

A Software Maintenance Process Model With Feature-based Tool and Reliability Metrics

نموذج عملي لادامة البرمجيات مع اداة
وقياسات لتحديد الموثوقية اعتمادا على المميزات

**Prepared by:
Abdallah Qaisi**

**Supervisors:
Dr. Ahmad Sharieh
Prof. Walid Salameh**

**A Dissertation Submitted in Partial
Fulfillment of the Requirements for
the degree of Doctor of Philosophy
in Computer Science**

**Graduate College of Computing Studies
Amman Arab University for Graduate Studies**

August 2008

AUTHORIZATION

I, the undersigned, Abdallah M. Qaisi, authorize Amman Arab University for Graduate Studies to reproduce this dissertation in whole or in part for purposes of research.

Name: ABDALLAH MUSTAFA QAISI
Signature: Abdallah Qaisi
Date: 9/13/2008

DISSERTATION COMMITTEE APPROVAL



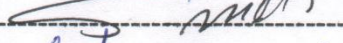

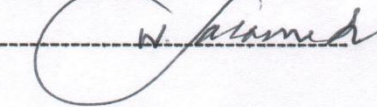
Date: August 12, 2008

Name: Abdallah M. Qaisi

Degree: Doctors of Philosophy in Computer Science

Title: A Software Maintenance Process Model With Feature-based Tool and Reliability Metrics

This dissertation, with the title "A Software Maintenance Process Model With Feature-based Tool and Reliability Metrics", was presented, examined, and approved on: August 12, 2008.

Examining Committee	Title	Signature
Prof. Fawaz Zghoul	Chair	
Dr. Ahmad Sharieh	Supervisor & Member	
Dr. Saad Abdel Sattar	Member	
Prof. Alaa Hamami	Member	
Prof. Walid Salameh	Co-supervisor & Member	

ACKNOWLEDGMENTS

First and foremost, I want to offer my thanks to Allah, the Greatest, who gave me hope and guidance. I would like to use this chance to thank the supervisors of this project, Dr. Ahmad Sharieh, and Prof. Walid Salameh, for the time they spent reviewing my dissertation and offering constructive remarks. Special thanks to Prof. Alaa Hamami and Dr. Omar Basheer for the words of encouragement and logistical help. Thanks to the dissertation committee for their valuable feedback and support.

Thanks to my friends Dr. Fawwaz Tubasi and Dr. Ahmad Qaisi who gave me the initial push to start, and to Abdelrahman and Nayef Qaisi who constantly pushed to keep it going. Special Thanks to my friends Ken Victor, Hassan Yacoub, and Haitham Rayeq for their technical support in developing the tool on the Macintosh and Windows.

PUBLICATIONS AND CONFERENCES

Parts of this research were presented and published at the 2008 International Conference on Software Engineering Research & Practice, Las Vegas, Nevada, July 2008, pp. 86-92.

DEDICATION

This dissertation is dedicated to my late father, Mustafa Qaisi, who gave me the motivation to pursue knowledge; my mother (Amina), for teaching me to be caring and responsible; and my wife (Samira) and four kids (Layla, Lorance, Muhannad, and Muhammad), for their love, patience, and support.

TABLE OF CONTENTS

Acknowledgments.....	iv
Publications and Conferences	v
Dedication.....	vi
Table Of Contents.....	vii
List of Tables	x
List of Figures	xii
List of Equations	xv
List of Symbols and Abbreviations	xvi
Abstract Of Dissertation	xvii
Arabic Summary	xxi
Chapter 1 Introduction	1
1.1 Motivation.....	1
1.2 Aim of the Work.....	3
1.3 Research Problems	6
1.3.1 Program Understanding is Difficult	6
1.3.2 Change Impact Analysis is Inaccurate	7
1.3.3 Regression Testing is Incomplete and Unfocused	7
1.3.4 Project Mismanagement.....	7
1.3.5 Development Tools Offer Limited Maintenance Support.....	8
1.4 Definitions.....	8
1.4.1 Software Maintenance.....	9
1.4.2 Software Maintenance Process Model	10
1.4.3 Program Understanding	13
1.4.4 Feature-Based Code Analysis.....	15
1.4.5 Change Impact Analysis.....	18
1.4.6 Regression Testing	20
1.4.7 Software Complexity Metrics	23
1.4.8 Software Reliability Metrics	29
1.4.9 Maintenance Management.....	30
1.5 Assumptions and Hypotheses	33
1.6 Research Importance	37

1.7	Scope and Limitations	39
1.8	Organization of the Dissertation.....	41
Chapter 2 Related Work		42
2.1	Introduction.....	42
2.2	Software Maintenance.....	42
2.3	Program Understanding and Visualization	44
2.4	Change Impact Analysis.....	48
2.5	Regression Testing	51
2.6	Feature-Based Code Analysis.....	53
2.7	Software Complexity and Reliability Metrics.....	57
2.8	Other Cost Factors	60
2.9	Summary	63
Chapter 3 Methodology.....		65
3.1	Introduction.....	65
3.2	Proposed Process Model	66
3.3	Proposed Metrics	69
3.3.1	Feature-based Function Maintainability (FBFM).....	70
3.3.2	Function Maturity (FM)	73
3.3.3	Function Reliability (FR)	76
3.3.4	Feature Reliability.....	79
3.3.5	Product Reliability.....	81
3.4	CMMR Tool	82
3.4.1	CMMR Parser.....	87
3.4.2	CMMR Viewer	95
3.4.3	Use of CMMR Tool.....	107
3.5	Summary	116
Chapter 4 Case Studies.....		118
4.1	Introduction.....	118
4.2	JContact – Open Source Java Project	119
4.2.1	Java CMMR on Windows	120
4.2.2	Java CMMR User Interface	121
4.3	A Commercial Macintosh Product	131
4.3.1	The Mac Product.....	132
4.3.2	CMMR Adoption and Feedback	136

4.4 iSpend - A Sample Macintosh Project	142
4.5 Summary	146
Chapter 5 Results and Discussion	148
5.1 Introduction.....	148
5.2 The Research Hypotheses	148
5.3 The New Process Model, Metrics, and CMMR Tool.....	160
5.4 Acceptance of CMMR	164
5.5 How the Main Research Claims Feared in Practice	165
5.6 Code Parsing.....	166
5.7 Multiple Languages and Platform Support	166
5.8 Actual Experimental Feedback.....	167
5.9 Summary	170
Chapter 6 Conclusions & Suggestions for future work.....	172
References	179
Appendices	187
Appendix A.....	187
Appendix C.....	232

LIST OF TABLES

Table	Page
1.1 Maintenance Activities not in Development 8	
2.1 Software Evolution Laws 31	
2.2 Impact of Key Adjustment Factors on Maintenance 34	
2.3 US Populations in Development and Maintenance 43	
3.1 Number of Releases Description Line 73	73
3.2 Lines of Code Description Line 73	73
3.3 Lines of Comments Description Line 73	
3.4 McCabe Cyclomatic Complexity Description Line 73	
3.5 Halstead Vocabulary Description Line 74	74
3.6 Halstead Length Description Line 74	74
3.7 Halstead Volume Description Line 74	
3.8 Maintenance Index Description Line 74	
3.9 Kafura Description Line 75	75
3.10 System Complexity Description Line 75	
3.11 Feature-Based Function Maintainability Description Line 75	
3.12 Function Maturity Description Line 75	
3.13 Function Reliability Description Line 76	
4.1 Macintosh Product Team Feedback 101	
5.1 Mac Product Feature vs. Project Metrics	

107		
5.2	Defect Detection via CMMR vs. Traditional Methods	
111		
5.3	Metric Values of a Function in Multiple Builds	113

LIST OF FIGURES

Figure		Page
1. 1	The IEEE-1219 Maintenance Process	
8		
3.1	Proposed Maintenance Process Model	
48		
3.2	Function Reliability Model	
55		
3.3	Feature Reliability Model	
57		
3.4	Product Reliability Model	
58		
3.5	CMMR Architecture	59
3.6	CMMR User Interface	
60		
3.7	CMMR Add Feature Dialog	
61		
3.8	CMMR Feature Tree Window	
65		
3.9	CMMR Tree View of a Product Feature	
69		
3.10	CMMR Comment View of a Function	
70		
3.11	CMMR Code View of a Function	
71		
3.12	CMMR Metric View of a Function	
72		
3.13	CMMR Add Function Dialog	
79		
3.14	CMMR Function Reliability Trend	
81		
3.15	CMMR Feature Reliability Trend	
82		
3.16	CMMR Product Reliability Trend	

83	
4.1	JContact Main Window
85	
4.2	CMMR Menu Bar and Menus (Java version)
87	
4.3	New Project Window (Java version)
88	
4.4	Add Feature Window (Java version)
88	
4.5	Add Contact Feature
89	
4.6	Product Features Menu (Windows version)
90	
4.7	Add Contact Feature in Build 2 with Two Changed Functions
91	
4.8	Method Source Code View
92	
4.9	Method Comments View
92	
4.10	Method Metrics View
93	
4.11	Method Reliability Chart
94	
4.12	Project Reliability Chart
94	
4.13	iSpend Main Window
102	
4.14	iSpend New Project Window
103	
4.15	Original Metric Window of a Function
104	
5.1	Original Source View of a Function
108	
5.2	Original Comments View of a Function
109	
5.3	Function Reliability Trend in a 3-Month Period
112	

5.4	Correlation of Proposed Metrics with Common Metrics
114	
5.5	Multiple Reliability Curves on Same Chart
117	

LIST OF EQUATIONS

Equation	Page
1. 1 McCabe's Cyclomatic Complexity	19
1. 2 Halstead Program Length	20
1.3 Halstead Vocabulary	20
1. 4 Halstead Volume	20
1. 5 Maintenance Index	20
1. 6 Software Reliability	21
3.1 Feature-Based Function Maintainability	51
3.2 Function Maturity	52
3.3 Function Reliability	54
3.4 Feature Reliability	56
3.5 Product Reliability	58

LIST OF SYMBOLS AND ABBREVIATIONS

CMMR	Complexity, Maintainability, Maturity, Reliability
DTrace	Mac OS X Debug Trace Services
FBFM	Feature-Based Function Maintainability
FM	Function Maturity
FR	Function Reliability
GUI	Graphical User Interface
H	Halstead Vocabulary
JPDA	Java Platform Debugger Architecture
LOC	Lines Of Code
MI	Maintainability Index
QA	Quality Assurance
R_{feature}	Feature Reliability
R_{product}	Product Reliability
CASE	Computer-Aided Software Engineering
V	Halstead Volume
VG	McCabe Cyclomatic Complexity Number of Graph G.

A Software Maintenance Process Model With Feature-based Tool and Reliability Metrics

Prepared by:
Abdallah Qaisi

Supervisors:
Dr. Ahmad Sharieh
Prof. Walid Salameh

ABSTRACT OF DISSERTATION

Software maintenance cost can be significantly higher than the initial development cost. The high maintenance cost is the result of several inefficiency factors that include: program comprehension, change impact analysis, regression testing, and reliability measurement. Current maintenance process models are not comprehensive enough to address these problems. Existing maintenance tools are not easy for the entire maintenance team to fully adopt and use during the various maintenance activities. Metrics for measuring code reliability during maintenance are very limited in practice.

This research introduces a new tool-based process model to help minimize the cost associated with the maintenance problems identified above. The process model and tool target the entire development team helping them improve their skills in software maintenance areas, such

as: program understanding, change impact analysis, regression testing, documentation, quality assessment, and code complexity metrics. Several new process-oriented metrics are introduced to help the team measure and track various attributes of the code base at three levels: function, feature, and product.

The new process model is a more detailed version of the IEEE-1219 standard, with emphasis on the major cost factors and the various responsibilities of the maintenance team. To assist the team in adopting the new process model, a maintenance tool “CMMR” was developed on two platforms: Objective C++ on the Macintosh, and Java on Windows. CMMR was designed to be easy to use by the entire maintenance team, developers and non-developers alike. It includes facilities to generate graphical views of the target system’s features based on dynamic analysis of code execution traces. Multiple graphical views are available at different levels of details to assist the team in various program comprehension activities. It includes feature-based support for more accurate change impact analysis and more focused regression testing. It offers an intelligent scheme for early defect

detection based on instantaneous tracking of code changes across multiple builds. CMMR also offers 16 metric measurements of various attributes of the product features and functions, such as complexity, maintainability, and reliability. The five new metrics introduced in this research are among these built-in metrics, which require breaking down the code base into basic tokens (keywords, operators, operands, etc.). The new metrics measure the reliability of an individual function based on its Function Maturity (age and number of releases), along with a new maintainability measure known as Feature-Based Function Maintainability. FBFM takes into account the maintainability of the function along with number of features that use the function. The other new metrics measure the feature reliability based on the reliability of its functions, and the product reliability based on the reliability of its features.

The proposed process model and tool were used in several case studies, one of which was a commercial Macintosh product. Initial feedback from participants working on these projects was very encouraging, and actual benefits were immediately observed; such as:

faster program comprehension, more focused regression testing, higher team productivity, higher code quality, and lower error injection. Some of the proposed metrics were refined as a result of actual usage. However, to see the full benefits of the new process model and tool, a full maintenance release cycle is needed. As for the metrics, like all other new software metrics, multiple release cycles are needed to refine them.

نموذج عملي لادامة البرمجيات مع اداة
وقياسات لتحديد الموثوقية اعتمادا على الميزات

اعداد: عبدالله مصطفى القيسي
اشراف: د. احمد الشرايعة، ا.د. وليد سلامة

الملخص

ARABIC SUMMARY

ان التكلفة المترتبة عن صيانة البرمجيات ومتابعتها تكون بالعادة اكبر من الكلف المترتبة عن تطوير النظام بحد ذاته. ويعود ذلك الى عدة امور منها: صعوبة فهم البرمجية، وتحليل نتائج اثر التغيير في البرمجيات، وفحص الارتداد البرمجي المترتب على اعادة صياغة البرمجية. وما يشاهد حالياً من انظمة صياغة البرمجيات ومتابعتها هو عدم تواجد الشمولية البرمجية فيها بما يكفي للحكم على جودة البرمجية ومحاولة فهم المشاكل وحلها في هذه النظم. كما ان الادوات البرمجية الحالية ليست على قدر كافي من السهولة للتعامل معها لصيانة البرمجيات ومتابعتها من قبل الفرق المسؤولة عن انتاج هذه الانظمة.

تقدم هذه الأطروحة نموذجاً عملياً وأدوات جديدة كحلاً للمسائل العالقة من خلال مقترح ثبتت فعاليته وذلك بتقليل الاثر الناتج عن صيانة البرمجيات من خلال تقليل الكلف الزمنية وبالتالي المالية. يهدف النموذج والحل المقترح في هذه الرسالة الى دمج آليات الصيانة للبرمجيات للمساعدة على فهم البرمجية ذاتها، وتحليل اثر التغيير على البرمجية، وفحص الارتداد على البرمجيات المتغيرة، والقدرة على التوافق مع المطلوب، بالاضافة الى قياس درجة النوعية للبرمجة ودرجة التعقيد والتوثيق. لقد تم تطوير العديد من المرجعيات البرمجية القياسية في هذا البحث لقياس درجة الوثوقية للمنتج البرمجي، والخصائص الفردية ووظائفها لهذه المنتجات. ولقد تم تطوير اداة تتعلق بالخصائص المفردة للمنتج البرمجي وتم استخدامها اثناء تطوير النظام وتطبيقه على حالات حقيقية واثبت نجاحه بما يتوافق مع الاهداف الموضوعية لهذا البحث.

اشتق نموذج العمليات المقترح من معيار IEEE-1219 وبشكل أكثر تفصيلاً مع التركيز على مسؤوليات فرق الإدامة للبرمجيات ويمكن النموذج من استخدام الأداة الجديدة والمسماة CMMR. وتمكن الأداة التي قد تم تطويرها لتعمل في بيئتي Java/C++ on Windows and Objective C/C++ on Macintosh من توليد واجهات رسومية تفاعلية بناءً على التحليل الديناميكي للبرنامج المعالج و تتبع تدفق تعليماته وبياناته. وتظهر الواجهات مستويات مختلفة التفاصيل لفريق الصيانة (الإدامة) لمتابعة نشاطاتهم أثناء تطوير البرمجيات. كما تمكن الأدوات إظهار التحريات عن الأعطال الممكنة حال حصولها مباشرة. ويقدم النظام المطور من النموذج ما مجموعه 16 مقياساً لخواص ووظائف البرمجية مثل: صعوبة فهم البرنامج (أو النظام) وإدامته ووثوقيته. كما تم إضافة خمسة معايير جديدة لقياس الصعوبة والإدامة والوثوقية. وتقيس المعايير الجديدة الوثوقية باعتبار العوامل المؤثرة مثل نضج المكونة البرمجية بناءً على عمرها وعدد إصداراتها.

تم تطبيق النموذج المقترح على عدة حالات دراسية، وإن أحداها هي من منتجات ماكتوش. وقد كانت التغذية الراجعة من الأشخاص الذين يعملون في هذا المنتج من خلال تطبيق الأداة المقترحة هنا ممتازة ومشجعة إلى حد كبير. وإن بعض أدوات القياس المقترحة في هذا البحث قد تم تطبيقها في منتجات برمجية حقيقية. ولمعرفة الفائدة الكبيرة من النموذج المقترح في هذه الأطروحة لا بد من دراسة دورة كاملة تتبني مراحل الصيانة للبرمجيات وتقرير دور وأثر تطبيق الأدوات القياسية المطلوبة لذلك.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Software maintenance is identified as a major cost factor [39]. It has received much less research than software development, despite costing a lot more [11]. Recently, there has been a growing interest in this area of software engineering, especially in the areas of processes and tools. However, the recent studies and maintenance tools are not comprehensive enough in terms of process model coverage and intended users. It is the strong opinion of the authors of this research that what is needed to address the ever-increasing cost of software maintenance is proper tools and process models.

A typical software maintenance process model includes several phases, each of which representing a significant cost factor. Moreover, different maintenance team members (developers, testers, documentation writers, and project managers) with a mixture of unique skills and specialties, usually carry out these process phases. Critical to the success of any new maintenance process and tool is the ability to address each and every maintenance phase and target every team member.

There are many maintenance tools and processes in research and practice today, as will be discussed in Chapter 2. However, many of these tools and processes are of limited practical use. For example, most of the test tools are disconnected from the actual source code being tested, while most development tools are intended for developers only; i.e. cannot easily be used by non-developers. This represents a major problem since software maintenance is a “social activity” that involves many stakeholders throughout the software lifecycle [59].

Another motivation for this research is the lack of practical reliability metrics in the area of software maintenance. Good software process models and tools are intended to improve software reliability, and reliability, in turn, reduces maintenance cost further. Most software reliability models and metrics ignore the maintenance process and focus on results, i.e. the number of observed defects, time to remove defects, etc. New process metrics are needed to allow the team, especially the project manager, to assess the reliability of the product and the individual features, and on build-by-build basis.

1.2 Aim of the Work

The aim of this research is to help software organizations significantly reduce the cost of software maintenance. It does so by tackling several maintenance cost factors at the same time. The research introduces a new maintenance process model along with a new maintenance tool that is intended to assist the maintenance team in adopting the new model. The process and tool are comprehensive enough to support all the major phases of the software maintenance process, and are easy to adopt and use by the entire team. Five new in-process metrics are also introduced in this research to support the new process model and help reduce maintenance cost. The new metrics are intended for measuring the **C**omplexity, **M**aintainability, **M**aturity, and **R**eliability of functions, features, and the product. These five metrics are implemented inside the tool, thus the tool is named “**CMMR**”.

The CMMR tool requires minimal setup, which involves identifying the target software project under maintenance along with the product features it supports. CMMR parses the target project code base and builds visual representations of the product features and functions. It offers multiple levels of details (graphical views) selectable by users

based on their level of programming expertise. Through its feature-based graphs, the tool helps each class of users attain and maintain better understanding of the project, so that code changes are safer, debugging more productive, testing more focused, documentation more reflective of code changes, and project management more effective.

CMMR offers 16 metric measurements in the area of software complexity, maintainability, and reliability, including the new ones introduced here. The proposed metrics are based on the premise that a software product is made up of one or many features (user scenarios), and each feature is implemented by one or many functions (source code procedure or method). Therefore, the reliability of a product should be computed from the reliability of its features, and the reliability of a feature should be computed from the reliability of its functions. The reliability of a function, in turn, is based on two new measurements: Feature-Based Function Maintainability (FBFM), and Function Maturity (FM).

This research claims that the code maintainability of a function increases as more features use that function. The higher the FBFM, the more likely the function will have defects, and the more maintenance is required to detect and fix these defects. Another claim is that the FM matters when measuring reliability. Maturity, in this context, refers to the time since the function's creation date and number of releases the function has been in. For example, a 5-year old function that was released to customers a few times is more reliable than a 1-month old unreleased function of equal complexity. Reliability in the context of this research is not a probability of failure, as it is commonly known. It's simply a number between 0 and 1 that represents the combined maturity and maintainability of the source code. Such number can be used to monitor and control code changes at the function and the feature levels.

The new metrics contribute to the reduction of maintenance cost in many ways. In coding, developers can minimize complexity and risk. In testing, testers can determine how much and where additional testing is required. In management, managers can get an accurate indication of the readiness of the features, and the product as a whole

, for delivery to the next phase of software maintenance.

1.3 Research Problems

The main problem this research intends to address is the ever-increasing cost of software maintenance. “Among the most challenging problems of software maintenance are: program comprehension, impact analysis, and regression testing” [9]. Other cost factors include project mismanagement, and inadequate development environments.

More details will follow next on each of these problem elements, with initial thoughts on how this research intends to address each element.

1.3.1 Program Understanding is Difficult

Software projects are getting too large and complex to fully comprehend, even for senior engineers. This problem is more evident in software maintenance, since in most software companies, many of the maintenance tasks are typically assigned to new engineers who lack enough understanding of the project to perform their jobs effectively. Very often, these new/novice engineers introduce new defects as they fix others. The missing element in this process is a facility that makes program comprehension easier for both expert and novice developers.

1.3.2 Change Impact Analysis is Inaccurate

Determining the effects of a proposed modification on the rest of the system is another major challenge [9]. Many defects are injected due to limited understanding of the full impact of making changes in *shared* code or data; i.e. common utility functions, generic classes, or global variables. Duplicating shared code is one way to eliminate the problem, but that proved ineffective as code cloning usually leads to more maintenance problems.

1.3.3 Regression Testing is Incomplete and Unfocused

Regression testing is another expensive testing process used to validate new versions of the software and to detect whether new faults have been added into the code [30]. Much of this cost is due to limited knowledge by testers about the nature of changes that go into each version. Testers tend to test many areas unaffected by the change, increasing cost and delaying defect detection and removal in the truly affected areas.

1.3.4 Project Mismanagement

Most of the problems with managing software maintenance projects are due to lack of timely information, or misinformation about the project. There are currently no existing tools with practical metrics that allow the project manager to measure the complexity and quality of the

project, feature-by-feature, and on build-by-build basis. A project manager often relies on his team members for task estimates and code reliability. Inaccurate information from the team causes the manager to make wrong and costly decisions resulting in schedule delays and further increase in maintenance cost.

1.3.5 Development Tools Offer Limited Maintenance Support

The tools available to developers, testers, and managers lack support for handling the cross-functional activities and process-oriented problems mentioned above. The tools don't offer any support for graphical views of the software project at the *feature* level, or any metrics for measuring the reliability of the software. Almost all of the tool offerings operate at the *module* (or *object*) level, which only makes sense to developers and only when there is one-to-one mapping between modules and features. Such mapping is usually intended by design but quickly deteriorates over time due to maintenance activities.

1.4 Definitions

This section defines the major concepts and keywords found in this research. Some of these terms are discussed in more details in the next chapter.

1.4.1 Software Maintenance

The IEEE definition of Software Maintenance is as follows: “*Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.*” [24]

This definition implies that software maintenance is all the work made on a software system after it becomes operational; i.e. after the first release to customers. This includes:

- Corrective maintenance - correction of defects.
- Perfective maintenance - enhancing the product to add new capabilities that originate from customers requests.
- Adaptive maintenance - adapting the product to changes in the environments; i.e. to make it run on a new operating system or a new hardware platform.

Some authors consider a fourth category: *preventive* maintenance, which includes the modifications to make the software more maintainable [50]. Software maintenance is therefore more than correcting errors. It includes all the changes made to the system after it has been delivered to customers at least once. Such changes involve many maintenance activities including: coding, testing, documentation,

and management. It's clear from the above definition and categorizations that software maintenance accounts for a huge amount of the total life cycle cost of a software system.

1.4.2 Software Maintenance Process Model

Most software vendors see no difference between development and maintenance and therefore use the same process model for both. While there are similarities between the two activities (i.e. they both include design, coding, testing, installation, and operation), there are many differences as well. Software maintenance includes several key process areas that are not present in development. Table 1.1 shows a listing of these areas.

Table 1.1: Some Maintenance Key Process Areas not Present in Development [62]

Management of problems
Acceptance of the software
Managing transition from development to maintenance
Role of the user, operators and support staff
Maintenance planning
Management of the maintenance personnel
Software management (improvements, performance)

Obviously, a standard development process cannot be adopted unaltered during maintenance. Several maintenance process models have been proposed, as a result. One such process model is the IEEE-1219 [24], shown in Fig. 1.1.

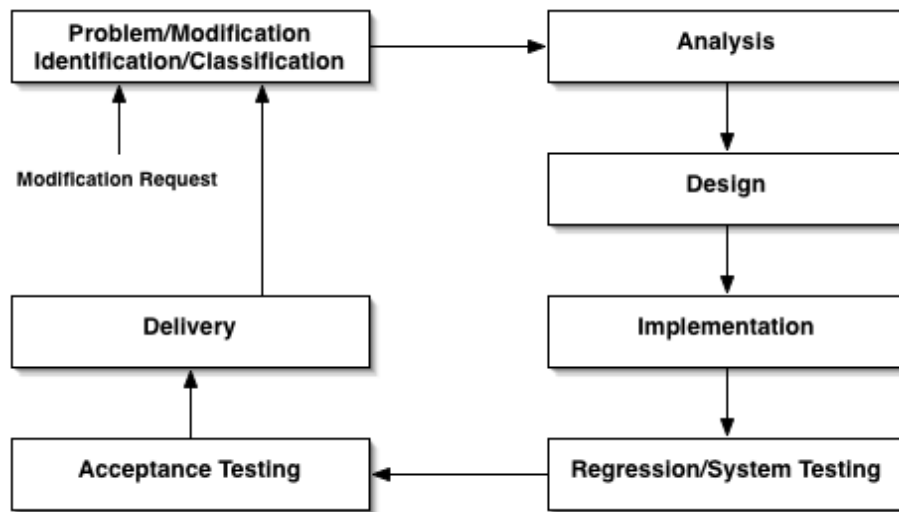


Fig. 1.1: The IEEE-1219 Maintenance Process [24]

The IEEE process model begins with the identification/classification of a problem or a modification to the source code. The next phase is analysis of the problem or task at hand. A preliminary plan is put in place for design, implementation, test, and delivery of the task. Next, comes the design phase where the solution is designed in more details. This phase includes several areas including the identification

of the code affected by the change, modification of documentation, and test plans. The next phase is implementation where the solution is actually implemented. This phase includes sub-tasks such as coding, unit testing, risk analysis, and review. The next phase, regression testing, is where parts of the system are validated after the change is made to ensure compliance with the requirements and that no other faults have been introduced. Acceptance testing is the final test that comprises all tests performed internally (a.k.a. alpha testing) and external (a.k.a. beta testing). The final phase is delivery, where the modified system goes into a release mode to the customer.

ISO-IEC/12207 is another process model that has been proposed for software maintenance. It's more comprehensive than the IEEE specification to include several more detailed areas, and more focus on the system's life cycle [27]. There are other slight variations of these two models in the market today [9], including the one introduced here.

One thing that these process models have in common is that they all list program comprehension, impact analysis, and regression testing as a core set of activities. Each of these phases is defined next.

1.4.3 Program Understanding

Program understanding is a domain of computing science dealing with the process used by engineers to understand programs before their modification. It's also known as "Program Comprehension", and the two terms will be used interchangeably within this dissertation. Program understanding is vital in software maintenance since it facilitates many techniques such as: removing defects, extending functionality, reverse engineering, and reengineering.

Program comprehension is a reading/viewing process of the source code and available documentation. It does not involve any writing or modification of code. If skipped or not thoroughly done prior to software modifications, bad fixes could potentially be injected in the software, and/or bad changes inserted in the code corrupting the program structure and leading to more costly maintenance. Available estimates indicate that the percentage of maintenance time consumed on program comprehension ranges from 50% up to 90% [15].

There are three strategies that can be employed during program comprehension: top-down, bottom-up, and combination of the two. The bottom-up approach starts with the source code and constructs the

high level design from it using chunking and grouping strategies [15]. The top-down approach is feature-based, and involves identification of the software components responsible for implementing a feature or a task [20]. The combined approach mixes the use of different methods as needed.

As systems increase in size and complexity, program understanding becomes more difficult. Tools are one way to help with program understanding, and a few good ones are starting to appear in research, and some in practice. At minimum, a program-understanding tool must be able to support one or more of the three methods above. It must maintain a repository of architectural and behavioral information about the program. It must organize that information and present it to the user in a visually comprehensive way.

For a program comprehension tool to be successful, it must have low setup cost. If the setup cost is relatively high, then the target program understanding tasks must be large and complex enough to justify the cost of installing, setting up, and using such tool. Another key to successful adoption of any new tool is that it must fit the environment

of the target user. For example, a Mac engineer will be biased against a tool if it did not have a graphical user interface (GUI) and it did not integrate with the current development environment on the Mac.

Automation is another key to the success of a program comprehension tool. Most tool automation work in this area is still limited to small project, and has not been proven on real-world legacy programs. The tool presented here is a step in that direction. Section 2.3 lists some examples of other tools in practice.

1.4.4 Feature-Based Code Analysis

This is a type of program-comprehension analysis that focuses on the identification of source code related to a user feature or concern. Methods used here fall into two categories: static and dynamic [67].

Static techniques involve examining the source code and design documents to create intermediate representations for further analysis. The source code may not be complete since there is no need to build and execute the program. Static analysis can therefore be used more rapidly and without preparation. The Unix “grep” tool, some commercial Computer-Aided Software Engineering (CASE) tools, and class dependency analysis tools all fall into this category.

Dynamic analysis, on the other hand, involves executing an instrumented version of the program to identify the execution path (call graph) of the intended feature. It requires having the entire code base in order to build and execute the program. Dynamic analysis has more setup cost due to several reasons: first, having to locate the full set of source code; second, adding profiling information so that when the program is run a trace of logging information is generated; third, mapping the log information to a form that is easy to view and extract information from. There is a loss in performance when running an instrumented program, so it's usually a common practice to disable all the profiling information when producing the final version of the program, or when the feature in question is completed (i.e. comprehended).

It's worth noting here that the Java Platform Debugger Architecture (JPDA) produces logging information without any code instrumentation, but this is limited to Java programs only. Linux-OS offers a similar capability by adding a wrapper process around the target program. Mac OS X 10.5 offers a similar capability called DTrace, which allows traces to be generated without adding profile

information. It's generally not recommended for a tool to touch the target code to add profile information, or otherwise.

Despite the setup cost and performance issues just explained, dynamic analysis produces more accurate results than static analysis. This accuracy is seen especially when dealing with object-oriented software where polymorphism and dynamic bindings make it hard to define the dependencies between the various objects [13]. Another challenge that this method handles with ease is dynamic callback mechanisms, which are often seen in network and some graphical user interface applications. Such dependencies may not be known until the program actually runs.

Recently, a few studies are beginning to cover the problem of locating features in source code [41][53][6], and a few tools have been developed [52][70][28] to assist in that direction. All these studies and tools are intended for use by developers and offer little to no support to non-developers. No tool or technique offers a complete solution to developers either. In most cases, further debugging techniques are needed to gain full understanding of the features.

1.4.5 Change Impact Analysis

Software change impact analysis estimates the potential affects of changes on the rest of the software. A major problem for developers is that “seemingly small changes can ripple throughout the system to cause major unintended impacts elsewhere” [34]. Developers are able to evaluate the change before actually committing to making the change, or even after making the change. This helps developers in two ways: estimating costs of proposed changes and selecting between different implementation alternatives, and reducing risks associated with releasing changed software [2].

Change impact analysis is such an important activity that it's shown as a distinct phase in maintenance process models. The aim of carrying out the analysis is to identify and minimize negative side effects. An ideal change analysis consists of identifying the changed code and related code affected by the change, and assessing the overall impact on certain metrics such as quality, size, complexity, performance, resource requirement, and regression testing. If evidence leads to dramatic increase in these metrics, other alternatives are considered, and the process is repeated until an acceptable solution is found.

Failure to assess the impact of a software change could lead to dramatic problems down the road, causing a significant increase in lifetime maintenance cost. The impact may not always be internal in the code base. There may be external constraints such as packaging, training, customer support, government regulation, etc. All these factors and others must be taken into account during a change impact analysis.

Impact analysis techniques fall into two categories: static and dynamic. Static techniques are predictive in nature; i.e. the analysis takes place before the change is made. Transitive closure of a call graph and static slicing are examples of this. Dynamic methods are based on program execution, as discussed in Section 1.4.4. Another classification of change impact analysis methods is in [34]. According to the article, the basic techniques for supporting change impact analysis fall into several categories: data flows, data dependency, control flow, program slicing, test coverage, cross referencing, browsing, logic-based defects detection, and reverse engineering algorithms.

Code coverage tools and techniques are often used during change impact analysis. Code coverage information includes profiling information gathered from a specific version of the program. They track which functions and which statements executed by each test, without tracking the frequency of the execution. For this information to be useful it must be updated all the time to keep it in sync with the evolving target program. Aristotle [23] is a good analysis system to measure code coverage.

1.4.6 Regression Testing

Regression testing is an expensive testing process used to validate new versions of the software and to detect whether new faults have been added into the code. “It has been estimated that regression testing may account for almost one-half of the cost of software maintenance” [30]. Much of this cost is due to limited knowledge by the testers about the nature of changes that go into each version. “Regression testing should be focused on those areas that are most likely to contain the introduced faults” [16]. This avoids the wasteful retesting of unaffected areas, focusing the testing effort on the impacted areas only, and immediately after the change is made. Better focus on impacted areas leads to better fault exposure capability, which leads to more efficient defect removal.

Most regression testing selection methods are based on code (known as “white-box” testing). Other techniques are based on specifications (known as “black-box” testing). Code-based regression techniques are language dependent and good for analysis at the unit level. They are not scalable to testing big components. They are time-consuming, and require that the tester understand the underlying code. Specification-based methods don’t require any programming experience, and are suitable for testing of all components regardless of size. However, the selection process is subjective, where each tester has his/her own criteria for the selection process.

Any test run can only identify defects found in a specific test. However, many defects remain uncovered due to being in other related test runs but neglected as being unrelated. In addition, there are many variables when running a test run which could hide certain defects; such as: program state, data values, system configuration, and operating conditions. The number of possible test runs ends up being very large, and running all possible tests becomes very expensive. In other words, even the most comprehensive testing method cannot detect all defects in a program. Although, regression testing only focuses on the impact

of recent changes to the code, the challenge of finding all defects caused by any particular change is not a trivial process.

To add to the challenge of testing, the number of potential defects in a program increases exponentially with the size of the program. The larger the program, the more risky the changes are, the more defects detected and not detected, the more cost associated with detecting/removing/verifying the defects, and the less reliable the program becomes. These factors contribute to the sad reality of today's program quality. As demand for quality increases, the current software quality practices will become less adequate in the future.

To illustrate how imperfect the current testing methods are, it's estimated that the current U.S. average for defect removal is only about 85% of the defects introduced during development and maintenance [10]. A program with one million lines of code will have 7500 defects at delivery. About 1/3, or 2500, will be serious enough to stop the application from running or create erroneous output.

A related issue that increases the number of latent defects is "bad fix injection". For any given defect repair, there is a good possibility that it

may contain or introduce other defects, some of which are more serious than the original. It's estimated that the average percentage of bad fixes is about 7% of all defect repairs [12]. Change impact analysis prevents bad fix injection during development, and regression testing prevents them from getting delivered to customers.

A good regression test has two main characteristics: first, identification and testing of the affected areas; second, avoiding wasteful testing of unaffected areas. So, it really boils down to finding all the features and user scenarios that were impacted by a particular change and testing them all. Not as trivial as it sounds. Automation tools, with the aid of metrics, play an important role in this area. Many regression testing selection techniques have been proposed and will be discussed in more details in Section 2.5.

1.4.7 Software Complexity Metrics

Software metrics are numerical data related to software to get better estimates on labor, resources, and reliability of programs. Example software metrics include: product size in terms of lines of code (LOC) or functionality, planned vs. actual cost, estimated vs. actual staffing levels, number of active defects, percentage of test cases passed,

code covered by unit testing, and code complexity. Metrics have been in use since the 1970s when Lehman [36] used them to analyze the evolution of the IBM OS/360 system. Lehmann, Perry, and Rami [35] explored the implication of the evolution metrics (number of modules per release) on software maintenance. Burd and Munro [8] analyzed the influence of changes on the maintainability of software systems.

Some of these metrics have been adopted and used by almost all software organizations, while others were rejected. Some of the metrics that have not been adopted are proven theoretically, but they don't work in practice on real life commercial projects. Others are not accepted because they may be used to measure performance – something that most software engineers are not comfortable with. In theory, metrics are designed to help software management do better planning, organizing, controlling, and improving of the software projects, but some managers do use them during performance evaluation. Metrics are also used in other managerial areas, such as: cost estimation, scheduling, resource allocation, and tracking activities. Although less common, metrics can be used during development to improve software maintainability, decrease

complexity, and insure high quality of the product.

A special class of software metrics measures the complexity of the source code. Such metrics are away to assess the quality and reliability of software [18]. Several maintenance characteristics are affected by code complexity, including: understandability, modifiability, maintainability, reliability, and testing. Code complexity is hard to define simply because it's subjective - code that is complex to one programmer may not be to another. Objective measures were introduced when McCabe introduced the Cyclomatic Complexity measure in 1976 [42].

McCabe's assumption was that complexity is related to the number of control paths in the code. He believed that the size of the code (LOC, for example) is irrelevant to complexity. He developed a method that maps a program to a directed, connected graph where the nodes represent decision statements, and edges represent control paths. He stated that the complexity of a program (named it "Cyclomatic Complexity") equals the number of enclosed regions in its mapped graph plus one. This number is basically calculated by counting the

number of decision points such as “if” blocks, “switch” cases, “do”, “while”, and “for” loops. He concluded that a program with a cyclomatic number higher than ten is problematic and needs reduction. There are several techniques to reduce the complexity of a function with a high complexity value, including: eliminating useless branches, unrolling loops, tuning switch/case statements, and breaking up the function into smaller functions.

McCabe’s Cyclomatic Complexity metric is shown in Equation (1.1):

$$VG = E - N + P \quad (1.1)$$

Where, VG: McCabe Cyclomatic Complexity

E: Number of edges of the function’s graph.

N: Number of nodes in the function’s graph

P: Number of connected components.

McCabe’s metric is easy to use and agrees with many empirical data, however some argue that the number of control paths does not fully describe code complexity. Others claim that some functions are long and complex by nature, and don’t lend themselves to have a VG value under 10. Examples of this include: a scheduler, locking procedure,

and performance critical code. A better indicator of complexity should take the number of statements (LOC) into account.

Halstead went a step beyond statements and LOC. He based his metrics on the number of operators and operands within the statements [21]. Halstead metrics are known as Halstead Software Science. He used a measure of each function in terms of operators and operands. He defined several metrics to compute program length (N), see Equation (1.2), vocabulary (h), see Equation (1.3), volume (V), see Equation (1.4), and others. Halstead metrics are criticized for being too difficult to compute, while others found faults in his assumptions and mathematical equations.

$$N = N1 + N2 \quad (1.2)$$

$$H = H1+H2 \quad (1.3)$$

$$V = N \log_2 H \quad (1.4)$$

Where, N: Program Length

H: Vocabulary

V: Volume

N1: Number of all operators in the code.

N2: Number of all operands in the code

H1: Number of unique operators in the code

H2: Number of unique operands in the code

Another code complexity metric is Maintainability Index (MI) [46][65], which is gaining popularity lately. The MI metric is composited from several other metrics: McCabe cyclomatic complexity, Halstead Volume, LOC, and Lines of Comments. MI is a good indicator of program maintainability, and because it's based on several reasonable complexity metrics, it's believed to be more accurate than each individual metric when used separately. The MI metric is shown in Equation (1.5):

$$MI = 171 - 5.2 \ln(VG) - 0.23V - 16.2 \ln(LOC) + 50 \sin(\sqrt{2.4 \text{avgPerCM}}) \quad (1.5)$$

Where, MI: Maintainability Index (0-171)

VG: McCabe Cyclomatic Complexity

V: Halstead Volume

LOC: Number of lines of code.

avgPerCM: Ratio of comments to source code

The higher the MI value, the better the maintainability of the function. Values higher than 84 indicate good maintainability. Range 65-85 indicates moderate maintainability. Range below 65 is considered low maintainability. This research uses a variation of MI in the computation of Feature-Based Function Maintainability Metric (see Section 3.3.1).

1.4.8 Software Reliability Metrics

The formal definition of software reliability is: “the probability of failure-free operation of the software for a specified period of time in a specified environment” [45]. In terms of measurement, there is still no good way of measuring software reliability. See Equation (1.6) for the original metric of calculating the software reliability.

$$\text{Reliability (R)} = \exp(-\lambda_t * t) \quad (1.6)$$

Where, λ_t : the number of failures/hour

t : the time period for which the reliability is to be calculated

Range of values for R is 0.000 to 1.000, with 0 indicating no reliability, and 1 indicating maximum reliability. But, like almost all software metrics and models, this metric has its unrealistic assumptions and limitations, such as specifying the time and environment in the above definition. In general, software reliability measurement cannot be performed easily and directly. Other related software attributes that lead to reliability are measured instead, such as code complexity, faults, and test coverage. It's well known that the high complexity of software is the major contributing factor of software reliability problems

. Several complexity metrics are based on program size measures, such as LOC, and are used for software reliability assessment [25].

When computing reliability it is important to focus on the trend of several reliability measures rather than one particular measure. Typically, these measures or computations take place per software revision (a.k.a. build), and/or when major changes are made in the software. Observing a trend gives a good idea about the improvement or deterioration of code quality and maintainability, which allows the manager to insure that trends are going in the right direction.

1.4.9 Maintenance Management

Management is “the process of designing and maintaining an environment in which individuals, working together in groups, accomplish efficiently selected aims” [64]. In software maintenance, the selected aim is to provide high quality product with minimal cost, and to maintain good maintainability of the project during the entire lifespan, not just the current release. The main responsibilities of a successful maintenance manager are: planning, organizing, staffing, leading, and controlling [64].

Several problems confront a software manager responsible for managing a maintenance project. First: inexperienced personnel. It's estimated that 60-80% of the maintenance staff is newly hired [48], and that 25% are students [62]. Second, many software organizations still perceive maintenance as non-strategic. Third, code maintenance is considered by many engineers to be non-glamorous when compared to development. So, high turnover and low morale are constant problems for the manager to contend with. Forth, software managers are often faced with budget constraints, and as a result tend to focus on short-term incremental changes, when the better strategy may be a total rework of the entire system [32] or portions of it.

Another problem that confronts managers is that, often, effective management requires the use of metrics to measure certain criteria such as LOC by developer, defects found by tester, etc. For managers to manage well they need to be able to accurately measure the progress of the project, which is solely based on the progress of the team working on it. So, measuring the productivity and efficiency of the individuals becomes a necessity for obtaining optimal project results. However, developers and testers don't respond very well when their

productivity is measured by how many lines of code they wrote in a single day, or how many defects they found or fixed.

What some managers elected to do instead of focusing on measuring team productivity in terms of numbers, they measure productivity in terms of teamwork. For example instead of counting lines of code, a manager would look at the maintainability of the code. Instead of focusing on short-term results (i.e. is the build on time? Or, is a particular function complexity under 10?), a manager would focus on long-term objectives; such as: is the quality acceptable? Is morale high? Nevertheless, a successful manager must know of all the measurement techniques and use them at the right place and at right time. A way of doing it is to reward those that meet certain criteria, and not reward others that don't.

Use of metrics in software engineering is one such sensitive area where managers and engineers don't agree on. Forcing a metric on a team is not recommended, so a good strategy to adopting a new metric should be a slow one. A slow strategy is one that creates a "measurement culture" and involves a few gradual steps: start small,

explain why, share the data, define data items and procedures, and understand trends [66]. Measurements are useless if not used to improve processes and work smarter to achieve the organizational goals.

1.5 Assumptions and Hypotheses

In addressing the high cost of software maintenance, this research builds its solutions on a few hypotheses and assumptions, which directly and indirectly contribute to the cost factors discussed in previous sections. Some of the assumptions are obvious while others are based on the author's past experience and review of related studies, and questionnaires. These assumptions are listed first, followed by the research hypotheses that are based on these assumptions.

Assumptions:

A1. Most maintenance tasks or defects are related to a single feature in the product.

A2. Graphical representations of the codebase aid in program comprehension.

A3. Source code is getting too complex for some testers to perform white-box testing.

A4. Testers and documentation writers are usually kept in the dark on code changes.

A5. Many defects are detected much later after they were introduced.

A6. Managers often far removed from the code base to make good and timely decisions.

A7. Almost all maintenance tools are for developers only.

Hypotheses:

Based on the above assumptions, the following research hypotheses were constructed.

H1. Keeping the code path of one particular feature isolated from other unrelated code complexities insures better program comprehension of the feature at hand. It also results in more focused testing, more accurate documentation, and more effective management.

H2. Maintaining and presenting each feature as a call graph further improves understanding, testing, documentation, and management of the feature. A tree representation, with nodes representing features and function names, and edges representing call chains, is the most natural call graph representation. This presentation is especially

helpful for unskilled engineers, or those who are new to the project. Other benefits include: reduction in the learning curve of the code base and the features it supports, better communication within the team, reduction in defect fix time, increase in fix quality, and decrease in error injection.

H3. Testers attempting to perform white-box testing have difficulties understanding the low-level source code and extracting the information necessary to determine the best testing strategy. Showing a graphical representation of each feature, in terms of underlying function names and the comments in these functions, is much less intimidating to testers. It allows them to easily attain the knowledge needed to perform their work more effectively, and without having to deal with the lower-level source code details.

H4. Keeping testers and documentation writers away from the source code delays the detection of defects, reduces the accuracy of defect reports and documentation, and increases the time needed to remove defects. Giving access to these graphical representations is invaluable to testers when accompanied with a mechanism to automatically

detect and track code changes (i.e. which functions changed, and what features were impacted). Regression testing becomes much more focused as a result.

H5. Quick detection of errors not only improves testing productivity. It also leads to more effective debugging and error removal, minimizes further deterioration of the source code, and dramatically decreases the overall maintenance cost. Detecting errors immediately after they are introduced makes error removal more efficient, because the changes that caused the errors are more likely to still be in the minds of the developers who made the changes.

H6. Hands-on observation of the project status and reliability metrics by the project manager results in better and more-timely project decisions. The ability to measure the quality of the product, at the feature level, and after each build, is important to managers. Other benefits from these metrics include: planning, staffing, leading, training, estimation, and controlling the project activities.

H7. For maintenance tools to be effective they need to be useable by the entire maintenance team, not just developers. Testers, writers, and

managers prefer to work at the product-feature level, so the existing “source code” based tools are not very helpful to them. What’s needed is common tool that meets the demands of powers users and hides any complexities from the novice.

1.6 Research Importance

Every software system that gets delivered to customers, and succeeds in the marketplace, must be maintained in order to maintain and build on that initial success. As software systems get larger, older, and more complex, the maintenance cost will continue to increase. Currently, maintenance cost accounts for 60-80% of the total lifetime cost [39], and if left unchecked, the cost will continue to increase. Passing that cost to customers is not a viable option in today’s competitive software industry. Reducing cost by improving the maintenance process and tools is the only viable approach.

Key to success of any new process model and tool is ease of initial adoption, and ultimately, a full adoption. Often times, processes and tools are rejected because they either require expensive setup, hard to use, or simply not practical enough. The ideas proposed here strike a balance between theory and practice. Together, the new process

model, the CMMR tool, and the built-in metrics promise to help software organizations maintain their software projects more effectively and efficiently.

The CMMR tool that is used to demonstrate this research targets the entire team regardless of their background and expertise. It may potentially become a product or a free software maintenance tool in the public domain. The complexity, maturity, and reliability metrics introduced here should be of great value, not only to practitioners, as will be shown in this research, but to theorists as well. These metrics could serve as basis for many derived metrics beyond what is shown here.

The proposed process model and tool are easy to adopt and use by the entire maintenance team. The initial setup, which is very minimal compared to the potential benefit, consists of setting up the tool's database with selected product features – an operation that could easily be done by a developer or a senior tester. The process is gradual in that not all product features need to be supported right away, only the ones that need more comprehension and/or precise regression testing.

1.7 Scope and Limitations

This section outlines the known scope and limitations of this research.

1. The main objective of this study is to reduce the overall maintenance cost of software systems. This cost typically includes other factors and activities beyond the four areas covered: development, testing, writing, and managing. Other areas not covered here include: requirement engineering, configuration and release, customer beta testing, deployment, and training. These activities are all outside the scope of this project and are not impacted by it. In other words, their contribution to the overall maintenance cost, believed to be relatively minimal, is not affected in any way.

2. It's difficult to quantify the savings in maintenance cost for all organizations and for all software projects. There are many variables involved here, such as: the level of acceptance and adoption of the process and the tool by different classes of users, product size and complexity, product age, among other factors. To obtain concrete numbers of the savings for any given organization and product, two full release cycles of equal levels of requirements are needed: one without the adoption of the proposed process model, tool, and metrics, and

another full cycle with. A comparison is then performed between the two costs, in terms of man months, or dollars, to determine the savings.

3. New software metrics take multiple release cycles and several refinements before they are accurate enough for actual use. The proposed metrics are no exception. They were updated several times during the case studies, and there will be more refinements as more case studies are conducted.

4. The CMMR tool could potentially store information related to the engineers, assigned features, product releases, defects, etc. Such relations can then be used to generate data that help project managers do better resource allocation, defect assignment, build comparisons, among other things. The tool is designed in a way that is extensible to include such information.

5. In terms of high-level language support, it's desirable to have the tool fully functional on both the Macintosh and the Windows environments, with full support for the three high-level programming languages (C/C++/Java). Due to cost constraints, a compromise was to have Java implementation on Windows, and C/C++/Objective

C/Objective C++ on the Macintosh. Other implementation flavors can be added in future revisions.

6. Not everything in the tool is fully automated. Some operations are manual, such as identification and naming of features, and management of builds. Key to accepting any tool and using it during software maintenance is minimal initial setup and maximum level of automation. The first objective is believed achieved in the first version of the tool. The second is partially realized with more automation planned in future versions.

1.8 Organization of the Dissertation

This completes the introduction chapter of this dissertation. The next chapter discusses related work. Chapter 3 discusses the methodology used in this research and highlights its major contributions. Chapter 4 contains the case studies performed as part of this research. Chapter 5 discusses the results of the study. Finally, Chapter 6 concludes this research by summarizing its results and highlighting the remaining works ahead.

CHAPTER 2

RELATED WORK

2.1 Introduction

This chapter discusses some of the related studies and research papers that were reviewed prior to the writing of this dissertation. The chapter is organized into seven main sections as related to software maintenance and its major cost factors: software maintenance, in general, program comprehension and visualization, change impact analysis, regression testing, feature-based code analysis, and software complexity and reliability metrics, and other cost factors. Each section discusses the major related studies and tools, listed in chronological order. This research does not attempt to compare these studies and solutions among themselves, as other excellent papers have already done that [63][23]. The purpose here is to provide background information for the proposed work in the next chapters.

2.2 Software Maintenance

The Lehman's laws of evolution state that for software to be successful it must continuously change over time. "A program that is used in a real world environment necessarily must change or become progressively less useful in that environment". Table 2.1 lists the Lehman's evolution laws.

Table 2.1: Software Evolution Laws [36]

Continuous Change	Systems must continually adapt to the environment to maintain satisfactory performance.
Continuing Growth	Function content of systems must be continually increased to maintain user satisfaction
Increased Complexity	As systems evolve they become more complex unless work is specifically done to prevent this breakdown in structure.
Declining Quality	System Quality declines unless it is actively maintained and adapted to environmental changes.

“Software Maintenance”, by Canfora and Cimitile [9] is a comprehensive on-line article on software maintenance. It defines software maintenance and categorizes its types, costs, and challenges. It then introduces general models and management of the maintenance process. Finally, it covers two areas that are related to maintenance, namely reverse engineering and reengineering. The article presents the two as solutions to many problem areas in software maintenance. The two solutions seem to be too dramatic, as most maintenance tasks don't require reverse engineering or reengineering. The article concludes by saying that better solutions are needed in light of many software systems growing in size, complexity, and age.

2.3 Program Understanding and Visualization

“Program understanding is the ill-defined deductive process of acquiring knowledge about a software project through analysis, abstraction, and generalization” [63]. This acquired knowledge aids in performing all types of maintenance work: adaptive, corrective, perfective, and preventative. Tilley and Smith [63] claim in their technical report titled “Coming Attraction in Program Understanding” that program comprehension tools must include support for data-gathering techniques, advanced schemes for organizing knowledge, and hypertext-based information exploration. Key to understanding legacy systems is organizing the knowledge about the subject project and presenting its architecture and design in a graphically intuitive way [3][34]. Such organization allows the user to maintain full view of the project as a whole, and selectively navigate through different parts of the project at the appropriate levels of details.

A variety of visualization tools and techniques are available to facilitate program understanding. They all make use of color and graphs to represent components of the program: such as objects, modules, call graphs, lines of code [51][29][17]. Some add metric information to

assist users in measuring complexity, among other things [51]. Others make extensive use of advanced visualization techniques [38][26]. Another family of tools focuses on recording program understanding. Bennet and Younger's paper "Model-Based Tools to Record Program Understanding" [4] provides a good summary of such tools.

Object-oriented (OO) programming languages, such as C++, are good for development but recent studies suggest that it may not be any better in the maintainability of programs as other third-generation languages (3GL) [31]. Wilde, et. al. [68] suggested that the large number of small methods in the OO environment make it difficult to trace program functions. In addition, OO aspects such as inheritance and dynamic binding contribute to the difficulty in determining program functionality.

On the importance of visualizing the software, Ball and Eick [3] wrote in their paper "Software Visualization in the Large": "Software visualization tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behavior. Pictures of the software can help slow knowledge decay by helping

project members remember--and new members discover--how the code works.” The article highlights the biggest problem in most existing software visualization tools in that they don’t scale well to large commercial projects, because their objective is to decompose the product into modules. The authors developed a program comprehension tool that visualizes the program’s text involving change history, difference between releases, and static and dynamic properties of the code. The tool is used daily within Bell Labs, the authors claim, for two decades by thousands of engineers. Again, only developers are targeted by their system.

Pinzger, et. al. [49] introduced a visualization approach that provides graphical views of source code and release data, in an effort to assist developers in understanding the system’s architecture and design concerns. The paper stresses the importance of extracting and building abstracted views of the system architecture and design as a prerequisite to successful program understanding and maintenance. It targets developers only and focuses mostly on source code evolution over multiple releases of the software. It does not offer any solutions to testers, writers, or managers, and it does not offer any dynamic

analysis of features, but the paper does point out that as possible future work.

The value of tools cannot be overestimated as it ranks second to having maintenance specialists with domain experience. Capers Jones [11] lists a variety of tools having key factors with positive impact on maintenance. Table 2.2 lists the top ten factors in ascending order.

Table 2.2: Impact of Key Adjustment Factors on Maintenance [11]

Maintenance Factors	Plus Range
Maintenance specialists	35%
High staff experience	34%
Table-driven variables and data	33%
Low complexity of base code	32%
Test coverage tools and analysis	30%
Code restructuring tools	29%
Reengineering tools	27%
High level programming languages	25%
Reverse engineering tools	23%
Complexity analysis tools	20%

Capers Jones [11] points out that “the imbalance between software development and maintenance is generating a significant burst of research into tools and methods for improving software maintenance performance”.

Another article “Visualization Techniques for Program Comprehension”, by Lemieux and Salois [37], provides a good explanation on the history and terminology of software visualization. The article also reviews several more recent visualization techniques, graphical views, and animations that illustrate program behavior. Storey, et. al. [61] in their paper “Remixing Visualization to Support Collaboration in Software Maintenance”, emphasize the importance of program understanding through visualization benefiting the entire team. For example, a developer may use it for change impact analysis, while a tester may use it for regression testing. Visualizing software goes beyond program understanding and covers other areas including communication and software evolution.

2.4 Change Impact Analysis

There are many papers written to cover various techniques for performing change impact analysis. Transitive closure of a call graph

[5] and static slicing [56] are examples of static analysis methods. Dynamic methods [33][2] are based on program execution.

Lee [34] wrote her dissertation titled “Change Impact Analysis of Object-Oriented Software” where she emphasized the need for mechanisms to understand how a change in a software system will impact the rest of the system. Object-oriented software was supposed to put an end to this problem with features like encapsulation, inheritance, aggregation, polymorphism, and dynamic binding. But the ripple effect problems are still there and are more difficult to detect and control than in procedural systems. The research introduced a set of data-dependency graphs, algorithms, and change impact metrics to evaluate the change impact quantitatively, and a prototype tool to evaluate the algorithms. The research also claims that it can assist testers during regression testing, and in supporting cost estimation and schedule planning. Although it claims support for regression testing, testers must deal with classes and objects rather than the more natural features that they and their customers are comfortable with.

A paper was written by Apiwattanapong, et. al. [2] on the issue of dynamic change impact analysis, titled “Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences.” The authors claim that there are two known dynamic impact techniques, as of May 2005: CoverageImpact and PathImpact. They introduced a new technique “Execute After” which is claimed to be better than both. All three techniques attempt to identify program entities (methods) that may be affected by a change for a given set of program executions. The result is a set of methods to analyze the effect of change and perform regression tests. Obviously, here again the research is operating at the code level and its results are only beneficial to developers. There is also the flawed assumption that *all* methods that get executed after a changed method are affected by that change and requiring regression test. Moreover, the research is still not applicable to real software released to real users, as its authors state.

There are other prototype tools to demonstrate change impact analysis methods, or use them in other software maintenance techniques. Rutgers' Prolang [55] includes several tools to help programmers with symantic change impact analysis. The tools, which run within Eclipse

interactive programming environment, are based on program executions of the program before and after a change is made. It combines static information to show the possible call graphs and their impact on testing. A related field combines change impact analysis with execution profiles in locating features. Rohatgi, et. al. [53] presented an approach and a case study for measuring the impact of change on all components in the trace and ranking the results. Their hypothesis is that the smaller the impact on the rest of the system the more likely the component is specific to the feature under study. Another recently emerging field is on predicting code changes. “Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach”, by Sharafat and Tahavildart [57], which uses change history and code metrics to determine the classes that are likely to change in the next release of the software system.

2.5 Regression Testing

A good study of available regression testing selection techniques is by Rothermel and Harrold [54], “Analyzing Regression Test Selection Techniques.” A simple risk model is proposed to compute the risk exposure of each function based on the probability of fault and cost (impact) of fault if it occurs in production. The selection is then based

on choosing the test cases with the highest risk exposure values. Determination of success of any selection technique is based on the number of defects found (defect find ratio), and the efficiency at which they were found (how quickly).

Another paper by Elbaum and Munson [16] evaluates regression test suites based on their fault exposure capability. The research developed a methodology based on test execution profiles and “evolutionary fault indexes” to provide an assessment of the overall regression testing activity and the suitability of each individual test. It does not really present a new test selection technique, yet it highlights the importance of identifying the areas of the code impacted by each change where faults are most likely to lie.

Obviously, testing is a tedious process and a lot of it is mechanical and repetitive. Automation tools, with built-in metrics, can be very helpful to testers. There are many test tools with various levels of automation, including: GJTester (testing Java code only), LDRA Testbed (application test tool), TestWorks (functional test tool), TestComplete, Vermont HighTest, Netvantage Functional Tester, JUnit

(for Java unit testing), ApTest Manager (test management tool), and WebKing, by Parasoft, for testing and analysis of websites and web applications. Misuse of such tools could lead to worse results than testing without them, so these tools must be used with caution. Most regression test selection methods are based on code (a.k.a. white-box testing). The method proposed in this research falls in this category. A few techniques are based on specifications (a.k.a. black-box testing) [14].

Many studies have been conducted on the importance of regression testing during software maintenance. In his study “Cost-Effective Regression Testing”, Khoury [30] claims that regression testing may account for almost one-half of the cost of software maintenance. He points out the importance of choosing the right regression test selection technique to improve the cost-effectiveness of regression testing.

2.6 Feature-Based Code Analysis

On the issue of locating features in source code, Wilde and Schully [69], considered by many to be the pioneers in this field, introduced “Software Reconnaissance” - a simple method of identifying the feature’s code components.

The goal of their research was to support developers in software maintenance activities and in extending the functionality of legacy code. Despite having no support to non-developer team members, the tool is very powerful in terms of excluding log noise that is not related to the feature under study.

Robillard and Murphy developed FEAT [52], a tool for locating, describing, and analyzing concerns in Java source code. It describes features/concerns in terms of graphs between program elements such as classes, methods, or any field in the project. These descriptions are presented to the developer visually inside their Java development environment for further analysis and comparison. Their tool resembles the tool proposed here in many ways except it addresses only one aspect of software maintenance - program understanding. It works as a plug-in for one particular development environment and for one particular language. Finally, it fully relies on the developer to compose each concern description, whereas the proposed tool is more automatic and dynamic, relying on the developer in only a couple of instances.

Recently, a few studies are beginning to cover the problem of locating features in source code. Marcus and Rajlich [41], in their paper “Identification of Concepts, Features, and Concerns in Source Code”, blame the problem of not having one-to-one correspondence between features and modules on “limitations of existing programming paradigms and languages, often combined with the lack of design expertise, resulting in a sad reality where concerns are implemented in several modules, often cross-cutting the primary decomposition of the system” [41]. Concerns or features are often seen sharing the same module, making it hard to comprehend the program and perform proper impact analysis and regression tests.

Feature-based code analysis is recognized by many [40][63][23][19] as a good technique to aid developers in program understanding tasks. Wong and Gkhale [70], in their paper “Static and dynamic distance metrics for feature-based code analysis”, presented new metrics to determine the “distance” between two related features, based on their execution profiles. To illustrate the use of their metrics, the authors developed a tool “SHARPE”, which provides a measure of how two features are related. Such measure can “serve as a good start to

understanding how a modification made to one feature is likely to affect other features”.

Many feature-based tools produce call graphs and present them as visual aid to understanding the feature at hand. A paper that relies on call-graphs for feature location is by Bohnet and Dollner [7]. Their method combines both static and dynamic methods to identify and explore feature call graph. “An effective 2½D visualization provides various visual cues that facilitate finding those paths in the function call graph that are essential for understanding feature functionality”. Their approach is limited to C/C++ since most legacy systems are written in those languages.

TraceGraph 4 is another tool to assist engineers in locating and understanding the code for a specific feature. It was originally developed at the university of West Florida then later adopted by Motorola in 2007 to see if it can be effective in the company’s large software systems. Jiang and Zhang [28], in their case study “TraceGraph 4: A Demonstration Case Study”, claim that the tool was indeed useful in maintaining real legacy systems. Like other software

maintenance tools, this tool does not offer support to non-developers, and its user interface is not intuitive enough for novice developers.

Bohnet and Dollner [6] presented a prototype tool in their paper for locating feature code. In addition to call graphs, the authors use module containment and data modification to help users extract the functions with the highest relevancy. The tool offers “graph pruning” capability, which allows the developer to remove irrelevant functions from the call graph.

2.7 Software Complexity and Reliability Metrics

Several studies have been published, and hundreds of metrics were invented covering both product and process metrics, in almost all phases of software maintenance. However, McCabe’s Cyclomatic Complexity metric remains to be the most widely used. Cyclomatic complexity metric was introduced by Tomas McCabe in 1976 [42], and has been extended a couple of times since then to include design and structural complexity [43], and be independent of the language format [44]. It has been applied successfully in several areas of software engineering, some of which happen to be the main focus of this research, such as program comprehension, change impact analysis, and regression testing.

Shepperd [58] criticized cyclomatic complexity for being based on poor theoretical foundations. He also claimed it can be outperformed by a simple lines-of-code (LOC) metric. There are many negative criticisms of McCabe's measure, but it must be taken into consideration that it was the first software measure put forward over 30 years ago, before many advances in programming and complexity theory. Fenton and Pfleeger [18] point out that cyclomatic complexity metric is useful when counting independent paths but does not give an accurate picture of the total complexity.

Another complexity metric is Halstead, which gives a true size measure of each function in terms of operators and operands [21]. Halstead metrics have seen limited use. They are used instead by other composite metrics such as Maintainability Index. Marciniak [40] describes Halstead complexity measures, along with other commonly known related measures.

Maintainability Index (MI) is another complexity metric used for measuring program maintainability. Welker [65] offers a good explanation of the MI measurement technique, which takes several

factors into computing the index; such as: Cyclomatic complexity, Halstead Volume, count of lines of code, percent of lines of comments, etc. Oman [46]. MI has received good reviews and was chosen by Software Engineering Institute [60] as the most suitable tool for measuring the maintainability of systems with high-quality requirements. It's also used by this research as a basis for some of the proposed metrics.

Measuring software reliability is not as easy and remains a difficult problem because the nature of software is not well understood. Since software reliability cannot be measured directly, then it's typical to measure something related to it, such as complexity, faults, and test coverage. "The current practices of software reliability measurement can be divided into four categories" [47]: product metrics (e.g. lines of code, function point, and complexity), project management metrics, process metrics, and faults and failure metrics. The reliability metrics proposed in this research belong to the first category, and are based on function complexity, maintainability, and maturity. As the function complexity increases, its maintainability and reliability decrease. Function maturity (number of releases it has been in) has a strong

correlation with reliability as well: as the maturity increases, the reliability increases.

McCabe, Halstead, and MI metrics are not the only complexity measures available for use. Others include: LOC, Kafura Fan-in/Fan-out, Card and Glass System Complexity. These metrics have been used to estimate the complexity of the maintenance effort, and are commonly used in predicting reliability [1][22].

2.8 Other Cost Factors

The major factors contributing to the high maintenance cost were covered by many studies, as shown in the previous sections. Adding to the maintenance cost are several other less major factors, such as software aging, limited tools support, and inexperienced personnel. These cost factors will be discussed in this section.

Much of the software today is decades old and still aging. Maintenance of such software becomes more difficult year by year since software updates gradually destroy its original structure and increase its entropy [11]. The word “entropy” means the tendency of systems to destabilize and become more difficult to maintain over time. A side effect of continuous changes is that software documentation becomes stale,

and not trustworthy for future maintenance use. Adding to the difficulty is that certain modules of the code have high error densities called “error-prone modules”, and certain error fixes introduce other errors - “bad fix injection”. Limited staff inexperience and tool support add to the cost of repairing defects, identifying and removing of error-prone modules, and common re-factoring activities.

As a result of these factors, more personnel are needed to perform maintenance. Table 2.3 lists some interesting figures of personnel needed to perform maintenance vs. development work. The table shows that a few decades ago, this ratio was about 1-2 maintenance personal for every ten developers. Currently, as programs got larger, older, more complex, and in need for more maintenance, that ratio stands around three to one.

Table 2.3: U.S. Software Populations in Development and Maintenance [11]

Year	# Developers	# Maintainers	Total	% Maintainers
1950	1,000	100	1,100	9.09%
1955	2,500	250	2,750	9.09%
1960	20,000	2,000	22,000	9.09%
1965	50,000	10,000	60,000	16.67%

1970	125,000	25,000	150,000	16.67%
1975	350,000	75,000	425,000	17.65%
1980	600,000	300,000	900,000	33.33%
1985	750,000	500,000	1,250,000	40.00%
1990	900,000	800,000	1,700,000	47.06%
1995	1,000,000	1,100,000	2,100,000	52.38%
2000	750,000	2,000,000	2,750,000	72.73%
2005	775,000	2,500,000	3,275,000	76.34%

The table does not count adding major features as maintenance otherwise the gap will be much bigger. On the other hand, the table considers individual costs as equal. In reality, the people who do maintenance tend to be less paid than original code developers because they are either new on the job or less capable. Often, the current programmers are not the ones who invented the code and/or no longer familiar with it. This increases both the cost and the code's entropy. Making matters worse, employee turnover in the software industry is a major problem.

Limited tool support contributes to the cost, as well. There are presently much fewer tools for managing maintenance activities than development activities. Most of the tools discussed in the previous section are prototype tools and not ready for actual use on commercial

projects. Tool support is needed mostly in the areas of program comprehension and regression testing. More tools are needed in the areas of code impact analysis, restructuring, test coverage, complexity analysis, better defect tracking, reverse engineering, and reengineering. Even the high-level programming languages and debuggers in use today are not “high” enough to support true maintenance. Automation tools have a positive impact on software productivity and quality, and can greatly help manage software complexity.

2.9 Summary

Software maintenance is an old subject that covers a wide range of software engineering areas and major challenges, as outlined in this chapter. Many papers have been written and tools developed to tackle the described maintenance cost factors. However, the coverage of each paper or tool was found to be limited to one or two of these challenges, but not all. For example, there are excellent program comprehension solutions but they don't offer any services in areas of change impact analysis or regression testing. There are testing solutions that could be useful to testers but not useful to developers or managers.

Having seen all the related studies and tools in this chapter, it is obvious that these solutions are disconnected from each other, and more importantly, disconnected from the actual maintenance process model itself. There is a disparate need for one comprehensive software maintenance solution. A solution that addresses all the maintenance cost factors at once, and is easy to use by the entire maintenance team. Such unique solution must therefore include a process model and a tool. This is what this research is all about.

CHAPTER 3

METHODOLOGY

3.1 Introduction

The methodology that this research follows to prove its hypotheses is to introduce a new maintenance process model, five new reliability metrics, and a new maintenance tool to help the maintenance team adopt the new model and metrics. As part of the research, the CMMR tool was used on several case study projects, including a real commercial software project during the early phases of a given maintenance release cycle that purposely follows the proposed maintenance process model. Performance measurement (such as productivity, quality, and duration) were taken and compared with previous release cycles. Further refinements in the tool and the built-in metrics were made based on the results and feedback from the maintenance team. Reduction in maintenance cost was noted as a result of using this methodology and the contributions of this research. The reduction in cost was seen in two important aspects of software maintenance: higher productivity and efficiency (i.e. faster time to delivery), and less defects escaping to field (i.e. better release quality).

The next few sub-sections explain each of the three contributions in some detail.

3.2 Proposed Process Model

The current maintenance process models, explained in Section 1.4.2, are obviously not working effectively, as indicated by the ever-increasing cost of software maintenance throughout the software industry. Either, the existing process models are not being fully adopted, or the problems are inherent in the models themselves. The author of this research believes it's a combination of both. In general, when it comes to adopting a new process model that requires a transition from traditional methods, most people resist the transition due to two reasons: fear of change of what the new process holds, and fear of loss of the investment in their traditional processes. The key elements to a successful transition to a new process model are: low setup cost, tool automation, and a gradual unforced transition.

The software maintenance process model proposed in this research, as shown in Fig. 3.1, is based on the IEEE-1219 model introduced in Section 1.4.2 (see Fig. 1.1). The proposed model includes more details and assumes the presence of CMMR - the maintenance tool that is

developed as part of this research. The maintenance activities: program comprehension, change impact analysis, regression testing, documentation update, and maintenance management are shown as distinct phases, and shaded to indicate the use of CMMR tool during that phase. The process model is a continuous loop of carrying out maintenance tasks until the manager determines that the product is feature-complete and ready for delivery. Unlike the IEEE model, the delivery phase comes after all the release requirements are fulfilled. In addition, each activity has the title of the person responsible for carrying out the activity. For example, determining the next task and deciding when to deliver are areas that fall under the manager's responsibility, while design and implementation are the developer's responsibility, etc.

When carrying out a shaded activity, it is expected that CMMR be running side-by-side next to the developer's development environment, the manager's project tracking tool, the tester's defect tracking tool, and the writer's documentation tool. Not all team members have to adopt this process model and tool to see the benefits. But to realize the full benefits of this and maximize the cost reduction,

it is recommended that all the team members fully adopt the CMMR tool and follow the proposed process model.

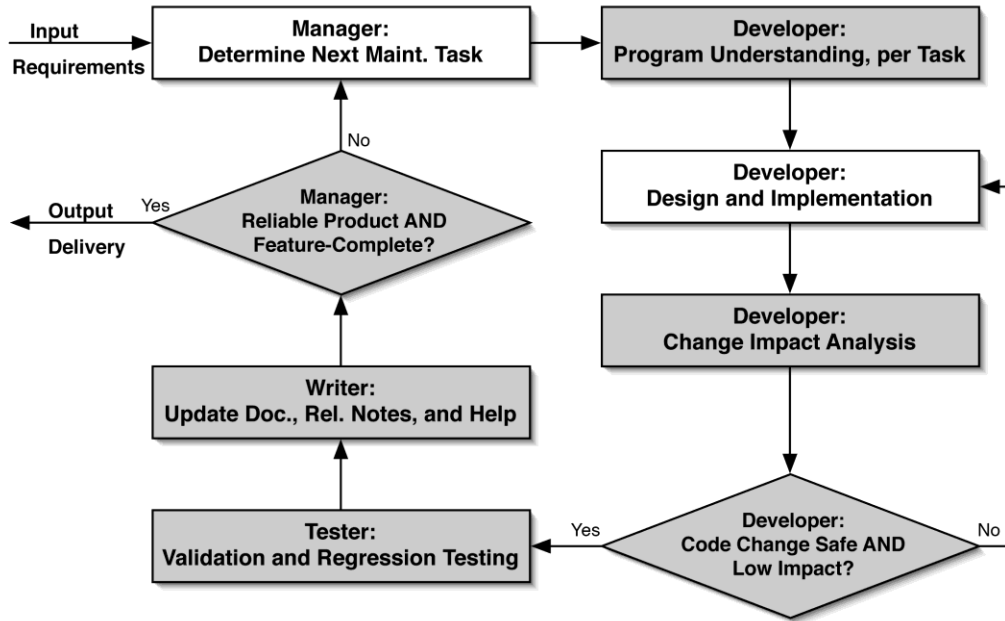


Fig. 3.1: Proposed Maintenance Process Model

The right-hand side of the diagram is for developers only. After understanding the program as related to the task at hand, and designing and implementing the solution, the third activity (change impact analysis) may find problems suggesting going back to get better understanding of the program as related to the task at hand, or its solution design and implementation. It's possible for some tasks to predict the change impact before the change is made.

When the results of the change impact analysis phase are positive, the task moves onto testers, to revalidate the change and perform regression testing, then to documentation writers, to update any related documentation. Regression testing under this model is feature-based, rather than module or function based. It is also more focused on the exact features that have been affected by a particular change, thus limiting the testing to only those areas. Documentation updates are performed on time and with better accuracy. More efficient management is made possible by various reliability measures that the manager can obtain directly via CMMR. The full benefits of the model will be illustrated using the tool to be discussed in more details in Section 3.4 - CMMR Tool.

3.3 Proposed Metrics

There are many known metrics that can be used during the various phases of software maintenance to measure reliability; such as product size (e.g. LOC, function points), function cyclomatic complexity, maintenance index, estimated vs. actual durations, number of defects found/fixed, fix backlog, fix response time, and fix quality. In practice, a small percentage of these metrics are actually being used. "Most technologies developed by the software community have not been

transferred into industrial use, and the number of papers on the software process modeling and technology presented at conferences and published in journals is decreasing” [19]. There are many reasons for this including: limited practicality in the processes and metrics themselves, people’s tendency to fear data that can be used to measure their performance, an bad use of good metrics which leads to bad management of software maintenance projects.

This research introduces five new metrics that are easy to compute and use, have low initial investment cost, yet, if applied correctly, they promise great reduction in long-term maintenance cost. The five new metrics are described next.

3.3.1 Feature-based Function Maintainability (FBFM)

It’s well known in the software engineering industry that the higher the function’s complexity, the more maintenance it will likely need [60]. The extra maintenance cost comes from additional testing effort, more time to comprehend the code, more risk in modifying it, and more errors the modification will leave behind. A very high complexity value may indicate a potential need to rewrite the function entirely rather than making small modifications to it. Rewriting the function may involve

reducing its nodes and paths, or breaking it into smaller manageable pieces (sub-functions).

This research claims that a function that is shared by multiple product features (or user scenarios) is more complex by design, requires more maintenance than its complexity value suggests, and is likely to introduce more errors than a single-feature function with the same complexity measurement. Sharing functions across features is a good software technique because it reduces the overall size of the software system and the maintenance cost. However, a shared function must be constantly maintained to insure it always works for every feature that uses the function, thus increasing its maintenance cost. The new metric, FBFM, introduced here and shown in Equation (3.1), is based on the Maintainability Index (MI) of the function adjusted for the number of features that use the function. It is therefore believed to be a better indicator than MI at measuring the function complexity and estimating its maintenance cost.

$$FBFM = \max\left(0, \frac{MI}{171} - \text{Log}_{10}(N + 9) - 1\right) \quad (3.1)$$

Where: *FBFM*: *Feature-Based Function Maintainability*

MI: *Function Maintenance Index*

N: Number of Features that use the function

The MI range is typically 0-171, whereas the range of FBFM is between 0 and 1. The higher the FBFM value, the better the maintainability of the function. The MI value in Equation (3.1) is divided by 171 to normalize it within the 0-1 range. The normalized MI value is then reduced by a logarithmic value of the actual number of features using the function. For unused functions, no computation of FBFM will be triggered, as only accessible functions get assigned FBFM values. For single-feature functions, FBFM equals MI, since the reduction is 0 ($\log 10 = 1$). If the value of N is higher or equal to 91 the reduction of the MI value takes it below zero, thus the use of the max function to keep the final FBFM value at the minimum zero level. A zero FBFM value indicates a very low maintainability value and a high potential for: errors, change impact, regression test, and maintenance cost.

The aim, of course, is to maximize the FBFM value for every function in the software project, which improves their reliability automatically, as well as the reliability of the features that use these functions, and the product reliability as a whole. Generally, shared functions tend to

be relatively small in size (i.e. low complexity value). One should avoid sharing a long function that has a high complexity value, as that increases its FBFM value and maintenance cost. During debugging, identifying shared functions is the first place to look for root causes of multiple feature failures.

3.3.2 Function Maturity (FM)

It is well known that the higher the complexity of a function, the higher the potential for having defects inside that function. While this metric is true, it ignores a very important aspect of software development, and that is: function maturity. A “mature” function that has undergone a lot of testing and been included in several releases will likely have a lower number of defects (i.e. higher reliability) than a brand new function of equal complexity.

The new metric “function maturity”, introduced here and shown in Equation (3.2), takes into account the maturity of the function with respect to the product’s maturity as a whole. Maturity is a new measure used in this context to indicate the number of times the function, or product, has been released to customers, and the age in days since creation date.

$$FM = \frac{C * R_f + A_f}{C * R_p + A_p} \quad (3.2)$$

Where: *FM: Function Maturity*

C: Average Release Cycle in days

R_f: Number of times the function has been released to customers.

A_f: Function's Age in days.

R_p: Number of times the product has been released to customers.

A_p: Product's Age in days.

A few notes on the terms in Equation (3.2):

- R_p is always a positive integer. If the product has not been released to customers, then it has not entered maintenance mode, and this metric (and most of this research) won't apply.
- R_f can be zero if the function was created after the last release. If a function is ported from another product, its R and A values start out as zeros.
- A_f indicates time in days since the function's creation date. The same for the product A_p value.
- Multiplying R by C gives it extra weight (age). This is saying that each day of a function's life prior to a release equals two days in age without release.
- R_f is always <= R_p, while A_f is always <= A_p.

The computed FM value will always fall between 0 and 1, with 0 indicating minimum maturity and 1 indicating maximum. Obviously, the higher the FM value, the more reliable the function is, and the less effort and cost needed to maintain it. This is a direct result of having tested the function heavily in prior releases. When a function is rewritten, or receives major changes, it is recommended to reset its creation date (i.e. $A_f = 0$), as if it were a brand new function, resulting in a new and lower FM value. Minor changes generally improve the function's maturity (i.e. defect repairs). Some defect repairs actually inject errors, and in reality, they interrupt maturity and decrease reliability. However, the percentage of bad repairs is about 7%, according to [12]. So, they are not captured by this metric and are considered as measurement error (i.e. noise).

When computing function maturity, the CMMR tool relies on date information derived from the CMMR project window, and the header comments in the source code specifying the creation date of each function; i.e. "yyyy/mm/dd". Any function missing a creation date comment will be assumed to be as old as the project. Bad or missing creation date entries could yield wrong computation of the function

maturity (FM) metric, and misleading reliability computation results.

3.3.3 Function Reliability (FR)

Feature-based Function Maintainability (FBFM) and Function Maturity (FM) metrics are combined into a new metric “Function Reliability”, as shown in Equation (3.3), which realistically computes the function reliability.

$$FR = \frac{FBFM + FM}{2} \quad (3.3)$$

Where: *FR*: *Function Reliability*

FBFM: *Feature-Based Function Maintainability*

FM: *Function Maturity*

Typically, in software development, the reliability of a function is directly related to the complexity and maintainability of the function [60]. But, as explained earlier, FM does matter in software maintenance and impacts the reliability of the function just as effectively as the function’s complexity, thus the average of the two in Equation (3.3).

Reliability here is therefore not a probability of failure, as it is commonly known (see Sections 1.4.8 and 2.7). It’s simply a number between 0 and 1 (higher is better), which represents the combined maturity and

maintainability of the source code. The significance of the FR value is not in the number itself as the value may be subjective (i.e. 0.7 FR may be acceptable in some environments but not in others). The significance comes to play when taking several measurements over time, plotting these values, and making sure the trend is going in the right direction and at the right speed. As a function grows in complexity and/or maturity, its reliability should be recomputed. This is done typically after each release or when the function changes. When these computed FR values are plotted over time, they follow a well-known exponential model (as shown in Fig. 3.2). The model shown in the figure is a special case of the Weibull distribution family, and is used widely for reliability growth studies in many fields.

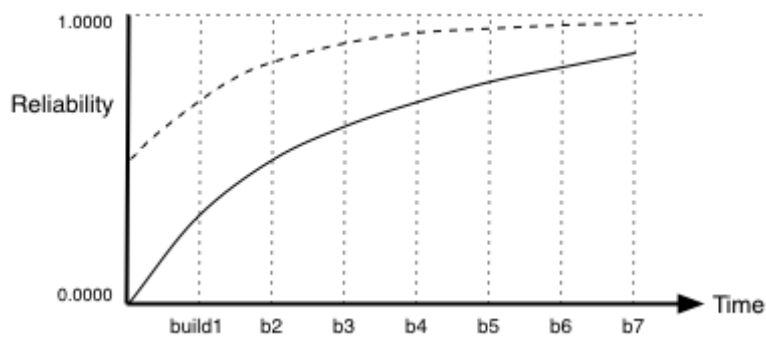


Fig. 3.2: Function Reliability Model

The starting point of the exponential curve in Figure 3.2 depends on whether the function is brand new or has been through one or many releases. The reliability of a brand new function follows a curve starting out at a value close to zero (due to zero maturity), and slopes up as the function reliability increases to a maximum of 1 without reaching 1. See the solid line in Figure 3.2, which represents a very complex new function. The reliability of an existing (mature) function, on the other hand, follows a similar pattern but the curve typically starts out at a point higher than zero. See the dashed line in Fig. 3.2. The units on the horizontal axis represent internal releases of the software system that are built for testing purposes and usually after some code changes are made.

Reliable functions don't typically change and thus have a constant complexity/maintainability value, however, they gradually increase in maturity, as they get included in product releases, so their reliability increases over time. As a function is changed to fix a defect or add a new enhancement, its complexity, maintainability, and reliability change as well, hopefully for the better. Any fluctuation in reliability results in a curve that may not be as nice and smooth as the ones

shown in Fig. 3.2. In making changes to a function, the developer must keep an eye on the FR values over time, and try to keep the reliability from degrading as much as possible.

3.3.4 Feature Reliability

A software feature is implemented as a sequence of functions and/or methods. The reliability of the feature is therefore dependent on the reliability of its underlying functions. A new reliability metric “feature reliability” is introduced here and shown in Equation (3.4). The metric is calculated by taking the average of all the reliability metrics of the underlying functions. This metric is intended for software managers and decision makers.

$$R_{feature} = \frac{\sum_{i=1}^n FR_i}{n} \quad (3.4)$$

Where $R_{feature}$: *Feature Reliability*

FR_i : *Function i's Reliability*

n : *Number of functions in feature*

Fig. 3.3 shows the reliability model of a feature (X) which consists of three functions (A, B, and C). Function A is a new and complex function so its reliability curve starts from zero, indicating high complexity and

lack of maturity, and slopes up with maturity and reduction in complexity. Function B is an existing function with moderate start value. Its reliability curve increases over time as well. Function C had a modification made at t_1 where the complexity increased dramatically dropping the reliability curve sharply. It took the function a few builds to recover its old reliability and glory.

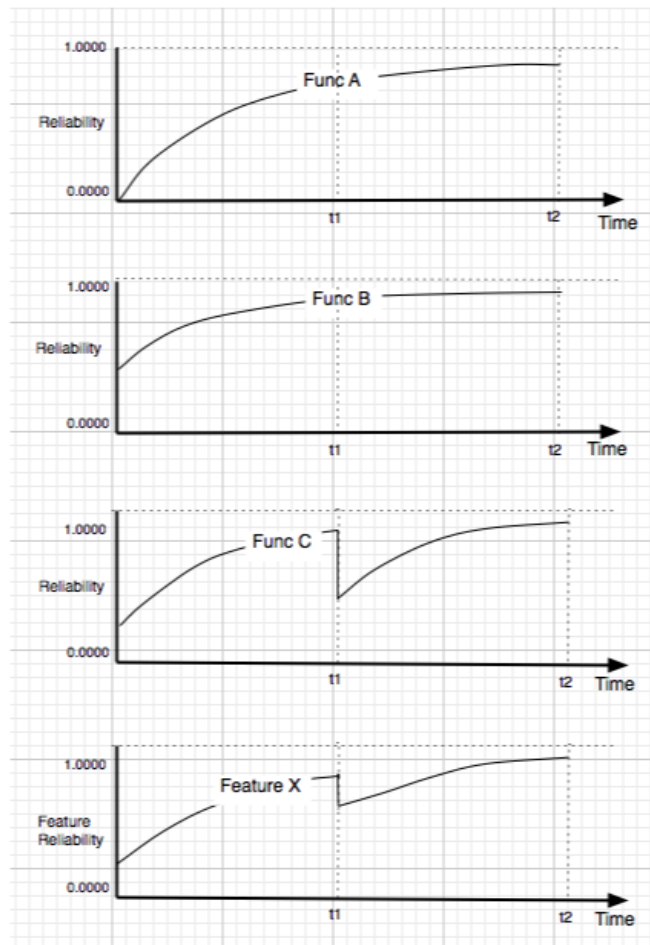


Fig. 3.3: Feature Reliability Model

The feature reliability curve is computed from the three function reliability values on build-by-build basis. Notice that, at any point in time, the feature is as reliable as the average reliable function. Of course, in real applications, features are typically composed of tens or hundreds of functions. Nevertheless, the same trend computation logic just is still applicable and used. The trend charts generated and presented by CMMR differ slightly from the one shown here. A future release of the tool will combine the curves onto one chart as shown in Fig. 3.3.

3.3.5 Product Reliability

A software product consists of one or more features. The reliability of the product is dependent on the average reliability of its features, assuming equal weights in terms of being critical. The new metric, as shown in Equation (3.5), can be used by management in deciding when to release a product, what features to release, or drop, etc.

$$R_{product} = \frac{\sum_{i=1}^n R_{feature_i}}{n} \quad (3.5)$$

Where $R_{product}$: Product Reliability

$R_{feature_i}$ Feature i Reliability

n : Number of features in product

Fig. 3.4 shows an example product made up of two features. Notice that at any point in time that the product reliability is equal to the average reliability of its two features.

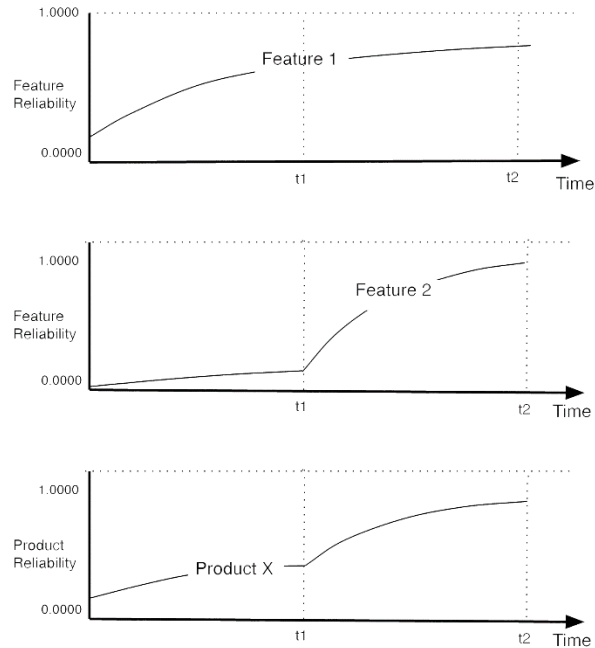


Fig. 3.4: Product Reliability Model

3.4 CMMR Tool

The CMMR tool creates and uses a database of features and functions in the software project, and tracks these relations on build-by-build basis. The tool reads the source code to parse for tokens (branches, operations, operators, LOC, comments etc.) and stores that data inside the database. It has occasional but controlled write access to the source code to insert function names used for profiling and trace

generation. During the execution of each individual feature the names of all visited functions are written to a log file. At the completion of the run, the tool takes the content of the log file and maps it to a tree representation for further use.

1. CMMR Architecture

In terms of architecture, CMMR is essentially made up of two major components: a parser and a viewer. A simple illustration of the two components and their interactions with the project and the database is shown in Fig. 3.5. More details on each component will follow in the next two sub-sections.

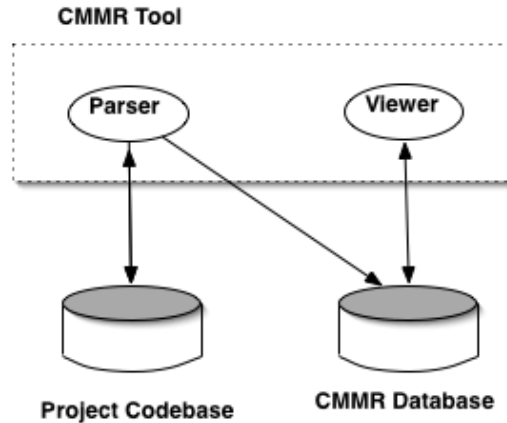


Fig. 3.5: CMMR Architecture

2. CMMR User Interface

In terms of user interface, CMMR is designed to be extremely easy to use by all its users, novice and experts alike. The tool's menu bar and menu commands are shown in Fig. 3.6.

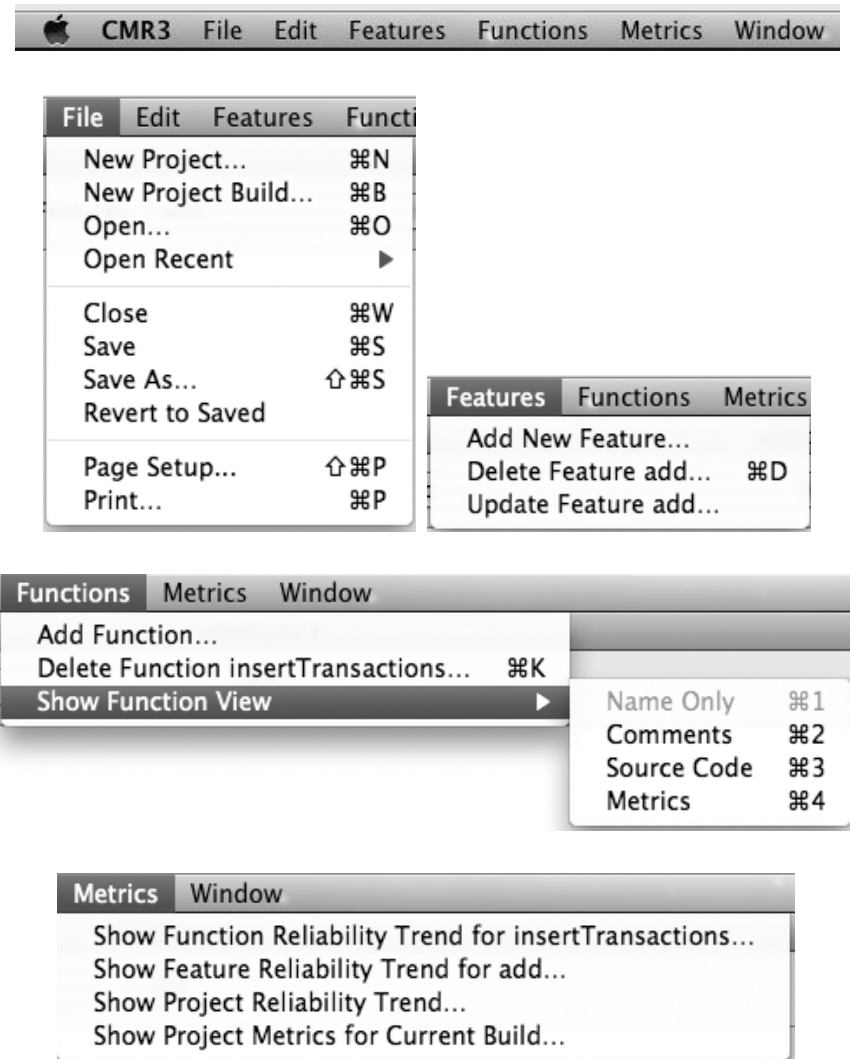


Fig. 3.6: CMMR User Interface

Project creation and build management are handled in the File menu. As shown, the feature management functionality is handled in the Features menu, while function management is in the Functions menu. All the metric-related commands are in the Metrics menu. CMMR can simultaneously handle multiple project document windows, or multiple build windows of the same project. The Window menu manages switching between different open windows.

3. Adding Features

Initially, the product features are entered into the database via Add New Feature command. When adding a feature, the user will have to provide the tool with the feature name (see Add Feature Dialog in Fig. 3.7) and perform the feature inside the running project. When the feature is executed to completion, the user would click Add Feature in CMMR. This prompts the tool to take the entire trace log of visited function names and relate them to the new feature in the database for later viewing.

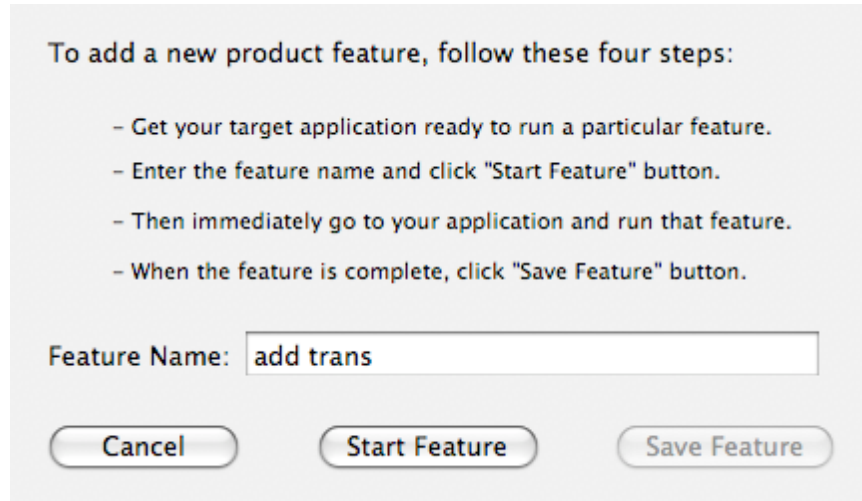


Fig. 3.7: CMMR Add Feature Dialog

This process is repeated until all the features are entered. Nodes and edges can be moved around for better layout. Additional nodes can be added manually (via Add Function Node) or removed (via Delete Function Node). After each build and whenever major changes are made to a certain feature, the database representation of the feature can be updated to reflect new code paths and/or new complexity metrics. Multiple metric measurements, taken at each build, help CMMR build its reliability curves for each function and feature (via Metrics menu commands).

The next two sections explain the two major components of CMMR, the parser and the viewer, in more detail.

3.4.1 CMMR Parser

To generate the database relations (features and functions), a trace log and source code parsers are needed and implemented as part of the tool. The log parser handles the parsing of the trace data and mapping it into a graphical representation. The source code parser handles parsing of the source code. It assumes C, C++, and Java function/method naming conventions and comments. The parsing is performed after each build in order to keep the information in the database up-to-date. The next few sections explain the major functionality of the CMMR parser.

1. Log Data to Graph Data Mapping

When the project is run in debug mode, all the visited functions are dumped to a log file for further analysis. Here is an example trace file with CMMR-specific profile data in it:

*CMMR Feature (compute reliability) Run Started: 2008-06-24
23:29:33 +0300*

*CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:71
CMMRFuncStart:computeFunctionReliability*

*CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:438
CMMRFuncStart:featuresThatLeadToNodesWithSameName
CMMRPathName:/Users/aqaisi/Desktop/CMMR*

*proj/MetricsComputer.mm CMMRLineNum:451
CMMRFuncEnd:featuresThatLeadToNodesWithSameName*

*CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:460
CMMRFuncStart:McCabeMetricsFromCodeAnalysis
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:726
CMMRFuncStart:stringWithCharString
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:737
CMMRFuncEnd:stringWithCharString*

...

*CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:502
CMMRFuncEnd:McCabeMetricsFromCodeAnalysis*

*CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:510
CMMRFuncStart:HalsteadMetricsFromCodeAnalysis
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:629
CMMRFuncStart:numOfOperatorsInCode
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:726
CMMRFuncStart:stringWithCharString
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:737
CMMRFuncEnd:stringWithCharString*

...

*CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:656
CMMRFuncEnd:numOfOperatorsInCode
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:665
CMMRFuncStart:numOfOperandsInCode
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:726
CMMRFuncStart:stringWithCharString*

```

CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:737
CMMRFuncEnd:stringWithCharString
...
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:717
CMMRFuncEnd:numOfOperandsInCode
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:533
CMMRFuncEnd:HalsteadMetricsFromCodeAnalysis

CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:542
CMMRFuncStart:MaintanabilityIndexFromCodeAnalysis
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:570
CMMRFuncEnd:MaintanabilityIndexFromCodeAnalysis

CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:578
CMMRFuncStart:KafuraMetricsFromCodeAnalysis
CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:600
CMMRFuncEnd:KafuraMetricsFromCodeAnalysis

CMMRPathName:/Users/aqaisi/Desktop/CMMR
proj/MetricsComputer.mm CMMRLineNum:173
CMMRFuncEnd:computeFunctionReliability

```

The above log data is an actual profile of running a feature inside CMMR itself – the computation of the function reliability metric (FR). In other words, CMMR is analyzing itself while computing FR. The log shows the function “ComputeFunctionReliability” at the top, with its full

file path name and its starting line number inside the file. The next line is the first function it calls “*featuresThatLeadToNodesWithSameName*” which too gives its file path and line number. The next line is “*McCabeMetricsFromCodeAnalysis*” which does the same but this function calls another function of its own “*stringWithCharString*” before returning. The remaining functions follow in the same way until the last line of the trace is reached, which shows the main function “*ComputeFunctionReliability*” returning.

The tree generation is essentially taking the above log file and reading it one line at a time, and for every function start “CMMRFuncStart” a new node is created. Then in looking for the end of the function “CMMRFuncEnd”, for any new function encountered, a new node is generated and attached to the parent node as a child node. This process continues recursively until we return from the main function.

The above sequence of trace statements actually contains duplicate entries (see the two “...” lines), which are purged to reduce the complexity of the tree. For example, the utility function “*stringWithCharString*” is found many times in the profile, in fact, as

many times as there are tokens in the source file. There is a loss of information in doing that, of course, but it's a tradeoff with complexity. A future update of CMMR will save this information inside the node, as it may be taken into account in metric computation.

When the entire log file is scanned, the full feature call graph is completed, and the tree is automatically shown to the user, as in Fig. 3.8. The graph tree shown is read from left to right and in depth-first order. The figure is an exact representation of the execution profile (the call sequence) of the "Compute Function Reliability" feature. A quick glance at the graph gives a good overview of the feature. If more details are needed about the feature or a particular function within the graph, whether it's source code, comments, or metrics, it's only a click away for the user, as will be shown in the next section – CMMR Viewer.

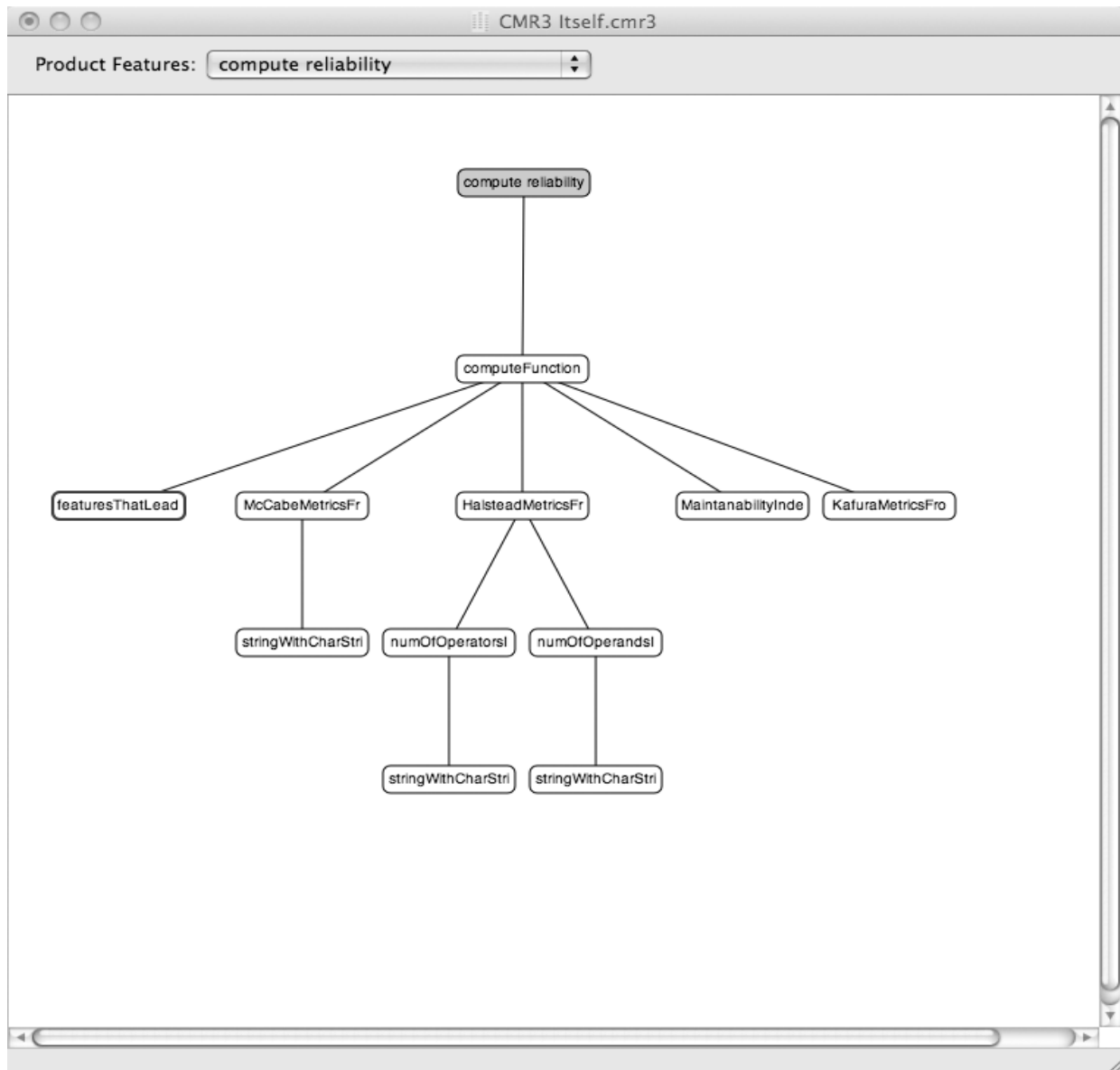


Figure 3.8: Compute Function Reliability Feature Tree Window

2. Source Code Extraction

In the process of mapping the trace statements to graphical nodes, the CMMR parser accesses the files containing all functions in the log data, extracts the source code, and stores it in the database. Among

the information stored for each function: source code lines, comment lines, starting and ending line numbers, creation date of the function (if available inside the comments), modification date (if this is a new build and the function had changed), and other information that is needed in metrics computations when requested later on. Appendix C at the end of this dissertation shows an actual source code listing of the parsing functionality.

3. Node Storage

In addition to code-related information (see previous section) the node also stores graphical information related to on-screen drawing, such as node name, location on screen, and tree representation, i.e. the parent node (incoming edge) and the children nodes (outgoing edges). When metrics are computed for a particular node, the results are also cached inside the node for quick access. The parser currently computes and stores the following metrics: McCabe Cyclomatic Complexity, five Halstead metrics, Maintainability Index, Kafura System Complexity, and the three function metrics proposed in this research: FBFM, FM, and FR.

4. Multiple Feature Management

More features can be added in the same as explained above. All the different features are grouped together in a popup menu (as shown at the top of the window in Fig. 3.8). In some cases, multiple variations of the same feature are added. For example, if the code path when saving a file on a local disk differs significantly from saving it on a remote disk, then perhaps the user can run both variations of the feature and name them “Save File” and “Save File On Remote Disk”. All the added features, their graphical representations, the functions source code and comments, and the metric computations are saved inside the CMMR database, which is simply the CMMR project document.

5. Multiple Build Management

When multiple builds are created, the entire database of features and functions is updated. This is done by taking each function of each feature and determining if the function’s code had changed since the previous build. The code stored inside each node is automatically compared with the current build’s source code, and if changes are found, the node is marked in red. If the function cannot be located, a comment is inserted in its node to remind the user to update the

feature. Updating a feature, which is available as a command in the Features menu, essentially re-runs the feature and allows for finding the function in its new location.

The information from each build (feature trees, metric computations, etc.) is stored on disk inside a separate version of the same document. For example, five builds of the target project will have five CMMR documents stored on disk. This arrangement allows CMMR to generate time-based reliability trend charts (see Section 3.4.3). The points in these charts are actually taken from these documents starting from the current build all the way back to the initial build.

3.4.2 CMMR Viewer

Graphical views are generated based on the function-feature relations, allowing users to better view the project structure at four different levels of details: Tree, Comment, Code, and Metric views. The user can switch between the four views via Show Function View commands, or by using a keyboard modifier key when clicking a function node. Control-click shows the function's Source Code window. Option-click shows the Metrics window. Command-click shows the function's Comments window. A double-click on a function node, opens the file

where the function is implemented inside the user's favorite editor.

The following is a brief description of each view.

1. Tree View

The default view (as shown in Fig. 3.9) is intended for the entire team and as a starting point for many tasks, such as debugging, testing, and documentation. This view shows a tree graph of a particular product feature with its underlying functions in the code path. A red function node indicates the function had changed in the current build. A green oval in the upper corner of the function node indicates the function is shared by a multiple number of features, and the actual number is shown inside the oval. Shared functions require extra attention during regression testing and error removal.

on screen)

2. Comment View

This view provides more information about each feature and function. A click on any function node shows the function's comments within and above the function (header comments). An example view is shown in Fig. 3.10. The window is a floating transparent window with its title set to the selected node's function name, and the content set to its comments - with the comment markers removed. The transparency allows data underneath it (nodes and edges) to be shown.

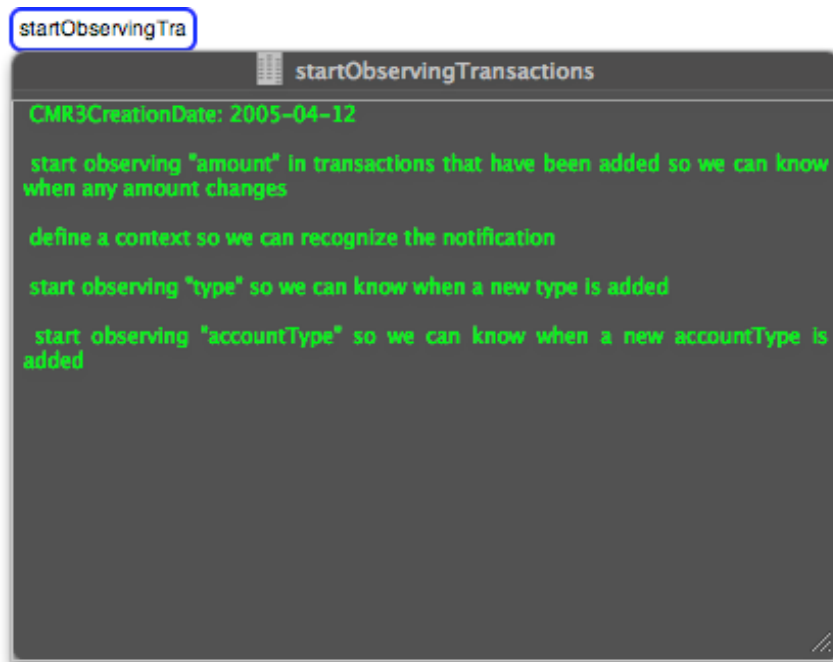
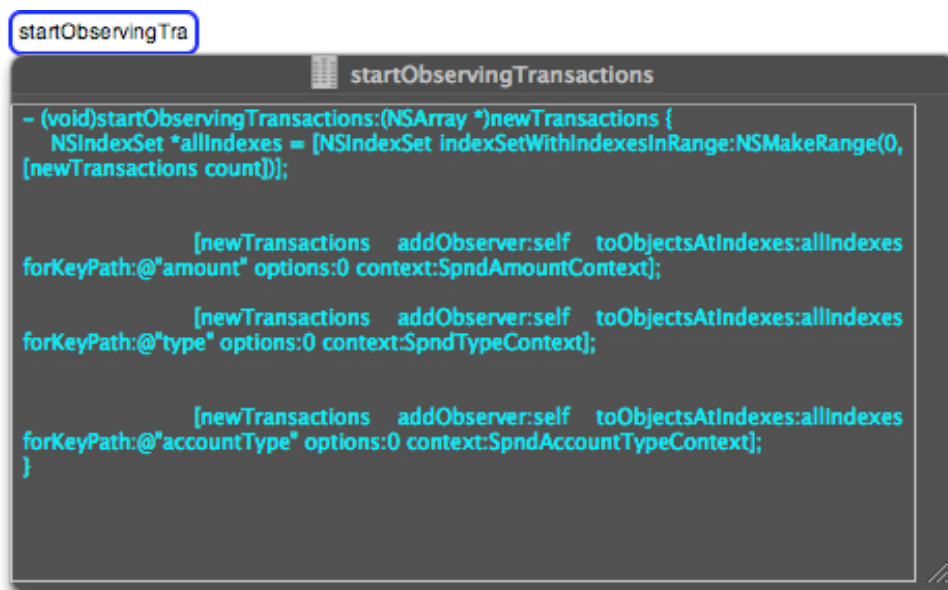


Fig. 3.10: Comment View of a function (on-screen text color is green).

These comments are extracted from the function's source code directly, and are updated on build-by-build basis. The information provided in this view is mostly intended to writers and testers who need an idea of how a feature works without actually seeing the underlying code or talking to the developers. Comments written in English are easier to understand than high-level language. Developers will have to be aware of this and are advised to thoroughly document their code and use descriptive names for functions/methods.

3. Code View

This view is intended for developers and technical testers and managers. This view shows the actual source code of the selected function in a special window (see Fig. 3.11).



```
startObservingTra
startObservingTransactions
- (void)startObservingTransactions:(NSArray *)newTransactions {
    NSMutableIndexSet *allIndexes = [NSMutableIndexSet indexSetWithIndexesInRange:NSMakeRange(0,
[newTransactions count])];

    [newTransactions addObserver:self toObjectsAtIndexes:allIndexes
forKeyPath:@"amount" options:0 context:SpndAmountContext];

    [newTransactions addObserver:self toObjectsAtIndexes:allIndexes
forKeyPath:@"type" options:0 context:SpndTypeContext];

    [newTransactions addObserver:self toObjectsAtIndexes:allIndexes
forKeyPath:@"accountType" options:0 context:SpndAccountTypeContext];
}
```

Fig. 3.11: Code View of a Function (on-screen text color is blue).

The user can only view the function's source code in this window. Editing requires the use of the development environment. Double clicking the node is a short cut to open the original file where the function is implemented inside the user's development environment or favorite text editor. The user is able to obtain full information on how to analyze, modify, and test each function, with full risk analysis in mind.

4. Metric View

This view is intended for managers, technical testers, and developers. The view shows several known complexity metrics, new metrics introduced in this research, and the factors used to compute these metrics, such as: number of features using the function, releases, LOC, comments, and aging information. See an example Metric view in Fig. 3.12. The on-screen text color is white.

Metric	Value	Description
Creation Date:	2005-04-12	—
Last Modified:	2005-04-12	—
Number of Features:	3	add trans
Number of Releases:	5	High - Excellent
Lines of Code (LOC):	51	Too Long
Lines of Comments:	30	Well Commented
McCabe Cyclomatic Complexity V(G):	18	Moderate
Halstead Vocabulary (n):	64	High #Unique Operations
Halstead Length (N):	163	Very High #Operations
Halstead Volume (V):	978	Very High #Operations
Maintainability Index (MI):	61.30278	Moderate Maintainability
Kafura Structure Complexity (Cp):	256	High Fan-in/Fan-out
Card & Glass System Complexity (C):	274	High
Feature-Based Maintainability (FBM):	0.3868819	Moderate
Maturity:	0.9548301	High - Very Good
Reliability:	0.670856	Moderate

Fig. 3.12: Metric View of a function

The Metric window shows 16 metric lines that are computed by CMMR for the selected function. Each line shows the metric name, the current measurement value, and a description line providing an intelligent assessment of the measurement or additional information. Tables 3.1 through 3.13 show how the description lines in the Metric window were derived from the computation results.

Table 3.1: Number of Releases Description Line

Number of Releases	Description Line
5 or higher	High – Excellent
3 - 4	Moderate
0 – 2	Low

Table 3.2: Lines of Code Description Line

LOC	Description Line
Higher than 40	Too Long
21 – 40	Long
0 - 20	Short - Excellent

Table 3.3: Lines of Comments Description Line

Lines of Comments / LOC Ratio	Description Line
Higher than 0.3 (i.e. 3 lines per 10 LOC)	Well Commented
0.1 – 0.3	Somewhat Commented
Less Than 0.1	No Comments!

Table 3.4: McCabe Cyclomatic Complexity Description Line

McCabe Cyclomatic Complexity*	Description Line
25 or higher	Very High
20 – 24	High
15 – 19	Moderate
0 – 14	Low – Excellent*

* The 0-14 range was adjusted from the original as suggested by McCabe (0-10), as the original range values were a little aggressive. Other range values were not given by McCabe so they were invented here as an educated guess.

Table 3.5: Halstead Vocabulary Description Line - (*# unique operators and operands*)

Halstead Vocabulary	Description Line
10 or higher	High # of Unique Ops
5 – 9	Moderate # of Unique Ops
0 – 4	Low # of Unique Ops - Excellent

Table 3.6: Halstead Length Description Line - (*total number of operators and operands*)

Halstead Length	Description Line
25 or higher	Very High # of Ops
20 – 24	High # of Ops
15 – 19	Moderate # of Ops
0 – 14	Low # of Ops - Excellent

Table 3.7: Halstead Volume Description Line - (*Length times Log 2 Vocabulary*)

Halstead Volume	Description Line
25 or higher	Very High # of Ops
20 – 24	High # of Ops
15 – 19	Moderate # of Ops
0 – 14	Low # of Ops - Excellent

Table 3.8: Maintenance Index Description Line

MI*	Description Line
50 or higher	Good Maintainability
35 – 49	Moderate Maintainability
0 – 34	Low Maintainability

* Initial values were very aggressive: 20 or higher = good maintainability; 10-20 = moderate maintainability, and 0-10 = low

maintainability. Range values were adjusted accordingly as shown.

Table 3.9: Kafura Description Line - $(Fan-in \times Fan-out) Squared$

Kafura	Description Line
100 or higher	High Fan-in/Fan-out
50 – 99	Moderate Fan-in/Fan-out
0 – 49	Low Fan-in/Fan-out

Table 3.10: System Complexity Description Line - $(Kafura + McCabe)$

System Complexity	Description Line
110 or higher	High
55 – 109	Moderate
0 – 54	Low - Excellent

Table 3.11: Feature-Based Function Maintainability Description Line
 $(\max(0, MI/171) - \log_{10} (\#features+9)-1)$

FBFM	Description Line
0.8 or higher	Very High - Excellent
0.6 – 0.799	High
0.3 – 0.599	Moderate
0 – 0.299	Low

Table 3.12: Function Maturity Description Line - (*# releases / # product releases*)

FM	Description Line
0.7 or higher	High – Very Good
0.3 – 0.699	Moderate
0 – 0.299	Low

Table 3.13: Function Reliability Description Line - (*Average of FM and FBFM*)

Function Reliability	Description Line
0.8 or higher	High – Very Good
0.6 – 0.799	Moderate
0 – 0.599	Low

To compute these metrics, CMMR starts out by parsing the code and extracting the lines of code (LOC) of each function, without comments and blank lines, then applying the various algorithms on the code. Surprisingly, no code was readily available on the Internet for common metrics like McCabe and Halstead; so all computations were reinvented inside the tool. The full listing of source code used by CMMR to compute the metrics is shown in appendix C at the end of this dissertation. It's supplied as is to help other researchers continue this research and/or use it in other code complexity computations.

5. Editor View

CMMR is not intended for editing and compiling source code. However, it does provide short cuts for doing so by opening any function inside the development environment by simply double clicking the function's node in the feature tree graph.

3.4.3 Use of CMMR Tool

The following section discusses how the CMMR tool is used and how it helps its users improve their productivity and the product quality. It is organized into four sections according to the type of intended users.

1. Developer's Use

In code maintenance, the developer is mostly dealing with customer feature requirements, or handling errors encountered by customers and testers when running a particular feature. When adding a new feature into a project or modifying an existing one to meet new requirements, it helps to use this tool as a starting point to get a better understanding of the project structure and the services and functions available. Visual representation of code is a much better alternative to program understanding than textual views.

When fixing bugs, developers typically follow a sequence of actions that start with bug analysis to find the root cause, determining a fix,

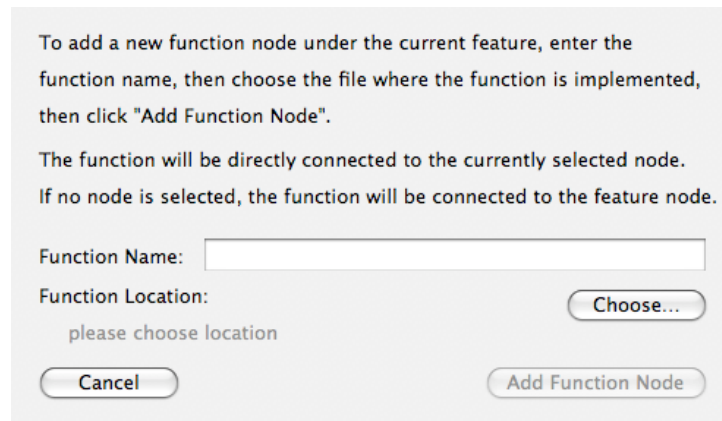
inserting the fix in the proper place (which may or may not be in the same place as the root cause), then verifying that the error is indeed fixed and that no new errors were introduced with the fix. Obviously, the developer can benefit from the tool and its multiple views in performing each of these steps. For example, finding the root cause of an error can be done initially by viewing the feature's code path in Tree View, then in more details, as needed, in the most suspect areas in the error code path. Most suspect area can be determined manually with the developer debugging various functions in the code path and tracking related variable names to see the starting point of where things go bad. The operation continues until the exact cause is determined.

After a fix is found and inserted in the correct place(s) in the code path, the developer verifies that the error is indeed fixed by running a few test scenarios. A very important step follows in insuring that all other features that use the effected code path are still functional and that no new errors were injected. Many errors are introduced in this step, because the developer is not aware of the full impact of the change just introduced, and as a result, does a partial job in verifying the fix.

This is where the tool is really handy because it allows the developer to “see the entire picture” of each change made or about to be made. Change impact analysis is assisted by actually seeing the exact set of functions that come after the selected function in the execution of the feature. And if the selected node shows multiple features using the function, the names of the features are available, further assisting the developer in the analysis of the change impact. This analysis can be performed immediately after the change is made. This way, any new possible errors or broken features are detected and fixed right away. If injected errors are found and fixed much later, they will increase maintenance cost. If the injected errors slip into the field and found by the customer, then the maintenance cost will even be higher. At any point in this process, the developer can always go back to previous phases of the process model to design a better solution.

An important role for the developer is to assist the tool in operations that require developer knowledge and cannot be automated. For example, a function may impact a feature without actually being called directly by it. There may be a global variable that is set in this function that gets used by the feature in later executions. Or perhaps, the

function may write data to a file that gets read later by the function. Such cases are not detected by the feature execution trace, and require the developer to intervene by actually specifying the function's impact on the feature. In other words, adding the function name to the feature relation manually. See Fig. 3.13.



The dialog box contains the following text and controls:

To add a new function node under the current feature, enter the function name, then choose the file where the function is implemented, then click "Add Function Node".

The function will be directly connected to the currently selected node. If no node is selected, the function will be connected to the feature node.

Function Name:

Function Location:

please choose location

Fig. 3.13: Add Function Dialog

2. Tester's Use

Upon request, the full code path of a particular feature is shown, by default in Tree mode. As further information is needed, the tester can request to see the comments of each function by command-clicking the function name and showing the Comment view. This retrieves the function comment from the node and shows them to the tester. This is the closest thing to White Box Testing without having testers read actual code, or engage in technical discussions with the developers.

The tool also helps testers run more focused and more productive regression testing. For each build, every single change made is identified automatically and shown to the tester in red-colored nodes, and the *exact* features that use such functions are pointed out. Only these features need to be regressed!

In addition to regression testing, many new defects can be detected by testers through browsing feature trees. Furthermore, when a new defect is found, the defect can be entered with the exact code path and minimal steps to reproduce. Developers really appreciate the usefulness of such defect reports as it helps during debugging a great deal.

More advanced testers can dig deeper into the code and perform white box testing to determine new errors via the Code View. They can also be helped by complexity and reliability measures available to them for each function and feature in the code path via the Metric View. Many errors can be detected this way which otherwise (i.e. in black box testing) can be very hard to detect.

In a way, the tool offers the tester the opportunity to selectively perform white box testing at four different levels according to their technical skills and/or level of knowledge needed to do the job. This should result in higher test effectiveness and dramatic decrease in number of defects escaping to the field.

3. Writer's Use

Tree and Comment views allow the documentation personal to document the features at hand much more accurately reflecting every code-change in the feature's behavior. The alternative is to rely on developers for constant input, which places a heavy burden on the developers, or risk the danger of the documentation getting stale.

4. Manager's Use

The tool is of great value to team leaders (project, QA, and development alike). It helps them make better decisions in all aspects of project management, from tracking and assignment of all known defects and tasks, to better assessment of the quality of each feature and the productivity of the team, to better and more timely decision making in the areas of resource planning, release milestones, and release notes.

Managers often make bad decisions because they rely on their teams for input that may be inaccurate and/or late. The CMMR tool reduces the manager's dependency on the team, and provides the manager with instant and more accurate information centralized in one place. This is made possible by the tool's reliability metrics and trends, which are available at three levels (function, feature, and product) and computed on build-by-build basis. Figures 3.14, 3.15, and 3.16 show these three reliability charts, respectively.

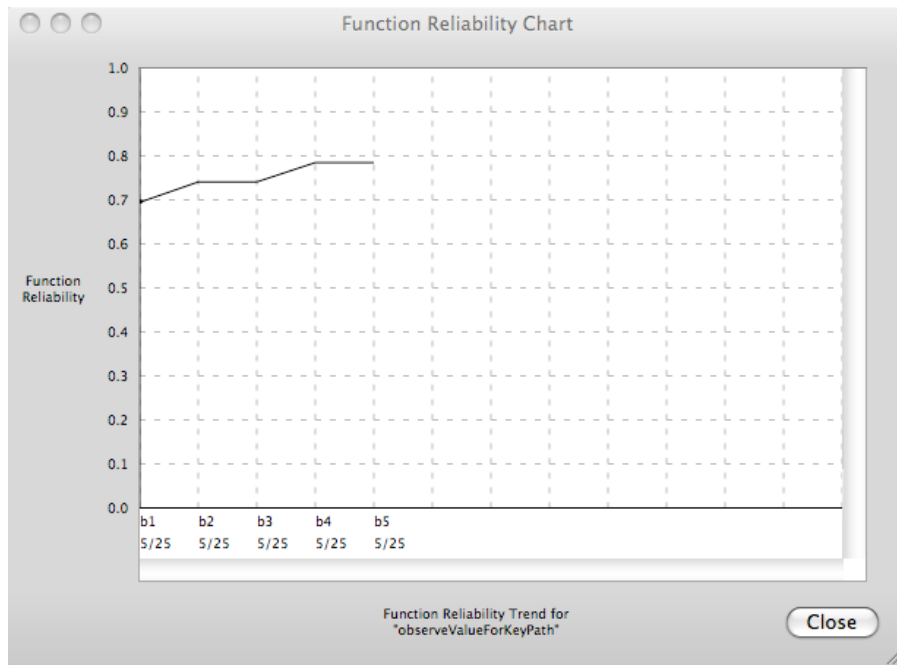


Fig. 3.14: Function Reliability Trend.

A function reliability measure, as a single number, is not that much

useful, however, a series of measurements taken over time across multiple builds and releases could be very useful. In general, a function reliability trend curve should never slope down unless new complexity is added to it. If a function is left unchanged for several builds and releases, its reliability trend will slope up by default due to an increase in maturity. The developer must keep these factors in mind when changing functions, and the best way to do that is to watch the curve progress before and after changes are made. If left unchecked, complexity will dramatically increase resulting in lower reliability measures of the changed functions and any features that use these functions.

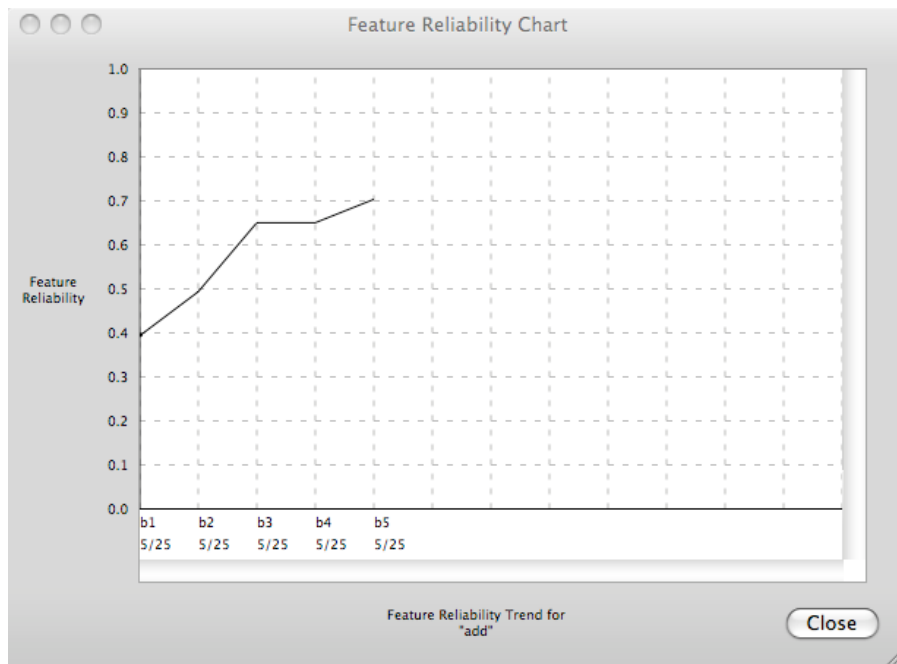


Fig. 3.15: Feature Reliability Trend.

Feature Reliability measures are more accurate than Function Reliability, whether derived from a single build or a series of builds. The reason for this is due to taking the average reliability measures from multiple functions, thus reducing any noise caused by outlier functions. Functions with very high or very low reliability will have little impact on the overall feature reliability.

The Project Reliability Trend (shown in Fig 3.16) is computed by taking the average of the reliability of features at each build. Of course, the assumption that all the product features have equal weights, in terms of importance to the project, may not always be true. Managers must take that into account when viewing these charts.

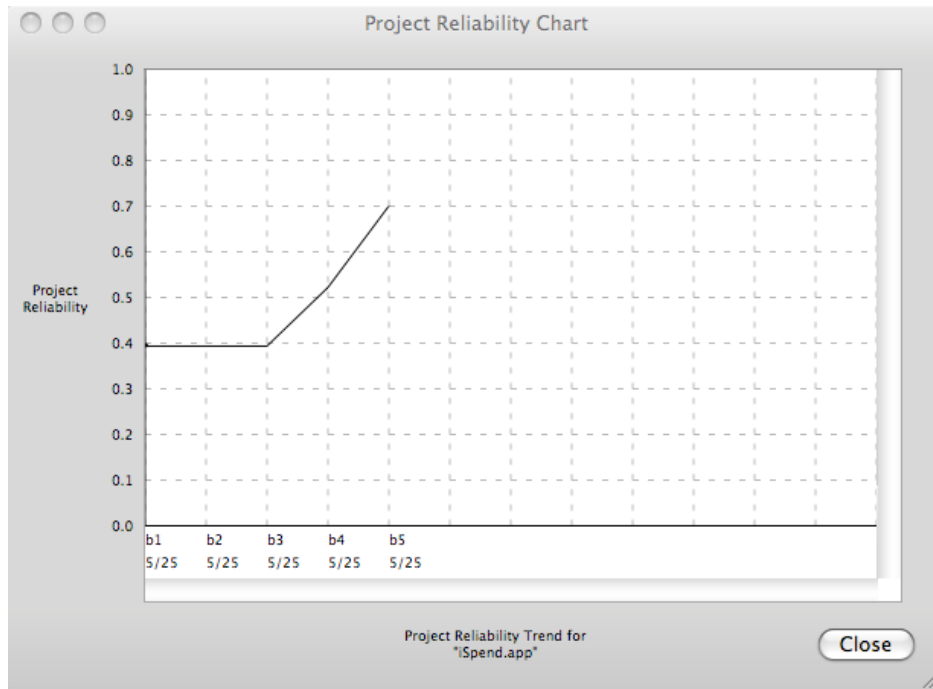


Fig. 3.16: Project Reliability Trend.

3.5 Summary

This concludes the methodology chapter where the major contributions of this research were discussed in some detail: the process model, the five metrics, and the CMMR tool. The design goals of the tool were two folds: first, to make it easy to use by the entire team with as little setup as possible, which was achieved by making the tool feature centric. Second, to serve as a companion tool to help adopt the proposed process model in all its phases, and the proposed metrics. None of the tools discussed in Chapter 2 were comprehensive enough to meet both criteria, and therefore cannot reduce the cost of software maintenance

by as much. As part of this research, the new process model, CMMR, and metrics were put to the test in several case studies. The next two chapters will discuss these case studies and the results obtained from them.

CHAPTER 4

CASE STUDIES

4.1 Introduction

In validating the ideas, techniques, models, and metrics presented in this research, the CMMR tool was used on five software projects, three of which will be discussed in this chapter. The first is a small Windows-based open-source Java project, named JContact. The second is a commercial C/C++/Objective C++ project for the Macintosh platform. The third is a sample open-source Macintosh Objective C++ project, named iSpend. This chapter discusses the work involved in preparing the tool to work with these three projects, some of the ideas gained from this experience, initial feedback from the team working on the projects, and a wish list of enhancements collected for a future update of the tool. The chapter is organized in three sections: Section 4.2: JContact, Section 4.3: A Commercial Macintosh Product, and Section 4.4: iSpend Mac project.

The formal results of the case studies and the research project as a whole are discussed in Chapter 5. It's worth noting that this is still work

in progress and some critical results data has yet to be realized. The CMMR tool was not available when the annual update of the Mac Product was started so it could not be scheduled in. Several aspects of the Mac Product used the CMMR tool and the proposed process model (as will be shown in Section 4.3), with a full utilization planned for the next maintenance cycle.

4.2 JContact – Open Source Java Project

This section discusses a case study that was conducted on JContact - a small Windows-based open-source project written in Java. The application allows the user to add records of contacts consisting of: contact name, sex, phone, and email address. Saved contacts can then be edited or deleted. The application also allows the user to save contacts to a file, and import contacts from an external file. A very simple application with several features, some of which have very complex call graphs. Fig. 4.1 shows the application's main window.

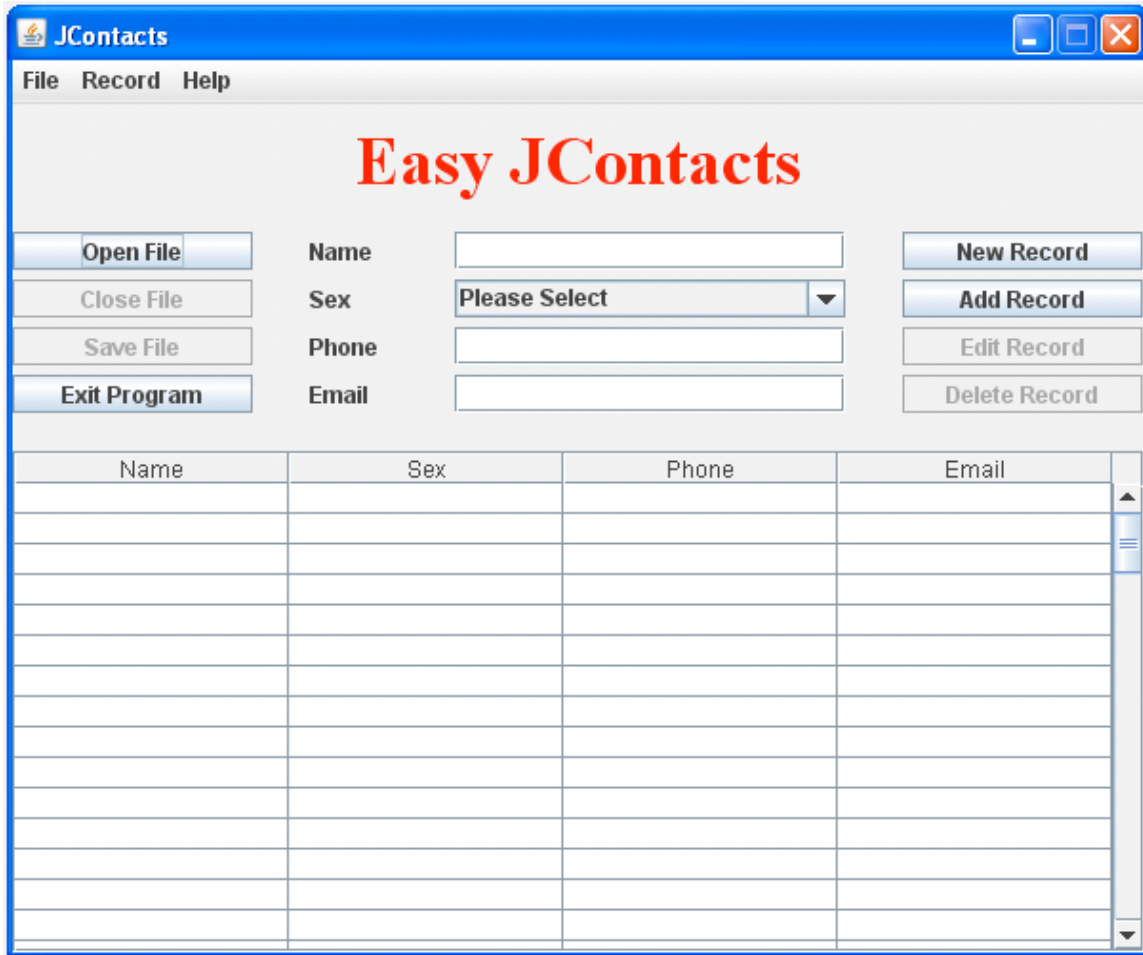


Fig. 4.1: JContact Main Window.

4.2.1 Java CMMR on Windows

Like the target application JContact, the Windows version of CMMR was also written in Java. Accordingly, it will be referred to as “Java CMMR”, from now on. It was designed to look and feel very similar to its Macintosh counterpart. For the most part, the feature-parity objective was accomplished, however there are some differences between the two implementations, which will be pointed out in the next

section (Use Interface). For example, a method name in Java tends to be very long, as it includes the full path name to the class interface as a prefix to the method name. The default node rectangle does not show the full name as a result, so a decision was made to make the rectangle resizable.

As far as architecture and design, the Java version of CMMR was designed to have two main packages: the analyzer, and the viewer. The analyzer deals with code analysis and conversion to XML. While the viewer reads the XML data, generates the graph trees based on that data, and presents to the user. The viewer also handles user interaction on the view area to support the other three views (source code, comments, and metrics). It also shows trend charts for the selected node. Appendix A shows the class diagram of all the Java packages and classes created by the Windows version of CMMR.

4.2.2. Java CMMR User Interface

In this section, Java CMMR user interface will be discussed and a few screenshots will be shown. The discussion will only highlight the key differences and areas that are unique to the Java implementation and the target JContact demo application.

1. Menu Bar and Menus

The menu bar and menu screen shots are very similar to that in the Mac version except the “Functions” menu title has been changed to “Methods”. See Fig. 4.2.

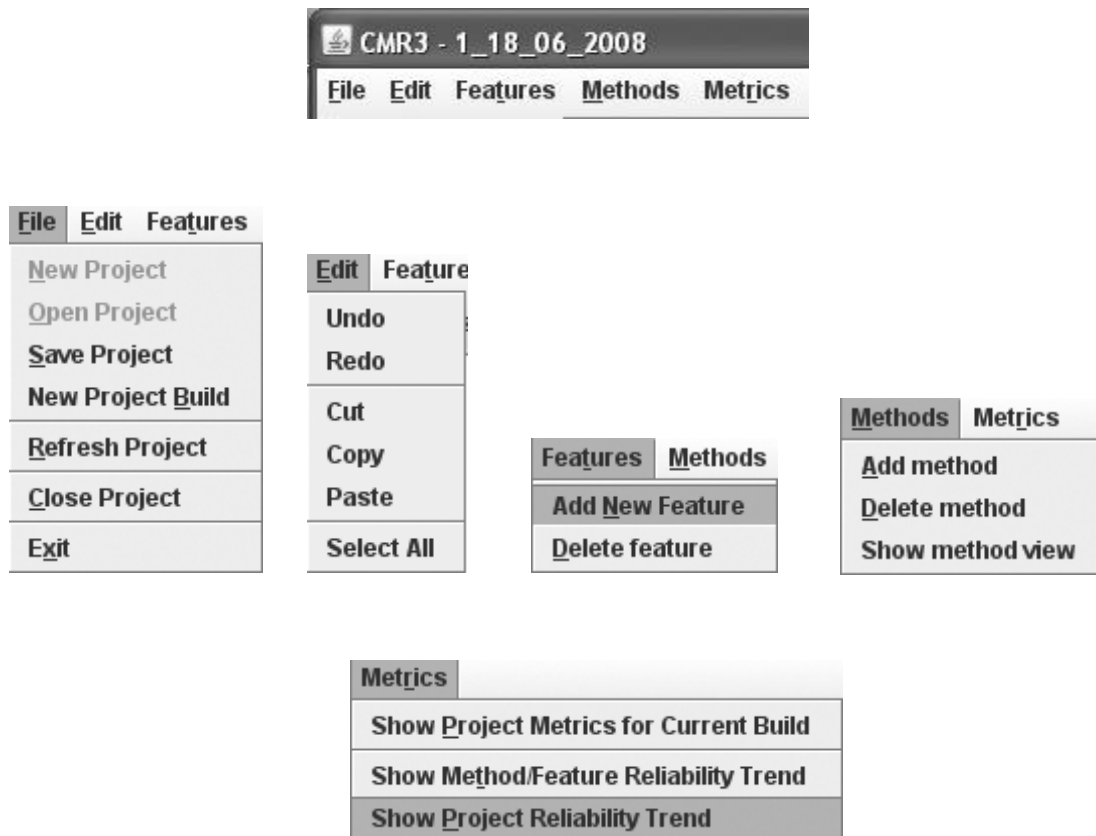


Fig. 4.2: Java CMMR Menu Bar and Menus.

2. New Project Window

In Java, the user must supply the path names to the source folder along with the libraries used by the Java application (see Fig. 4.3). The user

can browse to these folders (via the Browse buttons) or type the path names into the provided edit fields.

Target project sources	easyjcontacts\source\src	Browse
Target project libraries	easyjcontacts\source\lib	Browse
Target project location	easyjcontacts\source\out	Browse
Creation date	10/06/2005	
First release date	10/06/2006	
Last release date	10/06/2007	
Number of releases to date	10	

Next Cancel

Fig. 4.3: New Project Window (Java version).

3. Add Feature Window

Similar to the Mac version, JContact needs to be prepared to run a particular feature. The feature needs to be given a name, and the start and stop of the feature need to be defined, see Fig. 4.4. Unlike CMMR for the Mac, there was no need to touch the JContact source code directly to add profiling statements. Instead, Java CMMR operated on the intermediate Bytecode representation of the JContact code.

To add a new product feature, follow these four steps:

- Get your target application ready to run a particular feature.
- Enter the feature name and click "Start" button.
- Then immediately go to your application and run that feature.
- When the feature is complete, click "Stop" button.

Feature name Add Contact

Start Stop Cancel

Fig. 4.4: Add Feature Window (Java version).

4. Add Contact Feature Tree

After the feature has been run inside JContact, recorded, analyzed, it is converted to a graph as shown in Fig. 4.5.

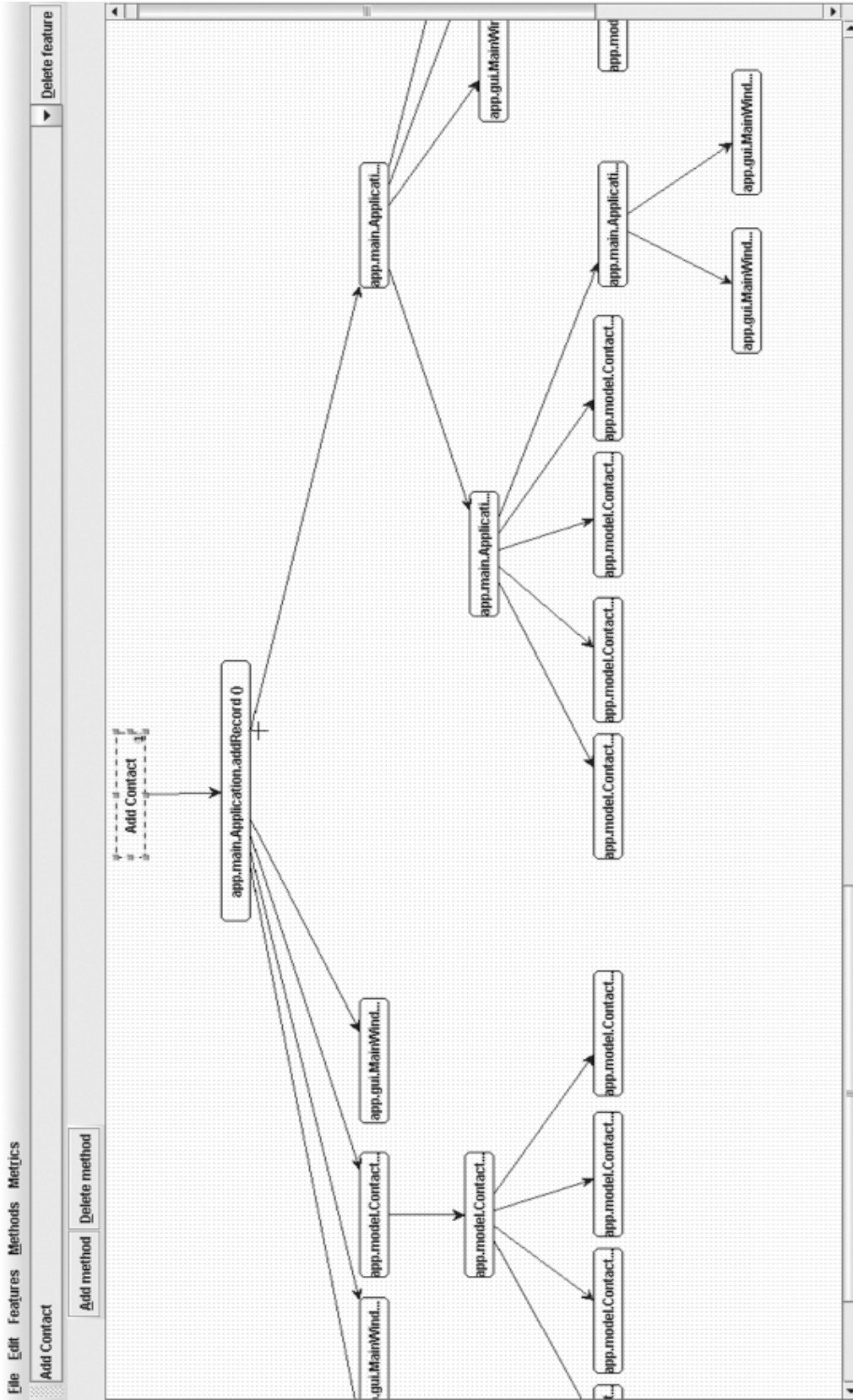


Fig. 4.5: Add Contact Feature Tree View.

5. Product Feature Menu

With a bunch of features added to the CMMR project, the Product Feature menu at the top of the window is populated with the feature names. Selecting a feature name switches the view to show the call graph tree of that particular feature. An example Product Features popup menu is shown in Fig. 4.6.

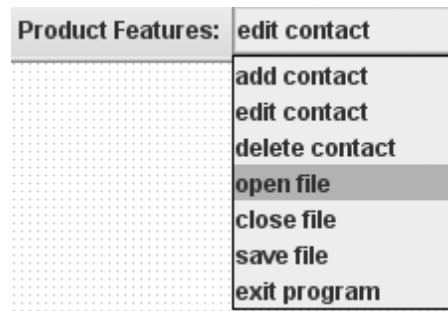


Fig. 4.6: Product Features Menu (Windows version).

6. Tree View with Changed Methods

After a new build is made, the feature functions are compared with the previous build and a couple of functions are found changed, Java CMMR highlights changed nodes with a red oval in the top right corner (see Fig. 4.7). If the function is shared by multiple features, the number features is also shown inside a green oval in the lower right corner of the function node.

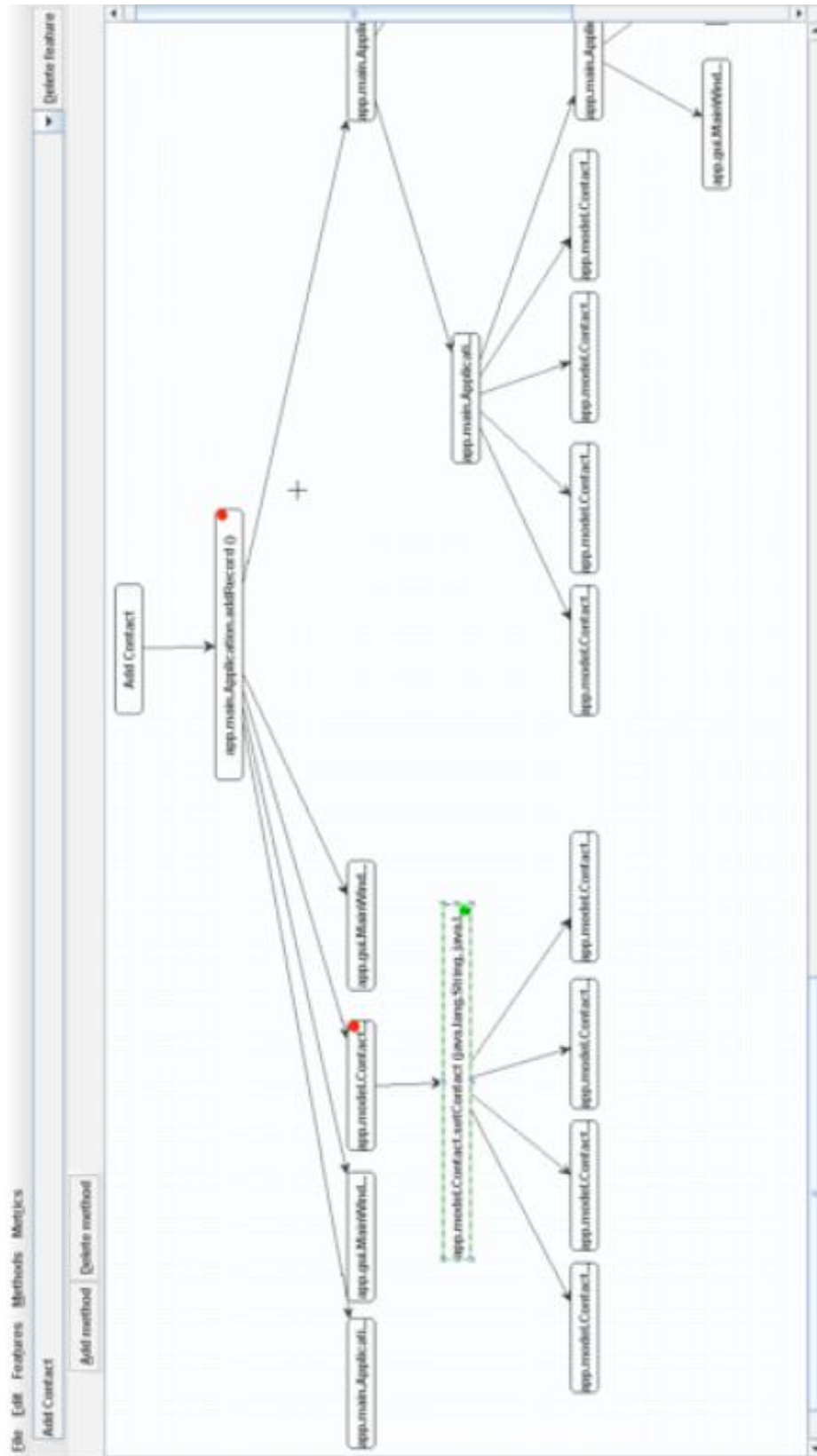


Fig. 4.7: Add Contact Feature in Build 2 with Two Changed Functions

7. Method Source Code View

A method node is selected and its Source Code View window is shown in Fig. 4.8.



```
app.main.Application.addRecord ()  
  
    if(validateField()) {  
        modified = true;  
        if(!edit) {  
            record = new Contact(window.getFieldValues());  
            records[numOfRecs] = record;  
            numOfRecs++;  
        }  
        else {  
            int i = dataTable.getSelectedRow();  
            records[i].setContact(window.getFieldValues());  
            edit = false;  
        }  
        window.enableButt(3, true);  
        resetFields();  
    }  
}
```

Fig. 4.8: Method Source Code View (on-screen text color is blue).

8. Method Comments View

A method node is selected and its Comments View window is shown in Fig. 4.9.

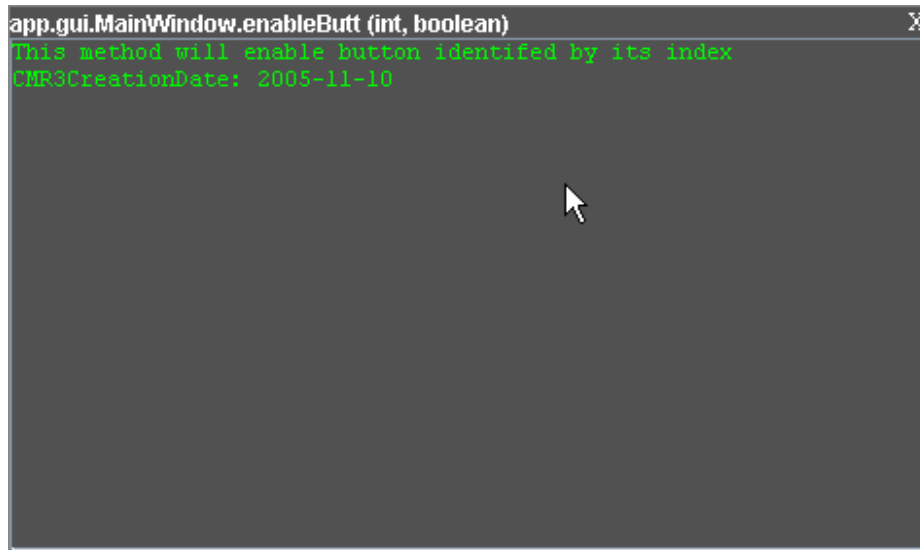


Fig. 4.9: Method Comments View (on-screen text color is green).

9. Method Metrics View

A method node is selected and its Metrics View Window is shown in Fig. 4.10. Metrics computed here are limited to the original version. More metrics will be computed and shown to match the Mac version of CMMR.

Metric:	Value:	Description:
Creation Date:	2005-06-10	-
Last Modified:	2005-06-10	-
Number of Features:	1	Add Contact
Number of Releases:	10	High - Excellent
Lines of Code:	15	Short - Excellent
Lines of Comments:	0	No Comments!
McCabe FCC:	0.5	Low - Excellent
FBFC:	0.5	Moderate
Maturity:	1.0	High - Very Good
Reliability:	0.75	Moderate

Fig. 4.10: Method Metrics View

10. Method Reliability Trend

A method was modified between two builds adding more code (complexity) thus reducing its reliability slightly. See Fig. 4.11.

11. Project Reliability Trend

The Project Reliability Trend chart was affected downward due to several changes in methods between three builds. See Fig. 4.12.

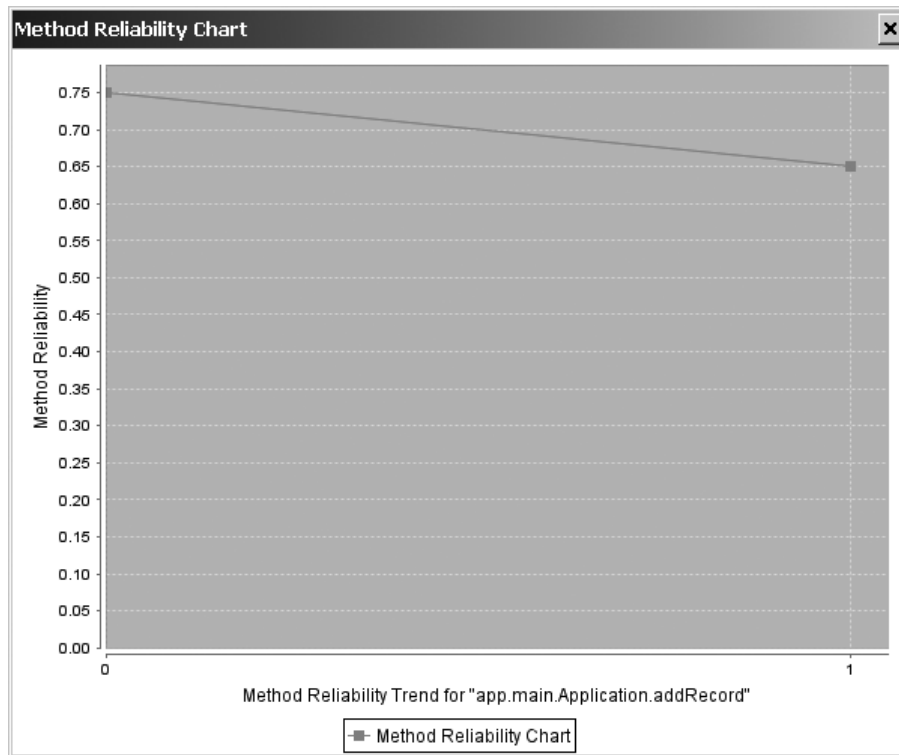


Fig. 4.11: Method Reliability Chart

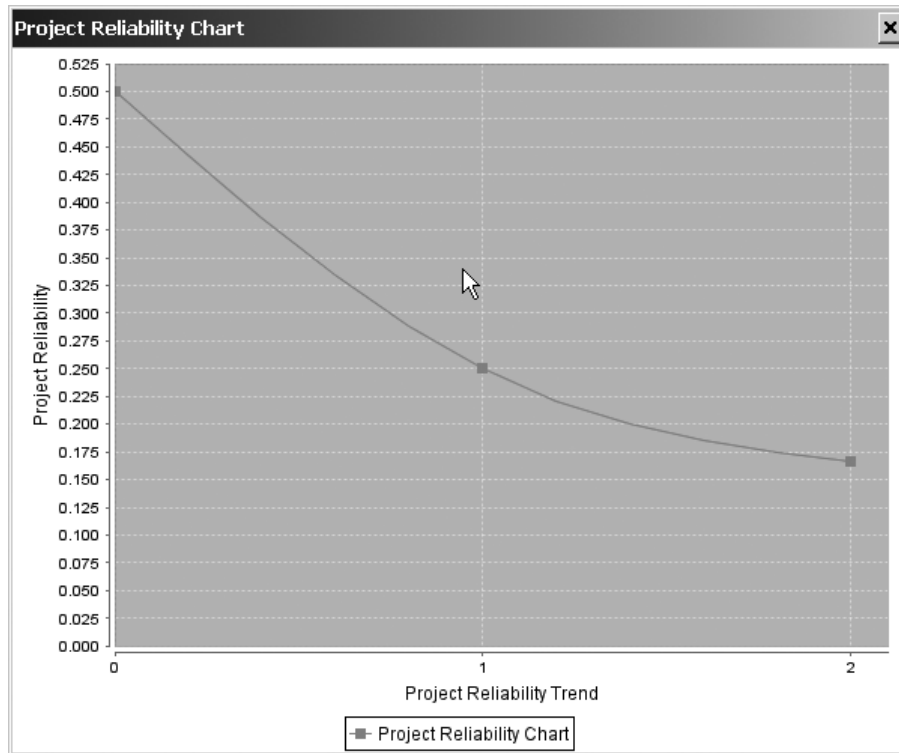


Fig. 4.12: Project Reliability Chart

4.3 A Commercial Macintosh Product

This section discusses a case study that was conducted on a Macintosh-based C/C++/Objective C/Objective C++ commercial product. Unfortunately, the identity of the product cannot be exposed at this time. It will therefore be referred to in this section as the “Mac Product”. The aim of this case study is to prove that the ideas proposed in this research are practical and can be used in real projects that are very large and complex; i.e. not for demo only.

4.3.1 The Mac Product

The Mac product is a software package developed for the Macintosh business market. It has been under development and maintenance for over 20 years. It ships to customers once a year followed by a couple of minor updates to remove latent defects and incompatibilities. The original product was the result of porting the Windows version of the product, modified to work on the Mac. As the two platforms evolved, the two products diverged in terms of functionality and underlying code base. The product receives a major update every year with several new features added, and several enhancements and bug fixes are made. The first six months of each cycle is spent adding and testing the required changes. The second six months are spent on perfecting the changes, with input from customers, then final delivery.

The Mac product is considered a large product delivering hundreds of major features to business customers. Among the many features is the ability to create and print various business reports and lists. The product has several platform-specific features that allow it to integrate with other applications such as Microsoft Office, Apple iLife suite, and others. Feature requirements are derived from customers and

approved by management prior to each maintenance cycle. The next three sections cover the current Mac Product's codebase, maintenance process model, and tools. This information is provided as a background to adopting the new metrics, process model, and tool.

1. Codebase

This Mac project consists of approximately 3800 files. The average size of each source file is about 770 lines of code for a total of 3 million lines of code. The average size of each function is about 20 lines of code (LOC) but some functions from old legacy code exceed 100 LOC. There are about 150,000 functions and methods in total. The code is written mostly in C and C-derived languages: C++, Objective C, and Objective C++. Over half of the code is legacy (ten years or older) written in C and C++. The other half is considered new and written in Objective C/C++.

Design documents of old legacy code are non-existent. Coding style and naming conventions are inconsistent. The code is somewhat commented. The overall code is very well structured yet very complex. Complexity comes from the vast size of the project and the turnover of the original code authors.

2. Development/Maintenance Process Model

The process is iteration-based. At the start of the iteration, a planning meeting is held to define and select a group of requirements (user stories) to be done in the iteration. Some rough estimates are made to determine what can be done, and by whom, in the two-week iteration. As developers complete their stories, the stories are assigned to testers to verify they are really completed. Limited regression testing takes place at this early phase of development. At the end of the iteration, a demo meeting is held for the entire team to demonstrate and review the changes just made, and to determine if more work is needed. Another planning meeting is held for planning the next iteration, and the cycle continues until all the requirements are completed. At which point, intensive system testing and defect removal takes place to get the product ready for beta testing by customers.

Beta testing usually uncovers a few areas that require more work and some defects that were not caught internally. The team attends to these issues and verifies them internally. A few more beta updates are sent out to give the customer a chance to verify their issues are indeed taken care of. This process continues until the team and customers

feel the product is complete in terms of features and quality. The product then enters the delivery phase. After delivery, a couple of minor updates are made and sent out, as needed. These minor updates are developed using the same process model but they tend to be short focusing on urgent defects only.

3. Tools and Automation

In the process just discussed, the team uses several tools to facilitate communication and ease the work. There is an on-line system that manages the iteration-based feature requirements and project scheduling. There is a defect tracking system that is used for handling defects and change management during the later stages of maintenance. Acceptance testing is semi-automated and regularly used. There are regular meetings that are managed by scheduling tools, emails, and chats are often used between the team members.

There are no maintenance tools to help the team in program comprehension, change impact analysis, regression testing, and measurement tasks. The first three tasks are handled on ad hoc basis and rely heavily on the experience and skill of the individual. There are no measurements of complexity or reliability whatsoever. In fact, the

only metrics used are: number of defects found and fixed, number of stories/features completed, and time to milestones (feature complete, beta testing, delivery to customer, etc.).

4.3.2 CMMR Adoption and Feedback

CMMR was introduced to the team as a tool to reduce maintenance cost. The cost reduction comes from several areas in the maintenance process which they were all too familiar with and knew were problematic (i.e. regression testing, program comprehension, change impact analysis, complexity measurement, etc.). This generated some excitement among the team, however some of the excitement quickly faded away after they learned that the tool is still not fully automated and that there is a setup cost associated with using it. The biggest complaint was about having to touch the code to add the profiling code. Although, such code appears only in the debug version of the project, it involves inserting code within actual codebase that ships to customers, and it does get in the way while viewing and editing the original code. They were informed that this area is still work in progress and that it can be avoided altogether with the use of a new operating system technology called DTrace, which is supposed to generate trace profiles automatically without adding any debug code.

Despite the limitations, some team members saw the tool's potential clearly, continued to use it, and gave valuable feedback (see next section), some of which were incorporated right away, while others are planned for a future release. They really appreciated the feature-based, multi-level approach to supporting different classes of users (i.e. the comment, source code, metric windows). Engineering Management specially liked the ability to track changes, measure feature complexity, in terms of number of functions and LOC, and the ability to watch trends of complexity and reliability of each feature and for the product as a whole.

QA folks enjoyed the “red nodes” – the ability to pinpoint the exact functions that were changed in a given build, allowing them to detect where code changes were made and which features were impacted. This is by far the best regression testing selection technique some of them have seen.

Project Management expressed concern about the possible use of metrics to measure performance – an area that is very sensitive to managers and engineers alike. Such concern may prevent the use of

CMMR or limit its use in order not to negatively effect employee moral.

Beyond that concern, management saw several benefits:

- Ability to understand the completeness and thoroughness of each feature.
- Ability to identify risks and areas that require additional resources.
- Ability to identify areas where engineer coaching is needed.
- Allows for better communication between the manager and the engineers.

Documentation folks have not provided any feedback so far, as they have yet to be assigned to the project. Their job starts after the completion of all feature development and the start of integration testing (alpha and beta testing).

Feature Requests

A list of feature requests was compiled from the team who used the CMMR tool. Table 4.1 summaries the list and shows which requests were actually implemented to accommodate the initial feedback and encourage continuous use. Other features were also added based on discussions with the team and not listed in the table. They include: adding metrics that indicate the size of functions, not just its structure

(i.e. Halstead), and adding more metrics that take both size and structure into account (i.e. MI). Finally, before CMMR was used on the Mac Product, the original feature reliability metric was equal to the function with the least reliability measure. This was based on the assumption that a single unreliable function will have the same negative impact on a feature as many unreliable functions. However, in working with the Mac Product, many functions were complex (low reliability, by design) causing the feature reliability to compute as being low. This assumption was therefore relaxed and now the feature reliability ignores outliers and takes the average reliability of the underlying functions. The same applies for the product reliability computation and trend.

Next year, the CMMR tool will be used in a full maintenance release cycle that follows the maintenance process model proposed here. Performance measurement (such as productivity, quality, and duration) will then be taken and compared with previous release cycles. Reduction in maintenance cost should be noted as a result of higher productivity and better release quality.

Table 4.1: Macintosh Product Team Feedback

Feature Request	Done
When adding a new feature, check for duplicate feature name	
When browsing between nodes when command/option/control key is down, enable use of arrow keys. Left arrow key moves to left-most child node. Right key moves to right-most child. Up arrow key moves to parent node. Down arrow key moves to first child.	Yes
Show "project metric window" when clicking on window title CMMR icon	
When opening source file in editor (via double click) scroll file into view and select function name.	
Make the color of changed function nodes red, rather than adding a red tiny bullet on top of the node. This makes the changed nodes more visible.	Yes
Add function should browse a source file, find the function name, extract its code, and setup node before drawing it. If not found, let user know.	Yes
When adding a feature and detecting files removed post an alert telling user that feature is stale and should be re-added (updated).	Yes

Show file path in metric window.	
Adjust location of nodes (feature node mostly) for better layout.	Yes
When deleting a node with children give user choice to delete the children nodes or have them get adopted by their grandparent node.	
When purging the list of log statements, detect loop vs. recursion duplicates and give an indication in tree/metric window.	
Fix code inside the node making sure it captures the entire function lines from { to }	Yes
Search for comments on top of the function name.	Yes
In metric window: number of releases - fix the description to base it on the number of releases vs. product releases.	
The tiny green “number of features” indicator: enlarge and move out of node slightly overlapping a corner (like Mail/XCode dock icons).	
Show more curves on the same trend chart. Perhaps multiple metric curves, or the function, feature, and product curves all on the same chart.	

4.4 iSpend - A Sample Macintosh Project

This Objective C++ project was developed by Apple to demonstrate the use of some of their Operating System technologies. It was chosen as a case study here because it provides a moderate set of features for CMMR to analyze, and it is open-source, which allows for exposing portions of its source code. The project allows the user to manage their spending by tracking each and every transaction made (see Fig. 4.13). Transactions can be added, edited, and deleted. The project includes a toolbar and offers extensive “undo” and search capabilities. It can save documents to disk and manage multiple documents at once. All these features were exercised under CMMR and several feature trees were generated.

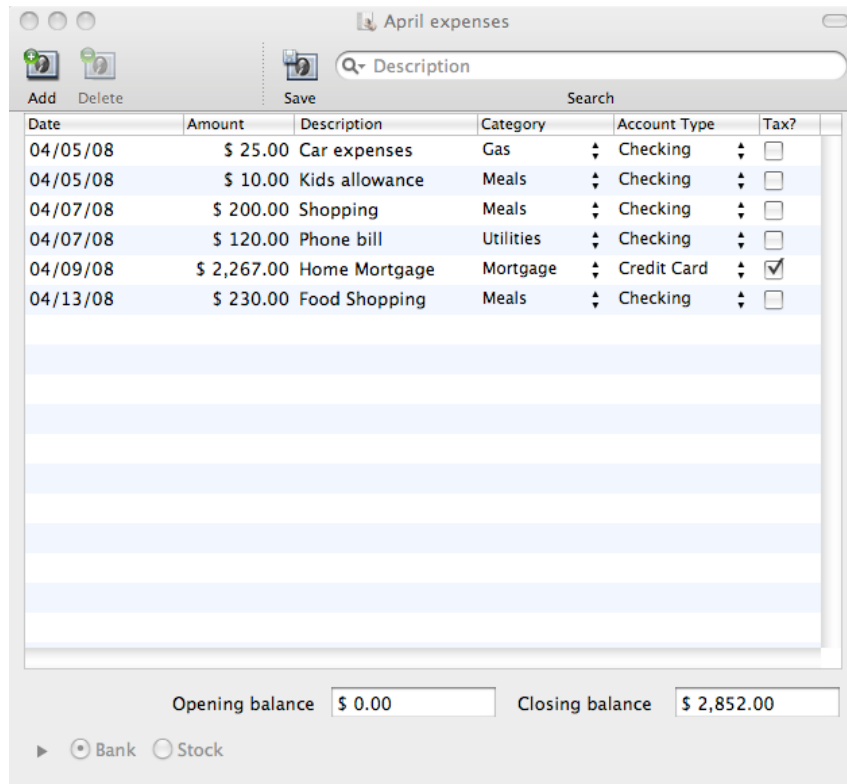


Fig. 4.13: iSpend Main Window

The iSpend source code project is relatively small. It's made up of 18 source files containing nine classes responsible for all the project functionality. Total lines-of-code in the project is about 2700 lines of code. Total functions/methods is a little over a 100. The project was manually instrumented by adding the "CMMRLog" statements so that all visited functions, at all levels, are dumped to the log file when running any feature. Instrumenting this project took less than an hour. Several builds were generated over a three-month period between April 2008 and June. Fig. 4.14 shows the CMMR project settings for

the iSpend project. Of importance are the date fields at the bottom of the window, which are used heavily by the Function Maturity metric.

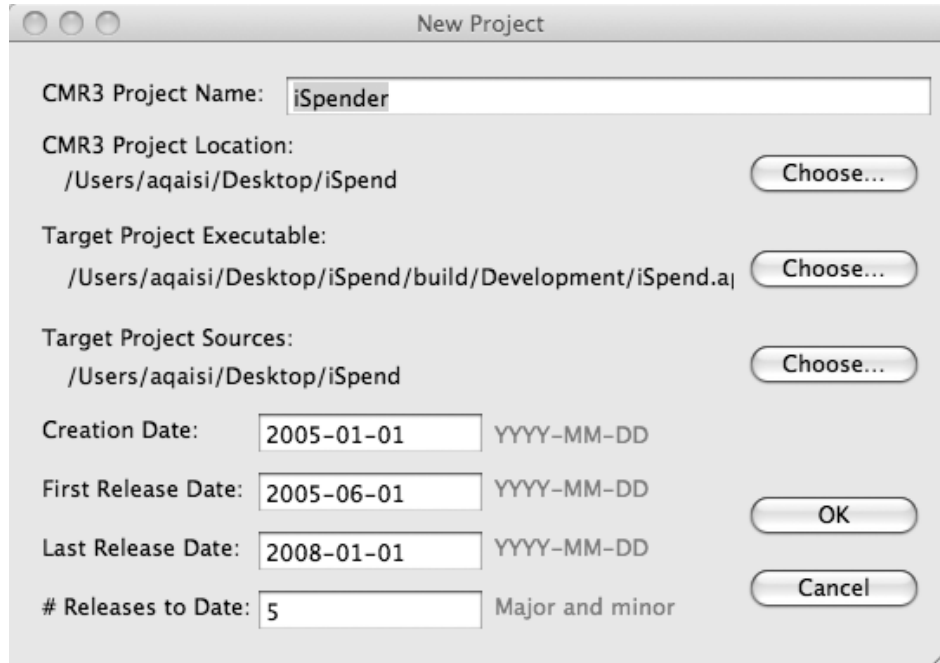


Fig. 4.14: iSpend New Project Window

The major focus in this particular case study was on metric measurements and reliability trends. In each build, several changes were made to a function named “observeValueForKeyPath” breaking it down into several sub-functions, replacing a complex portion of the code with a switch statement, and adding/removing “dummy” code and comments. In some cases, the function was modified to reduce complexity in the original code, and to inject and fix defects. The

function was originally long (51 LOC) but was reduced dramatically during maintenance.

The *observeValueForKeyPath* function is actually used when adding a spending transaction and when deleting it. Any changes in this function impact both features. The Metric window of this function's original code, i.e. before any changes were introduced, is shown in Fig. 4.15.

Metric	Value	Description
Creation Date:	2005-04-12	—
Last Modified:	2005-04-12	—
Number of Features:	3	add trans
Number of Releases:	5	High - Excellent
Lines of Code (LOC):	51	Too Long
Lines of Comments:	30	Well Commented
McCabe Cyclomatic Complexity V(G):	18	Moderate
Halstead Vocabulary (n):	64	High #Unique Operations
Halstead Length (N):	163	Very High #Operations
Halstead Volume (V):	978	Very High #Operations
Maintainability Index (MI):	61.30278	Moderate Maintainability
Kafura Structure Complexity (Cp):	256	High Fan-in/Fan-out
Card & Glass System Complexity (C):	274	High
Feature-Based Maintainability (FBM):	0.3868819	Moderate
Maturity:	0.9548301	High - Very Good
Reliability:	0.670856	Moderate

Fig. 4.15: *observeValueForKeyPath* Original Metrics View

The function is relatively mature and has been included in 5 releases, thus the high 0.953 FM value. However, the code is relatively long and somewhat complex yielding 0.387 FBM value. The FR value was computed from the average of FM and FBM value yielding a 0.671 value, which is considered moderate. The aim, of course, is to increase that value by reducing the complexity of the function.

Ten builds were generated over a period of three months, and in some of the builds, the function was simplified and new metrics were computed for it. Some of the changes in this function were non-functional in nature; i.e. injecting defects and fixing them, or adding/removing “dummy” code and comments. This was done to test the accuracy of the measurement and computation of the various metrics used by this research. Results and discussions of this area of the case study will be covered in some detail in the next chapter.

4.5 Summary

This chapter discussed three case studies where the contributions of this research were applied. Two of the case studies were open-source (one Windows-based and one Mac), while the third was a real commercial product with a huge code base and a large team working

on it. The focus of this chapter was to provide background information about the target projects and software systems, explain the deployment of the research contributions on these projects, and provide initial feedback from the maintenance team involved in these projects. The actual results of these case studies were not included in this chapter but will be covered in the next chapter.

CHAPTER 5

RESULTS AND DISCUSSION

5.1 Introduction

This chapter summarizes the major results of this research, which were based on actual case studies conducted in 2008 on five projects, some were open-source, and one was a commercial product. Three of these case studies were covered extensively in the previous chapter. This chapter focuses on the results obtained from the case studies and the overall research. The results are classified into several categories as shown in the next sub-sections. Some of these results were objective with actual data derived and analyzed (see Sections 5.2 through 5.4), while other results were subjective (see other remaining sections in this chapter) with data based mostly on feedback from the team who participated in the case studies.

5.2 The Research Hypotheses

This research started out with a few hypotheses (see Section 1.5). As the research evolved and the CMMR tool was put to use in several case studies, these assumptions were validated and the hypotheses proven. The following discussion shows how each hypothesis was proven experimentally.

H1&H2: Feature Trees Yield Better Program Comprehension. The first two hypotheses claim that isolating the code path of a single feature from the rest of unrelated code and presenting it as a tree results in better program comprehension. This was observed in all case studies, especially in the commercial Mac Product (see Section 4.3), which was very large and complex. To illustrate this point, Table 5.1 shows some common metrics of one particular project feature relative to the project as a whole. The feature was fully developed when the CMMR analysis started; i.e. it was in maintenance mode. More maintenance of the feature is expected in later stages of the maintenance cycle and before customer delivery.

Table 5.1: Mac Product Feature vs. Project Metrics

Metric	Feature	Project
# Lines of Code	8450	3000000
# Functions	640	150000
# Files	14	3800

The table shows a significant reduction of code size that a maintainer has to deal with when maintaining the feature. Moreover, keeping the feature code isolated (in a call graph tree) from the rest of the massive

project code insures better focus on the feature at hand. Results of the Mac Product case study indeed showed that program comprehension was significantly faster for both developers and testers.

The project team also agreed that most maintenance work is indeed feature-based and keeping the target feature isolated and graphically represented helps the team in several activities including maintenance, testing, management, and communication. One developer claimed that showing the feature in outline mode and with collapsible nodes is more ideal than a tree mode, since it allows for parts of a large feature to be hidden/exposed as necessary. While this is true, the tree offers other advantages over the outline mode in the areas of multiple views and future extensibility. Actually, both modes were originally planned for the first release of CMMR, but the outline mode could not be implemented in time, and was thus deferred to a future version.

H3: Easier White-Box Testing Through Comments. This hypothesis claims that white-box testing is a lot easier when viewing comments than when viewing actual source code. Comments are easily

understood by all testers and regardless of their programming skills. Understanding source code, on the other hand, requires special skills that most testers don't have. To illustrate this hypothesis, the two figures (Fig. 5.1 and Fig. 5.2) show the source code view and the comment view, respectively, for one function in the iSpend project (see case study in Section 4.4).



```
observeValueForKeyPath

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context
{
    if (context == SpndTransactionsContext) {
        NSArray *oldTransactions = [change objectForKey:NSKeyValueChangeOldKey];
        if ([oldTransactions count] > 0) {
            [[self undoManager] prepareWithInvocationTarget:self] insertTransactions:oldTransactions atIndexes:[change
objectForKey:NSKeyValueChangeIndexesKey];
            [self stopObservingTransactions:oldTransactions];
        }
        NSArray *newTransactions = [change objectForKey:NSKeyValueChangeNewKey];
        if ([newTransactions count] > 0) {
            [[self undoManager] registerUndoWithTarget:self selector:@selector(removeTransactionsAtIndexes:) object:[change
objectForKey:NSKeyValueChangeIndexesKey];
            [newTransactions makeObjectsPerformSelector:@selector(setDocument:) withObject:self];
            [self startObservingTransactions:newTransactions];
        }
    }
    else if (context == SpndAmountContext) {
        [self willChangeValueForKey:@"balance"];
        [self didChangeValueForKey:@"balance"];
    }
    else if (context == SpndTypeContext) {
        NSString *category = [object valueForKeyPath:@"*type"];
        if (category != nil && ![category isEqualToString:@""] && !_categories containsObject:category) {
            [self mutableArrayValueForKey:@"categories"] addObject:category;
        }
    }
    else if (context == SpndAccountTypeContext) {
        NSString *accountType = [object valueForKeyPath:@"*accountType"];
        if (accountType != nil && ![accountType isEqualToString:@""] && !_accountTypes containsObject:accountType) {
            [self mutableArrayValueForKey:@"accountTypes"] addObject:accountType;
        }
    }
    else {
        [super observeValueForKeyPath:keyPath ofObject:object change:change context:context];
    }
}
```

Fig. 5.1: observeValueForKeyPath Original Source View

```
observeValueForKeyPath
CMR3CreationDate: 2005-04-12

we'll be notified whenever there is a change to transactions, whether the array is replaced or an object is
added or removed, because of the observing we set up in -init
set oldTransactions to the array of transactions that have been removed
this change can be undone by adding oldTransactions back into the arrayController
set newTransactions to the array of transactions that have been added
this change can be undone by removing newTransactions from the arrayController
set the document in each transaction so that the transaction can find our undoManager
observeValueForKeyPath... gets called with one of the below keyPaths ("amount", "type", or "accountType")
when the value for that keyPath changes in a transaction, because of the observing we set up above
the amount has changed in one of the transactions. Cause balance to get updated
the type has changed in one of the transactions. If this is a type we haven't seen before, add it to _categories
get type
the account type has changed in one of the transactions. If this is an accountType we haven't seen before, add
it to _accountTypes
the notification wasn't recognized, so it was probably meant for someone else. Invoke super.
```

Fig. 5.2: observeValueForKeyPath Original Comments View

It's obvious that the code, shown in Fig. 5.1, is harder to comprehend than the English comments in Fig. 5.2. Advanced testers wishing to go beyond comments and actually see source code are of course allowed. However, most testers prefer to work with comments, at least initially then move to the source code, as needed. CMMR offers both views conveniently with one-click of a mouse. If the tester wants both views together, a double click opens the intended function inside the development environment.

Testers working on the Mac Product who used CMMR agreed that these views offer testers with multiple easy-to-use ways to perform white-box testing. One tester was able to learn a feature, start testing it, and produce defect reports on day one! Another tester spotted a defect just by viewing the code view of one function. This tester realized that she could not have spotted the same defect in traditional white-box testing methods (i.e. via viewing the source code directly without the CMMR tool). She wondered: “Without CMMR, how would I know that the function is part of the feature?” Then, assuming she knew, locating the function in the source repository is not a trivial task.

H4&H5: Better Tracking of Code Changes Yields More Focused Regression Testing. Hypotheses 4 and 5 claim that better tracking of code changes provides testers with a quick way to find all the impacted areas. This in turn results in more focused regression testing and more efficient error detection and removal. Both hypotheses were proven in all the case studies covered here. CMMR automatically detects code changes in every new build by actually comparing code inside the feature nodes against the latest build’s code. Any changes are highlighted in red for the tester to immediately spot. Showing the Metric

view of a red node also shows a list of all features impacted by that change. In a single click the tester gets so much information that would have taken hours otherwise. In fact, one tester of the Mac Product felt that this is by far the most advanced regression testing technique the tester has ever seen.

Table 5.2 illustrates these advantages in terms of number of defects found during the maintenance of JContact project (see case study in Section 4.2). In the case study, a change was made in JContact and a new build was generated and handed to two testers. One tester used CMMR and the other used traditional methods. The tester using CMMR reported three defects while the other tester reported two. Beyond defect count, there is a time factor that is just as important: how long after the build was available the defects were found. The defects reported by the tester using CMMR were immediately detected; i.e. within minutes after the build was available. Traditional methods do find bugs, but more often than not, the defects are found long after they were introduced.

Table 5.2: Defect Detection via CMMR vs. Traditional Methods

Testers	Defects Found	Time to Find Defects
Tester Using CMMR	3	1 min., 3 min., and 12 min.
Tester Not Using CMMR	2	5 min., 135 min.

Quick error detection leads to a quick error removal. Some errors are not detected until days or weeks have passed since they were originally injected. Such delay increases code decay and decreases developer's efficiency in error removal. No experimental data was available for this area of the project, but is planned as future work.

In general, testers using CMMR felt "closer" to the code but without having to deal with the complexity of the code. Testers and developers both appreciated the tool's facilities to detect defects more quickly and to report the defects more accurately, allowing for faster defect removal.

H6: Metrics and Reliability Trends Help Management. This hypothesis claims that having the code metrics and reliability charts,

available and updated at all times, helps managers make more timely and effective decisions. The engineering manager and the product manager of the Mac Product both agreed with this claim. These results were evident in the iSpend case study by tracking all code changes and function reliability measurements of a single function for a duration of three months (see Fig. 5.3).

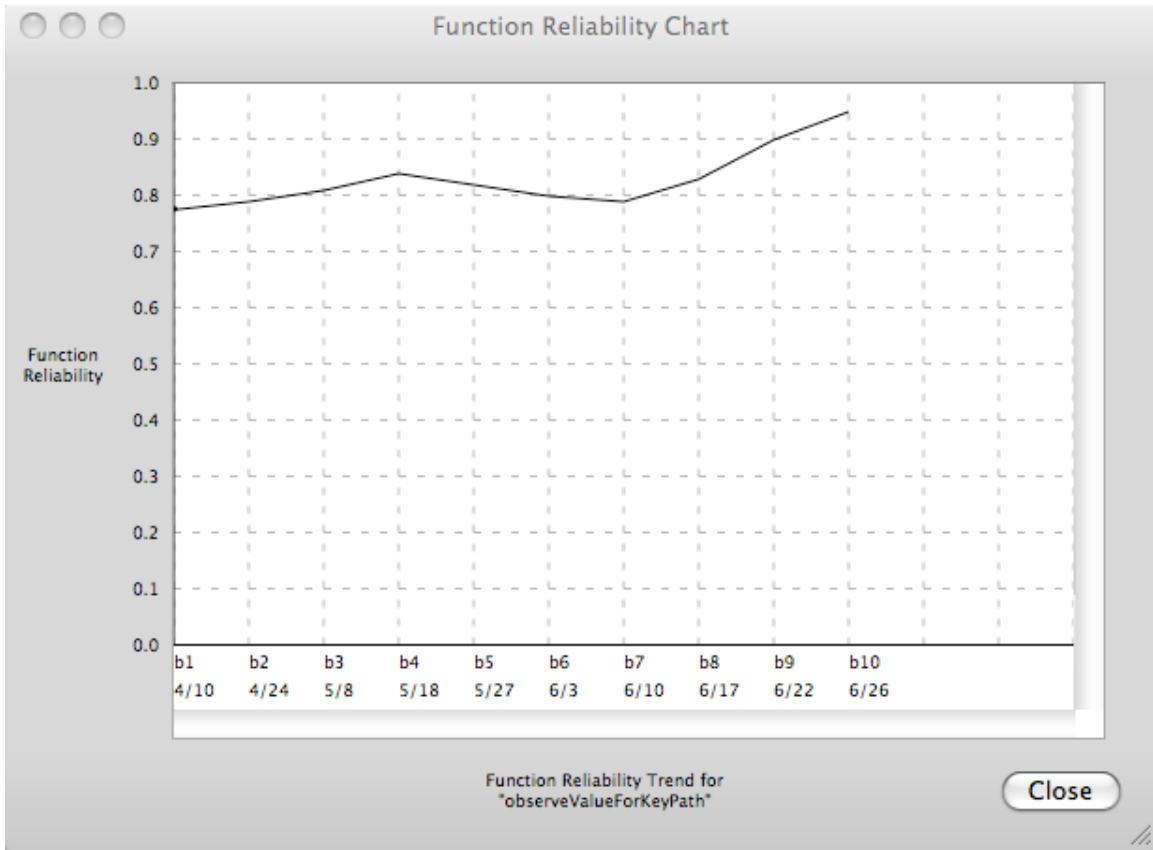


Fig. 5.3: Function Reliability Trend in a 3-month Period.

The figure shows the function “observeValueForKeyPath” starting out

with a moderate FR value, thanks to its high FM value (maturity). The objective of the engineering manager was to reduce complexity of this function and improve its reliability to an acceptable level (0.9 or higher). The strategy by the developer was to use a switch statement and break the function down into several small functions. In the next three builds, the results of these changes were evident by the reliability curve going up. However, these changes were visible to testers using CMMR who quickly reported a defect in b4. More code changes were made (code addition, mostly) in b6 and b7 to fix this defect. Beyond b7, the function was simplified further until its FR value reached a desired level of 0.95.

The reliability trend is not limited to managers only. It can be beneficial for all the team members. However, managers are typically the decision makers for feature-related issues. Decisions are usually based on data, and the more reliable the data is and the faster it is retrieved, the better the decisions are. The FR measurements used in Fig. 5.3 are based on other metrics that are computed from the actual source code inside the function. Table 5.3 shows the results of the individual metrics used in the 3-month trend analysis. These results were evident in the iSpend case study (Section 4.4).

Table 5.3: Metric Values of “observeValueForKeyPath” in Multiple Builds.

Metric	b1 4/1 0	b2 4/2 4	b3 5/8	b4 5/1 8	b5 5/2 7	b6 6/3	b7 6/1 0	b8 6/1 7	b9 6/2 2	b10 6/2 6
LOC	51	24	20	21	19	22	25	18	16	15
Comments	30	10	10	10	8	8	8	8	6	6
VG	18	16	14	14	13	15	16	13	12	11
n	64	30	28	28	28	30	32	30	28	26
N	163	96	88	88	88	102	118	113	90	85
V	978	384	352	352	352	408	472	452	360	340
MI	61	65	70	65	70	67	65	73	85	99
Cp	36	400	400	400	400	400	400	400	400	400
C	54	416	414	414	413	415	416	413	412	411
FBFM	0.2 8	0.3 0	0.3 3	0.3 0	0.3 3	0.3 1	0.3 0	0.3 5	0.4 2	0.5 0
FM	0.9 5	0.9 5	0.9 5	0.9 5	0.9 5	0.9 5	0.9 5	0.9 5	0.9 5	0.9 5
FR	0.6 2	0.6 3	0.6 4	0.6 3	0.6 4	0.6 3	0.6 3	0.6 5	0.6 9	0.7 3

The proposed metrics FBFM and FR are based on MI, a well-known maintenance metric. MI, in turn, is based on VG, LOC, among other popular metrics. A good correlation was found between the proposed

metrics and these industry standard metrics. To show the correlation, a chart was generated in Excel using values from Table 5.3. See Fig. 5.4.

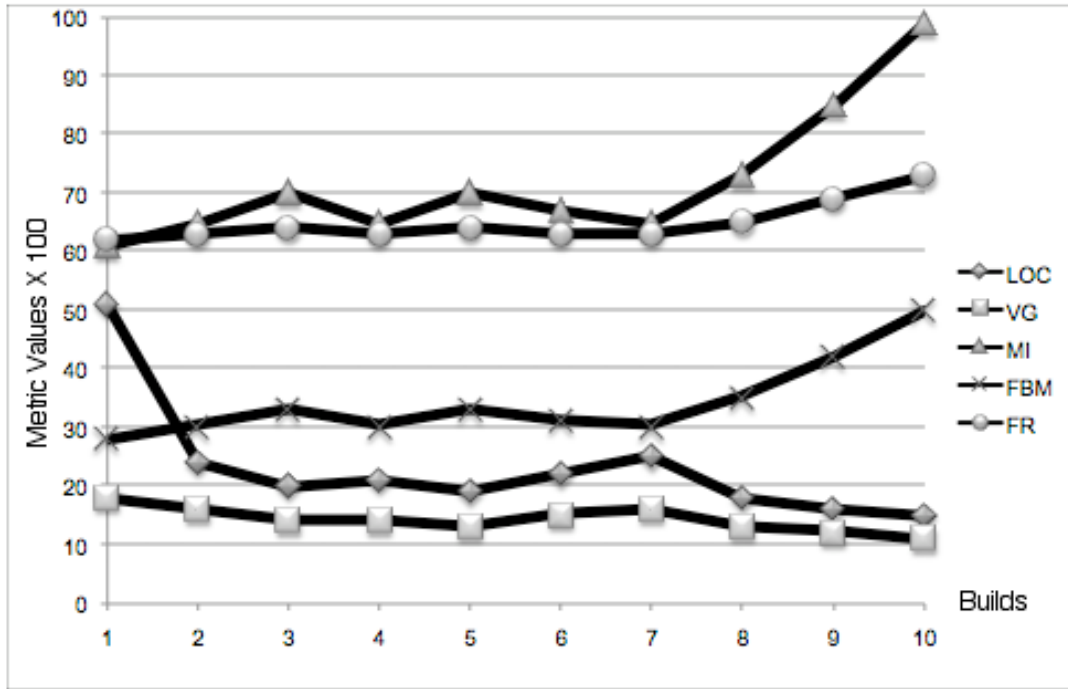


Fig. 5.4: Correlation of Proposed Metrics (FBFM and FR) With Common Metrics

The FBFM and FR values, in the figure, were normalized (multiplied by 100) to share the same chart and units with other metrics. As shown in the chart, the top two lines, which represent FR and MI, are positively correlated (reliability increases as maintainability index increases). The bottom three lines show FBFM negatively correlated with VG and LOC

(FBFM increases as complexity decreases). The third proposed metric, FM, is not shown in the figure because it's constant for the function under analysis in one given release cycle.

H7. The Best Maintenance Tools are the Ones Used by the Entire

Team. CMMR was designed for use by the entire team, and strong evidence suggests that it met the requirements of all the intended users. Maintenance is not a development-only activity. It involves testing, documentation, and management, among other things. In two of the case studies covered by this research, CMMR was the standard tool used by the team. Both team productivity and product quality increased, as a result. Actual result data was hard to quantify due to the limited time the tool was put into use. At minimum, CMMR needs a full release cycle of continuous use in order to retrieve data that can be quantified and compared against other release cycle. A potential for 25% reduction in maintenance cost is very likely. Higher savings are possible depending on the project and the level of adoption.

5.3 The New Process Model, Metrics, and CMMR Tool

Initial feedback from the case studies suggested that the new process model and the tool are indeed comprehensive enough to support all the various phases of the software maintenance process. They were

also easy to adopt and use by the development team: developers, testers, and project managers. Results from documentation writers are not available at this time due to scheduling conflicts.

The proposed process model was immediately accepted by the Mac Product team, for two reasons: first, it is very similar to what the company advocates; i.e. full program comprehension before making changes, identifying impact and regression as quickly as possible, etc. Second, it serves as a way to enforce the good habits and avoid the bad ones in order to increase team productivity and the quality of the product. One adjustment was suggested to remove the documentation phase from the process cycle and make it a post-delivery phase, or perhaps during beta testing phase. While this makes sense to this particular organization, the author decided to leave the process model as is to encourage documentation writers to work with the team during maintenance in order to have their work completed at the same time when the product features are completed. More research in this area is needed to see where in the maintenance cycle most software vendors prefer to have their documentation written.

Feedback on the new metrics were mixed mostly due to limited use. It's well known that metrics take years to fully develop and gain precision. Nevertheless, the metrics introduced in this research show signs that they will help reduce complexity, improve productivity, and increase quality and error injection. The proposed metrics were actually refined during the case studies on two fronts: first, adopting the maintainability index (MI) instead of McCabe cyclomatic complexity as a measure of complexity, and, second, computing the feature reliability by averaging the individual function reliability measurements, rather than taking the minimum. These two changes resulted in better assessment of complexity, with more complexity metrics taken into account, and reliability, with outlier functions becoming less significant.

The author expects other metric changes in computing Function Maturity (FM) in terms of internal builds, not just external releases. Another metric that may be refined is the Function Reliability (FR) where the function maturity and complexity factors are given equal weights. A better division may be to give more weight to complexity than to maturity. More case studies are needed to refine these two metrics.

The reliability trends were seen as a helpful tool to engineering managers to control complexity, as an indicator of the completeness and thoroughness of the features, and the readiness of the product for delivery to the next phase of software maintenance. Engineering managers requested more curves to be shown on the same chart, and be optionally turned on/off. This allows the user to see the impact of a change to a function on the reliability trend of the same function, the features that use the function, and the overall project - at the same time and on the same chart. An illustration of the requested feature planned for next release is shown in Fig. 5.5.

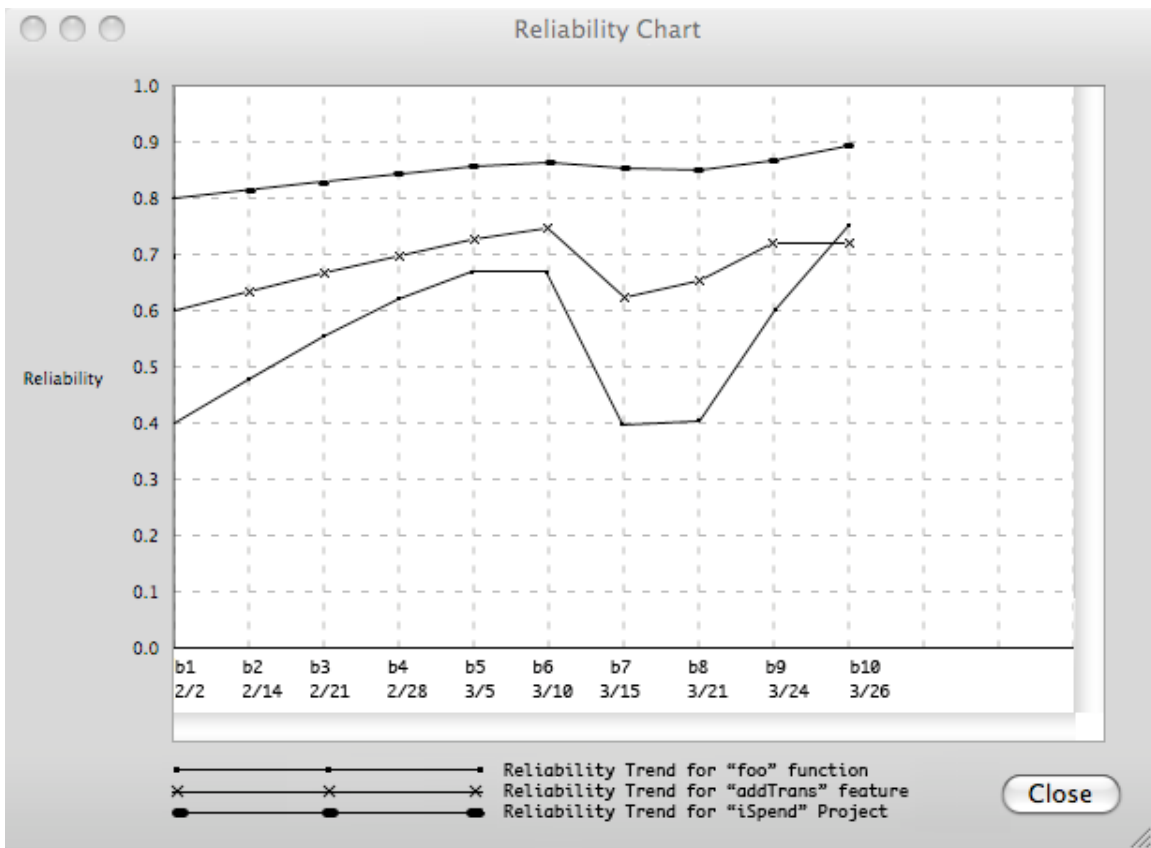


Fig. 5.5 – Multiple Reliability Curves on Same Chart.

One refinement to the process model was suggested: when a feature is under development, it is best to defer its CMMR analysis until the coding is complete and the unit testing has begun. The authors agree with this suggestion for that particular case study, and for certain major features that take weeks/months to develop and become ready for testing and maintenance. In general, all participants agreed that there would be a cost reduction in using the proposed model and tool, pending support for additional feature requests. However, they could not quantify the exact cost reduction, in terms of percentage of the overall estimated cost, without a full adoption of the process and tool for a full release cycle. This is a good likelihood for the next release of the Mac Product in 2009.

5.4 Acceptance of CMMR

Adopting CMMR as a new maintenance tool needs better planning and preparation so it can be scheduled in with the intended software organization. There is a natural tendency by some project owners to not accept process model changes or new tools, especially those that differ from the processes they follow and the tools they use. This was

not a concern for the Mac Product, but it is expected for other products, and must be dealt with as part of planning and preparation.

There was too much concern and sensitivity of some participants about the possibility of using the proposed metrics for performance evaluation and measurement. This issue is a valid one and will take time before the fear is completely eliminated. As mentioned earlier, a slow strategy is needed. One that involves a few gradual steps: start small, explain why, share the data, define data items and procedures, and understand trends. With the ultimate goal of creating a “measurement culture” that is willing to adopt and use the tool without any fear.

5.5 How the Main Research Claims Feared in Practice

This research made two claims that were put to the test during the case studies: first, the code complexity/maintainability of a function increases as more features use that function. The higher the feature-based function maintainability, the more likely the function will have defects, and the more maintenance is required to detect and fix these defects. The second claim: function maturity (its age and number of releases it has been in) matters when measuring reliability. All

participants agreed with these two claims based on actual experience and observations. These are facts that are seen in practice yet somehow neglected in research. The author believes that this research is the first one to point them out.

5.6 Code Parsing

The author confronted some difficulty finding open source code for computing popular metrics like McCabe and Halstead. The computations were therefore invented from scratch, which was not so trivial, and the results may not be optimal. A way around it would be to use a standard compiler for parsing the code, however most compilers are not “open” enough to allow for such customization. More investigation is needed in this area to insure accurate code analysis and calculations of metrics.

5.7 Multiple Languages and Platform Support

In terms of high-level languages and platform support, as usual, the more, the better. The first release of CMMR supported C/C++/ObjC/ObjC++ on the Macintosh, and Java on Windows. Obviously, there are other languages, platforms, and many combinations thereof. Obviously, there is a huge cost associated with implementing all of these combinations, however, there are current

discussions with some grant providers to get some financial support to do this development. Such new development will require adding more user options in CMMR's New Project window to allow the user to specify the language(s) of the target project, and the file extensions allowed for code analysis and search. Metrics computation may change as well especially for languages that are not derived from C.

In the first release of CMMR, two development projects were created and maintained for the Mac and Windows versions. Separating the development of the Mac version from the Windows version caused a feature disparity problem where one feature is implemented on one platform but not the other, or a feature is implemented slightly differently across the two platforms. Combining the two projects into one project with multiple targets is planned. This not only avoids the feature disparity problem, it also reduces future development and maintenance cost by avoiding redundant work and duplication of effort.

5.8 Actual Experimental Feedback

The following are some of the actual feedback from the engineers and managers who used CMMR during the maintenance of their Macintosh software project.

The Tech Lead had this to say prior to using the tool: “And I do think this is a valuable tool if it works as described”. He expressed concern over the tendency of some engineers to keep trying to prolong the life of legacy code rather than replacing it with something newer and better (i.e. re-engineering). In other words, he is more in favor of “development” rather than “maintenance” of old code. However, he agreed that in practice this is easier said than done. “That’s the main philosophical difference I have with your precepts. But even if one subscribes to that, the tool still seems very useful for tracking the changes. I’ll have to play with it more to get a real feel.”

One developers wrote: “With this tool, program understanding is easier, regression testing more focused, and project management better informed and their decisions more timely”. Another developer said: “I would like to give this a go on the SuperNav [feature]”. Another wrote: “I think that I am going to be working on the Dashboard [feature] next so maybe we can start fresh with CMMR on that”.

Testers were the most excited about the tool. One wrote: “First, let me say: Wow, this is an awesome tool. It looks like it could be really

useful. I really mean that too. I'd love to apply it to the automation code I'm building... As a QA person, I'm always interested in the downstream effects of code changes. Sometimes the Engineers change TONS of files between builds and there is no easy way to tell what those code changes could have affected. If we had CMMR running on the entire project it would make regression ever so much more focused." He continued: "the ability to see the code, the comments, and the metrics of any function is invaluable. Knowing that the data is available is the important part. Can we put the import feature into CMMR? It'd be interesting to use CMMR in a live environment between several folks."

Product Management had some positive remarks as well: "I think this would help us identify risks and also pinpoint areas where we would need additional resources or to provide coaching for a particular engineer". On the Metric window, the manager said: "It helps me understand quickly the level of completeness, thoroughness and risk tied to a given feature. The culmination of all these details for functions... would help me understand release completeness and risk."

Another product manager had this to say: "It would help me understand

how existing or new features are being built, allowing me to better follow engineering conversations... At least initially, I would use this more to educate myself than to help make decisions about resourcing or the staging of work.”

5.9 Summary

The research hypotheses were proven experimentally in this chapter. Some results were subjective while others were objective. Some data was easily obtainable for some areas of the work, such as number of defects, and various other metrics and trends. Other data was hard to get due to inherent limitations in the target case studies. When dealing with open-source case studies, the limitations had to do with the nature of the target project itself having no owner, or history data, or a maintenance team to work on the project in a given maintenance cycle. In the case of the commercial Mac Product, the limitations had to do with the limited time spent on the project, and the inability to expose the product’s source code and other company-confidential data to the public. Nevertheless, some result data was obtained, and more is desired and planned in a future study.

The feedback from the people involved in these case studies was very positive and encouraging. Their feedback was based on comparing the results of the proposed methodology against not using it; i.e. doing things in their own ad-hoc ways. As mentioned earlier, there are no other tools in the market today that the methodology introduced here can be fully compared against. Some small comparisons can be made nevertheless. One might be the ease of use of CMMR vs. other program comprehension tools. Another is being the only tool with the ability to move to a remote site away from the target project source code without missing a single benefit. Another is the automatic code change detection and the new regression testing method that is based on it.

CHAPTER 6

CONCLUSIONS & SUGGESTIONS FOR FUTURE WORK

This research addresses major cost factors of software maintenance, simultaneously, by introducing a tool-centric process model. The tool, CMMR, is feature-based and easy to use by the entire team in many areas of software maintenance, including program comprehension, change impact analysis, and regression testing. It offers graphical representations of the program features based on feature execution trace data. It has built-in metrics that help the team measure complexity, maintainability, maturity, and reliability of functions, features, and the overall project. It computes these metrics by parsing the code base and breaking it up into basic tokens (operators, operands, keywords, branch statements, etc.). CMMR was used in several case studies and the overall results suggest a significant cost reduction in software maintenance due to: (1) faster program comprehension, (2) more precise change impact analysis, (3) more focused regression testing, (4) better and more timely decision at the project management level, (5) faster defect detection and recovery, and (6) lower defect injection.

Together, the new process model, the CMMR tool, and the built-in metrics promise to help software organizations maintain their software projects more effectively and efficiently. The entire maintenance team: developers, testers, writers, and managers will benefit from the two core features it provides: first, the more natural feature-based organization of the code base; second, the tree representation of each feature allowing for faster and more accurate understanding of each feature, as shown in some of the case study results in Chapter 5.

Beyond the above main benefits, each class of users will find additional value in the tool tailored for their unique requirements. Developers will benefit greatly in the areas of program comprehension, change impact analysis, and complexity measurements. Testers will appreciate the automatic detection of changed functions and the new regression testing selection method that is based on it. Management will find the metric and reliability chart windows very valuable in managing risk associated with code complexity. Documentation writers will also benefit from the comments window to understand some details about the target features intended for documentation. Better communication among the entire team is a side benefit from this model. The combined

efficiency in coding, testing, documentation, management, and interpersonal communication leads to lower overall maintenance cost.

The CMMR tool differs from other maintenance tools in that it was designed from the ground up to be easy to use by technical and non-technical users. It's the only tool that targets the entire maintenance team and addresses the entire set of maintenance cost factors at once. In that regard, CMMR is in a class by itself, and is therefore hard to compare against other tools that may be specific to one class of users (i.e. developers) or a single cost factor (i.e. program comprehension).

The tool was demonstrated at the WorldComp '08 conference in Las Vegas, and was very well received by practitioners as well as scientists. Some of the audience estimated the potential cost reduction to be in the range of 30-35%. But that percentage was subjective and based on their specific project and the team working on it. In general, the cost reduction percentage obtained from this method or any other similar method is based on several conditions that vary from one software system to another. Any such figure is obtained through experiments and more case studies. There is no mathematical proof of

any percentage figure or claim.

Suggestions for Future Work

It's important to note that the tool development still works in progress with many features and enhancements deferred for future research.

Most of the planned future work lies in the functionality of the CMMR tool and the built-in metrics. The following is a list of suggestions of future work.

1. More comparison of results between the methodology proposed here and other existing methods. This study would require finding multi-purpose tools similar to CMMR (none exists as of this writing), and applying these tools on similar case studies and comparing their results with CMMR. Other requirements for such study include: full maintenance cycle, full access to the target project's source code, and full commitment of the maintenance team. Not a trivial study in terms of time and cost, but these requirements are essential for obtaining realistic comparison data.
2. Better error tracking per feature, and per function, within each project build, and across multiple builds.
3. Additional support related to engineers, such as specialties and assigned tasks.

4. More automation in the areas of code trace generation, code/comment parsing, log analysis, tree pruning, finding relocated functions, and detecting new builds.
5. Better trace handling in multithreaded applications. When running a feature to record its function names into a log file, it's required that no other feature be running concurrently in other threads inside the application. Multithreading causes mixing of function names belonging in different threads into one log file, which poses problems for the tool's code analysis. A future version of the tool can solve this problem by adding more intelligence to recognize the feature thread(s) and analyzing only the functions that belong to the feature threads. Another challenge is the ability to detect and remove redundant sub-graphs generated from code segments with recursion and loops.
6. More refinement of the five metrics introduced. Such refinement requires using the tool on several representative projects during a full maintenance cycle from start to finish. In the case studies presented here, insufficient time was given to retrieve data from a full maintenance cycle, and such time-based data is crucial to the

refinement of the time-dependent metrics and reliability models proposed. Of course, this goal requires prior arrangements with the companies that own the projects. The owners must commit to adopting the process model and using the CMMR tool in their maintenance workflow for a full release cycle, or even two, if possible.

7. An additional refinement of the metrics involves better and more accurate parsing. The author found some difficulty finding open-source parsers to use for standard complexity metrics (like McCabe and Halstead), so a lot of that work was invented internally at the risk of losing accuracy and not being standard. More work is needed on the tool to bring up the code parsing capabilities to a level comparable with compiler-based lexical analyzers.

8. Finally, the CMMR tool was developed on two platforms: Windows and Macintosh. Both versions need more work to bring them in sync together in terms of feature parity. Both versions could use more graphical support for large features with hundreds and thousands of nodes, such as zoom-in and zoom-out capabilities, multiple selection

alignment, grids, etc. Viewing multiple feature trees at once in 3-D is another area that needs to be explored. Such support promises to make the tool more in line with what the user expects from a graphical application. This in turn will facilitate editing, and provide further aid in program visualization and comprehension, tracking changes, and managing complexity.

REFERENCES

- [1] Abran, A., Silva, I., Primera, L. "Field studies using functional size measurement in building estimation models for software maintenance", *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1), 2002, pp. 31–64.
- [2] Apiwattanapong, T., Orso, A., Harrold, M. J., "Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences", *ICSE '05*, May 2005.
- [3] Ball, T., Eick, S. G., "Software Visualization in the Large." *Computer*, Volume 29, Issue 4, April 1996, pp. 33-43.
- [4] Bennett, K. H., Younger, E. J., "Model-Based Tools to Record Program Understanding", *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Napoli, Italy, IEEE Computer Society Press, 1993, pp. 87-95.
- [5] Bohner, S., Arnold, R., "Software Change Impact Analysis", IEEE Computer Society Press, 1996.
- [6] Bohnet, J., Dollner J., "Analyzing dynamic call graphs enhanced with program state information for feature location and understanding", *International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 915-916.
- [7] Bohnet, J., Dollner, J., "Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems", *SOFTVIS 2006*, Brighton, United Kingdom, 2006.
- [8] Burd, E., Munro, M., "An initial approach towards measuring and characterizing software evolution", *Proceedings of the Working Conference on Reverse Engineering*, WCRE '99, 1999, pp. 168–174.

- [9] Canfora, G., Cimitile, A., "Software Maintenance", November 2000. Retrieved August 16th, 2008 from:
<http://citeseer.ist.psu.edu/cache/papers/cs/25307/ftp:zSzzSzcs.pitt.eduzSzchangzSzhandbookzSz02.pdf/software-maintenance.pdf>
- [10] Capers Jones, "Patterns of Software System Failure and Success", International Thomson Computer Press, Boston, MA, 1995.
- [11] Capers Jones, "Software Productivity Research, Inc", Burlington, MA, Proceedings of the 2006 international workshop on Software quality, 2006.
- [12] Capers Jones, "Software Quality – Analysis and Guidelines for Success", International Thomson Computer Press, Boston, MA, 1997.
- [13] Chen, K., Rajich, V., "RIPPLES: tool for change in legacy software", Proceedings of the IEEE Int'l Conference on Software Maintenance, 2001, pp. 230-239.
- [14] Chen, Y., "Specification-based Regression Testing Measurement with Risk Analysis", Masters Thesis, University of Ottawa, Canada, 2002.
- [15] Detienne, F., "Software Design – Cognitive Aspects", Springer-Verlag London, Ltd., 2002.
- [16] Elbaum, S., Munson, J., "Evaluating Regression Test Suites Based on Their Fault Exposure Capability", Journal of Software Maintenance, Volume 12, Issue 3, 2000, pp. 171-184.
- [17] Feng, L., Maletic, J. I., Marcus, A., "Comprehension of Software Analysis Data Using 3D Visualization", Proceedings of the IEEE Int'l Workshop on Program Comprehension, 2003, pp. 105-114.

- [18] Fenton, N., Pfleeger, S.L., "Software Metrics: A Rigorous and Practical Approach", second ed. International Thomson Computer Press, London, UK, 1996.
- [19] Fuggetta, A., "Software Process: A Roadmap", Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, June 04-11, 2000, pp. 25-34.
- [20] Good, J., "Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension", Ph.D. Thesis, University of Edinburgh, 1999.
- [21] Halstead, M. H. "Elements of Software Science, Operating, and Programming", Systems Series Volume 7. New York, NY: Elsevier, 1977.
- [22] Harrison, M. S., Walton, G. H., "Identifying high maintenance legacy software", Journal of Software Maintenance and Evolution: Research and Practice, 14(6), 2002, pp. 429–446.
- [23] Harrold, M., Rothermel, G., "Aristotle, A system for research on and development of program analysis based tools", Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, 1997.
- [24] IEEE Std. 610.12, "IEEE Standard Glossary of Software Engineering Terminology 610.12-1990". In IEEE Standards Software Engineering, 1999 Edition, Volume One: Customer and Terminology Standards. IEEE Press, 1999.
- [25] Inoue, S., Yamada, S., "Discrete Program-Size Dependent Software Reliability Assessment: Modeling, Estimation, and Goodness-of-Fit Comparisons", December 2007.
- [26] Irwin, W., Churcher, N., "Object oriented metrics: Precision tools and configurable visualizations", Proceedings of the IEEE Symposium on Software Metrics, 2003, pp. 112-123.

- [27] ISO/IEC 12207, "Information Technology – Software Life Cycle Processes", Geneva, Switzerland, 1995.
- [28] Jiang, M., Zhang, J., Simmons, J., Edwards, D., Wilde, N., "TraceGraph 4: A Demonstration Case Study", SERC-TR-290, Software Engineering Research Center, July 2007.
- [29] Jones, J. A., Harrold, M. J., Stasko J., "Visualization of Test Information to Assist Fault Localization", Proceedings of the IEEE Int'l Conference on Software Engineering, 2002, pp. 467-477.
- [30] Khoury, M., "Cost-Effective Regression Testing", Seminar on Software Testing, Department of Computer Science, University of Helsinki, Autumn 2006. Retrieved August 16th, 2008 from: http://www.cs.helsinki.fi/u/khoury/st/cert_MaruanKhoury.pdf
- [31] Kiran, G. A. , Haripriya, S., Jalote, P., "Effect of object orientation on maintainability of software", Proceedings International Conference on Software Maintenance. IEEE Computer Society Press: Los Alamitos CA, 1997, pp.114–121.
- [32] Krishnan, M. S., Mukhopadhyay, t., Charles, H., Kriebel, "A Decision Model for Software Maintenance", Information Systems Research Vol. 15, No. 4, 2004, pp. 396–412
- [33] Law, J., Rothermel, G., "Incremental dynamic impact analysis for evolving software systems", IEEE Int. Symp. on Soft. Reliability Eng., 2003.
- [34] Lee, M. L., "Change Impact Analysis of Object-Oriented Software", ISE-TR-99-06, George Mason University, May 1999.
- [35] Lehman, M. M., Perry, D. E., Rami L, J. F., "Implications of evolution metrics on software maintenance", Proceedings of the International Conference on Software Maintenance, 1998, pp. 208–217.

- [36] Lehman, M. M., Rami L, J. F., Wernick, P. D., Turski, W. M., "Metrics and laws of software evolution—the nineties view", Proceedings of the 4th International Software Metrics Symposium, IEEE Computer Society Press, 1997, pp. 20.
- [37] Lemieux, F., Salois, M., "Visualization Techniques for Program Comprehension", Frontiers in Artificial Intelligence and Applications, Vol. 147, pp. 22-47, 2006.
- [38] Lewerentz, C., Simon, F., "Metrics-Based 3D - Visualization of Large Object-Oriented Programs". Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2002, pp. 70-77.
- [39] Li, W., Henry, S., "An Empirical Study of Maintenance Activities in Two Object-oriented Systems," Journal of Software Maintenance, Research and Practice, Volume 7, No. 2, 1995, pp. 131-147.
- [40] Marciniak, J., "Encyclopedia of Software Engineering", New York, NY, John Wiley & Sons, 1994, pp. 131-165.
- [41] Marcus, A., Rajlich, V., "Identification of Concepts, Features, and Concerns in Source Code", Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM2005), Budapest, Hungary, September 25-30, 2005, pp. 718-718.
- [42] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2 No. 4, 1976, pp. 308-320.
- [43] McCabe, T. J., Butler, C. W., "Design Complexity Measurement and Testing", Communications of the ACM 32, 12, December 1989, pp. 1415-1425.
- [44] McCabe, T. J., Watson, A. H., "Software Complexity." Crosstalk, Journal of Defense Software Engineering 7, 12, December 1994, pp. 5-9.
- [45] Musa, J., "Software Reliability Engineering", McGraw-Hill, 1998, pp. 15.

- [46] Oman, P., Hagemester, J. "Construction and Validation of Polynomials for Predicting Software Maintainability (92-01TR)". Moscow, ID: Software Engineering Test Lab, University of Idaho, 1992.
- [47] Pan, J., "Software Reliability", Carnegie Mellon University, 18-849b Dependable Embedded Systems, Spring 1999.
- [48] Pigoski, T. M., "Practical Software Maintenance – Best Practices for Managing Your Software Investment", John Wiley & Sons, New York, NY, 1997.
- [49] Pinzger, M., Fisher M., Lanza M., "Visualizing Multiple Evolution Metrics", ACM, June 2005, pp. 67-75.
- [50] Pressman, R. S., "Software Engineering – A Practitioner's Approach", McGraw-Hill, New York, NY, 2001.
- [51] Reiss, S. P., "Bee/Hive: A Software Visualization Back End", Proceedings of ICSE Workshop on Software Visualization, 2001, pp. 44-48.
- [52] Robillard, M., Murphy, G., "FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code", Proceedings of the 25th International Conference on Software Engineering, May 2003, pp. 822-823.
- [53] Rohatgi, A., Lhadj, H., Rilling, J., "Feature Location based on Impact Analysis", Proceeding of Software Engineering and Applications, 2007.
- [54] Rothermel, G., Harrold, M. J., "Analyzing Regression Test Selection Techniques", IEEE Transactions on Software Engineering, Vol. 22, No. 8, 1996, pp 529–551.
- [55] Ryder, B. G., "Helping Programmers Debug Code Using Semantic Change Impact Analysis", Rutgers Prolang, 2006.

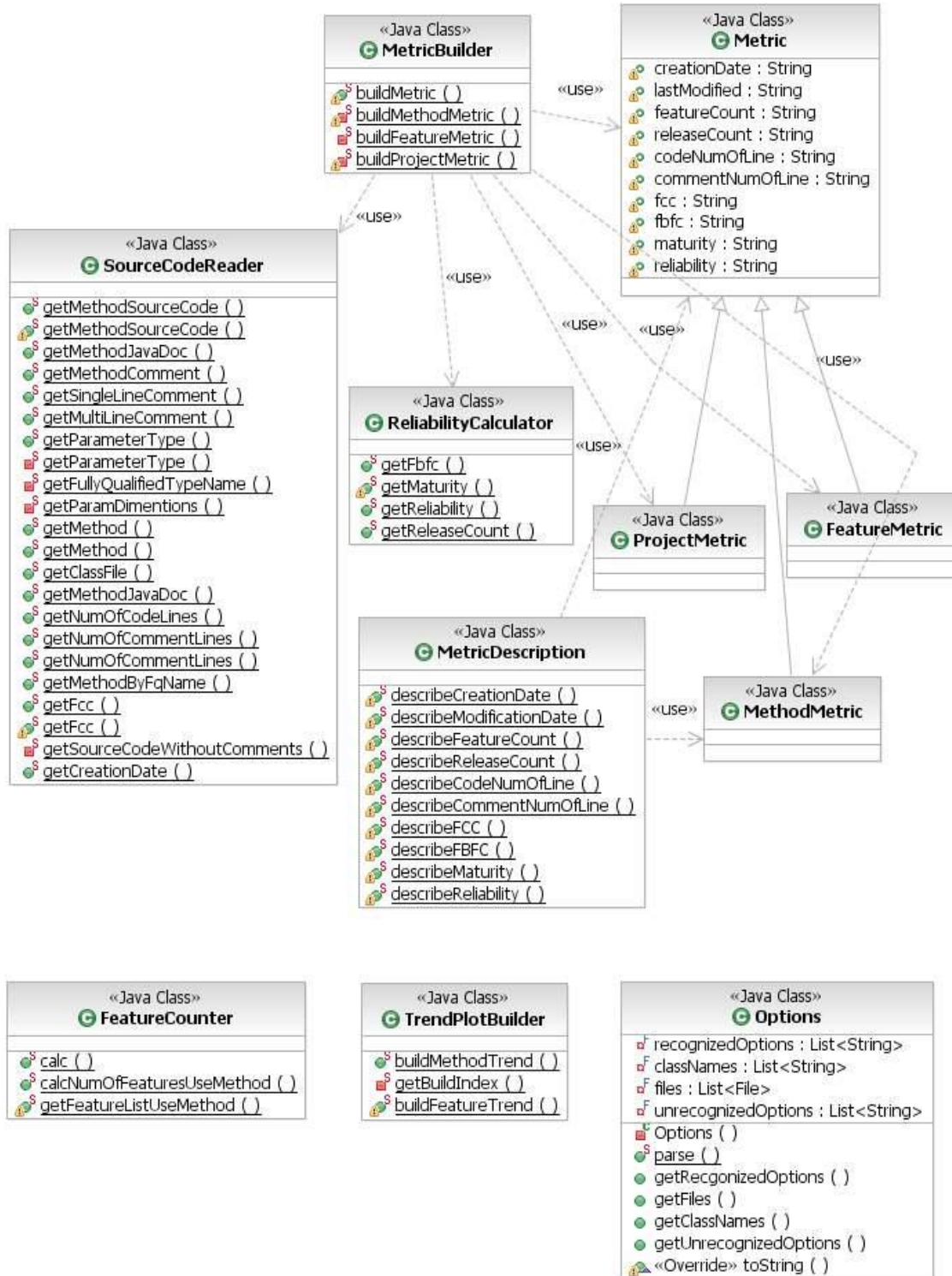
- [56] Ryder, B. G., Tip, F., “Change impact analysis for object-oriented programs”, Program Analysis for Software Tools and Engineering, 2001.
<http://www.prolangs.rutgers.edu/refs/docs/paste01.pdf>
- [57] Sharafat, A. R., Tahavildart, L., “Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach”, Journal of Software, Vol. 3, NO. 5, May 2008.
- [58] Shepperd, M., “A Critique of Cyclomatic Complexity as a Software Metric”; Software Engineering Journal, Vol. 3, 1988, pp. 30-36.
- [59] Sillito, J., Wynn, E., “The social context of software maintenance,” *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on Software Maintenance*, pp. 325–334, Oct. 2007.
- [60] Software Technology Review, “Maintainability index technique for measuring program maintainability”, SEI, www.sei.cmu.edu/str/descriptions/mitmpm_body.html, Last visited: July 2008.
- [61] Storey, M. A., Bennett, C. R., Bull, I., German, D. M., “Remixing Visualization to Support Collaboration in Software Maintenance”, Department of Computer Science, University of Victoria, May 2008.
- [62] Swanson, E. B., Beath, C. M., “Maintaining Information Systems in Organizations”, John Wiley & Sons, 1989.
- [63] Tilley, S. R., Smith, D. B., “Coming Attractions in Program Understanding”, Technical Report CMU/SEI-96-TR-019 ESC-TR-96-019, December 1996.
- [64] Wehrich, H., “Management: Science, Theory, and Practice”, Software Engineering Project Management, Second Edition, Thayer, R. H., ed., IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 4-13.

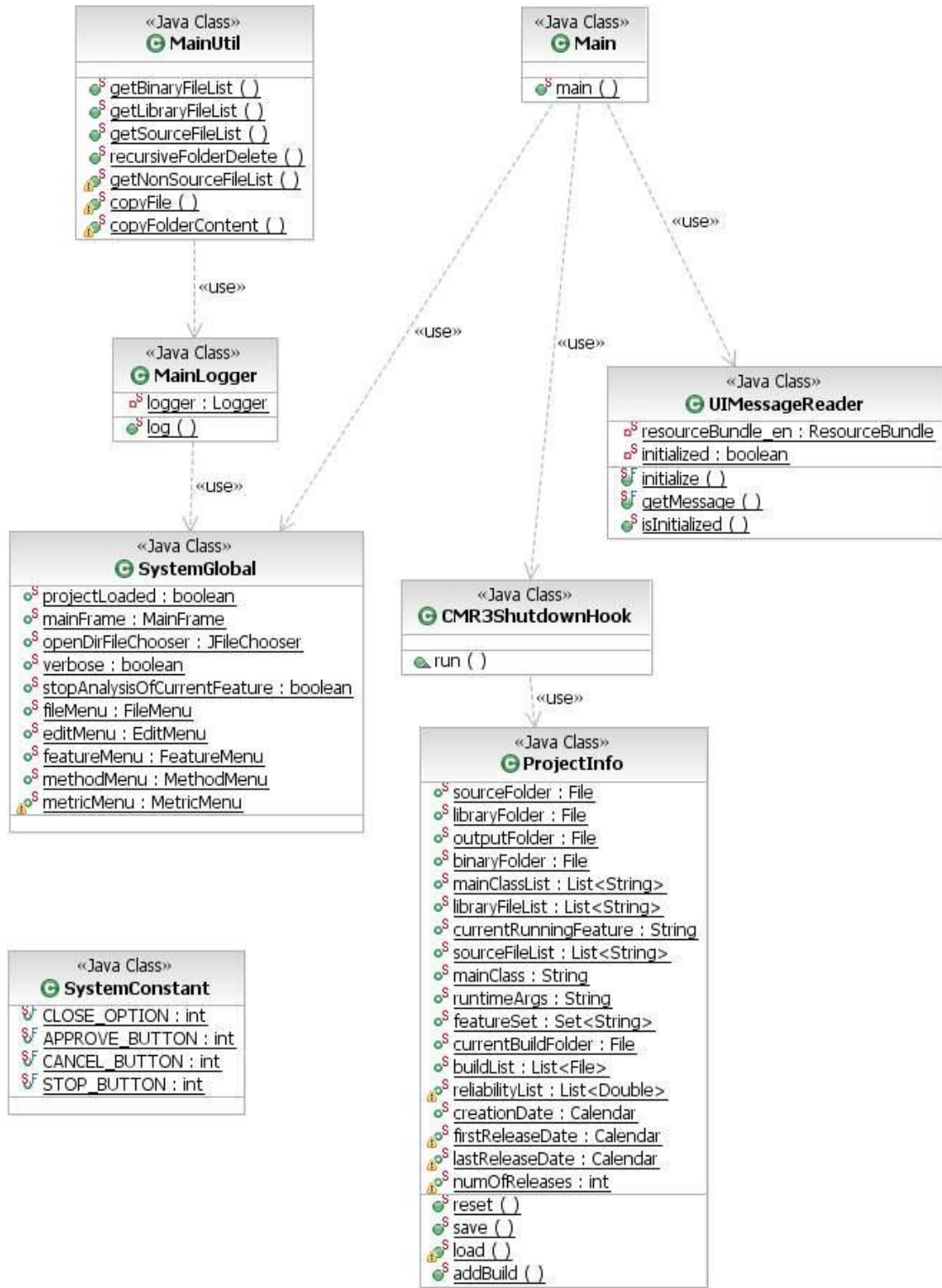
- [65] Welker, K. D., Oman, P. W., "Software Maintainability Metrics Models in Practice." Crosstalk, Journal of Defense Software Engineering 8, 11, November/December 1995, pp. 19-23.
- [66] Wieger, K. E., "A Software Metrics Primer", Software Development, 7(7), 2005, pp. 39–42.
- [67] Wilde, N., Buckellew, M., Page, H., Rajlich, V., Pounds, L., "A comparison of methods for locating features in legacy software", The Journal of Systems and Software, vol. 65, 2003, pp. 105-114.
- [68] Wilde, N., Matthews, P., Huitt, R., "Maintaining object-oriented software", IEEE Software; 10(1), 1993, pp. 75–80.
- [69] Wilde, N., Scully, M., "Software reconnaissance: Mapping program features to code" Journal of Software Maintenance: Research and Practice, Vol. 7, 1995, pp. 49–62.
- [70] Wong, W. E., Gokhale, S., "Static and dynamic distance metrics for feature-based code analysis", Journal of Systems and Software, Volume 74, Issue 3, February 2005, pp. 283-295.

APPENDICES

Appendix A CLASS DIAGRAM OF CMMR FOR WINDOW





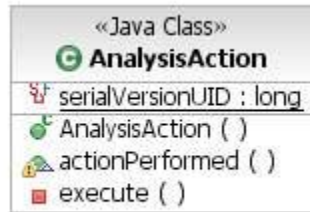


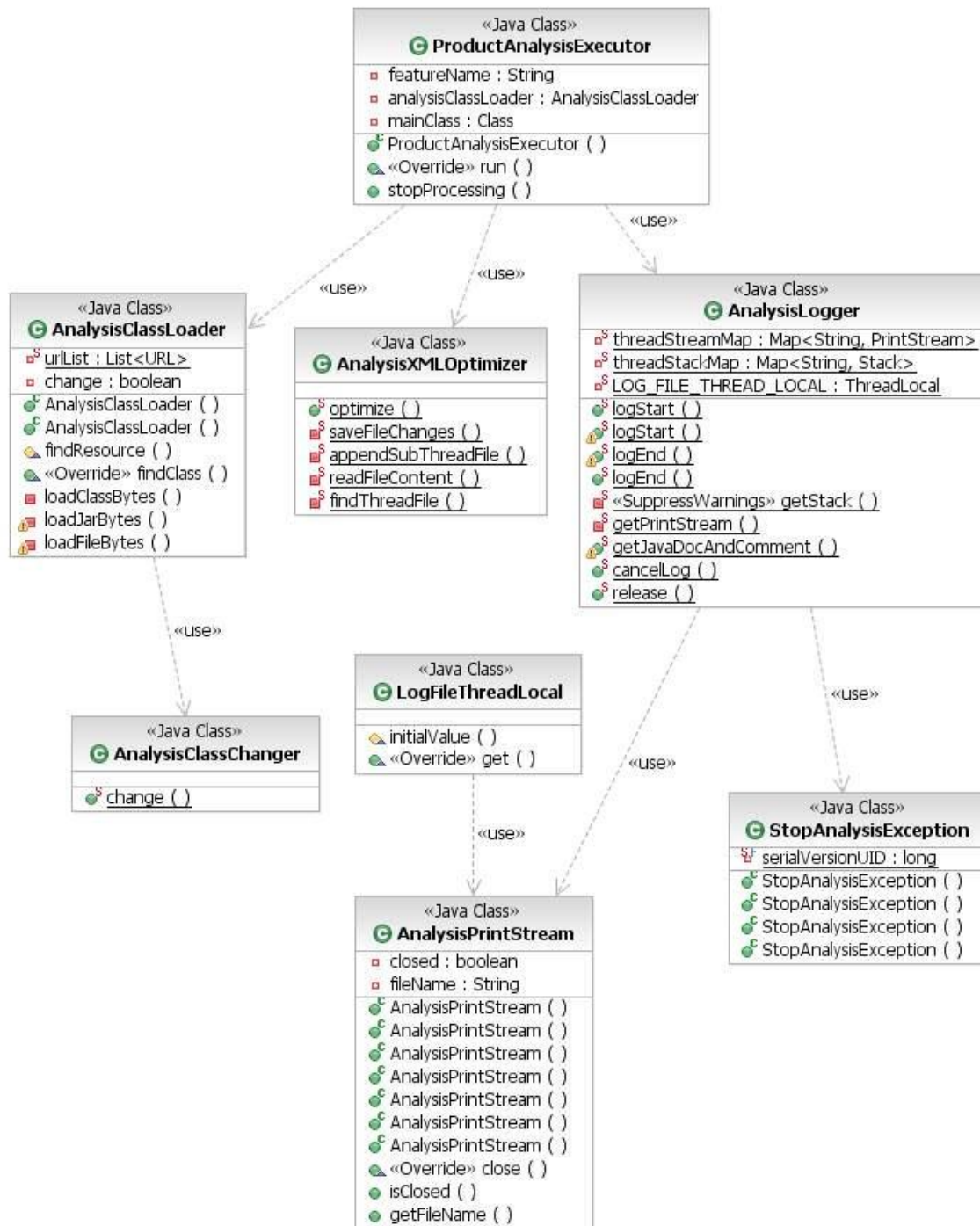
«Java Class»	
TrendDialog	
▢	title : String
▢	xAxisTitle : String
▢	yAxisTitle : String
▢	trendPointList : List<TrendPoint>
🌱	TrendDialog ()
▢	initialize ()
▢	getChartPanel ()
🔧	createDataSet ()

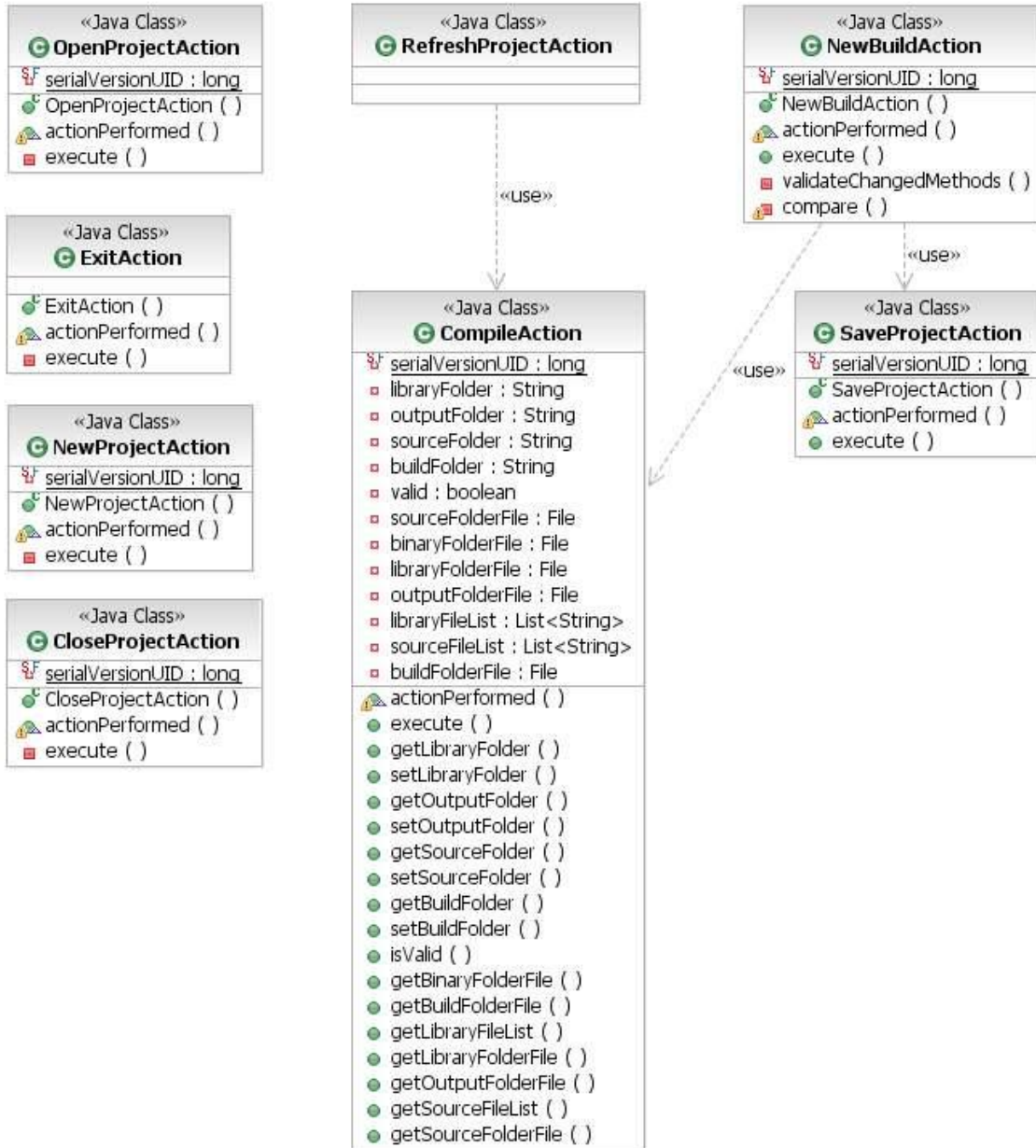
«Java Class»	
AnalysisDialog	
🔧	serialVersionUID : long
▲	executor : ProductAnalysisExecutor
▲	featureTextField : JTextField
▲	returnValue : int
▲	featureName : String
🌱	AnalysisDialog ()
▢	initialize ()
▢	createMainPanel ()
🔧	createControlPanel ()
🌱	getFeature ()
🌱	getReturnValue ()

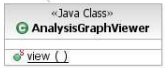
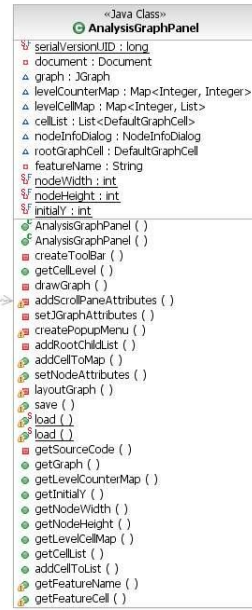
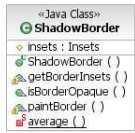
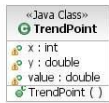
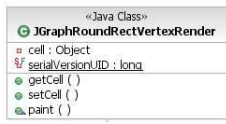
«Java Class»	
NewProjectDialog	
🔧	serialVersionUID : long
▢	classLoader : AnalysisClassLoader
▲	sourceFolderTextField : JTextField
▲	libraryFolderTextField : JTextField
▲	outputFolderTextField : JTextField
▲	numofreleasesTextField : JFormattedTextField
▲	creationDateTextField : JDateChooser
▲	firstReleaseDateTextField : JDateChooser
▲	lastReleaseDateTextField : JDateChooser
▲	mainClassComboBox : JComboBox
▲	runtimeArgsField : JTextField
▲	returnValue : int
▲	sourceFolder : String
▲	libraryFolder : String
▲	outputFolder : String
▲	mainClass : String
▲	runtimeArgs : String
🌱	NewProjectDialog ()
▢	initialize ()
▢	createCard1MainPanel ()
🔧	createCard1ControlPanel ()
▢	createCard2MainPanel ()
🔧	createCard2ControlPanel ()
▲	moveToCard2 ()
🌱	getLibraryFolder ()
🌱	getOutputFolder ()
🌱	getSourceFolder ()
🌱	getReturnValue ()

«Java Class»	
NewBuildDialog	
▲	versionNumTextField : JTextField
▲	versionNameTextField : JTextField
▲	returnValue : int
🌱	NewBuildDialog ()
▢	initialize ()
▢	createMainPanel ()
🔧	createControlPanel ()
🔧	getReturnValue ()
🌱	getVersionName ()
🔧	getVersionNum ()











«use»



«Java Class» FileMenu
<ul style="list-style-type: none"> serialVersionUID : long newProjectAction : Action openProjectAction : Action saveProjectAction : Action newBuildAction : Action refreshProjectAction : Action closeProjectAction : Action exitAction : Action
<ul style="list-style-type: none"> FileMenu () initialize ()

«Java Class» EditMenu
<ul style="list-style-type: none"> EditMenu () initialize ()

«Java Class» FeatureMenu
<ul style="list-style-type: none"> analysisAction : Action FeatureMenu () initialize ()

«Java Class» MethodMenu
<ul style="list-style-type: none"> addMethodAction : AddMethodAction deleteMethodAction : DeleteMethodAction
<ul style="list-style-type: none"> MethodMenu () initialize () changeCurrentGraph ()

«Java Class» MetricMenu
<ul style="list-style-type: none"> MetricMenu () initialize ()

«Java Class» ViewerUtil
<ul style="list-style-type: none"> centralizeDialog () centralizeFrame () createDirBrowseButton ()

«Java Class» StringUtil
<ul style="list-style-type: none"> rightP ()

APPENDIX B

HEADER FILES OF CMMR FOR MACINTOSH

CMMRAppDelegate.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
*/  
  
#pragma once  
  
@class DaughterWindowsController;  
  
@interface CMMRAppDelegate : NSObject  
{  
    // private data  
    @private  
    NSString*          deleteFeatureTemplate;  
    NSString*          updateFeatureTemplate;  
    NSString*          deleteFunctionTemplate;  
    NSString*          reliabilityFeatureTemplate;  
    NSString*          reliabilityFunctionTemplate;  
    NSMutableArray*    registeredDaughterWindowControllers;  
}  
  
// accessors / mutators  
@property (retain) NSString* deleteFeatureTemplate;  
@property (retain) NSString* updateFeatureTemplate;  
@property (retain) NSString* deleteFunctionTemplate;  
@property (retain) NSString* reliabilityFeatureTemplate;  
@property (retain) NSString* reliabilityFunctionTemplate;  
  
// public methods  
- (void) registerDaughterWindow:  
    (DaughterWindowsController*) inController;  
- (void) unregisterDaughterWindow:
```

```
(DaughterWindowsController*) inController;
- (BOOL) processEvent: (NSEvent*) inEvent;

@end
/*
  CMMRApplication.h

  Copyright (c) ASI
  March 2008
  Abdallah Qaisi

*/

#pragma once

@interface CMMRApplication : NSApplication {

    // private data
    @private

}

@end
```

CMMRDocument.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

*/

```
#import <Cocoa/Cocoa.h>
```

```
@class FDGraph;
```

```
@class FDGraphView;
```

```
@class FDNode;
```

```
@interface CMMRDocument : NSDocument
```

```
{
```

```
    FDGraph *graph;
```

```
    IBOutlet FDGraphView *graphView;
```

```
    IBOutlet NSTextField*    labelTextField;
```

```
    IBOutlet NSPopUpButton*  selectedFeature;
```

```
    @private
```

```
        NSMutableArray*      featuresArray;
```

```
        NSArrayController*   featuresArrayController;
```

```
        BOOL                  updatingCurrentFeature;
```

```
}
```

```
// creation
```

```
+ (NSError*) createCMMRProjectNamed: (NSString*) inName  
                        atLocation: (NSURL*)
```

```
inProjectLocation
```

```
forSourcesAt: (NSURL*)
```

```
inSourcesLocation
```

```
forExecutableAt: (NSURL*)
```

```
inExecutableLocation
```

```
projCreateDate: (NSString*)
```

```
createDate
```

```
projFirstRelDate: (NSString*)
```

firstRelDate
lastRelDate
numReleases;

projLastRelDate: (NSString*)
projNumRelease: (NSString*)

// accessors / mutators

- (FDGraph*) graph;
- (void) setGraph: (FDGraph*) value;

- (FDGraphView*) graphView;
- (void) setGraphView: (FDGraphView*) value;

- (NSPopUpButton*) selectedFeature;
- (void) setSelectedFeature: (NSPopUpButton*) value;

- (NSArrayController*) featuresArrayController;
- (void) setFeaturesArrayController: (NSArrayController*) value;

- (BOOL) updatingCurrentFeature;
- (void) setUpdatingCurrentFeature: (BOOL) value;

- // public methods
- (void) addFeatureNamed: (NSString*) inNewFeatureName
forFile: (NSURL*) inFeatureFileURL;

- (void) addFunctionNamed: (NSString*) inNewFunctionName
functionPath: (NSURL*) inFunctionPath;

- (FDNode*) currentFeatureNode;

- (NSString*) targetAppPath;
- (NSString*) targetAppPathLeafName;
- (NSString*) targetAppSourcePath;

- (NSString*) projectCreationDate;
- (NSString*) projectFirstReleaseDate;
- (NSString*) projectLastReleaseDate;
- (NSString*) projectNumReleases;

```

- (NSString*) currentBuildFolderPath;
- (NSDate*) previousBuildFolderDate;

// Features menu methods
- (IBAction)showNewFeatureSheet:(id)sender;
- (IBAction) selectedFeatureDidChange: (id) sender;
- (IBAction) deleteCurrentFeature: (id) sender;
- (IBAction) updateCurrentFeature: (id) sender;

// Functions menu methods
- (IBAction)showNewFunctionSheet:(id)sender;
- (IBAction)addFunctionNode:(id)sender;
- (IBAction)dismissNewFunctionSheet:(id)sender;
- (IBAction)deleteFunctionNode:(id)sender;
- (IBAction)changeView:(id)sender;

// Metric menu methods
- (IBAction) showNodeReliabilityTrend: (id) sender;
- (IBAction) showProjectCurrentBuildMetrics: (id) sender;
- (FDNode*) fetchSameNodeFromPreviousBuild: (FDNode *)
nodeLabel;
- (void) drawChartFromReliabilityArray: (FDNode *) functionNode
fromArray: (NSMutableArray *)functionReliabilityArray
forProjectName: (NSString *) projectName;

- (IBAction)inspectSelectedObject:(id)sender;

- (NSArray*) featuresArray;
- (unsigned) countOfFeaturesArray;
- (id) objectInFeaturesArrayAtIndex: (unsigned) theIndex;
- (void) getFeaturesArray: (id*) objSPtr range: (NSRange) range;
- (void) insertObject: (id) obj
inFeaturesArrayAtIndex: (unsigned) theIndex;
- (void) removeObjectFromFeaturesArrayAtIndex: (unsigned)
theIndex;
- (void) replaceObjectInFeaturesArrayAtIndex: (unsigned)
theIndex

```

```

        withObject: (id) obj;
- (NSInteger) insertWithSortIntoFeaturesArray: (FDNode*) obj;

- (void) handleNodeDoubleClick: (NSEvent*) inEvent;
- (void) handleNodeCommandClick: (NSEvent*) inEvent;
- (void) handleNodeControlClick: (NSEvent*) inEvent;
- (void) handleNodeOptionClick: (NSEvent*) inEvent;

- (NSString*) nameForNewBuildFolder;
- (NSMutableArray*) pastBuildFolders;
- (NSString*) projectFileLeafName;

```

```
@end
```

```
/*
```

```
    CMMRDocumentWindowController.h
```

```
    Copyright (c) ASI
```

```
    March 2008
```

```
    Abdallah Qaisi
```

```
#pragma once
```

```
// forward declarations
```

```
@class FDGraphView;
```

```
@class CMMRDocument;
```

```
@interface CMMRDocumentWindowController : NSObject {
```

```
    // outlets
```

```
    IBOutlet FDGraphView*
```

```
    graphView;
```

```
    IBOutlet NSWindowController*
```

```
    windowController;
```

```
    IBOutlet NSPopUpButton*
```

```
    selectedFeaturePopup;
```

```
    IBOutlet NSArrayController*
```

```
    featuresArrayController;
```

```
    // private data
```

```
    @private
```

```
}
```

```
@end
```


CMMRLog.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

*/

```
#define CMMRLogString(s,...) [CMMRLog logFile:__FILE__  
lineNumber:__LINE__ format:(s),##__VA_ARGS__]
```

```
@interface CMMRLog : NSObject  
{  
}
```

```
// invoked by macro's inserted in target app  
+(void) logFile: (char*) sourceFile lineNumber: (int) lineNumber  
format: (NSString*)format, ...;
```

```
+(void) setLogOn: (BOOL) logOn;
```

```
// delete any existing log file, and create another one  
+ (void) zapLogFile: (NSString*) inFeatureName;
```

```
// create path to our target app's log file in  
// "Application Support/CMMR" folder  
+ (NSString*) targetAppLogFilePath;
```

```
// log a string to the target app's log file;  
// will create the log file if necessary  
+ (void) logString: (NSString*) inString;
```

```
@end
```

CommentsWindowController.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

*/

#pragma once

// includes

#import "DaughterWindowsController.h"

// forward declarations

@class CMMRDocument;

@class FDNNode;

@interface CommentsWindowController :
DaughterWindowsController {

 // outlets

 IBOutlet NSTextView* commentsView;

 }

 // construction / initialization / destruction

 + (void) displayCurrentNodeComments: (CMMRDocument*)
 inDoc;

@end

DaughterWindowsController.h

Copyright (c) ASI

March 2008

Abdallah Qaisi

*/

```
typedef enum {
    SourceCodeDWindow,
    MetricsDWindow,
    CommentsDWindow
} DWindowFlavor;

// forward declarations
@class CMMRDocument;
@class FDNode;

@interface DaughterWindowsController : NSObject {
    IBOutlet NSPanel*    daughterWindow;

    // public data
    CMMRDocument*    document;
    FDNode*    node;
    int    activeDWindow;
}

// construction / initialization / destruction
+ (void) displayDaughterWindowForNib: (NSString*) inNibName
    forDocument: (CMMRDocument*) inDoc;

// accessors / mutators
@property (assign) CMMRDocument* document;
@property (assign) FDNode* node;
@property (assign) NSPanel* daughterWindow;
@property (assign) int activeDWindow;

// public methods
- (BOOL) moveDaughterWindow: (id) sender direction: (short)
    keyCode;
```

```

// action methods
- (IBAction) closeDaughterWindow: (id) sender;

@end

@interface DaughterWindowsController (subClassMethods)
    - (void) setWindowContents;
@end
/*
    FDEdge.h
    Based on Hillegass's FiveDegrees sample project

    Copyright (c) ASI
    March 2008
    Abdallah Qaisi
*/

#import <Cocoa/Cocoa.h>

@class FDNode;

@interface FDEdge : NSObject <NSCoding>
{
    // Weak references in non-GC apps
    FDNode *toNode;
    FDNode *fromNode;
}
@property (readwrite, assign) FDNode *toNode;
@property (readwrite, assign) FDNode *fromNode;

+ (id) edgeWithToNode: (FDNode*) inToNode fromNode: (FDNode*)
    inFromNode;

- (NSMutableSet*) featureNodes;

@end

```

FDGraph.h
Based on Hillegass's FiveDegrees sample project

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
*/  
#import <Cocoa/Cocoa.h>  
  
@class FDNNode;  
@class FDEdge;  
  
@interface FDGraph : NSObject <NSCoding>{  
    NSMutableSet *nodes;  
    NSUndoManager *undoManager;  
}  
  
+ (FDGraph *)graphWithData:(NSData *)d;  
- (NSData *)dataRepresentation;  
- (NSSet *)nodes;  
- (void)addNodesObject:(FDNode *)f;  
- (FDNode *)getNodeObject:(NSString *)nodeLabel;  
- (void)removeNodesObject:(FDNode *)f;  
- (NSUndoManager *)undoManager;  
- (void)addEdge:(FDEdge *)e from:(FDNode *)f to:(FDNode *)t;  
- (void)removeEdge:(FDEdge *)e;  
- (FDNode*) nodeNamed: (FDNode*) inNode;  
- (void) removeNodeTree: (FDNode*) inNode;  
  
@end
```

FDGraphView.m
Based on Hillegass's FiveDegrees sample project

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
*/  
  
#import <Cocoa/Cocoa.h>  
@class FDGraph;  
@class FDNodeDisplayer;  
@class FDNode;  
  
#define FDGRAPH_SELECT_MODE    0  
#define FDGRAPH_LINE_MODE     1  
  
#define NO_STATE                0  
#define DRAGGING_EDGE_STATE    1  
#define MOVING_NODE_STATE      2  
#define EDITING_TEXT_STATE     4  
  
@interface FDGraphView : NSView {  
    FDGraph *graph;  
    NSMutableArray *nodeDisplayers;  
    int eventState;  
    int mode;  
    FDNodeDisplayer *selectedNodeDisplayer;  
    FDNodeDisplayer *selectedNodeDisplayer2;  
    NSPoint downPoint;  
    NSPoint currentPoint;  
    NSTextStorage *editorTextStorage;  
}  
  
- (void)setGraph:(FDGraph *)g;  
- (void)setMode:(int)m;  
- (int)mode;  
- (void)selectNode:(FDNode *)n;  
- (FDNode *)selectedNode;
```

- (void)endEditingText;
- (void)beginEditingTextOfSelectedNodeDisplayer;

@end

/*

FDNode.h

Based on Hillegass's FiveDegrees sample project

Copyright (c) ASI

March 2008

Abdallah Qaisi

*/

#import <Cocoa/Cocoa.h>

@class FDEdge;

struct HalsteadMetrics

{

NSNumber* N1; // numOfUniqueOperands;

NSNumber* N2; // numOfOperands;

NSNumber* n1; // numOfUniqueOperators;

NSNumber* n2; // numOfOperators;

NSNumber* N; // length = N1 + N2

NSNumber* n; // vocabulary = n1+n2

NSNumber* V; // volume = N*log2 n (app physical size)

// more stuff to consider for future releases...

};

@interface FDNode : NSObject <NSCoding>

{

NSString* label;

NSPoint location;

NSMutableSet* edges;

BOOL isChanged;

BOOL isFeature;

NSNumber* numFunctions;

NSNumber* numFeatures;

NSNumber* numReleases;

```

NSInteger                nodeNumber;
NSString*                filePath;
NSString*                parentFeatureName;
NSNumber*                startLineNumber;
NSNumber*                endLineNumber;
NSString*                creationDate;
NSString*                modDate;
NSMutableArray*          comments;
NSMutableArray*          sourceCode;

NSNumber*                McCabeVG;// #code paths in
function
struct HalsteadMetrics  halsteadMetrics
NSNumber*                maintIndex;
NSNumber*                kafuraCp;// (fan_in*fan_out) pow2
NSNumber*                systemC; // system complexity
                        // kafura's structure complexity +
                        // McCabe data complexity

NSNumber*                fbm; // MI * log #features;
NSNumber*                fm;  // maturity
NSNumber*                fr;  // function reliability =
                        // average of fm and the complement of mi;
}

@property (copy) NSString *label;
@property (assign) NSPoint location;
@property (assign) BOOL    isChanged;
@property (assign) BOOL    isFeature;
@property (copy) NSNumber* numFeatures;
@property (copy) NSNumber* numFunctions;
@property (copy) NSNumber* numReleases;
@property (assign) int     nodeNumber;
@property (copy) NSString *filePath;
@property (copy) NSString *parentFeatureName;
@property (copy) NSMutableSet *edges;

```



```

@property (copy) NSNumber*    startLineNumber;
@property (copy) NSNumber*    endLineNumber;
@property (copy) NSString*    creationDate;
@property (copy) NSString*    modDate;
@property (copy) NSMutableArray* comments;
@property (copy) NSMutableArray* sourceCode;

@property (copy) NSNumber*    mcCabeVG;
@property (assign) struct HalsteadMetrics halsteadMetrics;
@property (copy) NSNumber*    maintIndex;
@property (copy) NSNumber*    kafuraCp;
@property (copy) NSNumber*    systemC;

@property (copy) NSNumber*    fbm;
@property (copy) NSNumber*    fm;
@property (copy) NSNumber*    fr;

+ (id) featureNodeWithName: (NSString*) inName;
+ (id) nodeWithName: (NSString*) inName sourceDoc: (NSString*)
inDoc;
- (void)addEdgesObject:(FDEdge *)e;
- (void)removeEdgesObject:(FDEdge *)e;
- (NSMutableSet*) featureNodes;

@end
/*
    FDNODEDISPLAYER.H
    Based on Hillegass's FiveDegrees sample project

    Copyright (c) ASI
    March 2008
    Abdallah Qaisi
*/

#import <Cocoa/Cocoa.h>

@class FDNODE;
@class FDGRAPHVIEW;

```

```

@interface FDNodeDisplayer : NSObject {
    FDNode *representedNode;
    FDGraphView *graphView;
    NSPoint transientLocation;
}

@property (readwrite, assign) NSPoint transientLocation;
- (void)invalidate;
- (NSRect)bounds;

- (id)initWithRepresentedNode:(FDNode *)f
    view:(FDGraphView *)v;
- (void)drawSelected:(BOOL)yn;
- (FDNode *)representedNode;
- (void)offsetByVector:(NSPoint)p;
- (void)syncNode;
- (BOOL)hitTest:(NSPoint)p;
- (NSTextStorage *)textStorage;
- (void)setTextStorage:(NSTextStorage *)ts;

+ (NSSize) nodeSize;

@end
/*
    FeatureLogParser.h

    Copyright (c) ASI
    March 2008
    Abdallah Qaisi

#pragma once

// element names in Log files
#define CMMR_Log_DocumentName
    @ "\tCMMRPathName:"
#define CMMR_Log_LineNumber
    @ "\tCMMRLineNum:"
#define CMMR_Log_FuncStart
    @ "CMMRFuncStart:"
#define CMMR_Log_FuncEnd
    @ "CMMRFuncEnd:"

```

```

// graphing support
#define y_IncrementPerLevel      100.0
#define x_IncrementPerNode      100.0

@class FDNode;
@class CMMRDocument;

@interface FeatureLogParser : NSObject {

    @private

// public methods
+ (FDNode*) parseFeatureFile: (NSURL *) inFileUrl forFeatureName:
    (NSString*) inName forDocument: (CMMRDocument*)
inDoc;
+ (NSString*) funcNameStartForLogLine: (NSString*) inLogLine;
+ (NSString*) funcNameEndForLogLine: (NSString*) inLogLine;

@end

@interface FeatureLogParser (PrivateUtilities)
+ (NSString*) filePathForLogLine: (NSString*) inLogLine;
+ (NSNumber*) lineNumberForLogLine: (NSString*) inLogLine;
+ (BOOL) sameFunctionLogLines: (NSString*) firstLogLine
    secondLine : (NSString*) nextLogLine;
+ (void) addChildrenTo: (FDNode*) inFromNode fromLogLines:
(NSMutableArray*) logLines atIndex: (int) arrayIndex;
+ (int) removeDuplicatesFromLogArray: (NSMutableArray*)
logLines;
@end

```

FeatureXMLParser.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

*/

#pragma once

// graphing support

#define y_IncrementPerLevel 100.0

#define x_IncrementPerNode 100.0

@class FDNode;

@class CMMRDocument;

@interface FeatureXMLParser : NSObject {

 @private

}

// public methods

+ (FDNode*) parseFeatureFile: (NSURL*) inFileUrl forFeatureName:
 (NSString*) inName forDocument: (CMMRDocument*)
inDoc;

@end

@interface FeatureXMLParser (PrivateUtilities)

+ (NSString*) stringForXMLNode: (NSXMLNode*)
 inXMLNode elementNamed: (NSString*) inElementName;

+ (NSString*) filePathForXMLNode: (NSXMLNode*) inXMLNode;

+ (NSNumber*) lineNumberForXMLNode: (NSXMLNode*)

inXMLNode;

```
+ (NSString*) creationDateForXMLNode: (NSXMLNode*)
inXMLNode;
+ (NSString*) modDateForXMLNode: (NSXMLNode*)
inXMLNode;
+ (NSNumber*) fccForXMLNode: (NSXMLNode*) inXMLNode;
+ (NSMutableArray*) commentsForXMLNode: (NSXMLNode*)
inXMLNode;
```

```
+ (void) addChildrenTo: (FDNode*) inFromNode
fromXMLNode: (NSXMLNode*) inXMLNode;
```

```
@end
/*
```

```
MenuIds.h
```

```
Copyright (c) ASI
March 2008
Abdallah Qaisi
```

```
#pragma once
```

```
// includes
#import <Cocoa/Cocoa.h>
```

```
const NSUInteger
cmd_AppMenu = 100,
cmd_About = 101,
cmd_Preferences = 102,

cmd_FileMenu = 200,
cmd_NewProject = 201,
cmd_NewProjectBuild = 202,
cmd_Open = 203,
cmd_OpenRecent = 204,
cmd_Close = 205,
cmd_Save = 206,
cmd_SaveAs = 207,
cmd_Revert = 208,
cmd_PageSetup = 209,
```

cmd_Print	= 210,
cmd_EditMenu	= 300,
cmd_Undo	= 301,
cmd_Redo	= 302,
cmd_Cut	= 303,
cmd_Copy	= 304,
cmd_Paste	= 305,
cmd_Delete	= 306,
cmd_SelectAll	= 307,
cmd_FeaturesMenu	= 400,
cmd_AddFeature	= 401,
cmd_DeleteFeature	= 402,
cmd_UpdateFeature	= 450,
cmd_FunctionsMenu	= 500,
cmd_AddFunction	= 501,
cmd_DeleteFunction	= 502,
cmd_ShowFunctionViewMenu	= 550,
cmd_ShowFunctionNameView	= 551,
cmd_ShowFunctionCommentView	= 552,
cmd_ShowFunctionCodeView	= 553,
cmd_ShowFunctionMetricView	= 554,
cmd_MetricsMenu	= 600,
cmd_ShowFunctionReliabilityTrend	= 601,
cmd_ShowFeatureReliabilityTrend	= 602,
cmd_ShowProjectReliabilityTrend	= 603,
cmd_WindowMenu	= 700,
;	

MetricsComputer.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
#pragma once
```

```
// forward declarations
```

```
@class FDNode;  
@class CMMRDocument;  
struct HalsteadMetrics;
```

```
@interface MetricsComputer : NSObject {
```

```
    @private  
}
```

```
// public methods
```

```
+ (float) computeFunctionReliability : (FDNode *)  
        functionName forDoc: (CMMRDocument*)theDoc;  
+ (float) computeFeatureReliability : (FDNode *)  
        featureNode forDoc: (CMMRDocument*)theDoc;  
+ (float) computeProjectReliability : (FDNode *) productNode  
        forDoc: (CMMRDocument*)theDoc;  
+ (void) featuresThatLeadToNodesWithSameName : (NSString *)  
        functionName featureNames : (NSMutableArray*)  
        theFeatures forDoc: (CMMRDocument *)theDoc;  
  
+ (void) McCabeMetricsFromCodeAnalysis : (FDNode *) theNode;  
+ (void) HalsteadMetricsFromCodeAnalysis : (FDNode *) theNode;  
+ (void) MaintainabilityIndexFromCodeAnalysis : (FDNode *) theNode;  
+ (void) KafuraMetricsFromCodeAnalysis : (FDNode *) theNode  
        forDoc:(CMMRDocument *)theDoc;
```

```
@end
```

```

@interface MetricsComputer (PrivateUtilities)
+ (int) numOfOperatorsInCode : (FDNode *) theNode
    uniqueOpArray: (NSMutableArray
*)uniqueOpsArray;
+ (int) numOfOperandsInCode : (FDNode *) theNode
    uniqueOperandArray: (NSMutableArray
*)uniqueOpsArray;

@end
/*
    MetricsWindowController.h

    Copyright (c) ASI
    March 2008
    Abdallah Qaisi

*/

#pragma once

// includes
#import "DaughterWindowsController.h"

// forward declarations
@class CMMRDocument;
@class FDNode;

@interface MetricsWindowController : DaughterWindowsController {

    // outlets in 1st column
    IBOutlet NSTextField* nfTitle;

    // outlets in 2nd column
    IBOutlet NSTextField* creationDateField;
    IBOutlet NSTextField* modificationDateField;
    IBOutlet NSTextField* nfField;
    IBOutlet NSTextField* nrField;
    IBOutlet NSTextField* locField;
    IBOutlet NSTextField* lcmField;
}

```



```

IBOutlet NSTextField* fccField;
IBOutlet NSTextField* nField;
IBOutlet NSTextField* lField;
IBOutlet NSTextField* vField;
IBOutlet NSTextField* miField;
IBOutlet NSTextField* cpField;
IBOutlet NSTextField* cField;

IBOutlet NSTextField* fbmField;
IBOutlet NSTextField* fmField;
IBOutlet NSTextField* frField;

// outlets in 3rd column
IBOutlet NSTextField* creationDateFieldDesc;
IBOutlet NSTextField* modificationDateFieldDesc;
IBOutlet NSPopUpButton* nfFieldDesc;
IBOutlet NSTextField* nrFieldDesc;
IBOutlet NSTextField* locFieldDesc;
IBOutlet NSTextField* lcmFieldDesc;

IBOutlet NSTextField* fccFieldDesc;
IBOutlet NSTextField* nFieldDesc;
IBOutlet NSTextField* lFieldDesc;
IBOutlet NSTextField* vFieldDesc;
IBOutlet NSTextField* miFieldDesc;
IBOutlet NSTextField* cpFieldDesc;
IBOutlet NSTextField* cFieldDesc;

IBOutlet NSTextField* fbmFieldDesc;
IBOutlet NSTextField* fmFieldDesc;
IBOutlet NSTextField* frFieldDesc;

}

// construction / initialization / destruction
+ (void) displayCurrentNodeMetrics: (CMMRDocument*) inDoc;

```

@end

NewBuildController.h

Copyright (c) ASI

March 2008

Abdallah Qaisi

#pragma once

// forward declarations

@class CMMRDocument;

@interface NewBuildController : NSObject {

// outlets

IBOutlet NSWindowController* windowController;

IBOutlet NSTextField* folderTextField;

IBOutlet NSWindow* newBuildWindow;

// private data

NSString* newFolderName;

}

// construction / initialization / destruction

+ (void) createNewBuildForDocument: (CMMRDocument*) inDoc;

// accessors / mutators

@property (copy) NSString* newFolderName;

// action methods

- (IBAction) okButtonAction: (id) sender;

- (IBAction) cancelButtonAction: (id) sender;

@end

NewFeatureController.h

Copyright (c) ASI

March 2008

Abdallah Qaisi

*/

// forward declarations

@class CMMRDocument;

@interface NewFeatureController : NSObject {

// outlets

IBOutlet UIWindow* newFeatureWindow;
IBOutlet NSTextField* featureNameTextField;
IBOutlet NSButton* startButton;
IBOutlet NSButton* saveButton;

// public data

NSString* featureName;

@private

CMMRDocument* document;
NSWindow* parentWindow;
NSString* logFilePath;

}

// construction / initialization / destruction

+ (void) createNewFeatureForDocument: (CMMRDocument*)
inDoc;

// action methods

- (void) cancelButtonHit: (id) sender;
- (void) saveButtonHit: (id) sender;
- (void) startButtonHit: (id) sender;

@end

@interface NewFeatureController (PrivateUtilities)

- (CMMRDocument*) document;
- (void) setDocument: (CMMRDocument*) value;
- (NSWindow*) newFeatureWindow;

```

- (NSTextField*) featureNameTextField;
- (void) enableButtons;
- (NSWindow*) parentWindow;
- (void) setParentWindow: (NSWindow*) value;
- (NSString*) logFilePath;
- (void) setLogFilePath: (NSString*) value;
@end
/*
NewFunctionController.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

*/

// forward declarations
@class CMMRDocument;

@interface NewFunctionController : NSObject {

// outlets
IBOutlet NSWindow*          newFunctionWindow;
IBOutlet NSTextField*       functionNameTextField;
IBOutlet NSTextField*       functionLocationTextField;
IBOutlet NSButton*          chooseButton;
IBOutlet NSButton*          addButton;

@private
CMMRDocument*               document;
NSString*                    functionName;
NSWindow*                    parentWindow;
NSURL*                        functionLocation;
}

// construction / initialization / destruction
+ (void) createNewFunctionForDocument: (CMMRDocument*)
inDoc;

```

```

// action methods
- (void) cancelButtonHit: (id) sender;
- (void) addButtonHit: (id) sender;
- (void) chooseButtonHit: (id) sender;

@end

@interface NewFunctionController (PrivateUtilities)
- (CMMRDocument*) document;
- (void) setDocument: (CMMRDocument*) value;
- (NSWindow*) newFunctionWindow;
- (NSTextField*) functionNameTextField;
- (void) enableButtons;
- (NSWindow*) parentWindow;
- (void) setParentWindow: (NSWindow*) value;

@end
/*
NewProjectController.h

Copyright (c) ASI
March 2008
Abdallah Qaisi
#pragma once

@interface NewProjectController : NSObject {

// outlets
IBOutlet NSTextField*      projectNameTextField;
IBOutlet NSTextField*      projectLocationTextField;
IBOutlet NSTextField*      projectSourceTextField;
IBOutlet NSTextField*      projectExecutableTextField;

IBOutlet NSTextField*      projCreationDateTextField;
IBOutlet NSTextField*      projFirstReleaseDateTextField;
IBOutlet NSTextField*      projLastReleaseTextField;
IBOutlet NSTextField*      projNumReleasesToDateTextField;

```

```

IBOutlet UIWindow*      newProjectWindow;

IBOutlet NSButton*      okButton;

// private data
@private
NSURL*                  projectLocation;
NSURL*                  sourcesLocation;
NSURL*                  executableLocation;
}

// action methods
- (IBAction) okButtonAction: (id) sender;
- (IBAction) cancelButtonAction: (id) sender;
- (IBAction) projectNameBrowseButtonAction: (id) sender;
- (IBAction) projectSourcesBrowseButtonAction: (id) sender;
- (IBAction) projectExecutableBrowseButtonAction: (id) sender;
@end

```

ReliabilityChartController.h

Copyright (c) ASI
 March 2008
 Abdallah Qaisi

*/

```

// keys used in our dictionary
#define key_ReliabilityDict_Index @"index"
#define key_ReliabilityDict_Date @"date"
#define key_ReliabilityDict_BuildN @"buildNumber"

// draw spacing
#define reliabilityChartYSpacing 30.0
#define reliabilityChartXSpacing 40.0

// forward declarations
@class FDNode;
@class ReliabilityChartView;
@class ReliabilityChartLabelsView;

```

```

@interface ReliabilityChartController : NSObject {
    IBOutlet NSWindow*           window;
    IBOutlet ReliabilityChartView* graphView;
    IBOutlet ReliabilityChartLabelsView* bottomLabelsView;
    IBOutlet NSTextField*        bottomTitle;
    IBOutlet NSTextField*        sideTitle;

    @private
    FDNode*                       plottedNode;
    NSArray*                       pointsArray;
}

// accessors / mutators
@property (retain) FDNode* plottedNode;
@property (retain) NSArray* pointsArray;
@property (assign) NSWindow* window;
@property (assign) ReliabilityChartView* graphView;
@property (assign) ReliabilityChartLabelsView*
bottomLabelsView;
@property (assign) NSTextField* bottomTitle;
@property (assign) NSTextField* sideTitle;

// public methods
- (void) setNode: (FDNode*) functionNode pointsArray:
(NSArray*)
functionRelArray forProjectName: (NSString *)
projectName;
- (IBAction) closeButtonHit: (id) sender;
@end

```

ReliabilityChartLabelsView.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
#pragma once
```

```
// forward declarations
```

```
@class ReliabilityChartController;
```

```
//=====
```

```
=====
```

```
// ReliabilityChartLabelsView
```

```
//=====
```

```
=====
```

```
@interface ReliabilityChartLabelsView : NSObject {
```

```
// outlets
```

```
IBOutlet ReliabilityChartController *reliabilityChartController;
```

```
@private
```

```
}
```

```
// accessors / mutators
```

```
@property (assign) ReliabilityChartController *relChartController;
```

```
@end
```


ReliabilityChartView.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
#pragma once
```

```
// forward declarations
```

```
@class ReliabilityChartController;
```

```
@interface ReliabilityChartView : NSView {
```

```
    // outlets
```

```
    IBOutlet ReliabilityChartController*   reliabilityChartController;
```

```
    @private
```

```
}
```

```
    // accessors / mutators
```

```
    @property (assign) ReliabilityChartController* relChartController;
```

```
@end
```

ReliabilityChartWindow.h

Copyright (c) ASI
March 2008
Abdallah Qaisi

```
#pragma once
```

```
// forward declarations
```

```
@class ReliabilityChartController;
```

```

@interface ReliabilityChartWindow : NSWindow {

    // outlets
    IBOutlet ReliabilityChartController*   reliabilityChartController;

    @private

    // accessors / mutators
    @property (assign) ReliabilityChartController *relChartController;

@end

```

SourceCodeParser.h

Copyright (c) ASI
 March 2008
 Abdallah Qaisi

```
*/
```

```

#pragma once
// forward declaration
@class FDNode;
@class CMMRDocument;

```

```

@interface SourceCodeParser : NSObject {

    @private

}
// public methods
+ (void) parseSourceAndSetNodesFor : (FDNode *)featureNode
        inDocument : (CMMRDocument *) inDoc
        forBuild: (NSString *)lastBuildDateString;
+ (void) parseFileForFunctionNode : (FDNode

```

```

*)newFunctionNode
    forDocument : (CMMRDocument *) inDoc
    forBuild: (NSString *)lastBuildDateString;

// delegate methods

// action methods

@end
// interface SourceCodeParser (PrivateUtilities)
@interface SourceCodeParser (PrivateUtilities)

+ (NSString*) funcNameStartForSourceLine: (NSString*)
inLogLine;

+ (NSString*) funcNameEndForSourceLine: (NSString*)
inLogLine;

+ (bool) findFuncLinesInFileLines : (NSArray *)theFileLines
funcLine: (NSMutableArray *)theFuncLines
funcNode: (FDNode *)newFunctionNode;

+ (bool) extractFuncLinesFromFileLines : (NSArray *)theFileLines
intoArray: (NSMutableArray *)
theFuncLines funcNode : (FDNode *) aNode;

+ (void) expandFuncLinesForNode :(FDNode *) aNode
fromFileLines : (NSArray *)theFileLines;

+ (void) separateCodeFromComments : (NSMutableArray
*)linesInFunc
sourceArray: (NSMutableArray *) theCode
commentsArray: (NSMutableArray *) theComments;

+ (void) setNodeFromCodeAndComment : (FDNode *) aNode
sourceArray: (NSMutableArray *)theCode
commentsArray: (NSMutableArray *)theComments

```

```

        forBuild: (NSString *)lastBuildDateString;

+ (NSString *) creationDateFromComments :
        (NSMutableArray *) theComments;

+ (NSString *) modDateFromComments :
        (NSMutableArray *) theComments;

+ (BOOL)   isChangedFromModDate : (NSString *)
nodeModDate
        forBuild: (NSString *)lastBuildDateString;

+ (BOOL)   isChangedFromCodeCompare : (FDNode *)aNode
        sourceCode : (NSMutableArray *)theCode;

@end
/*
    SourceCodeWindowController.h

    Copyright (c) ASI
    March 2008
    Abdallah Qaisi

#pragma once

// includes
#import "DaughterWindowsController.h"

// forward declarations
@class CMMRDocument;
@class FDNode;

@interface SourceCodeWindowController :
DaughterWindowsController {

    // outlets
    IBOutlet NSTextView* sourceCodeView;

```

```
// construction / initialization / destruction  
+ (void) displayCurrentNodeSourceCode: (CMMRDocument*)  
inDoc;
```

```
@end
```

Appendix C

SOURCE CODE IMPLEMENTATION FOR

METRIC COMPUTATIONS IN CMMR

MetricsComputer.mm

Computes reliability of function, feature, and product. Computation of function reliability is based on function maturity and feature-based function complexity. FBFM is based on

MI (maintainability index). MI is based on McCabe, LOC, and Halstead metrics. McCabe is based on number of branch statements.

Halstead is based on number of operators and operands. All these metrics and the logic used for their computations are located in this single module for convenience. Computation of feature reliability is omitted because it's simply the average of the metrics of its functions. Same for Product computation.

Copyright (c) ASI
March 2008
Abdallah Qaisi

*/

```
// includes
#import "MetricsComputer.h"
#import "FeatureLogParser.h"

#import "FDNode.h"
#import "FDEdge.h"
#import "FDGraph.h"
#import "CMMRDocument.h"
#import "FDGraph.h"
#import "FDGraphView.h"
```

```

// static functions
static NSString *stringWithCharString( const char*cString);

// implementation starts here
@implementation MetricsComputer
//=====
=====
// computeFunctionReliability
//=====
=====

+ (float) computeFunctionReliability : (FDNode *) functionNode
forDoc: (CMMRDocument*)theDoc
{
    float    functionReliability = 0;
    float    functionMaturity = 0;
    float    function_fbm = 0;
    int      numFeatures = 0;

    // if FR is already computed, return it
    if ((functionNode.fr && [functionNode.fr floatValue] != -1.0))
        functionReliability = [functionNode.fr floatValue];
    else
    {
        // first compute number of features that use this function
        NSMutableArray* theFeatures = [[NSMutableArray alloc] init];
        [self featuresThatLeadToNodesWithSameName :
functionNode.label
        featureNames: (NSMutableArray *)theFeatures
forDoc:theDoc];
        numFeatures = [theFeatures count];

        // all the following computations are code-based
        if (functionNode.sourceCode.count)
        {
            // compute the third-party metrics first
            // compute McCabe cyclomatic complexity (mcCabeVG)
            // based on number of tokens

```

```

[self McCabeMetricsFromCodeAnalysis : functionNode];

// compute Halstead metrics
[self HalsteadMetricsFromCodeAnalysis : functionNode];

// now maintainability index
[self MaintainabilityIndexFromCodeAnalysis :
functionNode];

// now compute kafura which is number of children times
// number of parents squared of the node in the entire tree
[self KafuraMetricsFromCodeAnalysis : functionNode
forDoc:theDoc];

// now compute system complexity = kafura + mccabe
functionNode.systemC = [NSNumber
numberWithFloat:[functionNode.kafuraCp
floatValue] +
[functionNode.m McCabeVG floatValue]];

// now, compute function maturity based on info in
function

functionMaturity =
[self FunctionMaturityFromCodeAnalysis: functionNode];

// Now, compute the feature-based function
maintainability.
// This is based on MI and #features that use the function.
float mi_normalized = [[functionNode maintIndex]
floatValue] / 171.0;
function_fbm = mi_normalized * log10f(
(float)numFeatures
+ 9); // see math.h
if (function_fbm > 1.0)
function_fbm = 1.0;
if (function_fbm < 0)
function_fbm = 0;

```



```

// Finally, we can compute the function reliability
// of the function in *this* build
functionReliability = (function_fbm + functionMaturity) /2;
if (functionReliability > 1.0)
    functionReliability = 1.0;
if (functionReliability < 0)
    functionReliability = 0;

// Store the main computed values in the FDNode
// so that they are cached for next time and for
// the metric window.
functionNode.fm =
    [NSNumber numberWithFloat: functionMaturity];
functionNode.fbm =
    [NSNumber numberWithFloat: function_fbm];
functionNode.numFeatures =
    [NSNumber numberWithFloat: numFeatures];
functionNode.numReleases =
    [NSNumber numberWithInt: functionReleaseCount];
functionNode.fr =
    [NSNumber numberWithFloat: functionReliability];
}
else
// no code, no complexity/maintainability, max reliability
functionNode.fr = [NSNumber numberWithFloat:
functionReliability = 1.0];
}

return functionReliability;
}

```

```

//=====
// McCabeMetricsFromCodeAnalysis
//=====
=====
+ (void) McCabeMetricsFromCodeAnalysis : (FDNode *) theNode
{
    NSMutableArray *theCode = theNode.sourceCode;

    // start with 2 for start/end of function to make a directed graph
    int numTokensFound = 2;
    char *searchTokens[] = { "if", "else", "?", "&&", "||", "and",
                             "or", "switch", "case", "for", "while",
                             "goto", "break", "continue", "catch"};
    int numSearchTokens = sizeof(searchTokens);
    int numCodeLines = theCode.count;
    for (int codeLineIndex = 0; codeLineIndex < numCodeLines;
         codeLineIndex++)
    {
        NSString *codeLine = [theCode objectAtIndex:codeLineIndex];
        for (int searchTokenIndex = 0;
             searchTokenIndex < numSearchTokens;
             searchTokenIndex++)
        {
            NSString *aToken = stringWithCharString(
                searchTokens[searchTokenIndex]);
            NSArray *arrayStrs =
                [codeLine componentsSeparatedByString: aToken];

            // found what could be a token, make sure it's not part of
            // a bigger word; i.e. "for" in "before"
            if (arrayStrs.count > 1)
            {
                bool beforeTokenOK = false;
                for (int i=0; i < arrayStrs.count; i++)
                {
                    bool isToken = false;
                    NSString *nsString =
                        [arrayStrs objectAtIndex:i];

```

```

        if ([nsString length])
        {
            const char *str = nsString.UTF8String;
            char lastChar = str[strlen(str)-1];
            char firstChar = str[0];
            if ( beforeTokenOK && firstChar &&
                !isalpha(firstChar) &&
                !isdigit(firstChar))

                isToken = true;

            if ( lastChar && !isalpha(lastChar) &&
                !isdigit(lastChar))
                beforeTokenOK = true;
            else
                beforeTokenOK = false;
        }
        else
            isToken = true; // token starting line

        numTokensFound += isToken ? 1 : 0;
    }
}

}

}

theNode.mcCabeVG = [NSNumber numberWithInt :
numTokensFound];
}
//=====
=====
// stringWithCharString
//=====
=====
static NSString *stringWithCharString( const char*cString)
{
    NSString *retString = @"";
    if (cString && cString[0])

```

```

retString = [NSString stringWithCString:cString
              encoding:NSUTF8StringEncoding];
if (!retString)
    retString = [NSString stringWithCString:cString
                encoding:NSMacOSRomanStringEncoding];
if (!retString)
    retString = [NSString stringWithCString:cString
                encoding:NSASCIIStringEncoding];
}

return retString;
}
//=====
=====
// HalsteadMetricsFromCodeAnalysis
//=====
=====

+ (void) HalsteadMetricsFromCodeAnalysis : (FDNode *) theNode
{
    HalsteadMetrics hal;

    // count operators
    NSMutableArray *uniqueOpsArr = [[NSMutableArray alloc] init];
    hal.N1 = [NSNumber numberWithInt : [self
numOfOperatorsInCode :
        theNode uniqueOperatorArray: uniqueOpsArr]];
    hal.n1 = [NSNumber numberWithInt : uniqueOpsArr.count];

    // count operands
    NSMutableArray *uniqueOperandsArr = [[NSMutableArray alloc]
init];
    hal.N2 = [NSNumber numberWithInt: [self
numOfOperandsInCode :
        theNode uniqueOperandArray: uniqueOperandsArr]];
    hal.n2 = [NSNumber numberWithInt : uniqueOperandsArr.count];

    int vocabulary = [hal.n1 intValue] + [hal.n2 intValue];
    hal.n = [NSNumber numberWithInt : vocabulary];
}

```

```

int programLength = [hal.N1 intValue] + [hal.N2 intValue];
hal.N = [NSNumber numberWithInt : programLength];
int volume = programLength * log2(vocabulary);
hal.V = [NSNumber numberWithInt : volume];

// store the metric in the node
theNode.halsteadMetrics = hal;

//=====
=====
// MaintainabilityIndexFromCodeAnalysis
//=====
=====

+ (void) MaintainabilityIndexFromCodeAnalysis : (FDNode *) theNode
{
    // maintainability index uses McCabeVG, Halstead Volume, LOC,
    // and percentage of locomments to LOC
    // I am using a 0-100 formula which excludes comment
percentage.
    // Others did too;i.e. // http://blogs.msdn.com/fxcop/archive // /2007/11/20/maintainability-index-range-and-meaning.aspx
    // Maintainability Index = MAX(0,(171 - 5.2 * ln(Halstead Volume)
-
    // 0.23 * (Cyclomatic Complexity) - 16.2 * ln(Lines of Code))*100 /
    // 171); 0-9 = Red, 10-19 = Yellow; 20-100 = Green

float halV = [theNode.halsteadMetrics.V doubleValue];
float theV = 5.2 * log( halV);// natural logarithm (base e = 2.178)
float theFCC = 0.23 * [theNode.m McCabeVG doubleValue];
float loc = theNode.sourceCode.count;
float theLOC = 16.2 * log(loc);
float commentsPerc = 0;
if (theNode.comments.count)
    commentsPerc = theNode.comments.count * 100.0 / loc;

// the comments portion of the metric does not yield good results and
// it's optional anyway, according to:
// http://www.sei.cmu.edu/str/descriptions/mitmpm.html
#if 0

```

```

float sqrt = sqrt( 2.4 * commentsPerc);
float theLOCPerc = 50.0 * sin( sqrt);
float maintIndex = (171.0 - theV - theFCC - theLOC +
    theLOCPerc);
#else
float theLOCPerc = commentsPerc * theLOC / 100.0;
float maintIndex = (171.0 - theV - theFCC - theLOC +
theLOCPerc)
    * 100 / 171;
#endif

    theNode.maintIndex = [NSNumber numberWithInt:
        maintIndex>0 ? maintIndex : 0];
}
//=====
=====
// KafuraMetricsFromCodeAnalysis
//=====
=====

+ (void) KafuraMetricsFromCodeAnalysis : (FDNode *) theNode
    forDoc:(CMMRDocument *)theDoc
{
    int numFanIn = 0, numFanOut = 0;
    FDGraph* graph = [theDoc graph];
    for (FDNode* aNode in [graph nodes])
    {
        if (![aNode isFeature] &&
            [aNode.label isEqualToString: theNode.label])
        {
            for (FDEdge* anEdge in aNode.edges)
            {
                if ([anEdge toNode] == aNode)
                    numFanIn++;

                if ([anEdge fromNode] == aNode)
                    numFanOut++;
            }
        }
    }
}

```

```

int kafura = pow (numFanIn * numFanOut, 2);
theNode.kafuraCp = [NSNumber numberWithFloat: kafura];
}

@end

@implementation MetricsComputer (PrivateUtilities)

//=====
// FunctionMaturityFromCodeAnalysis
//=====

+ (int) FunctionMaturityFromCodeAnalysis: (FDNode *) functionNode)
{
    NSDate* funcCreationDate = [NSDate
        dateWithString:functionNode.creationDate
        calendarFormat:@"%Y-%m-%d"];

    // fetch the project dates to set these
    NSDate* projCreateDate = [NSDate
        dateWithString:[theDoc projectCreationDate]
        calendarFormat:@"%Y-%m-%d"];

    NSDate* firstReleaseDate = [NSDate
        dateWithString:[theDoc projectFirstReleaseDate]
        calendarFormat:@"%Y-%m-%d"];

    NSDate* lastReleaseDate = [NSDate
        dateWithString:[theDoc projectLastReleaseDate]
        calendarFormat:@"%Y-%m-%d"];

    int numberReleasesToDate =
        [[theDoc projectNumReleases] intValue];

```

```

// compute function age (in days), project age (in days),
// and first release age (in days)
int functionAge = ( [funcCreationDate timeIntervalSinceNow]
                   * -1 ) / (24*60*60); // convert seconds to days
int productAge = ([projCreateDate timeIntervalSinceNow]
                  * -1) / (24*60*60);
int firstReleaseAge = ([firstReleaseDate
                       timeIntervalSinceNow] * -1 ) / (24*60*60);
int lastReleaseAge = ([lastReleaseDate
                      timeIntervalSinceNow] * -1 ) / (24*60*60);

// compute the average release duration (in days)
// span between first release and last diff in days
int releaseSpan = (firstReleaseAge - lastReleaseAge);
int averageReleaseDuration = releaseSpan /
                             numberReleasesToDate; // in days

// compute # times the function has been part of a release
int funcDaysAfter1stRel = (firstReleaseAge - functionAge);
int functionReleaseCount = numberReleasesToDate -
                           (funcDaysAfter1stRel / averageReleaseDuration);
int productReleaseCount = numberReleasesToDate;
if (functionReleaseCount < 0)
    functionReleaseCount = 0;
if (functionReleaseCount > productReleaseCount)
    functionReleaseCount = productReleaseCount;

// Now we are ready to compute the FM value
functionMaturity = ((float)averageReleaseDuration *
                   functionReleaseCount + functionAge) /
                   (averageReleaseDuration * productReleaseCount +
                    productAge);
if (functionMaturity > 1.0)
    functionMaturity = 1.0;
if (functionMaturity < 0)
    functionMaturity = 0;

```



```

    return functionMaturity;
}
//=====
=====
// numOperatorsInCode:
// first counts ops, search for mult char ops first then single
// char ops so we don't count && as 2 or 3 ops
//=====
=====

+ (int) numOperatorsInCode : (FDNode *) theNode
uniqueOperatorArray: (NSMutableArray *)uniqueOpsArray
{
    char *operators[] = { "+=", "-=", "&=", "^=", "|=", "/=", "<<=",
        "%=", "*=", ">>=", ">=", "&&", "::", "||",
        "->", "++", "=", ">=", "<=", "!", "##", "+",
        "-", "=", "&", "/", "%", "*", "[", "(",
        ">", "<", "&", "!", "~", "#", ":", "new",
        "delete", "sizeof"};

    int numOperators = sizeof(operators);
    int opsFound = 0;
    NSMutableArray *theCode = theNode.sourceCode;
    int numCodeLines=theCode.count, codeLineIndex=0;

    for (int searchTokenIndex = 0;
        searchTokenIndex < numOperators;
searchTokenIndex++)
    {
        for ( codeLineIndex = 0;
            codeLineIndex < numCodeLines; codeLineIndex++)
        {
            NSString *anOp = stringWithCharString(
                operators[searchTokenIndex]);
            NSString *codeLine = [theCode
objectAtIndex:codeLineIndex];
            NSArray *arrayStrs =
                [codeLine componentsSeparatedByString: anOp];
            int subStringCount = arrayStrs.count;

```

```

if (subStringCount > 1) // found an op
{
    opsFound += subStringCount-1;
    // add the token if not there already
    if ([uniqueOpsArray indexOfObjectIdenticalTo:
        anOp] == NSNotFound)
        [uniqueOpsArray addObject: anOp];
    break;
}
}
}
return opsFound;
}

//=====
=====
// numOfOperandsInCode
//
// Here we count the full words in code (separated by whitespace)
// then omit any keywords, which leaves the count of operands,
// then removing duplicates gives us the unique count.
//=====
=====

+ (int) numOfOperandsInCode : (FDNode *) theNode
uniqueOperandArray: (NSMutableArray *)uniqueOpsArray
{
    char *keywords[] = { "asm", "auto", "break", "case", "catch",
        "char", "class", "const", "continue", "default", "delete",
        "do", "double", "else", "enum", "extern", "float", "for",
        "friend", "goto", "if", "inline", "int", "long", "private",
        "protected", "public", "return", "overload", "register",
        "using", "operator", "signed", "sizeof", "static",
        "struct", "switch", "template", "this", "throw", "try",
        "typedef", "union", "unsigned", "virtual", "void",
        "volatile", "while", "self", "super", "short"};

```

```

int numKeywords = sizeof(keywords);
int operandsFound = 0;
NSMutableArray *theCode = theNode.sourceCode;
int numCodeLines=theCode.count, codeLineIndex=0;

// Lets build the keyword array
NSMutableArray *keyWordsArray = [[NSMutableArray alloc] init];
for (int searchTokenIndex = 0; searchTokenIndex <
numKeywords;
    searchTokenIndex++)
    [keyWordsArray addObject:
    stringWithCharString( keywords[searchTokenIndex])];

for (codeLineIndex = 0;
    codeLineIndex < numCodeLines; codeLineIndex++)
{
    NSString *codeLine = [theCode objectAtIndex:codeLineIndex];
    const char *str = codeLine.UTF8String;
    int len = strlen(str);
    bool inWord = false;
    char *theWordStart;
    for (const char *aChar = str; aChar<str+len; aChar++)
    {
        // valid chars in an identifier
        if (isalpha(*aChar) || isdigit(*aChar) || *aChar == '_' )
        {
            if (inWord == false)
                theWordStart = (char *)aChar;
            inWord = true;
        }
        else
        {
            if (inWord == true) // found a word
            {
                operandsFound++; // count it
                short wordLen = aChar - theWordStart;
                theWordStart[wordLen] = 0;
                NSString *anOperand =
                    stringWithCharString(

```

```

        (char *)theWordStart);
        BOOL notKeyWord = ([keyWordsArray
            indexOfObject: anOperand] ==
NSNotFound);
        BOOL notInArray = ([uniqueOpsArray
            indexOfObject: anOperand] ==
NSNotFound);
        if (notKeyWord && notInArray)
            [uniqueOpsArray addObject:
anOperand];
    }
    inWord = false;
}
}
}
return operandsFound;
}
@end

```