Maintenance of Hybrid Software that Combines Open Source and Closed Source Components

by

Xiangyi (Cindy) Xu

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Applied Science in Technology Innovation Management

Department of Systems and Computer Engineering

Carleton University

Ottawa, Canada, K1S 5B6

January 2011

© Copyright 2011 Xiangyi (Cindy) Xu
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Canadian
Abstract

The maintenance of hybrid software, software that contains both open source components and closed source components, has barely been studied. This thesis examines how the maintenance effort is affected by the architecture of the hybrid software and the characteristics of the open source components it uses.

Through empirical studies of three industrial case studies, two sets of propositions were developed. The first group of propositions reveals the effects of characteristics of open source components and the architectural relationship of open source components and closed source components on the maintenance effort of hybrid software. The second group of propositions provides suggestions for project managers of hybrid software to help reduce the maintenance effort.
# Table of Contents

Abstract ........................................................................................................................................ i

Table of Contents ......................................................................................................................... ii

Figures .......................................................................................................................................... iv

Tables ........................................................................................................................................... v

1. INTRODUCTION ................................................................................................................... 1
   1.1 Objective ............................................................................................................................. 1
   1.2 Relevance ............................................................................................................................ 2
   1.3 Deliverables ........................................................................................................................ 3
   1.4 Contributions ...................................................................................................................... 4
   1.5 Organization ....................................................................................................................... 5

2. BACKGROUND INFORMATION ............................................................................................ 6
   2.1 Research scope .................................................................................................................. 6
   2.2 Software maintenance ...................................................................................................... 9
       2.2.1 Classifications of software maintenance ................................................................. 10
       2.2.2 Software maintenance measurement ....................................................................... 11
   2.3 Component-based software development ...................................................................... 13
       2.3.1 What is component-based software development .................................................. 13
       2.3.2 Advantages of component-based system development ......................................... 14
       2.3.3 Challenges of component-based system development .......................................... 15

3. LITERATURE REVIEW ......................................................................................................... 16
   3.1 Maintenance of component-based software ................................................................... 16
       3.1.1 Challenges of maintaining component-based software ......................................... 16
       3.1.2 Measuring maintenance of component-based software ......................................... 17
   3.2 Maintenance of hybrid software ...................................................................................... 22
   3.3 Unique characteristics of FLOSS .................................................................................... 24
       3.3.1 Structure of an open source community ................................................................. 24
       3.3.2 Communications in open source communities ......................................................... 25
       3.3.3 Virtual organization and virtual project management ............................................ 27
       3.3.4 Innovation and evolutionary patterns of open source software ............................. 29
3.3.5 Licenses........................................................................................................... 31

4. RESEARCH DESIGN....................................................................................... 33

4.1 Selecting cases ............................................................................................. 33

4.2 Theoretical framework.................................................................................. 34

4.2.1 Components of software maintenance.................................................... 34

4.2.2 Change-based studies.............................................................................. 35

4.3 Data collection .............................................................................................. 37

4.3.1 Sources of the data .................................................................................. 38

4.3.2 Data collected for each case.................................................................... 39

4.4 Data analysis ................................................................................................ 41

4.5 Results ......................................................................................................... 41

4.5.1 Cases classification .................................................................................. 41

4.5.2 Proposition development ....................................................................... 43

4.5.2.1 Horizontal analysis (A)....................................................................... 44

4.5.2.2 Vertical analysis (B) .......................................................................... 49

4.6 Summary ..................................................................................................... 54

5. CONCLUSION ............................................................................................... 56

6. REFERENCES ................................................................................................. 59

7. APPENDICES ................................................................................................. 63

Appendix A. Lines of code for the cases .......................................................... 63

Appendix B. Results of change task analysis for 3 cases ............................... 64
Figures

Figure 2-1. Research scope of this paper ................................................................. 9
Figure 2-2. Matrix of purpose and measurement of software maintenance .............. 13
Figure 3-1. General structure of a FLOSS community (Nakakoji et al., 2002) .......... 25
Figure 3-2. Open source development team communications tools (Barnett, 2004) .... 26
Figure 3-3. Evolutionary patterns of the projects in (Nakakoji et al., 2002) ............ 30
Figure 4-1. A conceptual model for change-based studies (Benestad et al., 2009) ...... 36
Figure 4-2. Software architecture of the three cases ............................................... 43
Figure 4-3. Linux evolutionary pattern – forking (Nakakoji et al., 2002) ............... 46
Tables

Table 3-1. Summary of literature on measuring the maintenance of CBS ...................... 22
Table 3-2. Types of projects and their community structures (Nakakoji et al., 2002) ..... 28
Table 4-1: Components of a software maintenance framework (Grub et al., 2003) .... 34
Table 4-2. Characteristics of the FLOSS Component Adopted by the 3 Cases .......... 42
Table 4-3. Summary of propositions ........................................................................ 55
Table 7-1. Results of lines of code for case 1 (Low layer) ........................................ 63
Table 7-2. Results of lines of code for case 1 (High layer) ...................................... 63
Table 7-3. Results of lines of code for case 2 ......................................................... 64
Table 7-4. Results of lines of code for case 3 ....................................................... 64
Table 7-5. Results of change task analysis .............................................................. 65
1. INTRODUCTION

Software systems that are used actively need to be changed continuously. Making changes to existing software products is called software maintenance. It is the last step of the software life cycle, and also it is the most expensive step within the whole software life cycle. Results from studies undertaken to investigate the characteristics and costs of changes carried out on a system after delivery show estimated expenditure at 40%-70% of the costs of the entire life-cycle of the software system (Grub et al., 2003). The management and control of software maintenance is one of the greatest challenges facing software engineers (Grub et al., 2003). This is clearly demonstrated by the time spent and effort required to keep software systems operational after release. Understanding, managing, and reducing costs and risks of software maintenance and evolution are important goals for both research and practice in software engineering (Benestad et al., 2009).

1.1 Objective

There are many studies of component-based software maintenance from all aspects, for closed source software or open source software, respectively. There has not been much research specific to the maintenance of software that contains both open source components and closed source software components, or hybrid software. Using
open source components in proprietary software development brings about benefits such as lower development cost and time to market, and a higher starting point for innovation or technology. However at the same time, it introduces unique maintenance challenges that have not been studied thoroughly. In this study, we will investigate some of these challenges and solutions. The objective of this thesis is to examine how the maintenance effort is affected by the architecture of the hybrid software and the characteristics of open source components used in the software.

1.2 Relevance

These questions are important for project managers who are considering integrating open source components into their software product to provide new features. Answering these questions will help them:

1) Consider the cost and effort as a whole for both development and maintenance to make the right decision of whether they want to go down that path.
2) Understand what they should expect, if they decide to make use of open source components.
3) Plan in advance to make adjustments at an early stage of software design to make sure a hybrid software product is maintainable.

As for project managers who have been involved in hybrid software development, answering these questions will give them:
1) Theoretical framework for understanding the root cause of some of the maintenance challenges they are fighting day by day.

2) Guidelines to make the maintenance of hybrid software easier. Consequently, reduce the cost of software maintenance of such hybrid software.

This research will also interest project managers of open source projects. Enabling their open source projects to be more broadly used, especially by commercial users, is always a goal of open source project managers. Answering these questions will help them:

1) Understand what maintenance challenges proprietary software project managers using open source components are confronted with.

2) Examine what they can do to increase the usability and integrability for the open source projects for commercial users.

1.3 Deliverables

First of all, 3 cases were identified based on the criteria being pre-defined. Every case represents one real life project that has integrated with open source component and in-house developed closed source components. Then, we closely examined the maintenance of these three representative hybrid software systems for over 4 years, which is long enough to expose a number of maintenance problems. After that, multi-aspect and rich data was collected through data mining of internal databases and interviews. The data was then analyzed using the change-based analysis mechanism, which has been proved to
be capable of identifying and understanding factors that affect change effort during maintenance and evolution (Benestad et al., 2009). We used layered architectures to analyze the architecture of each project. Four layers were defined for the projects.

As the result, first of all, the three projects were categorized at the component level and architecture level. It was found the open source component adopted by each project has its own unique characteristics from the other ones. Also it was found each of the three projects has a unique architectural relationship between the open source components and closed source components. Secondly, 2 sets of propositions were developed based on the analysis results and conclusions were drawn accordingly.

1.4 Contributions

This research fills in the research gap of software maintenance that has been identified – software maintenance of hybrid software.

Through empirical studies of three industrial case studies, two sets of propositions were developed. The propositions indicated that the architectural relationship between open source components and closed source components, the maturity and scope of usage of the FLOSS component are affecting the total maintenance effort, average maintenance effort per task and the types of maintenance tasks. Also, propositions provided suggestions for hybrid project managers to consider while designing and maintaining hybrid software, which will help reduce the maintenance effort and cost.
1.5 Organization

The remainder of this thesis is organized as follows. Section 2 introduces some background information about this research. Section 3 provides a brief overview of research in component-based software maintenance and the unique characteristics of open source software. Section 4 describes the research method and the results. Finally, Section 5 draws conclusions and provides suggestions for future work.
2. BACKGROUND INFORMATION

2.1 Research scope

As open source software becomes more popular, more and more companies, including large companies, choose to integrate open source components into their proprietary commercial software. The major motivation for using open source components is to reduce development cost and time to market. As a result of such movement, hybrid software, which refers to software containing both open source components and closed source components, is gaining momentum in the software industry. An ActiveState (2008) white paper states that “even if you wanted to, it would be nearly impossible to run a software business using exclusively open source or proprietary software” and that “by 2012, 90% of commercial software will contain significant amounts of open source code, much of it coming in through embedded applications.” Sun's acquisition of MySQL is an example where closed source and open source software are combined in a product.

Proprietary software is defined by the GNU Project (2010) as “software that is not free or semi-free”. To use, redistribute or modify the software is restricted in certain way, which is defined by the licensing scheme. Proprietary software is also described as “closed source”, “non-free”, and “non-FLOSS”. There are many cases where the terms proprietary software and commercial software are used interchangeably – in company white papers, on websites, and even in some academic research. But the fact is: proprietary software does not fully equal to commercial software. Interestingly enough, it was discovered that over 90% proprietarily developed software is built without
commercial purpose or for some reason did not turn into commercial usage in the end (Wheeler, 2006). In the following chapters, the terms proprietary software and closed source software will be used interchangeably.

According to Wheeler (2006), open source software, or more precisely, FLOSS (Free-Libre/Open Source Software) is described as software whose licenses give users the freedom to run the program for any purpose, to study, to modify, and to redistribute copies of either the original or modified program without paying royalties to previous developers. It is apparent from this terminology that FLOSS includes two aspects: free and open source. Open source does not necessarily mean free; and free software does not necessarily mean open source. The GNU Project (2010) states that free software is a matter of the users' freedom to run, copy, distribute, study, change and improve. Open source software explicitly refers to software or programs that make their source code available to the public and allow their distribution in source as well as compiled form. However, since the difference between free source software and open source software is small, we will use the terms free software and open source software interchangeably in this thesis.

As Wheeler (2006) points out, treating open source and commercial as opposites is “fundamentally wrong.” There are enough examples already: FLOSS has been commercialized and indeed, very successfully. Wheeler (2006) recounts those companies such as Red Hat, Novell/SuSE, and others are selling their commercial distributions. Also, Red Hathas been listed on the NY Stock Exchange for years. Its income primarily depends on selling FLOSS products and providing support. This fully proves that open source have real commercial value. Asslett (2008) points out that Fortune magazine
published a list of its hot IPO tips for 2008. It was found three out of the five IPO companies, MySQL, Ingres and SugarCRM, are open source companies, while another, Parallels, is an open source project sponsor. In fact, after MySQL decided to go for an IPO, Sun Micro purchased MySQL for 1 billion US dollars. All of these evidences prove that there is no conflict between software being open source and its commercialization.

Based on previous research, we can conclude that closed source and open source are just two strategies of software development. Each has its own unique development practices. Whether or not software will be made into commercial software is another story.

With increased maturity and wider acceptance of open source, a new type of software was created, which contains both open source components and closed source components. The closed source components are either developed in house or bought from third party software component providers. This type of software is developed by proprietary companies. They choose open source components with the purpose to reduce their development cost and time. We call such software hybrid software. This type of software makes the definition of FLOSS more complicated. As the growth of FLOSS itself, it is no doubt that more and more proprietary companies will integrate open source components into their product lines. Companies, including large companies like IBM or HP have embraced this strategy already.

The debate about drawing clear lines between proprietary (or closed source) software, open source software and commercial software is far from a conclusion. The research scope of this thesis is commercial software that contains both closed source components and open source components, which we call hybrid software, as shown by the grey area in Figure 2-1.
2.2 Software maintenance

Edelstein (1993) defines software maintenance as "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."

According to traditional software engineering models, software maintenance is the last stage of the software life cycle and it mostly happens in serial after all the other phases like requirement engineering, design, implementation, testing and deployment. Software maintenance is a very important stage as it is the step to make sure software systems are functional, flexible, continuous available and correct operation.

The cost of software maintenance is high. Many studies estimate the cost of changes carried out on a system after delivery at 40–70% of the costs over the entire life-cycle of the system (Grub et al., 2003). Bernstein (1993) judges that out of $100 billion spent on software systems at least $70 billion is spent on system maintenance.
2.2.1 Classifications of software maintenance

Based on the characteristics of the changes being made through the software maintenance process, many researchers have classified software maintenance research accordingly. Changes have been consistently classified into 4 types (Grub et al. 2003):

1) Corrective maintenance: Maintenance performed to correct faults in hardware or software.

2) Adaptive maintenance: Software maintenance performed to make a computer program usable in a changed environment.

3) Perfective maintenance: Changes undertaken to expand the existing requirements of a system.

4) Preventive maintenance: Changes undertaken to prevent malfunctions or to improve maintainability of the software.

From the maintenance activity perspective, as P. Stafford (2003) explains: 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; and preventive maintenance deals with updating documentation and making the software more maintainable. All changes to the system can be characterized by one of these four types of maintenance. Corrective maintenance is considered as ‘traditional maintenance’ while the other types are considered as ‘software evolution’.

The proportions of maintenance effort for each type of maintenance are not the same. According to Lientz et al. (1980), who studied 487 data processing organizations:
corrective maintenance accounted for 20%, adaptive maintenance consumed 25%,
perfective maintenance accounted for 50%, and the remaining 5% was spent on
preventive maintenance (Lientz et al., 1980).

### 2.2.2 Software maintenance measurement

As for software project managers, they are always faced with the costly activities of
“managing, controlling and making the changes necessary to allow software systems to
evolve” (Grub et al., 2003). To reduce the cost of maintenance is a goal of every software
project manager.

However, often cost is difficult to estimate. It relies on other measurements.
Maintenance effort and maintainability are the two measurements that being used the
most to help estimate the cost associated with the maintenance job. Many models have
been developed to measure either maintainability or maintenance effort. On the other
dimension, from the perspective of purpose of measuring, assessment and prediction are
two broad purposes of maintenance measuring (Fenton, 1994). They are highly
interdependent. Figure 2-2 represents the matrix of the purpose and measurements for
measuring software maintenance. Assessment of the maintenance effort (grey area) is that
the area we will focus on in this thesis. Our goal is to examine the maintenance activities
that have been implemented to find out correlations of some factors and maintenance
effort.

1) Maintainability

Software maintainability is a measurement that indicates how maintainable a
piece of software is. It is defined as the “ease with which a software system or component
can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment" (IEEE, 1990). On one hand, higher maintainability of the software indicates lower maintenance cost; on the other hand, lower maintainability indicates higher software maintenance cost.

There is even an international standard for the evaluation of software maintainability. ISO 9126 part 1 represents the latest research in software quality control, software quality assurance and software process improvement. It defined maintainability as one of the six main software quality characteristics, which includes four sub-characteristics: adaptability, changeability, stability and testability.

Peercy (1981) sees the primary objective of software maintainability evaluation to “collect enough specific information to identify for which parts of the software and for what reasons maintainability may be a problem”. A secondary objective is to “assess the effectiveness of the evaluation process itself”. It is to help validate maintainability scores against an actual field maintenance level of effort.

2) Maintenance effort

Maintenance effort is a straightforward concept. It is normally measured in person-months or person-days. Maintenance effort is considered as one of the most important issues that should be kept as low as possible.
2.3 Component-based software development

Component-based software development significantly enhances software development and use (Brereton et al., 2000). In today’s software industry, with increased size and complexity of software products a single application is commonly made up of components produced by different developers using different practices as opposed to developed completely in-house from bottom up (Ardimento et al., 2007). Components can be developed internally or acquired externally from third parties, either as Commercial off the Shelf (COTS) or open source components.

2.3.1 What is component-based software development

The term “software component” has varying definitions. Software component was defined by Szyperski (1998) as a “unit of composition with contractually specified interfaces and explicit context dependencies only”. A software component can be deployed independently and is subject to be integrated with other software components by third parties. Brown et al. (1998), instead, view a software component as a nontrivial
and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture.

Component-based Software Development (CBSD) is a software engineering process to integrate existing components into a software system, with a potential benefit of delivering quality systems within a timely fashion by using quality, finished components. Those software components can be commercial off-the-shelf, public domain, freeware, shareware, or open source (Jeffrey, 1998).

The idea of CBSD is not new. It was introduced as early as 1968, by Mellroy (1996) in his address “Mass Produced Software Components” at the first NATO conference on software engineering in Garmisch, Germany. He advocated that the software industry should, like the hardware industry, be “industrialized.” As he pointed out, “software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance” (Douglas, 1969). However, at the time, this type of software development process was not well accepted due to lack of available components and integration technologies. It was within the last several years, along with the popularity of Object-Oriented Programming (OOP), CBSD was made possible.

2.3.2 Advantages of component-based system development

The emphasis of CBSD is integration, departing from the conventional software development process which is development centric. CBSD can make a software organization more productive, reduce its time to market, the cost of development and increase system quality (Sajjad et al., 2004).
Component-based system development has greatly helped software development technology get mature enough to overcome many of the technical issues involved, putting us in a position from where to start effectively tackling many of the quality aspects of software development. However, component-based system development does introduce new challenges (Sajjad et al., 2004).

2.3.3 Challenges of component-based system development

Padmal (2003) stresses that challenges and advantages co-exist within CBSD. He analyzed the challenges for the various CBSD stakeholders and identified three primary types of stakeholders: component developers, application assemblers and customers. Each group of stakeholders is facing different set of challenges from the other groups. For example: component developers have to accommodate the requirements from a broad range of domains to make the component suitable for the mass-market; application assemblers need to manage and maintain the components from several parties being used in the same system; customers are at risk of not having the expected functionalities or qualities. Padmal (2003) found that “risks faced by one stakeholder are transferred to the next, ultimately constraining each customers ability to leverage component technology in developing its application systems.”
3. LITERATURE REVIEW

3.1 Maintenance of component-based software

Component-based software development does not only bring about challenges to the software development, as we saw in chapter 2.3.3; more predominately, it brings about new challenges to the maintenance of such software for all stakeholders.

3.1.1 Challenges of maintaining component-based software

Component developers need to “think not just of maintaining a block of source code in a specific application, but of maintaining code that is reused in numerous customer applications” (Voas, 1998). This is because that each application may have slightly different requirements; consequently, component modifications may not work for all applications.

Customers of component-based systems risk not having their issues solved in time due to the maintenance challenges that system integrators are facing or the interaction between component vendors and system integrators.

System integrators have to maintain code that is not natively developed which they are not familiar with or not having internal domain experts. Even worse, sometimes, they have to maintain “black box” components, which limit visibility to documentation that describes the component’s operation and functionality. However, it is critical to understand component behavior during the evolution of components and applications, since maintenance is often necessitated because of changes in components (versions, availability, etc), changes of requirements and a host of other reasons (Andrews et al.,
In addition to maintaining components, system integrators will need to maintain the "interoperability" of components. Voas (1998) identified the maintenance challenges raised by component-based system like frozen functionality, incompatible upgrades, unreliable components and complex middleware.

The maintenance activities for system integrators range from component replacement to troubleshooting and configuration management. Key maintenance activities for the component-based system include component reconfiguration, configuration management, system tailoring, system monitoring, troubleshooting and repair (Sajjad et al., 2004).

In this thesis, we will focus on studying the maintenance challenges that system integrators are facing in the context of hybrid software, that contains both closed source software components and open source components.

3.1.2 Measuring maintenance of component-based software

Component-based software quality assurance and assessment are non-trivial tasks as they do not only depend on the qualities of the components involved, but also on the qualities of the framework and the integration process (Mahmooda et al., 2004).

Mari et al. (2003) introduce a framework of maintainability and techniques that promote maintainability for component-based systems. They argue that the maintainability of component-based systems needs to be considered at three abstraction levels: system, architecture and component. In system dimension, the maintainability requirement is considered from a business related point of view. In architecture, maintainability means a set of quality attributes, e.g. extensibility and flexibility. At the
component level, maintainability focuses on modifiability, reusability, integrability and testability.

Based on Mari et al. (2003)’s framework, we conclude that software components affect the maintainability of the whole system from component level as well as architecture level. However, much research regarding the maintenance of component-based software has only focused on component-level analysis or using traditional ways of analyzing the nature of the code in a hope to find correlations of component characteristics and maintainability or maintenance effort, without considerations or analysis of architecture factors. Table 3-1 summarizes the literature in this area.

1) Literature focused on component level and code natures

As Sajjad et al. (2004) have pointed out, measuring and assessing the quality attributes of a component-based system has changed from the traditional way, given that the quality of an individual component can influence the quality of the overall system. We can argue that maintainability of the overall system, as one of the six main software quality characteristics, is affected by the quality of its components.

Niessink et al. (1998) present two measurement procedures, their characteristics, the data gathered, and an analysis of the datasets. They used principal component analysis and multiple regression analysis to analyze the datasets. They identified several key factors that driving the software maintenance effort: amount of control-flow changes, increase in size, and algorithmic change.

Based on an empirical study of three industrial projects, Ardimento et al. (2004) report three significant correlations between the characteristics of a component itself and the maintainability of the overall system: time spent on training people is positively
correlated to the values of the mean effort spent for maintenance activities; a positive
correlation exists between functional coverage and maintenance effort; and a negative
correlation exists between maintenance effort and the level of support provided by the
vendor of the component. Based on these conclusions, they found that:

- Functionality of each component should be as concentrated as possible over a
  single aspect of the application domain.

- Training time offered by the component producer usually indicates the complexity
  of understanding it and if a component is difficult to understand, then it is also
difficult to maintain.

- Deep knowledge of the component is necessary for the organization before its
  adoption; therefore, a trial usage of components is advised before the final
decision about their adoption.

Again, in 2007, Ardimento (2007) completed another empirical study of 22
projects aimed at understanding how component characteristics affect CBSs
maintainability. The obtained results show that many characteristics included in the
framework are not meaningful, which indicates the failure of developing a framework
However, this study does further test the correlations of some characteristics of
components with the maintainability of the overall systems. More interestingly, they
analyzed several characteristics specifically to open source components. They discovered
that out of all variables they measured for open source components, only release rating is
correlated with the maintainability of the CBS. This can be explained as the higher the
number of published releases of a component, the higher the probability that new releases
introduce new functionalities, improve existing ones and fix bugs of previous releases,
reducing the effort spent for satisfying maintenance requests, which represents higher maintainability.

Aggarwal et al. (2005) predict the maintainability of the object-oriented (OO) systems through an Artificial Neural Network (ANN) based approach. They consider the principal components of eight OO metrics as independent variables, which include Lack of Cohesion, Number of Children, Depth of Inheritance and others. Maintainability was considered as a dependent variable. Results show that independent variables chosen for the study were able to predict the maintenance effort.

Shukla et al. (2008) adopted a similar approach – a neural network based approach to estimate software maintenance efforts. They chose 14 factors as cost drivers for their study and conducted the experiment by taking various options of number of hidden layers and number of hidden nodes. Input data selected for training was 60% of the total, while 20% each were used for validation and testing. MRE percentage (Mean Relative Estimation Error %) obtained from the experiment was around 5%. Results concluded that neural network was able to successfully model the maintenance effort.

2) Literature with architecture focus

Sajjad et al. (2004) emphasizes that the properties of a system are influenced more by the interaction of its components than by the properties of a single component. So the prediction of properties of component-based software cannot be done successfully without the consideration and analysis of architecture. However, not much research has been done predicting the maintainability with architecture analysis specifically for component-based software.
Software architecture is defined by Bass et al. (1998): The software architecture of a program or computer system refers to the structure of the system, which comprises software components, externally visible properties of those components, and the relationships among them.

Bengtsson et al. (1999) presented a method predicting software maintainability while designing software architecture. The paper proposed a method for software architecture analysis of modifiability based on scenarios. This method estimated the required maintenance effort of the system built based on the software architecture based on four inputs: (1) requirement specification, (2) design of the architecture (3) expertise from software engineers and, possibly, and (4) historical maintenance data. However, this research did not identify any characteristics of the architecture would increase or decrease the maintainability.

Arsanjani et al. (2002), through studying over 15 projects, summarizes the main and most common challenges and problems of maintaining component-based software are fourfold: (1) difficulty in providing harmonious functionality while integrating new or legacy components with legacy components; (2) architectural entropy, which disallows the application of cost-effective, non-intrusive changes as software evolves to meet changing requirements; (3) difficulty of managing the evolution of the heterogeneous distributed application portfolios; and (4) functional duplication across the application portfolio. They suggest that these challenges can be addressed a variation-oriented design process. The approach is based on defining a uniform mechanism for the management of component boundaries between systems. The process includes the identification of requirements for the behavior of a system and its components, formalizing and
abstracting them into the grammar of a domain-specific language. This approach enables a highly re-configurable architectural style to help build and maintain reusable components that are responsive and resilient to changing requirements.

<table>
<thead>
<tr>
<th>Without architecture analysis</th>
<th>Author</th>
<th>Method used</th>
<th>Number of projects</th>
<th>Measuring/Predicting</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ardimento et al., 2004</td>
<td>Empirical study</td>
<td>3</td>
<td>Maintainability</td>
<td>3 correlations identified</td>
<td></td>
</tr>
<tr>
<td>Ardimento et al., 2007</td>
<td>Empirical study</td>
<td>22</td>
<td>Maintainability</td>
<td>Framework not meaningful</td>
<td></td>
</tr>
<tr>
<td>Aggarwal et al., 2005</td>
<td>Artificial Neural Network (ANN) based</td>
<td>1</td>
<td>Maintenance effort</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Shukla et al., 2008</td>
<td>Neural network based</td>
<td>36</td>
<td>Maintenance effort</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>F. Niessink et al., 1998</td>
<td>Principal component analysis and multiple</td>
<td>2</td>
<td>Maintenance effort</td>
<td>Key factors identified</td>
<td></td>
</tr>
<tr>
<td>With architecture analysis</td>
<td>Bengtsson et al. 1999</td>
<td>Scenario based case study</td>
<td>Maintainability</td>
<td>Method to predict maintainability</td>
<td></td>
</tr>
<tr>
<td>Arsanjani et al. 2002</td>
<td>Case studies to identify root causes of key maintenance problems</td>
<td>15</td>
<td>Maintainability</td>
<td>Main challenges identified</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-1. Summary of literature on measuring the maintenance of CBS

3.2 Maintenance of hybrid software

Some think that open source software components are just software whose source code is freely available. When integrating open source software into their proprietary software, they consider the open source part to be no different than closed source components purchased from third party software companies, with the exception that the source code of the open source components is available from the start. It is probably true that using
open source saved a lot of development time and expenses by not having to purchase commercial components. However, problems were discovered while maintaining such software: not being able to find support; key new features being difficult to add; and not being able to upgrade to newer versions of open source projects due to architectural or systematic limitations.

A number of studies proposed structured, formal, or semiformal selection procedures that suggest various attributes to consider when choosing an open source component. However, not much research has been done to identify potential maintenance problems that could be brought about by not integrating open source components properly in the component-based software development. Ardimento et al. (2007) identified several unique characteristics of open source components in addition to other generic characteristics identified for software components that can potentially affect the maintenance of component-based software. Issues that were not considered during system integration may arise during maintenance. For example, system maintainers may not be familiar with some parts of a component. Changes to the open source component require significant insight into its implementation. These problems become more urgent as components evolve: there is no mechanism to force the developers of the open source component to change it in a specific way.

We have not found any empirical studies on software maintenance of hybrid software, that is, software that contains both open source components and closed source components. And we have not found any research examining the effect of the structural relationship between the open source components and closed source components on the maintenance effort.
3.3 Unique characteristics of FLOSS

What is unique about open source components compared to closed source components? In this chapter, we will identify the ways in which FLOSS components differ from closed source software components in order to be able to later analyze the effects of certain characteristics of FLOSS components on the maintenance effort of hybrid software.

3.3.1 Structure of an open source community

Most open source software is surrounded by a community, which includes the original developers, users and co-developers. Without a community, an open source project is not likely to be successful” (Nakakoji et al., 2002). Research on open source projects is incomplete without studying the associated communities.

Nakakoji et al. (2002) suggest the general construction of open source communities contains: project leader, core member, active developer, peripheral developer, bug fixer, bug reporter, reader and passive user, as shown in Figure 3-1. Not all of these types of members need to exist in any particular FLOSS community. The percentages of each type vary from project to project (Nakakoji et al., 2002). Typically, open source projects consist of a small number of core members, a much larger number of non-core developers like active developers/users, and a vast pool of users (Barnett, 2004).
Community members may function as multiple roles within a particular project, for example, a bug reporter could also be a bug fixer (Scacchi, 2004).

3.3.2 Communications in open source communities

The existence of a community allows efficient communication from all parties involved. Such communication brings about two obvious advantages:

1) Open source developers are themselves often end-users of their systems. Thus, less time is required to define requirements and design in comparison to traditional software engineering projects (Scacchi, June 2004).

2) As the source code of the projects is visible to the whole community, it enables peer code review from any member of the community. Linus Torvalds observed that a large user base can be more valuable (in terms of maintaining a high level of quality) than a large developer base. Similarly, Eric Raymond summarized that “given enough eyeballs, all bugs are shallow” (Barnett, 2004).
The formation and enactment of complex open source software development are performed by loosely coordinated software developers and contributors who may be globally dispersed (Scacchi, 2006). The fundamental technology that makes this happen is the Internet. Open source development and maintenance strongly rely on the Internet.

All steps in open source development, from requirement analysis, specification to version control, system build, and maintenance are implemented over Internet. It can be said that without the Internet the development of open source would not be possible. The whole procedure of open source development is “facilitated by communication and knowledge sharing infrastructures such as persistent chat rooms, newsgroup, issue reporting/tracking repositories, sharable design representations and many kinds of software informalisms” (Scacchi, 2003). “Informalisms” (Scacchi, 2003) refer to information sharing mechanisms that allow information to be publicly accessed, browsed, hyperlinked and updated on demand. Figure 3-2 gives some good examples of tools that are used by open source development teams to communicate.

<table>
<thead>
<tr>
<th>Open source team communications tools</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collaborative environment</strong></td>
</tr>
<tr>
<td>SourceForge</td>
</tr>
<tr>
<td><strong>Technical discussion</strong></td>
</tr>
<tr>
<td>Wikis, mailing lists, Web sites, FAQs</td>
</tr>
<tr>
<td><strong>Version control</strong></td>
</tr>
<tr>
<td>CVS and Subversion</td>
</tr>
<tr>
<td><strong>Issue tracking</strong></td>
</tr>
<tr>
<td>Bugzilla, Scarab</td>
</tr>
<tr>
<td><strong>Build systems</strong></td>
</tr>
<tr>
<td>Anthill, CruiseControl, Gump, Make, Maven, Tinderbox</td>
</tr>
</tbody>
</table>

**Figure 3-2. Open source development team communications tools (Barnett, 2004)**
3.3.3 Virtual organization and virtual project management

There is not strict boundary of open source organization. Margaret et al. (2003) found that “the beliefs in freedom, free software, and freedom of choice create a special bond for the people working on free software projects. These beliefs foster the values of cooperative work and community-building.” In most of open source development, the belief in free software and belief in freedom of choice are part of the virtual organizational culture. Unlike proprietary projects, which have a designated project manager and clear hierarchy, open source projects normally have a relative loose organization. The popular literature has described open source developers as members of a “geek” culture (Pavlicek, 2000), well-known for being technically savvy, but socially inept, and as participants in a “gift” culture (Berquist et al., 2001), in which the social status of individuals is measured by what they give away freely. In such a “free” environment, developers have control over what utilities and communication mechanisms to use, the tasks and roles they want to pursue, even the delivery schedule. Consequently, there is no single point of contact that controls budget, feature list, schedule, etc.

The management of open source projects is different from that of proprietary software development. It appears that most open source project development relies on virtual project management (VPM) to mobilize, coordinate, control, build, and assure the quality of FLOSS development activities. VPM invites or encourages system contributors to come forward and take a shared, individual responsibility that will serve to benefit the F/OSSD collective of user-developers. Open source project managers do not have the administrative authority to plan, manage, and control software development resources,
staff, schedule, and budget their projects. FLOSS projects enable a form of “participative management” by a decentralized community of developers (Scacchi, 2004).

Again, let us look at the organization, or say community structure, of open source development teams for different types of open source projects. As explained in Nakakoji (2002), the evolution of a system is linked to the evolution of its community. Different types of open source projects have a different community structure and evolve differently (as shown in Table 3-2). Nakakoji et al. (2002) identify three types of projects: exploration-oriented (centrally managed), utility-oriented (user managed), and service-oriented (managed by a council or foundation).

<table>
<thead>
<tr>
<th>Type</th>
<th>Objective</th>
<th>Control style</th>
<th>System evolution pattern</th>
<th>Community structure</th>
<th>Major problems</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploration-Oriented</td>
<td>Sharing innovations and knowledge</td>
<td>Cathedral-like central control</td>
<td>Single branch Feedback from the community</td>
<td>Project Leader Many Readers</td>
<td>Subject to split</td>
<td>GNU systems Jun Perl</td>
</tr>
<tr>
<td>Utility-Oriented</td>
<td>Satisfying an individual need</td>
<td>Bazaar-like decentralized control</td>
<td>Multiple versions coexist Tournament style</td>
<td>Many Peripheral Developers Peer support to Passive Users</td>
<td>Difficult to choose the right program</td>
<td>Linux system excluding the kernel</td>
</tr>
<tr>
<td>Service-Oriented</td>
<td>Providing stable services</td>
<td>Council-like central control</td>
<td>Single branch Patches merged through control</td>
<td>Core Members instead of a Project Leader Many Passive Users that develop systems for end-users</td>
<td>Less innovation</td>
<td>PostgreSQL Apache</td>
</tr>
</tbody>
</table>

Table 3-2. Types of projects and their community structures (Nakakoji et al., 2002)

Since the initial goal of most open source projects is to solve the problems encountered by a programmer or common problems among several programmers, there is much emphasis on features. This type of development is also known as feature-centric development (Levesque, 2004). Also, as there is no “powerful” project manager to determine what to do and what not to do, some important aspects of a programming project could go missing or be lack of focus such as: the user interface, documentation, coding standards, security, project direction, specified target audience, etc.
3.3.4 Innovation and evolutionary patterns of open source software

Software innovation was classified by previous researchers as: radical innovation and incremental innovation. Klincewicz (2005) attempts to provide an original classification of innovation in FLOSS based on four classes:

1) Radical innovation, technological modifications, platform modifications (passing a solution from one platform to another, a common practice among FLOSS developers) and market innovations providing new uses for existing technologies.

2) It has been argued that open source projects are more innovative than proprietary ones from a market perspective and a technology perspective (Dario et al., 2008). The reasons provided are:

- In a FLOSS world, barriers between companies and users are so low that new feature requests are getting into the mainstream of open source projects in a much faster manner than proprietary ones.

- The dynamic setup of the source code control and software building environment allows open source to “release early, release often.”

- As the source code and repository are all accessible to the users, users can even use inter-release packages if there are new features that they need. Or users can add the new features by themselves. “Giving users of a product access to its source code and the right to create derivative works allows them to help themselves, and encourages natural product evolution as well as preplanned product design” (O'Reilly, 1999).
The type of open source project affects its evolutionary pattern. According to Nakakoji et al. (2002), as the objectives of projects differ, they tend to have different evolutionary patterns. Some open source systems are showing patterns of splitting and merging their overall system architectures across releases (Figure 3-3). For example, in Linux there is less motivation for contributing back developed patches, as customers do not care about version updates and they prefer to stay with the current version of the system as long as the system is working, even though new versions are available.

Multiple implementations of the same functionality are allowed. Therefore, many branches evolved from a single program may exist (Nakakoji et al., 2002). This is so called forking scenario. On the other hand, there are cases where multiple projects merge into a single project, called joining. A good example is PostgreSQL. As new requirements emerge, active developers organize a team to implement them. However, these new implementations will exist for a relative long time, as patches are incorporated into the core version only when they are approved by the core members (Nakakoji et al., 2002).

Figure 3-3. Evolutionary patterns of the projects in (Nakakoji et al., 2002)
The evolution of open source software and the evolution of the surrounded community are mutually dependent on each other. "A large base of voluntarily contributing members is one of the most important success factors of FLOSS systems. The evolution of an FLOSS community is affected by the contributions made by its aspiring and motivated members. Such contributions not only transform the role and influence of their contributors in the community and thus evolve the whole community, but also are the sources of the evolution of the system" (Nakakoji et al., 2002). On the other hand, the evolution of an open source system will lead to changes to the community. "Any modification, improvement, and extension made to a FLOSS system – whether it is a bug fix, a bug report, or a patch – not only evolves the system itself but also redefines the role of those contributing members and thus changes the social dynamics of the FLOSS community" (Nakakoji et al., 2002).

3.3.5 Licenses

Licenses are something that cannot be avoided while using open source components. Proprietary software normally adopts the traditional software licensing approach where permission to use the software is granted to a customer in return for fee. Open source licensing approaches, in principle, define the right to modify and the right to redistribute the software. Open source licenses restrict how the software may be further changed or distributed, rather than impose a requirement that a fee be paid for it (Hiong, 2005).
There are many variations of open source licenses. The popular ones are Berkeley Software Distribution (BSD) License, GNU General Public License (GPL) and the GNU Library or "Lesser" General Public License (LGPL).

Integrating FLOSS components without understanding the licensing scheme being used by the components may result in one of the following situations:

1) Software cannot be redistributed because of the license limitations of a certain components.

2) License incompatibility among the open source components and closed source components force a switch of components even if the components work well from a technical perspective.

3) Due to the usage of a certain open source component, the license of this open source component forces the release of the source code of the proprietary components.

To avoid licensing incompatibility issues, a common approach is to add a wrapper around open source components to separate the open source part and the closed source part. The wrapper is released in open source manner. Obviously, adding wrappers introduces additional cost of development and maintenance.
4. RESEARCH DESIGN

The method being used by this study is "inducting theory using case studies - from specifying the research questions to reaching closure" (Eisenhardt, 1989). The research objective has been defined in chapter 1. The next step is to select cases.

4.1 Selecting cases

Selecting right cases is a very important step of building theory from case studies (Eisenhardt, 1989). Since this research is about proprietary software that has used open source components, it is very unlikely that detailed and first hand data can be collected from the public domain for such software. Consequently, the projects were selected from the company that I worked with as a product manager, where the original data of the software and people who developed and are maintaining such software are accessible. Of course, all these projects need to be development complete; and they need to have integrated with open source components.

As Pettigrew (1988) noted, given the limited number of cases which can usually be studied, it makes sense to choose cases such as extreme situations and polar types in which the process of interest is "transparently observable." Thus, the goal of theoretical sampling is to choose cases which are likely to replicate or extend the emergent theory. The literature review revealed that research for maintenance of component-based software needs to be done from two major perspectives: component level and architecture level. We would like to examine all selected cases from these two perspectives.
4.2 Theoretical framework

4.2.1 Components of software maintenance

As Grub et al. (2003) point out, “software maintenance is not an activity carried out in a vacuum. It affects, and interacts with the environment within which it is carried out. Indeed, it is changes and interactions in the surrounding environment that bring about the need for change.” As a consequence of this, they defined a Software Maintenance Framework to describe the software maintenance components (Table 4-1). We will be using this framework to examine the specific maintenance tasks being triggered by using open source components for each project.

Components of a software maintenance framework

<table>
<thead>
<tr>
<th>Component</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. User requirements</td>
<td>- Requests for additional functionality, error correction and improving maintainability</td>
</tr>
<tr>
<td></td>
<td>- Request for non-programming-related support</td>
</tr>
<tr>
<td>2. Organizational</td>
<td>- Change in policies</td>
</tr>
<tr>
<td>environment</td>
<td>- Competition in the market place</td>
</tr>
<tr>
<td>3. Operational</td>
<td>- Hardware innovations</td>
</tr>
<tr>
<td>environment</td>
<td>- Software innovations</td>
</tr>
<tr>
<td>4. Maintenance process</td>
<td>- Capturing requirements</td>
</tr>
<tr>
<td></td>
<td>- Creativity and undocumented assumptions</td>
</tr>
<tr>
<td></td>
<td>- Variation in programming practice</td>
</tr>
<tr>
<td></td>
<td>- Paradigm shift</td>
</tr>
<tr>
<td></td>
<td>- ‘Dead’ paradigms for ‘living’ systems</td>
</tr>
<tr>
<td></td>
<td>- Error detection and correction</td>
</tr>
<tr>
<td>5. Software product</td>
<td>- Maturity and difficulty of application domain</td>
</tr>
<tr>
<td></td>
<td>- Quality of documentation</td>
</tr>
<tr>
<td></td>
<td>- Malleability of programs</td>
</tr>
<tr>
<td></td>
<td>- Complexity of programs</td>
</tr>
<tr>
<td></td>
<td>- Program structure</td>
</tr>
<tr>
<td></td>
<td>- Inherent quality</td>
</tr>
<tr>
<td>6. Maintenance personnel</td>
<td>- Staff turnover</td>
</tr>
<tr>
<td></td>
<td>- Domain expertise</td>
</tr>
</tbody>
</table>

Table 4-1: Components of a software maintenance framework (Grub et al., 2003)
4.2.2 Change-based studies

Change-based studies analyze data that describes the individual changes made to the software systems. This method has been found effectively discovering cost and risk factors that are hidden. It views software maintenance and evolution as the aggregate of individual changes that are made to a software system throughout its life cycle. From Benestad et al. (2009) we learn that a change-based study can be used to:

- Identify and understand factors that affect change effort during maintenance and evolution.
- Measure performance trends during maintenance and evolutions.

These are the exact two aspects that we want to understand for maintaining hybrid software. So we decided to adopt this change-based study approach.

A conceptual model was proposed by Benestad et al. (2009) as shown in Figure 4-1. The basic idea is that a change task constitutes the fundamental activity around which software maintenance and evolution are organized. The following concepts are presented in this model:

- A change task is a coherent and self-contained unit of work that is triggered by a change request.
- A change request describes the requirements for the change task.
- A change task is manifested in a corresponding change set.
- A change set consists of a set of deltas.
- A delta is the differences between two revisions of the same component. A component can, in principle, be any kind of work product that is considered to be part of the system, although the reviewed studies focused primarily on measurement of source code components.

- Components can form a hierarchy in where a large component can be composed by components of finer granularity. A system is deployed to its users through releases.

- A release is composed by a set of components, each in a specific revision.

- A release can also be described by the change sets or the corresponding change requests that the release incorporates.

Figure 4-1. A conceptual model for change-based studies (Benestad et al., 2009)
4.3 Data collection

Based on the criteria defined in chapter 4.1, we identified 3 cases:

Case 1: Call control APIs for telephony applications that needs SIP capability. This project was integrated with open source SIP stack oSIP. All the other components within this project are closed source components and they were developed in house.

Case 2: Support for Asterisk platform using proprietary technologies. Asterisk is an open source platform that allows all kinds of telephony applications to be developed. Asterisk provides plug-in modules for closed source software companies to support this open platform. All the closed source components in this project were developed in house.

Case 3: Port of existing Windows hardware drivers to Linux hardware drivers but still keep the driver proprietary. All the drivers were developed in house and kept as closed source after the porting was complete. This is a very common type of project that can be found in companies that offer PC based hardware.

For the purpose of collecting more data and comparing the three projects in an unbiased manner, we collected data for the longest possible time with the same duration for all three projects, 4 years. Also, all data was collected after the first general available (GA) release of each project.

4.3.1 Sources of the data
4.3.1 Sources of the data

The data for the three cases were mainly collected from the following four sources:

1) Bug tracking system

Mantis\(^{(1)}\) has been used as the bug tracking system. The system has been customized to track both feature requests and bugs for all of the projects. The system is linked to the source control system SVN. Each issue being entered in the system for each project is considered as a change task or change set in our research. The change set is converted into a set of change tasks to estimate the effort. A lot of useful information was retrieved from this system, like the trigger for a change, why it was requested, who implemented the change, how long it took, links to the changed lines of code in SVN.

2) Source code control system

Subversion system is used as the source code control system. We logged into the SVN systems to count the lines of code for each project. Also it helped estimate the effort for some change tasks through the lines of code being changed when the effort is not clearly documented in the bug tracking system. When estimating effort through lines of code being changed, additional effort would be added for other members being involved for the same task, like QA, support, configuration. This source control system also helped confirm the component being change for each change task.

\(^{(1)}\) Mantis is a free popular web-based bugtracking system. It is written in the PHP scripting language and works with MySQL, MS SQL, and PostgreSQL databases and a webserver. It supports Windows, Linux, Mac OS, OS/2, and other operating systems. It is released under the terms of the GNU General Public License (GPL) (www.mantibt.org).
3) Internal specification

An open source wiki engine was used to track internal feature specifications and functional specification for all the projects and all releases. It also contains information of why there was a request of such feature and the initial estimation from developers for each feature. It helped the assessment of effort for some change tasks, especially feature requests. And it helped confirm the trigger of some of the change tasks.

4) Interview

In addition to these internal systems, interviews have been taken with developers, project managers, support personals, quality assurance people. The interviews were conducted to get subjective opinions, help effort assessment for some particular tasks, understand the challenges they are facing.

4.3.2 Data collected for each case

For each change task, data was collected from several aspects: maintenance effort; who were involved in the change; attribute of each change task, bug or feature request; what software component being changed; what triggered the change task. As for what triggered the change task, we used the maintenance framework defined by Grub et al. (2003), as explained in chapter 4.2.1.

Lines of code were collected for each project as the indicator of the complexity of each project. CLOC was used for the counting of lines of code for each project (CLOC). CLOC is an open source, very easy to use utility written in Perl, which can run on Linux,
Mac OS and Windows. The detailed results of the lines of code counting for all three cases can be found in Appendix A.

Case 1:

The project was firstly released on May 26th, 2006. 277 change tasks were collected from May 26th, 2006 until May 26th, 2010. There are total of 26575 lines of code, including 6712 blank lines and 4077 comment lines.

Case 2:

The project was firstly released on Aug 17th, 2006. 315 change tasks were collected from Aug 17th, 2006 until Aug 17th, 2010. There are total of 11773 lines of code, including 1748 blank lines and 1517 comment lines.

Case 3:

The project was first released on June 29th, 2005. 24 change tasks (specifically related to the support of Linux operating system) were collected from June 29th, 2005 until June 29th, 2009. The driver contains total of 68361 lines of code, and it supports both Windows and Linux. Assuming one third of the code was specifically designed for Windows, one third was for Linux and the left one third was common code for both platforms. So there are total of 22787 lines of code, including 4734 blank lines and 1578 comment lines.
4.4 Data analysis

The total effort for each project was the sum of the effort for each of the change tasks that have been collected. Since the complexity the three projects was different, to allow for a fair comparison in the following steps, lines of code has been used as a proxy of the complexity of each project.

We also calculated the sum of the effort for each type of maintenance tasks categorized from the perspective of the triggering, like user requirement, maintenance process, operational environment and software product. The percentage of effort for each maintenance component in the maintenance framework was calculated as well. Please refer to Table 7-5 in Appendix B for the results of data analysis.

4.5 Results

4.5.1 Cases classification

The 3 selected cases were classified from 2 aspects: component level and architecture level.

1) Component level

This research is about hybrid software that contains both closed source components and open source components. As for the closed source components being used by the three projects, they were all developed in house. We did not find any cases
that can be used for this study that are using closed source components from third party software companies.

As for the open source components being used by the 3 projects, we classified the components based on the matrix of maturity and reusability. On the dimension of maturity, software can be classified as mature ones and relative new ones. On the other reusability dimension, based on how broadly the FLOSS project being used, we categorized open source projects with broad usage and narrow usage. Software with broad usage is used in many scenarios and by many type of application; software with narrow usage can be used in certain scenarios and certain ways. The three selected projects cover three scenarios out of 4 scenarios in the matrix as shown in Table 4-2.

<table>
<thead>
<tr>
<th>OSS characteristics</th>
<th>Mature</th>
<th>Relatively New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broad usage</td>
<td>Case 3</td>
<td>Case 2</td>
</tr>
<tr>
<td>Narrow usage</td>
<td>n/a</td>
<td>Case 1</td>
</tr>
</tbody>
</table>

Table 4-2. Characteristics of the FLOSS Component Adopted by the 3 Cases

2) Architecture level

From the literature review, it was understood that the properties of a system are influenced more by the interaction of its components than by the properties of a single component. Hence, it is important to understand the architecture of each project.

We generalized all the selected cases to contain 4 abstraction layers, from the bottom to the top:
- Operating system layer (Windows or Linux);
- Lower layer: driver layer. The layer that allowing the PC hardware to talk to the operating system.
- Middle layer: the layer bridges the communication between the driver layer and the user interface layer.
- User interface layer: the layer that allows customers/users to build their own applications on top of it.

The three selected cases represent three different architectural relationships between FLOSS components and other closed source components (Figure 4-2).

**Case 1**
- Proprietary User Interface and Applications
- Proprietary Components
- Proprietary Hardware Driver
- Closed Source Operating System

**Case 2**
- Open Source User Interface and Applications
- Open Source Components
- Proprietary Hardware Driver
- Closed Source Operating System

**Case 3**
- Proprietary User Interface and Applications
- Proprietary Software Components
- Proprietary Hardware Driver
- Closed Source Operating System

**Figure 4-2. Software architecture of the three cases**

### 4.5.2 Proposition development

Propositions were developed based on comparing the results horizontally and vertically from the results being developed in Chapter 4.4, Table 7-5 in Appendix B.
4.5.2.1 Horizontal analysis (A)

Horizontal analysis is to compare, horizontally, the total maintenance effort and the effort for each maintenance category among three cases. It was designed to understand how the maintenance effort is affected by the characteristics of the FLOSS component and the structural relationships between the open source component and proprietary software components.

4.5.2.1.1 Examine the total maintenance effort

As we can see from the results, case 1 has the highest value of total maintenance effort. However, considering three projects have different complexity, the level of maintenance effort spent for each project is represented by the total maintenance effort divided by lines of code. The results showed case 2 required the highest level of maintenance effort; and case 3 has the lowest value.

Based on this result, we can say hybrid software that have an open source component running at the top layer requires the highest level of maintenance effort. This can be explained as caused by one of the characteristics of open source software that we analyzed in chapter 2.3.1, the structure of the open source community. The literature suggests that a successful open source project should have a large number of users, many of whom are both users and developers. To support such a type of open source platform using proprietary software, as case 2 did, means that the system would be exposed to broader range of users than the complete proprietary software.

This also provides access to wider pool of skill levels among the users, who would use the software in all kinds of different ways, and discover more bugs. However, since the underlying software is proprietary software, these users do not have access to it
and they are not able to fix the bugs themselves, even if some users have the skills to do so. Consequently, these tasks came back as change tasks to be fixed internally. On the other hand, when the open source components are at the other layers of the software (case 1 and case 3); the open source components are “hidden” through the proprietary user interface. As a result, although open source components are used, the scope of targeted users of the software has not been changed. They are still proprietary software users.

**Proposition A-1:** To support open source platform with proprietary software running at the user interface layer requires more maintenance effort compared to software that has integrated open source components at lower architectural levels of the system.

We also measured the average effort for the change tasks of each project. It was found that case 3 has the highest effort/task value. This can be explained since case 3 is at the driver layer, which is the bottom layer of a system right above the operating system. We know Linux operating system has many distributions and variants. This is caused by the evolutionary pattern of the Linux software, as we explained in Section 3.3.4. The evolutionary pattern of the Linux operating system is shown in Figure 4-3. As Nakakoji et al. (2002) explains, the forking scenario is frequently occurring in Linux. There is much less motivation and encouragement for contributing back the developed patches, as customers do not care about version updates and they prefer to stay with the current version of the system as long as the system is working, even if new versions are
available. Multiple implementations for the same functionality are allowed. Hence, many branches that have evolved from a single program may exist (Nakakoji et al., 2002).

Figure 4-3. Linux evolutionary pattern – forking (Nakakoji et al., 2002)

Therefore, when a change requires a formal patch release, the same version of software needs to be built upon different Linux operating system distributions and kernels, which requires a larger effort from configuration management and build management. In addition, since this layer is at the low level in the system, any change at this layer may have side effects to all the software layers above it. The system requires more thorough testing after changes. The same idea applies to the development stage.

*Proposition A-2: The average effort spent for each change task at the bottom layer software of the system to support open source operating system is higher than the effort required for each change task at the other layers of the system.*

4.5.2.1.2 Examine the effort spent for each categories of change tasks

1. In case 1, 68.84% and in case 2, 55.35% of the total maintenance effort was spent on completing the change tasks triggered by maintenance process, and most of them are error detection and correction type change tasks.
Integrating open source components in proprietary software has been proved easier said than done. Having FLOSS integrated with proprietary software requires full understanding of how exactly the component works to allow smooth communication to the other parts of the software. Otherwise, when the system has the FLOSS integrated without full understanding of this component or wrong understanding, ever since it is adopted by real customers, problems start to be reported.

Sufficient and accurate documentation of the FLOSS components will help the understanding of the FLOSS components. However, when the FLOSS project itself is not mature, it is normally not equipped with sufficient documentation. Even when there are documentations, due to the frequent changes of the open source components in their immature stage, the documentation gets out of date quickly. So FLOSS components may behave differently than the documentation says. Both case 1 and case 2 were integrated with relatively new open source projects and with less reliable documentations.

*Proposition A-3: When integrating immature open source components into proprietary software, the majority of the maintenance effort is spent on error detection and error correction.*

2. In case 2, 86 change tasks were from user requirement; while in case 3, 12 change tasks were from user requirement. That amounts to 27.30% and 50% of the total number of change tasks respectively for case 2 and case 3. In case 1, 52 change tasks were from the user requirements, which is only 18.77% of the total number of change tasks for case 1. The reason is that both case 2 and case 3 were integrated with open source components
with a broader range of usage compared to the one being integrated in case 1, which was specifically designed for SIP protocol. As a result of this, more user requirements were raised to require the system to be used in ways that the system was not designed for.

*Proposition A-4: Proprietary software integrated with open source components that have broad range of usage receives more requests from users; it consequently has a higher number of change tasks triggered by user requirements.*

3. In case 3, the effort spent for on change tasks for operational environment changes is 43.47%, whereas in both case 1 and case 2, the effort spent for that category of change tasks are close to 0%. This is easy to understand as both case 1 and case 2 are offering software at upper layer away from the operating system. So the changes of operating system or hardware are hidden away from them. However, in case 3, it is to support Linux driver which has to deal with the changes of operating system and hardware.

*Proposition A-5: Supporting open source operating system as Linux with proprietary driver software requires a large maintenance effort to deal with the frequent changes of the operating system or the hardware environment.*
4.5.2.2 Vertical analysis (B)

Vertical analysis is to compare the maintenance effort for each type of tasks within each case. It was designed to understand the major challenges for each project specifically and hopefully find out the reasons and corresponding solutions to alleviate them.

4.5.2.2.1 FLOSS at the middle layer – case 1

1. Case 1 project adopted FLOSS component that is under GPL licensing. A wrapper was added for the FLOSS component to avoid licensing incompatibility issue as explained in chapter 3.3.5.

Proposition B-1: Integrating open source components with GPL licensing with proprietary software brings about licensing incompatibility issue.

There is no doubt that the additional layer increases the maintenance cost. In this project, it was calculated that the maintenance effort spent for fixing issues at the wrapper is 256 man/days, which is 28.70% of the total maintenance effort for this project. This proves that the effort of maintaining additional wrapper is significant enough not to be ignored.

Proposition B-2: Solving licensing incompatibility issue of utilizing FLOSS by adding an open source wrapper requires additional maintenance effort.
There is no real solution to avoid having this layer unless company is willing and able to switch the other parts of the proprietary software to be open source software under the same licensing scheme. However, due to the intellectual property inside of the proprietary software, this is often not a feasible solution to proprietary software companies. This additional cost of maintenance for the wrapper layer should be considered while making the decision of adopting open source components.

2. In case 1, 68.84% and of the total maintenance effort was spent for the change tasks triggered by maintenance process, and most of them are error detection and correction type change tasks. From the interview, it was discovered the main developers for case 1 were proprietary software developers who used to be designing only proprietary software. They all indicated when they designed the code and made the modifications to the code, they mostly rely on reading the code or documentation itself. As discussed in chapter 4.5.2.1.2, the immaturity of this open source project means that it does not have sufficient documentation. Consequently, the communication with the community should leverage the unique communication ways of FLOSS as described in chapter 3.3.3. This introduces challenges to proprietary software developers, who are used to formal documents, meetings, conferencing, stand-by support lines and formal training to get to know how the component works. Not being able to communicate with the open source community efficiently definitely increases the effort required to support the maintenance tasks triggered by maintenance process.
**Proposition B-3**: When adopting immature open source component, proprietary software developers being able to efficiently communicate with the community of the open source components will help reduce the maintenance effort spent on error detection and correction for hybrid software.

4.5.2.2.2 FLOSS at the top layer – case 2

The goal of Case 2 project was to support open source platform with proprietary software components through the interface being provided by the open source project.

The results for case 2 showed that the majority of that effort was for change tasks triggered by maintenance process and user requirement, 55.35% and 27.35% respectively. It was discovered the same scenario in case 2 as case 1 through interview, the main developers for this project was used to be designing proprietary software only. So the proposition B-3 developed in chapter 4.5.2.1 applies to this case too.

Through the interview, we discovered that in this project there is no valuable intellectual property. It is running on top of proprietary software APIs to support the interface offered by the open source platform, which is a standalone project from the other proprietary software. There is no real reason that this layer cannot be released as open source project. We argue here, if this layer is released as open source project and contribute it back to the open source project community, which has thousands of active developers and users. The maintenance tasks will be co-worked by the community and the company. Taking advantage of the power of community to maintain it will largely offload a lot of maintenance burden from the company.
Proposition B-4: Supporting open source platform with proprietary software running under it, company should consider the possibility of releasing the proprietary software in open source format while designing it; which will largely reduce the maintenance effort of such software.

4.5.2.2.3 FLOSS at the bottom layer – case 3

The objective of the case 3 project was to port existing an Windows hardware driver to Linux using the same stream of source code. This is a very commonly seen project in a hardware development company that supplies proprietary drivers.

As we have pointed out above, the change tasks triggered by user requirements comprise 50% of the total change tasks for this project. The effort spent on meeting these change tasks is 52.17% of the total maintenance effort of the project. We examined all the change tasks triggered by user requirement and found most of them are for the “Linux-on-demand” program offered by the company. The reason to establish this program was to meet a requirement from Linux customers that use different distributions or different kernels of Linux than the ones being generally supported, sometimes even their own special customized distributions or kernels. In our study, this type of task has been defined as change set which contains a set of change tasks:

- Software built from SVN (Code change/fixes as required)
- Testing machine set up with the kernel/distribution required
- Release files tested
- Create install instructions, requirements, etc.
- Release ZIP/TAR/RPM files created
- Copy ZIP/TAR/RPM file to central location
- ZIP/TAR/RPM files posted to web/FTP site
- Tech Support given new link to FTP site
- Release history updated
- Company notice re: sending of build
- Release authorized

The total effort of the whole change set is 10 man/days in normal circumstances; sometimes can be much longer than that if there is interface incompatibility issue with the customized kernel/distribution. And that is just cost from effort perspective. To be able to support the customer that needs such build, new machines may need to be purchased to keep the regression testing running for future software releases. So this change set is expensive not only from effort perspective, but also from hardware infrastructure perspective.

The root cause of that many requests to support different distribution and kernels is the evolutionary pattern of the open source software component has - as we had explained in Section 3.3.4. The evolutionary pattern of the Linux operating system is as shown in Figure 4-3. As Nakakoji et al. (2002) explained this forking scenario is frequently occurring in Linux.

*Proposition B-5: The Linux evolutionary pattern determines that supporting Linux operating system with proprietary driver software will lead to a large effort to meet the user requirement to support variant Linux distributions and kernels.*
The real solution to this issue is to open up the proprietary driver and release the driver as open source. Or separate the components in the driver that cannot be released for intellectual property reason or other reasons and put them into binary library format. The ultimate goal is to make the driver “rebuildable” by the user in real time against the kernel or distribution that they want to use.

Proposition B-6: Allowing the driver rebuildable against the specific Linux distribution and kernel will largely reduce the maintenance effort triggered by the user requirement of supporting variant Linux distributions and kernels.

4.6 Summary

Two groups of propositions, with total of 12 individual propositions were developed, as demonstrated in Table 4-3:

- The first group (A) focused on analyzing the effect of FLOSS characteristics and the structural relationship of FLOSS component and proprietary components over maintenance effort and maintenance types.
- The second group (B) focused on solutions to the major maintenance challenges of each individual project. Suggestions were then given specifically for each project regarding what improvement can be done to possibly reduce the maintenance effort. They are realistic suggestions with the consideration of the unique situations of each project.
<table>
<thead>
<tr>
<th>Cause</th>
<th>Effect →</th>
<th>Aspects of Maintenance</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total Maintenance Effort</td>
<td>A-1</td>
<td>B-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maintenance Effort/Task</td>
<td>A-2</td>
<td>B-2</td>
<td>B-4</td>
</tr>
<tr>
<td>Architecture</td>
<td></td>
<td></td>
<td></td>
<td>B-5</td>
<td>B-6</td>
</tr>
<tr>
<td>Matureness of FLOSS</td>
<td></td>
<td>B-3</td>
<td></td>
<td></td>
<td>A-3</td>
</tr>
<tr>
<td>Scope of usage of FLOSS</td>
<td></td>
<td></td>
<td></td>
<td>A-4</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-3. Summary of propositions
5. CONCLUSION

We closely examined the maintenance of three representative hybrid software systems for over 4 years, long enough to observe a number of maintenance problems. Each of the three projects has a unique structural relationship between the open source components and closed source components. The open source component adopted by each project covers a certain quadrant in the characteristics matrix for open source projects as described in chapter 4.5.1 (Table 4-2).

The research collected multi-aspect and rich data through data mining of internal databases and interviews. The data was collected and analyzed using the change-based analysis mechanism, which has been proved to be capable of identifying and understanding factors that affect change effort during maintenance and evolution (Benestad et al., 2009).

From the results, two sets of propositions were developed. The first group of propositions described the effects of the structural relationship between open source components and proprietary software components, the maturity of and scope of usage of the FLOSS component over total maintenance, average effort per task and the types of maintenance tasks. It was found that hybrid software supporting an open source platform at the top layer, which directly interacts with open source users, requires the most maintenance effort compared to hybrid software that has open source component “hidden” at the other layers. It was also discovered that maintaining the proprietary driver software running on top of open source operating system requires the highest effort per maintenance task although the total effort is low due to the relatively low number of maintenance tasks.
We then studied each project from the perspective of determining what can be realistically improved with consideration of the unique situations of each project; and, thereafter, developed a second group of propositions. The second group of propositions provided suggestions for hybrid project managers to consider while designing and maintaining hybrid software. Releasing proprietary software as an open source project, whenever possible, can help reduce the maintenance effort. More efficient communication with the open source community will also help reduce the maintenance effort.

This thesis studied the maintenance of hybrid software with emphasis on understanding architectural effect over maintenance effort. It fills in the gap in the study of software maintenance from two perspectives:

1. An empirical study of software maintenance in hybrid software.
2. A study of the effect of structural relationship of open source components and proprietary software components on maintenance effort. All data was collected first-hand from internal bug tracking databases or source code databases and interviews with the original maintainers of the projects.

However, the study is limited in a way that all three cases were from a single small company located in Ottawa of Canada. No data was available from other companies or larger organizations that are globally operating.

Future research could be done with the analysis of other characteristics of FLOSS projects. Also, more cases could be collected to test the propositions being developed in
this thesis. It will be also interesting to study hybrid software that contains both open
source components and closed source components purchased from third party software
companies. On the other hand, this is a change-based study that assessed the maintenance
tasks that have been implemented and the effort; further study using the same method
could be done to help predict the effort/cost trends of such hybrid software in order to
plan and take corrective actions, when negative trends have been observed.
6. REFERENCES


G.S. Hiong (2005), Open Source And Commercial Software – An In-depth Analysis of The Issues, *Business Software Alliance*, Available at: www.bsa.org

GNU Project, Philosophy of the GNU Project, *GNU Website*, Available at: www.gnu.org/philosophy.


M. Aslett (2008), 451 CAOS Community, Blog of the 451 Group, Available at: blogs.the451group.com/opensource


P. Vitharana (2003), Risks and challenges of component-based software development, Communications of the ACM, Volume 46, Issue 8, Pages: 67 - 72


W. Scacchi (2004a), Free and open source development practices in the game community, *Software*, (21)1, 1, Jan-Feb, 59 – 66, IEEE.


7. APPENDICES

Appendix A. Lines of code for the cases

Case 1:
- Two parts: Higher layer + Lower layer.
- Total LOC for Case 1 = LOC of Higher layer + LOC of Lower layer = 26575

Lower Layer:
10 text files
10 unique files
30 files ignored

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Blank</th>
<th>Comment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>4</td>
<td>1266</td>
<td>906</td>
<td>6403</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>5</td>
<td>191</td>
<td>320</td>
<td>527</td>
</tr>
<tr>
<td>Make</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>SUM:</td>
<td>10</td>
<td>1464</td>
<td>1228</td>
<td>6952</td>
</tr>
</tbody>
</table>

Table 7-1. Results of lines of code for case 1 (Low layer)

Higher layer:
41 text files
41 unique files
0 files ignored

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Blank</th>
<th>Comment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>23</td>
<td>4687</td>
<td>2366</td>
<td>17778</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>18</td>
<td>561</td>
<td>483</td>
<td>1845</td>
</tr>
<tr>
<td>SUM:</td>
<td>41</td>
<td>5248</td>
<td>2849</td>
<td>19623</td>
</tr>
</tbody>
</table>

Table 7-2. Results of lines of code for case 1 (High layer)

Case 2:
- LOC = 11773

Channel Driver for Asterisk
49 text files.
41 unique files.
249 files ignored.

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Blank</th>
<th>Comment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>4</td>
<td>1588</td>
<td>1443</td>
<td>11051</td>
</tr>
</tbody>
</table>

63
Table 7-3. Results of lines of code for case 2

Case 3:
- Assume 1/3 codes are Linux specific; 1/3 are Windows specific; 1/3 are common code
- LOC = 68361/3 = 22787

Drivers (for both Windows and Linux)

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Blank</th>
<th>Comment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>65</td>
<td>10298</td>
<td>11700</td>
<td>35028</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>113</td>
<td>3368</td>
<td>8517</td>
<td>31510</td>
</tr>
<tr>
<td>C++</td>
<td>5</td>
<td>439</td>
<td>677</td>
<td>1515</td>
</tr>
<tr>
<td>DOS Batch</td>
<td>1</td>
<td>62</td>
<td>7</td>
<td>190</td>
</tr>
<tr>
<td>Make</td>
<td>8</td>
<td>27</td>
<td>6</td>
<td>87</td>
</tr>
<tr>
<td>Bourne Shell</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>Teamcenter Def</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SUM:</td>
<td>194</td>
<td>14201</td>
<td>20914</td>
<td>68361</td>
</tr>
</tbody>
</table>

Table 7-4. Results of lines of code for case 3

Appendix B. Results of change task analysis for 3 cases
<table>
<thead>
<tr>
<th>% of total effort for maintenance process</th>
<th>68.84%</th>
<th>55.35%</th>
<th>0.00%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort/task</td>
<td>2.87</td>
<td>1.79</td>
<td>0.00</td>
</tr>
<tr>
<td>Software product</td>
<td>76.00</td>
<td>77.5</td>
<td>2</td>
</tr>
<tr>
<td># of tasks for software product</td>
<td>33</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>% of total effort for software product</td>
<td>11.95%</td>
<td>15.82%</td>
<td>4.36%</td>
</tr>
<tr>
<td>Effort/task</td>
<td>2.30</td>
<td>1.55</td>
<td>2.00</td>
</tr>
<tr>
<td>Operational environment</td>
<td>4.00</td>
<td>8.00</td>
<td>22.50</td>
</tr>
<tr>
<td># of tasks for operational environment</td>
<td>1</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>% of total effort for operational</td>
<td>0.37%</td>
<td>1.58%</td>
<td>43.47%</td>
</tr>
<tr>
<td>environment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effort/task</td>
<td>4</td>
<td>1.6</td>
<td>2.25</td>
</tr>
</tbody>
</table>

Table 7-5. Results of change task analysis