Department of Computer Science
Tufts University

Software Maintenance
As Part of the Software Life Cycle

Comp180: Software Engineering

Prof. Stafford

Prepared by:
Kagan Erdil
Emily Finn
Kevin Keating
Jay Meattle
Sunyoung Park
Deborah Yoon

December 16, 2003

TABLE OF CONTENTS

SOFTWARE MAINTENANCE AS PART OF THE SOFTWARE LIFE CYCLE 4

1. INTRODUCTION 4

2. DESCRIPTION OF THE NATURE OF THE PHASE 5

2.1 FOUR TYPES OF SOFTWARE MAINTENANCE 5
2.2 THE SIGNIFICANCE OF SOFTWARE MAINTENANCE 7
2.3 SOFTWARE CHARACTERISTICS THAT AFFECT SOFTWARE MAINTENANCE EFFORT 8
2.4 SOFTWARE MAINTENANCE FROM A SERVICE PERSPECTIVE 9
2.5 SUMMARY OF THE NATURE OF THE MAINTENANCE PHASE 10

3. PROCESS 10

3.1 TRADITIONAL PROCESS MODELS 11
3.2 MAINTENANCE PROCESS MODELS 11
3.3 PROGRAM UNDERSTANDING 13
3.4 REVERSE ENGINEERING 13
3.5 REUSE AND REUSABILITY 14
3.6 MANAGEMENT AND ORGANIZATIONAL ISSUES 14
3.7 SUMMARY OF PROCESSES 15

4. TASKS 15

4.1 SUMMARY OF TASKS 16

5. TOOLS 16

5.1 COMMERCIALLY AVAILABLE PRODUCTS 18
5.2 SUMMARY OF TOOLS 18

6. ROLE OF SOFTWARE MAINTENANCE IN DEVELOPMENT METHODS 18

6.1 INTRODUCTION 18
6.2 ITERATIVE DEVELOPMENT 19
6.2.1 Rational Unified Process 20
6.2.2 Scrum 21
6.2.3 Case Study: Holland Railconsult (Switching to RUP) 21
6.2.4 Case Study: Micron's Facilties IS Team (Switching to RUP) 22
6.2.5 Agile development 22
6.2.5.1 What is Agile development? 23
6.2.5.2 When should you implement XP? 25
6.2.5.3 Success factors 25
6.2.5.4 Limitations 25
6.2.5.5 Effect on Maintenance stage 26
6.2.5.6 Case Study: IONA Technology (Applying XP to Maintenance) 26
6.2.6 Summary of Iterative development 27
6.3 COMPONENT BASED SOFTWARE DEVELOPMENT AND MAINTENANCE 28
6.3.1 What is CBSD? 28
6.3.2 What kind of roles the maintenance plays in CBSD? 29
6.3.2.1 The role of maintainers in CBSD 29
6.3.2.2 Major maintenance activities in CBSD 29
6.3.3 Advantages and disadvantages of CBSD in maintenance 30
6.3.4 Summary of CBSD 32
6.4 OPEN SOURCE 32
6.4.1 Differences with traditional project maintenance 33
6.4.1.1 Release date 33
6.4.1.2 Expectation of service 33

6.4.2 Advantages of the open source method *34*
6.4.3 Mozilla *35*
6.4.4 Summary of open source development *36*
6.5 SUMMARY OF THE ROLE OF MAINTENANCE IN DEVELOPMENT METHODS 36
7. CONCLUSION **37**
ACKNOWLEDGMENTS **38**
REFERENCES **38**
APPENDIX A: SOFTWARE MAINTENANCE COST IN SOFTWARE DEVELOPMENT **42**
APPENDIX B: BONSAI **43**
APPENDIX C: TINDERBOX **44**
APPENDIX D: BUGZILLA **45**
APPENDIX E: REQUIREMENT MANAGER IN TESTDIRECTOR **46**
APPENDIX F: TEST PLAN TREE IN TESTDIRECTOR **47**
APPENDIX G: CUSTOMIZABLE ACTION-DRIVEN WORKFLOW **48**
APPENDIX H: DEFINE CUSTOM VIEWS **49**

TABLE OF FIGURES

FIG. 1. THE QUICK FIX MODEL (TAKANG AND GRUBB [1996]) **12**
FIG. 2. THE REUSE MODEL (TAKANG AND GRUBB [1996]) **13**
FIG. 3. THE ITERATIVE CYCLE **20**
FIG. 4. THE COST OF CHANGE RISING EXPONENTIALLY OVER TIME (BECK [1999]) **24**
FIG. 5 XP, SCRUM, CRYSTAL, FDD, DSDM, ASD, PP, ISD, AM ARE ALL EXAMPLES OF AGILE METHODOLOGIES **25**
FIG. 6. AGILE PROCESSES FOLLOWED IN STAGES PRIOR TO MAINTENANCE **26**

SOFTWARE MAINTENANCE AS PART OF THE SOFTWARE LIFE CYCLE

KAGAN ERDIL
Tufts University
EMILY FINN
Tufts University
KEVIN KEATING
Tufts University
JAY MEATTLE
Tufts University
SUNYOUNG PARK
Tufts University
and
DEBORAH YOON
Tufts University

Maintenance plays an important role in the life cycle of a software product. It is estimated that there are more than 100 billion lines of code in production in the world. As much as 80% of it is unstructured, patched and not well documented. Maintenance can alleviate these problems. This paper describes the nature of software maintenance, why it is included in software development and how it's carried out. It discusses the role of maintenance played in iterative, agile, component-based and open source development models.

Categories and Subject Descriptors: D.2.7 [Software Engineering]; Distribution and Maintenance - *corrections*
General Terms: Design, Documentation, Management
Additional Key Words and Phrases: Case studies, Software maintenance, Software evolution, Process, Tasks, Tools, Reverse engineering, Software development, Iterative development, Agile development, Component-based development, Open source

---

1. INTRODUCTION

Software Development has many phases. These phases include Requirements Engineering, Architecting, Design, Implementation, Testing, Software Deployment, and Maintenance. Maintenance is the last stage of the software life cycle. After the product has been released, the maintenance phase keeps the software up to date with environment changes and changing user requirements.

The earlier phases should be done so that the product is easily maintainable. The design phase should plan the structure in a way that can be easily altered. Similarly, the

implementation phase should create code that can be easily read, understood, and changed. Maintenance can only happen efficiently if the earlier phases are done properly. There are four major problems that can slow down the maintenance process: unstructured code, maintenance programmers having insufficient knowledge of the system, documentation being absent, out of date, or at best insufficient, and software maintenance having a bad image. The success of the maintenance phase relies on these problems being fixed earlier in the life cycle.

Maintenance consists of four parts. Corrective maintenance deals with fixing bugs in the code. Adaptive maintenance deals with adapting the software to new environments. Perfective maintenance deals with updating the software according to changes in user requirements. Finally, preventive maintenance deals with updating documentation and making the software more maintainable. All changes to the system can be characterized by these four types of maintenance. Corrective maintenance is 'traditional maintenance' while the other types are considered as 'software evolution.'

As products age it becomes more difficult to keep them updated with new user requirements. Maintenance costs developers time, effort, and money. This requires that the maintenance phase be as efficient as possible. There are several steps in the software maintenance phase. The first is to try to understand the design that already exists. The next step of maintenance is reverse engineering in which the design of the product is reexamined and restructured. The final step is to test and debug the product to make the new changes work properly.

This paper will discuss what maintenance is, its role in the software development process, how it is carried out, and its role in iterative development, agile development, component-based development, and open source development.

2. DESCRIPTION OF THE NATURE OF THE PHASE

This section will cover what the software maintenance phase is about. As briefly seen in the introduction, software maintenance is not limited to the correction of latent faults. The term software maintenance usually refers to changes that must be made to software after they have been delivered to the customer or user. The definition of software maintenance by IEEE [1993] is as follows:

> The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

The following subsections will discuss different types of software maintenance, the significance and the characteristics of software maintenance.

2.1 Four types of software maintenance

There are four types of maintenance according to Lientz and Swanson: corrective, adaptive, perfective, and preventive [1980].

Corrective maintenance deals with the repair of faults or defects found. A defect can result from design errors, logic errors and coding errors (Takang and Grubb [1996]). Design errors occur when, for example, changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. Logic errors result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow or incomplete test of data. Coding errors are caused by incorrect implementation of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors. All these errors, sometimes called 'residual errors' or 'bugs', prevent the software from conforming to its agreed specification. The need for corrective maintenance is usually initiated by bug reports drawn up by the end users (Coenen and Bench-Capon [1993]). Examples of corrective maintenance include correcting a failure to test for all possible conditions or a failure to process the last record in a file (Martin and McClure [1983]).

Adaptive maintenance consists of adapting software to changes in the environment, such as the hardware or the operating system. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the system, for example, business rule, government policies, work patterns, software and hardware operating platforms (Takang and Grubb [1996]). The need for adaptive maintenance can only be recognized by monitoring the environment (Coenen and Bench-Capon [1993]). An example of a government policy that can have an effect on a software system is the proposal to have a 'single European currency', the ECU. An acceptance of this change will require that banks in the various member states, for example, make significant changes to their software systems to accommodate this currency (Takang and Grubb [1996]). Other examples are an implementation of a database management system for an existing application system and an adjustment of two programs to make them use the same record structures (Martin and McClure [1983]). A case study on the adaptive maintenance of an Internet application 'B4Ucall' is another example (Bergin and Keating [2003]). B4Ucall is an Internet application that helps compare mobile phone packages offered by different service providers. In their study on B4Ucall, Bergin and Keating discuss that adding or removing a complete new service provider to the Internet application requires adaptive maintenance on the system.

Perfective maintenance mainly deals with accommodating to new or changed user requirements. Perfective maintenance concerns functional enhancements to the system and activities to increase the system's performance or to enhance its user interface (van Vliet [2000]). A successful piece of software tends to be subjected to a succession of

changes, resulting in an increase in the number of requirements. This is based on the premise that as the software becomes useful, the users tend to experiment with new cases beyond the scope for which it was initially developed (Takang and Grubb [1996]). Examples of perfective maintenance include modifying the payroll program to incorporate a new union settlement, adding a new report in the sales analysis system, improving a terminal dialogue to make it more user-friendly, and adding an online HELP command (Martin and McClure [1983]).

Preventive maintenance concerns activities aimed at increasing the system's maintainability, such as updating documentation, adding comments, and improving the modular structure of the system (van Vliet [2000]). The long-term effect of corrective, adaptive and perfective changes increases the system's complexity (Takang and Grubb [1996]). As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it. This work is known as preventive change. The change is usually initiated from within the maintenance organization with the intention of making programs easier to understand and hence facilitating future maintenance work (Takang and Grubb [1996]). Examples of preventive change include restructuring and optimizing code and updating documentation.

Among these four types of maintenance, only corrective maintenance is 'traditional' maintenance. The other types can be considered software 'evolution'. The term evolution has been used since the early 1960s to characterize the growth dynamics of software (Chapin et al [2001]). Software evolution is now widely used in the software maintenance community. For example, *The Journal of Software Maintenance* added the term 'evolution' to its title to reflect this transition (Chapin and Cimitile [2001]).

2.2 The significance of software maintenance

As software systems age, it becomes increasingly difficult to keep them 'up and running' without maintenance. The following stories show the significance of the software maintenance phase in the software development life cycle.

Lost pet fees cost Toronto $700,000

"... the city [of Toronto] lost out on nearly $700,000 in pet fees [in 2000] because nearly half of Toronto's dog and cat owners were never billed [due to computerized billing system failure]. The staff who knew how to run the computerized billing system was laid off. [...] Only one city employee ever understood the system well enough to debug it when problems arose. That [employee was also laid off in 2000 due to downsizing] leaving no one to get things going again when the system ran into trouble and collapsed." (Bowker [2001])

UK and Y2K: $50 billion

"The Associated Press today [April 14, 1997] reports that Robin Guenier, head of the UK's TaskForce 2000, estimates that Y2K reprogramming efforts will cost Britain $50 billion dollars, three times the guesstimates of business consultants and computer service companies. Guenier suggested that 300,000 people may be required to tackle the problem. Coincidentally, that number is roughly equivalent to the number of full-time computer professionals in the UK." (Neumann [1997])

The first story implies the need of corrective maintenance. It is estimated that there are more than 100 billion lines of code in production in the world, and as much as 80% of it is unstructured, patched, and badly documented (van Vliet [2000]). It is necessary to keep these software systems operational. Errors and design defects in software must be corrected. Alternatively, the second story is an example of an adaptive change to the Y2K environment. Systems must also be adapted to changing environments and user requirement needs.

In fact, a substantial proportion of the resources expended within the Information Technology industry goes towards the maintenance of software systems. Annual software maintenance cost in the United States has been estimated to be more than $70 billion for ten billion lines of existing code (Sutherland [1995]). At the company level, Nokia Inc. used about $90 million for preventive Y2K-bug corrections (Koskinen [2003]).

Many studies were done to investigate the proportional software maintenance cost, in other words, the cost ratio of new development versus maintenance. The total cost of system maintenance is estimated to comprise at least 50% of total life cycle costs (van Vliet [2000]). The proportional maintenance costs range from 49 % for a pharmaceutical company to 75% for an automobile company in some studies according to Takang and Grubb [1996]. Zelkowitz et al [1979] also point out that in many large-scale software systems, only one-fourth to one-third of the entire life cycle costs can be attributed to software development. Most effort is spent during the operations and maintenance phase of the software life cycle as shown in Appendix A.

In their study of 487 data processing organizations, Lientz and Swanson [1980] reported on the proportion of maintenance effort allocated to each type of maintenance. Corrective maintenance accounted for slightly more than 20% of the total, on the average. Adaptive maintenance accounted for slightly less than 25%. Perfective maintenance accounted for over 50%. In particular, enhancements for users accounted for 42% of the total maintenance effort. Only 5% was spent on preventive maintenance activities.

2.3 Software characteristics that affect software maintenance effort

In order to increase the maintainability of software, we need to know what characteristics of a product affect its maintainability. There has been a great deal of speculation about what makes a software system more difficult to maintain. There are some program characteristics that are found to affect a product's maintainability. According to Martin and McClure [1983], these factors include system size, system age, number of input/output data items, application type, programming language, and the degree of structure.

Larger systems require more maintenance effort than do smaller systems, because there is a greater learning curve associated with larger systems, and larger systems are more complex in terms of the variety of functions they perform. Van Vliet [2000] points out that less maintenance is needed when less code is written. The length of the source code is the main determinant of total cost during maintenance as well as initial development. For example, a 10% change in a module of 200 lines of code is more expensive than a 20% change in a module of 100 lines of code. Older systems require more maintenance effort than do younger systems, because software systems tend to grow larger with age, become less organized with changes, and become less understandable with staff turnover.

Martin and McClure [1983] also discuss the factors that decrease maintenance effort. They are 1) Use of structured techniques, 2) Use of modern software, 3) Use of automated tools, 4) Use of data-base techniques, 5) Good data administration, and 6) Experienced maintainers.

2.4 Software maintenance from a service perspective

Niessink and van Vliet [2000] proposed software maintenance be seen as providing a service, whereas software development is concerned with the development of products. However, this is not yet widely recognized. Within the software maintenance domain, the focus is still on product aspects. The final phases of software development supposedly concern the delivery of an operations manual, installing the software, handling change requests and fixing bugs (van Vliet [2000]).

A service is defined as an essentially intangible set of benefits or activities that are sold by one party to another (Niessink and van Vliet [2000]). The main differences between products and services are as follows (van Vliet [2000]).

1) Services are intangible
2) Services tend to be more heterogeneous than products
3) Services are produced and consumed simultaneously, whereas production and consumption of products can be separated
4) Services are perishable, products are not.

The difference between products and services are not clear-cut. For example, babysitting is a 'relatively' pure service, while packaged food is a 'relatively' pure product. There is a product-service continuum for software development and maintenance. For example, adaptive maintenance can be seen as a hybrid of product and service, whereas corrective maintenance is a product-intensive service, and software operation is a relatively pure service. A custom software development is a service-intensive product (Niessink and van Vliet [2000]).

According to Niessink and van Vliet, customers judge the quality of software maintenance differently from how they judge the quality of software development. This implies a need to carry out software maintenance through different processes from those used by the average software development organization.

2.5 Summary of the nature of the maintenance phase

The traditional view of software maintenance deals with the correction of faults and errors that are found after the delivery of the product. However, as this section discussed, other significant changes are made to the product as software evolves. These changes can happen when the product needs to meet the new environment or new user requirements, or even to increase the product's maintainability. Adaptive, perfective, and preventive maintenance deal with these changes and these three types of maintenance are considered software 'evolution'.

There are a few aspects of software maintenance that set it apart from the other phases. Software maintenance cost comprises more than half of the total software development cost. Also, without software maintenance, it is impossible to change the problems within the product after its release, and many disasters can happen because of immature software.

Some characteristics of software that affect software maintenance are system size, age, and structure. Understanding the characteristics of software will facilitate maintaining the software more efficiently. It is also important to look at how software maintenance fits into the relationship between products and services. Software maintenance, including software operation, has relatively more aspects of a service than a product, whereas software development yields a product rather than a service.

3. PROCESS

A process model is the representation of the progress or course taken – the model of the process (Takang and Grubb [1996]). A process model gives and abstract representation of a way to build software. Looking at the traditional software models help see the difference between software development and software maintenance and understand the

need for maintenance conscious process models.

3.1 Traditional process models

Examples of traditional process models are the code and fix model, the waterfall model and the spiral models.

The code and fix model is a two-way phase model. The first phase of the model is writing the code, and the second phase is fixing it. The downfall of this model is that the code becomes hard to fix over time. In addition, this model does not give any room for future enhancements. This model is still used because, in real world applications the time required to identify and fix the problem is usually very limited, which does not spare any time for analysis and redesign.

The waterfall model gives a high level view of the software life cycle. The waterfall model is a tried and tested problem solving mechanism. Documentation is an integral part of the process. This model has various stages where the work of the each stage is “signed off” before proceeding to the next phase. The problem with this model is that it allows errors in the specification phase, which is more costly to correct at a later stage.

The spiral model is defined with 4 stages. First the identification of the objectives, constraints and alternatives is required. Then alternatives are assessed which helps correctly identifying the risks. The next phase is to develop the product. The final phase is to plan the next iteration of the spiral, which begins again with the first phase. The goal here is to identify and assess the high risk items so that they won’t turn into bigger issues down the line.

3.2 Maintenance process models

Traditional models fail to capture the evolutionary nature of software. Therefore different models are required that recognizes the requirement to build maintainability into the system. The five models that are used most in the industry are the quick fix model, Boehm’s model, Osborne’s model, the iterative enhancement model, and the reuse oriented model.

The quick fix model is an ad-hoc approach (see Figure 1). Its goal is to identify the problem and then fix it as quickly as possible. Due to time constraints, the model does not pay attention to the long-term affects of the fixes. The advantage of this model is that it gets work done quickly with lower cost. For example, if a system is developed and maintained by only one person, then that person will know the system well enough to make changes in a short time without the need to manage detailed documentation.

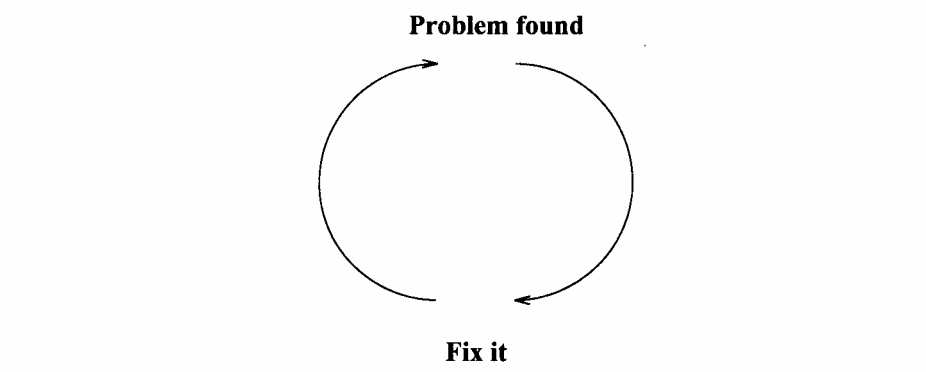


Fig. 1. The quick fix model (Takang and Grubb [1996])

A second model is Boehm's model. The foundation of Boehm's model is based on economic models and principles. The use of economic models helps us to better understand the problem and improve productivity in maintenance.

Osborne's model is concerned with the reality of the maintenance environment. In Osborne's point of view, technical problems that arise during maintenance are due to poor communication and control between management. Osborne recommends four strategies to address these issues.

1) Maintenance requirements need to be included in the change specification.
2) A quality assurance program is required to establish quality assurance requirements.
3) A metrics needs to be developed in order to verify that the maintenance goals have been met.
4) Managers need to be provided with feedback through performance reviews

The iterative enhancement model considers that changes made to the system during the software lifetime make up an iterative process. This model was adapted from development to maintenance. The model has three stages. First, the system has to be analyzed. Next, proposed modifications are classified. Finally the changes are implemented. This model is not effective when the documentation of the system is not complete, as the model assumes that a full documentation of the system exists.

The reuse oriented model assumes that existing program components could be reused (see Figure 2). The steps for the reuse model are identifying the parts of the old system which have the potential for reuse, fully understanding the system parts, modifying the old system parts according to the new requirements, and integrating the modified parts into the new system.

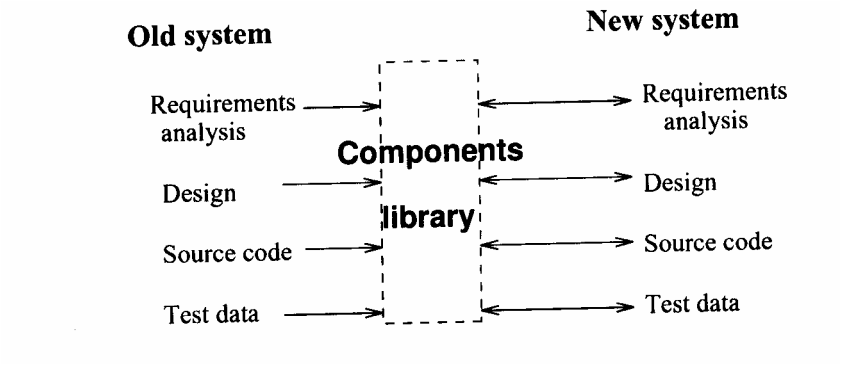


Fig. 2. The reuse model (Takang and Grubb [1996])

All of these models have their strengths and weaknesses. Therefore, usually more than one model is necessary for all maintenance activities. The best approach is to combine the models when required.

3.3 Program understanding

Program understanding means having the knowledge of what the software system does, how it relates to its environment, identifying where in the system changes are to be effected and having an in-depth knowledge of how the parts to be corrected or modified work. (Takang and Grubb [1996]). In order to successfully make changes to the system, the problem of the domain, effects of the execution, relation of cause-effect, relation of product-environment and features of decision-support need to be understood.

Every member of the maintenance team needs a comprehensive understanding of the system. Members of the team consist of managers, analysts, designers, and programmers. There are strategies that could be used to effectively form a mental model for the members of the team. These strategies are the top-down model, the bottom-up/chunking model, and the opportunistic model (Takang and Grubb [1996]).

3.4 Reverse engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at higher levels of abstraction (Chikofsky and Cross [1990]). Reverse engineering is required when the process to understand a software system would take a long time due to incorrect, out of date documentation, complexity of the system and the insufficient knowledge of the maintainer of the system.

The goals of reverse engineering are to recover lost information, to facilitate migration between platforms, to improve or provide documentation, to provide alternative views, to extract reusable components, to cope with complexity, to detect side effects and to reduce maintenance effort.

There are three types of abstraction: function, data and process abstraction. Function abstraction consists of eliciting functions from the target system. When using an abstraction process, one finds out information about what the function does. Particularly, one does not need to know how it operates. Data abstraction focuses on data objects, while process abstractions focus on the exact order in which the operations were performed.

3.5 Reuse and reusability

The goals of reuse during maintenance are to increase productivity, increase quality, facilitate code transportation, reduce maintenance time and effort, and improve maintainability. The definition of software reuse is as follows: The reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development or maintenance of that other system (Biigerstaff et al. [1989])

Software reuse is derived from the process, the personnel, and the product. The process is an activity or action performed by a machine or person. A methodology could be reused on different application problems. Object-oriented design provides a perfect example of reuse in development. Boehm's COCOMO model is another example of process reuse. COCOMO is a model that allows one to estimate the cost, effort and schedule when planning a new software development activity.

The reuse of personnel consists of reusing the knowledge of the people that have faced and overcome issues in previous projects and applying this knowledge to the new projects. An example of this is the "lessons learned" from previous projects. To maximize the effectiveness, the knowledge acquired needs to be transformed to a reusable form through domain analysis.

Product reuse consists of reusing the previously created projects. Data, design, and programs are all products that can be reused. Data formats such as XML could be easily used between applications. Architectural and detailed design could be used for redeployment of similar products, which would increase productivity and improve product quality. Program reuse uses code components such as modules, packages, procedures, functions and routines. Program components could be easily integrated into a new software system without a need for adaptation.

3.6 Management and organizational issues

Larger and complex software projects require significant management control. They also introduce challenges to management as complex software systems are a crucial part of the organization. Also, the maintenance of large software systems requires a large

numbers of employees. Therefore, management needs to find ways to increase productivity and ensure job satisfaction, which can be achieved by employing the right people, as well as motivating and training employees. Another factor that affects maintenance is selecting an appropriate way to organize maintenance tasks. This will increase productivity, control cost and deliver a quality system to the customer.

3.7 Summary of processes

Traditional life cycle models do not take account of the evolutionary nature of the software systems; therefore, different models are required for maintainability. Program understanding is a crucial part of maintenance since over half of the time and effort is spent on effecting change. Improving the performance on maintenance jobs will lead to higher productivity and successful evolution of software products. Software reuse also increases productivity and improves maintainability and quality of the software system by using the existing software components.

4. TASKS

Maintenance tasks can be grouped into five categories: analysis/ isolation, design, implementation, testing, and documentation (Basili et al.[1996])

Analysis/isolation tasks consist of impact analysis, cost benefit analysis, and isolation. Impact analysis and cost benefit analysis consist of analyzing different implementation alternatives and comparing their effect on schedule, cost, and operation. Isolation refers to the time spent trying to understand the problem or the proposed enhancements to the system.

Design consists of redesigning the system based on the understanding of the necessary changes. It also entails semiformal documentation, like release review documents.

Implementation entails code and unit testing. Code and unit testing refer to the time spent coding and testing the changes. It also consists of semiformal documentation, like the software modification test plan. Unit testing is performed by the maintainer who has made the changes. Unit testing is usually done locally on the user's workstation.

Testing consists of integration, acceptance and regression testing. Integration testing refers to the time spent on the integration of the components, while acceptance testing entails verifying that the changed system adheres to the user requirements. Acceptance testing is performed by the end users to ensure that the desired changes have been implemented successfully. Regression testing refers to the time spent ensuring that the changes did not affect the functionality of the other parts of the software.

Documentation consists of system, user and other documentation. System

documentation refers to the time spent writing or revising the system description document. User documentation entails writing or revising the user's guide and other formal documentation, excluding system documentation. Documentation is very important since the future changes will rely on the documentation of the previous changes/modifications.

4.1 Summary of tasks

Maintenance tasks are grouped into 5 categories. Among the maintenance task categories, code/unit testing takes the most effort of the programmer. A significant time is also spent on the design activities such as understanding of the necessary changes and semiformal documentation.

5. TOOLS

A software maintenance tool is an artifact that supports a software maintainer in performing a task (Takang and Grubb [1996]). The use of tools for software maintenance simplifies the tasks and increases efficiency and productivity.

There are several criteria for selecting the right tool for the task. These criteria are capability, features, cost/benefit, platform, programming language, ease of use, openness of architecture, stability of vendor, and organizational culture.

Capability decides whether the tool is capable of fulfilling the task. Once it has been decided that a method can benefit from being automated, then the features of the tool need to be considered for the job.

The tool must be analyzed for the benefits it brings against its cost. The benefit indicators of a tool are quality, productivity, responsiveness, and cost reduction. The environment that the tool runs on is called the platform. The language of the source code is called the programming language. It's important to select a tool that supports a language that is an industry standard.

The tool should have a similar feel to the ones that the users are already familiar with. The tool should have the ability to be integrated with different vendors' tools. This will help when a tool will need to run with other tools. The openness of the architecture plays an important role when the maintenance problem is complex. Therefore, it is not always sufficient to use only one tool. There may need to be multiple tools running together.

It is also important to consider the vendor's credibility. The vendor should be capable of supporting the tool in the future. If the vendor is not stable, the vendor could run out of business and not be able to support the tool. Another important factor is the culture of the organization. Every culture has its own work pattern. Therefore, it is important to take into consideration whether the tool is going to be accepted by the target users.

The chosen tools must support program understanding and reverse engineering, testing, configuration management, and documentation (Takang and Grubb [1996]). Selecting a tool that promotes understanding is very important in the implementation of change since a large amount of time is used to study and understand programs.

Tools for reverse engineering also accomplish the same goal. The tools mainly consist of visualization tools, which assist the programmer in drawing a model of the system. Examples of program understanding and reverse engineering tools include the program slicer static analyzer, dynamic analyzer, cross-referencer and dependency analyzer. (Takang and Grubb [1996]).

Slicing is the mechanical process of marking all the sections of a program text that may influence the value of a variable at a given point in the program (M. Weiser. [1984]). Program slicing helps the programmers select and view only the parts of the program that are affected by the changes. Static analyzer is used in analyzing the different parts if the program such as modules, procedures, variables, data elements, objects and classes. A static analyzer allows general viewing of the program text and generates summaries of contents and usage of selected elements in the program text, such as variables or objects (Takang and Grubb [1996]).

A dynamic analyzer could be used to analyze the program while it is executing. A data flow analyzer is a static analysis tool that allows the maintainer to track all possible data flow and control flow paths in the program (Vanek and Davis [1990]). It allows analysis of the program to better outline the underlying logic of the program. It also helps display the relationship between components of the system. A cross-referencer produces information on the usage of a program. This tool helps the user focus on the parts that are affected by the change.

A dependency analyzer assists the maintainer to analyze and understand the interrelationships between entities in a program (Takang and Grubb [1996]). Such a tool provides capabilities to set up and query the database of the dependencies in a program. It also provides graphical representations of the dependencies.

Testing is the most time consuming and demanding task in software maintenance. Therefore, it could benefit the most from tools. A test simulator tool helps the maintainer to test the effects of the change in a controlled environment before implementing the change on the actual system. A test case generator produces test data that is used to test the functionality of the modified system, while a test path generator helps the maintainer to find all the data flow and control flow paths affected by the changes.

Configuration management benefits from automated tools. Configuration management and version control tools help store the objects that form the software system. A source control system is used to keep a history of the files so that versions can

be tracked and the programmer can keep track of the file changes.

5.1 Commercially available products

There are numerous products on the market available for software maintenance. One type of product is bug tracking tools, which play an important role in maintenance. Bugzilla by the Mozilla Foundation is an example of such a tool (see appendix D). Other bug tracking products are Test Director by Mercury Interactive (see appendices E and F), Silk Radar by Segue Software (see appendix G and H), SQA Manager by Rational software, and QA director by Compuware.

ProTeus III Expert CMMS by Eagle Technology, Inc. is a maintenance software package that lets users schedule preventative maintenance, generate automatic work orders, document equipment maintenance history, track assets and inventory, track personnel, create purchase orders, and generate reports. Microsoft Visual Source Safe is a source control system tool that is used by configuration management.

Products that are specific to programming languages are CCFinder and JAAT which is specifically designed for JAVA programs (Kamiya et al [2001]). CCFinder identifies code clones in JAVA program. JAAT executes alias analysis for JAVA programs. For C++ programs, there is a tool called OCL query-based debugger which is a tool to debug C++ programs using queries formulated in the object constraint language, OCL (Hobatr and Malloy [2001]).

5.2 Summary of tools

The task of software maintenance has become so vital and complex that automated support is required to do it effectively. The use of tools simplifies tasks, increase efficiency and productivity. There are numerous tools available on the market for maintenance.

6. ROLE OF SOFTWARE MAINTENANCE IN DEVELOPMENT METHODS

6.1 Introduction

The earlier development of a software product can have a large impact on the maintenance of the product. Thus, an examination of maintenance in various development methods is presented below. The first method, iterative development, is a method in which each part of the product is created in iterations; for example, an iteration done for each new feature added to a product. Iterative development is particularly useful in maintenance because a new iteration can be done for each bug that is fixed and each feature that is added. This is more efficient than doing all of the maintenance at once because each added part of the product is fully working before the development

team moves onto the next one. This makes it easier to code and debug each part. In addition, it allows us to have a working product at the end of each iteration.

As projects get bigger, it becomes harder for the customer to define requirements early on. Agile is a form of iterative development that focuses on adapting to user requirements throughout the development process. This is accomplished by releasing a working product at the end of each iteration. Some popular methods of agile development are Extreme Programming, SCRUM, Crystal, and FDD.

The complexity of software makes their change through maintenance and evolution inevitable, and this intensifies the problem of coping with their complexity throughout their long lives. An approach to reduce this complexity to manageable levels is building systems out of components. Component-Based Software Engineering (CBSE) is based on the idea of developing software systems by selecting appropriate off-the-shelf components and then assembling them with a well-defined software architecture. CBSD has an emphasis in construction of software systems that makes use of reusable components.

The concept of open source is to release the source code of the product to the public so that others can modify it to add new features. Technically, open source entails only the philosophy of allowing end users to view and modify the source code. However, most open source projects follow similar practices during their development. Some of the best known examples of open source software are Linux, an operating system, and Mozilla, a web browser.

6.2 Iterative development

Iterative development is a method of software development in which the process is broken into small iterations. An iteration is performed for each new feature added to the product. At the beginning of an iteration, there is a meeting to plan out steps to be taken during the iteration. During these meetings, the developers and customers discuss user requirements, plan steps to be taken, and estimate costs. Iterative development implements just-in-time planning, in which decisions are made as changes occur. The following figure shows how an iteration works in the maintenance phase (Figure 3).

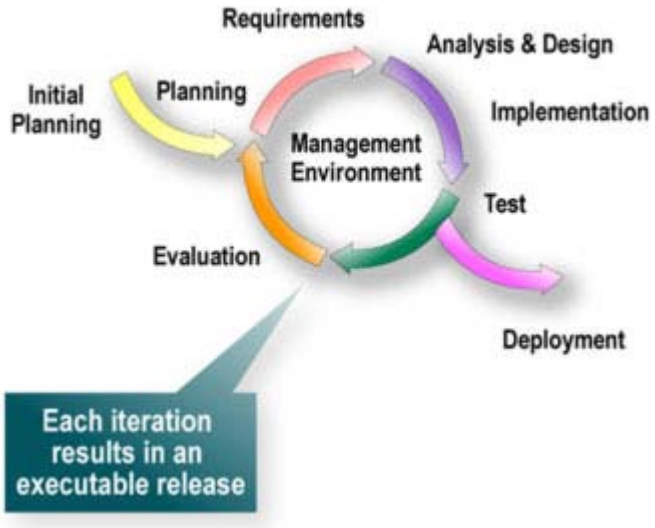


Fig. 3. The Iterative Cycle

There are multiple methods that use the iterative approach. Two of the most used methods are Rational Unified Process (RUP) and Scrum.

6.2.1 Rational Unified Process

With RUP, developers aim to fix risks as early as possible. The further they get into the development process, the more costly it becomes to fix risks. Therefore, it is important to fix them early. RUP also stresses the importance of documenting user requirements. User requirements should be referred to through each step of the life cycle. This ensures that the developers do not lose sight of the goals set in the beginning. Another important idea in RUP is to create a system that adapts well to change. Then, if a problem arises or the user requirements change, it will be much easier to accommodate changes. The RUP method also requires that systems be built with components. Building component based systems increases reusability and makes it easier to fix bugs. More on component based systems will be discussed later in the paper. The most important concept in RUP is that the team works together to make a quality product.

In RUP, earlier iterations are more concerned with requirements, analysis, and design, while later iterations concentrate on implementation and testing. RUP breaks up the development process into four phases, in which each phase can have one or more iterations. The first phase is the inception phase. In this phase, the developers lay out the user requirements, fix as many risks as possible, and get the stakeholders to approve the requirements before the beginning of the project. In the second phase, the elaboration phase, the developers design, implement, and test the architecture. In the third phase, the

construction phase, the product goes from the architectural stage to the first working version. This is the phase where most of the implementation gets done. In the final phase, the transition phase, the developers make sure that the product fulfills the user requirements and test it for release.

Because RUP creates systems that adapt well to change, it makes the maintenance phase easier. It is much easier to add new features to a system that adapts well to changes. The component based nature of RUP allows the system to be debugged and changed in parts.

6.2.2 Scrum

Another important form of iterative development is Scrum. Scrum is used mostly for maintenance on existing systems. In this method, an iteration is performed for each new feature added to the system, and at the end of each iteration, there is a working form of the product. Scrum's main idea that sets it apart from the other methods is its daily meetings. In Scrum, the team gathers every day for a meeting and asks three questions: "What did we accomplish yesterday?", "Were there any obstacles?", and "What will we accomplish today?" During these meetings, the team discusses what has been done and what to do next. There are four phases in the Scrum method: planning, architecture, development sprints, and closure. In the planning phase, the developers define the changes to be made, plan the schedule, and estimate costs. In the architecture phase, they plan how the changes will be implemented. The development sprints consist of developing the new functionality. Finally, during the closure phase, the developers plan for release.

Scrum is a good method to use for maintenance, because a new iteration can be done for each bug fixed or new feature added. The developers can go through the four stages to plan and execute the iteration. The daily meetings ensure that adequate progress is made throughout the iteration.

6.2.3 Case Study: Holland Railconsult (Switching to RUP)

Of these three methods of iterative development, RUP is possibly the most well-known method of software development. There exist a couple of success stories about companies that switched to RUP. One interesting success story is that of Holland Railconsult. This company used to use the waterfall model to create products, but switched over to RUP. When using the waterfall model, they had no documentation on how the system worked. Therefore, anyone who created a tool for the system had to maintain and answer any questions about it. Holland Railconsult wanted to change the system so they could have a team of developers and a team of maintainers. Another problem with their old system

was that when changing the software, they went straight to the code without planning or designing the changes.

After switching over to RUP, Holland Railconsult experienced many improvements in their methods and products. One of the greatest advantages of using an iterative method, was that they validated the product with the customer during the development process instead of just at the end. This made it easier to ensure that the final product was much closer to what the customer wanted. This also saved the company time and money that would have been spent after the product was done to make it fit the user requirements. Another advantage was that when they made changes to the system, they changed the models in addition to the code. This ensured that the models were consistent with the existing code. The iterative nature of the method made it much easier to add someone to the group, because there was plenty of documentation to train the person. In addition, the person could learn about the system by going through one iteration. This also saved the company time and money that they would have spent training the person. RUP also allowed for better communication between developers and better communication between the stakeholders and customers. Overall, RUP saved time, reduced cost, increased productivity, increased flexibility, and increased quality (Holland Railconsult [2003]).

6.2.4 Case Study: Micron's Facilities IS Team (Switching to RUP)

Another company that switched to RUP is Micron's Facilities IS Team, or FIST. They used RUP to get to Capability Maturity Model (CMM) level five. RUP provided them with a common communication method with users, a common documentation style, consistent business use-case documentation, and consistent analysis design documentation. FIST used a six week iteration cycle in which they had a deliverable to the customer every six weeks. In the first week, they analyzed cost and schedule. In the second week, they worked on the design. In the third and fourth weeks, they developed the components of the system. In the fifth week, they built and tested the product. In the sixth week, they deployed the application. After deployment, they prepared for the next cycle. After switching to RUP, they were left with a system with increased up-time and maintainability (Micron [2003]).

6.2.5 Agile development

Industry and technology move extremely fast, requirements change at rates that swamp traditional methods and customers have become increasingly unable to state their needs upfront while, at the same time, expecting more from their software. These changes in the software industry led to the development of Agile Methods, a form of iterative

development. Agile Methods are a reaction to traditional ways of developing software and acknowledge the need for an alternative to documentation driven, heavyweight software development processes. For the most part Agile processes are nothing new. The core values of Agile methodologies is what make it different from the rest (Cohen, et al. [2003]). This section discusses the core values and focus of Agile development, its success factors and limitations and when to deploy Agile development practices.

6.2.5.1 What is Agile development?

Agile development is a group of iterative software development methodologies that includes Extreme Programming or *XP* for short, Scrum, Standard & Poor's, Feature Driven Development, Crystal, and Adaptive Software Development. All Agile development methodologies share the following ideas (The Agile Manifesto):

- **Individuals and interactions** are valued over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

While Agile techniques vary in practices and process, they share common characteristics, which include development in small iterations, focus on interaction, communication, and the reduction of resource intensive processes. It is a software development paradigm that allows for tight collaboration between teams, intense customer involvement, and immediate feedback. Highest priority is given to satisfying the customer though early and continuous delivery of valuable working software.

Embracing change is the cornerstone of Agile development. It recognizes that over the course of the software development life cycle, change will occur as a natural part of the process. These changes will be in response to changes within the business itself, changes external to the business such as market changes, and new ideas and information that is discovered along the way. Being able to adapt to change more efficiently reduces the overall cost of change and of the project; in other words, flattening the cost of change over time curve in the following figure.

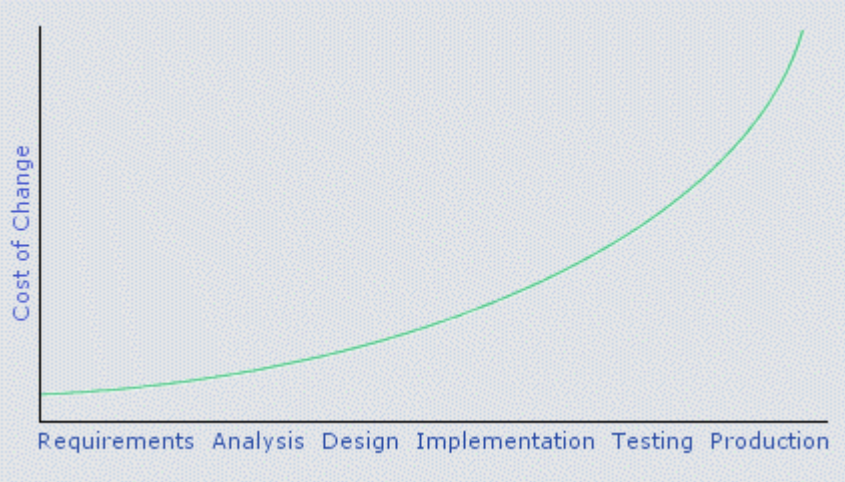


Fig. 4. The cost of change rising exponentially over time (Beck [1999])

Developing in iterations makes Agile processes adaptive – changing requirements are welcomed even in late development stages especially if they can enhance the customer's competitive advantage. Well-planned iterative projects start from the expectation that people (developers, users, other stakeholders) are not very good at figuring out how they will actually use the product, estimating costs, prioritizing features, or anticipating the problems they will encounter during development. The methodology is designed to help manage the risks associated with errors and omissions in assumptions and estimates. Consensus is built around a broad vision for the product rather than pretending to build consensus around the details. Each development iteration results in the creation of usable software for the customer that essentially is replaced every few weeks. The software is then used and tested by the customer who then provides direct feedback to the developers. This process might lead to feature additions and other changes in requirements.

After each iteration, stakeholders add, remove, and re-prioritize features. For re-prioritization to be successful there has to be high levels of face-to-face interaction between the stakeholders to ensure clear communication, feedback, and progress. High levels of interaction provide the stakeholders with an avenue for expressing ideas, thoughts, understanding, and innovation. Everyone involved with the development process is expected to maintain a constant pace indefinitely. Continuous attention is placed on technical excellence since good design enhances agility. Simplicity is valued as it keeps the team from getting bogged down and over-complicating the application over iterations. If the method is light and simple, modifications are easier. Projects are built around motivated individuals who need to be placed in an environment that supports their every need. Best architectures, requirements and designs emerge from self-

organizing teams.

Processes and behaviors within the organization are scrutinized at regular intervals and improvements are undertaken in order to become more effective and time efficient. Progress is measured in terms of delivery of working software.

	XP	Scrum	Crystal	FDD
Team Size	2 - 10	1 - 7	Variable	Variable
Iteration Length	2 weeks	4 weeks	< 4 months	< 2 weeks

Fig. 5 XP, Scrum, Crystal, FDD, DSDM, ASD, PP, ISD, AM are all examples of Agile methodologies.

6.2.5.2 When should you implement XP?

Extreme Programming is ideal when:

- The current software development process does not accept change.
- Requirements are nonexistent or change constantly.
- The current software development process takes too long.
- The software development team contains a mix of senior and junior level developers.
- It is difficult to produce quality software.
- Project stakeholders want to see interim releases.

6.2.5.3 Success factors

XP leads to increased control on the project because of high levels of interaction and straightforwardness of processes. It reduces time-to-market of products which increases competitive advantage for the customer as well as the developers, as time equals money. Small iterations result in reduced rework and scrap which lowers development and manufacturing costs which in turn maximizes the companies return on investment and probability of project success. Improved risk management enhances the ability of the project to adapt to change by minimizing the cost of change, also lowering the total cost of a project.

6.2.5.4 Limitations

Agile practices work if performance requirements are made explicit early, and if proper levels of testing can be planned for. Because of the nature of its process it does not work for systems that have criticality, reliability, and safety requirements. It best fits applications that can be built bare bones very quickly, especially applications that spend most of their lifetime in maintenance. Extensive documentation is not encouraged which makes later stages in the software development cycle such as maintenance harder to implement. Reduction in formal communication *could* lead teams to discount system

requirements.

6.2.5.5 Effect on Maintenance stage

Since Agile development considers setting up tools such as comprehensive documentation to be a secondary objective, subsequent stages in the software's life cycle, such as maintenance, are harder to implement. However, the best way to go about the actual maintenance stage is likely the Agile programming method because of its very nature. It keeps things simple, focuses on delivering working software *quickly* and welcomes and responds nimbly to changing requirements (such as fixing newly discovered bugs quickly). The following figure clearly shows these concepts.

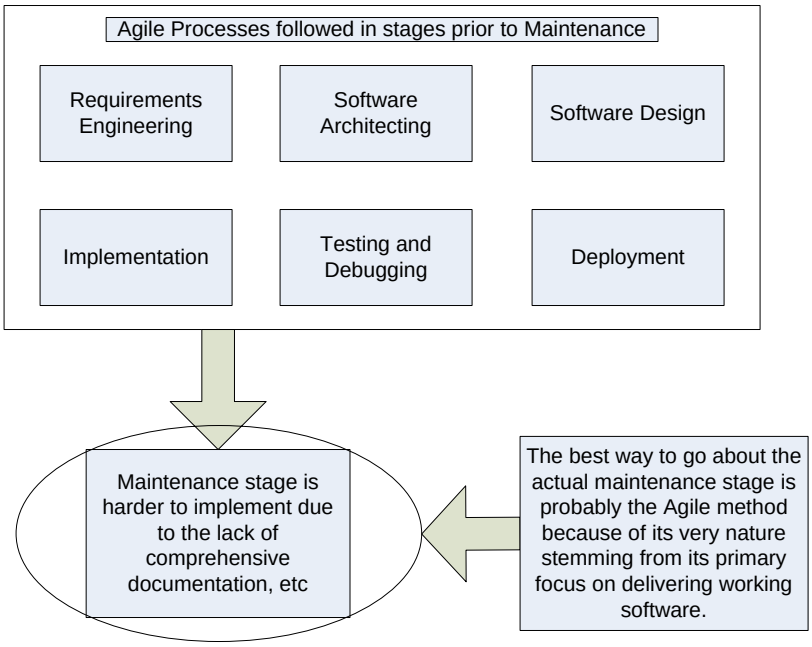


Fig. 6. Agile processes followed in stages prior to maintenance

XP in a maintenance environment (also known as extreme maintenance) is very common. According to Kent Beck, "Maintenance is really the normal state of an XP project. (Cohen, et al. [2003])" The project evolves over time because of frequent iterations. The first iteration can be considered the initial release. Therefore, all following iterations are, by definition, the maintenance stage of the development cycle.

6.2.5.6 Case Study: IONA Technology (Applying XP to Maintenance)

Poole and Huisman examined some of the problems experienced by IONA Technology's Orbix Generation 3 maintenance and enhancement team in 2001 and how the adoption of Extreme Programming further improved the team's ability to deliver quality support and enhancements to the products they work on.

As one of the foremost adopters of XP in the world, IONA has used XP to help

expand the breadth of best practices in product development, team size, and geographic distribution. Initially, XP was brought in at the senior level, but quickly gained ground amongst developers and managers alike. Some issues faced by IONA before the implementation of XP were:

- Testing. Quality of IONA's product was never its high point.
- Visibility issues stemming from lack of communication.
- Employee morale was low.
- Personal work practices were an issue.

Management wanted to do more with less: higher productivity, coupled with increased quality, decreased team size, and improved customer satisfaction. The engineers wanted more time to do a good job instead of always feeling pressured to deliver fix after fix. So they started looking at XP and learned that many of its elements come naturally to teams working the maintenance game (Poole and Huisman [2001]).

On implementing Agile methodologies, productivity improved by more than 67% (based on the next most productive 5-week period) and the level of interaction, communication, and morale improved noticeably. A big story board was set up to prioritize tasks and discuss projects on a daily bases. This improved visibility, giving others, especially managers, an opportunity to see what people were working on and their actual progress. The board focused people's attention on the fact that they were allowing issues to go unverified for significant amounts of time. There was a dramatic increase in the number of issues on which people started actively working. It is hypothesized that visibility alone was a strong motivational factor in this turnaround (Poole and Huisman [2001]).

IONA's positive experience with XP practices and the radical improvements it witnessed in team productivity and quality over a short period of time, demonstrates how effective Agile software development can be in a maintenance environment.

6.2.6 Summary of Iterative development

As user requirements become more volatile, iterative development becomes more important. Agile, a form of iterative development, is specifically designed to deal with changing user requirements. Because of the nature of the maintenance phase, it makes sense to use an iterative method. Iterative development allows the developers to validate the product throughout the process instead of at the end. Trying to validate the product at the end wastes time and money. By following the methods of iterative development, companies can increase productivity, lower their costs, and deliver better products to the customers.

6.3 Component Based Software Development and Maintenance

Component Based Software Development (CBSD) provides a method of constructing software systems that makes use of reusable components. CBSD is generally considered a good way to increase cost efficiency in software development (Szyperski [1997]). It improves more than just the documentation. It also provides increased reliability of the software when it is up and running, decreasing pressure on maintenance (Szyperski [1997]).

6.3.1 What is CBSD?

There are two main elements to CBSD: the component architecture and the component based development procedure. The architecture acts as a standard foundation for the reuse of software components, as this reutilization will not be able to take place if architecture is not standardized (Ning [1996]). The component development process is then able to use components as a central aspect of software development (Ning [1996]), and the need for standard software architectures is essential for this reuse.

The main elements of this component architecture are the component framework, the components, the component contracts, the coordination services, and the glue codes (Szyperski [1997]). The component framework affords a selection of coordination of runtime services, which support the component model and enforce the model (Ning [1996]). The component model is made up of the component types, the interfaces of these types and the rules of the patterns of interaction which could be allowed between these component types (Gao et al [2003]).

Some examples of component models are: EJB (Enterprise JavaBeans) from Sun Microsystems, COM and COM+ from Microsoft and CORBA (Common Object Request Broker Architecture).

Enterprise JavaBeans (EJB) provides a standard for developing reusable, portable components in Java. The Java language provides standards for application development. Java's promise is that it can be written once and run on any platform. With the Java language, also comes a standard framework for putting an application together. EJB provides an interlocking network of components that communicate with one another in a standardized way. The JavaBeans Component architecture, in theory, will run on any operating system and within any application environment.

EJB provides standardized software building blocks that interface with other code components while hiding the internal implementation of their work. In object-oriented programming, the concept of encapsulation allows each object the ability to encapsulate its own data and member functions. Each object is a small application that can effectively work in tandem with other objects, each performing a specific task, and

ultimately making up a large application.

A component is a software performance that can take place on either a logical device or a physical device. The component will execute the imposed interfaces and must satisfy specific component contracts (Gao et al [2003]).

The component contracts make sure that the components, which are developed independently, will obey the rules to ensure the interaction of the components in a predictable manner, so that their deployment and use in both standard build time and standard run time environments can be assured.

The coordination service is provided by the component framework. Examples of this may be seen with transaction services (Gao et al [2003]). Glue codes provide a key factor in CBSD, as they allow the independent components to operate together (Gao et al [2003]). This is necessary as it would be very unusual to find a situation where there can be automatic connections between the components. Also, the software that is complied with the use of components cannot operate without the glue code.

6.3.2 What kind of roles the maintenance plays in CBSD?

6.3.2.1 The role of maintainers in CBSD

The role of maintenance in CBSD is somewhat different from that in other custom-built systems. According to SEI, maintenance of CBSD differs from maintenance of custom-built systems in the following ways (Vidger [1999]):

1) System developers do not have access to the source code.
2) Maintenance and evolution of the component is controlled by a third party.
3) Maintenance is done at the component level rather than the source code level.

6.3.2.2 Major maintenance activities in CBSD

The maintenance activities for CBSD are specified as followed by SEI: (Vidger [1999])

1) Gluing and Wrapping : Even though CBSD suggests the concept of building a system out of components, components are not just “plug-and-play”. Wrappers around the components should be developed, and “glue codes” that enable the components to interact together should be implemented so that the system is coherent. As the system as a whole evolves, those wrappers and glue codes should be changed in order to accompany the changes.
2) Tailoring: The systems developer has to “tailor” the generic functionality of the component in order to make it fit into the system’s unique requirements. Even though the functionalities of the components get changed, these are not done at the source code level. They are done at the component level, including changes such as frameworks and scripting. The maintenance of the software system

must reflect the modified business process.

3) Fault identification and isolation: Since the maintenance in CBSE is not done at the source code level, developers can't change the source code of the component to fix the problem. The maintainers must identify the component causing the failure and calculate the next step to fix the problem. For example, get a new component and replace the old one, or have the component builders who originally produced the component to fix it.
4) Updating component configuration: Upgrading the component configuration is crucial in maintaining CBSD. Upgrading of configuration includes:
    i) Replacing components with newer versions with added functionalities as they are released by the component developers. Since components do not rely on the surrounding system, and the surroundings of a component have no need to know exactly how it performs its duties, components can be removed and replaced seamlessly. For example, if a failure of a system occurs while the root cause lies in a component, a revised component with patches in it from the component development company can be delivered to the maintainers and can be plugged into the system.
    ii) Substituting similar components with better functionalities from different vendors.
    iii) Adding or deleting components as the requirements of the system change.
5) Monitoring and auditing system behavior. Maintainers must be able to monitor the load, performance, usage, and the failure of each component.
6) Component testing. Before a new component is to be added to a system, the maintainers have to carefully test the component to determine the behavior of the component, differences from previous version, etc.

6.3.3 Advantages and disadvantages of CBSD in maintenance

There are both advantages and disadvantages to using this type of system. It is generally seen as a way of increasing levels of productivity, and also improving on the quality of software that is developed. However, it should also be remembered that this is still a relatively new area that is building on the success of object oriented methodology, but providing increased flexibility.

The advantage is found in the underlying concept; in large system, many parts of the software are repeated. In development, upgrading and maintaining a large system can result in increasing costs as the developers must make adjustments, which increases the

risk of conflict and clashes. In order to prevent these risks, parts are written only once and reused several times in the system. Therefore, a shift in design emphasis is seen from the design of the overall to the composition of the individual components.

However, reuse of the components is complex. A catalogue of the different components must be kept, and the developers and maintainers must have a good understanding of the different interfaces and the intricacies of the system. Building a component for reuse purpose or changing an existing component to make it reusable may also add additional costs. For example, in the case studied at NASA by Barry Boehm (Pree [1997]), it was found that a few changes (12%) on a single component increased the reuse cost by 55% compared to the development of the particular component built from scratch. This cost is associated not only with the physical changes, but also with the cost of understanding of requirements of the component (Pree [1997]). In addition, there is increased risk associated with this type of development. The reused components may save time, but may also increase the risk of corruption or unreliability due to interactions, making the maintenance process more difficult.

The existence of self-managed components introduces another advantage of CBSD. To explain this, an analogy may be made with a car. Inside a car's engine there is a combustion engine. This is managed by systems in place designed to control the engine. Through this system that a car is able to comply with the environmental regulation that control exhaust from the engine. This control mechanism also has an interface to control the engine for the user and for the purpose of diagnostics.

Applications that have built-in controls self-manage and lead to a decrease in the level of input labor on the maintenance side. It would also increase the efficiency of the system. This is implemented through of agents, which facilitate self-management during the lifecycle of the application.

The facilitation of the application and deployment occur when the frameworks and components have built-in control to allow their own installation and modification. The maintainers can then quietly test and monitor operations, making adjustments where necessary, undertaking work that would otherwise have been labor intensive, if at all possible.

This also results in a decrease in the levels of errors and down time as the flaws are detected early and remedied before they have noticeable manifestations. The system can give greater information to the administrator, such as notification where a fault is occurring. This leads to a simplified method of tracing the fault and remedying it. Remedying any problems in this nature may be simpler and take less time, thus increasing the efficiency of the maintenance staff.

6.3.4 Summary of CBSD

There are many disadvantages to CBSD, which can increase costs and create difficulties. However, when the system has been developed with the right components and the proper associated elements, such as the glue code and the architecture, this can be a very effective way of building and managing software system in the future. This system is likely to develop and evolve over the coming years.

6.4 Open Source

While development techniques such as those discussed above are solely methods to create software, open source would be better described as a "philosophy of software." In such development, the source code is distributed alongside of, or in place of, compiled binaries. It is also known as free software; however, this "is a matter of liberty, not price. To understand the concept, you should think of 'free' as in 'free speech,' not as in 'free beer'" (GNU Project [2003]). Though the terms "open source" and "free software" have been embraced by different groups and have gained slightly different connotations, this distinction has minimal impact on the maintenance process. As such, these terms will be treated synonymously. A summary of the open source process is given by Open Source Now, a group devoted to spreading open source software: "With open source software, the source code is open. You can see it, change it, improve it. And it's protected by a special license so if anyone else improves it, they can't redistribute it without the source code" [2003].

In its strictest definition, open source does not embody any particular approach to developing software so long as the source code remains available to the end user. However, in practice, most open source projects follow a similar methodology stemming from the fact that open source code allows for very different development techniques. It is important to note that these techniques were created by those writing software without any commercial backing. For example, the GNU Project was launched in 1984 to develop a "Unix-like operating system". This, one of the first major free software efforts, was started by Richard Stallman while at MIT (GNU Project [2003]). In 1991, Linus Torvalds began development on what would become the Linux operating system while he was a student at the University of Helsinki in Finland (Raymond *History* [1999]). While these techniques have successfully been applied to commercial development by corporations such as Netscape, the fact that the techniques were created in a non-commercial setting affects the entire development cycle, including maintenance.

6.4.1 Differences with traditional project maintenance

6.4.1.1 Release date

In open source projects, the importance of the release date is greatly diminished as there is typically no customer awaiting delivery. Therefore, the source code and the program are made available before the release date in exactly the same method as they are after the release. This blurs the line between maintenance and the earlier portions of the development life cycle. Maintenance is typically defined as the portion of the product life cycle after the official release and after a completed application has been delivered to the customer (Van Vliet [2000]). Thus, maintenance traditionally begins after the software has been used in a production environment, commonly designated version 1.0. However, open source applications will often be put into use as soon as they can be compiled to accomplish something useful. This stems from the "release early, release often" philosophy popularized by the development of Linux (Raymond *Cathedral* [2000]). This is necessary so that user feedback can be incorporated: unless the program is made available before version 1.0, there is no user base to test and debug the software, making the development of a release quality application difficult.

Due to this lack of distinction between product creation and maintenance, the open source model is akin to an incremental model of development, where the user is given a series of prototypes or partially working pieces of software (Van Vliet [2000]). However, in open source projects, these increments occur far more often. The "release often" portion of the Linux philosophy maximizes the usefulness of user feedback, as this feedback is related to a version of the software that is hours or days old, not weeks or months (Raymond *Cathedral* [2000]). However, because there is no set date when a final product is delivered, version 1.0 is important only for the accompanying sense of accomplishment. In terms of deliverables to the end user, there is no difference between this release and that of version 0.9 or 1.1.

6.4.1.2 Expectation of service

While the distinction between "free as in speech" and "free as in beer" is important, many open source projects are also offered free of charge. This results in another important difference when compared to traditional development methods: because the user has not paid for the software, there is no explicit expectation of service from the developer. If a user pays for an application that does not work, the developer is expected to offer assistance. If a user pays for a service contract and later desires a new feature, the developer is expected to implement this feature. However, if no money is paid, such expectations are not necessarily the case. Open source projects, though, offers users a new option: fix the problem or implement the feature on their own. Because they have

access to the source, users are, in a sense, co-developers (Raymond *Cathedral* [2000]).

However, these tasks are far from simple for open source projects of any significant complexity. Oftentimes, corporations do not wish to devote resources to maintaining an open source project; they simply want software that works and someone to turn to when problems arise. Thus, companies such as Red Hat offer distributions of Linux which come with service contracts to "deploy, integrate, update, manage, train, [and] support" the operating system and bundled applications (Red Hat [2003]). Red Hat collects new patches and updates, and delivers them to their customers in an easy-to-install package. Robert Young, the founder of Red Hat, likens his business to a car company: "Honda buys tires from Michelin, airbags from TRW, and paint from Dupont and assembles these diverse pieces into an Accord that comes with certification, warranties, and a network of Honda and independent repair shops. Red Hat takes compilers from Cygnus, web servers from Apache, an X Window System from the X Consortium… and assembles these into a certifiable, warranted" OS [1999]. Through this certification and warranty, Red Hat provides the maintenance activities that would normally be provided by the developer.

6.4.2 Advantages of the open source method

The differences between open source and more traditional development methods also lead to a number of advantages during the maintenance phase. As mentioned above, open source projects allow any end user to double as a co-developer, as users can examine the source code and suggest bug fixes. This leads to what has been described as Linus's Law: "Given enough eyeballs, all bugs are shallow" (Raymond *Cathedral* [2003]). In other words, "given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone" (Raymond *Cathedral* [2003]). Brooks made note of this phenomenon in *The Mythical Man Month* (Raymond *Cathedral* [2003]): "The total cost of maintaining a widely used program is typically 40% or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. *More users find more bugs*" [1995] (emphasis added). Open source builds on the advantage of "more users" by allowing all of these users, if they so desire, to see the source code. Thus, they can make more intelligent bug reports, or even offer a fix for a bug (Raymond *Cathedral* [2000]).

One of the corollaries to Linus's Law states that "debugging is parallelizable." "Debugging requires debuggers to communicate with some coordinating developer, but does not require significant coordination between debuggers" (Raymond *Cathedral* [2000]). Because of this, Brooks's Law is avoided. While there may be more people working on the project in the form of end users with access to the source code, communication between all of these people is not required. Thus the coordination

overhead is not paid and development is not slowed by their presence. To the contrary, these users allow bugs to be found and fixed at a remarkable rate (Raymond *Cathedral* [2000]).

In addition, open source software means that a user is not locked into a single company for maintenance. As noted above, the maintainer of the project is not necessarily the same company that developed the software. This means that the user can choose a maintainer that fits their needs and is free to change this maintainer at any time. Because anyone can modify the source code to correct problems and add features, if customers are no longer satisfied with Red Hat Linux, they can easily switch to a competitor such as SuSE or Mandrake without needing to purchase entirely new software.

6.4.3 Mozilla

These advantages led Netscape Communications Corporation to make an unprecedented move on January 22, 1998, when they announced plans to release the source code for their Communicator 5.0 web browser (Netscape [1998]). At that time, no "proprietary software had ever been released under a free software license" (Hamerly [1999]). This led to the development of what was later named Mozilla (Raymond *Cathedral* [2000]), now under control of the Mozilla Foundation (Mozilla Organization *Mozilla.Org Announces* [2003]). An examination of this project gives a view of what is involved in the maintenance of open source software and serves to demonstrate the above-mentioned principles in action.

This project has a core development team in charge of maintaining the source by both writing code themselves and deciding what submitted code is accepted. Because Mozilla has a highly modular code base, each major module, such as the image library or the XML parser, has a designated owner who knows that code best and makes the final decision as to "what should go into the module and what" should not (Hamerly [1999]).

A number of tools make this task possible. An examination of these tools gives an idea of what goes into the maintenance of a typical open source project. Three main programs are used by the Mozilla team: Bonsai, Tinderbox, and Bugzilla. (See appendices B, C, and D, respectively, for screen shots of these tools.) Bonsai acts as a revision control system and allows developers to check-in and -out code. In addition, it constantly runs test on the code in the background. If any major errors are found, the developers are alerted and further check-ins are prevented until the problem has been identified (Hamerly [1999]). Tinderbox acts as an interface to Bonsai. It allows the developers to see what is happening to the source by showing who submitted what code and what file versions went in to a particular build of the software (Hamerly [1999]).

The third tool, Bugzilla, as the name implies, is a defect tracking system created for Mozilla. While it has now taken on a life of its own as a separate open source project, it is still used in the maintenance of Mozilla. It allows bugs to be reported and commented on. In addition, a specific person can be assigned to fix a problem, and dependencies and interactions between related bugs can be noted (Mozilla Organization *Bugzilla* [2003]). Through the use of these tools, bugs can be found and discussed, patches can be submitted by the public, and once accepted, these fixes can be merged into the source. The current state of the source code can then be visualized to decide when to release a new version.

6.4.4 Summary of open source development

Despite a lack of distinction between the maintenance phase and earlier phases of development, maintenance remains an important part of the open source software life cycle. The customer generally has a choice of maintenance companies, as the maintainer is often different than the group that initially developed the program. This, along with the advantages of Linus's Law, serves to distinguish open source maintenance from that of other development methods.

6.5 Summary of the role of maintenance in development methods

As there are various methods of developing software, different approaches of maintenance activities are adopted for each different development method. Because Agile development processes focuses on changing the system to make it better able to adapt to user requirements throughout the software's lifetime, maintenance becomes easier. It is also good to use an iterative method for the maintenance phase because an iteration is done for each change made to the system and each bug that is fixed. This allows for smaller and more frequent releases. This ensures that each added part of the product is fully working before moving onto the next change. Using smaller releases also makes it easier to code and debug each part. In addition, it allows the developers to have a working product at the end of each iteration. On the other hand, component based software design is built on the idea of developing software systems by selecting appropriate off-the-shelf components and then assembling them with a well-defined software architecture. The maintenance of such systems is done at the component level rather than at the source code level. Therefore, if a defect is found in a specific part, the maintainers can seamlessly replace that module with a working one. This is done easily by modifying the glue code. Finally, open source development allows each user to serve as a co-developer. This leads to the creation of better software and allows users to customize a program on their own. Any of these development methodologies can be used to make the maintenance phase more productive and efficient.

7. CONCLUSION

Maintenance clearly plays an important role is the life cycle of a software product. As noted earlier, the cost of maintenance in the United States has been estimated at more than $70 billion annually for more than ten billion lines of existing code (Sutherland [1995]). While "traditional maintenance" applies only to corrective maintenance – fixing bugs in the code, the maintenance phase also incorporates three other main aspects that are considered to be elements of software evolution. Adaptive maintenance serves to modify the software for use in new environments, such as a new hardware platform or interfacing with a different database system. Perfective maintenance concerns adding new functionality to a product, typically as a result of a customer request. Finally, preventive maintenance increases the maintainability of a system, through updating documentation or changing the structure of the software.

There are a number of models of maintenance that serve to organize the five main tasks of the phase: isolating and analyzing the problem, designing a fix, implementing this fix, testing the resulting system, and updating documentation to reflect the changes made. A number of tools, such as automated analyzers and configuration management tools, aid in the accomplishment of these tasks.

Maintenance is heavily impacted by the methods used to develop a product. Thus, different development methods result in different maintenance procedures. Iterative development results in the creation of a working product after each iteration. Therefore, maintenance tasks are carried out on each working product created. This serves to ensure that problems will not go undiagnosed and unfixed for long. Agile development, a similar method to iterative, considers the creation of documentation a secondary objective, thus preventive maintenance can become problematic. However, because of the iterative nature of agile development, corrective and perfective maintenance are a natural extension of the development life cycle. Component Based Software Development shifts the focus of the phase to maintaining the interaction between components rather than at the source code level, as the maintenance of the specific components falls to their developer. Open source development blurs the line between maintenance and earlier phases of the software life cycle due to the typical lack of a definite release date. However, it offers the advantage of allowing users to double as co-developers, resulting in a large debugging team.

As the cost of maintenance has been estimated at 50% of total life cycle costs (Van Vliet [2000]), it is apparent that further study into this field will be necessary. Cost savings in this area can have a large impact on the overall life of a software project. This paper has presented an overview of this phase of the project life cycle and its role in

various means of development in the hopes of aiding this further study.

ACKNOWLEDGMENTS

Many thanks to Professor Stafford and Kevin Simmons for their help and invaluable support throughout the research and writing phases of this paper.

REFERENCES

BASILI, V. et. al 1995. *Understanding and Predicting the Process of Software Maintenance Releases.* University of Maryland, College Park, MD.

BECK, K. et al. 2003. Agile Manifesto. http://www.agilemanifesto.org

BECK, K. 1999. *Extreme Programming Explained.* Addison-Wesley, Boston, MA.

BERGIN, S., AND KEATING, J. 2003. A case study on the adaptive maintenance of an Internet application. *Journal of Software Maintenance and Evolution : Research and Practice* 15, 245-264.

BOWKER, P. 2001. Lost pet fees cost Toronto $700,000. http://catless.ncl.ac.uk/Risks/subj2.1

BROOKS, F. P. 1995. *The Mythical Man Month*. Addison-Wesley, Boston, MA.

CAMPBELL, B. 2003. Explanation of iterative development. http://www.allpm.com/article.php?sid=215

CHAPIN, N., HALE, J.E., KHAN, K. MD., RAMIL, J.F, AND TAN, W. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution : Research and Practice* 13, 3-30.

CHAPIN, N., AND CIMITILE, A. 2001. Announcement. *Journal of Software Maintenance and Evolution : Research and Practice* 13, 1.

COENEN, F. AND BENCH-CAPON, T. 1993. *Maintenance of Knowledge-Based Systems: Theory, Techniques and Tools.* Hartnolls Ltd, Bodmin, Cornwall, UK.

COHEN, D. et al., 2003. *"Agile Software Development," Data & Analysis Center for Software (DACS)*. Rome, NY.

GAO, J., TSAO, H., JACOB, S. AND WU, Y. 2003. *Testing and Quality Assurance for Component-based Software*, Artech House, Boston, MA.

Gill, G. K. 1990. *Cyclomatic complexity metrics revisited : an empirical study of software development and maintenance*. Center for Information Systems Research, Sloan School of Management, MIT Press, Cambridge, MA.

GNU PROJECT. 2003. *Free Software Foundation.* http://www.fsf.org/

GARY, H., DAVID, D., AND JOHN, F. 1997. Component-Based Software Development / COTS Integration, Carnegie Mellon Software Engineering Institute. http://www.sei.cmu.edu/str/descriptions/cbsd_body.html

HAMERLY, J., PAQUIN, T. AND WALTON, S. 1999. Freeing the Source: The Story of Mozilla. Open Sources: Voices from the Open Source Revolution. Chris Dibona, Sam Ockman, and Mark Stone. O'Reilly.
http://www.oreilly.com/catalog/opensources/book/toc.html

HAMILTON, G. 1997. JavaBeans 1.01 Specification, Sun Microsystems.
http://www.javasoft.com/beans/docs/beans.101.pdf

HOBATR, C. AND MALLOY, B. 2001. Using OCL-Queries for Debugging C++. IEEE 839-840.

HOLLAND RAILCONSULT. 2003. Advantages of changing from waterfall model to RUP at Holland Railconsult.
http://programs.rational.com/success/Success_StoryDetail.cfm?ID=277

IEEE. 1993. *IEEE Standard for Software Maintenance*. IEEE Std 1219-1993. Institute of Electrical and Electronics Engineers, inc., New York, NY.

KAMIYA, T. et al. 2001. Maintenance Support Tools for JAVA Programs: CCFinder and JAAT. IEEE. 837-838

KEMERER, C. F. 1992 *Empirical research on software maintenance,* Chris F. Kemerer, A. Knute Ream II. Center for Information Systems Research, Sloan School of Management, MIT Presss, Cambridge, MA.

KOSKINEN, J. 2003. Software Maintenance Cost.
http://www.cs.jyu.fi/~koskinen/smcosts.htm

KROLL, P. AND KRUCHTEN, P. 2003. *The Rational Unified Process Made Easy*. Addison-Wesley.

KRUCHTEN, P. 2000. http://www.therationaledge.com/content/dec_00/m_iterative.html

KRUCHTEN, P. 2001. Switching from waterfall model to RUP.
http://www.therationaledge.com/content/sep_01/t_waterfall_pk.html

LIENTZ, B.P. AND SWANSON, E.B.1980. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Reading, MA.

LIPPERT, M. et al. 2002. *Extreme Programming in Action*. John Wiley & Sons, Ltd, England.

MARTIN, J. AND MCCLURE, C.1983. *Software Maintenance: The Problem and Its Solutions.* Prentice Hall, Englewood Cliffs, NJ.

MICRON. 2003. Success story for Micron's switch to RUP.
http://acm.isu.edu/ISU/download.asp?dl=2

MATENA, V., HAPNER, M. AND STEARNS, B. 2000. *Applying the Enterprise JavaBeans: Component-Based Development for the J2EE Platform.* Addison-Wesley, New York, NY.

MOZILLA ORGANIZATION. 2003. Bugzilla: Bug Tracking System.
http://www.bugzilla.org/

MOZILLA ORGANIZATION. 2003. Mozilla.Org Announces Launch Of The Mozilla Foundation To Lead Open-Source Browser Efforts. http://www.mozilla.org/press/mozilla-foundation.html

NETSCAPE COMMUNICATIONS CORPORATION. 1998. Netscape Announces Plans To Make Next-Generation Communicator Source Code Available Free On The Net. http://wp.netscape.com/newsref/pr/newsrelease558.html

NIESSINK, F. AND VAN VLIET, H. 2000. Software maintenance from a service perspective. *Journal of Software Maintenance and Evolution : Research and Practice* 12, 103-120.

NING J.Q. 1996. A Component-Based Software Development Model, In *Proceedings of the Annual International Computer Software and Applications Conference* (COMPSAC'96), pp.389–394, IEEE.

NEUMANN, P. G. 1997. UK and Y2K: $50 billion. http://catless.ncl.ac.uk/Risks/19.07.html#subj6.1

OBJECT MENTOR, INC. 2003. Description of iterative development. http://www.objectmentor.com/writeUps/IterativeDevelopment

OPEN SOURCE NOW. 2003. http://www.redhat.com/opensourcenow/

TATTERSALL, G. 2003. Description of Iterative Development. http://www.therationaledge.com/content/oct_02/m_iterativeDevelopment_gt.jsp

POOLE, C. AND HUISMAN, J. 2001 "Using Extreme Programming in a Maintenance Environment," *IEEE Software*, vol. 18, no. 6, pp. 42-50,.

PREE, W. 1997. Component-Based Software Development—A New Paradigm in Software Engineering? *Software—Concepts and Tools*, 18: 169–174

RAJIV D. 1992. *Banker Software complexity and software maintenance costs*. Center for Information Systems Research, Sloan School of Management, MIT Press, Cambridge, MA.

RAYMOND, E. S. 1999. A Brief History of Hackerdom. Open Sources: Voices from the Open Source Revolution. Chris Dibona, Sam Ockman, and Mark Stone. O'Reilly. http://www.oreilly.com/catalog/opensources/book/toc.html

RAYMOND, E. S. 2000. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/

RED HAT, INC. 2003. http://www.redhat.com/

SCHNEBERGER, S. L. 1997. *Client/server software maintenance.* McGraw-Hill, New York, NY.

SCHWABER. K. 2003. Outline of Scrum. http://www.jeffsutherland.com/oopsla/schwapub.pdf

SUTHERLAND, J. 2003. Articles on Scrum. http://jeffsutherland.com/papers/OTUG2003/Inventing_Scrum_files/frame.htm

SUTHERLAND, J. 1995. *Business Objects in Corporate Information Systems*. ACM Computing Surveys 27:2:274-276.

SZYPERSKI, C. 1997. *Component Software: Beyond Object-Oriented Programming*. Addison- Wesley, New York, NY.

TAKANG, A.A., AND GRUBB, P.A. 1996. Software *Maintenance Concepts and Practic*. Thompson Computer Press London, UK.

VAN Vliet, H. 2000. *Software Engineering:Principles and Practices*, 2nd Edition. John Wiley & Sons, West Sussex, England.

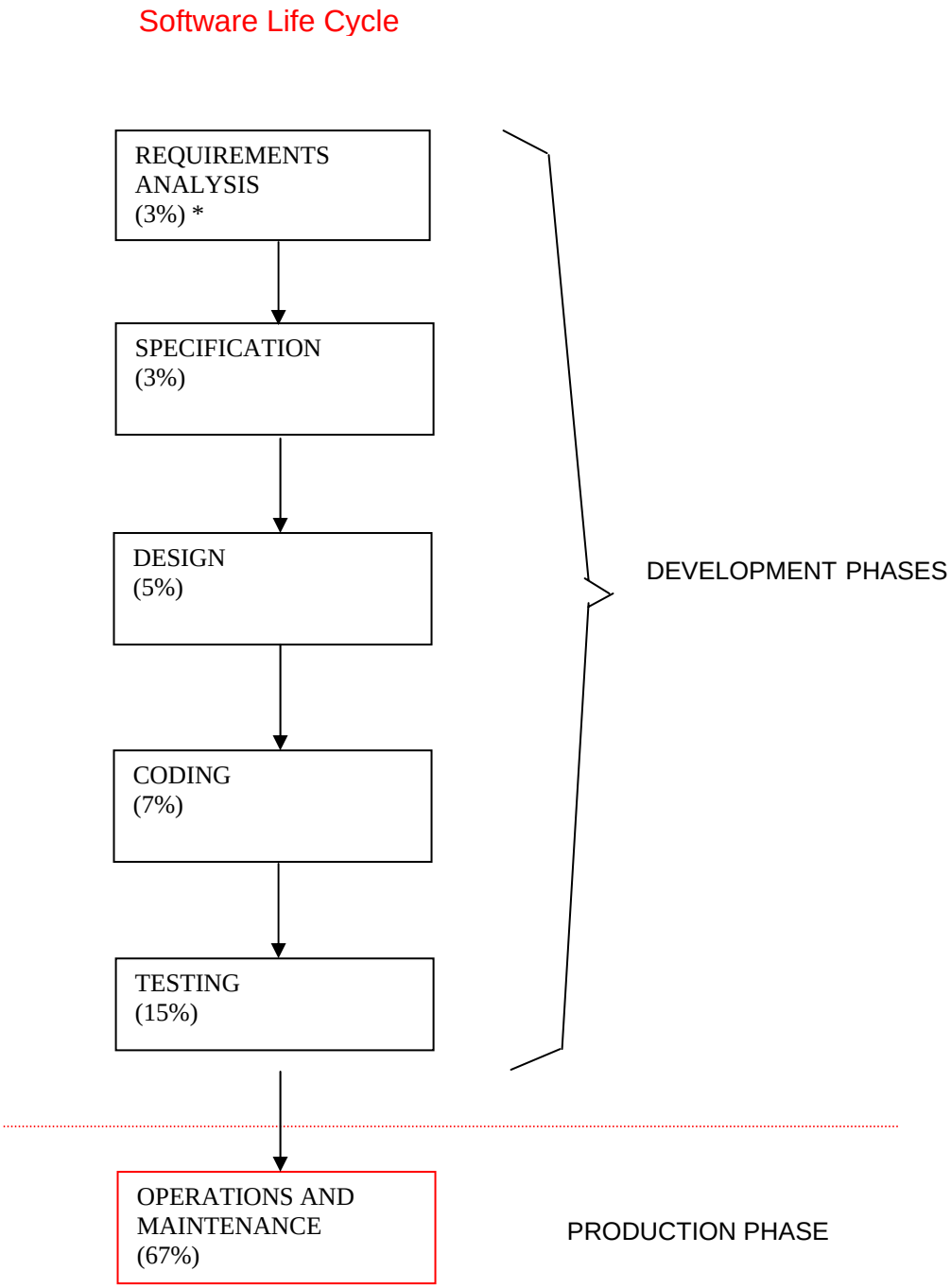
VIGDER, M. R. 1999. Building Maintainable Component-Based Systems, Carnegie Mellon Software Engineering Institute. http://www.sei.cmu.edu/icse99/papers/38/38.htm

WELLS, D. 1999. Rules of Extreme Programming. http://www.extremeprogramming.org/rules/iterative.html

YOUNG, R. 1999. Giving It Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry. Open Sources: Voices from the Open Source Revolution. Chris Dibona, Sam Ockman, and Mark Stone. O'Reilly. http://www.oreilly.com/catalog/opensources/book/toc.html

ZELKOWITZ, M. V., SHAW, A. C., AND GANNON, J. D. 1979. *Principles of Software Engineering and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

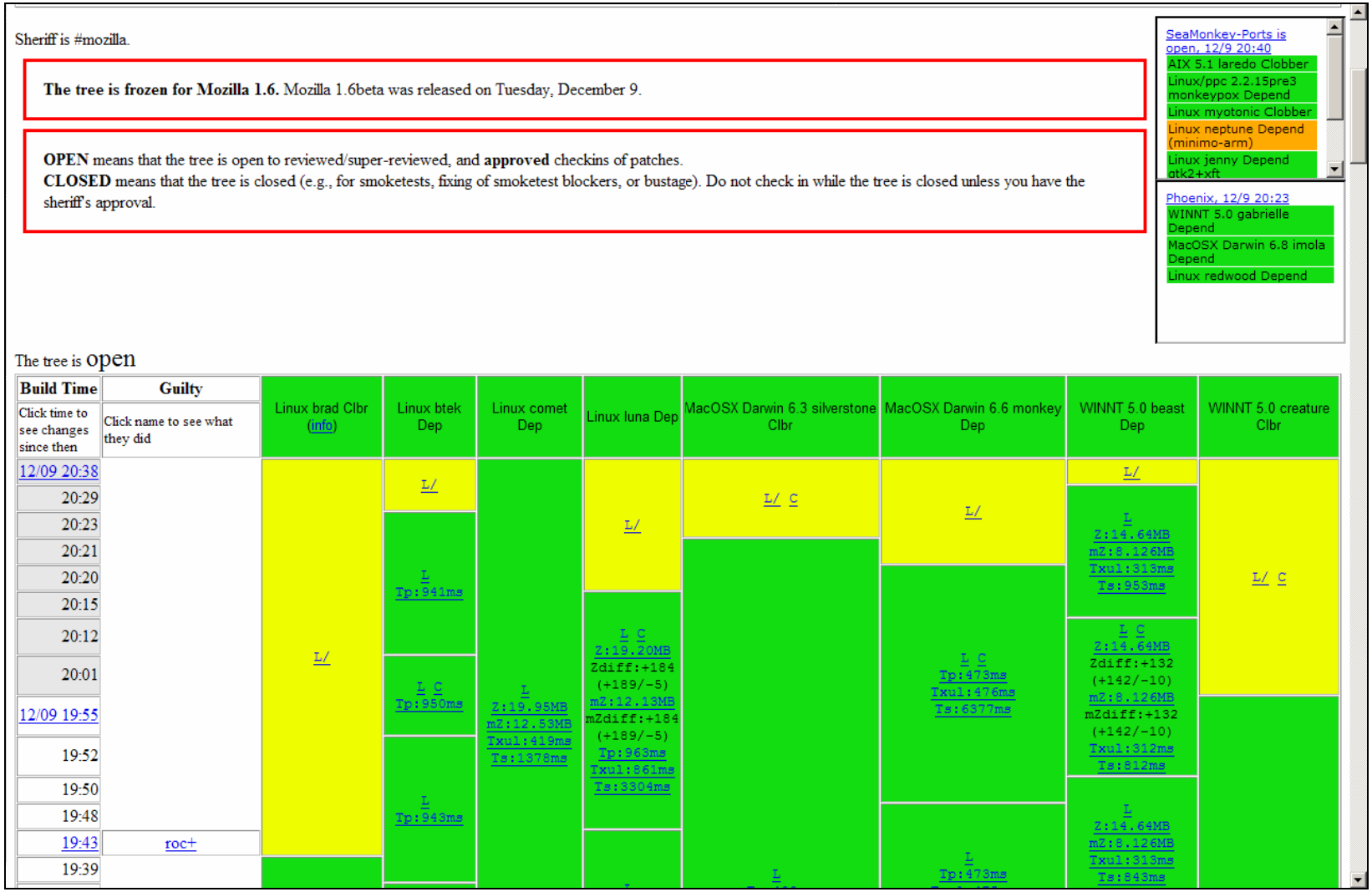
APPENDIX A: SOFTWARE MAINTENANCE COST IN SOFTWARE DEVELOPMENT



Maintenance phase costs most of the software life cycle
(Zelkowitz et al. [1979])

Bonsai is the revision control system used by the Mozilla development team. It allows users to check-in and -out code, and constantly runs tests on the code in the background. If any major errors are found, the developers are alerted and further check-ins are prevented until the problem has been identified (Hamerly, 1999).

This screen shot shows the main screen of Bonsai, which allows a user to search for a file they plan to work on. It can be accessed at http://bonsai.mozilla.org.



Sheriff is #mozilla.

The tree is frozen for Mozilla 1.6. Mozilla 1.6beta was released on Tuesday, December 9.

OPEN means that the tree is open to reviewed/super-reviewed, and **approved** checkins of patches.
CLOSED means that the tree is closed (e.g., for smoketests, fixing of smoketest blockers, or bustage). Do not check in while the tree is closed unless you have the sheriff's approval.

SeaMonkey-Ports is open, 12/9 20:40
- AIX 5.1 laredo Clobber
- Linux/ppc 2.2.15pre3 monkeypox Depend
- Linux myotonic Clobber
- Linux neptune Depend (minimo-arm)
- Linux jenny Depend atk2+xft

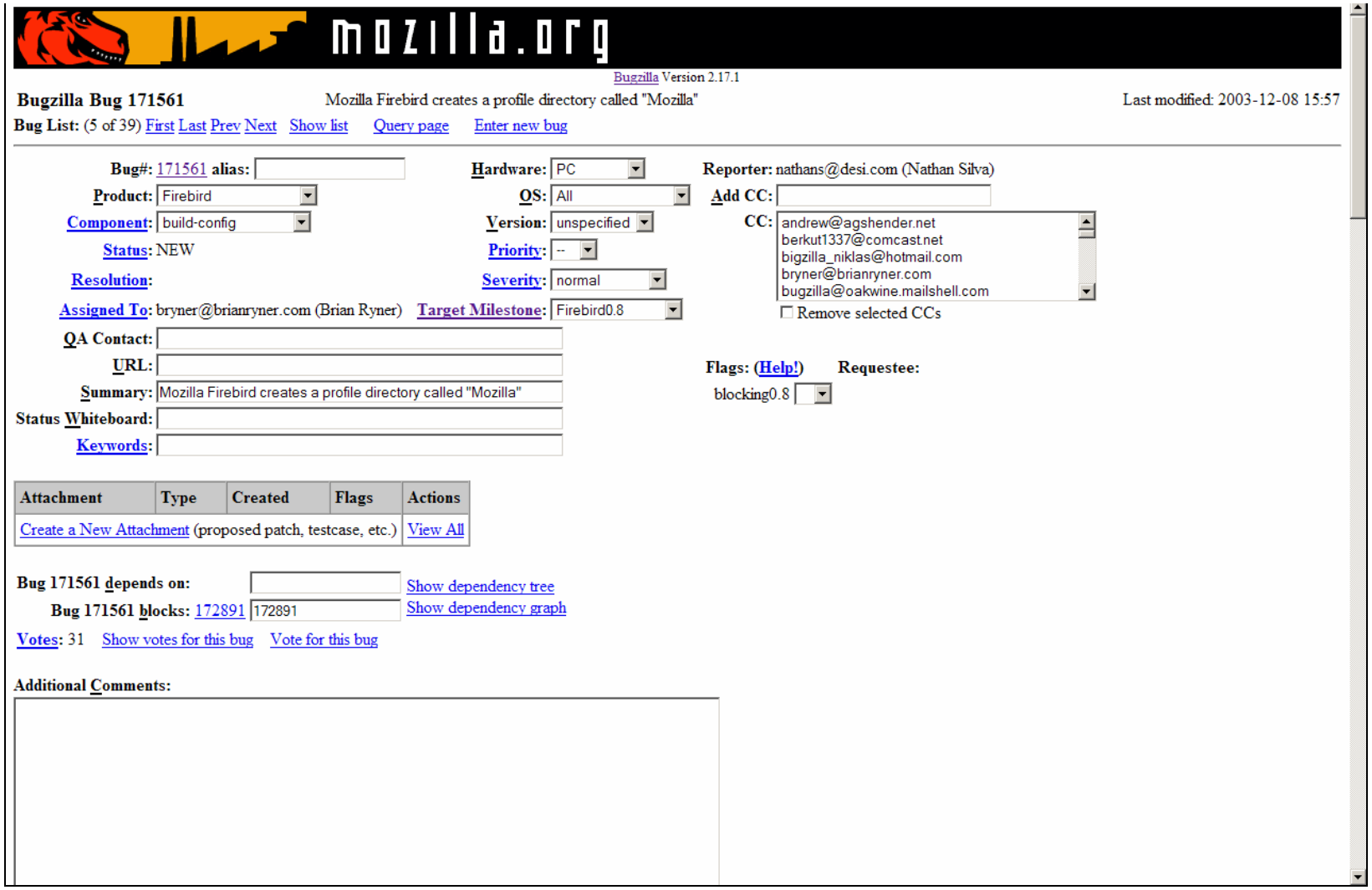
Phoenix, 12/9 20:23
- WINNT 5.0 gabrielle Depend
- MacOSX Darwin 6.8 imola Depend
- Linux redwood Depend

The tree is open

Build Time	Guilty	Linux brad Clbr (info)	Linux btek Dep	Linux comet Dep	Linux luna Dep	MacOSX Darwin 6.3 silverstone Clbr	MacOSX Darwin 6.6 monkey Dep	WINNT 5.0 beast Dep	WINNT 5.0 creature Clbr
Click time to see changes since then	Click name to see what they did								
12/09 20:38		L/	L/	L Z:19.95MB mZ:12.53MB Txul:419ms Ts:1376ms	L/	L/ C	L/	L/	L/ C
20:29								L Z:14.64MB mZ:8.126MB Txul:313ms Ts:953ms	
20:23									
20:21									
20:20			L Tp:941ms						
20:15									
20:12					L C Z:19.20MB Zdiff:+184 (+189/-5) mZ:12.13MB mZdiff:+184 (+189/-5) Tp:963ms Txul:861ms Ts:3304ms		L C Tp:473ms Txul:476ms Ts:6377ms	L C Z:14.64MB Zdiff:+132 (+142/-10) mZ:8.126MB mZdiff:+132 (+142/-10) Txul:312ms Ts:812ms	
20:01			L C Tp:950ms						
12/09 19:55									
19:52			L Tp:943ms						
19:50									
19:48									
19:43	roc+						L Tp:473ms	L Z:14.64MB mZ:8.126MB Txul:313ms Ts:843ms	
19:39									

Tinderbox is a tool used by the Mozilla development team which acts as an interface to Bonsai. It allows the developers to see what is happening to the source by showing who submitted what code and what file versions went in to a particular build of the software (Hamerly, 1999).

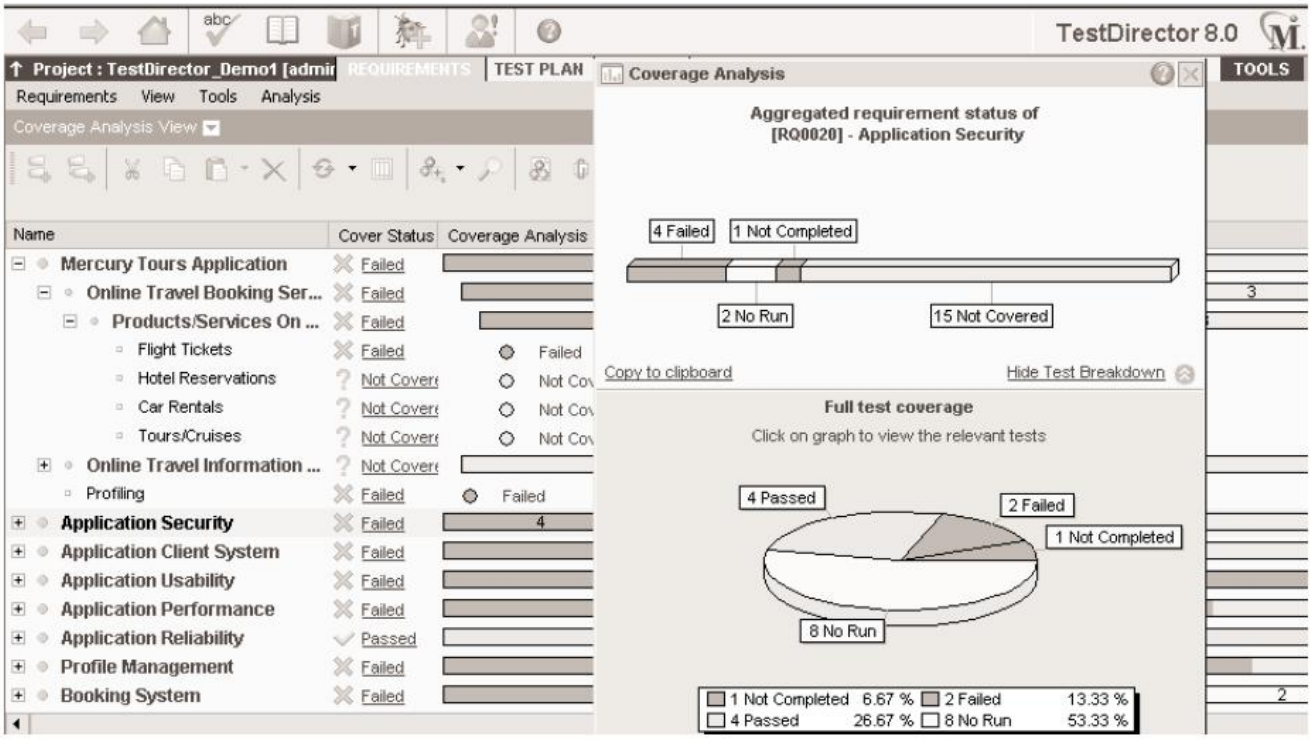
This screen shot shows the SeaMonkey source tree, which is the main suite within the Mozilla family of products. It shows the times at which various files of source code were checked-in, as well as the file size and author. This tool may be accessed at http://tinderbox.mozilla.org/.



Bugzilla is a defect tracking system created by the Mozilla development team. It allows bugs to be reported and commented on. In addition, a specific person can be assigned to fix a problem, and dependencies and interactions between related bugs can be noted (Mozilla Organization *Bugzilla* 2003).

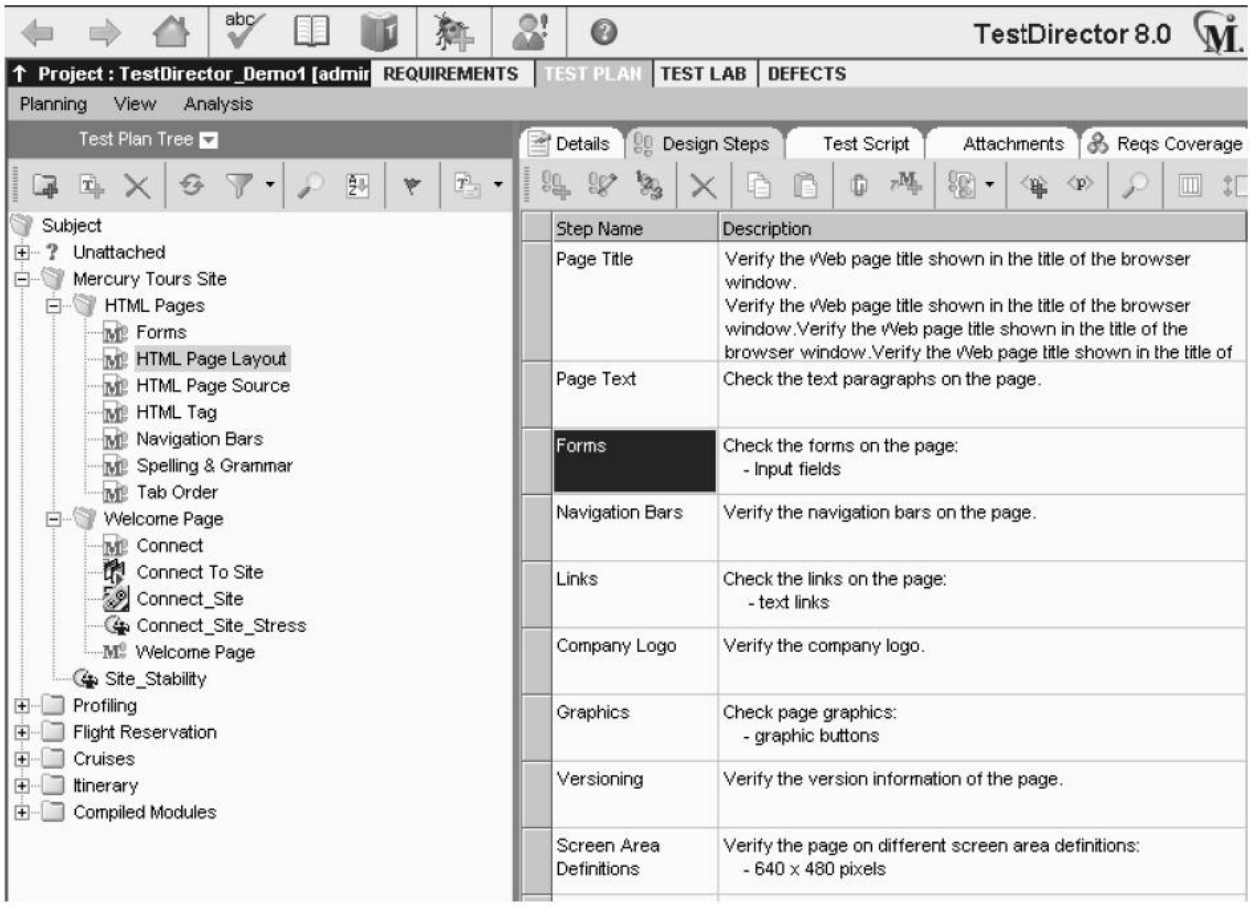
This screen shot shows a bug report for Firebird, the next generation browser from Mozilla. It displays a summary of the bug, who has been assigned to fix it, and the target fix time. Below this information, discussion on the bug is available in the form of a message board (not shown). This tool may be accessed at http://www.bugzilla.org.

APPENDIX E: REQUIREMENT MANAGER IN TESTDIRECTOR



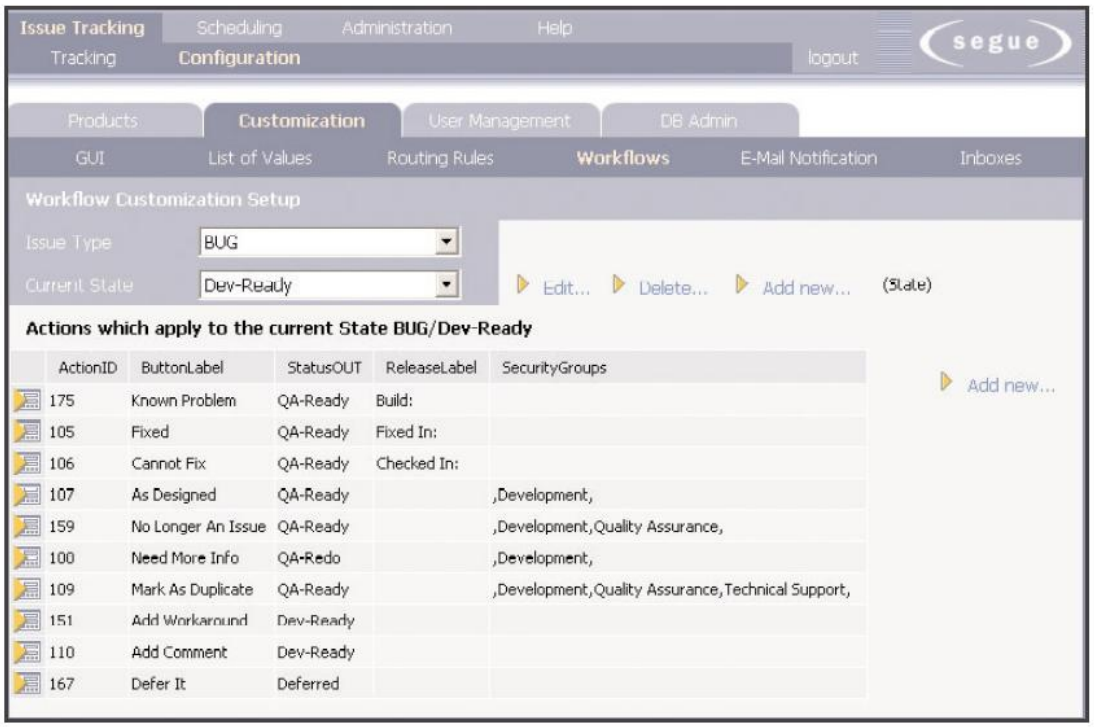
TestDirector's Requirements Manager links test cases to testing requirements, ensuring traceability.

APPENDIX F: TEST PLAN TREE IN TESTDIRECTOR



The test plan tree in TestDirector is a graphical representation of the organization's test plan.

APPENDIX G: CUSTOMIZABLE ACTION-DRIVEN WORKFLOW



APPENDIX H: DEFINE CUSTOM VIEWS