of the experiment show that differences in process structure had little effect on defect detection. This reinforces the results of our previous experiment. That work showed that single session inspection by a small team is the most efficient structure for the software inspection process (fewest personnel and shortest interval, with no loss of effectiveness—see summary in Section 2.4 above).

Given the amount of research that has advocated changes to the structure of the inspection process, these results raise an interesting research challenge, "Can we demonstrate that process structure actually increases defect detection?" Do these effects depend on the environment, the type of documents being inspected, the background and experience for the reviewers. How do these effects interact with the process techniques being used? Clearly, more experimentation is needed to address these questions.

Also, we should not neglect research to find better techniques with which to review (e.g., systematic reading techniques[1] for the preparation step, meetingless techniques[10, 20, 13] for the collection step, etc.).

7 Future Work

7.1 Framework For Further Study

Our study revealed a number of influences affecting variation in the data, some internal and some external to the inspection process.

Internal sources included factors from the process structure (the manner in which the steps are organized into a process, e.g., team sizes, number of sessions, etc.), and from the process techniques (the manner in which each step is carried out, the amount of effort expended, and the methods used, e.g., reading techniques, computer support, etc.).

External sources included factors from the process inputs (differences in reviewers' abilities and in code unit quality) and from the process environment (changes in schedules, priorities, workload, etc.).

7.2 Premise for Improving Inspection Effectiveness

We believe that inspection techniques have not been studied adequately. Moreover, our results suggest that better techniques may have a strong effect on inspection performance. Thus, we will continue to investigate and improve them [24].

7.3 Need for Continued Study of Inspection Interval

We have not yet adequately studied the factors affecting interval data. Some of the factors are found in process structure (specifically repairing in between sessions) and process inputs, but much of its variance is still unaccounted for. To address this, we must examine the process environment, including workloads, deadlines, and priorities.

Acknowledgments

We would like to recognize Stephen Eick and Graham Wills for their contributions to the statistical analysis. Art Caso's editing is greatly appreciated.

References

[1] Victor R. Basili and Harlan D. Mills. Understanding and documenting programs. *IEEE Trans. on Software Engineering*, SE-8(3):270–283, May 1982.

[2] Victor R. Basili and David M Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. on Software Engineering*, SE-10(6):728–738, Nov. 1984.

[3] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language*. Wadsworth and Brooks/Cole, 1988.

[4] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Trans. on Software Engineering*, 15(10):1294–1304, Oct. 1989.

[5] George E. Box, William G. Hunter, and J. Stuart Hunter. *Statistics for Experimenters*. John Wiley and Sons, Inc., 1978.

[6] John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tuckey. *Graphical Methods For Data Analysis*. Chapman & Hall, 1983.

[7] John M. Chambers and Trevor J. Hastie, editors. *Statistical Models in S*. Wadsworth & Brooks, 1992.

[8] Chris Chatfield. Model uncertainty, data mining and statistical inference. *Journal of the Royal Statistical Society, Series A*, 158(3), 1995.

[9] Bill Curtis. Substantiating programmer variability. *Proceedings of the IEEE*, 69(7):846, July 1981.

[10] Alan R. Dennis and Joseph S. Valacich. Computer brainstorms: More heads are better than one. *Journal of Applied Psychology*, 78(4):531–537, April 1993.

[11] Stephen G. Eick, Clive R. Loader, M. David Long, Lawrence G. Votta, and Scott Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, Melbourne, Australia, May 1992.

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

[13] Philip M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering*, pages 113–122, Sorrento, Italy, May 1994.

[14] David A. Kenny. *Correlation and Causality.* John Wiley & Sons., New York, 1979.

[15] John C. Knight and E. Ann Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, Nov. 1993.

[16] Brett Kyle. *Successful Industrial Experimentation*, chapter 5. VCH Publishers, Inc., 1995.

[17] David A. Ladd and J. Christopher Ramming. Software research and switch software. In *International Conference on Communications Technology*, Beijing, China, 1992.

[18] David A. Ladd and J. Christopher Ramming. Two application languages in software production. In *USENIX Symposium on Very-High-Level Languages*, Oct. 1994.

[19] P. McCullagh and J. A. Nelder. *Generalized Linear Models.* Chapman and Hall, 2nd edition, 1989.

[20] J.F. Nunamaker, Alan R. Dennis, Joseph S. Valacich, Douglas R. Vogel, and Joey F. George. Electronic meeting systems to support group work. *Communications of the ACM*, 34(7):40–61, July 1991.

[21] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 215–222, Aug. 1985.

[22] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Experimental software engineering: A report on the state of the art. In *Proceedings of the 17th International Conference on Software Engineering*, pages 277–279, Seattle, WA, April 1995. Invited talk and short paper appear in the proceedings.

[23] Dewayne E. Perry, Nancy A. Staudenmayer, and Lawrence G. Votta. Understanding and improving time usage in software development. In Alexander Wolf and Alfonso Fuggetta, editors, *Software Process*, volume 5 of *Trends in Software: Software Process*. John Wiley & Sons., 1995.

[24] Adam A. Porter, Lawrence G. Votta, and Victor R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Trans. on Software Engineering*, 21(6):563–575, June 1995.

[25] Adam A. Porter, Lawrence G. Votta, Harvey P. Siy, and Carol A. Toman. An experiment to assess the cost-benefits of code inspections in large scale software development. In *The Third Symposium on the Foundations of Software Engineering*, Washington, D.C., Oct. 1995.

[26] G. Michael Schneider, Johnny Martin, and Wei-Tek Tsai. An experimental study of fault detection in user requirements documents. *ACM Trans. on Software Engineering and Methodology*, 1(2):188–204, Apr. 1992.

[27] Scott Vander Wiel and Lawrence G. Votta. Assessing software designs using capture-recapture methods. *IEEE Trans. on Software Engineering*, 19(11):1045–1054, Nov. 1993.

A Project Overview

In this appendix, we present the compiler project in some detail so that the reader may be aware that the code units being inspected in the experiment are not uniform and may vary in number of injected defects in a systematic manner, as when certain functionalities are tied to certain developers. For an experiment to gain credibility, it must be replicated in many different settings, which may or may not give consistent results. In either case, a description of the original project may be used to strengthen corroborating results or give non-corroborating results some alternative explanation.

A.1 Project Background

The 5ESS is Lucent Technologies' flagship local/toll switching system, containing an estimated 10 million lines of code in product and support tools. At the heart of the 5ESS software is a distributed relational database with information about hardware connections, software configuration, and customers. For the switch to function properly, this data must conform to certain integrity constraints. Some of these are logical constraints; for example, "call waiting and call forwarding/busy should never be active on the same line." Other constraints exist to document data design choices (redundancy, functional dependencies, distribution rules) that support efficient 5ESS operation and call processing. Enforcing these constraints are done through data audits, which check for all data violations on a snapshot of the database, and transaction guards, which ensure that incremental changes to the database leave it in a consistent state.

PRL5[17], a declarative language based on first-order predicate logic, was created to specify these integrity constraints. PRL5 specifications were to be translated automatically into data audits and transaction guards in C, with is then compiled on multiple platforms. Due to the constantly changing integrity constraints to be provided to different communication service providers worldwide, compilation speed is crucial. The generated C code also had to be optimized to make as few disk accesses as possible. (For more details on the history of PRL5, see Ladd and Ramming[18]).

The basic compilation scenario for P5CC is shown in Figure 22. The assignment of the development team is to implement P5CC as well as the P5CC runtime library.

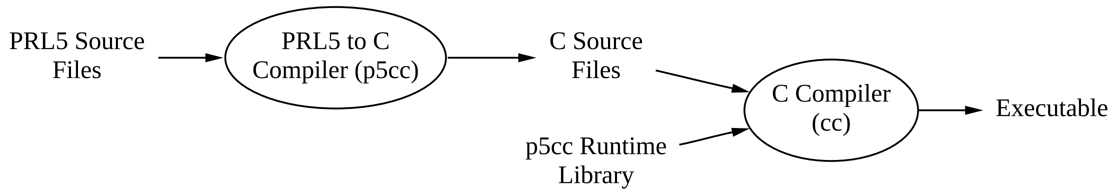

Figure 22: **Basic Compile Scenario for P5CC.** The PRL5 source files are translated into C using P5CC. This output, along with the P5CC runtime library, is then passed to a regular C compiler to produce an executable file.

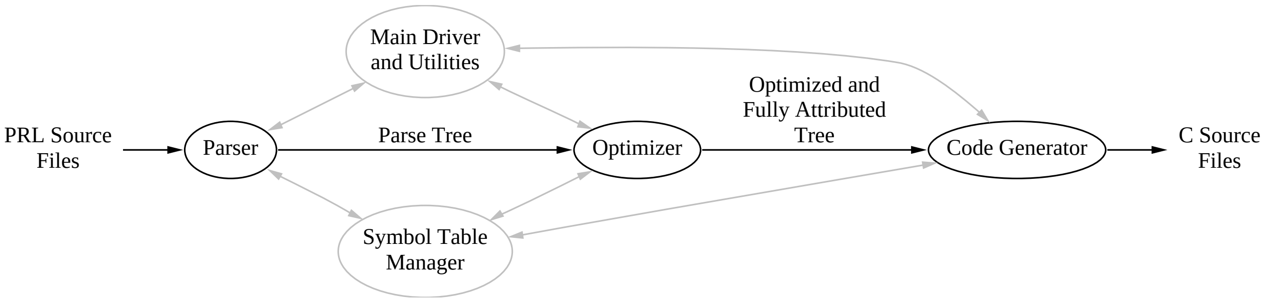

Figure 23: **Major Compiler Components.** The PRL5 source files is passed through the main translation pipeline from Parser, to Optimizer, to Code Generator. During the process, calls are also made to the symbol table manager and utilities.

A.2 P5CC Components

Figure 23 is a diagram depicting the components of the compiler. This is a multi-pass compiler creating intermediate trees between the parser, optimizer, and code generator. For our discussion, the parser refers to the code which checks both the syntax and semantics of the input program. It also includes code to implement the preprocessor and lexical analyzer. The optimizer refers to code which performs query optimization and also to all code that performs transformations on the parse tree, "massaging" it into a form suitable for processing by the code generator. The code generator refers to code that converts the transformed tree into opcodes, and eventually to C language statements. It also generates source file-dependent code needed for runtime support, such as memory management. The symbol table manager refers to all code which interface with the symbol table, handling access to all of the source program's data definitions. The main driver and utilities contains the main program which calls each routine, as well as miscellaneous utilities used by the other major functions.

B Measuring Effectiveness

In any empirical study, it is important to precisely define the variables we intend to measure. To measure the inspection's defect detection effectiveness, there are several choices, each with its own advantages and disadvantages. The artifact has a certain total defect density, ρ_{TOTAL} (defects divided by size), before the inspection. A certain portion of it, $\rho_{OBSERVED}$ is detected by the inspection and the rest, $\rho_{REMAINING}$ remains in the artifact after the inspection. We can write $\rho_{TOTAL} = \rho_{OBSERVED} + \rho_{REMAINING}$. When assessing inspection effectiveness, some natural metrics include the remaining defect density in the artifact, $\rho_{REMAINING}$ and the percentage of defects removed $\rho_{OBSERVED}/\rho_{TOTAL}$. The first gives an excellent view of the suitability of the artifact for use in subsequent phases of the software life cycle. The second gives an excellent view of the effectiveness of the process.

Unfortunately, neither quantity can be measured directly since we never know how many actual defects exist in the original artifact. There is no easy solution to this problem. Attempts at capture-recapture sampling techniques[11, 27] have been disappointing. Longitudinal studies which track the artifact and the defects found throughout its life cycle take a long time to complete and therefore are not capable of providing immediate feedback. In addition, it may be extremely difficult to determine whether additional defects found were due to mistakes in implementing the original requirements or in customer-requested enhancements.

An alternative metric is the observed defect density, $\rho_{OBSERVED}$. It has the advantage of being available as soon as the inspection has been completed. However, it is just a surrogate measure and does not give an accurate picture of the inspection's effectiveness. For example, if the number of observed defects is low, there is no way to tell if it is because the inspection was poorly executed, or if the artifact did not have many defects to start with.

Another issue is whether to use observed defect density or just the actual number of observed defects. The choice depends on the analysis technique we will be using later on. The defect density is approximately normally distributed and we can use simple and well-known linear models[5] to analyze the data. On the other hand, the actual number of defects is a more natural response variable because we can think of the inspection process as a process for counting the number of

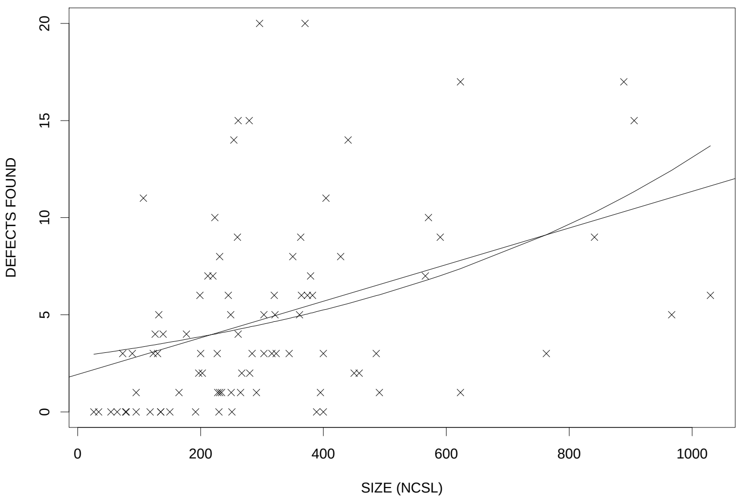

Figure 24: **Size vs. Number of Defects.** This is a scatter plot showing the relation between the size of the code and the number of defects found. The straight line is fitted using linear modeling while the curved line is fitted using generalized linear modeling.

defects. In this case, we can apply the methods of generalized linear modeling[19], in particular, the Poisson family of generalized linear models. Figure 24 shows the number of defects versus size. The dashed straight line and the curved solid line show the fitted data when we try to explain the variance with just the size variable, using linear modeling and generalized linear modeling, respectively. We can see that they give approximately the same fit so we can use either one. We decided to use the generalized linear model because it is more natural for our problem (fitted values will always be nonnegative counts). Hence we used actual number of defects as our measure of effectiveness.

C Statistical Modeling in S

C.1 Statistical Modeling

A statistical model takes the general form, $y = \mu(X_1) + \epsilon(X_2)$, where y is the vector of observed data, μ is a function taking as input a set X_1 of factors with associated coefficients, $x_{11}, \ldots, x_{1n}$ and giving as output $\hat{y}$, the expected value of y. The residuals ϵ represent a process whose effects are ignored or whose presence is unknown to us. Model formulation deals mainly with describing μ, specifying factors and the interaction between them. Model fitting deals with moving factors between X_1 and X_2 and adjusting the coefficients to give the best fit to y. A model may be considered adequate when the residuals $y - \hat{y}$ are independently distributed with zero mean and constant variance.

S is a programming environment for data analysis[3, 7]. In this appendix, we will outline our approach in using S to build and analyze statistical models for defect and interval data.

C.2 Model Formulation

C.2.1 Identifying Possible Variables

The possible factors to be incorporated into the model are usually determined from prior knowledge of the process being modeled. The initial model is normally specified with the full set of available factors.

Note that the effect caused by one factor may depend on the level of another factor. Each set of possibly related factors can be represented as an additional interaction term in the model[7, p. 22]. Since we had a limited number of observations and had difficulty interpreting interaction terms we avoided fitting interactions between factors. (One way to visualize these interaction effects is to plot the residuals against the appropriate products.)

C.2.2 Selecting the Model Function Form

With our defect data, the linear model[5] for defect density and the generalized linear model[19] for defect counts appear to perform equally well (see Appendix B). Defect counts are naturally modeled as a counting process. In our data the counts were too small to use Gaussian approximation. Hence, we used the generalized linear model (as opposed to linear model where the distribution of the response variable is assumed to be Gaussian).

For the interval data, the linear model was used because the distribution of intervals approximated a normal distribution.

S offers two functions for specifying models, `lm()` for linear models and `glm()` for generalized linear models. Both take as basic parameters a model specification and the data for the model. In addition, in `glm()`, we can specify a distribution family (Poisson, Gaussian, etc.)

C.3 Model Fitting

We did model fitting by iteratively adding or dropping factors and adjusting the coefficients to give the best fit for the given data (the particular implementation was S function `step`)[7, pp. 233-238]. In each iteration, a new factor is added to the model if it significantly reduces the residual variance. Conversely, a factor may be dropped if its removal does not significantly increase the residual variance.

While it is desirable to add as many explanatory factors in the model, there is the danger of adding too many factors. This is known as overfitting[8]. The problem is that while the model might fit the particular dataset well, it may be inexplainable, may not make physical sense, or have no predictive power when used on a different dataset.

We looked for a parsimonious model with the help of stepwise model selection. In stepwise model selection, we start with an existing model and iteratively add or drop one term, minimizing the number of parameters while maximizing the fit according to some specified criterion. In S, we used the function `step()`, increasing the scale parameter until the number of factors in the model are sufficiently reduced. There are other methods and criteria to select the best model but this is beyond the scope of this paper.

The model selection algorithm may not give the best model in terms of explaining the physical meaning of the factors it manipulates. At the end, we must use our prior knowledge of the process in order to fine-tune the model to one that is interpretable.

C.3.1 Calculating the Significance

To calculate the significance of a factor's contribution into the model, we used the `summary.aov()` function to perform analysis of variance, passing the model specification into it, with the factor of interest at the end of the formula. For example, if we have a model $y \sim a + b + c$, we perform the analysis of variance on $y \sim b + c + a$, $y \sim a + c + b$, and $y \sim a + b + c$ to calculate the significance of the contributions of a, b, and c to the model. Essentially, this is how `step()` determines which factor to retain and which to drop.

C.4 Model Checking

Once a model has been specified and fitted, it is checked to see if it is an adequate model. The model is adequate when it sufficiently explains the variance, i.e., adding the additional factors does not substantially reduce the residual variance. An informal check is to look for patterns in the set of residuals. The presence of patterns in the residuals is taken as an indication of presence of a better model.

D The Inspection Process

The following process description is a summary of the coding and inspection process followed by the inspectors. We took the organization's written process methodology and modified it to accommodate our experimental treatments.

1. Modification Requests (MR's) are issued whenever additions or enhancements to code are needed.

2. A developer accepts one or more MR's and develops the necessary code.

3. The author then makes a code unit available for inspection. A code unit may implement one or more MR's.

4. A particular inspection treatment and a review team is randomly assigned to the inspection.

5. The author contacts the review team and schedules the inspection meeting. (If the treatment calls for 2 sessions with no repair in between, the author contacts 2 sets of reviewers and sets up two separate meetings.)

6. Prior to the meeting, the reviewers analyze the code unit looking for defects.

7. The author and reviewers conduct the collection meeting. One of the reviewers is assigned to be the moderator. He makes sure the meeting does not get bogged down on any single point of discussion. Another reviewer is assigned as the reader. He guides the inspection team during the meeting by reading through the code and orally summarizing the logic of each coding section (paraphrasing). (The amount of detail is left to the discretion of the reader.)

8. After the meeting the author collects the consolidated list of issues. Issues are the potential defects discovered during the inspection.

9. The author determines which issues must be repaired, and does so.

10. The author brings the reworked code to the inspection moderator who ensures that all issues have been addressed and signs off the inspection.

11. If the treatment calls for 2 sessions with repair in between, then the author repeats all inspection steps one more time.