b17760793

QA
76.76
.D47
L53
2005

ANALYSIS ON RELATIONSHIP BETWEEN CODE QUALITY AND CODE COVERAGE IN AN XP ENVIRONMENT: A CASE STUDY ON THE SWURV PROJECT

by

Hua Li

Master of Computer Science

Beijing Institute of Technology, Beijing, China, 1997

A thesis
presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in the program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2005

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signature

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature

Hua Li, "Analysis on relationship between code quality and code coverage in an XP environment: A case study on the Swurv project," degree of Master of Applied Science, Electrical and Computer Engineering, Ryerson University, 2005

Abstract

The thesis uses hypothesis testing and correlation analysis methods to explore the relationship between structural code coverage and the quality of software developed in an eXtreme Programming (XP) environment, via a case study of a commercial software product. We find that improving code coverage is helpful to detect residual defects, but it is not enough, and we also need other testing, like acceptance testing, in the process of XP software development to provide good quality software products.

In addition, in order to investigate why the strength of association between code coverage and residual defect density is not as strong as that presented in prior work, a detailed defect root cause analysis is performed, showing that over 96% of bugs cannot be detected by improving code coverage. Based on the defect categories and distribution of defect root cause, six improvement actions are proposed for future XP projects.

Acknowledgements

I would like to gratefully acknowledge the enthusiastic supervision of Dr. Denise Woit during this work. She not only provides me general guidance and numerous stimulating discussions, but also helps experimental setup, and the opportunity to work and study at Swurv company; I wish to acknowledge assistance of some technical problems from members of the Swurv company, and in particular, I would like to acknowledge the help of Joey Coleman for his restoring old versions of Swurv project.

Finally, I am forever indebted to my husband, and parents for their understanding, endless patience and encouragement when it was most required.

Table of Contents

List of Tables

List of Figures

List of Appendix

Chapter 1
Introduction

This introduction covers the following topics to provide an overview of background material necessary to understand the work in this thesis: Extreme Programming, automated testing for web applications, and the Swurv project (the case study product). Following this, the thesis motivation and organization are presented.

1.1 Extreme Programming

Extreme Programming (XP), originated by Kent Beck, has been proven to be an effective and successful software development process in projects for many companies [34]. Organizations increasingly face a rapidly changing business and technological environment. In such a dynamic environment, traditional software development approaches, which assume that all requirements can be anticipated at the beginning of projects and will remain stable, are unlikely to be successful [32]. The goal of Extreme Programming is to allow an organization to be able to deliver and change quickly. Change is smoothly facilitated in XP, by empowering developers to confidently respond to changing customer requirements, even late in the life cycle.

1.1.1 Core Practices of XP

Extreme Programming is described in terms of 12 core practices [37]:

- **The Planning Game**: Business and development cooperate to produce the maximum business value as rapidly as possible.
- **Small Releases:** Start with the smallest useful feature set. Release early and often, adding a few features each time.
- **System Metaphor:** Each project has an organizing metaphor, which provides an easy to remember naming convention.
- **Simple Design:** Always use the simplest possible design that gets the job done.
- **Test Driven Development:** Before programmers add a feature, they write a test for it. In XP, there are two kinds of tests:

A. Unit tests are automated tests written by the developers to test functionality as they write it. Each unit test typically tests only a single class, or a small cluster of classes.

B. Acceptance tests (also known as functional tests) are specified by the customer to ensure that the overall system is functioning as specified. Ideally acceptance tests should be automated.

- **Refactoring:** Eliminate any duplicate code generated in a coding session.
- **Pair Programming:** All production code is written by two programmers sitting at one machine.
- **Collective Code Ownership:** No single person "owns" a module. Any developer is able to work on any part of the code base at any time.
- **Continuous Integration:** All changes are integrated into the code base on a minimum of a daily basis. The tests have to run 100% both before and after integration.
- **40-Hour Work Week:** Overtime is treated as a sign that something is wrong with the process.
- **On-site Customer:** The development team has continuous access to a real live customer.
- **Coding Standards:** Everyone codes to the same standards.

1.1.2 The Good, The Bad and Modified Practices for XP

Recently, there have been some discussions about which practices of XP are useful and which need to be adapted (the lists are given below). From some real projects' experience, some XP practices are not considered readily applicable to every project [30,32,33,35]. However, as XP has provided many effective practices to develop software at Internet speed, to accommodate changes in requirements, XP is becoming increasingly important and deserves to be tailored to suit the nature of different systems and development environments. This is especially true for complex large-scale enterprise systems [32]. I summarized XP's good and bad aspects, and modified practices, according to papers based on authors' real world project experience.

1. The Good Aspects:

- Testing [31,34,35]
- Pair Programming [34]
- Small Release [34]
- Common Code Ownership [34]
- Continuous Integration [31,34,35]
- Metaphor [31,34]
- On-Site Customer [34]
- Test-driven development [35]
- Stand-up meetings [35]
- Refactoring [35]
- Simplest thing possible [31,35]
- Extremely human-focused [31]

Rumpe's study [34], Quantitative Survey on Extreme Programming Projects, was conducted among people who had experience in applying XP, and presented the following encouraging data [34]:

- Almost all of the projects were rated successful.
- 100% of the asked developers would reuse XP in the next project, when appropriate.
- The most useful XP elements were common code ownership, testing and continuous integration, metaphor and on-site customer.
- The most important success factors were: testing, pair programming and the focus of XP on the right goals.

Rasmussen [35] summarizes several good aspects of their customer's project. He stresses that Test-driven development is very useful. He points out, "*Tests are an important side effect, but more important is the resulting design the technique helps produce.*"[35]

2. The Bad Aspects:

- Limited documentation (User Story, code and test cases only) [33]

  Nawrocki proposes that the most important issues are: maintenance problems resulting from very limited documentation (XP relies on code and test cases only) [33]. Also, XP relies on oral communication, but oral communication is susceptible to lapses of memory. Again it would be useful to have some written documents.

- Insufficient automated functional tests [30]

  Elssamadisy mentions, "*All the unit tests are passing and the system is still broken. An insufficient suite of automated tests may result in late delivery.*"[30]

- Developers' lack of a holistic understanding of the place each story has in the total application [30].

  Elssamadisy noticed, "W*hen the story cards are written and analyzed, the responsible party for a card feels that he or she cannot be sure all functionality has been accounted for in the functional tests developed for that card. [30].*" With the number of implemented story cards growing, customers began to miss this or that dependency that should have been addressed in the story card.

- Design at development time [31,32]

  Glass points out that the main disadvantage of XP is ongoing design at development time. From his experience, " ' *analysis paralysis' might be okay for tiny projects (Beck says XP is for small to medium-sized projects), but it's a disaster in the making for larger ones.*"

- Pair Programming [31,32]

  Some people argued that pair programming is not good since most programmers do not like to hold ongoing conversations with a pair mate while working in creative mode.

3. The Modified Practices:

- Designing upfront [32]

  Development of large and complex system requires a carefully designed architecture. In this way, we can avoid the bad effects of "Design at development time".

- Surrogate customer engagement [32]

  It is difficult to get a real customer to work on site, so an on-site surrogate customer is integrated into the project.

- Flexible pair programming [32]

  Pair programming is used in a flexible way. Only analysis, design, test case development, and unit testing are done in pairs. Developers do try to code in pairs but may return to solo coding. This is inherently tied to the personality of the developers.

- Provide the development team with some synoptic "picture" of the whole that promotes a holistic understanding of the place each story has in the total application [30]

- Make testers a part of the team [35]

  Better results will be achieved when testers are made a core part of the team. They can aid customers in defining and automating acceptance tests as well as play the role of analyst and flush out hidden requirements and assumptions [35].

1.1.3 Acceptance Testing in Extreme Programming

Ron Jeffries [38] indicates that the failure to have customer-owned acceptance tests is one of the most common mistakes in XP. Acceptance tests can enhance the communication between customers and programmers. By defining acceptance tests at the beginning of every iteration, the customer communicates to the programmers what she wants, by telling them how she will confirm that they've done what is needed; by implementing acceptance tests, programmers show customers that the story has been implemented correctly. Furthermore, successful acceptance tests are customer-owned and automatic. However, "automatic" requirement is difficult for some cases, such as GUI applications [36,39,40].

There are some researchers who explore how to automate Acceptance tests. Lisa Crispin et al [36] reported that the "automatic" criterion has given them trouble in some cases. For example, for web applications, they have not found a cost-effective way to automate JavaScript testing. They test JavaScript manually and carefully control their JavaScript libraries to minimize changes and thus the required retesting. They also propose principles for acceptance tests.

- Test must be modular and self-verifying to keep up with the pace of development.

- Verify the minimum criteria for success.

Because the unit tests are comprehensive, we don't need to duplicate them in Acceptance tests.

- Perform each function in one and only one place.

  This is to minimize maintenance time.

- Define modules that can be reused, even for unrelated projects.

- Do the simplest thing that works.

  This XP value applies as much to testing as to coding.

Finsterwalder [39] suggests that acceptance tests need to be repeatable and should be run at least once a day. This way, any defects that have been introduced into the application can be detected and corrected as soon as possible. In XP, acceptance tests should be automated. But it is not always worth automating every test. Especially for GUIs that change often, it does not pay off to invest in GUI test automation. Since their application is designed so that no business logic is mixed with the GUI handling code, *"the verification that the thin GUI layer is working well can often be done manually."*

Anderson [40] introduced a successful project using the strategy "Usage-First Design". "*We simulated Test-First Design by insisting that new code was always written 'usage-first' acceptance tests.*" It is difficult to automate some products (such as interactive GUIs). They solved this by creating a scripting interface that slots in just beneath the GUI layer itself. But this method depends on the architecture of the software application.

1.2 Automated Testing for Web-based application

As we discussed above, XP requires automated unit testing and acceptance testing. Moreover, Swurv, the project studied in this thesis, is a web-based application. So, in this section, we did a survey on automated testing for web-based applications. Automated tests are crucial for Web-based projects. They enable continuous modifications to an existing code base without the fear of damaging existing functionality. They are executed at will and don't carry the costs and inconsistencies associated with manual tests [43].

1.2.1 Black box and White box

There are two fundamental approaches for automated web testing:

- White Box/Structural tests, where an application is tested from the inside using its internal application programmatic interface (API).
- Black Box/Functional tests, where an application is tested using its outward-facing interface (usually a graphical user interface).

Sometimes white box tests are sufficient. In general, white box unit tests are sufficient to test programmatic interfaces (APIs). For example, a unit structural test is sufficient to test a square root function [43].

But for web applications, unit tests are not enough since most unit testing tools do not support testing long event functions. What is a long event function? For example, in WordPad, the event "Find Next" (obtained by clicking on the Edit menu) can only be executed after at least six events have been performed. Web applications, however, usually include many longer event functions. For example, in a flight booking web application, if a user wants to book a flight ticket, she/he has to fill in a query form, submit it and then get a list of choices, choose one option and fill some personal information to book a flight. In order to test the above 'booking' function, we have to test the continuous events. In general, unit tests only test an event/function in a test. Memon [42] shows that testing these longer event sequences may help detect a larger number of faults than short event sequences.

Black Box tests are useful in testing long event functions and evaluating the overall health of the application and its ability to provide value to users. Most Black Box tests emulate the sequence of interactions between a user and an application in its exact order. Functional testing plays an important role in XP methodology, which stresses functional testing so that developers can get feedback about the overall state of their system [41].

The two testing strategies are complementary: Structural tests may reveal intricate bugs that functional testing would miss, but functional tests are required to validate the application's overall performance.

1.2.2 Web Testing Tools

In today's market, a number of Automated Test Tools have been developed for web applications.

1. Functional Testing Tools

Giora Katz-Lichtenstein [44] introduced historical and recent methods to do automated black box testing for web applications. She points out that, "historically, writing Black Box tests was difficult. Applications used different GUI technologies and communication protocols; thus, the potential for tool reuse across applications was low. Commercial testing-tool packages relied on capturing mouse clicks and keystrokes, which proved brittle to change. The combination of wide usage of HTTP/HTML standards and an emerging number of open source projects makes Black Box testing an easier task." [44]

Nowadays, many companies and developers prefer to use open source Java packages like HttpUnit or tools based on HttpUnit [41,44]. "The open-source HttpUnit ... is free of licensing fees and very simple to use... Using HttpUnit as a foundation, you can build complex test suites at minimal cost." [8]

HttpUnit covers the useful spectrum of HTTP/HTML standards functionality, including forms, frames, JavaScript, SSL, cookies, and headers. HttpUnit can be used to test a web site written in any programming language. It can be used with a framework such as JUnit.

2. Structural Testing Tools

White-box or Structural tests validate each method independently. The most popular framework for Structural testing in Java is JUnit, but it lacks for server-side testing. Unlike a stand-alone application, servlets and EJBs in Web Applications depend on objects created and managed by the server; in other words, they necessarily rely on the container's context. For example, many servlet methods expect an HttpServletRequest and an HttpServletResponse object as parameters, making it impossible to test the methods without valid instances of these objects [45]. In these cases, there are two general approaches:

1) Mock objects (ServletUnit, etc)[45]
2) In-container testing (Cactus)[45]

Mock Objects are test-oriented replacements for concrete implementations. They are configured to simulate the server side object. In contrast to stubs, they also keep details of the expected interactions. (A stub object provides nothing more than the implementation of an interface.) Mock Objects let us test methods even before the domain objects are ready, i.e. before the implementation is ready or before a choice of implementation has been made. Thus, for example, it is possible to write servlet code before choosing a container. This is in accordance with XP that says: "do not commit to infrastructure choice before you have to" and "write unit tests first" [46].

There are some testing frames using mock objects: ServletUnit (the most famous and popular tool for unit testing servlet) [62], MockEJB [61], etc.

In-container testing is to run the tests on the server, using real objects provided by the Web server [45]. It tends to be more coarse-grained and it also ensures that developed code will run in the container. Until now, from my research, the unique package supporting in-Container testing is Cactus.

Cactus's founder, Vincent Massol, stresses that these two ways are complementary. [46] Mock objects let us test methods even before the domain objects are ready, and it is in accordance with XP's directive to "write unit test first". A cactus test gives us 'enough' confidence that our tests will run fine when deployed [47]. Furthermore, sometimes, we need to check in-container memory objects to verify applications.

1.3 Swurv Project Introduction

The Swurv company specializes in developing Internet technology products. The product in our study is Swurv Environment (SE). SE is a web-based delivery platform that creates a central meeting place for employees, clients and suppliers [48]. SE was developed by a team of 7-12 developers and the development lasted for about 20 months.

Table 1.1 gives statistic information about the Swurv project (SE) from Clover --- a code coverage tool for Java [21].

Table 1.1: Swurv Project Statistics

Project stats:	LOC:	215,921		Methods:	11,484
	NCLOC:	181,111		Classes:	1,343
	Files:	1,295		Packages:	20

In Table 1.1, "LOC" stands for lines of code and "NCLOC" stands for non-commented lines of code. From the above table, we can see 1,343 java classes, 20 java packages and 11,484 methods were coded for the Swurv project. The Swurv project is a large and real commercial project, which is suitable for our case study.

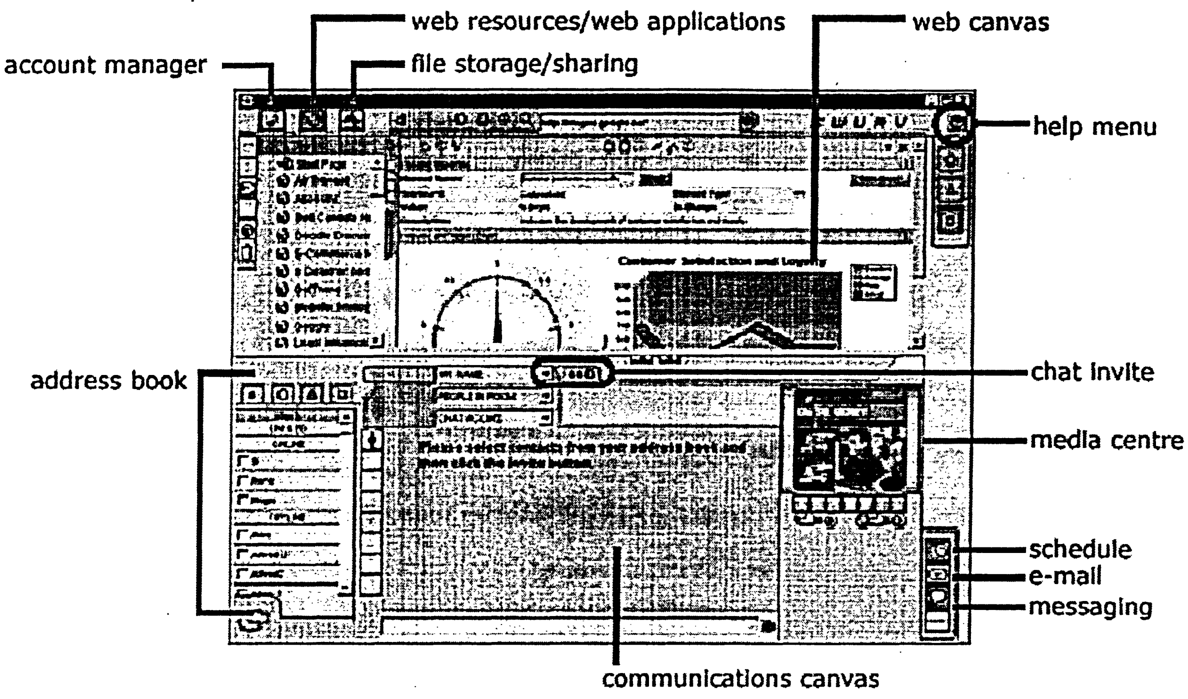

Figure 1.1: Swurv project [48]

1.3.1 Testing

In Swurv, developers apply JUnit and ServletHarness, an in-house Mock objects test tool for servlets, to write automated unit testing. These tests are executed every evening. When developers come back to work, the first thing they do is to check the testing results. In this way, they get quick feedback as to whether the newly added features work with other parts of the project.

As for acceptance testing, the customer manually tests the functions of the project daily. If a bug is detected, he writes a bug report using Bugzilla (a kind of bug tracking system [63]).

In terms of automated testing for web-based applications, Swurv project has applied the `mock objects strategy` to do automated structural testing. They lack in-container structural testing and automated functional testing.

1.3.2 XP Practices in Swurv

Swurv applies the Extreme Programming methodology to develop products, including SE. The XP practices Swurv follows include:

- **The Planning Game**
- **Small Releases**
- **Simple Design**
- **Unit Test Driven Development**
- **Refactoring**
- **Pair Programming**
- **Collective Code Ownership**
- **Continuous Integration**
- **40-Hour Work Week**
- **On-site Customer**
- **Coding Standards**

The practices of XP that Swurv does not follow:

- **System Metaphor**

  System Metaphor, a compact mental image of the system, is a communication mechanism [60]. It is not very indispensable and can be replaced by other methods, such as design discussion, or documents. Rumpe [34] mentions, "It is used by 40% XP projects" [34]. Ron Jeffries [60] indicates that, "When working without a clear metaphor (which probably half of all projects do), expect to need more diagrams, more design discussions, and perhaps more documents."

- **Acceptance Tests before Development**

  Before programmers add a feature, the customer seldom writes detailed acceptance tests. However, sometimes, the customer would do manual acceptance testing after programmers submit a new feature.

We will suggest an effective set of XP improvement actions on the basis of root causal defect analysis for the Swurv project in Chapter 5.

1.4 Motivation and Thesis Organization

In this section, I will introduce this thesis's motivation and organization. However, first, I will explain the following terms:

- Code coverage: it is expressed in terms of a ratio of the metric items executed or evaluated at least once to the total number of metric items (see section 2.1.1 for detailed information).
- Residual defects: they are referred to the bugs detected from a software production operational usage after XP development team releases the product to the customer.
- Quality: for the purposes of this thesis, product quality is considered to be inversely proportional to the number of residual defects.
- Defect root causal analysis: it is the process of analyzing a defect to determine its root causes.

1.4.1 Motivation

Usually, software developers tend to ignore unit testing or simply test several methods of the newly developed code/features. They argue that testing is the task of testers and they cannot write too much testing code since they have a tight schedule. However, Extreme programming (XP) stresses that unit testing done by developers is very important. Project managers want to check if developers have done adequate unit tests for modules. Since it is not convenient for project managers to get the data of how well developers have tested for unit functional tests, they tend to use code coverage tools, such as JProbe, Clover etc, to get the structural coverage data of modules. But, does high code coverage mean less defects? Hence, we want to show if code coverage has a strong relationship with remaining defects through our case study.

In XP, developers are encouraged to apply simple design to implement everything, and avoid putting effort to useless things. They begin with a small simple component and frequently refactor out duplicate code to keep a good simple structure. Thus, programs are easy to understand and maintain. Also, because of the simple clear structure, buggy code may jump out at developers. However, in real life, developers and project managers usually do not schedule enough time for refactoring. In this thesis, I want to explore if the number of revisions, including the number of refactorings, number of bug fixes and number of adding new functions, has any correlation to residual defects. Because small components are encouraged in XP, I also wish to study whether component size has any correlation to residual defects. If such relationships exist, an XP team will be more confident with the practices of simple design and refactoring.

In addition, currently, Extreme Programming (XP) methodology is applied to more and more real world projects. Swurv has also practiced XP for almost 3 years. However, are there any XP practices needing to be modified? Addressing this question, we want to do root cause analysis for Swurv defects, and then propose recommendations.

To summarize, the main goals of this thesis are:

G1: To study if there is any correlation between code coverage and residual defects.

G2: To study if there is any correlation between code coverage and number of component revisions, between residual defects and component size, and between residual defects and number of component revisions.

G3: To explore if there are any XP practices that could benefit from modification.

We hope to provide data to help XP project managers revise their development practices in future projects. Then, how to implement these goals?

1. For Goal Gl and G2, firstly, we need to collect and analyze component data, such as code coverage, number of revisions, component size, and residual defects, from the project. Then, we will use correlation analysis methods to determine if any relationship exists among the data.

2. For Goal G3, we need to perform defect root causal analysis (DRCA). First of all, we will collect bugs, and then categorize the underlying causes of each collected bugs. After calculating the distribution of defect root causes, we can find which phases inject most of the collected bugs. Then, we can propose improvement actions for XP practices to help prevent and eliminate bugs.

Thus, after I finish the above two steps, my main contributions are in the following aspects:

- To provide real correlation data between code coverage and residual defects in an XP environment. From our survey, no previous work has studied an XP environment.
- To provide real correlation data between code coverage and number of component revisions, between residual defects and component size, and between residual defects and number of component revisions. These data are from a large, real-world web-based

project, using actual XP software bugs. No previous work studies such correlation from real bugs in large, real-world web-based XP applications.

- To provide detailed steps of how to perform DRCA and DRCA's results based on a real-world case study to help XP project managers improve development practices.

1.4.2 Organization

This thesis is organized as follows:

Chapter 1: Firstly, Extreme Programming methodology is introduced, including a survey on the good, the bad, and the modified XP practices. Then, basic automated web testing methods are presented. The Swurv project under study is also introduced in this chapter.

Chapter 2: The background and related prior work necessary to understand the context and details of the techniques developed in this thesis are provided here.

Chapter 3: Experimental design and basic data are introduced. First of all, the process of data collection and filter are interpreted; then, the correlation analysis methods we use are introduced; last, we present some basic Swurv statistic data and defect distributions.

Chapter 4: Our hypotheses are presented and tested by correlation analysis methods, and results are presented here.

Chapter 5: Based on the defect causal distribution analysis, appropriate improvement actions for the Swurv project are proposed.

Chapter 6: Results of this thesis are summarized and discussed, and I provide proposals for future work.

Chapter 2
Background and Related Work

The research presented in this thesis focuses on two goals. One is to analyze correlation between code coverage and remaining defects, and the other is through defect causal analysis to explore if there are any XP practices applied by the Swurv project needing to be modified. This chapter introduces the relevant terms, and presents the background and prior related research in each of these areas. However, very little research has been done in applying defect causal analysis to explore XP practices. Since Extreme Programming has some common characters with traditional software development, some of the methods and results from the prior work on defect causal analysis for non-XP projects are surveyed.

2.1 Structural Test Coverage

Structural testing techniques use a program's structure to guide the development of test cases. Chen [18] has shown that white-box testing is generally more effective than methods such as random and functional testing [18]. One of the main tasks of this thesis is to study if the structural test coverage is useful. Is there any relationship between structural code coverage and remaining defects? Very little research is done to show us the relationship between code coverage and remaining defects. Most of them study code coverage and *detection* of defects. I have classified my survey into three categories:

1. Structural Test Coverage Criteria

2. Correlation between testing coverage and the detection of defects.

3. Using structural testing to guide functional testing

Hence, in the subsequent section, some of the terms, approaches and conclusions related to these studies are presented.

2.1.1 Introduction to Structural Test Coverage Criteria

Coverage criteria are a set of rules used to help determine whether a test suit has adequately tested a program and to guide the testing process [17]. Structural coverage techniques give a measure of how well the structural elements of the program are executed by a given test suite [18]. A **coverage** metric is expressed in terms of a ratio of the metric items executed or evaluated at least once to the total number of metric items. This is usually expressed as a percentage.

Coverage = items executed at least once / total number of items.

Theoretically, there are several standard structural coverage criteria: Statement coverage, Branch coverage and Path coverage. Beizer [20] presents their definitions. In real life, researchers/engineers also apply other criteria, such as conditional coverage, method coverage and total coverage, etc.

Path Testing [20]--- Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program. If we achieve this prescription, we are said to have achieved 100% **path coverage.**

Statement Testing [20]--- Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% **statement coverage.**

Branch Testing [20]--- Execute enough tests to ensure that every branch alternative has been exercised at least once under some test. If we do enough tests to achieve this prescription, then we have achieved 100% **branch coverage.**

Conditional Testing --- For each condition in a decision, we must get TRUE and FALSE at least once. A condition is a Boolean sub-expression, separated by logical-and and logical-or if they occur. Condition coverage measures the sub-expressions independently of each other. This measure is similar to branch testing, but has better sensitivity to the control flow. If we do enough tests to achieve this, then we have achieved 100% **conditional coverage.** However, full condition coverage does not guarantee full branch coverage.

Method Testing --- Each method in a class or module must have been executed at least once. If we do enough tests to achieve this, then we have achieved 100% **method coverage.**

Total Coverage --- The "total" coverage percentage of a class (or file, package, project) is provided as a quick guide to how well the class is covered - and to allow ranking of classes. The total percentage coverage (TPC) is calculated using the formula: [21]

TPC = (CT + CF + SC + MC)/(2*C + S + M)

where

CT - conditionals that evaluated to "true" at least once

CF - conditionals that evaluated to "false" at least once

SC - statements covered

MC - methods entered

C - total number of conditionals

S - total number of statements

M - total number of methods

In the real world of testing, to achieve 100% of path coverage is impractical since some loops might never terminate. Automated collection and analysis is considered essential if a metric is to be used in real software development. Through software testing coverage analysis tools, some test coverage metrics, which are most suitable for automated collection and analysis, can be automatically evaluated. For example, **statement** and **module/method** coverage are widely used in most coverage tools, such as Rational Pure Coverage[64], JProbe coverage[65], Clover coverage tools[21], etc. **Branch/Conditional** coverage metrics are also used in some coverage tools, such as Testworks[66] and Clover. In the meantime, a method of combining these metrics into a single index is recommended and implemented in some tools, such as Clover, to get an overall coverage evaluation.

2.1.2 Correlation Between Structural Test Coverage and Detection of Defects

Several researchers have explored the relationship between structural test coverage (or code coverage) and the detection of defects [3,6,12,13,55]. Williams [6] used mathematical analysis to validate the relationship between code coverage and the detection of defects. He indicated that his analysis result agrees well with the experimental results of previous work: A linear increase in code coverage results in an exponential increase in errors found [6].

Malaiya et al [13] present a model for test effectiveness that relates test coverage and defect coverage. The model is fitted to 4 data sets for programs with real defects. The data used (defect coverage and test coverage) are from integration and acceptance-test phases.

The model is described by the following equation: [13]

$C_0 = -A_i + B_i*C_i$, where

C_0 is defect coverage;

C_i is test coverage, including block, decision, c-use and p-use test coverage defined below;

A_i and B_i are the parameters for the linear approximation for C_i;

In the paper [13], Malaiya et al present definitions of coverage measures they use:

- Statement (or block) coverage: The fraction of the total number of statements (blocks) that have been executed by the test data. (Block refers to statement blocks of a program.)
- Branch (or decision) coverage: The fraction of the total number of branches that have been executed by the test data.
- C-use coverage: The fraction of the total number of c-use pairs that have been covered during testing. A c-use pair includes 2 points in the program, a point where the value of a variable is defined or modified, followed by a point where it is used for computation (without the variable being modified along the path).
- P-use coverage: The fraction of the total number of p-use pairs that have been covered during testing. A p-use pair includes 2 points in the program, a point where the value of a variable is defined or modified, followed by a point in a branching statement where it is used in a predicate (without modifications to the variable along the path).

Ye and Malaiya [12] have validated the above model [13,19]. They propose, " for the values of C_i greater than a knee-value C_i _knee, this model can be approximated by a linear function.... It has been shown that this model can be used to estimate the residual defect density. "

Piwowarski [3] surveyed testing experience of using coverage measurements during function testing at IBM. They showed an almost linear correlation between the number of errors removed and code coverage. They propose that a 10% improvement in code coverage would lead to approximately a 70% removal rate during function test.

Karcich [55] collected failure and module coverage data during the system testing phase of a commercial software-hardware system. They measured module and branch coverage. They found when the coverage increases, so do the errors detected. They also applied the correlation analysis technique. The correlation coefficient between error and module coverage was found to be 0.76 and that between error and branch coverage was 0.60. Based on the Table 3.7 (Descriptors of Association for Pearson's r), the association between error and test coverage are strong.

Krishnamurthy [16] presented experimental data that is indicative of a high correlation between code coverage and software reliability: the reliability increases with coverage. The reliability is computed by:

$Rp_{i'} = 1 - FPp_{i'}$

Where

$p_{i'}$: Erroneous program with i remaining faults, $i \in \{0,1,\ldots,N\}$

$Rp_{i'}$: True reliability of pi′

$FPp_{i'}$: Failure probability of $p_{i'}$

The code coverages they measured are block, decision, and all-uses. Block and decision are presented above. In reference [17], Hong Zhu gives the definition of all-uses criterion as the following:

All-uses criterion: A set **P** of execution paths satisfies the all-use criterion if and only if for all definition occurrences of a variable *x* and all use occurrences of *x* that the definition feasibly reaches, there is at least one path *p* in P such that *p* includes a subpath through which that definition reaches the use.

Krishnamurthy [16] examined 3 Unix programs with about 200-1000 LOC. Faulty versions of programs were created by introducing defects into each of them. These seeded defects were all structural/coding defects [11], such as data initialization and boundary errors.

The code coverage (block, decision, and all-uses) was measured using random testing. They used the correlation analysis technique and observed that the degree of correlation between all-uses coverage and reliability has an inverse relationship with the fault density, and a direct relationship with program size. The correlation coefficient of the most complex program---sort, varying from 0.63 to 0.87, is higher than those of other 2 programs, varying from 0.12 to 0.45 [16]. However, their conclusions were based on relatively small programs. According to Karcich [55], for a valid conclusion, the system should be 5000 lines of code or larger.

The above prior work showed that structural test coverage has strong correlation with the detection of defects and software reliability. However, there are works that argue that *good coverage does not imply good fault detection* [5]. Even the high coverage level test sets were not terribly likely to detect the faults [10]. Achieving 100% statement coverage does not imply correctness [5,6,8,15]. The contradiction has led some researchers to think that if tests are designed simply to satisfy the coverage criteria, then their coverage may not be correlated with defect detection. *It is dangerous to rely on structural coverage alone* [5]. Combining code coverage with other testing techniques, such as functional testing, may enhance the effectiveness of functional testing. Studies relating code coverage with functional testing are outlined in Section 2.1.3.

2.1.3 Using code coverage with functional testing

The following prior research proposes that functional testing with code coverage could enhance test effectiveness.

Kantamneni [1] proposed a technique called "Structurally Guided Black Box Testing". Structural information is used to automatically guide black-box test case generation. They argued this technique which can help black-box testing to cover those *hard to test* branches will significantly enhance test effectiveness based on the idea: *If the code is never executed in the first place then faults can never be detected.*

Chen [7] proposed a methodology to integrate the black- and white-box techniques based on mathematical theorems. The black-box technique is used to select test cases. The white-box technique is mainly applied to determine whether two objects resulting from the program execution of a test case are observationally equivalent. It is also used to select test cases in some situations.

Edward Kit [14] mentions that Logic coverage (code coverage) is also important for two reasons:

(1) It indirectly improves function coverage. In some functional tests, white-box methods are used to increase logic coverage of functional tests. During functional tests, code coverage is measured; if it is too low, testers will make more functional test cases to increase code coverage. Thus, indirectly, code coverage helps to execute some functions hard to discover.

(2) It is necessary for the testing of logic paths that are not discernible from the external functionality (e.g., a math function that uses completely different algorithms, depending on the values of the input arguments).

The above studies indicate that code coverage can enhance the effectiveness of functional testing. Based on our survey of Structural test coverage, we are encouraged to believe that higher code coverage may result in less residual defects and thus improve the reliability of the software.

2.2 Unit Testing Efficacy

Unit testing is the process of running the executable of a module (functions, small programs, classes, etc) with the view of inducing failures and identifying subsequent faults [9]. Several prior works evaluated it experimentally [2,9,15]. They concluded that unit testing is effective:

- Unit tests are very effective in testing class or component functionality [2,9,15];
- Unit tests force clarification of class behavior and may lead to class redesign [15];
- Unit tests are cost-effective. The time spent on function test/system test is expected to be shorter than the previous model (without unit testing) since more extensive testing of each component's function has occurred during unit test [2];

However, Younessi [9] also mentions it is not very effective for certain defects. There is a large class of defects for which unit testing has exceptionally low efficacy. Further investigation shows that most structural defects, e.g., maintainability defects such as absence of a comment or a poorly named variable, are exceptionally improbable if not impossible to be detected using unit testing [9].

In addition, based on their results, Younessi [9] points out that unit tests "uncovers far fewer [defects] than we would expect ". This leads us to explore which kinds of bugs unit testing is not effective to detect and what countermeasures we should apply. This is discussed through defect causal analysis in Chapter 5.

2.3 Software Defect Causal Analysis

The common motivation of prior work on software faults and DRCA is to help develop strategies for preventing defects or detecting them earlier. Based on the goals of this thesis (introduced in section 1.4), the motivations for me to do DRCA are the following points:

- To analyze the reason why the degree of correlation between code coverage and the number of remaining defects is lower than we expected;
- To propose modified practices for future projects;

My survey on DRCA focuses on: defect categories, ways of collecting data, and distribution of defect root causes. The following are some interesting findings from the survey.

1. Defect categories are not unique

Different motivation leads to different defect categories. There is no universally correct way to categorize bugs. Beizer [20] provided a sample categorization according to their occurrence in the development life cycle in one project; the defect categories are: requirements, features and functionality, structural bugs, data, implementation and coding, integration, system and software architecture, test design and execution, others. These bugs were caught in independent testing, integration testing and system testing. The number of bugs detected by developers at the component (unit test) level is unknown.

Whitaker [23] presented an estimate of the root causes of software errors discovered in certain military electronic systems. His defect cause categorizes are a combination of software development phase triggers (like Requirements translation, Logic design, etc) and human, and environment triggers.

Leszak et al [24] used a novel strategy to analyze defect root cause. They allowed for multiple root causes to be defined as well as for no root causes. They provided a set of four root cause classes: phase-related, human related, project-related, and review-related (see Table 2.1).

Table 2.1: Multiple root causes

Phase triggers	Human related triggers	Project triggers	Review triggers
Requirements	Change	Time pressure	No or incomplete

	coordination		review
Architecture	Lack of domain knowledge	Management mistake	Not enough preparation
High level design	Lack of system knowledge	Caused by other product	Inadequate participation
Component Spec/Design	Lack of tools knowledge	N/A	N/A
Component implementation	Lack of process knowledge		
Load building	Individual mistake		
N/A	Introduced with other repair		
	Communication problem		
	Missing awareness of need for documentation		
	N/A		

2. Collecting bugs

There are two ways of collecting bugs: sampling [24,25] and exhaustive analysis of all reported problems collected during some particular parts of the software development processes [20,23].

3. Defect root causes

Some researchers present statistical data about percentages of software errors that occur in each development phase [20,24]. There is a common result: most bugs originate from component spec/design and implementation. Based on the data provided by Beizer [20], over 57% of bugs in the project originate from component spec/design and implementation. 47.6% bugs originate from *data* and *structural bugs* and 9.9% are from *Implementation and coding.* Leszak [24] observed that 70% of defects originate from components: 30% from spec/design and 40% from component implementation.

Then, how are we to decrease bugs originating from component spec/design and implementation in future projects? Leszak [24] proposes that for component design, they will improve the requirements management and systems engineering processes, specifically capturing non-functional SW requirements. As for component implementation, they suggest bettering unit tests by increasing test coverage, by creating complete test specifications, by systematic case selection, and by test automation, etc.

Yu [29] lists the most frequent errors made during coding and concludes that the top two causes that have the greatest effects are: Inadequate attention to details (75%) and inadequate considerations to all relevant issues (11%); Fenton [28] also indicates that volume of program design documents have significant influence on the error rate.

4. Correlation between error rate and component size

Basili [26] presents the data showing a higher error rate in smaller sized modules. They also gave some tentative explanations for this behavour: the majority of the modules examined were small, causing a biased result; larger modules were coded with more care than smaller modules because of their size; and errors in smaller modules were more apparent. There may still be numerous undetected errors present within the larger modules since all the "paths" within the larger modules may not have been fully exercised.

Fenton et al [27] conclude that there is weak support for the statement that "Size metrics (such as Loc) are good predictors of number of pre-release faults in a module", which means that larger components have more errors. Also they indicate that there is no evidence that module size has a significant impact on fault density.

Fenton [28] collected the number of errors remaining in a program at the beginning of the testing phase of development and used correlation analysis and one-way analysis of variance to examine the relationship between error rate and error factors. They concluded that large modules contained many residual defects, and program size has no substantial influence on the error rate.

The above conflicting results lead us to examine if there is correlation between error rate and component size in the Swurv project.

2.4 Conclusion

This chapter presented an overview of related work that serves as the foundation for the research of this thesis. The survey on structural test coverage presents two results: one is that structural test coverage has strong correlation with the detection of defects and software reliability, and the other is that code coverage can enhance effectiveness of functional testing. Also, the survey on effectiveness of unit testing gives us the result: Unit testing is efficient. Thus, these results suggest that unit tests with higher code coverage should result in less residual defects. We discuss the related hypotheses in section 4.1.

The survey on effectiveness of unit testing also shows that unit tests may be blind for some kinds of defects, such as maintainability defects. In section 5.1, we will analyze this based on the defect root causal analysis.

Results of our survey on DRCA support the design of our defect causal analysis: our defect categories and our sampling method to collect bugs (see Chapter 3). In the meantime, it leads us to set up two more objectives besides proposing improvement actions:

- Does the Swurv project have a lot of bugs injected from component spec/design and implementation phases? I discuss this in Chapter 5.1.
- Is there correlation between error rate and component size? This is discussed in Chapter 4.2.

Chapter 3
Experimental Design and Case Study

Experimental design is very critical to case study. It applies principles and techniques at the data collection stage so as to ensure the generation of valid, defensible, and supportable conclusions. In this chapter, we detail the methods and techniques we used to collect and analyze the data, and present some basic statistical data of the Swurv project.

3.1 Data Collection

In this thesis, we hope to provide a contribution to the study of correlation between residual defects and structural test coverage. In other words, we collected data to examine the hypothesis: the higher structural test coverage we achieve in the unit testing phase, the less residual defects we have in system testing. From my survey, this technique is novel in the following aspects:

- Our data is obtained from an Extreme Programming (XP) domain.

  No related prior work has been studied in an XP domain.

- Our data involves coverage of unit tests, making our results particularly relevant in the XP domain.

  The main testing in XP is unit testing. Developers write, maintain and control unit testing. In XP, any residual bugs detected at system-level are incorporated into the unit test repository, as rules of XP dictate that developers must write unit tests to expose all detected bugs. System-level tests are relatively few and simple; they are created to minimally [36] satisfy the customer, not to serve traditional system-level verification. Thus, for XP projects, the appropriate coverage measure is that of unit testing.

  No prior work has used unit test coverage data. In prior work [3,13,55], coverage data are measured from system-level tests only.

- All our data is actual data from development of a commercial project.

In some prior studies [12,16], data is not from commercial products. Their defects are not real operational defects. Thus, our collected data and calculated results are more applicable to real life than those in prior work [12,16].

- Our defects are actual operational defects, detected from product usage.

  In some other prior studies [12,16], synthetic errors were created and manually seeded into mature software to simulate bugs. Thus, our collected data and calculated results are more realistic than those in prior work [12,16]. In the prior work [3,13,55], the defects are obtained from system tests or functional tests, with no relation made to actual operational product defects. In our work, the defects used are actual defects encountered during 9 months of product usage.

- Our results are based on data collected from more than 100 different classes.
  Some prior works base their results on data from one program/module [12] or 3 classes/modules [16]. In other prior work [13], the data was collected and analyzed based on four datasets. Therefore, their sample sizes are smaller than ours. As our data are collected and analyzed from over 100 classes, its diversity gives our results more generality.

We wish to answer the question: During XP development, if developers had used code coverage as a measure of unit testing adequacy, would this have resulted in less operational defects at release time? We studied:

A. Whether classes with higher code coverage at release time have less residual defects than others.

B. Whether we could improve defect detection of each class by improving its code coverage.

Other prior works studied only the last sub-point above. They are concerned with the relationship between the code coverage and the number of detected defects of the same program/module/dataset, and more specifically, whether more of one class's defects can be detected by improving the class's code coverage.

We also study whether we can detect the residual defects by improving classes code coverage. However, after detailed analysis, we found very few detects (only 3.28%) can get detected by improving code coverage (See section 5.1.2).

3.1.1 Swurv project data

In order to do analysis on correlation between classes' structural testing coverage and the number of residual defects, we need to set a start date to get each class's structural testing coverage, and its number of residual defects which were detected from the start date to the effective end of development for this project.

To get testing coverage of a specific date, we have to get back that day's version of the project from Concurrent Versions System (CVS) (see section 3.1.1.1) and rerun the whole project. The start date is set as '2003/01/01', which is the furthest back date when Swurv project has the required data to rerun. The end date of collecting bugs is '2003/09/24', which was the effective end of development for this project. During the period from 2003/01/01 to 2003/09/24, operational acceptance tests (system tests), which identified residual bugs used in correlation analysis, were being performed. At the same time, developers added a small amount of new functionality and fixed the detected bugs.

Therefore, during collecting data phase, our main aim is to get:

- The number of remaining defects of each class after 2003/01/01. The number of remaining defects of each class is obtained by adding up the bugs a class was involved in after 2003/01/01.
- The coverage data of each class before the first fault was detected by system tests after 2003/01/01.

1. Which classes are involved in a bug?

Swurv applied CVS for software version control. CVS is a powerful method of recording the history of your source files and allowing many developers to work on the same source code. Each developer checks out a copy of the current version of the source code from CVS and then is able to work on their own personal copy separately from other developers. When they have made updates, they commit them back to the CVS repository. CVS automatically reconciles conflicts, and if they can't be reconciled, it flags them for the developer to manually reconcile.

In Swurv, developers are required to submit comments together with the updated source programs, such as the Bug-ID for the bug they fixed and a simple description of the update and the bugs. Therefore, if we know a bug was detected after the specified date 2003/01/01, then we could find which classes are involved in this bug by searching the file "ChangeLog" provided by

CVS. For example, in the file "ChangeLog" of the Swurv project, there is a line to describe bug 583:

```
2003-09-05 Developer-name <Developer-Email-Address>
        */usr/local/moly-cvs/swurv/issues/tracking, /usr/local/moly-
cvs/swurv/servlets/AddressCardDeleteGUI.sjf:

        fixed bug 583
```

From the information, we know the file 'AddressCardDeleteGUI.sjf' is the only class file involved in bug 583 (The Swurv project uses the ".sjf" file extension for java source files. They are converted to ".java" files by the Swurv pre-processor.).

From CVS, We can also find out the exact changes made to the 'AddressCardDeleteGUI.sjf' file by using the log and diff commands:

```
$ cvs log servlets/AddressCardDeleteGUI.sjf

----------------------------

revision 1.77

date: 2003/09/05 15:03:27;  author: tsingh;  state: Exp;  lines: +58 -28

fixed bug 583

----------------------------

revision 1.76

date: 2003/08/12 18:37:04;  author: ssmith;  state: Exp;  lines: +6 -1

bugzilla 569
```

```
.....
```

Thus, we know revision 1.77 fixed the bug 583. Then we use the following command to get the modification code to fix the bug:

```
$ cvs diff -r 1.77 -r 1.76 servlets/AddressCardDeleteGUI.sjf
```

The following display lists the modification. The front sign '<' means the line was deleted, and '>' means the line was newly inserted into the new version.

```
diff -r1.77 -r1.76
```

```
35a36
>       var alreadySubmitted = false;
38c39,42
<             $TOP_Frame.$ABSTRACT_POPUP_COF_FRAME_NAME.submitForm();
---
>             if(!alreadySubmitted) {
>                alreadySubmitted = true;
>                $TOP_Frame.$ABSTRACT_POPUP_COF_FRAME_NAME.submitForm();
>             }
43a48
> .....
```

The above information helps us to analyze each bug's root cause.

2. Bugs

How can we know which bugs were detected after our start date---2003/01/01? As Swurv uses "Bugzilla" to record bugs, we can search Bugzilla to get the desired bug list. Bugzilla is a database for bugs that lets people report bugs and assigns these bugs to the appropriate developers. Developers can use Bugzilla to keep a to-do list as well as to prioritize, schedule and track dependencies.

Thus, for each bug in the Bugzilla bug list, we can search 'ChangeLog' file to find which classes are involved in the bug. After searching all the bugs of the result bug list, the number of residual bugs for each class can be calculated.

3. Code coverage

We use the Clover code coverage tool to obtain code coverage for classes. Clover reports Total/Overall, Method, Statement, and Conditional coverage at the project level down to each line of source code. It provides two report formats: .html and .xml. Thus, we can write scripts to get coverage data of each class from the .xml report file.

3.1.2 Independent Variables

We examine correlation of pairs of independent variables in Chapter 4. These independent variables are listed and explained as follows:

Total-Bugs: the number of remaining defects in each class after 2003/01/01

Bug Density: calculated by Total-Bugs/Kncloc, kncloc = ncloc/1000 (See definition below)

Overall-%: a quick guide to how well a class is covered, reported as 'Total' coverage by Clover

Statement-%: the statement coverage of a class.

Conditional-%: the conditional coverage of a class.

Method-%: the method coverage of a class.

Class Age: the number of days from the class's creation date to the coverage date. A script was written to get the creation date from CVS.

No. of Revisions: the number of revisions of a class from creation to the coverage date. It was extracted from CVS by a script.

Ncloc: the abbreviation of non-commented lines of code. NCLOC is a measure of the uncommented size of a class. It was extracted from the Clover's xml report by a java program.

3.1.3 Data collection and Filter Process

The actual data collection process includes the following data query and filter process:

1. Query Bugzilla to get bugs detected after 2003/01/01 to 2003/09/24.
   My query criterion is: The bugs that were fixed from 2003/01/01 to 2003/09/24. The query returned a list of 120 bugs. After reading their descriptions, I deleted some invalid bugs and some bugs due to system administrator's improper configuration.

   In this study, we assume that the Swurv project has no remaining bugs after 2003/09/24, as the development was completed at that time, and all relevant bugs were fixed.

2. Get the related classes of each bug from CVS "ChangeLog".
3. Search the related classes' coverage data from coverage.html provided by Clover.

We used Clover to collect coverage data of unit tests of the Swurv project on the first day of each month from 2003/01/01 to 2003/09/01. We do not have testing coverage data of each day during the period, so we are restricted to mining the monthly coverage data. Although we could use CVS to recover old code and unit tests, we cannot re-run the tests to obtain coverage at a given time-point, because this requires re-building the entire project environment at that time-point, which is beyond the scope of this thesis.

Therefore, for a class, such as com/swurv/base/BookmarkBranch.sjf, and a bug, such as 533, which was detected on date 2003/06/17, the coverage data we use is from 2003/06/01, as it is the latest recorded coverage data for the class com/swurv/base/BookmarkBranch.sjf on or before the bug detection date.

After this process, a list of classes which are involved in one or more bugs, their coverage data, and detection date for each bug were collected. (See Table 3.1)

Table 3.1: An illustration of data collected after step 3

Bug-ID	Classname	Detection-date	Coverage-date	Overall-%	Statement-%	Conditionals-%	Methods-%
533	com/swurv/base/BookmarkBranch.sjf	2003-06-17	2003-06-01	77.60	80.80	68.20	84.40
546	com/swurv/base/UserManager.sjf	2003-07-02	2003-07-01	67.80	69.70	60.80	78.20
573	com/swurv/base/UserManager.sjf	2003-07-02	2003-07-01	67.80	69.70	60.80	78.20
589	com/swurv/base/UserManager.sjf	2003-08-21	2003-08-01	63.90	64.80	60.40	70.00
585	com/swurv/base/BookmarkBranch.sjf	2003-08-14	2003-08-01	63.90	64.80	60.40	70.00
607	com/swurv/base/BookmarkBranch.sjf	2003-08-25	2003-08-10	79.00	81.2	75	77.4
……							

4. For each class, calculate the total remaining bugs, Total-Bugs, and select the earliest coverage data after 2003/01/01.

   The value of 'Total-Bugs' is used to calculate defect density and the selected coverage data is used to calculate correlation coefficients. The reason why we use the earliest coverage is that we need a date at which all bugs in Total-Bugs existed for that class (some bugs are fixed after 2003/01/01). Table 3.2 gives a sample of the result list---BugClasses.

Table 3.2: An illustration of data collected after step 4

Total-Bugs	Classname	Overall -%	Statement-%	Conditionals-%	Methods-%
12	com/swurv/base/BookmarkBranch.sjf	77.6	80.8	68.2	84.4
5	com/swurv/base/BookmarkTrunk.sjf	85.5	87.7	76.9	88.6
5	servlets/DownloadJavascript.sjf	71.9	80	100	20
4	com/swurv/base/FileRef.sjf	90.8	91.1	82.6	96.5
4	com/swurv/base/WebStorageBranch.sjf	52.3	55.6	33.3	52.3
4	com/swurv/letterbox/IncomingMessage.sjf	80.5	82.3	72.9	91.7
4	servlets/ECN_GUI.sjf	100	100	n/a	100
……					

5. Considering the case that some classes are not used by the Swurv project, but still stay in CVS, we have to exclude them to ensure that our study is based on valid data, since we don't know how many residual bugs they have. We do this as follows:

   Select out a list of classes which are all operational Swurv classes as of 2003/09/24 named EffectClasses from CVS. EffectClasses are divided into 2 subsets (lists): EffectBugClasses and EffectNoBugClasses as follows (using set BugClasses from step 4):

$$\text{EffectBugClasses} = \text{EffectClasses} \cap \text{BugClasses}$$

$$\text{EffectNoBugClasses} = \text{EffectClasses} - \text{BugClasses}$$

   EffectBugClasses--- a file to store classes which were all operational Swurv classes as of 2003/09/24, and from which there were one or more bugs detected between 2003/01/01and 2003/09/24

EffectNoBugClasses--- a file to store classes which are all operational Swurv classes as of 2003/09/24, and from which no bug was detected between 2003/01/01and 2003/09/24.

6. From set EffectBugClasses, delete non-relevant classes, such as test stub classes, classes for generating html elements, and classes whose overall coverage is equal to 0.

   We must delete classes with reported coverage of 0 because 0 is not their true coverage value, but was reported so by clover because of errors in the test scripts. We cannot rerun the corrected tests to obtain the true coverage values because recreating the proper environment is beyond the scope of this thesis.

7. Extract coverage data for classes in set EffectNoBugClasses from the .xml coverage report on 2003/01/01.

   Of the seven steps above, the first three steps are mainly done manually, and the last four steps mainly executed by scripts. After the above steps, we have a list of classes which have class name, the number of remaining defects (Total-Bugs), and coverage data. Statistical correlations are computed between the remaining errors and four coverage measures. The correlation coefficients (see Table 3.3) are significantly lower than prior work suggests (see section 2.1.2).

Table 3.3: Correlation Coefficients Before Defect Analysis

	Total/Overall	Statements	Conditionals	Methods
Defects	-0.034	-0.114	0.05	-0.145

Table 3.4 groups classes by the number of bugs they are involved in and presents the average coverage for each group. For example, for the first group, which has 196 classes, zero bugs were detected in these classes. Its average coverage of Total/overall, statements, conditionals, methods are 79.67, 81.94, 70.36, and 88.56, respectively.

Table 3.4: Average Coverage for Different Bug Number Group

No. of Bugs	No. of classes	Total/Overall-%	Statements-%	Conditionals-%	Methods-%
0	196	79.67	81.94	70.36	88.56

1	58	79.67	81.84	72.01	82.54
2	23	70.90	73.62	60.61	85.72
3	30	69.78	71.23	61.26	75.36
4	16	77.44	78.79	70.71	80.60
5	13	84.68	85.78	78.94	87.94
6	7	80.23	84.81	88.23	59.03
7	17	78.33	79.92	69.82	83.06
11	1	86.70	88.00	79.70	95.40
16	1	82.80	85.7	73.90	87.90

8. Implement defect root cause analysis to get each bug's type.
   From Table 3.3, our correlation coefficients are really low, suggesting there is no correlation between remaining defects and code coverage. We want to explore the reason, so we decided to do defect root cause analysis.

   During the defect root cause analysis, we found some bugs had occurred in newly added functions/code as per customer's requirements. We also found that some bugs had the same root cause (if we fixed one, the others were fixed too). Additionally, some classes were not really involved in a bug because developers fixed several bugs together and submitted/commented all classes at the same time. The above cases should be excluded from our data set of correlation analysis. In the later correlation analysis in Chapter 4, we deleted these cases. Here I would like to recommend if you plan to do a similar study, you should require developers to log bugs' information formally and in detail.

   After this step, the new correlation coefficients were calculated and are presented in Table3.5 (the old ones were in Table 3.3). New coverage averages were also calculated, and are presented in Table 3.1 in section 3.3 (the old ones were in Table 3.4).

Table 3.5: Correlation Coefficients After Defect Analysis

	Total/Overall	Statements	Conditionals	Methods
Defects	-0.05	-0.031	0.02	-0.2

The correlation coefficients between the number of defects and structural testing coverage are still low. The above data suggests there is no association between them.

9. Get the creation date, class age and number of revisions for each class by running a C script using CVS commands. For each class in EffectClasses, its class size (Ncloc) was extracted from the .xml coverage report.
   Why do we collect these data? A close correlation between program size and the number of errors in each program has already been suggested in prior work [26,27]. We would like to determine if our data corroborates prior work, or provides a counter-example. In addition, we also want to examine the relationship between defect density and other independent variables, such as class age, and the number of revisions.

Thus, our collected data is a list of classes which has class name, the number of remaining defects (Total-Bugs), defect density (Total-Bugs/Kncloc, kncloc = ncloc/1000), coverage data, the number of revisions, class age, and class size (Ncloc), etc. (see Table 3.6. As it is too wide, I separate it into 2 tables.) We will do correlation analysis based on these data in Chapter 4.

Table 3.6: An illustration of data collected after step 9

No. of Bugs	Classname	Overall-%	Statement-%	Conditional-%	Methods-%
12	com/swurv/base/BookmarkBranch.sjf	77.6	80.8	68.2	84.4
5	com/swurv/base/BookmarkTrunk.sjf	85.5	87.7	76.9	88.6
5	servlets/DownloadJavascript.sjf	71.9	80	100	20
4	com/swurv/base/FileRef.sjf	90.8	91.1	82.6	96.5
4	com/swurv/base/WebStorageBranch.sjf	52.3	55.6	33.3	52.3
......					

No. of Bugs	Classname	BugDensity	ClassAge (Days)	No. of Revision	Loc	Ncloc
12	com/swurv/base/BookmarkBranch.sjf	26.2	659	92	530	458
5	com/swurv/base/BookmarkTrunk.sjf	9.2	659	161	599	541
5	servlets/DownloadJavascript.sjf	2.7	708	114	1909	1827
4	com/swurv/base/FileRef.sjf	10.2	312	39	443	391
4	com/swurv/base/WebStorageBranch.sjf	50.6	141	15	95	79

......

3.2 Correlation Analysis Techniques

3.2.1 Introduction

In most statistical packages, correlation analysis is a technique used to measure the association between two variables. The most widely used type of correlation coefficient is Pearson's r. However, for some cases, Pearson's r is not suitable as outlined below. In my thesis, I will use Pearson and Spearman correlation coefficient (r_{rank}).

1. Pearson's r

A correlation coefficient, r, is a statistic used for measuring the strength of a supposed linear association between two variables. Generally, the correlation coefficient varies from -1 to +1. A positive correlation coefficient indicates a direct relationship, and a negative correlation coefficient indicates an inverse relationship between two variables. Correlations of –1.00 and 1.00 are perfect correlations. The value near 0 indicates no linear relationship between the variables. A value of zero exactly conveys that two variables are unassociated [53]. Table 3.7, from reference [53], presents qualitative descriptors for size of relationship for r:

Table 3.7: Descriptors of Association for Pearson's r [53]

Correlation	Size of Association	Strength of Association
About .10 (or -.10)	Small	Weak
About .30 (or -.30)	Medium	Moderate
About .50 (or -.50)	Large	Strong
About .70 (or -.70)	Very large	Very Strong

- Assumptions of Pearson's r

Pearson correlation has the following assumptions:

a) For a valid confidence interval around the correlation coefficient both variables must be normally distributed [52,54].

b) Random sampling is conducted [54].

c) The data is measured on a continuous scale [54].

In practice, many researchers ignore assumption (a). They follow a rule of thumb that if your sample size is 50 or more, then serious biases are unlikely [20]. Bittick [54] mentioned in "introduction to correlation" that when using Pearsons' r, " the Central Limit Theorem states that a sample size of 30 or more will satisfy requirement (b)." The minimum of sample sizes of our data is 66.

Moreover, most of our data are measured on a continuous numeric scale, such as bug density, and testing coverage data. As to class age, class size (ncloc), and the number of class revisions, they are discrete numerical variables, but can be treated as continuous for the purpose of analysis. In short, all of our empirical data can be applied to Pearson correlation analysis technique.

- Statistical significance test of Pearson's r

The significance test of Pearson's r conveys the probability of obtaining one's study sample correlation given that the population correlation is 0.00. In our study, we will use non-directional hypothesis pairs to test the correlation between variables. The non-directional hypothesis states [53]:

Null: The correlation (Pearson's r) in the population from which the sample was randomly selected equals 0.0.

Research: The correlation (Pearson's r) in the population from which the sample was randomly selected does not equal 0.0.

In symbols, the non-directional pair is:

H_0: $\rho = 0$ H_1: $\rho \neq 0$

Statistical theory indicates that the larger the sample size, the smaller the absolute value of of r needed to reject the null hypothesis. A straightforward way to assess the significance of r is to consult a table of critical values. We follow this approach. Reference [53] gives critical values of Pearson's r for (1) .01 and .05 significance levels, for (2) one-tailed and two-tailed tests, and for (3) varied degrees of freedom (N-2). We reproduce this table as Table 3.8. With the help of the table, we evaluated the Pearson's r at the .01 and .05 significance level, and then decide to accept or reject the hypotheses as follows. If the calculated value of $|r|$ is less than or equal to the critical value of r, accept the null hypothesis - there is no proof of significant correlation between the variables. If the calculated value of $|r|$ is greater than the critical value of r, we will reject the null hypothesis and accept the research hypothesis.

We use two tailed tests since we don't know if it is a negative or positive association between variables. Note that H_1 states '......does not equal 0.0......'. It does not state whether the difference between the two variables is greater or less than zero.

Table 3.8: Critical value for Pearson's r

	Two Tailed Test			One Tailed Test	
N	df	P=.01	P=.05	P=.01	P=.05
72	70	.302	.302	.195	.274
82	80	.283	.283	.183	.257
202	200	.181	.181	.116	.164
302	300	.148	.148	.095	.134

2. Spearman's r_{rank}

Spearman correlation coefficient is a commonly used nonparametric correlation coefficient. Nonparametric correlational techniques are designed to estimate the correlation or association between variables measured on nominal and/or ordinal scales, or metric variables that have been reduced to nominal and/or ordinal scales. For Spearman's r_{rank}, the descriptors in Table 3.7 may be used to interpret the size and strength of the association [53].

- Assumptions of Spearman's r_{rank}

1. This method is sensitive only to the ordinal arrangement of values. The two variables are ordinal or metric variables that have been reduced to an ordinal scale of measurement
2. Nonparametric methods are most appropriate when the sample sizes are small. When the data set is large (e.g., n > 100) it often makes little sense to use nonparametric statistics at all.

- Statistical significance test of Spearman's r_{rank}

In testing the significance of Spearman's r_{rank}, the non-directional pair is the same as that of Pearson's r:

H_0: $\rho = 0$ H_1: $\rho \neq 0$

Given a sample size of N cases, a statistical table can be used to evaluate the test (we reproduce the table given in [57] as Table 3.9. When sample size is greater than or equal to 30, the critical value of Pearson's r and Spearman's r_{rank} are nearly

Table 3.9: Critical values of Spearman's r_{rank}

TABLE G Critical Values of r_s at the .05 and .01 Levels of Significance (α)

	α			α	
N	*.05*	*.01*	*N*	*.05*	*.01*
5	1.000	—	16	.506	.665
6	.886	1.000	18	.475	.625
7	.786	.929	20	.450	.591
8	.738	.881	22	.428	.562
9	.683	.833	24	.409	.537
10	.648	.794	26	.392	.515
12	.591	.777	28	.377	.496
14	.544	.714	30	.364	.478

Source: E. G. Olds, *The Annals of Mathematical Statistics*, "Distribution of the Sum of Squares of Rank Differences for Small Numbers of Individuals," 1938, vol. 9, and "The 5 Percent Significance Levels for Sums of Squares of Rank Differences and a Correction," 1949, vol. 20, by permission of the Institute of Mathematical Statistics.

identical [53]. Hence, when $N \geq 30$, we can use the critical value of Pearson's r (see Table 3.8); when $N < 30$, we can consult the Table 3.9 [57]. Suppose, for example, that N=8 and a correlation value r rank of .91 is obtained. Since .91 exceeds the critical value at the p=0.01 level of significance (0.881, see Table 3.9), then we are more than 99% confident in rejecting the null hypothesis H_0 that states no correlation between variables.

3. Spearman or Pearson?

Both Spearman correlation and Pearson correlation have assumptions (see Sections 3.2.1.1, and 3.2.1.2). First of all, researchers [59] suggest that if variables' values are at the interval or ratio level of measurement, then use the Pearson correlation coefficient; If variables' values are at the ordinal level of measurement (ranks), then use the Spearman rank-difference correlation coefficient.

Secondly, Pearson correlation calculations are based on the assumption that both X and Y values are sampled from populations that follow a Gaussian distribution, at least approximately. With large samples, this assumption is not too important. However, if you don't wish to make the Gaussian assumption, select nonparametric (Spearman) correlation instead.

In addition, with Spearman's r_{rank}, you are effectively calculating Pearson's r on the rank-ordered positions of the original data instead of on the original values themselves. Thus, Pearson's r uses more of the information in the original data than Spearman's r_{rank}. This leads to the fact that Spearman's r_{rank} only has about 91% of the power of Pearson's r [58]. Power refers to the test's ability to reject the null hypothesis when it IS actually false. Therefore, for the same data-set, if the assumptions underlying r are not violated, Pearson's r test is more likely to give the correct decision.

Since assumptions from both Pearson and Spearman hold for our data, we use both correlation significance tests in our analysis (see Chapter 4 for a description of why and when we use each.)

4. What is statistical significance?

The statistical significance (p-level) of a result is an estimated measure of the degree to which it is "true" (in the sense of "representative of the population"). More technically, the value of the p-level represents a decreasing index of the reliability of a result. The higher the p-level, the less we can believe that the observed relation between variables in the sample is a reliable indicator of the relation between the respective variables in the population.

Specifically, the p-level represents the probability of error that is involved in accepting our observed result as valid, that is, as "representative of the population." For example, a p-level of .05 (i.e.,1/20) indicates that there is a 5% probability that the relation between the variables found in our sample is a "chance" (in other words, assuming that in the population there was no relation between those variables). In many areas of research, the p-level of .05 is customarily treated as a "border-line acceptable" error level. With large samples, researchers typically choose the .01 level since choosing .01 level increases the size of association required for rejection [53].

3.2.2 Applying Correlation analysis in Data Mining

Correlation analysis arises as a basic probabilistic vehicle that is used to quantify associations (links) between individual variables [50]. It is applied to reveal and quantify relationships between variables in any task of data mining [49,50,51]. Especially, it is used to explore the correlation between coverage data and reliability or detected faults [16,55].

Curry [49] used the correlation analysis method to explore the relation between existing amount-of-reuse metrics in the C Programming Language. They used 2 indexes to assess the results:

1. the correlations between these metrics within each project.
2. the significance of the correlations within each project at the 0.049 level.

Takahashi [51] also use correlation analysis and one-way analysis of variance to analyze correlations between each of the error factors and the number of errors found in a program. He explained that previous work showed a close correlation between program size and the number of errors in each program. Thus, the number of errors and the error factors, which are closely related to program size, are normalized by using KLOC.

3.2.3 Analyse-it Tool

Analyse-it is an add-in to Microsoft Excel allowing one to analyse data straight from an Excel worksheet [52]. Statistical functions are provided for descriptive statistics, testing normality, comparing groups, correlation, and multiple linear & polynomial regression analysis. In my thesis, I will use this tool to calculate the correlation coefficient (Pearson'r and Spearman's r_{rank}) between fault rate and other factors with a significance at the 0.01 and 0.05 level.

3.3 Basic statistical data

The results reported in this thesis are based on empirical data obtained from the Swurv project. After filtering invalid bugs, and bugs occurring in newly added functions, the basic information of this study is reported below:

- There are 61 remaining bugs detected by operational usage (system tests) since January 1, 2003.
- 74 classes are involved with from 1 to 12 bugs detected by operational usage (system tests).
- 133 classes are not involved in bugs. They represent 40% of the project's code.
- The total studied buggy classes have over 43,000 Ncloc.
- The average overall, statement, conditional, and method coverages are 79.53, 81.4, 61.8, and 85.6 respectively. They are comparatively high. In reference [3], they observed that

beyond a certain range (70% -80%), increasing statement coverage becomes difficult and is not cost effective.

The statistical mean/average value is by far the most commonly used measure of central tendency. In order to find some tendency between a class's remaining defects and other independent variables, I also wrote a program to calculate the mean values. I firstly divided collected data into several bug no. groups according to how many bugs a class is involved in, and then I calculated the mean values for different bug no. groups(See Table 3.10)

Table 3.10: Average data by bug no. group for all classes

BugNoGroup	TotalClasses	Overall (%)	Statement (%)	Conditional (%)	Method (%)	Age (days)	Revisions	Ncloc
0	133	79.74	81.57	57.14	85.35	519	22.38	132.5
1	42	81.37	83.29	71.32	89.28	568	72.9	328.2
2	19	72.8	74.02	69.78	80.68	424	45.16	208.9
3	6	83.18	84.83	73.8	90.97	809	159.5	673.5
4	4	80.9	82.25	62.93	85.13	391	55	377.5
5	2	78.7	83.85	88.45	54.3	683	137.5	1184
12	1	77.6	80.8	68.2	84.4	659	92	458

Table 3.10 shows that:

- The Non-bug class group (BugNoGroup 0) has the smallest size---their average Ncloc is 132.5. The BugNoGroups 1, and 2 (in which classes have 1 or 2 remaining bugs) have the third and second smallest average class size respectively. This leads us to study in Chapter 4 whether class size is correlated with the number of remaining defects or remaining defect density.
- The Non-bug class group (BugNoGroup 0) has the least revisions while BugNoGroup 3, in which classes have the most revisions, has the highest overall code coverage. Is the number of revisions a good predictor of fault or code coverage? In the Chapter 4, we will explore whether the number of revisions has correlation with the number of remaining defects and code coverage.
- Our data shows no clear relationship between remaining defects and average code coverage. However, prior work shows a high correlation between code coverage and

software reliability [16] and a linear correlation between the number of errors removed and code coverage [3,6,12,13]. In Chapter 5, we do defect root causal analysis to explore the reasons that result in our results. Moreover, in reference [12], researchers suggest that the relation model between defect coverage and code coverage can be used to estimate the residual defect density. In other words, higher coverage means lower residual defect density. We will validate if this hypothesis holds for our data in Chapter 4.

- Our data shows no clear relationship between class age and remaining defects. It suggests the number of remaining defects is not affected by how long the class has been created.

We also calculated mean values according to the overall/total coverage range. First of all, the collected classes are divided into several groups according to the code coverage. Then, the average number of bugs and average bug densities are calculated (see Table 3.11).

Table 3.11: Average data by overall/total coverage for all classes

Overall/Total Coverage Range (X)	Total Classes	Average No. of Bugs	Average BugDensity
X < 40	8	0.38	3.46
40=<X<50	4	0.00	0.00
50=<X<60	11	1.09	6.08
60=<X<70	21	0.29	1.46
70=<X<80	40	1.05	4.02
80=<X<90	70	0.76	2.51
X >= 90	52	0.06	3.17

Average No. of Bugs = Total Number of bugs / TotalClasses

Average BugDensity = Total Number of bugs / Total class size (Ncloc)

From table 3.11, we cannot see a clear reduction in average number of bugs or average bugdensity with code coverage growth. However, it is still possible a relationship exists, even though it is not clearly discernable with mean value analysis. As we know, the statistical mean represents the "balance point". The mean value is substantially affected by extreme values.

Especially, if there are values much higher or lower than others, the mean value is not an effective represent/explanation for samples. Thus, we will do our further study using correlation analysis on defect density and testing coverage in Chapter 4 to determine if a relationship exists.

3.4 Defect Categories and Distribution

From our collected data, some classes have higher code coverage, but don't have less remaining defects as expected. We want to analyze what causes this result and if residual defects can be detected by improving code coverage of related classes. Therefore, we categorize our collected defects based on root cause analysis. In this section, we introduce the defect categories we will use. We also present a grouping of bugs into these categories--- the root cause defect distribution.

3.4.1 Defect Categories

We analysed the bugs (defects) and divided them into the following six categories:

0. Those that were detected before the start date of 2003/01/01, or were detected after 2003/01/01 but occurred because of code that was added after 2003/01/01
   Since these bugs did not exist when we executed unit tests to get classes' coverage, these bugs are not related to the study. We should not consider these bugs to do correlation analysis.

1. Incomplete/incorrect requirements in stories.

2. Missed "default" cases: Programmers missed some cases from "common programming experience" (not in stories); for customers, these cases are too common to be ignored / missed.

3. Programmers missed/misunderstood cases from stories.

4. Component Design and Implementation bugs, including incomplete/incorrect design and coding errors.

4.1 Can be detected by improving code coverage
   If the related code was implemented and not covered by unit testing, it is possible to detect the defect by executing the related code. We assume that proper verification would check if the executed result is the expected result.

4.2. Cannot be detected by improving code coverage

4.2.1 Component Design problem--- inadequate/wrong features (w.r.t requirements), algorithm, performance. The appropriate code does not exist for some features.

4.2.2 Occurred in long-event functions--- need acceptance tests.

As we mentioned in section 1.2, long-event functions may need functional / acceptance tests since unit test tools will not keep status between events.

4.2.3 Lack of verification --- test produced incorrect output, but the test case did not catch this.

4.2.4 GUI Bugs

These bugs cannot be trapped by JUnit tests. In order to detect these bugs, it is necessary to check the GUI. As it is difficult to automate GUI testing, the Swurv project abandoned automated unit testing for the GUI. Thus, in our study, unit tests cannot detect GUI bugs.

5. Invalid Bugs. This includes unrelated bugs, repeated bugs, bugs that are related to classes whose coverage are unavailable, etc.

   Unrelated: Sometimes, developers fixed several bugs and committed involved classes together, so we may ascribe an unrelated bug to a class during collecting defects for the class. Thus, we refer to this bug as unrelated to the class.

   Repeated: if the same root cause resulted in several bugs, and if developers fixed one, the others got fixed, then we only consider the first bug while computing the correlation coefficient. The other bugs are repeated.

   Unavailable coverage: As we can not reinstate the proper environment from 2003/01/01 to run some tests to get accurate classes' coverage, these classes are not considered in computing correlation. This will not affect our correlation analysis, as we have enough sample classes to get a valid statistical correlation coefficient without them.

Structural testing, which generates test cases based on program structural information, probably will not detect the bugs which belong to specification/requirements defects (categories 1, 2, 3) but it may detect the implementation bugs (part of category 4). Thus, only improving appropriate classes' coverage cannot detect bugs in categories 1,2,3, but may detect some bugs in category 4.

Defect categorization is not easy. Sometimes, it seems some faults fit more than one category. Therefore, our categorizing rule is: if a bug belongs to several categories, it is ascribed to one category in terms of its origin/root cause instead of its consequences.

3.4.2 Defect Distribution

Based on the above bug categories, the Swurv bug distribution was calculated. The result is listed in Table 3.12. Bugs of Category 0 and 5 should be excluded from our statistical analysis for the reasons outlined in section 3.4.1.

Table 3.12: Distribution of Bugs Collected

Bug Category	Number
0 Occurred in newly added functions or detected before 2003/01/01	21
1 Incomplete/incorrect requirements in stories	4
2 Missed "Assumption" cases	11
3 Programmers missed/misunderstood cases from stories	3
4 Component Design and Implementation Bugs	43
5 Invalid Bugs	8
Total	90

Table 3.13 and Figure 3.1 give the Swurv bug distribution after filtering invalid bugs and bugs occurring in newly added functions or detected before 2003/01/01 (categories 0 and 5).

Table 3.13: Statistic of Swurv Bugs Distribution

Bug Category	Number	Percent
1 Incomplete/incorrect requirements in stories	4	6.56
2 Missed "Assumption" cases	11	18.03
3 Programmers missed/misunderstood cases from stories	3	4.92
4 Component design and implementation bugs	43	70.49
4.1 Can be detected by improving code coverage	2	3.28
4.2. Can not be detected by improving code coverage	41	67.21
4.2.1 Component design problem	18	29.51

4.2.2 Occurred in long-event functions	3	4.92
4.2.3 Lack of verification	2	3.28
4.2.4 GUI bugs	18	29.51
Total	61	

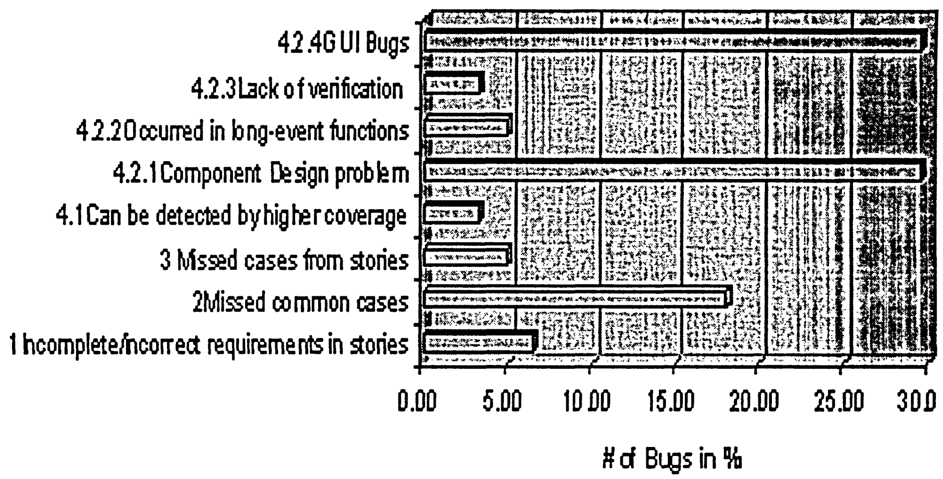

Figure 3.1: Swurv Bugs Distribution

In chapter 5, we will discuss the Swurv bug distribution and propose appropriate countermeasures to reduce the number of bugs in each category in future projects.

Chapter 4
The Hypotheses Tested and Results

After the data were collected, we proceeded to compute the correlations, and test several hypotheses which relate to defect density--- code coverage, class size, class age and the number of revisions. We used the defect density instead of defect number because in prior work, Fenton et al [28] observed that the number of errors is closely related to the program size. Therefore, we needed to normalize the number of defects by class size. We did this with defect density, which was expressed by the following equation:

Defect density = No. of defects / KLOC

where KLOC = Ncloc/1000

For most of our hypotheses, we used Pearson's r and Spearman's r_{rank} tests. Three indexes are used to evaluate the significance test of Pearson's r and Spearman's r_{rank}:

1. The correlation coefficient between independent variables;
2. The significance of the correlation at the .01 and .05 levels;
3. The sample size (N) or degrees of freedom (N-2). If the sample data includes non-bug classes, N is around 206, so the degree of freedom is 204; if sample data excludes non-bug classes, N is 74, so the degree of freedom = 74-2 = 72.

It is important to note that we excluded non-bug classes while calculating Pearson's r and Spearman's r_{rank} between defect density and other metrics. For significance tests of Pearson's r, the reason is because of the assumption that: "The data is measured on a continuous scale and normally distributed." There are a lot of non-bug classes, and their defect density is 0, the minimum. If we included them, the distribution would be severely skewed. For significance tests of Spearman's r_{rank}, on the one hand, the sample size is required not to exceed 100; on the other hand, we want to be consistent with the sample size of Pearson's r. Thus, we excluded non-bug classes from tests on initial values.

Since assumptions from both Pearson and Spearman hold for our data, we used both correlation significance tests on our data. However, in some cases, the tests may produce different conclusions for the same hypothesis. Based on the discussion in 3.2.1.3, Pearson's r test is more likely to give the correct decision if the assumptions underlying r are not violated [58]. Therefore, we depended more on Pearson's r.

Table 4.1 and Table 4.2 present the results of Pearson's r and Spearman's r_{rank} respectively.

Table 4.1: Results of Pearson's r

Variable Pair	Pearson's r		Sample Size N	Absolute Critical Value	
	p = .01	p=.05		p = .01	p=.05
BugDensity vs. Overall	-0.27	-0.27	74	0.30	0.23
BugDensity vs. Statement	-0.29	-0.29	74	0.30	0.23
BugDensity vs. Conditional	-0.14	-0.14	66	0.31	0.24
BugDensity vs. Methods	-0.35	-0.35	73	0.30	0.23
BugDensity vs. ClassAge	-0.21	-0.21	74	0.30	0.23
BugDensity vs. Revisions	-0.32	-0.32	74	0.30	0.23
BugDensity vs. Ncloc	-0.42	-0.42	74	0.30	0.23
Ncloc vs. Overall	-0.02	-0.02	207	0.18	0.135
Ncloc vs. Statement	-0.02	-0.02	207	0.18	0.135
Ncloc vs. Conditional	0.24	0.24	199	0.181	0.138
Ncloc vs. Method	-0.05	-0.05	206	0.18	0.135
Revisions vs. Overall	0.06	0.06	207	0.18	0.135
Revisions vs. Statement	0.06	0.06	207	0.18	0.135
Revisions vs. Conditional	0.22	0.22	199	0.181	0.138
Revisions vs. Method	0.0	0.0	206	0.18	0.135
Revisions vs. Ncloc	0.76	0.76	207	0.18	0.135

Table 4.2: Results of Spearman's r_{rank}

Variable Pair	Spearman's r_{rank}		Sample Size N	Absolute Critical Value	
	p = .01	p=.05		p = .01	p=.05
BugDensity vs. Overall	-0.14	-0.14	74	0.30	0.23
BugDensity vs. Statement	-0.12	-0.12	74	0.30	0.23
BugDensity vs. Conditional	-0.11	-0.11	66	0.31	0.24
BugDensity vs. Methods	-0.07	-0.07	73	0.30	0.23
BugDensity vs. ClassAge	-0.38	-0.38	74	0.30	0.23
BugDensity vs. Revisions	-0.64	-0.64	74	0.30	0.23
BugDensity vs. Ncloc	-0.85	-0.85	74	0.30	0.23
Ncloc vs. Overall	0.11	0.11	74	0.30	0.23
Ncloc vs. Statement	0.09	0.09	74	0.30	0.23
Ncloc vs. Conditional	0.13	0.13	66	0.31	0.24
Ncloc vs. Method	-0.04	-0.04	73	0.30	0.23
Revisions vs. Overall	0.13	0.13	74	0.30	0.23
Revisions vs. Statement	0.06	0.06	74	0.30	0.23
Revisions vs. Conditional	0.23	0.23	66	0.31	0.24
Revisions vs. Method	-0.16	-0.16	73	0.30	0.23
Revisions vs. Ncloc	0.78	0.78	74	0.30	0.23

4.1 Hypotheses relating to code coverage

It is widely believed that there is linear correlation between the number of errors removed and code coverage [3,12,13,19]. Especially in reference [12], researchers showed that code coverage could be used to estimate the residual defect density. Therefore, we investigated 4 related hypotheses:

Hypothesis 1a: Overall code coverage can be used to estimate the residual defect density;

Hypothesis 1b: Statement code coverage can be used to estimate the residual defect density;

Hypothesis 1c: Conditional code coverage can be used to estimate the residual defect density;

Hypothesis 1d: Method code coverage can be used to estimate the residual defect density;

4.1.1 Hypothesis 1a

Null: The correlation between overall code coverage and residual bug density in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between overall code coverage and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.3 and the corresponding sample correlation is plotted in Figure 4.1.

Table 4.3: Pearson's r of overall vs. Bug Density

n	74
Pearson's r statistic	-0.27
P(significance level)	0.01 or 0.05

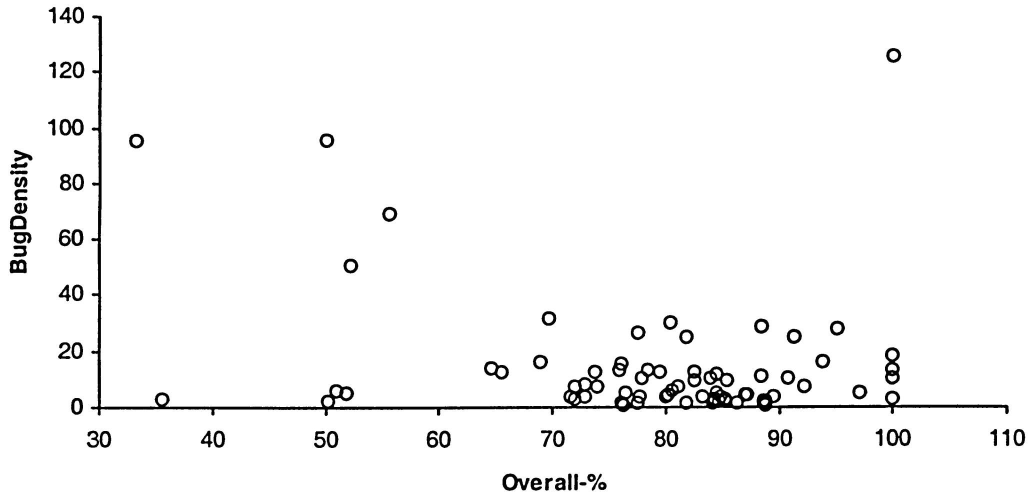

Figure 4.1: Sample Relation between Bug Density and Overall Code Coverage

When n=74, the critical value at significance level 0.01 is $r_{crit, p=0.01} = 0.30$ and the critical value at significance level 0.05 is $r_{crit, p=0.05} = 0.23$.

As Pearson's $r = -0.27$, $|r| > r_{crit, p=0.05}$, we must reject the null hypothesis, concluding that there is a negative correlation between overall and residual bug density at the 5% significance level. Since $|r| < r_{crit, p=0.01}$, we cannot reject the null hypothesis at the 1% significance level. This indicates the association is not strong, but it is close to moderate based on Table 3.7.

(2) The significance test of Spearman's r_{rank}

Table 4.4: Spearman's r_{rank} of Overall vs. Bug Density

n	74
Spearman's r_{rank} statistic	-0.14
P(significance level)	0.01 or 0.05

The result is listed in Table 4.4. Since $|r_{rank}| = 0.14 < r_{crit, p=0.05} = 0.23 < r_{crit, p=0.01} = 0.30$, we cannot reject the null hypothesis, concluding that there is no clear correlation between overall and residual bug density at the 5% significance level.

However, based on Table 3.7, $|r_{rank}| = 0.14$ can be interpreted as a weak correlation.

As stated above, we depend more on Pearson's r since it is more effective to disclose correlation. We conclude that there is a correlation between overall code coverage and residual defect density, and the size of association is close to moderate.

4.1.2 Hypothesis 1b

Null: The correlation between statement coverage and residual bug density in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between statement coverage and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.5. As the corresponding sample correlation plot is similar to that of overall vs. bug density, it is not given here.

Table 4.5: Pearson's r of Statement vs. Bug Density

n	74
Pearson's r statistic	-0.29
P(significance level)	0.01 or 0.05

As Pearson's r = -0.29, and $| r | > r_{crit,\ p=0.05} = 0.23$, we must reject the null hypothesis, and conclude that there is a negative correlation between statement coverage and residual bug density at the 5% significance level. Since $| r | < r_{crit,\ p=0.01} = 0.30$, we cannot reject the null hypothesis at the 1% significance level.

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.6. Since $| r_{rank} | = 0.12 < r_{crit,\ p=0.05} = 0.23 < r_{crit,\ p=0.01} = 0.30$, we cannot reject the null hypothesis, concluding that there is no clear correlation between statement and residual bug density at the 5% significance level.

Table 4.6: Spearman's r_{rank} of Statement vs. Bug Density

n	74
Spearman's r_{rank} statistic	-0.12
P(significance level)	0.01 or 0.05

Similar to the significance test of Pearson's r for hypothesis 1a, the Pearson's r_{rank} shows that there is close to moderate association between statement code coverage and residual defect density.

4.1.3 Hypothesis 1c

Null: The correlation between conditional code coverage and residual bug density in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between conditional code coverage and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is given in Table 4.7. Here, the Pearson's $r = -0.14$, and $| r | < r_{crit,\ p=0.05} = 0.24$. Thus, we cannot reject the null hypothesis at the 5% significance level. However, according to Table 3.7, $| r | = 0.14$ can be interpreted as a weak correlation between them.

Table 4.7: Pearson's r of Conditional vs. Bug Density

n	66
Pearson's r statistic	-0.14
P(significance level)	0.01 or 0.05

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.8. Since $| r_{rank} | = 0.11 < r_{crit,\ p=0.05} = 0.24$, we cannot reject the null hypothesis, concluding that there is no clear correlation between conditional coverage and residual bug density at the 5% significance level.

Table 4.8: Spearman's r_{rank} of Conditional vs. Bug Density

n	66
Spearman's r_{rank} statistic	-0.11
P(significance level)	0.01 or 0.05

Based on both correlation tests, we can conclude that there is an unclear/weak correlation between conditional code coverage and residual defect density.

4.1.4 Hypothesis 1d

Null: The correlation between method code coverage and residual bug density in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between method code coverage and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is presented in Table 4.9. Here, the Pearson's $r = -0.35$. Since $| r | > r_{crit.\ p=0.01} = 0.30$, we should reject the null hypothesis at the 1% significance level. Moreover, according to Table 3.7, $| r | = 0.35$ can be interpreted as a moderate correlation between them.

Table 4.9: Pearson's r of Method vs. Bug Density

n	73
Pearson's r statistic	-0.35
P(significance level)	0.01 or 0.05

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.10. Since $| r_{rank} | = 0.07 < r_{crit,\ p=0.05} = 0.24$, we cannot reject the null hypothesis, concluding that there is no clear correlation between method coverage and residual bug density at the 5% significance level.

Table 4.10: Spearman's r_{rank} of Method vs. Bug Density

n	73
Spearman's r_{rank} statistic	-0.07
P(significance level)	0.01 or 0.05

As we stated that when there is conflict results between Pearson's r and Spearman's r_{rank},, we choose the result of Pearson's r. We can conclude that there is a correlation between method code coverage and residual defect density, and the size of association is moderate.

4.1.5 Conclusion

The above tests do not support a strong linear association between code coverage and residual bug density, which was shown or suggested by some prior works [12,16]. However, our empirical tests showed that there are close to moderate linear associations between them. This suggests that higher code coverage, such as overall, statement and method may help to eliminate residual defect density, though this effect is not strong.

Is this in conflict with the result that only 3.28% of residual bugs (Category 4.1) can be detected by improving code coverage (detailed in section 3.4)? No. The reason is "Unit tests force clarification of class behavior and may lead to class redesign." [15]. Based on our root cause analysis, over 58 % of the residual defects are requirements and design problems which cannot be detected by only improving code coverage. However, from my experience and prior study [15], when developers try to improve code coverage of a class, they may make an effort to clarify the class's features and review the class's design and therefore they improve code coverage and also detect/eliminate some design bugs. This can explain how higher code coverage also results in fewer features/design bugs.

Furthermore, we observed that both correlation tests suggest that conditional coverage has no or very weak linear correlation with residual defect density. If this is verified in future works, after conditional coverage is improved to a bottom value, developers can ignore conditional coverage and concentrate on improving other coverage metrics, such as statement and method, which are easier to improve than conditional coverage. Therefore, in this way, testing efforts/costs can be reduced.

Our results have one limitation: our collected average code coverage is relatively high compared with that of other prior works. The average overall, statement, conditional, and method coverages are 79.53, 81.4, 61.8, and 85.6 respectively. This difference may mean that defects which can easily be detected by improving code coverage were already detected before our starting date of: 2003/01/01. Future study can collect code coverage and residual defects from an earlier date to verify our results---from a relative high coverage level, there is no strong linear correlation between code coverage and residual defect density.

4.2 Hypotheses about program size

In the software engineering area, there is a heated debate about the association between program size and residual defect density. Basili [26] presented data to support that there is a higher error rate in smaller sized modules. Fenton et al [28] indicated that there is no evidence from their empirical study that module size has a significant impact on fault density. They both gave some tentative explanations for the result of higher error rate in smaller sized modules:

- Larger modules are coded with more care than smaller modules because of their size;
- Errors in smaller modules are more apparent;
- There may still be numerous undetected errors present within the larger modules since all the "paths" within the larger modules may not have been fully exercised;
- The error rate may result from other explanatory factors, most notably, testing effort and operational usage.

We hope to provide some evidence to support their results and explanations. In the Swurv project, we don't have available strong data to quantify design, implementation, testing effort or operational usage. However, indirectly, code coverage may tell us about testing effort, and the number of revisions can tell us about implementation and refactoring effort. Therefore, the following 3 related hypotheses were tested:

Hypothesis 2a: Program size can be used to predict residual defect density.

Hypothesis 2b: Larger programs have higher code coverage.

Hypothesis 2c: Larger programs are implemented with more revisions.

4.2.1 Hypothesis 2a

Null: The correlation between class size (Ncloc) and residual bug density in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between class size (Ncloc) and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.11. As Pearson's r = -0.42, and $|r| > r_{crit,\ p=0.01} = 0.30$, we must reject the null hypothesis, and conclude that there is a negative correlation between class size (Ncloc) and residual bug density at the 1% significance level. Based on Table 3.7, the size of the association is moderate.

Table 4.11: Pearson's r of Ncloc vs. Bug Density

n	74
Pearson's r statistic	-0.42
P(significance level)	0.01 or 0.05

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.12. Since $|r_{rank}| = 0.85 > r_{crit,\ p=0.01} = 0.30$, we must reject the null hypothesis, concluding that there is a negative correlation between class size (Ncloc) and residual bug density at the 1% significance level. Based on Table 3.7, the strength of association is 'very strong'. Their sample relationship was plotted (See Figure 4.2).

Table 4.12: Spearman's r_{rank} of Ncloc vs. Bug Density

n	74
Spearman's r_{rank} statistic	-0.85
P(significance level)	0.01 or 0.05

Combining the results of both correlation tests, our empirical result shows that there is a moderate to strong negative association between program size and residual defect density. It means, in our collected data, larger classes have smaller defect density.

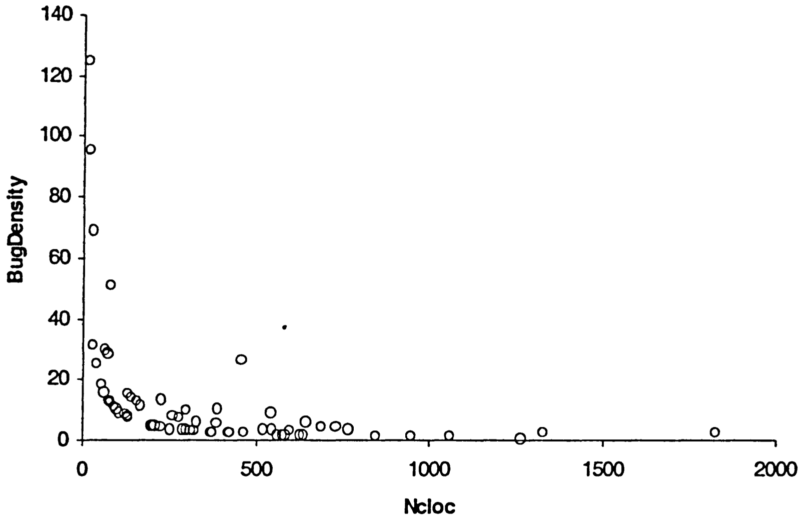

Figure 4.2: Sample Relationship Between Program Size and Residual Bug Density

4.2.2 Hypothesis 2b

Null: The correlation between class size (Ncloc) and code coverage in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between class size (Ncloc) and code coverage in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.13.

Table 4.13: Pearson's r of Code Coverage vs. Ncloc

Variable pair	Pearson's r (p=0.01)	n	Critical value (p=0.01)	Critical value (p=0.05)
Ncloc vs. Overall	-0.02	207	0.18	0.135
Ncloc vs. Statement	-0.02	207	0.18	0.135
Ncloc vs. Conditional	0.24	199	0.181	0.138
Ncloc vs. Method	-0.05	206	0.18	0.135

We can see the values of Pearson's r between class size and most code coverage (overall, statement and method) are really small, and their absolute values are smaller than $r_{crit,\ p=0.05}$ = 0.135. Only Pearson's r between class size and conditional coverage, r = 0.24, is higher than the critical value at p=0.01. Therefore, we must reject the null hypothesis for conditional coverage, and accept the null hypotheses for other code coverage. In other words, we can conclude that there is a weak to moderate positive correlation between class size and conditional coverage and no linear correlation between class size and overall, statement or method coverage.

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.14.

Table 4.14: Spearman's r_{rank} Code Coverage vs. Ncloc

Variable pair	Pearson's r (p=0.01)	n	Critical value (p=0.01)	Critical value (p=0.05)
Ncloc vs. Overall	0.11	207	0.18	0.135
Ncloc vs. Statement	0.09	207	0.18	0.135
Ncloc vs. Conditional	0.13	199	0.181	0.138
Ncloc vs. Method	-0.04	206	0.18	0.135

Since all of | r_{rank} | (absolute values) are smaller than $r_{crit,\ p=0.05}$ = 0.135, we cannot reject the null hypothesis, concluding that there is no correlation between class size (Ncloc) and residual bug density at the 5% significance level. However, we also observe that r_{rank} between conditional code coverage and residual defect density (r_{rank} = 0.13) is very close to $r_{crit,\ p=0.05}$ = 0.138. Thus, we can see the significance test of Spearman's r_{rank} agrees well with Pearson' r test above: Only conditional code coverage has weak correlation with class size. In general, conditional coverage is harder to improve than other code coverage. Hence, our results give only weak support to the idea that more testing efforts have been made for larger classes.

From both results of correlation tests, we can safely conclude that for the Swurv project, we cannot find clear evidence that larger classes have higher overall/statement/method code coverage. However, we observe that larger classes have a little higher conditional coverage.

4.2.3 Hypothesis 2c

Null: The correlation between class size (Ncloc) and the number of the class's revisions in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between class size (Ncloc) and the number of the class's revisions in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.15, and the sample relationship was plotted (See Figure 4.3).

Table 4.15: Pearson's r of Ncloc vs. Revisions

n	207
Pearson's r statistic	0.76
P(significance level)	0.01 or 0.05

The Pearson's r = 0.76, which is larger than $r_{crit, p=0.01} = 0.18$. Therefore, we must reject the null hypothesis and conclude that there is a very strong positive linear correlation between class size and the number of the class's revisions.

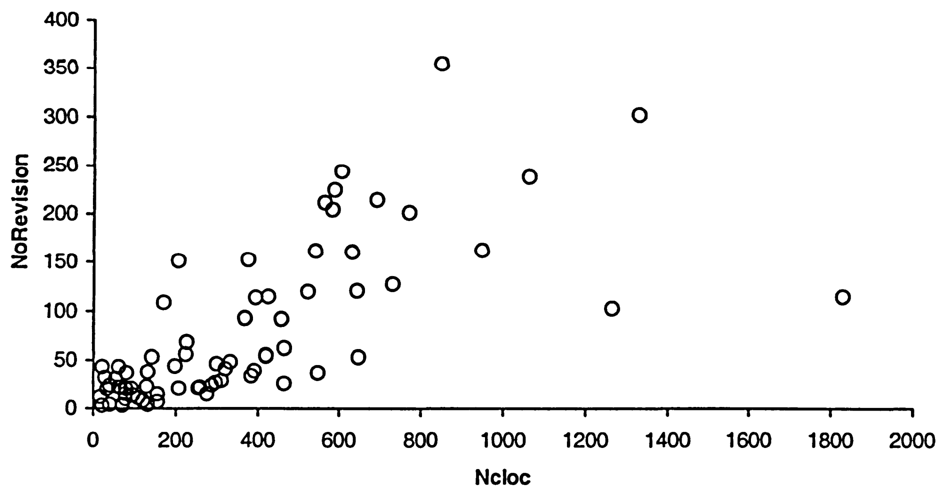

Figure 4.3: Sample Relationship Between Ncloc and Revisions

(2) The significance test of Spearman's r_{rank}

The result is listed in the Table 4.16. Since $r_{rank} > r_{crit,\ p=0.01} = 0.30$, we must reject the null hypothesis, and conclude that there is a very strong positive linear correlation between class size and the number of the class's revisions.

Table 4.16: Spearman's r_{rank} of Ncloc vs. Revisions

n	74
Pearson's r statistic	0.78
P(significance level)	0.01 or 0.05

In conclusion, both significance correlation tests indicate that there is a very strong linear association between a class size and the number of its revisions. The result indirectly indicates that larger programs have been implemented with more effort under the assumption more revisions stand for more effort of implementation and refactoring.

4.2.4 Conclusion

In this section, we have presented the evidence to support the hypothesis that program size has a strong linear correlation with residual defect density. It suggests that larger classes have smaller residual defect density. Should we then propose that developers should build large classes instead of several smaller classes? Definitely not. In our study, we also observed that non-bug classes have the smallest average class size.

The conflicting results lead us to suppose that the association between size and fault density is not a causal one. Perhaps that smallest classes have no bugs is because errors in smaller modules are more apparent. Thus, all of the bugs in smaller classes have been detected before the start of our data collecting period: 2003/01/01.

Perhaps that larger classes have smaller residual defect density is due to indirect influences by other factors. Our tests do not present a clear proof. However, our data indicates that there is a possibility that larger programs are designed and tested with more effort based on tests of hypothesis 2b: Larger programs have higher conditional code coverage, and hypothesis 2c: Larger programs are implemented with more revisions.

Though higher conditional code coverage does not necessarily stand for more testing effort, these two results suggest the association between class size and residual defect density may be due to other factors, such as testing effort, or refactoring and implementation effort. Therefore, we suggest researchers collect related data to quantify these factors in the future.

4.3 Hypothesis about class age

Traditional software development experience leads us to believe that, in a project, classes developed earlier should be more reliable than newly developed classes. The reason is programmers and testers must have tested older classes longer than newer classes, and older classes are more stable. Is this true in XP projects? In order to answer this question, I tested the following hypothesis:

Hypothesis 3a: Older classes have smaller residual bug density than newer classes.

4.3.1 Hypothesis 3a

Null: The correlation between class age and residual bug density in the population equals 0.0.

H_0: $\rho = 0$

Research: The correlation between class age and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.17.

Table 4.17: Pearson's r of Class Age vs. Bug Density

n	74
Pearson's r statistic	-0.21
P(significance level)	0.01 or 0.05

As Pearson's r = -0.21, and $| r | < r_{crit,\ p=0.05} = 0.23$, we cannot reject the null hypothesis, and conclude that there is no clear correlation between class age and residual bug density at the 5% significance level. However, the value of Pearson's r (-0.21) is very close to the critical value. Furthermore, from Table3.7, the strength of association can be explained as weak. Thus, we conclude that they may have a weak correlation.

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.18. Since r_{rank} =-0.38, $| r_{rank} |$ = 0.38> $r_{crit,\ p=0.01}$ = 0.30, we must reject the null hypothesis, concluding that there is a negative correlation between class size (Ncloc) and residual bug density at the 1% significance level. Based on Table 3.7, the strength of association is 'moderate'.

Table 4.18: Spearman's r_{rank} of Class Age vs Bug Density

n	74
Spearman's r_{rank} statistic	-0.38
P(significance level)	0.01 or 0.05

In short, from the two correlation tests, there is a weak to moderate negative correlation between class age and bug density. That is, we have weak to moderate evidence to support hypothesis 3a: older classes have smaller residual bug density than newer classes.

4.3.2 Conclusion

The result that there is no strong correlation between class age and bug density is expected. The reason is that in Extreme Programming (XP) projects, older classes are modified almost as often as the new classes.

The XP rule of simple design (always use the simplest possible design that gets the job done) means that developers may frequently add new functions into older classes. When developers create a new class, they write only the part necessary to implement the functions of the current story/iteration. After a new story is published, they can add new functions into older classes without limitation. In addition, the rule of refactoring results in modification of classes. XP requires developers often refactor out any duplicate code generated in a coding session. So, the older classes in XP projects are modified more than those in traditional projects. We cannot 'trust' old classes more than new classes as in traditional projects. Therefore, we have to test old classes as much as new classes. That is why XP requires the rule of continuous testing: XP teams have to make all test cases of the software work at all times. Moreover, these tests (Unit tests and Acceptance tests) should be automated so that it is possible to run them daily.

4.4 Hypothesis about the number of revisions

In this study, the number of revisions was collected at the class level, that is, how many revisions a class had before we collected the defects. A class's number of revisions includes the number of refactorings, the number of bug-fixes and the number of times new functions were added. Refactoring is an XP discipline for restructuring an existing body of code, altering its internal structure without changing its external behavior. We are concerned with refactoring's effects on software quality. However, the number of refactorings is not available in the Swurv project. We are hoping to get some idea of that through the analysis of correlation between the number of revisions and residual bug density. Therefore, we tested the following hypotheses:

Hypothesis 4a: The number of revisions has a negative association with residual bug density.

Hypothesis 4b: The number of revisions has a correlation with code coverage.

4.4.1 Hypothesis 4a

Null: The correlation between the number of revisions and residual bug density in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between the number of revisions and residual bug density in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The result is listed in Table 4.19. As Pearson's r = -0.32, and $| r | > r_{crit, p=0.01} = 0.30$, we must reject the null hypothesis, and conclude that there is a negative correlation between the number of revisions and residual bug density at the 1% significance level. From Table 3.7, the strength of association can be explained as "moderate". Thus, we conclude that they have a moderate negative correlation.

Table 4.19: Pearson's r of the number of Revisions vs. Bug Density

n	74
Pearson's r statistic	-0.32
P(significance level)	0.01 or 0.05

(2) The significance test of Spearman's r_{rank}

The result is listed in Table 4.20. Since r_{rank} =-0.64, $| r_{rank} | = 0.64 > r_{crit, p=0.01} = 0.30$, we must reject the null hypothesis, concluding that there is a negative correlation between the number of revisions and residual bug density at the 1% significance level. Based on the Table 3.7, the strength of association is "strong".

Table 4.20: Spearman's r_{rank} of the number of Revisions vs Bug Density

n	74
Spearman's r_{rank} statistic	-0.64
P(significance level)	0.01 or 0.05

In conclusion, the two tests convey that there is a moderate to strong negative correlation between the number of revisions and residual bug density. In other words, a class having more revisions has a smaller bug density. The sample correlation plot is given in Figure 4.4.

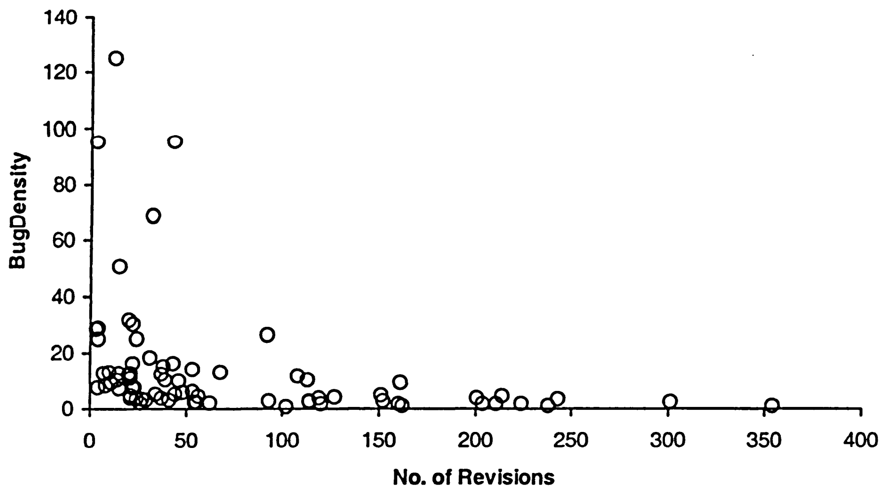

Figure 4.4: Sample relation Between No. of Revisions and Bug Density

4.4.2 Hypothesis 4b

Null: The correlation between the number of revisions and code coverage in the population equals 0.0. H_0: $\rho = 0$

Research: The correlation between the number of revisions and code coverage in the population does not equal 0.0. H_1: $\rho \neq 0$

(1) The significance test of Pearson's r

The results of Pearson's r between revisions and code coverage are listed in Table 4.21. We can see that only the Pearson's r of revisions vs. conditional is bigger than the critical values $r_{crit,\ p=0.01} = 0.181$ and $r_{crit,\ p=0.05} = 0.138$. Other Pearson's rs are smaller than the critical value at the 0.05 significance level. Thus, for conditional coverage, we must reject the null hypothesis, and conclude that there is a positive correlation between the number of revisions and conditional coverage at the 1% significance level. From Table 3.7, the strength of association can be explained as "weak". However, for other code coverage, we cannot find a correlation with the number of revisions.

Table 4.21: Pearson's r of Revisions vs Code Coverage

Variable pair	Pearson's r (p=0.01)	n	Critical value (p=0.01)	Critical value (p=0.05)

Revisions vs. Overall	0.06	207	0.18	0.135
Revisions vs. Statement	0.06	207	0.18	0.135
Revisions vs. Conditional	0.22	199	0.181	0.138
Revisions vs. Method	0.0	206	0.18	0.135

(2) The significance test of Spearman's r_{rank}

The results of Spearman's r_{rank} of Revisions vs. code coverage are listed in the Table 4.22. From this table, all r_{rank} are smaller than their corresponding critical values at the 5% significance level, so we cannot reject the null hypothesis, concluding that there is no clear correlation between the number of revisions and code coverage at the 5% significance level.

Table 4.22: Spearman's r_{rank} of Revisions vs. Code Coverage

Variable pair	Spearman's r_{rank} (p=0.01)	n	Critical value (p=0.01)	Critical value (p=0.05)
Revisions vs. Overall	0.13	74	0.30	0.23
Revisions vs. Statement	0.06	74	0.30	0.23
Revisions vs. Conditional	0.23	66	0.31	0.24
Revisions vs. Method	-0.16	206	0.18	0.135

It seems that there is a difference for conditional coverage in the above two tests. However, we notice that the Spearman's r_{rank} between conditional coverage and the number of revisions ($r_{rank} = 0.23$) is very close to its $r_{crit,\ p=0.05} = 0.24$; furthermore, the Spearman's r_{rank} does not consider non-bug classes due to its assumption that sample size should be less than 100. Therefore, we accept the result of Pearson's test that there is a weak positive correlation between conditional coverage and the number of revisions, and no clear correlation between other code coverage and the number of revisions.

4.4.3 Conclusion

The evidence we found in support of Hypothesis 4a (the number of revisions has a negative association with residual bug density) is relatively strong. It indicates that more revisions may result in a higher quality of software products. Is this because more revisions lead to higher code coverage? From the testing Hypothesis 4b, we observed the number of revisions has a weak correlation with code coverage.

As we stated above, fixing detected bugs, refactoring and adding new functions will increase the number of revisions. Which is the real causal reason of improved quality? We can't find an answer from our study due to lack of available data, such as the number of refactorings and the number of bug fixes for each class. Therefore, we suggest that in future research projects, developers record each submission's reason: adding new functions, fixing bugs or refactoring.

If a new revision is for adding new functions, it may not improve the class's quality. However, if the new revision is for fixing a bug, the class's quality can be improved since the number of bugs in the class has been reduced; if the new revision is for refactoring, it may make the program easier to understand so that the bugs just leap out, or are less likely to be introduced in the future.

The most interesting data should be the number of refactorings at the class level. In reality, due to tight project schedules, the project manager may not arrange time for developers to do refactoring. If other researchers do find a strong correlation between the number of refactorings and residual defect density, developers and program managers will be encouraged to do more refactoring to improve the quality of their projects.

4.5 Summary of Associations

Based on our significance tests of Pearson's r and Spearman's r_{rank}, we found that several variable pairs have moderate or strong associations (see Table 4.23). The further conclusion is presented in Chapter 6.

Table 4.23: Variable Pairs with Moderate/Strong Association

Variable Pair	Strength of Correlation	Direction
BugDensity vs. Overall	Close to moderate	Negative
BugDensity vs. Statement	Close to moderate	Negative
BugDensity vs. Methods	Moderate	Negative
Revisions vs. Ncloc	Strong	Positive
BugDensity vs. Revisions	Strong	Negative
BugDensity vs. Ncloc	Strong	Negative

Chapter 5
Defect Distribution Analysis and Improvement Actions

Through defect root cause analysis, we obtained Swurv's statistical distribution of residual defect root causes shown in Table 3.13. From the statistical distribution, we can derive several general results:

- Bugs in component design and implementation consume 70.49% of all defects;
- Only 3.28% of all bugs (Category 4.1) can be detected by improving code coverage;

In this chapter, further discussion of the above results is presented and, furthermore, appropriate improvement actions for the Swurv project are proposed.

5.1 Defect Distribution Analysis

5.1.1 Category 4--- Component Design and Implementation Bugs

From Table 3.13, we can see that bugs of category 4--- Component Design and Implementation Bugs--- represent 70.49% of Swurv bugs. In other words, defects are injected into the system predominantly (70.49%) during the component design and implementation phase. This result is very similar with the result presented in reference [24] "A Case Study in Root Cause Defect Analysis". In their project, 71% of bugs originated from the component spec. / design and implementation phase. However, their defects include bugs detected by unit test space[24]; in my study, the bugs were detected only by system tests (after unit tests). So, our defect statistics do not include defects detected by unit tests. It is reported that unit tests are efficient in detecting defects of functionality [9,56], which refers to missing or wrong functionality (w.r.t. requirements) and mostly are injected in building phase [24]. Therefore, if we collected defects detected by unit tests, the rate of component design and implementation bugs would probably be more than 71%.

Moreover, in Beizer's project [20], over 57% bugs originated from the component spec/design and implementation phase (structural, data bugs, implementation and coding faults). Therefore, comparatively speaking, the number of Swurv bugs occurring in the component design and implementation phase is relatively high and we should propose improvement methods to reduce bugs in this category.

5.1.2 Why improving code coverage cannot help detect residual bugs?

An interesting outcome of the analysis is the observation that only 3.28% of residual bugs (Category 4.1) can be detected by improving code coverage. It means even if a project's code coverage is improved to 100%, it is not likely to detect over 96% of the residual defects. This result is supported by Frankl's study [10]. Frankl mentioned that in most subjects (programs), even the high coverage level test sets were not terribly likely to detect faults. This result leads me to analyze why high code coverage cannot help to detect residual bugs for each category.

Category 1,2,3---missing/incorrect requirements/features

Since code to implement missing functions does not exist, developers probably will not diagnose that there are missing/incorrect features by executing/covering all the existing program structural elements, such as statements, conditionals and methods. Thus, these bugs would not be detected by improving code coverage. However, they could be eliminated by design review and acceptance tests before implementation.

Category 4.2.1--- Component Design Problem

Bugs of category 4.2.1 include inadequate/wrong features (w.r.t requirements), algorithm, performance, etc. They represent 29.51% of all the studied bugs and 43.9% of the component design and implementation bugs that may not be detected by improving code coverage. This rate is relatively large, so I divided it into the following types:

4.2.1.1: Deadlock, multi-thread coordination: 2 (Bug ID: 553, 621)

4.2.1.2: Used wrong API/java package: 1 (Bug ID: 595)

> This is due to lack of experience. Bug 595 is a result of using a timezone insensitive java package to get the email's sent time. Unit tests cannot obtain correct sent time to validate if the time is correct. Developers fixed this bug by using a timezone sensitive package.

4.2.1.3: Missed code for some special cases/exceptions: 12 (Bug ID: 492, 510, 569, 516,518,539,547,548,649,582,587,608)

4.2.1.4: Inheritance problem: 2 (584,585)

Developers inherited existing parent class's methods to implement some functions for a subclass, so they didn't implement new methods for the subclass. However, they did not consider differences in handling different class objects.

From the data, we observe that missing code for some special cases/exceptions represents 67% of root causes of component design problem bugs. For these bugs, effective design reviews or software inspections can detect these missing of cases.

Category 4.2.2---bugs occurred in long-event functions

Even if the related codes were all covered by unit tests, the bug was not detected. The reason is the long-event functions require several events to be performed continuously and sequentially. The latter event may check/use the states resulting from the former event's execution. But most unit tests will reset states after executing an event, and some unit test tools do not support continuous testing of several events.

For example, for bug 609, the bug was described in Bugzilla as "once you step out of a room (go to email and back to chat) clicking on a user name from before you left and whispering (kicking out, adding to a chat room), may not work, or may do the operation on another user." In order to test the above case, we have to keep all the states while continuously executing the following six events: enter a chat room, whisper with a user, step out of a room, go back to the former chat room, click on the user name to whom you whispered, whisper to him/her.

Acceptance/ Functional tests are useful in testing long event functions. Several web functional testing tools, such as HttpUnit, can be used to test long-event functions for web-based applications.

Category 4.2.3--- lack of verification

As for bugs in category 4.2.3, related code was covered but the unit tests lacked verification to check if the executed value is as expected. As we know, a programmer's coding error, such as a logic error, boundary error, etc, cannot be avoided completely. Thus, appropriate verification is necessary for important objects/variables. However, in the Swurv project, programmers didn't write enough verification in their unit tests. In some cases, even though codes involved in a bug were executed, unit tests still could not detect the bug.

Thus, we can conclude if the unit test relies on only structural coverage, it may not detect some bugs. Tests need to be designed more carefully and formally so that they will find and check all important values.

Category 4.2.4---GUI bugs

The bugs in category 4.2.4 occurred in HTML/Javascript codes. The Swurv unit tests will not test HTML/Javascript codes. Therefore, these bugs may not get detected by improving code coverage. This is why GUI bugs consume 29.51% of all the studied bugs. Then, how do we eliminate GUI bugs? From prior work, automated acceptance tests and good design may help.

Automated GUI testing is difficult. Lisa Crispin [36] proposed several guidelines for acceptance tests for web applications. She suggested verifying the minimum key criteria for success and doing the simplest thing that works. A few GUI tests are better than none. In this way, we can eliminate some GUI bugs.

Good architecture design can help GUI tests too. In reference [36], Lisa Crispin mentioned, "we carefully control our JavaScript libraries to minimize changes and thus the required retesting." Finsterwalder [39] said, "As most parts of our application are designed so that no business logic is mixed with the GUI handling code, the verification that the thin GUI layer is working well can often be done manually."

Our analysis thus far indicates that the defects of the following cases are probably not detected by improving code coverage: missing/incorrect specification/features/cases, component design problem (problems of correctness and completeness with respect to the requirements, internal consistency, performance, O-O design problems, etc) long-event functions, lack of verification and GUI bugs (Swurv automatic unit tests do not verify the GUI). Above analysis pointed out that important areas to look for improvements are good architecture design, design reviews, and acceptance testing. We will further discuss them below.

5.2 Improvement Actions

The Swurv project applied the Extreme Programming (XP) methodology from the beginning of the project. However, we observed that quite a few bugs existed after the delivery to the customer. Our goal is to find and suggest an effective set of XP improvement actions, so that for future XP projects:

- the overall number of defects is significantly reduced
- the defects are avoided and detected earlier in the lifecycle
- the modified set of XP rules are still a lightweight methodology, and will not add burden to the programmers and the customer.

Based on the statistical distribution of defect root cause, the most important task is to reduce in the areas of bugs: Component design and GUI. Therefore, we advocate several improvement actions for design, component implementation and acceptance tests.

5.2.1 Design up front

XP advocates the rule of "simple design"----design the simplest thing that will work now. The motivation for this is that requirements will invariably change, so most of a "big up-front design" ends up being thrown away, and a simple design is easy to adapt to requirements changes.

Design up-front in XP usually involves a system metaphor and CRC cards. The system metaphor is used to replace architecture design and describes the basic elements and relationships of the application [32]. The system metaphor does not seem to address bigger architectural issues such as overall architecture, classic design patterns, e.g. Model-View-Control pattern, etc.

As we know, decisions made during the architectural and design phase of the application development lifecycle have a significant impact on the software product's long-term performance and scalability. Furthermore, upfront architectural design reduces the development time for new functionalities [32]. With a good upfront architectural design, each new functionality is based on an infrastructure backbone and design patterns. With this approach, the time and effort needed for implementing new functionalities is much reduced when compared to developing from scratch.

The Swurv project is a complex web-based application and it does not separate presentation, control and business logic. GUI handling codes are mixed with business processing logic. It is proved in many web applications that if we can use some design model to separate business logic from GUI handling code, the number of GUI bugs will be efficiently reduced [36,39]. Good design patterns, such as MVC, make code easy to reuse, efficiently improve project velocity and detect bugs. Especially, for performance problems, good architecture strategies can prevent and solve them. However, architecture design requires significant planning. Compared to the cost of detecting and fixing bugs, having a good architecture design is cost-effective. Therefore, in order to reduce the overall number of defects, especially GUI bugs, and performance problems, we propose:

IP1: For complex projects, like the Swurv project, a big upfront architectural design is necessary.

In XP, CRC cards are used to replace high level design. CRC design sessions are suggested to last 10-30 minutes and allow the entire project team to contribute to the design. It is considered that the more people who can help design the system the greater the number of good ideas incorporated. However, in reality, people argue that an experienced senior designer/analyst can do a better design than a group of programmers. Large complex design requires strong experience. Considering 29.51% of the studied bugs are component design problems, in order to reduce the number of component design problems, we propose:

IP2: Important/common functions should be designed by an experienced designer/analyst.

5.2.2 Component implementation

In the Swurv project, missing handlings for some important cases/exceptions represent 67% of the bugs of the component design problem. After defect root cause analysis, we discovered most of them are a result of lack of experience. Thus, though peer inspection is provided in XP projects by pair programming, inspections by a senior designer/analyst/tutor are still necessary. We propose:

IP3: The designer/analyst/coach should frequently review/inspect programmers' artifacts.

Based on prior work [15] and the results of Chapter 4 (see section 4.1.5), unit tests can not only reduce the number of defects, but improve the design as well. Also, from our defect analysis, 3.28% residual bugs are due to lack of verification. Thus, we suggest:

IP4: Developers should better unit tests so that unit tests have higher test coverage of unit functional and code coverage and complete verification for important values.

5.2.3 Acceptance testing

Acceptance tests can enhance the communication between customers and programmers [38]. In the Swurv project, considering that automating acceptance tests is difficult, the customer did manual acceptance tests after programmers delivered the finished functions to him. From the residual defect distribution shown in Table 3.13, we found 23% of all residual defects resulted from programmers' missing/misunderstanding common cases/features. If acceptance tests are defined at the beginning of each iteration, programmers will clearly grasp the use cases they should handle, and therefore efficiently eliminate this kind of bug.

XP rules require that acceptance tests should be automated so they can be run often to test changes that affect many objects. Also, automated acceptance tests can help to test long-event functions that unit tests cannot detect. Furthermore, from my experience, automated acceptance tests can help to detect some GUI bugs by using some testing tools, like HttpUnit. For testing the GUI part in the acceptance tests, somehow you will have to emulate human interaction to get the system's response---- complicated textual output (e.g. an HTML report). Then scripts can test if the links/functions work, if the complicated *textual* output (e.g. an HTML report) is correct by simply comparing any captured output with expected output. However, it is difficult for the on-site customer to write automated acceptance tests. So we suggested:

IP5: It is necessary to add a tester to help customer define acceptance tests at the beginning of every iteration.

IP6: While implementing the new iteration, the tester should write automatic acceptance tests.

5.3 Summary of Improvement Actions

Based on our defect distribution analysis, six improvement actions for the Swurv project were derived in this Chapter as follows:

IP1: For complex projects, like the Swurv project, a big upfront architectural design is necessary.

IP2: Important/common functions should be designed by an experienced designer/analyst.

IP3: The designer/analyst/ should frequently review/inspect programmers' artifacts.

IP4: Developers should better unit tests so that unit tests have higher test coverage of unit functional and code coverage and complete verification for important values.

IP5: It is necessary to add a tester to help customer define acceptance tests at the beginning of each iteration.

IP6: While implementing the new iteration, the tester should write automatic acceptance tests.

The proposed improvement actions require adding an experienced analyst and a tester to the XP team. The experienced analyst is responsible for architecture design and the important functions' design, and also she/he needs to discuss/review/inspect developer's artifacts frequently; the tester can aid customers in defining and automating acceptance tests as well as flush out hidden requirements and assumptions. If such an analyst and a tester existed in the XP team, we may prevent and effectively eliminate most of the Swurv's residual defects due to missing/misunderstanding features, GUI bugs, component design and implementation problems, etc. Furthermore, the proposed improvement actions will not add burdens to customers or developers.

Chapter 6
Conclusions and Future Work

The correlation-based results we have presented (which are summarized in Table 6.1) are based on the Swurv project.

Table 6.1: Support for the Hypothesis provided

Number	Hypothesis	Strength of Correlation	Direction
1a	Overall code coverage can be used to estimate the residual defect density	Close to moderate	Negative
1b	Statement code coverage can be used to estimate the residual defect density	Close to moderate	Negative
1c	Conditional code coverage can be used to estimate the residual defect density	None	N/A
1d	Method code coverage can be used to estimate the residual defect density	Moderate	Negative
2a	Program size can be used to predict residual defect density.	Strong	Negative
2b	Larger programs have higher code coverage.	None for most, weak for conditional coverage	Positive
2c	Larger programs are implemented with more revisions.	Strong	Positive
3a	Older classes have smaller residual bug density than newer classes.	Weak to moderate	Negative
4a	The number of revisions has negative association with residual bug density	Moderate to strong	Negative
4b	The number of revisions has correlation with code coverage.	None for most, weak for conditional coverage	Positive

Since there are only 7-12 developers in this project, we cannot make claims about the generalization of these results. However, given the rigor and extensiveness of the data collection and also the strength of some of the observations, we feel that the following findings are helpful to the wider community:

- There is no strong (weak to moderate) linear correlation between code coverage and residual defect density when code coverage is beyond a certain range.

This result moderately supports that higher code coverage can lead to less residual defect density. As the association is not strong, it does not support that we can predict the residual defect density based on the code coverage as presented in prior work [12]. From this result, we indicate that improving code coverage is helpful to detect residual defects, but it is not enough, and we also need other testing, like acceptance testing, in the process of software development to provide good quality software products.

Our defects were collected while average code coverage of the Swurv classes was relatively high. In our study, the average overall, statement, conditional, and method coverages are 79.53, 81.4, 61.8, and 85.6 respectively. Our result agrees well with the result in reference [3]: "beyond a certain range (70% -80%.), increasing statement coverage becomes difficult and is not cost effective."

Thus, we can conclude that, beyond a certain range, code coverage cannot be used to predict the residual defect density.

- Compared with other kinds of code coverage in this thesis, conditional coverage has the weakest linear association with the residual defect density.

This indicates that increasing conditional coverage may be the least cost effective after conditional coverage is improved to a certain range (60%).

- There is a strong linear correlation between program size and residual defect density. It suggests that larger classes have smaller residual defect density. Also, we observed that non-bug classes have the smallest average class size. The conflicting results lead us to suppose that the association between size and fault density is not a causal one.

- No strong correlation exists between class age and residual bug density.

Earlier created classes may have the same residual defect density as newly created classes. Therefore, we have to test old classes as much as new classes. It is important to follow the XP rule of "continuous testing"---XP teams have to make sure all test cases of the software work at all times.

Since our study doesn't give strong support that classes with higher code coverage have fewer residual defects, we did defect root cause analysis on residual defects. We found for over 96% of residual defects, improving code coverage cannot detect them. In order to effectively reduce the number of residual defects for future XP projects, we propose six improvement actions to enhance design, component implementation and acceptance testing. They are:

IP1: For complex projects, like Swurv project, a big upfront architectural design is necessary.

IP2: Important/common functions should be designed by an experienced designer/analyst.

IP3: The designer/analyst/ should frequently review/inspect programmers' artifacts.

IP4: Developers should better unit tests so that unit tests have higher test coverage of unit functional and code coverage and complete verification for important values.

IP5: It is necessary to add a tester to help customer define acceptance tests at the beginning of every iteration.

IP6: While implementing the new iteration, the tester should write automatic acceptance tests.

As to the suggestions for future studies, it is hoped that further research in software engineering will be conducted to support our findings and propose more meaningful results. We suggest:

1. To collect residual defects and code coverage earlier when average code coverage is lower. In this way, researchers can investigate if there is a critical range for average code coverage. When unit tests reach the range of this study, there is no strong linear correlation between code coverage and residual defect density.
2. To investigate which factors, such as testing effort, number of refactorings and implementation effort, etc, lead to larger classes having smaller defect density. In order to do this study, researchers need to collect enough information. Thus, we recommend that when developers submit code, they should record appropriate information, such as implementation hours, testing hours, and refactoring hours.
3. To explore the correlation between the effort of refactoring and residual defect density. Our result indicates that more revisions may result in higher quality software products. Both fixing detected bugs and refactoring may increase the number of revisions with

software quality. If researchers do find a strong correlation between the number of refactorings and residual defect density, developers and project managers will be encouraged to follow the XP "refactoring" rule to improve the quality of their projects.

Appendix A
Definitions of Terms

Analyse-it: Analyse-it is an add-in to Microsoft Excel allowing one to analyse data straight from an Excel worksheet [52].

Bugzilla: Bugzilla is a database for bugs that lets people report bugs and assigns these bugs to the appropriate developers [63].

Clover: Clover is a code coverage tool that can obtain code coverage for classes, packages and project [21].

CVS: Concurrent Versions System. CVS is a powerful tool for software version control. It provides method of recording the history of your source files and allowing many developers to work on the same source code [67].

DRCA: Defect root causal analysis. It is the process of analyzing a defect to determine its root causes.

Swurv: Swurv company. It specializes in developing Internet technology products [48].

Bibliography

[1] Harish V. Kantamneni, Sanjay R. Pillai, Yashwant K. Malaiya, "Structurally Guided Black Box Testing," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 2002, pp. 137-138.

[2] Jim D.Creasman, Christopher J.Born, "Design, Implementation, and Case Study of a Function Level Unit Test Environment," in *1992 IEEE Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, 27-29 May 1992, pp. 286-296.

[3] Piwowarski, P.; Ohba, M.; Caruso, J., "Coverage measurement experience during function test," in *Proceedings of the 15th International Conference on Software Engineering*, 17-21 May 1993, pp. 287-301.

[4] James Ramsey, Victor R. Basili, "Analyzing the Test Process Using Structural Coverage," in *Proceedings of the 8th international conference on Software engineering*, Aug. 1985, pp. 306-312.

[5] Daniel Hoffman, Paul Strooper and Lee White, "Boundary Values and Automated Component Testing," *Journal of Software Testing, Verification, and Review*, vol. 9, pp. 3-26, 1999.

[6] T.W.Williams, M.R.Mercer, J.P.Mucha, and R.Kapur, "Code Coverage, What Does It Mean in Terms of Quality?" in *2001 IEEE Proceedings Annual Reliability and Maintainability Symposium*, 22-25 January 2001, pp. 420-424.

[7] Huo Yan Chen, T. H. Tse, F. T. Chan, T. Y. Chen, "In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs," *ACM Transactions on Software Engineering and Methodology*, vol. 7, No. 3, pp. 250-295, July 1998.

[8] Silke Kuball, Gordon Hughes, "Scenario-Based Unit Testing For Reliability," in *IEEE Proceedings Annual Reliability and Maintainability Symposium*, 2002, pp. 222-227

[9] Houman Younessi, Panlop Zeephongsekul, Winai Bodhisuwan, "A General Model of Unit Testing Efficacy," *Software Quality Journal*, vol. 10, pp. 69-92, 2002.

[10] Phyllis G. Frankl, Oleg Iakounenko, "Further Empirical Studies of Test Effectiveness," *ACM SIGSOFT Software Engineering Notes*, vol.23, pp.153-162, Nov. 1998.

[11] M. Chen, A. P. Mathur and V. J. Rego, "Effect of testing Techniques on software reliability estimates obtained using a time domain model," *IEEE Transactions On Reliability*, Vol. 44, No. 1, March 1995, pp. 97-103.

[12] Ran Ye and Yashwant K. Malaiya, "Relationship Between Test Effectiveness and Coverage," in *Proceedings of the thirteenth International Symposium on Software Reliability Engineering*, Nov. 12-15, 2002, pp. 159-160.

[13] Yashwant K. Malaiya, Michael Naixin Li, James M. Bieman, and Rick Karcich, "Software Reliability Growth With Test Coverage," *IEEE Transactions on Reliability*, vol. 51, Issue: 4, Dec. 2002, pp. 420-426.

[14] Edward Kit; *Software Testing in the Real World---improving the process*. Addison-Wesly, 1995.

[15] Brad Long and Paul Strooper, " A Case Study in Testing Distributed Systems," in *3rd International Symposium on Distributed Objects and Applications*, 17-20 Sept. 2001, pp. 20-29.

[16] Saileshwar Krishnamurthy, Aditya P. Mathur, "On Predicting Reliability of Modules Using Code Coverage," in *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative Research*, November 12-14, 1996, p. 22.

[17] Hong Zhu, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, No.4, December 1997, pp. 366-427.

[18] Mei-Hwa Chen; Mathur, A.P.; Rego, V.J., "Effect of testing techniques on software reliability estimates obtained using a time-domain model," *IEEE Transactions on Reliability*, vol. 44, Issue: 1, March 1995, pp. 97-103.

[19] Y.K. Malaiya, N. Li, J. Bieman, R. Karcich and B. Skibbe, "The Relationship between Test Coverage and Reliability," in *International Symposium on Software Reliability Engineering*, Nov. 1994, pp. 186-195.

[20] Boris Beizer, *Software Testing Techniques*. New York: John Wiley & Sons, 1990.

[21] Cenqua Pty Ltd., "Clover Frequently Asked Questions," 1999, http://www.thecortex.net/clover/FAQ.html.

[22] StatSoft Inc., "Basic Statistics," 1984,

http://www.statsoft.com/textbook/stbasic.html.

[23] Jerry C. Whitaker, "Software failures for fault-tolerant systems," *Reliability Engineering*, pp. 18-169, 2004,

http://www.tvhandbook.com/support/pdf_files/Chapter18_13.pdf

[24] Marek Leszak, Dewayne E. Perry and Dieter Stoll, "A Case Study in Root Cause Defect Analysis," in *Proceedings of the 22nd international conference on Software engineering*, June 2000, pp. 428-437.

[25] Card, D.N., "Learning from our mistakes with defect causal analysis," *IEEE Software*, vol. 15 Issue: 1, pp. 56-63, Jan.-Feb. 1998.

[26] Victor R. Basili and Barry T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 27, pp. 42-52, January 1984.

[27] Norman E. Fenton and Niclas Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE transaction on software engineering*, vol. 26, pp. 797-814, August 2000.

[28] Fenton, N.E. and Neil, M., "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, pp. 675-689, Sept.-Oct. 1999.

[29] Weider D. Yu, "A Software Fault Prevention Approach in Coding and Root Cause Analysis," *Bell Labs Technical Journal*, pp. 3-21, April - June 1998.

[30] Amr Elssamadisy, Gregory Schalliol, " Industry track papers and presentations: technology trends: Recognizing and responding to "bad smells" in extreme programming," in *Proceedings of the 24th international conference on Software engineering*, May 2002, pp. 617-622.

[31] Glass, R.L., "Extreme programming: the good, the bad, and the bottom line," *IEEE Software*, vol. 18, pp. 112-111, Nov.-Dec. 2001.

[32] Lan Cao; Mohan, K.; Peng Xu; Ramesh, B., "How extreme does extreme programming have to be adapting XP practices to large-scale projects," in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, 5-8 Jan. 2004, pp. 83-92.

[33] Nawrocki, J.; Jasinski, M.; Walter, B., "Wojciechowski, A.; Extreme programming modified: embrace requirements engineering practices," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering*, 9-13 Sept. 2002, pp. 303-310.

[34] Bernhard Rumpe, Astrid Schroeder, "Quantitative Survey on Extreme Programming Projects," in *Proceedings of XP2002*, 2002, pp. 95-100.

[35] Rasmussen, J., "Introducing XP into Greenfield Projects: lessons learned," *IEEE Software*, vol. 20, pp. 21-28, May-June 2003.

[36] Lisa Crispin, Carol Wade and Tip House, "The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment," *Software Testing Analysis & Review*, pp. 96-104, May 2001.

[37] John Brewer, Jera Design, "Extreme Programming FAQ," 2001,

http://www.jera.com/techinfo/xpfaq.html.

[38] Ron Jeffries, "Introduction to the Korean Edition," July 2002, http://www.xprogramming.com/xpmag/koreanintro.htm#N67.

[39] Malte Finsterwalder, "Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment," 2004,

http://www.agilealliance.org/articles/articles/AutomatingAcceptanceTests.pdf.

[40] Johan Anderson, Geoff Geoff, and Peter Sutton, "XP with Acceptance-Test Driven Development: A rewrite project for a resource optimization system," 2004, http://www.carmensystems.com/research_development/ articles/ctrt0302.pdf.

[41] Kulvir Singh Bhogal, "HttpUnit: A Civilized Way to Test Web Applications in WebSphere Studio," March, 2003,

http://www-
106.ibm.com/developerworks/websphere/library/techarticles/0303_bhogal/bhogal.html.

[42] Atif M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," Ph.D. dissertation, University of Pittsburgh, Pittsburgh, Pennsylvania, 2001.

[43] Bret Pettichord, "Seven Steps to Test Automation Success," presented at *STAR West*, San Jose, November 1999.

[44] Giora Katz-Lichtenstein, "Black Box Web Testing with HttpUnit," July 2003,

http://www.onjava.com/pub/a/onjava/2003/05/07/blackboxwebtest.html.

[45] Benoît Marchal, "Views from Abroad: Cactus -- No Pity for Bugs," 2004, http://www.developer.com/java/other/article.php/793701.

[46] Vincent Massol, "Subject: Mock Objects for servlets (was Re: Roadmap for Cactus 1.1)," 17 Jun 2001,

http://www.mail-archive.com/jakarta-commons@jakarta.apache.org/msg02227.html.

[47] Vincent Massol, "Mock Objects vs. In-Container testing," 2004, http://jakarta.apache.org/cactus/mock_vs_cactus.html

[48] Swurv holdings, "Swurv Project", 2004

http://www.swurv.com/

[49] William Curry, Giancarlo Succi, Michael Smith, Eric Liu and Raymond Wong, "Empirical Analysis of the Correlation between Amount-of-Reuse Metrics in the C Programming Language," in *Proceedings of the 1999 symposium on Software reusability*, May 1999, pp. 135-140

[50] Witold Pedrycz, Michael H. Smith, "Granular Correlation Analysis in Data Mining," in 1999 *IEEE International Fuzzy Systems Conference Proceedings*, 22-25 Aug. 1999, pp. 1235-1240

[51] Muneo Takahashi, Yuji Kamayachi, "An empirical study of a model for program error prediction," in *Proceedings of the 8th international conference on Software engineering*, August 1985, pp. 330-336.

[52] Analyse-it Ltd., "Analyse-it Home," 2004,

http://www.analyse-it.com/start.htm.

[53] James A. Rosenthal, *Statistics and Data Interpretation for the Helping Professions.* Wadsworth, 2001.

[54] R. M. Bittick, "Introduction to Regression Analysis," 2004, http://somc.csudh.edu/sp04_pub305/Correlation%20and%20Regression%20Intro1.ppt.

[55] Richard M. Karcich, Robert Skibbe, "On Software Reliability and Code Coverage," in *Proceedings of 1996 Aerospace Applications Conference*, vol. 4, Feb. 1996, pp. 297-308.

[56] Weider D. Yu, Alvin Barshefsky, and Steel T. Huang, "An Empirical Study of Software Faults Preventable at a Personal Level in a Very Large Software Development Environment," *Bell Labs Technical Journal*, pp. 221-232, Summer 1997.

[57] Charles M. Friel, Criminal Justice Center, Sam Houston State University, "Nonparametric Correlation Techniques," 2004,

http:// www.shsu.edu/~icc_cmf/cj_685/mod12.doc.

[58] Oxford University; "Geographical Techniques Online," 2004, http://techniques.geog.ox.ac.uk/mod_2/week_7/lecture-7.htm.

[59] John Wasson, "Statistics in Educational Research - An Internet Based Course," 2004, http://www.mnstate.edu/wasson/ed602lesson15.htm.

[60] Ron Jeffries, "Are we doing XP?" Nov. 2000, http://www.xprogramming.com/xpmag/are_we_doing_xp.htm.

[61] MockEJB website, 2004,

http://www.mockejb.org/.

[62] HttpUnit, ServletUnit website, 2004,

http://httpunit.sourceforge.net/.

[63] Bugzilla.com, "Bugzilla!" 2004,

http://www.bugzilla.com/.

[64] Rational Ltd., "Rational PureCoverage: What it does?" 2004,

http://engineering.arm.gov/base/pure/purecoverage/html/html/ht_intro_pc.htm.

[65] Quest Software Ltd., "JProbe® Suite Comprehensive Java Performance Tuning," 2004,

http://www.quest.com/jprobe/.

[66] Testworks Group, "Solutions in Assembly, Inspection and Test," 2004,

http://www.testworks.co.uk/.

[67] CollabNet Inc, "Welcome to CVS home!" 2003,

https://www.cvshome.org/.