

DISSERTATION

AN INTEGRATED METHOD FOR IMPROVING TESTING EFFECTIVENESS AND  
EFFICIENCY

Submitted by

Catherine V. Stringfellow

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2000

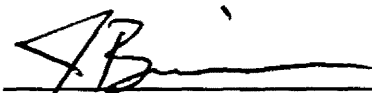


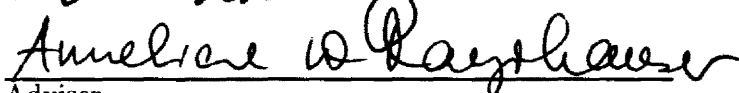

QA  
76.76  
.T48  
S77  
2000

COLORADO STATE UNIVERSITY

July 6, 2000

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY CATHERINE V. STRINGFELLOW ENTITLED AN INTEGRATED METHOD FOR IMPROVING TESTING EFFECTIVENESS AND EFFICIENCY BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

  
\_\_\_\_\_  
  
\_\_\_\_\_  
  
\_\_\_\_\_  
  
\_\_\_\_\_  
Adviser  
  
\_\_\_\_\_  
Department Head

QA  
76.76  
.T48  
S97  
2000

COLORADO STATE UNIVERSITY

July 6, 2000

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY CATHERINE V. STRINGFELLOW ENTITLED AN INTEGRATED METHOD FOR IMPROVING TESTING EFFECTIVENESS AND EFFICIENCY BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

[REDACTED]

---

[REDACTED]

---

[REDACTED]

---

[REDACTED]

---

Adviser

[REDACTED]

---

Department Head

## ABSTRACT OF DISSERTATION

### AN INTEGRATED METHOD FOR IMPROVING TESTING EFFECTIVENESS AND EFFICIENCY

The aim of testing is to find errors and to find them as early as possible. Specifically, system testing should uncover more errors before release to reduce the number of errors found in post-release. System testing should also prevent the release of products that would result in discovery of many post-release errors. Studies indicate that post-release errors cost more to fix than errors found earlier in the life cycle. The effectiveness and efficiency of system testing depends on many factors, not only the expertise and quality of the testers and the techniques they employ.

This dissertation develops an integrated method using various techniques that will improve testing effectiveness and efficiency. Some of these techniques already exist, but are applied in a new or different way.

The integrated method enables root cause analysis of post-release problems by tracing these problems to one or more factors that influence system testing efficacy. Development defect data help to identify which parts of the software should be tested more intensely and earlier because they were fault-prone in development. Based on assessment results, testers can develop testing guidelines to make system test more effective. A case study applies this evaluation instrument to existing project data from a large software product (medical record system). Successive releases of the software product validate the method.

During system testing, testers may need to determine quantitatively whether to continue testing or to stop, recommending release. Early stopping could decrease the cost of testing, but has the disadvantage of possibly missing problems that would have been detected, had

system testing continued. Testers need to evaluate the efficiency of currently used methods and to improve the efficiency of future testing efforts. This dissertation develops empirical techniques to determine stopping points during testing. It proposes a new selection method for software reliability growth model(s) that can be used to make release decisions. The case study compares and evaluates these techniques on actual test result data from industry.

Quality assessment of multiple releases of the same product forms the basis of longitudinal decisions, such as re-engineering. Techniques using data from prior releases help to identify parts of the system that are consistently problematic. This information aids in developing additional testing guidelines for future releases of the product. This dissertation adds to a study that adapted a reverse architecting technique to identify fault relationships among system components based on whether they are involved in the same defect fix. The case study applies this technique to identify those parts of the software that need to be tested more.

Results of the case study demonstrate that the integrated method can improve the effectiveness and efficiency of system test. The method identified problematic software components using data from prior releases and development. Results of prioritizing show that fault-prone components tested earlier reveal more defects earlier. Development should, therefore, have more time to fix these defects before release. The method was also able to estimate remaining defect content. The estimates were used to make release decisions. Based on data from post-release and interviews with the test manager, the method recommended the right release decisions.

Catherine V. Stringfellow  
Department of Computer Science  
Colorado State University  
Fort Collins, Colorado 80523  
Summer 2000

# Acknowledgements

I am very grateful to my advisor, Dr. Anneliese von Mayrhauser, for her guidance and support throughout my research. My thanks go to Dr. Claes Wohlin for his assistance in carrying out work on my research and providing feedback on several chapters of this thesis. My thanks also go to Dr. Robert France, Dr. Jim Bieman, and Dr. Don Zimmerman for serving as members on my committee.

I am grateful to my colleagues at New Mexico Highlands University for their support and for filling in for me during my absence.

I am grateful to my family and my friends for their support and encouragement. I would like to thank my undergraduate advisor, friend, and mentor, Dr. Gary Huckabay, for encouraging me to go to graduate school. I especially thank my parents for their love and support and for teaching me the value of education and determination.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	1
1.2	Objectives . . . . .	5
1.3	Organization . . . . .	7
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	Fault-Proneness . . . . .	9
2.1.1	Methods based on Product Metrics . . . . .	11
2.1.2	Methods based on Defect or Change Data . . . . .	17
2.2	Fault Architecture . . . . .	24
2.2.1	Reverse Architecture . . . . .	24
2.2.2	Fault Architecture Technique . . . . .	27
2.2.3	Defect Cohesion and Defect Coupling Measures . . . . .	30
2.2.3.1	Defect Cohesion Measures . . . . .	31
2.2.3.2	Defect Coupling Measures . . . . .	32
2.2.4	Determining Fault-Prone Components and Fault-Prone Relationships	33
2.3	Methods that Use Prioritization to Improve Efficiency . . . . .	34
2.3.1	Prioritize Test Cases . . . . .	35
2.3.2	Prioritizing Modules for Testing . . . . .	37
2.4	Defect Estimation using Static Models . . . . .	37
2.4.1	Experience-based Methods . . . . .	38
2.4.2	Methods based on Capture-Recapture Models . . . . .	43
2.4.2.1	Notation for Capture-Recapture Methods . . . . .	45
2.4.2.2	Estimators for Capture-Recapture Methods . . . . .	46
2.4.2.3	Studies on Capture-Recapture . . . . .	49
2.4.3	Detection Profile Method and Cumulative Method . . . . .	51
2.4.4	Comparison of Capture-Recapture, Detection Profile and Cumulative Methods . . . . .	54
2.4.5	Methods that Integrate Experience . . . . .	56
2.5	Defect Estimation using Dynamic Models . . . . .	62
2.5.1	Software Reliability Models . . . . .	62
2.5.2	Software Reliability Models in Practice . . . . .	69

2.6	Research Matrix . . . . .	72
<b>3</b>	<b>Approach</b>	<b>77</b>
<b>4</b>	<b>Case Study</b>	<b>85</b>
4.1	Data . . . . .	85
4.2	Validity of the Case Study . . . . .	86
<b>5</b>	<b>Fault-prone Component Analysis</b>	<b>90</b>
5.1	Approach . . . . .	90
5.1.1	Determination of Fault-prone Components . . . . .	91
5.1.2	Consideration of Other Indicators . . . . .	92
5.1.3	Comparison to Fault-Prone Components in Post-Release . . . . .	92
5.2	Results . . . . .	93
5.2.1	Fault-prone Component Analysis Applied to Release 1 . . . . .	93
5.2.2	Evaluation of Other Indicators . . . . .	96
5.2.3	Comparison to Fault-Prone Components in Post-Release . . . . .	98
5.2.4	Proposed Testing Guidelines . . . . .	99
5.2.5	Fault-Prone Component Analysis Applied to Releases 2 and 3 . . . . .	99
5.2.6	Cross Release Analysis . . . . .	102
5.3	Summary . . . . .	103
<b>6</b>	<b>Fault Architecture Analysis</b>	<b>105</b>
6.1	Approach . . . . .	105
6.2	Release Analysis . . . . .	105
6.3	Single Phase Analysis . . . . .	109
6.4	Case Study Results . . . . .	110
6.4.1	Release Analysis . . . . .	110
6.4.1.1	Defect Cohesion Measures . . . . .	110
6.4.1.2	Fault Component Directory Structure . . . . .	112
6.4.1.3	Defect Coupling Measures . . . . .	113
6.4.1.4	Component Level Fault Architecture Diagrams . . . . .	114
6.4.1.5	Lift the Fault Relationships to the Subsystem Level . . . . .	118
6.4.2	Single Phase Analysis . . . . .	120
6.4.2.1	Fault Architecture Diagrams for Development and Test . . . . .	120
6.4.2.2	Fault Architecture Diagrams for Post-Release . . . . .	124
6.5	Summary . . . . .	128
<b>7</b>	<b>Prioritizing Testing Activities</b>	<b>130</b>
7.1	Evaluating the Effect of Testing Problem Components Early . . . . .	130
7.2	Results . . . . .	132
7.2.1	Prioritization based on New and Fault-Prone Components . . . . .	132
7.2.2	Prioritization based on Fault-Prone Relationships . . . . .	134
7.3	Summary . . . . .	136



<b>8</b>	<b>Analysis of Static Defect Estimation</b>	<b>138</b>
8.1	Approach . . . . .	138
8.1.1	Estimating Components with Defects in Post-Release and Not in Test	138
8.1.2	Estimating Still Defective Components (Components with Defects in Test and Post-Release) . . . . .	142
8.2	Case Study Data . . . . .	143
8.3	Results . . . . .	146
8.3.1	Estimation of Defective Components in Post-Release and Not in Test	146
8.3.1.1	Evaluation of Estimates using Three Test Sites . . . . .	146
8.3.1.2	Evaluation of Estimates using Two Test Sites . . . . .	148
8.3.1.3	Evaluation of Estimates Obtained Earlier in System Test . . . . .	151
8.3.2	Release Decisions based on Estimates . . . . .	153
8.3.2.1	Release Decisions using Three Test Sites . . . . .	153
8.3.2.2	Evaluation of Decisions in Last Week (Three Sites) . . . . .	154
8.3.2.3	Evaluation of Earlier Decisions (Three Sites) . . . . .	157
8.3.2.4	Release Decisions using Two Test Sites . . . . .	160
8.3.2.5	Evaluation of Decisions in Last Week (Two Sites) . . . . .	160
8.3.2.6	Evaluation of Earlier Decisions (Two Sites) . . . . .	163
8.3.3	Estimation of Still Defective Components . . . . .	164
8.4	Summary . . . . .	165
<b>9</b>	<b>Analysis of SRGM Selection Method</b>	<b>168</b>
9.1	Approach . . . . .	168
9.2	Results . . . . .	172
9.3	Summary . . . . .	180
<b>10</b>	<b>Integration Analysis</b>	<b>181</b>
<b>11</b>	<b>Conclusion</b>	<b>187</b>
11.1	Quality Assessment . . . . .	187
11.2	Test Guidelines and Strategies . . . . .	189
11.3	Release Decisions . . . . .	189
11.3.1	Static Defect Estimation . . . . .	190
11.3.2	SRGM Selection Method . . . . .	191
11.4	Multiple Release Assessment . . . . .	191
11.5	Validity of Integrated Method . . . . .	192
<b>12</b>	<b>Future Work</b>	<b>195</b>
<b>A</b>	<b>Case Study Data</b>	<b>199</b>
A.1	Failure data used for SRGM selection method. . . . .	199
A.2	Defect Data for Fault-Prone Analysis . . . . .	200
A.2.1	Defects Found in Development and System Test . . . . .	200
A.2.2	Diffusion Matrices based on Severity for Release 2 and Release 3 . . . . .	200
A.3	Cumulative Defects . . . . .	202
A.4	Subsystems and Problematic Components . . . . .	204
A.5	Data used in Static Defect Estimation Methods . . . . .	205
A.5.1	Resulting Estimates . . . . .	205

A.5.2 Estimation Errors . . . . .	207
A.5.3 $\chi^2$ analysis . . . . .	208
<b>REFERENCES</b>	<b>209</b>

# List of Figures

1.1 Methodolody to improve testing effectiveness and efficiency. . . . .	1
2.1 An illustration of the different types of capture-recapture models. . . . .	44
2.2 Plots for the a) Detection Profile Method and b) Cumulative Method. . . . .	53
3.1 Techniques used to improve testing effectiveness and efficiency. . . . .	77
6.1 Example of a Fault Component Directory Structure Diagram. . . . .	108
6.2 Fault Component Directory Structure. . . . .	112
6.3 Release 1 Component Level Fault Architecture. . . . .	115
6.4 Release 2 Component Level Fault Architecture. . . . .	116
6.5 Release 3 Component Level Fault Architecture. . . . .	116
6.6 Release 1 Fault Architecture Diagram. . . . .	118
6.7 Release 2 Fault Architecture Diagram. . . . .	119
6.8 Release 3 Fault Architecture Diagram. . . . .	119
6.9 Cumulative Release Diagram. . . . .	120
6.10 Fault Architecture Component Level Diagrams for development in Release 1. . . . .	121
6.11 Fault Architecture Component Level Diagrams for system test in Release 1. . . . .	122
6.12 Fault Architecture Component Level Diagrams for development in Release 2. . . . .	122
6.13 Fault Architecture Component Level Diagrams for system test in Release 2. . . . .	123
6.14 Fault Architecture Component Level Diagrams for development in Release 3. . . . .	123
6.15 Fault Architecture Component Level Diagrams for system test in Release 3. . . . .	124
6.16 Fault Architecture Component Level Diagrams for post-release in Release 1. . . . .	125
6.17 Fault Architecture Component Level Diagrams for post-release in Release 2. . . . .	125
6.18 Fault Architecture Component Level Diagrams for post-release in Release 3. . . . .	126
7.1 Release 1 cumulative defect curves (Guideline 4). . . . .	133
7.2 Release 2 cumulative defect curves (Guideline 4). . . . .	133
7.3 Release 3 cumulative defect curves (Guideline 4). . . . .	134
7.4 Release 1 cumulative defect curves (Guideline 5). . . . .	135
7.5 Release 2 cumulative defect curves (Guideline 5). . . . .	135
7.6 Release 3 cumulative defect curves (Guideline 5). . . . .	136
8.1 Relative errors and mean absolute relative error for all releases (three sites). . . . .	147

8.2	Relative errors and mean absolute relative error for all releases (2 sites). . .	149
9.1	Flowchart for SRGM Selection Method. . . . .	170
9.2	Plot of Release 1 data and SRGM models not rejected. . . . .	174
9.3	Plot of Release 2 data and SRGM models not rejected. . . . .	177
9.4	Plot of Release 3 data and SRGM models not rejected. . . . .	179
10.1	Flowchart for applying the integrated method. . . . .	182

# List of Tables

1.1	Research questions for each part of methodology. . . . .	3
1.2	Relationship between methodology phases, methods and objectives. . . . .	5
2.1	Studies on product-based methods. . . . .	11
2.2	Studies on fault-proneness based on defect or change data. . . . .	18
2.3	Studies on defect estimation methods based on defect or change data. . . . .	39
2.4	Statistical methods for capture-recapture models. . . . .	45
2.5	Jackknife estimators $\hat{N}_{hk}$ for $k = 1, \dots, 5$ . . . . .	47
2.6	Studies on Capture-recapture methods. . . . .	49
2.7	Studies on Capture-recapture, DPM and Cumulative Methods. . . . .	55
2.8	Studies that integrate experience. . . . .	57
2.9	Briand's DPM strategies summarized. . . . .	61
2.10	Software reliability growth models used in this study. . . . .	69
2.11	Research Matrix for Fault-Prone Classification Methods. . . . .	73
2.12	Research Matrix for Reverse Architecting and Code Decay Methods . . . . .	74
2.13	Research Matrix for Defect Estimation Methods. . . . .	74
4.1	Attributes recorded for defect reports. . . . .	85
4.2	Attributes recorded for repair reports. . . . .	86
5.1	Diffusion Matrix for Release 1 using a threshold of 10%. . . . .	94
5.2	Statistics on fault-prone components using order of magnitude threshold. . . . .	94
5.3	Diffusion matrix for Release 1 using order of magnitude threshold. . . . .	95
5.4	Diffusion Matrix including new components as fault-prone in Release 1. . . . .	96
5.5	Maximum defects for a component by severity level for Release 1. . . . .	96
5.6	Diffusion Matrix for Release 1 by severity 1. . . . .	97
5.7	Diffusion Matrix for Release 1 by severity 2. . . . .	97
5.8	Diffusion Matrix for Release 1 by severity 3. . . . .	97
5.9	Diffusion Matrix for Release 1 by severity 4. . . . .	97
5.10	Diffusion Matrix for System Test versus Post-Release in Release 1. . . . .	98
5.11	Testing guidelines derived from Release 1 data. . . . .	99
5.12	Diffusion matrix for Release 2. . . . .	99
5.13	Diffusion Matrix including new components as fault-prone in Release 2. . . . .	100

5.14	Fault-prone components in System Test versus Post Release in Release 2. . . . .	100
5.15	Diffusion Matrix for Release 3. . . . .	101
5.16	Diffusion Matrix including new components as fault-prone in Release 3. . . . .	101
5.17	Diffusion Matrix for System Test versus Post Release in Release 3. . . . .	101
5.18	Diffusion Matrix for fault-prone components across Releases. . . . .	102
5.19	Diffusion Matrix for fault-prone components across Releases 2 and 3. . . . .	102
6.1	Number of components identified as fault-prone in Releases 1 – 3. . . . .	111
6.2	Number of releases in which components were fault-prone. . . . .	111
6.3	Fault relationship information. . . . .	113
6.4	Fault-prone relationship information using the defect coupling measures. . . . .	114
6.5	Fault relationship information for all releases by development phase. . . . .	121
6.6	Testing guidelines derived from applying set of methods. . . . .	124
7.1	Testing guidelines derived from applying set of methods. . . . .	131
8.1	Release data for three test sites (two sites are in parentheses). . . . .	144
8.2	Multiplicative factors for all releases. . . . .	145
8.3	Actual values used to determine correct release decisions. . . . .	145
8.4	Ranking of estimators over three releases (three sites). . . . .	148
8.5	Ranking of estimators over three releases (three sites). . . . .	150
8.6	Release 1 decisions earlier in test (three sites). . . . .	154
8.7	Release 2 decisions earlier in test (three sites). . . . .	155
8.8	Release 3 decisions earlier in test (three sites). . . . .	156
8.9	Ranks for estimators for three test sites. . . . .	159
8.10	Release 1 decisions earlier in test (two sites). . . . .	160
8.11	Release 2 decisions earlier in test (two sites). . . . .	161
8.12	Release 3 decisions earlier in test (two sites). . . . .	162
8.13	Ranks for estimators for two test sites. . . . .	163
8.14	Release data and estimates for Releases 2 and 3. . . . .	165
9.1	Release 1 predictions and correlation values. . . . .	173
9.2	Final estimates and errors by SRGM models not rejected in Release 1. . . . .	175
9.3	Release 2 predictions and correlation values. . . . .	176
9.4	Final estimates and errors by SRGM models not rejected in Release 2. . . . .	177
9.5	Release 3 predictions and correlation values.. . . . .	178
9.6	Final estimates and errors by SRGM models not rejected in Release 3. . . . .	179
10.1	Testing guidelines derived from applying set of methods. . . . .	183
10.2	Recommended release decisions in the last week of system test. . . . .	184
A.1	Cumulative number of failures for all three releases. . . . .	199
A.2	Statistics on defects found by development and system test. . . . .	200
A.3	Diffusion Matrix for Release 2 by severity 1. . . . .	200
A.4	Diffusion Matrix for Release 2 by severity 2. . . . .	200
A.5	Diffusion Matrix for Release 2 by severity 3. . . . .	201
A.6	Diffusion Matrix for Release 2 by severity 4. . . . .	201
A.7	Diffusion Matrix for Release 3 by severity 1. . . . .	201
A.8	Diffusion Matrix for Release 3 by severity 2. . . . .	201

A.9	Diffusion Matrix for Release 3 by severity 3. . . . .	201
A.10	Diffusion Matrix for Release 3 by severity 4. . . . .	201
A.11	Release 1 cumulative defects by week for unprioritized and prioritized testing.	202
A.12	Release 2 cumulative defects by week for unprioritized and prioritized testing.	203
A.13	Release 3 cumulative defects by week for unprioritized and prioritized testing.	204
A.14	Subsystem containment of problematic components. . . . .	204
A.15	Release 1 estimates (3 sites). . . . .	205
A.16	Release 2 estimates (3 sites). . . . .	205
A.17	Release 3 estimates (3 sites). . . . .	205
A.18	Release 1 estimates (2 sites). . . . .	206
A.19	Release 2 estimates (2 sites). . . . .	206
A.20	Release 3 estimates (2 sites). . . . .	206
A.21	Estimation Errors for Static Methods (3 sites). . . . .	207
A.22	Estimation Errors for Static Methods (2 sites). . . . .	207
A.23	$\chi^2$ for each estimator for three releases (three sites). . . . .	208
A.24	$\chi^2$ for each estimator for three releases (two sites). . . . .	208

# Chapter 1

## Introduction

### 1.1 Problem Statement

In software testing, a tester is faced with two challenges: limited testing time on the one hand and delaying the release of software when it is likely to still contain too many or costly errors. The first challenge relates to testing efficiency, while the second relates to testing effectiveness. Since testing can only show the presence of errors, and not their absence, a low testing yield may be due to the high reliability of the software under test, in which case testing should stop sooner. A low testing yield, however, may also be due to an ineffective testing technique. This is why testers use an arsenal of techniques to ensure software is sufficiently tested.

To improve testing effectiveness and efficiency, test managers need to

- determine what parts of software are problematic.
- create testing guidelines to focus on those problematic parts.
- make decisions to continue testing or to stop and release the software.

This dissertation proposes an integrated method that includes a set of techniques to evaluate and improve testing effectiveness and efficiency. Figure 1.1 illustrates this methodology. The

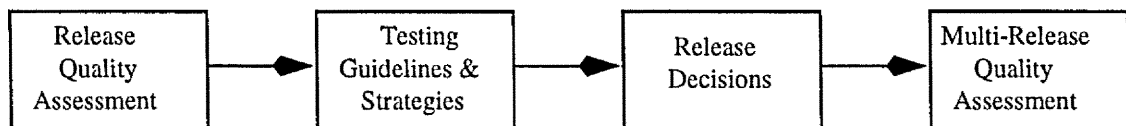


Figure 1.1: Methodology to improve testing effectiveness and efficiency.



methodology consists of four parts: quality assessment, testing strategies, release decisions and multiple release assessment.

Quality assessment of the software drives effectiveness and efficiency issues. It focuses testing on parts of the software likely to be problematic so that testing is more effective in uncovering defects. It also sets priorities, based on identifying the parts of the software that should be tested earlier to improve efficiency. Establishing the focus of testing and the priorities during testing enables testing guidelines and strategies to be developed.

Quality assessment also aids in making release decisions. Knowledge of remaining defect content or defect rates can be used to make decisions about the quality of the software and whether the software quality is acceptable so that it may be released. Testing strategies that improve the efficiency of the testing process can accelerate the gains in software quality so that release may occur sooner.

Most software testing today is developed over a series of releases. In the best of cases, it continually improves. In the worst case, it decays. Thus it is important to assess the quality of software over several releases. A multiple-release assessment identifies problems like code decay and deterioration of key components. This kind of assessment can identify long-term priorities and contributes to the development of testing guidelines and strategies. It also aids in making longitudinal decisions regarding the software.

Table 1.1 shows the research questions for each part of the methodology. The type of data that is available drives the kinds of quality assessment questions that can be asked. For example, if no data is available on severity of errors, then one cannot determine if components with severe problems have more problems.

Answers to quality assessment questions will aid in the development of testing guidelines and strategies. They may enable us to propose better guidelines to improve testing effectiveness and efficiency. Guidelines will enable testers to focus on the parts of the software that cause the most problems. Problems identified early enough in test have a better chance of being fixed before release.

The decision to stop testing is usually controlled by management based on marketing goal. In some organizations, testing stops when testing yield saturates, where yield is defined

Table 1.1: Research questions for each part of methodology.

	Questions
Quality Assessment	<p>How good is the software?</p> <p>How many parts of the software have problems?</p> <p>What parts of the software are the worst?</p> <p>What kinds of components are likely to have problems after release?</p> <p>Do components with severe problems have more problems?</p> <p>What parts of a system were problematic in the previous release?</p> <p>What are the relationships between fault-prone components in a system?</p> <p>How many defective components remain after testing?</p> <p>How many defects remain after testing?</p>
Testing Guidelines and Strategies	<p>What part of the software should testing focus on?</p> <ul style="list-style-type: none"> <li>- Should testing focus on parts that had many problems in development?</li> <li>- Should testing focus on parts that had severe problems in development?</li> <li>- Should testing focus on parts that had many modifications?</li> <li>- Should testing focus on parts that were new?</li> </ul> <p>How can the testing process be made more efficient, that is, what changes can be made to the testing process that would result in earlier release of software?</p> <p>When should problematic parts of the software be tested?</p> <p>Will proposed testing guidelines not only improve effectiveness, but also improve testing efficiency?</p>
Release Decisions	<p>Can defect estimations made using testing data be used to determine the right point at which to stop testing and release software?</p>
Multi-Release Quality Assessment	<p>What kinds of problems occur in release after release?</p> <p>Do persistent problems indicate code decay?</p> <p>Do persistent problems indicate a need for re-architecting?</p>

in terms of number of faults (or failures) exposed. Unfortunately, it is not always clear whether the software is ready for release at that point. It would be helpful if testers could use testing data to determine the right point at which to stop testing and release software. If defects have their root causes based on problems known earlier in the life cycle, the number of problems that are left may be estimated. For example, techniques like *capture-recapture* can be used on earlier data to forecast the number of problems that are likely left in the software at the time of release [49, 69, 70]. Depending on the estimates, testers can make decisions on whether to recommend release or to test more. Using quantitative techniques to estimate remaining defect content will enable testers to assess the quality of the software and give management information to make the right release decisions.

Answers to questions regarding multiple release assessments may also affect testing strategies, as well as longitudinal decisions.

The integrated method this dissertation proposes may be used to answer these questions and to provide ways to improve testing effectiveness and efficiency through empirical analysis. Various methods exist that can be applied to assess the software and to make release decisions. This dissertation investigates several approaches that can be used within this methodology, including GYR analysis [41, 40, 65] to identify fault-prone components, fault architecture techniques [40, 65, 66] to identify fault relationships between components, static defect estimation techniques such as capture-recapture models [6, 7, 17, 49, 64, 69, 70, 77], curve-fitting methods, such as the Detection Profile Method and the Cumulative Method [69, 7], and experience-based methods [5, 78] to estimate number of components with defects, and software reliability growth models (SRGMs) [23, 24, 29, 38, 37, 74, 75, 76] to estimate the total number of defects. This dissertation also proposes some new techniques to estimate defect content.

Since no one tool or method works on all data, the integrated method uses a set of methods that complement each other with selection criteria for each. Multiple evaluations provide more credible information for decision making.

A case study applies the integrated method to empirically validate it. The case study evaluates current testing practices of a group of testers from industry for effectiveness and efficiency. This study collected defect data from several development life-cycle phases from three releases of a large medical record system. Like all case studies, this one has high external validity, because it uses industry data. Internal and external validity are addressed in 4.2.

Given that testing can consume significant effort, especially when high reliability requirements must be met, or field failures are very costly, improvements in the testing process have potential for a very big pay-off. Even high quality environments can benefit from using the integrated method that this dissertation proposes to make improvements in effectiveness and efficiency.

## 1.2 Objectives

Many attributes of a software project influence the software testing effort, both its effectiveness and efficiency. Some are complexity of the problem, schedule urgency, and the quality of work during design and implementation. The objectives of the integrated method are to provide the following:

1. Evaluation of factors influencing system test deficiencies and their causes.
2. Estimation of remaining problems to assess software quality.
3. Evaluation of methods to be used for release decisions.
4. Evaluation of methods to develop guidelines for testing improvement.

Table 1.2: Relationship between methodology phases, methods and objectives.

Methodology Phase	Methods	Objectives
Release Quality Assessment	Fault-prone component analysis Fault Architecture analysis Defect Estimation	1, 2
Testing Guidelines and Strategies	Focus testing Reorder testing activities	4
Release Decisions	Capture-recapture models and curve-fitting Experienced-based defect estimation SRGM selection method	3
Multi-Release Quality Assessment	Fault-prone component analysis Fault Architecture analysis	1

Table 1.2 shows the relationship between the parts of the methodology, the methods used in each part, and the objectives. To achieve the first objective, defect data from development is analyzed to determine whether it can be used to guide testing. Of particular interest is whether defects during development are related to the occurrence of defects during testing. The rationale for such a relationship is that:

- Components with severe or systemic problems during development carry a higher risk of not being completely fixed at the start of system test.

- They are more likely to exhibit long term problems.
- These problems are more likely to be severe.

Development defect analysis of software components can guide testing with the goal of focusing on parts of the software that were fault-prone during development. A method in Ohlsson et al. [41] identifies fault-prone components across releases. This thesis applies the method in a novel way to identify fault-prone components not only across releases, but within releases. Development defect data identifies the parts of the software that need to be tested more because they were fault-prone during development. Development defect data identifies components with problems that should be tested more. System test data from the same release validates the predictions. Analysis of post-release defects determines what components are missed in system test. This thesis replicates a study [40, 65, 66] using our data to identify relationships among system components based on whether they are involved in the same defect report. The resulting fault architectures for three releases indicate the most fault-prone relationships. Our case study uses fault architectures to identify fault-prone components after release and our predictions are validated using post-release data.

Another important issue for testers is to determine to what degree they prevent post-release defects and how many components make it into release with undetected defects that show up after release. To achieve the second objective, our case study analyzes defect reports from development, system test and post-release for the purposes of estimating remaining problems. Several defect content estimation techniques are analyzed and applied in a case study. The techniques analyzed include capture/recapture models [17, 64, 70], curve fitting models [37, 69], and a simple experience-based model.

System testing is an important and costly phase of software development. It is not always clear whether software has been tested enough, or should continue to be tested. Testing too long wastes valuable resources. Testing too little has the disadvantage of possibly missing problems that would have been detected, had system testing continued. This causes costly post-release problems. During system testing it might be desirable to determine quantitatively whether to stop or continue testing. The same information used in determining what to test and how much to test, could also be valuable in guiding when

to stop testing. To achieve the third objective, this dissertation investigates methods that can aid in making release decisions. Methods to estimate remaining defect content are one way to determine whether to stop testing. Static defect estimations include [17, 64, 70, 69]. Dynamic methods include [23, 29, 38, 37, 74, 76].

To achieve the fourth objective, a set of methods to improve testing effectiveness and efficiency are applied and evaluated. Results of assessment methods aid in developing guidelines for system test. For example, components that are fault-prone during development should be system tested more intensely and at the earliest possible time. This should shift higher defect intensities earlier in the test cycle, giving developers more time to fix remaining problems.

To achieve the fourth objective, this dissertation develops methods to aid in making release decisions. This dissertation uses capture-recapture and curve-fitting methods in a new way to estimate the number of defective components after release that had no defects in system test. This dissertation also proposes a new method to select the best software reliability growth model(s) to use to estimate the number of remaining defects for the purpose of making release decisions.

### 1.3 Organization

Chapter 2 presents techniques that have been used to identify fault prone components and relationships with the goal of using that information to make testing more effective. It also provides background on defect estimation techniques.

Chapter 3 describes the approach of the integrated method. Several ways are proposed to use defect data from earlier life cycle phases to identify fault-prone components and develop fault architectures to determine components that may be tested earlier and more intensely. Several ways are also proposed to use this data in estimating defect content to make system release decisions.

Chapter 4 describes the data used in the case study that applies the integrated method. It also discusses the issues of internal and external validity.

Chapters 5 – 9 describe the methods in the integrated method and applies them to defect data of three releases of a large medical record system. Chapter 5 does fault-prone component analysis to develop testing guidelines for system test. Testing guidelines developed using the first release are applied to the second release to evaluate the effectiveness of those guidelines. The effect of the testing guidelines on efficiency improvement are evaluated.

Chapter 6 applies the fault architecture technique to the defect data to identify problematic parts of the software that should be tested more.

Chapter 7 evaluates the effect of a testing guideline based on the fault-prone analysis on the defect exposure profile. It looks at the effect this has on the cumulative defect curve that may in turn affect release decisions.

Chapter 8 applies static defect estimation methods to estimate defect content in a new way. It estimates the number of components that have defects in release that do not have defects in test. Chapter 9 introduces a selection method for software reliability growth models to estimate the total number of defects. The purpose of these methods are to make release decisions.

Chapter 10 analyzes how the set of methods work together to make decisions to improve testing effectiveness and efficiency.

Chapter 11 draws conclusions, pointing out advantages and limitations of the integrated method. Chapter 12 presents possible further work in this area.

# Chapter 2

## Background

The approach for the integrated method presented in Chapter 3 is based on several existing methods. The integrated method identifies components that are fault-prone or are in fault-prone relationships to guide testing and improve effectiveness. It also uses defect content estimation techniques to help make release decisions. The method uses stopping rules to improve efficiency. Section 2.1 presents existing approaches to determine fault-prone parts of software. Section 2.2 presents methods that identify components in fault-prone relationships. Section 2.3 describes techniques that reorder testing activities to improve efficiency and effectiveness. Section 2.4 presents and compares several methods that estimate remaining defect content. Section 2.5 describes several well-known software reliability growth models and their underlying assumptions. It also reports on empirical evaluations that highlight both the fragility and robustness of using SRGMs when assumptions of these models are violated.

### 2.1 Fault-Proneness

Frankl, et al. [21] see two main goals in testing: to achieve quality, by probing the software for defects that can be removed, and to assess quality to gain confidence that software is reliable. Depending on the goal, different testing strategies are used: Debug testing is used to achieve quality and operational testing is used to assess quality. The testing strategy used also determines the kinds of defects found. Debug testing focuses on finding defects with a higher probability of being detected, not necessarily those most important, while operational testing focuses on finding defects that are most likely to occur



in the field, with probabilities in proportion to their severity. Their study found that when testing software for the purpose of improving quality (i.e., removing defects), it helps if the tester has good intuition and insight in devising testing strategies, otherwise operational testing is indicated. Aids to improve or validate a tester's intuition would be helpful. Knowing which components are fault-prone allows the tester to concentrate on devising testing strategies for those components.

Several static models exist to determine fault-proneness or estimate defect content. These models assume the general form [14] of

$$y = f(x_1, x_2, \dots, x_n).$$

The dependent variable  $y$  is a defect metric, such as the number of defects found in a phase of the development life cycle. The independent variables,  $x_i$  may be product- or process-related. An example of a product-related independent variable is size, which may be expressed in terms of lines of code (LOC), number of operands, or McCabes' complexity measure [14]. Studies in [2, 42, 43, 8, 57] investigate models based on product metrics. Examples of process-related static models include experienced-based models [5, 78], which predict the number of faults in release based on the number of faults in earlier phases of development, capture-recapture models [6, 7, 17, 49, 64, 69, 70, 77], the Detection Profile Method, and the Cumulative Method [69, 7]. Capture-recapture models, the Detection Profile Method and the Cumulative Method predict the number of defects remaining based on how many reviewers found each defect during code inspection. These models are static because the estimate of the number of defects is based on the current values of the independent variables, ignoring the rate of change of any metric over time. If the variables can be measured earlier than the dependent variable, these models have the potential to make predictions and guide activities in later phases, like system test.

Much of the research in the area of defect analysis on software components has focused on identifying fault-prone components [8, 30, 33, 41, 42, 43, 57] or predicting the number of defects remaining in components [5, 17, 69, 70, 78] based on their characteristics. It is useful to know which components are fault-prone during prior release or in earlier life cycle activities, as these components should be tested more intensely. Data from previous

releases can be used, but if this data is not available, data from earlier life cycle phases may be helpful.

There are many methods [2, 8, 42, 43, 57] based on product metrics to identify components that are fault-prone. Section 2.1.1 describes some of these methods. Our focus, however, is on methods that are based on defect data and change data. Section 2.1.2 describes methods that are based on defect and change data to identify fault-prone components.

### 2.1.1 Methods based on Product Metrics

Table 2.1 summarizes the studies described in this section. It identifies the dependent and some of the independent variables used in the models, as well as the techniques used in the studies.

Table 2.1: Studies on product-based methods.

Study	Dependent Variable	Independent Variables/Metrics	Technique	Results
Ash et al. [2]	Software Maintainability Index	Control and information structure and typography (LOC, # modules, type of module, V(g))	HPMAS Weight and Trigger Point Range	Identifies components that need work.
	Software Maintainability Index	Halstead's volume, extended cyclomatic complexity, LOC, % comments, # modules	Polynomial Linear Regression	Identifies modules for maintenance.
N. Ohlsson et al. [42]	# Trouble reports (fault-proneness)	Design metrics # decisions, new and modified signals (SigFF), #comparisons, #paths, #branches, depth, #loops, etc.	Spearman's rank order correlation	Best predictors: new & modified signals (SigFF) and # decisions in module.
N. Ohlsson et al. [43]	# Failure reports	Same as [42]	Alberg diagram diagram (correlation)	Best predictors: SigFF with # conditions or # decisions or McCabe's measure.
Briand et al. [8]	# faults in test	OO design measures 28 coupling, 10 cohesion, and 11 inheritance	Alberg diagram (correlation)	Best: 7 coupling, 4 inheritance strongly related (better than size).
Schneidewind et al. [57]	Low or high module quality	13 metrics: LOC, # operators and operands, #comments #nodes, edges, paths in graph, path length	Boolean Discriminant Functions	Comments and statements identify low quality modules best.

Ash et al. [2] provide a method to track fault-prone components across releases. They describe methods that may be used for quantifying software maintainability from software metrics. The metrics are used to drive the maintenance process to prevent further code degradation. Two methods were applied to industrial software systems, because they were

easy to implement and use by software engineers. The results were compared to the subjective evaluation of software maintainability performed by actual maintainers of the systems.

The Hierarchical Multidimensional Assessment (HPMAS) method [2] models software maintainability as a hierarchical structure of a set of software metrics. The suggested hierarchical model divides maintainability into three dimensions (or attributes):

1. Control structure: Refers to the way in which the software is decomposed into algorithms.
2. Information structure: Refers to the choice and use of data structure and data flow techniques.
3. Typography, Naming and Commenting: Refers to the layout of the code and the naming and commenting of code.

The dimensions are measured by metrics at a lower level. If a metric lies within an acceptable range, no penalty is applied. If, however, it falls out of the range, then a penalty in proportion to its deviation is applied. The dimension maintainability,  $DM_{dimension}$ , is calculated as:

$$DM_{dimension} = 1 - \frac{\sum w_i D_i}{\sum w_i}$$

where

$D_i$  is the proportional deviation of the metric value from the optimum range of values.

$w_i$  is a weighted value between zero and one, inclusive.

The overall maintainability index is the product of the maintainability of all three dimensions. The rationale for multiplying the the dimensions' maintainability is that it gives a lower overall maintainability. This implies that a low maintainability in one dimension will reduce other aspects of maintainability and the maintainability of the entire system.

Ash et al. [2] applied the HPMAS method to assess overall maintainability of several system before and after perfective maintenance modifications. The HPMAS maintainability index changed very little, even so the system complexity increased due to additional error

checking in the code. The HPMAS method applied on a module-by-module basis demonstrated that it was able to determine which old and new components are satisfactory and which ones need work.

Ash et al. [2] also applied another method based on polynomial regression models. The polynomial regression models use regression analysis on polynomial equations to explore the relationships between software maintainability and software metrics. The metrics include Halstead's volume, extended cyclomatic complexity, lines of code, and, when available, the number of components. All of these are size measures. The authors constructed several linear regression models where the measures were used as dependent variables. The models were ordered based on how well they correlated to the system maintainers' subjective evaluation. Models using Halstead's volume and effort metrics were judged to be the best predictors of maintainability for their data. Ash et al. [2] used the following model to measure overall maintainability,  $M$ :

$$M = 171 - 5.2\ln(\text{aveVol}) - 0.23\text{aveV}(g') - 16.2\ln(\text{aveLOC}) + 50\sin(\sqrt{2.46\text{perCM}})$$

where

$\text{aveVol}$  is the average Halstead's volume per module.

$\text{aveV}(g')$  is the average extended cyclomatic complexity per module.

$\text{aveLOC}$  is the average lines of code per module.

$\text{perCM}$  is the average percent of lines of comments per module.

Ash et al. [2] applied the metric polynomial regression model to several versions of a system, two of which had been re-engineered. The metrics show that successive versions showed decreased maintainability, until a re-engineering effort occurred. Only the last version did not degenerate and this was attributed to a new maintenance process that used metrics to guide the process. The metric polynomial regression model was also applied to several different software systems for comparison purposes. The metric polynomial model matched the expected results of an informal evaluation of software system engineers with experience with the systems. A module-by-module analysis performed on the two systems

successfully predicted the system that was easier to maintain. It also showed that individual modules could be identified for maintenance.

Using these two models, the authors show that it is possible to quantitatively analyze code quality through maintenance changes using the HPMAS model. The polynomial regression model can be used to track code health across versions, to compare whole software systems and to keep track of code health.

Ohlsson, Helander and Wohlin [42] derive a model for identifying fault-prone modules based on software design metrics, rather than code metrics. The goal is to identify a small portion of modules that contribute to a large number of faults early enough in the software life cycle to improve their quality. The authors studied several design metrics to determine those that have the ability to indicate the modules that are likely to be fault-prone.

The data showed that 20 percent of the modules were responsible for approximately 60 percent of the trouble reports from testing. To identify these modules, modules were ranked in decreasing order of the number of trouble reports written against them. Using Spearman's rank-order correlation coefficient, this ordering was compared to orders predicted by design metrics. Design metrics were considered individually and in combination.

Evaluation of the prediction model using a threshold of 20 percent resulted in identification of modules responsible for approximately 50 percent of the faults. A threshold of 30 percent found approximately 60 percent of the faults. The best prediction model for this data was based on the average combination of two design metrics, new and modified signals (used for communications between modules) and number of decisions within a module.

Ohlsson and Alberg [43] propose a model to predict the number of functional test failure reports associated with software modules based on design metrics for the modules. The prediction model is based on an earlier model in Alberg et al. [1] that multiplies the factors  $S$  and  $SigFF$ , where  $S$  is the number of statements in a module and  $SigFF$  is the number of new or modified signals. Their interest was in discovering design metrics that can be used instead of  $S$ , so that the model can be applied earlier in the software life cycle.

Twenty-seven design metrics were collected. Using regression analysis and a correlation coefficient of 0.40 or higher, the number of metrics used was reduced to eleven. Examples

of these include number of decisions, number of components, number of paths, number of loops, number of calls to subroutines, etc. The effects of adding and multiplying pairs of measures were investigated. Results showed that the original predictive model using  $S \cdot SigFF$  gave a coefficient of 0.76 for their data, while the new models ranged in values from 0.58 to 0.68. The models using  $SigFF$  with either the number of conditions, decisions, or McCabe's measure were found to be good predictors. The authors also evaluated the two best prediction models based on conditions or a modified McCabe's measure. To compare the prediction models, the authors used a graphical method that plots cumulative number of faults against modules that are ordered by applying different predictors. First, modules are ranked in decreasing order with respect to a predictor. Then the cumulative number of faults for different percentages of the modules are plotted. Percentages of modules are on the x-axis and cumulative number of faults for different percentages of the modules are on the y-axis. The prediction models are compared to an Alberg diagram [43]. In an Alberg diagram, modules are ranked in decreasing order with respect to the number of faults in the module, then the cumulative number of faults for different percentages of the modules are plotted.

The models accurately predicted 25 percent of the modules causing approximately 53% of the trouble reports, while using  $SigFF$  alone predicted 46%. These new models were not only available for use earlier, but were slightly better at identifying the most fault-prone modules.

Briand et al. [8] explore the relationships between object-oriented design measures for classes and the probability of detecting fault-prone classes. They investigate 28 coupling measures, 10 cohesion measures and 11 inheritance measures. To reduce the number of measures, the distributions and variance of each measure was examined. Only seven measures with a high variance that differentiated classes were selected for possible inclusion in the predictive model. These included seven coupling measures and four inheritance measures. The cohesion measures collected were found not to be strongly related to fault-proneness.

Using the data collected, 85 percent of the components were correctly identified as faulty and those components contained 95 percent of the faults that occurred in system

test. Because the prediction model was applied to the same set of data from which it was derived, the model was evaluated and compared to a predictor based on size. Results showed that 27% of the predicted classes accounted for 64 percent of the faults found in test. This model performed better than a predictor based on size alone.

Schneidewind [57] developed a model to validate and apply metrics for quality control using software from the space shuttle flight system. Metrics are used as early indicators of software quality problems and act as predictors for quality factors that are not available until after coding. Some of the metrics include the number of unique operators and operands, total number of operators and operands, number of executable statements, number of comments, number of nodes, edges and paths in the control graph, maximum and average path length. Schneidewind's contribution is in the use of Boolean discriminant functions (BDFs). BDFs classify components to be of low or high quality using information in the form of critical values. Critical values are threshold values of metrics that are used to reject or accept modules during the quality control process.

Schneidewind uses a three step process for selecting metrics for quality control, where quality is measured as the number of discrepancy reports (*drs*):

1. Identify and rank a set of candidate metrics according to their ability to discriminate between two sets of modules (for example, those with  $dr = 0$  and those with  $dr > 0$ ).
2. Determine critical values for the metrics. The method tests whether the cumulative distribution functions (CDFs) for the two sets are from different populations. The value corresponding to the maximum vertical difference between the CDFs of the two sets is the critical value (if the difference is significant).
3. Find the optimal combination of metrics and critical values.

In the experiment, 13 metrics were collected. Results show that two candidate metrics, comments and statements (both size measures), had a high degree of association with *drs*. Using contingency table analysis, one set of randomly selected modules validated these metrics with respect to the *drs*. A second set of randomly selected modules applied the validated metrics to identify problems early in order to resolve them.

Certain metrics are dominant, that is they make fewer mistakes in classifying quality than do others, and do not require additional metrics to accurately classify quality. This effect is called dominance. Concordance is the degree to which a set of metrics produces the same result in classifying software quality. Results show that properties of dominance and concordance are evident in the selection and validation process using statistical methods. A point is reached when adding additional metrics does not contribute to the improvement of quality classification, and the cost of additional metrics increases. A rule for adding metrics can be applied based on properties of BDFs.

Schneidewind compared the BDFs' ability to classify quality with linear discriminant functions. Linear discriminant functions use linear vectors of metrics. In this experiment, Schneidewind used a set of nine metrics (of the 13 collected). Results show that BDFs perform better in identifying low quality modules.

The research in this subsection focused on identifying fault-prone modules based on product metrics. Almost all of them use measures that are equivalent to a size measure. If fault-proneness is defined in terms of the number of defects reported for a module, one would expect a larger module to contain a larger number of defects. All size metrics should perform the same. Since it is possible for two modules to be roughly the same size, with one being fault-prone and the other not being fault-prone, a prediction model that is fully capable of explaining variation should probably also measure process data. Process data can include defect or change data. The inclusion of metrics that measure the development process would be useful in a software quality model.

The case study in this thesis does not consider product metrics, because the product data needed were not available.

### **2.1.2 Methods based on Defect or Change Data**

The following table summarizes the studies based on defect or change data in this section. Table 2.2 describes for each study, the techniques, dependent and independent variables, and the results.



Table 2.2: Studies on fault-proneness based on defect or change data.

Study	Purpose	Dependent variables	Independent variables	Technique	Best performers	Results
Ohlsson et al. [41]	Classify fault-prone modules	G, Y, R classification	size metrics, structural metrics, fault data	Ranking & Principal Components Analysis	Structural metrics (state and McCabe's measure)	Large change in size or large # faults and small change in size means fault-prone.
Khoshoftaar et al. [30]	Predict code churn	code churn (GC metric)	Software complexity metrics	Neural networks vs. Regression model	Neural networks	Useful tool for software maintenance.
Khoshoftaar et al. [33]	Predict debug code churn & classify fault-prone modules	debug code churn	process metrics (dev.code churn) vs. product metrics  vs. combined	module-ordering based on development (ranking)	module-ordering based on development code churn	More dev. code churn means more debug code churn. Larger modules have more code churn. Product metrics poor for debug code churn or fault-proneness. Dev.code churn & combined models good predictors. Combined model confirms other two models.
Eick et al. [18]	Predict code decay using change data	Code decay (effort to implement change)	span; # lines added changed, deleted; # developers	Scatter plots Regression analysis	span & size of change	Large spans mean more effort. Large, recent changes better predictor than size.
Khoshoftaar et al. [31, 32]	Evaluate impact of reuse on software quality	# faults	product metrics vs. product metrics and reuse	PCA Regression analysis  Informed prior probabilities & misclass. costs	Models with reuse	Reuse metrics enhance classification models. Informed prior probabilities & misclass. costs improve models.

Ohlsson, et al. [41] use defect data to predict fault-prone components. They combine prediction of fault-prone components with code decay analysis. Code decay is defined as code that is more difficult to change than it should be. Components are identified as problematic (or persistently fault-prone) in regards to code decay. A component is ranked based on its number of defects. Ranks and changes in ranks classify components as green, yellow or red (GYR) over a series of releases. The amount of code decay is interpreted from the classification.

- G indicates normal evolution, component is easily updated.
- Y indicates code decay, component is a candidate for reengineering.
- R indicates that the component is difficult and costly to maintain and is in need of reengineering.

To determine what methods and measures would be suitable to identify components in need of additional attention, either through testing or re-engineering, several product and process measures were used and evaluated. Examples of these measures include:

- size measures: lines of code (LOC), number of if statements, etc.;
- structural measures: cyclomatic complexity and amount of modified code, since it measures the amount of structural changes;
- fault data.

In total they used ten design and nine code measures. The approach involves three steps:

1. Rank components based on a product measure or their number of defects.
2. Identify components above some threshold.
3. Classify a component as red, if it is fault-prone in two releases. Classify a component as yellow, if it is fault-prone in only one release. Classify a component as green, if it is not fault-prone in either release.

To validate their approach, Ohlsson, et al. [41] used GYR analysis to determine the number of components correctly identified and incorrectly identified as fault-prone. Results showed that it is possible to classify components as green, yellow or red using simple methods, such as ranking. This kind of method is useful in pin-pointing components that may need special attention.

The case study in this thesis uses methods based on defect and change data. As in Ohlsson et al. [41], defect data is used to predict fault-prone components by using GYR analysis across releases. In addition to performing across-release analysis, the case study in this thesis involves across-phase GYR analysis to determine whether it is possible to predict fault-prone components in development and system test, as well as system test and post-release.

Other methods that identify fault-prone components using defect data do so by estimating defect content [4, 5, 78]. Those methods are described in Section 2.4.

Methods that predict whether or not a component will be fault-prone based on change data include [30, 33, 18, 25]. Khoshgoftaar et al. [30] use neural networks to make predictions on code churn, where code churn refers to new and changed code. (The assumption here is that modules with higher code churn are more fault-prone.) The authors collected gross change metrics (GC) for each module. (GC is the sum of the number of lines added and deleted. A modified line is reported as one added line and one deleted line.) The predictive quality of a multiple regression model from the principal components of software metrics was compared to a neural network trained with a set of principal components. (Principal components analysis is a statistical technique to reduce the dimensionality of a multivariate data set. This technique identifies and reduces the number of product metrics used to construct regression models.) The neural network model provided more accurate predictions of GC than did the multiple regression model. This suggests that there is some nonlinearity in the relationship between software complexity metrics and GC. Results indicate that the neural network model may be useful as a tool for diagnosing software maintenance..

Khoshgoftaar et al. [33] describe the concept of module ordering using data on code churn during development to predict rank-order of modules based on code churn due to fixes. A module-order model ranks modules according to some relative quantitative quality factor, such as development code churn. This module-ordering identifies modules that are likely to have problems. They recommend these modules should be considered for reliability enhancement. If process metrics from development are available early enough in the life cycle, this will enable reliability enhancement activities to prevent problems later in the life cycle.

Khoshgoftaar et al. compare the module-order model based on development code churn to two others; one based on product metrics alone and one that combined product metrics with development code churn. These three models were also evaluated in terms of their ability to classify fault-prone modules. The conclusions were:

1. Based on product metrics, larger modules had higher debug code churn, measured as the amount of new or changed lines of code due to bug fixes.

2. Based on development code churn, modules with more changes during development had higher debug code churn.
3. The combined model confirmed the conclusions of the other two models.
4. In looking at the module-ordering, the results showed that:
  - (a) Models based on product metrics were poor.
  - (b) The development code churn model and the combined model had good performance and both were robust. Even though their predictor uses an arbitrary cutoff percentile, they found that the performances for different cutoffs were nearly the same.
5. In regards to fault-prone classification, where fault-prone is defined as the top quartile, the results showed that:
  - (a) Models based on product metrics were worse than the other two models at classifying fault-prone modules.
  - (b) The development code churn model and the combined model were better, with development code churn best. (Product metrics introduced noise in the combined model for a range of percentiles).

The authors determined that module-order models are appropriate when a threshold to define fault-prone modules cannot be determined.

Eick et al. [18] look at change management data to assess code decay. They provide examples of causes, symptoms, risks and indicators of code decay. Examples of causes include: inappropriate architecture, violations of original design principles, imprecise requirements, time pressure, inadequate tools, especially for change process, as well as individual and organizational variability. Symptoms of code decay of interest here include a history of frequent changes, a history of faults, and widely dispersed changes. Risk factors are not causes or indicators of decay, but are of concern. They include: large module sizes, age of code, complexity, personnel and organizational change, and requirement changes.

Eick et al. [18] investigated several measures using change data as potential code decay indicators. They included number of files touched by a change request (span), number of lines added and number of lines deleted for a change request, number of delta changes associated with a change request, the date of changes, the time required to implement a change request, and the number of developers implementing a change. Results show that the span of changes increases over time, providing clear evidence that the span indicates code decay. In addition, span accompanied or possibly contributed to a breakdown in the modularity of code. Changes with large spans tended to require more effort. Fault analyses identified change as a causal element for faults, but did not identify modules with more decay. The authors also investigated the weighted time-damp model proposed in [25], and discussed in Section 2.4, with respect to their set of data. The results showed that large recent changes to a module add to fault potential and are a better predictor than its size. The investigation also provided evidence that some modules are more decayed than others, as measured by the number of changes to the module in the past, the age of their changes and their sizes.

The authors made four specific conclusions. Over time, span (the number of files touched) increases for each change request. Decrease in modularity can be measured by increases in number of modules touched by a change request. Frequency and recency of change contributes to module fault rate. Span and size of change are predictors for effort to implement a change.

Khoshgoftaar et al. [31, 32] incorporated module reuse as an additional independent variable in a software quality model based on product metrics. Software components may have different development histories that affect quality levels. Modules may be reused with modification, without modification, or be new. Reused modules have more testing and operational use than new modules, so are expected to have better reliability. Based on the case study [31], reuse measures were shown to significantly enhance classification models that identify fault-prone modules. Reuse measures also predicted the number of faults in each module better. In [32], the authors also apply informed prior probabilities of classifications to reduce the number of misclassifications for fault-proneness of modules. Informed prior

probabilities are probabilities determined by beliefs or data that is available prior to building the model. The misclassifications considered include misclassification of fault-prone modules (fault-prone modules classified as not fault-prone) and misclassification of not fault-prone modules (modules that are fault-prone, but not classified as such). The study showed that classification models can benefit by considering informed prior probabilities and costs of misclassifications. Unfortunately, the data in our case study does not identify modules that are reused with modification.

The studies in [30, 33, 41] use product metrics as predictors of fault-proneness or gross change. The metrics include number of if statements, number of operands, and McCabe's complexity measure. There is a fundamental concern with the use of these measures and with what they are supposed to measure. These metrics are all correlated to size. These studies also use change metrics as independent variables to predict fault-proneness or gross change. It is possible that larger files will have more changes, and size may be correlated to change measures. Another problem is the assumption in these three papers that change is bad. These models assume change is a potential source of future problems, when it could have been a result of a reengineering effort that made the system better than before.

The studies in [25, 18] consider other change metrics that include the number of changes to a file, the number of files touched by a change, the effort of a change (in terms of time required to implement the change), as well as the age of a change. The assumption is that the more files involved in a change, the more often a file is changed and the more effort it takes to implement the change. All are indications of a break-down in modularity or how difficult it is to fix a file correctly the first time. This may in turn indicate how fault-prone a module is.

Why a file is changed should be taken into consideration, because one cannot assume that change is bad. Methods for analyzing defect and change data can be used to predict components that are fault-prone and in need of more testing. These components may have the following characteristics:

- They have a high number of defects in prior releases.
- They are new.

- They have a large amount of modified code.
- They have been recently modified.

The methods presented identify components that may be problematic. Focusing attention on these components during testing may improve the effectiveness of testing by uncovering more defects in test rather than after release when they are more costly to fix.

Defect and change databases often do not record detailed change data (number of lines added, deleted or changed). Lack of this kind of data rules out using methods described in [30, 31, 32, 33, 18, 25]. Instead, other approaches must be used.

## 2.2 Fault Architecture

One can build a fault architecture in two ways:

1. Use an existing architecture and mask the components and relationships that are not fault-prone using the same measures of fault-proneness in [41, 65, 66].
2. If an existing architecture does not exist, reverse architecting techniques may identify it by using fault data extracted from the system.

### 2.2.1 Reverse Architecture

Reverse architecting is a type of reverse engineering. Tilley et al. [62] describe an approach to reverse engineering that is used in aiding program understanding for software evolution. The process of reverse engineering identifies system components and their dependencies and generates abstractions of them to make them more understandable. This involves extracting artifacts from the source code (or in our case, the defect data) and representing them in a manageable form so the system's structural and functional characteristics can be analyzed.

A reverse engineering approach should consist of the following phases [62]:

1. **Extraction Phase:** This phase extracts information from sources such as source code, documentation, and documented system history (e.g., defect reports and change management data).

2. Abstraction Phase: The phase abstracts the extracted information based on the objectives of the reverse engineering activity. The potentially very large amount of extracted information is distilled into a manageable amount.
3. Presentation Phase: This phase presents the abstracted information into a representation that is understandable to the user.

The purpose of the reverse architecting activity drives what information is extracted, how it is abstracted, and presented. If we are interested in a high level architecture of the system, then we would not want to extract too much information during phase 1, otherwise there would be too much information to abstract.

By applying reverse engineering techniques, Krikhaar [34] derives an architectural model for several complex systems using metrics that measure import relations (e.g. `#include` statements) and use relations (e.g. call statements, type, constant and variable usage). The architecture is a description of the system and its components, as well as their relationships. The main goal is to create a representation of the system at a higher level of abstraction to provide a better understanding of the architecture to more easily assess and identify parts of the system that may require maintenance or enhancement.

Reverse architecting concerns activities that make software architectures explicit using reverse engineering techniques. Reverse architecting follows the reverse engineering phases. Depending on the required models, the appropriate steps are executed. The approach consists of three steps:

1. Extract the import relations from files, which are assigned to subsystems by use of a directory structure (creating part-of relationships). Import relations are then derived at the subsystem level as follows: If two files in different subsystems have an import relationship, the two subsystems to which the files belong have one as well. Results are then presented.
2. Analyze the part-of hierarchy for the system (e.g. files, clusters, subsystems) in general. The part-of relations are extracted at file level and derived for the various levels. The results are presented at various different levels of abstraction.



3. Extract and analyze use relations at the code level. Use relationships include not only import statements, but call and called-by relationships, and global and shared variables, constants and structures. These and other use relations are best extracted using source code parsers. Analogous to part-of relations, the use relations are abstracted for various levels and the results are presented.

There are many ways to adapt this framework to a specific reverse engineering architecting objective. One way, for example, identifies fault-prone relationships between components.

Feijs et al. [20] describe a relational approach to support the analysis of software architectures. The relational approach supports many techniques that apply to analysis and structuring tasks, including:

- Lifting. Import and use relations at a higher level are obtained by union of import and use relations at a lower level according to the part-of relation.
- Checking rules. For example, typically each .c file in a C program must import at least one .h file; and it is not allowed that an .h file import other .h files (unless they are libraries).
- Impact analysis that allows one to determine which files require re-test as the consequence of changing a component.
- Finding unused and unavailable components.
- Studying alternative structures.
- Identifying components in top and bottom layers.
- Checking for cycles in uses relationships. This allows one to check for dependencies in software layers. For example, higher levels may use lower layers, but lower layers may not use higher levels.

The extraction-abstraction-presentation model is useful in reverse engineering. It allows visualization of the architecture at the highest level, while making sure that the high level

views and the real system are related in an accurate way. This makes it easier to perform improvement activities. At any point in time, it is possible to see and correct the evolving system's structure.

Gall et al. [22] use a reverse architecting technique to identify the logical coupling of modules using change data across releases. Their work differs from other work in that they build an architecture based on logical coupling using change data rather than syntactical coupling, which is usually based on code or design. Modules are logically coupled if they have identical change behavior during software development. Their approach consists of two processes: the first identifies change patterns among modules and the second reveals dependencies hidden among them. A change pattern is an observed pattern in change sequences, where a change sequence is a sequence of releases in which a module has been changed. Observed patterns in change sequences define potential logical coupling among specific modules. Patterns with long subsequences indicate stronger coupling. The logical couplings are verified by further examining change reports of modules with the same change sequence for causes of the changes. If the change reports identify the same reason for the change, then the logical coupling is verified. This can be easily described using graphs, where nodes represent subsystems and weighted edges represent the amount of coupling. Determination of interrelationships among modules aids in identifying modules that should undergo restructuring, re-engineering or in our case more testing. The advantage of this technique is that it does not require analyzing millions of line of code, but instead analyzes change data for the release, which is more manageable and usually available.

### 2.2.2 Fault Architecture Technique

Von Mayrhauser, et al. [40, 65] developed an adaptation of reverse architecting based on the need to represent defect relationships between components and the ability to focus on the most problematic parts of the architecture by quickly filtering out information.

Methods in [40, 41, 65] combine prediction of fault-prone components with code decay analysis. They look at relationships between components and identify the relationships that are fault-prone to indicate underlying systemic architecture problems.

In [40], the authors identify software components that are fault-prone across releases to analyze code decay. Components are fault-prone based on the number of times they have been fixed. The study looked at four releases and included 130 components of a large system software product. Some of the 28 measures collected include:

- number of changed files in each release.
- number of defect reports in each release.
- average number of files changed in a component per defect.
- number of unique files changed for each component.
- number of defects with more than one component changed.
- changes in size.

The approach [65] consists of the following steps :

1. Identify fault-prone components. Apply GYR [41] to identify fault-prone components over successive releases. Components with problems in several releases indicate possible code decay. The authors consider a component fault-prone, if it is among the top quartile in terms of defect reports in a given release. Several factors determine the threshold chosen. One of the goals is to provide a manageable amount of data.
2. Create a fault component directory structure. A component is a collection of files within the same subdirectory. The directory structure of the software provides the “part-of” relationship. Leaves in this structure are the fault-prone components identified by GYR in step 1. Internal nodes represent subsystems (subdirectories at higher levels in the directory structure) that contain fault-prone components. Since only fault-prone components are included, this does not represent the entire directory structure. This directory structure is referred to as a *Fault Component Directory Structure*.
3. Determine fault-prone relationships. How many fault relationships a component has with others is based on the number of other components involved in the same defect

fix. If the number of fault relationships is quite high, narrow the number of components with fault relationships. This study set the threshold to the top 10 percent components. Fault relationships among these components that are above a threshold set to an order of magnitude (10%) less than the largest number of fault relationships are considered fault-prone.

4. Create a fault architecture diagram for each release. Abstract fault relationships to the next higher subsystem level. Subsystem levels are based on the level of depth in the *Fault Component Directory Structure*. Two subsystems are fault related, if they contain components that are fault related. (This represents Krikhaar's second step [34].) Nodes represent components. Arcs between two nodes show that components are fault-prone in their relationship. Weights on the arcs indicate the number of defect reports associated with two components or subsystems.

The resulting fault architectures indicate for each release the components that have the most fault relationships. Changes in patterns or persistent fault relationships between components across releases indicate systemic problems related to components and system architecture.

5. Aggregate the fault architecture diagrams into a cumulative release diagram. Nodes in the cumulative release diagram aggregate nodes that occur in at least one fault architecture diagram. Two nodes are related, if there is a fault relationship between corresponding nodes in at least one fault architecture diagram. Annotations on the edges indicate the releases in which the relationships had problems.

A small number of components were identified as problematic in terms of having a high number of fault relationships. Results of the case study showed trends across releases. The study identified problematic component relationships. The cumulative release diagram identified persistent problems. Fault architecture diagrams and cumulative release diagrams draw attention to fault relationships that may need corrective maintenance, re-architecture, or more testing.

### 2.2.3 Defect Cohesion and Defect Coupling Measures

In design, cohesion is a measure of the internal consistency within components. It is a single component measure - an attribute of individual modules describing the extent to which the individual parts of a component perform the same task. In testing, defect cohesion refers to a measure that quantifies the number of individual parts of a component, e.g. files, that had to be changed to correct the same defect report. Defect cohesion measures the fault-proneness internal to a component.

In design, coupling measures the degree of interaction between components. Two components are coupled when parts of one component use parts of another. In testing, defect coupling measures the degree of interaction of components in terms of the work needed to repair the same defect. Two components are related or “coupled” if some or all of their files are changed to repair a defect. Defect coupling measures the fault-proneness of the relationship between components.

Whereas high cohesion and low coupling desirable in design, low defect cohesion and low defect coupling are desirable in testing: They indicate a lower degree of fault-proneness internal to and between components.

Files are changed (or fixed) for components in response to defects reported. A fault relationship between two components exists, if they are both involved in defect repair for a given defect, that is if files belonging to both components were changed to repair a single defect. If two files belonging to the *same* component were changed, the defect was *local* to the component. Otherwise, it represents a defect that is common to both components and it is referred to as a *relationship* defect. Two measures used to determine the fault-proneness of components and component relationships are:

- A local measure,  $Co_{\langle C \rangle}$ , used to measure defect cohesion for component,  $C$ .
- A relationship measure)  $Re_{\langle C_j, C_k \rangle}$ , used to measure defect coupling between two components,  $C_j$  and  $C_k$ .

Von Mayrhauser et al. [66] provide variants of the defect cohesion and defect coupling measures to more clearly distinguish defect fixes that involve single versus multiple file

changes in a component. These measures are more sensitive to the number of files changed in each component to fix a given defect. They also present several options for abstracting the extracted fault-coupling relationships between components to the subsystem level.

### 2.2.3.1 Defect Cohesion Measures

The defect cohesion measures are defined as follows:

- Basic Defect Cohesion Measure:

The basic defect cohesion measure for component  $C$  is defined as:

$$Co_{\langle C \rangle} = d \quad (2.1)$$

where

$d$  is the number of defect reports written against a component.

- Multi-file Defect Cohesion Measure:

Merely counting defects does not differentiate between a defect in a component that requires modifying one file or many files. If we assume that a defect is more complex when its repair involves more of the component's files, a more detailed defect cohesion measure is needed.

Multi-file defect cohesion counts the pairwise defect relationships between files within the same component,  $C$ , where those files were changed as part of a defect repair:

$$Co_{\langle C \rangle} = \sum_{i=1}^n C_{i\langle C \rangle} \quad (2.2)$$

where

$$C_{i\langle C \rangle} = \begin{cases} 1, & f_{d_i} = 1 \text{ and only 1 component is involved in fixing } d_i \\ \frac{f_{d_i}(f_{d_i}-1)}{2}, & f_{d_i} \geq 2 \end{cases}$$

and

$f_{d_i}$  is the number of files in component  $C$  that had to be changed to fix defect  $d_i$ .

$n$  is the number of defects that necessitated changes in component  $C$ .

This provides an indication of local fault-proneness. Unless only one file is changed, this measure will be much larger than the basic cohesion measure. It penalizes components that are only involved in a few defects, but where each defect repair involved multiple files.

### 2.2.3.2 Defect Coupling Measures

The defect coupling measures are defined as follows:

- **Multi-file Defect Coupling Measure:** Two or more components are fault related, if their files had to be changed in the same defect repair (i.e. in order to correct a defect, files in all these components needed to be changed).

For any two components  $C_1$  and  $C_2$ , the relationship measure  $Re_{\langle C_1, C_2 \rangle}$  is defined as:

$$Re_{\langle C_1, C_2 \rangle} = \sum_{i=1}^n C_{1d_i} \times C_{2d_i}, \quad C_1 \neq C_2 \quad (2.3)$$

where

$C_{1d_i}$  and  $C_{2d_i}$  are the number of files in components  $C_1$  and  $C_2$  that were changed to fix defect  $d_i$ .

$n$  is the number of defects that necessitated changes in components  $C_1$  and  $C_2$ .

- **Cumulative Defect Coupling Measure:** A component  $C$  can be fault-prone with respect to relationships if none of the individual defect coupling measures are high, but there are a large number of them (the sum of the defect coupling measure is large). In this case the defect coupling measure for a component  $C$  is defined as:

$$TR_C = \sum_{i=1}^m Re_{\langle C, C_i \rangle} \quad C \neq C_i \quad (2.4)$$

where

$m$  is the number of components other than  $C$ .

$Re_{\langle C, C_i \rangle}$  is the defect coupling measure between  $C$  and  $C_i$ .

The multi-file defect coupling measure emphasizes pairwise coupling related to code changes for defects involving a pair of components. The cumulative defect coupling measure is concerned with components in many fault relationships with two or more components.

The primary use of these measures is to provide an ordinal scale to rank components. The exact values of the measures or the range of values is secondary. Ohlsson et al. [40] use the values to rank the components and then identify the top 25 percent in this rank order as fault-prone. The multiplicative nature of the multi-file defect cohesion and coupling measures magnifies individual differences among components. This accentuates differences between components that are close in measurement values for the defect cohesion and defect coupling measures and reduces ties in ranks.

#### **2.2.4 Determining Fault-Prone Components and Fault-Prone Relationships**

The basic strategy in [40, 65, 66] uses defect cohesion measures for components and defect coupling measures between components to assess how fault-prone components and component relationships are. If the objective is to concentrate on the most problematic parts of the software architecture, these measures are used with filters to identify

- the most fault-prone components only (setting a threshold based on the defect cohesion measure);
- the most fault-prone component relationships (setting a threshold based on the defect coupling measure).

Von Mayrhauser et al. [65] consider a component fault-prone in a release if it is among the top 25 percent in terms of defect reports written against the component. In general, one would set the threshold based on available resources, quality, and objectives of the analysis (most problematic versus all components that have problems). The 25 percent threshold provided a manageable number of problematic components for further analysis. Similarly, a threshold may distinguish between component relationships that are fault-prone and those that are not. The threshold was set to an order of magnitude (or ten percent) less than the maximum value for the defect coupling measure. Setting the threshold is a



subjective decision and depends on the objectives of the investigation and the number of fault relationships.

Von Mayrhauser et al. [66] determine defect cohesion and defect coupling measures for all components. Then they filter based on the defect coupling measure to focus on the most problematic relationships. A component  $C$  can be fault-prone with respect to relationships for two reasons:

1. The defect coupling measure is high for a particular pair of components  $\langle C, C_i \rangle$ .
2. None of the individual defect coupling measures are high, but there are a large number of them (the cumulative defect coupling measure is large).

It is this second reason that prompted them to determine a threshold based on the sum of the defect coupling measures for a component. In their study, the threshold for including a component in the fault architecture is set as 10 percent of the highest  $TR_C$  measure. The threshold for including a fault relationship in the fault architecture is set at 10 percent of the highest  $Re_{\langle C, C_i \rangle}$ .

These two thresholds identify the components and component relationships that provide the lowest level of the *Fault Architecture*. The fault architecture may have components with a high  $TR_C$  measure, but a low  $Re_{\langle C, C_i \rangle}$  measure. In this case, the component has no fault-prone relationships, to denote the second situation above.

The data available in our study enables fault-architecture analysis for several releases. This study, therefore, replicates the studies in [41, 65, 66]. This study also explores the filtering and threshold setting techniques in those studies. In addition, this study performs fault-architecture analysis for the development, system test and post-release phases of each release, since data collected in this study identifies the phase in which a defect was reported.

## 2.3 Methods that Use Prioritization to Improve Efficiency

Stopping rules may improve testing efficiency by helping software developers determine when testing should stop. Potentially, weeks of testing may be saved. Another way to improve testing efficiency is to test components in a different order, so that parts of the

software that are more likely to have faults are tested earlier. Already, fault-prone component analysis and fault architecture models have been described to help in identifying these software parts for more intense testing, but often these parts can also be tested *earlier*. Assigning priorities to testing activities based on how fault-prone components are should improve the rate of fault detection. If more defects are found earlier, defects not only have a better chance of being fixed before release, but fewer defects will be left to uncover late in testing, so testing can stop sooner, saving more weeks.

There are two ways to prioritize test activities. They are: prioritize test cases [48] or prioritize testing of components [33].

### 2.3.1 Prioritize Test Cases

Several techniques are described in [48] for ordering the execution of test cases during regression testing to maximize some objective function, such as the fault detection rate. Test cases may be scheduled not only to improve the rate of fault detection, but may also be scheduled to achieve a faster rate of code coverage, to exercise features in order of expected frequency of use, or to exercise components in order of historical failure rate. Rothermel et al. [48] are concerned with techniques for prioritizing test cases for the purpose of improving the rate of fault detection. Nine test prioritization schemes were investigated:

1. No prioritization (an untreated test suite used as a control).
2. Random prioritization.
3. Optimal prioritization (using program with known faults in experiment as an upper bound for comparison purposes ).
4. Total branch coverage prioritization.

For any test, the number of decisions(branches) exercised in a program can be determined. These tests can be prioritized according to the number of branches they cover by sorting them in order of total branch coverage achieved.

5. Additional branch coverage prioritization.

Having executed a test and covering certain branches, more branch coverage may

be gained in subsequent tests by covering branches that have not yet been covered. Additional branch coverage prioritization iteratively selects test cases that yield the greatest branch coverage, then adjusts the coverage information on subsequent tests to indicate their coverage of branches not yet covered. This process is repeated until all branches have been covered by at least one test case.

6. FEP-total (total fault-exposing potential) prioritization.

Approximation of the fault-exposing potential (FEP) of a test case is obtained using mutation analysis.

7. FEP-additional prioritization.

Total FEP prioritization is extended to create additional-FEP prioritization in a method analogous to the extensions made to total branch coverage prioritization to additional branch coverage prioritization.

8. In order of total statement coverage.

9. In order of additional statement coverage.

Rothermel et al. [48] applied these nine techniques to seven programs. The measurement used to assess and compare the techniques was a weighted average of the percentage of faults detected over the life of the test suite.

Results show that optimal prioritization greatly improved the rate of fault detection and that all the prioritization techniques showed significant improvement compared to the test suite without prioritization. The FEP-based prioritization techniques performed better than all others, although not significantly. Considering the expense of the FEP-based assessment technique they are using, the technique may not be cost effective. Other FEP-based techniques, however, might be cost effective. Total-branch and total-statement coverage performed better than their more expensive additional counterparts.

Results show that test case prioritization can improve the rate of fault detection. Techniques that incorporate static measures of fault-proness may provide improvements and warrant investigation. Little research has been done to prioritize test cases based on process data.

### 2.3.2 Prioritizing Modules for Testing

Methods have been described to identify components that are fault-prone which may then be tested earlier. In [33], the authors suggest that module-ordering may recommend modules that are likely to have problems for reliability enhancement. If defect metrics from development are available early enough in the life cycle to can be used by testers to uncover more defects, problems in post-release may be reduced. Unfortunately, little empirical research has been done in this area. Validated methods that prioritize modules in testing are needed.

## 2.4 Defect Estimation using Static Models

Several approaches that use process metrics exist to estimate defect content. First, it is possible to build prediction models from historical data. This approach is referred to as experience-based, since it is based on building models from data collected previously. Studies in [4, 5, 13, 25, 78] predict the number of defects either within one release or between releases. Some experience-based approaches that estimate fault content are briefly described in Section 2.4.1.

Second, prediction models can be built using various statistical methods using data available only from the current project [6, 8, 10, 17, 19, 59, 64, 69, 70]. This approach is used to estimate the fault content with data from the current project, and hence the methods are more direct. Two types of models can be used for this approach, namely capture-recapture models [17, 10, 64, 70, 6, 8, 19, 59] or different curve-fitting approaches [69, 8, 59]. These models can briefly be described as follows:

- Capture-recapture models, i.e., models using the overlap and non-overlap between reviewers (or test sites in our case) during defect detection to estimate the remaining defect content. The models have their origin in biology where they are used for wildlife population estimations and management [10, 44]. Two or more independent counters study a population in a specific area and count the animals they encounter. If the number of animals seen by several counters is large, then the counters have probably covered most of the population. If the overlap is small, then there are probably many

animals remaining that were not counted. The same principle is applied to “bugs” in software systems.

- Curve fitting models, i.e., models that plot test data from the test sites and fit a mathematical function. The function is then used to estimate the remaining defect content [69].

These two types of models are discussed in more detail in Sections 2.4.2 and 2.4.3. Methods that integrate experience are described in Section 2.4.5.

Our main focus is on using the capture-recapture and curve-fitting methods to estimate the number of components with defects in post-release that showed no defects in testing. This is different from, but complements, the approach taken when identifying fault-prone components. The objective is to not only estimate fault-proneness as such, but to also estimate the number of components that seem fine during testing but exhibit problems during operation. This estimate can be used as an additional criterion to determine when it is suitable to stop testing and release software.

### 2.4.1 Experience-based Methods

Methods that use historical data for predicting the number of defects remaining in components may be based on defect data [5, 78] or code change data [4, 13, 25]. The data may come from either a prior phase within the release or from prior releases. Table 2.3 summarizes the studies described in this section. It identifies the assumptions, the dependent and independent variables used in the models, as well as the techniques used in the studies. All these models [4, 5, 13, 25, 78] assume that repair is imperfect.

Yu et al. [78] explore the relationship between the number of faults per module and the prior history of a module. They investigate two software defect models and propose a revised model to estimate the number of remaining defects in software. All three models use the number of defects detected in earlier phases of the life cycle, such as design reviews, code inspection, and unit test, to predict the number of defects found later in system test and post-release.

Table 2.3: Studies on defect estimation methods based on defect or change data.

Study	Assumptions	Dependent Variable	Independent Variables	Technique	Results
Yu et al. [78] (Case study: 2 similar systems)	Repair imperfect; Defect removal iterative w/i phase; Defect detection ratio decreases each iteration; No new functionality after unit test.	# remaining defects at end of phases w/i system	# defects found  # defects repaired	Linear Regression	Model building worked for study.
Biyani et al. [5] (Case study: 4 releases)	Repair imperfect.	# post release faults per module	# development defects per module in prior releases	Correlation	Exploratory: Showed relationships exist.
Basili et al. [4] (Case study)	Repair imperfect	Error Proneness (#defects)	Module size New or modified module	Least Squares Linear Regression	Module type good predictor.
Christenson et al. [13] (Simulation & Case study of switch systems)	Repair imperfect; Bad fixes and fixes-on-fixes can be measured; 1st and 2nd order faults have equal likelihood of being found.	# remaining defects	# fixes on fixes	Simulation: seed errors sampling  Case study: correlation	# Fixes-on-fixes is a good predictor.
Graves et al. [25] (Case study)	Repair imperfect; Error distribution Poisson.	# remaining defects in a module	# file changes in a module	Generalized linear regression models	Weighted time-damp model best.

The first model described by Yu et al. [78] uses linear regression analysis to determine if a relationship exists between earlier defects and later defects. Results of their study using two data sets show that a very strong correlation exists between earlier and later defects, suggesting that the number of earlier defects is a good predictor of the number of later defects. Unfortunately, the authors could not explain the physical meaning of the parameters for the regression equation. This would make this model difficult to adapt to other environments or projects. The second model described by Yu et al. [78] assumes defects introduced in one phase as the result of a defect repair will not be detected until a later phase. It uses the number of defects found and the number of defects repaired in a phase to estimate the number of defects remaining at the end of the phase. Parameters include the defect detection ratio (ratio of the number of detected defects to the number of defects during the previous phase) and the defect correction ratio (ratio of the number of correctly fixed defects to the number of detected defects). Methods to obtain the number of defects during the previous phase are not given.

Yu's model [78] is a revision of the second model. Yu's model assumes that the defect removal process is iterative, that is, that defects introduced during a correction can be detected later in the same phase. This model assumes that the defect detection ratio decreases after each iteration, due to a decrease in available test time and successive repair. The model performed well on the two data sets used. Including only the first two iterations in the model resulted in good accuracy, since defects found and fixed dropped off rapidly. Other studies provided data to estimate the model's parameters. These values may only be of use if the environments of the other studies are similar. The process used by developers and testers in the projects used in this case study are more accurately modeled by the third model by Yu et al. [78]. The models described in [78], however, require no new functions be added after unit test. The process used in the projects in our case study involved new functionality that was added in "drops" during system test; that is, several new components were added to the system and delivered to system test during the course of system testing. This is probably a fairly common procedure in industry.

Biyani, et al. [5] explore the relationship of the number of faults per module to the prior history of a module. Specifically, they use faults discovered in development to predict faults remaining in the field. They also use faults discovered in previous releases to predict faults in the current release. Their method attempts to correlate the number of defect-free modules in the field to the number of defects found in development, as well as the average number of defects found in the field per module to the average number of defects found in development per module. In their study of four releases of a commercial software product, modules with more development defects tended to have more field defects. In addition, in comparing defects in a current release to those in previous releases, they concluded that the prior release is sufficient for predicting the number of defects during development or in the field. In their study, defect data from development and data from the prior releases are good measures for assessing the relative quality of software.

Unlike Biyani, et al. [5], this thesis is concerned not so much in predicting the number of defects between development and post-release and across releases, but in how defect data can be used to improve system testing, in both its effectiveness and efficiency. In this,

various aspects of assessment and prediction are combined, and the data is used to support suggested improvement activities.

Change data is used as the basis of making predictions about defect content in [4, 13, 25]. Basili and Perricone [4] describe a case study that uses four releases of a satellite planning project to explore relationships between frequency and distributions of errors versus software size, software complexity, and reuse. They discovered that 89 percent of the repairs involved changing only one module. In addition, the largest source of errors was attributed to the specification/requirements phase with the majority of these errors involved in modified modules. This might indicate that specification were not well enough defined for reuse. New and modified modules differed in the types of defects and the efforts to repair them. New and modified modules, however, behaved similarly with respect to the number of defects, with modified modules contributing slightly more errors. In their study, a higher error rate existed in smaller sized modules. (They also found that cyclomatic complexity was correlated with module size.) They interpreted this result to mean that either the number of modules examined was too small, hence the study caused a biased result, or developers took more care in developing larger modules.

Christenson and Huang [13] investigated a “fix-on-fix” model to predict the number of defects remaining in software. A “fix-on-fix” is a software change to fix a defect introduced in an earlier software fix. The model assumes that defect repair is imperfect and may result in “bad fixes” (fixes that are not defect free). The authors refer to defects in newly developed software as “first-order faults” and defects in fixed software as “second-order faults.” Fixes-on-fixes are software changes needed to repair second-order faults. The number of defects remaining in software is a function of the number of “fixes-on-fixes”. Assuming each fault has an equal probability of being found, the set of faults found and repaired is considered as a statistical sample of the total fault population. In this case, the ratio

$$\# \text{ of fixes - on - fixes} / \# \text{ all fixes}$$

in the sample is used to estimate the ratio

$$\# \text{ 2nd order faults} / \# \text{ all faults.}$$



If the first ratio can be estimated, then one can predict the number of faults remaining. In a simulation, Christenson and Huang [13] used error seeding and statistical sampling to estimate the first ratio. A case study used several past releases of a large switching system to derive the first ratio. The “fix-on-fix” model performed reasonably well in predicting the number of remaining defects. The fewer the number of defects in fixed code, the fewer the number of remaining defects in a module. Measurements for bad fixes and “fix-on-fixes” are often not available.

Graves, et al. [25] developed a model that attempts to predict the number of defects found within a module based on the file changes made to the module. They also compared their model to several other models. First, they investigated what they call the “stable model.” This model assumes that the number of future faults in a module is a constant multiplier of the number of faults found in the module in the past. This indicates that testing intensity is not an important factor. It turns out that the stable model is a good model, with the advantage that it is simple. Two other models they investigated did not seem to be successful. The first unsuccessful model used the number of developers who worked on or changed a module. The other unsuccessful model was concerned with the extent to which a module is connected to other modules as measured by the number of other modules changed together with the module. Using these measures did not help in predicting the number of faults. In addition, models based on product measures, such as size, complexity, etc., did not perform any better at predicting the number of faults than size alone.

The models Graves et al. [25] found successful involved measures of changes to code within a module. The number of changes to the code in a module over its entire history, a touch count, appeared to be a successful predictor of faults. In addition, when large, recent changes are weighted more heavily than smaller, older changes, the model improved. Older modules with the same number of defects appear to be lower in the number of faults. They concluded that delta changes to code are a much better measure of fault likelihood than number of lines of code. The consideration of age improves the model. The model assumes new faults are continuously being added to the system as changes are made to it.

The change data available in the case study in this thesis does not include all changes made to the source code. It only includes information on changes made to repair a defect. In addition, it does not include data to measure the number of lines added, modified and deleted. This means that the experience-based methods based on detailed code change data cannot be used.

## 2.4.2 Methods based on Capture-Recapture Models

The major application of capture-recapture models in software engineering has so far been during the review or inspection process. Different capture-recapture models use different assumptions regarding reviewers and defects. Reviewers may have the same or different ability of finding defects. Defects themselves may be equally difficult to find or vary in how difficult they are to detect. Thus, capture-recapture models address four different situations:

1. Reviewers are assumed to have the same ability to find defects, and different defects are found with the same probability. This type of model is denoted  $M_0$ . It neither takes variations in the reviewers' ability nor in the detection probabilities into account.
2. Reviewers are assumed to have the same ability to find defects, though different defects are found with different probabilities. This type of model is denoted  $M_h$  (variation by heterogeneity<sup>1</sup>). It takes the detection probabilities into account, but not the reviewers' ability.
3. Reviewers are assumed to have different ability to detect defects, and all defects are found with the same probability. This type of model is denoted  $M_t$  (variation by time<sup>2</sup>). It takes the reviewers' ability into account, but not varying detection probabilities.
4. Reviewers have different profiles for detecting defects, and different defects are found with different probabilities. This type of model is denoted  $M_{th}$  (variation by time and

---

<sup>1</sup>The use of the words heterogeneity and time has its origin in biology.

<sup>2</sup>See footnote 1.

heterogeneity). It takes both variations in the reviewers' ability and in the detection probabilities into account.

In addition to the assumptions of each model regarding detection probability by defect and by reviewer, it is assumed that the reviewers work independently.

The assumptions for the four types of models are illustrated in Figure 2.1 for five defects and three reviewers. The plot is similar to a figure in [7]. The heights of the columns represent detection probability. The probabilities in the figure are for illustration purposes

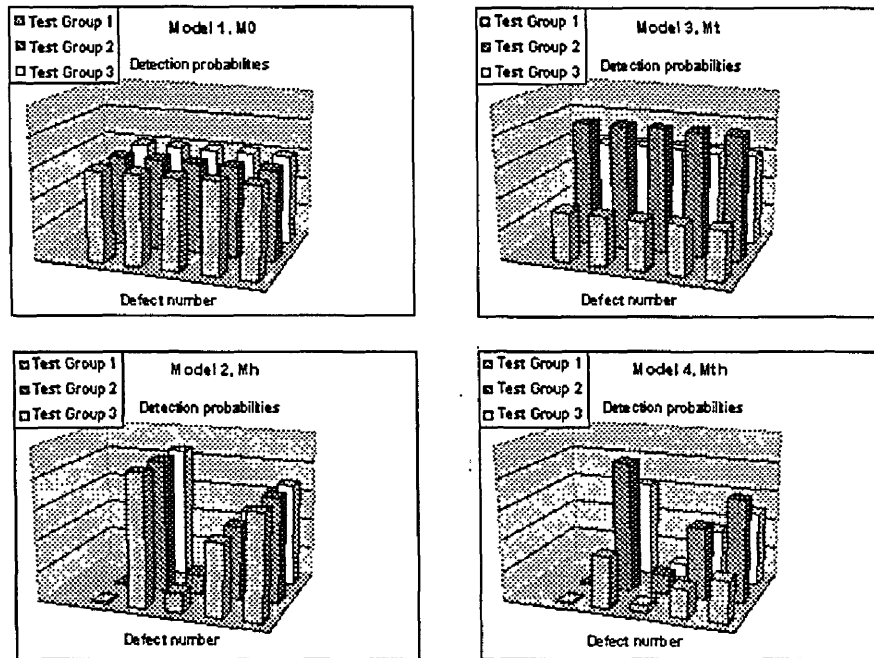


Figure 2.1: An illustration of the different types of capture-recapture models.

and not actual values. Clearly, the model of type 4 is the most realistic. It also requires more complicated statistical methods and it is more difficult to get stable estimates.

Eick et al. [17] combine results of two or more independent reviewers of requirements documents and design documents with a statistical inference method, capture-recapture, to draw conclusions about the remaining number of defects. Two or more independent reviewers inspect a document. If many of the defects were found by more than one reviewer, then most of the defects have probably been found. If, however, very few defects were found

by two or more reviewers, then there are probably many defects that have not yet been found.

The various capture-recapture models used in software engineering estimate the total number of defects. The estimated number of remaining defects can be determined by subtracting the number of defects found from the estimated total. One of the major differences between capture-recapture models are the assumptions made about the probabilities of detecting a defect. In addition, several statistical approaches have been applied to capture-recapture models for the purpose of estimating the total number of defects. [17, 49, 64, 70] apply the maximum-likelihood estimator (mtml). The Chapman estimator for the Mt model (mtChpm) is used in the case of two reviewers [19]. The jackknife method (mhjk) has been applied and compared to the mtml in [6, 64]. Briand et al. [6] investigated the Chao estimator for the Mth model proposed in [10] using a case study. Table 2.4 shows the capture-recapture models suitable for inspections [6] along with their estimators. Table 2.4 also contains references in which the models are described in more detail.

Table 2.4: Statistical methods for capture-recapture models.

Reviewer ability	Detection probabilities	
	Equal	Different
Equal	M0: Maximum-Likelihood (m0ml) [44]	Mh: Jackknife (mhjk)[44]
Different	Mt: Maximum-Likelihood (mtml) [44, 64] Chapman (mtChpm) [19]	Mth: Chao (mthChao)[10]

#### 2.4.2.1 Notation for Capture-Recapture Methods

Estimators for the capture-recapture models are discussed in the next section. The notation for these estimators are defined below.

$N$  is the actual number of defects in the inspected object.

$n$  is the number of observed defects in the inspected object.

$m$  is the number of inspectors.

$f_k$  is the number of defects found by exactly  $k$  inspectors.

$$S = \sum_{i=1}^m f_i.$$

$n_i$  is the number of defects found by inspector  $i$ .

$p_i$  is the probability of detecting a defect by inspector  $i$ .

### 2.4.2.2 Estimators for Capture-Recapture Methods

The maximum-likelihood method [64] is based on the assumption that all defects are found by a specific reviewer with equal probability. A simple example for the Mt model that uses only two inspectors follows. The equation is:

$$N = \frac{E(n_1)E(n_2)}{E(f_2)} \quad (2.5)$$

An estimator for the the number of defects can be derived as:

$$\hat{N} = \frac{n_1 n_2}{f_2} \quad (2.6)$$

This estimator is known as the Lincoln-Peterson Estimator [7].

The simplest of all models, M0, results from the assumption that defect detection probabilities do not vary by reviewer nor by individual defect. The following equation is maximized for the m0ml estimator [44]:

$$L(N) = \log \binom{N}{n} + \sum_{j=1}^m n_j \log \sum_{j=1}^m n_j + (Nm - \sum_{j=1}^m n_j) \log (Nm - \sum_{j=1}^m n_j) - Nm \log (Nm) \quad (2.7)$$

This function is maximized numerically over  $N \geq n$  to determine  $\hat{N}$ , the estimate for the number of faults. Subtracting the number of faults found at the review from the estimate of the total number of faults gives the estimate for the number of remaining faults. If most faults are found by two or more reviewers, then few faults are undiscovered. Otherwise, additional reviews are required to find more faults.

The Mt model assumes reviewers have different probabilities in finding defects. For the more general case of the Mt model where there may be two or more reviewers, the following mathematical equation is maximized [44]:

$$L(N) = \log \binom{N}{n} + \sum_{j=1}^m n_j \log n_j + \sum_{j=1}^m (N - n_j) \log (N - n_j) - Nm \log N \quad (2.8)$$

The following equation gives the Chapman estimator for the Mt model [19] (mtChpm) in the case of two reviewers:

$$\hat{N} = \frac{(n_1 + 1)(n_2 + 1)}{(n + 1)} - 1 \quad (2.9)$$

The jackknife method [64] may also be used to determine the total number of faults. It is based on the assumption that each reviewer has the same probability of finding a specific defect, while the defects are found with different probabilities.

Table 2.5: Jackknife estimators  $\hat{N}_{hk}$  for  $k = 1, \dots, 5$ .

$\hat{N}_{h0} = S$
$\hat{N}_{h1} = S + \left(\frac{m-1}{m}\right) f_1$
$\hat{N}_{h2} = S + \left(\frac{2m-3}{m}\right) f_1 - \frac{(m-2)^2}{m(m-1)} f_2$
$\hat{N}_{h3} = S + \left(\frac{3m-6}{m}\right) f_1 - \left(\frac{3m^2-15m+19}{m(m-1)}\right) f_2 + \frac{(m-3)^3}{m(m-1)(m-2)} f_3$
$\hat{N}_{h4} = S + \left(\frac{4m-10}{m}\right) f_1 - \left(\frac{6m^2-36m+55}{m(m-1)}\right) f_2 + \left(\frac{4m^3-42m^2+148m-175}{m(m-1)(m-2)}\right) f_3 - \frac{(m-4)^4}{m(m-1)(m-2)(m-3)} f_4$
$\hat{N}_{h5} = S + \left(\frac{5m-15}{m}\right) f_1 - \left(\frac{10m^2-370m+125}{m(m-1)}\right) f_2 + \left(\frac{10m^3-120m^2+485m-660}{m(m-1)(m-2)}\right) f_3$ $- \left(\frac{(m-4)^5-(m-5)^5}{m(m-1)(m-2)(m-3)}\right) f_4 + \frac{(m-5)^5}{m(m-1)(m-2)(m-3)(m-4)} f_5$

The jackknife estimator (mhjk) for the Mh model [9] estimates  $N_j$ .  $N_j$  is chosen as some  $N_{ji}$  as described below. Table 2.5 shows the formulas from [9] for  $\hat{N}_{jk}$  for order  $k \leq 5$ , where  $k \leq m$  is the jackknife order.

According to Burnham and Overton [9], in looking at the mean squared error of  $\hat{N}_{jk}$ , the unique minimum is usually achieved at  $k = 1, 2$ , or  $3$ .

$\hat{N}_j$  is chosen by testing sequentially the hypotheses

$$H_{0i} : E(\hat{N}_{j,i+1} - \hat{N}_{ji}) = 0 \quad (2.10)$$

versus

$$H_{\alpha i} : E(\hat{N}_{j,i+1} - \hat{N}_{ji}) \neq 0, \text{ for } i \leq 4 \quad (2.11)$$

Choose  $\hat{N}_j = \hat{N}_{ji}$  such that  $H_{0i}$  is the first null hypothesis not rejected. The test statistic is

$$T_i = \frac{\hat{N}_{j,i+1} - \hat{N}_{ji}}{\left(\frac{S}{S-1} \left(\sum_{i=1}^m a_i^2 f_i - (\hat{N}_{j,i+1} - \hat{N}_{ji})^2 / S\right)\right)^{1/2}} \quad (2.12)$$

where the coefficients for  $f_i$  in the formulas for  $N_{ji}$  in Table tab:jke are the constants  $a_{i1}, \dots, a_{ik}$ .

It is expected that the significance levels,  $P_i$ , will be increasing. If  $P_{i-1}$  is small, for example  $P_{i-1} \leq 0.05$ , and  $P_i$  is much larger than 0.05, then choose  $\hat{N}_j = \hat{N}_{ji}$ .

The Mth model assumes that the defect detection probabilities vary by reviewer and by individual defect. An estimator used for this model is the MthChao [10]. The mathematical formula for the MthChao estimator is:

$$\hat{N}_i = \frac{n}{\hat{C}_i} + \frac{f_1}{\hat{C}_i} \hat{\gamma}_i^2, \quad i = 1, 2, 3 \quad (2.13)$$

where

$\hat{C}_1 = 1 - f_1 / \sum_{k=1}^m k f_k$ ,  $\hat{C}_1$  is an estimator of the expected sample coverage, where sample coverage is defined as the total individual detection probabilities of the found defects.

$\hat{C}_2$  and  $\hat{C}_3$  are bias-corrected versions of  $\hat{C}_1$  and defined as:

$$\hat{C}_2 = 1 - \frac{f_1 - 2f_2 / (m-1)}{\sum_{k=1}^m k f_k}$$

$$\hat{C}_3 = 1 - \frac{f_1 - 2f_2 / (m-1) + 6f_3 / (m-1)(m-2)}{\sum_{k=1}^m k f_k}$$

$\hat{\gamma}_i^2$  is the estimate of the square of the coefficient of variation and is defined as:

$$\hat{\gamma}_i^2 = \max\left\{\frac{n}{\hat{C}_i} \sum_{k=1}^m k(k-1) f_k / \left(2 \sum_{j < k} \sum f_j f_k\right) - 1, 0\right\}, \quad i = 1, 2, 3.$$

### 2.4.2.3 Studies on Capture-Recapture

Table 2.6 summarizes the studies described in this section. It describes the type of studies and purpose of the study. It also identifies the estimators used and the best performers.

Table 2.6: Studies on Capture-recapture methods.

Study	Type of Study	Purpose	# of reviewers	Estimators applied	Best performers	Results
Eick et al. [17]	Experiment to review 13 req. & design documents.	Evaluate CRC models (m0ml)	4-12	m0ml	-	Can apply CRC to review data to estimate remaining defects.
Briand et al. [6]	Industry experiment Inspection of req. documents.	Compare models and evaluate impact of # reviewers.	2-7	m0ml mtml mhjk mthChao	mhjk	CRC best with 4 or more reviewers. Models underestimate.
Vander Wiel et al. [64]	Simulation reviews with 1 or 2 types of faults	Study effects of broken assumptions.	5	mtml mhjk	mtml	For 1 fault type or 2 fault types grouped: mtml best. 2 fault types: neither
Wohlin et al. [70]	Student experiment to inspect technical documents.	Group faults to correct underestimates.	7-8	mtml mhjk	Best model depends on type of group, organization, and documents.	Grouping faults improves estimates.

Vander Wiel and Votta [64] used Monte Carlo simulations to study the use of the maximum-likelihood and jackknife estimators in the capture-recapture model. The simulations included data that did not meet the assumptions for the estimators to study the effects of broken assumptions. Results showed that the mtml performs better than the mhjk in the case where there is only one fault type. In the study with two fault types, where fault types are pooled, neither the mtml nor the mhjk does well. When the fault types are grouped, that is, the fault types are estimated separately, the mtml is significantly improved, while there is no effect on mhjk. Results showed that the mhjk estimator greatly overestimated, when reviewers detection probabilities differ. In addition, both the mhjk and mtml estimators produced poor estimates when the detection probabilities of faults differ. The mtml estimated too low, but greatly improved, when faults were analyzed by types. The authors recommend that when dividing faults into types, the groups should be kept to a small number so that within each group it is reasonable that a reviewer has a fixed probability of finding a fault. In addition, the groups should be formed so that they have a large enough number of faults and so that some of the faults in each group are discovered by more than one reviewer. (This avoids infinite ML estimates.)



Capture-recapture models have several potential problems. First, the model implies that when there are too many reviewers, it is unlikely any faults remain. In practice, this is not true, because some faults are harder to find than others. Capture-recapture models also imply that the number of faults found during the inspection meeting are part of the estimate of remaining faults. (The assumption is that faults are found by independent reviewers, so faults found in a meeting are part of the remaining faults.) But if a large number of faults are found during the meeting, then there are probably an even larger number remaining, since some faults are difficult to find individually. In addition, the model does not work if there is no overlap: It gives an infinite estimate.

To correct underestimates of existing capture-recapture estimation methods, a technique that groups faults may be combined with the capture-recapture models [70]. Wohlin et al. [70] separate faults into two (or more) classes based on the number of reviewers who find the specific fault. This technique recognizes that certain faults are more or less difficult to find. The more reviewers that find a fault, the easier it is to find. The maximum-likelihood estimator is then applied to each class.

Faults found by more than some percentage of reviewers are put in class 1. The others are put in class 2. An alternative method is to put all faults found by exactly one reviewer in class 1 and the others in class 2. The latter method distinguishes between unique faults and faults found more than once, whereas the first method is concerned with how many times a fault is found. Applying the mtml on each fault class, the number of the remaining faults in each class are estimated and aggregated.

A problem with grouping faults occurs if all reviewers find different faults. The solution to this is to use an experience-based method: Faults found by a single reviewer are multiplied by an experience-based constant. Several studies that combine capture-recapture models and experience-based approaches are described in Section 2.4.5. Another problem occurs when the number of faults found during the review meeting is greater than the estimate of the number of remaining faults. This implies the estimate is unreliable.

Briand et al. [6] performed an experiment using data from inspections to compare capture-recapture models. Specifically, they evaluated the models and their estimators

by comparing the estimated number of defects to the actual number of defects in the documents. They also studied the impact of the number of inspectors on the accuracy of the models and estimators. Results showed that, generally, the models tended to underestimate. Results according to number of inspectors are as follows:

- When the number of inspectors is less than four, no model is sufficiently accurate in terms of its relative error and the variability in relative error.
- For two inspectors, the estimators for Mh and Mt have low relative error variability, but they tend to underestimate. If estimators consistently underestimate, they may be adjusted by some factor appropriate for the environment in which they are applied. This indicates that calibration may make the estimators for Mh and Mt good candidates in the case of two inspectors.
- For three inspectors, the jackknife estimator (mhjk) performs the best in terms of the size of relative error, although it tends to overestimate. Calibration of the mhjk estimator might make it a candidate in actual practice.
- For four or more inspectors, the mhjk and mtml perform the best, either providing overestimates or underestimates, respectively. Calibration may significantly improve them.

Results also show that the Mth model, which allows two sources of variation (i.e. variations across detection probabilities for defects and reviewers) did not seem to perform better than the Mt and Mh models, which have only one source of variation. The MthChao estimates had low relative errors, but they tended to have high variability and generated extreme outliers. The Mt and Mh models had low variability, but they tended to either overestimate or underestimate. Calibration of the Mt and Mh models may make them useful in actual practice.

### 2.4.3 Detection Profile Method and Cumulative Method

Both the Detection Profile Method and the Cumulative Method [69] use curve fitting in order to estimate the remaining defect content. These methods make less restrictive

assumptions than the capture-recapture models. There are no assumptions about independent reviewers. The contributions of individual reviewers in finding defects are expected to vary.

The Detection Profile Method assumes:

1. More reviewers find more defects. With enough reviewers, all defects will be found.
2. Some defects are found by only one reviewer.
3. The curve that models the data is an exponential or linear decreasing function. (A linear model is proposed in [8] as a way of coping with data sets where the exponential model failed.)

The Cumulative Method assumes:

1. More reviewers find more defects. With enough reviewers, all defects will be found.
2. The curve that models the data is an exponential increasing function.
3. The number of unique defects can be derived from the cumulative number of defects found – all remaining defects are assumed to be unique.

These two methods are described as follows:

1. Detection Profile Method: The defects are sorted in *decreasing* order with respect to the number of reviewers that found a defect. The defect number is on the x-axis and the number of reviewers who found that defect are on the y-axis. Figure 2.2 a) shows an example of a plot of defects using the Detection Profile Method. The plot can be approximated with a decreasing function. Both exponentially and linearly decreasing functions have been evaluated. The exponential curve [69] has the form:

$$f(k) = me^{-bk} \tag{2.14}$$

where

$k$  is the defect number,  $1 \leq k \leq n$ .

$m$  is the number of reviewers.

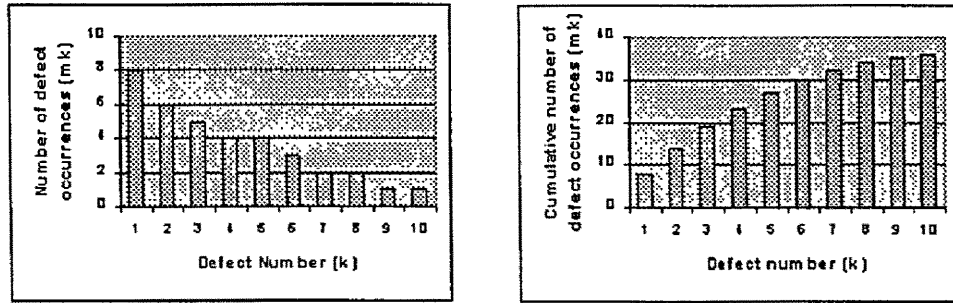


Figure 2.2: Plots for the a) Detection Profile Method and b) Cumulative Method.

$f(k)$  is the number of reviewers finding defect  $k$ .

$b$  describes how the exponential function decreases.

The fitted curve is an approximation of the bars in the Figure 2.2 a). The bars, however, represent integer values. Function values of the fitted curve in the range of 0.5 – 1.5, for example, are interpreted as bars of height 1, while values below 0.5 are interpreted as no bar. In other words, the values of the function are rounded to the nearest integer value. The total number of defects is estimated when the fitted curve reaches a threshold value of 0.5 defect occurrences. The threshold cannot be greater than or equal to 1, because the estimate will be less than or equal to the actual number of defects found. The threshold cannot be less than or equal to 0.5, because this represents a defect that was found 0 times (or never found). The estimate of the total number of defects from the curve is obtained as follows: The estimated defect content is equal to the largest integer value on the x-axis for which the curve is greater than 0.5. Subtracting the number of defects found from the estimated total number of defects gives an estimate of the number of remaining defects. The dpm(exp) estimator is based on an exponential curve fit and the dpm(linear) estimator is based on a linear curve fit.

2. Cumulative Method: The defects are sorted in *ascending* order with respect to the number of reviewers that found a defect. The defect number is on the x-axis and the cumulative number of defect occurrences are on the y-axis. For example, if five

reviewers detect the first defect and four reviewers detect the second, then the first bar is five units high and the second bar is nine units high. The defects are sorted in the same order as for the model of type 1. Figure 2.2 b) shows an example of a plot for the cumulative method. Here plotting cumulative defects leads to an *increasing* approximation function. The increasing exponential model proposed in [69] has the form:

$$M(k) = n(1 - e^{-dk}) \quad (2.15)$$

where

$k$  is the defect number,  $1 \leq k \leq n$ .

$M(k)$  is the cumulative number of reviewers having found  $k$  defects or  $\sum_{i=1}^k f_i$ .

$n$  is the total number of defects detected.

$d$  describes how the exponential function increases.

The remaining defect content is estimated as the asymptotic value of the increasing curve minus the cumulative number of defects found so far.

One of the advantages of these two curve fitting methods is that they are easy to implement using a spreadsheet application. They are also easy to visualize. These methods may, therefore, appeal to industry.

#### 2.4.4 Comparison of Capture-Recapture, Detection Profile and Cumulative Methods

Table 2.7 summarizes the studies described in this section. It describes the type of studies and purpose of the study. It also identifies the estimators used and the best performers.

Wohlin and Runeson [69] evaluated the Detection Profile Method and the Cumulative Method and compared them to the mtml. Their results showed that a rank of the three estimators based on their performance (or how close they came to actual values) did not come out in the same order all the time. The estimate from the Cumulative Method was

Table 2.7: Studies on Capture-recapture, DPM and Cumulative Methods.

Study	Type of Study	Purpose	# of reviewers	Estimators applied	Best performers	Results
Wohlin et al. [69]	Experiment using review data.	Evaluate DPM & Cumulative Methods.	4-5 7-8	mtml DPM Cum.	-	No statistical difference among them. Use mean of all 3.
Petersson et al. [46]	Case study of review documents.	Estimate intervals. Bias correction approaches.	3-4	m0ml mtml mhjk DPM, Cum.	-	1st bias correction approach: Multiply estimate by factor.
Theilin et al. [59]	Simulated experiment of inspections.	Evaluate effects of perspective-based reading on estimates.	3,6	m0ml mtml mhjk mthChao DPM	3 insps: mhjk, dpm 6 insps: all but dpm	CRCs overestimate but robust in perspective-based readings.

often the most conservative estimate. The mean error and the standard deviation of the estimates from the Detection Profile Method were often the lowest. Results of their case study, however, showed no statistical significance between these three methods. Because the three methods produced varying estimates, Wohlin and Runeson concluded that the mean value of the estimates from the three approaches was the most sensible approach to obtaining an estimate for the number of remaining defects.

Petersson et al. [46] investigated several software defect estimation methods to develop a technique that can give an estimation interval. The approach involves looking at existing estimation methods to determine whether there are methods that consistently under- or over-estimate that can be used to provide a lower and an upper bound for an estimation interval. They found that all the estimators (m0ml, mtml, mhjk, dpm, and cumulative) tended to underestimate, except for the mthChao estimator, which has been shown to have large variances for a small number of reviewers. They In an effort to correct for what appears like a systematic bias, they applied two approaches for bias correction. The first correction multiplies the estimate by a factor,  $x$ . Both corrections adds the number of faults found multiplied by a factor  $y$  to the estimate. Both models increase the variance, the first slightly more. The bias-corrected estimates are used as the upper limit. (The number of defects found can be used as a lower limit.) The first correction produces intervals that cover the correct value more often than the second (not surprising since it produces wider intervals), approximately 79 percent of the time. The advantage of using interval estimates

is that they are more likely to be trusted. The disadvantage is that they require historical data to determine the multiplicative factors for the bias correction.

Thelin and Runeson [59] present a simulated experiment to evaluate capture-recapture models when perspective-based reading techniques are used in inspections. In perspective-based readings[59], inspectors are assigned different perspectives in reading documents in order to gain better detection coverage of defects. Less overlap in faults found are expected. This contradicts the principles underlying capture-recapture, because the latter are based on overlap among faults found by different inspectors. The simulated experiment used two sizes of inspection teams. A team size of three inspectors was chosen to represent a more typical industry setting, where the inspectors, for example, would take on roles of designer, tester or user. A team size of six was also chosen to investigate how more reviewers would affect the behavior of the estimators. The number of faults simulated for a document is 30. to represent the three perspectives one third of the faults it is assumed to have a high probability of being detected. The other two thirds of the faults have a low probability of detection . Results of the simulation show that, as expected, the capture-recapture estimators overestimate in most cases when using simulated perspective-based reading data. However, the capture-recapture estimators are robust in that differences in means and standard deviations of relative errors among different perspectives were not significant. Because the capture-recapture estimators are robust, they can be used in perspective-based reading inspections. In the case of three inspectors, two estimators, the mhjk and the dpm, provide better estimates than the others in terms of mean relative error. In the case of six inspectors, all estimators investigated, except for the dpm estimator, estimated well.

#### **2.4.5 Methods that Integrate Experience**

Defect estimation methods that attempt to incorporate experience include [7, 49, 71]. Table 2.8 summarizes the studies described in this section. It describes the type of studies and purpose of the study. It also identifies the estimators used and the best performers.

The estimation of remaining faults after inspection using capture-recapture and methods based on experience is presented in [49]. Incorporating experience makes the

Table 2.8: Studies that integrate experience.

Study	Type of Study	Purpose	# of reviewers	Estimators applied	Best performers	Results
Runeson et al. [49]	Experiment using inspection data.	Classify faults and apply exp-based factors	8	mtml Exp.base		Combined method less sensitive to team composition
Wohlin et al. [71]	Experiment using code inspection data.	Compare CRC/exp-based methods.	8	CRCs DPM Cum. EBMs	mhjk and some EBMS	EBMs using review effectiveness and individual or average basis with MLE & Chao worst.
Briand et al. [7]	Experiment using requirements inspection data	Evaluate selection method for CRC/EDPM	2-6	CRCs DPM (exp) DPM (lin) EDPM	-	Selection strategy between EDPM and CRC better than models alone.

capture-recapture models more stable with respect to variations in inspection teams. This model does not make the same assumptions made by the mtml, that is, different defects have the same probability of being detected. Nor does it make the assumption that different reviewers have the same probability of finding a specific defect, as the jackknife method does. Different reviewers do not have similar profiles, some are specialists on certain kinds of faults.

Faults are divided into two classes:

1. faults found by a single reviewer.
2. faults found by more than one reviewer.

Two different methods are applied to the classes. A multiplicative factor based on experience is applied to faults in class 1. The maximum-likelihood estimator (mtml) is applied to faults in class 2. The multiplicative factor takes into account different reviewer profiles based on historical data. Initially the multiplicative factor is set to 2.11 (based on the value from a previous experiment [70]). The factor is updated when new data on each reviewer becomes available through a moving average. Runeson and Wohlin [49] shows that existing capture-recapture models are sensitive with regard to the composition of the inspection team. Combining capture-recapture with the fault classification method and applying an experience-based multiplicative factor to the first class made them less sensitive to the composition of the inspection team.



Wohlin et al. [71] describe an empirical study for estimating defect content for only two reviewers. Because estimates are unreliable if there are too few reviewers (the estimates have high relative errors and/or high relative error variability) several experience-based approaches are evaluated using historical data. The experience-based approaches are compared to the capture-recapture, detection profile and Cumulative Methods that only use data from the current review in terms of the relative errors of their estimates. Three aspects that are considered include:

- Computation of review effectiveness and efficiency. Review effectiveness measures the number of defects found compared to the total number of defects in the review object. Review efficiency measures the number of defects per unit of time. The review effectiveness and efficiency is derived in three ways:
  1.  $\sum_i n_i / \sum_i N_i$ , where  $n_i$  is the observed number of defects in review  $i$  and  $N_i$  is the actual number of defects in review  $i$ . This is an effectiveness measure denoted as P1.
  2.  $(\sum_i^a n_i / N_i) / a$ , where  $a$  is the number of reviewers. This is a mean effectiveness measure denoted as P2.
  3. Efficiency, denoted as P3, is derived based on the effectiveness per time unit, it includes review time as a parameter. Effectiveness of a specific review is divided by the time spent on the review.
- Use of experience in terms of review effectiveness and efficiency. These measures are applied in three ways:
  1. Individual reviewers base, denoted as I. (Estimation is based on the experience of individual reviewers.)
  2. Group experience base, denoted as G, (Reviewers are grouped according to their backgrounds).
  3. Average reviewer base, denoted as A. (Estimations is based on an average for all reviewers).

- Estimation approach. The two estimation approaches considered include:
  1. Mean of individual estimates, MV.
  2. Maximum-likelihood method, MLE.

Combining the alternatives for the three aspects results in 18 ( $3 \times 3 \times 2$ ) different experience-based methods (EBMs). To reduce this number, Wohlin et al. [71] compared the alternatives for each of the three aspects to each other using statistical methods. For derivation of review effectiveness and efficiency, [71] evaluated the three alternatives, P1, P2, and P3, in all possible pairwise comparisons by applying Fisher's PLSD (Protected Least Significant Difference), a parametric test similar to a t-test [71]. The alternative that includes review time is significantly worse than the other two. Thus, it is better to use review effectiveness than review efficiency.

The three alternatives for use of review effectiveness measure were also evaluated in all possible pairwise comparisons. An ANOVA test showed no statistical differences.

[71] applied a t-test (for the absolute error) to compare the two estimation approaches. The test showed there is a significant difference between the two approaches. Looking at the estimates from the two approaches, the maximum-likelihood estimator performed better than the averaging approach.

This left six ( $3 \times 2 \times 1$ ) EBMs to evaluate. The six EBMs include:

- A. EBM (P1-I-MLE)
- B. EBM (P1-G-MLE)
- C. EBM (P1-A-MLE)
- D. EBM (P2-I-MLE)
- E. EBM (P2-G-MLE)
- F. EBM (P2-A-MLE)

These six EBMs were compared against the direct estimation methods `m0ml`, `mtml`, `mhjk`, `dpm(exp)`, `nthChao`. The methods were evaluated using the mean absolute relative error, standard deviation, and maximum error.

Evaluations show that the `nthChao` estimator was the only method that showed significant statistical results. It performed much worse than the others. The authors performed a subjective evaluation to identify the models that they considered best and that warranted further study. Results of the subjective evaluation identified four experience-based methods (A, C, D, F) and the `mhjk` estimator as candidates for further study. These four EBMs use review effectiveness rather than efficiency on an individual or average basis with the maximum-likelihood estimation approach.

One problem with capture-recapture models is that they often provide extreme under- and over-estimations. That, in turn, provides little confidence in the methods. Briand, et al. [7] address this problem by enhancing the Detection Profile Method and integrating it with capture-recapture models. Their technique uses selection criteria to decide which model to apply based on the characteristics of the data.

The Detection Profile Method (`dpm`) requires the defects to be exponentially decreasing in terms of the number of reviewers that found them, and that some defects are found by only one reviewer. These assumptions may not always be met. If the data is not exponential, it may result in high estimates, especially if at least one inspector found no defects. Their method improves on the `dpm` by

1. Taking into account the different shapes a fitted curve might have; and
2. Providing selection criteria to choose between these curves.

Two curve shapes are considered: exponential and linear. Use of an exponential fit versus a linear fit is determined based on the  $R^2$ -value (the coefficient of determination that measures the goodness of fit of the curve to the data). The fit that gives the largest  $R^2$ -value is selected. If the number of defects found by exactly one inspector is zero, the exponential curve may still provide a good fit, or it may not. In this case, an exponential fit is selected

only if the following ordering holds:

$$f_1 \geq f_2 \geq f_3 \geq \dots \geq f_k \quad (2.16)$$

where

$k$  is the number of inspectors.

$f_i$  is the number of defects found by  $i$  inspectors.

If the underlying data is indeed exponential, this ordering will hold and an exponential curve will fit better than a linear curve.

The authors evaluated four dpm strategies:

1. Always fit exponentially (used in [69]).
2. Always fit linearly.
3. Select either the exponential fit or the linear fit based on  $R^2$ .
4. Select either the exponential fit or the linear fit based on the strict ordering criterion.

Table 2.9 shows the strategies investigated by Briand et al. [7]. It also shows the results of the comparisons among the strategies.

Table 2.9: Briand's DPM strategies summarized.

Strategy	Description	Results
1	DPM (exponential)	
2	DPM (linear)	
3	Select DPM (exp or linear) based on $R^2$	
4	Select DPM (exp or linear) based on strict ordering (EDPM)	Best of 1-4. 4 comparable to 5
5	CRC models (mhjk, mhChao, mtml)	
6	Selection between EDPM and CRC (mhjk)	6 better than 4 or 5 alone in some cases and never worse.

Results show that the fourth strategy, the strict criterion ordering, is the best of the four strategies. Briand, et al. [7] compared the strict ordering strategy, called the Enhanced Detection Profile Method (EDPM) with the capture-recapture models. They applied the Mh model, since that model was preferred in their previous study [6], using the jackknife

estimator and the Chao estimator. They also used the estimator  $Mt(mtml)$ , since it was used for comparison purposes in [69]. Results show that the EDPM is comparable, but not a major improvement over existing capture-recapture models.

The authors also present a selection strategy between EDPM and capture-recapture models. If  $R^2 \geq 0.8$  and is statistically significant at the level of 0.01, then EDPM should be selected, otherwise a capture-recapture model should be selected, specifically  $Mh(mhjk)$ . ( $R^2$  can be high and the confidence level low, if there are few inspectors and few defects are found.) Results show that when using the selection strategy between the EDPM and the capture-recapture models, in some cases estimates were better than when using either EDPM alone or capture-recapture alone. In no case was the selection strategy between EDPM and capture-recapture worse.

The static defect estimation methods investigated in this thesis include Capture-recapture methods, the Detection Profile Method, the Cumulative Method, as well as a simple experienced-based method proposed in Chapter 8.1.1. The case study in this thesis applies the defect estimation methods, not to estimate the number of defects remaining after inspection, but rather to estimate the number of components that have defects after release that were defect-free in system test. In addition, the case study uses defect data from different test sites, rather than different reviewers. Chapter 8.1.1 describes the reasons for this novel approach. This thesis also proposes and evaluates a method to use defect estimations as one criterion in making decisions regarding continuing and stopping system test.

## 2.5 Defect Estimation using Dynamic Models

### 2.5.1 Software Reliability Models

Static and dynamic software reliability models exist to assess the quality aspect of software. These models aid in software release decisions [14]. While a static model uses software metrics, like complexity metrics, results of inspections, etc. to estimate the number of defects in the software, a dynamic model uses the past failure discovery rate during

software execution or cumulative defect profile over time to estimate the number of failures. It includes a time component, typically time between failures.

Musa et al. [37] make a distinction between failures and faults. A *failure* is a departure from how software should behave during operation according to the requirements. Failures are *dynamic*: The software must be executing for a failure to occur. A fault is a defect in a program, that when executed causes a failure(s). While a fault is a property of the program, a failure is the property of the program's execution.

Dynamic models measure and model the failure process itself. Because of this, they include a time component, typically, they are based on recording times  $t_i$  of successive failure  $i$  ( $i \geq 1$ ). Time may be recorded as execution time or calendar time. These models focus on the failure history of software. Failure history is influenced by a number of factors, including the environment within which the software is executed and how it is executed. A general assumption of these models is that software must be executed according to its operational profile, that is test inputs are selected according to their probabilities of occurring during actual operation of the software in a given environment [36]. Many dynamic models have been developed [23, 29, 38, 37, 74, 76], based on various sets of assumptions about the software and its execution environment.

Many dynamic models try to assess whether a given testing approach is likely to discover more failures [15, 16, 77], increase coverage [27, 28, 12], attain a given reliability criterion [35, 37, 77], or meet some other software testing objective to determine when to stop testing. [37, 38, 63] use SRGMs as stopping rules.

Many dynamic models have been developed [23, 29, 38, 37, 74, 76], based on various sets of assumptions about the software being executed and the execution environment. Trachtenberg [63] presents a framework for the dynamic models. The Trachtenberg-General model assumes that software failures occur when software faults are encountered during software execution. The rate at which failures are experienced is defined as

$$\frac{df}{dt} = \lambda = \frac{df}{de} \frac{de}{dx} \frac{dx}{dt} = s \cdot d \cdot w$$

where

$x$  is the number of executed instructions.

$e$  is the number of encountered errors.

$f$  is the number of failures.

$S, s$  are the initial and current average size of remaining errors (measured as failures per encountered error). The average size of remaining errors is proportional to the probability of failure of the remaining software error.

$d$  is the apparent error density (measured as number of encountered errors per executed instruction).

$D = dR/r$  is the actual error density.

$R$  and  $r$  are the initial and current remaining errors, respectively.

$W, w$  are the initial and current workload (measured as the number of instructions per unit time).

The classical models of software reliability can be derived from Trachtenberg's General Model by modifying assumptions for the parameters,  $s$ ,  $d$ , and  $w$ . Musa's model [37], for example, assumes that the average size of remaining errors,  $s$ , is constant, that the apparent error density,  $d$ , is the same as the actual error density,  $D$ , and that the workload,  $w$ , is constant. In this model, the failure intensity decreases a constant amount each time a defect is removed. (The general assumption is that all defects are corrected when discovered.) The failure rate decreases at the same rate at which remaining errors decrease. The failure intensity,  $\lambda$ , is given in [37] as

$$\lambda(\mu) = \lambda_0(1 - \mu/v_0)$$

where

$\lambda_0$  is the initial failure intensity (or hazard rate).

$v_0$  is the total number of failures that would occur in infinite time.

$\mu$  is the expected number of failures at a time  $t$ .

The failure intensity function can also be expressed in Trachtenberg's model as

$$\lambda = -\frac{dr}{dt} = SDW e^{SDWt/R}$$

Musa's basic model [37] uses failure data from execution to characterize the failure process. The expected number of failures is expressed as a function of time  $t$ .

$$\mu(t) = v_0(1 - e^{-\frac{\lambda_0}{v_0}t}) \quad (2.17)$$

In the absence of failure data,  $\lambda_0$  and  $v_0$  must be predicted. If failure data exists, both parameters are estimated using maximum likelihood estimation or some other suitable method.

Given a target failure intensity, one can derive the number of additional failures or the additional amount of execution time needed to reach the desired failure intensity. This provides valuable information to system testers in making decisions to release software.

Musa's basic model [37] makes the following assumptions:

- Test input is selected randomly from a complete set of inputs anticipated in actual operation. (The operational profile specifies the probability that a given input will be selected.)
- All software failures are observed.
- Failure intervals are independent of each other.
- The execution time between failures is exponentially distributed and the failure rate is constant during the interval between failures. (The failure rate changes at the correction of each defect.)
- The failure rate is proportional to the number of failures remaining.
- The failure detection rate is proportional to the failure rate.

There is a whole body of knowledge of software reliability growth models [36, 37, 23, 29, 63, 74, 76] with many studies and applications of the models in various contexts [38, 72, 73]. Like the Musa model, the Goel-Okumoto (G-O) model, the Goel-Okumoto S-shaped model [74], the Gompertz model [29], and the Yamada Exponential model [76] all assume testing follows an operational profile. They also assume that the software does not change, except that defects are fixed when discovered. The models differ in their assumptions either in terms of workload, error size, or failure intensity.



Some models use a non-homogeneous Poisson process (NHPP) to model the failure process. The NHPP is characterized by its expected value function,  $\mu(t)$ . This is the cumulative number of failures expected to occur after the software has executed for time  $t$ .  $\mu(t)$  is nondecreasing in time  $t$  with a bounded condition,  $\mu(\infty) = a$ , where  $a$  is the expected number of failures to be encountered if testing time is infinite. Different nondecreasing functions  $\mu(t)$  give NHPPP models. Musa's basic model [37], the G-O model, the Goel-Okumoto S-shaped model [74], the Gompertz model [29], and the Yamada Exponential model [76] are all based on an NHPP.

The expected value function for failure intensity can be put into two shape classes: concave and S-shaped [72]. S-shaped models are first convex, then concave. The S-shaped growth curves start at some fixed point and increase their growth rate monotonically to reach an inflection point. After this point, the growth rate approaches a final value asymptotically. The S-shaped models reflect an assumption that early testing is not as efficient as later testing, so there is a period during which the failure-detection rate increases. This period terminates, resulting in an inflection point in the S-shaped curve, when the failure-detection rate starts to decrease.

Software reliability growth models predict the number of failures,  $\mu$ , at time  $t$ , or  $\mu(t)$ . The G-O model [23] (similar to the Musa model) is a concave model. It uses the function

$$\mu(t) = a(1 - e^{-bt}), \quad a \geq 0, \quad b > 0 \quad (2.18)$$

where

$a$  is the expected total number of failures that would occur if testing was infinite.

It is the upper limit that the reliability (or number of failures) approaches asymptotically as  $t$  approaches infinity. ( $a = v_0$  in Eq. 2.17.)

$b$  is the rate at which the failures detection rate decreases. It is a shape factor for the curve. ( $b$  is  $\lambda_0/v_0$  in Eq. 2.17.)

The assumptions for this model are the same as for the Musa model.

The delayed S-shaped model [74] is a modification of the G-O model to make it S-shaped. An S-shaped reliability growth curve describes a reliability growth trend with a lower rate

of failures occurring during the early stages of development and a higher rate later. It is given by:

$$\mu(t) = a(1 - (1 + bt)e^{-bt}), \quad a \geq 0, \quad b > 0 \quad (2.19)$$

where

$a$  is the expected total number of failures that would occur if testing was infinite.

( $a = v_0$  in Eq. 2.17.)

$b$  is the failure detection rate during the steady-state, that is the value to which the rate converges as  $t$  approaches infinity. (The failure intensity rate initially increases from  $t = 0$  to  $t = 1/b$  and then gradually decreases, approaching zero.)

The delayed S-shaped model makes the following four assumptions:

- The failure detection process is a non-homogeneous process, that is the characteristics of the probability distribution vary over time.
- The time to failure of an individual fault follows a gamma distribution with a shape parameter of 2.
- Each time a failure occurs, the error that caused it is immediately fixed, and no other errors are introduced.
- The initial error content of the system is a random variable.

The G-O models have two parameters; other models may have more parameters. The Yamada exponential model and the Gompertz model are two such examples.

The Yamada Exponential, a concave model [76], attempts to account for differences in testing effort. It does not assume that testing effort is constant over the testing period. It is given by the equation:

$$\mu(t) = a(1 - e^{-bc(1 - e^{-dt})}), \quad a \geq 0, \quad bc > 0, \quad d > 0 \quad (2.20)$$

where

$a$  is the expected total number of failures that would occur if testing was infinite.

( $a = v_0$  in Eq. 2.17.)

$b$  is the failure detection rate per unit testing-effort. ( $b$  is  $\lambda_0/v_0$  in Eq. 2.17.)

$c$  and  $d$  are parameters in the testing-effort function. To account for a variable amount of effort,  $c$  and  $d$  are based on assuming an exponential form for the testing effort function [3]. The parameters are estimated using least-squares.

The Yamada exponential model makes the following five assumptions:

- The failure process is a non-homogeneous process, that is the characteristics of the probability distribution vary over time.
- Each time a failure occurs, the error that caused it is immediately fixed, and no other errors are introduced.
- Testing-effort is described by an exponential curve.
- The expected number of failures in a time interval to the current testing-effort expenditures is proportional to the expected number of remaining errors.

Another popular model to estimate remaining failures is the Gompertz model [29]. It has been widely used to estimate software error content [74]. It works by fitting a curve to the data using regression analysis. The Gompertz model [29] is an S-shaped model. It is given by the following equation:

$$\mu(t) = a(b^{c^t}), \quad a \geq 0, \quad 0 \leq b \leq 1, \quad 0 < c < 1 \quad (2.21)$$

where

$a$  is the expected total number of failures that would occur if testing was infinite.

( $a = v_0$  in Eq. 2.17.)

$b$  is the rate at which the failures detection rate decreases. ( $b$  is  $\lambda_0/v_0$  in Eq. 2.17.)

$c$  models the growth pattern (small values model rapid early reliability growth, and large values model slow reliability growth).

Table 2.10 summarizes the models used in this study.

Two common ways for estimating the function's parameters from data are the maximum likelihood and regression methods. The maximum likelihood method estimates the

Table 2.10: Software reliability growth models used in this study.

Model	Type	Equation ( $\mu(t)$ )	Reference
G-O or Musa	Concave	$a(1 - e^{-bt}), a \geq 0, b > 0$	[23]
Delayed S-shaped	S-shaped	$a(1 - (1 + bt)e^{-bt}), a \geq 0, b > 0$	[74]
Gompertz	S-shaped	$a(b^{c^t}), a \geq 0, 0 \leq b \leq 1, c > 0$	[29]
Yamada Exponential	Concave	$a(1 - e^{-bc(1 - e^{-dt})}), a \geq 0, bc > 0, d > 0$	[76]

parameters by solving a set of simultaneous equations, usually numerically. Methods for estimating parameters for the G-O model and the delayed S-shaped mode are provided in [23] and [74], respectively. For the Yamada exponential model, the methods to estimate the initial values are provided in [76]. [29] provides methods to estimate initial values for the Gompertz model. Parameter estimates may also be obtained using nonlinear regressions. This approach fits the curve to the data and estimates the parameters from the best fit to the data, where fit is defined as the difference between the data and the curve function fitting the data.

### 2.5.2 Software Reliability Models in Practice

Goel discussed the applicability and limitations of software reliability growth models during the software development life cycle in [24]. He proposed a step-by-step procedure for fitting a model and applied the procedure to a real-time command and control software system. His procedure selects an appropriate model based on an analysis of the testing process and a model's assumptions. A model whose assumptions are met by the testing process is applied to obtain a fitted model. A goodness-of-fit test is performed to check the model fit before obtaining estimates of performance measures to make decisions about additional testing effort. If the model does not fit, additional data is collected or a better model is chosen. He does not describe how to look for a better model. The problem with this method is that, in practice, many of the models' assumptions are violated, hence none

of the models are appropriate. The method does not allow for the fact that many of the models are robust when assumptions are not met.

SRGMs have many assumptions that must be met regarding testing and defect repair that are not valid in actual software development and test environments [73]. The realities are:

- It is difficult to define operational profiles and perform operational tests.
- Defects may not be repaired immediately.
- Defect repair may introduce new defects.
- New code is frequently introduced during the test period.
- Failures are often reported by many groups with different failure-finding efficiency.
- Some tests are more or less likely to cause failures than others.

For the software practitioner, the assumptions or conditions for the SRGMs are an open problem, because they are often violated in one way or another. Several studies have found that even so, SRGMs perform well in practice. Two industrial applications of reliability measurement are described in [38]. Despite distributional assumption violations, in both cases, reliability measurements performed well in predicting failure rates, demonstrating that reliability measurement may be used in industry.

Wood compared the assumptions of SRGMs to Tandem's defect removal environment and points out two key differences [73]. These include the introduction of new code during system test and varying defect-finding efficiency of tests. Wood also points out the effects of violating the model's assumptions. They are:

- The number of defects increase during testing rather than remaining constant.
- The defect-finding efficiency of tests per unit of time varies rather than remaining constant.

Wood [73] proposed three approaches to accommodate model assumption violations with advantages and disadvantages discussed for each. The easiest approach is to ignore the

violations. This keeps the model simple, but causes some loss of accuracy. Parameter estimation, however, may compensate. The second solution proposed involves modifying the data. This approach is easy to implement, because the standard models can be used with the modified data. Data modification, however, needs to be based on a mathematical model. A third approach is to derive new models to fit the test environment. These models are more complex. While they are more accurate, they are more difficult to derive and to apply. In Wood's experimentation with data from Tandem, the simple models performed reasonably well despite assumption violations, although the confidence limits were wide. In some cases, the data had to be modified.

In [72], Wood applied eight reliability models to a subset of software products with four releases to determine which model performed the best in predicting the number of residual defects. This study shows that software reliability growth models based on cumulative defects predict the number of remaining defects that are close to the number of defects reported in post-release. The cumulative number of defects by week are fitted with software reliability growth models. If the correlation is good, the function can predict the number of remaining defects in the system.

One may use a software reliability growth model in industrial applications in two ways:

1. During system test to predict the additional test effort needed to achieve a desirable quality level in terms of number of remaining failures.
2. At the end of system test to predict the number of remaining failures that could be reported in post-release.

A useful model must become and remain stable. Predictions week by week should not vary much. According to Wood [72], the prediction should not vary by more than 10 percent from week to week. Stability requires a reasonable amount of data that may not be available until several weeks have passed. A useful model must also be reasonably accurate at predicting the number of failures in post release.

In [72], predictions were attempted using execution time, calendar time and number of test cases. In the environment in Wood's study [72], execution time was a better measure

of amount of testing. Using calendar time, curve fits diverged in earlier test weeks, making it impossible to obtain predictions. In later weeks, predictions were unstable, especially in comparison to execution time.

## 2.6 Research Matrix

An integrated approach to assessing and guiding testing activities needs to combine the following:

1. Assessment (Single and Multiple Releases)
  - Analysis of fault-prone modules or components.
  - Assessment of code decay and architectural decay.
2. Test Guidance and Release Decisions
  - Prioritizing testing activities.
  - Defect estimation.

The following tables illustrate existing work in these areas. Existing assessment methods are classified by type of data used in the analysis, and whether the analysis applies to individual modules, module interactions, single releases, across releases, or across development phases.

Tables 2.11 – 2.13 summarize existing work related to testing effectiveness and efficiency. The columns in each table indicate subareas of research, for example, whether fault-proneness is based on measures within a module or between modules. Research is also classified by analysis across releases, across development phases, or neither. The tables also distinguish existing work by the type of data used: product measures, such as code complexity, number of lines of code, number of operands, etc.; or defect data, or change data. Asterisks (\*) in Tables 2.11 – 2.13 show the areas in which the work in this study will contribute.

Table 2.11 classifies research in the area of fault-prone analysis. The columns indicate the fault-prone analysis methods that use product metrics, defect data or change data. (Change

data may refer to changes made to modules to fix defects or reuse and modification data.) The rows identify whether fault-prone components are identified based on module characteristics or characteristics between modules (inter-module). Analysis for fault-proneness

Table 2.11: Research Matrix for Fault-Prone Classification Methods.

	Data Type		
	product metrics	change data	defect data
<b>MODULE</b>			
Multi releases	[4], [41], [31], [32]	[4], [40], [31], [32]	[41], *
Across phases	[8], [30], [33], [42], [43]	[30], [33]	*
Single release	[25], [57]	[25]	
<b>INTER-MODULE</b>			
Across phases	[8], [42], [43]	See Table 2.12	

is sparse when it is based on defect data or change data alone. This is especially true in across-phase defect data analysis. Components identified as fault-prone in earlier phases of development based on defect data should probably be targeted for more intense testing. It also might help to track them in later phases and across releases to see whether they continue to have problems. Thus, fault-prone components during development might be used to guide testing. Yet, so far, methods have not been developed to do this.

Table 2.12 classifies research related to reverse architecting and code decay. The row and column categories for Table 2.12 are similar to the ones in Table 2.11. Two columns separate the research in reverse architecting and code decay. This is because the work on code decay analyzes the architectures built to identify parts of the software that may be problematic and in need of more attention.

All work on code decay used industry data, except for [62]. Most work concentrated on identifying fault-prone modules. Very little work has been performed in identifying fault-prone relationships, that is, inter-module problems. Table 2.12 also shows that more work is needed to analyze fault-prone relationships across development phases using defect data. Only one group of researchers [30, 33] has done any work in this area.

Table 2.13 classifies research in the area of defect estimation that may be used to make release decisions. The columns identify several categories of defect estimation methods.



Table 2.12: Research Matrix for Reverse Architecting and Code Decay Methods

	Reverse Architecting	Code Decay		
	product metrics	product metrics	change data	defect data
<b>MODULE</b>				
Multi releases	[62]	[2], [40]	[18], [22], [40]	[40], *
Across phases		[30], [33]	[30], [33]	*
Single release		[25]	[25]	
<b>INTER-MODULE</b>				
Multi releases	[34], [62], [65], [66]	[40]	[40], [65], [66]	[40], [65], [66]
Across phases				
Single release	[20]			

They include capture-recapture models, curve-fitting methods, experienced-based methods and reliability models. Some integrate several methods. Some do not belong to any of the categories and are classified as “other.” Row categories are similar to the ones in Tables 2.11 and 2.12.

Table 2.13: Research Matrix for Defect Estimation Methods.

	CRC	Curve-fit	Exp.Based	SRGMs	Other	
	defect	defect	defect	failures	product	change
<b>MODULE</b>						
Multi releases			[5]			
Across phases			[5], [78]			
Single Release					[13], [25]	[13], [25]
<b>SYSTEM</b>						
Single release	[6], [7], [17], [19], [46], [49], [59], [71], *	[7], [46], [59], [69], *	[46], [49], *	[37], [38], [39], [72], [73], [77], *		

Table 2.13 shows there has been a great deal of work in estimating the number of defects remaining in a system. (All work in Table 2.13 used industry data, except for [59], which used simulations.) Not as much research has been done in estimating the number of defects remaining in a module. Defect estimation methods for modules are more difficult to develop, because there is usually not as much data (per module). An area that has a need for more work is in the use of product metric data for defect estimation. So far, product metrics have mostly been used to identify fault-prone components, not to estimate remaining defects in them or the system.

Another area lacking is in the application of defect estimation methods across releases or phases. Capture-recapture and curve-fitting methods have been used in inspections. They also have potential for other phases of development, such as testing. In addition, these techniques have been used to estimate the remaining number of defects, but may potentially be used to estimate the number of components with defects, if the system is large enough and has many components.

SRGMS have underlying assumptions that may be violated in practice. Despite this, many are quite robust and may be used for defect estimation to make release decisions. The problem is that it may not be obvious which model(s) to use as failure data is collected. More research is needed in this area.

In addition, there is very little research in prioritizing testing activities. Rothermel et al. [33] investigates ways to prioritize test cases using white box techniques. While [33] suggests that code churn can predict fault-prone modules and may recommend modules for earlier testing, no research has been performed using black box techniques to prioritize testing activities to improve efficiency. Our case study investigates this area.

Tables 2.11 – 2.13 clearly illustrate where research is lacking. In the area of fault-prone analysis, little research has been done to identify components that are fault-prone using defect data from earlier phases of development. In the area of fault architecture analysis, there is no research on using defect data from different phases of development to build fault architectures. To address some of the deficiencies in prior research, this thesis will answer the following research questions.

- Can analysis of defect and change data from earlier phases of development be used to guide system testing?
- Can reverse-architecting techniques be used to build fault-architectures?

There is already a great deal of research in the area of defect estimation. Most defect estimation methods attempt to estimate the number of remaining defects in a system. A few methods are concerned with estimating the number of remaining defects in modules. No method estimates the number of remaining components with defects, which may also

be used to make release decisions. Two major areas of defect estimation, capture-recapture and curve-fitting, estimate remaining defects after reviews. They do not estimate remaining defect content after testing. If there are several testers or testing groups, there is potential that these techniques can be applied in the system test or acceptance test phases. This thesis will answer the research questions:

- Can the number of remaining components with defects be estimated?
- Can defect estimation be used to make release decisions?

A lot of research has occurred on the subject of SRGMs. Many of the approaches are based on models with assumptions about distributions, independence of tests, etc. Most existing SRGMs either use data from previous releases, subjective data based on developers' experience, or a great deal of data within the testing phase of the current release. This thesis proposes a selection method to determine model(s) that are appropriate to use to estimate defect content for the purpose of making release decisions.

This chapter described the research in regards to testing effectiveness and efficiency and pointed out the subareas that could benefit from more research. To address this, this thesis presents an integrated approach and applies it in a case study involving a large medical record system. Of particular interest is the use of defect and change data from development to guide system testing and using defect data from system test to make release decisions. The following chapters describe this approach and apply it in a case study.

# Chapter 3

## Approach

The approach used to develop an integrated method to improve testing effectiveness and efficiency involves several techniques. Release Quality Assessment uses fault-prone component and fault architecture analysis techniques. Assessment results aid in developing testing guidelines and strategies to focus testing on the problematic parts of the software and prioritize testing activities. Corresponding to the Release Decision part of the integrated method, static and dynamic defect estimation methods are used to make release decisions. A testing strategy that recommends prioritizing testing activities has the potential to enable earlier release decisions. Multi-Release Quality Assessment performs fault-prone component analysis and fault-architecture analysis across several releases. The analysis enables additional testing strategies and guidelines to be developed. Figure 3.1 shows the types of analysis for each part of the integrated method as described in Section 1.1.

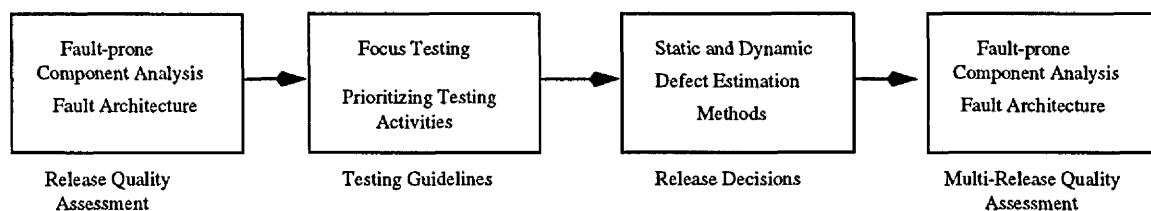


Figure 3.1: Techniques used to improve testing effectiveness and efficiency.

The type of data available drives the questions that can be asked, as well as the kinds of methods that can be used. The approach uses defect data (failure reports) from development, testing, and post-release over several releases of a large system based on the idea that

past behaviour is often the best predictor of future behavior. Questions that are important to testers are:

- Do components that have errors after release have problems in other phases of the software development life cycle?
- Does product quality during development have any impact on the number of defects found in testing or after release?
- Is it possible to identify parts of the system that are fault-prone during development, and because of that cause extra testing work?
- Is it possible to use defect data from development to identify better testing guidelines during system test?
- Is it possible to identify fault-prone relationships between components early? How can this information be used for more effective tests?
- Should parts of a system that were identified as problematic in the previous release, be tested more intensely and earlier in the next release?
- Is it possible to estimate the number of components that will still have defects after release by looking at the number of components that had defects during system test?
- Can data from system test guide release decisions?

An integrated method was developed to answer these questions and to improve both testing effectiveness and efficiency. Techniques used in each part of the integrated method are described below.

#### 1. Perform Release Quality Assessment.

Many attributes of a software project influence the software testing effort, both its effectiveness and efficiency. Example attributes are complexity of the problem, schedule urgency, and the quality of work during design and implementation. Of particular interest is whether defects found during development are related to the occurrence of defects during testing. The rationale for such a relationship is that:

- Components with severe or systemic problems during development carry a higher risk of not being completely fixed at the start of system test.
- They are more likely to exhibit long term problems.
- These problems are more likely to be severe.
- If this is the case, defect data from development can guide testing.

The approach requires the analysis of system defect reports from development and system test. The approach uses the following techniques to perform the assessment.

- (a) Use GYR analysis [40] to identify fault-prone components within development and system test.
- (b) Create a fault-prone component directory structure [40, 65, 66] to identify fault-prone components within development and system test.
- (c) Create component level fault architectures for development and system test to identify components in fault-prone relationships.

An important issue for testers is to what degree they prevent post-release defects and which types of components make it into release with undetected defects that show up after release. To address this issue, the approach also applies fault-prone component analysis and fault architecture analysis to post-release defect data.

## 2. Develop Testing Guidelines and Strategies.

The approach recommends basing testing guidelines and strategies on quality assessment results.

- (a) Identify components that should be tested more thoroughly.

Components that are problematic in earlier development life-cycle phases are likely to have problems during system test and post-release. Efforts to test these components more thoroughly aids in finding the defects in test, rather than in post-release.

- (b) Test problematic components earlier.

Specifically, components that are fault-prone during development or earlier releases should be system tested at the earliest possible time. This would shift higher defect intensities earlier in the test cycle, giving developers more time to fix remaining problems.

### 3. Make Release Decisions.

Two factors that influence testing efficiency are *cost* and *yield*. Cost can be expressed in terms of cost of test generation (e.g. random test generation is cheaper than test generation by symbolic execution), cost of test execution (thus one wants to execute as few tests as possible and automate execution), and cost of test validation (self validating tests are cheaper than tests that require a fair amount of manual tester effort). Yield can be expressed either in coverage elements found (for example number or proportion of branches covered) or faults exposed (alternatively one could record failures). This thesis is concerned with *yield* in terms of defect reports (or failures). Given that they also report when they occurred, this case study uses defect reports as failures or faults exposed, depending on the analysis technique applied.

Software developers are concerned with finding defects as early as possible in the development of software. Certainly, they would prefer to find defects in system test rather than after release of the software. Methods that estimate defect content after release aid software testers in making decisions to stop testing and release software. Steps in this approach are:

- (a) Use static defect estimation techniques to estimate the remaining number of defective components in the software.

Traditionally, capture-recapture models and curve-fitting methods are used to estimate the remaining number of defects based on inspection reports from several reviewers. The approach applies these methods in a novel way to estimate the number of components that have defects after release but are defect-free in system test. The approach also uses a simple experience-based method that

does not require product or process metrics, although it does require data from previous releases.

Estimation is based on data provided by test groups from different test sites. Each test site takes the role of the “reviewer” in the models. Test groups should test the software in parallel, that is they should “review” the same system. Similarly, one could define subgroups within a system test group that test the same software.

Quality of estimation is evaluated by

- Estimation error.
- Decision error.

The decision to stop or continue testing is based on the estimated number of components that are not defect-free after release for which no defects were reported in system test.

- (b) Apply the SRGM selection method to estimate the number of failures that will occur after release.

Methods that estimate remaining failures in software can help test managers make release decisions during testing. Various software reliability growth models [23, 24, 29, 38, 37, 74, 75, 76] have been used to estimate remaining defect content. The problem with using SRGMs to estimate defect content is that they have underlying assumptions that are often violated in practice. Despite the fact that empirical evidence has shown that many of the models are quite robust in practice, it is often difficult to decide which model to apply in light of these assumption violations. The “best model” can vary across systems, and even releases. One cannot select one model and apply it in later releases or other systems.

Thus it is important to have an (iterative) selection method for these models that is based on the failure data collected. The approach in this thesis applies a series of models as soon as testing has progressed to a point in the test plan that it makes sense to worry about whether to stop testing. The selection method



determines the best model(s) for estimating the total number of failures in the software. From this estimate, the expected number of remaining failures is calculated.

(c) Use the estimates to make release decisions.

The approach compares the estimates from the static and dynamic estimation methods to acceptability thresholds. If the estimates are above the thresholds, the approach recommends stopping test and releasing the software.

#### 4. Perform Multiple-Release Quality Assessment.

Identification of fault-prone components and fault-prone component relationships is desirable so that steps can be taken to more thoroughly expose the nature of problems. This may include more intensive testing of fault-prone components and relationships, or performing code decay analysis. Code decay analysis involves identifying code that increasingly becomes problematic over time and more difficult to maintain, with the objective of taking steps to prevent further degradation.

Of particular interest is whether problems identified in earlier releases are indicators of problems in future releases. The rationale for such a relationship is that parts of the software with severe or systemic problems in earlier releases are more likely to exhibit long term problems, and that these problems are more likely to be severe. If this is the case, one can use defect reports from earlier releases to guide testing.

Defect analysis identifies both components and relationships between components that are problematic. A defect cohesion measure at the component level is an indicator of problems local to the component, while a defect coupling measure between two components is an indicator of relationship problems between components [66]. High values in either are undesirable, indicating problems. The problems they indicate are of different types. High defect cohesion measures identify components that have problems locally, that is, they have internal problems. High defect coupling measures identify relationships between components that are broken.

The approach uses the defect coupling measures in a reverse architecting technique presented in [40, 65, 66] to derive "fault architectures." One can identify and highlight problematic components and relationships that occur over multiple releases, and ignore components and component relationships that are not problematic.

The steps in the approach are:

- Uses two defect cohesion measures to identify fault-prone components.
- Uses two defect coupling measures to identify components that have many fault relationships.
- If the objective is to focus on the most problematic parts of the software architecture, use filters. Our approach investigates a method of setting the threshold based on order of magnitude, that is to 10% of the largest measure.
- Perform cross-release and cumulative release analyses using Fault-Prone Component Analysis and the Fault Architecture technique.

Identifying problematic parts of the system will enable system testers to focus testing on these parts of the system. Persistent problems involving multiple components over several releases may indicate the need for more intense testing or rearchitecting.

Chapter 4 describes the data used in the case study that applies the integrated method. Chapters 5 – 9 describe the techniques in the integrated approach in more detail and applies them to the data.

Chapter 5 describes the approach taken to determine components that are fault-prone. It applies the approach to develop testing guidelines for system test to improve testing effectiveness. Testing guidelines developed using the first release are applied to the second release to evaluate the effectiveness of those guidelines. The effect of the testing guidelines on efficiency improvement are evaluated.

Chapter 6 describes the fault architecture technique and applies it to fault data to identify problematic parts of the software that should be tested more. This assessment may be used to develop additional testing guidelines and to make longitudinal decisions.

Chapter 7 describes the approach of prioritizing testing of fault-prone components. It looks at the effect this has on the cumulative defect curve that may in turn affect release decisions.

Chapter 8 describes static defect estimation methods to estimate defect content and applies them in a new way. It estimates the number of components that have defects in release that do not have defects in test. The estimates are then used to make release decisions.

Chapter 9 introduces a selection method for software reliability growth models to estimate the total number of failures. The purpose of this method is to make release decisions.

# Chapter 4

## Case Study

### 4.1 Data

The defect data come from a large medical record system, consisting of 188 software components. Each component contains a number of files that are logically related. The components vary in the number of files they contain, ranging from 1 to over 800 files. In addition, components may contain lower level components. Initially, the software consisted of 173 software components. All three releases added functionality to the product. Between three to seven new components were added in each release. Over the three releases, fifteen components were added. Many other components were modified in all three releases. Of the 188 components, 99 had at least one defect in Releases 1, 2 or 3.

Table 4.1 shows some of the attributes for defect reports that the tracking database records. (The table does not list attributes that were not used in the analysis.)

Table 4.1: Attributes recorded for defect reports.

defect number
release identifier
phase in which defect was reported (development, test, post-release)
test site reporting defect
defective entity (code component(s), type of document by subsystem)
whether the component was new for a release
date the defect was reported
defect classification
“drop” in which the defect occurred

The defect classification indicates whether the defect is valid or not. Only valid defects are considered in this study. Software is released to testers in stages or “drops.” Developers work on drop “i+1” when testers test drop “i.” Each successive drop includes more of the functionality expected for a given release. Release 1 has three drops. Testing on the first drop started during week 54, on the second drop during week 63, and on the third drop during week 70. Release 2 has two drops. Testing on the first drop started during week 17, and on the second drop during week 22. Release 3 had one drop.

Table 4.2 shows the attributes for repair reports that the tracking database records.

Table 4.2: Attributes recorded for repair reports.

file change number
release identifier
defect report identifier for which the repair is being made
file identifier for the file being changed to repair the defect
type of change, e.g., delta, create, delete, rename, link
date of the file change
component to which the file belongs

There are approximately 6500 files in the system. The number of file changes related to defect repairs in a release was approximately 4200–5000. Of these, approximately 4000–4700 were delta changes and approximately 90 were new files created for 15–20 components.

## 4.2 Validity of the Case Study

A case study is only as good as the data on which it is based and as such has limitations. Questions concerning the validity of the study include:

1. What is the quality of the data and the process used to report the data?
2. Are there any other factors that affect which methods may be used?
3. Are the results of the case study generalizable?

The kind of data an organization reports drives the type of analysis. Most techniques require defect reports to have a unique identifier and information on release, report date, and life cycle phase. A fix report requires an identifier, information on release, the defect it

is attempting to repair, the file(s) being changed, and the component or module the file(s) belong(s). Only an organization with a good defect reporting process can take advantage of the techniques in the integrated method.

Interviews with testers at the beginning of the study allowed us to evaluate the process and the quality of the data. Specific interview questions include:

- What data about the product or test process are collected?
- Is data collected, analyzed, or processed after testing stops?
- How is the data recorded?
- How can one access the data?

An on-line tracking database records detailed defect reports (see Table 4.1) and repair reports (see Table 4.2). Developers, testers, and customers have access to the data through the tracking database. The tracking database does some validation of the data recorded for each defect report. For example, characters cannot be entered in numeric fields and dates must be entered in a specific format. All attributes used in the analysis were filled out for every defect report.

The organization in this case study is very good at documenting problems. The organization has a high quality development and testing environment. Assessment results show the software to be of a very high quality and it is reasonable to assume that the defect reporting process is likewise of very high quality.

Interviews with testers aided in understanding the testing process more fully. They helped to ascertain which factors other than testing activities could affect the data. These interviews also determined whether underlying assumptions of models were met, what the quality expectations were (this was important as it drove setting thresholds for the integrated method), what aspects of the testing process could be changed to improve testing effectiveness and efficiency, etc. Questions included:

- Does each group test the same system?
- Are system test sites independent? Specifically, do they know what defects the other groups reported? If so, how is this information used?
- Is there information on duplicate defects in the reports?
- Are all duplicates consistently recorded?

All test sites test the same system. They all have access to the on-line tracking database and know what defects the other groups report. There is information on duplicate defects in the reports, but duplicates are not consistently recorded. Test sites are encouraged to check the database before recording a defect and to not report duplicates. This has an impact on the capture-recapture and curve-fitting techniques which led to an adaption of these static defect estimation methods.

Interview questions that concern the assumptions for software reliability growth models include:

- Is testing performed according to specifications? Component by component?
- Are old components retested after new components are added?
- Are there failures during testing that require the faults causing the failures to be fixed to enable test to exercise all code?
- When system is in testing, is the system stable? That is, are changes only a result of failure correction?
- Do testers test a fully integrated system? If not, do we know the dates when new components became available for testing?

Testing is performed on components. Old components are retested after new components are added, although not necessarily in the same drop. Prior to the start of system testing, the system test group performs a qualification test to determine if the system is ready for test. This reduces the number of severe failures that would prevent testers from exercising all the code. During testing, the system changes. Some changes result from added functionality.

Interview questions concerning currently used methods to make release decisions include:

- Does the testing group currently use any stopping rules?
- How does the testing group determine when they are done testing and what criteria do they use?

System testers do not currently use stopping rules to make release decisions. Release decisions are schedule driven or based on the number of defects found in the last few weeks.

Like all case studies, this one has external validity, because it uses data from industry. It is, however, only one case study. The case study involves one environment and three releases of the same project. Not all developing and testing environments are of such high quality. Each project and environment has characteristics that need to be taken into account when determining what techniques to apply. Some of the characteristics that are important in using the integrated method include:

- An environment with a good defect reporting proces. Defects are reported via an on-line tracking database that validates data as it is entered and ensures all required attributes have values.
- A configuration management system for reporting changes to files.
- A tracking database that makes automated extraction easy.
- A project large enough so that there is enough data describing the results of testing (at least ten weeks).

Future work that applies this method to other projects and in other environments will improve its external validity.



## Chapter 5

# Fault-prone Component Analysis

### 5.1 Approach

Fault prone components are identified and analyzed not only across releases, but also within releases comparing across development and test. Within a release, development information is used to guide testing, rather than tracking components across releases or looking at relationships between components to analyze code decay. Defect data from development is used to improve system testing, both its effectiveness and efficiency. Various aspects of assessment and prediction are combined, and the data is used to support suggested improvement activities. The first release of a software product is used to derive testing guidelines. Successive releases validate them. This thesis derives testing guidelines for the case study in Sections 5.2.1 – 5.2.4. Section 5.2.5 validates them for our case study. The approach consists of the following steps:

1. Identify whether fault-prone components during development are a good predictor of fault-prone components during test and derive a test guideline for this situation.
2. If prediction has false positives and false negatives, determine whether other attributes of components or defects would improve this prediction. Examples of such attributes are whether a component is new or how severe a defect is. We restrict ourselves to data that is usually available in a defect database. We derive a test guideline for this situation.

3. Evaluate effects of fault-prone components on post-release problems. Derive additional test guidelines.
4. Summarize applicable test guidelines and evaluate how well they work on successive releases.
5. Perform cross release analysis.

### 5.1.1 Determination of Fault-prone Components

Components are ranked by the number of defects written against them during development and system test. Ohlsson, et al. [41] used such a defect ranking on overall defects in successive release to identify components that are fault-prone across releases to identify possible code decay. By contrast, we use this defect ranking to identify components that are fault-prone during development versus those that are fault-prone during system test.

Similar to [41] we consider a component "red," if it is fault-prone in both development and test. It is "green," if it is fault-prone in neither development nor test. Finally, a component is "yellow," if it is fault-prone in either development or test, but not both. Testing should focus on the "red" components. They should be tested as early and as thoroughly as possible.

Setting the threshold for the number of defects that makes a component fault-prone can be done in two ways:

- as a percentage of the ranked components (e.g., the top 25 percent)
- as a function of the total number of defects.

The decision is subjective. In our case, we wanted a fairly low threshold to avoid a large number of components that are fault-prone in either development or test, but not both. This reduces "false positives" and "false negatives" when using development fault-prone classification for prediction.

The threshold was set at about an order of magnitude less than the largest number of defects written against a component. The approach should also attempt to identify about the same number of components as fault-prone in development and test.

A guideline that de-emphasizes testing of components that are not fault-prone during development would generally work well and potentially save a lot of effort, since the vast majority of components fall into that category. A testing guideline that emphasizes more thorough testing of components that were fault-prone during development would correctly test some components more thoroughly and "over-test" others (those that were fault-prone during development, but not fault-prone during system test).

### 5.1.2 Consideration of Other Indicators

If the predictions of fault-prone components lead to many false positives or false negatives, consideration is given to other attributes of components and defects in order to improve the prediction. Attributes to consider include whether a component is new, the number of changes made to its files in development, etc. To reduce false negatives, one would want to consider components with these attributes as fault-prone, as they would also benefit from more thorough testing. We may still over-test components that are fault-prone during development, but are not fault-prone during system test. However, we would no longer miss components that are not fault-prone during development, but fault-prone during system test.

Another possible consideration is with respect to defect severity level. Severity levels range from 1 to 4. Level 1 is the highest. Analysis may indicate benefits in using severity levels for prediction of fault-prone components.

### 5.1.3 Comparison to Fault-Prone Components in Post-Release

Our next question was: Is system test finding all problems and what is the nature of the problems that are first detected in the field? To investigate this, we compared fault-proneness in system test to post-release fault problems. Because the number of post-release faults is so much lower than the number of faults found in system test, the fault-prone threshold for post-release defects per component is set to a smaller number.

Components that are both fault-prone in system test and post-release are identified. Harder to identify before release are the components that are normal during system test, but fault-prone after release. Other indicators of components, e.g. whether they were new

for the release, whether they experienced a lot of change due to enhancements, etc. are investigated. If these indicators point to a higher likelihood of post-release problems, they could be used to identify potentially fault-prone components for more thorough system test. If these types of components are easily identified, components of those types are grouped with components that are fault-prone during system test. Examples of these types of components include new ones or those affected by changes or enhancements.

## 5.2 Results

Setting the threshold to a straight percentage of the number of components may not work, especially for high quality code. In this study, a threshold of 25 percent is too much. It requires defects to be found in 45 components. In Release 1, development found defects in exactly 45 components, and many of these components only had one defect. (See Table A.2.) A component having one reported defect arguably is not fault-prone. System test only had 32 components with a reported defect. The threshold needs to be set to a lower percentage of components. This case study explored the threshold set to 10 percent of the components. An alternative is to set the threshold to an “order of magnitude” (or 10 percent) less than the total number of defects. The advantage is that the number of components considered fault-prone can vary without having to consider explicitly how high or low the quality of the software is.

### 5.2.1 Fault-prone Component Analysis Applied to Release 1

Table 5.1 shows the diffusion matrix for Release 1 using 10 percent of the ranked components as a threshold. A diffusion matrix shows the number of components that are fault-prone or normal during development and system test. Table 5.1 shows that 10 of the 18 components identified as fault-prone in development stayed that way in system test (top left cell). Of these ten, five were new components. The other eight are normal in system test (top right cell). Using development fault-prone components as a predictor of fault-proneness during test would classify these eight components incorrectly. If the test guideline stated to test development fault-prone components more thoroughly during system test, this would

cause system test to overtest these eight components. In addition, Release 1 had eight fault-prone components that were normal (not fault-prone) in development. Had the test guideline stated to test components less intensively, if they were not fault-prone during development, these components would not be tested enough and test could miss defects.

Table 5.1: Diffusion Matrix for Release 1 using a threshold of 10%.

Threshold = 10% 18 components	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	10 (5 new)	8 (1 new)
Development Normal	8 (2 new)	154

The threshold may also be set as a function of the number of defects. Table 5.2 shows the highest number of defects written against a single component in development and test for each release. We set the threshold at about an order of magnitude less than the largest number of defects written against a component during development. For Release 1, the maximum number of defects in development is 143. The threshold was an order of magnitude lower, set at ten defects per component. This threshold applied to both development and system test. Table 5.2 shows the number of components that were over this threshold and considered fault-prone and the corresponding percentage of components that were fault-prone.

Table 5.2: Statistics on fault-prone components using order of magnitude threshold.

	Release 1 (180 components)		Release 2 (185 components)		Release 3 (188 components)	
	development	test	development	test	development	test
Max defects in a comp.	143	27	115	32	7	41
# comps over threshold (10 defects for Rel.s 1&2) (4 defects for Rel. 3)	10	6	6	5	1	3
% comp. over threshold	5.6%	3.3%	3.2%	2.7%	0.5%	1.6%
Mean # defects for comp. over threshold	36.06	19.33	35.17	20	7	19.33
Std. Dev.	40.57	6.12	39.97	7.38	0	18.82

In Release 1, ten components were fault-prone in development, six were fault-prone during system test. Table 5.3 shows to what degree a component that is fault-prone (or normal) during development stays that way during system test. If one were to predict fault-proneness

Table 5.3: Diffusion matrix for Release 1 using order of magnitude threshold.

Threshold $\geq 10$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	5 (4 new)	5 (1 new)
Development Normal	1 (1 new)	169 (1 new)

during testing based on whether a component was fault-prone during development, the diffusion matrix can be used to evaluate the quality of the prediction model. In this case, it correctly predicts five components as fault-prone during system test and 169 components as normal during system test. It shows five false positives (development fault-prone, but normal during system test) and one false negative (normal during development, but fault-prone during system test). If one were to use the classification of fault-prone components during development as a guide to increase testing effort for fault-prone components and to decrease test effort for normal components, this would lead to “overtesting” five components (an inefficiency) and not testing one component enough (a decrease in effectiveness). This is not quite the accuracy desired.

In this case study, using an order of magnitude difference in the number of defects for a component also included a component that was consistently fault-prone in later phases (system test and post-release) of Release 1 and throughout all phases (development, system test, and post-release) in Release 2. Because the quality of a system may not be known prior to system test, it would be difficult to set the threshold based on a percentage of ranked components. Therefore, setting the threshold as a function of the maximum number of defects found in a single component is recommended.

A testing guideline that emphasizes more thorough testing of components that were fault-prone during development would work well, catching components that are fault-prone in system test, and overtesting a few. A guideline that de-emphasizes testing of components that are not fault-prone during development would generally work well and potentially save a lot of effort, since the vast majority of components fall into that category.

## 5.2.2 Evaluation of Other Indicators

Table 5.3 also showed how many of the components in each category were new for Release 1. Interestingly, all but one were fault-prone in either development or test, and thus could benefit from more thorough testing. This led to a second testing guideline: Test new components more thoroughly. It is based on the assumption that new components will be fault-prone during system test and should be tested more thoroughly. This guideline removes the false negative in the diffusion matrix of Table 5.3 by grouping it correctly with the components that are fault-prone during system test. We still over-test components that are fault-prone during development, but are not fault-prone during system test. However, this guideline no longer misses the new component that is not fault-prone during development, but fault-prone during system test. Table 5.4 shows the results of this analysis.

Table 5.4: Diffusion Matrix including new components as fault-prone in Release 1.

Threshold $\geq 10$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	6 (5 new)	6 (2 new)
Development Normal	0	168

Another possible refinement of test guideline 1 is to refine it with respect to defect severity level. Severity levels range from 1 to 4. Level 1 is the highest. Table 5.5 shows the maximum number of defects found during development and system test for a component for each severity level.

Table 5.5: Maximum defects for a component by severity level for Release 1.

	Severity Level			
	1	2	3	4
Development	54	47	27	15
System Test	5	17	21	6

Because the maximum number of defects in each severity level is lower than the cumulative number of defects, one needs to set a new threshold. An order of magnitude less than the number of defects in development is 6, 4, 3 and 2 defects for severity levels 1, 2, 3 and 4,

respectively. The threshold is set to 4, the mean average of these, and used for all levels. The results of the fault-prone analysis between development and testing for Release 1 are presented in Tables 5.6 – 5.9. The analysis does not indicate any benefits in using severity levels for prediction of fault-prone components. Analyzing defects by severity level does not improve the predictions. Thus testing guidelines based on severity level would not improve the process.

Table 5.6: Diffusion Matrix for Release 1 by severity 1.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1 (1 new)	10 (5 new)
Development Normal	1 (1 new)	168

Table 5.7: Diffusion Matrix for Release 1 by severity 2.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1	6 (3 new)
Development Normal	3 (2 new)	170 (2 new)

Table 5.8: Diffusion Matrix for Release 1 by severity 3.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	4 (4 new)	3
Development Normal	3 (2 new)	170 (1 new)

Table 5.9: Diffusion Matrix for Release 1 by severity 4.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	3 (2 new)
Development Normal	2 (2 new)	175 (3 new)



### 5.2.3 Comparison to Fault-Prone Components in Post-Release

Table 5.10 shows the results of comparing components that are fault-prone in system test to components that are fault-prone in post-release. Using the order of magnitude threshold identified components with two or more post-release defects as fault-prone, seven in all.

Table 5.10: Diffusion Matrix for System Test versus Post-Release in Release 1.

Post release threshold $\geq 2$ defects System Test threshold $\geq 10$ defects	Prediction (Post Release)	
	Fault-prone	Normal
System Test Fault-prone	1	5 (5 new)
System Test Normal	6	168 (2 new)

Only one component was fault-prone both in system test and post-release (it was also fault-prone in development). It is not a new component and easily identifiable by the high number of defects throughout the life cycle. Obviously, such a brittle component is cause for concern. Five components were fault-prone during system test, but were repaired before release and problem-free (all of them new components). This indicates thorough testing and excellent repair work for new components.

None of the components that are normal during system test, but fault-prone after release, are new. These old components may have been affected by changes or enhancements. One possible cause for missing these components may be that regression test did not test them thoroughly enough. The defect data is not detailed enough to explain this phenomenon. A suggestion for improving testing would be to assess (and possibly improve) change impact analysis and regression testing.

Overall, development and system test are doing a very good job of testing and repairing new components. The exposure rate of defects for new components is high and these components have very few defects after release indicating that the problems were corrected. Analysis across phases within the same release, therefore, will not help guide system test in preventing post-release defects.

## 5.2.4 Proposed Testing Guidelines

To fine tune the testing process, one should develop guidelines that are specific to the situation in a project group and are based on project data from that environment. Data from Release 1 was used to derive the guidelines. Table 5.11 shows these guidelines. There

Table 5.11: Testing guidelines derived from Release 1 data.

Testing Guidelines	
1.	Test development fault-prone components more thoroughly. (Test components not fault-prone in development less.)
2.	Test new components more thoroughly.

were also several candidate guidelines that did not work well in this environment. They related to defect severity levels and the quality of the repair work between development and system test, and before release.

## 5.2.5 Fault-Prone Component Analysis Applied to Releases 2 and 3

To evaluate whether the test guidelines are useful, they were applied and evaluated on Release 2 and Release 3 of the same system.

In Release 2, the maximum number of defects per component found in development is 115. In system test it is 32. We chose the same threshold (10) as in Release 1. Any component with more than ten defects was considered fault-prone.

In Release 2, six components were identified as fault-prone during development. Only two of them were also fault-prone during system test. Applying guideline 1 would have over-tested the other four. There were five fault-prone components in system test, three of which were not fault-prone during development and would not have been tested enough according to guideline 1 (cf. Table 5.12). This latter misclassification is more dangerous.

Table 5.12: Diffusion matrix for Release 2.

Threshold $\geq 10$	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	2 (1 new)	4
Development Normal	3 (2 new)	176 (2 new)

Applying the more conservative guideline 2 improves matters. Table 5.13 shows the results. While we “over-test” more components than with the first guideline, we only undertest one (old) component that is fault-prone during system test.

Table 5.13: Diffusion Matrix including new components as fault-prone in Release 2.

Threshold $\geq 10$	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	6 (5 new)	4
Development Normal	1	174

As in Release 1, defect data analyzed for severity do not give better results and do not warrant additional test guidelines. (See Appendix A.2.2.)

Next, we evaluated whether system test defects are a good indicator of post-release defects. Table 5.14 shows the results. They quite strongly point out that any component that was fault-prone during test was **not** fault-prone after release. This is good news: the problems that were identified during testing were corrected before release. Development and system testing are doing a very good job of eliminating problems. Improvements in testing should focus on the five components that were not fault-prone during development or test, but showed post-release problems. These components were not new. One possibility is to investigate whether the problems could have been prevented with more extensive change impact analysis and regression testing. This requires more detailed data than we had available.

Table 5.14: Fault-prone components in System Test versus Post Release in Release 2.

Post Release Threshold $\geq 2$ System Test Threshold $\geq 10$	Prediction (Post Release)	
	Fault-prone	Normal
System Test Fault-prone	0	5 (1 new)
System Test Normal	5	175 (4 new)

In Release 3, the maximum number of defects per component found in development is 7. In system test it is 41. We set the threshold to 4. Any component with more than four defects was considered fault-prone.

In Release 3, only one component was identified as fault-prone during development. The component was also fault-prone during system test. Applying guideline 1 would not have resulted in over-testing any components. There were 3 fault-prone components in system test, two of which were not fault-prone during development and would not have been tested enough according to guideline 1 (cf. Table 5.15). This latter misclassification is more dangerous.

Table 5.15: Diffusion Matrix for Release 3.

Threshold $\geq 4$	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1	0
Development Normal	2 (1 new)	185 (2 new)

Table 5.16 shows the results of applying the more conservative guideline 2. While we “over-test” two more components than with the first guideline, we only undertest one (old) component that is fault-prone during system test.

Table 5.16: Diffusion Matrix including new components as fault-prone in Release 3.

Threshold $\geq 4$	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	2 (1 new)	2 (2 new)
Development Normal	1	183

Next, we evaluated whether system test defects are a good indicator of post release defects for Release 3. Table 5.17 shows the results. Again, they point out that any component that was fault-prone during test was **not** fault-prone after release: The problems that were identified during testing were corrected before release.

Table 5.17: Diffusion Matrix for System Test versus Post Release in Release 3.

Post Release Threshold $\geq 2$ System Test Threshold $\geq 4$	Prediction (Post Release)	
	Fault-prone	Normal
System Test Fault-prone	0	3 (2 new)
System Test Normal	2	183 (1 new)

### 5.2.6 Cross Release Analysis

When successive releases are to be tested, it might be useful to have a test guideline on how to treat components that have been fault-prone in development, test, or in the field in prior releases.

A test guideline would be to treat a component as fault-prone (and focus attention on it) in system test, if it was fault-prone during system test or in the field in the prior release. Between releases, pay particular attention to components that had development and post-release problems in prior releases, but where system testing found few problems. Likewise, identify components that had problems in development, test, and after release in the prior release.

Tables 5.18 and 5.19 show the diffusion matrix analyses of this approach for the medical record system. The results indicate that a guideline to consider a component fault-prone in system test, if it was fault-prone during system test or in the field in the prior release does not help in this case study. This is because problems, once identified, are fixed.

Table 5.18: Diffusion Matrix for fault-prone components across Releases.

	Release 2			
Release 1	Test Fault-prone	Test Normal	PostRelease Fault-prone	PostRelease Normal
Test or PostRelease Fault-prone	3	9	3	9
Test or PostRelease Normal	2 (2 new)	171 (3 new)	5	168 (5 new)

Table 5.19: Diffusion Matrix for fault-prone components across Releases 2 and 3.

	Release 3			
Release 2	Test Fault-prone	Test Normal	PostRelease Fault-prone	PostRelease Normal
Test or PostRelease Fault-prone	1	11	2	10
Test or PostRelease Normal	2 (2 new)	174 (1 new)	0	176 (3 new)

We also investigated whether there are some components that are not fault-prone during system test, but are fault-prone in development and after release. This would be an indication of insufficient testing. Only one component was fault-prone in development and post-release in all three releases that was not fault-prone in system test in any of the releases. This component needs to be tested more thoroughly. Further, there was only one component that was fault-prone in development, system test and post-release for the first two releases and for system test in Release 3. This is not so much an indication of insufficient testing, but of insufficient repair. In both cases, the number of such components is very small (one). This speaks for the quality of the development organization.

### 5.3 Summary

We evaluated development, system test and post-release defect data to determine whether additional test guidelines based on this data might be helpful. The following ideas were helpful.

1. Use an order of magnitude less than the maximum defects in a component to set the threshold for fault-prone components. This is less arbitrary than selecting a percentage of components as fault-prone.
2. Test components that are fault-prone during development more thoroughly. They are likely to be fault-prone during system test. De-emphasize testing of components that are not fault-prone during development.
3. Test new components more thoroughly, whether they are fault-prone during development or not.
4. Pay particular attention to components that had development and post-release problems, but where system testing found few problems (in our case, this was only one component). Likewise, evaluate the component that had problems in development, test, and after release in the prior release.

5. Improve impact analysis of enhancements to existing components and determine whether improvements in regression testing could have prevented problems from slipping through.

The following ideas are not helpful.

1. Analyze defects found in components by severity level. We surmise that this is because severity is not a good indicator of how complete repair will be.
2. Set the threshold for fault-prone components to percentage values, especially when the quality of the system is not known.

This being a case study, one cannot expect these guidelines to improve every project, although they certainly are sensible. The project studied has characteristics that need to be taken into account when determining whether applying these guidelines would improve system test performance:

- Few problems remain undetected.
- The number of post-release problems is very low.
- Most known problems are fixed before release.
- In each release, the code is of very high quality. Only two components out of 188 are fault-prone in all releases.

One might say, we are “gilding the lily”. Further, even the “almost perfect” project can benefit from the guidelines we derived from the defect data. For high quality development environments like the one analyzed, the key issue for testers is where to put emphasis, and where not to. This has the greatest potential for being more efficient without sacrificing effectiveness.

## Chapter 6

# Fault Architecture Analysis

### 6.1 Approach

In order to validate that an assessment tool is useful, it is important to empirically evaluate whether it works on more than one project. We chose a very different project (medical record system) from that used in the prior case study (system software) [65, 66]. This thesis adds to the study in [65, 66] by applying the approach to a new data set. In addition, rather than just applying one defect cohesion measure and one defect coupling measure, the approach applies all measures in Section 2.2.3 so as to compare how results might differ. Unlike the prior case study, which only used post-release data, this study uses defect data from development, system test and post-release.

### 6.2 Release Analysis

The steps in the approach are:

1. Determine both defect cohesion measures of Section 2.2.3 and identify fault-prone components.

Rather than choose between the two defect cohesion measures described in [40, 66], our approach uses both to identify problematic components that should be considered for more intense testing or for rearchitecting.

This study considers a component,  $C$ , fault-prone, if the defect cohesion measure,

$$Co_{\langle C \rangle} \geq 0.1 * Co_{max}$$



where

$Co_{<C>}$  is defined in 2.1 or 2.2.

$$Co_{max} = \max \{ Co_{<C_i>}, 0 \leq i \leq n \}.$$

$n$  is the number of system components.

2. Determine both defect coupling measures and identify fault-prone component relationships.

Unlike the prior studies [40, 65, 66], this study sets the threshold to an order of magnitude (or 10 percent) less than the highest defect coupling measure. This is an alternative to setting the threshold to some percentage of fault-relationships. The purpose is to focus on fault-relationships with the highest ranks. This reduces the number of relationships, so that testing or reengineering efforts may focus on components that are much worse than others.

A component is considered fault-prone, if either of the following applies:

- (a) A fault relationship between two components  $C_1, C_2$  is fault-prone, if

$$Re_{<C_1, C_2>} \geq 0.1 * Re_{max}$$

where

$$Re_{max} = \max \{ Re_{<C_i, C_j>}, 0 \leq i, j \leq n, i \neq j \}$$

$n$  is the number of system components.

- (b) A component  $C$  is fault-prone, if

$$TR_{<C_i>} \geq 0.01 * TR_{max}$$

where

$$TR_{max} = \max \{ TR_{<C_i>}, 0 \leq i \leq n \}$$

$n$  is the number of system components.

### 3. Create a *Fault Component Directory Structure*.

The fault-prone component directory structure shows the components identified as fault-prone according to either one of the defect cohesion measures.

Components carry the prefix **d**, if they are identified as fault-prone using only the basic defect cohesion measure 2.1. They carry the prefix **f**, if they are identified as fault-prone by only the multi-file defect cohesion measure 2.2. If there is no prefix, the component is identified as fault-prone using both measures. Components marked in bold are fault-prone because of fault relationships. The components not in bold are fault-prone components with internal problems instead of fault-prone relationships with other components.

### 4. Create the *Component Level Fault Architecture Diagrams*.

Components are included in the diagrams, if they have been identified as fault-prone with respect to at least one of the defect coupling measures.

In addition, those components that are fault-prone according to the defect cohesion measures are included in an inset to the diagram for comparison purposes. Components that are fault-prone according to both a defect cohesion measure and a defect coupling measure are indicated in bold.

Analysis of the component level diagrams can give developers and testers an indication of what kind of problems the system has and where they are. For example, one can determine if most problems are local or involve relationships between components.

Comparing across releases makes it possible to identify components that are repeatedly fault-prone or in fault-prone relationships. Developers and testers can then focus their attention on them in terms of either re-engineering them or testing them more intensely.

### 5. Lift.

The lift operation abstracts the fault-prone relationships at the component level to the subsystem level. These are presented in *Fault Architecture Diagrams* for each release

and a *Cumulative Release Diagram*. A subsystem is defined as the level immediately below the system level (or root).

Figure 6.1 shows an example of a fault component directory structure. In this example,  $S_1$  and  $S_2$  are subsystems. Not all components are at the same level. A component belongs to a subsystem, if there is a path from the subsystem to the component. In the example, component  $C_{11}$  belongs to  $S_1$ . Components  $C_{21}$ ,  $C_{22}$ , and  $C_{23}$  belong to subsystem  $S_3$ .

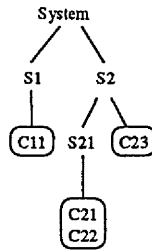


Figure 6.1: Example of a Fault Component Directory Structure Diagram.

The steps the lift operation are:

- (a) Determine defect coupling measures for the subsystem level.

For any two subsystems,  $S_1$  and  $S_2$ , the defect coupling measures are defined as follows.

Let  $C_{11}, \dots, C_{1n}$  be components in subsystem  $S_1$  and  $C_{21}, \dots, C_{2m}$  be components in subsystem  $S_2$ , then

$$Re_{\langle S_1, S_2 \rangle} = \sum_{i=1}^n \sum_{j=1}^m Re_{\langle C_{1i}, C_{2j} \rangle} \quad (6.1)$$

where  $Re_{\langle C_{1i}, C_{2j} \rangle} \geq \text{threshold}$  for the multi-file defect coupling measure for the release.

$$TR_S = \sum_{i=1}^n TR_{C_i} \quad (6.2)$$

where  $TR_{C_i} \geq \text{threshold}$  for cumulative defect coupling measure for the release.

- (b) Create subsystem level diagrams.

We construct *Fault Architecture Diagrams* for the three releases using the defect coupling measures for the subsystem level. Nodes in the diagram represent

subsystems that occur in at least one fault architecture diagram. Arcs indicate the magnitude of the problem between the subsystems. Nodes in the diagram represent subsystems that occur in at least one fault architecture diagram. Arcs indicate the magnitude of the problem.

Next we aggregate the fault architecture diagrams into a cumulative release diagram. The *Cumulative Release Diagram* illustrates persistent problems within and between subsystems. This diagram aggregates subsystem level relationships across releases. The arc annotations in the diagram describe fault relationships between subsystems that persisted across releases.

The questions we address are:

- What are the subsystems with the majority of problems?
- Are problems between subsystems or are they local within the system?
- Do problems in one release occur in other releases?

Analysis of the subsystem level diagrams can give developers and testers an indication of whether problems exist between subsystems or are local within systems. In addition they indicate problems that occur over several releases.

### 6.3 Single Phase Analysis

Studies in [41, 40, 65, 66] used defect measures from successive release to identify components that are fault-prone or are in fault-prone relationships across releases to identify possible code decay. By contrast, this study also applies these techniques to identify components that are fault-prone or in fault-prone relationships during development, system test and post-release to determine those components that should be tested more intensely.

This study derives and analyzes component level fault architecture diagrams for each of the development phases. The questions we address are:

- Is system test finding all problems?
- Do problems in development appear in test?
- What is the nature of the problems that are detected in the field?

To investigate this we perform across-phase analysis between earlier phases of development and post-release, comparing fault problems in development, system test, and post-release. Comparing across development phases within a release makes it possible to see whether some components are repeatedly in fault-prone relationships, whether the problems are prevalent in particular phases, and whether they are repaired. For example, few fault-prone relationships during development, but many during testing indicate that development is creating defects, but either does not find them or does not correct them during development. A preponderance of fault-prone relationships during post-release indicates insufficient system testing. The objective of this analysis is to assess the quality of the product and identify possible architectural problems. By analyzing earlier defect data, the information may be used to guide testing, that is to improve its effectiveness.

We investigate whether development fault architectures can identify the parts of the software that need to be tested more intensely. We validate these assessments using system test data from the same release. We also use the development and system test fault architectures to identify fault-prone components after release and validate our assessments using post-release data.

## 6.4 Case Study Results

### 6.4.1 Release Analysis

#### 6.4.1.1 Defect Cohesion Measures

This study identified the most fault-prone components in each release using the two ways to measure defect cohesion. Table 6.1 shows how many components were identified as fault-prone in each release. The last row shows the number of components that are considered fault-prone by at least one of the defect cohesion measures. Order of magnitude was used to set the threshold.

If there is a large overlap in the number of components that are considered fault-prone by both rankings, one can assume that the two measures are similar. In this case it means that components with a lot of defect reports also require changes in multiple files. If there are a lot of components that are fault-prone according to one method and not the other, the

Table 6.1: Number of components identified as fault-prone in Releases 1 – 3.

	Release 1	Release 2	Release 3
Basic defect cohesion measure	17	18	3
Multi-file defect cohesion measure (Eq. 2.2)	6	10	2
Both combined	18	24	5

measures are different. In this case it means a component with few defect reports may not require changes in multiple files to repair its defects. Further a component that required many multiple file changes may have few defects reported.

Table 6.1 shows that the basic defect cohesion measure results in identifying 17 fault-prone components in Release 1, while the multi-file defect cohesion measure identifies six fault-prone components. Between the two methods, there is an overlap of five components. Thus, the multi-file measure mostly flags components that have already been identified as fault-prone by the basic measure. However, the basic measure includes components not flagged by the other measure. In Release 2, the two defect cohesion measures identify 18 and 10 fault-prone components with an overlap of four components between the two methods. Table 6.1 shows that in Release 3, the defect cohesion measure identifies three components, the multi-file defect cohesion measure identifies two components. There is no overlap between the fault-prone components in Release 3. (The *Fault Component Directory Structure* shown in the next section displays these components.)

The two methods for computing the defect cohesion measure identify a few of the same components as fault-prone. There are sufficient differences between what the two measures identify as fault-prone components that it warrants using both methods. This is more noticeable when looking at the *Fault Component Directory Structure* (see Figure 6.2).

Table 6.2: Number of releases in which components were fault-prone.

		Number of Times Fault-Prone			
		0	1	2	3
Number of Components	Basic defect cohesion measure	161	15	11	1
	Multi-file defect cohesion measure (Eq. 2.2)	174	10	4	0
	Both combined	156	21	11	1

Table 6.2 shows many of the 188 components were never identified as fault-prone by either method for computing defect cohesion. The last row shows that 33 components were identified as fault-prone in at least one release using one of the defect cohesion measures. At most one or two components were identified as fault-prone in all three releases and these were components that had defect cohesion measures much, much higher than any other components.

### 6.4.1.2 Fault Component Directory Structure

Figure 6.2 shows the *Fault Component Directory Structure*. Components in the diagram include those identified as fault-prone using either the basic defect cohesion measure (labeled with prefix 'd') or the multi-file defect cohesion measure (labeled with prefix 'f') in at least one release. Several components are identified as fault-prone using both measures (no label).

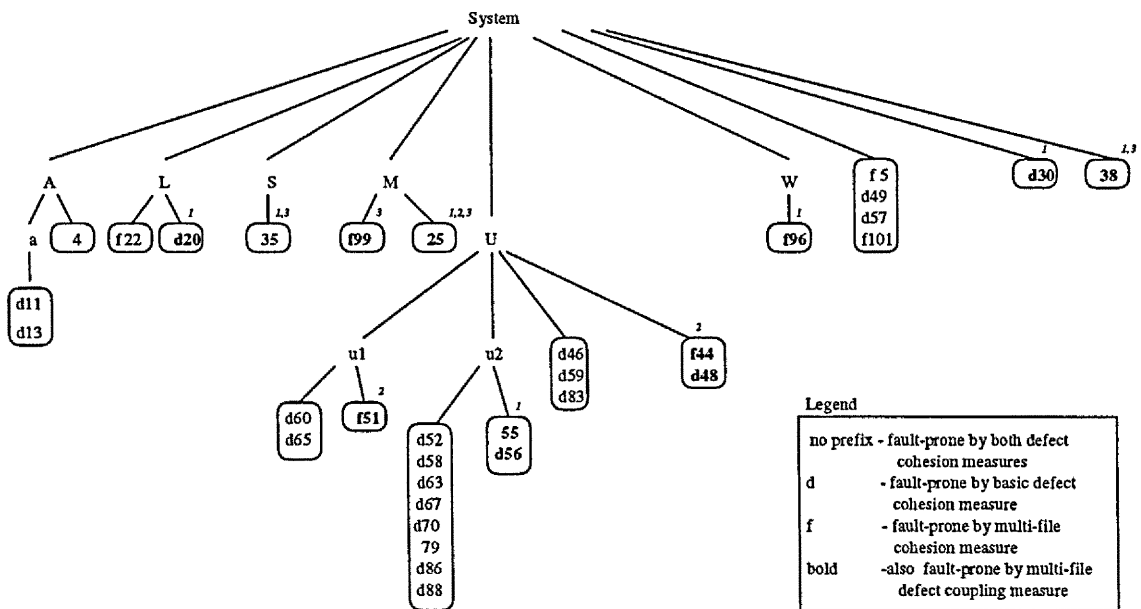


Figure 6.2: Fault Component Directory Structure.

Not all the components identified as fault-prone using one method were fault-prone using another. The two methods for computing the defect cohesion measure occasionally identify different components as fault-prone. There are 32 components that are fault-prone using either the basic defect cohesion measure or the multi-file defect cohesion measure in the three

releases. Of these 32, 20 are fault-prone according to the basic defect cohesion measure, that is they had a lot of defect reports, but not a lot of file changes. Six components are fault-prone according to the multi-file defect cohesion measure, indicating they had a lot of file changes but not a lot of defect reports. Six components are fault-prone according to both measures, that is they had a lot of defect reports and a lot of file changes. Since there is not a large overlap of components that are fault-prone by both measure (6 out of 32), this indicates that the measures identify different components and both measures should be used to determine components that require more intense testing or reengineering, rather than choosing one method over another.

It should be noted that components vary in size and in the number of files they contain. Component 38, for example, which has 230 files, may be considered a subsystem. Component 79, which is several levels down in the existing hierarchy, consists of 183 files. Component 86, which is at the same level as component 79, consists of 1 file.

#### 6.4.1.3 Defect Coupling Measures

Table 6.3 summarizes the results of applying the defect coupling measure. Column 2 identifies the number of components involved in fault relationships for each release. Column 3 gives the number of fault relationships (arcs) between components. This shows that

Table 6.3: Fault relationship information.

	Number of Components	Number of Relations
Release 1	65	245
Release 2	74	61
Release 3	34	56

between Release 1 and Release 3, the number of fault relationships decreased to less than one quarter of the number of such relationships in Release 1. Similarly, the number of components in fault relationships was cut almost in half by Release 3.

To determine which are the most problematic of these relationships, we apply the order of magnitude threshold. Table 6.4 shows the results. Column 2 identifies the number of components involved in fault-prone relationships for each release using the multi-file



defect coupling measure. Column 3 identifies the number of fault-prone relationships between components. Column 4 identifies the number of components involved in fault-prone relationships using the cumulative defect coupling measure. (The number of fault-prone relationships is the same for both measures.) Column 5 identifies the number of components in fault-prone relationships using either measure.

Table 6.4: Fault-prone relationship information using the defect coupling measures.

	$Re_{\langle C, C_i \rangle}$ measure		$TR_C$ measure	Combined
	Components	Num. Relations	Components	Components
Release 1	15	10	21	21
Release 2	10	7	10	10
Release 3	4	2	6	6

These are much more manageable numbers of components to focus attention. Table 6.4 also shows that using both defect coupling measures to identify components with a lot of fault relationships does not result in more components than using the  $TR_C$  measure alone. In this study, then, using only the  $TR_C$  measure to identify components for more intense testing is sufficient.

Based on these results, we updated the *Fault Component Directory Structure* in Figure 6.2, marking components in fault-prone relationships in bold. Bold components are annotated with the release identifiers in which they were considered relationship fault-prone. The components not in bold are fault-prone components with internal problems.

#### 6.4.1.4 Component Level Fault Architecture Diagrams

Figures 6.3 – 6.5 show the *Fault Architecture Component Level Diagrams* for Releases 1, 2, and 3. Note that the *Fault Architecture Component Level Diagrams* include components at different levels of the existing logical structure. In addition, the components are of various sizes in terms of the number of files they include. The diagrams show the components in fault-prone relationships (arcs are annotated with the  $Re_{C, C_i}$  value). Components are also included, if they have high cumulative defect measures (The  $TR_C$  measure is shown in parenthesis). Components in bold are also fault-prone according to one of the defect cohesion measures.

Figure 6.3 shows ten fault-prone relationships involving 15 components in Release 1. Six components have no fault-prone defect relationships with any other component. These components, 10, 46, 51, 65, 79 and 114 are included, because they have high cumulative defect measures. The other 15 components were fault-prone according to both defect coupling

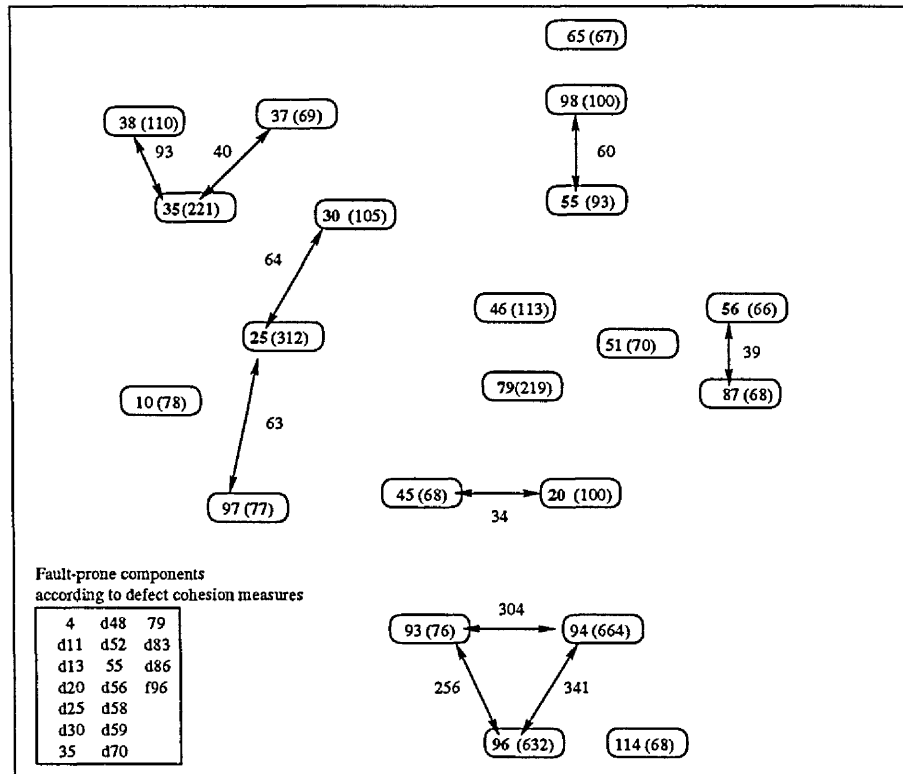


Figure 6.3: Release 1 Component Level Fault Architecture.

measures. Several components were involved in more than two fault-prone relationships. Components 25 and 35 were involved in two fault-prone relationships. Components 93, 94 and 96 were in fault-prone relationships together. All other components had only one fault-prone relationship.

In Release 2 there are seven fault-prone relationships involving ten components. Component 25 is again in a fault-prone relationship, but this time with a different component. All of the components were fault-prone based on either of the two measures. No additional components were identified based on the cumulative defect coupling measure. In effect, both measures included exactly the same components in Release 2. Figure 6.4 shows fewer components and fewer fault-prone relationships, indicating that some prior problems have

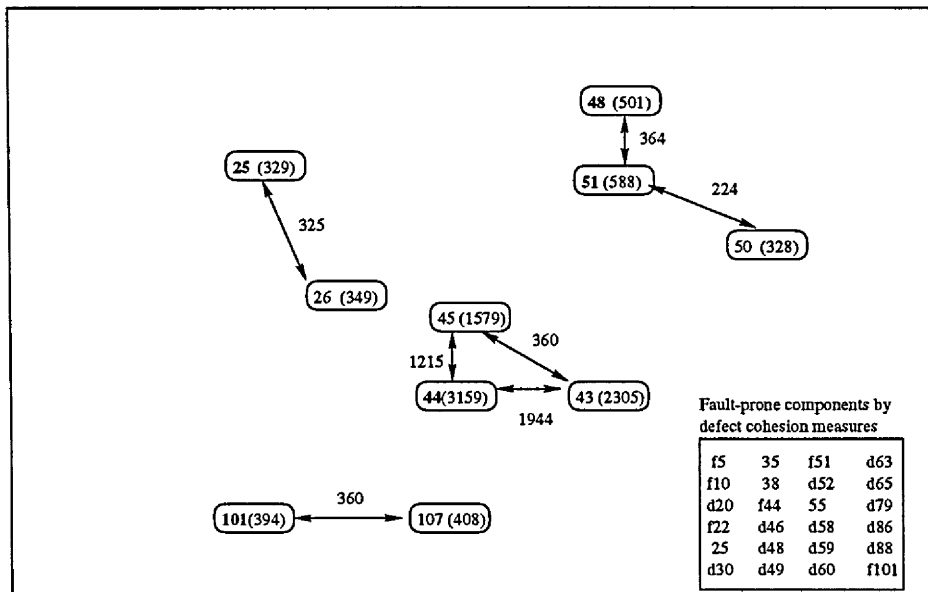


Figure 6.4: Release 2 Component Level Fault Architecture.

been successfully fixed. This trend continues into Release 3. There is, however, one component (25) that was relationship fault-prone in all three releases. There are two fault-prone

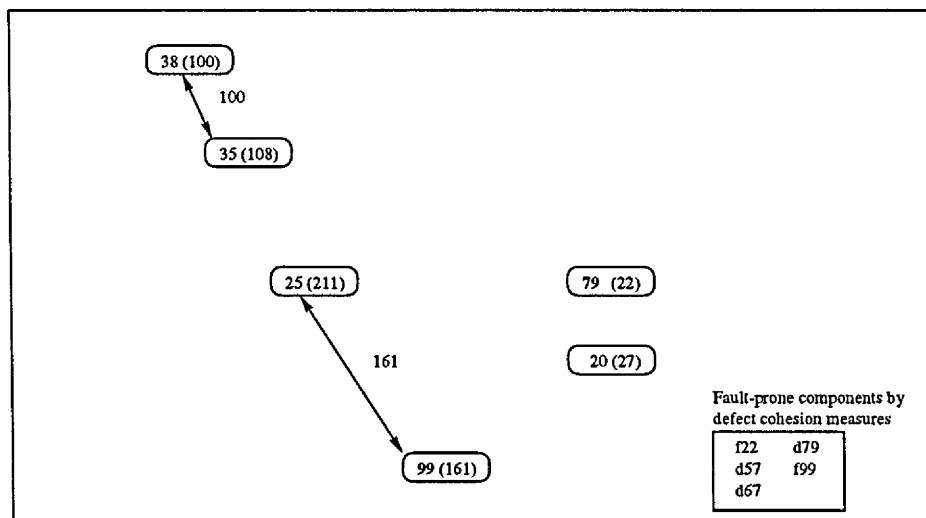


Figure 6.5: Release 3 Component Level Fault Architecture.

relationships involving four components and two other components that have high cumulative defect coupling measures. Component 35 reappears again in fault-prone relationships (it had disappeared in Release 2).

Using the  $Re_{\langle C, C_i \rangle}$  by itself did not include any more fault-prone components than the  $TR_C$  measure. Since all components considered fault-prone by the  $Re_{\langle C, C_i \rangle}$  measure are also fault-prone by the  $TR_C$  measure, the  $TR_C$  might be considered a good choice for a defect coupling measure all on its own.

The component level diagrams indicate that the most fault-prone parts of the software are not due to relationship problems, but to local problems, that is, problems internal to the components.

Analysis of the fault component directory structure shows that 32 components out of 188 are locally fault-prone. This is a low number, less than 20 percent of the components have problems. Out of these 32 components, almost half are also relationship fault-prone in at least one release. The fault-prone components are at various levels in the directory structure.

In looking at the component level fault architecture diagrams, one can see that many components that are in fault-prone relationships are also locally fault-prone (8 out of 21 in Release 1, 5 out of 10 in Release 2, and 2 out of 6 in Release 3). In addition, several components are in more than one fault-prone relationship. More attention should be focused on these components.

A cross-release analysis of the component level fault architecture diagrams indicates that the number of components in fault-prone relationships and those with a large number of fault relationships decreases in successive releases. This indicates that components are being repaired. The number of fault-prone components increased in Release 2. Problematic parts of the software had less to do with relationships between components and more to do with internal problems. These problems should be less difficult to fix. In Release 3, both kinds of problems continue to decrease. Cross-release analysis also reveals that a few components are consistently problematic: Components 25, 35, 38, and 45 are fault-prone in at least two releases. Components 25 and 35 are fault-prone both in terms of relationship problems and local problems. System developers and testers should focus on these components in successive releases.

### 6.4.1.5 Lift the Fault Relationships to the Subsystem Level

Fault architecture diagrams illustrate subsystems that have a majority of the problems. They also indicate whether relationship problems are between subsystems or more within subsystems.

Figures 6.6 – 6.8 show there are only six fault-prone relationships between subsystems in the three releases. There are few fault relationships between components, hence there are few between subsystems, and the degree of fault-proneness is small. (Table A.14 in Appendix A.4 shows to which subsystem problematic components belong.)

Figure 6.6 shows that in Release 1, subsystems M, S, 30, 38, U, and L have fault-prone relationships with other subsystems. Subsystems S, M, U also have fault-prone relationships internal to the subsystem. W has only internal fault-prone relationships.

Figure 6.7 shows that in Release 2, we have three fault-prone relationships between subsystems. The first is between subsystems M and X. The second is between subsystem 101 and subsystem 107. The third is between the subsystems U and B.

Figure 6.8 shows that in Release 3, we have only one fault-prone relationship between subsystems. It is between subsystems S and 38. Subsystem M has fault-prone relationships that are internal to the subsystem.

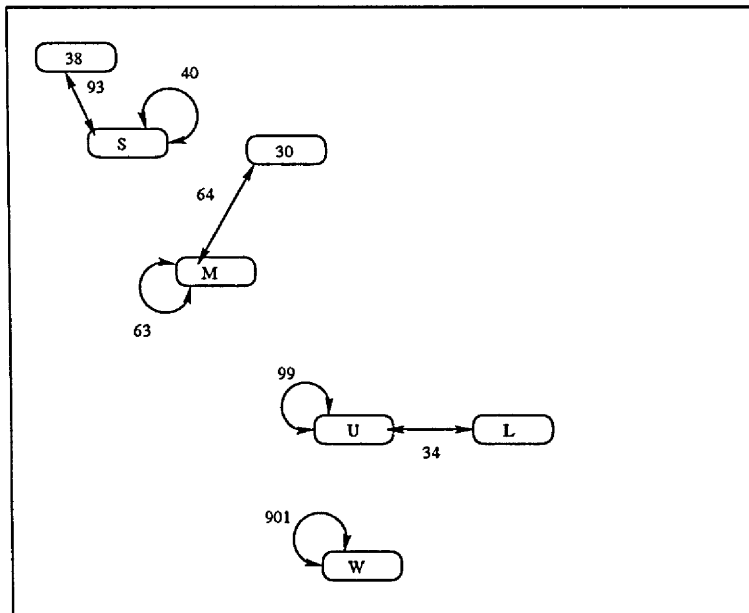


Figure 6.6: Release 1 Fault Architecture Diagram.

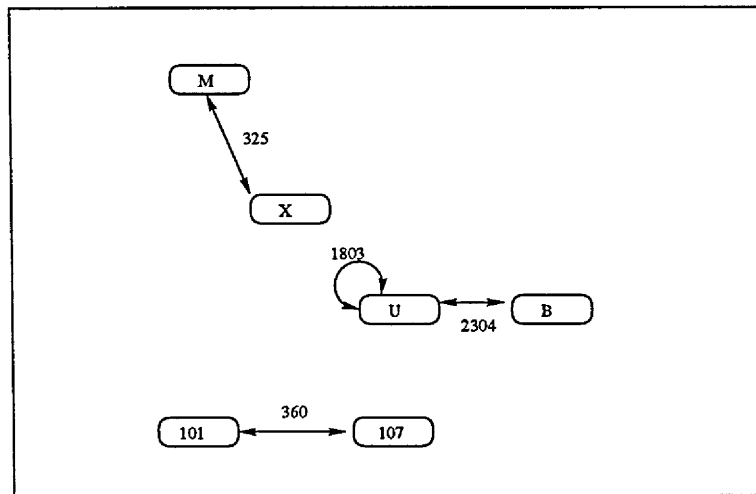


Figure 6.7: Release 2 Fault Architecture Diagram.

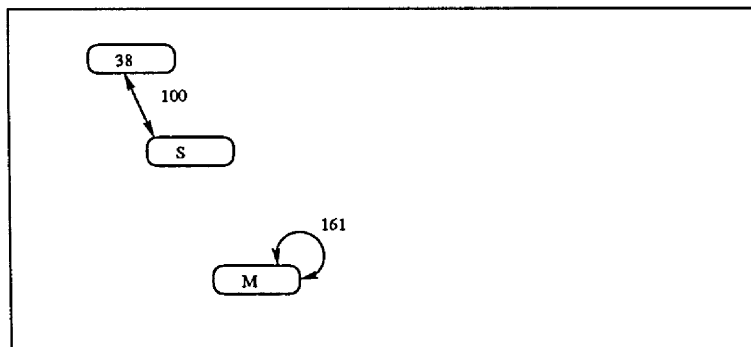


Figure 6.8: Release 3 Fault Architecture Diagram.

So far, we have only analyzed each release individually. When successive releases are to be tested, it might be useful to have a test guideline on how to treat components and relationships that have been fault-prone in prior releases. The purpose of a test guideline would be to focus attention on components in system test that had fault-prone relationships in the prior release. Figure 6.9 shows the *Cumulative Release Diagram* for the system in this study.

The subsystem level diagrams reveal that the most problematic relationships between subsystems were different for each release, save for one or two relationships. Only six fault-prone relationships exist between subsystems in the three releases. Three subsystems, 38, U and M are in fault-prone relationships with two other subsystems. In this system, the

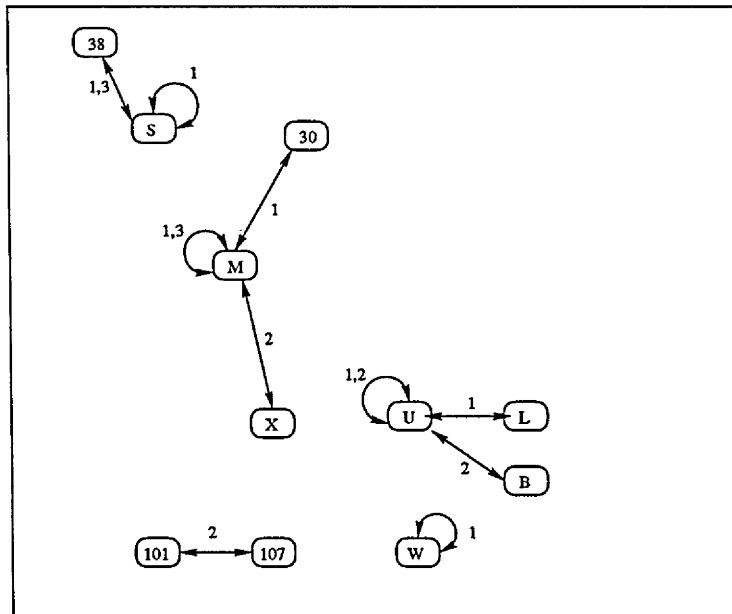


Figure 6.9: Cumulative Release Diagram.

most problematic parts of the software are internal to the subsystems, rather than between subsystems.

Results indicate that a test guideline that would focus attention in system test on components in a system that had fault-prone relationships in the prior release does not help in this case study. Figure 6.9 shows only one fault-prone relationship between subsystems occurring more than once. This is the fault relationship between subsystems S and 38. This relationship is fault-prone in Release 1 and Release 2. There are two reasons why such a guideline does not help in this study:

1. There are few fault-prone relationships.
2. Problems, once identified, are fixed.

## 6.4.2 Single Phase Analysis

### 6.4.2.1 Fault Architecture Diagrams for Development and Test

Clearly, the number of reported defects for development phases is much lower than the number of reported defects for the entire release. Row 1 in Table 6.5 shows the number of fault relationships that exist in development, system test, and post-release in Releases 1 – 3.

Row 2 shows the number of relationships that are fault-prone according to either defect coupling measure. Row 3 shows the number of components that are in fault-prone relationships according to either measure.

Table 6.5: Fault relationship information for all releases by development phase.

	Release 1			Release 2			Release 3		
	dev	test	post	dev	test	post	dev	test	post
# fault relationships	211	126	27	20	7	1	12	12	4
# fault-prone relationships	8	32	9	7	7	1	3	4	1
# comp. in fault-prone relns.	11	33	14	10	10	2	4	6	1

Figures 6.10 and 6.11 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for development and system test for Release 1. Numbers annotating the arcs are  $Re_{\langle C, C_i \rangle}$  measures. Numbers in parentheses are  $TR_C$  measures. Of the eleven components that are in fault-prone relationships in development, seven are also in fault-prone relationships in system test. These components include 25, 30, 35, 38, 96, 97, and 98. The components with which they are relationship fault-prone, however, are not necessarily the same. Only the two relationships between components 35 and 38 and components 25 and 30 are fault-prone in both development and system test. Components 25 and 35 are in fault-prone relationships in multiple releases as well.

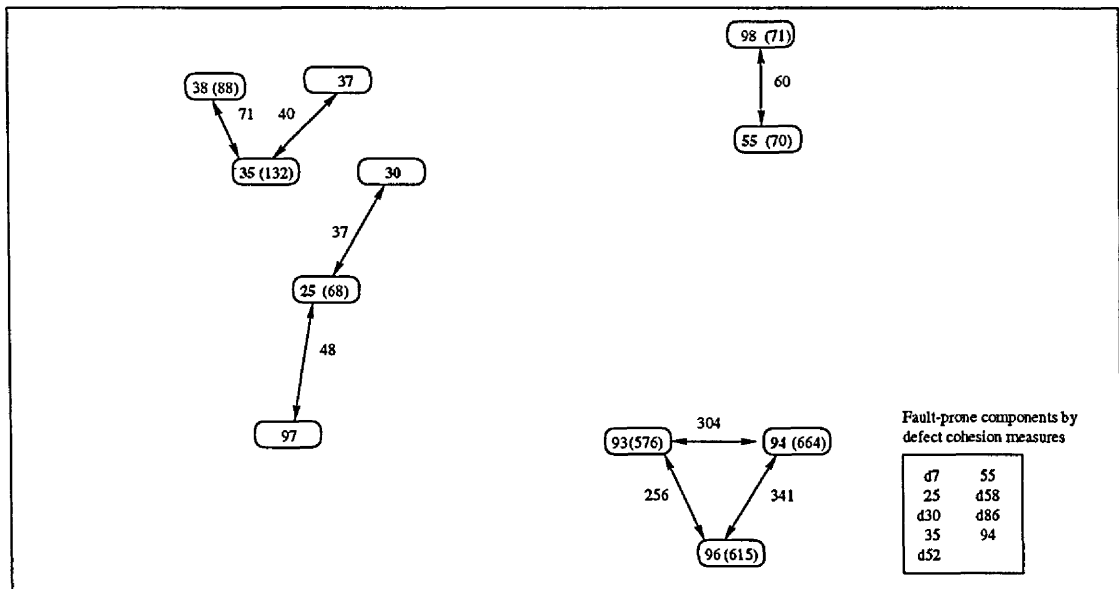


Figure 6.10: Fault Architecture Component Level Diagrams for development in Release 1.



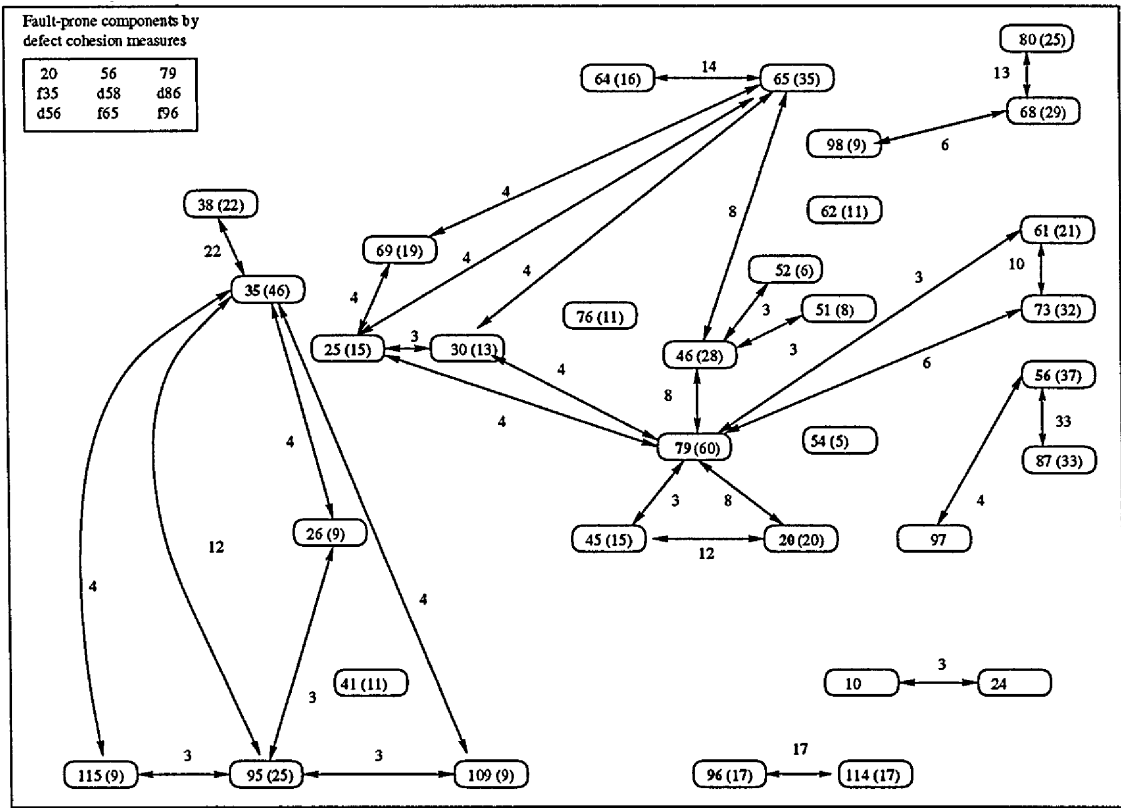


Figure 6.11: Fault Architecture Component Level Diagrams for system test in Release 1.

Figures 6.12 – 6.13 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for development and system test for Release 2. In Release 2,

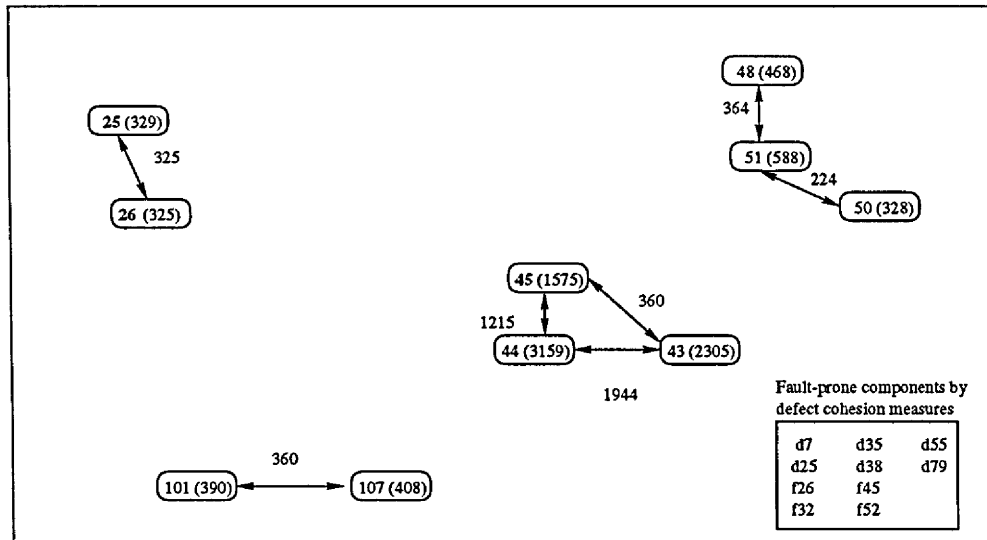


Figure 6.12: Fault Architecture Component Level Diagrams for development in Release 2.

none of the fault-prone relationships in development exist in system test. The components in fault-prone relationships are also different. A test guideline that recommends testing components in fault-prone relationships in development more intensely would not work well in Release 2.

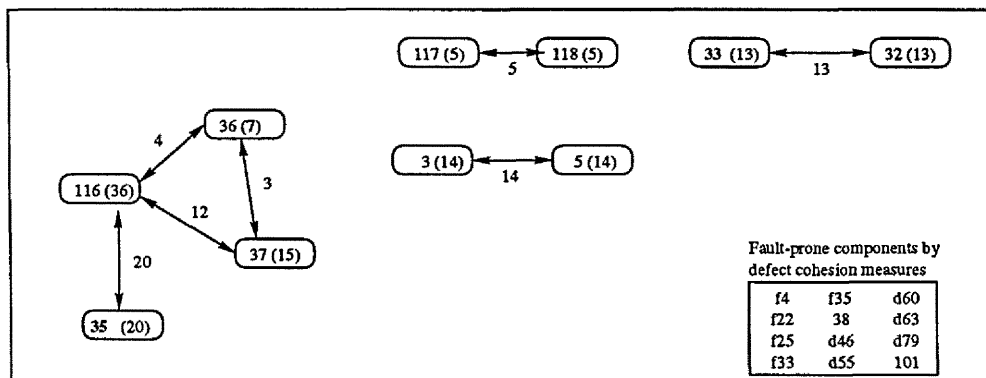


Figure 6.13: Fault Architecture Component Level Diagrams for system test in Release 2.

Figures 6.14 and 6.15 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for development and system test for Release 3. In Release 3, only one component that is in a fault-prone relationship in development is also in a fault-prone relationship in system test. This is again component 25. The components with which it has fault-prone relationships are different in development and system test. Because component 25 is in fault-prone relationships in five phases in the three releases with many other components, this component may require attention.

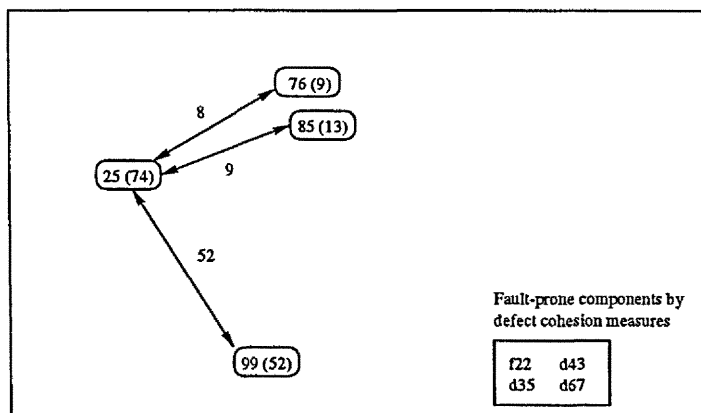


Figure 6.14: Fault Architecture Component Level Diagrams for development in Release 3.

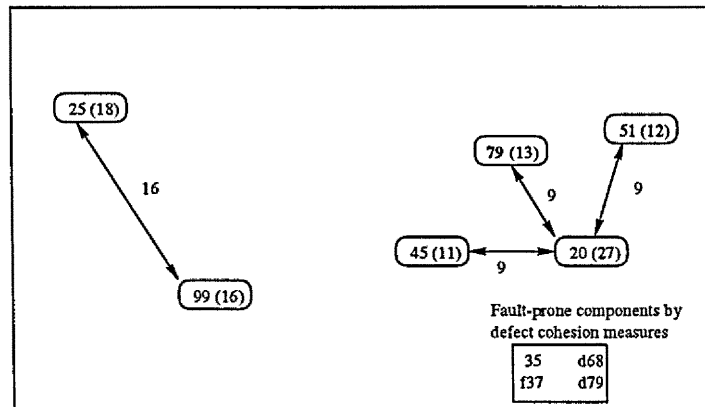


Figure 6.15: Fault Architecture Component Level Diagrams for system test in Release 3.

There are only four components that are repeatedly in fault-prone relationships throughout the development and system test phases of all three release. These are not new components and easily identifiable by the high number of defects all the way through the life cycle. These components include 25, 35, 38 and 99. Obviously, such brittle components are causes for concern. The analysis in this section leads to testing guideline 3 shown in Table 6.6.

Table 6.6: Testing guidelines derived from applying set of methods.

Testing Guidelines	
3.	Test components that are repeatedly fault-prone according to defect cohesion and defect coupling measures more thoroughly.

#### 6.4.2.2 Fault Architecture Diagrams for Post-Release

Figures 6.16 to 6.18 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for post-release for Releases 1, 2 and 3. The numbers annotating the arcs are the  $Re_{<C,C_i>}$  measure. The numbers in parentheses are the  $TR_C$  measure.

Of the nine fault-prone relationships in post-release in Release 1, five were also fault-prone in system test. Seven out of 11 components in fault-prone relationships were also fault-prone in development or system test. This indicates that system test was identifying and testing most of the components that have relationship problems in post-release. These problems, however, did not get fixed before release.

Release 2 had only one fault relationship in post-release. Most of the problems in Release 2 were internal to the components. Component 26 was, however, in a fault-prone relationship in development.

In Release 3, there were four fault relationships. Only one relationship was fault-prone. This relationship was between components 35 and 38, the same relationship that was fault-prone in system test in Release 1. It was not, however, fault-prone in development or system test in Release 3. This indicates that system test did not adequately test this relationship.

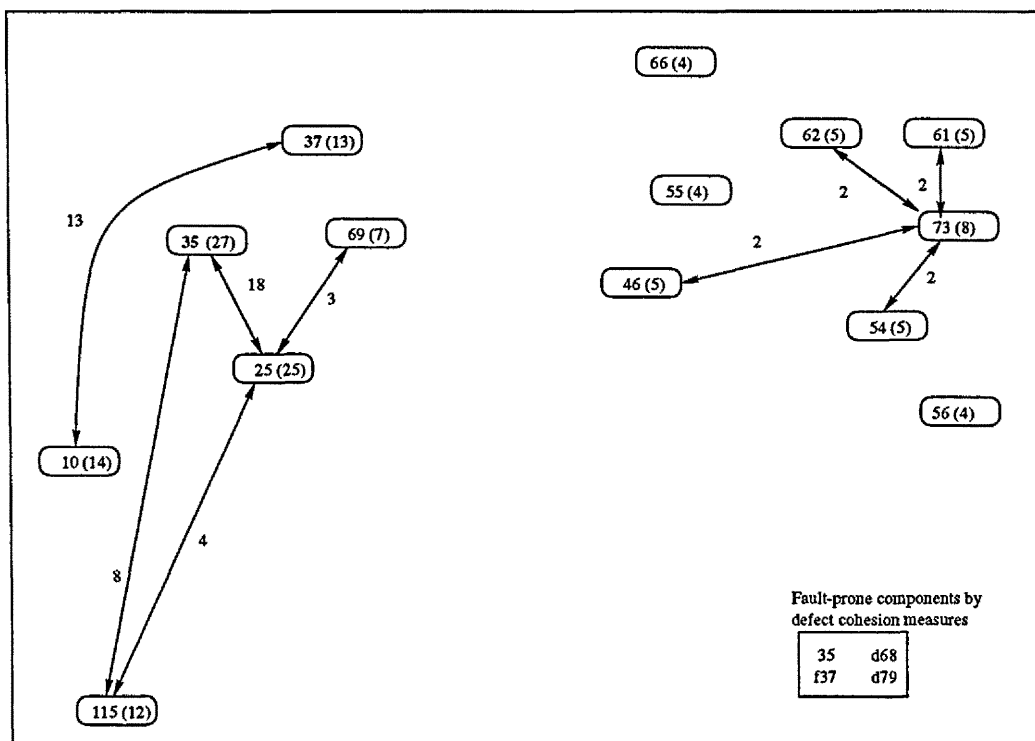


Figure 6.16: Fault Architecture Component Level Diagrams for post-release in Release 1.

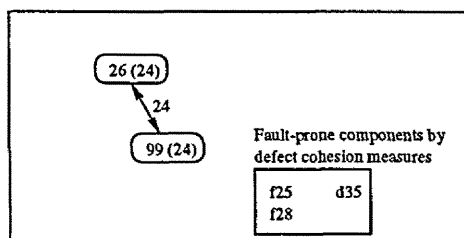


Figure 6.17: Fault Architecture Component Level Diagrams for post-release in Release 2.

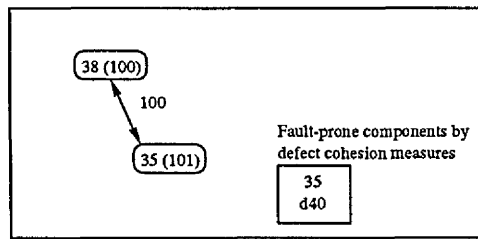


Figure 6.18: Fault Architecture Component Level Diagrams for post-release in Release 3.

The most problematic relationships tended to be different for the development and system test phases of each release. This indicates problems identified in development are usually fixed before system test. The most problematic relationships in post-release tended to be different than those in development and system test. This indicates some problems are getting through, but because the defect coupling measures are small in post-release, there are few problematic relationships.

Hard to identify before release are the components that are normal during system test, but are in fault-prone relationships after release. The analysis quite strongly points out that any component that was in a fault-prone relationship during test was **not** fault-prone after release. This is good news: the problems that were identified during testing were corrected before release. Development and system testing are doing a very good job eliminating problems. None of the components in fault-prone relationships are new, but they may have been affected by changes or enhancements. One possible cause for missing these components may be because regression test did not test them thoroughly enough. The defect data is not detailed enough to explain this phenomenon. A suggestion for improving testing would be to assess and improve impact analysis and regression testing.

Using these diagrams we performed a within-release analysis to determine whether additional test guidelines based on this data might be helpful. The following guidelines were helpful.

1. Test components that are fault-prone during development more thoroughly. They are likely to be fault-prone during system test.

2. Test components that are fault-prone during development as early in the drop as possible. It will give development more flexibility to fix defects and allow greater savings in test time when using statistical stopping rules.
3. Between releases, pay particular attention to components that had development and post-release problems, but where system testing found few problems. Likewise, evaluate the components that had problems in development, test, and after release in the prior release.
4. Improve impact analysis of enhancements on existing components and determine whether improvements in regression testing could have prevented relationship problems from slipping through.
5. Specifically test components 25, 35, and 38 more intensely and earlier. They are fault-prone repeatedly according to the defect cohesion measures and the defect coupling measures.

This being a case study, we do not expect these guidelines to improve every project, although they certainly are sensible. The project we studied has certain characteristics that need to be taken into account when determining whether applying these guidelines would improve system test performance:

- Few problems remain undetected. The number of post-release problems is very low. Using an order of magnitude threshold, less than ten percent of the components are fault-prone or are in fault-prone relationships in Release 1. In Releases 2 and 3, less than two percent of the components are fault-prone or are in fault-prone relationships.
- Most problems that are detected are fixed before release.
- In each release, the code is of very high quality. There are only nine fault-prone relationships between 26 subsystems in the three releases.

Overall, development and system test are doing a very good job at testing and repairing components. Within release analysis therefore will not help guide system test on prevention of post-release defects.

## 6.5 Summary

This paper replicated a study to evaluate the usefulness of using defect reports to derive fault architectures. Unlike the previous study [66], it uses both defect cohesion measures to identify fault-prone components and both defect coupling measures to identify components in fault relationships. It also different in that it set the threshold to an order of magnitude less than the largest measure, rather than to some percentage of components. Defect reports are easily available and can be used to identify parts of the software that are fault-prone. We applied several measures that identified the most fault-prone parts of the system in three releases. We also applied the measures to identify the most fault-prone relationships between components in the development, system test and post-release phases of each release. The methods for computing the defect cohesion measures and the defect coupling measures differed in how they treated defect reports. The basic defect cohesion measure is based simply on the number of defect reports for the component. The multi-file defect cohesion measure is based on defect fix reports and is sensitive to the number of files changed. The defect cohesion measures identified different components as fault-prone, hence we recommend using both.

In terms of the defect coupling measures, we investigated the multi-file defect coupling measure and the cumulative defect coupling measure. The multi-file defect coupling measure identifies components in fault-prone relationships, while the cumulative defect coupling measure identifies components that are in many fault relationships. In this study, the multi-file defect coupling measure did not identify many more problematic components than the cumulative defect coupling measure. The multi-file defect coupling measure does, however, identify pairwise coupling problems between components, while the cumulative defect coupling measure identified components with defect coupling problems with many other components.

Based on the measures, we created *Fault Architecture Component Level Diagrams* and *Fault Architecture Diagrams* for each release. We also created *Fault Architecture Component Level Diagrams* for the development, system test and post-release phases for each release, as well. The fault architecture technique visualized problems due to architecture fairly well.

It identified the most problematic component relationships in every release and for several phases of each release. Using these diagrams we were able to determine guidelines that should be helpful to software developers and testers.

The study in [40, 65, 66] differed from our study in two ways:

1. The technique was applied for a different purpose.
2. The quality of the system was different.

The purpose of the earlier study was to analyze code decay to recommend components for reengineering. In the study in [40, 65, 66], the system had more fault-prone components and more long term problems. Our study analyzed a very high quality software product, thus we were interested in identifying fault-prone components that require more thorough testing. The fault architecture method works in both cases. Because the system in the earlier study had more problems than the one analyzed in our study, the authors applied an additional step to reduce the number of components and fault relationships to a more manageable number prior to measuring defect coupling. This step would have eliminated too many components in our study. Future work should investigate whether using order of magnitude to set the threshold in systems of lower quality (such as the one in [40, 65, 66]) would make that step unnecessary.

Even the “almost perfect” project can benefit from the analysis and guidelines derived using fault architecture techniques. For high quality development environments like the one we are currently analyzing, the key issue for testers is where to put emphasis, and where not to. This technique enables developers and testers to determine the most problematic parts of the software. They can then focus their attention on these parts.



## Chapter 7

# Prioritizing Testing Activities

### 7.1 Evaluating the Effect of Testing Problem Components Early

Knowing which components are fault-prone is useful in two ways:

1. It identifies components that should be tested more thoroughly.
2. It can also be used to identify which components should be tested earlier. The assumption is that fault-prone components are more likely to have more failures during testing. Testing them early would shift these defects earlier in the testing cycle and thus give development more time to repair them.

Whether or not it is feasible to prioritize testing activities depends on several factors. Components may have to be tested in a specific order, if they depend on other components working correctly. In such a situation, prioritizing may not be possible. However, when the software development process contains a “qualification” phase, that is, a phase before system test that determines test readiness, it may be possible to arrange testing activities. In the qualification phase, the system test group runs a subset of the test cases they have developed. If during the qualification testing, the software is sent back to development, it software has major problems. This means that when system test starts, components do not have to be tested in a specific order. In addition, system test identifies and schedules features and major components for testing, as well as regression testing. Therefore, it is possible to arrange the schedule for testing specific components based on priorities. In

addition, if the software is released to system testing in stages or “drops,” tests can be ordered according to priorities within the same drop.

Prioritization strategies applied to the data include:

1. Test new components and components that were fault-prone in development earlier.
2. Test components that were in fault-prone relationships in development earlier.

These components are tested as early in a drop as possible. Earlier identification of defects gives development more time and flexibility for correction. Other components should be tested late in the test period for a given drop. Table 7.1 shows adds to the guidelines based on these two prioritization strategies.

Table 7.1: Testing guidelines derived from applying set of methods.

Testing Guidelines	
4.	Test new components and components fault-prone during development as early as possible.
5.	Test components that are repeatedly fault-prone according to defect cohesion and defect coupling measures earlier.

To evaluate the effect of these two guidelines, this study identified the weeks in system test that tested new components, components that were fault-prone during development, or components that were in fault-prone relationships during development. To simulate testing these components earlier, test results for those weeks were switched with those of an earlier test week (during the same drop) during which non-fault-prone components were tested.

The reordered curve should indicate earlier discovery of defects. It should be smoother because defect spikes caused by late testing of fault-prone components have disappeared. Beyond the benefit of earlier defect identification, applying a statistical stopping rule [12, 15, 16, 35, 47, 50, 51, 52, 53, 56] to the cumulative defect curve will allow for earlier stopping than in the original curve, reducing the elapsed time needed for testing. Therefore, the number of weeks that are expected to be saved by prioritizing testing should be a factor in parameterizing the stopping rule. Depending on the stopping rule, several weeks may be saved.

More work is required to develop an approach for applying stopping rules to reordered testing activities. Testing fault-prone components more thoroughly affects testing effectiveness. Efficiency may be improved by testing fault-prone components earlier. Questions concerning prioritizing testing activities that still need to be answered include:

- Will ordering testing activities by fault-proneness improve efficiency?
- How much will prioritizing testing activities help? Can this be measured, i.e., in “time available to repair”?

## 7.2 Results

### 7.2.1 Prioritization based on New and Fault-Prone Components

Release 1 has three drops in system test. Testing on the first drop started during week 54, on the second drop during week 63, and on third drop during week 70. (See Appendix A.3 for cumulative defect data.) Figure 7.1 shows cumulative defects per week for the original test process, and for the one that tests components that were fault-prone during development as early in a drop as possible. The reordered curve clearly indicates earlier discovery of defects. It also is smoother, especially at the end, because defect spikes caused by late testing of fault-prone components have disappeared. Beyond the benefit of earlier defect identification, applying a statistical stopping rule [39, 54] to the cumulative defect curve will allow for earlier stopping than in the original curve, reducing the elapsed time needed for testing. There were no defects in the last six weeks of testing and testing could have stopped earlier. Depending on the method used to make release decisions, this could save six to eight weeks for Release 1. The approximate cost per week of testing is about \$100K. Thus this testing guideline could easily have saved \$600-800K. This is a substantial amount of money.

As in Release 1, we evaluated the effect of the test guideline 4 by shifting testing of new and development fault-prone components to an earlier week in the same drop. Figure 7.2 shows the results.

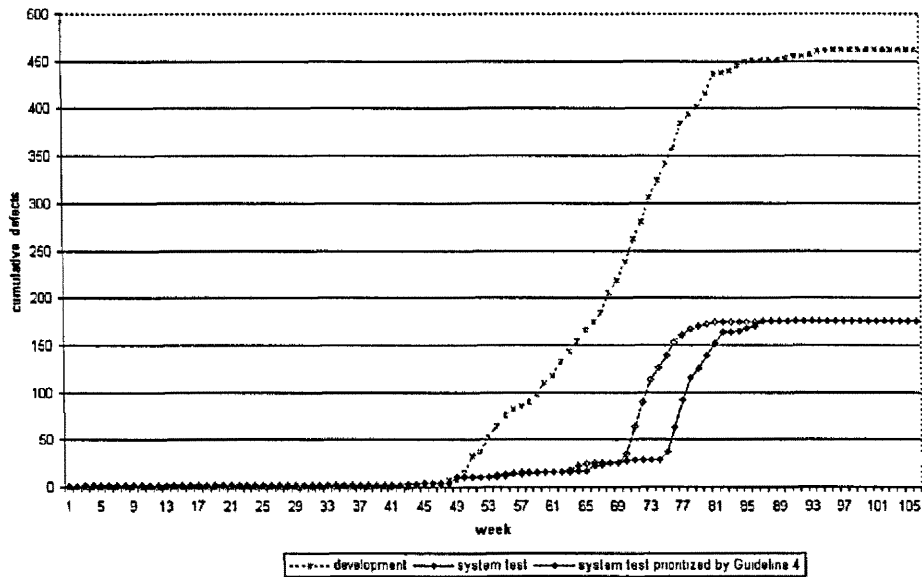


Figure 7.1: Release 1 cumulative defect curves (Guideline 4).

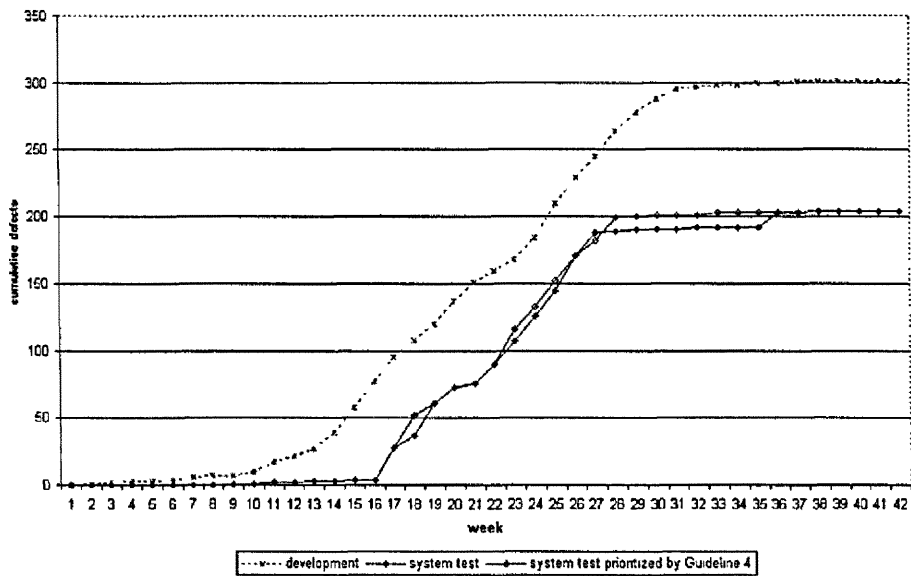


Figure 7.2: Release 2 cumulative defect curves (Guideline 4).

As in Release 1, defect intensity is higher earlier in testing. Most defects are found earlier, as well, giving developers more weeks to fix problems. As before, stopping rules are also likely to identify earlier end points for testing, saving effort and calendar time for the testers. However, there were fewer defects in Release 2 overall, and thus the benefits of this rule were not as spectacular as in Release 1.

Figure 7.3 shows the results of shifting testing of development fault-prone components to an earlier week in the release. (Release 3 had only one drop.)

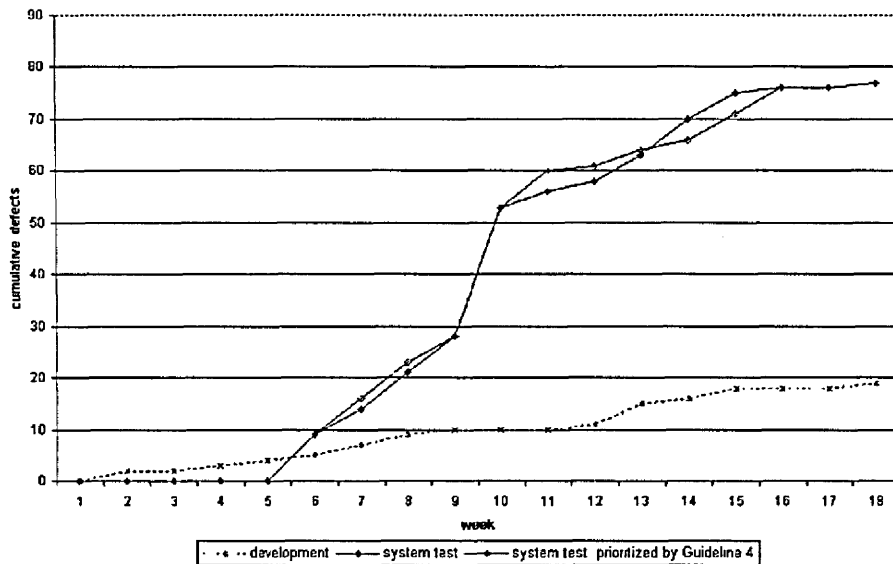


Figure 7.3: Release 3 cumulative defect curves (Guideline 4).

In Release 1, defect intensity is slightly higher earlier in testing. There were fewer defects in Release 3 overall, and thus the benefits of this rule were limited. Several defects were found earlier, which would give developers more time to fix them. System test found one defect in week 16, none in week 17, and one in week 18. The guideline to test fault-prone components earlier would not have uncovered these two defects. Therefore, reordering would not have saved any time in system test.

## 7.2.2 Prioritization based on Fault-Prone Relationships

The second prioritization strategy results in cumulative defect curves that are similar to those in Section 7.2.1 for Release 1 and Release 2. In Release 1, there is no difference in the effects of prioritization based on test guideline 5 in the first two drops. In the third drop, only one week is affected differently by the two prioritization strategies. One week with a lower defect yield is delayed eight weeks in the second strategy. This results in a slight improvement over the first strategy.

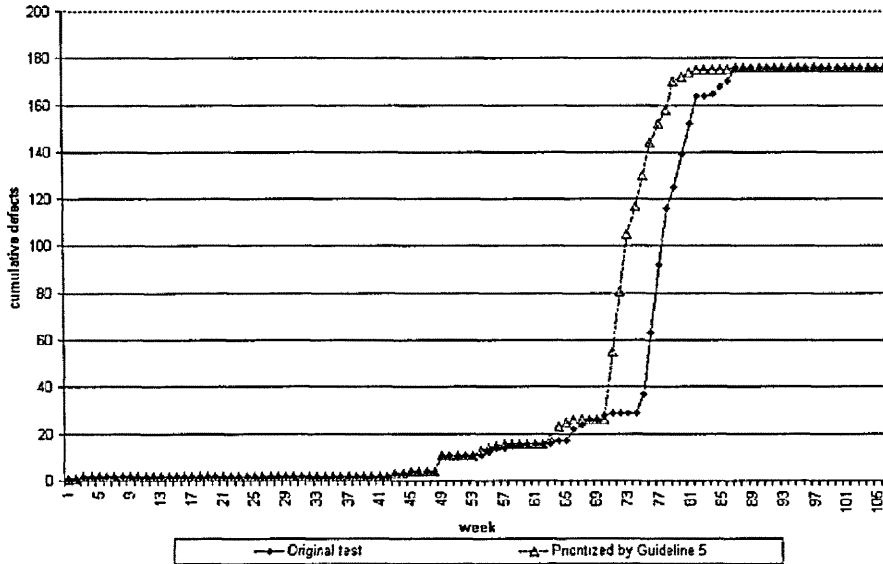


Figure 7.4: Release 1 cumulative defect curves (Guideline 5).

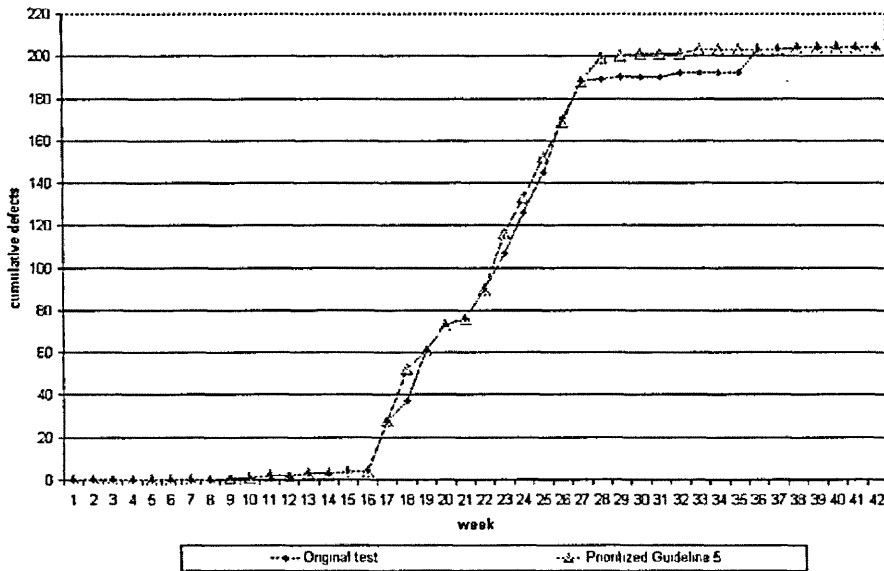


Figure 7.5: Release 2 cumulative defect curves (Guideline 5).

In Release 2, the effect of prioritization based on test guideline 5 is similar to Release 1. There again was no difference between the two strategies in the first drop. In the second drop, only one week was affected. One week is moved up two weeks in the second strategy. The effect worsens efficiency for one week, but in the following week, it improves. In the

next week, the cumulative number of defects using both strategies is the same and remains the same for the rest of system test.

The effect of prioritization based on test guideline 5 is slightly worse in Release 3. It is not only worse than the effects of prioritization based on test guideline 4, it is worse than using no prioritization strategy. Figure 7.6 shows the cumulative defects curves for Release 3.

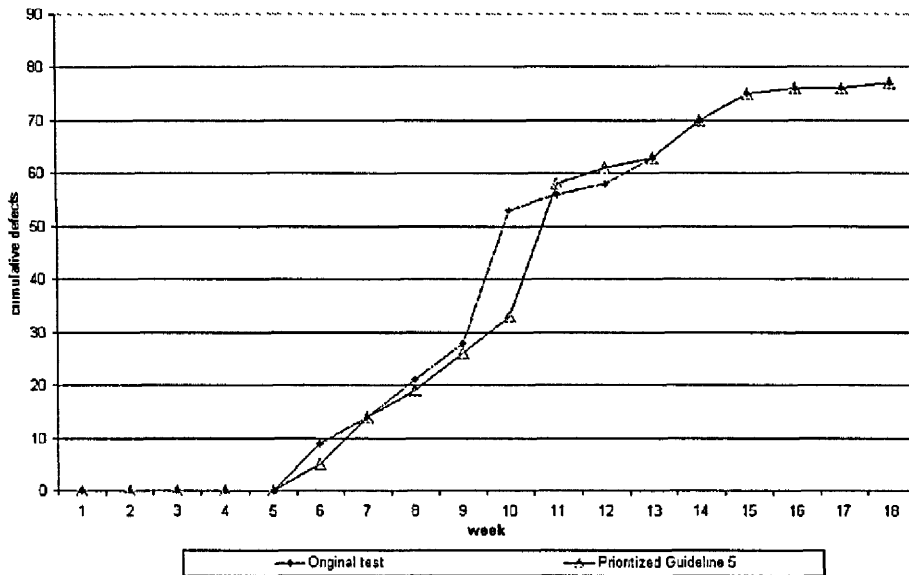


Figure 7.6: Release 3 cumulative defect curves (Guideline 5).

### 7.3 Summary

The priorities clearly indicate earlier discovery of defects for the first two releases. The curves are smoother, especially at the end, because defect spikes caused by late testing of fault-prone components have disappeared. The results in the first release were better than the second and third releases, because there were fewer defects in later releases. Beyond the benefit of earlier defect identification, applying a method to make release decisions will allow for earlier stopping when testing activities are prioritized than when they are not, reducing the elapsed time needed for testing. A guideline to test new components, components that are fault-prone in development, or components that are in fault-prone relationships

in development, as early as possible has the potential of saving weeks of testing time and hundreds of thousands of dollars without penalty.



## Chapter 8

# Analysis of Static Defect Estimation

### 8.1 Approach

#### 8.1.1 Estimating Components with Defects in Post-Release and Not in Test

During maintenance and evolution of systems, testers and developers alike worry about negative effects of enhancement and maintenance activities. Of particular concern are old components that appear fine during development and test of a release, but have problems after the software is released. If testers were able to estimate how many components are likely to exhibit such problems, this knowledge could be used to decide whether to release the system or test it further. The test manager could set a threshold of how many such components are acceptable. If the estimate falls below, software is ready to be released. If the estimate is higher, more (thorough) testing is advisable.

The approach described here applies methods based on capture-recapture models and curve-fitting methods in a novel way to estimate the total number of components that have defects in system test and post-release. From this estimate, one can derive the expected number of components that will have defects after release that were defect-free in system test. The approach uses defect data from groups at different test sites. These methods are used differently here than they have been in prior studies, in which they were primarily used to estimate the number of defects remaining after inspection by several reviewers.

Data is often available from several test sites, such as a system test group at the developing organization, an internal customer, an external customer, or an independent quality assurance organization. If this is the case, it is possible to use capture-recapture and curve fitting models to estimate the number of components that will have problems after release, but were defect-free during testing. Each test site reports the components for which defects were found.

The steps in the approach are:

1. Collect defect data for components from the different test sites at the end of each week. For each test site, a component is given a value of 0, if the test site has never reported a defect for it. Otherwise the component is given the value of 1.
2. Apply capture-recapture and curve-fitting estimation methods to the data. The estimates give the sum of
  - the number of components with defects found in testing plus
  - the number of components that are expected to have defects in operation even though they were defect-free in testing.

Several estimators used in capture-recapture models ( $m0ml$ ,  $mtml$ ,  $mhjk$ ,  $mthChao$  and  $mtChpm$ , see Table 2.4) and curve-fitting methods ( $dpm(exp)$ ,  $dpm(linear)$ , cumulative) are applied to the data. Because the  $mtChpm$  estimator is used in the case of two reviewers [19], it is also evaluated.

3. Apply our proposed experience-based estimation method to the data. This method is based on simple multiplication factors and is applied to releases for which historical data is available.

The experience-based method uses a multiplicative factor that is calculated by using data from previous releases and applied to the current release. This factor is used to estimate the number of components that have defects in post-release that were defect-free in testing.

This estimate refers to the number of “missed” components, those components for which system test should have found defects. This estimate can be compared with the ones obtained using the capture-recapture and curve-fitting methods.

The following formula is used to compute the number of “missed” components.

$$[mf_i * t_i] \tag{8.1}$$

where

$i$  is the current release.

$mf_i = \sum_{k=1}^{i-1} p_k / \sum_{k=1}^{i-1} t_k$  is the multiplicative factor for release  $i$ .

$p_k$  is the number of components with defects after release that were defect-free in test in release  $k$ .

$t_k$  is the number of components that were defect-free in testing in release  $k$ .

4. Calculate the number of components that are expected to have defects in the remainder of system test and in post-release that are currently defect-free in testing by subtracting the known value of the number of components with defects found in testing so far.
5. Compare the estimated number of components that are not defect-free but for which no defects have been reported in system test to a predetermined decision value. Use the information as an input to decide whether or not to stop testing and release the software.

In a study in [19], estimates have been used for review decisions. By contrast, the estimates in this study are used for release decisions. Estimates of the number of components with post-release defects that do not have defects in system test may be used as one of several criteria when deciding whether to stop test and release software. In this context, it is more important that the correct decision is made than that the estimate is completely accurate.

The estimate must be checked against some threshold to make a decision. For example, if defects were found in 50 components and no defects were found in 60 components

during system testing and estimation methods predict that 10 components out of the 60 would have defects after release, would testing continue or stop? Threshold values reflect quality expectations. These can be common for all projects or specific to a particular type of project, such as corrective maintenance, enhancement, safety-critical, etc.

While the availability of defect data from several test groups or test sites makes it possible, in principle, to apply existing capture-recapture and curve-fitting models, it is by no means clear whether the estimators work well in practice. Thus we present a case study to answer the following questions:

- Which estimators work best?
- How robust are these estimators when only two independent test sites are available?
- Should the focus be on using only one estimate?
- At what point in the testing process can/should these estimates be applied?

To answer these questions, the estimators are evaluated using two types of measures:

- Estimation error (including relative error <sup>1</sup> and mean absolute relative error <sup>2</sup>).
- Decision error (is the decision to release or continue testing made correctly).

The decisions based on the expected number of components with defects in post-release, but not in system test, are evaluated and compared to the correct decisions using three different threshold values. The number of correct decisions is the number of times the decisions based on the estimates predict the correct decisions.

The estimators are ranked and analyzed to see whether they have similar behavior for the data sets. In general, the Mh model using the mhjk estimator has worked best

---

<sup>1</sup>Relative error is defined as (estimate - actual total)/actual total

<sup>2</sup>Mean absolute relative error is defined as the absolute value of the mean relative errors for all three releases

for applications published within the software engineering field so far [6]. That does not, however, mean that it is necessarily the best for this specific application, since the models are used in a new context.

A related question is at what point in the test cycle one should and could apply these estimates for decision making purposes? We attempt to answer this question by using the estimators up to five weeks before the scheduled end of testing. As before, we evaluate the quality of the estimator and the quality of the decision made.

### 8.1.2 Estimating Still Defective Components (Components with Defects in Test and Post-Release)

Capture-recapture and curve-fitting methods do not estimate the components that are found to have defects in system test and are still defective. An assumption for the models is that defects are corrected and do not show up in post-release. Components that have defects in system test cannot be assumed to be defect-free in post-release, as other defects may be reported. Therefore, other ways must be used to estimate the number of components that still contain defects as the capture-recapture and curve-fitting models cannot cope with them.

An experience-based estimation method based on a simple multiplication factor is presented below. It is used to estimate the number of components that have defects in system test and post-release. It is applied to releases for which historical data is available.

The simple estimation-based method is based on using a multiplicative factor that is calculated by using data from previous releases and applied to the current release. This factor is used to determine the number of “still defective” components, those components that had defects in system test and still had defects after system test.

The following formula is used to compute the number of “still defective” components.

$$[sf_i * dt_i] \tag{8.2}$$

where

$i$  is the current release.

$sf_i = \sum_{k=1}^{i-1} b_k / \sum_{k=1}^{i-1} dt_k$  is the multiplicative factor for release  $i$ .

$b_k$  is the number of components with defects in both test and post-release in release  $k$ .

$dt_k$  is the number of components that had defects in testing in release  $k$ .

## 8.2 Case Study Data

Interviews with testers ascertained that assumptions for the capture-recapture models are met: Three system test sites receive the same system for testing at the same time. The three sites and their main responsibilities are:

1. The system test group at the developing organization tests the system against design documents.
2. The internal customer tests the system against the specifications.
3. The external customer tests the system with respect to their knowledge of operational use.

These different views may have the effect of reducing the overlap of non-defect-free components. Perspective-based reading [59] shows that the capture-recapture models are robust with respect to different views. This is important, because different test sites focus on different testing goals for the software.

Interviews with the testers uncovered other factors that affect the estimates. First, the internal customer test site may have under-reported defects. The primary responsibility of this test site is writing specifications. In addition to writing specifications, they test against the specifications. Because of this, the number of defective components is estimated with and without this site. This way we also evaluate how well the methods perform when data exists for only two test sites.

Second, some scrubbing of the data occurred. Before reporting a defect, the personnel at the test sites are encouraged to check a centralized database to avoid reporting duplicate defects. Some defects reported are later classified as duplicates, but the partial prescreening has the effect of reducing overlap and hence increasing estimates. This prescreening has a greater impact on components with few defects. For example, if a component has only one

defect and a test site finds and reports the defect, no other test site will report any other defects (and are not likely to report the same defect). The component will be categorized as defect-free for those other test sites, reducing overlap. One way to compensate for this problem is to look at estimators that tend to underestimate. If overlap is reduced due to prescreening, estimates will be higher. Estimators that tend to underestimate will compensate for defect scrubbing.

Table 8.1 shows the actual values for components with defects and components without defects for test and post-release. Actual values are shown for all three releases using data from three test sites, and where different for two test sites, they are shown in parentheses. The values in Column 6 (the actual number of components with defects in test and post-release) are used to evaluate the estimates obtained in the study. The values in Column 5 (the actual number of components that have defects in post-release that did not have defects in system test) are used to evaluate the release decisions.

Table 8.1: Release data for three test sites (two sites are in parentheses).

	# all comp.	# comp. defect-free in test	# defective comp. in test	# defective comp. in post release not in test	total # defective comp.
Release 1	180	128	52 (51)	7	59 (58)
Release 2	185	125	60 (59)	5	65 (64)
Release 3	188	154	34	6	40

Columns 3 and 5 of Table 8.1 also show the data from Release 1 and Release 2 used to compute the multiplicative factors and the estimates for Release 2 and Release 3, respectively. Table 8.2 shows the computation for the multiplicative factors using data from Release 1 and applied to Release 2, and data from both Releases 1 and 2, applied to Release 3. Since data from more than one previous release is available, the cumulative data over the previous releases is used. It is possible, however, to use only one previous release that is similar to the current release.

Table 8.3 shows the data used in earlier test weeks to make release decisions for Releases 1, 2, and 3. Release decisions are based on the actual number of defective components remaining to be found after the test week indicated through post-release shown in

Table 8.2: Multiplicative factors for all releases.

	Multiplicative Factor	
	3 Sites	2 sites
Release 1	–	–
Release 2	7/128	7/129
Release 3	$(7+5)/(128+125)$	$(7+5)/(129+126)$

Columns 3, 5 and 7. The values in parenthesis are the values for two test sites. (The number of defects found for most of the weeks using only two test sites is one less than it was for three test sites in Release 1. In Releases 2 and 3, the actual number of defects found for all of the weeks using only two test sites is the same as it was for three test sites.)

Table 8.3: Actual values used to determine correct release decisions.

	Release 1		Release 2		Release 3	
	defective components in test	defective components remaining	defective components in test	defective components remaining	defective components in test	defective components remaining
5 weeks earlier	48 (47)	11 (12)	56	9	29	11
4 weeks earlier	48 (47)	11 (12)	56	9	31	9
3 weeks earlier	49 (48)	10 (11)	56	9	33	7
2 weeks earlier	49 (48)	10 (11)	57	8	33	7
1 week earlier	49 (48)	10 (11)	60 (59)	5	33	7
End date	52 (51)	7	60 (59)	5	34	6

In Release 1, four to five weeks earlier than the actual end of system test, the number of defective components remaining is 11 (or 12 using two test sites). Because this number is larger than all three thresholds, the correct answer at all three thresholds is to continue testing. One to three weeks before the end of test, the correct answer at thresholds 2 and 5 is to continue testing and at the threshold of 10, the correct answer is to stop testing. The correct answer at the actual end of system testing indicates that testing should continue if the threshold is 2 or 5 and stop if the threshold is 10. One would expect that as more weeks of testing occurs, one would see more decisions to stop. The correct release decisions are the same for three test sites and two test sites. These decisions are used to evaluate the decisions made based on the estimates.

Table 8.3 shows there were a few weeks that had no change in the number of components found to have defects. For example, in Release 3, there was no change one to two weeks prior



to the end of system test. All components were available for test during this time. They did not come in late, nor were any new components integrated during this time. Defects were found in these weeks, but they were found in components that already had reported defects.

Appendix A.5.1 presents the estimates obtained using the static estimation methods for three and two test sites. The errors and relative errors for the estimators are shown in tables in Appendix A.5.2.

## 8.3 Results

### 8.3.1 Estimation of Defective Components in Post-Release and Not in Test

#### 8.3.1.1 Evaluation of Estimates using Three Test Sites

Figure 8.1 shows the evaluation of the estimates for all three releases for three test sites. (Negative values are a result of underestimation.) The results are encouraging. The relative errors show that most estimators provide estimates close to the actual values, except for mtChao.

$\chi^2$  analysis showed only one estimator, the mthChao, was significantly different from the actual release values and the other estimators' values at a level of  $p = 0.001$ . The mthChao, therefore is not recommended for use. The statistical null hypothesis (the estimator's values are the same as the actual values) is not rejected for all other estimators at  $p = 0.001$ . While  $\chi^2$  analysis also showed that the other estimators were not significantly different from each other, in practice several estimators perform better (and had lower  $\chi^2$  values). (See Table A.23 in the Appendix for the  $chi^2$  values).

In Release 1, the actual value was between the estimates given by the m0ml, mtml, dpm (linear) and mhjk methods. The m0ml, mtml, and dpm (linear) estimators normally have a tendency to underestimate. The partial scrubbing of defect data for duplicates may have reduced the overlap of the test groups leading to higher estimates for some of the estimators. Estimates that usually tend to underestimate (m0ml, mtml) worked well for this situation: They did not underestimate quite so much. Both dpm estimators basically plot data and

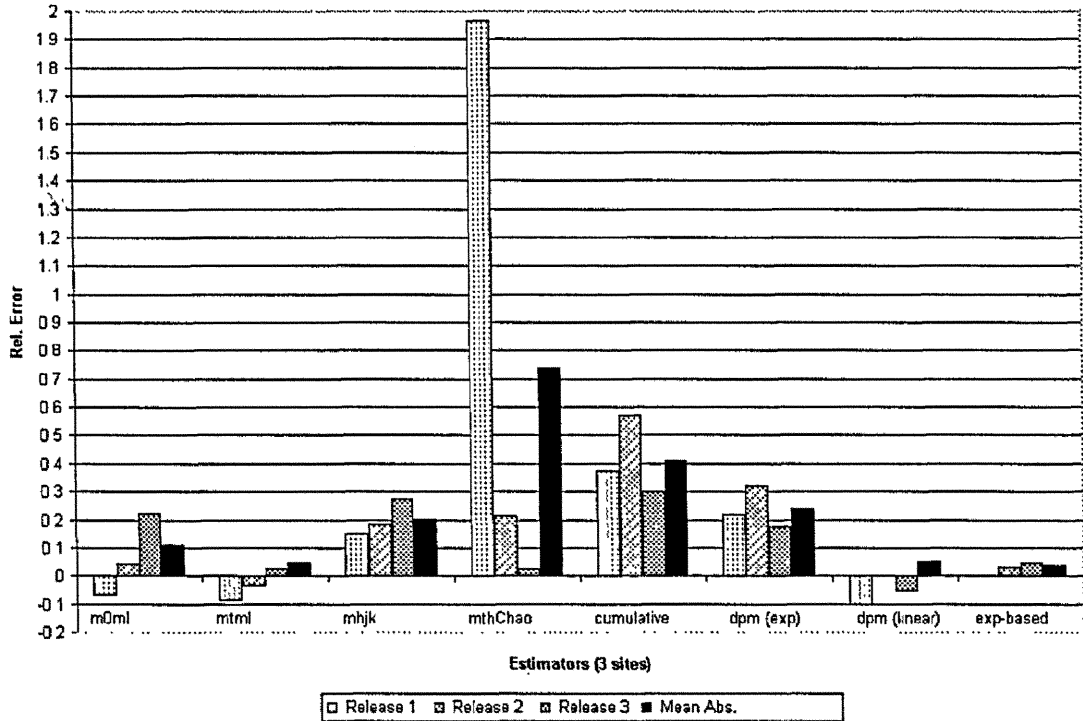


Figure 8.1: Relative errors and mean absolute relative error for all releases (three sites).

do not have the same assumptions normally involved in capture-recapture models, hence they are less sensitive to varying defect detection probabilities or test site abilities. The mhjk has shown promising results in software engineering before [6] and is what we would have guessed to be the best estimate. It overestimated here, probably because of the partial scrubbing of data. Thus, it is good that the actual value turned out to be between the estimates provided by these four estimators as it gives us lower and upper limits.

Figure 8.1 shows that the second and third releases have similar results. In Release 2 and Release 3, the same four capture-recapture and curve-fitting estimators (dpm(linear), mtml, m0ml and mhjk) perform the best. Actual values were close to their estimates, frequently occurring between the estimates provided by the mtml or dpm(linear) and mhjk methods. The mtml and dpm(linear) estimation methods slightly underestimated and the mhjk slightly overestimated. The cumulative, dpm(exp) estimators still overestimate greatly.

The mthChao, the cumulative, and the dpm(exp) estimators did not perform well in most of the releases. They tended to overestimate greatly. The mthChao greatly

overestimated in the first, but performed well in Release 3. Its inconsistency, however, makes it difficult to trust.

The experienced-based method performed very well in the second and third releases. It is interesting to compare these results with the results for the capture-recapture and curve-fitting estimation methods. The capture-recapture estimates were close to the experience-based estimates and are quite good.

Table 8.4 shows the results of evaluating the estimators using the mean of the absolute relative error to rank the estimators over all three releases. The experienced-based method performs the best overall. The mthChao and cumulative do not perform well. The capture-recapture and curve-fitting estimators that perform the best are the mtml, dpm(linear), m0ml, and mhjk. These estimators have relative errors close to the experience-based method and show that they are as good as a method that requires history.

Table 8.4: Ranking of estimators over three releases (three sites).

Estimator	Ranking
m0ml	4
mtml	2
mhjk	5
mthChao	8
cumulative	7
dpm (exp curvefit)	6
dpm (linear curvefit)	3
experience-based	1

These results indicate that capture-recapture and curve-fitting methods, in particular the mtml, dpm(linear), m0ml, and mhjk estimators, are able to estimate quite well the total number of components that have defects in test and post-release. The expected number of remaining components with defects that were defect-free in testing can then be computed from these estimates.

### 8.3.1.2 Evaluation of Estimates using Two Test Sites

Figure 8.2 shows the evaluation of the estimation methods applied to data from two test sites for all three releases. The two test sites include the system test group at the developing

organization and the external customer, which is probably the more common situation. Naturally, one would expect capture-recapture and curve-fitting methods to produce less accurate predictions as the number of reviewers (here, test sites) shrinks. However, the results are quite reasonable. Several of the estimators have low relative errors.

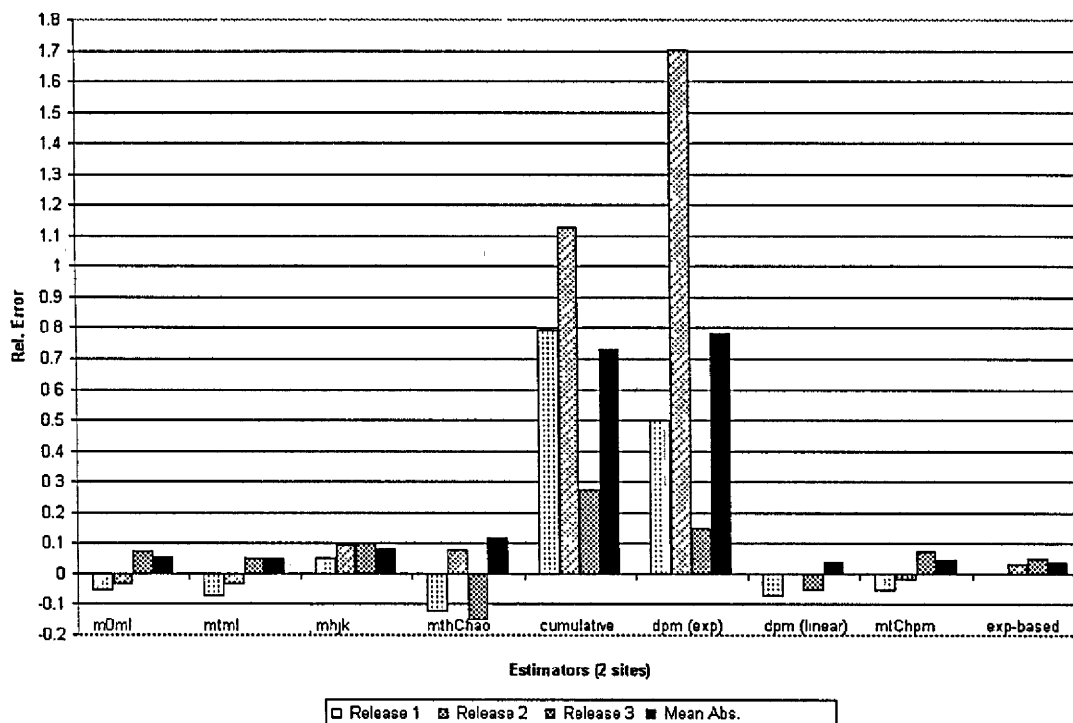


Figure 8.2: Relative errors and mean absolute relative error for all releases (2 sites).

$\chi^2$  analysis shows that none of the estimators were significantly different from the actual values or from each other at  $p \leq 0.20$ . (See Table A.24 in the Appendix for the  $\chi^2$  values). Using data from only two test sites, the mtml, m0ml, dpm(linear), mtChpm, and the mhjk estimators performed the best in terms of their errors. The m0ml, mtml, and mtml estimators again tended to underestimate slightly. The mhjk slightly overestimated. The MtChpm estimator, which may be used only in the case of two test sites, performed very well and is comparable to the m0ml and mhjk. The mthChao, cumulative and dpm (exp) did not perform well using two test sites. The cumulative and dpm(exp) overestimated too much. The mthChao, which usually tends to overestimate, underestimated in Releases 1 and 3. In fact, it gave the lower bound for estimates, the number of components that are

known to have defects in testing. As such, the mthChao method provided an estimate that is suspect. This effect may be due to the fact that data from only two test sites was used. In any case, the estimates it provided were so low, that its relative error ranks it as one of the worst performing estimators. The mthChao estimator is not useful in this situation.

As in the case of three test sites, the mean of the absolute relative error across the three releases is used to evaluate and rank the estimators using two test sites. Table 8.5 shows the ranking of the estimation methods applied to data from two test sites for all three releases. The experience-based method and the dpm (linear) rank the highest overall. The mthChao, cumulative, and dpm (exp) rank lowest. The m0ml, mtml, mhjk and mtChpm have mean absolute relative errors that are less than 0.100. They are almost as good as the experience-based method and do not require any history.

Table 8.5: Ranking of estimators over three releases (three sites).

Estimator	Ranking
m0ml	5
mtml	3
mhjk	6
mthChao	9
cumulative	8
dpm (exp curvefit)	7
dpm (linear curvefit)	1
mtChpm	4
experience-based	2

The experience-based method depends on releases being similar. If historical data is available and releases are similar, the experience-based method using the multiplicative factors should be used. The capture-recapture and curve-fitting methods are independent of data from previous releases. Several of the estimates from the capture-recapture and curve-fitting methods have relative errors that are almost as low. These include the mtml, dpm(linear) and the mjhk. If no historical data is available or releases are dissimilar, these capture-recapture and curve-fitting methods provide reasonable estimates.

The results for the estimation methods based on capture-recapture models do not worsen for two test sites, in most cases they improve. This is due to the fact that the third test

site had less overlap with the first two test sites than the first two test sites had with each other. The cumulative and  $dpm(exp)$  estimators, which are curve-fitting methods, worsen (as expected) with one less test site. Most unexpectedly, the  $dpm(linear)$  estimator, which is also a curve-fitting method, performs very well and improves with only two test sites. The curve-fitting methods are most useful when there are a number of reviewers and several reviewers find most of the defects[6].

These results demonstrate that capture-recapture and curve-fitting methods are able to estimate remaining defective components well when only two test sites are involved in system testing. Since this is probably a more common situation, this is good to know.

### 8.3.1.3 Evaluation of Estimates Obtained Earlier in System Test

Estimates obtained in earlier weeks and the decisions based on those estimates were evaluated for Releases 1, 2, and 3 using three test sites and two test sites. Given that some of the estimators performed rather badly so far, not all estimators were applied in earlier test weeks. The  $m0ml$ ,  $mtml$ ,  $mhjk$ , and  $dpm(linear)$ , the  $mtChpm$  estimators and the experienced-based method are the only ones applied and evaluated. The estimates from the  $mtChao$ ,  $dpm(exp)$  and cumulative methods were not considered. They recommended testing to continue too long.

Over the last few weeks of system test, the estimates changed. In all releases, estimates tended to approach the actual number of defective components in test and post-release as the estimators were applied in later weeks. As more testing occurred, estimation errors decreased. For example, in Release 1, all the estimators underestimated until the last week of system test. In earlier weeks, all the estimators, except for  $mhjk$ , gave estimates that were below the actual number of defective components found. The  $mhjk$  estimates were slightly higher than the others. In Release 3, the  $mtml$  started underestimating and then by the end of system test overestimated slightly.

We also applied the experience-based method to Release 2 and Release 3 in a similar approach using data available at earlier points in time. The estimates for defective components relies on the multiplicative factor based on data from earlier releases, as well as

the number of components that have been found to have defects at various points in time. Because the previous release determines the multiplicative factor used in the current release, the factor's value does not change when applying it to earlier weeks of data in the release for which estimates are derived.

Applying the experienced-based method at earlier weeks in system test provided estimates that are quite good. Estimates were close to the estimates obtained by other estimation methods applied at earlier weeks. For example, in Release 2, the experience-based estimates fell between the lower estimates provided by the m0ml, mtml, and mtChpm methods (which under-estimated), and the higher estimates provided by the mhjk and the dpm (linear) methods (which over-estimated). The experience-based tends to underestimate at earlier weeks, then slightly overestimates closer to the actual end of system test.

Results indicated that capture-recapture methods hold up well in earlier weeks for the case of two test sites. In Release 1 and Release 2, the m0ml and mtml estimates for two test sites were approximately the same as for three test sites. The mhjk and the dpm (linear) performed better with two test sites. The mhjk overestimated less using two test sites and the dpm (linear) underestimated less. Using data from two test sites then did not worsen the performance of the estimators and in some cases improved them.

In Release 2, the m0ml and mtml estimators performed worse in the case of two test sites than in the case of three, but still had relative errors less than 0.10. The mtChpm estimator was only slightly better than the m0ml and mtml estimators. The dpm (linear) overestimated in earlier weeks, but had smaller errors than when it underestimated using data from three test sites.

The estimators in Release 3 performed no worse, and in the case of the mhjk performed better using data from two test sites rather than three. The mtml and dpm(linear) estimators provide estimates that are almost the same as those for three test sites. The mhjk overestimated less with two test sites and the estimates were closer to the actual values.

Overall, the mhjk estimator, which overestimated in the case of three test sites, still overestimated, but not quite as much. The m0ml and mtml performed about the same in both cases. The mtChpm performed as well as the m0ml and mtml estimators. The

curve fitting method, dpm (linear), was not consistent: In Release 1 it performed better, in Release 2 it performed worse, and in Release 3 it performed about the same.

The decision to stop or continue testing at the end of a particular week, however, is based on the actual number of defective components found in system test at that point in time and it is the decision made, rather than the estimate, that is of most concern.

### 8.3.2 Release Decisions based on Estimates

Interviews conducted with the developers indicated that it is acceptable to have two components with no reported defects in system test, but with post-release defects. More than ten such components is unacceptable. Specifically, defects in new components that add functionality are more acceptable than defects in old components. It is most important that old functionality has not been broken. Most of the new components are found to have defects in system test. Only old components had post-release defects, but no defects in system test. These are exactly the components the developers worry about.

Given that some of the estimators performed rather badly, not all estimators are used. The  $m0ml$ ,  $mtml$ ,  $mhjk$ , and  $dpm(\text{linear})$  estimators are used for the case with three test sites. These estimators, as well as the  $mtChpm$  estimator, are evaluated for the case with two test sites. The estimates from the  $mtChao$ ,  $dpm(\text{exp})$  and cumulative methods are not considered as their relative errors are too high and would cause testing to continue too long.

The release decisions based on the estimates are compared to the correct decision based on the actual values and evaluated.

#### 8.3.2.1 Release Decisions using Three Test Sites

Tables 8.6 – 8.8 show the decisions based on the estimates and the number of correct decisions for the four estimators  $m0ml$ ,  $mtml$ ,  $mhjk$  and  $dpm(\text{linear})$  using three test sites for Releases 1, 2 and 3. The correct decisions are shown in bold.



Table 8.6: Release 1 decisions earlier in test (three sites).

5 weeks earlier	Threshold			# correct decisions
	2	5	10	
m0ml	stop	stop	stop	0
mtml	stop	stop	stop	0
mhjk	continue	continue	continue	3
dpm (linear)	continue	stop	stop	1
4 weeks				
m0ml	stop	stop	stop	0
mtml	stop	stop	stop	0
mhjk	continue	continue	stop	2
dpm (linear)	continue	stop	stop	1
2-3 weeks				
m0ml	stop	stop	stop	1
mtml	stop	stop	stop	1
mhjk	continue	continue	stop	3
dpm (linear)	continue	stop	stop	2
1 week				
m0ml	stop	stop	stop	1
mtml	stop	stop	stop	1
mhjk	continue	continue	stop	3
dpm (linear)	continue	stop	stop	2
last week				
m0ml	continue	stop	stop	2
mtml	stop	stop	stop	1
mhjk	continue	continue	continue	2
dpm(linear)	stop	stop	stop	1

### 8.3.2.2 Evaluation of Decisions in Last Week (Three Sites)

To illustrate the quality of the estimators in making decisions, thresholds of 2, 5 and 10 were chosen and evaluated. The estimate of the number of components with defects in post-release but not in test is compared against these three thresholds to determine whether the decision made, using the estimate, is correct. If, for example, the threshold value is 2, then testing would stop if the estimated number of components with defects in post-release that were defect-free in testing was less than or equal to 2. The correct answer is 7 in Release 1. So the correct *decision* would be to continue testing. If, for example, the threshold value is 10, the correct decision would be to stop testing, since 7 is less than 10.

Table 8.7: Release 2 decisions earlier in test (three sites).

3-5 weeks earlier	Threshold			# correct decisions
	2	5	10	
m0ml	continue	continue	stop	3
mtml	stop	stop	stop	1
mhjk	continue	continue	continue	2
dpm (linear)	continue	stop	stop	2
experience-based	continue	continue	stop	3
2 weeks				
m0ml	continue	continue	stop	3
mtml	stop	stop	stop	1
mhjk	continue	continue	continue	2
dpm (linear)	continue	stop	stop	2
experience-based	continue	continue	stop	3
1 week				
m0ml	continue	continue	stop	2
mtml	continue	stop	stop	3
mhjk	continue	continue	continue	1
dpm (linear)	continue	stop	stop	3
experience-based	continue	continue	stop	2
last week				
m0ml	continue	continue	stop	2
mtml	continue	stop	stop	3
mhjk	continue	continue	continue	1
dpm(linear)	continue	stop	stop	3
experience-based	continue	continue	stop	2

Tables 8.6, 8.7 and 8.8 show the results of the decision analysis in the last week of system test for all three releases using data from three test sites. In Release 1, the m0ml and mhjk estimators provide the correct decision most often. The m0ml recommends stopping a bit too soon. The mhjk recommends continuing testing a little too long. Testing a bit too long, in most cases, is probably preferable to stopping too soon. In Release 2, the mtml and dpm(linear) perform the best, both providing three correct decisions. The decisions based on m0ml and mhjk would result in continuing testing a bit longer than the other estimators. In Release 3, mtml is the only estimator that leads to three correct decisions. The decisions based on the dpm(linear) estimator result in a recommendation to stop testing too early.

Table 8.8: Release 3 decisions earlier in test (three sites).

5 weeks earlier	Threshold			# correct decisions
	2	5	10	
m0ml	continue	continue	continue	3
mtml	continue	continue	stop	2
mhjk	continue	continue	continue	3
dpm (linear)	continue	stop	stop	1
experience-based	continue	continue	continue	3
4 weeks				
m0ml	continue	continue	continue	2
mtml	continue	stop	stop	2
mhjk	continue	continue	continue	2
dpm (linear)	continue	stop	stop	2
experience-based	continue	continue	continue	2
1-3 weeks				
m0ml	continue	continue	continue	2
mtml	continue	continue	stop	3
mhjk	continue	continue	continue	2
dpm (linear)	continue	stop	stop	2
experience-based	continue	continue	continue	2
last week				
m0ml	continue	continue	continue	2
mtml	continue	continue	stop	3
mhjk	continue	continue	continue	2
dpm(linear)	continue	stop	stop	2
experience-based	continue	continue	stop	3

The decisions based on m0ml and mhjk result in recommendations to continue testing longer than the others.

Not only do these methods provide reasonable estimates of the number of components that will have post-release defects, but no defects in system test, the estimates give a good basis for a correct release decision for the three threshold values analyzed. The mtml estimator makes the largest number of correct decisions for all three threshold values both in Release 2 and Release 3. In Release 1, it recommends stopping too soon. The mhjk estimator consistently recommends to continue testing, because it typically overestimates. In all releases, the mjhk estimator would cause testing to continue until a threshold value of about 15-16.

It is probably preferable to continue testing too long rather than stopping testing too soon and releasing the software. Because of this, the preferred estimator would be one that provides a slight overestimate. Analysis of the estimates provided by the methods shows that the mhjk estimator tends to slightly overestimate in this situation. (Other estimators that overestimate, do so too much.) An estimator that slightly overestimates will best support making the correct decision. This analysis is supported by the opinion of one tester who believed that too many defects were found after Release 1 and Release 2 and that testing should have continued a bit longer.

Estimations provided by the experience-based method, using multiplicative factors, are also analyzed from a decision point of view. Because estimations using such a method can only be made for releases with historical data, decisions for stopping test based on estimations can only be made for Release 2 and Release 3. Compared to decisions based on estimation from the capture-recapture and curve-fitting methods, the experience-based method works quite well. In Release 2, the experience-based method gives us decisions on a par with m0ml. Since testing a bit longer is preferable to stopping testing too soon, this method provides us with a conservative, but not too conservative decision. In Release 3, the experience-based method gives us decisions on a par with mtml. A more conservative method like the mhjk would recommend testing a little longer than mtml and the experience-based method.

### **8.3.2.3 Evaluation of Earlier Decisions (Three Sites)**

Table 8.6 shows that the m0ml, mtml, and dpm (linear) estimators recommend stopping testing as early as five weeks before the end of system test in Release 1. These decisions do not agree with the correct answer. The mhjk estimator makes the correct decisions at the threshold values for 2 and 5 for the last six weeks for system test. At the threshold value of 10, it makes the correct decision at every week, but two. These two weeks are the fourth week before the actual end of system test and the last week of system test. In these cases, the mhjk estimator recommends that testing continue when the correct answer is to stop. The mhjk estimator is, therefore, a little conservative.

For an example of how this method works, consider the following scenario. Assume that the threshold is ten components and the mhjk estimator is used to make decisions to continue or stop testing in Release 1. Five weeks before the actual end of system test, the recommendation based on the mhjk estimator is to continue testing. This is the correct decision at this time. The following week (4 weeks earlier), the recommendation is to stop testing. The decision is incorrect: It should recommend testing to continue. It would be better if testing continues until a stop decision consistently occurs a certain number of weeks in a row. If we assume that testing continues until three stop decisions occur in a row, testing would correctly continue, because only one stop decision has occurred at this point. The following week (three weeks earlier), the recommended decision is to stop testing. This recommendation occurs again at two weeks before the actual end of system test. Testing would now have had three weeks in a row in which a stop was recommended. If testing stopped now, they would be making a correct decision. This decision results in saving two weeks of testing. There is a potential that some defects would be missed, but it would be within the threshold set at 10.

Tables 8.7 – 8.8 show that most of the estimators, except for mhjk, improve in the second and third releases. The mhjk estimator recommends testing continue for all weeks at all thresholds. The mhjk decisions agree with the correct answer in the earlier weeks, but are perhaps conservative in the last two weeks. The m0ml and experienced-based method perform better than the mhjk estimator in making decisions in Release 2.

Mtml and the dpm (linear) recommend stopping testing too soon for at least two of the thresholds in Release 2. In Release 3, the dpm (linear) estimator incorrectly recommends stopping six weeks before the end of system test at thresholds 5 and 10. Mtml recommends stopping two to five weeks before the actual end of system test and then in the last two weeks recommends testing continue at the threshold of 2. (A threshold of 2 is very sensitive to changes in estimates.) In Release 3, the mtml estimator also incorrectly recommends stopping too soon in the fourth week week before the actual end of system test at the threshold level 5 and then in the last three weeks recommends testing continue, reversing its decision.

Tables 8.6 – 8.8 show that when m0ml, mtml and dpm (linear) provide incorrect decisions, they typically indicate that testing should stop when the correct answer is to continue. Whenever the mhjk estimator is incorrect, it most often says to continue when the correct answer is to stop.

Table 8.9 ranks the estimators based on the number of correct decisions over the last six weeks (the last week and the previous five weeks) for three test sites. The columns indicate the estimators' ranks for all three releases and an overall rank. Table 8.9 shows that the experience-based method performed very well in Release 2 and Release 3, in which historical data was available. The overall rankings show the m0ml and mhjk estimators perform the best in making correct decisions. When a conservative decision is desired, the mhjk estimator should be used, otherwise the m0ml should be appropriate.

Table 8.9: Ranks for estimators for three test sites.

Estimator	Release 1 Rank	Release 2 Rank	Release 3 Rank	Overall Rank
m0ml	3	1	3	1
mtml	3	4	1	2
mhjk	1	4	3	2
dpm(linear)	2	3	5	4
experience-based	-	1	2	-

The m0ml and mhjk estimators and the experienced-based estimation method perform very well when used to make decisions. The m0ml and experience-based estimation methods tend to recommend stopping sooner than the mhjk estimation method. If system testers want to save testing time, the m0ml and experience-based methods should perform well in providing information to aid in making the decision to continue or stop test. The mhjk estimation method is recommended in situations in which system testers want to be more conservative - that is they would prefer to continue testing longer in order to have fewer defects reported in post-release.

### 8.3.2.4 Release Decisions using Two Test Sites

Tables 8.10, 8.11, and 8.12 show the results of the decision analysis for all three releases using data from only two test sites. The Chapman estimator is included in the decision analysis as it had low relative errors. The correct decisions are shown in bold.

Table 8.10: Release 1 decisions earlier in test (two sites).

5 weeks earlier	Threshold			# correct decisions
	2	5	10	
m0ml	stop	stop	stop	0
mtml	stop	stop	stop	0
mhjk	<b>continue</b>	stop	stop	1
mtChpm	stop	stop	stop	0
dpm (linear)	<b>continue</b>	stop	stop	1
4 weeks				
m0ml	stop	stop	stop	0
mtml	stop	stop	stop	0
mhjk	<b>continue</b>	stop	stop	1
mtChpm	stop	stop	stop	0
dpm (linear)	<b>continue</b>	stop	stop	1
1-3 weeks				
m0ml	stop	stop	stop	0
mtml	stop	stop	stop	0
mhjk	<b>continue</b>	stop	stop	1
mtChpm	stop	stop	stop	0
dpm (linear)	<b>continue</b>	<b>continue</b>	stop	2
last week				
m0ml	<b>continue</b>	stop	<b>stop</b>	2
mtml	stop	stop	<b>stop</b>	1
mhjk	<b>continue</b>	<b>continue</b>	<b>stop</b>	3
dpm(linear)	<b>continue</b>	<b>continue</b>	continue	2
mtChpm	<b>continue</b>	stop	<b>stop</b>	2

### 8.3.2.5 Evaluation of Decisions in Last Week (Two Sites)

Table 8.10- 8.12 show that based on data from two test sites the m0ml, dpm(linear), and mtChpm estimator lead to the correct decision for two threshold values in Release 1. The m0ml, mtml, dpm(linear) and mtChpm perform well in Release 2; all three lead to three correct decisions. All the estimators, except for dpm (linear), perform equally well at

Table 8.11: Release 2 decisions earlier in test (two sites).

3-5 weeks earlier	Threshold			# correct decisions
	2	5	10	
m0ml	stop	stop	stop	1
mtml	stop	stop	stop	1
mhjk	continue	continue	stop	3
mtChpm	stop	stop	stop	1
dpm (linear)	continue	continue	continue	2
2 weeks				
m0ml	stop	stop	stop	1
mtml	stop	stop	stop	1
mhjk	continue	continue	stop	3
mtChpm	stop	stop	stop	1
dpm (linear)	continue	continue	continue	2
1 week				
m0ml	continue	stop	stop	3
mtml	continue	stop	stop	3
mhjk	continue	continue	stop	2
mtChpm	continue	stop	stop	3
dpm (linear)	continue	continue	continue	1
last week				
m0ml	continue	stop	stop	3
mtml	continue	stop	stop	3
mhjk	continue	continue	continue	1
dpm(linear)	continue	stop	stop	3
mtChpm	continue	stop	stop	3

all thresholds earlier in system test in Release 3. The mhjk estimator leads to three correct decisions in Releases 1 and 3, but in Release 2, it results in an incorrect recommendation to continue testing. Again, it is probably more desirable to continue testing too long, than it is not to test long enough. The m0ml, mtChpm and mjhk estimators appear to be the type of estimators to best support the correct decision for two test sites.

Comparing decisions based on estimations using two test sites for Release 2, the experience-based method performs better than the capture-recapture and curve-fitting methods. It recommends testing a little longer than the correct answer, but not as long as mjhk, which was determined to be the preferred estimator in Release 2. In Release 3, the



Table 8.12: Release 3 decisions earlier in test (two sites).

5 weeks earlier	Threshold			# correct decisions
	2	5	10	
m0ml	continue	continue	stop	2
mtml	continue	continue	stop	2
mhjk	continue	continue	stop	2
mtChpm	continue	continue	stop	2
dpm (linear)	continue	stop	stop	1
4 weeks				
m0ml	continue	continue	stop	3
mtml	continue	continue	stop	3
mhjk	continue	continue	stop	3
mtChpm	continue	continue	stop	3
dpm (linear)	continue	stop	stop	2
1-3 weeks				
m0ml	continue	continue	stop	3
mtml	continue	continue	stop	3
mhjk	continue	continue	stop	3
mtChpm	continue	continue	stop	3
dpm (linear)	continue	stop	stop	2
last week				
m0ml	continue	continue	stop	3
mtml	continue	continue	stop	3
mhjk	continue	continue	stop	3
dpm(linear)	continue	stop	stop	2
mtChpm	continue	continue	stop	3

experience-based method performs as well as several of the other estimators, including the mhjk.

Overall, the experience-based method performs very well when used to make decisions. Decision making based on estimates using some of the capture-recapture and curve-fitting estimation methods results in decisions that are as good. If historical data is available and releases are similar with regards to defects and their exposure, the experience-based estimation method should be used to complement capture-recapture and curve-fitting estimation methods and provide more input into making the decision to stop or continue testing. If no historical data is available or releases are not similar, capture-recapture and curve-fitting methods may be used to make good decisions based on the estimates they provide.

### 8.3.2.6 Evaluation of Earlier Decisions (Two Sites)

The same kind of analysis was performed using data from two test sites to evaluate the estimators' release earlier in test. Tables 8.10 – 8.12 show the decisions based on the estimates and the number of correct decisions for the five estimators m0ml, mtml, mhjk, mtChpm and dpm (linear) for two test sites for all releases.

Tables 8.10 and 8.11 show the decisions based on estimates earlier in system test are not very good in the first and second releases. Most of the estimators incorrectly recommend stopping at the three threshold values. Mhjk and dpm (linear) provide the greatest number of correct decisions 2–5 weeks before the actual end of system test. They recommend testing continue when other estimators incorrectly recommend testing should stop. It is only in the actual last week of testing, that the number of correct decisions improve for the other estimators. Table 8.12 shows that all the estimators, except for dpm (linear), perform equally well in Release 3 at all thresholds earlier in system test. Most of the estimators correctly recommend testing continue at the thresholds of 2 and 5 for all weeks. The same estimators make the correct recommendation at a threshold of 10, recommending testing at this threshold stop about five weeks earlier.

Table 8.13 ranks the estimators based on the number of correct decisions over the last six weeks for two test sites. The columns indicate the estimators' ranks for each release and an overall rank. The mhjk, the m0ml and the mtChpm perform the best overall. The mhjk, however, is consistently ranked first or second for all three releases using data from two test sites. Based on this analysis, the mhjk is recommended for use in the case of two test sites to make decisions in continuing or stopping testing.

Table 8.13: Ranks for estimators for two test sites.

Estimator	Release 1 Rank	Release 2 Rank	Release 3 Rank	Overall Rank
m0ml	3	3	1	2
mtml	5	3	1	5
mhjk	2	1	1	1
mtChpm	3	3	1	2
dpm(linear)	1	2	5	4

The m0ml and the mhjk estimators ranked the highest in making release decisions, whether three test sites or two test sites were used. The mhjk estimator tends to be more conservative, recommending testing continue, when the correct answer is to stop. If a conservative decision is desired, one should use the mhjk estimator. Otherwise the m0ml would be appropriate.

In this case study, the best results come from using two test sites rather than three test sites. (Because the internal customer test site may have under-reported defects, it may have affected the results for three test sites.) For the environment in this study, two test sites are recommended. Since the methods do perform well when data exists for only two test sites, it is feasible to use our approach with two sites, which may be a more common situation. If there are three test sites and all three test sites have good defect data, then three test sites may be recommended.

### 8.3.3 Estimation of Still Defective Components

Section 8.1.2 described the approach for a simple estimation-based method to estimate the number of “still defective” components, those components that had defects in system test and still had defects after system test.

The multiplicative “still defective” factor was computed using data from Release 1 and applied to Release 2. A multiplicative factor was computed using data from both Release 1 and 2 and applied to Release 3. Since data from more than one previous release is available, the cumulative data over the previous releases may be used. It is possible, however, to use only one previous release that is similar to the current release.

Table 8.14 shows the data from Release 1 used to estimate for Release 2, as well as the calculations for the estimates for Release 2 and the data from Releases 1 and 2 used to estimate for Release 3.

For the third release, cumulative values are taken from the previous releases and used to derive the multiplicative factors. The sum of the number of components that had defects in both test and post-release in Releases 1 and 2 is used for the “still defective” factor. Still defective components are overestimated by over 50 percent .

Table 8.14: Release data and estimates for Releases 2 and 3.

	Previous Release			Current Release		
	# comps w/ defects in test	# with defects in test & post-release	Fraction still defective	# comps w/ defects in test	Estimated # comps still defective	Correct # comps still defective
Release 2	52	28	28/52	60	$\lceil 28/52 * 60 \rceil = 33$	16
Release 3	60	16	16/60	34	$\lceil \frac{28+16}{52+60} * 34 \rceil = 14$	6

Using data only from only Release 2 to estimate for Release 3 improves the estimates slightly. The still defective components are estimated as 10. In terms of new functionality, Release 3 had fewer new components than Release 2 and Release 2 had fewer new components than Release 1. In this respect, Release 3 is more similar to Release 2 than it is to Release 1.

This method does not estimate still defective components very well, but on the other hand capture-recapture and curve-fitting methods can not estimate for this type of component at all.

## 8.4 Summary

This study evaluated the use of several existing methods to estimate total and remaining defects in code in a new context. Several test groups or sites concurrently test a software product. Developers want to know whether their software is ready for release, and how many components will exhibit defects after release that had not shown defects during system test. This gives an indication of how many components were “missed” (tested inadequately) during system test.

Results show that capture-recapture and curve-fitting methods may be used to estimate the number of components that have defects after release, but no defects in testing. The estimates from several capture-recapture and curve-fitting methods have low relative error and compare favorably with experience-based estimates used as a point of reference. Errors from the best estimators based on capture-recapture and curve-fitting have relative errors between 0.05 and 0.2 compared to the experience-based method that has absolute relative errors between 0.03 to 0.05. The experience-based method, however, depends greatly

on previous releases being similar. The capture-recapture and curve-fitting methods are independent of data from previous releases.

Capture-recapture and curve-fitting estimators that perform the best in this study include the `m0ml`, `mtml`, `dpm(linear)` and `mjhk`. The `m0ml`, `mtml` and `dpm(linear)` estimators usually have a tendency to underestimate slightly. Due to the partial prescreening of the data to reduce duplicate reporting, these estimators provide estimations that appear to compensate for defect scrubbing. The `mjhk` tends to overestimate slightly. The `mthChao`, `dpm(exp)` and cumulative methods overestimate too much. Because testing too long is preferable to not testing long enough, the `mjhk` estimator will probably perform the best, especially if there is no prescreening. Estimators for two different capture-recapture models provide a “window”. The `mtml` or the `dpm(linear)` estimators are good choices for the lower bound of the range and the `mjhk` estimator is a good choice for the upper bound. These estimates from capture-recapture and curve-fitting estimation methods may be complemented with estimates provided by the experience-based method and the testers’ personal experiences.

Estimates can be used as an input to making decisions on whether to stop test and release software. Results show that estimates from capture-recapture and curve-fitting methods using several threshold values provide a good basis for a correct decision for stopping. The `mjhk` estimator appears to perform the best as a basis for decision making. It tends to recommend testing a bit longer and is more conservative than the `m0ml`, `mtml`, and `dpm(linear)` methods. The `mthChao`, `dpm(exp)` and cumulative estimators do not perform well in making decisions.

Results show that the same estimators that performed well using data from three test sites also performed well when using data from two test sites. In some cases, the estimators performed better. The estimators are also shown to be quite robust for two test sites, even when the sites test the system differently.

This approach is a new application of estimation methods based on capture-recapture and curve-fitting models. These methods may be used to estimate the number of components that have defects in post-release that did not have defects in testing. These are the

components that were missed in testing. These estimations in turn may be used to recommend decisions on whether to continue testing or to stop testing and release software. The recommendation may then be used as one of several criteria to make a decision. If historical data is available and releases are similar, a simple experience-based method using multiplicative factors should be used as a complement to the capture-recapture and curve-fitting estimation methods. If, however, no historical data is available or releases are dissimilar, capture-recapture and curve-fitting methods work quite well to provide estimates and to help make decisions for continuing or stopping testing.

## Chapter 9

# Analysis of SRGM Selection Method

### 9.1 Approach

There are three main risks involved in applying the software reliability growth models to estimate failures. They are:

- Not any one model will match a company's development and test process exactly. This will result in a curve fit that may not converge or may have a low correlation value.
- Data is grouped by week resulting in a reduction in the number of data points. This means it will take longer for the predicted total number of failures to stabilize.
- Testing effort may vary week to week. This will be more of a problem with a small amount of data, that is when there are few test weeks.

To handle these risks, multiple failure estimation methods and models may be applied.

The selection method applies software reliability growth models to cumulative failure data grouped by weeks to determine how well the method predicts the expected total number of failures. More specifically, we consider

1. Can one use calendar time (in our case weeks) when the original models assume execution time?

2. How robust are the models used in this case study when assumptions are not met?  
The models are robust when failures occur one at time. In this study, failures are grouped in weeks.
3. How does one decide in the testing process which model to use, when a plethora of models exist? Not one model may fit all the data.
4. Can we put a process framework around the models to make the models useful?
5. Can the selection method aid in choosing the model and making decisions to stop testing?

Cumulative number of failures by week are fitted with software reliability growth models. If the correlation is good, the function can predict the number of failures that will occur after release. The predicted number of total failures is a model parameter that is statistically estimated from test time and failure data. One may use different models, data, and statistical techniques. The ones we use include:

- Test time: We investigate the use of calendar time, since data on test cases run and execution time are often not available.
- Defect data: We collect failure data from system test after all functionality has been added. The models assume that the code being tested does not change during test, except for defect repair, and that the effects of repair are minimal, that is not a lot of code is changed. The data is grouped by week.
- Growth models: We investigate two concave models, the G-O model [23] and the Yamada exponential model [76], and two S-shaped models, the delayed S-shaped model [74] and the Gompertz model [29]. These models are applied to the data at the end of each week. We evaluate the models in terms of:
  - Curve fit. How well a curve fits is given by the  $r$ -value (a correlation coefficient). The  $r$ -value is the normalized difference between the data points and the fitting function (or regression model). It has a value between 0 and 1. An  $r$ -value



of 0 implies there is no correlation between the fitting function and the data. A perfect fit has a value of 1.

- Prediction stability. A prediction is stable if the prediction in week  $i$  is within 10 percent of the prediction in week  $i - 1$ ).
- Predictive ability. Error of predictive ability is measured in terms of error (estimate - actual) and relative error (error / actual).

- Statistical technique: A mathematical function is fit to the data. Parameter estimates are obtained using nonlinear regressions.

Figure 9.1 shows the steps of the approach. The steps are described in more detail as follows:

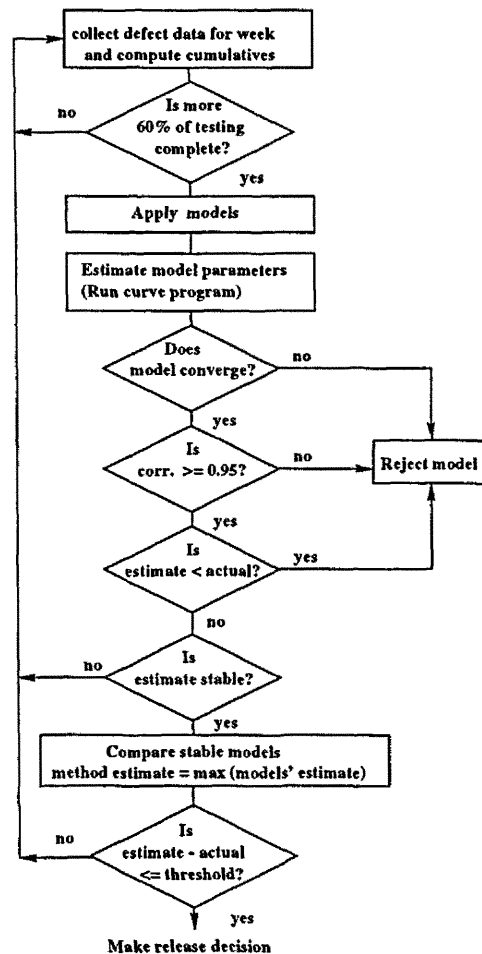


Figure 9.1: Flowchart for SRGM Selection Method.

1. Record the cumulative number of failures found in system test at the end of each week.
2. When at least 60 percent of planned testing is complete, apply the software reliability growth models at the end of each week using a commercial curve fit program that allows the user to enter the equations for the models. The study applying this method uses the G-O, delayed S-shaped, Gompertz, and Yamada Exponential models. (Empirical studies in Musa et al. [37] and Wood [72] indicate that the models typically do not become stable until 60 percent of testing is complete.)
3. The curve fit program estimates a model's parameters by attempting to fit the model to the data.
4. If a fit cannot be performed, a model is said to diverge and the curve fit program outputs a message to that effect. This situation occurs if the model is not appropriate for the data or not enough data has yet been collected. If this situation occurs and more than 60 percent of testing is complete, then the model is probably not appropriate for the data and should not be considered in future weeks.

If a curve can be fit to the data for the model, the results of the fit are computed. The program computes how well the curve fits (the correlation or  $r$ -value, which has a value between 0 and 1), as well as the estimate for the expected number of total failures ( $a$  parameter).

5. The  $r$ -values of the various models for the week should be over 0.95, if the fits are good. If a correlation value is not over 0.95 and more than 60 percent of testing is complete, then the model is not appropriate for the data and is not considered in future weeks.

(We are looking for a few models that fit the best. An  $r$ -value that is too low, for example may include too many models that must be applied week after week that may not perform as well. Alternatively, an  $r$ -value set too high may eliminate too many models, in fact it may eliminate all of them.)

6. The curve fit program provides an estimate for the parameter  $a$ , which is an estimate for the expected number of total failures. If a model's predictions for expected number of total failures are lower than the actual number of failures already found and have been consistently so in prior weeks, the model chosen is inappropriate for the data and should not be used in future weeks.
7. If no model has a stable prediction for the current week, that is within 10 percent of the prediction of the previous week, additional testing effort is required. System test tests and collects failure data for another week.
8. If there is at least one stable model, the method estimate is taken to be the maximum estimate of all stable models. (This is a conservative choice. Alternatively, the method estimate could be the minimum or the mean.) System test determines whether additional testing effort is required by comparing the method estimate to the actual number of failures found. If the method estimate is much higher than the actual number of failures found, additional testing effort may be considered for at least another week. If the difference between the prediction and the actual number of failures found is below the acceptability threshold, the decision to stop testing may be considered.
9. System test should apply the models that were not eliminated in previous weeks at the end of system test to estimate the number of remaining failures that could be reported in post-release. The number of failures in post-release may be estimated by subtracting the number of known failures in system test from the predicted total number of failures.

## 9.2 Results

Our study shows that the method for selecting software reliability growth models based on cumulative failures performs very well in predicting the number of failures close to the total number of failures reported in test and post-release.

Tables 9.1 – 9.5 show the data from three releases using the G-O, delayed S-shaped, Gompertz and Yamada Exponential software reliability growth models to predict the total number of failures. The columns show the test week, the cumulative number of failures found by test week, the prediction for total number of failures (parameter  $a$ ), and the r-value (goodness of fit). The week a model is rejected is indicated by an ‘R’. The week a model stabilizes is indicated by an ‘S’ in the estimate column. (If a model destabilizes, it is indicated by a ‘D’.)

Table 9.1 shows that according to our process framework, the G-O model and the Yamada Exponential model would be rejected as appropriate models at week 11 (60 percent of the way through testing). The G-O model is rejected because the r-value is 0.916, less than the 0.95 value the approach recommends. The Yamada Exponential model is rejected because it does not converge in week 11, indicating that a good curve fit with this model is not possible for the data.

Table 9.1: Release 1 predictions and correlation values.

Test week	Failures found	Delayed S-shaped		G-O		Gompertz		Yamada	
		Est.	r-value	Est.	r-value	Est.	r-value	Est.	r-value
11	139	830	0.950	6722	0.916R	2431	0.970	–	– R
12	152	562	0.962	6910	0.934	796	0.975	–	–
13	164	451	0.970	6889	0.946	412	0.979	–	–
14	164	345	0.972	6054	0.955	276	0.979	–	–
15	165	287	0.971	4906	0.960	227	0.979	5625	0.962
16	168	255	0.972	2574	0.961	207 S	0.979	6557	0.961
17	170	236 S	0.973	1539	0.961	197	0.980	4925	0.961
18	176	226	0.974	986	0.962	193	0.981	1008	0.961

Because visual inspection of the cumulative failure curve in Release 1 indicated that it was more S-shaped than concave, we expect the delayed S-shaped model and the Gompertz model to perform better. Figure 9.2 shows the plot of the failure data from Release 1. It also shows the curves for the delayed S-shaped model and the Gompertz curve. The figure clearly shows that the data has an S-shaped curve. The correlation values in Table 9.1 show that the S-shaped models did provide a good fit to the data in Release 1.

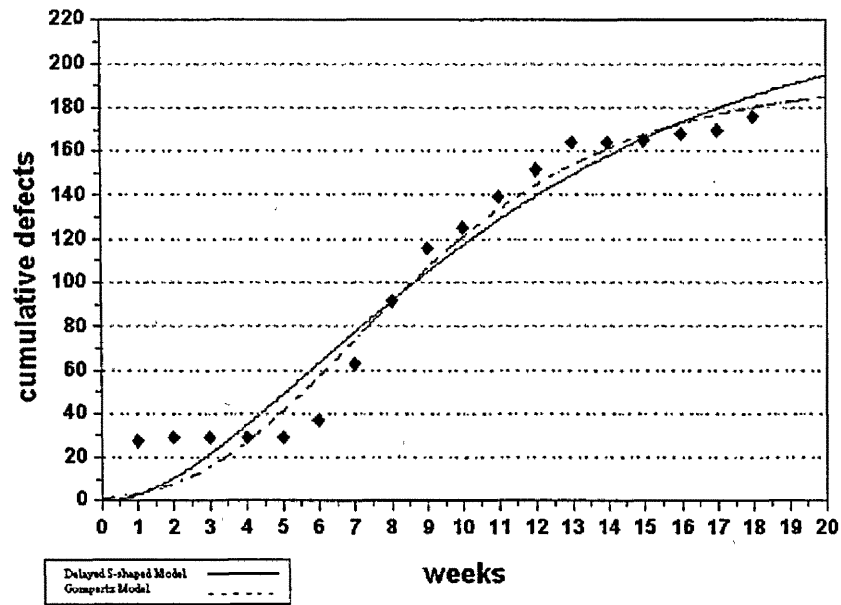


Figure 9.2: Plot of Release 1 data and SRGM models not rejected.

The Gompertz model applied to Release 1 is the first model to stabilize. It stabilizes at week 16. This is the point at which decisions about continuing or stopping testing can be made. The Gompertz model predicts the number of failures at week 16 is 207. Since the delayed S-shaped model is the only stable model at week 16, the method estimate is taken to be the estimate of the G-O S-shaped model. The predicted number of failures, 207, is compared to the actual number of failures, 168. The prediction is 23 percent more than the actual number and probably not below the acceptability threshold. If this is the case, testing continues.

In weeks 17 and 18, both the delayed S-shaped and Gompertz models are stable. The selection method estimate is the maximum estimate of the two models, 236 in week 17 and 26 in week 18. Week 17 and week 18 estimates are 39 percent and 28 percent more, respectively, than the actual number of failures. If this is below the acceptability threshold, testing should continue. System test actually stopped during week 18.

Used as a guide for system testing, the selection method suggests that system testing should have continued beyond week 18. Considering that in post-release, 55 more failures occurred, system testing should probably have continued. This decision is supported by the

opinion of one tester who believed that too many failures occurred after Release 1 and that testing should have continued a bit longer.

Table 9.2: Final estimates and errors by SRGM models not rejected in Release 1.

Model	estimate (compare to 231)	r-value	error	relative error
Delayed S-shaped	226	0.974	-5	-0.022
Gompertz	193	0.981	-38	-0.165

Together with the 176 failures found by the end of system test and the 55 failures found in post-release, the total number of failures is 231 in Release 1. The estimates given by the growth models not rejected should be compared to this value. Table 9.2 shows the prediction provided by the delayed S-shaped model underestimated by five failures, with a relative prediction error of -0.02. This is a very good performance for the model. The Gompertz model underestimated by 38 failures, with a relative error of -0.165. While the Gompertz model provided a better curve fit to the data than did the delayed S-shaped model according to the correlation r-value, it underestimated more and had a higher relative error in its prediction of total number of failures. The selection method correctly chose the best stable model, the delayed S-shaped model, to make release decisions in Release 1.

The method for selecting software reliability growth models based on cumulative failures performed very well in Release 1 in predicting the number of failures close to the total number of failures reported in test and post-release. The approach was applied in Releases 2 and 3, as well.

Table 9.3 shows the SRGMs applied to Release 2. All models, except for the Yamada Exponential model would be rejected according to the approach. Sixty percent of testing is completed by week 11 in Release 2. At that time, the G-O S-shaped model has both a correlation value that is too low and an estimate for the total number of failures that is less than the actual number of failures found - both reasons to reject the model. The Gompertz model has a correlation value that is too low. Table 9.3 shows that there are several weeks in which no new failures occurred. These include weeks 9, 10, 12, 13, 14, and 16. This may

reflect a decrease in testing effort by system test and it may affect the fit of some of the S-shaped models and the prediction values.

Table 9.3: Release 2 predictions and correlation values.

Release 2									
Test week	Failures found	Delayed S-shaped		G-O		Gompertz		Yamada	
		Est.	r-value	Est.	r-value	Est.	r-value	Est.	r-value
11	192	186	0.893R	195	0.964	200	0.986	262	0.967
12	192	187	0.898	195	0.966	198	0.986	283S	0.970
13	192	188	0.902	194	0.969	197	0.986	284	0.970
14	192	188	0.906	194	0.969	196	0.986	320D	0.971
15	203	190	0.906	195R	0.969	197R	0.986	286	0.972
16	203	191	0.907	196	0.969	198	0.986	265S	0.973
17	204	193	0.907	197	0.969	199	0.986	249	0.973

The S-shape models, delayed S-shaped and the Gompertz, did not fit the data for Release 2 well. Because visual inspection of the cumulative curve indicated that it was more concave than S-shaped, we expect the G-O concave model and the Yamada exponential concave model to perform better.

We evaluated the G-O concave model and the Yamada exponential concave model on Release 2 on a week by week basis. Table 9.3 shows that the G-O concave model stabilized at week 7, well before 60 percent of testing was complete. As a predictor of total number of failures, the G-O concave model did not perform well, it predicted fewer total failures than the actual number already found after week 14. According to the approach described, the G-O model should be applied until week 15, until the estimate becomes less than the actual number of failures that occurred. At this point, the G-O model is rejected.

The Yamada exponential concave model performed very well. At week 11, for example, the prediction was 262 failures, while only 192 had been found in system test by that time. The Yamada exponential model would suggest that system testing continue, if 70 failures were unacceptable. By week 17, the model predicts a total of 249 failures, while only 204 have been found. Again, system test should have probably continued testing beyond week 17, since 41 failures occurred in post-release. The Yamada exponential concave model attempts to account for testing effort and this may be the reason it works better on Release 2.

Figure 9.3 shows the plot of the failure data from Release 2. It also shows the curve for the Yamada exponential model. The figure clearly shows that the data has a concave curve.

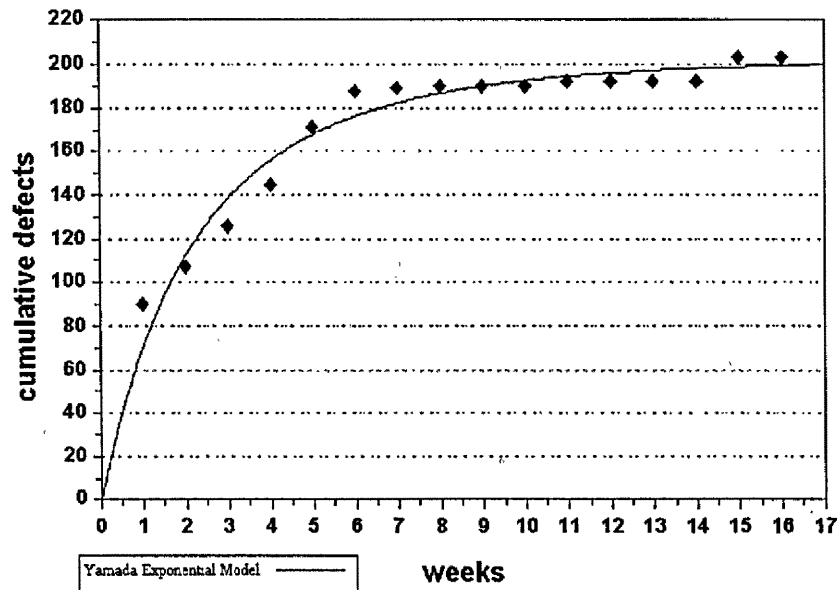


Figure 9.3: Plot of Release 2 data and SRGM models not rejected.

Table 9.4 shows the final curve correlations, predicted total failures and the relative error for Yamada exponential model, the only one not rejected in Release 2.

Table 9.4: Final estimates and errors by SRGM models not rejected in Release 2.

Model	estimate (compare to 245)	r-value	error	relative error
Yamada Exponential	249	0.973	4	0.016

The Yamada exponential concave model has a fairly good correlation at 0.973. It predicts 249 total failures. The actual total number of failures in Release 2 was 245. The Yamada exponential model over-estimates by four failures and has a relative error of 0.016. This model performed very well for Release 2. Other models also investigated in [72] did not fit much better than the ones shown in Table 9.3. The models had correlations between 0.867 – 0.978 and predicted total failures in the range of 203 – 212, underestimating by 33 – 42 defects.



Table 9.5 shows the S-shaped models had a good fit to the data in Release 3. The correlation values for the S-shaped models are even better than in Release 1, with correlations between 0.974 and 0.979 at week 8, 60 percent of the way through testing.

Table 9.5: Release 3 predictions and correlation values..

Release 3									
Test week	Failures found	Delayed S-shaped		G-O		Gompertz		Yamada	
		Est.	r-value	Est.	r-value	Est.	r-value	Est.	r-value
8	63	83	0.974	396	0.966	74	0.979	779	0.966
9	70	84S	0.980	293	0.972	77S	0.983	297	0.972
10	75	86	0.984	214	0.977	80	0.986	216	0.977
11	76	85	0.986	159	0.978	81	0.988	161	0.978
12	76	84	0.987	129	0.977	80	0.989	130	0.977
13	77	83	0.988	114	0.976	80	0.990	115	0.976

The selection method did not reject any of the models investigated in Release 3. Only two models, the S-shaped models stabilized, both at week 9. The G-O S-shaped model is very stable for the last six weeks with predicted failures ranging between 83 and 86. The Gompertz model is also quite stable with predicted failures ranging between 77 and 80.

Table 9.5 shows that the concave models had a high enough correlation value at week 8 and through the remainder of test, but the predictions never stabilized. (The ‘-’ in the tables indicate that the models did not converge.) So while all four models must be applied according to the approach, the estimates for the concave models should not be used to make decisions to stop testing.

The selection method uses the estimates of the delayed S-shaped model, because those estimates are higher. At week 8, the delayed S-shaped model predicts the total number of failures is 83 and the actual number of failures found by that time is 63. This is a 20 percent difference between the estimate and the actual number of failures. This might recommend a continuation of testing, if it is below the acceptability threshold. At week 12, the delayed S-shaped model predicts eight more failures than the actual number of failures that have occurred. This is within 10 percent . If this is above the acceptability threshold, the system test manager may consider stopping test.

Figure 9.4 shows the plot of the failure data from Release 3. It also shows the curves for the delayed S-shaped model and the Gompertz.

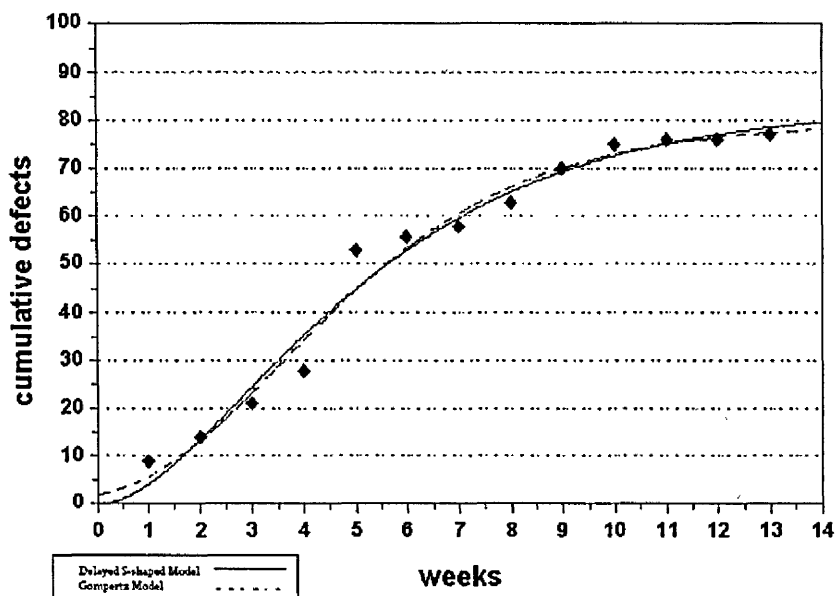


Figure 9.4: Plot of Release 3 data and SRGM models not rejected.

Table 9.6: Final estimates and errors by SRGM models not rejected in Release 3.

Model	estimate (compare to 83)	r-value	error	relative error
Delayed S-shaped	83	0.988	0	0
Gompertz	80	0.990	-3	-0.036

In Release 3, post-release reported six failures. Together with the 77 failures found by the end of system test, this is 83 failures. Table 9.6 shows that the final prediction value using the delayed S-shaped model is exactly 83 failures, giving an error of 0, while the final prediction value using the Gompertz model is 80 failures, an underestimate of 3. As in Release 1, the Gompertz model has a higher correlation value, and thus a better curve fit to the data, but the estimate of the total number of failures has a higher error - the Gompertz model underestimates more than the delayed S-shaped model. Based on this observation, the estimates based on the delayed S-shaped model are better than the Gompertz, if testers want to be more conservative. The selection method correctly chooses the best model, the delayed S-shaped model, for Release 2.

### 9.3 Summary

Results show that the selection method based on empirical data works well in choosing a software reliability growth model that predicts number of failures. The selection method is robust in the sense that it is able to adjust to the differences in the data. This enables it to differentiate between the models: Different models were selected in the releases.

At least one model of those investigated is acceptable after at least 60 percent of testing is complete. In the first and third release, the S-shaped models performed well in predicting the total number of failures. These two releases had data that exhibited an S-shape. The data in Release 2 was concave, rather than S-shaped. It is no surprise that the S-shaped models did not perform well on this data. The Yamada exponential concave model, however performed very well on the data from Release 2. (Other concave models underpredict the total number of failures.)

Software reliability growth models may provide good predictions of the total number of failures or the number of remaining failures. Wood's empirical study [72] has shown that predictions from simple models of cumulative failures based on execution time correlate well with field data. Our study has shown that predictions from simple models based on calendar time correlate well with data from our environment. The empirical selection method described in this paper aids in choosing the appropriate model, when assumptions are not met.

## Chapter 10

# Integration Analysis

Since no one tool or method works on all data, the integrated method uses a set of methods that complement each other with selection criteria for each. Multiple evaluations provide more credible information for decision making. A question is: How does one make decisions in system test to improve effectiveness and efficiency using all the methods that have been described? Figure 10.1 shows a flowchart for applying the methods.

This thesis proposed several assessment techniques to improve effectiveness by identifying components on which testing should focus more attention. Using GYR analysis (which ranks according to the basic defect cohesion measure), the multi-file defect cohesion measure and the defect coupling measures, different components are identified as fault-prone. These techniques actually complement each other. This is useful in situations where the types of system problems are unknown. The types of problems include:

1. Problems that are mainly internal to components.
  - (a) These problems may involve changing only one file per defect repair.
  - (b) They may involve changing many files per defect repair.
2. Problems that are between components.

In Release 1 of this study, the number of components that had internal problems and the number of components that had problems with other components was the same (21). Most of the components that had internal problems involved changing only one file per defect. If only one measure had been applied, some components would have been overlooked. For

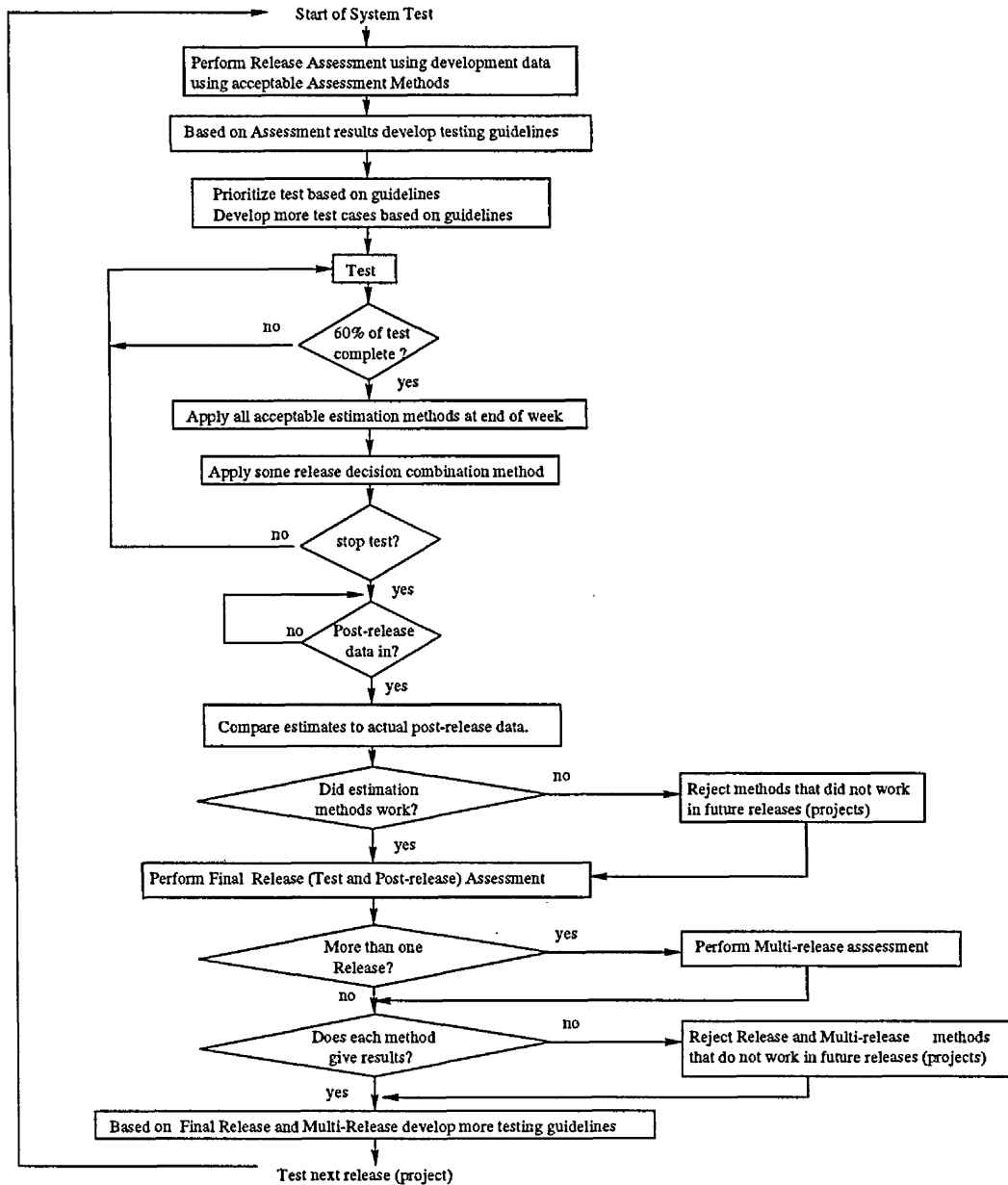


Figure 10.1: Flowchart for applying the integrated method.

example, if only the multi-file defect coupling measure had been applied in Release 1, 19 components would have been missed. It is necessary to apply all the methods that identify fault-prone components, because they may be fault-prone for different reasons.

Applying all assessment techniques allows us to develop a more complete set of testing guidelines. Table 10.1 shows the complete set of guidelines for this case study and the section that they come from.

Table 10.1: Testing guidelines derived from applying set of methods.

Testing Guidelines	Section
1. Test development fault-prone components more thoroughly. (Test components not fault-prone in development less.)	5.2.1
2. Test new components more thoroughly.	5.2.2
3. Test components that are repeatedly fault-prone according to defect cohesion and defect coupling measures more thoroughly.	6.4.2.1
4. Test new components and components that are fault-prone during development as early as possible.	7.2.1
5. Test components that are repeatedly fault-prone according to defect cohesion and defect coupling measures earlier.	7.2.2
6. Assess and improve impact analysis and regression testing to catch old components that are not fault-prone in development or system test, but are fault-prone in post-release.	5.2.3 6.4.2.2

Improving efficiency happens in two ways:

1. Catch problems earlier so they can be fixed before release.
2. Recognize when testing can stop and software can be released.

Guidelines 4 and 5 are concerned with efficiency. They recommend prioritizing testing so that certain components are tested earlier. This study applied prioritization according to Guideline 4 in Section 7.2.1. Prioritization according to Guideline 5 was applied in Section 7.2.2. This study also analyzed the effects of applying both prioritization strategies. The two prioritization strategies have a similar effect on the cumulative defect curve, especially in the first two releases. The improvement in efficiency by applying both guidelines is the same as the improvement in efficiency by the individual guideline that performed the best. However, since one would normally not know a priori which would be the better one, both should be used. In Release 1, the effect of applying both strategies is the same as applying only Guideline 5. In Release 2 and Release 3, the effect is the same as applying only Guideline 4. The effects of applying both guidelines, for this case study, were not additive. Again, since the types of problems present in a system may not be known, it would be difficult for a tester to identify the optimal guideline. In this case, both Guidelines 4 and 5 should be applied.

This thesis proposed several estimation methods to make release decisions. The best methods for this case study include:

- Capture-recapture methods: m0ml, mtml, and mhjk.
- Curve-fitting methods: dpm(linear).
- Experience-based method.
- SRGM selection method.

All these methods are applied iteratively (on a weekly basis) to determine whether testing should stop.

Table 10.2 shows the recommended decisions for the various methods in the last week of system test for the case study. Because the m0ml, mtml and dpm(linear) estimators produce similar estimates, they are grouped together. The table also shows the correct release decision based on the opinion of system test group. Their opinion is based on the number of defects found in post-release. (Thresholds for the capture-recapture, curve-fitting and experience-based methods were set to five defective components. For the SRGM selection method, the threshold was set to ten defects.)

Table 10.2: Recommended release decisions in the last week of system test.

Method	Release 1	Release 2	Release 3
m0ml	stop	stop	continue
mtml	stop	stop	continue
dpm(linear)	stop	stop	stop
mhjk	continue	continue	continue
exp-based	–	continue	continue
SRGM selection	continue	continue	stop
subjective opinion	continue	continue	stop

These methods may be combined in several ways:

1. Logical AND, that is, all methods must say stop (the most conservative).
2. Sequential (apply one rule after another).
3. Majority (least conservative).

The first combination results in decisions to continue testing in all three releases. Based on subjective expert opinion, these decisions are correct for the first two releases, but not the third.

A sequential combination may be defined as follows:

1. If mhjk says stop, stop. (Mjkh is the most conservative estimator.)  
If the mhjk says continue, go to the next rule.
2. If the SRGM selection method and at least one other estimation method says stop, stop. Otherwise go to the next rule.
3. If at least one of the following methods, m0ml, mtml, dpm(linear) and the experienced-based method say stop, stop. Otherwise go to the next rule.
4. Continue test.

This sequential combination results in the correct decisions for all three releases.

The third way of combining methods recommends stopping test, if the majority of the methods recommend stopping. Because the m0ml, mtml and dpm(linear) estimates give estimates that are close to each other, they are grouped together. If at least one of them recommend stopping, the group recommendation is to stop. For Release 1, this gives one recommendation to stop and two to continue, so the decision is to continue. For Release 2, this gives one recommendation to stop and three recommendations to continue, so the decision is to continue. For Release 3, this gives two recommendations to continue and two recommendations to stop, a tie. In the case of a tie, the system test group may decide to be conservative and continue testing for one more week. Alternatively, they may check the number of defects found in the last week. If the number of defects found in the last week



is more than five, then testing should continue. In Release 3, only one defect was found in the last week, so testing should stop. Using the majority of methods in this way results in the correct decision for all three releases.

Using only one method to make release decisions will not necessarily work all the time. (Although the SRGM selection method works for all releases in this study, more case studies are needed to validate this.) Requiring all methods to recommend stopping is probably too conservative. A sequential combination or a majority of recommendations made by several estimation methods would be more robust.

# Chapter 11

## Conclusion

This dissertation proposed an integrated set of methods to improve the effectiveness and efficiency of system testing. This includes performing quality assessment on a release to develop testing guidelines and strategies and make release decisions. It also involves performing quality assessment on multiple releases in order to make longitudinal decisions. Quality assessment includes methods that analyze component fault-proneness, estimate defect content, and derive fault architectures. Testing guidelines and strategies resulting from quality assessment recommend testing fault-prone components more thoroughly and earlier, by prioritizing testing activities. Defect estimation methods are used to make release decisions.

Using the techniques proposed in the integrated method, this thesis was able to answer the research questions for the integrated method posed in Chapter 1. The following sections address these questions. In addition, it summarizes how the answers are dealt with from a decision point of view.

### 11.1 Quality Assessment

This thesis investigated a fault-prone component analysis method based on the GYR technique and the fault architecture technique by applying them in a new case study. This thesis proposed that one should set thresholds using order of magnitude, rather than setting thresholds to percentages. This should work well in situations where the quality of the system is unknown.

Using fault-prone component analysis and the fault architecture technique, the approach in this thesis was able to evaluate the quality of the software and answer the research questions posed in Chapter 1. Quality assessment results for this project show that:

- In each release, the code is of very high quality.
- Only two components out of 188 are fault-prone in all releases.
- Components that are new or fault-prone in development are fault-prone in system test.
- Components with severe problems do not necessarily have more problems.
- Most problems are internal to components, rather than between components.
- Few problems remain undetected.
- The number of post-release problems is very low.
- Most known problems are fixed before release.

The quality of a release may be assessed using defect data with the idea that past behavior is often the best predictor of future behavior. GYR analysis identifies components that need more attention in system test using defect data from development. The fault architectures visualized the kinds of problems the system had, supporting work in a prior study. In addition to applying GYR analysis and the fault architecture technique to perform release assessment, this thesis proposes using these methods in a new way to perform phase assessment within the release. Fault architectures may be derived not only for releases, but also for development phases within a release, in particular development, system test, and post-release. Setting thresholds to an order of magnitude less than the largest measures works very well and should be considered over the use of a percentage. The number of defective components after release may be estimated using data from three and two test sites. The SRGM selection method also performed well in estimating the number of remaining defects.

## 11.2 Test Guidelines and Strategies

Quality assessment of a release may be used to determine test guidelines that might be helpful in improving effectiveness. Assessment methods were able to find several useful guidelines, as well as several ideas that were not helpful. These guidelines have the potential of saving weeks of testing time and hundreds of thousands of dollars without penalty.

The key issue for testers is where to put emphasis, and where not to. Assessment methods are useful in determining guidelines and strategies for testers. Testing problematic components more thoroughly and earlier has the potential to not only improve testing effectiveness, but efficiency. Prioritizing testing has the effect of altering the cumulative defect curves. These guidelines should result in testers finding more defects earlier. The altered curves showed that in the last few weeks before system test actually stopped testing, no defects would have been found. System test would have probably stopped earlier, if no defects were being in those last few weeks.

This being a case study, one cannot expect the particular guidelines developed for this project to improve every project, although they are very sensible. Each project has characteristics that need to be taken into account when determining the guidelines that would best improve system test performance within their environment.

## 11.3 Release Decisions

To make release decisions, this thesis proposed using existing static methods (capture-recapture and curve-fitting) in several new ways.

1. Apply these methods to defect data from system test, rather than review/inspection data.
2. Test sites, rather than individuals, provide the data.
3. Estimate remaining defective components, rather than remaining defects.

A simple experienced-based methods was also proposed to estimate defective components with the goal of making release decisions. This thesis also defined a selection method

to determine the best software reliability growth model(s) to apply to data to estimate remaining defects.

Chapter 1 asked the question: Can defect estimations made using testing data be used to determine the right point at which to stop testing and release software? Results from this case study show that defect estimation methods can provide system test with the ability to make release decisions.

Static and dynamic defect estimation methods provided good estimates. Actual post-release defect content was close to the estimates using system test data. This study was able to use estimates in both types of methods to make decisions to stop testing and release software. Defect estimation methods applied weekly can identify the point at which software is ready for release, thereby saving weeks of testing.

### 11.3.1 Static Defect Estimation

Capture-recapture and curve-fitting methods were successful in estimating the number of components that have defects after release, but no defects in testing. These are the components that were missed in testing. These estimations in turn may be used to recommend decisions on whether to continue testing or to stop testing and release software.

Capture-recapture and curve-fitting estimators that perform the best in this study include the  $m0ml$ ,  $mtml$ ,  $dpm(\text{linear})$  and  $mjhk$ . The  $mjhk$  tended to overestimate slightly. The  $mtml$  or the  $dpm(\text{linear})$  estimators are good choices for the lower bound of the range and the  $mjhk$  estimator is a good choice for the upper bound. These estimators may provide a “window”.

Estimates from these methods provide a good basis for a correct decision to stop testing and release software. The  $mjhk$  estimator appears to perform the best as a basis for decision making. It tends to recommend testing a bit longer and is more conservative than the  $m0ml$ ,  $mtml$ , and  $dpm(\text{linear})$  methods. Because testing too long is preferable to not testing long enough, the  $mjhk$  estimator will probably perform the best, especially if there is no prescreening. Estimators were also shown to be quite robust for two test sites, even when the sites test the system differently.

The estimates from several capture-recapture and curve-fitting methods compare favorably with experience-based estimates. If historical data is available and releases are similar, a simple experience-based method should be used as a complement to the capture-recapture and curve-fitting estimation methods. If, however, no historical data is available or releases are dissimilar, capture-recapture and curve-fitting methods work quite well to provide estimates and to help make decisions for continuing or stopping testing.

### 11.3.2 SRGM Selection Method

The selection method based on empirical data works well in choosing a software reliability growth model that predicts number of defects. The selection method is robust in the sense that it is able to adjust to the differences in the data and to differentiate between the models.

Software reliability growth models may provide good predictions of the total number of defects or the number of remaining defects. Wood's empirical study [72] has shown that predictions from simple models of cumulative defects based on execution time correlate well with field data. Our study has shown that predictions from simple models based on calendar time correlate well with data from our environment. The empirical selection method described in this paper aids in choosing the appropriate model, when assumptions are not met.

## 11.4 Multiple Release Assessment

The study successfully applied defect cohesion and defect coupling measures to identify the most fault-prone parts of the system in three releases. The approach recommends using both the basic and multi-file defect cohesion measures to identify fault-prone components. In our case study defect cohesion measures identified different components as fault-prone. In terms of the defect coupling measures, this thesis investigated the multi-file defect coupling measure and the cumulative defect coupling measure. All of the components that were fault-prone according to the multi-file defect coupling measure were also fault-prone by the

cumulative defect coupling measure. The cumulative defect coupling measure does identify additional components as fault-prone.

The case study showed that defect measures were useful in deriving fault architectures. The fault architectures visualized problems across releases well. Using the fault architectures, it was possible to perform multiple release assessment. This was helpful in answering the research questions in Chapter 1 concerning Multiple Release Assessment. It identified the problems that occurred in release after release. It also indicated that the system had very few persistent problems. Only two components out of 188 consistently had problems.

If the data had been different, the answers to the questions from Chapter 1 may have been different. If the data had been different, the tools that would have worked may have been different. The fault-prone component analysis method (GYR method) did not work well in this case study. This is another argument for using multiple tools in an integrated method.

## 11.5 Validity of Integrated Method

Questions concerning how well the integrated method works and the validity of the study include:

1. How good is the integrated method? That is, can it be empirically validated? Is it objective, robust, and easy to use [14]?
2. Can the integrated method be used in other projects or environments?

This thesis empirically validated the integrated method in a case study. The integrated method was appropriate for the project in which it was applied. The study applied the method to the first release of a large software product. Successive releases validated the method. Techniques in the method give estimates close to actual values. Interviews with testers confirmed other results.

Objectivity was ensured by collecting data by executing SQL commands within the database and obtaining measures algorithmically. The integrated method is robust. Small changes in the data, resulted in small changes in the methods' results. For example, defect

estimation methods applied in successive weeks result in small changes in estimates. The capture-recapture, curve-fitting and experienced-based techniques worked as well with two test sites as three. The SRGM selection method worked well in selecting a model based on the data. Different models were selected in different releases.

The integrated method is easy to use - most of the techniques are easily implemented in a commercial database and spreadsheet package. Other techniques can be implemented in tools or downloaded from the web. There is some effort in reporting the data, but it is worth the benefits. Any organization that wants to produce high quality software should have a good reporting process in place. Many software configuration management tools are available to aid in reporting this data.

Not one method works on all projects or in all environments. Hence, the integrated method employs a strategy of applying a set of techniques that complement each other, so that it will work in more than one environment. Other studies have used some of the techniques successfully in other environments. The integrated method described in this dissertation also proposes some of the following ways to overcome some of the problems these techniques have shown in practice:

- Use indicators appropriate to the project to determine fault-prone parts of the software. Indicators investigated in this study included defect severity, development fault-prone components, and components in fault-prone relationships. Alternative indicators might be documentation defects, components with many defects in inspection, components with many failures in regression testing, etc.
- Use several techniques to identify fault-prone parts of the software.
- Use metrics that more clearly differentiate between fault-proneness and set thresholds to focus attention on the most problematic parts of the software.
- Use the selection method this thesis proposes to determine the software reliability growth model(s) that best fits the data in a given project to make release decisions.
- Use capture-recapture, curve-fitting, and experience-based methods to aid in making release decisions. This thesis showed how these techniques could be used in a new



way in system test to estimate the number of components with defects in post-release that did not have defects in test.

- Apply a series of techniques in early releases or projects to determine those that work best for later projects.

The process model will work if there is a systematic approach to determine

- the best methods to assess software for future releases and projects.
- the best models to make release decisions as testing progresses.

The integrated method allows developers to assess the quality of software. For development environments like the one we analyzed, the key issue for testers is where to put emphasis, and where not to. The integrated method enables developers and testers to determine the most problematic parts of the software. They can then focus their attention on these parts. Testers also want to know when their software is ready for release. The integrated method provides quantitative information to guide testers in making the right decision. Knowing what to focus on and when will help improve testing effectiveness and efficiency.

# Chapter 12

## Future Work

More work needs to be done in several areas. These areas include:

- Fault-prone analysis.

The study in this thesis investigated only a few indicators for fault-proneness, i.e. development fault-prone components and severity of defects. Future work can add to this study by identifying and evaluating other indicators of fault-proneness. Examples include documentation defects (defect reports written against corresponding documentation components) and defects found during regression.

- Fault architecture.

Several activities remain:

1. Investigate other ways to measure defect cohesion and coupling. One method might be to measure cohesion and coupling using number of reports to repair a defect, so that the method is not as sensitive to the number of file changes as in the multi-file methods. For example, if a repair involves changing one or more files in a component, the measure is set to 1, otherwise it is set to 0.

Future work should apply these alternative measures to each release, as well as to the development, system test and post-release phases for each release. The different measures should be analyzed and compared.

2. Fault-prone relationships in this thesis were based on a threshold set to an order of magnitude (ten percent) less than the maximum value for the measures.

Alternatively, one can look at the top percentage of relationships ranked according to the measures. This should be done for the project in this case study and the study in [65]. The effects of using different thresholds should be analyzed and evaluated.

3. Adapt the procedure in two ways:
    - Exclude outliers in threshold calculations.
    - Identify multiple data sets by plotting the distribution.
  4. Evaluate the various fault-architecture methods by looking at the stability of the components between different releases.
- Evaluate other methods to make release decisions.

This thesis investigated methods that estimate defective components and number of defects after release to make release decisions. Another way is to use statistical stopping rules [12, 15, 16, 35, 47, 50, 51, 52, 53, 56].

1. The key issue is how does one parameterize these functions? This is not easy to do. The rules should be based on some other indicator that is available and measurable before testing is completed. An indicator available from development is preferable.

We explored a stopping rule similar to the linear one in [12] for the project, but had no success at operationalizing the parameters. Data does not always exhibit one-to-one linear relationship between weeks and cumulative defects. Future work should explore using data from earlier life cycle phases to operationalize the parameters. Methods to derive parameters using data from development may include:

- Using the maximum number of defects found in week during development prior to the start of system test to determine parameter values.
- Using the number of components that contribute to some percentage, say 98 percent, of the defects found in development prior to the start of system test. This idea is based on Albert diagrams [43].

Both these methods provide values that are close to the optimal value. There are, however, problems with these two methods. The problem with the first method for determining parameter values is that it depends on testing intensity during development, which may vary. This is why the maximum value is chosen to arrive at the most conservative value possible.

The problem with the second method is that it depends on an estimate for the expected number of weeks in system test. This estimate is determined by the system test group prior to the start of system test. The better the estimate is, the more conservative the stopping rule. More analysis is needed to determine how the stopping rule parameter is affected if the estimates are close to the actual number of weeks spent in system test.

The risk here is that a general method for determining the parameters will not be found. If this is the case, then earlier releases may be useful in deriving the values of parameters to be used in succeeding releases. That means that there will not be a stopping rule for the first release of any product based on this method.

2. Evaluate the effect of prioritizing testing activities on stopping rules, in particular how it impacts the determination of parameters.

Currently no method exists for determining the parameters for a stopping rule for prioritized system test. Analyses of comparisons made between actual development data and reorder development data have not yielded a method that will work for this case study.

The risk here is analogous to the risk described in the last section for determining parameter values. A general method for determining these values may not be found. Again, if this is the case, then Release 1 will be needed to derive the values of parameters to be used in succeeding releases.

- Calibration and customization method.

A method for calibrating and customizing the approach for different projects and environments is needed. This involves:

- Determining indicators of fault-proneness for the particular environment.
  - Determining how to set thresholds for the project or environment for fault-prone component analysis, fault architecture analysis, and defect estimation methods.
  - Determining the particular capture-recapture and curve-fitting methods that are appropriate for the environment.
  - Choosing appropriate stopping rules and operationalizing their parameters.
- Evaluate the effects of guidelines and strategies proposed in this case study to future releases of the same project to determine if effectiveness and efficiency improve.
  - Validation of method on more case studies.

The integrated method must be validated on other projects and other environments to improve the external validity of the case study.

# Appendix A

## Case Study Data

### A.1 Failure data used for SRGM selection method.

In system test and post release, a defect report is actually a failure. This study uses failure data from the last drop of system test and post-release. Table A.1 shows cumulative number of failures by week in the last drop of system test for all three releases.

Table A.1: Cumulative number of failures for all three releases.

Test week	Cumulative number of failures		
	Release 1	Release 2	Release 3
1	28	90	9
2	29	107	14
3	29	126	21
4	29	145	28
5	29	171	53
6	37	188	56
7	63	189	58
8	92	190	63
9	116	190	70
10	125	190	75
11	139	192	76
12	152	192	76
13	164	192	77
14	164	192	
15	165	203	
16	168	203	
17	170	204	
18	176		
post release	231	245	83

## A.2 Defect Data for Fault-Prone Analysis

### A.2.1 Defects Found in Development and System Test

Table A.2 shows the number of defects found in development and system test, as well as the number of components with a defect in Releases 1, 2 and 3. This data is used in assessing fault-proneness of components.

Table A.2: Statistics on defects found by development and system test.

	Release 1 (180 components)		Release 2 (185 components)		Release 3 (188 components)	
	development	test	development	test	development	test
Total number of defects	460	177	299	204	19	77
# comp. with a defect	45	32	41	39	11	19
Mean # defects per comp.	2.56	0.98	1.62	1.10	0.10	0.41
Std. Dev.	12.38	3.74	9.11	3.67	0.59	3.12

### A.2.2 Diffusion Matrices based on Severity for Release 2 and Release 3

The results of the fault-prone analysis based on severity between development and testing for Release 2 are presented in Tables A.3 - A.6. The results of the fault-prone analysis based on severity between development and testing for Release 3 are presented in Tables A.7 - A.10. Analysis does not indicate any benefits in using severity levels for prediction of fault-prone components.

Table A.3: Diffusion Matrix for Release 2 by severity 1.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1	4
Development Normal	3 (1 new)	177 (4 new)

Table A.4: Diffusion Matrix for Release 2 by severity 2.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	2	3
Development Normal	3 (2 new)	177 (3 new)

Table A.5: Diffusion Matrix for Release 2 by severity 3.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	2	3
Development Normal	3 (2 new)	177 (3 new)

Table A.6: Diffusion Matrix for Release 2 by severity 4.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	0
Development Normal	0	185 (5 new)

Table A.7: Diffusion Matrix for Release 3 by severity 1.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	0
Development Normal	0 (1 new)	188 (3 new)

Table A.8: Diffusion Matrix for Release 3 by severity 2.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	0
Development Normal	1 (1 new)	187 (2 new)

Table A.9: Diffusion Matrix for Release 3 by severity 3.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	0
Development Normal	2 (2 new)	186 (1 new)

Table A.10: Diffusion Matrix for Release 3 by severity 4.

Threshold $\geq 4$ defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	0
Development Normal	1 (1 new)	187 (2 new)



### A.3 Cumulative Defects

Tables A.11 – A.13 show the cumulative defects by week for unprioritized testing (original data) and prioritized testing by Guideline 3 and Guideline 5. Guideline 3 states that

Table A.11: Release 1 cumulative defects by week for unprioritized and prioritized testing.

Week	Test	Prioritized Guideline 3	Prioritized Guideline 5	Week	Test	Prioritized Guideline 3	Prioritized Guideline 5
1	1	1	1	54	11	13	13
2	1	1	1	55	12	14	14
3	2	2	2	56	14	15	15
4	2	2	2	57	14	16	16
5	2	2	2	58	15	16	16
6	2	2	2	59	15	16	16
7	2	2	2	60	16	16	16
8	2	2	2	61	16	16	16
9	2	2	2	62	16	16	16
10	2	2	2	63	16	18	18
11	2	2	2	64	17	23	23
12	2	2	2	65	17	25	25
13	2	2	2	66	22	26	26
14	2	2	2	67	24	26	26
15	2	2	2	68	26	26	26
16	2	2	2	69	26	26	26
17	2	2	2	70	28	35	26
18	2	2	2	71	29	64	55
19	2	2	2	72	29	90	81
20	2	2	2	73	29	114	105
21	2	2	2	74	29	126	117
22	2	2	2	75	37	139	130
23	2	2	2	76	63	153	144
24	2	2	2	77	92	161	152
25	2	2	2	78	116	167	158
26	2	2	2	79	125	170	170
27	2	2	2	80	139	172	172
28	2	2	2	81	152	174	174
29	2	2	2	82	164	175	175
30	2	2	2	83	164	175	175
31	2	2	2	84	165	175	175
32	2	2	2	85	168	175	175
33	2	2	2	86	170	175	175
34	2	2	2	87	176	176	176
35	2	2	2	88	176	176	176
36	2	2	2	89	176	176	176
37	2	2	2	90	176	176	176
38	2	2	2	91	176	176	176
39	2	2	2	92	176	176	176
40	2	2	2	93	176	176	176
41	2	2	2	94	176	176	176
42	2	2	2	95	176	176	176
43	3	3	3	96	176	176	176
44	3	3	3	97	176	176	176
45	4	4	4	98	176	176	176
46	4	4	4	99	176	176	176
47	4	4	4	100	176	176	176
48	4	4	4	101	176	176	176
49	11	11	11	102	176	176	176
50	11	11	11	103	176	176	176
51	11	11	11	104	176	176	176
52	11	11	11	105	176	176	176
53	11	11	11	106	176	176	176

system test should test components that are new or fault-prone during development earlier. Guideline 5 states that system test should test components that are repeatedly fault-prone in several life cycle phases and/or releases based on the defect cohesion and defect coupling measures earlier.

Table A.12: Release 2 cumulative defects by week for unprioritized and prioritized testing.

Week	Test	Prioritized Guideline 3	Prioritized Guideline 5
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	1	1	1
10	1	1	1
11	2	2	2
12	2	2	2
13	3	3	3
14	3	3	3
15	4	4	4
16	4	4	4
17	28	28	28
18	37	52	52
19	61	61	61
20	73	73	73
21	76	76	76
22	90	90	90
23	107	116	116
24	126	133	133
25	145	152	152
26	171	171	169
27	188	182	188
28	189	199	199
29	190	200	200
30	190	201	201
31	190	201	201
32	192	201	201
33	192	203	203
34	192	203	203
35	192	203	203
36	203	203	203
37	203	203	203
38	204	204	204
39	204	204	204
40	204	204	204
41	204	204	204
42	204	204	204

Table A.13: Release 3 cumulative defects by week for unprioritized and prioritized testing.

Week	Test	Prioritized Guideline 3	Prioritized Guideline 5
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	9	9	5
7	14	16	14
8	21	23	19
9	28	28	26
10	53	53	33
11	56	60	58
12	58	61	61
13	63	64	63
14	70	66	70
15	75	71	75
16	76	76	76
17	76	76	76
18	77	77	77

## A.4 Subsystems and Problematic Components

Table A.14 identifies the subsystems and the components they contain that are considered fault-prone in the fault architecture analysis. Components 30, 38, 101, and 107 are also

Table A.14: Subsystem containment of problematic components.

Subsystem	Components
A	4, 11, 13
B	43
L	20, 22
M	25, 97, 99
S	35, 37
W	93, 94, 96, 114
X	26
U	44, 45, 46, 48, 50, 51, 52, 55, 56, 58, 59, 60, 63, 65, 67, 70, 79, 83, 86, 87, 88, 98

considered subsystems and are problematic. Subsystems 5, 10, 49 and 57 do not contain components and are not fault-prone nor are they in fault-prone relationships.

## A.5 Data used in Static Defect Estimation Methods

### A.5.1 Resulting Estimates

Tables A.15 – A.20 show the estimates provided by the Capture-recapture methods, the curve-fitting methods and the experienced-based method used in the case study. The following tables show the results of applying the estimators in the last six weeks of system test for three test sites and for two test sites.

Table A.15: Release 1 estimates (3 sites).

	m0ml	mtml	mhjk	dpm(lin)	mthChao	cum	dpm (exp)	exp.based
5 weeks earlier	47	46	61	44	–	–	–	–
4 weeks earlier	47	46	58	45	–	–	–	–
2-3 weeks earlier	48	47	58	46	–	–	–	–
1 week earlier	47	46	59	46	–	–	–	–
Actual end date	55	54	68	53	175	81	72	–

Table A.16: Release 2 estimates (3 sites).

	m0ml	mtml	mhjk	dpm(lin)	mthChao	cum	dpm (exp)	exp.based
3-5 weeks earlier	59	55	66	57	–	–	–	64
2 weeks earlier	64	59	72	62	–	–	–	64
1 week earlier	69	63	78	64	–	–	–	67
Actual end date	68	63	77	65	79	102	86	67

Table A.17: Release 3 estimates (3 sites).

	m0ml	mtml	mhjk	dpm(lin)	mthChao	cum	dpm (exp)	exp.based
5 weeks earlier	42	36	44	33	–	–	–	37
4 weeks earlier	43	36	45	35	–	–	–	39
1-3 weeks earlier	47	40	49	37	–	–	–	41
Actual end date	49	41	51	38	41	52	47	42

Table A.18: Release 1 estimates (2 sites).

	m0ml	mtml	mhjk	mtChpm	dpm(lin)	mthChao	cum	dpm (exp)	exp.based
5 weeks earlier	47	47	52	48	50	-	-	-	-
4 weeks earlier	46	46	51	47	52	-	-	-	-
1-3 weeks earlier	47	47	52	47	54	-	-	-	-
Actual end date	55	54	61	55	62	51	104	87	-

Table A.19: Release 2 estimates (2 sites).

	m0ml	mtml	mhjk	mtChpm	dpm(lin)	mthChao	cum	dpm (exp)	exp.based
3-5 weeks earlier	57	57	64	58	71	-	-	-	64
2 weeks earlier	59	58	66	59	71	-	-	-	64
1 week earlier	59	58	66	59	71	-	-	-	66
Actual end date	62	62	70	63	64	59	136	109	66

Table A.20: Release 3 estimates (2 sites).

	m0ml	mtml	mhjk	mtChpm	dpm(lin)	mthChao	cum	dpm (exp)	exp.based
5 weeks earlier	37	36	37	37	33	-	-	-	37
4 weeks earlier	37	37	39	37	35	-	-	-	39
1-3 weeks earlier	41	40	42	41	37	-	-	-	41
Actual end date	43	42	44	43	38	34	51	46	42

## A.5.2 Estimation Errors

Tables A.21 and A.22 shows the errors, the relative errors and the mean absolute relative errors for the estimators for three and two test sites, respectively. It also shows the overall ranking based on the mean absolute relative errors.

Table A.21: Estimation Errors for Static Methods (3 sites).

Estimator	Relative Error				Overall
	Release 1	Release 2	Release 3	Mean Abs.	Ranking
m0ml	-0.068	0.046	0.225	0.1130	4
mtml	-0.085	-0.031	0.025	0.0470	2
mhjk	0.153	0.185	0.275	0.2043	5
mthChao	1.966	0.215	0.025	0.7353	8
cumulative	0.373	0.569	0.300	0.4140	7
dpm (exp curvefit)	0.220	0.323	0.175	0.2393	6
dpm (linear curvefit)	-0.102	0.000	-0.050	0.0507	3
experience-based	-	0.031	0.050	0.0405	1

Table A.22: Estimation Errors for Static Methods (2 sites).

Estimator	Relative Error				Overall
	Release 1	Release 2	Release 3	Mean Abs.	Ranking
m0ml	-0.052	-0.031	0.075	0.0527	5
mtml	-0.070	-0.031	0.050	0.0503	3
mhjk	0.052	0.094	0.100	0.0820	6
mthChao	-.121	0.078	-.150	0.1163	9
cumulative	0.793	1.125	0.275	0.7310	8
dpm (exp curvefit)	0.500	1.703	0.150	0.7843	7
dpm (linear curvefit)	-0.069	0.000	-0.050	0.0397	1
mtChpm	-0.052	-0.016	0.075	0.0477	4
experience-based	-	0.031	0.050	0.0405	2

### A.5.3 $\chi^2$ analysis

Tables A.23 and A.24 shows the  $\chi^2$  analysis between each estimator and the actual values for three sites and two sites, respectively. For all estimators, except the experience-based method, the degrees of freedom is 2 and the critical value for  $\chi^2_{0.05}$  is 5.991. For the experience-based method, the degrees of freedom is 1 and the critical value for  $\chi^2_{0.05}$  is 3.842.

Table A.23:  $\chi^2$  for each estimator for three releases (three sites).

Estimator							
m0ml	mtml	mhjk	mthChao	Cum	dpm(exp)	dpm(lin)	exp-based
0.352	0.153	0.138	23.396	0.605	0.220	0.173	0.004

Table A.24:  $\chi^2$  for each estimator for three releases (two sites).

Estimator								
m0ml	mtml	mhjk	mthChao	Cum	dpm(exp)	dpm(lin)	mtChpm	exp-based
0.207	0.173	0.033	0.081	3.842	2.167	0.172	0.193	0.000

# References

- [1] H. Alberg, O. Johansson, N. Ohlsson, "Predicting Error-Prone Software Modules," *Nordic Telecom Seminar*, (1993), Stockholm.
- [2] D. Ash, J. Alderete, P. Oman and B. Lowther, "Using Software Models to Track Code Health," *Procs. International Conference on Software Maintenance*, (September 1994), Victoria, British Colombia, Canada, pp. 154 – 160.
- [3] V. Basili, M. Zelkowitz, "Analyzing medium-scale software development," *Procs. Third International Conference on Software Engineering*, (1978), pp. 116 – 123.
- [4] V. Basili, B. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27, no. 1, (January 1984), pp. 42 – 52.
- [5] S. Biyani, P. Santhanam, "Exploring Defect Data from Development and Customer Usage on Software Modules over Multiple Releases," *Procs. Ninth International Conference on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 316 – 320.
- [6] L. Briand, K. El Emam, B. Freimut, O. Laitenberger, "Quantitative Evaluation of Capture-Recapture Models to Control Software Inspections," *Procs. of the 8th International Conference on Software Reliability Engineering*, (November 1997), Albuquerque, NM, pp. 234 – 244.
- [7] L. Briand, K. El Emam, B. Freimut, "A Comparison and Integration of Capture-Recapture Models and the Detection Profile Method," *Procs. Ninth International Conference on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 32 – 41.
- [8] L. Briand, J. Daly, V. Porter, J. Wüst, "Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems," *Procs. Ninth International Conference on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 344 – 353.
- [9] K. Burnham, W. Overton, "Estimation of the Size of a Closed Population when Capture Probabilities vary Among Animals," *Biometrika*, vol. 65, no. 3, (December 1978), pp. 625 – 633.
- [10] A. Chao, "Estimating Population Size for Capture-Recapture Data when Capture Probabilities Vary by Time and Individual Animal," *Biometrics*, vol. 48, (March 1992), pp. 201 – 216.
- [11] S. Chen, S. Mills, "A Binary Markov Process Model for Random Testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 3 (March 1996), pp. 218 – 223.



- [12] T. Chen, I. Munn, A. von Mayrhauser, A. Hajjar, "Using Statistical Stopping Rules for More Efficient Verification of Behavioral Models," *Procs. of VLSI'99*, (August 1999), Portugal.
- [13] D. Christenson, S. Huang, "Estimating the Fault Content of Software using the Fix-on-Fix Model," *Bell Labs Technical Journal*, vol. 1, no. 1, (Summer 1996), pp. 130 – 137.
- [14] S. Conte, H. Dunsmore, V. Shen, *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1986.
- [15] S.R. Dalal and C.L. Mallows, "When Should One Stop Testing Software," *J. Am. Stat. Assn.*, vol. 83, (1988), pp. 872-879.
- [16] S. R. Dalal, C. L. Mallows, "Some Graphical Aids for Deciding When to Stop Testing Software," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, (February 1990), pp. 169 – 175.
- [17] S. Eick, C. Loader, M. Long, L. Votta, S. VanderWeil, "Estimating Software Fault Content Before Coding," *Procs. International Conference on Software Engineering*, (1992), Melbourne, Australia, pp. 59 – 65.
- [18] S. Eick, T. Graves, A. Karr, J.S. Marron, A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," to appear in *IEEE Transactions on Software Engineering*.
- [19] K. El Emam, O. Laitenberger, "Evaluating Capture-Recapture Models with Two Inspectors," Fraunhofer Institute for Experimental Software Engineering, Germany, (1999), ISERN-99-08, pp. 1 – 50.
- [20] L. Feijs, R. Krikhaar, R. van Ommering, "A Relational Approach to Software Architecture Analysis," *Software Practice and Experience*, vol. 28, no. 4, (April 1998), pp. 371 – 400.
- [21] P. Frankl, R. Hamlet, B. Littlewood, L. Strigini, "Evaluating Testing Methods by Delivered Reliability," *IEEE Transactions on Software Engineering*, vol. 24, no. 8 (August 1998), pp. 586 – 601.
- [22] H. Gall, K. Hajek, M. Jazayeri, "Detection of Logical Coupling Based on Product Release History," *Procs. of the International Conference on Software Maintenance*, (November 1998), Washington, D.C., pp. 190 – 198.
- [23] A.L. Goel, K. Okumoto, "A Time Dependent Error Detection Model for Software Reliability and Other Performance Measures," *IEEE Transaction on Reliability*, vol. R-28, no. 3, (August 1979), pp. 206 – 211.
- [24] A.L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE Transaction on Reliability*, vol. 11, no. 12, (December 1985), pp. 1411 – 1421.
- [25] T. Graves, A. Karr, J. Marron, H. Siy, "Predicting Fault Incidence using Software Change History," to appear in *IEEE Transactions on Software Engineering*.
- [26] D. Hamlet, J. Voas, "Faults on its Sleeve: Amplifying Software Reliability Testing," *Procs. International Symposium on Software Testing and Analysis*, (August 1993), Cambridge, MA, pp. 89 – 98.
- [27] W. Howden, "Systems Testing and Statistical Test Data Coverage," *Procs. COMP-SAC 97*, (August 1997), Washington, D. C., pp. 500 – 505.

- [28] W. Howden, "Confidence-based Reliability and Statistical Coverage Estimation," *Procs. of the International Symposium on Software Reliability Engineering*, (November 1997), pp. 283 – 291.
- [29] D. Kececioglu, *Reliability Engineering Handbook*, Vol. 2, Prentice-Hall, Englewood Cliffs, N.J., 1991.
- [30] T. Khoshgoftaar, R. Szabo, "Improving Code Churn Predictions During the System Test and Maintenance Phases," *Procs. International Conference on Software Maintenance*, (September 1994), Victoria, British Columbia, Canada, pp. 58 – 66.
- [31] T. Khoshgoftaar, E. Allen, K. Kalaichelvan, N. Goel "The Impact of Software Evolution and Reuse on Software Quality," *Empirical Software Engineering*, vol. 1, (1996), pp. 31 – 44.
- [32] T. Khoshgoftaar, E. Allen, "Classification of Fault-Prone Software Modules: Prior Probabilities, Costs, and Model Evaluation," *Empirical Software Engineering*, vol. 3, (1998), pp. 275 – 298.
- [33] T. Khoshgoftaar, E. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software," *Procs. Ninth International Symposium on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 344 – 353.
- [34] R. Krikhaar, "Reverse Architecting Approach for Complex Systems," *Proceedings of the International Conference on Software Maintenance*, (September 1997), Bari, Italy, pp. 1 – 11.
- [35] B. Littlewood and David Wright, "Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software," *IEEE Transactions on Software Engineering*, vol. 23, no. 11, (November 1997), pp. 673 – 683.
- [36] M. Lyu, editor, *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, NY, 1996.
- [37] J. Musa, A. Iannino, K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, NY, 1987.
- [38] J. Musa, A. Ackerman, "Quantifying Software Validation: When to Stop Testing," *IEEE Software*, (May 1989), pp. 19 – 27.
- [39] J. Musa, "Applying Failure Data to Guide Decisions," *Software Reliability Engineering*, McGraw-Hill, New York, NY, 1998.
- [40] M. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin, "Code Decay Analysis of Legacy Software through Successive Releases," *Procs. IEEE Aerospace Conference*, (March 1999), Section 7.401.
- [41] M. Ohlsson, C. Wohlin, "Identification of Green, Yellow and Red Legacy Components," *Procs. International Conference on Software Maintenance*, (November 1998), Washington, D.C., pp. 6 – 15.
- [42] N. Ohlsson, M. Helander, C. Wohlin, "Quality Improvement by Identification of Fault-prone Modules Using Software Design Metrics," *Procs. International Conference on Software Quality*, (October 1996), Ottawa, Canada, pp. 1 – 13.
- [43] N. Ohlsson, H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, (December 1996), pp. 886 – 894.

- [44] D. Otis, K. Burnham, G. White, D. Anderson, "Statistical Inference from Capture Data on Closed Animal Populations," *Wildlife Monographs*, no. 62, (October 1978).
- [45] D. Parnas, A. van Schouwen, , S. Kwan, "Evaluation of Safety-Critical Software," *Communications of the ACM*, vol. 33, no. 6 (June 1990), pp. 636 – 648.
- [46] H. Petersson, C. Wohlin, "An Empirical Study of Experience-based Software Defect Content Estimation Methods," *Procs. of the International Symposium on Software Reliability Engineering*, (November 1999), Boca Raton, Florida, pp. 126 – 135.
- [47] P. Randolph, M. Sahinoglu, "A Stopping Rule for a Compound Poisson Random Variable," *Applied Stochastic Models and Data Analysis*, vol. 11, (June 1995), pp. 135-143.
- [48] G. Rothermel, R. Untch, C. Chu, M. Harrold, "Test Case Prioritization: An Empirical Study," *Procs. of the International Conference on Software Maintenance*, (August 1999), Oxford, England, pp. 179 – 188.
- [49] P. Runeson, C. Wohlin, "An Experimental Evaluation of an Experience-Based Capture-Recapture Method in Software Code Inspections," *Empirical Software Engineering: An International Journal*, vol. 3, no. 4, (December 1998) pp. 381-406.
- [50] M. Sahinoglu, "The Limit of Sum of Markov Bernoulli Variables in System Reliability Evaluation," *IEEE Transactions on Reliability*, vol.39, (April 1990), pp. 46-50.
- [51] M. Sahinoglu, "Negative Binomial Density of the Software Failure Count," *Proc: Fifth Int. Symp. Computer and Information Sciences(ISCIS)*, vol. 1, (October 1990), pp. 231 – 239.
- [52] M. Sahinoglu, "Compound Poisson Software Reliability Model," *IEEE Trans. Software Engineering*, vol. 18, (July 1992), pp. 624 – 630.
- [53] M. Sahinoglu, U. Can, "Alternative Parameter Estimation Methods for the Compound Poisson Software Reliability Model with Clustered Failure Data," *Software Testing Verification and Reliability*, vol. 7, no.1, (March 1997), pp. 35 – 57.
- [54] M. Sahinoglu, A.K. Alkhalidi, "A Compound Poisson LSD Stopping Rule for Software Reliability," *5th World Meeting of ISBA, Satellite Meeting to ISI-97*, (August 1997), Istanbul.
- [55] M. Sahinoglu, A.K. Alkhalidi, "Bayesian Stopping Rule for Software Reliability," *5th World Meeting of ISBA, Satellite Meeting to ISI-97*, (August 1997), Istanbul.
- [56] M. Sahinoglu, A. von Mayrhauser, A. Hajjar, T. Chen, C. Anderson, "On the Efficiency of a Compound Poisson Stopping Rule for Mixed Strategy Testing," Technical Report, Colorado State University, 1999.
- [57] N. Schneidewind, "Software Metrics Model for Quality Control," *Procs. International Symposium of Software Metrics*, (November 1997), Albuquerque, New Mexico, pp. 127 – 136.
- [58] R. Tesoriero, M. Zelkowitz, "A Model of Noisy Software Engineering Data (Status Report)," *Procs. International Conference on Software Engineering*, (April 1998), Kyoto, Japan, pp. 461 – 464.
- [59] T. Thelin, P. Runeson, "Capture-Recapture Estimations for Perspective-Based Reading - A Simulated Experiment," *Proceedings of the Conference on Product Focused Software Process Improvement*, (June 1999), Oulu, Finland, pp. 182 – 200.

- [60] P. Thevenod-Fosse, H. Weselynck, "An investigation of statistical software testing," *Journal of Software Testing Verification, and Reliability*, vol. 1, no. 2, (July-September 1991), pp. 5 - 25.
- [61] P. Thevenod-Fosse, H. Weselynck, Y. Crouzet, "Software statistical testing," In: *Predictably Dependable Computing Systems*, Eds. B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood, Springer Verlag, (1995), pp. 253 - 272.
- [62] S. Tilley, K. Wong, M. Storey, H. Muller, "Programmable Reverse Engineering," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, (December 1994), pp. 501 - 520.
- [63] M. Trachtenberg, "A General Theory of Software Reliability Modeling," *IEEE Transactions on Reliability*, vol. 39, no. 1, (April 1990), pp. 92 - 95.
- [64] S. Vander Wiel, L. Votta, "Assessing Software Designs Using Capture-Recapture Methods," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, (November 1993), pp. 1045 - 1054.
- [65] A. von Mayrhauser, J. Wang, M. Ohlsson, C. Wohlin, "Deriving a Fault Architecture from Defect History," *International Conference on on Software Reliability Engineering*, (November 1999), pp. 295 - 303.
- [66] A. von Mayrhauser, J. Wang, M. Ohlsson, C. Wohlin, "Choices for deriving a Fault Architecture from Defect History," accepted by *Journal of Software Maintenance*.
- [67] G. Walton, J. Poore, C. Trammell, "Statistical Testing based on a Software Usage Model," *Software Practice and Experience*, vol.5, no.1, (January 1995), pp. 98 - 108.
- [68] J. Whittaker, M. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10 (October 1994), pp. 812 - 824.
- [69] C. Wohlin, P. Runeson, "Defect Content Estimations from Review Data," *Procs. International Conference on Software Engineering*, (April 1998), Kyoto, Japan, pp. 400 - 409.
- [70] C. Wohlin, P. Runeson, "An Experimental Evaluation of Capture-Recapture in Software Inspections," *Journal of Software Testing, Verification and Reliability*, vol. 5, no. 4 (1995), pp. 213 - 232.
- [71] C. Wohlin, H. Petersson, M. Host and P. Runeson "An Empirical Study of Defect Content Estimation for Two Reviewers," submitted to *ACM Transactions on Software Engineering and Methodology*, (1999).
- [72] A. Wood, "Predicting Software Reliability," *IEEE Computer*, vol. 29, no. 11, (November 1996), pp. 69 -78.
- [73] A. Wood, "Software Reliability Growth Models: Assumptions vs. Reality," *Procs. of the International Symposium on Software Reliability Engineering*, vol. 23, no. 11, (November 1997), pp. 136 - 141.
- [74] S. Yamada, M. Ohba, S.Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection," *IEEE Transaction on Reliability*, vol. R-32, no. 5, (December 1983), pp. 475 - 478.
- [75] S. Yamada, M. Ohba, S.Osaki, "Software Reliability Growth Modeling: Models and Applications," *IEEE Transaction on Reliability*, vol. 11, no. 12, (December 1985), pp. 1431 - 1437.

- [76] S. Yamada, H. Ohtera, H. Narihisa, "Software Reliability Growth Models with Testing Effort," *IEEE Transaction on Reliability*, vol. 35, no. 1, (April 1986), pp. 19 – 23.
- [77] M. Yang, A. Chao, "Reliability-Estimation & Stopping-Rules for Software Testing Based on Repeated Appearances of Bugs," *IEEE Transactions on Reliability*, vol. 44, no. 2 (June 1995), pp. 315 – 321.
- [78] T. Yu, V. Shen, H. Dunsmore, "An Analysis of Several Software Defect Models," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, (September 1988), pp. 1261 – 1270.