

## ABSTRACT

Title of Document: A FRAMEWORK FOR SOFTWARE  
RELIABILITY MANAGEMENT BASED ON  
THE SOFTWARE DEVELOPMENT PROFILE  
MODEL

**Arya Khoshkhou, PhD, 2011**

Directed By: Associate Professor Michel Cukier, ENRE  
Professor Ali Mosleh, ENRE

Recent empirical studies of software have shown a strong correlation between change history of files and their fault-proneness. Statistical data analysis techniques, such as regression analysis, have been applied to validate this finding. While these regression-based models show a correlation between selected software attributes and defect-proneness, in most cases, they are inadequate in terms of demonstrating causality. For this reason, we introduce the Software Development Profile Model (SDPM) as a causal model for identifying defect-prone software artifacts based on their change history and software development activities. The SDPM is based on the assumption that human error during software development is the sole cause for defects leading to software failures. The SDPM assumes that when a software construct is touched, it has a chance to become defective. Software development activities such as inspection, testing, and rework further affect the remaining number of software defects. Under this assumption, the SDPM estimates the defect content of software artifacts based on software change history and software development

activities. SDPM is an improvement over existing defect estimation models because it not only uses evidence from current project to estimate defect content, it also allows software managers to manage software projects quantitatively by making risk informed decisions early in software development life cycle. We apply the SDPM in several real life software development projects, showing how it is used and analyzing its accuracy in predicting defect-prone files and compare the results with the Poisson regression model.

A FRAMEWORK FOR SOFTWARE RELIABILITY MANAGEMENT BASED ON  
THE SOFTWARE DEVELOPMENT PROFILE MODEL

By

Arya Khoshkhou

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park, in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2011

Advisory Committee:

Associate Professor Michel Cukier, Committee Chair and Advisor

Professor Ali Mosleh, Co-Chair and Co-Advisor

Professor Peter Sandborn, Regular Member

Professor Aris Christou, Regular Member

Associate Professor Atif Memon, Dean's Representative

© Copyright by  
Arya Khoshkhou  
2011

## Dedication

This work is dedicated to my loving wife Azita, my daughter Arianna, and my son Ervin, who patiently tolerated me for these long years in completing my Doctorate Degree, and my parents for their support and encouragement.

## Acknowledgements

I would like to give special thanks to my advisors Associate Prof. Michel Cukier and Prof. Ali Mosleh for their time, support and valuable advice over the last four years. I am deeply grateful to them for being such great mentors.

I would like also to extend my appreciation to the members of my committee for their time, support and assessment of this research:

- Professor Peter Sandborn
- Professor Aris Christou
- Associate Professor Atif Memon

I also want to give special thanks to my co-worker Brent Olson for his expertise and help and support in collecting the data used for this research.

# Table of Contents

Dedication .....	ii
Acknowledgements .....	iii
Table of Contents .....	iv
List of Tables .....	viii
List of Figures .....	ix
Chapter 1: Introduction .....	1
1.1 Background .....	1
1.2 Motivation .....	3
Chapter 2: Literature Review .....	7
2.1 Overview .....	7
2.2 Software Reliability and Defect Estimation Models .....	7
2.2.1 Jelinski-Moranda Model (1972) .....	7
2.2.2 Goel-Okumoto Imperfect Debugging Model (1978) .....	8
2.2.3 Goel-Okumoto Imperfect Nonhomogeneous Poisson Process Model (1979) .....	8
2.2.4 Littlewood Models (1981) .....	9
2.2.5 Goel Generalized Nonhomogeneous Poisson Process Model (1982) .....	9
2.2.6 Musa-Okumoto Logarithmic Poisson Execution Time Model (1983) .....	10
2.2.7 The Delayed S and Inflection S Models (1983) .....	11
2.2.8 The inflection S model (1984) .....	11
2.2.9 Shigeru Yamada et al. Software Reliability Growth Models with Testing-Efforts (1986) .....	12
2.2.10 Crow, L.H.: Evaluating the reliability of repairable systems (1990) .....	13
2.2.11 Crow, L.H. et al.: Principles of successful reliability growth applications (1994) .....	13
2.2.12 Khoshgoftaar, T.M. et al.: Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System (1996) .....	14
2.2.13 Malaiya, Y. K & Denton, J. D.: Estimating the Number of Residual Defects (1997) .....	14
2.2.14 Kai-Yuan Cai: On Estimating the Number of Defects Remaining in Software (1998) .....	15
2.2.15 Munson, J.C. & Elbaum, S.G.: Code Churn: Measure for Estimating the Impact of Code Change (1998) .....	15
2.2.16 Chulani, S. & Boehm B.: Constructive Quality Model (COQUALMO) (1999) .....	16
2.2.17 Neufelder, A.M.: How to Predict Software Defect Density during Proposal Phase (2000) .....	17
2.2.18 Graves, T.L. et al.: Predicting Fault Incidence Using Software Change History (2000) .....	17
2.2.19 Smidths, C. & Stutzke, M.: A Stochastic Model of Fault Introduction and Removal during Software Development (2001) .....	17
2.2.20 Malaiya, Yashwant K et al.: Software Reliability Growth with Test Coverage Model (2002) .....	18

2.2.21	Nikora, A.P. & Munson: Developing Fault Predictors for Evolving Software Systems (2003).....	19
2.2.22	Bai, Chenggang et al.: On the Trend of Remaining Software Defect Estimation (2003).....	19
2.2.23	Sherriff, M., Nagappan, N. et al.: Early Defect Estimation Model (2005) 20	
2.2.24	Nagappan, N. & Ball, T.: Use of Code Churn to Predict Defect Density (2005) .....	20
2.2.25	Nachiappan Nagappan and Thomas Ball: Static Analysis Tools as Early Indicators of Pre-Release Defect Density (2005).....	21
2.2.26	Ching-Pao Chang et al.: Defect Prevention in Software Processes: An Action-Based Approach (2006).....	22
2.2.27	Ceylan, Evren et al.: Software Defect Identification Using Machine Learning Techniques (2006) .....	23
2.2.28	Askari, M. & Holt, R.: Information Theoretic Evaluation of Change Prediction Models for Large-Scale Software (2006) .....	23
2.2.29	Nagappan, N. et al.: Mining Metrics to Predict Component Failures (2006) .....	24
2.2.30	Bernstein, A. et al.: Improving Defect Prediction Using Temporal Features and Non-linear Models (2007).....	25
2.2.31	Norman Fenton : Using Bayesian Nets to Predict Software Defects and Reliability (2007) .....	26
2.2.32	Norman Fenton et al.: Predicting Software Defects in Varying Development Lifecycles Using Bayesian Nets (2007).....	26
2.2.33	Oral, A.D. & Bener, A.B. Paper (2007) .....	26
2.2.34	Bergander, Torsten et al.: Software Defects Prediction Using Operating Characteristic Curves (2007).....	27
2.2.35	Karim O. Elish & Mahmoud O. Elish: Predicting Defect-Prone Software Modules Using Support Vector Machines (2007).....	28
2.2.36	Y. Hong, et al.: A Value-Added Predictive Defect Type Distribution Model based on Project Characteristics (2008).....	28
2.2.37	Bai, Cheng-Gang, et al.: On the Trend of Remaining Software Defect Estimation (2008).....	28
2.2.38	Mirosław Staron & Wilhelm Meding: Predicting Weekly Defect Inflow in Large Software Projects Based on Project Planning and Test Status (2008)29	
2.2.39	Haider, Syed et al.: Estimation of defects based on defect decay model: ED3M (2008) .....	29
2.2.40	Jiang, Y. et al.: Comparing design and code metrics for software quality prediction (2008) .....	30
2.2.41	Scott, H. & Wohlin, C.: Capture-Recapture in Software Unit Testing (2008) .....	31
2.2.42	Walia, G. S. & Carver, J. C.: Evaluation of Capture-Recapture Models (2008) .....	31
2.2.43	Cheung, L. et al.: Early Prediction of Software Component Reliability (2008). .....	32



2.2.44	Moser, R. et al.: A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction (2008) .....	32
2.2.45	Nagappan, N. & Murphy, B. & Basili: The Influence of Organizational Structure on Software Quality: An Empirical Case Study (2008) .....	33
2.2.46	Afsharian, S. et al.: A Framework for Software Project Estimation Based on Cosmic, DSM and Rework Characterization (2008).....	34
2.2.47	Illes-Seifert, T. & Paech, B.L.: Exploring the Relationship of History Characteristics and Defect Count: An Empirical Study (2008) .....	35
2.2.48	Meneely, A. et al.: Predicting Failures with Developer Networks and Social Network Analysis (2008) .....	36
2.2.49	Ostrand T.J. and Weyuker E.J.: Progress in Automated Software Defect Prediction (2009).....	36
2.2.50	Conclusion.....	37
2.3	Overview of Defect Estimation Models .....	37
2.4	Current State of Software Defect Estimation Models.....	41
2.5	Our Objective in the Context of the Current State of Research.....	43
Chapter 3:	Software Development Profile Model .....	45
3.1	Proposed Work.....	45
3.2	Methodology .....	45
3.3	Software Development Profile Matrix.....	48
3.4	Estimating Change Set Reliabilities Using a Binary Decision Diagram ....	50
3.5	Estimating Total Number of Defective Constructs in Change Sets.....	59
3.5.1	Chao's Heterogeneity-Time Model.....	63
3.6	Modeling Dependencies.....	67
3.6.1	Modeling Dependencies among Change Sets .....	69
3.6.2	Updating Model Parameters using Bayesian Belief Network (BBN) .....	71
3.7	Properties and of Software Development Profile Model.....	74
3.8	Software Development Profile.....	75
Chapter 4:	Case Studies .....	77
4.1	Comparing Test Case Results with Existing Models.....	79
4.1.1	Poisson Regression Model Setup .....	82
4.2	Software Development Profile Estimation Tool (PET) .....	83
4.3	Case Study 1: CCD 693- RRACS Interface .....	87
4.3.1	Software Project Background and History .....	87
4.3.2	Case Study Measurements.....	89
4.3.3	Model Parameter Estimation .....	91
4.3.4	Case Study Results .....	98
4.3.5	Poisson Regression Model Results.....	101
4.4	Case Study 2: CCD 762 – IMF (health care) Changes for PY 2011 .....	106
4.4.1	Software Project Background and History .....	106
4.4.2	Case Study Measurements.....	107
4.4.3	Model Parameter Estimation .....	108
4.4.4	Case Study Results .....	110
4.4.5	Poisson Regression Model Results.....	112
4.5	Case Study 3: CCD 770R2- BMF Health Care Changes for PY 2011 .....	116
4.5.1	Software Project Background and History .....	116

4.5.2	Case Study Measurements.....	117
4.5.3	Model Parameter Estimation .....	118
4.5.4	Case Study Results .....	120
4.5.5	Poisson Regression Model Results.....	123
4.6	Case Study 4: CCD 700- More BMF and Help Tag Changes for PY 2010 128	
4.6.1	Software Project Background and History .....	128
4.6.2	Case Study Measurements.....	129
4.6.3	Model Parameter Estimation .....	130
4.6.4	Case Study Results .....	132
4.6.5	Poisson Regression Model Results.....	135
4.7	Case Study 5: CCD 689- IMF Changes for PY 2010 .....	141
4.7.1	Software Project Background and History .....	141
4.7.2	Case Study Measurements.....	142
4.7.3	Model Parameter Estimation .....	143
4.7.4	Case Study Results .....	144
4.7.5	Poisson Regression Model Results.....	146
4.8	Case Study Conclusion .....	152
Chapter 5:	Summary of Contributions and Future Research Directions .....	154
5.1	Summary of Contributions.....	154
5.2	Limitations of this Research .....	155
5.3	Future Research Directions.....	156
Chapter 6:	Glossary .....	185
Chapter 7:	Bibliography .....	187

## List of Tables

Table 1: SDPM Sample Inspection Worksheet .....	62
Table 2: Examples of CS Codes .....	85
Table 3: Number of Constructs Modified during Each Change Set .....	92
Table 4: SDPM Parameter Estimation .....	92
Table 5: Parameter Updates Based on External Factors .....	95
Table 6: Construct Reliability Estimations .....	98
Table 7: Estimated Number of Defective SLOCs.....	99
Table 8: Coefficient of Correlation – Case Study 1 - (SDPM model).....	101
Table 9: Coefficient of Regression – Case Study 1 - (Poisson Regression).....	101
Table 10: Estimated Number of Defects (Poisson Model) .....	102
Table 11: Coefficient of Correlation (Poisson Regression).....	104
Table 12: Parameter Estimation for DIS/CS 18.4.....	108
Table 13: Construct Reliability Estimations – DIS/CS 18.4.....	109
Table 14: DIS/CS 18.4 Case Study Results .....	110
Table 15: Correlation Analysis – DIS/CS 18.4.....	112
Table 16: Coefficient of Regression – Case Study 2 - (Poisson Regression).....	112
Table 17: Estimated Number of Defects-Case Study 2 - (Poisson Model) .....	113
Table 18: Parameter Estimation for DIS/CS 18.5.....	118
Table 19: Construct Reliability Estimation – DIS/CS 18.5 .....	120
Table 20: Case Study Results – DIS/CS 18.5 .....	121
Table 21: Correlation Analysis – DIS/CS 18.5.....	123
Table 22: Coefficient of Regression – Case Study 3 - (Poisson Regression).....	124
Table 23: Estimated Number of Defects-Case Study 3 - (Poisson Model) .....	124
Table 24: Model Parameters – DIS/CS 17.4.....	130
Table 25: Construct Reliability Estimation – DIS/CS 17.4 .....	132
Table 26: Case Study Results – DIS/CS 17.4.....	133
Table 27: Correlation Analysis DIS/CS 17.4.....	135
Table 28: Coefficient of Regression – Case Study 4 - (Poisson Regression).....	135
Table 29: Estimated Number of Defects-Case Study 4 - (Poisson Model) .....	136
Table 30: Coefficient of Correlation – Poisson Model.....	139
Table 31: Model Parameters – DIS/CS 17.3.....	143
Table 32: Case Study Results – DIS/CS 17.3 .....	145
Table 33: Correlation Analysis DIS/CS 17.3.....	146
Table 34: Coefficient of Correlation – Poisson Regression.....	147
Table 35: Estimated Number of Defects-Case Study 5 - (Poisson Model) .....	148
Table 36: Coefficient of Correlation – Poisson Model.....	150

## List of Figures

Figure 1: Actual vs. Estimated Defect Density .....	21
Figure 2: Classification of Software Reliability Models based on SDLC .....	39
Figure 3: Software Development Stream.....	46
Figure 4: Software Development Binary Decision Diagram.....	51
Figure 5: Defect Injection Probability of Project 1- Before and After Code Inspection .....	54
Figure 6: Defect Injection Probability of Project 1 Before and After Testing.....	56
Figure 7: Capture-Recapture Model Assumptions .....	60
Figure 8: SDPM within the Formal Inspection Process .....	63
Figure 9: Modeling Dependencies among Change Sets .....	70
Figure 10: Example of Bayesian Belief Network Used in Conjunction with SDPM. 73	
Figure 11: Software Development Profile Model - Scalability .....	76
Figure 12: Defects are counted only once in the stream they were injected.....	81
Figure 13: PET – SDPM Profile Estimation Tool .....	84
Figure 14: PET - Change Matrix .....	86
Figure 15: PET – Estimated Number of Defective Constructs.....	87
Figure 16: Software Development Activities .....	89
Figure 17: CCD 693 Binary Decision Diagram.....	97
Figure 18: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 17.10 .....	105
Figure 19: CCD 762 Timeline and Development Activities .....	106
Figure 20: Software Change Matrix – DIC/CS 18.4 release .....	107
Figure 21: CCD 762 Binary Decision Diagram.....	109
Figure 22: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 18.4 .....	115
Figure 23: CCD 770 Timeline and Development Activities .....	117
Figure 24: Software Change Matrix – DIC/CS 18.5.....	118
Figure 25: Binary Decision Diagram – DIS/CS 18.5 .....	119
Figure 26: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 18.5 .....	127
Figure 27: CCD 700 Timeline and Development Activities .....	129
Figure 28: Software Change Matrix – DIS/CS 17.4.....	130
Figure 29: Binary Decision Diagram – DIS/CS 17.4 .....	131
Figure 30: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 17.4 .....	140
Figure 31: CCD 689 Timeline and Development Activities .....	142
Figure 32: Change Set Matrix – DIS/CS 17.3 .....	143
Figure 33: Binary Decision Diagram – DIS/CS 17.3 .....	144
Figure 34: SDPM – Estimated # of Defective SLOCs vs. Observed # of Defective SLOCs – DIS/CS 17.3 .....	151

# Chapter 1: Introduction

## 1.1 Background

Despite hundreds of software reliability and defect estimation models developed over the past few decades, the software reliability discipline is still struggling to establish a reliability estimation and prediction framework [25]. Over the years, many new models have been proposed, discussed, modified and generalized, while some have suffered much criticism [64]. Even today the field of software reliability engineering remains an active area in software engineering. Historically, software reliability engineering has been influenced greatly by hardware reliability theories. This influence has helped statisticians to develop numerous new software reliability models. On the other hand, it has connected software reliability too strongly to hardware reliability theory. This connection has had an adverse effect on the development of new theories in software reliability engineering. Since software is fundamentally different from hardware, many of the proposed software reliability and defect estimation models have limited applicability dictated by their hardware-based assumptions. These assumptions and limitations make many existing software reliability models impractical to use and difficult to validate for software. Furthermore, many of the existing reliability and defect estimation models, like the hardware-based models, rely on observed failure data that is mainly available towards the end of the development life cycle, too infrequent in cases of safety critical applications.

There is a great need to develop new theories for software reliability and defect estimation which can be used to help manage the reliability of software products

while it is still in development. Unlike reliability estimation models that assess the reliability of software systems in production or before release, software reliability management models provide a framework for managing the reliability of software products. However, still today, many software reliability models rely on defect data, which are not available during early development phases. Reliability management models need to start early in the development process and continue throughout the entire development lifecycle. Software reliability management models provide a great value to software managers, practitioners and users.

In [35], we introduced Software Development Profile Model (SDPM) as a causal model for identifying defect-prone software artifacts based on software development activities and software change history. Throughout this dissertation, we use the term “software construct” [9] as the smallest software piece for which data is collected. Depending on the software development project, a construct can be a software line of code (SLOC), function point (FP), function, class, source statement (SS), or any other software unit. In addition, we use the term software artifact [36] as a product that is created during software development containing software constructs. A software artifact can be a source file, a software module, or a software document such as the Software Requirements Specifications (SwRS) produced during software development. SDPM assumes that when a software construct is touched, it has a chance to become defective. Other activities such as inspection and testing are defect factors that affect detection and removal of software defects. Under this assumption, SDPM estimates the reliability of software constructs based on the software change history and development activities. The reliability of software constructs are then

used to estimate the defect content of various software artifacts. Since SDPM uses software change history and software development activities to estimate software defect content, managers can use SDPM to make risk informed decisions and adjust software development activities early in the development lifecycle to manage software defect content.

## **1.2 Motivation**

Knowing which files are most likely defective early in the software development life cycle can be very valuable for software managers. Finding these defects while the software is still in development can help companies better manage the reliability of their software products by making risk informed decisions to use resources more effectively and by focusing efforts on mission critical modules, resulting in more reliable systems at reduced costs [53].

The relationship between fault-prone software modules and other measurable software attributes has been studied by many authors. In the article “Code Churn: A Measure for Estimating the Impact of Code Change” [43] Munson used the rate of change in relative complexity as the index for the rate of fault injection. The relationship between change history and fault-proneness of software modules has been discussed widely in other recent literature as well as [26], [34], [42], [44], [45]. Recent empirical studies show a strong correlation between the change history of a file and its fault-proneness [29], [40], [45-47], [63]. Researchers have applied statistical data analysis techniques such as regression analysis to show the correlation between change history and fault-proneness [45], [46], [63]. These models are generally based on data fitting techniques and rely on historical data. While they

suggest a relationship between fault proneness and certain aspects of the software product, they generally fall short of demonstrating a causal relationship [33]. Causality is defined as a relationship between an event A (the cause) and an event B (the effect), where the second event is understood as a consequence of the first. While correlation is a necessary condition for a causal relationship, it is not sufficient enough to make a causal inference with reasonable confidence. Regression models can also be used to investigate certain software characteristics, such as file size or file age, to show a relationship between these attributes and fault-proneness. Likewise, this correlation, however interesting, does not imply causality. In other words, this relationship cannot be used to imply that large file size causes additional defects in the file. It is not surprising to see inexperienced developers write larger files or modules; thus both large file sizes and large numbers of defects in such artifacts can be caused by lack of experience.

Bayesian Belief Network (BBN) has been used by numerous authors to build a causal model for software defect prediction [23], [24] . Existing causal models are often high level causal relationships as described in [24] and don't consider software development activities. They are often based on the broad assumption that poor quality of development increases the number of defects, or high quality testing increases the proportion of defects found. While these assumptions are valid, they can't be used to model day-to-day software development activities. The motivation behind this work is to introduce a "causal" model that can be used to capture software development activities. This is important because it allows software managers to manage software development's daily activities. This is also an improvement over



existing causal models because incorporating software development activities allows more evidence to be taken into account, resulting in more accurate predictions. This information can then be used to understand the cause and effect relationship and take proactive steps to reduce production level software defects.

A review of current literature on software reliability management shows that there is a great need for new theories in software reliability management. Apart from the aforementioned impracticality during early stages of development due to a reliance on defect data, another shortcoming is that there are simply too few of them available. Furthermore, many proposed software reliability management models are less quantitative and less statistical-based compared to software reliability models [32]. Because of this, there is a need for developing new theories that can be used to manage the reliability of software products during early stages of software development lifecycle.

This dissertation introduces a new causal model for estimating software defect based on software development activities and software change history and presents five case studies showing how it is used in real industrial software development projects. Unlike software reliability and defect estimation models that assess the software product at a given snapshot in time, the proposed model provides a framework for estimating the software defect content and defect-prone files throughout the development lifecycle. We will provide a brief history of software reliability in section 2.1. In Section 2.2, we will provide a literature review of related software reliability and defect estimation models. We will discuss the current status of software reliability in section 2.4 and provide the objective of this dissertation in

Section 2.5. In Section 3 we will discuss in detail the concept of Software Development Profile Model. We will provide five real life case studies in Section 4 that the author was directly involved with and SDPM. In this section we will investigate the performance of SDPM and provide the results. In Section 5 we will provide the summary of contributions and future research directions.

## Chapter 2: Literature Review

### 2.1 Overview

The development of software reliability theory made its greatest jump during the 1970s [8]. During this period many new software reliability and defect estimation models were introduced and software reliability engineering earned recognition among practitioners. In this section we will provide a literature review to cover software reliability and defect estimation models from the 1970's to the present. This section is by no means a complete review of all software reliability models. It is intended to list selected historical models that have influenced the current state of software reliability models and papers relevant to our research. In [13], the authors provide a more complete list of software reliability models.

### 2.2 Software Reliability and Defect Estimation Models

#### 2.2.1 Jelinski-Moranda Model (1972)

The Jelinski-Moranda (J-M) Model was one of the earliest models in software reliability engineering [64]. It estimated time between failures. J-M assumes  $N$  software defects at the beginning of testing, and failures occur randomly, and the relationship between defects and faults is constant. It also assumes the repair time is negligible and no new defects are introduced. Therefore, the software failure rate is constant and decreases over time. The instantaneous hazard function between times of two failures is:

$$Z(t_i) = \emptyset[N - (i - 1)]$$

It is assumed that the number of initial software defects is fixed and annotated by  $N$ .

### 2.2.2 Goel-Okumoto Imperfect Debugging Model (1978)

Unlike the J-M model, which assumed perfect fixes with negligible repair times (perfect debugging), Goel-Okumoto proposed a more realistic imperfect debugging model. In practice, when defects are fixed, new ones are introduced. In this model the hazard function between (i-1)-th and i-th failure is:

$$Z(t_i) = [N - p(i - 1)]\lambda$$

Where N is the number of defects at the start of testing, p the probability of imperfect debugging, and  $\lambda$  is the failure rate per fault.

### 2.2.3 Goel-Okumoto Imperfect Nonhomogeneous Poisson Process Model (1979)

The NHPP (Goel and Okumoto, 1979) was concerned with modeling the number of failures observed in given testing intervals. Goel and Okumoto propose that the cumulative number of failures observed at time t,  $N(t)$ , can be modeled as a nonhomogeneous Poisson process, with a time dependent failure rate. They propose that the time-dependent failure rate follows an exponential distribution. The model is:

$$P\{N(t) = y\} = \frac{[m(t)]^y}{y!} e^{-m(t)}, y = 0, 1, 2, \dots$$

Where

$$m(t) = a(1 - e^{-bt})$$

$$\lambda(t) = m'(t) = abc^{-bt}$$

In this model,  $m(t)$  is the number of expected number of failures observed by time  $t$ ;  $\lambda(t)$  is the failure density;  $a$  is the expected number of failures to be observed eventually,  $a$  and  $b$  are the fault detection rate per fault. Fitting the model curves from actual data and projecting the number of faults remaining in the software is done mainly by means of the mean value, or cumulative density function. The fundamental difference between this model and other models is that it treats the total number of defects to be detected ' $a$ ' as a random variable, which is assumed to depend on the test and other environmental factors.

#### 2.2.4 Littlewood Models (1981)

The Littlewood model (LW) is similar to the J-M model. The LW differs in that it assumes different defects have different sizes, and therefore contribute differently to the software failure. The larger the defect, the easier it is to be identified. Therefore, over time larger defects are identified and removed and the size of remaining defects decreases. Littlewood developed other models based on nonhomogeneous Poisson process, where the failure rate is assumed not to be constant from one failure to the next.

#### 2.2.5 Goel Generalized Nonhomogeneous Poisson Process Model (1982)

Goel (1982) proposed a generalization of the Goel-Okumoto NHPP model by adding one more parameter to the mean value function and failure density function.

$$m(t) = a(1 - e^{-bt^c})$$

$$\lambda(t) = m'(t) = abc^{-bt^c}t^{c-1}$$

Where  $a$  is the expected number of failures to be eventually detected,  $b$  and  $c$  are constants that reflect the quality of testing. This mean value function and failure density is actually the Weibull distribution.

#### 2.2.6 Musa-Okumoto Logarithmic Poisson Execution Time Model (1983)

In the Musa-Okumoto (M-O) model, as in the NHPP model, the observed number of failures by a certain time,  $t$ , is also assumed to be nonhomogeneous Poisson process. However, its mean value function in the M-O model is different. The basic assumption here is that later fixes have a smaller effect on the software's reliability than earlier ones. The logarithmic Poisson process is claimed to be superior for highly non-uniform operational user profiles, where some functions are executed much more frequently than others. Also, the process models the number of failures in a specified execution time instead of calendar time. The model consists of two components, the execution time component and the calendar time component, which provides a systematic approach to convert results to calendar time. The mean value function of this model is:

$$u(t) = \frac{1}{\theta} \ln(\lambda_0 \theta t + 1)$$

Where  $\lambda$  is the initial failure intensity and  $\theta$  is the rate of reduction in the normalized failure intensity per failure.

### 2.2.7 The Delayed S and Inflection S Models (1983)

With regard to the software defect removal process, Yamada et al. (1983) argue that a testing process consists of not only defect reduction process, but also a defect isolation process. Because of the time needed for failure analysis, significant delay can be expected between the first failure observation and the time of reporting. This model uses the delayed S-shaped reliability growth model, in which the observed growth curve of the cumulative number of detected defects is S-shaped. The model is based on the nonhomogeneous Poisson process but with a different mean value function to reflect the delay in failure reporting,

$$m(t) = k[1 - (1 + \lambda t)e^{-\lambda t}]$$

Where  $t$  is time,  $\lambda$  is the error detection rate, and  $k$  is the total number of defects or total cumulative defect rate.

### 2.2.8 The inflection S model (1984)

In 1984, Ohba proposed another S-shaped reliability growth model—the inflection S model (Ohba, 1984). The model describes a software failure detection phenomenon with a mutual dependence on detected defects. This means that the more defects we detect, the more undetected failures become detectable. This assumption brings a certain realism into software reliability modeling and is a significant improvement over other earlier models, namely the independence of faults in a program. Based on the Nonhomogeneous Poisson process, the mean value function is

$$I(t) = K \frac{1 - e^{-\lambda t}}{1 + ie^{-\lambda t}}$$

Where  $t$  is time,  $\lambda$  is the error detection rate,  $i$  is the inflection factor, and  $K$  is the total number of defects or total cumulative defect rate.

#### 2.2.9 Shigeru Yamada et al. Software Reliability Growth Models with Testing-Efforts (1986)

Software Reliability Growth Models are concerned with the relationship between the cumulative number of defects detected and the time span of the software reliability. This paper assumes that the error detection rate is proportional to the current error content. The test effort is defined by exponential and Rayleigh curves.

Assumptions:

- A software system is subject to failure at random times caused by defects remaining in the software
- Each time an error occurs, it is immediately removed and no errors are re-introduced
- The testing effort is described by exponential or Rayleigh curve
- The s-expected number of errors detected in the time interval  $(t, t+1]$  to the current testing-effort expenditures is proportional to the s-expected number of remaining errors.
- The error detection is NHPP



#### 2.2.10 Crow, L.H.: Evaluating the reliability of repairable systems (1990)

The Weibull-Poisson process (WPP) for representing the reliability of complex repairable systems is discussed in [20]. The emphasis is on estimation and other statistical methods for this model when data have been generated by multiple systems. Examples and procedures specifically illustrating these methods are given for several real-world situations. In addition to maximum likelihood estimation methods, goodness-of-fit tests and confidence interval procedures are discussed and illustrated by numerical examples. It is noted that in the case of one system the model reduces to a model for reliability growth. Confidence intervals for the WPP shape parameter and growth rate are given.

#### 2.2.11 Crow, L.H. et al.: Principles of successful reliability growth applications (1994)

This paper discusses the successful application of integrated reliability growth testing (IRGT) to the development of a large switching system, and demonstrates the results obtained using a case study. In usual applications of reliability growth testing, it is customary to dedicate development test items for a period of time and implement design changes to improve the reliability of a fielded product. In IRGT, reliability growth is demonstrated through design changes which occur during development testing. Crow et al. [19] identify the lessons learned from the application of IRGT principles. The success of the IRGT program provided the Switching System Pilot Project with several benefits, including: timely analysis of failed items; accurate problem classification; timely and accurate laboratory failure rates; early identification of pattern failures; metrics demonstrating; achieved reliability growth during development testing. While the Switching System Pilot Project IRGT effort

was largely successful, a configuration management problem area was identified in terms of providing adequate configuration data for reliability analysis.

#### 2.2.12 Khoshgoftaar, T.M. et al.: Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System (1996)

This study defines fault-prone as exceeding a threshold of debug code churn, defined as the number of lines added or changed due to bug fixes. Previous studies have characterized reuse history with simple categories. The study presented in [34] quantifies new functionality with lines of code. The paper analyzes two consecutive releases of a large legacy software system for telecommunications. The authors applied discriminant analysis to identify fault prone modules based on 16 static software product metrics and the amount of code changed during development. Modules from one release were used as a fit data set and modules from the subsequent release were used as a test data set. In contrast, comparable prior studies of legacy systems split the data to simulate two releases. The authors validated the model with a realistic simulation of utilization of the fitted model with the test data set. Model results could be used to give extra attention to fault prone modules and thus reduce the risk of unexpected problems.

#### 2.2.13 Malaiya, Y. K & Denton, J. D.: Estimating the Number of Residual Defects (1997)

Malaiya & Denton argue in [38] that estimating the remaining defects in highly reliable software is challenging since remaining defects are hard to detect. Several different software defect estimation techniques are discussed, including: sampling based methods, fault seeding, estimations based on empirical models and exponential Software Reliability Growth Models (SRGM). Malaiya et al. propose a model

relating the density of remaining defects with test coverage measures. Their model assumes that, at the beginning of the test, defect coverage starts slowly but improves linearly over time.

#### 2.2.14 Kai-Yuan Cai: On Estimating the Number of Defects Remaining in Software (1998)

In [12] the author presents an analysis of the method of dynamic software reliability models, and that of empirical models, particularly of the Halstead model. He develops a new static model for estimating the number of remaining defects and uses a set of real data to test his model. The new model coincides with the Mills model in a particular case and shows its attractiveness in its applicability to a broader scope of circumstances. Bayesian versions of the Mills model and the new model are also developed.

#### 2.2.15 Munson, J.C. & Elbaum, S.G.: Code Churn: Measure for Estimating the Impact of Code Change (1998)

The focus of this paper is on the precise measurement of software development processes and product outcomes. Tools and processes for the static measurement of the source code have been installed and made operational in a large embedded software system. Source code measurements have been gathered unobtrusively for each build in the software evolution process. The measurements are synthesized to obtain the fault surrogate. The complexity of sequential builds is compared and a new measure, code churn, is calculated. In a “Code Churn: Measure for Estimating the Impact of Code Change” [43], the authors demonstrate the effectiveness of code complexity churn by validating it against the testing problem reports.

2.2.16 Chulani, S. & Boehm B.: Constructive Quality Model (COQUALMO)  
(1999)

The authors claim that cost, schedule, and quality are highly correlated factors in software development [18]. COQUALMO is an extension to the existing COCOMO II model presented earlier [10]. Constructive Quality Model is based on two sub-models: defect introduction and defect removal models. The total number of defects introduced is modeled by:

$$\text{total defects introduced} = \sum_{j=1}^3 A_j * (\text{size})^{B_j} * \prod_{i=1}^{21} (\text{DI} - \text{driver})_{ij}$$

And the number of remaining defects is modeled by:

$$DReS_{Est,j} = C_j \cdot DI_{Est,j} \cdot \prod_i (1 - DRF_{ij})$$

Where

$DReS_{Est,j}$ = Estimated number of residual defects for j-th artifact

$C_j$ = Calibrated constant for j-th artifact

$DI_{Est,j}$ = Estimated number of defects of artifact type j introduced

$i$ = 1 to 3 for each DR profile

$DRF_{i,j}$ = Defect Removal Fraction for defect removal profile I and artifact type j

COQUALMO is initially calibrated using expert judgments. When more data on actual completed projects is available the it can be calibrated using Bayesian approach.

2.2.17 Neufelder, A.M.: How to Predict Software Defect Density during Proposal Phase (2000)

The author developed a method in [48] to predict defect density based on empirical data. The author evaluated the software development practices of 45 software organizations. The resulting polynomial was:

$$\textit{Predicted Defect Density} = 0.000003 \cdot x^2 - 0.003635 \cdot x + 1.30437$$

Where  $x$  is the resulting score from a questioner form provided in the model.

2.2.18 Graves, T.L. et al.: Predicting Fault Incidence Using Software Change History (2000)

In this paper Graves et al. attempt to investigate the process by which software changes and the effects of said change on software reliability. The authors [26] find that the change history contains more useful information than a snapshot of the code. For example, the number of lines of code in a module is not as helpful in predicting the number of future defects once one has taken into account the number of times the module has been changed. The authors use change management data from a very large software system to predict the fault distribution over different modules. They argue that the number of times code has been changed is a better indication of how many faults it will contain than its size.

2.2.19 Smidths, C. & Stutzke, M.: A Stochastic Model of Fault Introduction and Removal during Software Development (2001)

A stochastic model is sought that represents the injection and removal of software faults during software development. The authors describe in [61] a stochastic model

that relates the software failure density function to development and debugging error occurrence throughout all phases of software development life-cycle. In this model the data from development and debugging errors are used to create an early prediction of software reliability. Model parameters are derived based on data reported in open literature and other projects.

Model assumptions:

- *Development errors follow a NHPP intensity function  $V(t)$*
- *Software fault count is described by a NMBDWI*
- *Software fault detection follows NHPP*
- *Software failure is caused by exactly 1 fault*

#### 2.2.20 Malaiya, Yashwant K et al.: Software Reliability Growth with Test Coverage Model (2002)

This paper models the relationships between testing time, code coverage, and software reliability. In [39] an LE (logarithmic-exponential) model is presented that relates testing effort to test coverage (block, branch, computation-use, or predicate-use). The model is based on the hypothesis that the enumerable elements (like branches or blocks) for any coverage measure have various probabilities of being exercised; likewise defects have various probabilities of being encountered. This model allows the direct relation of a test-coverage measure with defect-coverage one. The model is fitted to 4 data-sets for programs with real defects. In the model, defect coverage can predict the time to next failure. This paper makes the assumption that

both defect coverage and code coverage are based on the M-O model following a logarithmic model.

#### 2.2.21 Nikora, A.P. & Munson: Developing Fault Predictors for Evolving Software Systems (2003)

The authors have shown in previous work that there is a significant linear relationship between code churn and the rate at which faults are inserted into the system, measured in terms of the number of faults per unit change in code churn. In [50] they investigate this relationship with a flight software technology development effort at the jet propulsion laboratory (JPL) and succeed in resolving the limitations of the earlier work in two distinct aspects. First, they have developed a standard for the enumeration of faults. Second, they have developed a practical framework for automating the measurement of these faults. In this paper, the authors analyze the measurements of structural evolution and fault counts obtained from Nikora and Munson's JPL flight software technology development effort. The results of this study indicate that the measures of structural attributes for the evolving software system are suitable for forming predictors of the number of faults inserted into software modules during their development.

#### 2.2.22 Bai, Chenggang et al.: On the Trend of Remaining Software Defect Estimation (2003)

Software defect curves describe the behavior of the estimated number of remaining software defects as software testing proceeds. They are of two possible patterns: single trapezoidal-like curves or multiple trapezoidal-like curves. In [3] the authors present some necessary conditions for software defect curves from the Goel-Okumoto NHPP model. These conditions can be used to predict the effect of the detection and

removal of a software defect on the variations of the estimated number of remaining defects. In this paper the authors use a field software reliability dataset to justify the trapezoidal shape of software defect curves and the author's theoretical analysis.

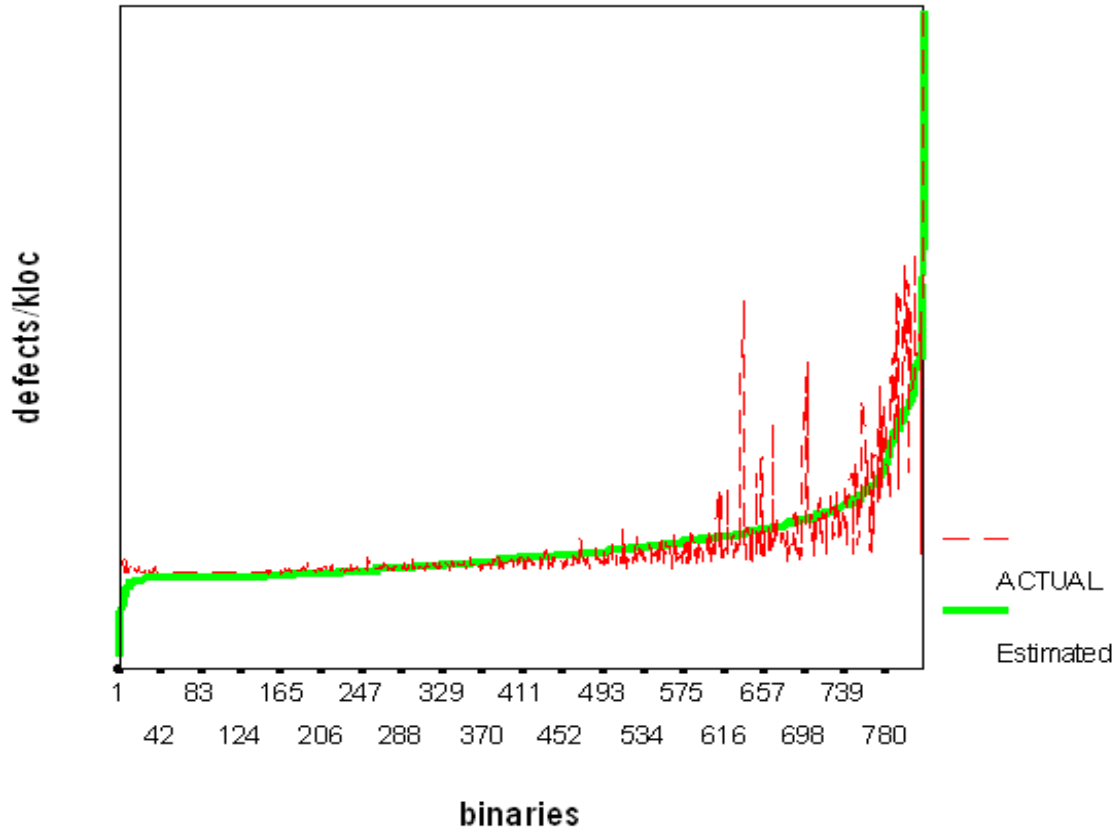
#### 2.2.23 Sherriff, M., Nagappan, N. et al.: Early Defect Estimation Model (2005)

This paper presents a suite of in-process metrics that leverages the software testing effort to create a defect density prediction model for use throughout the software development process. A case study conducted with Galois Connections, Inc. in a Haskell programming environment indicates that the resulting defect density prediction is indicative of the actual system defect density [59].

#### 2.2.24 Nagappan, N. & Ball, T.: Use of Code Churn to Predict Defect Density (2005)

Software systems evolve over time due to changes in requirements, optimization of code, security fixes, reliability bugs, etc. Code churn, which measures the changes made to a component over a period of time, quantifies the extent of this change. In [45] the authors present a technique for early prediction of system defect density using a set of relative code churn measures that relate the amount of churn to other variables such as component size and the temporal extent of churn. Using statistical regression models, they show that while absolute measures of code churn are poor predictors of defect density, the set of relative measures of code churn is highly predictive of defect density. A case study performed on Windows Server 2003 indicates the validity of the relative code churn measures as early indicators of system defect density. Furthermore, the code churn metric suite is able to discriminate between fault-prone and non-fault-prone binaries with an accuracy of 89.0 percent.





**Figure 1: Actual vs. Estimated Defect Density**

#### 2.2.25 Nachiappan Nagappan and Thomas Ball: Static Analysis Tools as Early Indicators of Pre-Release Defect Density (2005)

The authors believe that there is a strong positive correlation between the static analysis defect density and pre-release defect density determined by testing [44]. Using the two static analysis tools, PREFIX and PREFast, the authors tested their hypothesis. The results show that the static defect density is correlated to the pre-release defect density determined by various testing-activities.

#### 2.2.26 Ching-Pao Chang et al.: Defect Prevention in Software Processes: An Action-Based Approach (2006)

In [15] the authors argue that in order to accurately predict the number of defects in a given piece of software, one needs to look at the software development process. They use the Work Breakdown Structure (WBS) to identify all actions that are performed during software development. Factors causing defects vary according to the different attributes of a project, including the experience of the developers, the product's complexity, the development tools and the schedule. The most significant challenge for a project manager is to identify actions that may incur defects before the action is performed. Actions performed in different projects may yield different results, which are difficult to predict in advance. To alleviate this problem, they propose an Action-Based Defect Prevention (ABDP) approach, which applies the classification and Feature Subset Selection (FSS) technologies to project defects during execution. Accurately predicting actions that cause many defects by mining records of performed actions is a challenging task due to the rarity of such actions. To address this problem, the under-sampling is applied to the data set to increase the precision of predictions for subsequence actions. To demonstrate the efficiency of this approach, it is applied to a business project, revealing that under-sampling with FSS successfully predicts the problematic actions during project execution. The main advantage of utilizing ABDP is that the actions likely to produce defects can be predicted prior to their execution. The detected actions not only provide the information to avoid possible defects, but also facilitate the improvement of software development process.

#### 2.2.27 Ceylan, Evren et al.: Software Defect Identification Using Machine Learning Techniques (2006)

In [14], different machine learning algorithms are evaluated in terms of their ability to identify and locate possible defects in a software project. In the proposed methodology the dataset is first normalized and cleaned against correlated and irrelevant values, and then machine learning techniques are applied for error prediction. The defect prediction can be done in two parts. First, it can be used to predict if the code is defective or not. Second, it can be used to predict the magnitude of the possible defect such as its severity, priority, etc. This paper is focused on the second type of predictions. By doing so, the authors argue that they are providing the software quality practitioner with an estimation of “which modules may contain more faults.” This information can be used to allocate the scarce testing and validation resources on the modules that are predicted to be “most defective.”

#### 2.2.28 Askari, M. & Holt, R.: Information Theoretic Evaluation of Change Prediction Models for Large-Scale Software (2006)

In [2], the authors analyze the data extracted from several open source software repositories and show that the change data follows a Zipf<sup>1</sup> distribution. Based on the extracted data, they develop three probabilistic models to predict which files will have changes or bugs.

The first model is Maximum Likelihood Estimation (MLE), which simply counts the number of events, i.e., changes or bugs, that happen to each file and normalizes the

---

<sup>1</sup> The Zipf distribution, sometimes referred to as the zeta distribution, is a discrete distribution commonly used in linguistics, insurance, and the modeling of rare events

counts to compute a probability distribution. The second model is Reflexive Exponential Decay (RED) in which the authors postulate that the predictive rate of modification in a file is incremented by any modification to that file and decays exponentially. They also assume that the predictive rate of bugs induced by any event decays exponentially. The third model is called RED-Co-Change. With each modification to a given file, the RED-Co-Change model not only increments its predictive rate, but also increments the rate for other files that are related to the given file through previous co-changes. The authors then present a theoretic approach to evaluate the performance of different prediction models.

In this approach, the closeness of model distribution to the actual unknown probability distribution of the system is measured using cross entropy. They then evaluate the prediction models empirically using the proposed theoretical approach for six large open source systems. Based on this evaluation, the authors argue that, of the three prediction models, the RED-Co-Change model most accurately predicts the distributions of all the studied systems.

#### 2.2.29 Nagappan, N. et al.: Mining Metrics to Predict Component Failures (2006)

In [46] the authors present an empirical study of the post-release defect history of five Microsoft software systems. They discovered that failure-prone software entities are statistically correlated with code complexity measures. However, they did not observe a single set of complexity metrics that could act universally as the best predictor of defects. Using principal component analysis on the code metrics, they built regression models that accurately predicted the likelihood of post-release defects for new

entities. The approach can easily be generalized to arbitrary projects; in particular, predictors obtained from one project can also be significant for new, similar projects.

#### 2.2.30 Bernstein, A. et al.: Improving Defect Prediction Using Temporal Features and Non-linear Models (2007)

In this paper [7] the authors argued that temporal features (or aspects) of the data are central to predicting performance. They used non-linear models instead of traditional regressions and argued that non-linear models are necessary to uncover some of the hidden interrelationships between the features and the defects and maintain the accuracy of the prediction in some cases.

Using data obtained from the CVS and Bugzilla repositories of the Eclipse project, the authors extracted a number of temporal features, such as the number of revisions and number of reported issues within the last three months. They then used these data to predict both the location of defects (i.e., the classes in which defects will occur) as well as the number of reported bugs in the next month of the project. They used standard tree-based induction algorithms in place of traditional regression models.

They claimed that using non-linear models uncovers the hidden relationships between features and defects, presenting them in easy to understand form. Results also showed that, using temporal features, their model could predict both whether a source file will have a defect with an accuracy of 99% (area under ROC curve 0.9251) as well as the number of defects with a mean absolute error of 0.019 (Spearman's correlation of 0.96).

#### 2.2.31 Norman Fenton : Using Bayesian Nets to Predict Software Defects and Reliability (2007)

In [23], Fenton argued that predicting software defects by complexity and size measures alone will not provide a meaningful estimate because the number of defects detected is related to the amount of testing that is performed. Moreover, complex systems generally have a lower test effectiveness and therefore lower number of discovered defects. Fenton further argued that modeling the complexities of software development using new probabilistic techniques presents a positive way forward. In this paper Fenton suggested using Bayesian Networks (BNs) for predicting software defects and software reliability. This approach allows for the incorporation of causal process factors while combining qualitative and quantitative measures, hence, it overcomes some of the limitations of traditional software metrics methods.

#### 2.2.32 Norman Fenton et al.: Predicting Software Defects in Varying Development Lifecycles Using Bayesian Nets (2007)

In [24], the authors extended their earlier work by describing a general method of using BNs for defect prediction. The limitation of the earlier work was the need to build a different BN for each software development lifecycle to reflect the variation in both the number of testing stages in the lifecycle and the available metric data. To overcome this limitation, the authors described a BN that models the creation and detection of software defects without commitment to a particular development lifecycle.

#### 2.2.33 Oral, A.D. & Bener, A.B. Paper (2007)

This paper examines defect prediction techniques from an embedded software point of view. In [52], the authors presented the results of combining several machine

learning techniques for defect prediction. They believed that the results of this study will help us to find better predictors and models for this purpose.

#### 2.2.34 Bergander, Torsten et al.: Software Defects Prediction Using Operating Characteristic Curves (2007)

The authors propose in [6] a software defect prediction technique using Operating Characteristic curves in order to predict the cumulative number of failures at any given time. The core idea behind their methodology is to use geometric insight in helping construct a prediction method to predict the cumulative number of failures at specific times.

The assumption was that the software failure data is usually available to the user in three basic forms:

- A sequence of ordered failure times  $0 < t_1 < t_2 < \dots < t_n$
- A sequence of inter failure times  $\tau_i$  where  $\tau_i = t_i - t_{i-1}$  for  $i = 1, \dots, n$
- Cumulative number of failures.

The cumulative number of failures  $N(t_i)$  detected by time  $t_i$  (i.e., the cumulative number of failures over the period  $[0, t_i]$ ) defines a non-homogeneous Poisson process (NHPP) with failure intensity or rate function  $\lambda(t_i)$  such that the rate function of the process is time-dependent. The mean value function  $m(t_i) = E(N(t_i))$  of the process is given by

$$m(t_i) = \int_0^{t_i} \lambda(u) du$$

At present software reliability modeling is considered a part of software quality and is listed as a key quality measure for software quality. Currently, the software reliability engineering discipline is saturated with software reliability models and many new models are either generalizations of older models or special cases of existing models [64].

2.2.35 Karim O. Elish & Mahmoud O. Elish: Predicting Defect-Prone Software Modules Using Support Vector Machines (2007)

This paper evaluates the capability of Support Vector Machines (SVM) in predicting defect-prone software modules and compares its prediction performance against eight statistical and machine learning models in the context of four NASA datasets. The results in [22] indicate that the prediction performance of SVM is generally better than, or at least competitive with, the compared models. The authors argue that their method can enable software developers to focus quality assurance activities and allocate effort and resources more efficiently.

2.2.36 Y. Hong, et al.: A Value-Added Predictive Defect Type Distribution Model based on Project Characteristics (2008)

In [28], the authors aim to predict the type and distribution of in-process defects. They proposed a process which includes several steps: 1) analysis of literature, 2) behavior analysis, 3) data gathering, 4) statistical modeling, 5) regression analysis, 6) model validation, 7) gathering of more data for refining the model in the future.

2.2.37 Bai, Cheng-Gang, et al.: On the Trend of Remaining Software Defect Estimation (2008)

In [4], the concept of Remaining Software Defect Estimation (RSDE) curves is proposed. An RSDE curve charts the dynamic behavior of RSDE as software testing



proceeds. Generally, RSDE changes over time and displays two typical patterns: single mode and multiple modes. This behavior is due to the different characteristics of the testing process, i.e., testing under a single testing profile or under multiple testing profiles with various change points. By studying the trend of the estimated number of remaining software defects, RSDE curves can provide further insights into the software testing process. In particular, in this study [4], the Goel-Okumoto model is used to estimate this number on actual software failures and to derive some properties of RSDE curves. In addition, the authors discuss some theoretical and applicability issues regarding the RSDE curves.

#### 2.2.38 Mirosław Staron & Wilhelm Meding: Predicting Weekly Defect Inflow in Large Software Projects Based on Project Planning and Test Status (2008)

In this paper the authors present a new method for predicting the number of defects reported into the defect database on a weekly basis. The method proposed in [60] is based on using project progress data, in particular information about the test progress, to predict defect inflow for the next three weeks. The results show that the prediction accuracy of the models is up to 72% (mean magnitude of relative error for predictions of 1 week in advance is 28%) when used in ongoing large software projects. The method is intended to help project managers more accurately adjust resources in their projects, since they would be notified in advance about any potentially large effort needed to correct defects.

#### 2.2.39 Haider, Syed et al.: Estimation of defects based on defect decay model: ED3M (2008)

In this paper a new approach called ED3M is presented that estimates the total number of defects in an ongoing testing process. ED3M is based on estimation

theory. Unlike other existing approaches the technique presented here does not depend on historical data from previous projects or any assumptions about requirements and/or testers' productivity. It is an automated approach that relies only on the data collected during an ongoing testing process. In [27], the ED3M approach was evaluated using five data sets from large industrial projects and two data sets from the literature. In addition, a performance analysis was conducted using simulated data sets to explore the model's behavior using different models for the input data. The authors argue that the ED3M approach provides accurate estimates with as fast or faster convergence times compared to well-known alternative techniques, all while only using defect data as the input.

#### 2.2.40 Jiang, Y. et al.: Comparing design and code metrics for software quality prediction (2008)

In this paper the authors compare the performance of predictive models which use design-level metrics with those that use code-level metrics and those that use both. In [31], they analyze thirteen datasets from NASA's Metrics Data Program which offers design as well as code metrics. Using a range of modeling techniques and statistical significance tests, they confirm that models built from code metrics typically outperform design metrics based models. However, both types of models prove to be useful as they can be constructed in different project phases. Code-based models can be used to increase the performance of design-level models and thus increase the efficiency of assigning verification and validation activities late in the development lifecycle. They also conclude that models that utilize a combination of design and code level metrics outperform models which use either one or the other metric set.

#### 2.2.41 Scott, H. & Wohlin, C.: Capture-Recapture in Software Unit Testing (2008)

This paper presents a method for estimating the total amount of software failures using a/the capture-recapture method. The method presented in [58] combines the results from having several developers test the same unit with capture-recapture models to create an estimate of “remaining” number of failures. The evaluation of the approach consists of two steps: first a pre-study where the tools and methods are tested in a large open source project, followed by an add-on to a project at a medium-sized software company. The evaluation was a success. An estimate was created, and it can be used both as a quality gatekeeper for units and an input to functional and system testing.

#### 2.2.42 Walia, G. S. & Carver, J. C.: Evaluation of Capture-Recapture Models (2008)

This paper argues that previous research on evaluated capture-recapture models were mostly done on artifacts with a known number of defects. Therefore, before applying capture-recapture models in real development, an evaluation of those models on naturally-occurring defects is imperative.

The study in [62] is based on the data drawn from two inspections of real requirements documents created as part of a capstone course. The results show that estimators change from being negatively biased after one inspection to being positively biased after two.

The findings contradict the earlier results which suggested that a model which includes two sources of variation is a significant improvement over models with only one source of variation. The study also suggests that estimates are useful in determining the need for artifact re-inspection.

#### 2.2.43 Cheung, L. et al.: Early Prediction of Software Component Reliability (2008).

Authors in [17] argue that the ability to predict the reliability of a software early in the development (e.g., during architectural design) can help to improve the system's quality and save on cost. Existing architecture-level reliability prediction approaches focus on system-level reliability and assume that the reliabilities of individual components are known. In general, this assumption is unreasonable, making component reliability prediction an important missing ingredient in the current literature. Early prediction of component reliability is a challenging problem because of the uncertainties associated with components under development. The authors address these challenges in developing a software component reliability prediction framework. They do this by exploiting architectural models and associated analysis techniques, stochastic modeling approaches, and information sources available early in the development lifecycle. They evaluate their framework to illustrate its utility as an early reliability prediction approach.

#### 2.2.44 Moser, R. et al.: A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction (2008)

In this paper the authors analyze two different defect prediction metrics. The authors in [42] choose one set of product-related and one set of process-related software metrics and use them for classifying Java files from the Eclipse project as defective or defect-free. They built classification models using three common machine learners: logistic regression, Naive Bayes, and decision trees. To allow different costs for prediction errors, the authors performed cost-sensitive classification, which proved to be successful. The authors claimed having over 75 percentage of files correctly

classified with less than 30 percentage false positive. Results indicated that for the Eclipse data, process metrics were more efficient defect predictors than code metrics. In general, the authors aim to answer one or several of the following questions in [42]:

- Which metrics that are easy to collect during the early phase of software development are good defect predictors?
- Which models, quantitative, qualitative, hybrid, etc., should be used for defect prediction?
- How accurate are those models?
- How much does it cost a software organization to utilize defect prediction models and what are the benefits?

#### 2.2.45 Nagappan, N. & Murphy, B. & Basili: The Influence of Organizational Structure on Software Quality: An Empirical Case Study (2008)

In this paper the authors presented a metric scheme to quantify organizational complexity in relation to the product development process. They also used the proposed metrics to identify defect-prone files. In the case study presented in [47], the organizational metrics, when applied to data from Windows Vista, were statistically significant predictors of failure-proneness. The precision and recall measures for identifying failure-prone binaries, using the organizational metrics, were significantly higher than those derived from traditional metrics (churn, complexity, coverage, dependencies, and pre-release bug measures). The authors concluded that the results provide empirical evidence that the organizational metrics are related to, and can effectively predict, defect-proneness.

One of the organizational metrics used in this paper is Edit Frequency (EF). This measures the total number of times the source code that makes up the binary is edited. An edit is an instance when an engineer checks code out of the VCS, alters it and checks it back in again. This is independent of the number of lines of code altered during the edit. This measure serves two purposes. One being that, if a binary had too many edits it could be an indicator of the lack of stability/control in the code from the different perspectives of reliability, performance etc., this is even if a small number of engineers were making the majority of the edits. Secondly, it provides a more complete view of the distribution of the edits: did a single engineer make the majority of the edits, or were they evenly distributed amongst the engineers? The EF cross balances with NOE and NOEE to make sure that a few engineers making all the edits do not inflate our measurements and ultimately affect our predictive model. Also, if the engineers who made most of the edits have left the company (NOEE) then it can lead to the above discussed issues of knowledge transfer.

#### 2.2.46 Afsharian, S. et al.: A Framework for Software Project Estimation Based on Cosmic, DSM and Rework Characterization (2008)

In this paper the authors propose a framework, developed by Ericsson R&D Italy, for project time and cost estimation for software development projects in the telecommunications domain. The customization of Design Structure Matrix (DSM), the application of COSMIC and the study of defect complexity curves are the components of their estimation framework presented in [1]. The authors argue that rework is the main cause of software deviations.

2.2.47 Illes-Seifert, T. & Paech, B.L.: Exploring the Relationship of History Characteristics and Defect Count: An Empirical Study (2008)

In this paper, the authors present the results of an empirical study, exploring the relationship between history characteristics and defects in software entities. In [29], they analyze and present nine open source Java projects. The results show that there are some history characteristics that highly correlate with defects in software, e.g., the number of changes and the number of distinct authors performing changes to a file. The number of co-changed files does not correlate with the defect count. The following three hypotheses were tested in the study:

*H1: The more distinct authors changing a file, the higher the file's defect count will be. The rationale behind this hypothesis is that "too many cooks spoil the broth."*  
**CONFIRMED!**

*H2: The more changes made to a file, the higher the defect count will be. The rationale behind this hypothesis is that a high amount of changes indicates that particular parts of the problem are not well understood and often need rework resulting in fault-prone files.* **CONFIRMED!**

*H3: The higher the number of co-changed files, the higher the defect count. The rationale behind this hypothesis is that a local change, affecting just one file, will cause fewer defects than changes affecting more files.* **REJECTED!**

2.2.48 Meneely, A. et al.: Predicting Failures with Developer Networks and Social Network Analysis (2008)

In [40], the authors examine developer collaboration and combine this information with code churn in an effort to predict failures at the file level. They conducted a case study involving a mature Nortel networking product of over three million lines of code. Failure prediction models were developed using test and post-release failure data from two releases, then validated against a subsequent release. One model's prioritization revealed 58% of the failures in 20% of the files compared with the optimal prioritization that would have found 61% in 20% of the files, indicating that a significant correlation exists between file-based developer network metrics and software failures.

2.2.49 Ostrand T.J. and Weyuker E.J.: Progress in Automated Software Defect Prediction (2009)

The authors developed a tool to predict which files are most likely to have defects in future releases. The tool proposed in [63] is based on a regression model and uses the system's defect history to produce a list of possible fault-prone files. The proposed method extracts data from configuration management to predict fault-prone files in the current release. The model is based on each file's change history, fault history, size, and the programming language. The file history used by the tool is based on the number of previous releases that contained a specific file. For change history the tool uses the number of submitted MRs. The tool uses an automated Configuration Management device to gather information needed to predict fault-prone files with limited user interaction. Development of such automated tools and the shift in focus toward change history is a clear indication that new theories in software reliability are



being developed. There are however some shortcoming with the proposed tool. Since software has no aging property, mere presence of a file in the solution should not affect its fault-proneness. The tool also uses the number of Modification Requests (MR) in order to track changes made to the file's configuration management. While the number of submitted MRs is a good indication of the number of changes, it ignores the size and impact of the change. MRs vary greatly in size. Some MRs impact a large portion of the code, replacing almost the entire file, while others might simply change a single character. Using the number of MRs to track changes is a convenient method that captures the number of changes made to a file, but it is not a true indicator of the file's change history.

#### 2.2.50 Conclusion

This section has provided a synopsis of some of the most significant and relevant software reliability models that have been published in the field of software reliability. The literature review has addressed some of the issues related to software reliability. Numerous additional papers were reviewed during the research but are not presented above. Not all papers reviewed are presented here due to their relative importance and the sheer volume of available literature

### **2.3 Overview of Defect Estimation Models**

In section 2.2 we provided a literature review of many existing software reliability and defect estimation models. In order to categorize these models, a suitable classification is needed. Due to the large number of models available, it is difficult to find one method of classification; thus different classifications have been suggested. One method of classification can be based on the probabilistic assumptions made in

the model. Classification based on these assumptions is helpful because it provides insight for development of new models based on more realistic assumptions than existing categories. Some software reliability models incorporate a stochastic process in their description of the failure phenomenon, such as the Markov process assumption, or the non-homogeneous Poisson process. Other models are based on the subjective knowledge of the failure data or the Bayesian inference. Some do not consider the dynamic aspects of failure process, such as input-domain based models, fault seeding and tagging models. In [13], Cai provides the following classification based on the model assumptions:

#### **Markov Models**

- Jelinski-Moranda (J-M) Model
- Schick-Wolverton Model
- Shanthikumar Model
- Littlewood Semi-Markov Model

#### **Nonhomogeneous Poisson Process Models**

- Goel-Okumoto (GO) model
- S-Shaped NHPP Model
- Musa Time Execution Model

#### **Bayesian Models**

- Littlewood-Verrall (LV) Model
- Langberg-Singpurwalla Model

#### **Statistical Data Analysis Methods**

- Crow and Singpurwalla Model

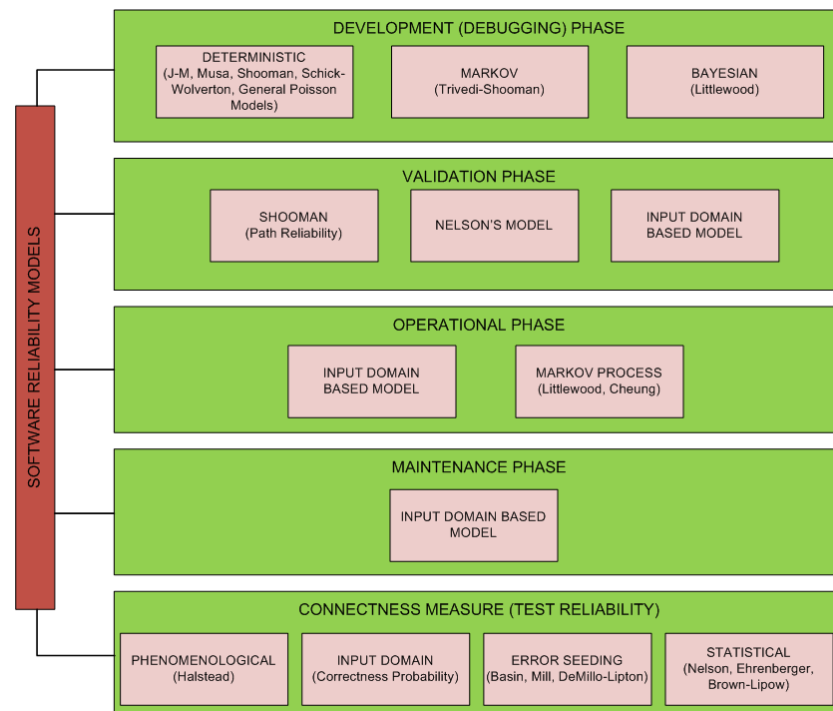
### Input-Domain-Based Models

- Nelson Model

### Seeding and Tagging Models

- Mills Model
- Peterson Model
- Lipow Model
- Software Metrics Models

Ramamoorthy and Bastani [57] provide a categorization of existing software reliability models according to the phase of the software development where the model is most appropriate (figure 2).



**Figure 2: Classification of Software Reliability Models based on SDLC**

In general, existing software reliability and software defect estimation models can be grouped into two broad categories: static models, and dynamic models [32]. Static models use different attributes from current or similar projects to estimate the technical reliability measures of the current project. These models also use some characteristics of the current project as the input parameters. Static models are called static because the coefficients of their parameters are static and are estimated based on a number of selected factors from previous projects. Dynamic models, on the other hand, are based on statistical distributions and use observations from the current project to estimate defect content and a software product's reliability. Dynamic models use the observed defects during the software development phase to estimate an end-product's reliability or defect content at release time. By using data obtained from the current project, dynamic models can provide a more accurate prediction specific to the project. Dynamic software reliability and defect estimation models can be divided into two classes: those that use data obtained during the entire software development life cycle to estimate model parameters, and those that focus on the data obtained during the back-end of the project, specifically the testing phase. Since more defect data is typically available during the testing of the final software product at the end of the project, most of the existing dynamic models belong to this group. Models that are based on exponential distribution and other reliability growth models usually belong to the back-end testing phase category as well. The Rayleigh model is an example of a dynamic model that can be used throughout the software development lifecycle. Rayleigh distribution is a special case of Weibull distribution; Its PDF increases to a peak and then decreases at a decelerating rate. The Rayleigh model is

based on the assumption that the software defect removal pattern follows the Rayleigh distribution. Under this assumption, the defect data obtained from each software development phase can be used to obtain a Rayleigh model that fits the defect pattern to estimate the expected number of remaining defects after the software is released.

## **2.4 Current State of Software Defect Estimation Models**

In the previous section we categorized software reliability and defect estimation models into two categories: static and dynamic models. We claimed that the parameters of static models are estimated based on a number of factors that may relate to software defects. The correlation between code churn and defect-proneness has been studied by a number of research teams. In [49], Munson et al. studied the change in relative software complexity in over 18 software builds and estimated the fault surrogate in the software product. The authors used a set of complexity measures that are known to be highly correlated to software faults for estimating the software fault surrogate. They discovered a strong relationship between software faults and certain aspects of software complexity. The authors used the rate of change in relative complexity as the index of the rate of fault injection. They developed a regression model relating complexity measures of the code to code faults. In [53], Ostrand et al. used a negative binomial regression model utilizing four years of data from previous releases to show a correlation between selected predictor variables and the numbers of faults observed in files.

The list of software defect factors seems inexhaustible, especially when we consider that multiple measures can apply to a single factor. The complexity among various

factors and measures has led to many arguments and controversies [13]. After studying a number of software defect factors over seven case studies in [37], the authors have noted that the differences in the number of defects could not be explained by any combination of software structural metrics. This implies that there is a need for models that incorporate software process. The main limitation is that regression models can only show a correlation between variables and do not prove causality. Since factors that affect defect content are different and vary from project to project, the assumption that the same correlation always exists between selected predictor variables in any software development project is unfounded.

Fenton et al. in [24], review various approaches for software defect prediction and concludes that traditional regression modeling alone is inadequate. In [24] the authors claim that causal models are needed for more accurate prediction. Khoshgoftar and Goel [34] explored the relationship between debug code churn and fault-prone modules. The authors analyzed two consecutive releases of a large communication software to identify fault-prone modules based on the number of debug code changes during development. They labeled fault-prone modules as those that exceeded a threshold of debug code churn. Their model can be used to focus extra attention on fault-prone modules and thus reduce the risk of unexpected problems. In [26], T.L. Graves et al. analyzed the effect of code change on software complexity and argued that the change in code has an impact on the fault surrogate. Moser et al. in [42] showed that program quality metrics are closely related to software complexity metrics and code churn. Other researchers have also examined code churn and its relationship to defect density. The authors in [59] present a suite of

in-process metrics that leverage the software testing effort to create a defect density prediction model for use throughout the software development process. A case study conducted with Galois Connections, Inc. in a Haskell programming environment indicated that the resulting defect density prediction is indicative of the actual system defect density. In [44], Nagappan and Ball find a significant linear relationship between code churn and the rate at which faults are inserted into the system in terms of number of faults per unit change in code churn. In [45], Nagappan and Ball present a technique for early prediction of system defect density using a set of relative code churn measures that relate the amount of churn to other variables like component size and the temporal extent of churn. Results from [45] also show that there are some change history characteristics that highly correlate with defects in software, e.g., the number of changes and the number of distinct authors performing changes to a file. While the relationship between code churn and software defect density has been discussed by many researchers, to our knowledge no causal model has been proposed that captures the change history of products and the software development activities.

## **2.5 Our Objective in the Context of the Current State of Research**

While the relationship between code churn and software defect density has been discussed by many researchers, to our knowledge no causal model has been proposed that can be used to identify defect-prone artifacts based on software development activities and change history. Thus, our objective is to introduce a causal model that uses software development activities and change history to identify defect-prone software artifacts early in the development lifecycle. A model that draws from

current software development activities to estimate the defect content in a software product is very useful to the software engineering community. Using observations from an ongoing software development project not only provides more accurate defect prediction, it also supplies the framework software managers need to make risk informed decisions early in the software development lifecycle. Rather than relying on defect data which is mostly available toward the end of the software development lifecycle, the SDPM can be used throughout a project's development to produce a more reliable product. Our objective is also to investigate the accuracy of the estimate provided by the SDPM in five real life software development projects.



## Chapter 3: Software Development Profile Model

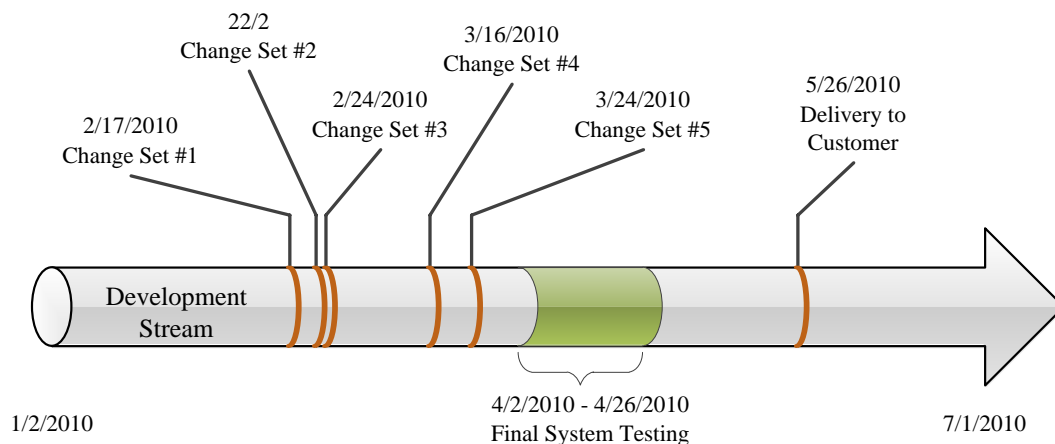
### 3.1 Proposed Work

Most large software systems are developed in phases over a long period of time and follow a specific software development lifecycle model. During software development, due to human error, defects are injected into the software. Some are identified and removed, while others go undetected. These “latent defects” are passed into the next software development phase and can be observed and reworked in the subsequent change sets. Before the software is released into production, it undergoes a period of final system testing and acceptance testing to ensure it meets all the customer requirements before it is released for production. Any defects that are not identified during Software Acceptance Testing (SAT) will be released into production and can cause software failure. We define the Software Development Profile (SDP) as all internal and external factors that affect the software product while it is being developed. While most software development projects follow a specific development model (Waterfall, Prototype, Agile, etc.) they nevertheless experience a unique software development profile during development. Software Development Profile should be considered to obtain more accurate defect count estimation.

### 3.2 Methodology

Software development processes rely heavily on human judgment and therefore cannot be completely automated. While the use of different Computer Aided Software Engineering (CASE) tools during software development has improved control and productivity in recent years, software development remains a very hands-

on activity. Since causes leading to software failures are all due to human error during implementation of software requirements overtime, software change history contains useful information about software defect content. Configuration Management (CM) tools are used to manage software changes in large scale software development projects. Therefore, the CM tool can be used to obtain information about software's change history. Software engineers are required to check out desired artifacts from the development stream before making any changes. A development stream is a database containing the chronology of all development activities [36]. After software changes are made and inspected, the artifacts are delivered back to the stream in the form of a change set. Fig. 1 shows an example of a software development stream. Since the content of a software development stream only changes when change sets are delivered, it is logical to divide the software development process into a number of successive intervals and model software development activities in each individual change set.



**Figure 3: Software Development Stream**

When a software construct (SLOC, module, function point, requirement statement, etc.) is created or modified during software development, there is a chance it is can be injected with a defect. Let's define:

$$z^c = \begin{cases} 1, & \text{if a construct did not become defective} \\ 0, & \text{if a construct became defective} \end{cases} \quad (1)$$

If we could identify all defective constructs, we could simply rework the defects and produce the perfect software. However, since we don't know which constructs were injected with a defect,  $Z^c$  is a random variable. We define  $r^c$  as the probability of  $\{z^c = 1\}$ . This means  $r^c$  is the probability that a given construct touched in the change set  $c$  did not become defective. The probability  $r^c$  would depend on the effectiveness of software development activities such as defect detection and removal and therefore may not be the same across all change sets. Other internal and external factors such as the developer's skill level, schedule pressure, size of the change, etc. can influence this probability as well. Let's formally define

$$r^c = \Pr\{z^c = 1\}; \quad \text{therefore } \Pr\{z^c = 0\} = 1 - r^c \quad (2)$$

as the reliability of change set  $c$ . In the remainder of this paper we use the term “*change set reliability*” as the probability that a given construct touched in a change set is defect free. Estimating the  $r^c$  based on software development activities is desirable and is discussed in the following section. Modeling change set reliabilities based on software development activities would allow software managers to make

risk-informed decisions and adjust software development activities to improve the reliability of their product.

### 3.3 Software Development Profile Matrix

Since change sets have different reliabilities and a given construct can be touched in different change sets, change set reliabilities need to be maintained. This data can be captured in the Software Development Profile Matrix  $X = (x_{i,c})$  where:

$$x_{i,c} = \begin{cases} r^c, & \text{if } i\text{-th construct was touched in Change set } c \\ 1, & \text{otherwise} \end{cases} \quad (3)$$

The size of the Software Development Profile Matrix is  $(n \times c)$  where  $n$  is the number of constructs present in the software stream when the  $c$ -th change set is delivered. As an example, the following matrix captures the software development profile of a software development stream containing nine constructs modified over nine change sets.

$$X_{i,c} = \begin{pmatrix} 1 & 1 & r^3 & r^4 & 1 & 1 & r^7 & 1 & 1 \\ r^1 & 1 & 1 & 1 & 1 & 1 & 1 & r^8 & 1 \\ 1 & r^2 & 1 & 1 & r^5 & 1 & 1 & 1 & r^9 \\ 1 & 1 & 1 & r^4 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & r^7 & 1 & r^9 \\ 1 & 1 & r^3 & r^4 & r^5 & 1 & 1 & 1 & r^9 \\ r^1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ r^1 & 1 & 1 & 1 & 1 & r^6 & r^7 & r^8 & 1 \\ 1 & 1 & 1 & r^4 & r^5 & 1 & 1 & 1 & r^9 \end{pmatrix} \quad (4)$$

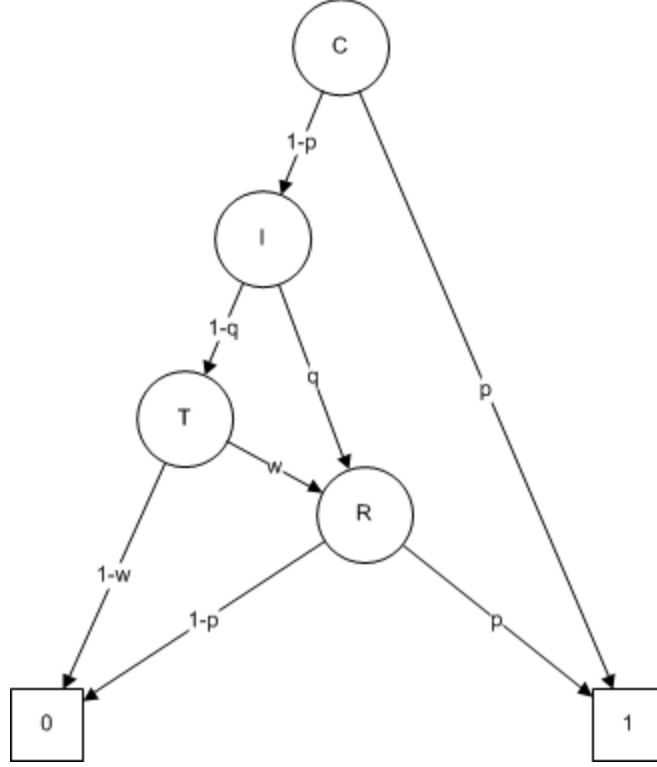
In the matrix above, columns represent change sets and rows correspond to specific constructs. Based on our assumption, when a construct is touched in a change set, it has a change to become defective. Since the probability of a given construct becoming defective is different in each change set, the value  $r^c$  is stored in the change matrix. If the construct is not modified in the change set, its reliability remains unchanged and its value is marked with a 1. Using the information stored in the Software Development Profile Matrix, we can then estimate the reliability of a given construct changing over multiple change sets. In this section we provide an approximation by assuming the change sets are independent. In section 3.6 we will improve this approximation by incorporating the dependencies between the change sets. If we assume that the change sets are independent (an assumption that will be removed in section 3.6), then the probability of a construct being defect free can be estimated by the following approximation:

$$R_i^c = \prod_{j=1}^c x_{i,j} \quad (5)$$

We define “*construct reliability*” as the probability of a given construct being defect free represented hereafter by  $R_i^c$ . Estimator (5) simply means that for a construct “i” to be defect free, it can’t be injected with a defect in any change set during which it was modified.

### 3.4 Estimating Change Set Reliabilities Using a Binary Decision Diagram

Previously, we discussed the notion that software development activities such as defect detection or removal activities can influence change set reliabilities. Software inspections or tests are examples of defect detection activities. In this section we will show how the effectiveness of such activities can be used to estimate the change set reliabilities. Software development activities during each change set can be modeled using a Binary Decision Diagram (BDD). BDD is a data structure that is used to represent Boolean functions and the relationship between them. Each decision node represents an activity that occurs in the change set. Example activities in a change set include software constructs being modified, inspected, integration tested, or reworked before delivery to the development stream. Figure 4 illustrates how these activities are modeled using a Binary Decision Diagram. In this example the node C represents the coding activity in the change set, the node I represents the code inspection activity, and node T represents testing. In this example, the rework activity, shown here as the node R, occurs after the inspection and testing is complete. The edges of the BDD represent the probability of success for each node. For instance,  $p$  in this example is the probability that a given construct that was modified did not become defective, while  $q$  is the probability that a defective construct is observed during the inspection process and  $w$  represents the probability that a defective construct, unnoticed during the inspection process, is observed during the testing phase.



**Figure 4: Software Development Binary Decision Diagram**

Based on the BDD representation of software development activities, the probability that a given construct does not become defective in change set  $c$  can be estimated by:

$$r^c = p^c + (1 - p^c) \cdot q^c \cdot p^c + (1 - p^c) \cdot (1 - q^c) \cdot w^c \cdot p^c$$

Where:

$\bar{p}^c = (1 - p^c)$  = defect injection probability

$q^c$  = probability of observing a defective construct during inspection

(detection probability during inspection)

$w^c$  = probability of observing a defective construct during I & T

(detection probability during testing)

We further assume that the defect injection probability during initial coding is the same as the defect injection probability during rework. This is not an unreasonable assumption, since the defects found during inspection or testing are generally corrected by the original author. To estimate  $r^c$ , we first need to calculate model parameters  $p^c, q^c$  and  $w^c$ . Let's define

$N^c$  as the number of constructs that become defective in change set c,

$S^c$  as the number of constructs touched in change set c,

$\bar{p}^c = (1 - p^c)$  as the injection probability.

If we assume that the constructs become defective independently, then the probability of  $N^c$  constructs becoming defective given  $S^c$  and  $\bar{p}^c$  can be described using the binomial distribution:

$$Pr(N^c | S^c, \bar{p}^c) = \binom{S^c}{N^c} (\bar{p}^c)^{N^c} \cdot (1 - \bar{p}^c)^{S^c - N^c} \quad (6)$$

The current state of knowledge about  $\bar{p}^c$  is unknown prior to inspection and testing activities. Bayesian theorem can be used to obtain an “updated” state of knowledge based on the number of defective constructs observed during inspection or testing. Let's use the Beta distribution to describe  $\bar{p}^c$ . Using Beta distribution to describe probabilities is a reasonable assumption because it has a flexible distribution between 0 and 1. The functional form of the Beta family of distributions is:

$$B(x^c | a^c, b^c) = \frac{\Gamma(a^c + b^c)}{\Gamma(a^c) \cdot \Gamma(b^c)} \cdot (x^c)^{a^c - 1} \cdot (1 - x^c)^{b^c - 1} \quad (7)$$



Where  $a^c$  and  $b^c$ , both non-negative, are parameters of the distribution and determine its shape. More specifically, parameters  $a=1$  and  $b=1$  describe no prior knowledge about  $\bar{p}^c$ . The binomial distribution in Eq. (6) and the evidence obtained from inspection activities can be used to update the state of our knowledge about  $\bar{p}^c$ . Using the Beta distribution shown in Eq. (8) with the conjugate likelihood given by Eq. (6) provide updates for the parameters  $a' = a + N$  and  $b' = b + (S - N)$  and the state of knowledge about  $\bar{p}^c$ .

$$Pr(\bar{p}^c | a^c, b^c) = \frac{[(S^c+2)]}{[(N^c+1) \cdot (S^c-N^c+1)]} \cdot (\bar{p}^c)^{N^c} \cdot (1 - \bar{p}^c)^{S^c-N^c} \quad (8)$$

Figure 5 shows the defect injection probability of project 1 with no prior knowledge ( $a=1, b=1$ ) and after the evidence was obtained during the inspection process.

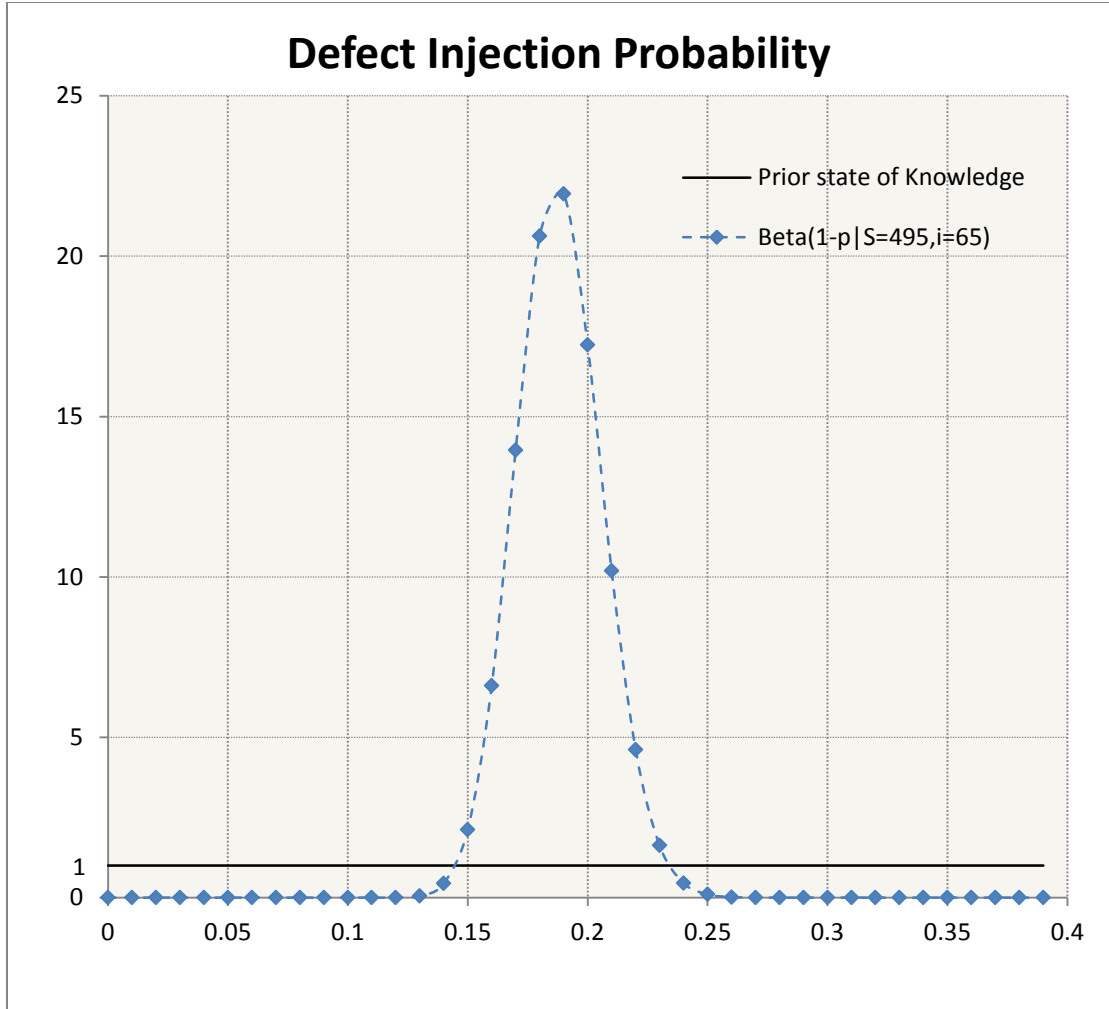


Figure 5: Defect Injection Probability of Project 1- Before and After Code Inspection

The posterior mean value for  $\bar{p}^c$  is

$$\bar{p}^c = \frac{a'}{(a' + b')} = \frac{(a + N)}{(a + b + S)}$$

For  $N \gg a$ , and  $S \gg b$  the posterior mean value for  $\bar{p}^c$  is simply:

$$\hat{p}^c = 1 - \frac{N^c}{S^c} \quad (9)$$

Bayesian theorem states that as more evidence becomes available, uncertainty decreases. The spread of the Beta distribution can be used to reflect our uncertainty about the value of the unknown  $\bar{p}^c$ . The coefficient of variation (CV) of the Beta distribution shown in Eq. (10) can be used to express our uncertainty of the state of knowledge.

$$CV = \sqrt{\frac{b}{a(a+b+1)}} \quad (10)$$

In this case, with  $a=1$  and  $b=1$ , the coefficient of variation is 0.5774, whereas in Figure 5 ( $S=495$  and  $N=95$ ), the coefficient of variation is reduced to 0.0916. The coefficient of variation will further be reduced if additional information becomes available through further testing of the changes (Figure 6).

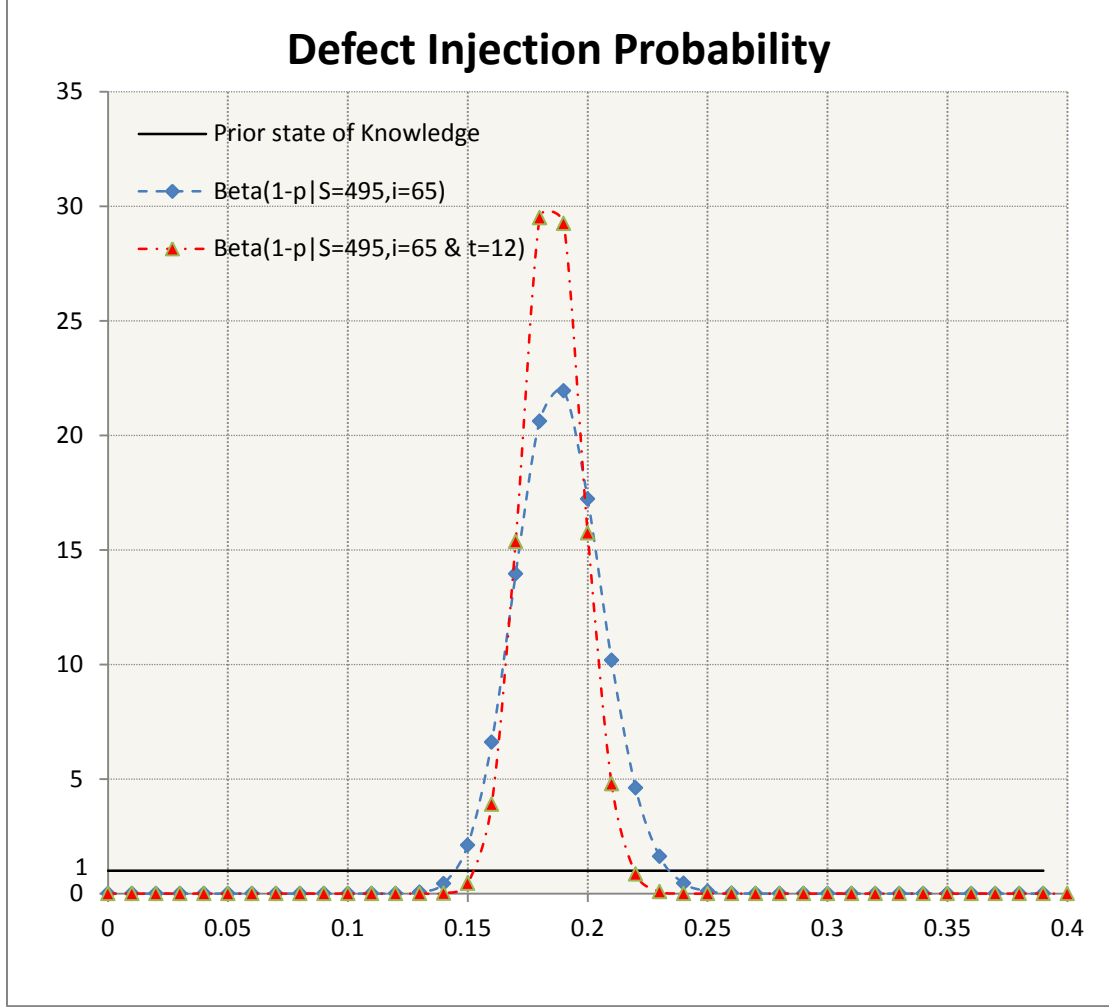


Figure 6: Defect Injection Probability of Project 1 Before and After Testing

In this case shown in Figure 6 ( $S=495$ ,  $i=65$ ,  $t=12$ ) the coefficient of variation is further reduced from 0.0916 to 0.0676 after the evidence from testing is available. Similarly, the detection probabilities  $q^c$  and  $w^c$  can be estimated using Bayesian inference. Let's define

$i^c$  = Number of defective constructs identified by the inspection of change set c

$t^c$  = Number of defective constructs identified during testing of change set c

If we assume that defective constructs are observed independently, then the probabilities of  $i^c$  and  $t^c$  can be described by the following binomial distributions:

$$\mathbf{Pr}(i^c | N^c, q^c) = \binom{N^c}{i^c} (q^c)^{i^c} \cdot (1 - q^c)^{N^c - i^c} \quad (11)$$

$$\mathbf{Pr}(t^c | (N - i)^c, w^c) = \binom{(N - i)^c}{t^c} (w^c)^{t^c} \cdot (1 - w^c)^{(N - i)^c - t^c} \quad (12)$$

Our state of knowledge on  $q^c$  and  $w^c$  can be described by Bayesian theorem using the Beta distribution as the posterior distribution. As more evidence becomes available through inspection and testing, our degree of knowledge increases and the uncertainty decreases. If we assume no prior knowledge ( $a=1$  and  $b=1$ ), using  $N^c, i^c$ , and  $t^c$  as the evidence, the updated degree of knowledge about the probability of observing a defect during inspection and testing can be written as:

$$\mathbf{Pr}(q^c | N^c, i^c) = \frac{\Gamma(N^c + 2)}{\Gamma(i^c + 1) \cdot \Gamma((N - i)^c + 1)} \cdot (q^c)^{i^c} \cdot (1 - q^c)^{N^c - i^c} \quad (13)$$

$$\mathbf{Pr}(w^c | (N - i)^c, t^c) = \frac{\Gamma((N - i)^c + 2)}{\Gamma(t^c + 1) \cdot \Gamma((N - i)^c - t^c + 1)} \cdot (w^c)^{t^c} \cdot (1 - w^c)^{(N - i)^c - t^c} \quad (14)$$

where  $D$  is the number of defective constructs observed during inspection and testing ( $D = i^c + t^c$ ). The posterior mean values and coefficient of variation of  $q$  and  $w$  are given below by Eq. (15) and eq. (16).

$$\hat{q}^c = \frac{i^c+1}{N^c+2}; \quad CV = \sqrt{\frac{[(N-i)^c+1]}{(i^c+1)(N^c+3)}} \quad (15)$$

$$\hat{w}^c = \frac{t^c+1}{(N-i)^c+2}; \quad CV = \sqrt{\frac{[(N-D)^b+1]}{(t^b+1)((N-i)^b+3)}} \quad (16)$$

Using posterior mean values of the parameters provided in Eq. (9), Eq. (15) and Eq. (16), we can estimate the reliability of change set c as:

$$\hat{r}^c = Pr[\mathbf{construct} = 1] = \left(1 - \frac{N^c}{S^c}\right) \left(1 + \frac{i^c+t^c}{S^c}\right) \quad (17)$$

If we define  $D^b = (i^b + t^b)$  as the number of defective constructs observed during testing and inspection, then the reliability of change set c can be rewritten as:

$$\hat{r}^c = Pr[\mathbf{construct} = 1] = \left(1 - \frac{N^c}{S^c}\right) \left(1 + \frac{D^c}{S^c}\right) \quad (18)$$

The above equation (18) describes the relationship between the reliability of the change set c, size of the change set, and software development activities during change set c. Based on Eq. (18), in case of perfect coding where no defects are injected ( $N=i=t=0$ ), the change set reliability is 1. However, in case of imperfect coding, even if all defective constructs are observed during testing and inspection ( $N=D=i+t$ ), the reliability of the change set will be  $1 - \left(\frac{N^c}{S^c}\right)^2$ , which is less than 1, due to the defect injection probability during rework.

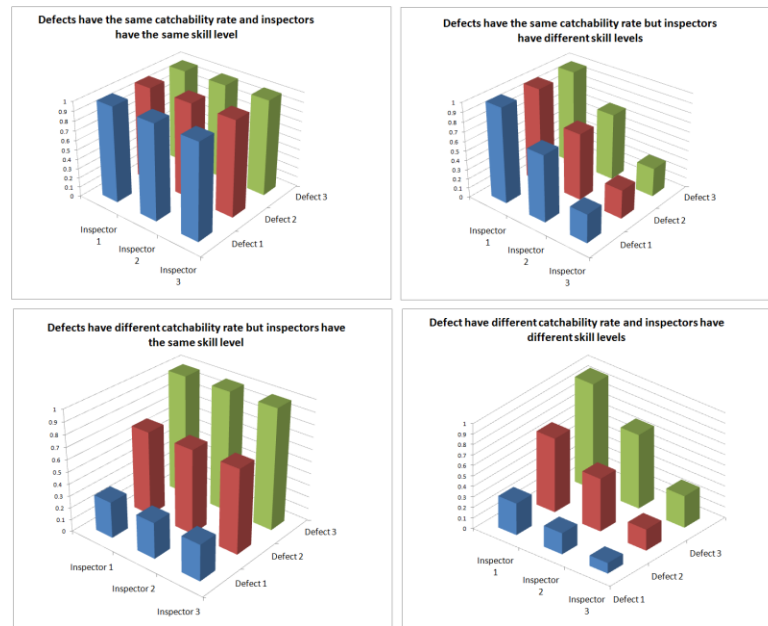
In this section we used the size of change set  $c$  ( $S^c$ ) and the number of defective constructs observed during defect detection activities to estimate model parameters  $p^c, q^c$ , and  $w^c$ . While  $S^c$ ,  $i^c$  and  $t^c$  are values that can be directly obtained from the configuration management tool and the inspection and testing process, the total number of constructs that become defective in each change set, ( $N^c$ ), needs to be estimated. Section 3.5 describes how to estimate this number during change set  $c$  using Capture-Recapture method.

### **3.5 Estimating Total Number of Defective Constructs in Change Sets**

In this chapter we discuss how the capture-recapture method can be used to estimate the number of defective constructs in a change set. Several studies in software engineering have considered the use of capture-recapture models for estimating the number of defects in an inspection package. Originally proposed by biologists to estimate animal populations, different variations of capture-recapture have been employed to estimate the defect content in an inspection package. Inspection is a formal, rigorous and in-depth technical review designed to identify problems as close to their point of origin as possible [56]. Inspection was first described by Fagan in 1976, and since then inspections have been established as state of the practice and have evolved to become a mature empirical research area [55][5]. A number of authors have studied the robustness of various capture-recapture techniques have been researched [11], [41], [54], [58], [62]. Capture-recapture uses the overlap between the findings observed among different inspectors to estimate the total fault content in an inspection package. If the overlap between the findings observed by inspectors is

large, it is assumed that few defects remain, and if the overlap is small, then many more defects are assumed undetected.

Different variations of the capture-recapture method have been proposed in the literature. The simplest version assumes that all defects have the same probability of being observed and all inspectors have the same skill level. This assumption is not very realistic for the software inspection process, since inspectors generally have different skill levels, and defects have different probability of being detected. The most realistic version of the capture-recapture model takes this into account, assuming defects have different probabilities of being found and inspectors' different skill levels. Four different capture-recapture assumptions are graphically illustrated in Figure 7 [11]. To estimate the total number of defective constructs in a change set, we use the capture-recapture method proposed by Chao [16].



**Figure 7: Capture-Recapture Model Assumptions**



To implement capture-recapture in the inspection process several requirements have to be met. These necessary assumptions are as follows:

- The defect population must be closed. This means that while the inspection is ongoing, no further defects are injected into the artifacts. This requirement is easily met because no modification is made to the artifact while the inspection is in progress.
- Each inspector will receive the same material. This assumption is also realistic, because the moderator prepares the inspection material and submits the same material to all inspectors for review.
- Each inspector reviews the material independently. This, too is realistic, because each inspector is given sufficient time to review the material prior to the formal inspection meeting.
- Inspectors do not discuss or share their findings until everyone has submitted his findings. It is very critical to the successful capture-recapture model that this requirement is met. As described above, CR uses the overlap between defects found by different inspectors. Sharing defect information will increase the overlap and therefore result in an underestimation of defect data.
- Inspectors must keep accurate data of their findings.
- The moderator needs to use the independent findings and estimate the number of constructs that remain defective.
- Due to the nature of the software and dependencies among artifacts, constructs can become defective as the result of a change in a related artifact. To capture such defects, it is assumed that the inspection package includes not only

artifacts that were modified, but also all related artifacts. While there are various methods for modeling internal dependencies among modules, we found it sufficient to capture this dependency by inspecting related artifacts during the inspection process.

Table 1 shows a sample inspection worksheet that can be used to estimate the number of defective constructs in an inspection package. Columns e1-e4 represent the inspection findings for each inspector.

**Table 1: SDPM Sample Inspection Worksheet**

Defect Description	Detection Probability	Inspectors				$f_i$
		e1	e2	e3	e4	
Module1 line 243	p1	1	1	1	1	4
Module1 line 622	p2	1	1	0	1	3
Module2 line 41	p3	1	1	0	1	3
Module13 line 24	p4	0	1	0	0	1
Module 21 line 2	p5	1	1	0	1	3
Module 34 line 1233	p6	0	1	0	0	1
	$n_j =$	4	6	1	4	15

The format inspection process is illustrated in Figure 8. SDPM is used after the formal inspection meeting to estimate the build reliability. This worksheet is used during the inspection of the case studies provided in Chapter 4.

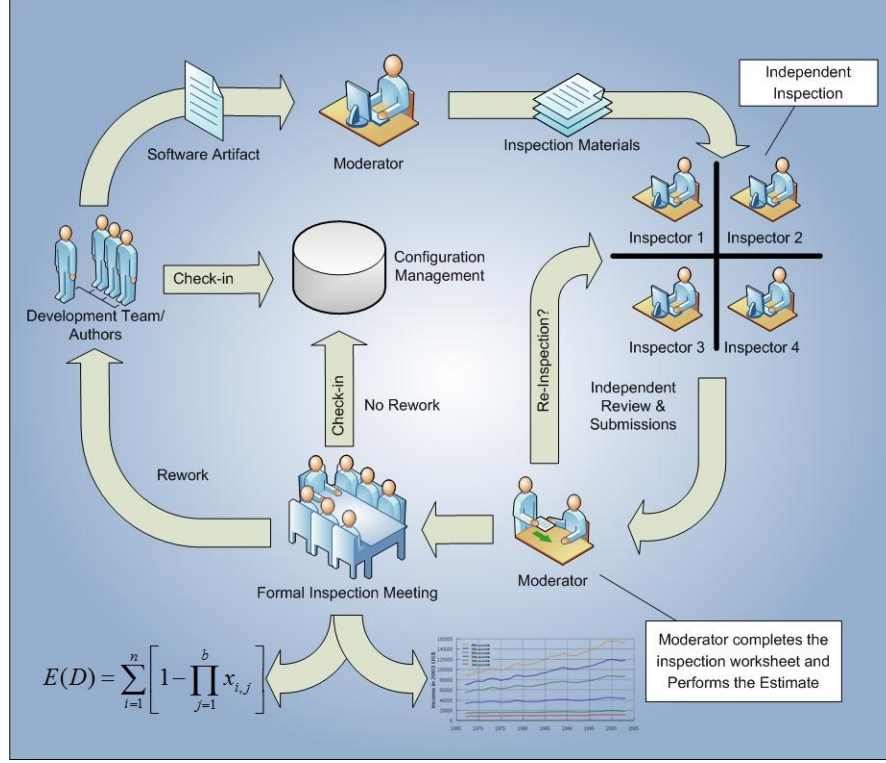


Figure 8: SDPM within the Formal Inspection Process

### 3.5.1 Chao's Heterogeneity-Time Model

In this section we will describe how Chao's Heterogeneity-Time model is used to estimate the number of defective constructs  $N^c$  in a change set [16]. Chao's model makes the following assumptions:

$N^c$ : True but unknown number of defective constructs in a change set  $c$

$t^c$ : Number of inspectors inspecting change set  $c$

$p_i^c$ : Unknown detection probability of  $i$ -th defective constructs in change set  $c$

$P^c = (p_1^c, p_2^c, \dots, p_N^c)$  has sample mean  $\tilde{p} = \frac{\sum_{i=1}^{N^c} p_i^c}{N^c}$

$e_j^c$ : Unknown skill level of the  $j$ -th participating inspector

$p_{ij}^c$ : The detection probability of the  $i$ -th defective construct by the  $j$ -th inspector reviewing change set  $c$

$C^c$ : Inspection coverage of change set  $c$

$n_j^c$ : Number of defective constructs observed by the  $j$ -th inspector inspecting change set  $c$

$f_k^c$ : Number of defective constructs identified exactly by  $k$  inspectors in change set  $c$

$D^c$ : The number of distinct defective constructs observed during the inspection of change set  $c$

Chao [16] formulated an estimator that allows the probability to vary with heterogeneity and time. Let us assume that the number of constructs that become defective in change set  $c$  is  $N^c$ , and there are  $t$  inspectors participating in the inspection process. Let's also assume that defects are indexed  $1, 2, \dots, N^c$  and  $p_{ij}^c$  is the detection probability of the  $i$ -th defective construct observed by the  $j$ -th inspector. Chao in [16] assumes that  $p_{ij}^c = p_i^c e_j^c$ , and  $0 < p_i^c e_j^c \leq 1$  for  $i = 1, 2, \dots, N^c$  and  $j = 1, 2, \dots, t^c$ . Unlike previous authors that assume  $(p_1^b, p_2^b, \dots, p_N^b)$  and  $(e_1^b, e_2^b, \dots, e_t^b)$  are random samples from an unknown distribution, Chao [16] treats them as fixed parameters.

In previous section we developed a worksheet to capture the inspection findings per inspector during each inspection. Table 1 captures this information in the form of an  $N \times t$  matrix  $X = (X_{ij})$  where:

$$X_{ij} = I[\text{the } i - \text{th defect is detected by the } j - \text{th inspector}]$$

Where  $I[A]$  is 1 if event A occurs and 0 otherwise. If we assume that there is no interaction between the inspectors and each inspector reviews the materials independently, then the number of distinct defects observed during the inspection can be written as:

$$D^c = \sum_{i=1}^N I[\sum_{j=1}^t X_{ij}^c \geq 1] \quad (19)$$

And the number of defective constructs observed exactly by k inspectors can be written as:

$$f_k^c = \sum_{i=1}^N I[\sum_{j=1}^t X_{ij}^c = k], k = 0, 1, \dots, t \quad (20)$$

It is obvious that only  $D^c$  defective constructs are detected and  $f_0^c$  represents the number of defective constructs that are not observed. The total number of defective constructs in change set c is  $N^c = f_0^c + D^c$ . The sample coverage  $C^c$  is defined as the proportion of detection probabilities of the observed constructs over all defect detection probabilities:

$$C^c = \frac{\sum_{i=1}^N p_i^c \cdot I[\text{the } i - \text{th defective construct is detected}]}{\sum_{i=1}^N p_i^c}$$

If all  $p_i^c$  are equal, then  $C^c = \frac{D^c}{N^c}$ . In that case the estimator for the number of defective constructs in change set c would be

$$\hat{N}^c = \frac{D^c}{C^c} \quad (21)$$

Chao [16] provides the following three estimators for sample coverage  $C^c$  when  $p_i^c$  are unequal:

$$\begin{aligned}\hat{c}_1^c &= 1 - \frac{f_1^c}{\sum_{k=1}^t k f_k^c} \\ \hat{c}_2^c &= 1 - \frac{f_1^c - \frac{2f_2^c}{(t-1)}}{\sum_{k=1}^t k f_k^c} \\ \hat{c}_3^c &= 1 - \frac{f_1^c - \frac{2f_2^c}{(t-1)} + \frac{6f_3^c}{(t-1)(t-2)}}{\sum_{k=1}^t k f_k^c}\end{aligned}$$

$\hat{C}_2$  and  $\hat{C}_3$  are bias-corrected versions of defect coverage  $\hat{C}_1$ . Based on the coverage factors, the estimated number of defective constructs in change set c can be estimated by

$$\hat{N}_m^c = \frac{D^c}{\hat{C}_m^c} + \frac{f_1^c}{\hat{C}_m^c} \gamma_m^2 \text{ for } m = 1, 2, 3 \quad (22)$$

Where

$$\gamma_m^2 = \max \left\{ \frac{\frac{D^c}{\hat{C}_m^c} \sum_{i=1}^k k(k-1) f_k^c}{2 \sum_{k=2}^t \sum_{j=1}^{k-1} n_j n_i} - 1, 0 \right\} \quad (23)$$

$\gamma_m^2$  is the coefficient of variation. When  $\gamma_m^2$  is relatively small, then the number of defective constructs,  $\hat{N}_m^c$ , can be estimated by

$$\hat{N}_m^c \approx \frac{D^c}{\hat{C}_m^c} \quad (24)$$

### 3.6 Modeling Dependencies

Dependency among components is an important factor when it comes to quantifying risk, reliability and safety models. Generally, there are two approaches to incorporating dependencies in a probabilistic model. The first approach is the explicit modeling approach, where we define the sources of dependencies, such as internal, external, design, human interaction, environmental, etc. and include them in the overall physical model of the system. The second approach is the implicit modeling approach. In an implicit dependency modeling approach we try to cover the probabilistic impact of dependencies on the overall risk or reliability of the system without modeling the detailed mechanism of the interdependencies.

In the SDPM, we recognize two types of dependencies: intrinsic and extrinsic dependencies. Intrinsic dependencies are those in which functional status of one construct affects the functional status of another. Such dependencies generally stem from the way the software is designed. This type of dependency is important because the modification of one construct can cause related constructs to become defective. Intrinsic dependencies primarily exist between software constructs in related artifacts and numerous models have been proposed to capture such dependencies [21], [51]. Modeling dependencies among constructs can improve the estimation of defect injection probability ( $p$ ) by including the probability of constructs that can become defective even if they are not modified in a change set. In the SDPM, we estimate this probability based on the total number of defective constructs obtained by capture-

recapture method. Since such dependencies mostly exist among constructs within related software artifacts, we were able to estimate them indirectly by including related artifacts in the inspection process.

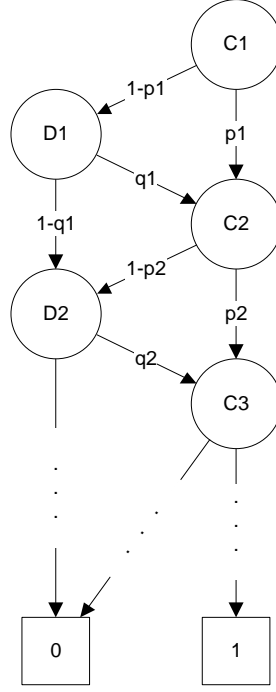
Another type of dependency is the dependency between change sets. Since software constructs can be modified in multiple change sets, internal dependencies exist among change sets. In Section 3.4, we assumed that change sets are independent and, based on this assumption, we provided an approximation for estimating the reliability of constructs that were touched in more than one change set. We assumed change sets were independent such that defects injected in one change set could only be observed and reworked during the same change set. By making this assumption, we estimated the reliability of the constructs as the product of change set reliabilities during which they were modified. In reality, however, defective constructs can be observed during the inspection or testing of any subsequent change sets. In Section 3.6.1 we improve upon the approximation by including the future detection probabilities in the estimation.

Extrinsic dependencies, on the other hand, are those in which the coupling mechanism is not inherent in the design of the software. Such dependencies are often external to the software product. Dependencies due to common environmental factors, such as overall schedule pressure, maturity level of the organization, skill level of the development team, or lack of management oversight belong to this category. In Section 3.6.2 we discuss how Bayesian Belief Networks (BBNs) can be used to capture external dependencies and incorporate the state of our knowledge to update and improve SDPM parameter estimates.



### 3.6.1 Modeling Dependencies among Change Sets

In Section 3.4, we assumed that software development activities were independent from each other, and software constructs that become defective in one change set can only be detected only during the next inspection or testing activity. This is an unrealistic assumption, as defective constructs can be observed and repaired during the inspection or testing of future change sets. In this chapter we improve our approximation by removing the independency assumption. However, to remove the independency assumption we need to add two new assumptions. The first assumption is that a defective construct must be observed before it is reworked. The second assumption is that when a defective construct is observed, it is reworked in the immediately next change set. Both these assumptions are in general reasonable, because latent defects are more likely to be observed during testing or inspection than unit testing. Furthermore, latent defects are generally reworked as soon as they are discovered. The exceptions are fixes that are either too complex or require input from a customer or third party. In any case, these assumptions are more reasonable than our initial independency assumption. Figure 9 shows a Binary Decision Diagram (BDD) used to model the probability of constructs that are touched in multiple change sets.



**Figure 9: Modeling Dependencies among Change Sets**

In this BDD, the nodes marked with C represent the coding activity and those marked with D represent defect discovery activities such as inspection or testing. Based on the BDD shown in Figure 9, a construct can be modified in multiple change sets either by the implementation of a new functionality, or as the result of the rework of an observed defect. Based on the BDD above, the probability of a given construct that is modified in two change sets  $i$  and  $j$  can be estimated by:

$$R^{i,j} = p^i \cdot p^j + (1 - p^i) \cdot q^{j-1} \cdot p^j \quad (25)$$

This means that the probability of a construct modified in two change sets being correct, is the probability of the construct being correct in both change sets,

represented by  $p^i \cdot p^j$ . Otherwise, if it became defective in the first change set, it must be observed during the defect discovery activity of the (j-1)th change set and be correctly modified in change set j,  $(1 - p^i) \cdot q^{j-1} \cdot p^j$ . This process can be written recursively as:

$$R_i^c = \begin{cases} 1, & \text{if the construct is not touched in } c \\ R_i^{c-1} \cdot (1 - q^{c-1}) \cdot p^c + q^{c-1} \cdot p^c, & \text{Otherwise} \end{cases} \quad (26)$$

### 3.6.2 Updating Model Parameters using Bayesian Belief Network (BBN)

As discussed earlier, causal models can provide more accurate predictions by allowing evidence and expert judgment to be taken into account when estimating model parameters. Rather than relying only on structural software measurements and historical data, the Software Development Profile Model can be used in conjunction with Bayesian Belief Networks to make inferences about the uncertain states of model parameters when limited information is available. BBNs can also be used to incorporate specification of probabilistic dependencies between variables and factors that have widespread influences. In general, there are two types of dependencies among change sets that need to be considered when updating model parameters. The first type of dependency is the dependency on factors that affect the overall software development project, such as process quality, overall staff quality, requirements and specification quality or test process quality. These factors, for example, impact all change sets and their affect should be captured to improve model parameter estimation when such information becomes available. On the other hand, there are

factors that do not affect the entire project, only individual change sets, such as the level of testing effort during one specific change set, resource availability, or current schedule pressure. While these factors do not affect the entire software development project, when available they can be used to update model parameters. Unlike existing regression models that are inadequate at capturing such dependencies, the SDPM can be used in conjunction with Bayesian Belief Networks to capture this information and provide a more accurate prediction.

A Bayesian Belief Network (BBN) is a directed acyclic graph (DAG) with nodes representing random variables, each with associated probability tables. An arrow from one node to another represents probabilistic influence. Figure 10 shows how a Bayesian Belief Network can be used in conjunction with SDPM to update model parameters. In this example we selected factors that affect all change sets (shown in blue), as well as factors that affect only individual change sets (shown in orange). In this model, the variables Test Process Quality (TPQ), Development Process Quality (DPQ), Staff Quality (SQ), and Requirements & Specification Quality (RSQ) are factors that are common to all change sets. As the names of the variables indicate, these are generally process, organizational or program level qualities that affect the entire project.

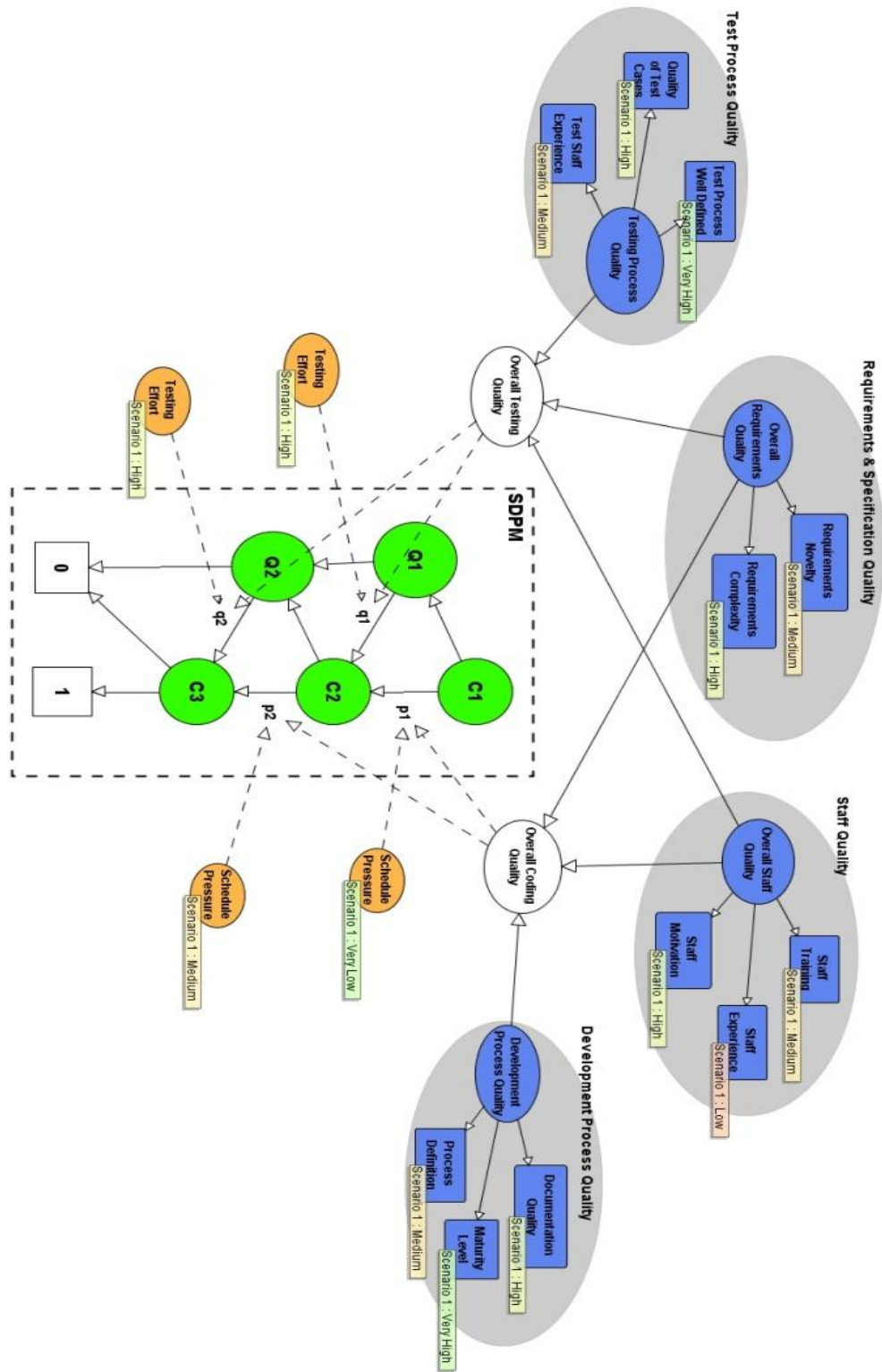


Figure 10: Example of Bayesian Belief Network Used in Conjunction with SDPM

Consider the example shown in Figure 10. In the example above, overall test process quality depends on test staff experience, quality of test cases, and how well the test process is defined. The influence level of the process quality indicators are judged by experts and are assigned numerical values between 0 and 1. Similarly the testing process quality and the testing effort influence the effectiveness of the process. Again, the relative level of influence of these two factors can be assessed by an expert. Once the relationships between the variables are defined, the BBN can be used to update Software Development Profile Model parameters ( $p$ ,  $q$ , and  $w$ ). In the example above we show how the probability of observing a defect in test can be updated. This is especially useful when objective evidence is lacking.

### **3.7 Properties and of Software Development Profile Model**

Modeling software development using the Software Development Profile Model provides some unique advantages

#### **1. Flexibility**

The proposed Software Development Profile Model is not dependent on a particular type of software artifact or unit of measurement. Software systems consist of executable and non-executable files but models based on observed defects fail to identify defects in non-executable files. Since the SDPM uses capture-recapture during inspection to estimate the number of defective constructs, it can be used successfully on executable and non-executable files alike, including configuration files, system documentation, user documentation, and other artifacts.

#### **2. Scalability**

The proposed Software Development Profile Model can be applied to the entire software solution or any subset of the system that might be of interest. It is often necessary to make a statement only about the defect content of a subset of the system. This becomes important with reuse-based software development, COTS integration, and partial exclusions such as auto-generated code.

### 3. Measurability

The proposed Software Development Profile Model provides a method for estimating the number of defective constructs in a software artifact. The estimator provided in (9) can be used to estimate the number of defective constructs in a given module.

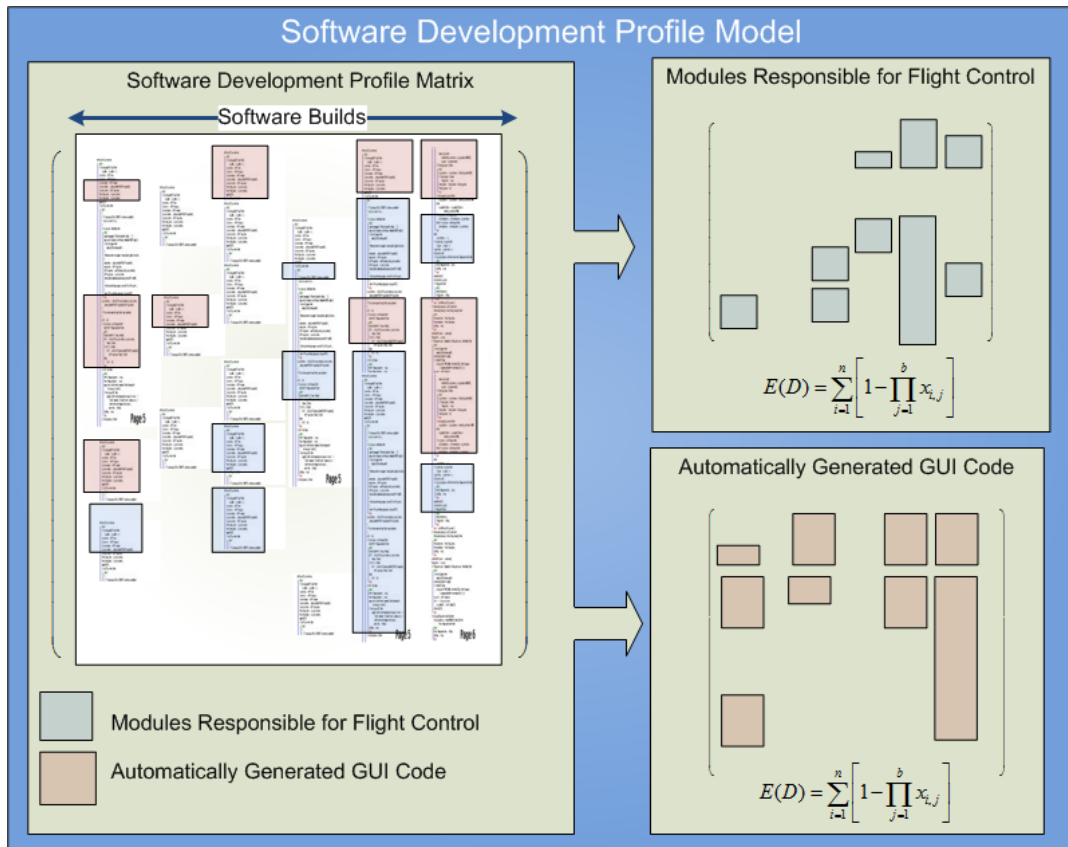
$$E^c(M) = \sum_{i=1}^n [1 - R_i^c] \quad (27)$$

Where n is the total number of constructs in the module M during change set c.

### 3.8 Software Development Profile

We formally define Software Development profile (SDP) as the listing of all software constructs in the software development stream after the change set c is delivered, together with their reliabilities  $R_i^c$ .

$$SDP^c = \{(i, R_i^c), \forall i \in \text{stream}\} \quad (28)$$



**Figure 11: Software Development Profile Model - Scalability**



## Chapter 4: Case Studies

In this chapter we present five case studies of software development projects that the author was personally involved in from 2007 – 2011. These are real projects with real customers and deliveries. We used these case studies to showcase how effective SDPM can be in relevant software development projects and how well it predicts software defect content. The purpose of presenting these case studies is not only to assess the accuracy of the SDPM's predictions, but also to investigate the usability of the SDPM in real life industrial projects. This chapter is divided into five sections, each detailing one case study. We will also discuss regression based defect estimation methods and compare the case study results with the negative binomial regression model.

We will first provide a brief background for each software development project and then describe step-by-step how measurements are taken and model parameters are estimated. Using the model parameters, we will then estimate the defect content of the files and identify those files that are most likely defect-prone. Finally, we will compare the predicted results with the actual defects observed during the final system and acceptance testing. We will use the coefficient of correlation to compare the SPDM results with the existing regression based defect estimation methods. To reduce the placebo effect and to prevent files from being treated differently, the development and test team members were not informed of the intent or the prediction results of the case studies until the end of all five projects. The predicted results were kept unpublished during final system and acceptance testing to allow the test team to perform their final system testing without bias or special attention to any identified

defect-prone modules. The test cases for the final system testing were developed based on the overall system requirements covering not only the software changes but also overall system functionality. On the other Software Integration (SWIT) test cases and System Integration (I&T) test cases were developed based on the requirements targeting only modified software functionalities.

In response to usability analysis, we noticed that the measurements needed for estimating SPDM model parameters were already being collected by the program with the exception of the number of constructs that remain defective after each inspection. To capture this information we used Chao's estimator to estimate the number of defective constructs in each change set as described in Section 3.5. The inspectors were asked to review the inspection artifacts independently and document their findings prior to the formal inspection meeting. While proper inspection processes requires inspectors to review the inspection package independently prior to the formal inspection meeting, the inspectors were not required to document findings at the construct level. To reduce the impact of this extra effort on the development team, the inspectors were asked to submit their findings to the moderator via email prior to the meeting, and the moderator himself performed Chao's estimation [16]; thus the SDPM had no significant impact on the development team. The time that the moderator needed to perform the analysis and perform the estimation was between one to three hours per inspection, depending on the number of issues observed during the inspection. This number was an increase of less than 10% in the total inspection time.

To make measurements consistent across all projects, we developed the Software Development Profile Estimation Tool (PET) and several Perl scripts that we used on all projects. We also used COTS products such as Microsoft Excel to perform calculations related to the reliability estimation

#### **4.1 Comparing Test Case Results with Existing Models**

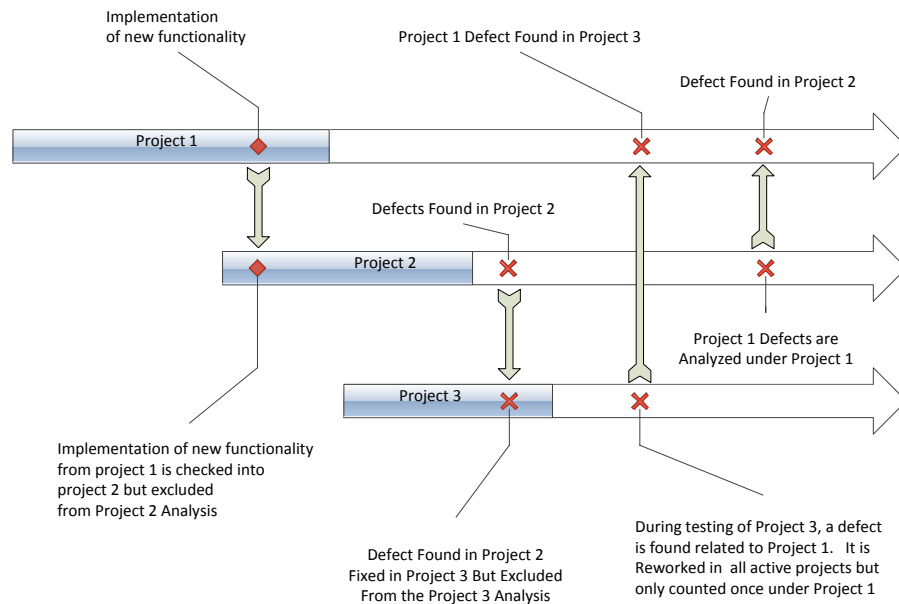
We are also interested in comparing the SDPM estimates with existing defect estimation models. Due to the large number of defect estimation models that have been proposed over the years, a comparison among all models is unrealistic and outside the scope of our current research. However, since we had access to extensive software defect data, going back over 40 releases, it made sense to compare SDPM with a regression based model. The main idea behind this comparison is to illustrate that, even with long historical software defect data, software development activities from current project can provide a better future software defect estimate. It is important to mention, that because of differences between the two models, a direct comparison between the SDPM and other models is not possible. First, there is a difference between the units of measurement among the two models. While the SDPM provides an estimate for the number of defective constructs, existing models are based on the number of defects per file. Second, regression based models are based on the defects observed during testing and operation and are unable to identify defects in non-executable files. But perhaps the main difference between the two models is that the SDPM is a causal model for estimating the number of defective constructs based on the development activities and the software changes in a specific

project. When evidence on software development activities or change history is unavailable, the SDPM assumes no defects have been introduced.

We selected five independent software development projects from the same software system to evaluate the results of the SDPM and the regression based model. Since all five projects shared the same history, files and operational profile, the structural software measurements used in the regression models will be similar among all five projects. File age, file size, change history, and the number of previously observed defects are some examples of variables used for the regression models. The idea behind this selection is that given a common history, it is expected that regression models would estimate similar defect prediction for a given file across all five projects.

Selecting five projects within the same product presented its own challenges. While all software development projects were developed by different teams, we had to excluding unrelated code changes during the analysis of each project. There were two main reasons for excluding unrelated code changes. First, when defects were observed in one project they were resolved in all active software development streams. Since these defects were included in the analysis of each case study, counting them more than once would make the analysis invalid. Also, when defects were identified during final system testing, they were assigned to the project to which the defect belonged. Once a fix was identified, it was fixed in the software stream that it was introduced to and then delivered to all parallel streams. Next, implementation of new functionalities had to be delivered to all streams with future release dates. This is a common practice to ensure future software releases have all

the functionalities that previous releases had already implemented. However, to keep the projects independent we did not count the changes due to implementation of new functionality in parallel development stream. Figure 12 shows examples of defects and new functionality changes that were counted once in various projects.



**Figure 12: Defects are counted only once in the stream they were injected**

To exclude unrelated code changes and defects, the script developed (shown in Appendix A) to capture software change history was modified to count any change set with multiple deliveries only once. The logic behind the script was simple. Any change set with multiple deliveries was excluded from the analysis if it had been delivered to a previous stream.

#### 4.1.1 Poisson Regression Model Setup

In addition to the data that we collected for our case studies, we also collected data for a sequence of 40 previous releases in order to compare our case study results with the Poisson regression model. Poisson regression model extends linear regression in order to handle positive outcomes such as the number of defects. For outcomes such as the number of defects per file, which is a non-negative number, it is unrealistic to assume that the expected value is an additive function of the explanatory variables [53]. The explanatory variables were selected similar to the negative binomial regression model proposed by Robert Bell et al. [53]. The main advantage of negative binomial regression is that fits data that is over-dispersed, which is normally observed with software defects. SAS provides a feature to correct for over-dispersion called the Pearson adjustment. In SAS JMP, we enabled the Over-dispersion Test and Intervals feature to fit the data using Poisson distribution. We used SAS JMP version 8.0.1 in our case study. We used data from over 40 previous to predict which files are most likely to be defective in the next release.

Suppose that we want to make predictions for release 40. In that case, we build our model using data from releases 1 to 39 based on observations in the regression for each combination of file and release in which the file existed. To give an example, suppose that File A was added to the system at release 3 and remained in the system beyond release 40. File A would contribute thirty-eight observations to the regression, one for each release from 3 to 40. Some predictor variables would remain constant across these observations (notably, Programming Language), while others might change (e.g., SLOC or PriorFaults). Additional predictor variables are New,

Changed, Unchanged, and Age. In this example, for Release 3, File A would have New=1, Age = 0, and PriorFaults=0, Changed=1, Unchanged=0. For later releases, we would have New=0, but Age and PriorFaults greater or equal 0. We define the Age of a file as the number of previous releases the file featured in, so Age=0 is the same as New=1. Similar to the Negative Binomial Regression Model proposed by Bell et al. [53], we take square roots of prior defects and logarithm of SLOC to reduce skewness of those predictors and improve the fit.

For the regression model, we assume that the number of observed defects in each file has a Poisson distribution and that its mean  $\mu_i$ , is related to the factors used as predictor variables. A log linear relationship between the mean and the factors is specified by the log link function. The log link function ensures that the mean number of observed defects predicted from the fitted model is positive. Mathematically we write this relationship as:

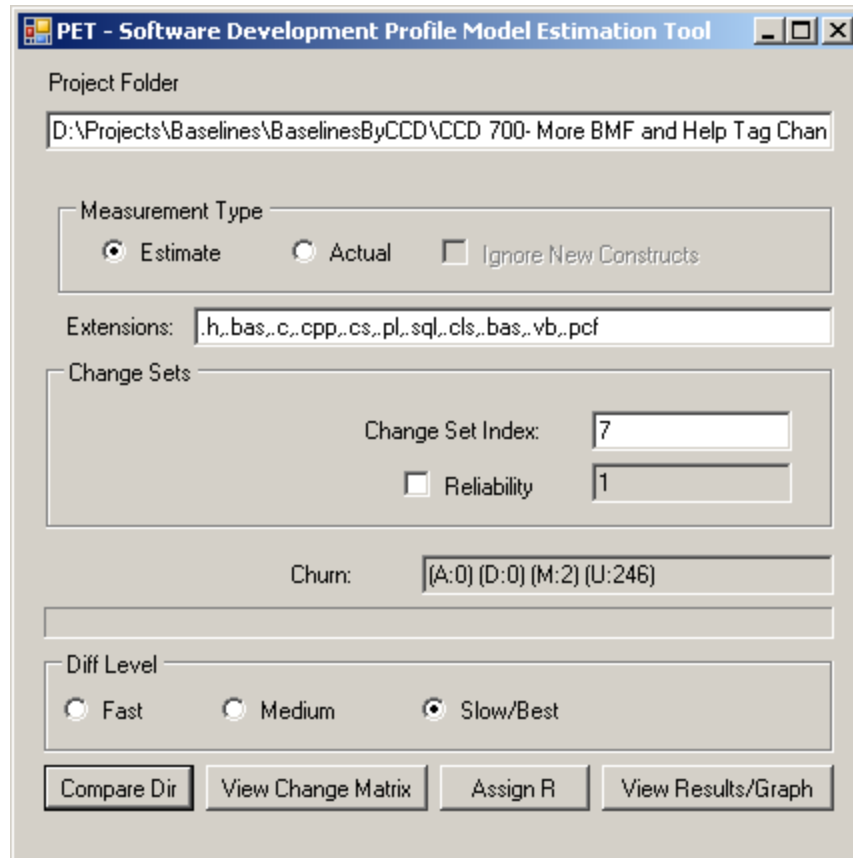
$$\log(\mu_i) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \dots + \beta_n x_n$$

The  $\beta_i$  are the regression coefficients, and the  $x_i$  are the predictor variables. Given this setup, we estimate the mean value of the number of defects by:

$$\mu_i = (e^{\beta_0})(e^{\beta_1 x_1})(e^{\beta_2 x_2}) \dots (e^{\beta_n x_n})$$

## 4.2 Software Development Profile Estimation Tool (PET)

In order to perform the estimation consistently across all projects, we developed the Software Development Profile Estimation Tool (PET) shown in Figure 13.



**Figure 13: PET – SDPM Profile Estimation Tool**

Using the PET tool, we are able to analyze software change history, generate software change matrices, assign reliability factors to software constructs, and ultimately estimate the number of defective constructs consistently across all projects. The process has two steps. We first run the script described in Appendix A on each development stream to generate a directory structure that contains software activities unique to each project. The Perl script extracts software activities automatically from the CM tool. The output of the script is used by the PET tool for further analysis.

In the PET tool we first select the location containing the directory structure created by the Perl script. The tool compares the content of each change set and assigns a



change set ‘CS’ code to each construct, based on the change set during which it is modified. Since constructs can be modified in more than one change set, we use the following binary convention to capture this information:

$$CS = \sum_{i=1}^n I(i) * 2^i \quad (29)$$

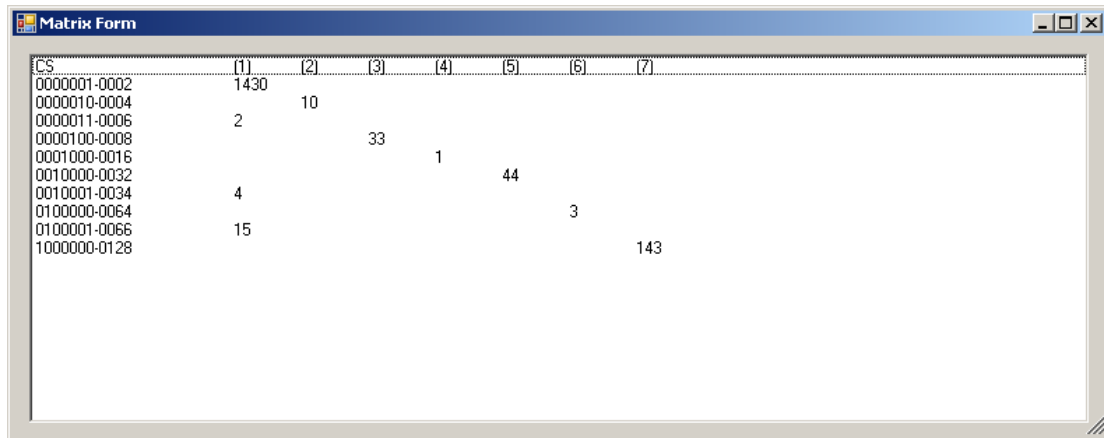
Where  $i$  is the change set number and  $I(i)$  is the usual indicator function defined as:

$$I(i) = \begin{cases} 1, & \text{if the construct was touched in change set } i \\ 0, & \text{otherwise} \end{cases}$$

**Table 2: Examples of CS Codes**

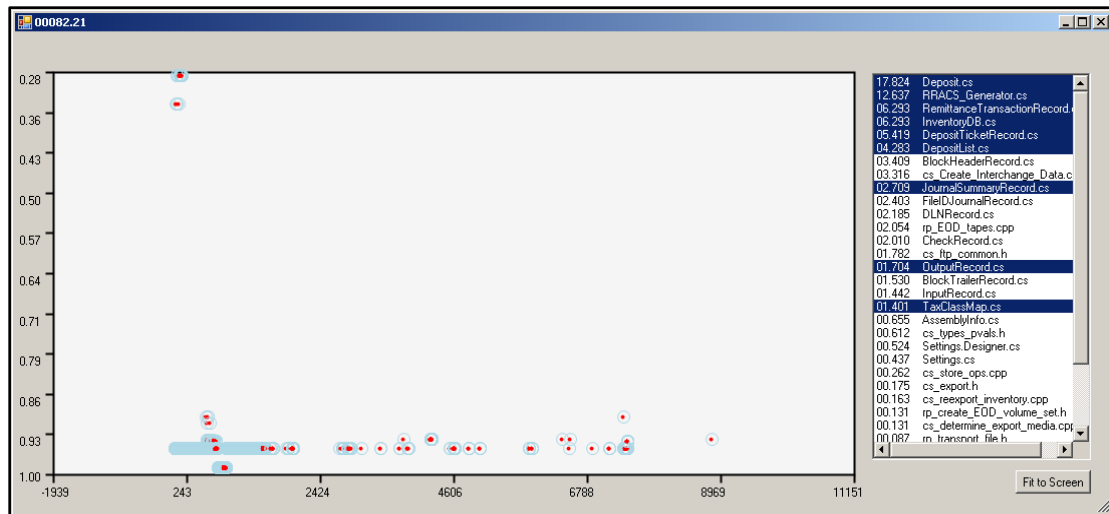
Binary Encoding	CS Code	Touched in Change sets
$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1$	2	1
$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1$	4	2
$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$	6	1 and 2
$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$	14	1, 2 and 3
$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$	30	1 through 4
$0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$	62	1 through 5
$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$	254	1 through 7

Table 2 shows some examples of the change set ‘CS’ codes. PET tool uses the CS-code to generate the Change Matrix shown below.



**Figure 14: PET - Change Matrix**

The information from the Change History Matrix is used to estimate the model parameters. While model parameter and construct reliabilities are calculated using Microsoft Excel, the PET tool is used to assign the probabilities to the software constructs. Once construct reliabilities are captured, the PET tool is used to display construct reliabilities and estimate the defect content of each file. Figure 15 shows how this information is represented. The x-axis of the graph represents the index of the constructs, while the y-axis represents the probability that the construct is defect-prone. Files corresponding to the constructs that are displayed in the graph are shown on the right side sorted by estimated number of defects in descending order. The user is able to zoom in by selecting a specific area of the graph to view the corresponding files.



**Figure 15: PET – Estimated Number of Defective Constructs**

The PET tool was also used to capture the changes during final system testing. This information was used to validate the SDPM estimation. We made the assumption that any change made to a file during the final system and acceptance testing phase was due to defect resolution. This is generally a realistic assumption, since no related development activities occur in the software stream during final system and acceptance testing. By selecting the “Actual” option under the Measurement Type, the PET tool recursively counts the constructs modified in each file during the final system testing phase and generates a report.

### **4.3 Case Study 1: CCD 693- RRACS Interface**

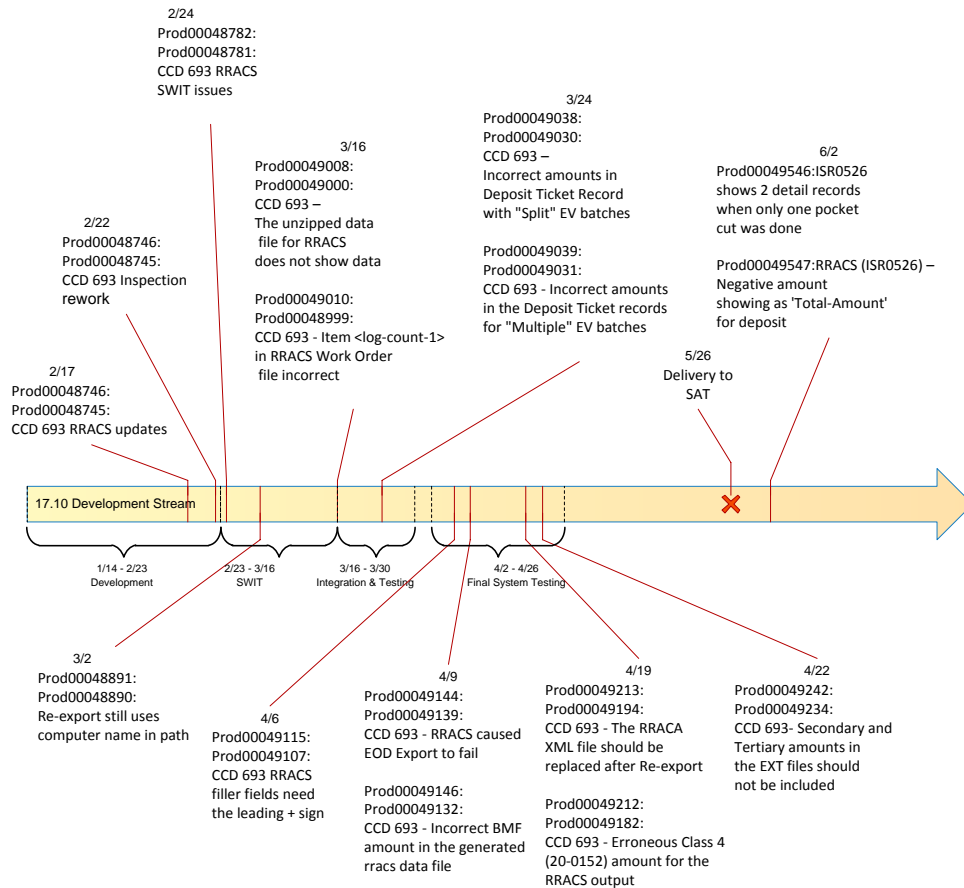
#### **4.3.1 Software Project Background and History**

For this case study, we selected a software development project from a maintenance contract. The duration of the project was 12 months, from July 2009 to August 2010. The purpose of this case study is to validate the Software Development Profile Model by demonstrating its use as a causal model for showing the causal relationship

between change history, the software development activities, and the defect-proneness of files. The project started with the Authorization to Proceed (ATP) July 31, 2009, followed by analysis and planning. The coding phase started on January 14, 2010 and finished as scheduled on March 16, 2010. The activities during the development phase directly related to this project consisted of the initial coding (including unit testing), inspection of the change sets, inspection defects rework, and software integration testing (SWIT) activities. After the development phase, software changes were handed to an independent integration and testing (I&T) team for validation. The handoff occurred on time on March 16, 2010. After fourteen days, the I&T phase was completed on March 30, 2010. During the I&T phase, independent test engineers performed in-depth tests of the software based on the test plan developed from the new software requirements provided by the customer.

Four I&T defects were identified during the formal I&T phase, documented and assigned to the development team for resolution. After all I&T defects were reworked and code changes were complete, the changes were inspected, and delivered to the development stream for a final build. The final build was conducted and the final version of the software was ready for complete final system testing on April 1<sup>st</sup> final system testing was performed to ensure that no additional defects were introduced during the repair process. The regression was conducted systematically based on the plan developed by the test team to validate common software functions. The duration of the final system testing lasted from April 2, 2010 until April 26, 2010.

Figure 16 shows the three software development phases and major activities during each of the three development phases.



**Figure 16: Software Development Activities**

#### 4.3.2 Case Study Measurements

The development stream was created on February 16 to allow developers to begin development activities and check-in their software updates in the configuration management (CM) tool. On February 17, 2010 the first set of changes was made and delivered to the stream. A formal inspection meeting was scheduled for February 22, 2010. The inspection package contained 1152 SLOC changes. Once the inspection

package was created, it was sent to the inspectors three days prior to the formal meeting on Feb 22, 2010. Each inspector was asked to follow the modified inspection process by reviewing the changes independently and submitting any findings prior to formal inspection by the moderator. During the inspection meeting defective constructs were reviewed, invalid findings were eliminated, and finally twenty-four constructs (SLOCs) were identified as defective ( $i=24$ ). Using the overlap between inspectors, we estimate the number of remaining constructs using the capture-recapture model proposed by Chao [16]. Based on the independent review of the code changes by four independent inspectors, a total of eighty-five defective SLOCs were estimated ( $N=85$ ).

After the formal inspection, the findings were handed to the development team for rework. While analyzing the software development activities, we noticed that developers occasionally combined unrelated code changes under the same activity to save time. While this is not recommended and uncommon, we were able to identify such deliveries and exclude them from the analysis.

The code updates addressing inspection defects were delivered to the stream on February 22, 2010. After the inspection process and rework, software integration testing (SWIT) started. During SWIT testing, additional defects were identified resulting in code changes which were delivered to the development stream on February 24, March 3 and March 16, 2010. After the SWIT phase, the software build was handed to I&T for system integration testing. The I&T team found two additional defects which were both resolved on March 24, 2010. The I&T phase

concluded on March 30, 2010 without any additional findings. After the I&T phase the final software build was conducted and ready for final system testing.

In the next section we discuss how the above measurements of the software changes history obtained from the CM tool and the sequence of software development activities can be used as an input to the Software Development Profile Model for estimating the defect content of software artifacts.

#### 4.3.3 Model Parameter Estimation

In this section we discuss how model parameters can be calculated based on the measurements taken for each change set. Table 3 shows the size of the software changes made in the development stream. There are eight columns and rows representing the size of change in eight change sets. Each column represents the number of constructs that are touched in each change set. In each column, the first entry represents the number of constructs that were modified or created during this project, followed by the number of constructs that were touched again in subsequent change sets. Column 1, for example, shows the change history of constructs that were initially created or modified during change set 1. All constructs that were implemented in change set 1 are divided into 1093 SLOCs that were only changed in change set 1, twenty-nine modified in change sets 1 and 2; two changed in change sets 1 and 3; five modified in change sets 1, 2 and 3; four modified again in change set 6; eleven modified in change sets 1 and 7; and finally eight SLOCs modified in change sets 1, 2 and 7. Therefore, the total number of constructs that were modified in change set 1 is 1152. Columns 2 through 8 show the change history for constructs that were implemented during each change sets respectively.

**Table 3: Number of Constructs Modified during Each Change Set**

CS	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
00000001-0002	1093							
00000010-0004		155						
00000011-0006	29							
00000100-0008			44					
00000101-0010	2							
00000111-0014	5							
00001000-0016				33				
00010000-0032					1			
00100000-0064						44		
00100001-0066	4							
01000000-0128							22	
01000001-0130	11							
01000011-0134	8							
10000000-0256								143

Using the data obtained from the software change history, we now estimate the model parameters for each change set based on the SDPM described in Section 3. Table 4 shows the parameter estimations for all change sets. Each change set has only two parameters  $p$  and  $q$ , because testing activities occurred in separate change sets.

**Table 4: SDPM Parameter Estimation**

Change Set	Type	Size	C / R	Est. Defects SLOC	Observed SLOCs	q(i)	p(i)	r(i)
48746	Dev	1152	Y	85.00	24	0.2824	0.9262	0.9471
48746	Inspection	197	N	14.54	2	0.0265	0.9262	0.9282
48782	SWIT	51	N	3.76	0	0.0000	0.9262	0.9262
48891	SWIT	33	N	2.43	0	0.0000	0.9262	0.9262
49008	SWIT	1	N	0.07	4	0.0501	0.9262	0.9299
49010	SWIT	48	N	3.54	15	0.1890	0.9262	0.9402
49038	I&T	41	N	3.03	2	0.0297	0.9262	0.9284
49039	I&T	143	N	10.55	0	0.0000	0.9262	0.9262

In Table 4 the first column represents the activity number. The second column shows the type of change and the third shows the size of change set. Column 4 is used to



indicate if capture-recapture method was used to estimate the defect content of the change set. While it is required to inspect every software change, it is unrealistic to conduct a formal inspection for small code changes. During this software development project, the inspection process allowed small changes to be reviewed by the inspectors without holding a formal inspection meeting. For change sets in which no capture-recapture was performed, we used the defect injection probability “ $p$ ” estimate from change set 1. Column 5 shows the estimated SLOCs that became defective in each change set. Column 6 shows the number of defective SLOCs that were observed during inspection or testing of each change set. It is further assumed that all defective SLOCs observed are reworked in the next change set. Based on these estimates change set reliabilities are estimated which are shown in Column 9.

Since defective constructs can be observed and reworked in the subsequent change sets, we need to use the software development process to estimate the probability of each construct being defect free. As discussed by the authors [37], the software development process should not be ignored when modeling software defect content. Modeling the development process is important to software organizations because it allows software managers to adjust development activities and improve the outcome of the project.

In this case study each change set consists of a coding followed by a defect discovery activity. After the first change set, the initial implementation of the new functionality is followed by the inspection process. The next change, which consists of coding activity due to inspection rework and possible additional changes, is followed by the SWIT testing activity. The software development activities in project 1 are modeled

using the BDD shown in Figure 17. Each coding activity is followed by a defect discovery activity. Activities Coding1 through Coding8 represent the coding activities. The edges  $p_1, p_2, \dots, p_8$  represent the probability that a given construct that is modified has not been injected with a defect. Similarly,  $1 - p_1, 1 - p_2, \dots, 1 - p_8$  are the defect injection probabilities, which are the probabilities that a given modified construct is injected with a defect. The defect discovery activities which follow coding are labeled Insp, SWIT and I&T. Edges  $q_1, q_2, \dots, q_7$  represent the probability that a defective construct from previous coding activities is observed during the defect discovery. Based on the Binary Decision Diagram, we can estimate the probability of a given construct being defective according to Equation 20. Construct reliabilities are shown in Table 6. Column 1 represents the change set codes according to Equation 22. Column 2 shows the number of SLOCs modified in various change sets. Construct reliabilities are estimated based on the change sets during which constructs are modified, as described in Equation 20. According to Table 6, the probability of a given construct implemented only in change set 1 to be defective is estimated as  $r_i^1 = 0.94705$ . In change set 2, twenty-nine SLOCs originally modified in change set 1 were reworked. Equation 20 states that for these constructs to be correct, they have to be implemented correctly in change sets 1 and 2, or, if they became defective in change set 1, they must have been observed during inspection and correctly reworked in change set 2. Under this assumption, the probability of these constructs being correct is  $r_i^{1,2} = 0.89102$ . All other construct reliabilities are estimated similarly.

As described in Section 3.6.2, the SDPM allows for our state of knowledge to be used to improve the model estimates by updating model parameters when new information becomes available. The new information can either come based on expert judgment or additional information obtained outside the project. Once model parameters are estimated, they can be updated using Bayesian inference. We asked the Technical Project Manager (TPM) to provide us with his judgment on the quality of the changes that were made during the coding phase. Based on the requirements volatility and the skill level of the developer that worked on specific change sets, we updated the values of  $p_7, p_8$  which also resulted in new estimates for  $q_7, q_8$ . We used the updated parameters and calculated new construct reliability estimates which are shown in Table 5.

**Table 5: Parameter Updates Based on External Factors**

Change Set	Type	Size	C / R	Est. Defective SLOC	Observed Defective SLOCs	$q'(i)$	$p'(i)$	$r'(i)$
48746	Coding	1152	Y	85.00	24	0.2824	0.9262	0.9471
48746	Inspection	197	N	14.54	2	0.0265	0.9262	0.9282
48782	SWIT	51	N	3.76	0	0.0000	0.9262	0.9262
48891	SWIT	33	N	2.43	0	0.0000	0.9262	0.9262
49008	SWIT	1	N	0.07	4	0.0501	0.9262	0.9299
49010	SWIT	48	N	3.54	15	0.1890	0.9262	0.9402
49038	I&T	41	N	20.5	2	0.0236	0.5000	0.5118
49039	I&T	143	N	1.43	0	0.0000	0.9900	0.9900

After estimating the reliability of all software constructs, we used the defect content estimator described in Section 3.3 to estimate the number of defective constructs in

files modified in the development stream. The list was then sorted in descending order based on the estimated number of defective constructs in each file. Table 7 shows the defect-prone files in descending order. The first column shows the file names, the second shows the magnitude of change in each file in SLOCs. The third column represents the estimated number of defective SLOCs based on the SDPM estimator. In the next section we will compare the SDPM estimation with files that were modified during final system testing in order to assess the accuracy of the findings.

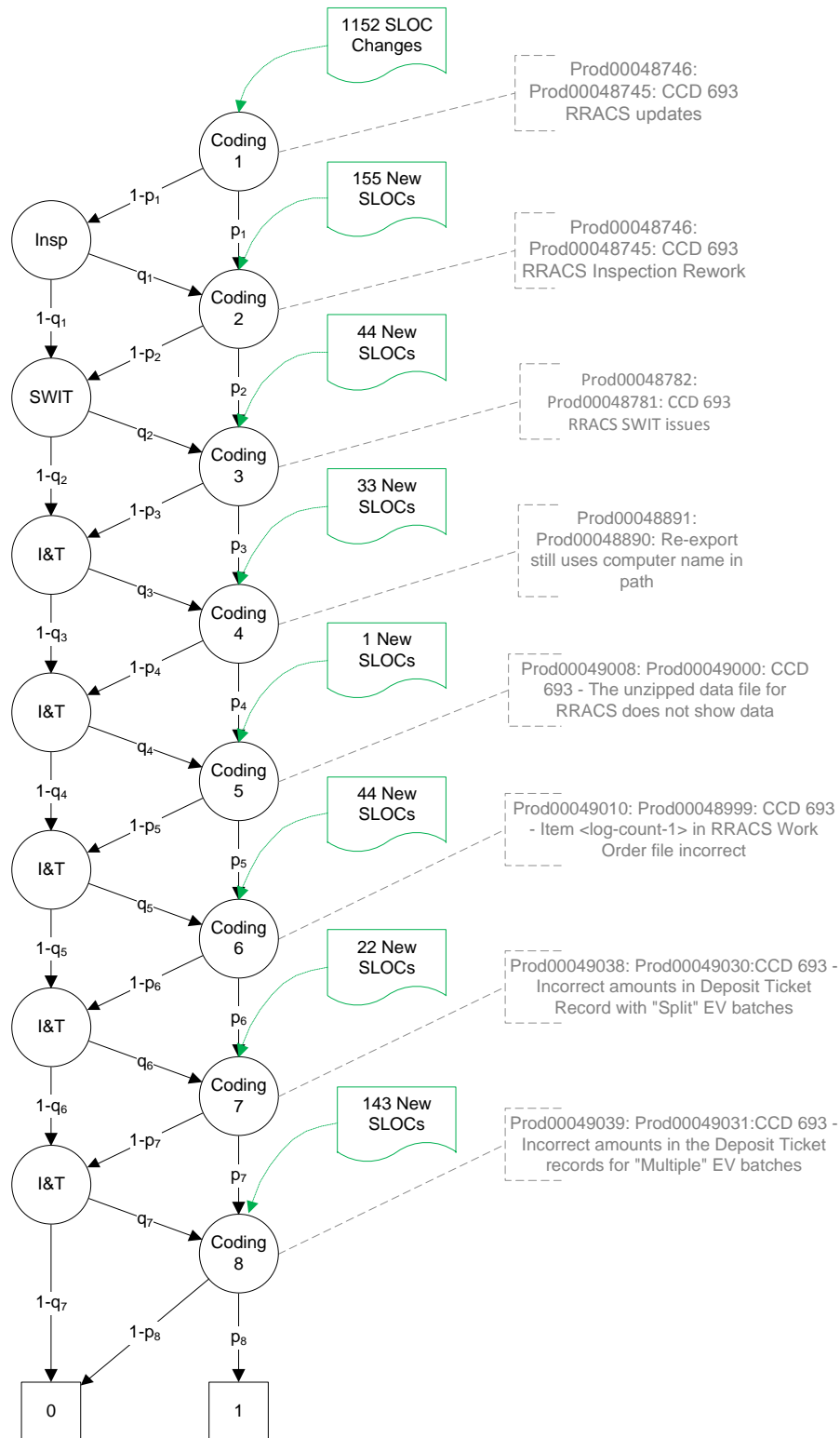


Figure 17: CCD 693 Binary Decision Diagram

**Table 6: Construct Reliability Estimations**

CS	Size	SDPM Model		R(i)	R'(i)
0002	1093	r(1)	r(1)	0.9470	0.9471
0004	155	r(2)	r(2)	0.9281	0.9282
0006	29	r(1,2)	$r1*p2+(1-r1)*q1*p2$	0.8910	0.8910
0008	44	r(3)	r(3)	0.9262	0.9262
0010	2	r(1,3)	$r1*p3+(1-r1)*q2*p3$	0.8784	0.8785
0014	5	r(1,2,3)	$r(1,2)*p3+(1-r(1,2))*q2*p3$	0.8279	0.8280
0016	33	r(4)	r(4)	0.9262	0.9262
0032	1	r(5)	r(5)	0.9299	0.9299
0064	44	r(6)	r(6)	0.9401	0.9402
0066	4	r(1,6)	$r1*p6+(1-r1)*q5*p6$	0.8796	0.8796
0128	22	r(7)	r(7)	0.9284	0.5118
0130	11	r(1,7)	$r1*p7+(1-r1)*q6*p7$	0.8864	0.4785
0134	8	r(1,2,7)	$r(1,2)*p7+(1-r(1,2))*q6*p7$	0.8444	0.4558
0256	143	r(8)	r(8)	0.9262	0.9900

#### 4.3.4 Case Study Results

In the previous section we used a real life software development project and described how SDPM was used as a causal model to predict the number of defective constructs in files modified during the software development process.

In order to determine the accuracy of the SDPM estimation, we examined files modified during the final system testing phase. We made the assumption that any change made to files during the final system testing phase is due to a defect resolution. This is generally a fair assumption since no development activities occur in the software stream during final system testing. The only exceptions are software updates due to changes in parallel software streams, which as described previously, were not included in this analysis.

Nine files were modified during final system testing as the result of defects found related to the current software development project. By comparing the files that were modified during final system testing with files identified as defect-prone by the SDPM, we observed that all nine modified files were on the list of defect-prone files. In addition, we used the SLOCCO tool to further investigate the number of SLOCs that were modified during final system testing with the number of defective SLOC estimated by the SDPM. SLOCCO is a custom tool that is used to compare two source files, and calculating the SLOC volatility between the two versions. Column 6 in Table 7 shows the number of SLOCs that were modified during final system testing for each file. Column 5 shows the number of defective SLOCs estimated by SDPM. We noticed that seven out of nine files modified during final system testing were on top of the list of defect-prone files estimated by the SDPM.

**Table 7: Estimated Number of Defective SLOCs**

File Name	Size (SLOC)	Churn (SLOC)	Est. # of Defective SLOCs per File (Updated Parameters)	Est. # of Defective SLOCs per File (Initial Parameters)	Observed SLOC changes in Files during final system testing
\\rp\\RRACS\\RRACS\\RRACS_Generator.cs	270	270	12.6373	16.2737	15
\\rp\\RRACS\\RRACS\\Deposit.cs	136	136	17.8235	8.8629	20
\\rp\\RRACS\\RRACS\\InputRecords\\RemittanceTransactionRecord.cs	144	144	6.2928	8.5536	5
\\rp\\RRACS\\RRACS\\InventoryDB.cs	144	144	6.2928	8.5536	6
\\rp\\RRACS\\RRACS\\TaxClassMap.cs	143	143	1.4014	8.4942	1
\\rp\\RRACS\\RRACS\\OutputRecords\\DepositTicketRecord.cs	124	124	5.4188	7.3656	3
\\rp\\RRACS\\RRACS\\DepositList.cs	98	98	4.2826	5.8212	5
\\rp\\RRACS\\RRACS\\InputRecords\\BlockHeaderRecord.cs	78	78	3.4086	4.6332	#N/A
\\cs\\cs_Create_Interchange_Data\\cs_Create_Interchange_Data.cpp	693	72	3.3157	4.3327	#N/A
\\rp\\RRACS\\RRACS\\OutputRecords\\JournalSummaryRecord.cs	62	62	2.7094	3.6828	1
\\rp\\RRACS\\RRACS\\OutputRecords\\FileIDJournalRecord.cs	55	55	2.4035	3.267	#N/A

\\rp\\RRACS\\RRACS\\DLNRecord.cs	50	50	2.185	2.97	#N/A
\\rp\\rp_perform_EOD_export\\rp_EOD_tapes.cpp	1531	47	2.0539	2.7918	#N/A
\\rp\\RRACS\\RRACS\\InputRecords\\CheckRecord.cs	46	46	2.0102	2.7324	#N/A
\\rp\\RRACS\\RRACS\\OutputRecords\\OutputRecord.cs	39	39	1.7043	2.3166	1
\\rp\\RRACS\\RRACS\\InputRecords\\BlockTrailerRecord.cs	35	35	1.5295	2.079	#N/A
\\rp\\RRACS\\RRACS\\InputRecords\\InputRecord.cs	33	33	1.4421	1.9602	#N/A
\\gen\\include\\cs_ftp_common.h	339	30	1.782	1.782	#N/A
\\rp\\RRACS\\RRACS\\Properties\\AssemblyInfo.cs	15	15	0.6555	0.891	#N/A
\\gen\\include\\cs_types_pvals.h	216	14	0.6118	0.8316	#N/A
\\rp\\RRACS\\RRACS\\Properties\\Settings.Designer.cs	12	12	0.5244	0.7128	#N/A
\\rp\\RRACS\\RRACS\\Settings.cs	10	10	0.437	0.594	#N/A
\\cs\\cs_store_ops\\cs_store_ops.cpp	1041	6	0.2622	0.3564	#N/A
\\gen\\include\\cs_export.h	131	4	0.1748	0.2376	#N/A
\\cs\\cs_reexport_inventory\\cs_reexport_inventory.cpp	689	3	0.1625	0.1782	#N/A
\\cs\\cs_Tape_Tools\\cs_determine_export_media.cpp	244	3	0.1311	0.1782	#N/A
\\gen\\include\\rp_create_EOD_volume_set.h	80	3	0.1311	0.1782	#N/A
\\gen\\include\\cs_types_common.h	335	2	0.0874	0.1188	#N/A
\\rp\\rp_perform_EOD_export\\rp_perform_EOD_export.cpp	429	2	0.0874	0.1188	#N/A
\\rp\\rp_perform_EOD_export\\rp_transport_file.h	160	2	0.0874	0.1188	#N/A
\\cm\\isrp_build_gui.pl	1446	1	0.0594	0.0594	#N/A
\\gen\\include\\cs_common.h	37	1	0.0594	0.0594	#N/A
\\rp\\rp_perform_EOD_export\\rp_EOD_tapes_private.h	113	1	0.0437	0.0594	#N/A

We used the coefficient of correlation to evaluate the performance of the SDPM which is shown in Table 8.



**Table 8: Coefficient of Correlation – Case Study 1 - (SDPM model)**

	<i>Est. # of Defective SLOCs (Initial Estimate)</i>	<i>Est. # of Defective SLOCs (Updated Parameters)</i>	<i>Observed SLOC changes during final system testing</i>
Est. # of Defective SLOCs (Initial Estimate)	1		
Est. # of Defective SLOCs (Updated Parameters)	0.82828	1	
<b>Observed SLOC changes during final system testing</b>	<b>0.64840</b>	<b>0.98597</b>	<b>1</b>

While the initial estimates suggest a correlation between the estimated number of defective constructs and the observed SLOC changes, the updated parameters shows a stronger correlation.

#### 4.3.5 Poisson Regression Model Results

We used defect data from releases 10.4 to 17.9 to estimate the number of defects in Release 17.10 files. To fit the data, we used the Poisson regression model as described in Section 4.1.1. The predictor variables used in this case study were logarithm of the SLOCs, square root of prior defects, age, and file status (New, Changed, and Unchanged). Table 9 shows the regression coefficients.

**Table 9: Coefficient of Regression – Case Study 1 - (Poisson Regression)**

<b>Coefficient</b>	<b>Estimate</b>	<b>Std Error</b>	<b>L-R ChiSquare</b>	<b>Prob&gt;ChiSq</b>	<b>Lower CL</b>	<b>Upper CL</b>
Intercept	-0.690	0.071	98.174	3.83E-23	-0.878	-0.551
Log(SLOC)	0.181	0.015	148.714	3.31E-34	0.152	0.210
Sqrt(PriorDef)	-0.442	0.027	276.596	4.14E-62	-0.495	-0.388
Age	-0.106	0.005	696.348	1.86E-153	-0.116	-0.096
New[0]	-1.758	0.038	2863.009	0.00E+00	-1.834	-1.684
Changed[0]	-1.970	0.034	4486.093	0.00E+00	-2.038	-1.904
Unchanged[0]	0	.	.	.	.	.

Using the coefficient of regression we estimated the number of defects in Release 17.10 and sorted them in descending order to identify the defect prone files. Table 10 below shows the files in descending order based on Poisson model estimates. Column two shows the actual number of defects observed during software final system testing. By comparing the number of observed defects with the raking assigned by Poisson model, we can see that the model performs well in identifying defect prone files.

**Table 10: Estimated Number of Defects (Poisson Model)**

File Name	Defects	Status Changed, Unchanged New			Age	Log (SLOC)	Sqrt (Prior Defects)	Poisson Model
\RRACS_Generator.cs	2	0	0	1	0	2.111	0.000	0.126653
\RemittanceTransactionRecord.cs	1	0	0	1	0	2.037	0.000	0.124987
\DepositTicketRecord.cs	3	0	0	1	0	1.978	0.000	0.123644
\InventoryDB.cs	1	0	0	1	0	1.672	0.000	0.116993
\Deposit.cs	3	0	0	1	0	1.633	0.000	0.116178
\DepositList.cs	2	0	0	1	0	1.447	0.000	0.112327
. \JournalSummaryRecord.cs	1	0	0	1	0	1.431	0.000	0.112006
\de_DEDatastoreBuild.sql	NA	0	0	1	0	1.322	0.000	0.109816
\OutputRecord.cs	1	0	0	1	0	0.845	0.000	0.100734
\de_Programs.bat	NA	1	0	0	1	2.083	0.000	0.091729
\drop_unauthorized_dbas.sql	NA	0	0	1	0	0.000	0.000	0.08645
\de_mod13212fn.vb	NA	0	0	1	0	1.949	1.000	0.079104
\BlockHeaderRecord.cs	NA	0	0	1	0	1.716	1.000	0.075833
\FileIDJournalRecord.cs	NA	0	0	1	0	1.380	1.000	0.071363
\DLNRecord.cs	NA	0	0	1	0	1.362	1.000	0.071124
\CheckRecord.cs	NA	0	0	1	0	1.301	1.000	0.070347
\BlockTrailerRecord.cs	NA	0	0	1	0	0.903	1.000	0.06546
\InputRecord.cs	NA	0	0	1	0	0.845	1.000	0.064777
\AssemblyInfo.cs	NA	0	0	1	0	0.477	1.000	0.060605
\Settings.Designer.cs	NA	0	0	1	0	0.301	1.000	0.058704
\de_mod13200fn.vb	NA	1	0	0	8	2.647	1.000	0.031146
\de_mod11214fn.vb	NA	1	0	0	8	1.708	1.000	0.026276

\de_mod11204fn.vb	NA	1	0	0	8	1.708	1.000	0.026276
\de_clssection03.vb	NA	1	0	0	8	1.591	1.000	0.025728
\de_clssection03.vb	NA	1	0	0	8	1.580	1.000	0.025675
\de_extractZipCodeCityStateDB.bat	NA	1	0	0	14	2.444	0.000	0.024742
\de_DEDatastoreBuild.sql	NA	1	0	0	14	1.322	0.000	0.020196
\de_mod11212fn.vb	NA	1	0	0	9	1.708	1.414	0.019687
\de_clssection03.vb	NA	1	0	0	9	1.568	1.414	0.019196
\de_checkZipCodeCityStateDB.bat	NA	1	0	0	14	1.869	1.000	0.014339
\de_mod11200fn.vb	NA	1	0	0	14	1.708	1.000	0.013925
\de.bat	NA	1	0	0	14	1.944	1.414	0.012106
\de_CreateMessageLoader.bat	NA	1	0	0	14	1.875	1.414	0.011955
\cs_format_block_analyze.cpp	NA	1	0	0	23	2.356	0.000	0.009395
\sp_eop_global.h	1	1	0	0	23	2.057	0.000	0.0089
\cs_captured_data_store.h	NA	1	0	0	23	1.833	0.000	0.008546
\de_clssection03.vb	NA	1	0	0	14	1.556	2.449	0.007145
\cs_read_completed_key_entry_data.cpp	NA	1	0	0	23	2.093	1.000	0.005761
\rp_EOD_tapes_private.h	NA	1	0	0	23	1.556	1.000	0.005227
\rp_create_EOD_volume_set.h	NA	1	0	0	23	1.415	1.000	0.005095
\sp_view_ke_data.cpp	1	1	0	0	24	2.427	1.414	0.004584
\sp_eop_ke3_processing.cpp	1	1	0	0	23	2.794	2.000	0.004205
\de_clsblockdata.vb	NA	1	0	0	14	3.113	4.359	0.004076
\sp_eopinit.cpp	1	1	0	0	31	2.301	0.000	0.003989
\cs_format_block.cpp	NA	1	0	0	25	2.576	2.000	0.003272
\cs_types_common.h	NA	1	0	0	32	2.413	1.000	0.002355
\cs_types_pvals.h	NA	1	0	0	32	2.248	1.000	0.002286
.\cm\isrp_build_gui.pl	1	1	0	0	23	2.818	3.464	0.002213
\cs_SA_Dialog.rc	NA	1	0	0	32	3.028	1.414	0.002192
\rp_transport_file.h	NA	1	0	0	33	2.021	1.000	0.001973
\ReportAPI.cpp	NA	1	0	0	31	2.938	2.000	0.001851
\cs_common.h	NA	1	0	0	32	1.176	1.414	0.001568
\cs_determine_export_media.cpp	NA	1	0	0	31	1.886	2.000	0.00153
\cs_export.h	NA	1	0	0	35	1.531	1.000	0.001462
\cs_store_ops.cpp	NA	1	0	0	32	2.612	2.236	0.001414
\rp_EOD_tapes.cpp	NA	1	0	0	33	2.623	2.236	0.001275
\cs_ftp_common.h	NA	1	0	0	31	2.230	2.646	0.001225
\rp_perform_EOD_export.cpp	NA	1	0	0	32	2.072	2.449	0.001167
\cs_reexport_inventory.cpp	NA	1	0	0	32	2.465	2.646	0.00115
\sp_release_block.cpp	NA	1	0	0	33	3.018	2.828	0.001054
\cs_Create_Interchange_Data.cpp	NA	1	0	0	32	2.436	3.000	0.000978

Similar to the SPDM results, we used the coefficient of correlation to evaluate the performance of the Poisson regression. By comparing the coefficient of correlation between the SDPM and Poisson model, we noticed that the estimate provided by the SDPM is more correlated with the defects observed during final system testing than the Poisson model.

**Table 11: Coefficient of Correlation (Poisson Regression)**

	<i>Estimated Number of Defects</i>	<i>Observed Defects</i>
Estimated Number of Defects	1	
Observed Defects	<b>0.520645</b>	1

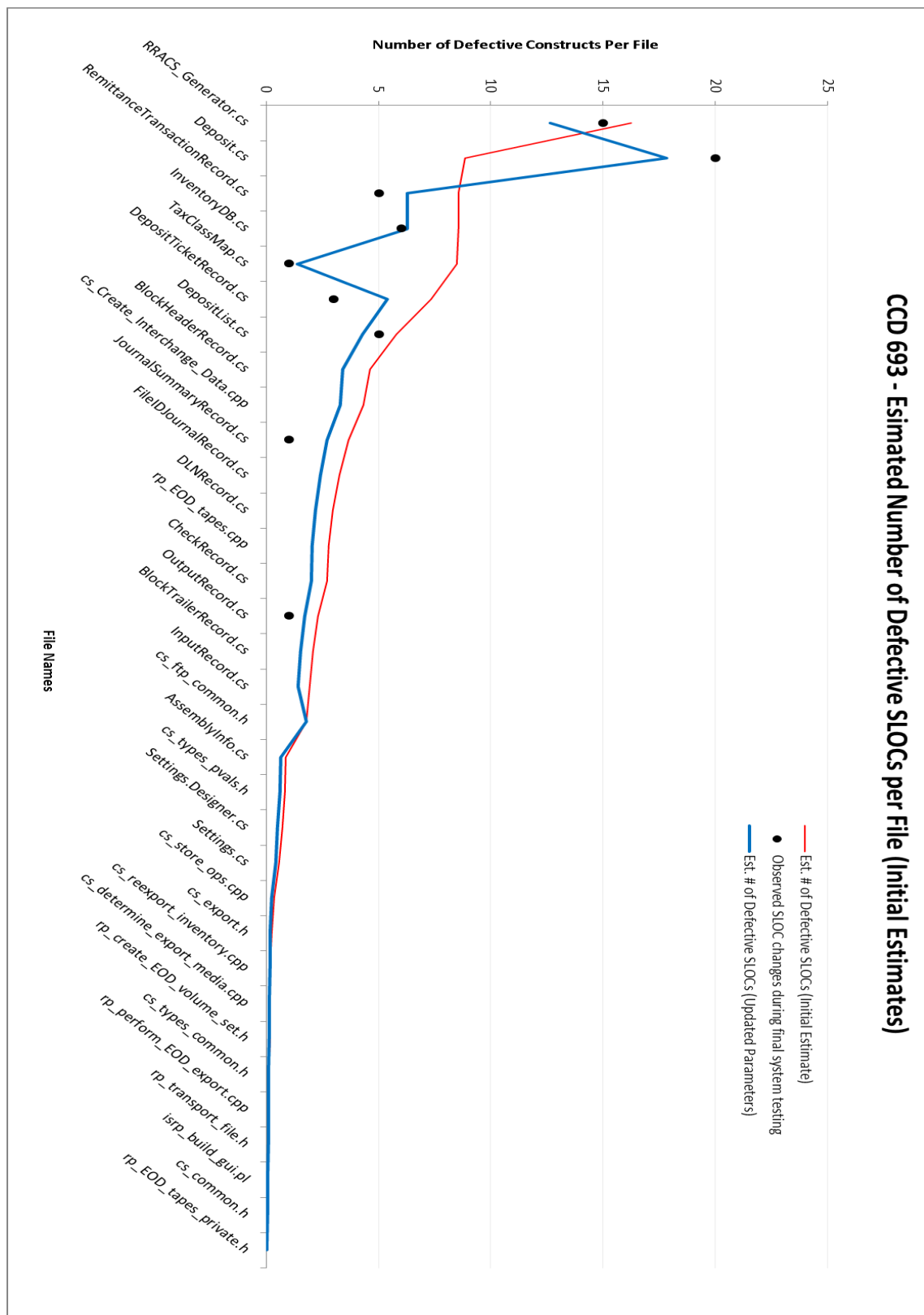


Figure 18: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 17.10

## 4.4 Case Study 2: CCD 762 – IMF (health care) Changes for PY 2011

### 4.4.1 Software Project Background and History

For this case study we selected a software development project that was intended to deliver three new functionalities as part of the DIS/CS 18.4 release. Figure 19 shows the timeline of software development activities for this project. The development phase started on August 26, 2010 and ended on September 20, 2010. During the development phase code changes were delivered in three change sets. The three change sets were also used to deliver rework needed to address observed inspection and SWIT issues. The three major enhancements delivered with this release were:

- CCD 762 – IMF (health care) Changes for PY 2011
  - Changes to PRP"s 15 and 31
- CCD 764 – OLG Changes for PY 2011
  - Update program numbers referenced for two OLG programs including PRP 4 and 5
- CCD 773 – PY 2011 HIRE Changes II
  - Legislative 2011 tax year changes including PRP 45 and 54

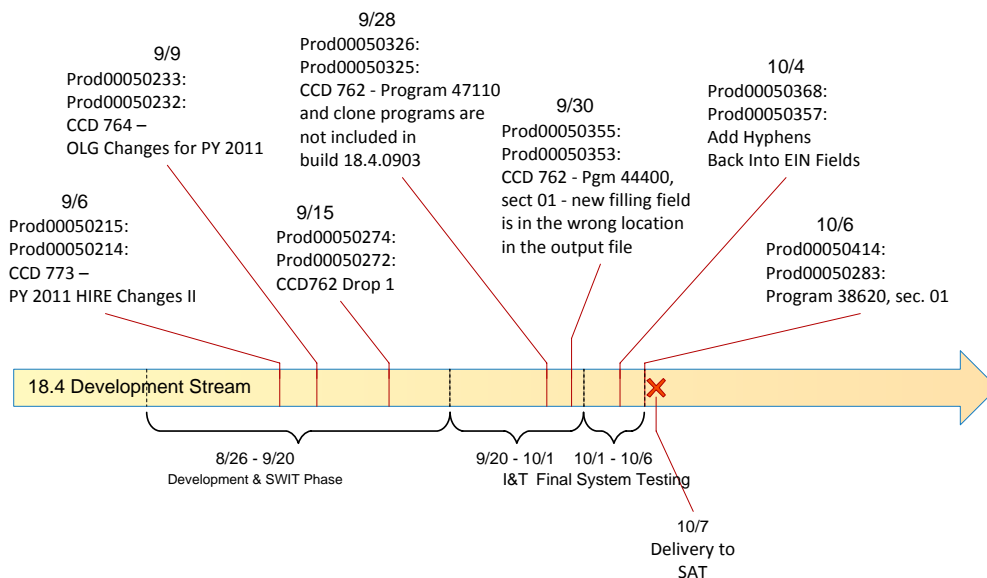


Figure 19: CCD 762 Timeline and Development Activities

#### 4.4.2 Case Study Measurements

In the previous section, we provided a timeline of the software development activities related to the DIS/CS 18.4 software release. On September 6, 2010 the first set of changes were implemented and delivered to the stream. A formal inspection was held and inspection findings, along with the implementation of the second set of enhancements, were delivered on September 9, 2010. Out of 268 SLOC changes delivered in the first change set, twenty-nine had to be reworked due to issues observed during inspection. The largest code churn was delivered with the implementation of CCD 762 Drop1 under change set 3, with 2574 SLOC changes. Code changes were inspected but no major issues were observed during the inspection. After development was complete and code changes were delivered to the stream, the software build was handed to I&T for integration and testing. The I&T team identified two issues which were reworked and delivered in change sets 4 and 5. Figure 20 shows the software change matrix for the DIS/CS 18.4 release.

CS	(1)	(2)	(3)	(4)	(5)
00001-0002	268				
00010-0004		71			
00100-0008			2547		
01000-0016				1	
01100-0024			93		2
10000-0032					2
10100-0040			15		

Figure 20: Software Change Matrix – DIC/CS 18.4 release

#### 4.4.3 Model Parameter Estimation

In this section, we discuss how model parameters are calculated based on the measurements taken in each change set. Table 12 shows the summary of the measurements taken for each change set along with the estimates of the model parameters. Once model parameters  $p_i$  and  $q_i$  are estimated, we calculate the change set reliabilities, shown in Column 8 of Table 12. Since software constructs can be modified in more than one change set, we use the Binary Decision Diagram shown in Figure 21 to estimate construct reliabilities.

**Table 12: Parameter Estimation for DIS/CS 18.4**

Change Set	Size	Cap-Recap	(2) Est. New Defects	(1)Observed (Modified/Fixed)	$p(i)$	$q(i)$	$r(i)$
50215	268	Y	48	29	0.8209	0.6042	0.90972
50233	71	N	13	3	0.8209	0.0813	0.83285
50274	2655	N	476	39	0.8209	0.0762	0.83210
50326	94	N	17	23	0.8209	0.0461	0.82768
50355	17	N	3	1	0.8209	0.0020	0.82119





After estimating the reliability of all software constructs, we used the defect content estimator described in Section 3.3 to estimate the number of defective constructs in files modified in the development stream. The list was then sorted in descending order based on the estimated number of defective constructs in each file. In the next section we discuss the results of the case study by comparing the SDPM estimates with the actual SLOC changes during the final system testing.

#### 4.4.4 Case Study Results

In this section, we compare the number of defective constructs estimated by SDPM with the number of constructs modified in each file during final system testing of the DIS/CS 18.4 release. Table 14 shows the defect-prone files in descending order. The first column shows the file names, the second shows the file size, and the third gives the magnitude of change in each file in SLOCs. The fourth and fifth columns represent the estimated number of defective SLOCs based on the SDPM estimator and the observed SLOC changes during final system testing respectively. We use the coefficient of correlation to assess the performance of the SDPM with the observed number of defective SLOCs in each file. We also use the coefficient of correlation to show that the SDPM provides a better estimate than change alone.

**Table 14: DIS/CS 18.4 Case Study Results**

File Name	SLOC	Churn	Est. # of Defects Per File	Observed Defects
de_100000.PCF	8793	964	174.4283	93
de_46125.PCF	936	475	79.61	#N/A
de_46121.PCF	1582	391	65.5316	#N/A
de_43110.PCF	1848	391	65.5316	#N/A
de_44400.PCF	430	127	23.1377	13

de_11502.PCF	438	201	18.1503	#N/A
de_47110.PCF	450	75	12.57	#N/A
de_11640.PCF	512	54	9.0288	2
de_enumcommonsectionfieldnumbers.vb	1608	52	8.7152	#N/A
de_mod46120fn.vb	248	41	6.8716	#N/A
de_mod43110fn.vb	291	39	6.5364	#N/A
de_11300.PCF	412	64	5.7792	2
de_11650.pcf	232	17	2.8424	2
de_mod46125fn.vb	197	16	2.6816	#N/A
de_clsform8919.vb	15	13	2.1788	#N/A
de_clsform8888.vb	114	11	1.8436	#N/A
de_clssection04.vb	42	9	1.6733	3
de_clssection04.vb	49	9	1.6509	2
de_mod44400fn.vb	95	9	1.5084	#N/A
de_clssection05.vb	38	7	1.1732	#N/A
de_clssection05.vb	42	7	1.1732	#N/A
de_mod47110fn.vb	76	6	1.0056	#N/A
de_cls46120.vb	169	4	0.6704	#N/A
de_cls43110.vb	274	3	0.5028	#N/A
de_clsForm8941.vb	3	3	0.5028	#N/A
de_mod11300fn.vb	73	2	0.1806	#N/A
de_clssection03.vb	30	1	0.1721	1
de_clstaxpr15.vb	80	1	0.1676	#N/A
de_clstaxpr31.vb	54	1	0.1676	#N/A
de_clssection01.vb	160	1	0.1676	#N/A
de_clssection03.vb	35	1	0.1676	#N/A
de_clssection03.vb	38	1	0.1676	#N/A
de_mod11502fn.vb	78	1	0.0903	#N/A

Table 15 shows the coefficient of correlation between size of change (churn), SDPM estimate and the number of defective SLOCs. Based on the table, the SDPM provides a good estimate for the number of defective SLOCs. From the coefficient of correlation in Table 15, the SDPM provides a better estimate than the churn alone. Figure 22 shows the estimated number of defective constructs in each file and the number of observed SLOCs modified during final system testing.

**Table 15: Correlation Analysis – DIS/CS 18.4**

	<i>Churn</i>	<i>Est. # of Defective SLOCs</i>	<i>Observed SLOC Changes During Final System Testing</i>
Churn	1		
Est. # of Defective SLOCs	0.9954	1	
<b>Observed SLOC Changes During Final System Testing</b>	<b>0.9975</b>	<b>0.9988</b>	<b>1</b>

#### 4.4.5 Poisson Regression Model Results

We used defect data from releases 10.4 to 18.3 to estimate the number of defects in Release 18.4 files. To fit the data, we used the Poisson regression model as described in Section 4.1.1. Similar to case study 1, the predictor variables used in this case study were logarithm of the SLOCs, square root of prior defects, age, and file status (New, Changed, and Unchanged). Table 16 shows the regression coefficients. As expected, the values of the coefficients of regression are similar to the coefficients estimated in case study 1 because the files share the same structural measures.

**Table 16: Coefficient of Regression – Case Study 2 - (Poisson Regression)**

<b>Coefficient</b>	<b>Estimate</b>	<b>Std Error</b>	<b>L-R ChiSquare</b>	<b>Prob&gt;ChiSq</b>	<b>Lower CL</b>	<b>Upper CL</b>
Intercept	-0.74818	0.071982	112.5288	2.74E-26	-0.94849	-0.60762
Log(SLOC)	0.190285	0.015085	159.9289	1.17E-36	0.160728	0.219859
Sqrt(PriorDef)	-0.43636	0.027608	262.5228	4.84E-59	-0.49063	-0.38241
Age	-0.10177	0.004673	750.9367	2.51E-165	-0.1111	-0.09278
New[0]	-1.79809	0.038935	2903.793	0	-1.87512	-1.72246
Changed[0]	-1.99454	0.034621	4446.066	0	-2.06316	-1.92741
Unchanged[0]	0	.	.	.	.	.

We used the coefficient of regression to estimate the expected number of defects per file in Release 18.4 and identify files that will most likely be defective. The results are shown in Table 17 below. As this table indicates, the Poisson model did not perform well to identify defect-prone files, based on the defects observed during the testing on Release 18.4. In fact, the results were so poor that we were unable to calculate the coefficient of correlation between the actual number of defects and the Poisson regression estimates. After reviewing the defects, we noticed that the majority of defects in Release 18.4 were in non-executable .PCF files. Since regression models are built based on defects observed during testing and operation, the model does not perform well for non-executable files. This is a known disadvantage with Regression based models and the non-executable files are generally excluded from such models [53].

**Table 17: Estimated Number of Defects-Case Study 2 - (Poisson Model)**

File Name	Defects	Status Changed, Unchanged, New			Age	Log (SLOC)	Sqrt (Prior Defects)	Poisson Model
\de_mod46120fn.vb	NA	1	0	0	22	2.39	1.00	0.0070
\de_mod46125fn.vb	NA	1	0	0	22	2.29	1.00	0.0069
\de_mod43110fn.vb	NA	1	0	0	22	2.46	1.41	0.0059
. \de_mod44400fn.vb	NA	1	0	0	22	1.98	1.41	0.0054
\de_clssection03.vb	NA	1	0	0	22	1.58	1.41	0.0050
\de_clssection05.vb	NA	1	0	0	22	1.58	1.41	0.0050
\de_clssection03.vb	NA	1	0	0	22	1.54	1.41	0.0050
\de_cls46120.vb	NA	1	0	0	22	2.23	1.73	0.0049
\de_mod11502fn.vb	NA	1	0	0	23	1.89	1.41	0.0048
\de_mod11300fn.vb	NA	1	0	0	23	1.86	1.41	0.0048
\de_clssection07.vb	NA	1	0	0	22	1.28	1.41	0.0047
\de_cls43110.vb	NA	1	0	0	22	2.44	2.00	0.0046
\de_clssection04.vb	1	1	0	0	22	1.69	1.73	0.0044
\de_clssection05.vb	NA	1	0	0	22	1.62	1.73	0.0044
\de_clssection04.vb	1	1	0	0	22	1.62	1.73	0.0044

\de_clssection01.vb	NA	1	0	0	22	2.20	2.00	0.0044
\de_mod47110fn.vb	NA	1	0	0	25	1.88	2.00	0.0030
de_enumcommonsectionfield numbers.vb	NA	1	0	0	28	3.21	2.00	0.0029
de_clssection02.vb	NA	1	0	0	25	0.90	1.73	0.0028
de_enummessages.vb	NA	1	0	0	28	2.27	2.00	0.0024
de_clsform8888.vb	NA	1	0	0	28	2.06	2.00	0.0023
de_clssection03.vb	1	1	0	0	25	1.48	2.45	0.0023
de_clspipelinebh.vb	NA	1	0	0	28	2.36	2.24	0.0022
de_clsirpbh.vb	NA	1	0	0	28	2.30	2.24	0.0022
de_clstaxpr31.vb	NA	1	0	0	28	1.73	2.00	0.0022
de_clstaxpr15.vb	NA	1	0	0	28	1.90	2.24	0.0020
de_clsform1040xs02.vb	NA	1	0	0	28	1.20	2.00	0.0020
de_clsform8919.vb	NA	1	0	0	28	1.18	2.00	0.0019
de_clsfield.vb	NA	1	0	0	28	2.59	3.00	0.0016
de_clsiffeeiflookup.vb	NA	1	0	0	28	2.57	3.87	0.0011
ReportAPI.h	NA	1	0	0	39	1.86	1.00	0.0011
de_ctlfield.vb	NA	1	0	0	28	3.14	4.58	0.0009
de_clsstatemachine.vb	NA	1	0	0	28	3.45	6.00	0.0005
ReportAPI.cpp	NA	1	0	0	45	2.95	2.00	0.0005

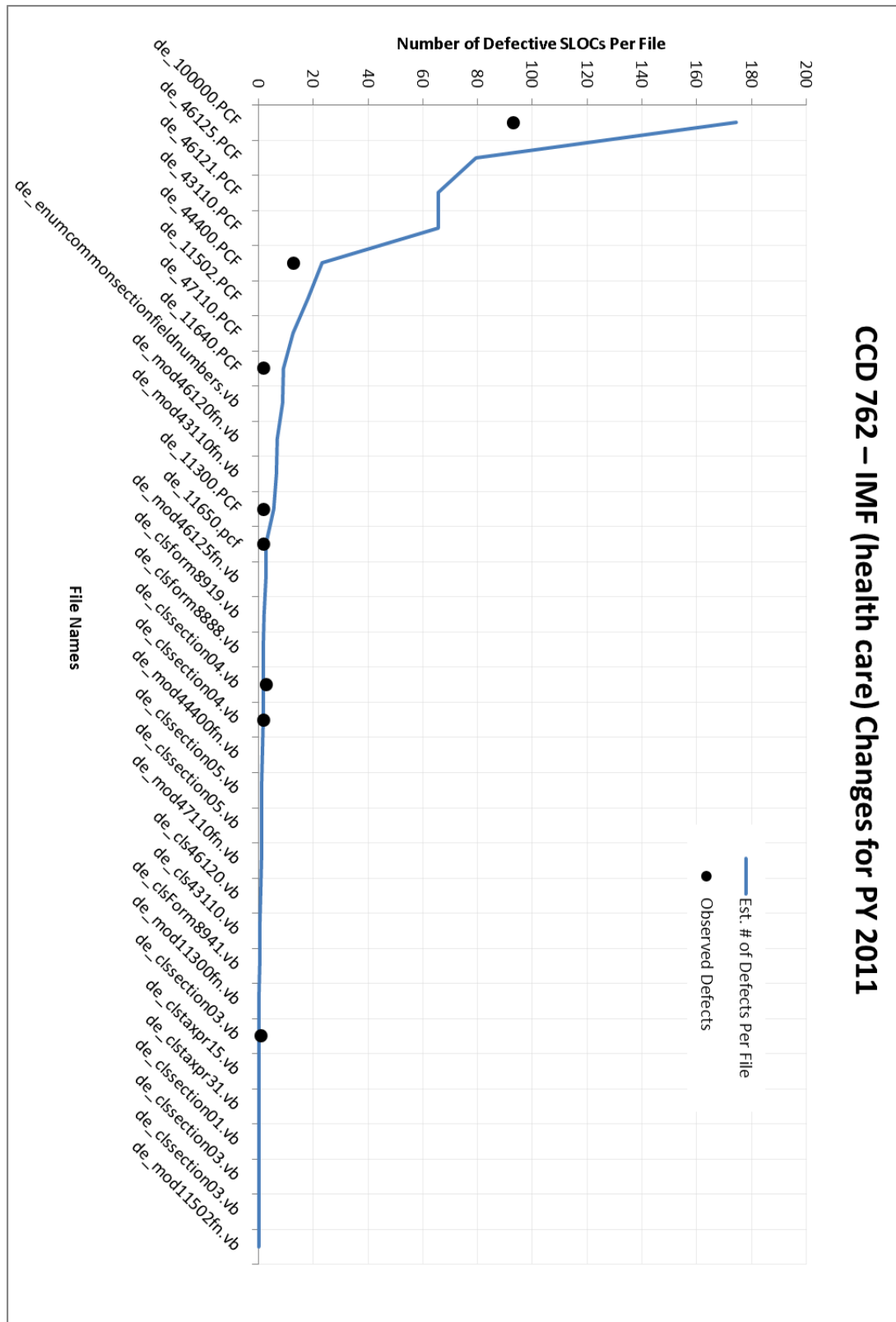


Figure 22: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 18.4

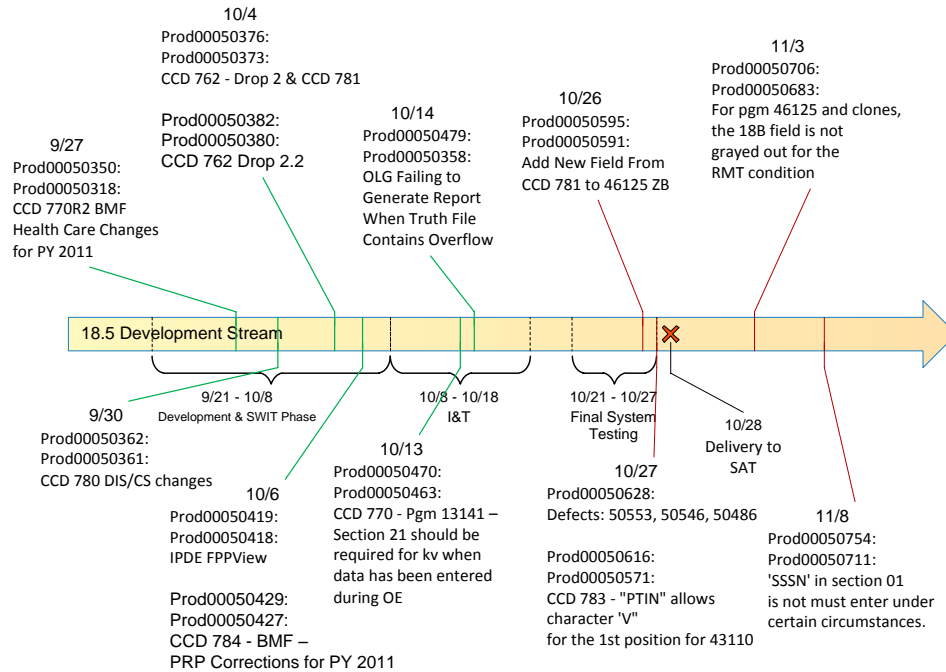
## **4.5 Case Study 3: CCD 770R2- BMF Health Care Changes for PY 2011**

### **4.5.1 Software Project Background and History**

For this case study we selected a software development project that was intended to deliver six functionalities as part of the DIS/CS 18.5 release. Figure 23 shows the timeline of software development activities for this project. The development phase started on September 21, 2010 and ended on October 10, 2010. During the development phase, code changes were delivered in four change sets. The four change sets were also used to deliver code changes needed to address observed inspection and SWIT issues. The major enhancements delivered with this release were:

- CCD 762 – IMF (health care) Changes for PY 2011
  - Changes to PRP"s 33 and 36
- CCD 770 – BMF (Health Care) Changes for PY 2011
  - Changes to PRP"s 01, 27, 32, 39, 47, 48, 50, 51 and 54
- CCD 780 – RP Changes PY 2011, PY 2010
  - Update EOD code
- CCD 781 – IMF (Health Care) Changes II for PY 2011
  - Changes to PRP 31
- CCD 783 – PRP 31 Corrections for PY 2011
  - Correct PRP 31
- CCD 784- BMF – Corrections for PY 2011
  - Correct PRP"s 43, 84 and 90





**Figure 23: CCD 770 Timeline and Development Activities**

#### 4.5.2 Case Study Measurements

In the previous section we provided a timeline of the software development activities related to the DIS/CS 18.5 software release. On September 27, 2010, the first set of changes were implemented and delivered to the stream. A formal inspection was held and inspection findings, along with the implementation of the second set of enhancements, were delivered on September 30, 2010. Out of 2864 SLOC changes delivered in the first change set, ninety-three were reworked in change set 2, and three were modified again in change set 7. Figure 24 shows the software change matrix for the DIS/CS 18.5 release. This release was different from the other case studies in that it consisted of six independent enhancements within the same release.

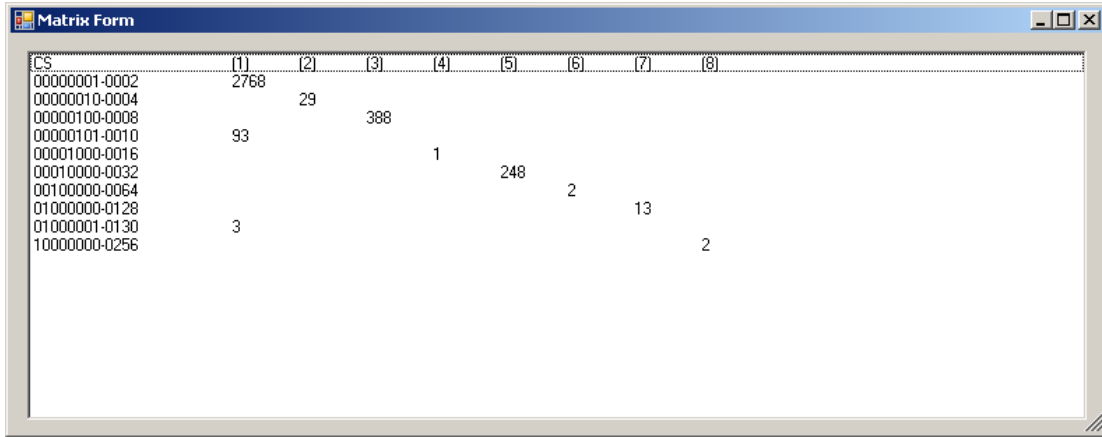


Figure 24: Software Change Matrix – DIC/CS 18.5

#### 4.5.3 Model Parameter Estimation

In this section we discuss how model parameters are calculated based on the measurements taken in each change set. Table 18 shows the summary of the measurements taken for each change set along with the estimates of the model parameters. Once model parameters  $p_i$  and  $q_i$  are estimated, we calculate the change set reliabilities, which are shown in Column 8 of Table 18.

Table 18: Parameter Estimation for DIS/CS 18.5

Change Set	Size	Cap-Recap	(2) Est. New Defects	(1)Observed (Modified/ Fixed)	p(i)	q(i)	r(i)
50350	2768	Y	150	140	0.9458	0.9333	0.9937
50362	29	N	2	1	0.9458	0.0522	0.9485
50376	481	Y	42	35	0.9127	0.5730	0.9584
50382	1	N	0	0	0.9458	0.0000	0.9458
50419	248	N	13	23	0.9458	0.4372	0.9682
50429	2	N	0	1	0.9458	0.0212	0.9469
50470	16	N	1	3	0.9458	0.0625	0.9490
50479	2	N	0	2	0.9458	0.0416	0.9479

Table 18 shows the probability of constructs being defect-free based on the Binary Decision Diagram shown in Figure 25.

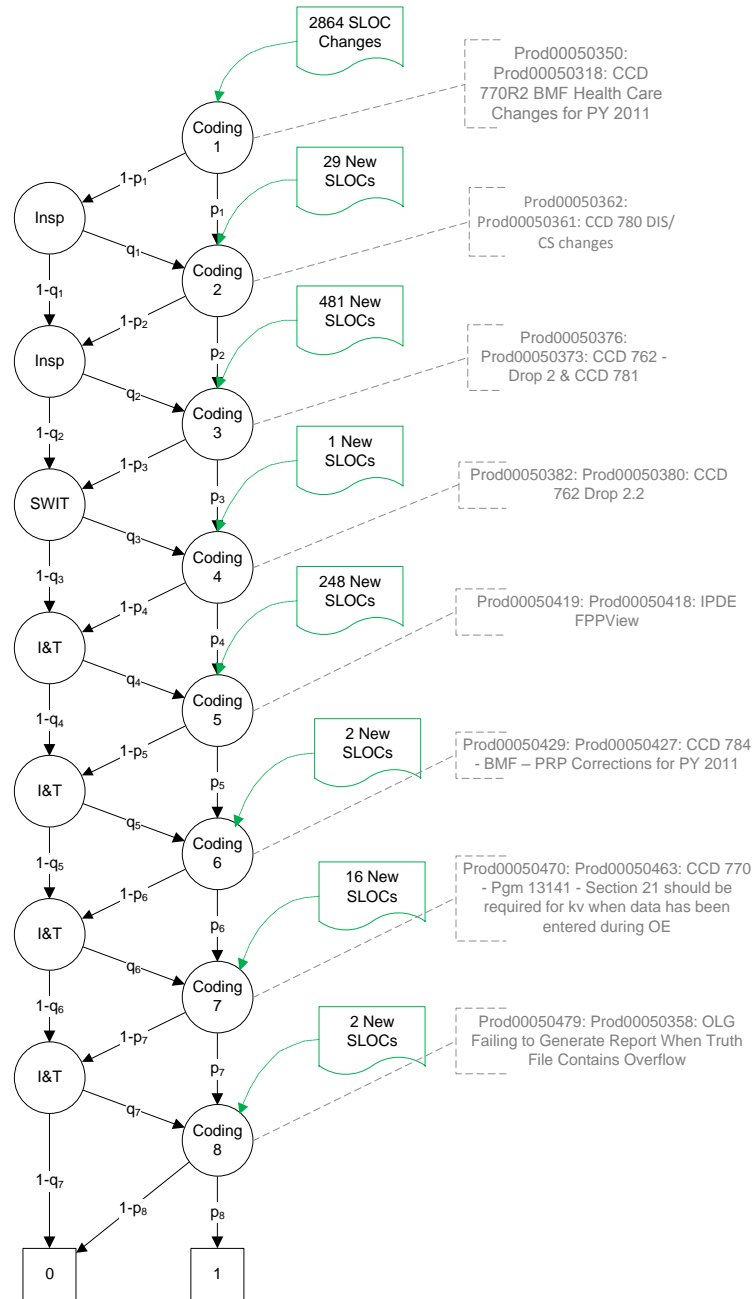


Figure 25: Binary Decision Diagram – DIS/CS 18.5

**Table 19: Construct Reliability Estimation – DIS/CS 18.5**

Change Sets	Churn	Probability		Probability
0002	2768	r(1)	r(1)	0.99365
0004	29	r(2)	r(2)	0.94849
0008	388	r(3)	r(3)	0.95835
0010	93	r(1,3)	$r1*p3+(1-r1)*q2*p3$	0.90719
0016	1	r(4)	r(4)	0.94581
0032	248	r(5)	r(5)	0.96822
0064	2	r(6)	r(6)	0.94690
0128	13	r(7)	r(7)	0.94901
0130	3	r(1,7)	$r1*p7+(1-r1)*q6*p7$	0.93993
0256	2	r(8)	r(8)	0.94794

After estimating the reliability of all software constructs, we use the defect content estimator described in Section 3.3 to estimate the number of defective constructs in files modified in the development stream. The list was then sorted in descending order based on the estimated number of defective constructs in each file. In the next section we discuss the results of the case study by comparing the SDPM estimates with the actual SLOC changes during final system testing.

#### 4.5.4 Case Study Results

In this section we compare the number of defective constructs estimated by the SDPM with the number of constructs modified in each file during final system testing of the DIS/CS 18.5 release. Table 20 shows the defect-prone files in descending order. The first column shows the file names, the second file size, the third gives the magnitude of change in each file in SLOCs. The fourth and fifth columns represent the number of defective SLOCs based on the SDPM estimator and the observed SLOC changes during final system testing respectively. We use the coefficient of correlation to assess the performance of the SDPM with the observed number of

defective SLOCs in each file. We also use the coefficient of correlation to show that the SDPM provides a better estimate than size of change alone.

**Table 20: Case Study Results – DIS/CS 18.5**

<b>File Name</b>	<b>SLOC</b>	<b>Churn</b>	<b>Est. # of Defective SLOCs per file</b>	<b>Observed # of Defective SLOC per File</b>
de_100000.PCF	8781	92	8.5376	12
de_44110.PCF	1100	150	6.255	#N/A
de_ctlFPPView.vb	163	163	5.1834	#N/A
de_46122.PCF	684	121	5.0457	4
de_11500.PCF	866	411	2.6304	#N/A
de_13141.PCF	732	387	2.4768	#N/A
de_11509.PCF	742	348	2.2272	#N/A
de_11508.PCF	738	346	2.2144	#N/A
de_13420.pcf	662	332	2.1248	#N/A
de_ctlFPPView.Designer.vb	66	66	2.0988	#N/A
de_11540.PCF	648	304	1.9456	#N/A
de_46125.PCF	934	41	1.7097	2
de_13170.PCF	472	214	1.3696	#N/A
de_11501.PCF	456	206	1.3184	#N/A
rp_write_assembled_transport_data.cpp	1039	17	0.8755	#N/A
de_cls13141.vb	72	20	0.8689	#N/A
de_46121.PCF	1578	18	0.7506	#N/A
de_43110.PCF	1844	16	0.6672	1
de_44400.PCF	430	13	0.5421	#N/A
de_mod44110fn.vb	185	10	0.417	#N/A
de_11900.PCF	946	60	0.384	#N/A
de_frmipde.designer.vb	244	12	0.3816	#N/A
rp_EOD_tapes.cpp	1532	5	0.2575	#N/A
de_mod46127fn.vb	122	6	0.2502	#N/A
de_frmipde.vb	557	7	0.2226	#N/A
de_clssection03.vb	100	5	0.2085	1
de_11910.PCF	512	31	0.1984	#N/A
rp_write_assembled_transport_data.h	72	3	0.1545	#N/A
de_mod13141fn.vb	130	22	0.1408	#N/A
de_clssection04.vb	42	3	0.1251	#N/A

de_mod13420fn.vb	119	19	0.1216	#N/A
sp_format_on_line_grader_report.cpp	1415	2	0.1042	#N/A
TaxClassMap.cs	143	2	0.103	#N/A
de_clsprogram45500.vb	530	2	0.103	#N/A
de_enumcommonsectionfieldnumbers.vb	1606	1	0.0928	#N/A
de_mod46125fn.vb	196	2	0.0834	#N/A
de_11511.PCF	368	10	0.064	#N/A
de_clssection05.vb	33	1	0.0542	1
de_clssection01.vb	36	1	0.0531	#N/A
de_35713.PCF	416	1	0.0531	#N/A
de_clstaxpr33.vb	79	1	0.0417	#N/A
de_clssection05.vb	32	1	0.0417	1
de_clssection04.vb	49	1	0.0417	#N/A
de_12220.PCF	878	5	0.032	#N/A
de_cls13420.vb	52	4	0.0256	#N/A
de_clssection13.vb	4	4	0.0256	#N/A
de_cls13170.vb	92	3	0.0192	#N/A
de_clssection21.vb	3	3	0.0192	#N/A
de_cls12220.vb	57	3	0.0192	#N/A
de_cls12200.vb	57	3	0.0192	#N/A
de_cls12100.vb	54	3	0.0192	#N/A
de_cls11900.vb	55	3	0.0192	#N/A
de_mod11900fn.vb	164	3	0.0192	#N/A
de_cls11540.vb	55	3	0.0192	#N/A
de_cls11511.vb	54	3	0.0192	#N/A
de_cls11509.vb	53	3	0.0192	#N/A
de_cls11508.vb	51	3	0.0192	#N/A
de_cls11503.vb	55	3	0.0192	#N/A
de_cls11502.vb	52	3	0.0192	#N/A
de_cls11501.vb	92	3	0.0192	#N/A
de_cls11500.vb	52	3	0.0192	#N/A
de_clsForm8941v2.vb	3	3	0.0192	#N/A
de_12200.PCF	596	3	0.0192	#N/A
de_12100.PCF	694	3	0.0192	#N/A
de_11503.PCF	354	3	0.0192	#N/A
de_11502.PCF	440	3	0.0192	#N/A
de_mod11511fn.vb	66	1	0.0064	#N/A

Table 21 shows the coefficient of correlation between size of change (churn), SDPM estimate and number of defective SLOCs. Based on Table 21, the SDPM provides a

good estimate of the number of defective SLOCs. Judging by the coefficient of correlation in the table, the SDPM provides a better estimate than the churn alone. Figure 26 shows the estimated number of defective constructs in each file and the number of observed SLOCs modified during final system testing.

**Table 21: Correlation Analysis – DIS/CS 18.5**

	<i>Churn</i>	<i>Est. # of Defective SLOCs per file</i>	<i>Observed SLOC Changes During Final System Testing</i>
Churn	1		
Est. # of Defective SLOCs per file	0.54716	1	
Observed SLOC Changes During Final System Testing	<b>0.69641</b>	<b>0.95450</b>	1

#### 4.5.5 Poisson Regression Model Results

We used defect data from releases 10.4 to 18.4 to estimate the number of defects in Release 18.5 files. To fit the data, we used the Poisson regression model as described in Section 4.1.1. Similar to case study 1 and 2, the predictor variables used in this case study were logarithm of the SLOCs, square root of prior defects, age, and file status (New, Changed, and Unchanged). Table 24 shows the regression coefficients. As expected, the values of the coefficients of regression are similar to the coefficients estimated in case study 1 and 2 because the files share the same structural measures.

**Table 22: Coefficient of Regression – Case Study 3 - (Poisson Regression)**

Coefficient	Estimate	Std. Error	L-R ChiSquare	Prob> ChiSq	Lower CL	Upper CL
Intercept	-0.7475	0.0712	114.7925	0.0000	-0.9415	-0.6084
logSLOC	0.1902	0.0149	163.3089	0.0000	0.1610	0.2194
SqrtPriorD	-0.4359	0.0273	267.7662	0.0000	-0.4896	-0.3826
Age	-0.1020	0.0046	786.4533	0.0000	-0.1111	-0.0932
New[0]	-1.8001	0.0385	2975.5713	0.0000	-1.8763	-1.7253
Changed[0]	-1.9973	0.0343	4554.6000	0.0000	-2.0652	-1.9308
Unchanged[0]	0.0000	.	.	.	.	.

We used the coefficients of regression to estimate the expected number of defects per file in Release 18.5 and identify files that will most likely be defective. The results are shown in Table 23 below. As this table indicates, the Poisson model did not perform well to identify defect-prone files, based on the defects observed during the final system testing. Once again, by reviewing the results from the SDPM, the defect prone files in this case study were non-executable files that can't be detected by Regression models. In fact, the estimate was so poor that no coefficient of correlation could be calculated.

**Table 23: Estimated Number of Defects-Case Study 3 - (Poisson Model)**

File Name	Defects	Status (Changed, Unchanged, New)			Age	Log (SLOC)	Sqrt (Prior Defects)	Poisson Model
\de_ctIFPPView.vb	NA	0	0	1	0	5.09	1.00	0.1334
\de_ctIFPPView.Designer.vb	NA	0	0	1	0	4.19	1.00	0.1123
\de_clssection13.vb	NA	0	0	1	0	1.39	1.00	0.0659
\de_clssection21.vb	NA	0	0	1	0	1.10	1.00	0.0624
\de_clsForm8941v2.vb	NA	0	0	1	0	1.10	1.00	0.0624
\TaxClassMap.cs	NA	1	0	0	14	3.89	1.41	0.0174
\de_mod46120fn.vb	NA	1	0	0	23	5.51	1.00	0.0114
\de_mod46125fn.vb	NA	1	0	0	23	5.28	1.00	0.0109



\de_mod46127fn.vb	NA	1	0	0	23	4.80	1.00	0.0099
\de_mod13420fn.vb	NA	1	0	0	23	4.78	1.00	0.0099
\de_mod43110fn.vb	NA	1	0	0	23	5.67	1.41	0.0098
\de_clssection01.vb	NA	1	0	0	23	5.78	1.73	0.0087
\de_cls44110.vb	NA	1	0	0	23	4.89	1.41	0.0084
\de_clssection03.vb	1	1	0	0	23	4.62	1.41	0.0080
\de_mod11900fn.vb	NA	1	0	0	24	5.10	1.41	0.0079
\de_clssection01.vb	NA	1	0	0	23	3.58	1.00	0.0079
\de_clssection01.vb	NA	1	0	0	23	5.69	2.00	0.0076
\de_mod13141fn.vb	NA	1	0	0	24	4.87	1.41	0.0076
\de_cls13420.vb	NA	1	0	0	23	3.95	1.41	0.0070
\de_mod44110fn.vb	NA	1	0	0	23	5.22	2.00	0.0069
\de_clssection01.vb	NA	1	0	0	23	5.59	2.24	0.0067
\de_clssection05.vb	1	1	0	0	23	3.50	1.41	0.0065
\de_cls11503.vb	NA	1	0	0	24	4.01	1.41	0.0064
\de_cls11900.vb	NA	1	0	0	24	4.01	1.41	0.0064
\de_cls11500.vb	NA	1	0	0	24	3.95	1.41	0.0064
\de_cls11502.vb	NA	1	0	0	24	3.95	1.41	0.0064
\de_cls11501.vb	NA	1	0	0	24	4.52	1.73	0.0062
\de_clssection04.vb	NA	1	0	0	23	3.89	1.73	0.0061
\de_cls13141.vb	NA	1	0	0	24	4.28	1.73	0.0059
\de_clssection04.vb	NA	1	0	0	23	3.74	1.73	0.0059
\de_clsprogram45500.vb	NA	1	0	0	24	6.27	2.65	0.0058
de_enumcommonsectionfieldnumbers.vb	NA	1	0	0	29	7.39	2.00	0.0057
\de_clssection05.vb	1	1	0	0	23	3.47	1.73	0.0056
\de_mod11511fn.vb	NA	1	0	0	26	4.19	1.41	0.0054
\de_cls13170.vb	NA	1	0	0	26	4.52	1.73	0.0050
\de_clssection02.vb	NA	1	0	0	23	2.08	1.41	0.0049
\de_clssection02.vb	NA	1	0	0	23	2.08	1.41	0.0049
\de_cls12220.vb	NA	1	0	0	26	4.04	1.73	0.0046
\de_cls11540.vb	NA	1	0	0	26	4.01	1.73	0.0046
\de_cls11511.vb	NA	1	0	0	26	3.99	1.73	0.0045
\de_cls12100.vb	NA	1	0	0	26	3.99	1.73	0.0045
\de_cls11509.vb	NA	1	0	0	26	3.97	1.73	0.0045
\de_cls11508.vb	NA	1	0	0	26	3.93	1.73	0.0045
\de_enummessages.vb	NA	1	0	0	29	5.23	2.00	0.0038
\de_cls12200.vb	NA	1	0	0	26	4.04	2.24	0.0037
\de_clssection01.vb	NA	1	0	0	26	5.37	3.00	0.0034

\rp_cddb_tape.h	NA	1	0	0	32	3.78	1.00	0.0033
\de_frmipde.designer.vb	NA	1	0	0	29	5.50	2.65	0.0030
cs_write_formatted_key_entry_data.cpp	NA	1	0	0	38	6.54	1.00	0.0030
\de_clssection03.vb	1	1	0	0	26	3.40	2.45	0.0030
\de_clssection01.vb	NA	1	0	0	26	4.63	3.00	0.0030
\de_clstaxpr33.vb	NA	1	0	0	29	4.37	2.24	0.0029
\de_clsform1065xs01.vb	NA	1	0	0	29	4.86	2.65	0.0027
\de_frmipde.vb	NA	1	0	0	29	6.32	3.32	0.0026
\de_clsimfeeiflookup.vb	NA	1	0	0	29	5.95	3.87	0.0019
\rp_write_assembled_transport_data.h	NA	1	0	0	48	3.04	0.00	0.0009
\rp_write_assembled_transport_data.cpp	NA	1	0	0	48	6.06	2.00	0.0006
\rp_EOD_tapes.cpp	NA	1	0	0	48	6.04	2.45	0.0005
\cs_export.h	NA	1	0	0	50	3.53	1.00	0.0005
sp_format_on_line_grader_report.cpp	NA	1	0	0	48	6.45	3.32	0.0004

## CCD 770R2- BMF Health Care Changes for PY 2011

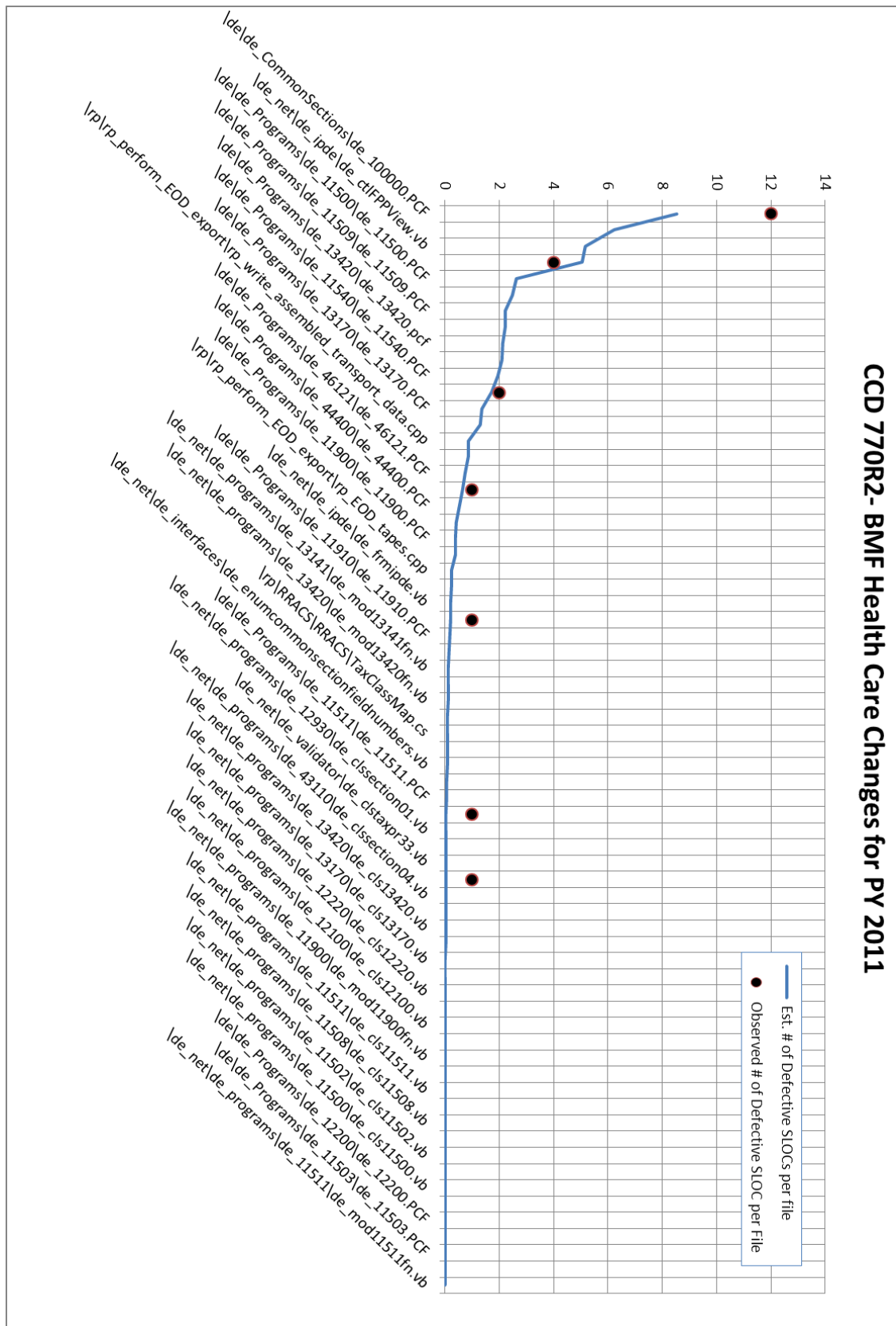


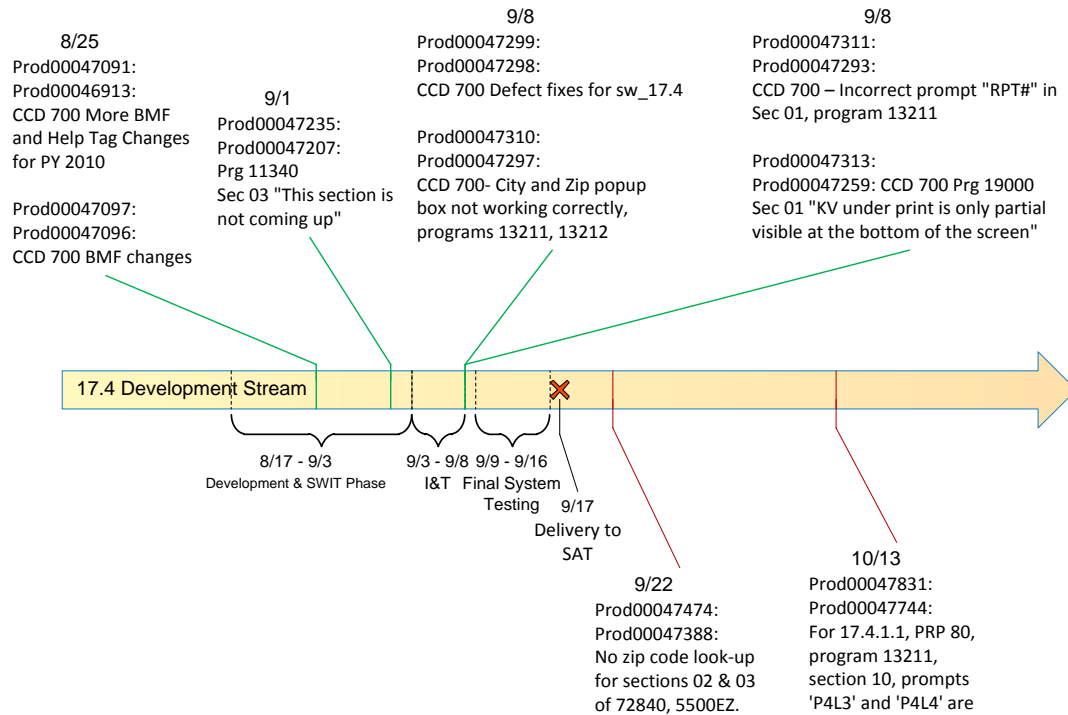
Figure 26: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 18.5

## **4.6 Case Study 4: CCD 700- More BMF and Help Tag Changes for PY 2010**

### **4.6.1 Software Project Background and History**

For this case study we selected a small software development project intended to only deliver one new function as part of the DIS/CS 17.4 release. Figure 27 shows the timeline of software development activities for this project. The development phase started on August 17, 2009 and ended on September 3, 2009. During the development phase, code changes were delivered in three change sets. The same change sets were also used to deliver code changes needed to address observed inspection and SWIT issues. What makes this case study different from other case studies is that a large number of files are modified to deliver only one enhancement which is shown below:

- CCD 700- More BMF and Help Tag Changes for PY 2010
  - Modify PRPs 11, 20, 21, 26, 39, 40, 41, 43, 44, 46, 49, 51, 52, 53, 55, 56, 57, 58, 80, 81, 82, 87, 88, 89
  - Modify 24 BMF programs and create 2 new BMF programs and Help Tag changes



**Figure 27: CCD 700 Timeline and Development Activities**

#### 4.6.2 Case Study Measurements

In the previous section, we provided a timeline of the software development activities related to the DIS/CS 17.4 software release. On August 25, 2009 the enhancements were delivered to the stream in two change sets 47091 and 47097. The change set 47097 also addressed 23 SLOCs that were identified as defective during the inspection of 47091. Change set 47235 was used to resolve an issue identified during the SWIT testing. Figure 28 shows the software change matrix for DIS/CS 17.4 release.

CS	(1)	(2)	(3)	(4)	(5)	(6)	(7)
0000001-0002	671						
0000010-0004		2178					
0000100-0008			3				
0001000-0016				294			
0001001-0018	8						
0010000-0032					43		
0010010-0036		23					
0100010-0068		1					
1000010-0132		2					

**Figure 28: Software Change Matrix – DIS/CS 17.4**

#### 4.6.3 Model Parameter Estimation

In this section we discuss how model parameters are calculated based on the measurements taken in each change set. Table 24 shows the summary of the measurements taken for each change set, along with the estimates of the model parameters. Once model parameters  $p_i$  and  $q_i$  are estimated, we calculate the change set reliabilities, which are shown in Column 8 of Table 24.

**Table 24: Model Parameters – DIS/CS 17.4**

Change Set	Size	Cap-Recap	(2) Est. New Defects	(1)Observed (Modified/Fixed)	p(i)	q(i)	r(i)
47091	679	Y	207	23	0.6951	0.1111	0.71869
47097	2204	N	110	3	0.9500	0.0100	0.95047
47235	3	N	0	39	0.9500	0.1299	0.95617
47299	302	N	15	23	0.9500	0.0729	0.95346
47310	66	N	3	1	0.9500	0.0031	0.95015
47311	1	N	0	2	0.9500	0.0063	0.95030
47313	2	N	0	0	0.9500	0.0000	0.95000

Table 25 shows the probability of constructs being defect-free based on the Binary Decision Diagram shown in Figure 29.

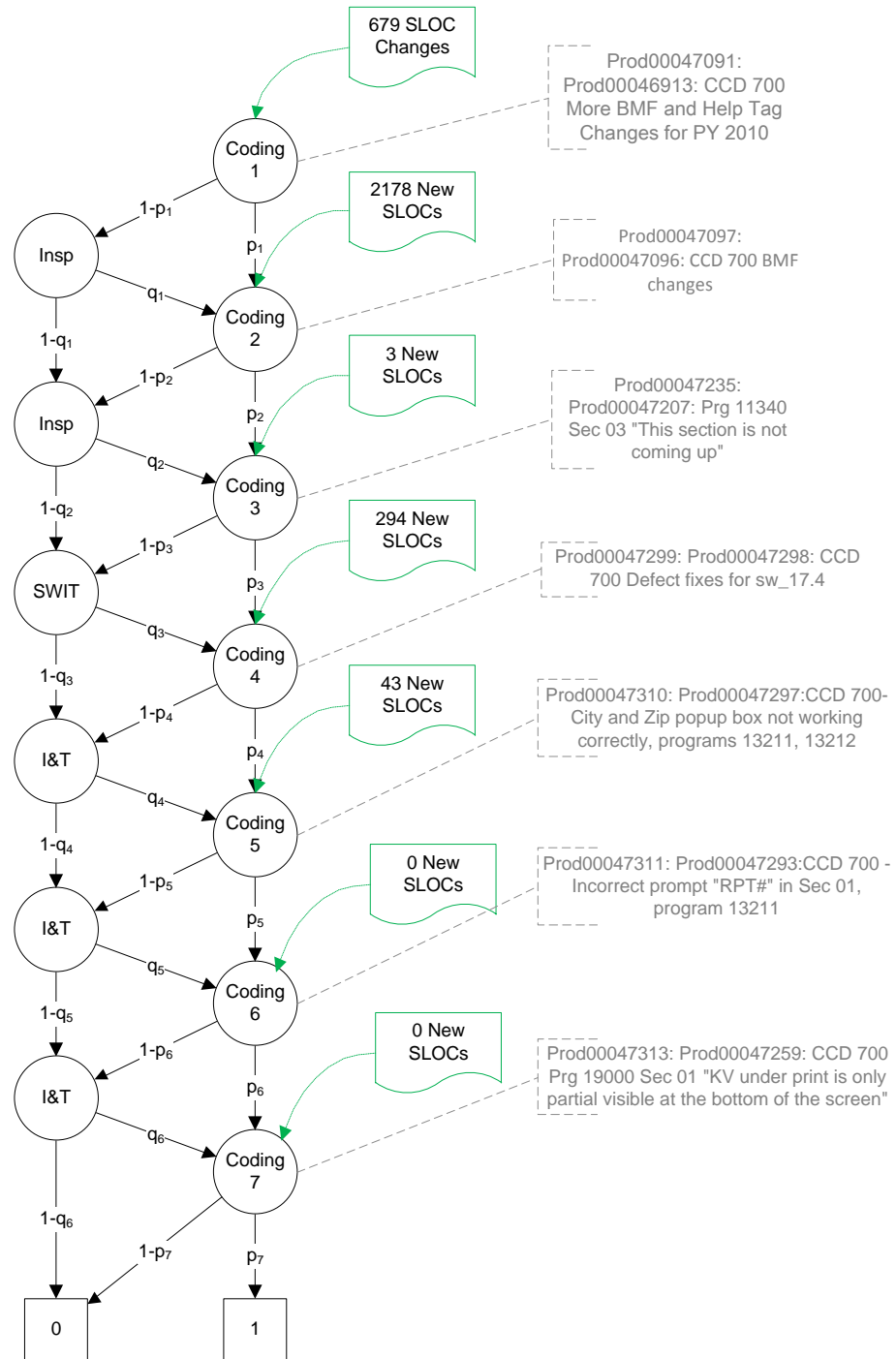


Figure 29: Binary Decision Diagram – DIS/CS 17.4

**Table 25: Construct Reliability Estimation – DIS/CS 17.4**

Change Sets	Churn	SDPM Model		Probability
0002	671	r(1)	r(1)	0.7187
0004	2178	r(2)	r(2)	0.9505
0008	3	r(3)	r(3)	0.9562
0016	294	r(4)	r(4)	0.9535
0018	8	r(1,4)	$r1*p4+(1-r1)*q3*p4$	0.7175
0032	43	r(5)	r(5)	0.9501
0036	23	r(1,5)	$r1*p5+(1-r1)*q4*p5$	0.7022
0068	1	r(1,6)	$r1*p6+(1-r1)*q5*p6$	0.6836
0132	2	r(1,7)	$r1*p7+(1-r1)*q6*p7$	0.6844

After estimating the reliability of all software constructs, we use the defect content estimator described in Section 3.3 to estimate the number of defective constructs in files modified in the development stream. The list is then sorted in descending order based on the estimated number of defective constructs in each file. In the next section, we discuss the results of the case study by comparing the SDPM estimates with the actual SLOC changes during final system testing.

#### 4.6.4 Case Study Results

In this section we compare the number of defective constructs estimated by the SDPM with the number of constructs modified in each file during final system testing of the DIS/CS 17.4 release. Table 26 shows the defect-prone files in descending order. The first column shows the file names, the second shows the file size, and the third gives the magnitude of change in each file in SLOCs. The fourth and fifth columns represent the number of defective SLOCs based on the SDPM estimator and the observed SLOC changes during final system testing respectively. We use the coefficient of correlation to assess the performance of the SDPM with the observed



number of defective SLOCs in each file. We also use the coefficient of correlation to show that the SDPM provides a better estimate than change alone.

**Table 26: Case Study Results – DIS/CS 17.4**

<b>File name</b>	<b>SLOC</b>	<b>Churn</b>	<b>Est. # of Defects Per File</b>	<b>Observed Defects</b>
de_13410.pcf	1654	250	68.625	13
de_13212.pcf	492	492	24.354	#N/A
de_13211.pcf	390	390	19.305	2
de_12100.PCF	686	64	17.568	#N/A
de_12300.PCF	646	303	16.1241	#N/A
de_12500.PCF	676	315	15.5925	#N/A
de_15540.pcf	174	44	12.078	#N/A
de_cls45blank.vb	41	41	11.2545	#N/A
de_15560.PCF	172	38	10.431	#N/A
de_12402.PCF	374	38	10.431	#N/A
de_11330.pcf	286	35	9.6075	#N/A
de_11340.pcf	400	186	9.1896	#N/A
de_enumcommonsectionfieldnumbers.vb	1522	73	9.1855	#N/A
de_19000.PCF	248	112	6.0632	#N/A
de_11507.PCF	372	22	6.039	#N/A
de_16010.PCF	210	20	5.49	#N/A
de_clsform8038xs01.vb	103	103	5.3589	2
de_59600.PCF	172	18	4.941	#N/A
de_12404.PCF	220	90	4.455	#N/A
de_12403.PCF	212	86	4.257	#N/A
de_mod13212fn.vb	84	84	4.158	#N/A
de_12410.PCF	464	12	3.294	#N/A
de_mod13211fn.vb	66	66	3.267	#N/A
de_13200.PCF	2544	66	3.267	#N/A
de_11800.PCF	1300	10	2.745	#N/A
de_cls13211.vb	50	50	2.475	#N/A
de_11100.PCF	752	9	2.4705	#N/A
de_cls13212.vb	48	48	2.376	#N/A
de_12701.PCF	190	7	1.9215	#N/A
de_12201.PCF	270	6	1.647	#N/A
de_mod12701fn.vb	32	5	1.3725	#N/A
de_mod12402fn.vb	67	5	1.3725	#N/A

de_mod12300fn.vb	110	5	1.3725	#N/A
de_12702.PCF	204	4	1.098	#N/A
de_12310.PCF	2234	4	1.098	#N/A
de_mod15560fn.vb	31	3	0.8235	#N/A
de_mod15540fn.vb	31	3	0.8235	#N/A
de_mod13410fn.vb	287	3	0.8235	#N/A
de_mod11800fn.vb	223	3	0.8235	#N/A
de_12400.PCF	408	3	0.8235	#N/A
de_12320.PCF	592	3	0.8235	#N/A
de_mod12702fn.vb	32	2	0.549	#N/A
de_mod12410fn.vb	81	2	0.549	#N/A
de_mod12400fn.vb	70	2	0.549	#N/A
de_mod12201fn.vb	48	2	0.549	#N/A
de_mod12100fn.vb	114	2	0.549	#N/A
de_mod11330fn.vb	47	2	0.549	#N/A
de_mod11100fn.vb	124	2	0.549	#N/A
assemblyinfo.vb	11	11	0.5445	#N/A
assemblyinfo.vb	11	11	0.5445	#N/A
de_mod12404fn.vb	43	11	0.5445	#N/A
de_mod12403fn.vb	39	8	0.396	#N/A
de_71700.PCF	190	8	0.396	#N/A
de_mod71700fn.vb	32	7	0.3465	#N/A
de_mod12320fn.vb	103	1	0.2745	#N/A
de_mod19000fn.vb	43	4	0.198	1
de_clssection02.vb	4	4	0.198	1
de_clssection03.vb	4	4	0.198	1
de_clssection04.vb	4	4	0.198	#N/A
de_clssection10.vb	4	4	0.198	#N/A
de_clssection11.vb	4	4	0.198	#N/A
de_mod11340fn.vb	71	4	0.198	#N/A
de_mod13200fn.vb	438	3	0.1485	#N/A
de_mod12500fn.vb	112	2	0.099	#N/A

Table 27 shows the coefficient of correlation between size of change (churn), SDPM estimate and the number of defective SLOCs. Based on Table 27, the SDPM provides a good estimate for the number of defective SLOCs. Based on the coefficient of correlation in shown below, the SDPM provides a better estimate than

the churn alone. Figure 30 shows the estimated number of defective constructs in each file and the number of observed SLOCs modified during final system testing.

**Table 27: Correlation Analysis DIS/CS 17.4**

	<i>Churn</i>	<i>Est. # of Defective SLOCs</i>	<i>Observed # of Defective SLOCs During Final System Testing</i>
Churn	1		
Est. # of Defective SLOCs	0.61643	1	
Observed # of Defective SLOCs During Final System Testing	0.45062	<b>0.97922</b>	<b>1</b>

#### 4.6.5 Poisson Regression Model Results

We used defect data from releases 10.4 to 17.3 to estimate the number of defects in Release 17.4 files. To fit the data, we used the Poisson regression model as described in Section 4.1.1. Similar to case study 1, 2 and 3, the predictor variables used in this case study were logarithm of the SLOCs, square root of prior defects, age, and file status (New, Changed, and Unchanged). Table 28 shows the regression coefficients. As expected, the values of the coefficients of regression are similar or close to the coefficients estimated in previous case studies because the files share the same structural measures.

**Table 28: Coefficient of Regression – Case Study 4 - (Poisson Regression)**

<b>Coefficient</b>	<b>Estimate</b>	<b>Std. Error</b>	<b>L-R ChiSquare</b>	<b>Prob&gt; ChiSq</b>	<b>Lower CL</b>	<b>Upper CL</b>
Intercept	-0.5403	0.0574	91.3337	0.0000	-0.6531	-0.4282
Log(SLOC)	0.1623	0.0121	180.4330	0.0000	0.1386	0.1861

Sqrt(PriorDef)	-0.4873	0.0222	504.1644	0.0000	-0.5310	-0.4438
Age	-0.1195	0.0050	901.7879	0.0000	-0.1294	-0.1099
New[0]	-1.8633	0.0350	4270.4099	0.0000	-1.9327	-1.7953
Changed[0]	-2.1260	0.0314	7229.3271	0.0000	-2.1883	-2.0652
Unchanged[0]	0.0000	.	.	.	.	.

We used the coefficients of regression to estimate the expected number of defects per file in Release 17.4 and identify files that will most likely be defective. The results are shown in Table 23 below. As this table indicates, the Poisson regression model was able to identify executable files that were likely to be defect prone, leaving out non-executable files.

**Table 29: Estimated Number of Defects-Case Study 4 - (Poisson Model)**

File Name	Defects	Changed Unchanged New			Age	Log (SLOC)	Sqrt (Prior Defects)	M
de_clsform8038xs01.vb	2	0	0	1	0	4.63	0.00	0.1918
de_mod13211fn.vb	0	0	0	1	0	4.19	1.00	0.1096
de_cls13211.vb	0	0	0	1	0	3.91	1.00	0.1048
de_cls13212.vb	0	0	0	1	0	3.87	1.00	0.1041
de_clssection03.vb	0	1	0	0	2	3.81	0.00	0.1015
de_clssection03.vb	0	1	0	0	2	3.81	0.00	0.1015
de_cls45blank.vb	0	0	0	1	0	3.71	1.00	0.1015
assemblyinfo.vb	0	0	0	1	0	2.94	1.00	0.0895
assemblyinfo.vb	0	0	0	1	0	2.94	1.00	0.0895
de_cls44318.vb	0	1	0	0	1	4.14	1.00	0.0742
de_cls44317.vb	0	1	0	0	1	4.14	1.00	0.0742
de_clssection10.vb	0	0	0	1	0	1.39	1.00	0.0695
de_clssection11.vb	0	0	0	1	0	1.39	1.00	0.0695
de_clssection02.vb	0	0	0	1	0	1.39	1.00	0.0695
de_clssection03.vb	0	0	0	1	0	1.39	1.00	0.0695
de_clssection04.vb	0	0	0	1	0	1.39	1.00	0.0695
assemblyinfo.vb	0	1	0	0	1	2.94	1.00	0.0611
assemblyinfo.vb	0	1	0	0	1	2.94	1.00	0.0611

de_mod44318FN.vb	0	1	0	0	1	2.89	1.00	0.0606
de_mod44317fn.vb	0	1	0	0	1	2.77	1.00	0.0594
de_mod13212fn.vb	0	1	0	0	5	4.43	1.00	0.0482
de_clssection06fc1048.vb	0	1	0	0	1	1.39	1.00	0.0474
de_clssection06fc1049.vb	0	1	0	0	1	1.39	1.00	0.0474
de_clssection58.vb	0	1	0	0	1	1.10	1.00	0.0453
de_clssection57.vb	0	1	0	0	1	1.10	1.00	0.0453
de_clsSection57.vb	0	1	0	0	1	1.39	1.41	0.0388
de_clssection03.vb	0	1	0	0	6	2.94	1.00	0.0336
de_clssection02.vb	0	1	0	0	6	2.94	1.41	0.0275
de_mod13200fn.vb	0	1	0	0	13	6.08	1.00	0.0242
de_mod13410fn.vb	0	1	0	0	13	5.66	1.00	0.0226
de_mod46125fn.vb	0	1	0	0	13	5.21	1.00	0.0210
de_mod12320fn.vb	0	1	0	0	13	4.63	1.00	0.0192
de_mod44400fn.vb	1	1	0	0	13	4.45	1.00	0.0186
de_mod12410fn.vb	0	1	0	0	13	4.39	1.00	0.0184
de_mod43110fn.vb	0	1	0	0	13	5.58	1.41	0.0183
healthchecks.xml	0	1	0	0	15	5.46	1.00	0.0172
de_mod12500fn.vb	0	1	0	0	14	4.72	1.00	0.0172
de_mod12300fn.vb	0	1	0	0	14	4.70	1.00	0.0172
de_mod11330fn.vb	0	1	0	0	13	3.85	1.00	0.0169
de_cls44110.vb	0	1	0	0	13	4.96	1.41	0.0165
de_mod12400fn.vb	0	1	0	0	14	4.25	1.00	0.0160
de_mod12402fn.vb	0	1	0	0	14	4.20	1.00	0.0159
de_clssection03.vb	0	1	0	0	13	4.67	1.41	0.0158
de_clssection01.vb	1	1	0	0	13	4.49	1.41	0.0153
de_mod12201fn.vb	0	1	0	0	14	3.87	1.00	0.0150
FileVersionHealthCheck.cs	0	1	0	0	15	4.38	1.00	0.0145
de_mod11100fn.vb	0	1	0	0	14	4.82	1.41	0.0143
DatabaseBackupHealthCheck.cs	0	1	0	0	15	4.25	1.00	0.0142
DiskSpaceHealthCheck.cs	0	1	0	0	15	4.20	1.00	0.0141
de_mod71700fn.vb	0	1	0	0	14	3.47	1.00	0.0141
de_mod12702fn.vb	0	1	0	0	14	3.47	1.00	0.0141
DatabaseRowCountHealthCheck.cs	0	1	0	0	15	4.16	1.00	0.0140
EnvironmentVariableHealthCheck.cs	0	1	0	0	15	4.14	1.00	0.0139
de_cls43110.vb	0	1	0	0	13	5.65	2.00	0.0139

DatabaseScalarQueryHealthCheck.cs	0	1	0	0	15	4.09	1.00	0.0138
de_clssection04.vb	0	1	0	0	13	3.81	1.41	0.0137
ServiceStateHealthCheck.cs	0	1	0	0	15	4.03	1.00	0.0137
VerifyAutoPurgeHealthCheck.cs	0	1	0	0	15	3.97	1.00	0.0135
EventLogHealthCheck.cs	0	1	0	0	15	3.85	1.00	0.0133
de_mod11340fn.vb	0	1	0	0	14	4.26	1.41	0.0131
de_clssection05.vb	0	1	0	0	13	3.50	1.41	0.0130
de_mod44110fn.vb	0	1	0	0	13	5.20	2.00	0.0129
FolderReplicationHealthCheck.cs	0	1	0	0	15	4.63	1.41	0.0123
de_enumcommonsectionfieldnumbers.vb	2	1	0	0	19	7.33	1.41	0.0118
de_cls46125.vb	0	1	0	0	13	4.63	2.00	0.0118
de_clssection05.vb	0	1	0	0	13	3.66	1.73	0.0115
de_mod15560fn.vb	0	1	0	0	14	3.43	1.41	0.0114
de_mod12100fn.vb	0	1	0	0	16	4.74	1.41	0.0111
de_mod15540fn.vb	0	1	0	0	16	3.43	1.41	0.0090
de_mod47110fn.vb	1	1	0	0	16	4.28	1.73	0.0088
de_mod12404fn.vb	0	1	0	0	19	3.76	1.00	0.0081
de_mod12701fn.vb	0	1	0	0	19	3.47	1.00	0.0077
de_mod11800fn.vb	0	1	0	0	19	5.41	1.73	0.0074
de_cls47110.vb	0	1	0	0	16	4.19	2.24	0.0068
de_mod19000fn.vb	0	1	0	0	19	3.76	1.41	0.0066
de_enummessages.vb	0	1	0	0	19	5.23	2.00	0.0063
de_clssection03.vb	0	1	0	0	16	3.40	2.24	0.0060
de_ctlprpview.vb	0	1	0	0	19	5.45	2.24	0.0058
de_clssections.vb	0	1	0	0	19	5.45	2.24	0.0058
setupworkstationdatastores.bat	1	1	0	0	28	4.65	0.00	0.0052
de_clstaxpr31.vb	0	1	0	0	19	3.99	2.00	0.0052
de_clstaxpr15.vb	0	1	0	0	19	4.38	2.24	0.0049
de_mod12403fn.vb	0	1	0	0	19	3.66	2.00	0.0049
de_clstaxpr33.vb	0	1	0	0	19	4.37	2.24	0.0049
de_clscheduledec.vb	0	1	0	0	19	3.50	2.00	0.0048
cs_create_cddb.cpp	0	1	0	0	22	5.84	2.83	0.0033
EEIFDatabase.cs	0	1	0	0	26	6.73	2.24	0.0031
MainForm.cs	0	1	0	0	26	5.88	2.00	0.0030
rp_EOD_tapes_private.h	0	1	0	0	28	3.56	1.00	0.0027
cs_sql_eeif_initialize.cpp	0	1	0	0	29	5.35	1.41	0.0026

Install_ISRP.bat	0	1	0	0	25	3.95	2.65	0.0018
de_clsstatemachine.vb	2	1	0	0	19	7.93	5.74	0.0016
cs_ftp_export.cpp	0	1	0	0	36	4.53	1.73	0.0008
cs_store_ops.cpp	0	1	0	0	37	6.01	2.24	0.0007
cs_end_of_shift.cpp	1	1	0	0	37	6.02	2.83	0.0006
rp_perform_EOD_export.cpp	0	1	0	0	37	4.77	2.45	0.0005

We used the coefficient of correlation to compare the results of the SDPM with the Poisson regression model. As Table 30 indicates, the SDPM performed better than the Poisson regression model in identifying defect prone files.

**Table 30: Coefficient of Correlation – Poisson Model**

	<i>Estimated number of Defects</i>	<i>Observed Number of Defects</i>
Estimated number of Defects	1	
Observed Number of Defects	0.46747	1

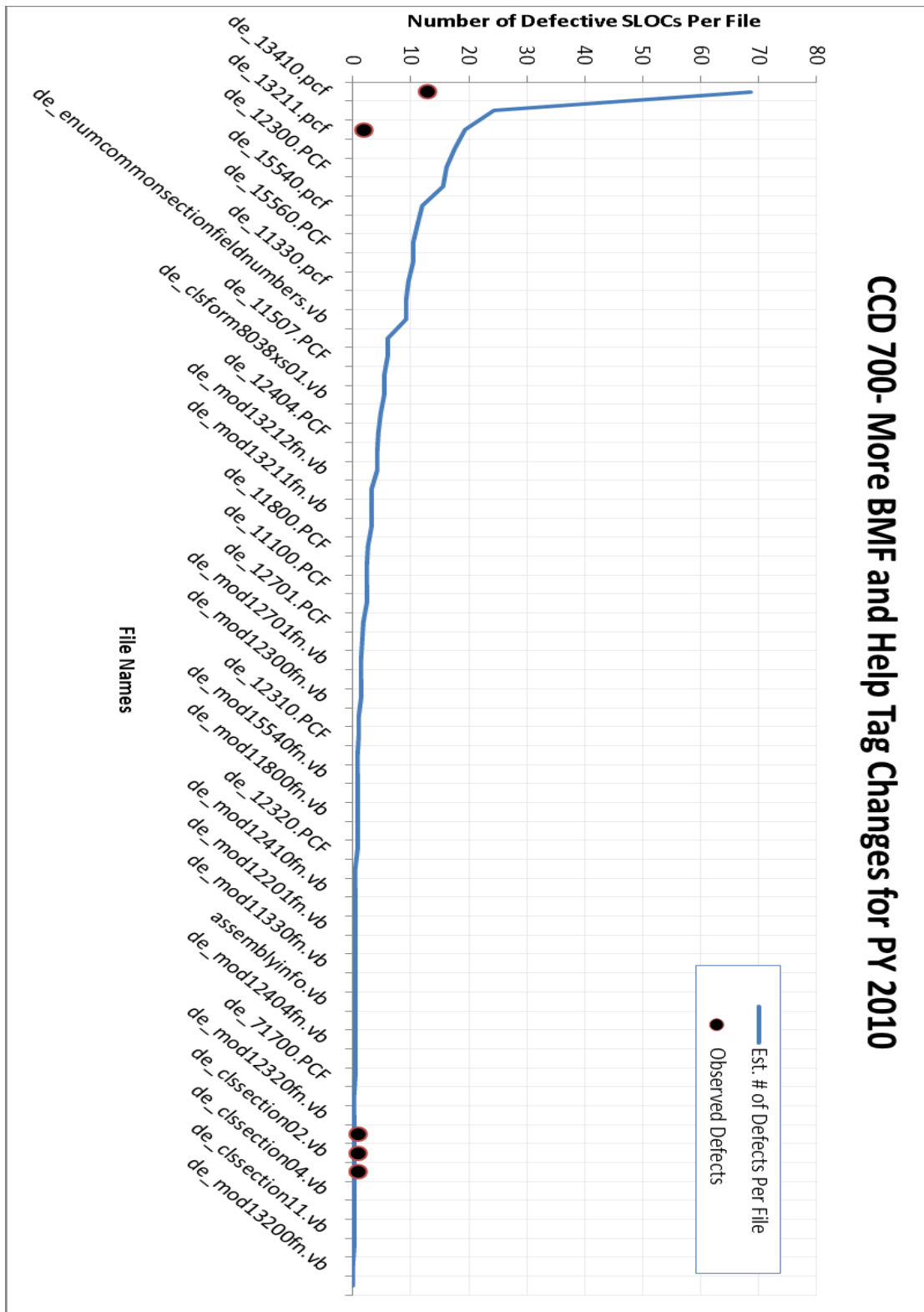


Figure 30: SDPM - Est. # of Defective SLOC vs. Observed Number of Defective SLOCs – DIS/CS 17.4

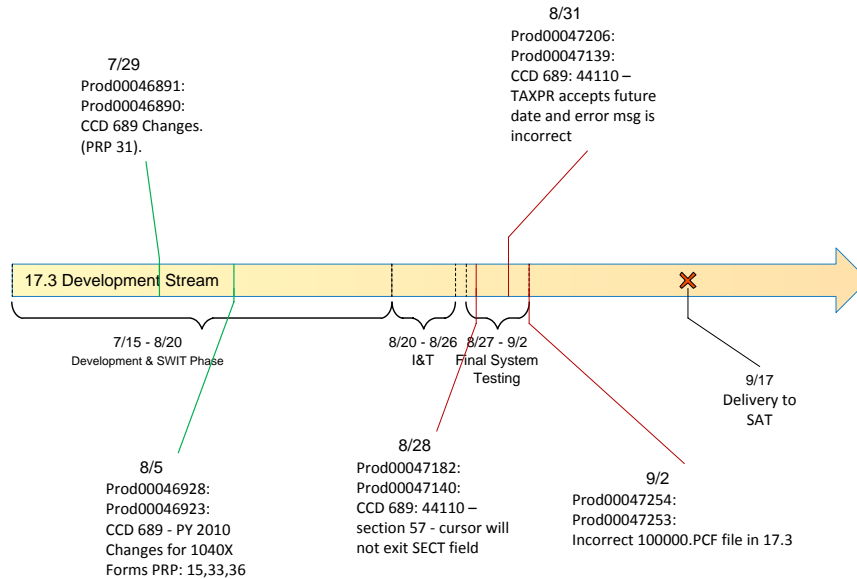


## **4.7 Case Study 5: CCD 689- IMF Changes for PY 2010**

### **4.7.1 Software Project Background and History**

For this case study we selected a software development project that is intended to deliver one enhancement as part of the DIS/CS 17.3 release. Figure 31 shows the timeline of software development activities for this project. The development phase started on July 15, 2009 and concluded on August 20, 2009. During the development phase, code changes were delivered in 2 change sets. The two change sets were also used to deliver code changes needed to address inspection defects. The first part of the code changes was delivered on July 29, 2009 modifying 1287 SLOCs. The second set of changes were delivered on August 5, 2009 modifying 577 SLOCs. From 577 SLOCs modified in change set 2, 200 overlapped with SLOCs modified in change set 1, 67 of which addressed defective SLOCs identified during the inspection process. In this case study no defects were identified during SWIT and I&T testing. CCD 689 was a relatively small project implementing one major enhancement. The major functions being delivered with DIS/CS 17.3 are listed below:

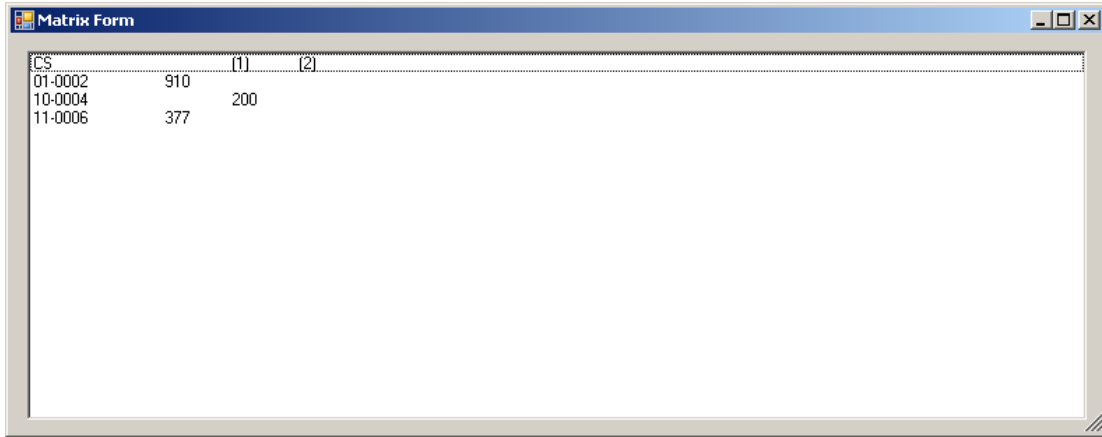
- CCD 689- IMF Changes for PY 2010
  - IMF changes for 2010; PRP 15, 31, 22 and 36 changes; 7 IMF programs impacted.



**Figure 31: CCD 689 Timeline and Development Activities**

#### 4.7.2 Case Study Measurements

In previous section, we provided a timeline of the software development activities related to DIS/CS 17.3 software release. On July 29, 2009 the first set of enhancements was delivered to the stream. A formal inspection was held and inspection findings along with the implementation of the second set of enhancements were delivered on August 5, 2009. Out of 1287 SLOC changes delivered in the first change set 377 were reworked in change set 2. Figure 32 shows the software change matrix for DIS/CS 17.3 release. In change set 2, 200 additional SLOCs were updated to implement the second set of changes needed for this release.



CS	(1)	(2)
01-0002	910	200
10-0004	377	200
11-0006	377	200

**Figure 32: Change Set Matrix – DIS/CS 17.3**

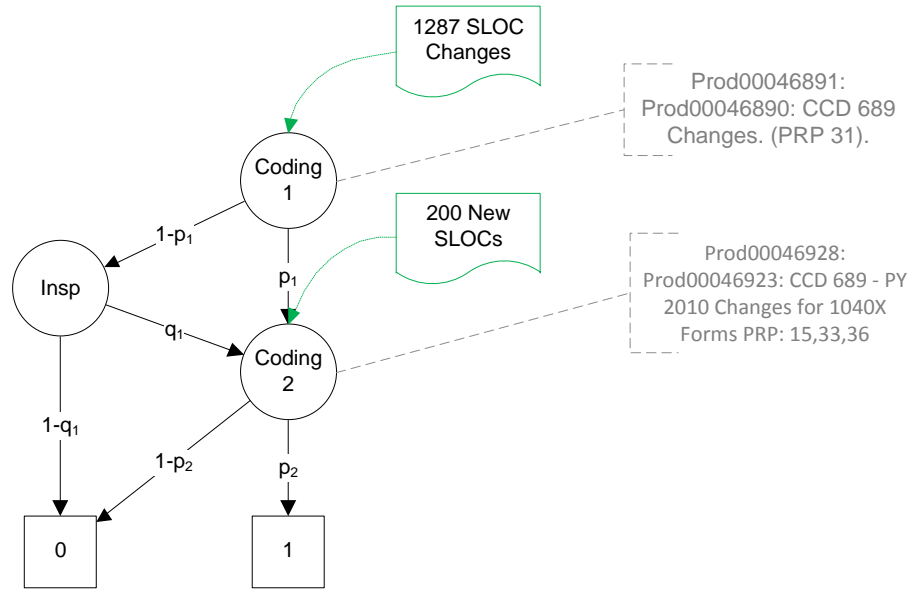
#### 4.7.3 Model Parameter Estimation

In this section we will discuss how model parameters are calculated based on the measurements taken in each change set. Table 31 shows the summary of the measurements taken for each change set along with the estimates of the model parameters. Once model parameters  $p_i, q_i$  are estimated, we calculate the change set reliabilities, which is shown in column 8 of Table 31.

**Table 31: Model Parameters – DIS/CS 17.3**

Change Set	Size	Cap-Recap	(2) Est. New Defects	(1) Observed (to be reworked)	q(i)	p(i)	r(i)
46891	1287	Y	132	67	0.50758	0.89744	0.94949
46928	200	N	21	0	0.00000	0.89744	0.89744

Table 31 shows the probability of constructs being defect free based on the Binary Decision Diagram shown in Figure 33.



**Figure 33: Binary Decision Diagram – DIS/CS 17.3**

#### 4.7.4 Case Study Results

In this section we will compare the estimated number of defective constructs estimated by SDPM with the number of constructs modified in each file during final system testing of DIS/CS 17.3 release. Table 32 shows the defect-prone files in descending order. The first column shows the file names, the second column shows the file size, the third column gives the magnitude of change in each file in SLOCs. The third and fourth columns represent the estimated number of defective SLOCs based on the SDPM estimator and the observed SLOC changes during final system testing respectively. We use the coefficient of correlation to assess the performance of SDPM with the observed number of defective SLOCs in each file. We also use the coefficient of correlation to show that SDPM provides a better estimate than the churn alone.

**Table 32: Case Study Results – DIS/CS 17.3**

File Name	SLOC	Churn	Est. # of Defective SLOCs	Observed SLOC changes during final system testing
de_100000.PCF	8297	628	59.867	522
de_43110.PCF	1710	273	13.7865	1
de_44110.PCF	1090	119	12.2094	1
de_46121.PCF	1440	148	7.474	
de_46125.PCF	874	98	4.949	
de_46122.PCF	676	40	4.104	
de_mod43110fn.vb	266	31	1.5655	
de_clscheduledec.vb	33	30	1.515	
de_47110.PCF	430	14	1.4364	
de_mod44110fn.vb	182	12	1.2312	1
de_mod46121fn.vb	226	23	1.1615	
de_44400.PCF	402	22	1.111	
de_cls44110.vb	142	3	0.3078	
de_cls46121.vb	179	5	0.2525	
de_clssection05.vb	36	5	0.2525	
de_cls43110.vb	283	5	0.2525	
de_cls47110.vb	66	2	0.2052	
de_mod46122fn.vb	120	2	0.2052	
de_mod46125fn.vb	184	4	0.202	
de_clsSection57.vb	4	4	0.202	
de_clssection57.vb	3	3	0.1515	
de_clssection58.vb	3	3	0.1515	
de_clstaxpr15.vb	80	1	0.1026	
de_clstaxpr31.vb	54	1	0.1026	
de_clstaxpr33.vb	75	1	0.1026	
de_clssection03.vb	30	1	0.1026	
de_mod47110fn.vb	72	1	0.1026	
de_clssection03.vb	107	1	0.1026	
de_clssection04.vb	45	2	0.101	
de_clssection05.vb	39	2	0.101	
de_cls46125.vb	103	1	0.0505	
de_clssection05.vb	33	1	0.0505	
de_mod44400fn.vb	86	1	0.0505	
de_cls46125.vb	103	1	0.0266	
de_clssection05.vb	33	1	0.0266	
de_mod44400fn.vb	86	1	0.0266	

Table 33 shows the coefficient of correlation between size of change (churn), SDPM estimate and the number of defective SLOCs. Based on Table 33 SDPM provides a good estimate for the number of defective SLOCs. Based on the coefficient of correlation in Table 33, SDPM provides a better estimate than the churn alone. Figure 34 shows the estimated number of defective constructs in each file and the number of observed SLOCs modified during final system testing.

**Table 33: Correlation Analysis DIS/CS 17.3**

	<i>Churn</i>	<i>Est. # of Defective SLOCs</i>	<i>Observed SLOC changes during final system testing</i>
Churn	1		
Est. # of Defective SLOCs	0.976673	1	
Observed SLOC changes during final system testing	<b>0.917232</b>	<b>0.976679</b>	1

#### 4.7.5 Poisson Regression Model Results

We used defect data from releases 10.4 to 17.2 to estimate the number of defects in Release 17.3 files. To fit the data, we used the Poisson regression model as described in Section 4.1.1. Similar to previous case studies the predictor variables used in this case study were logarithm of the SLOCs, square root of prior defects, age, and file status (New, Changed, and Unchanged). Table 34 shows the regression coefficients. As expected, the values of the coefficients of regression are similar or close to the

coefficients estimated in previous case studies because the files share the same structural measures.

**Table 34: Coefficient of Correlation – Poisson Regression**

<b>Coefficient</b>	<b>Estimate</b>	<b>Std. Error</b>	<b>L-R ChiSquare</b>	<b>Prob&gt; ChiSq</b>	<b>Lower CL</b>	<b>Upper CL</b>
Intercept	-0.53660	0.05814	87.76055	0.00000	-0.65091	-0.42299
Log(SLOC)	0.16125	0.01227	173.43862	0.00000	0.13721	0.18531
Sqrt(Prior Def)	-0.48666	0.02250	490.90084	0.00000	-0.53086	-0.44264
Age	-0.11875	0.00509	843.19031	0.00000	-0.12888	-0.10893
New[0]	-1.86038	0.03550	4137.83521	0.00000	-1.93065	-1.79148
Changed[0]	-2.12136	0.03180	7011.85550	0.00000	-2.18446	-2.05978
Unchanged[0]	0.00000	.	.	.	.	.

We used the coefficients of regression to estimate the expected number of defects per file in Release 17.3 and identify files that will most likely be defective. The results are shown in Table 35 below. As this table and the SDPM analysis indicate, the SDPM performed well in identifying defect prone files based on the software development activities from the current project, but failed to identify latent defects that already existed in the software product. On the other hand, while the Poisson regression model was able to identify one a newly created file as defective, it failed to identify most defect-prone files.

**Table 35: Estimated Number of Defects-Case Study 5 - (Poisson Model)**

File Name	Defects	Changed Unchanged New			Age	Log (SLOC)	Sqrt (Prior Defects)	Poisson Model
de_clsSection57.vb	2	0	0	1	0	1.39	0.00	0.11379
de_cls44318.vb	0	0	0	1	0	4.14	1.00	0.10909
de_cls44317.vb	0	0	0	1	0	4.14	1.00	0.10909
assemblyinfo.vb	0	0	0	1	0	2.94	1.00	0.08992
assemblyinfo.vb	0	0	0	1	0	2.94	1.00	0.08992
de_mod44318FN.vb	0	0	0	1	0	2.89	1.00	0.08914
de_mod44317fn.vb	0	0	0	1	0	2.77	1.00	0.08746
de_clssection06fc1049.vb	0	0	0	1	0	1.39	1.00	0.06994
de_clssection06fc1048.vb	0	0	0	1	0	1.39	1.00	0.06994
de_clssection58.vb	0	0	0	1	0	1.10	1.00	0.06677
de_clssection57.vb	0	0	0	1	0	1.10	1.00	0.06677
de_mod46125fn.vb	0	1	0	0	12	5.21	1.00	0.02402
de_mod44312fn.vb	0	1	0	0	11	4.13	1.00	0.02270
mod_registerlist.vb	0	1	0	0	11	5.06	1.41	0.02156
de_mod44303fn.vb	0	1	0	0	11	3.76	1.00	0.02140
de_mod44400fn.vb	0	1	0	0	12	4.45	1.00	0.02125
de_mod43110fn.vb	0	1	0	0	12	5.58	1.41	0.02084
mod_createlist.vb	0	1	0	0	11	4.84	1.41	0.02080
de_mod44313fn.vb	0	1	0	0	11	3.30	1.00	0.01985
de_cls44110.vb	0	1	0	0	12	4.96	1.41	0.01883
de_mod13131fn.vb	0	1	0	0	13	5.61	1.41	0.01858
de_clssection01.vb	0	1	0	0	12	3.61	1.00	0.01855
assemblyinfo.vb	0	1	0	0	11	2.71	1.00	0.01806
assemblyinfo.vb	0	1	0	0	11	2.71	1.00	0.01806
de_clssection03.vb	0	1	0	0	12	4.67	1.41	0.01799
de_clssection01.vb	0	1	0	0	12	3.18	1.00	0.01730
de_clssection01.vb	0	1	0	0	12	3.18	1.00	0.01730
de_mod11900fn.vb	0	1	0	0	13	5.12	1.41	0.01717
de_mod44110fn.vb	1	1	0	0	12	5.20	1.73	0.01679
de_cls43110.vb	0	1	0	0	12	5.65	2.00	0.01583
de_clssection04.vb	0	1	0	0	12	3.81	1.41	0.01565
de_mod35713fn.vb	0	1	0	0	13	4.29	1.41	0.01502
de_clssection05.vb	0	1	0	0	12	3.50	1.41	0.01489
de_enumcommonsectionfieldnumbers.vb	0	1	0	0	18	7.33	1.41	0.01354
de_cls46125.vb	0	1	0	0	12	4.63	2.00	0.01345
de_cls11680.vb	0	1	0	0	13	4.48	1.73	0.01326
de_clssection05.vb	0	1	0	0	12	3.66	1.73	0.01310
de_mod11509fn.vb	0	1	0	0	15	4.81	1.41	0.01289



de_mod11508fn.vb	0	1	0	0	15	4.81	1.41	0.01289
de_mod11540fn.vb	0	1	0	0	15	4.75	1.41	0.01277
de_ctlfielddisplayorder.designer.vb	0	1	0	0	18	5.93	1.41	0.01081
de_mod47110fn.vb	0	1	0	0	15	4.28	1.73	0.01013
de_ctlzerobalance.designer.vb	0	1	0	0	18	5.49	1.41	0.01007
de_ctlfields.designer.vb	0	1	0	0	18	6.26	1.73	0.00977
de_enummessages.vb	2	1	0	0	18	5.23	1.41	0.00965
de_POMDatastoreBuild.sql	0	1	0	0	18	4.64	1.41	0.00878
de_ctlfieldoutputorder.designer.vb	0	1	0	0	18	4.57	1.41	0.00869
de_CreateMessageLoader.bat	0	1	0	0	18	4.32	1.41	0.00833
de_cls47110.vb	0	1	0	0	15	4.19	2.24	0.00781
de_ctlenumerations.designer.vb	0	1	0	0	18	3.83	1.41	0.00770
de_ctlprpview.vb	1	1	0	0	18	5.45	2.00	0.00752
de_clssections.vb	1	1	0	0	18	5.45	2.00	0.00752
de_ctlsections.designer.vb	0	1	0	0	18	4.57	1.73	0.00744
de_clssection03.vb	0	1	0	0	15	3.40	2.24	0.00688
de_ctlfields.vb	0	1	0	0	18	6.78	2.65	0.00681
de_clstaxpr33.vb	1	1	0	0	18	4.37	2.00	0.00632
de_clstaxpr31.vb	0	1	0	0	18	3.99	2.00	0.00594
de_ctlsections.vb	0	1	0	0	18	5.89	2.65	0.00589
de_clstaxpr15.vb	0	1	0	0	18	4.38	2.24	0.00564
de_clsschedulec.vb	0	1	0	0	18	3.50	2.00	0.00549
de_frmipde.vb	0	1	0	0	18	6.30	3.16	0.00490
cs_dis_epmf_lookup.cpp	0	1	0	0	27	4.93	1.00	0.00386
EEIFDatabase.cs	0	1	0	0	25	6.73	2.24	0.00359
MainForm.cs	0	1	0	0	25	5.88	2.00	0.00351
isrp_build.bat	0	1	0	0	27	3.47	1.00	0.00305
sp_eop_ke3_processing.cpp	0	1	0	0	27	7.15	2.24	0.00303
cs_entity_check.cpp	0	1	0	0	28	5.30	1.41	0.00298
cs_eeif_lookup_private.h	0	1	0	0	28	4.67	1.41	0.00269
sp_remove_ghostblock.cpp	0	1	0	0	27	5.30	2.00	0.00252
sp_eop_ke3_processing_training_block.cpp	0	1	0	0	34	6.44	1.00	0.00215
sp_release_block.cpp	0	1	0	0	37	7.54	3.00	0.00068

We used the coefficient of correlation to compare the results of the SDPM with the Poisson regression model. By comparing the coefficient of correlation from Table 36 with the coefficient of correlation from SDPM provided in Table 33, we observe that

the SDPM performed better than the Poisson regression model in identifying defect prone files.

**Table 36: Coefficient of Correlation – Poisson Model**

	<i>Estimated # of Defects</i>	<i>Observed # of Defects</i>
Estimated # of Defects	1	
Observed # of Defects	0.63078	1

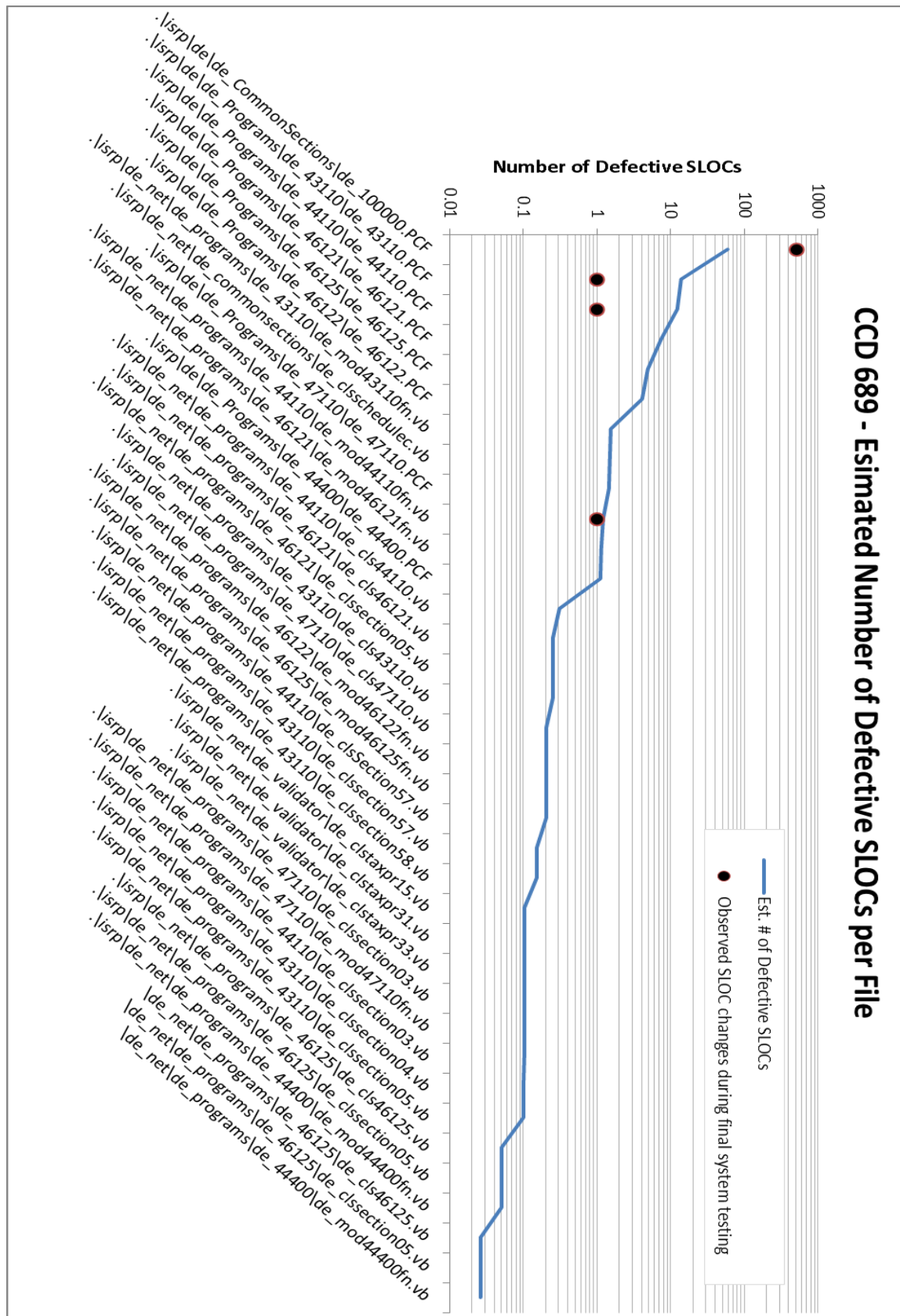


Figure 34: SDPM – Estimated # of Defective SLOCs vs. Observed # of Defective SLOCs – DIS/CS 17.3

#### **4.8 Case Study Conclusion**

In this chapter, we presented five industrial software development projects and studied how the Software Development Profile Model is used in real life projects. In each case study, we used the SDPM to estimate the number of defective constructs per file. We then compared the results with the number of SLOCs that were modified in each file during final system testing. To make this comparison valid, we excluded any code changes during the final system testing phase that were not related to the current development. We then analyzed the results using the coefficient of correlation between our estimate and the actual code changes and by comparing the ranking of files. In all five case studies the number of defective constructs estimated by SDPM was strongly correlated with the actual number of SLOCs modified during final system testing. Further, in all five case studies the number of SLOC changes during final system testing had a stronger correlation with the SDPM estimate than size of code change during development alone. This implies that software development process attributes should be considered in defect estimation. We sorted the files that were modified in each software development project in descending order and plotted them against the number of SLOCs modified in each file during final system testing. Again, the SDPM performed well by identifying defect prone files listed on top of the list.

Although we noticed a strong correlation between the SDPM estimates and the actual modified SLOCs during regression, its absolute predictive accuracy varied from project to project. Our investigation into this error shows the need to ensure the model closely matches the project. For example, the error can either be due to

inaccuracies in the estimation of total inspection defects or by failure to incorporate all evidence. In general, the SDPM performed best in Case Studies 1 through 4, where the requirement volatility was comparatively low. These four projects followed the waterfall model, where the requirements were finalized before development started. In Case Study 5, requirements were changed by the customer later in the development lifecycle, causing an unexpectedly large number of code changes to appear during final system testing.

We also used the Poisson regression model to evaluate the SDPM in comparison with an existing defect estimation model. As discussed in Section 4.1, a direct comparison was not possible, due of the differences in each model's measurement units and assumptions. In general, we observed that the SDPM performed better than the regression based model in identifying defect prone files in all five projects. The advantage of the SDPM is that it can estimate defect content of both executable and non-executable files. Since regression based models are based on defect data observed during the previous releases, they are unable to identify defects in non-executable files. The regression based model performed well in identifying latent defects that the SDPM was unable to identify due to lack change history and software development activities data from previous releases.

## Chapter 5: Summary of Contributions and Future Research Directions

### 5.1 Summary of Contributions

In Chapter 3, we introduced the Software Development Profile Model as a causal model for identifying defect prone software artifacts based on change history and software development activities. Rather than relying on defect data from previous projects or static software attributes to predict defect content, the SDPM assumes that human error during software development is the sole cause of software defects, and software development activities such as inspection, testing, and rework, further affect the total number of remaining software defects. Based on these assumptions, we proposed the SDPM as a causal model for estimating the number of defective constructs in software artifacts. Understanding the relationship between software development activities, change history and defect content can be crucial to the development of more reliable software products. It provides software managers with a framework for managing and adjusting software development activities more effectively. Rather than using defect data which is mostly available toward the end of the software development lifecycle, the SDPM can be used throughout the development process to measure defect content based on software development activities. Furthermore, using observations from an ongoing software development project provides more accurate defect prediction.

In Chapter 4, we investigated the relationship between the number of defective constructs estimated by the SDPM, and the number of defective constructs observed during final system testing using five real life software development projects. In all

five case studies we showed that the number of defective constructs estimated by the SDPM was strongly correlated to the actual number of SLOCs modified during final system testing. We also showed that the SDPM can be used to identify defect-prone software artifacts early in the development process without relying on defect data.

In Case Study 1, we show how additional evidence can be taken into account as it becomes available to update model parameters. We used the Bayesian Belief Network (BBN) to capture external factors and expert judgment to update the model parameters and provide a more accurate estimation.

## **5.2 Limitations of this Research**

In this section we discuss the limitations of the SDPM based on the model's assumptions and discuss future research directions. First, it is important to note that the number of remaining defects is not usually a direct measure of software reliability. A software program may contain many defects, each with a very low rate of occurrence, and such product can be more reliable than another software product which contains fewer defects each with a high rate of occurrence. Hence, the total rate of failure, that is the failure intensity of a software artifact, is a better measure that needs to be considered in the context of software reliability analysis. Similarly, we use the number of defective constructs in files as the measure of defect-proneness. We assume that files containing more defective constructs are more likely to be defective in production. While there is a correlation between the number of defective constructs in a file and its defect-proneness, considering the logical file structure and inter-modular coupling among constructs might provide a better measure of defect-proneness.

We also discussed dependencies among related software artifacts. We recognized that modifying one artifact can cause others to become defective. We captured this dependency by assuming that all related artifacts are known, included and reviewed during the inspection process. By including all related artifacts in the inspection, we assumed that we are able to estimate the number of defective constructs in related artifacts. Several models have been proposed to quantify the dependencies among related artifacts [21][51]. Modeling dependencies among software artifacts qualitatively rather than subjectively can improve the estimation, especially for larger software development projects or when file dependencies are unknown.

### **5.3 Future Research Directions**

Most existing software reliability models contain a parameter which represents the number of faults in the software. If the number of faults is assumed to be finite, then there is a need to estimate the number of remaining defects [64]. The SDPM can be used in conjunction with different software reliability models to estimate the reliability of the software product early in its lifecycle.

Further, the SDPM has not yet been used in software development project following agile methods. Agile methods break software development activities into small increments with minimal planning. Each increment allows a team to work through a full software development lifecycle, including requirements analysis, design, coding and testing. Since iterations are small, multiple iterations may be needed to deliver functionality. Because agile involves minimal planning, the SDPM can be used to identify defect-prone artifacts based on development activities and the size of each



change so that resources can be more effectively focused on defect-prone software artifacts.

In chapter 4 we recognized three types of dependencies, the dependency among constructs, the dependency among change sets and the external dependencies. In a software program there is also an additional dependency between artifacts. In software engineering, the term coupling is used to describe the degree to which software artifacts rely on each other. Low coupling is usually a sign of well-structured software program. Since coupling among artifacts can have ripple effect on other less defect-prone artifacts, modeling coupling as a dependency is a recommended future research topic.

In Chapter 4, we used Bayesian Belief Network (BBN) to capture the extrinsic dependencies. An example was provided to show how common environmental factors and local factors are used to update the model parameters and to provide a more accurate estimation. We did not discuss however, the importance measure of the external factor. Since not all factors affect the number of remaining defects equally, we recommend further sensitivity analysis of the external factors as future research area. Such sensitivity analysis can provide software managers with the tool needed to understand which factors can provide a better return on investment.

## Appendix A: Script Developed to Generate Change Sets

The script below was developed to examine the software stream to identify all activities in the change set and create a directory structure that can be used by the PET tool for SDPM analysis.

```
#####
#
#   Author: Brent Olson
#   Purpose:
#   Provided a baseline, examine that baseline to determine
#   its contents.
#
#   for each of the activities unique to that stream (meaning
#   that we exclude activities with equivalent check-ins
#   in earlier streams), create a directory structure that looks
#   like this:
#
#   compare_dirs
#       baseline
#           latest
#           previous
#       activity1
#           latest
#           previous
#       activity2
#           latest
#           previous
#       activity3
#           latest
#           previous
#       .
#       .
#       .
#       .
#
#   "latest" contains the latest versions of files touched by
#   that activity.  "previous" contains versions of the files
#   touched by the activity, but contains the version of the file
#   that existed before it was modified by that activity
#
#####

use CQPerlExt;
#use Win32::ODBC;
use Env "USERNAME";

#-----
#   set some base variables
#-----

my $pvob = "\\isrp_pvob";
my $temp_dir;
my $baseline;
```

```

if ( defined $ENV{"TMP"} ) {
    $temp_dir = $ENV{"TMP"};

    print "\n\n#####\n#          Copying files to: " .
$ENV{"TMP"} . "\\baseline_compare\n#####\n";

}
else {
    $temp_dir = "C:\\TEMP";
}

my $out_file = "$temp_dir\\baseline_compare\\copy_baseline_output.txt";
my $compare_directory = "baseline_compare";

if (! -d "$temp_dir\\baseline_compare") {
    mkdir ("$temp_dir\\baseline_compare") or die "\nERROR: cannot mkdir
\\$temp_dir\\baseline_compare\" because $!";
}

#-----
# verify input
#-----

if (! $ARGV[0]){
    usage("You must provide a baseline.");
}
else {
    $baseline = $ARGV[0];

    # if the baseline has an @, then it includes a pvob qualifier
    if ($baseline =~ /\@/) {
        ($baseline, $pvob) = split /\@/, $baseline;
    }

    print `cleartool lsbl $baseline\@$pvob 2>&l`;
    usage("Not a valid baseline") if ($?);
}

# -----
# remove the comparison directory and the output file
# -----

clean_up();

# -----
# copy out the entire baseline so that SLOCCO can look at it and give us
# details on how many total SLOC exist. We do this by using SLOCCO to compare
# the baseline against itself.
# -----
#$whole_baseline_file_hash = run_file_comparison_for_all_files($baseline);
#clean_up();

# -----
# now use the baseline to get a list of the activities included, exclude those
# from previous releases, and then copy out the relevant files and SLOCCO it
# we define activities of this release to include those things checked in for this
# baseline,
# but excluding those activities with corresponding checkins in earlier streams
# -----
my @all_activities = get_all_activities($baseline);

```

```

print "\nall activities means @{all_activities}";

my @release_activities =
activities_from_baseline_excluding_earlier_releases($baseline);
print "\njust the release activities means @{release_activities}";

# update the compare dir for the first copy
$compare_directory = "baseline_compare\\$baseline";

copy_files_for_these_activities(@release_activities);

print "\nDone copying everything, now I'm going to make copies for each of the
activities included";

foreach $act (@release_activities) {

    print "\n\nRunning for $act:";

    $compare_directory = "baseline_compare\\$act";

    my @one_act = ($act);
    copy_files_for_these_activities(@one_act);

    print "\n\tDone copying for $act ....";

}

$compare_directory = "baseline_compare\\$baseline";

print "\n\n#####\n#";
print "    Copying Complete!!  Please check $temp_dir\\baseline_compare for your
files...";
print "\n#####";

exit 0;

```

```

#-----
#-----
#  SUBROUTINES
#-----
#-----

```

```

#-----
# sub clean_up
#
#     remove the output file and the
#     comparison directories if they
#     exist (from the last time they
#     were run)
#-----

sub clean_up {

    # remove the output file if it exists already
    if (-f $out_file) {
        #print "\nRemoving the output file from the last time this was run...
(file: $out_file)";
        print `del /q /f \"$out_file\" 2>&1`;
        print "\nWarning: $out_file not removed!" if (-f $out_file);
    }
}

```

```

# remove the comparison directories if they already exist
if (-d "$temp_dir\\$compare_directory" ){
    #print "\nRemoving the comparison directory from the last time this was
run... (dir: $temp_dir\\$compare_directory)";
    print `rmdir /s/q \"${temp_dir}\\$compare_directory\" 2>&1`;

    print "\nWarning: $temp_dir\\$compare_directory not completely removed"
if (-d "$temp_dir\\$compare_directory" ) ;
}

# now, recreate the base that you've just removed, since we'll be runing this
thing multiple times and at different directory depths
mkdir ("${temp_dir}\\$compare_directory") or die "Can't mkdir on
$temp_dir\\$compare_directory because $!";

}

```

```

#-----
# sub get_cq_info_for
#
#-----

sub get_cq_info_for {

    my @activities = @_;

    #print "\n\nInside get_cq_info_for I have @{activities}";

    my %act_info;
    #my @INSPECTS = ("$inspection");

    my $CQsession = CQSession::Build();
    $CQsession->UserLogin("xxxxxxx", "xxxxxxx", "xxxxx", "");

    my $query_def_obj = $CQsession->BuildQuery("BaseCMActivity");
    my $filterOp = $query_def_obj-
>BuildFilterOperator($CQPerlExt::CQ_BOOL_OP_AND);
    # $filterOp->BuildFilter("Inspection_ID", $CQPerlExt::CQ_COMP_OP_LIKE,
    \@INSPECTS);
    $filterOp->BuildFilter("id", $CQPerlExt::CQ_COMP_OP_IN, \@activities);

    $query_def_obj->BuildField("id");
    $query_def_obj->BuildField("Inspection_ID");
    $query_def_obj->BuildField("Parent_Defect_Record");
    $query_def_obj->BuildField("Parent_Enhancement_Record");
    $query_def_obj->BuildField("Parent_Defect_Record.Resolution_new");
    $query_def_obj->BuildField("Parent_Defect_Record.Swit_Test_Status");
    $query_def_obj->BuildField("Parent_Defect_Record.Unit_Test_Status");
    $query_def_obj->BuildField("Parent_Enhancement_Record.Resolution_New");
    $query_def_obj->BuildField("Parent_Enhancement_Record.Swit_Test_Status");
    $query_def_obj->BuildField("Parent_Enhancement_Record.Unit_Test_Status");
    $query_def_obj->BuildField("Headline");
    $query_def_obj->BuildField("State");
    $query_def_obj->BuildField("Owner");
    $query_def_obj->BuildField("ucm_stream");

    # unfortunately, querying on the State fields below causes the
    # query to return an empty results set ... not sure why...
    # but I think it has to do with the fact that the baseCMActivity also has a
State
    # $query_def_obj->BuildField("Parent_Defect_Record.State");
    # $query_def_obj->BuildField("Parent_Enhancement_Record.State");

    # now that I think about this more, I've seen this before, and it is the case

```

```

that queries against
    # child record fields whose names also appear in the parent record fail to
    behave as would be expected

    # create a results object and run the query
    my $result_set_obj = $CQsession->BuildResultSet($query_def_obj);
    print $result_set_obj->Execute();

    while ( $result_set_obj->MoveNext() == $CQPerlExt::CQ_SUCCESS ) {

        my $id = $result_set_obj->GetColumnValue(1);
        my $stream = $result_set_obj->GetColumnValue(14);

        # don't bother adding an activity that isn't associated with a stream
        # since our activity list comes from a baseline comparison, it's very
unlikely that this
        # will be empty
        if (! $stream) {
            print "\n\nWarning: $id appears to not be associated with a
stream: excluding from this list";
            next;
        }

        $act_info{$id}{'inspection'} = $result_set_obj->GetColumnValue(2);

        # print "\n" . $result_set_obj->GetColumnValue(1) . $result_set_obj-
>GetColumnValue(2) . $result_set_obj->GetColumnValue(3) . $result_set_obj-
>GetColumnValue(4) . $result_set_obj->GetColumnValue(5) . $result_set_obj-
>GetColumnValue(6) . $result_set_obj->GetColumnValue(7) . $result_set_obj-
>GetColumnValue(8) . $result_set_obj->GetColumnValue(9) . $result_set_obj-
>GetColumnValue(10);

        $act_info{$id}{'defect'} = $result_set_obj->GetColumnValue(3);
        $act_info{$id}{'enhancement'} = $result_set_obj->GetColumnValue(4);
        $act_info{$id}{'defect_resolution'} = $result_set_obj-
>GetColumnValue(5);
        $act_info{$id}{'defect_swit'} = $result_set_obj->GetColumnValue(6);
        $act_info{$id}{'defect_unit'} = $result_set_obj->GetColumnValue(7);
        $act_info{$id}{'enhancement_resolution'} = $result_set_obj-
>GetColumnValue(8);
        $act_info{$id}{'enhancement_swit'} = $result_set_obj-
>GetColumnValue(9);
        $act_info{$id}{'enhancement_unit'} = $result_set_obj-
>GetColumnValue(10);
        $act_info{$id}{'headline'} = $result_set_obj->GetColumnValue(11);
        $act_info{$id}{'state'} = $result_set_obj->GetColumnValue(12);
        $act_info{$id}{'owner'} = $result_set_obj->GetColumnValue(13);
        $act_info{$id}{'stream'} = $result_set_obj->GetColumnValue(14);

        #if ($act_info{$id}{'stream'} eq "") {
        #print "\n\n#####SOME_MESSAGE#####";
        #}

        #print "\n\t$act_info{$id}{'owner'}";

        # since we can't use these, we may have to look up this information
        # separately later
        #
        # $act_info{$id}{'defect_state'} = $result_set_obj->GetColumnValue(11);
        # $act_info{$id}{'enhancement_state'} = $result_set_obj-
>GetColumnValue(12);

    }

    CQSession::Unbuild($CQsession);

    return \%act_info;
}

```

```

#-----
#
# sub check_CC_for_file_versions
#
#     calls get_a_view_for_this_stream
#     calls first_version_is_smaller
#
#
#-----

sub check_CC_for_file_versions {

    my %hash = %{$_[0]};

    foreach $key (keys %hash) {

        #print "\n    * $key $hash{$key}{'stream'}";

        #print "a view for this would be: " .
get_a_view_for_this_stream($hash{$key}{'stream'});

        $hash{$key}{'view'} = get_a_view_for_this_stream(
$hash{$key}{'stream'} );

        my $view = $hash{$key}{'view'};

        my $view_drive = get_view_drive();
        chdir ("$view_drive\\$view");

        #print `cleartool lsactivity -long $key@$pvob 2>&1`;
        @output = `cleartool lsactivity -long $key@$pvob 2>&1`;

        # initialize a place on the hash for file information
        %hash{$key}{'files'} ;

        foreach $line (@output) {

            next if $line !~ /\Q$view\E/;
            #print "$line";
            $line =~ s/^\s+//;
            $line =~ s/\s+$//;
            #$line =~ s/Q:\\\\Q$view\E\\\\/;
            $line =~ s/\Q$view_drive\E\\\\Q$view\E\\\\/;

            # we're taking the output of the lsactivity and putting it into
            # a file and the version specific information (or version tree
address)

            my $file, my $version;
            ($file, $version) = split /\@\/, $line;

            #print "\n\tThat's $file and version extension $version";
            #print "\n\t $file and $version";

            # set the current version that we're working on
            $hash{$key}{'files'}{$file}{'cur_version'} = $version;
            # set some temp vars to the already record earliest and latest
            # (if they don't exist, then we'll set them... see below...)
            my $early = $hash{$key}{'files'}{$file}{'earliest_version'} ;
            my $late = $hash{$key}{'files'}{$file}{'latest_version'} ;

            #print "\n\tfor $file, Comparing $early and $late against
$version";

            # if the version is ealier than what we've already recorded,
update

            if ( first_version_is_smaller( $version, $early )) {

```

```

                                $hash{$key}{'files'}{$file}{'earliest_version'} =
$version ;
                                }

                                # likewise, if we see that this version is the latest, select
that                                if (first_version_is_smaller($late, $version)) {
                                $hash{$key}{'files'}{$file}{'latest_version'} = $version
;                                }

                                #print "\n\t -" . $hash{$key}{'files'}{$file}{'cur_version'};

                                }

                                }

                                return \%hash;

}

#-----
#
#
#   sub first_version_is_smaller
#
#       compares two strings.  the strings look like this:
#
#       \main\se_7.2_Dev\se_7.3_CDev\1      \main\se_7.2_Dev\se_7.3_CDev\7
#       \main\se_7.2_CDev\2      \main\se_7.2_CDev\3A
#
#       The sub must look at the last whole integer and compare those
#       We do not have to confirm that both versions are on the same
#       branch because clearcase activities are tied to streams
#
#       thus it's highly unlikely that the versions being compared
#       will not be on the same branch
#
#
#-----

sub first_version_is_smaller {

    return 1 if (! $_[0] );
    return 1 if (! $_[1] );

    my @first_array = split /\//, $_[0];
    my @second_array = split /\//, $_[1];

    if ( $first_array[$#first_array] < $second_array[$#second_array] ) {
        return 1;
    }
    else {
        return 0;
    }
}

#-----
#
#   sub get_a_view_for_this_stream
#
#       call this, pass a stream name in a string,
#       and get back a view.  the view is either located
#       or created.
#
#       calls start_or_make_a_view
#       expects that $pvob is a global variable populated with

```



```

#       the relevant pprob from clearcase (we only have one pprob in ISRP)
#
#-----

sub get_a_view_for_this_stream {

    # create a private instance of stream based on the input
    my $stream = $_[0];

    # do a little error checking -- if we don't have a stream at this point se
    should just stop
    die "ERROR: get_a_view_for_this_stream was passed an empty \"$stream\" if
    ($stream eq "");

    # look for a view that has the stream name in it
    my $view_drive = get_view_drive();
    my $cmd_output = `dir $view_drive\\`;
    die "ERROR: some problem checking view dir using \"dir $view_drive\\\": $! -
    $cmd_output" if ($?);

    # note: you can't redirect stderr to stdout as it changes the output
    # and I don't feel like addressing it now
    # my $cmd_output = `dir q:\\ 2>&1`;

    # do a minimal amount of error handling
    # these msgs should come through stderr
    if ( $cmd_output =~ /The device is not ready/
        or
        $cmd_output =~ /is not a recognized device/
        or
        $cmd_output =~ /is not a recognized device/
    ) { #then
        die "Some problem when looking at $view_drive\\ : $cmd_output";
    }

    # the output is separated by some kind of whitespace
    my @views = split /\s+/, $cmd_output;

    # get those views whose names contain the stream
    # note that perl searches on variables require encapsulation in \Q and \E
    my @matching_views = grep (/\\Q$stream\\E/, @views);

    #print "\n\n Here are the views that I found matching stream $stream:";
    #foreach $guy (@matching_views){
    #    print "\n\t$guy";
    #}

    #if ($#matching_views < 0) {
    #    #    print "\n\tNo (already running) views found for this stream";
    #}

    if ($#matching_views < 0) {

        print "\n\nNo matching views found... I'll try making one...\n";

        @matching_views = ( start_or_make_a_view($stream) );

    }

    # return the view at the top of the list;
    return $matching_views[0];

}

```

```

#-----
#
# sub start_or_make_a_view
#   returns the name of a view that is currently running
#   based on the stream name provided
#
#   first, check to see if a view already exists (based
#   off of our expected viewname.  if it does, ensure it's
#   started and return that
#
#   if it doesn't already exist, create a new view and return
#   the view name
#
#   expects that global variable $pvob is populated
#-----

sub start_or_make_a_view {

    # grab the input as string
    my $stream = $_[0];

    # first check to see if the view already exists
    # $USERNAME is populated from the use Env "USERNAME" statement above
    my $output = `cleartool lsvview ${USERNAME}_XX_${stream} 2>&1`;

    # if it's not there, make a view,
    # otherwise, ensure that the view is started and return that

    if ($output =~ /cleartool: Error/) {

        # make a view
        #print "\n\tview    ${USERNAME}_XX_${stream}    does    not    exist.
Creating...";
        #print "\n\tRunning: \"cleartool mkview -tag ${USERNAME}_XX_${stream}
-stream ${stream}\@${pvob} -stgloc -auto 2>&1\" ";
        #   cleartool    mkview    -tag    cmbuild2_XX_se_7.1_Dev    -stream
se_7.1_Dev@\isrp_pvob -stgloc -auto
        #
        my $output = `cleartool mkview -tag ${USERNAME}_XX_${stream} -stream
${stream}\@${pvob} -stgloc -auto 2>&1 `;

        if ( $output =~ /Created view/ ) {
            #print "\n\t${USERNAME}_XX_${stream} created";
            return "${USERNAME}_XX_${stream}";
        }
        else {
            die "ERROR:  I  can't  seem  to  make  this  view:
${USERNAME}_XX_${stream} \n\n\tHere's my output: \n$output";
        }

    }
    else {

        #use the view that already exists, if you can

        if (substr($output,1,1) eq "**") {

            #print "\n\tView already started";
            return "${USERNAME}_XX_${stream}";

        }
        else {
            #print "\n\tView  ${USERNAME}_XX_${stream}  exists  but  isn't
started.  Starting...";

            if ( `cleartool startview ${USERNAME}_XX_${stream}` eq "" ) {
                #print "\n\tView started";
                return "${USERNAME}_XX_${stream}";
            }
        }
    }
}

```

```

        else {
            die "The view $view exists, but I can't start it...";
        }
    }
}

#-----
# sub create_compare_dir
#
#     use the activities hash you created to get lists of
#     old files vs new files.  use these lists to create
#     directories in your temp folder.  later we'll compare
#     these two folders against eachother
#-----

sub create_compare_dir {

    %hash = %{$_[0]};

    #print "\njust for reference, our hash was " . \%hash;

    #print "\n\tCopying Files: ";

    setup_base_dirs();

    #-----
    #     now we go through the activities, and for each, look at each of the files
    #     associated and create a directory tree under latest and previous that
    #     corresponds to the directory tree for the file
    #-----
    foreach $key ( keys (%hash) ) {

        #print "\n\t\t$key:";
        #foreach $inner_key (keys %{ $hash{$key} } ) {
        #    print "\n$key : $inner_key : $hash{$key}{$inner_key}";
        #}

        if ( ! -d "$temp_dir\\$compare_directory"){
            mkdir( "$temp_dir\\$compare_directory") or die "ERROR: cannot
make $temp_dir\\$compare_directory because of $!";
        }

        # foreach of the files, create the empty directory structure
        # that you need in order to do the comparison
        foreach $file ( keys %{ $hash{$key}{"files"} } ) {

            ##print "\n$file:\n\t" . $hash{$key}{"files"}{$file};

            #foreach $other_key (keys %{ $hash{$key}{"files"}{$file} } ) {
            #print "\n$other_key";
            #}

            #print
            "\n$file:                                "
            $hash{$key}{"files"}{$file}{"latest_version"};
            #print
            "\n$file:                                "
            $hash{$key}{"files"}{$file}{"earliest_version"};

            # split up the file string to get an array of
            # directories
            my @dirs = split /\//, $file;
            # pop off the last one -- that's the filename!
            pop @dirs;

            my $already_created_dir = "";
            my $this_dir = "";

```

```

my $that_dir = "";

foreach $dir (@dirs) {
    $this_dir = $temp_dir . "\\$compare_directory\\latest\\"
. $already_created_dir . $dir ;
    $that_dir = $temp_dir
. "\\$compare_directory\\previous\\" . $already_created_dir . $dir ;

    # if you create a directory, you may print to the screen
    that you've done so

    if (! -d $this_dir) {
        die "ERROR: cannot make $this_dir: $!" if (!
mkdir ($this_dir));
        #print "ERROR: cannot make $this_dir: $!" if (!
mkdir ($this_dir));
    }

    if (! -d $that_dir) {
        die "ERROR: cannot make $that_dir: $!" if (!
mkdir ($that_dir));
        #print "ERROR: cannot make $that_dir: $!" if (!
mkdir ($that_dir));
    }
    #print "\nCreated dir $this_dir" if (mkdir ($this_dir));
    #print "\nCreated dir $that_dir" if (mkdir ($that_dir));

    if ($already_created_dir eq "") {
        #print "\n\t(Setting \ $already_created_dir to
$dir\\)";
        $already_created_dir = "$dir\\";
    }
    else {
        #print "\n\t(Setting \ $already_created_dir to
$already_created_dir" . "$dir\\)";
        $already_created_dir = $already_created_dir .
"$dir\\";
    }
}

#-----
# now perform the copy
#-----

# get the earliest version associated with the activity
my @array = split /\\/,
$hash{$key}{"files"}{$file}{"earliest_version"};

# we need to compare the latest version with the version just
previous to the
# earliest version, so take the last element off the array,

$array[$#array]--;
my $orig_ver_number = pop @array;
$orig_ver_number--;
push @array, $orig_ver_number;

# put the array back together to get a string
my $prev_version = join '\\', @array;

my $orig_file = $file . "\\@" . $prev_version;
my $latest_file = $file . "\\@" .
$hash{$key}{"files"}{$file}{"latest_version"};
# print "\nI'm going to copy out $orig_file and $latest_file";

# just skip to the next entry if this is a directory: no need
to copy those

```

```

        #print "\nChecking to see if $orig_file is a directory";
        next if (-d $orig_file);

        ##
        #   uncomment here if you want to see what files are being
copied
        ##

        #print ".";
        #print "\nCopying $file...";
        #print "\t" . " " `copy` "$latest_file\"
"\$temp_dir\\$compare_directory\\latest\\$file\" 2>&1`;
        #print "\t" . " " `copy` "$orig_file\"
"\$temp_dir\\$compare_directory\\previous\\$file\" 2>&1`;

        if ( length($latest_file) > 255 or length($orig_file) > 255 ) {

            # grab the current directory and store it so that we can
change back to where we were
            my $current_directory = `cd`;
            chomp $current_directory;

            my @split_dirs_for_latest = split /\//, $latest_file;
            my @split_dirs_for_orig = split /\//, $orig_file;

            # we take the $latest_file and the $orig_file and split
them up by directories / branches
            # then we take the first half and change directory to
that half, before running the copy command
            # on the second half
            # (scalar flattens the array and returns the number of
elements)
            my $halfway_latest = int ( scalar @split_dirs_for_latest
/ 2 );
            my $sub_path_latest = join "\\",
@split_dirs_for_latest[0 .. $halfway_latest];
            my $copy_path_latest = join "\\",
@split_dirs_for_latest[ ($halfway_latest + 1) .. $#split_dirs_for_latest];

            my $halfway_orig = int ( scalar @split_dirs_for_orig / 2
);
            my $sub_path_orig = join "\\", @split_dirs_for_orig[0
.. $halfway_orig];
            my $copy_path_orig = join "\\", @split_dirs_for_orig[
($halfway_orig + 1) .. $#split_dirs_for_orig];

            my $drive = "G";

            # change to a directory that's somewhere close to half
way down the path
            # note that this might be a real directory, or might be
a branch off of the file you're copying
            chdir "$current_directory\\$sub_path_latest" or die
"\nERROR: I can't change directory to $sub_path_latest because $!";
            print `subst $drive: . 2>&1`;
            die "\nERROR in subst command to copy: $latest_file: cmd
is 'subst $drive: . 2>&1' error msg is $!" if ($?);

            # now copy the file
            `copy` "$drive:\\$copy_path_latest\"
"\$temp_dir\\$compare_directory\\latest\\$file\" `;
            print "\nWarning: error copying
$drive:\\$copy_path_latest to $temp_dir\\$compare_directory\\latest\\$file: $!" if
($?);

            print `subst $drive: /d 2>&1`;
            die "\nERROR in un-substing $drive using command 'subst
$drive: . 2>&1' error msg is $!" if ($?);

```

```

# change to a directory that's somewhere close to half
way down the path
# note that this might be a real directory, or might be
a branch off of the file you're copying
chdir "$current_directory\\$subs_path_orig" or die
"\nERROR: I can't change directory to $subs_path_orig because $!";
print `subst $drive: . 2>&1`;
die "\nERROR in subst command to copy: $orig_file: cmd
is 'subst $drive: . 2>&1' error msg is $!" if ($?);

# print "\nnow trying to copy from: " . `cd`;
# now copy the file
`copy "$drive:\\$copy_path_orig\"
"$temp_dir\\$compare_directory\\previous\\$file\" `;
print "\nWarning: error copying $drive:\\$copy_path_orig
to $temp_dir\\$compare_directory\\previous\\$file: $!" if ($?);

print `subst $drive: /d 2>&1`;
die "\nERROR in un-substing $drive using command 'subst
$drive: . 2>&1' error msg is $!" if ($?);

# change back to where you started
chdir $current_directory or die "\nERROR: I can't change
back to directory $current_directory because $!";

}
else {
`copy "$latest_file\"
"$temp_dir\\$compare_directory\\latest\\$file\" `;
print "\nWarning: error copying $latest_file to
$temp_dir\\$compare_directory\\latest\\$file: $!" if ($?);
`copy "$orig_file\"
"$temp_dir\\$compare_directory\\previous\\$file\" `;
print "\nWarning: error copying $orig_file to
$temp_dir\\$compare_directory\\previous\\$file: $!" if ($?);
}
}

}

}

#-----
# sub compare_directories
#
# use the SLOCCO tool to compare the two
# directories and generate an output file
#-----

sub compare_directories {

# note: the jar file referenced below needs to (apparently) be in the current
# working directory in order for things to work.
#print "\nChanging directory to \"c:\\CM\\scripts\\inspection check\". MAKE
SURE THIS IS UPDATED BEFORE RELEASING THIS SCRIPT!";
my $analysis_dir = "z:\\CM\\scripts\\baseline_analysis";
print "\nChanging directory to $analysis_dir.";
#chdir("c:\\CM\\scripts\\inspection check");
chdir($analysis_dir) or die "ERROR: cannot change to directory $analysis_dir:
$!";

my $latest = "$temp_dir\\$compare_directory\\latest";
my $previous = "$temp_dir\\$compare_directory\\previous";
my $slocco_jar = "c:\\cm\\SLOCCO\\slocco.jar";
my $slocco_jar = "slocco.jar";
my $slocco_settings = "c:\\cm\\SLOCCO\\isrp_slocco.xml";

```



```

        my $file_name = $2;
        my $total_lines = $3;
        my $comments = $4;
        my $sloc = $5;
        my $added = $6;
        my $modified = $7;
        my $deleted = $8;
        my $unchanged = $9;

        # fill up our %file_data_from_slocco hash
        $file_data_from_slocco{$long_file}{file_name} = $file_name;
        $file_data_from_slocco{$long_file}{total_lines} =
$total_lines;

        $file_data_from_slocco{$long_file}{comments} = $comments;
        $file_data_from_slocco{$long_file}{sloc} = $sloc;
        $file_data_from_slocco{$long_file}{added} = $added;
        $file_data_from_slocco{$long_file}{modified} = $modified;
        $file_data_from_slocco{$long_file}{deleted} = $deleted;
        $file_data_from_slocco{$long_file}{unchanged} = $unchanged;

    }

}

#foreach $file ( keys %file_data_from_slocco ) {
#print "\n";
#print $file_data_from_slocco{$file}{file_name} ;
#print " ";
#print $file_data_from_slocco{$file}{total_lines} ;
#print " ";
#print $file_data_from_slocco{$file}{comments} ;
#print " ";
#print $file_data_from_slocco{$file}{sloc} ;
#print " ";
#print $file_data_from_slocco{$file}{added} ;
#print " ";
#print $file_data_from_slocco{$file}{modified} ;
#print " ";
#print $file_data_from_slocco{$file}{deleted} ;
#print " ";
#print $file_data_from_slocco{$file}{unchanged} ;
#}

return \%file_data_from_slocco;

}

#-----
# sub get_latest_activities
#
# run a cleartool diffbl command to get the activities
# added to this baseline (as compared to it's immediate
# predecessor)
#-----

sub get_latest_activities {

    my $baseline = $_[0];
    my @activities;

```



```

my @output = `cleartool diffbl -pred $baseline\@$pvob 2>&1`;
# complain if the previous system call does not return success
die "some error in output; @output" if ($?);

#print "\n\ndiffbl returns @output";

@output = grep (! /Prod\d{10}\@$pvob "deliver /, @output);

foreach $line (@output) {

    my $act = substr($line,3,12);
    #print "\nadding $act to list of activities";
    push @activities, $act;

}

# should probably take out the grep for deliveries and instead
# simply write a function to only include basecms
#@activities = return_only_baseCMS(@activities);

return @activities;

}

#-----
# sub get_all_activities
#
#     purpose is to take a baseline and return all the
#     baseCMactivities that went into the baseline
#     (since the foundation baseline)
#-----

sub get_all_activities {

    my $baseline = $_[0];

    my $stream = `cleartool desc -fmt "[%bl_stream]p" baseline:$baseline\@$pvob`;
    chomp $stream;
    die "ERROR: description of baseline $baseline failed: $!" if ($?);

    my $prev_bl = `cleartool desc -fmt "[%found_bls]p" stream:$stream\@$pvob`;
    chomp $prev_bl;
    die "ERROR: describing the stream $stream failed: $!" if ($?);

    my @output = `cleartool diffbl $baseline\@$pvob $prev_bl\@$pvob 2>&1`;
    #print "\n `cleartool diffbl $baseline\@$pvob $prev_bl\@$pvob 2>&1`";
    die "some error in calling diffbl: @output" if ($?);

    #print "\n\ndiffbl returns @output";

    # note that when you have a variable in a regular expression block,
    # it needs to be enclosed in \Q and \E to get perl to interpret it properly
    @output = grep (!/Prod[0-9]{8}\@Q$pvob\E "deliver/, @output);

    foreach $line (@output) {

        my $act = substr($line,3,12);
        #print "\nadding $act to list of activities";
        push @activities, $act;

    }

    # should probably take out the grep for deliveries and instead
    # simply write a function to only include basecms
    #@activities = return_only_baseCMS(@activities);

```

```

        return @activities;
    }

#-----
#   sub copy_files_for_these_activities
#
#       accepts a list of baseCMactivities
#
#       from that list, get data on all files associated with that
#       activity, including the latest version in clearcase, and the
#       earliest version
#
#       then copy it out
#-----

sub copy_files_for_these_activities {

    my @activities = @_;

    #print "\n\nInside of copy_files_for_these_activities: @activities";

    my $hash_of_activity_data = get_cq_info_for(@activities);

    #print "\n\nhere are the latest activities: @{latest_activities}";
    #print "\n\nhere are the full list of activities for this baseline, since the
foundation: @{all_activities}";

    #-----
    #   just verify that you got something back from CQ
    #-----

    my @acts = keys %{$hash_of_activity_data};
    $number = $#acts + 1;
    #print "\nThere are $number of activities.";
    if ( $number < 1 ) {
        print "\n\nSorry: there are $number activities found in \ $hash_of-
activity_data\n\n";
        return;
    }

    #-----
    #   now we check clearcase to see what
    #   the activity has as far as files are
    #   concerned -- load into the data hash
    #-----

    $hash_of_activity_data = check_CC_for_file_versions($hash_of_activity_data);

    #-----
    #   copy out the files into two directories
    #   (one for the previous versions, another
    #   for those checked in against our activities
    #-----

    create_compare_dir($hash_of_activity_data);

    print "\nDone creating directories: $temp_dir\\$compare_dir";

    return ($insp_results, $hash_of_activity_data);
}

##-----

```

```

##   sub run_comparison_for_activity_list
##
##       accepts a list of baseCMactivities
##
##       from that list, get data on all files associated with that
##       activity, including the latest version in clearcase, and the
##       earliest version
##
##       then
##-----
#
#sub run_comparison_for_activity_list {
#
#my @activities = @_ ;
#
#my $hash_of_activity_data = get_cq_info_for(@activities);
#
##print "\n\nhere are the latest activities: @{\latest_activities}";
##print "\n\nhere are the full list of activities for this baseline, since the
foundation: @{\all_activities}";
#
##-----
##   just verify that you got something back from CQ
##-----
#
#my @acts = keys %{$hash_of_activity_data};
#$number = $#acts + 1;
##print "\nThere are $number of activities.";
#if ( $number < 1 ) {
#print "\n\nSorry: there are $number activities found in \\\$hash_of-activity_data\n\n";
#return;
#}
#
#
##-----
##   don't need anymore since I added a check in the get_cq_info_for
##   subroutine
##-----
##$hashref = remove_activities_with_empty_streams(\%hash);
##$hash_of_activity_data
remove_activities_with_empty_streams($hash_of_activity_data);
#
#
##-----
##   now we check clearcase to see what
##   the activity has as far as files are
##   concerned -- load into the data hash
##-----
#
#$hash_of_activity_data = check_CC_for_file_versions($hash_of_activity_data);
#
##just some checking to verify hash contents -- delete as needed
##my %h = %{$hash_of_activity_data};
##foreach $x (keys %h) {
##   print "\n$x and $h{$x}";
##
##   my %g = %{$h{$x}};
##
##   foreach $y (keys %g){
##       if ($y ne "files") {
##           print "\n\t$y $g{$y}";
##       }
##       else {
##           my %z = %{$g{$y}};
##
##           print "\n\thas files:";
##           foreach my $file (keys %z) {
##               print "\n\t\t$file: $z{$file}";
##           }
##       }
##   }
## }

```

```

##}
#
###-----
##  copy out the files into two directories
##  (one for the previous versions, another
##  for those checked in against our activities
##-----
#
#create_compare_dir($hash_of_activity_data);
#
##-----
##  compare the two directories using the
##  SLOCCO tool provided by LMCO
##-----
#
#compare_directories();
#
##-----
##  now grab the slocco output and process
##  it
##-----
#
$insp_results = extract_data_from_slocco_output();
#
#return ($insp_results, $hash_of_activity_data);
#}

#-----
# sub activities_from_baseline_excluding_earlier_releases
#
#      accept a clearcase baseline
#
#      first call get_all_activities to get a list of all the activities
#      get an ordered list of all streams and determine what streams happened
#      before the stream belonging to the baseline, through the foundation baseline
#      for the stream in question
#      strip out all the activities from previous releases (if a defect appears
#      in an earlier stream, then remove it from the list)
#-----

sub activities_from_baseline_excluding_earlier_releases {

    my $baseline = $_[0];
    my @all_activities = get_all_activities($baseline);

    #my $stream = get_stream_from_baseline($baseline);
    #my @previous_projects = get_sorted_projects_for($stream);
    # now get a list of activities included in those projects
    #my @excluded_activities =
get_activities_for_these_projects(@previous_projects);

    my @excluded_activities = get_excluded_activities(@all_activities);

    my @only_activities_for_this_release = reconcile_activity_lists(
\@all_activities, \@excluded_activities );

    return @only_activities_for_this_release;

}

#-----
#      sub get_excluded_activities
#-----

sub get_excluded_activities {

    my @activities = @_;

```

```

my @excluded_acts = ();

# foreach activities, look up it's parent, grab all associated child activies,
# and check each of them for the project they belong to

my $CQsession = CQSession::Build();
$CQsession->UserLogon("xxxxx", "xxxxx", "xxxxx", "");

foreach $act (@activities) {

    my @associated_records;
    my $has_earlier_content = 0;

    my $activity_object = $CQsession->GetEntity("BaseCMActivity", "$act");

    # get the parent record, whether defect or enhancement
    my $field_info_obj = $activity_object->GetFieldValue("Parent_Defect_Record");
    my $defect_parent = $field_info_obj->GetValue();
    $field_info_obj = $activity_object->GetFieldValue("Parent_Enhancement_Record");
    my $enhancement_parent = $field_info_obj->GetValue();

    # get the project that this activity is a part of
    my $project = $activity_object->GetFieldValue("ucm_project")->GetValue();

    if ($defect_parent ne "") {
        my $parent_object = $CQsession->GetEntity("Defect", "$defect_parent");
        @associated_records = split /\n/, $parent_object->GetFieldValue("Child_Defect_Record")->GetValue;
        #print "\n\nfor $act: we have other records associated with the parent defect $defect_parent: @{$associated_records}";
    }
    elsif ($enhancement_parent ne "") {
        my $parent_object = $CQsession->GetEntity("EnhancementRequest", "$enhancement_parent");
        @associated_records = split /\n/, $parent_object->GetFieldValue("Child_Enhancement_Record")->GetValue;
        #print "\n\nfor $act: we have other records associated with the parent enhancement $enhancement_parent: @{$associated_records}";
    }
    else {
        die "\n\nI can't find a parent for this baseCMactivity: $act";
    }

    # foreach of the baseCMs that share a parent with the activity in question, pull out the project
    # that they're a part of and compare to see if it appears to be an earlier release
    # if it is, then record the activity in question as an excluded activity

    foreach $baseCM (@associated_records) {

        $activity_object = $CQsession->GetEntity("BaseCMActivity", "$baseCM");

        my $other_project = $activity_object->GetFieldValue("ucm_project")->GetValue();

        # if the prefixs of the project do not match, then just ignore it

        # for example, we don't want to compare se_ with sw_
        next if ( substr($project, 0, 3) ne substr($other_project, 0, 3) );

        # convert the project from, for example, sw_17.12 to two

```

```

numbers, 17 and 12
(my $prj_1, my $prj_2) = split /\./, substr($project, 3);
(my $o_prj_1, my $o_prj_2) = split /\./, substr($other_project,
3);

# use the values extracted to compare and find whether the $act
in question
# has checkins in an earlier release, if it does, set your flag
$has_earlier_content = 1 if ($prj_1 > $o_prj_1);
$has_earlier_content = 1 if ( ($prj_1 == $o_prj_1) and ($prj_2
> $o_prj_2) );

# if so, stop further checking
last if ($has_earlier_content);

}

push (@excluded_acts, $act) if ($has_earlier_content);

}

CQSession::Unbuild($CQsession);

return @excluded_acts;

}

#-----
#      sub get_stream_from_baseline
#-----

sub get_stream_from_baseline {

    my $baseline = $_[0];

    my $stream = `cleartool lsbl -fmt %[bl_stream]p $baseline@$pvob 2>&1`;
    die "ERROR: unable to check baseline to find foundation stream in
get_stream_from_baseline: $!\noutput is $stream\n" if ($?);

    #print "\n$baseline is from $stream";
    return $stream;

}

#-----
#      sub get_foundation_stream
#-----

sub get_foundation_stream {

    my $stream = $_[0];

    my $found_baselines = `cleartool lsstream -fmt %[found_bls]p $stream@$pvob
2>&1`;
    die "\nERROR: unable to check stream to look for foundation baselines in
get_foundation_stream: $!\noutput is $found_baselines\n" if ($?);

    #
    # error out if multiple foundation baselines are found
    #
    if ( $found_baselines =~ /\s/ ) {

```

```

        # for now, we need to just complain that multiple baselines were found
and then die
        # this shouldn't occur, and if it does then we'll just have to rework
the logic to handle multiple components
        die "\nERROR: multiple components found associated with $stream:
multiple foundation baselines: $found_baselines. \n\tYou will need to retool this
script before you can run this against $ARGV[0].";
        #my @multiple_streams = split /\s/, $found_baselines;

    }

    my $found_stream = get_stream_from_baseline($found_baselines);

    print "\nstream $stream is based on $found_stream";

    return $found_stream;

}

#-----
#
#-----

#-----
# sort_projects
#   for a given list of clearcase projects, sort the list from earliest to
#   latest. List looks something like this:
#
#       sw_16.9
#       sw_17.1
#       sw_17.2
#       sw_17.3
#       sw_17.4
#       sw_16.10
#       sw_17.5
#       sw_17.6
#       sw_17.8
#       sw_17.7
#       sw_17.9
#       sw_17.10
#       sw_17.11
#
#   Notice the prefix_XX.YY format. We sort by XX first, and then by YY.
#
#-----

sub sort_projects {

    print "\n I've been asked to sort @_";

    my @projs = @_;
    my @sorted_list;

    my $prefix = substr $projs[0], 0, 3;

    # load up @sorted_list with all the projects, but with the three
    # char prefix stripped out
    foreach $x (@projs) {
        push ( @sorted_list, substr ($x, 3) );
    }

    print "\nWith the prefixes stripped off, the list looks like this:
@{sorted_list}";

```

```

    for ($i=0; $i <= $#sorted_list; $i++) {

        # first we convert all the numbers to XXX.YYY format, adding zeros
        where appropriate

        print "\n\tFor $sorted_list[$i], we split it into ";

        my @thing = split /\./, $sorted_list[$i];

        print "$thing[0] and $thing[1]";

        for ($j = 0; $j < 2; $j++){

            if ($thing[$j] =~ /^[0-9]$/ ) {
                $thing[$j] = "00" . $thing[$j];
            }

            if ($thing[$j] =~ /^[0-9][0-9]$/ ) {
                $thing[$j] = "0" . $thing[$j];
            }

        }

        $sorted_list[$i] = "$thing[0].$thing[1]";

    }

    print "\n\nRefactoring to deal with the zeros, and it looks like this:
    @{sorted_list}";

    @sorted_list = sort (@sorted_list);

    # now we need to strip out the extra zeros that we added in order to do the
    sort
    # there must be an elegant way to do this. I'm open to suggestions...

    for ($i=0; $i <= $#sorted_list; $i++) {

        print "\n\tlooking at $sorted_list[$i]";

        if ($sorted_list[$i] =~ /\..000$/ ) {
            $sorted_list[$i] =~ s/\..000/\..0/;
        }

        if ($sorted_list[$i] =~ /^00/ ) {
            $sorted_list[$i] =~ s/^00//;
        }

        if ($sorted_list[$i] =~ /\..00\d/ ) {
            $sorted_list[$i] =~ s/\..00/\..0/g;
        }

        if ($sorted_list[$i] =~ /^0/ ) {
            $sorted_list[$i] =~ s/^0//;
        }

        if ($sorted_list[$i] =~ /\..0\d\d/ ) {
            $sorted_list[$i] =~ s/\..0/\..0/;
        }

        if ($sorted_list[$i] =~ /\..0\d/ ) {
            $sorted_list[$i] =~ s/\..0/\..0/;
        }

        print "    changed to $sorted_list[$i]";

        # now put the prefixes back on, and return the list

```





```

        if ( $defect && $enhance ) {
            die "\nERROR! $id unexpectedly linked to defect $defect and
enhancement $enhance ";
        }
        elsif ( $defect ) {
            push @{$parent_records{$defect}}, $id;
            print "\nrecord $id has parent $defect (d)";
        }
        elsif ( $enhance ) {
            push @{$parent_records{$enhance}}, $id;
            print "\nrecord $id has parent $enhance (e)";
        }
        else {
            print "\n\nWARNING: id $id has no parrent! Please
investigate!\n\n";
        }
    }

    # don't attempt to build up an exclusion list unless we actually provided some
    # activities in the first place.
    if ( $#excluded_activities != -1 ) {

        print "\n\nTo be excluded:";

        # now do the same for the exclusion list

        $query_def_obj = $CQsession->BuildQuery("BaseCMActivity");
        $filterOp = $query_def_obj-
>BuildFilterOperator($CQPerlExt::CQ_BOOL_OP_AND);
        $filterOp->BuildFilter("id", $CQPerlExt::CQ_COMP_OP_IN,
        \@excluded_activities);

        $query_def_obj->BuildField("id");
        $query_def_obj->BuildField("Parent_Defect_Record");
        $query_def_obj->BuildField("Parent_Enhancement_Record");

        # create a results object and run the query
        $result_set_obj = $CQsession->BuildResultSet($query_def_obj);
        print $result_set_obj->Execute();

        while ( $result_set_obj->MoveNext() == $CQPerlExt::CQ_SUCCESS ) {

            my $id = $result_set_obj->GetColumnValue(1);
            my $defect = $result_set_obj->GetColumnValue(2);
            my $enhance = $result_set_obj->GetColumnValue(3);

            if ( $defect && $enhance ) {
                die "\nERROR! $id unexpectedly linked to defect $defect
and enhancement $enhance ";
            }
            elsif ( $defect ) {
                $parents_of_excluded_ones{$defect} = $id;
                print "\nrecord $id has parent $defect (d)";
            }
            elsif ( $enhance ) {
                $parents_of_excluded_ones{$enhance} = $id;
                print "\nrecord $id has parent $enhance (e)";
            }
            else {
                print "\n\nWARNING: id $id has no parrent! Please
investigate!\n\n";
            }
        }

    }

    CQSession::Unbuild($CQsession);

```

```

        # remove any keys that appear in the list of excluded activities and also
        exist in the
        # list of activities from the stream being analysed
        foreach $excluded_parent (keys %parents_of_excluded_ones) {
            print "\nLooking to exclude $excluded_parent";
            delete $parent_records{$excluded_parent} if (
$parent_records{$excluded_parent} );
        }

        my @abbreviated_list = ();

        #print "\n\nGot to this point and we now have the following:";
        #
        foreach $key (keys %parent_records) {

            my @list = @{$parent_records{$key}};

            foreach $id (@list) {
                #print "\n$id has parent $key";
                push @abbreviated_list, $id;
            }

        }

        return @abbreviated_list;
    }

}

#-----
# sub get_view_drive
#-----

sub get_view_drive {

    my @use_output = `net use 2>&1`;
    die "ERROR: \"net use\" call failed: $! - @{$use_output}" if ($?);

    my $line;
    my $drive;
    my $junk;

    ($line) = grep ( /\\\\\view                               ClearCase Dynamic Views/,
@use_output);
    ($junk, $drive, $junk) = split /\s+/, $line;

    #print "\nMy grep found this drive: $drive.  \ $line is $line";

    return $drive;

}

#-----
# sub setup_base_dirs
#
#     create the base directories for comparison
#     there are three:
#         $compare_directory\
#             latest\
#             previous\
#-----

sub setup_base_dirs {

    if (! -d "$temp_dir\\$compare_directory") {
        #print "\n$temp_dir\\$compare_directory  doesn't  already  exist...
making...";
        mkdir ($temp_dir . "\\$compare_directory\\")
    }
}

```

```

    }

    if (! -d "$temp_dir\\$compare_directory\\latest") {
        #print "\n$temp_dir\\$compare_directory\\latest doesn't already
exist... making...";
        mkdir ($temp_dir . "\\$compare_directory\\latest")
    }

    if (! -d "$temp_dir\\$compare_directory\\previous") {
        #print "\n$temp_dir\\$compare_directory\\previous doesn't already
exist... making...";
        mkdir ($temp_dir . "\\$compare_directory\\previous")
    }
}

#-----
# sub get_component_from_baseline
#
#         given a baseline, determine which component the baseline is
#         associated with
#
#         for now, assume a single component, error out if there are
#         multiple
#-----

sub get_component_from_baseline {

    my $baseline = $_[0];

    my $comp = `cleartool desc -fmt %[component]p baseline:$baseline\@$pvob 2>&1`;
    die "ERROR: trouble describing baseline $baseline for $pvob: $comp: $!" if
($?);
    chomp $comp;

    #print "\n\nComponent is: $comp";

    # let's keep in mind that we have single components in our projects in this
environment
    # if this changes
    die "ERROR: \ $comp contains multiple components: $comp.\n\nI wasn't made to
handle this case. Please refactor." if ($comp =~ /\s+/);

    return $comp;
}

#-----
# sub usage
#-----

sub usage {

    my $msg = $_[0];

    if ($msg) {
        print "Error: $msg";
    }

    print "\n\n";

    print "\n\n\tUsage: ratlperl $0 <baseline>";
    print "\n\n\tExample: ratlperl $0 sw_17.11.1006.4";
    print "\n\n\tExample: ratlperl $0 se_7.9.0.4@\\isrp_pvob\n\n\n";
    exit 1;
}

```

## Chapter 6: Glossary

Artifact:	A software artifact is a product that is created during software development containing software constructs. Software artifacts can be source files, software modules, or software documents such as the Software Requirements Specifications (SwRS) produced during software development [36].
Construct:	The smallest software piece for which data is collected. Depending on the software development project, a construct can be a software line of code (SLOC), function point (FP), function, class, source statement (SS), or any other software unit [9].
Error:	Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification [30].
Failure:	(1) The termination of the ability of a functional unit to perform its required function [30]. (2) An event in which a system or system component does not perform a required function within specified limits. A failure may be produced when a fault is encountered [30].
Fault:	(1) An accidental condition that causes a functional unit to fail to perform its required function [30]. (2) A manifestation of an error in software. A fault, if encountered, may cause a failure. Synonymous with bug [30].
Inspection:	A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include code inspection; design inspection [30].
Measure:	A quantitative assessment of the degree to which a software product or process possesses a given attribute.
Metric:	A quantitative measure of the degree to which a system, component, or process possesses a given attribute.
Module:	(1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine [30]. (2) A logically separable part of a program. Note: The terms “module,” “component,” and “unit” are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized [30].
Software Reliability:	The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to, and use of, the system as well as a function of

the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [30].

#### Software

#### Reliability

Management: The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources (cost), schedule, and performance [30].

## Chapter 7: Bibliography

- [1] S. Afsharian, M. Giacomobono, and P. Inverardi, "A framework for software project estimation based on COSMIC, DSM and rework characterization," in *30th International Conference on Software Engineering, ICSE 2008 - 1st Business Impact of Process Improvements, BIPI-2008, May 13, 2008 - May 13, 2008*, Leipzig, Germany, 2008, pp. 15-23.
- [2] M. Askari and R. Holt, "Information theoretic evaluation of change prediction models for large-scale software," in *2006 International Workshop on Mining Software Repositories, MSR '06, Co-located with the 28th International Conference on Software Engineering, ICSE 2006, May 20, 2006 - May 28, 2006*, Shanghai, China, 2006, pp. 126-132.
- [3] C. Bai, K.-Y. Cai, and T. Y. Chen, "An Efficient Defect Estimation Method for Software Defect Curves," in *Proceedings: 27th Annual International Computer Software and Applications Conference, COMPSAC 2003, November 3, 2003 - November 6, 2003*, Dallas, TX, United states, 2003, pp. 534-539.
- [4] C.-G. Bai, K.-Y. Cai, Q.-P. Hu, and S.-H. Ng, "On the trend of remaining software defect estimation," *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans*, vol. 38, no. 5, pp. 1129-1142, 2008.
- [5] V. R. Basili et al., "Empirical investigation of perspective-based reading," *Empirical Software Engineering*, vol. 1, no. 2, pp. 133-164, 1996.
- [6] T. Bergander, Y. Luo, and A. B. Hamza, "Software defects prediction using operating characteristic curves," in *2007 IEEE International Conference on Information Reuse and Integration, IEEE IRI-2007, August 13, 2007 - August 15, 2007*, Las Vegas, NV, United states, 2007, pp. 713-718.
- [7] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *IWPSE'07: Ninth International Workshop on Principles of Software Evolution - In conjunction with the 6th ESEC(European Software Engineering Conference)/FSE(Foundations of Software Engineering) Joint Meeting, September 3, 2007 - September 4, 2007*, Dubrovnik, Croatia, 2007, pp. 11-18.
- [8] W. Blischke, *Reliability : modeling, prediction, and optimization*. New York: Wiley, 2000.
- [9] W. Blischke, *Case studies in reliability and maintenance*. Hoboken NJ: John Wiley, 2003.
- [10] C. B. Boehm et al., "Cost models for future software life cycle processes," presented at the Annals of Software Engineering, 1995.
- [11] L. C. Briand, K. E. Emam, B. G. Freimut, and O. Laitenberger, "Comprehensive evaluation of capture-recapture models for estimating software defect content," *IEEE Transactions on Software Engineering*, vol. 26, no. 6, pp. 518-540, 2000.
- [12] K.-Y. Cai, "On estimating the number of defects remaining in software," *Journal of Systems and Software*, vol. 40, no. 2, pp. 93-114, 1998.
- [13] K.-Y. Cai, *Software defect and operational profile modeling*. Boston: Kluwer Academic Publishers, 1998.

- [14] E. Ceylan, F. O. Kutlubay, and A. B. Bener, "Software defect identification using machine learning techniques," in *32nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA, August 29, 2006 - September 1, 2006*, Cavtat/Dubrovnik, Croatia, 2006, pp. 240-246.
- [15] C.-P. Chang and C.-P. Chu, "Defect prevention in software processes: An action-based approach," *Journal of Systems and Software*, vol. 80, no. 4, pp. 559-570, 2007.
- [16] A. Chao, S.-M. Lee, and S.-L. Jeng, "Estimating Population Size for Capture-Recapture Data When Capture Probabilities Vary by Time and Individual Animal," *Biometrics*, vol. 48, no. 1, pp. 201-216, Mar. 1992.
- [17] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *30th International Conference on Software Engineering 2008, ICSE'08, May 10, 2008 - May 18, 2008*, Leipzig, Germany, 2008, pp. 111-120.
- [18] S. Chulani and B. Boehm, "Modeling software defect introduction and removal: COQUALMO (CONstructive QUALity MOdel)," *USC-CSE Technical Report*, pp. 99-510, 1999.
- [19] L. H. Crow, P. H. Franklin, and N. B. Robbins, "Principles of successful reliability growth applications," in *Reliability and Maintainability Symposium, 1994. Proceedings., Annual, 1994*, pp. 157-159.
- [20] L. H. Crow, "Evaluating the reliability of repairable systems," in *1990 Proceedings - Annual Reliability and Maintainability Symposium, January 23, 1990 - January 25, 1990*, Los Angeles, CA, USA, 1990, pp. 275-279.
- [21] E. L. Droguett, A. Mosleh, and C. Smidts, "Identification and Quantification of Software Dependencies in Reliability Models," *Probabilistic Safety Analysis and Management-PSAM*, vol. 4.
- [22] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649-660, 2008.
- [23] N. Fenton, M. Neil, and D. Marquez, "Using Bayesian networks to predict software defects and reliability," *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, vol. 222, no. 4, pp. 701-712, 2008.
- [24] N. Fenton et al., "Predicting software defects in varying development lifecycles using Bayesian nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32-43, 2007.
- [25] S. Ghose, "ANALYSIS OF ERRORS IN SOFTWARE RELIABILITY PREDICTION SYSTEMS AND APPLICATION OF MODEL UNCERTAINTY THEORY TO PROVIDE BETTER PREDICTIONS," University of Maryland, 2006.
- [26] T. L. Graves, A. F. Karr, U. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, 2000.
- [27] S. W. Haider, J. W. Cangussu, K. M. L. Cooper, and R. Dantu, "Estimation of defects based on defect decay model: ED3M," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 336-356, 2008.



- [28] Y. Hong, J. Baik, I.-Y. Ko, and H.-J. Choi, "A value-added predictive defect type distribution model based on project characteristics," in *7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008, May 14, 2008 - May 16, 2008*, Portland, OR, United states, 2008, pp. 469-474.
- [29] T. Illes-Seifert and B. Paech, "Exploring the relationship of history characteristics and defect count: An empirical study," in *2008 Workshop on Defects in Large Software Systems 2008, DEFECTS'08, July 20, 2008 - July 20, 2008*, Seattle, WA, United states, 2008, pp. 11-15.
- [30] Institute of Electrical and Electronics Engineers, *IEEE software engineering standards collection*. Institute of Electrical and Electronics Engineers, 1991.
- [31] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *30th International Conference on Software Engineering, ICSE 2008 - 4th International Workshop on Predictor Models in Software Engineering, PROMISE 2008, May 12, 2008 - May 13, 2008*, Leipzig, Germany, 2008, pp. 11-18.
- [32] S. Kan, *Metrics and models in software quality engineering*, 2nd ed. Boston: Addison-Wesley, 2003.
- [33] A. Kaw, *Numerical methods with applications*, 2nd ed. [Morrisville N.C.: Lulu Enterprises, 2009.
- [34] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Proceedings of the 1996 7th International Symposium on Software Reliability Engineering, ISSRE'96, October 30, 1996 - November 2, 1996*, White Plains, NY, USA, 1996, pp. 364-371.
- [35] A. khoshkhou, M. Cukier, and A. Mosleh, "A Framework for Software Reliability Management Based on Software Development Profile Model," presented at the 10th International Probabilistic Safety Assessment and Management Conference, Seattle, 2010.
- [36] H. Kou, *studying micro-processes in software development stream - Google Search*. Citeseer.
- [37] B. Lennselius and L. Rydstrom, "Software fault content and reliability estimations for telecommunication systems," *Selected Areas in Communications, IEEE Journal on*, vol. 8, no. 2, pp. 262-272, 2002.
- [38] Y. K. Malaiya and J. Denton, "Estimating the number of residual defects," presented at the hase, vol. 98, pp. 13-14.
- [39] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 420-426, 2002.
- [40] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, SIGSOFT 2008/FSE-16, November 9, 2008 - November 14, 2008*, Atlanta, GA, United states, 2008, pp. 13-23.
- [41] J. Miller, "Estimating the number of remaining defects after inspection," *Software Testing Verification and Reliability*, vol. 9, no. 3, pp. 167-189, 1999.

- [42] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *2008 ACM/IEEE 30th International Conference on Software Engineering, ICSE 2008, May 10, 2008 - May 18, 2008, Leipzig, Germany, 2008*, pp. 181-190.
- [43] J. C. Munson and S. G. Elbaum, "Code churn: a measure for estimating the impact of code change," in *Proceedings of the 1998 IEEE International Conference on Software Maintenance, ICSM, November 16, 1998 - November 20, 1998, Bethesda, MD, USA, 1998*, pp. 24-31.
- [44] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *27th International Conference on Software Engineering, ICSE 2005, May 15, 2005 - May 21, 2005, Saint Louis, MO, United states, 2005*, vol. 2005, pp. 580-586.
- [45] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th International Conference on Software Engineering, ICSE 2005, May 15, 2005 - May 21, 2005, Saint Louis, MO, United states, 2005*, vol. 2005, pp. 284-292.
- [46] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *28th International Conference on Software Engineering 2006, ICSE '06, May 20, 2006 - May 28, 2006, Shanghai, China, 2006*, vol. 2006, pp. 452-461.
- [47] N. Nagappan, B. Murphy, and V. R. Basili, "The influence of organizational structure on software quality: An empirical case study," in *30th International Conference on Software Engineering 2008, ICSE'08, May 10, 2008 - May 18, 2008, Leipzig, Germany, 2008*, pp. 521-530.
- [48] A. M. Neufelder, "How to predict software defect density during proposal phase," presented at the National Aerospace and Electronics Conference, 2000. NAECON 2000. Proceedings of the IEEE 2000, 2000, pp. 71-76.
- [49] A. P. Nikora and J. C. Munson, "Determining fault insertion rates for evolving software systems," presented at the Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on, 1998, pp. 306-315.
- [50] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," 2003.
- [51] A. J. Offutt, M. J. Harrold, and P. Kolte, "Software metric system for module coupling," *Journal of Systems and Software*, vol. 20, no. 3, pp. 295-308, 1993.
- [52] A. D. Oral and A. B. Bener, "Defect prediction for embedded software," in *22nd International Symposium on Computer and Information Sciences, ISCIS 2007, November 7, 2007 - November 9, 2007, Ankara, Turkey, 2007*, pp. 346-351.
- [53] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," presented at the Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, 2004, p. 96.
- [54] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin, "Capture-recapture in software inspections after 10 years research - Theory, evaluation and application," *Journal of Systems and Software*, vol. 72, no. 2, pp. 249-264, 2004.

- [55] A. A. Porter, L. G. Votta Jr., and V. R. Basili, "Comparing detection methods for software requirements inspections: a replicated experiment," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 563-575, 1995.
- [56] S. Rakitin, *Software verification and validation for practitioners and managers*, 2nd ed. Boston: Artech House, 2001.
- [57] C. V. Ramamoorthy and F. B. Bastani, "SOFTWARE RELIABILITY - STATUS AND PERSPECTIVES.," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 354-371, 1982.
- [58] H. Scott and C. Wohlin, "Capture-recapture in software unit testing - A case study," in *2nd International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9, 2008 - October 10, 2008*, Kaiserslautern, Germany, 2008, pp. 32-40.
- [59] M. Sherriff, N. Nagappan, L. Williams, and M. Vouk, "Early estimation of defect density using an in-process Haskell metrics model," in *1st International Workshop on Advances in Model-Based Testing, A-MOST '05, May 15, 2005 - May 21, 2005*, St. Louis, MO, United states, 2005.
- [60] M. Staron and W. Meding, "Predicting weekly defect inflow in large software projects based on project planning and test status," *Information and Software Technology*, vol. 50, no. 7-8, pp. 782-796, 2008.
- [61] M. A. Stutzke and C. S. Smidts, "A stochastic model of fault introduction removal during software development," *IEEE Transactions on Reliability*, vol. 50, no. 2, pp. 184-193, 2001.
- [62] G. S. Walia and J. C. Carver, "Evaluation of capture-recapture models for estimating the abundance of naturally-occurring defects," in *2nd International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9, 2008 - October 10, 2008*, Kaiserslautern, Germany, 2008, pp. 158-167.
- [63] T. J. O. E. J. Weyuker, "Progress in Automated Software Defect Prediction," presented at the Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008, Revised Selected Papers, 2009, p. 200.
- [64] M. Xie, *Software reliability modelling*. World Scientific, 1991.