AN OPEN SOURCE SOFTWARE SELECTION PROCESS AND A CASE STUDY

A Record of Study

by

GUOBIN HE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF ENGINEERING

August 2007

Major Subject: Engineering

College of Engineering

AN OPEN SOURCE SOFTWARE SELECTION PROCESS AND A CASE STUDY

A Record of Study

by

GUOBIN HE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF ENGINEERING

Approved by:

Chair of Committee,	Dick B. Simmons
Committee Members,	William M. Lively
	Jianer Chen
	Sheng-Jen Hsieh
	John Fierstien
Head of Doctor of Engineering Programs,	N.K. Anand

August 2007

Major Subject: Engineering

College of Engineering

ABSTRACT

An Open Source Software Selection Process and a Case Study. (August 2007)

Guobin He,

B.S., National University of Defense Technology;

M.S., Beijing University of Aeronautics and Astronautics

Chair of Advisory Committee: Dr. Dick B. Simmons

In this study, I design an empirical open source software selection process, which reuses some ideas from Commercial Off-the-Shelf selection methods and addresses the characteristics of the open source software. Basically, it consists of three basic steps: identification, screening and evaluation. The identification step is to find all possible alternatives to open source software that can meet the high level requirements. The next step is screening, in which the refined requirements are applied to filter the alternatives. The evaluation step is based on the Analytic Hierarchy Process, in which the alternatives are inspected from functional suitability, source code, support strength and popularity. In more detail, under functionality suitability criterion, alternatives to open source software are evaluated in viewing of how much functionality can fit in with the functional user requirements. The source code of the alternatives is evaluated from six criteria: programming language, code size, code comment, code intra-module complexity and code inter-module complexity. The evaluation of support strength depends on the evaluation of field support and support resources. The field support includes commercial support and community support. The community support specifically refers to the direct responses from the community to the support requests. Aside from field support, open source software projects also provide various support-related resources such as, documents, wiki, blog, etc. To determine the popularity of the alternatives, I evaluate them from software use, development participation and web popularity.

In the case study, I utilize the process to select the best open source unified modeling language tool from the ten alternatives for the software development process. After the screening phase, the four competitive alternatives, BOUML, ArgoUML, UMLet and Violet, are evaluated from functionality, source code, support strength and popularity criteria. The evaluation result indicates that ArgoUML is the best tool for the requirement. The case study demonstrates the effectiveness of the selection process. Various important attributes of open source software are taken into consideration systematically and the final decision is reached based on comprehensive investigation and analysis. The process provides an operable solution to the open source selection problem in practice.

DEDICATION

This record of study is dedicated to my parents, my wife and my baby, for their love and support.

ACKNOWLEDGEMENTS

Two years ago, I decided to focus on my career development in industry. Therefore, I switched from the Doctor of Philosophy program to the Doctor of Engineering program. Now in retrospect, I feel I made a correct decision and am better prepared for the future career challenges. Thanks to the College of Engineering at Texas A&M University for setting up such a good program to help students fulfill their interests and prepare them for future career goals. Particularly I would like to thank my advisor, Dr. Dick B. Simmons, for his constant understanding, encouragement, advice, patience and guidance. He helped me select the appropriate topic and patiently guided me through the record of study process.

I would like to express my gratitude to my committee members, Dr. William M. Lively, Dr. Jianer Chen and Dr. Sheng-Jen Hsieh for their support. It is a wonderful experience for me to work with these professors.

Many thanks go to Dr. Dezhen Song for his involvement on my advisory committee, particularly in the final defense.

Special thanks to Mr. John Fierstien, for his involvement on my advisory committee as my internship supervisor. Mr. John Fierstien brought me into a fast growing company and gave me the opportunity to work on several projects.

I would like to thank Guangtong Cao, Rui Li, Yingwei Yu, Xingfu Wu, Lui Cheng and Xu Yang for their friendship and help during my study.

In the end, I owe a special debt of gratitude to my wife, my baby and my parents for their great love and sacrifices.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Page

1 INTRODUCTION

1.1 Open Source Software

1.1.1 History

At the early stage of computer software development, the computer professionals shared the software freely. However, with the popularity of computer software, people started to charge fee for software use and release software binary code instead of source code. The software became proprietary. On the one hand, the emergence of proprietary software attracted many companies to enter the software market and led to software industry boom in 1990s; on the other hand, it was believed to suppress knowledge sharing and distribution and give rise to monopoly. In 1984, Richard Stallman, launched the GNU project aiming at developing a free operating system. He believed that the source code is fundamental to the furthering of computer science and freely available source code is truly necessary for innovation to continue [1]. In 1985, he established The Free Software Foundation (FSF) to promote free software movement, a social movement which advocates users' rights to use, study, copy, modify, and redistribute computer programs [2]. The name of Open Source Software was accepted as a new label for free software in "Open Source Summit" in 1998. In the same year, the organization, Open Source Initiative (www.opensource.org) (OSI), was founded to promote open source software. OSI gives an open source definition and certifies open source licenses. Also, the open

---

This record of study follows the style and format of *IEEE Transactions of Software Engineering*.

source movement was launched to advocate open source software. Open source movement focuses on advocating the benefits of open source software. In contrast, free software movement emphasizes the freedom of use. Although OSI's open source software is defined differently from the FSF's free software, in practice, people treat open source software and free software as the same. They use Free/Open Software (FOSS), Free/Libre/Open Source Software (FLOSS) or open source software/free software (OSS/FS) as the general term. In our paper, we stay with the term - open source software (OSS).

1.1.2 The Formal Definition

As mentioned, FSF and OSI have different definitions for Open Source/Free Software. FSF defines free software as software with four freedoms [3]. These freedoms include the freedom to run the program, for any purpose (freedom 0); the freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is the precondition for this; the freedom to redistribute copies so you can help your neighbor (freedom 2) and the freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this. In this paper, we use the definition from Open Source Initiative [4][5]. This definition gives a list of criteria which open source software should confirm to:

1. Free Redistribution. There is no restriction in the license from selling or giving away the software as a component.
2. The software should include source code.

3. The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
4. Integrity of the author's source code.
5. No discrimination against persons or groups.
6. No discrimination against fields of endeavor.
7. Any users whom the software is distributed to have all the rights defined in the license.
8. License must not be specific to a product.
9. License must not restrict other software.
10. License must be technology-neutral.

There are two related terms we should distinguish from OSS. The first is "Freeware". There is no definition for this term but it refers to the type of software which can be used without any cost and in unlimited time. Freeware does not expose the source code, so it is not OSS. The second is "Shareware". In many cases, before releasing a new version of proprietary software, the software developers would issue free trial versions. These trial versions are called shareware. Shareware usually comes with partial features of its proprietary counterpart and its use is limited to a certain time period. It is only a piece of binary code and does not expose the source code.

1.2 OSS Development Process

The OSS development process is different from the traditional development process. Basically traditional development process involves requirement analysis, design, implementation, testing and maintenance stages [6]. The whole process is undertaken by teams or groups comprised of software professionals. This is not the case for OSS development. According to the popular article by Eric S. Raymond: The Cathedral and the Bazaar [7], OSS development model can be divided into two phases: the cathedral phase and the bazaar phase.

Typically the idea of an OSS project appears as a special requirement, i.e. “something the computer ought to be doing for me” [7]. Eric Raymond describes this requirement as “a developer’s personal itch” [7]. At the same time, the requirement cannot be fulfilled by software accessible to a developer. Otherwise, the developer will choose either to use it or even directly participate in the development. After some analysis on risk, schedule and requirement [6], the developer (or a team) may start his own project if he feels he has strong motivation. The developer will implement a software prototype for the project. The implementation also may follow a traditional development process and therefore experience more or less requirement analysis, design, implementation and testing. Then the developer publicizes the prototype with its source code, which ends the cathedral phase. In order to successfully enter the bazaar phase, the prototype should be extensible to allow multiple developers to work on the project simultaneously [6][8][9]. More important, even if the prototype does not work well currently, it should present a plausible promise to “convince potential co-developers that it can be evolved into something really neat in the foreseeable future” [7]. At the same time, the effective communication and license model should also be established.

The Bazaar phase starts the community building of the OSS. The OSS project success is closely related with whether the community building is active or not. The community is comprised of developers and users but the distinction between developer and users is blurred. Despite the project is still controlled by a core development team, any users can voluntarily become co-developers. According to the tightness of the connection with the development, people propose an onion-like model [10][11][12][13] for the community illustrated in figure 1-1. Core developers undertake most development and administration work. Co-developers submit code patches and review the code. The code patches will be checked into the code by core developers after being reviewed and accepted. Active users do not work on the source code but they make their contribution via using the software and submitting bug reports or feature requests. Passive users are

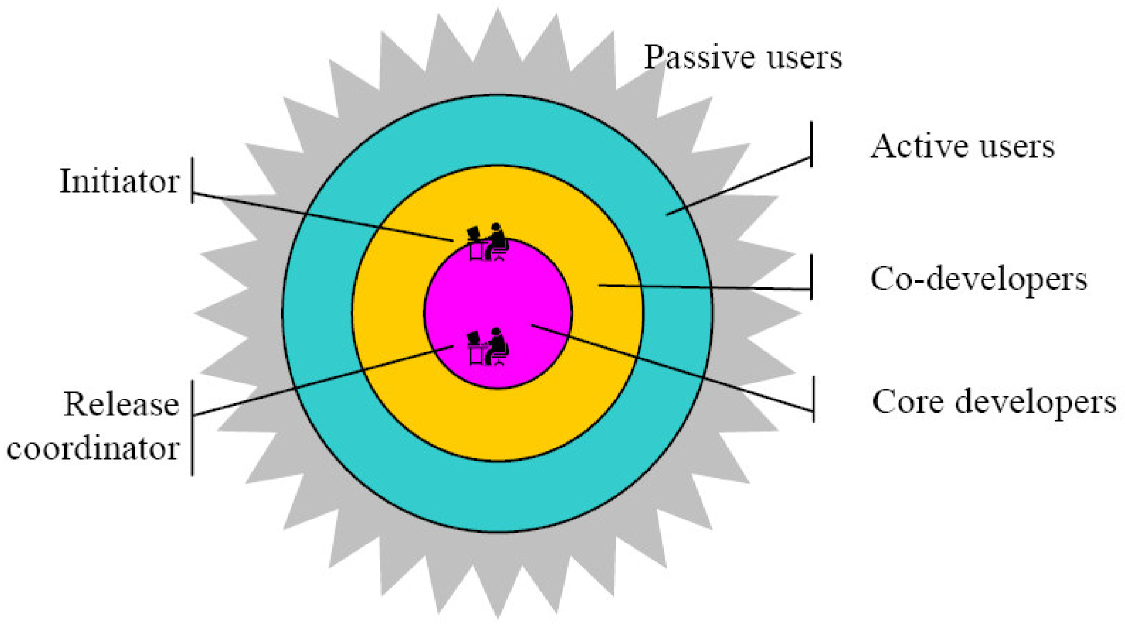

Figure 1-1 OSS Community Model (excerpted from [11])

just regular software users. The bazaar phase is characterized with peer review, concurrent development and opening up requirement [6]. Peer view is helpful for finding and solving the problems in the project. As Eric Raymond said [7], "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone”. Concurrent development means during the same time period the development proceeds by multiple developers and in various activities such as fixing bugs or implementing new features. In bazaar phase, OSS requirements are shaped by the multiple voluntary feature requests and suggestions. Although too many multiple requirement propositions may pose a problem to project management, such opening can help OSS developers understand what requirements users really need or prefer.

1.3 Software Evaluation

Software evaluation is an important field in Software Engineering. People want to have an idea on how good the software is, which will help select the software or improve the software development. There is a standard ISO 9126 [60] drawn up for software evaluation. This standard discusses the software evaluation from four respects: quality model, external metrics, internal metrics and quality in use metrics. The quality model sets a set of characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. Since the topic of software evaluation is too broad, here we narrow down our discussion to COTS (commercial-of-the-shelf) evaluation. We are interested in COTS evaluation mainly because it is similar to OSS in some respects. First, there exist a large number of COTS products. When people try to use COTS, they may face several options and they need to select the best fit via evaluating each COTS; Second, COTS promotes software reuse, which is also involved in OSS adoption. Some OSS is released in the form of reusable component. More important, people may integrate OSS into their software in order to reduce work and save time. The difference is that OSS reuse mostly belongs to white box reuse while COTS reuse is a black box reuse; Third, during the integration, the adaption work may be required for both software. In the following paragraphs, we will introduce COTS evaluation methods: OTSO [14][15], CRE[16], PORE[17] and Opal[18]

OTSO (Off-The-Shelf Option) model introduces a multiple criteria decision technique, Analytic Hierarchy Process (AHP), into the COTS selection. It defines six phases: search, screening, evaluation, analysis, deployment and assessment. At the beginning, a user searches the possible COTS alternatives. Next he selects the COTS alternatives which he believes need to be evaluated in more details. Then he evaluates these alternatives with usually multiple criteria. The evaluation results should be carefully analyzed and the best fit is selected. After the COTS is deployed, the assessment regarding the success of adopting COTS takes place, which aims to improve the selection process in the future. The evaluation criteria are generated during the search, screening and evaluation phases. Generally they can be categorized into four areas: functional requirements, product quality characteristics, strategic concerns, and domain and architecture compatibility [14]. The multiple evaluation criteria are organized into a hierarchy with the aid of AHP. Under the hierarchy, COTS alternatives are compared and ranked step by step. In the end, the one with the highest priority is the best alternative.

CRE (COTS-Based Requirements Engineering) emphasizes on non-functional requirements modeling to assist the process of evaluation and selection of COTS products [16]. It is an iterative process consisting of four phases: identification, description, evaluation and acceptance. In the identification phase, evaluation criteria are defined in view of the factors which may affect the COTS selection. The criteria and the requirement description, particularly the non-functional requirements, are refined in the description phase. At the same time, the requirements are also prioritized and the product information is acquired. At the evaluation stage, users evaluate COTS alternatives based on cost versus benefit comparison. The authors proposed several methods regarding decision making such as weighted scoring method (WSM) and AHP. The final acceptance phase involves purchasing and legal issue resolving.

PORE (Procurement-oriented requirements engineering) is a template-based method regarding requirements acquisition for selecting COTS products. The selection proceeds in the fashion of incrementally rejecting COTS products by checking the compliance with the requirement. It consists of three templates which are designed to acquire requirements for COTS selection at three stages. Template 1 focuses on acquiring the requirement from product supplier data. Template 2 guides the acquire requirement from supplier-led product demonstration. In particular, it helps examine individual requirement compliance. Template 3 assists evaluating COTS with the requirement acquired through product exploration. PORE is an iterative process. Each stage can be repeated for multiple times.

OPAL is a method with a supporting tool called OPAL. It consists of three phases: identifying goals and requirements, preparing call for tenders and selecting best COTS bid. In the first phase, the customer defines and weights the requirements. Then in the next phase, these requirements are compiled into the software requirement specifications and transformed into a questionnaire. Each question in this questionnaire has a scoring scale and the COTS suppliers need to complete this questionnaire. Finally in the third phase the best COTS product is selected by evaluating the requirement weights and the scores in the questionnaire.

From the description of these COTS selection methods, we can see some common characteristics among of these methods. First, the selection in these methods is comprised of several phases, which commonly include identification and evaluation activities. The examples are OTSO, CRE and OPAL. Second, COTS venders play an important role in these selection methods. In CRE, OPAL and OPAL, the product information is acquired from the vender side. Third, multiple criteria decision technique is widely adopted in these COTS methods such as AHP in OTSO and CRE.

1.4 OSS Selection Process: State of the Art

1.4.1 The Challenge

After many years' development, there has been a large repository of OSS accumulated. For instance, more than 10,000 open source software is registered on a single website, SourceForge.net. Due to developers' common interest or motivation, multiple OSS may share the identical functionality. As an IT manager decides what open source software his team or company will use as their tools or platforms, he may run into a selection problem. For instance, for automatic documentation tool, he has at least 13 options. (http://en.wikipedia.org/wiki/Comparison_of_documentation_generators). Hence which one is the best match is a serious question for him to answer. OSS has a lot attributes. Even though most of these attributes are open to the public, it is still hard and error-prone to select the best alternative without a systematic process. Directly applying the proprietary software selection processes such as COTS selection models on OSS is inappropriate. Essentially, they are different in two aspects: to begin with, proprietary software does not come with its source code. Hence source code is not considered in any of these proprietary software selection methods. However, source code is an important factor affecting both OSS reuse and maintenance. Next, usually OSS is unfinished and still under development. No parities would ensure its quality, functionalities and support. Users have to adopt it at their own risk. In another word, OSS suppliers will not get involved into this selection. It rests on users' own decision to evaluate the software and make sure it can be used in their projects. To solve the problems, people in software industry start to work on this topic and have proposed a couple of models.

1.4.2 The Navica/Golden Open Source Maturity Model

Navica/Golden Open Source Maturity Model (OSMM) was created by Navica's CEO, Bernard Golden. It is a formal process to assess the maturity level of open source software[19]. People can access this model and use it on their own purpose, which means the model is open source as well.

Basically, OSMM consists of three phases: assessing key product element maturity, assigning weighting factor and calculating product maturity score [20]. Key elements include product software, support, documentation, training, integrations and professional services. These elements are crucial for software success. The assessment of each key element can be divided into four steps: defining what the specific requirement; locating the necessary resources, for example, the means of acquiring technical support; assessing the extent the usefulness of the open source software, for example, is the software well documented for further development, and as the last step, assigning maturity score according to the result from the third step. In phase two, each key element is assigned with a weight factor based on its importance to overall maturity. Then in phase three, the final overall maturity score is calculated by using the following equation: overall maturity score = $\Sigma_{(i=1, n)} (w_i*m_i)$ (m_i is the i-th key element and w_i is the weighting score for m_i).

To facilitate the evaluation process, OSMM provides a set of document templates such as product requirements template, product software template and professional support template etc. These templates set a framework and check list for each phase or step in OSMM. For instance, the technical support maturity assessment template gives a list of how many points each type of technical support are assigned. A user can use the support checklist to arrange his/her support assessment activities. The reviewed result can be recorded in the support assessment table.

1.4.3 CapGemini Open Source Maturity Model

CapGemini[21] uses its model to provide consultation service for its customers. It is an evaluation work flow, which involves the interaction between its consultants and customers. The kernel index in this model is “product indicators”. These indicators are results of measuring a number of objective and measurable facts [21] related with the open source software. Product indicators can be grouped into four groups: product, integration, use and acceptance. Table 1-1 [21] could give us a clear idea of each indicator and what characteristics of open software it measures.

Table 1-1 CapGemini Model

Group	Indicator	Purpose	Immature	Mature
Product	Age	The active development time	The project has just started	The software has been developed for a while
	Selling Points	The features which can attract users	None or under-developed	Acceptable and useful
	Developer Community	The group of people who develop the software	Small unorganized group	Active developer community
	Human Hierarchy	The organization of development	Few lead developer	Multiple lead developers
	Licensing	Legal issues	Unclear or unsuited	Commercial and open source uses
Integration	Collaboration	Integration with other software	No consideration yet	Adequate attention to integration with other products
	Modularity	Whether if it can tailored to meet specific requirements	Monolithic code	Tailor-able if necessary
	Standards	Supporting current standards	Incompliant with standards	Compliant with current standards
Use	Support	How to get technical support	Restricted and far from enough support	Highly accessible support
	Deployment	The means of supporting deployment	Hard to deploy and maintain	Easy to deploy and maintain via training and documentation.
Acceptance	User Community	Software user group	Small	Large and well organized
	Market	How much the software has been accepted	Few reference	Many references and successful application cases.

1.4.4 Business Readiness Rating

Business Readiness Rating Model [22] is developed by Spike Source, the Center for Open Source Investigation at Carnegie Mello West, and Intel Corporation. The authors of this model claim to make a model which is Complete, Simple, Adaptable, and Consistent. Analogue to previous models, this model is comprised of four phases: Quick Assessment Filter, Target Usage Assessment, Data Collection& Processing and Data Translation. The rating has five ranks from 1, “Unacceptable”, to 5, “Excellent” [22].

In the first Quick Assessment Filter phase, the final usage is determined, which can be categorized into four types: mission-critical, regular, development and experimentation. With the final usage, users can remove some inappropriate candidates by checking a couple of properties such as licensing, compliance with standards, stable supporting organizations, references etc. This evaluation is coarse-grained, just in aim for filtering out some candidates which are obviously unable to fit the future purpose. Next, users need to select several most important assessment categories. The model recommends these categories should be less than 7. These assessment categories include functionality, usability, quality, security, performance, scalability, architecture, support, documentation, adoption, community and professionalism. These categories are similar to the categories in Navica/Golden and CapGemini. The third phase, data collection and processing, is a time-consuming process, because all measurement data should be collected for each category and compared with a normalized scale. This comparison is to answer the questions such as “I know the software has been downloaded for 2000 times per month, is this a good indicator for its maturity?”. In the last phase, data translation, the final Business Readiness Rating score is calculated based on the ratings computed in the previous phase for each category and the weighting factors. Different from the weighting factors used in Navica/Golden, these weighting factors measures the extent of importance of each metric in each category.

1.4.5 Karin van den Berg's Open Source Evaluation Model

This Open Source Evaluation Model[23] is described in Karin van den Berg's master thesis. The author presents the criteria which are collected from the open source evaluation literature. The criteria cover the important aspects of open source software: community, release activity, longevity, license, support, documentation, security, functionality, integration, modularity, standards, collaboration with other software and software requirements [23]. There are two steps in the evaluation process. First, four criteria, functionality, community, release activity and longevity, are used to select the candidates. The selection is either an elimination of the candidates which do not meet the minimal requirement on functionality and release activity or a ranking base on the four criteria listed above [23]. The first selection method is called Elimination by Aspects model and the second one is the Linear Weighted Attribute Model. The author gives a case study to show the effectiveness the model by using Course Management System.

1.4.6 David A. Wheeler's Open Source Evaluation Model

According to the model [24], OSS is evaluated in four steps: identify, read reviews, compare and analyze. Identify step is to find out what potential candidates are available. David A. Wheeler gave several recommendations to fulfill this task. The first is the well-known open source software lists such as his own Generally Recognized as Mature (GRAM) list, the IDA Open Source Migration Guidelines and the table of equivalents/replacements/analogs of Windows software in Linux [24]. The second choice is doing a search in open source websites. After identification, users need to read existing reviews. The author pointed out users should notice the popularity and market share as they read the reviews. He listed two reviews in his paper. One is a Content Management Problems and Open Source Solutions and the other is Software Configuration Management Systems [24]. The third step is comparing the crucial software attributes with users' requirements. These attributes are functionality, cost, market share, support, maintenance, reliability, performance, scalability, usability, security, flexibility/customizability, interoperability and legal/license issues [24]. At last, users need to do an analysis among the most competitive candidates. The author recommended using the software on representative work loads [24], which is close to software testing.

1.4.7 Summary

These OSS evaluation models are intuitive and not as well defined as those COTS evaluation methods. They give an OSS selection guideline which is still not quite operable. For instance, in BRR, to measure the reference deployment, users have to find out whether the software is scalable and tested in real use. But how to do that is questionable since few OSS provide such information. Also, BRR mentions measuring the difficulty to enter core developer team. If the waiting only takes a while, then the software would be ranked as excellent on this point. But how long is a while? Although most of methods point out the various attributes that OSS users should consider during the selection process, how to organize the various attributes and make a wise decision has not been elegantly solved. One example is OSMM. It treats support and documentation as parallel evaluation criteria. But in reality, the documentation is a means of support for OSS projects. In addition, these OSS evaluation methods do not emphasize the source code evaluation. But source code is a most important component of OSS, which may affect the OSS maintenance and further development.

In order to solve these problems, we design a new OSS selection process. Our process reuses the mature practices which have been adopted by proprietary software evaluation methods. For instance, as with OTSO and CRE, the process uses analytic hierarchy process to organize the multiple criteria and reach a wise decision. Also, our process takes into account the characteristics of OSS. One example is the criterion design. To ensure the practicability, we provide several indicators which can help measure the OSS alternatives under certain criteria. For sake of simplicity, our selection process does not consider the non-functional requirements. But we believe it can be extended to handle these requirements without major changes. In the end, we will give a case study to show the effectiveness of our selection process.

1.5 OSS Selection Process Based on Analytical Hierarchy Process

1.5.1 The OSS Selection Process

We design a new empirical OSS selection process (figure 1-2). This process reuses some ideas from COTS selection methods and addresses the OSS own characteristics. Basically it consists of three basic steps: identification, screening and evaluation. The identification step is similar to the search phase in OTSO and the identification phase in CRE. The goal is also to find all possible alternatives which may meet our major requirements. At this step, we only need to apply the high-level requirements. For example, if we want to find OSS which supports Internet Relay Chat (IRC), then IRC is the criterion regarding identifying the potential OSS. The OSS information sources are similar to the sources discussed in COTS selection process [66][16], which include Internet, publications, conference, expert/co-workers and Linux distributions. Specifically, OSS websites are an important resource on the Internet [24]. These websites can be categorized into two types. One type is the project host websites, which provide necessary infrastructure for OSS development. An example is Sourceforge.net (www.sourceforge.net), the largest OSS host website in the world. Another example is GNU's Savannah (http://savannah.gnu.org/). The other type website is OSS index websites. These websites usually collect an OSS repository and provide some information or comments related with each OSS project. The examples are Freshmeat (freshmeat.net), IceWalkers (www.icewalker.com), Ohloh (www.ohloh.net) and Free Software Directory (http://directory.fsf.org/). The next step is screening, in which the refined requirements are applied to reduce the OSS alternatives. The requirements used in this step aim at the OSS distinct properties such as underlying platform, implementation language, dependent modules, standard compliance and license. Usually the OSS project will provide such information directly. Therefore the screening would not take a large amount of time. The criteria can be expressed as a scope or range which users can accept. For example, users require the OSS should be able to run on either Windows or Linux. Then the OSS alternatives only running on MacOS or some embedded OS such as vxWorks and WinCE are screened out in this step. The remaining OSS alternatives after this step are believed to be the competitive candidates. Locating the best fit among them requires a scrutiny, which is undertaken in evaluation process. Among these steps the evaluation is the most important. It is worth noting that the process is iterative. If we realize that our initial criteria are too strict and should be adjusted during the evaluation or screen, we can always go back to the prior step and start from there again.

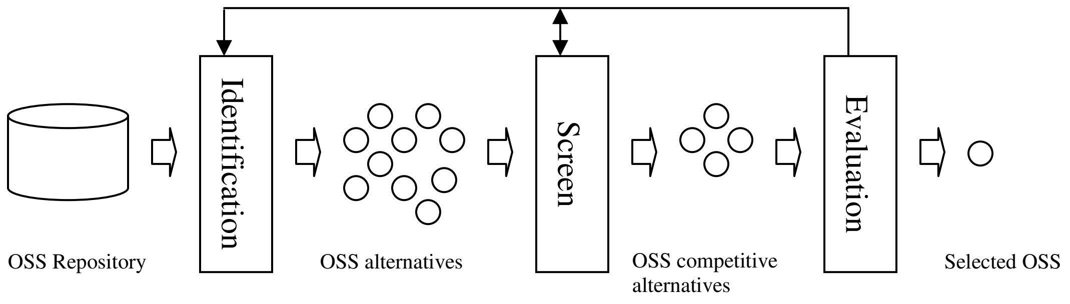

Figure 1-2 OSS Selection Process

The evaluation step is based on the Analytic Hierarchy Process. OSS alternatives are inspected carefully from functional suitability, source code, support strength and popularity (figure 1-3). As for functional suitability, we sort out the OSS alternatives according to how well they satisfy the functional requirements. Under the source code criteria, we want to find out which OSS alternative we are more willing to work on for maintenance and reuse. Support strength evaluation refers to examining the support availability of the OSS. Popularity evaluation means determining how popular each OSS alternative is relative to others.

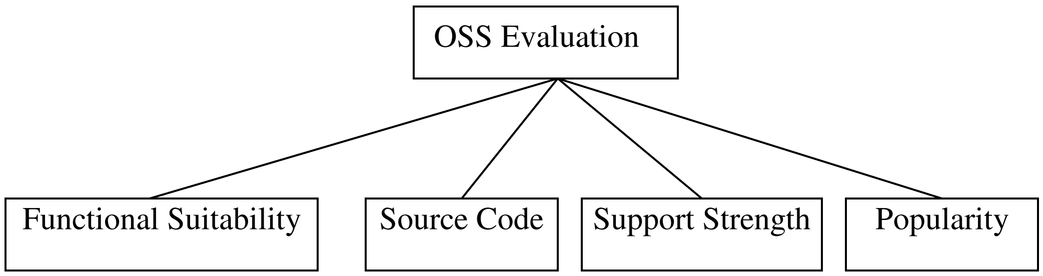

Figure 1-3 OSS Evaluation

Here is the organization of the whole report. The rest of this section will give an introduction of AHP. Next we will give a description of our OSS multiplicity observation. In section 3, we will discuss OSS license because it is an important issue in the screen step. Since evaluation is the major step in our OSS selection process, we will focus on it in the next two sections: section 4 for functional suitability and source code; section 5 for support strength and popularity. In the last section, we will give a case study regarding the application of our selection process.

1.5.2 The Analytic Hierarchy Process (AHP)

AHP is a decision making technique proposed by Thosmas L. Satty. It allows decision makers to model a complex problem with multiple attributes or criteria into a hierarchical structure [25] [26][27]. Basically AHP can be decomposed into five steps:

1. Establishing the hierarchical structure in view of the objective, criteria, alternatives and their relationships.
2. Do pair comparison between the elements regarding each criterion at its subsequent level

3. Compute the priority vector based on the pair comparison results.
4. Compute the consistency ratio. If the consistency ratio is out of a reasonable scope, tune the pair comparison value until the consistency is acceptable
5. Develop the final priority vector of the alternatives for the final objective.

The first step is establishing the hierarchical structure in view of the objective, criteria, alternatives and their relationships. The objective is at the top of the hierarchy. It is the ultimate evaluation goal. Below the objective is multiple criteria (C_1, …, C_m), which need to be examined in order to make a decision on the objective. Each criterion may depend on sub-criteria on the next level but the criteria on the same level should be independent. This criteria refinement continues until the evaluation of alternatives (A_1, …, A_n) can be carried out. Figure 1-4 gives an example of such hierarchy. From this figure, we can see Criterion$_2$ at the second level depends on three sub-criteria, C_{21}, C_{22} and $C_{23.}$ Under these sub-criteria are alternatives.

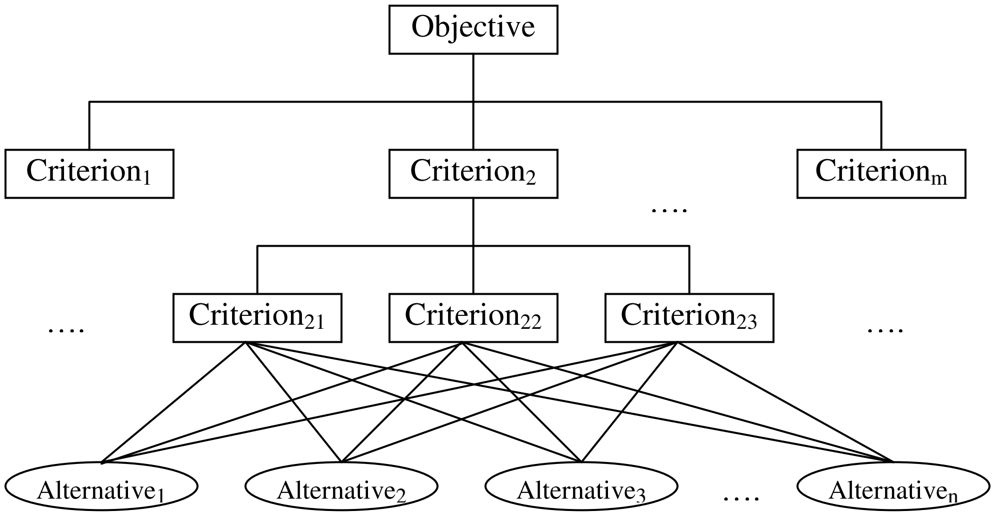

Figure 1-4 Analytic Hierarchy Process

At the second step, the dependent elements of each criterion at the subsequent level are pair wise compared and the priority of these elements is inferred from the comparison results. For example, under C_{21}, the alternative group, A_1, …, A_n, are compared in pairs. Thereby we have a reciprocal matrix A_{n*n}. The element of this matrix, a_{ij}, is the relative preference between A_i and A_j evaluated with respect to C_{21}. In another word, A_i is a_{ij} times as preferable as A_j. The value is a number on the scale recommended by Thomas Satty. Table 1-2 shows more details of the scale [27]. The upper limit is set as 9 because it is sufficient to make a distinction and compliant with the psychological limit of human [27][29].

Table 1-2 AHP Ranking Scale

Intensity of importance	*Definition*
1	Equal importance
3	Weak importance of one over another
5	Essential or strong importance
7	Very Strong or demonstrated importance
9	Absolute importance
2,4,6,8	Intermediate values between adjacent scale values
Reciprocals of above nonzero	If activity I has one of the above nonzero numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i
Rationals	Ratios arising from the scale

The priority rankings are derived by computing the eigenvector from the pair wise matrix. Let us give an explanation about this computation [26][27]. Assume the exact priority vector of $A_1, \ldots, A_n$ is $w=\{w_1, \ldots, w_n\}$. Then if the pair wise comparison is precise, element a_{ij} in matrix A should equal to w_i/w_j. Then we have the following equation [27]:

$$\begin{bmatrix} \frac{w_1}{w_1} & \frac{w_1}{w_2} & \frac{w_1}{w_3} & \cdots & \frac{w_1}{w_n} \\ \frac{w_2}{w_1} & \frac{w_2}{w_2} & \frac{w_2}{w_3} & \cdots & \frac{w_2}{w_n} \\ \cdots\cdots & \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ \frac{w_1}{w_n} & \frac{w_2}{w_n} & \frac{w_3}{w_n} & \cdots & \frac{w_n}{w_n} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_n \end{bmatrix} = n \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_n \end{bmatrix}$$

From the matrix theory, we know w is an eigenvector of A with eigenvalue n. There are several methods to estimate eigenvector. One method is the process of averaging over the normalized columns [27][28]. Below are the details of this process. The second method is simpler: First, multiply the n elements in each row and take the nth root and then normalize the results [27]. In addition, it is possible that we have the priority vector without going through the pair comparison. In this case, we must make sure the priority ratios truly represent the judgment.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots\dots & \dots\dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} => \begin{bmatrix} \frac{a_{11}}{\sum_{i=1}^{n} a_{i1}} & \frac{a_{12}}{\sum_{i=1}^{n} a_{i2}} & \frac{a_{13}}{\sum_{i=1}^{n} a_{i3}} & \dots & \frac{a_{1n}}{\sum_{i=1}^{n} a_{in}} \\ \frac{a_{21}}{\sum_{i=1}^{n} a_{i1}} & \frac{a_{ss}}{\sum_{i=1}^{n} a_{i2}} & \frac{a_{23}}{\sum_{i=1}^{n} a_{i3}} & \dots & \frac{a_{2n}}{\sum_{i=1}^{n} a_{in}} \\ \dots\dots & \dots\dots & \dots\dots & \dots & \dots \\ \frac{a_{n1}}{\sum_{i=1}^{n} a_{i1}} & \frac{a_{2n}}{\sum_{i=1}^{n} a_{i2}} & \frac{a_{n3}}{\sum_{i=1}^{n} a_{i3}} & \dots & \frac{a_{nn}}{\sum_{i=1}^{n} a_{in}} \end{bmatrix}$$

$$=> \begin{bmatrix} \sum_{j=1}^{n} \frac{a_{1j}}{\left(\sum_{i=1}^{n} a_{in}\right)} \Big/ n \\ \sum_{j=1}^{n} \frac{a_{2j}}{\left(\sum_{i=1}^{n} a_{in}\right)} \Big/ n \\ \dots \\ \sum_{j=1}^{n} \frac{a_{nj}}{\left(\sum_{i=1}^{n} a_{in}\right)} \Big/ n \end{bmatrix}$$

The pair wise comparison is based on subjective judgment, which may bring the inconsistency problem. The inconsistency means there exists elements a_{ij} and a_{jk} in A, $a_{ik} \neq a_{ij} * a_{jk}$. For instance, if A_i is 2 times more important than A_j and A_j is 3 times more important than A_k, then A_i should be 6 times more important than A_k. However, A_i may be deemed as only 4 times more important than A_k, which leads to an inconsistency. In this case, based on the matrix theory [26][27], the priority vector w should satisfy the equation $Aw=\lambda_{max}w$. λ_{max} is the maximum eigenvalue of A. The inconsistency can be estimated by measuring how close λ_{max} is to n, i.e. calculating the consistency index

$(\lambda_{max}$-n)/(n-1). The consistency index divided by the average consistent index of the randomly generated reciprocal matrix (its elements are from 1 to 9) is consistency ratio [27]. AHP allows inconsistency to some extent. In practice, if the consistency ratio is less than 0.1, we believe the inconsistency is acceptable.

Once all priority vectors under each criterion have been determined, we can continue to compute the final priority vector of the alternatives for the objective. The computation is from bottom to top. For instance, assume the priority vectors under C_{21}, C_{22} and C_{23} are v_1, v_2 and v_3. The priority ranking vector of $C_{21,}$ C_{22} and C_{23} under C_2 is u. Then the priority vector of the alternatives for criterion C_2 equals to $[v_1\ v_2\ v_3]*u$. $[v_1\ v_2\ v_3]$ is an n*3 matrix. In the same way, we compute the final priority vector after all priority vectors of the alternatives under criteria C_1, …, C_m have been computed.

2 OPEN SOURCE SOFTWARE MULTIPLICITY OBSERVATION

2.1 What Is Open Source Software Multiplicity?

The reason behind the multiple choices for OSS users is multiple OSS projects share common features. We call this phenomenon as open source software multiplicity in this report. An OSS project has many features. These features are listed in the project's description, document or website. Project description is the mission statement for the project. Usually it is a short introduction written by project initiator to brief the selling points of the project. For example, according to OSS project Notepad++'s description (http://sourceforge.net/projects/notepad-plus/), its features include source code editor, written in C++ with Win32 API, supporting editing several programming languages and customizable GUI. All of these features can be divided into two categories: functionality and non-functionality. In this example, C++ and Win32 API implementation is the non-functional feature and the rest belong to the functionality category. Among the functional features, source code editor is the most important since it defines the basic usage for this OSS. The other functional features can be seen as extension or enhancement based on this feature. In the paper, we define an OSS project's fundamental functionality as its key feature. If two OSS projects have the identical key features, they are deemed as similar projects or we call the two projects match. In practice, key feature can be used in the OSS identification step. It can be extracted from the high-level requirement and serve as the key words to search the OSS projects. The result is a list of similar OSS projects.

OSS project hosting websites have done some work on partitioning OSS projects based on common functional features. Projects are clustered into different topics or categories to help users locate the software projects. Sourceforge.net provides an up to four level software map. Likewise, Freshmeat.net has two level software categories. In Sourceforge.net, a topic is first proposed by certain open source software developers, and then publicly reviewed by all website users. If it is agreed by many people, it may be added into the Software Map [30]. In Freshmeat.net, the categories are determined by website administrators. It is project initiator who decides which topic or category the OSS project belongs to. The classification provides a means of project functionalities filtering. For instance, if a user wants to have a FTP client on SourceForge.net, she can simply check the software list under the topic directory path: Internet->File Transfer Protocol (FTP). The shortcoming of these classifications is that they are a bit over coarse-grained. In the previous example, there are 620 software projects under the FTP topic. Comparatively, key feature is more precise and closer to the real functional requirements from users. Also, these classifications tolerate some ambiguity and arbitrariness. For instance, both StarUML (http://sourceforge.net/projects/staruml/) and UMLet (http://sourceforge.net/projects/umlet/) are UML modeling tool, but on SourceForge.net, StarUML is under topic CASE and UMLet's topic is undefined.

In this report, we will give a rough observation of OSS multiplicity. It relies on the subjective judgment and the result may not be precise. However, our goal is limited, i.e. to indicate the possible extent of the multiplicity. In another word, we want to investigate the problem: for an OSS project, is it possible that its key feature overlap with any other OSS project key features? The answer is not of benefit to the OSS selection process but it can let us know more about the necessity of OSS selection. Since the OSS project repository is huge, it is impossible to check every OSS project. We will use Simple Random Sampling (SRS) approach to get a sample which can represent the whole OSS repository. In order to reduce subjectivity and ambiguity, a set of rules regarding key feature extraction and similarity criteria will be defined. An alternative of SRS may be statistical clustering, which partition a set into subsets (clusters). OSS project multiplicity can be measured based on the cluster sizes. Compared with SRS, this approach can give a more precise estimate of OSS project multiplicity. However, we do not use it here because, first, how to group OSS projects into different categories is not our main concern; second, it may involve too much work. Key feature extraction and similar project searching rely on personal judgment and are hard to automate.

2.2 Sampling Design

2.2.1 Simple Random Sample

Simple Random Sampling is one of the most widely used sampling techniques. A simple random sample with size *n* is drawn from the population *N* without replacement such that each possible sample of size n has the same chance of being chosen [32]. In our SRS design, the population is the OSS projects registered on SourceForge.net. Until November 12, 2006, there are 114,701 project registered on SourceForge.net. We assume these OSS projects are indexed from 1 to 114,711. The sample is determined by the index set generated by a free randomizer, Research Randomizer (www.randomizer.org). In order to measure OSS project multiplicity, we can estimate the average number of similar OSS projects from the sample at confidence level 95% by using the following equation:

$$n = N\,\sigma^2/((N\text{-}1)D+\sigma^2) \qquad \text{where } D=B^2/4$$ [31]

(n is the sample size, N is the population size. B is the bound on the error of estimation. σ^2 is the population variance.). However, it is not easy to calculate the appropriate sample size because the population variance is not available in this equation. To circumvent this problem, we shift the sampling objective to the population proportion. Specifically, OSS multiplicity can be measured by estimating the proportion of OSS projects in the population which have other similar OSS projects. Then the sample size calculation equation at the same confidence level becomes:

$$n = Npq/((N\text{-}1)D+pq) \qquad \text{where } D=B^2/4$$ [31]

n, N and B are the same as the previous equation. p is the proportion to be estimated and q is 1-p. Even though p is also unknown here, we can assume p=0.5 to assure the maximum variability [31][32]. Now if B=10%, N=110,000, n=110,000*0.25/275.2475=100. Hence, we can randomly choose 100 OSS projects as the sample from the population. To be more conservative, we select 120 projects as our sample in the real observation.

2.2.2 Sampling Implementation

2.2.2.1 Sampling Process

Sampling process can be seen as a simulation of users' OSS searching, i.e. locating the potential candidates which match major functionality requirements. The difference is that in our sampling, the functionality requirements are the key feature extracted from each sample OSS project. Basically, the process consists in three big steps: key feature extraction, key word searching and similar project matching. In the first step, we need to decide the key feature from the OSS project description. Aside from the project description, project website, FAQ and documents are also helpful. In the following section, we will discuss the key feature extraction rules. There exist some cases in which it is impossible to summarize the OSS project's functionalities. The project description is not well written and there are no website, FAQ and documentations. For example, project team-i-share's goal is simply depicted as "all-sharing application" (http://sourceforge.net/projects/team-i-share/), from which the key feature cannot be defined. As such, its key feature is empty and it does not have any similar OSS projects. The output of this step is the key words concluded from the key feature. For instance, the key words for project Notepad++ are "source code editor". The second step is searching the possible similar candidates with the key words from the population. We use the searching function provided by Sourceforge.net. It returns a list of OSS projects sorted by the relevance, which is calculated by looking at the number of times the search string is found in the name and project description [33]. The search function can dramatically filter out unrelated projects but there are still too many OSS projects included into the list that are not the real match. In order to find out the similar OSS projects, we have to go through the list, read each project's description and even documents and website if available. Since the sampling involves personal knowledge and judgment, several rules are stipulated to minimize possible subjective mistake and ambiguity.

2.2.2.2 Key Feature Extraction

OSS project descriptions do not have a standard format. Mostly they are just a paragraph including the main goal and the important features the projects will implement. However, this is not always the case. Some OSS projects give users some reference software projects or products. Some OSS projects list what tools they use and what platforms they are aiming at. Different OSS project description may give rise to different understanding of its key feature, which poses a problem for extracting key feature. To alleviate this problem, we classify OSS projects based on their goals and set up the rules to guide key feature extraction accordingly.

Generally speaking, OSS projects can be classified into Extension, Specialization, Porting, Replacement, Tool, Localization, User Interface and Originality. Extension refers to enhancing or adding functions to an existing software product. For example, project ZenStar mIRC Script (http://sourceforge.net/projects/zenstar/) is an extension of mIRC, an Internet Relay Client (IRC), (http://www.mirc.com) because it is designed to add functionality such as auto-nick completion, to mIRC. Specialization emphasizes on improving nonfunctional features such as performance, storage, response time etc. For example, project Business Maker ERP (http://sourceforge.net/projects/custom-erp/) advocates enhancing modularity into ERP system. OSS projects in Porting aim at porting some software products from one platform to another platform or implementing them with different languages. For example, GridLayout Panel for .NET (http://sourceforge.net/projects/dnetgridlayout/) implements Layout management concept or layout manager, which originally is on Java, on .NET platform. Replacement means providing an open source solution as an alternative to a software product. In many cases, the reference software is proprietary software or widely accepted open source software. For example, Ao Server Mod By Shura (http://sourceforge.net/projects/aoserverbyshura/) is a project aiming at implementing a free alternative to massive multiplayer online role-playing games. Tools, also called as utilities, are mainly for supporting software development. They are composed of script, API, SDK, library, framework, kit etc. For example, project blex (http://sourceforge.net/projects/blex/) is a platform-independent software development kit. Many OSS projects are created to support foreign languages, for instance, Arabic encoding, Chinese input etc. Therefore we have a Localization class to represent these OSS projects. User Interface is a class of OSS projects which provide a user interface or front end for some software products. For example, project TortoiseDarcs (http://tortoisedarcs.sourceforge.net/) is a GUI frontend for darcs, an open source source code management system. In reality, these OSS projects can also be ascribed to Extension or Specialty (for enhancing user-friendly). However, since the number of this type of OSS projects is substantial, we put them into a separate class. The last class is Originality. By name the project in this class should contain innovative work or ideas. In addition, it cannot be merged into any other classes. Actually some of major motivations of OSS developers are intellectual challenge and self-actualization, which drive them to do something new and innovative [34][35]. One example is Arcade Mass Conspiracy (http://sourceforge.net/projects/arcadeworkers/), which is to produce arcade games in functional language, Ocaml.

The OSS project classification is helpful for determining key features. Some types of features are crucial for a certain class of OSS projects but may not be considered as the key feature for another class of OSS projects. For OSS projects in Extension, Specialization and Replacement classes, key feature should include the reference software because it is a determinant of the software basic use. In particular, the key features of Extension projects should subsume the added enhancements, for they constitute the most important selling points for these projects. But the nonfunctional enhancement in OSS Specialization projects will not be treated as key feature because these features are hard to evaluate and different users may have different standards. The verification of these features tends to happen in the later screening stage. Usually the key feature does not contain programming languages. However, for OSS Tools projects, it is added up to keep key feature in line with their ultimate goal. Based on the same reason, for Porting projects, we include the underlying platform into their key features.

2.2.2.3 Similarity Criteria

After we decide the key feature for the OSS project, the search with the input of the key words generates a list of OSS projects. There are several criteria to help locate the similar projects among this list. For simplicity, suppose we have sample project A and project B in the generated project list. We use FK_1 to denote the key feature of sample project A and FK_2 as the key feature of project B. First, if A is in Extension class and B is similar to A, B should also belong to Extension class as well. This is also the case for projects in Localization, UI and Tool classes. Second, FK_1 contains multiple key features. If any key feature in FK_1 is also in FK_2, then B is an A's similar project. The example is a Localization class project: Write Dari and Farsi with Naveesinda (http://sourceforge.net/projects/naveesinda2-0/). It provides the inputting function on English keyboard for a set of languages such as Farsi, Dari, Arabic, Pashtu, Russian, Cyrillic etc. If another project also implements the inputting function for any of these languages, we believe it matches project Write Dari and Farsi with Naveesinda. Third, if FK_1 is a subset of FK_2 then B matches A. For example, project CDDA Ripper XP (http://sourceforge.net/projects/cddarip/) is an audio CD ripper program. Project

BonkEnc Audio Encoder (http://sourceforge.net/projects/bonkenc/) has more features. It is not only a CD ripper but also an audio encoder and converter. Base on the third rule, we believe BonkEnc Audio Encoder matches CDDA Ripper XP. Forth, if FK_2 is a special case of FK_1, or in another word, FK_2 is a non-functional enhancement of FK_1, B matches A. For example, project Snakelets (http://sourceforge.net/projects/snakelets/) is a web application server. Project Porcupine Web Application Server (http://sourceforge.net/projects/porcupineserver/) is an object oriented web application server, which can be seen as a specialization of web application server. As such, project Porcupine matches project Snakelets.

2.3 Sampling Results

The following pie chart (figure 2-1) shows the percentage of each class of OSS projects accounts for among the sample set, which reflects the distribution of OSS project developers' interests. The chart shows the doing something new is the focus for OSS project developers. But there are still enough interests on providing development support and re-implementing a software product to different platforms or with different languages.

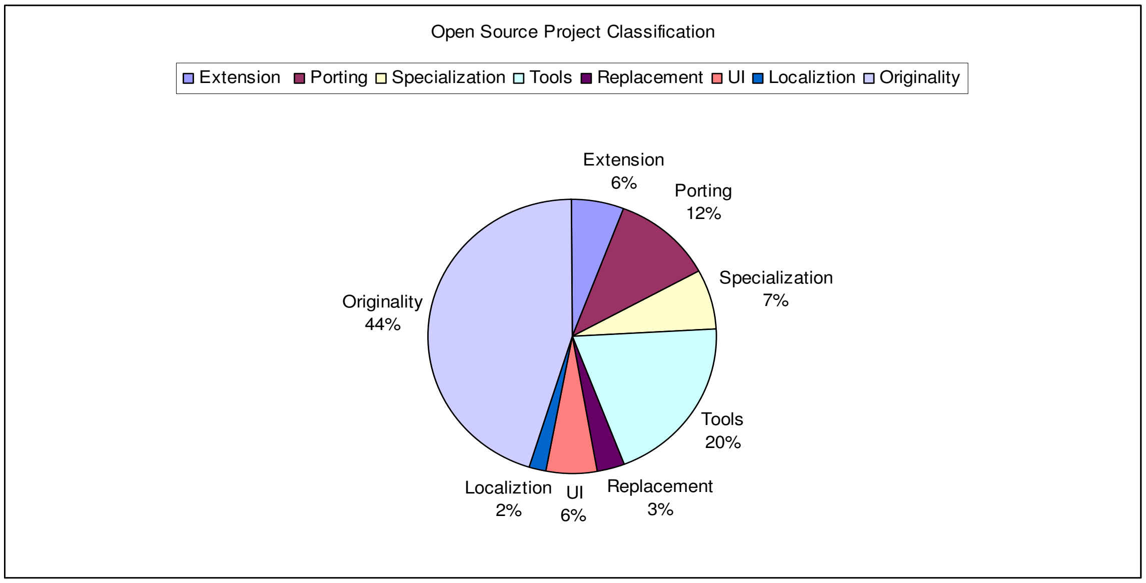

Figure 2-1 Open Source Sample Project Classifications

Table 2-1 is the sampling result. More than half of the sample projects have no less than 1 other similar projects. If we raise the threshold to 3, there are still 25.8% projects left. The confidence interval in this sampling test is 10% (the actual interval is a bit smaller than 10% since we select more than 100 sample projects). The percentage range column lists the range of the percentage estimate in the population, which is (percentage in the sample set – 10%, percentage in the sample set + 10%). From this column, we can easily conclude that there are a substantial percentage of OSS projects which share the same fundamental functionality feature with multiple other OSS projects. Therefore, if we need to use some OSS projects for our project, we may have to make a selection among these similar projects to determine which one is the best fit.

Table 2-1 Sampling Results

Sample Projects	Percentage in the sample set	Percentage Range in the population
Sample projects which has no less than 1 similar projects	52.5%	42.5%~62.5%
Sample projects which has no less than 2 similar projects	36.7%	26.7%~46.7%
Sample projects which has no less than 3 similar projects	25.8%	15.8%~35.8%

3 OPEN SOURCE LICENSING

3.1 Introduction: Intellectual Property Protection

The development of OSS has profoundly affected how software is licensed and distributed. In this section, we will discuss briefly the intellectual property law and introduce the common categories of OSS license. The intellectual property law distinguishes three kinds of creations - copyright, patent and trademark.

3.1.1 Copyright

Per the Copyright Law of the United States of America, "Copyright protection subsists in original works of authorship fixed in any tangible medium of expression, now known or later developed, from which they can be perceived, reproduced, or otherwise communicated, either directly or with the aid of a machine or device." [37]. The 1976 Copyright Act generally gives the copyright owner the exclusive right to reproduce the copyrighted work, to prepare derivative works, to distribute copies or phonorecords of the copyrighted work, to perform the copyrighted work publicly, or to display the copyrighted work publicly [37]. Since copyright protection subsists in the original works of authorship, neither a copyright notice nor a registration is required to obtain a copyright. Copyright simply exists when an original work is created. This rule also applies to software. For example, a software program that is written by an engineer is automatically protected by copyright law without the requirement of filing a registration. The copyright law of the United States of America also states that copyright only protects the expression of the original work of authorship, not "the idea, procedure, process, system, method of operation, concept, principle, or discovery, that described, explained, illustrated, or embodied in such work" [37]. For example, a description of a product could be copyrighted, but this would only prevent others from copying the description; it would not prevent others from writing a description of their own or from making and using the product. This is the fundamental difference between copyright and patent, which will be discussed later in this section.

Not in all circumstances the creator of an original work is the owner of the copyright. Works that are created by employee within the scope of ownership are "works made for hire" [36]. Works that are created for hire are owned by the employer, not the employee, even though he or she is the creator of the original work. Another important concept discussed in the U.S. Copyright law is that the compilation and derivative works are also copyrightable. "The copyright in a compilation or derivative work extends only to the material contributed by the author of such work, as distinguished from the preexisting material employed in the work, and does not imply any exclusive right in the preexisting material. The copyright in such work is independent of, and does not affect or enlarge the scope, duration, ownership, or subsistence of, any copyright protection in the preexisting material." [37].

In addition, the protection of copyright is limited to a certain period of time. Under the current U.S. Copyright law, copyrights last for the life of the author with additional 70 years. For corporate copyright, it lasts the shorter of 95 years from publication or 120 years from creation.

3.1.2 Patent and Trademark

Patents in the United States are governed by the Patent Act (35 U.S. Code), which established the United States Patent and Trademark Office (the USPTO). Section 101 of the U.S. Patent Act defines the general requirements for a patent as “Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvements thereof, may obtain a patent, subject to the conditions and requirements of this title.” [38] Therefore, for an invention to be patentable, it must be new, useful and unobvious. [36] The rights granted under patent law are very different than rights granted under copyright law discussed above. Patent law gives the patent holder the right to exclude others from making, using, offering to sell, selling or importing the invention. In contrast, copyright law protects the expression of the original work of authorship, which prevents others from copying or modifying the original work. Copyright law does not protect against someone else from independently creating the same or similar expression. [39]

A trademark is a word, name, phrase, symbol, design or combination of those, that is used to indicate the source of the goods or services and to distinguish them from the goods or services of others. Trademark rights are to prevent others from using a confusingly similar mark, but not to prevent others from selling the same goods or providing the same services under a clearly different mark [40]. Under the U.S. trademark law, the trademark owner must maintain the quality of the goods or services that are under his or her trademark when the trademark is licensed to others [36]. Thus, based on the OSS principle that the OSS license “must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software” [36], open source license can not include trademark license since an OSS licensor can not maintain control over the quality of the derivative works. [36] However, certain OSS license includes a trademark protection clause to prevent licensees using its trade name or trade marks.

3.1.3 Differences between Copyright and Patent

Table 3-1 below is a brief comparison between copyright and patent:

Table 3-1 Copyright and Patent Comparison

Criteria	Copyright	Patent
Subject Matter	Expression of ideas	Ideas
Rights	Prevent others from copying or modifying of an original work	Prevent others from making, using, selling, offering to sell or importing the invention
Standard	Low - original work of authorship	High - new, useful, unobvious
Registration	Not necessary	Strict procedures to obtain patent registration
Duration	Copyrights last for the life of the author plus 70 years, or for a work of corporate authorship, the shorter of 95 years from publication or 120 years from creation	New patents last for 20 years from the date the patent application is filed

3.2 Categories of Open Source Software License

Open source license comply with the intellectual property law. It is built upon the foundation of intellectual property law, primarily on copyright law. The OSS is owned by its author, who licenses the software to the public under generous terms [36]. The software that is distributed under the licenses which comply with the Open Source Definition and are approved by the Open Source Initiative board of directors is OSI certified open source software (www.opensource.org). OSI certification has become a standard in the OSS community such that the world's largest OSS host web site SourceForge.net requires that the software development projects seeking to be hosted on its website should either use OSI approved licenses or meet the requirements of the Open Source Definition set by the OSI [41].

By the time of this writing, there are over 50 OSI approved licenses listed on the OSI website (www.opensource.org). The large number of the OSS licenses makes it very difficult to understand the characteristics of each license and the differences among them. This fact has brought attention of the OSI board of directors. A License Proliferation Committee was formed to identify and mitigate or remove issues caused by license proliferation. The OSI License Proliferation committee started to divide the OSI approved licenses into groups and help people initially picking a license to use one of the more popular licenses, thereby helping to reduce the numbers of different licenses commonly used. [42] The following groups are listed on the OSI website [42]:

- Licenses that are popular and widely used or with strong communities
- Special purpose licenses

- Licenses that are redundant with more popular licenses
- Non-reusable licenses
- Other/Miscellaneous licenses

From a licensor's point of view, this classification methodology gives some guidance as to which ones are more commonly used, so that a licensor can choose from a smaller population. However, this classification methodology does not divide licenses according to their critical characteristics. The licenses within the same category might have very diverse purposes and suitable to different OSS sub-communities. We will focus our discussion on the nine licenses that are "most popular and widely used or with strong communities" [42] hereafter. This category includes Apache License 2.0, New BSD license, GNU General Public License (GPL version 2), GNU Library or "Lesser" General Public License (LGPL version 2), MIT license, Mozilla Public License 1.1 (MPL), Common Development and Distribution License, Common Public License and Eclipse Public License. OSI recommends developers to consider these license first when they are selecting a license for their OSS [42] and we believe these licenses are more important for our selection process. We will also try to give a more useful classification method based on the characteristics of these nine licenses.

3.3 Licenses That Are Popular and Widely Used or With Strong Communities

We studied 122,798 OSS project on Sourceforge.net with registration date on or before April 22, 2007. We found out that total 89,141 projects, or 72.59% of the population, are licensed under the nine licenses that are defined by OSI as "popular and widely used or with strong communities". Thus, our discussion will be focused on these nine licenses.

As mentioned in the section above, even though the nine licenses were recommended by the OSI as "most popular, widely used or with strong community", choosing one from them can still be confusing. Each of these nine licenses has its own characteristics. They serve the different purposes of licensors and impose different level of restrictions to the licensees. We will use figure 3-1 to explain the development pattern of an "open source community". Understanding of such pattern will help us to better learn what type of OSS community a licensor wants to create, and in turn what level of restrictions are applied by different type of licenses. The figure is revised from the "Virtuous Cycle Model of Free and Open Source Community Activity" from the "Free and Open Source Software Licensing White Paper" by Sun Microsystems Inc. [44]

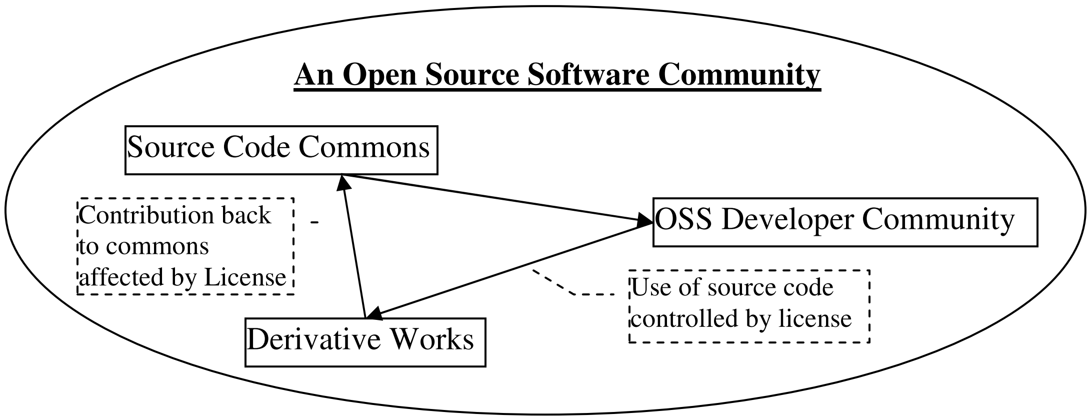
An Open Source Software Community
Source Code Commons
OSS Developer Community
Derivative Works
Contribution back to commons affected by License
Use of source code controlled by license

Figure 3-1 Open Source Software Community Activity (revised and excerpted from [44])

Figure 3-1 shows an endless cycle that the OSS developer community shares the source code commons, creates derivative works based on the source code commons and contributes back to the source code commons. [44]

Derivative works is defined in the U.S. Copyright Law as "a work that is based on (or derived from) one or more already existing works, is copyrightable if it includes what the copyright law calls an 'original work of authorship'."[43] The U.S. Copyright Law also regulates that "only the owner of copyright in a work has the right to prepare, or to authorize someone else to create, a new version of that work." [43] However, one of the OSS principles is that OSS license "must allow modifications and derived works" [36]. Therefore, all OSS licenses must grant the rights to licenses to create derivative works. For a licensor, the decision making of selecting a license for his or her software development depends on what kind of a community he or she wants to create. [44] Different license promotes different structure of how the derivative works can be created based on the original commons, and more importantly, how the derivative works must be distributed. A license with no restrictions of how the derivative works should be distributed will theoretically create a relatively "quiet" community since the licensees who create the derivative work do not have to contribute back to the source code commons. On the contrary, a license with restrictions that the derivative works must be licensed under the same license term as the original source code commons would help build up a rather "growing" community because licensees are committed to distribute the derivative works under the same OSS license, which in turn will keep this community growing. For a licensee, it is important to understand the level of restrictions when selecting OSS, and comply with the responsibility and commitment required by such license.

With understanding of the development pattern of the OSS community, we examine several OSS license classification methodology available today, such as OSI classification discussed in the previous section; the academic licenses and reciprocal licenses discussed in [36]; "the "two principle model" defined in [45]; the "three major categorizations" introduced by Sun Microsystems Inc. [44]; and the "three classes of licenses" based on the "restrictiveness of the agreement" described in [46]. We believe the classification methodologies introduced by Sun Microsystems Inc. and discussed in [46] are meaningful for the OSS selection, which categorizes open source licenses by the level of restrictions applied to the derivative works. This classification considers two critical characteristics of the OSS licenses: [46]

- Whether the open source license requires that derivative works to be distributed under the same terms and conditions of the same open source license. We consider these licenses as "restrictive". [46] This requirement is sometimes referred to as "copyleft" [47], which the Free Software Foundation describes as "a general method for making a program or other work free, and requiring all modified and extended versions of the program to be free as well." [47]
- Whether works that are not derived from program under such license must be distributed under the same terms and conditions of such license if they are distributed with the modifications derived from programs under such license as a whole. We consider such licenses as "highly restrictive" [46]. This requirement is sometimes referred to as "strong copyleft" [48].

Using this classification methodology, we divided the nine "Licenses that are popular and widely used or with strong communities" [42] into the following three categories:

3.3.1 Unrestrictive Licenses

Licenses within this category grant all necessary copyrights and patent rights to licensees free of charge, "including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software" [49]. These licenses place no restrictions as to what licenses need to be used for derivative works. The following licenses belong to this category:

3.3.1.1 BSD License (Berkeley Software Distribution License)

The original BSD license is the first open source license that was designed to promote free use, modify and distribute some software from University of California without any return obligation whatsoever from the licensee. [36] The license permits the redistribution and use the licensed software or its modifications in source and binary forms [50], thus any derivative works based on an OSS product licensed under BSD license can be distributed in any way the creator of the derivative works desires, even in commercial licenses.

This is not to say that BSD license does not impose any conditions on the licensee. BSD imposes three conditions [50]. First, the redistribution of source code must include the copyright notice which is Copyright(c) <YEAR>, <OWNER>, the three conditions and the warranty and liability disclaimer; second, the redistribution of binary code must include the same items in its documents and or other materials provided with the distribution; the third condition is also called the no-endorsement clause stating that "neither the name of the licensor nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission". [50] None of these conditions impose any restrictions to licensee to use, modify, or redistribute the software and no restrictions were set by BSD license as to what license the derivative work must use.

One thing we should notice is that the BSD license we discuss in last paragraphs is actually new BSD license. Its old version includes an advertisement clause which requires the display of an acknowledgement of the University of California, Berkeley and its contributors. This clause has been removed in 1999. However, when we select the OSS alternatives with BSD license, we should notice what version it uses.

3.3.1.2 MIT License

The MIT license [49], also called X license or X11 license, is almost equivalent to the BSD license except that the MIT license gives more clarification about the rights, i.e. "to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software"[49]. Also it does not have the no-endorsement clause.

3.3.1.3 Apache License

The Apache license [51] provides not only grants of copyright license but also grants of patent license which includes the rights of making, using, selling, offering to sell or importing. Particularly as for patent right grants, Apache license contains a termination clause which takes effective if a licensee initiates a patent litigation against the work contributor. Also it includes a clause to protect "Apache" trademark by stating that "This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file" [51].

Similar to BSD and MIT licenses, it has the warranty disclaimer and the limitation of liability clause.

3.3.1.4 Summary

In summary, the non-restrictive licenses share the following characteristics:

- Grant licensee the necessary copyrights and patent rights to use, copy, sell and distribute the software.
- Unrestrictive development of derivative works [44]
- The derivative works can be licensed in any way as the developer desires. No restriction to the way how the derivative works should be licensed.

3.3.2 Highly Restrictive License

We will introduce the Highly Restrictive License before we discuss the Restrictive License, because the most representative highly restrictive license is GNU General Public License (GPL), which is the first OSS license that introduces the idea that any derivative works created from the GPL-licensed software must be distributed under the same license. Similar with unrestrictive licenses, restrictive licenses and highly restrictive licenses also "gives licensees legal permission to copy, distribute and/or modify the software". [52] The key difference between restrictive/highly restrictive licenses and unrestrictive licenses is that the derivative works created based on the software under restrictive licenses and highly restrictive licenses must be licensed under the same terms and conditions in the restrictive licenses. As we discussed previously, such restriction helps to keep the derivative works based on the public commons remain open to public, thus to keep the open source software community growing.

3.3.2.1 GNU General Public License (GPL)

GPL is the most influential OSS license. It aims at ensuring the freedom “to share and change free software – to make sure the software is free for all its users” [52]. It requires any work as a whole must be licensed without any change and under the same GPL’s terms as long as it “contains or is derived” [52] from a portion of GPL licensed program. We need to pay close attention to the scope defined in the GPL, since this is the key difference between restrictive license and highly restrictive license. GPL makes clear that the restrictions applies to non-derivative work if it is distributed with the original GPL licensed work or the derivative work.[52] The restrictions loses effectiveness on the portion of work that are not derived from the GPL-licensed software, can be “reasonably considered independent and separate works in themselves”, and separately distributed. [52] GPL license writers further clarify their intent “to exercise the right to control the distribution of derivative or collective works based on the Program”. [52] Furthermore, GPL does not allow the sublicense the program which is mainly for the prevention of imposing additional restrictions. As with the unrestrictive licenses, GPL also has no warranty and limited liability clauses.

3.3.2.2 GNU Lesser General Public License (LGPL)

The GNU Lesser General Public License (LGPL) is revision based on GPL. It is intended for software library, which is defined in LGPL as “a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables."[53] The main difference between the GPL and the LGPL is that the latter distinguish "the work uses the library" from the derivative work. "The work uses the library" is defined as a program which is compiled or linked with the library and free of any portion of the library and its derivative work. [53] Under LGPL, the program parts other than the library may be licensed under other OSS license or even proprietary licenses, but the source code of the library must be provided. Also, LGPL requires that if the program is an "executable linked with the library" [53], it must provide "object code and/or source code" [53] such that the user can relink the modified library.

3.3.2.3 Summary

In summary, the key characteristic of the highly restrictive license are the follows:

- Grant licensee the necessary copyrights and patent rights to use, copy, sell and distribute the software.
- Unrestrictive development of derivative works [44]
- Modifications derived from software under such license must be distributed under the same terms and conditions
- Works that are not derived from software under such license must be distributed under the same terms and conditions if they are distributed with the software as a whole.

3.3.3 Restrictive Licenses

As discussed in the section of the highly restrictive licenses, the key difference between restrictive license and highly restrictive license is whether works that are not derived from program under a license must be distributed under the same terms and conditions of such license if they are distributed with the modifications derived from program under such license as a whole. The most popular restrictive licenses are Mozilla Public License (MPL), Common Development and Distribution License (CDDL), Common Public License (CPL) and Eclipse Public License (EPL).

3.3.3.1 Mozilla Public License (MPL)

In the late 1990's, due to Microsoft bundling its web browser, Internet Explorer, with the windows operating system, Netscape's web browser product Netscape Communicator rapidly lost its market share. Instead of shutting down the development of this software, Netscape decided to license the Netscape Communicator to the public under an OSS license [36] [54]. Netscape does not want to use the unrestrictive licenses because these licenses do not require modifications to the source commons to be contributed back to the community. It does not choose GPL license either since the company would like to be able to collect modifications to the code made by open source developers and re-license them for use in commercial products. Thus, the Mozilla Public License (MPL) was created by Netscape to meet its special requirements.

MPL is often considered as a hybridization of the BSD license and GNU General Public License. It is worth noting that MPL license uses files to define its governing scope rather than the program in GPL. The restrictions in MPL is applied on the original code and the modifications which is recursively defined as the work derived from the original code or the modifications. MPL defines this work as covered code and requires it to be governed by the license terms. The work other than the covered code is not restricted. To put it another way, "someone can take an MPL-licensed work and build upon it with new components. The resulting work can be distributed with the MPL covering the use of the original work and any license covering the rest. Clearly in this way a company could add closed source components to an MPL-licensed work and thus build a proprietary product." [55] We consider the MPL as restrictive license since MPL requires modifications to be licensed under MPL. But its restriction is weaker than GPL's since MPL allows the added work to use any license as the developer desires. In practice, if we adopt an OSS project with MPL license into our software product, we can limit our modifications within the same files as the OSS source code. These files are distributed under MPL license in the software release. Other source files can either be kept proprietary or released under other licenses. In addition, as with Apache license, MPL also provides the grants of patent license and contains a termination clause.

3.3.3.2 Common Development and Distribution License (CDDL)

The Common Development and Distribution License (CDDL) [56][57] was created by Sun Microsystems, Inc. based on the Mozilla Public License, version 1.1. CDDL is by large the same license as MPL with some improvements, such as "clarified the definition of Modifications, to make it easier for readers to understand what is covered by the license and what is not" and "focused the 'patent peace' provisions to cover only software released under this license" [56].

3.3.3.3 Common Public License (CPL)

The Common Public License (CPL) [58] is a license template created by IBM. Similar to GPL, CPL requires that the changes and additions (contributions) to the CPL-licensed software (Program) must be licensed under the same terms and conditions of CPL if the contributions are distributed under source code form. However, it also states the contributions exclude the works which are “separate modules of software distributed in conjunction with the Program under their own license agreement”, and “not derivative works of the Program.” [58] Therefore, the restriction can be avoided if both conditions are met. CPL emphasizes that it is users’ own risk to make sure “the Program does not infringe the patent or other intellectual property rights of any other entity” [58]. It is important to point out that in order to promote the ability of working with both proprietary software and OSS, CPL includes commercial distribution clauses specifying the responsibilities of commercial distributors.

3.3.3.4 Eclipse Public License (EPL)

Eclipse Public License (EPL) is very similar to CPL. The only slight difference is that EPL does not contain the statement regarding license termination in the case of patent litigation specifically against contributors. [59]

3.3.3.5 Summary

In summary, the restrictive licenses share the following characteristics:

- Grant licensee the necessary copyrights and patent rights to use, copy, sell and distribute the software.
- Unrestrictive development of derivative works [44]

- Modifications derived from software under such licenses must be distributed under the same terms and conditions of such licenses

3.4 Comparison

Table 3-2 summarizes the important characteristics of the nine licenses discussed in the section above. In the OSS selection process, this table can serve as a useful reference when we need to decide if an OSS license is compliant with our selection goal. In particular, there are two things we need to pay attention to. One is that we need to make sure if there is no patent infringement or third party intellectual property right violation. For those licenses which do not explicitly grant patent license, we may need to apply for a patent license from the licensor if the derivative work involves the patent rights [36]. The other thing is we should check whether there are any requirements in the license such as including a copyright notice in the source code.

Table 3-2 OSS Licenses Characteristics Comparison

Licenses	Access to Source Code	No Restriction to Create Derivative Work	Restriction on Licensing Derivative Works	Restrictions on Licensing Non-Derivative Works Distributed with Derivative Works	Trademark protection	Explicit Grant of Patent License
BSD	Y	Y	N	N	N	N
MIT	Y	Y	N	N	N	N
Apache	Y	Y	N	N	Y	Y
GPL	Y	Y	Y	Y	N	N
LGPL	Y	Y	Y	Y	N	N
MPL	Y	Y	Y	N	N	Y
CDDL	Y	Y	Y	N	N	Y
CPL	Y	Y	Y	N	N	Y
EPL	Y	Y	Y	N	N	Y

4 OPEN SOURCE SOFTWARE FUNCTIONALITY AND SOURCE CODE ANALYSIS

4.1 Open Source Software Functionality

4.1.1 Software Functionality

As we evaluate a software product, functionality is one of prime factors which need careful consideration. In ISO 9126 standard, functionality is defined as a part of software quality. Specifically, it refers to the capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions [60]. It consists of five sub-characteristics: suitability, accuracy, interoperability and security [61]. Suitability means the whether the software functionality can meet users' requirements. Accuracy refers to the correctness of software functionality. Interoperability is the capability of the software interacting with other software. Security measures the capability of the software in perspective of hacking prevention. Usually software functionality is evaluated in two scenarios: in the software testing stage or before adopting a third-party software product. The evaluation can be categorized into four levels: functional testing, checklist, component testing and formal proof [61]. Functional testing covers how well the system executes the functions it is supposed to execute – including user commands, data manipulation, searches and business processes, user screens, and integration [62]. In order to measure software functionality in an objective way, function point metrics were developed by Allan Albrecht of IBM in 1979 [63]. It reflects the size of functionality in users' point of view.

There is a lot of work related with software functionality evaluation which has been done. Barbara Ann Kitchenham & Lindsay Jones suggest a feature analysis evaluation should include five areas: scope of the evaluation, basis of the evaluation, roles and responsibilities, procedures, assumptions and constraints made and time scale and effort involved. The evaluations could be carried out in four ways: screening mode, case study, formal experiment and survey [64]. In accompany with software quality definition in ISO 9126, ISO/IEC also issued an evaluation process standard: ISO 14598. The process consists of four phases [65]. The first phase is establishing evaluation requirements, which means identifying evaluation purpose and setting up a quality model. The second phase is specifying the metrics related to the ISO 9126 characteristics. Thereafter the evaluation goes to the third phase: designing the activities and plan the necessary resources. The last phase is carrying out the evaluation and recording the results. Requirements analysis Software functionality evaluation has been intensively applied in COTS selection. OTSO (Off-The-Shelf-Option) method uses the required main functionality (e.g., "visualization of earth's surface" or "hypertext browser") and some key constraints (e.g., "must run on Unix and MS-Windows" or "cost must be less than $X") as its COTS search criteria. [66] In Function Fit Analysis [67] mentioned in [68], the functional suitability is evaluated by 'fit' calculation, i.e. calculating the percentage of COTS out-of-box functions over the required functions. The method is simple since it ignores functions which may require modification or enhancement work. But these functions should be taken into consideration for evaluating OSS functional suitability.

Procurement-Oriented Requirement Engineering (PORE) [17] points out the necessity of analyzing the functional requirements but it does not give a detailed description of how to accordingly filter the COTS alternatives. Alejandra Cechich and Mario Piattini presented a method to evaluate COTS functional suitability in the early stage [68]. In their method, functionality is divided into several sets according to how COTS' functionality matches with required functionality [68]. More details regarding this method will be discussed later.

4.1.2 OSS Functionality Overview

OSS functionality is mainly proposed from two sources. As we have discussed in the first section, the first is the OSS project authors. The second is OSS project developers and users. The functional requirements are requested from the project communities via various ways such as email and discussion board. Normally these requirements are simply assertions without references. In some case, the requirements are elicited because some developers want them and they are willing to provide efforts to make them operational. [6][69].

Open source software functionality is described in many places on its project website. The first is the project news or feature list. When a new version is released, the project team will post a message which may include a brief introduction for the new version's functionality. Some OSS projects will summarize its major functions on the feature list. The second is the documents such as user manuals. The documents clearly present what features the software possesses in more details. The third is the feature request tracking system. This system keeps a list of feature requests submitted by users. The requests are reviewed by developers. If a request is accepted, the task of implementing the request will be assigned to a developer. The request has two statuses. One is open denoting it has not been handled yet. The other is closed which means the request has been handled. Going through these requests could give us an idea of what functional changes or enhancement may be involved for the future versions. The forth is the change log or CVS log. According to GNU coding standards, these logs record all changes developers make to the source code which includes the enhancements to the software functionality. Besides, these logs also list what functional errors or bugs have been fixed by code changes.

On Sourceforge.net, OSS projects are categorized into different topics according to their functionality. [70] shows the number of projects in each top categories. However, their revelation is only one snapshot of the project repository on Sourceforge.net. We did a similar survey but with the aim of disclosing OSS functional distribution in a longer time period. Figure 4-1 shows the number of registered OSS projects in different years on sourceforge.net. We can see Internet, Software Development, System, Communication, Games/Entertainment, Multimedia and Scientific/Engineering stay the top seven topics from year 2001 to year 2007. There are some changes in the rest of the topics. The number of Database projects was surpassed by the number of Office/business projects around 2005; the same case for Desktop Environment and Education projects. These ordering changes show more efforts from the open source community have been invested to Office/Business and Education areas. The projects in topic Formats and Protocols grow fastest. In 2001, there were only 14 such projects but in 2007 the projects boom to 1708, 122 times as many as six years ago.

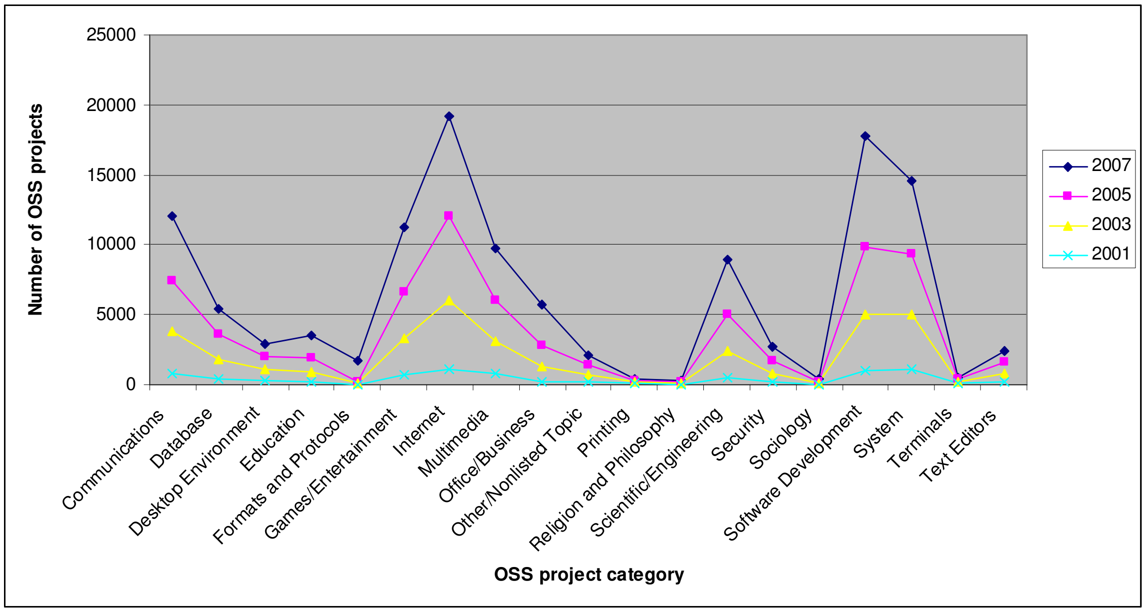

Figure 4-1 OSS Project Distribution on Different Topics

4.1.3 A Case Study of OSS Functionality Evolution

To gain a better understanding of OSS functionality, let us take a look at how OSS functionality evolves. According to Lehman's software evolution law VI: functional content of a program must be continually increased to maintain user satisfaction over its lifetime [71]. Here we use OSS project Emule (http://sourceforge.net/projects/emule/) as a case study to examine if this law is still applicable to open source software which has a different development process. The software metric for measuring functionality evolution is the number of features added per release version. The data source is the change log. As we have mentioned in previous paragraphs, the change log records the changes which the new version possesses in comparison with the old versions. In our cast study we only count the number of newly added features. We investigate the change log of project Emule from its first version (version 0.02) to current version (version 0.47c). The development time spans more than 4 years, from July, 2002 to September, 2006. Figure 4-2 clearly shows the functionality evolution of OSS project Emule is compliant with the Lehman's software evolution law VI.

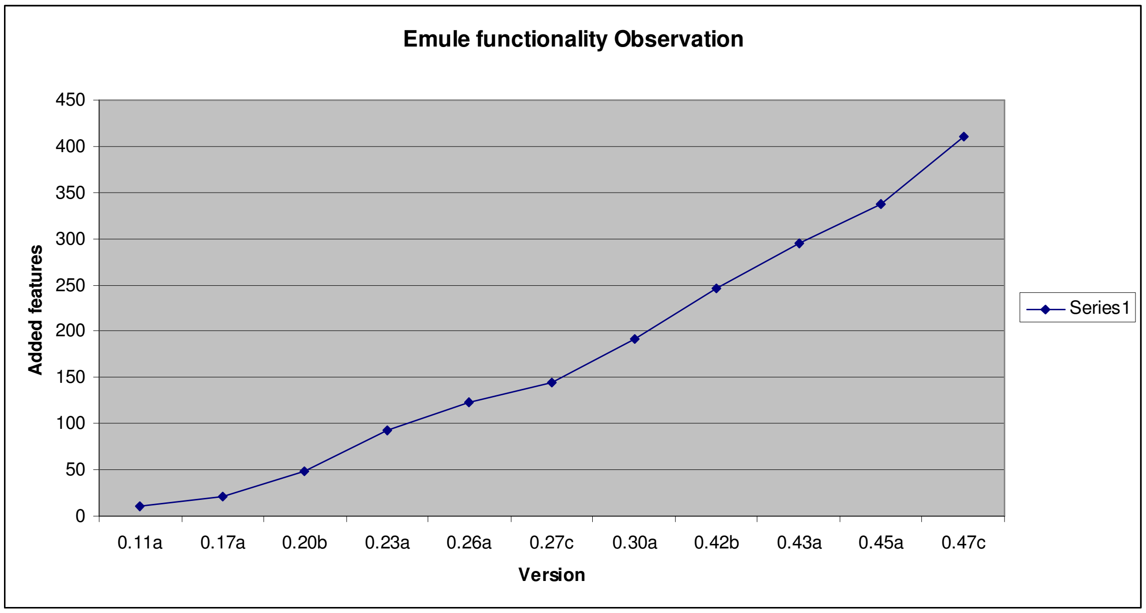

Figure 4-2 Observation of Functional Evolution of OSS Project Emule

4.2 Open Source Software Functionality Suitability Evaluation

4.2.1 OSS Functional Suitability

Analog to evaluating proprietary software functional suitability, OSS functional suitability should also take into consideration how much the software functionality can fit in with the requirements. The difference is, for users, proprietary software is a "black box". If there is a feature which is neither provided by the software nor extendable from the existing API or interface, users may have to dismiss the software. In the worst scenario, they cannot find any potential matches and have to start to implement their own software from the scratch. OSS can be used as a black box as well. But at the same time, it can be seen as a “white box”, since its source code is always available. Users could implement the features they want and adapt OSS for their own purposes via working on the source code. Therefore, OSS features could be inspected from three criteria: OSS out-of-box functions and functions which can be extended from the source code. On the one hand, since OSS functional suitability has something in common with proprietary software functional suitability, we might be able to reuse some existing functional suitability evaluation. On the other hand, OSS functional suitability has its own characteristics. We have to change or extend the evaluation method to adapt it to open source software.

Here we choose the COTS assessment developed by Alejandra Cechich and Mario Piattini [68] as the starting point of our OSS functional suitability evaluation. In our point of view, compared with other methods, this approach is more formally defined and the authors provided a case study to illustrate its effectiveness. Let us give a brief introduction of Alejandra Cechich and Mario Piattini’s COTS functional suitability assessment. This method is adapted from a component assessment invented by Dr. Alexander [72]. It assumes there is a framework A defined with scenarios and the component type that it can adopt is C. The specification of A defines the specification of C, S_c. The specification of component K_i, S_{ki}, should be compliant with S_c. There are two mappings mentioned in Dr. Alexander’s method: synthetic mapping and semantic mapping. The first evaluates interface matching and the second one measures behavior compatibility. The Alejandra Cechich and Mario Piattini’s functional suitability evaluation focuses on semantic mapping measured by semantic distance. Suppose the function is the mapping relationship between values in the input domain and output range. It is believed the semantic inconsistency is caused by the mismatching in input domains, output ranges or mappings between input domain and output ranges [72]. For example, figure 4-3 (modified from the figure in [72][68]) illustrates a typical situation in which inconsistent mappings exist between the intersecting domains and ranges. From the figure, we could see functional mappings a->α, c->γ , d->ε and e->λ cannot be matched because the argument of function in S_c falls out of the domain intersection such as a-> α, the image of the function falls out of the range intersection such as c-> γ and d-> ε or the arguments of the functions in S_k falls out of the domain intersection such as e-> λ. Even if both the argument and image of function fall into the respective intersections, the inconsistency may also arise due to the unequal mappings such as b->β and b->δ.

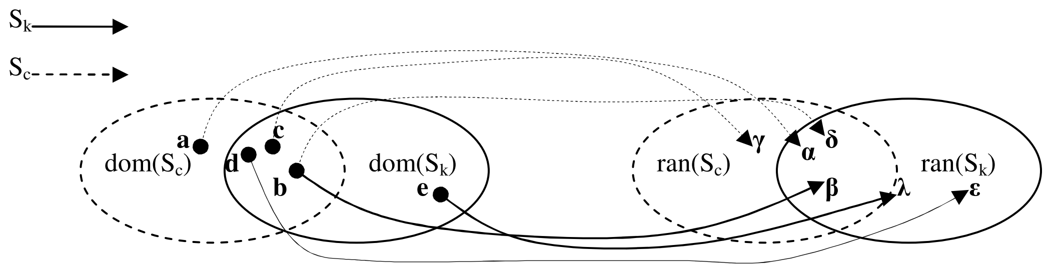

Figure 4-3 Example of Inconsistent Mappings (modified and excerpted from [72])

To analyze the semantic inconsistence systematically, Alejandra Cechich and Mario Piattini designed two groups of scenario-based measures. One is called component level and the other is solution level. These two groups are similar except component level measures estimate the inconsistency for a particular component and solution level measures include all components in the solution. In Component level group, there are four criteria: Compatible Functionality (CFc), Missed Functionality (MFc), Added Functionality (AFc) and Component Contribution (CC_f). CFc is the number of common functional mapping shared by S_k and S_c. MFc is the number of functional mappings in Sc but not in S_k. Conversely, AFc is the number of functional mappings in S_k but not in S_c. CC_f indicates the percentage of functional requirements in S_c could be satisfied by S_k.

Alejandra Cechich and Mario Piattini's functional suitability evaluation method has some limitations. First of all, it treats every function equally. But in reality, some functions are definitely more important than others. For instance, assume we have component C_1 and C_2. $CFc_1 > CFc_2$ but C_2 has a critical function which C_1 does not have. Then it is not clear for users which component they should choose. Second, the metrics used in the method are only a coarse indicator of suitability on analyzed components [68]. The authors did not provide a screening process based on these metrics. To adapt this method to evaluate OSS functional suitability, we will enhance it with Analytic Hierarchy Process (AHP). For sake of simplicity, in this paper we do not consider the functional suitability on solution level. We assume our goal is finding out one OSS alternative rather than several OSS alternatives to meet our functional requirements.

4.2.2 OSS Functional Suitability Evaluation Process

As mentioned, our functionality suitability measurement is based on AHP. Here we reuse notation S_k and S_c. S_k refers to the function set provided by the OSS and S_c refers to the function set from users' requirements. In our measurement, the alternatives are OSS alternatives. The criteria to evaluate the functional suitability are divided into two levels. The criteria in the first level are out-of-box function compatibility (OF): the functions shared by S_k and S_c, extensible function compatibility (EF): functions which are a subset of S_c and could be extended from the OSS, and contributive function compatibility (CF): functions which are provided by S_k but not required by S_c. If we use equation to define these sets, we have CF = $S_k \cap S_c$; EF = $S_c - S_k$; and CF = $S_k - S_c$. At the second level, OF is further divided into functions in the S_c: $f_1, \ldots, f_n$.

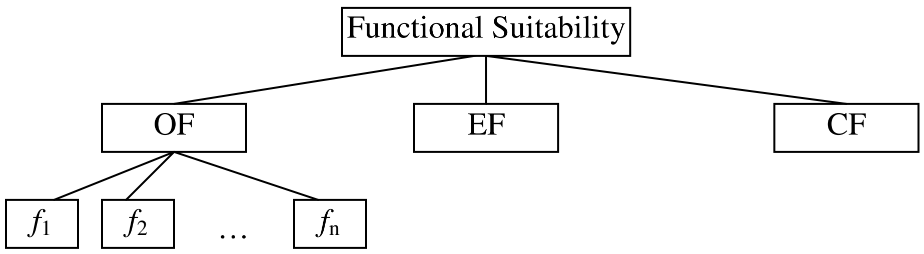

Figure 4-4 Functional Suitability Evaluation Hierarchy

Let us discuss this hierarchy (figure 4-4) in more details from the bottom up. First, we pair-wise compare functions $f_1, \ldots, f_n$ in S_c according to the relative importance of the function. The priority vector of the pair wise matrix is computed and the vector values serve as the weight for the criteria. Under each function, we compare how well each OSS alternative implements it relative to other OSS alternatives. Secondly, as we compare the OSS alternatives under extensible function compatibility, the major concern is the amount of work to implement the missing functions. In the evaluation process, we estimate the workload for an OSS alternative based on the number and complexity of functions which needs to be implemented. We prioritize all functions in S_c according to their complexity. Assume an OSS alternative's EF = $\{f_i, \ldots, f_{i+k}\}$and $w(f_{i'})$ is the complexity weight of function $f_{i'}$ ($i \le i' \le i+k$). Then its extension workload can be estimated via the equation $w(f_i)+\ldots+w(f_{i+k})$. The workload estimation serves as an indicator when we compare OSS alternatives in terms of extensible function compatibility. In practice, if we cannot evaluate the implementation complexity of functional requirements, we can simply use the number of missing functions as the indicator. In the end, the evaluation of OSS alternatives as to contributive function compatibility bases on our judgment of how these functions could help meet the possible requirements in the future even if they are currently redundant. Similarly, if we happen not to have any preference, we can simply employ the number of the contributive functions as well.

In order to carry out the functional suitability evaluation, we need to determine S_c and S_k. The measurement of S_c fits in with a functional user requirements (FUR) extracting model presented in the standard ISO/IEC 19761, COSMIC Full Function Points (COSMIC-FFP) [73]. The model is called "pre-implementation FUR model"[73] in that the extraction is based on the software engineering artifacts existing before the software is being implemented. The artifacts are requirements definition artifacts, data analysis/modeling artifacts and artifacts from functional decomposition of requirements. As for measuring S_k, the functions provided by the OSS alternatives are initially derived from functionality related artifacts such as software documents, feature list and mission statement etc. Since OSS does not have the venders' guarantee and its functionality related artifacts may contain imprecise information, we need to design test cases to make sure how much each claimed function is fulfilled; thereby S_k can be determined in the end.

4.3 Open Source Software Source Code Evaluation

4.3.1 OSS Source Code Overview

One of the major difference between OSS and the proprietary software is the OSS's source code is open to public, which lends users more flexibility when they adopt OSS in their project. If users are not satisfied with the OSS, they can customize it to fit their own purposes. In addition, since OSS support is not as reliable as the support of proprietary software (we will discuss it in next section), in many cases users have to maintain the code in order to solve the problems on their own. How hard users can do this is determined by the complexity of the source code. Normally, the more complicated the source code is, the more customization or maintenance work would be involved. Therefore, it is important to evaluate the complexity of the OSS source code and taking it into consideration in our OSS selection process.

OSS source code has gained a lot of attention. Godfrey and Tu did a case study on Linux kernel to investigate the evolution of OSS [74]. Figure 4-5 [74] is one of their results showing the fast size growth from June 1994 to Dec 1999 for Linux kernel. In

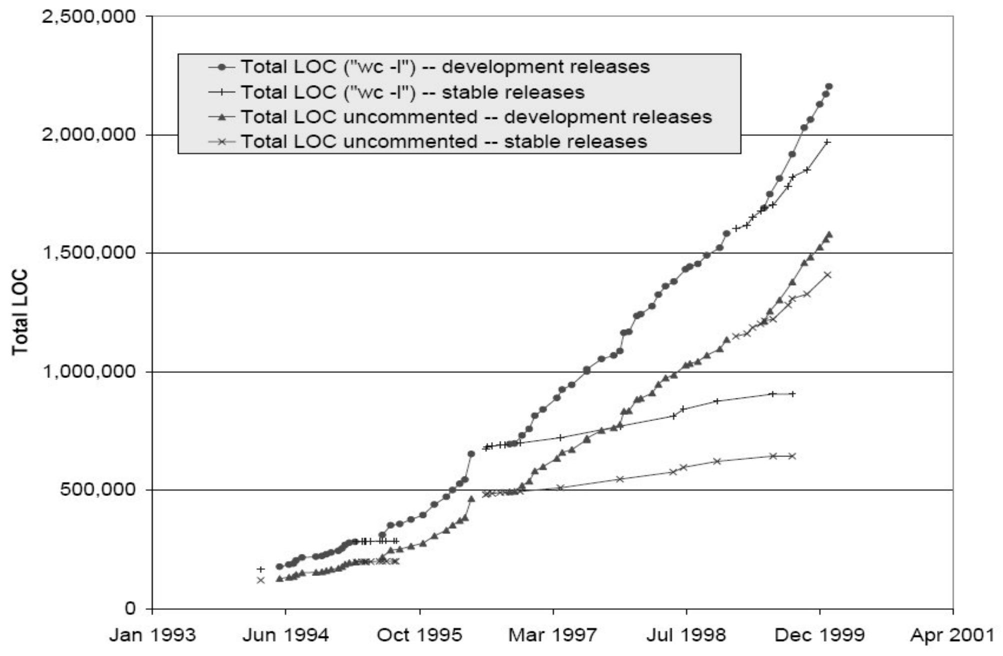

Figure 4-5 Linux Kernel Code Size Growth (excerpted from [74])

particular, the development releases even grew at a super-linear rate over time. Lehman's software evolution law II [71] states "as a program is evolved its complexity increases unless work is done to maintain or reduce it". The growth of Linux kernel indicates this law may also be applicable to OSS. Since code size does not reflect the program complexity, here we do two small case studies on OSS project Emule to further examine Lehman's program complexity evolution law on OSS. In the first experiment, we measured the McCabe's cyclomatic number (MVG) per module. In the second we used information flow complexity metric IF4 [75]. The details of these metrics will be presented later. Here we need to make clear that MVG measures the control complexity within a module and IF4 measures the structural complexity of the source code. From figure 4-6 and figure 4-7, we can see by and large the source code complexity keeps increasing as

Emule evolves. We noticed that on version 0.42, the values of both complexity metrics declined. Our guess is there was some maintenance work done on that version. Aside

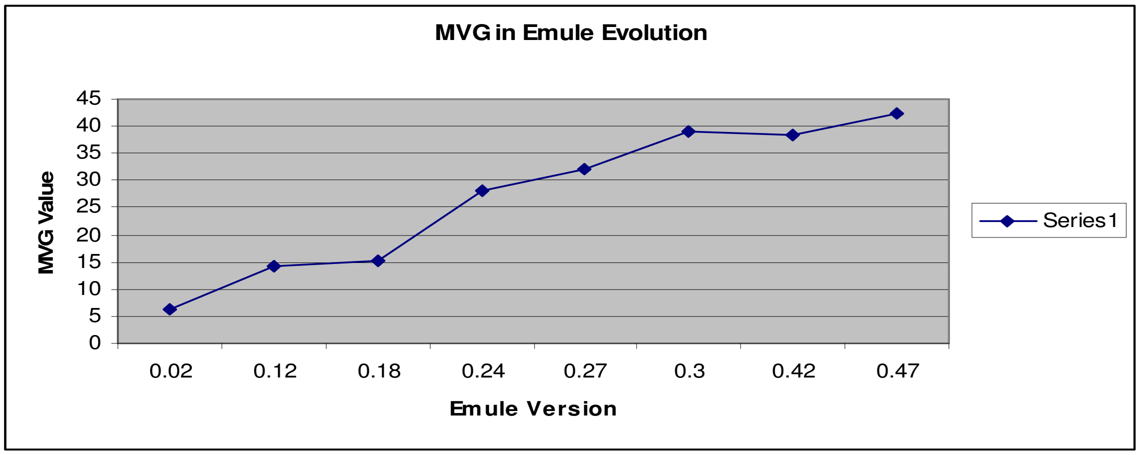

Figure 4-6 Control Complexity Growth of OSS Project Emule

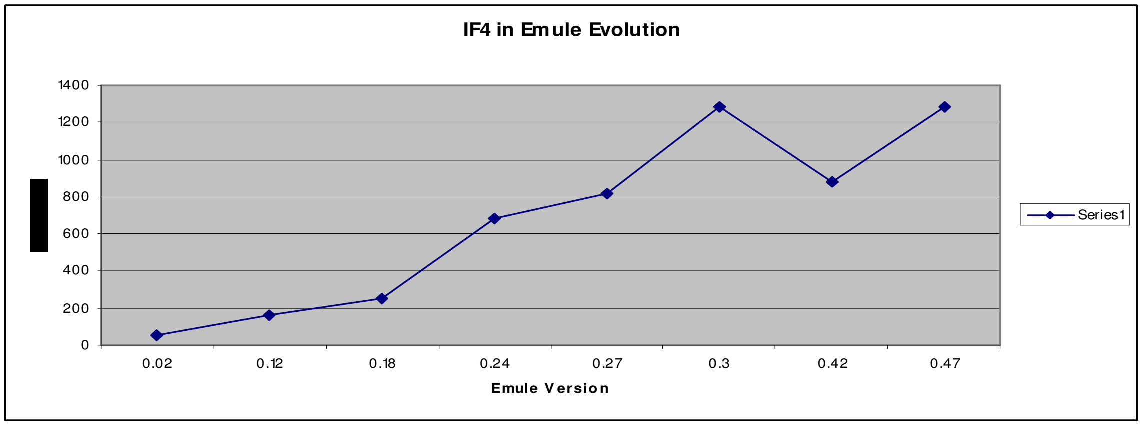

Figure 4-7 Structural Complexity Growth of OSS Project Emule

from the source code complexity evolution, people also care about OSS source code quality. There are mixed voices over this issue. Some researchers found out the structural code quality of the Linux applications provides results higher than that which someone countering open source might expect but lower than the quality implied by the standard

[76]. However, some study showed that the LINUX TCP stack (2.4.19) is the best in class with 0.1 defects per KLOC compared with commercial stacks with an average of 0.55 defects per KLOC [77].

4.3.2 OSS Source Code Evaluation Process

Analog to functionality evaluation, we also apply AHP approach to evaluate OSS source code. The sub-criteria under criterion source code are programming language, code size, code comment, code intra-module complexity and code inter-module complexity (figure 4-8). These sub-criteria set a framework for investigating the source code when a user wants to find out which OSS product is relatively easier for her/him to maintain, port or extend. Under each sub-criterion, the user could compare OSS alternatives based on the measuring results of corresponding software metric. In the rest of this section, we will discuss these source code sub-criteria in details.

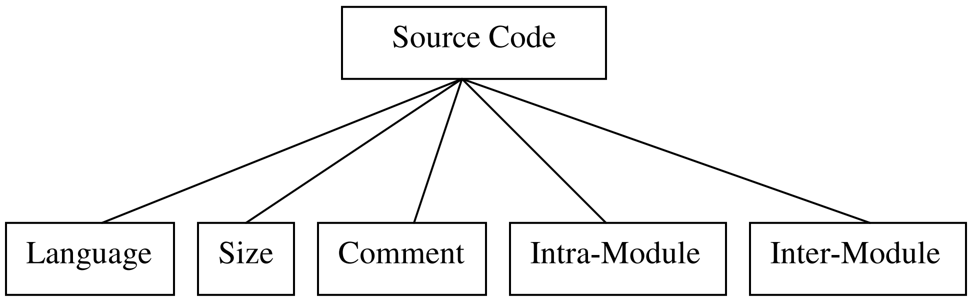

Figure 4-8 Source Code Evaluation Hierarchy

Programming language is no doubt an important factor which the user will take into account at the beginning of source code evaluation. Due to effects of the user's experiences and habit and programming language's own complexity, the user normally has a preference over programming languages. On the other hand, various OSS is written with different languages. The following figure gives an overview of the programming languages used in OSS development. The OSS projects in the figure are registered on Sourceforge.net before March 10, 2007 (119644 in total). We use the search and filtering tools provided by Sourceforge.net. Figure 4-9 discloses two facts: first, multiple programming languages have been used in OSS source code; second, Java, C++, C and PHP are the most popular languages in OSS development. To calculate the score for the programming language criterion, the user needs to compare the OSS alternatives pair wise according to his/her own preference and the specific requirements from the future development such as the interaction with other components or performance.

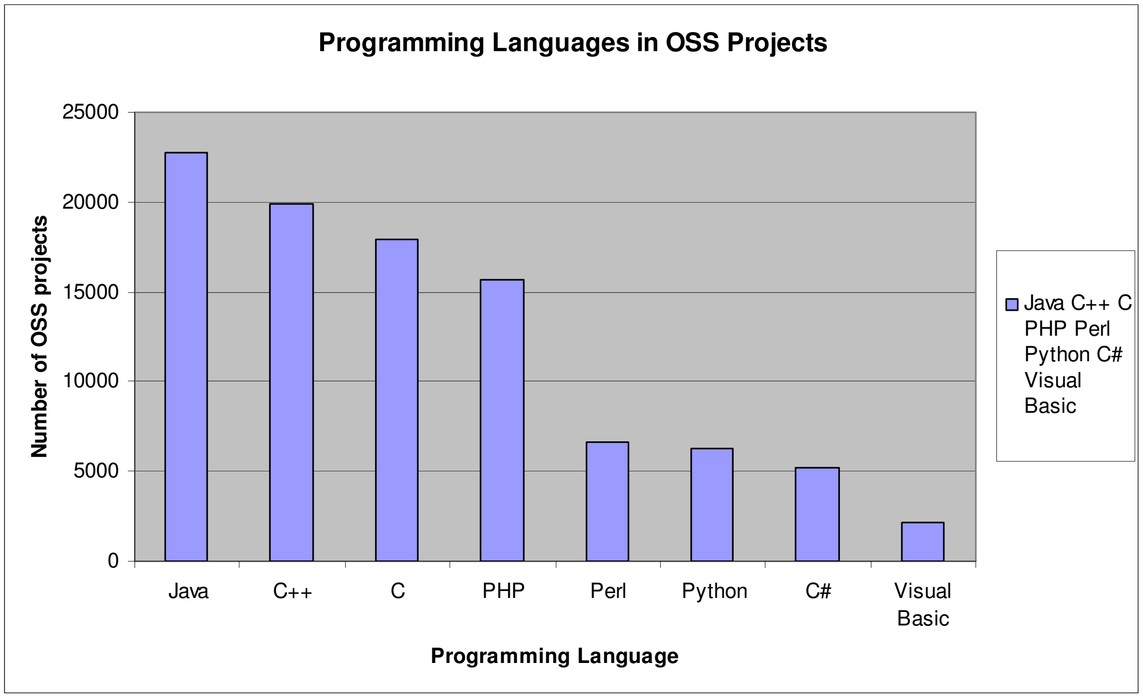

Figure 4-9 Programming Languages in OSS Projects

Aside from programming language criterion, we will employ software metric to help prioritize OSS alternatives. People have come up many metrics for measuring software source code such as source lines of code (LOC), Cohesion, Software Package Metrics, Coupling, Cyclomatic complexity etc. Here we will adopt a subset of these metrics into our OSS selection process to indicate the potential complexity of the source code. The determination of metrics is based on two concerns: the applicability and the availability. The applicability means the metrics should be applicable to both Object-Oriented program and procedural program. After all, OSS alternatives may consist of both programs. The availability means these metrics can be measured by some software tools. As such, we employ lines of code (LOC), Lines of comments (COM), McCabe's cyclomatic number (MVG))/Weighted methods per class (WMC) and Information flow [76] in our process. These metrics are applicable to both procedural program and object-oriented program (we treat WMC as an extension of cyclomatic number to Object-oriented program). Some metrics such as Coupling Between Objects (CBO) and the Lack of Cohesion of Methods (LCOM) are only applicable for object-oriented program. Our metrics may not be optimal but they can give users an approximate indication of the source code complexity. In addition, they are relatively easy to be measured. For Java and C++ code, these metrics could be measured automatically by an open source code analyzing tool, CCCC (http://sourceforge.net/projects/cccc) [78].

Code size sub-criterion is evaluated based on the metric Line of Code (LOC). Lines of Code is a traditional software metric. It refers to the number of lines in the source code excluding the blank and comments [79]. Normally people prefer the program with the smaller size if they could have multiple options which could provide the same functionality. The program with smaller LOC tends to have better understandability and maintainability. When we compare the OSS alternatives, we think the OSS with smaller source code is superior to the OSS with bigger code. However, minor gap between source code sizes would not bring much difference for code complexity.

Comments help users understand the source code, thereby lowering the effort on development and maintenance. Metric Lines of comments (COM) measures the understandability of the programs and thereby indicate indirectly the maintainability as well. To calculate this metrics, simply count the number of lines of comments in the program. The problem of COM is: large code may have more lines of comments but these comments are sparser than the smaller code with fewer comments. To overcome this problem, we use an extension of COM, Line of code per line of comment (L_C). It equals to LC divided by COM. The comparison of L_Cs between two OSS alternatives determines the value in the pair wise matrix under comment sub-criterion.

Intra-module complexity is evaluated according to metric McCabe's cyclomatic number or weighted methods per class .McCabe's cyclomatic number (MVG) [80] is a software metric measuring the number of independent paths through a program. The calculation is based on the control flow graph of the program. Assume we have program P and its control flow graph G. P's cyclomatic number $MVG(P) = E – N + R$. E is the edge number of G. N is the vertex number of G and R is the number of connected components in G. In practice, for structured program, the calculation can be simplified to counting the number of decision point in the program. McCabe's cyclomatic number could reveal the understandability and testability of the program. It mainly measures the complexity for procedural program and object oriented program at method level. For object-oriented program, Chidamber and Kemerer [81] proposed weighted methods per class (WMC). It can be either calculated by counting the number of member functions of a class or summing up the cylomatic number of the member functions. In our process we take the latter approach since it is more sophisticated. It is worth noting that module initially refers to a function or procedure but now its scope has been expanded to class as well. Therefore we replace the average WMC per class with the average MVG per module. This metric can be applied to both procedural programs and object-oriented programs.

Metric Information Flow [75] (IF4) is employed to evaluate inter-module complexity. It measures how well a program complies with structure design. The calculation is based on two sub-metrics: Fan-in and Fan-out. Fan-in refers to how many external modules cause the information flowing into the given module. Conversely, Fan-out counts the external modules which lead to the information flowing out of the given module. Usually Fan-in and Fan-out are computed based on the call graph. To estimate the overall complexity in the information flow, the inventors of Fan-in and Fan-out, Henry and Kafura came up with the composite metric IF4. There are several formulas to compute IF4 from Fan-in and Fan-out such as (procedure length)*(Fan-in*Fan-out)^2. In our process we use the formula (Fan-in*Fan-out)^2 adopted in [78]. High IF4 implies more complex inter-module control flow and tighter coupling. As we mentioned before, for object-oriented programs, there is also a coupling metric: Coupling Between Object (CBO). The metric counts the number of other classes which a class couples with. The coupling relation includes the object of this class calling or being called by the objects of other classes. From the definition, we can see this metric is also related to the information flow both into and out of the class. In order to have a metric which can be applied to both procedural program and object-oriented program, we employ IF4 per module to measure the inter-module coupling complexity.

4.4 Summary

In this section, we discuss how to evaluate OSS functional suitability and source code complexity. Our OSS functional suitability evaluation method is extended from Alejandra Cechich and Mario Piattini's COTS functional suitability evaluation method. It combines a screening process based on the priorities of multiple functional requirements. To evaluate source code complexity, we propose a set of software metrics: LOC, COM, MVG per module and IF4 per module. These metrics serve as indicators to expose the source code complexity from different respects.

5 OPEN SOURCE SOFTWARE SUPPORT AND POPULARITY ANALYSIS

5.1 Open Source Software Support Evaluation

5.1.1 OSS Support Overview

One of the difficulties OSS adopters will encounter is the support issue. Due to the importance of support in OSS adoption, we evaluate the support strength in our selection process. As with proprietary software, some OSS equips commercial support from some professional companies. For instance, some top IT companies have provided the OSS support service such as HP, IBM and RedHat. HP boasts its over 6,500 support professionals involving in implementing and supporting Linux and open source projects (http://opensource.hp.com/). IBM claims that "it provides support for both open source offerings and for associated IBM commercial software offerings" [82]. In recent years many small companies entered this market and start to provide support for pre-selected OSS packages, also known as certified bundle [83] or software stack [84]. One example is OpenLogic (www.openlogic.com) which supports more than 200 certified OSS projects. Although OSS users can acquire commercial support, there is still a significant amount of OSS without any commercial support service. People believe the lack of support hinders open source popularity [85]. Nevertheless, this does not necessarily mean the users have to completely rely on themselves to tackle the technical problems regarding software operation or further development. OSS provides some means to help users seek support such as document, wiki, blog, IRC, tracker system, mailing list, forum, direct developer emails etc. This support is contributed from the whole OSS community ranging over core developers, co-developers, active users and regular users. Normally the community members are geographically distributed and the information is shared via the Internet. If a user has a question, he/she could either look for the answer from the existing resources or directly ask for help via certain online communication medium such as mailing list or forum. Karim R. Lakhani and Eric von Hipper investigated the effectiveness of the forum support and underlying motivation of help providers in open source Apache system [86]. Their study shows that among their 4.5-month real-time sample "only about one-fourth" of the questions posted do not receive an answer" and a majority of information seekers feel the replies to their questions are helpful [86]. Since Apache system is a pretty mature system and has been developed for a long time, this result may not hold for all OSS but it points out how effective the forum support can reach. In terms of motivation, the authors did a survey on the support question providers. The potential motivations included expecting reciprocity, gaining reputation, helping the cause etc. The survey confirms that providing the help is essentially voluntary [86] but no motivation is the dominant reason for providing such support.

5.1.2 OSS Support Evaluation Criteria

To evaluate the strength of OSS support, we need to determine what criteria should be included into our process. In view of the OSS support characteristics, we set up two criteria on the second level of evaluation hierarchy (figure 5-1): field support and support resources. The field support includes commercial support and community support. The community support specifically refers to the direct responses from the community to the support requests. Usually this support occurs on the online information sharing platforms comprised of tracker system, mailing list, forum, IRC and direct developer emails. The reason of taking commercial support under consideration is because it is more reliable and responsive. Even though commercial support may add some costs, it could save a lot of efforts and time for OSS users. For mission critical or deadline restrictive software projects, users may prefer purchasing the service to ensure the smooth adoption. However, it is also likely that users do not need any commercial supports. They prefer OSS alternatives with stronger community support. The details regarding how to evaluate the OSS community support will be discussed later. Besides field support, OSS projects also provide various support-related resources such as documents, wiki, blog etc. OSS documents are an important support resource. Some projects even mark their documents with the support tag. One example is OSS project Emule. It explicitly puts a document repository on its “Help&Support” webpage. OSS documents include user manual, FAQ, reference, tutorials, developer guide/cookbook and publications. Usually when users want to learn OSS in more details, they will start to read the documents first. Aside from the documents, there are some other support-related resources such as screenshot, feature list, news, related links, blog, mailing list or forum archives etc. They may not be as closely related with support as the documents. But users can still find useful information out of these resources. The support resources are evaluated from three respects: relevance, capacity and understandability. Relevance refers to how closely the resources are related with the users’ support requirements. For example, if a user is concerned with the further development, then the OSS alternative with developer guide, cook book or any other development related materials will be preferred than those OSS alternatives which do not possess such resources. Capacity indicates how much information is contained in these resources. The more information support resources contain, the better chance the support request can be satisfied. Under understandability criterion, OSS alternatives are evaluated according to how difficult the information can be acquired and understood from the support resources. Or put it in another way, understandability measures the effectiveness of the supportive information sharing.

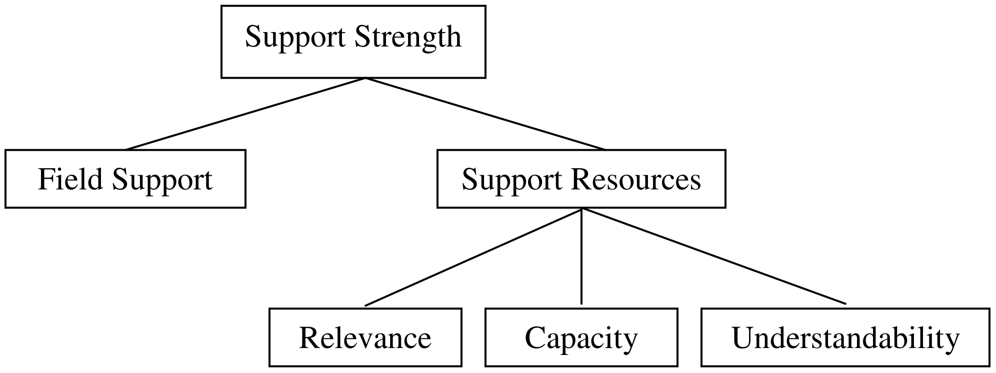

Figure 5-1 Support Strength Evaluation Hierarchy

5.1.3 OSS Community Support Evaluation

5.1.3.1 Community Support Strength Indicators

Since users request community support mainly via tracker system, mailing list, forum, IRC and direct developer email, it is necessary to evaluate the support strength from each of these communication channels. The following table lists a sampling result from the 96 OSS projects regarding support channels. The population is over 60,000 OSS projects on Sourceforge.net with files. The confidence level is 95% and confidence interval is 10%.

From table 5-1, we can see tracker system, mailing list and forum are fairly common among OSS projects but few OSS projects have adopted IRC for support purpose. For sake of simplicity, we do not consider IRC in our community support evaluation. Also, since usually the direct email does not have any historic records and the exchange emails between the help seeker and the developer are not open to the public, it is hard or even impossible to evaluate its support strength. Therefore in the evaluation we only take into consideration whether it exists or not. To be more accurate, all other things being equal, an OSS project with direct email support is deemed to have a better support than the OSS project without direct email support.

Table 5-1 OSS Community Support Sampling Results

Community Support	Mailing List	Forum	Tracker System	IRC	Direct Email
Percentage of Samples	24%	79%	89%	0	17.7%

To facilitate the measurement of the support strength, we designed several indicators for each support channel. The indicators aim to reflect the vigorousness of support activities on these channels and they should not be hard to measure. More important, we should be able to integrate these indicators to evaluate the strength of whole community support. The reason for the integration requirement is because OSS may not treat these support channels equally. Some projects only rely on one or two channels to provide support. For instance, project Emule does not have any mailing list or tracker system. Its community support activities are mainly carried out through the forum.

Therefore, when we compare its community support with other projects, we cannot do it in the fashion of channel versus channel. We have to treat all supports from the different channels as a whole. It is worth noting that due to the loosely OSS project management style, the evaluation will encounter the problems such as lack of historical information. It is hard to precisely determine the community support strength. Our strategy is making the best approximation with the available information.

5.1.3.2 Tracker System

Tracker system is a tool mostly provided by OSS host websites to track issues such as bugs, feature and support. We are interested in support request tracker, which is installed for users to seek help from the project developers. To better explain how the tracker system works, let us use the tracker system from Sourceforge.net as an example. After a user submits a support request, the request will be assigned with "open" status. The status will become "pending" if the project administrators need more input regarding the request. Once the request is answered or rejected, its status will be changed to "close". If the request is not answered in a predefined time period, its status will be set as "deleted" [87]. To evaluate the support from support request tracker, we came up with two indicators. The first is the mean response time. The response time on the tracker system can be defined as the time interval between request submission and close. This indicator directly shows the responsiveness of support. But this indicator is not easy to get measured without appropriate tools. In addition, it may miss some requests which are set as private and not displayed in the front end. The second indicator is the average number of closed support entries in a certain time period (in our process we use one week). This indicator indirectly addresses the major concern: "Has the support been active lately"? On

Sourceforge.net, the value of this indicator can be acquired from the tracker activity statistics which records the number of tracker items closed for the time period [88].

5.1.3.3 Mailing list

Mailing list is an important community support channel. In most cases, the OSS mailing list subscription is publicly open. Some OSS projects even provide a group of specialized mailing list. People have observed that successful OSS projects tend to make better use of mailing list than unsuccessful projects [89]. The observation was made by checking whether a sample project has no mailing list, one mailing list or multiple specialized mailing lists. One possible indicator could be the subscription number for the support-related mailing lists. Higher mailing list subscription number means more people joining the list and users could expect to get better support if the project mailing list owns a large member set. However, many OSS project mailing lists do not expose the subscriber number and this indicator cannot be integrated with the tracker system indicator either. Therefore, we employ an indicator compatible with the tracker system indicator: the number of replying messages in a certain time period such as one week. The advantage of this indicator is that it is not hard to measure if the project keeps the archive of historical mails. For those projects which do not come with such archive, users can subscribe the mailing list and spend one week on recording the number of replying messages in the mailing list. Its disadvantage is that, similar to the tracker system, the measurement may miss some replies since they are sent only to the original sender instead of the whole mailing list group.

5.1.3.4 Forum

Compared with mailing list and tracker system, forum is more informal. On some forums, users can submit their posts anonymously. Usually the forum keeps an archive of old posts. All messages and replies are posted on the forum, which lend itself to the analysis. OSS host website provides forum for the projects but some projects set up their own forums, which are in many cases more sophisticated. Similar to mailing list, many OSS projects also divide their forum into multiple sub-forums. Each sub-forum either focuses on a certain topic or mainly attracts certain specific groups like developers. For instance, OSS project Emule equips with its own forum (http://forum.emule-project.net/). The forum consists of nine sub-forums ranging over General Discussions, Support, Hardware Help, Bug Reports, Feature Requests and Translations etc. If users want to seek support, they could go to Support and Hardware Help sub-forums for help. The basic unit in a forum is post. The post could initiate a discussion or respond to another post. The discussion starting post is called as a topic. In order to measure the support strength from the forum, we choose the number of replying posts in a certain time period (here we use the one week as well). The large number of replying posts implies the forum participants are more willing to answer the questions. In addition, it is not hard to measure and can be integrated with mailing list and tracker system indicators.

5.1.3.5 Wrap-up

After we calculate the indicator values for each support channels, we use their sum as the overall indicator for community support strength. Since the community activity is fairly dynamic and the indicator is only a rough estimation for the community support strength, we have to take a conservative approach when we compare alternatives based on it. To be more exact, one alternative's community supports are preferable to the other only if there is a significant difference between the indicator values of these two alternatives. In addition, regarding the direct developer email, it affects the community support comparison only in the case where two alternatives have pretty close indicator values but one has the developer email support and one does not. We believe this strategy is reasonable since the developers are normally fewer than users.

5.2 Open Source Software Popularity Evaluation

5.2.1 OSS Popularity Criteria

When people have multiple choices of OSS alternatives, one easy way to find the best fit is simply adopting the most popular one. This strategy works in some cases since a popular OSS project tends to more mature. The OSS development process shows that the community participation is an important factor related with the success. Popular OSS more likely goes through more thorough development and testing by its large and active community. However, this strategy is a bit over simplistic. First of all, it is hard, or even impossible to exactly measure the OSS software popularity. The popularity of proprietary software can be measured by counting the number of its release or license, which can be acquired from software vender. But OSS usually is freely distributed. It is hard to collect such data. Second, this strategy ignores other requirements. Even if we know an OSS project is more popular than the other, it is still inappropriate to select the first one since the latter one may have some features we want but the first one does not have. Popularity discloses a part but not all of the necessary information regarding the software. Third, there is a possibility that among the OSS candidates we could not find an OSS which is significantly more popular than any other projects. In another word, these projects' popularities are closely matched. Then the strategy is not applicable any more. There are several ways to measure OSS project popularity. Freshmeat.net, one of the largest index website for OSS on Unix and cross-platform, defines a popularity measure as "((record hits + URL hits)*(subscriptions +1))^(1/2)" (http://freshmeat.net/faq/view/30/). However, this method is not general enough. Not all of OSS host or index websites provide such statistics data.

A simple popularity criterion is software use. The indicator is number of downloads. This criterion is similar to the release or license count used to measure proprietary software. High number of downloads implies that the OSS has been widely adopted and put into use. However, as [90][91] point it out, the potential problem of this measure is the ignorance of other software distribution channels. In reality, OSS may be distributed via other media, for instance, the off-line hardcopy. Particularly, an important channel is Linux distributions such as RedHat, SuSE and Debian [90]. Selected OSS packages and tools have been included into the distributions as a part of the whole solution such that users could install them without looking for the downloading source. Nevertheless, even though the number of downloads is not a reliable measure for OSS project use, it is still useful. The popularity of the OSS with 1 million downloads is very likely more popular than the OSS with only 1 hundred downloads. After all, on-line downloading is one of the major OSS distribution means. More importantly, most OSS projects provide the download statistics directly. To solve the problem, we can check whether OSS has been included into certain Linux distributions. There are various websites where we can look up that information. For instance, website http://packages.debian.org/stable lists the package issued with Debian Linux and www.rpmfind.net keeps a repository of RPMs and the distribution information. However, these websites do not provide the downloading statistics. Given the large volume of Linux distributions, we can simply deem the software included into the distributions as popular software but this workaround does not completely solve the problem. In our process, we combine software use with other criteria. More precisely, we employ two other criteria to evaluate popularity: development participation and web popularity (figure 5-2). There is also unreliability involved into the OSS alternative evaluation under these two criteria. But our idea is combining these multiple independent criteria together to mitigate the overall unreliability.

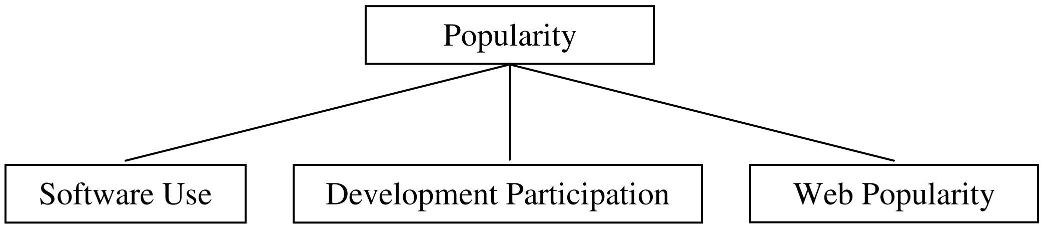

Figure 5-2 Popularity Evaluation Hierarchy

5.2.2 Development Participation

The OSS developers voluntarily join the project development motivated by their interest or intention to gain experiences and reputation from the project growth. The developer here refers to both core developers and co-developers. OSS with a larger developer group implies it attracts more attention from the community and share more common interests.

Conversely, more developers can speed up the development process, improve the support and eventually enhance the OSS popularity. Hence, we believe the number of developers could indicate the OSS popularity. Practically, there are two ways of counting developers. Usually OSS projects will list the members formally joining the development. More broadly, the active participants can also be counted via examining the mailing and other fora [90]. In our process, we follow the first method because it is much simpler. It is noteworthy that this indicator may not be consistent with the number of downloads. Otherwise, it is redundant and can be replaced with the latter. There exist cases where an OSS project with high download number has fewer developers than an OSS project with much lower download number. For example, we have two file sharing OSS projects: ABC [Yet Another Bittorrent Client] (http://sourceforge.net/projects/pingpong-abc) and Suicide Gnutella Client (http://sourceforge.net/projects/suicide). ABC has 4 developers and Suicide Gnutella Client has 5 developers. But project ABC has been downloaded for more than 10 million times and there are only more than 1 thousand downloads for the latter.

5.2.3 Web Popularity

OSS project popularity can be estimated by checking how many times it is referred on the Internet. An OSS project with many web references is more likely to be a popular project. The web references can be counted by using web search engine such as Google, Yahoo or Windows Live. [92] introduces three methods regarding estimating the web references. The simplest method is counting the number of pages containing all of the words in project's name [92]. This method is simple and gives a rough estimation of the web references but it also counts the web pages which refer to other meanings of words building up the name of the project [92]. For instance, an OSS's name is Oscar (http://sourceforge.net/projects/oscar). Certainly most of web pages returned by search engines with this name keyword will not be related with the project. A strict alternative is license-reference counting [92]. The method only counts the web pages which contain the whole phrase defined in the software license terms. Compared with the first method, it will include far less redundant web pages, which makes it accurate. The disadvantage of this method is it is not applicable to OSS projects whose licenses do not define a name for reference. The third method can be seen as a compromise of the last two. It is stricter than the first method but less conservative and more applicable than the second one [92]. The method counts the back links to project's homepage on the Internet [92]. Among the back links there are some internal links originated from the same domain as the project's homepage. Whether these internal links are excluded from the total count of the back links subdivides the method into two variants. In our process, we choose the variant which discounts the internal links as the web popularity indicator. The indicator's value can be measured by using search engines such as Yahoo site explorer or Alltheweb.com. In addition, this indicator is not consistent with the two previous indicators. The counterexample is project BBman (http://bbman.sourceforge.net/) and PCMan X (http://pcmanx.sourceforge.net/). Both projects are telnet clients, have the same license and oriented to the same platforms. The back link count of BBman is around 30 (measured from Alltheweb.com) and the count of PCMan X is around 6. In contrast, BBman has only 1 developer and PCMan X has 11 developers. BBman has been downloaded by 23,194 times on Sourceforge.net and the number of downloads for PCMan X is 58131 (All the data is measured on May 6, 2007).

6 A CASE STUDY: USING OSS SELECTION PROCESS TO SELECT OPEN SOURCE UML DESIGN TOOL

6.1 Motivation

UML (Unified Modeling Language) is a graphic-notation based object modeling language. Currently UML is widely applied into software development process with the dominance of object technology. According to Computer World Survey in 2005 [93], 33% of developers reported that UML was in use at their organization. In reality, it has become a standard visual modeling tool for project manager, architect and developers in software industry. With the aid of UML, it is much easier for software development participants to share their ideas, locate the problems and refine the design and implementation. Aside from the software development, it can also be used into business modeling [94], database design [95] and software testing [96]. To facilitate the UML modeling process, people have invented many software tools. The most notable one is IBM Rational Rose. Rose is a proprietary UML visual modeling software tool. It supports the UML standards and can generate code for various programming languages such as C++, Java and Visual Basic etc. Other proprietary UML tools include Visual Paradigm, Borland Together and Microsoft Visio. Besides the proprietary tools, there are also various UML design open source tool. For example: ArgoUML, BOUML and so on. As mentioned before, compared with proprietary UML tools, these tools are free and customizable. Since each of these tools has its own merits and shortcomings, it is hard to tell which one is the best in general. Additionally, the selection should also combine with the specific requirements. In this case study, we would like to utilize our OSS selection process to determine the best open source UML tool among a few alternatives for our usage. The study will demonstrate how the selection process proceeds. The scenario and requirements considered in this case study are summarized from the author's internship in a private software company for Gas&Oil applications. In the rest of the report, we will call this internship worksite as company S.

6.2 Unified Modeling Language: A Brief Introduction

Let us introduce more details of UML. There are many versions of UML: from version 1.1, the first mature version released in 1997, to version 2.1.1, the newest version released in February 2007. Particularly version 1.4 has been accepted by ISO as ISO/IEC 19501 and became an international standard. In our report, we discuss UML version 2.0 released in August, 2005 since it is the latest version many UML tools support. UML is not a method but it complies with and lends itself to most Object-oriented Analysis & Design (OOA&D) methodologies. Before starting UML-based development project, it is recommended to select a methodology [97]. There are various methodologies available such as Rational Unified Process, and Agile Programming etc. UML 2.0 defines thirteen diagrams, which can be categorized into two large types: structure diagrams and behavior diagrams [98]. Structure diagrams include the class diagram, object diagram, component diagram, composite structure diagram, package diagram and deployment diagram. Behaviors diagrams include activities diagram, interaction diagram, state machines diagram and use case diagram. In the following paragraphs, we will give more details of these diagrams based on the UML version 2.0 Superstructure specifications [98] and other introductory materials [97][99][100].

In UML structural diagrams, class diagram describes the classes and their relationships in a software system. Object diagram is a snapshot of the interaction among class objects during the execution. Component diagram illustrates the architectural layout of the software. Composite structure diagram shows the internal composition and interaction within a class at runtime. Deployment diagram models the hardware infrastructure underlying the software and how the software connects with it. Package diagram depicts the logic units and their relationship in the software. The graphic nodes of structure diagrams are class, interface, instance specification and package. The edges (or paths) depict the relationships among these nodes which include association, aggregation, composition, dependency, generalization, interface realization, realization, usage etc. Association is a general relationship depicting the connections between the instances of the associated types. Aggregation is a refined association which only allows the part-whole relationship. Composition requires one associated type instance containing an instance of the other type. Generalization means one associated type is a general form of the other. If the change in one type will lead to the change in the other one, then their relationship is defined as dependency. Usage is a dependency with one more constraint: the operation of one type depends on whether the other type is present or not. Realization refers to the case where one associated type contains the implementation of the other. If the type without the implementation is an interface, then the relationship becomes interface realization.

UML behavior diagrams focus on the operation and communication related with the software. Among the behavior diagrams, activity diagram models the workflow in the business process. It emphasizes sequence and conditions for coordinating low-level behaviors [98]. State machine model describes the state transition during the execution, which can help users better understand the runtime behavior of the software. Use case diagram has been widely used in business or requirement analysis. This diagram makes a description of how users interact with the system. In particular, in behavior diagrams there is a subset called interaction diagrams. Interaction diagrams place extra emphasis on the communication part in software behavior. It includes communication diagram, interaction overview flow diagram, sequence diagram and timing diagram. Communication diagram evolves from the collaboration diagram from previous UML versions. It shows the communication in the form of message sequence among the objects or parts. Interaction overview flow diagram is a newly introduced diagram in UML 2.0 and can be seen as a specialized activity diagram. The graphic nodes in this diagram are frames. Users can either inline other interaction diagrams or specify what activity or operation happens in these frames. Sequence diagram models the interaction sequence at runtime. The diagram is a good tool to mirror the complex runtime collaboration involving several participants. Time diagram is also a new diagram introduced by UML 2.0. It explores the time constraints related with the operation or activity.

6.3 Open Source UML Tool Identification and Screen

As we start our open source UML tool selection process, we identify the multiple potential UML tools. The identification criterion is that the tool should support UML modeling. Thereby we have ten open source UML tools: StarUML (http://staruml.sourceforge.net/en/), ArgoUML (http://argouml.tigris.org/), BOUML (http://bouml.free.fr/), UMLet (http://sourceforge.net/projects/umlet/), Gaphor(http://gaphor.devjavu.com/), Dia(http://live.gnome.org/Dia), Violet(http://alexdp.free.fr/violetumleditor/page.php), Astade (http://astade.tigris.org/), UML Pad (http://web.tiscali.it/ggbhome/umlpad/umlpad.htm) and Umbrello UML Modeller (http://uml.sourceforge.net/index.php).

In the screen step, we reject OSS alternatives according to our refined requirements. We aim at finding an OSS UML tool which can run on windows platform and supports multiple UML diagrams. We require the tool should be implemented with C++ or Java since we are familiar with these two languages. In addition, the tools should not depend on other modules which we have no prior experiences to work with. According to the requirements, we can simply check these tools' supporting platforms, programming languages and dependent modules. Thereby StarUML, Gaphor, Dia, Astade, UML Pad and UMbrello UML Modelers are screened out in this process. Umbrello UML Modeller is not selected since it mainly works on Linux platform. StarUML does not pass the screen phase because StarUML is implemented with Borland Delphi. Dia, Gaphor, Astade and UML Pad depend on other modules which make them inappropriate for further evaluation. In more details, Dia and Gaphor depend on GTK+ toolkit. UML Pad and Astade are based on wxWidgets. Besides, the selected tool is mainly for internal use, which does not exceed the license restrictions of these tools. Therefore the license requirement does not have any alternatives rejected in this step.

Now there are four competitive alternatives left: ArgUML, BOUML, UMLet and Violet. Before we evaluate these tools, we give a brief introduction about them. ArgoUML is an OSS project host on Tigris.Org. The first version of ArgoUML was distributed by University of California at Irvine in 1998. In 1999 it became an OSS project. It is written with Java and therefore platform independent. In the case study, we will use its newest version: version 0.24. BOUML is claimed as a tool supporting UML 2. It runs on multiple platforms including Windows. In our case study, we use its version 2.23.1. UMLet is a UML tool developed with Java. It runs either stand-alone or as an Eclipse plug-in. We evaluate its version 7.1. Violet is an UML editor developed with Java. The authors claim it is easy to learn and use. In the evaluation, we use its version 0.20. In addition, to facilitate our evaluation, we use a free online AHP tool, Web-HIPRE (http://www.hipre.hut.fi/). This tool is developed by Systems Analysis Laboratory of Helsinki University of Technology. It is a Java applet for multiple criteria decision analysis [102]. The tool uses the consistency measure (CM) instead of consistency ratio to estimate the consistency. The use of the CM is similar to the CR [102]. We still try to control the CM lower than 0.1 to make sure the inconsistency is within the acceptable limit in our evaluation.

6.4 Open Source UML Tools Evaluation

6.4.1 Functional Requirement Analysis

As mentioned, our ultimate goal is to find an open source UML modeling tool supporting our development process. The ideal tool would meet most of our requirements and leave the extension work to the minimum. Despite the fact that the development process of company S is not strictly defined and still under renovation, the process can approximately be attributed to an iterative and incremental development process. In the development there are two major threads: product and project. The project is a part of the product. It represents a major task within the product development such as implementing a feature or a component. The development process involves several roles: project manager, department manager, developer, tester and user. Project manager is usually an expert on the application area, i.e. geography and geology. He takes charge of interaction with users, writing project specification, planning, project organizing and educating engineers with the domain knowledge. Department manager coordinates the development of whole product. He oversees all projects and occasionally participates into the critical development incidents, for instance, specification release or modification, difficult implementation or design issue discussion etc. Developer and tester assume the responsibilities of specification implementation and verification. In particular, the developer should also design the software. Normally the design is object-oriented. If any developer or tester encounters a technical or scheduling problem which they cannot handle by themselves, they will discuss with the project manager or the department manager depending on the problem scope to find out a solution. Sometimes the solution may involve updating the specification or schedule. During the whole development, users can give their comments or expectation on the software.

The development process requires modeling from many aspects. The following use case diagram (figure 6-1) illustrates the use cases involving the modeling. Specifically, in the process, project manager needs to model the workflow and requirements for the project. He also models the project architecture based on the business logic analysis. Department manager has the similar modeling requirement but his focus is software product. Aside from the requirement and architecture, he needs to describe the execution environment for the product. Comparatively, developer does more modeling work during the implementation. He models the software structure, runtime operation and program state transition. The software structure is mainly defined on the class level. To test the program, tester also needs to gain an understanding of the program status and how the modules interact with each other.

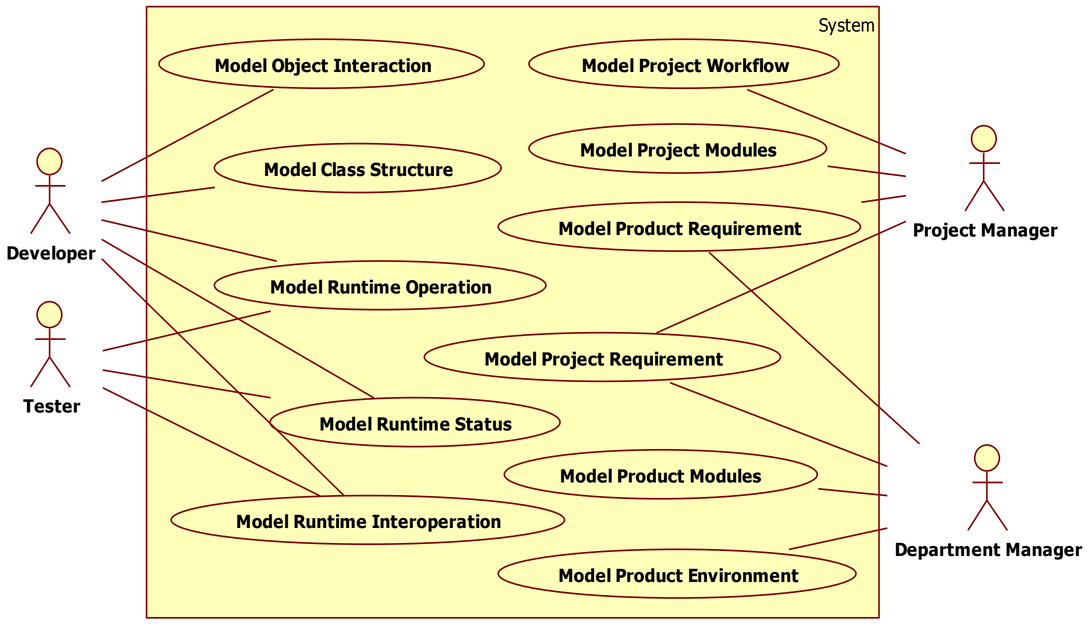

Figure 6-1 Use Case of the Modeling Tool

According to the modeling requirements in the development process, we can determine our requirement from UML. In UML, use case diagram is a good tool for department manager, project manager to model requirements. Similarly, activity diagram and communication diagram can be used by department manager and project manager to describe the business workflow. To model high-level architecture, project manager and department manager can use component diagram. In particular, department manager can describe the product actual running time environment or configuration via deployment diagram. Developer modeling activities need support from class diagram, communication diagram, sequence diagram, object diagram and state machine diagram. These diagrams help developer model program's structure and dynamic behavior including runtime operation, status, and interoperation among modules. For tester, state machine diagram, sequence diagram and communication diagram can assist analyzing the interaction and status of the program. In summary, there are nine UML diagrams needed in the development process: use case diagram, activity diagram, communication diagram, component diagram, class diagram, sequence diagram, object diagram, state machine diagram and deployment diagram.

Besides UML diagrams, the development process has some extra requirements. First of all, we need the support for XML Metadata Interchange (XMI). XMI is also a standard of OMG [101]. It is an XML-based data exchange format for sharing objects using XML. One of its major applications is supporting the interchange of UML models. We hope this feature could help us share the UML models on various UML modeling tools and ensure the reuse if we change our UML modeling tool in the future. Second, the feature of diagram exporting is also useful for us. In many cases, we need to add the diagrams into related documents or presentation slides. For instance, the developer can put a sequence diagram into the document, based on which he can discuss with tester or manager the scenario of a program bug or logic problem. Third, we prefer the tool which provides the better manipulation of diagram such as zooming in/out, undo/redo, customizing font size and background color etc.

6.4.2 Functional Suitability

To measure functional suitability, we need to examine OSS out-of-box function compatibility, extensible function compatibility and contributive function compatibility. Most importantly, we need to instantiate the hierarchy regarding functional suitability first. The following tables (table 6-1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16) show the comparisons with respect to the criteria at each level of functional suitability evaluation hierarchy. Before each table is presented, we will give a brief introduction about how comparison is made. The consistency measure value is also listed.

Table 6-1 is the result of pair wise comparison between each functional suitability criterion. In our case study, we prefer the tools which can meet more of our requirements. The contributory functions are a plus but not as important as other criteria. (CM=0.085)

Table 6-1 Functional Suitability Pair Comparison

Functional Suitability	**OF**	**EF**	**CF**	**Priority**
OF	1	5	6	0.726
EF	1/5	1	2	0.172
CF	1/6	½	1	0.102

ArgoUML, Violet, BOUML and UMLet provide the basic elements of use case diagram such as actor, use case, extend, include, generalization etc. Violet does not have a subject frame. UMLet is more compatible with UML 2.0 specification in that it provides more elements defined in UML 2.0 such as extension point. Also it provides some predefined templates for users' convenience. (CM=0.094)

Table 6-2 Use Case Diagram Pair Comparison

Use Case	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/3	1/3	1/5	0.078
ArgoUML	3	1	1	1/3	0.200
BOUML	3	1	1	1/3	0.200
UMLet	5	3	3	1	0.522

ArgoUML and BOUML have relatively better support for activity diagram. They are furnished with almost all necessary diagram nodes including action, final, initial, decision, joint etc. BOUML provides more decision and joint nodes but it does not have signal nodes. UMLet and Violet provide basic graphic nodes for activity diagram. (CM=0.067)

Table 6-3 Activity Diagram Pair Comparison

Activity	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/3	1/3	1	0.124
ArgoUML	3	1	1.5	3	0.414
BOUML	3	0.67	1	3	0.338
UMLet	1	1/3	1/3	1	0.124

Violet and UMLet do not explicitly support communication diagram. But we can still use UMLet to draw this diagram by utilizing the elements originally set for other diagrams. ArgoUML has the best support for this diagram. BOUML supports this diagram but it does not have the message node. (CM=0.078)

Table 6-4 Communication Diagram Pair Comparison

Communication	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/7	¼	½	0.068
ArgoUML	7	1	3	4	0.564
BOUML	4	1/3	1	2	0.238
UMLet	2	1/4	½	1	0.130

ArgoUML and Violet do not support the component diagram. UMLet does not directly support it either but it provides some nodes with which users can draw diagrams close to it. BOUML supports this diagram (CM=0.085).

Table 6-5 Component Diagram Pair Comparison

Component	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1	1/5	1/3	0.095
ArgoUML	1	1	1/5	1/3	0.095
BOUML	5	5	1	3	0.560
UMLet	3	3	1/3	1	0.249

In terms of class diagram, all of four tools support the diagram very well. Relatively UMLet is a bit weak since it is not as easy to use as other tools (CM=0).

Table 6-6 Class Diagram Pair Comparison

Class	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1	1	2	0.286
ArgoUML	1	1	1	2	0.286
BOUML	1	1	1	2	0.286
UMLet	1/2	1/2	1/2	1	0.143

The sequence diagram is well supported by the four tools as well. Particularly, UMLet and BOUML provide more nodes to describe the communication such as asynchronized message. The ranking of the tools on this function is fairly close. (CM=0).

Table 6-7 Sequence Diagram Pair Comparison

Sequence	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1	1/2	1/2	0.167
ArgoUML	1	1	1/2	1/2	0.167
BOUML	2	2	1	1	0.333
UMLet	2	2	1	1	0.333

ArgoUML does not support object diagram. BOUML supports it but the function is fairly hard to use. Comparatively, drawing object diagrams on Violet or UMLet is simple and effective.

Table 6-8 Object Diagram Pair Comparison

Object	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	3	2	1	0.351
ArgoUML	1/3	1	1/2	1/3	0.109
BOUML	1/2	2	1	1/2	0.189
UMLet	1	3	2	1	0.351

All of four tools support state machine diagram but Violet and UMLet only provide the basic nodes. In contrast, ArgoUML and BOUML have more nodes which give users more flexibility and convenience (CM=0.069).

Table 6-9 State Machine Diagram Pair Comparison

State Machine	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/4	1/3	1	0.116
ArgoUML	4	1	1.5	3	0.442
BOUML	3	2/3	1	2	0.304
UMLet	1	1/3	1/2	1	0.138

ArgoUML, BOUML and UMLet support deployment diagram fairly well. Violet does not have this feature (CM=0.06).

Table 6-10 Deployment Diagram Pair Comparison

Deployment	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/5	1/5	1/5	0.062
ArgoUML	5	1	1	1.5	0.342
BOUML	5	1	1	1.5	0.342
UMLet	5	2/3	2/3	1	0.254

With BOUML, users can import and export XMI up to version 2.1. ArgoUML has this function as well but this support is only compliant with XMI 1.2 standard. UMLet and Violet do not have this function (CM=0.093).

Table 6-11 XMI Pair Comparison

XMI	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/5	1/5	1	0.081
ArgoUML	5	1	1/2	5	0.346
BOUML	5	2	1	5	0.492
UMLet	1	1/5	1/5	1	0.081

All of these four tools support export the diagram into certain types of image. Modelers can use ArgoUML to generate png, gif, svg, ps and eps image files. BOUML supports png and svg file export. UMLet can generate jpg, svg, pdf and eps formats. Violet supports jpg and bmp. Particularly BOUML and Violet supports the direct diagram export to the clipboard. (CM=0.05).

Table 6-12 Export Pair Comparison

Export	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	2	1/2	2	0.263
ArgoUML	1/2	1	1/3	1	0.141
BOUML	2	3	1	3	0.455
UMLet	1/2	1	1/3	1	0.141

ArgoUML and BOUML provide plentiful operations for manipulating the diagrams. In ArgoUML, users can do operations such as search, zoom in/out, pan etc. BOUML's diagram operations include copy/paste, display style, font, zoom in/out, and search. UMLet and Violet have relatively fewer operations. With UMLet, users can undo/redo, pan or change the node color. Violet supports zoom in/out and undo/redo features. (CM=0.059)

Table 6-13 Manipulation Pair Comparison

Manipulation	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/4	1/5	1.5	0.099
ArgoUML	4	1	1	5	0.400
BOUML	5	1	1	5	0.424
UMLet	2/3	1/5	1/5	1	0.077

The following matrix (table 6-14) is the pair wise comparison regarding out-of-box function compatibility criterion. We emphasize on use case, class, sequence, state machine diagrams since we believe these diagrams are more important to the development. Diagram manipulation is also important because we want to have more flexibility to draw or modify the diagrams. Activity diagram, component diagram, communication diagram and export support are relatively less important. The least important functions are object diagram, deployment diagram and XMI support. (CM=0.084) (Note: to save the space we use the alphabets on the header line. A represents use case, B for activity, C for communication and so on and so forth).

Table 6-14 Out-of-Box Function Compatibility Pair Comparison

OF	A	B	C	D	E	F	G	H	I	J	K	L	Priority
Use Case	1	5	5	5	1	1	7	1	7	5	1	7	0.158
Activity	1/5	1	1	1	1/5	1/5	3	1/5	3	1	1/5	3	0.039
Communication	1/5	1	1	1	1/5	1/5	3	1/5	3	1	1/5	3	0.039
Component	1/5	1	1	1	1/5	1/5	3	1/5	3	1	1/5	3	0.039
Class	1	5	5	5	1	1	7	1	7	5	1	7	0.158
Sequence	1	5	5	5	1	1	7	1	7	5	1	7	0.158
Object	1/7	1/3	1/3	1/3	1/7	1/7	1	1/7	1	1/3	1/7	1	0.018
State Machine	1	5	5	5	1	1	7	1	7	5	1	7	0.158
XMI	1/7	1/3	1/3	1/3	1/7	1/7	1	1/7	1	1/3	1/7	1	0.018
Export	1/5	1	1	1	1/5	1/5	3	1/5	3	1	1/5	3	0.039
Manipulation	1	5	5	5	1	1	7	1	7	5	1	7	0.158
Deploy	1/7	1/3	1/3	1/3	1/7	1/7	1	1/7	1	1/3	1/7	1	0.018

In terms of extensible functional compatibility, in order to meet the requirements, we have to do more extension work on Violet than other UML tools. Violet lacks the support of component diagram, deployment diagram, XMI. The extension work on UMLet would also be significant. We need to add up XMI support and several diagram operations. The extension work on ArgoUML is adding object diagram support.

Table 6-15 Extendable Function Compatibility Pair Comparison

EF	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1/5	1/5	1	0.083
ArgoUML	5	1	1.5	5	0.460
BOUML	5	2/3	1	5	0.375
UMLet	1	1/5	1/5	1	0.083

As for contributory functions, ArgoUML and BOUML have some functions which are highly preferable. For example, both tools support code generation. ArgoUML even has the critics feature to help the modeler make high quality diagram. BOUML can import model from Rational Rose and generate html files. (CM=0.086)

Table 6-16 Contributory Function Compatibility Pair Comparison

CF	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	5	3	9	0.585
ArgoUML	1/5	1	1/3	2	0.103
BOUML	1/3	3	1	5	0.256
UMLet	1/9	1/2	1/5	1	0.055

In the end, the OSS alternative priority vector regarding functional suitability is <Violet:0.127, ArgoUML: 0.324, BOUML: 0.356, UMLet: 0.193>.

6.4.3 Code Complexity

As discussed in the previous section, we evaluate the code complexity from programming language, code size, comment percentage, intra-module complexity and inter-module complexity. We use OSS tool CCCC [78] to calculate the values of the corresponding metrics. Table 6-17 shows the data we collected by using this tool.

Table 6-17 Collected Data for Source Code

	Violet	ArgoUML	BOUML	UMLet
Language	Java	Java	C++	Java
LOC	17508	145200	101253	4750
L_C	2.024	1.33	5.237	3.674
MVG per module	3.34	10.666	33.804	2.878
IF4 per module	1559.72	6237.55	18182.403	630.029

The following five tables (table 6-18, 19, 20, 21, 22) show the pair wise comparison results under source code sub-criteria. Regarding the programming language, we prefer C++ over Java since we have more experience on C++ programming. Therefore, BOUML is ranked higher than other UML tools. (CM=0). As UML tools are compared in view of their sizes, the tool with smaller size is more preferable. The reason is the smaller code is easier to understand and maintain. (CM=0.088). Under comment criterion, the tools with low L_C are deemed better than the ones with high L_C (CM=0.072). Smaller MVG per module means the code is less complicated within its modules on average (CM=0.05). Likewise, small IF4 per module indicates the loose coupling among the source code modules (CM=0.099).

Table 6-18 Programming Language Pair Comparison

Programming Language	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1	1/3	1	0.167
ArgoUML	1	1	1/3	1	0.167
BOUML	3	3	1	3	0.5
UMLet	1	1	1/3	1	0.167

Table 6-19 Code Size Pair Comparison

Size	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	4	3	1/3	0.243
ArgoUML	1/4	1	1/2	1/8	0.061
BOUML	1/3	2	1	1/6	0.099
UMLet	3	8	6	1	0.596

Table 6-20 Comment Pair Comparison

Comment	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	2/3	3	2	0.318
ArgoUML	1.5	1	3	2	0.390
BOUML	1/3	1/3	1	2/3	0.117
UMLet	1/2	1/2	1/5	1	0.175

Table 6-21 Intra-Module Complexity Pair Comparison

Intra-Module	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	2	3	1	0.351
ArgoUML	1/2	1	2	1/2	0.189
BOUML	1/3	1/2	1	1/3	0.109
UMLet	1	2	3	1	0.351

Table 6-22 Inter-Module Complexity Pair Comparison

Inter-Module	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	3	4	2/3	0.334
ArgoUML	1/3	1	2	1/4	0.128
BOUML	1/4	½	1	2/9	0.082
UMLet	1.5	4	4.5	1	0.455

We compare the relative importance of sub-criteria under source code (table 6-23). At the first place, we favor the source code which is implemented with our acceptable languages and contain enough comments. Among size, intra-module complexity and inter-module complexity, we think size and intra-module complexity is a bit more important than inter-module complexity.

Table 6-23 Source Code Pair Comparison

Source Code	Language	Size	Comment	Intra-Module	Inter-Module	Priority
Language	1	2	1.5	3	4	0.355
Size	1/2	1	2/3	2	3	0.199
Comment	2/3	1.5	1	2.5	3	0.259
Intra-Module	1/3	½	2/5	1	1	0.109
Inter-Module	1/4	1/3	1/3	2/3	1	0.087

The overall priority vector regarding the source code is <Violet:0.254, ArgoUML:0.203, BOUML:0.246, UMLet:0.297>.

6.4.4 Support Strength

Let us examine the alternatives under relevance criterion first. We want to get an idea from the support resources regarding how to use the software, the software architecture and its user interface. Respectively, ArgoUML provides user manual, quick guide, user interface tour and developer cookbook. BOUML has the screen shots, online help file and tutorials. UMLet gives some sample diagrams and FAQ. Violet only provides a user interface tour and demo. Accordingly, the alternatives are compared regarding relevance in table 6-24 (CM=0.094):

Table 6-24 Relevance Pair Comparison

Relevance	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/6	1/3	1	0.085
ArgoUML	6	1	4	6	0.614
BOUML	3	¼	1	3	0.216
UMLet	1	1/6	1/3	1	0.085

In terms of capacity (table 6-25), ArgoUML's support resources are very informative. After going through its various support resources, we get a good understanding of this software. BOUML's support resources are relatively weaker but they still include a lot of information such as the comparison between BOUML with other UML tools. UMLet and Violet's support resources are less informative than ArgoUML and BOUML. (CM=0.088)

Table 6-25 Capacity Pair Comparison

Capacity	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/8	¼	½	0.061
ArgoUML	8	1	3	6	0.596
BOUML	4	1/3	1	3	0.243
UMLet	2	1/6	1/3	1	0.099

The support resources of these four alternatives are all well written and easy to understand. Therefore we treat them equally on this point (table 6-26). (CM=0)

Table 6-26 Understandability Pair Comparison

Understandability	Violet	ArgoUML	BOUML	UMLet	Priority
Violet	1	1	1	1	0.25
ArgoUML	1	1	1	1	0.25
BOUML	1	1	1	1	0.25
UMLet	1	1	1	1	0.25

Table 6-27 is the sub-criteria comparison under the support resource criteria. In this case study, we emphasize on the relevance of the resource. Besides, between capacity and understandability, we favor the capacity. (CM=0.042)

Table 6-27 Support Resource Pair Comparison

Support Resource	Relevance	Capacity	Understandability	Priority
Relevance	1	3	7	0.669
Capacity	1/3	1	3	0.243
Understandability	1/7	1/3	1	0.088

We do not want to buy commercial support for our OSS UML tool. Thus the commercial support is ignored in the field support evaluation. We count the number of replies in mailing list, tracker system and forum from Jan 24 to Jan 30, 2007. ArgoUML provides community support via mailing list and the indicator value is 94. BOUML uses only mailing list as well and the value is 16. Violet and UMLet have tracker system, mailing list and forum set up on Sourceforge.net but there are nearly no activities on them (table 6-28). (CM=0.071).

Table 6-28 Field Support Pair Comparison

Field Support	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/7	1/3	1	0.079
ArgoUML	7	1	4	7	0.635
BOUML	3	1/4	1	3	0.207
UMLet	1	1/7	1/3	1	0.079

For the whole support strength (table 6-29), we think support resources are slightly importantly than field support since the resources are more dependable. We have the small comparison as following (CM=0):

Table 6-29 Support Strength Pair Comparison

Support Strength	**Field Support**	**Support Resources**	**Priority**
Field Support	1	1/2	0.333
Support Resources	2	1	0.667

Finally the support strength priority vector is <Violet:0.088, ArgoUML:0.597, BOUML: 0.220, UMLet:0.095}.

6.4.5 Popularity

According to ArgoUML website download statistics, ArgoUML has been downloaded around 1,290,000 times till Feb 2007. BOUML's download totals more than 90,000. On

Sourceforge.net, Violet has been downloaded 48 till Feb, 2007. It has two websites, one of which provides download without any statistics data. UMLet is registered on Sourceforge.net where its download count is 0. It includes another download source on its website also without any statistics data. As such, in pair wise comparison (table 6-30), we assume both tools' software use is less preferable than ArgoUML and BOUML but the difference is not significant. (CM=0.065).

Table 6-30 Software Use Pair Comparison

Software Use	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/4	1/2	1	0.121
ArgoUML	4	1	3	4	0.538
BOUML	2	1/3	1	2	0.220
UMLet	1	1/4	1/2	1	0.121

On ArgoUML's website, it lists around 30 developers. BOUML is developed by only one developer. UMLet and Violet has two developers (table 6-31). (CM=0).

Table 6-31 Development Participation Pair Comparison

Development Participation	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/5	1	1	0.125
ArgoUML	5	1	5	5	0.625
BOUML	1	1/5	1	1	0.125
UMLet	1	1/5	1	1	0.125

In the case study, we use Alltheweb.com to compute the indicator for web popularity. Violet's websites (Violet has two websites) are referred for199 times. Also, ArgoUML, BOUML and UMLet's external back link numbers are respectively 4,550, 309 and 553. The results are shown in table 6-32. (CM=0).

Table 6-32 Web Popularity Pair Comparison

Web Popularity	**Violet**	**ArgoUML**	**BOUML**	**UMLet**	**Priority**
Violet	1	1/5	1	1	0.125
ArgoUML	5	1	5	5	0.625
BOUML	1	1/5	1	1	0.125
UMLet	1	1/5	1	1	0.125

Table 6-33 shows how we compare the popularity related criteria. We prefer development participation and web popularity since we believe the indicator values under these two criteria are more reliable than the indicator value under software use.

Table 6-33 Popularity Pair Comparison

Popularity	**Software Use**	**Development Participation**	**Web Popularity**	**Priority**
Software Use	1	1/5	1/5	0.091
Development Participation	5	1	1	0.455
Web Popularity	5	1	1	0.455

The final priority vector regarding the popularity is <Violet:0.125, ArgoUML:0.617, BOUML:0.134, UMLet:0.125>

6.4.6 Wrap up

In the evaluation, we compare the criteria under the objective in the hierarchy (table 6-34). We emphasize the functional suitability because we want to reduce the secondary development work. Source code is important than support strength and popularity because after all we have to deal with it when we adopt the OSS UML tool. There is no preference between support strength and popularity. (CM=0.05)

Table 6-34 OSS Evaluation Pair Comparison

OSS Evaluation	Functional Suitability	Source Code	Support Strength	Popularity	Priority
Functional Suitability	1	2	3	3	0.455
Source Code	1/2	1	2	2	0.263
Support Strength	1/3	1/2	1	1	0.141
Popularity	1/3	1/2	1	1	0.141

The last step of the evaluation is developing the final priority vector of the alternatives for the final objective. The priority vector is <Violet:0.155, ArgoUML:0.372, BOUML:0.277, UMLet:0.197>. Figure 6-2 shows how much functional suitability, source code, support strength and popularity contribute to the final priority. From this vector, it is easy to see ArgoUML is the best OSS UML tool for our requirement.

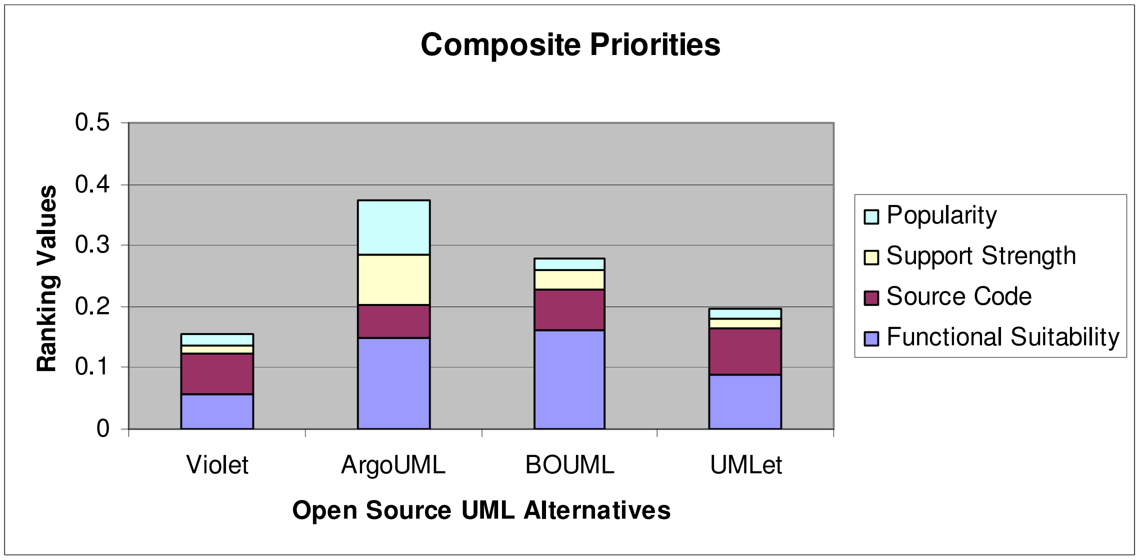

Figure 6-2 Composite Priority

6.5 Summary of the Case Study

This case study illustrates how to apply the OSS selection process to solving the problem of selecting the best open source UML tool. From this case study, we can see with the aid of the selection process, we find the best open source UML modeling tool out of ten alternatives for our development. In the evaluation AHP helps effectively organize the multiple attributes and reach the final decision. There are three lessons learned from this case study. First, the process still requires more CASE tool support. For example, we need a software metric measurement tool supporting more programming languages. Second, the redundant criteria should be recommended in the case that the evaluation regarding certain criteria cannot proceed if there is some information missing. Third, when we set the function criteria under out of box function compatibility, we should be careful to make sure the function criteria are independent.

7 SUMMARY

I propose an open source software selection process which consists of three phases: identification, screening and evaluation. In the first phase, potential open source alternatives are identified based on the high-level requirements; then the refined requirements such as underlying platform, implementation language, dependent modules, standard compliance and license, are applied to reduce the alternatives. The evaluation phase is a scrutiny on the remaining competitive alternatives from functionality, source code, support strength and popularity. The multi-criteria decision technique, analytic hierarchy process, plays an important role in the evaluation. It helps users organize various open source software related criteria and make an informed decision by calculating the relative priority rankings of alternatives under each criterion.

The selection process is operable and effective to solve the problem in practice, which is demonstrated in the cases study. Via following the process, I successfully select the best unified modeling tool among ten potential alternatives to satisfy the development requirements. The decision is well-founded since it takes into consideration not only the attributes of the alternatives but also the specific requirements.

REFERENCES

[1] C. DiBona, S. Ockman and M. Stone, *Open Sources Voice from the Open Source Revolution*, O'Reilly & Associates, 1999.

[2] "The Free Software Foundation", http://www.fsf.org, accessed on April 1, 2007.

[3] "The Free Software Definition", http://www.fsf.org/licensing/essays/free-sw.html, accessed on April 1, 2007.

[4] S. Hissam, C. B. Weinstock, Daniel Plakosh and Jayatirtha Asundi, "Perspectives on Open Source Software, Software Engineering Institute", technical report, CMU/SEI-2001-TR-019, Carnegie Mellon University , Nov 2001.

[5] "The Open Source Definition Version 1.8", http://www.opensource.org/docs/definition.html, accessed on April 1, 2007, 2001.

[6] A. Senyard, M. Michlmayr, "How to Have a Successful Free Software Project", *Proc. the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pp.84-91, Dec 2004.

[7] E.S. Raymond, *The Cathedral and the Bazaar*, O'Reilly Media Inc., 2001.

[8] B. Arief, C. Gacek, and T. Lawrie, "Software Architectures and Open Source Software – Where can Research Leverage the Most?" *Proc. Making Sense of the Bazaar: 1st Workshop on Open Source Software Engineering*, May 2001.

[9] I. Stamelos, L. Angelis, A. Oikonomou, and G.L. Bleris, "Code Quality Analysis in Open Source software development", *Information Systems Journal*, vol 12, issue 1, pp 43-60, Jane 2002.

[10] K. Crowston and J. Howison, "The Social Structure of Free and Open Source Software Development", *First Monday*, vol 10, number 2, Feb 2005.

[11] K. Crowston, H. Annabi, J. Howison, C. Masango, "Effective Work Practices for Software Engineering: Free/Libre Open Source Software Development", *Proc. 2004 ACM workshop on Interdisciplinary Software Engineering Research*, pp18-26, 2004.

[12] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two Case Studies of Open Source Software Development: Apache And Mozilla", *ACM Transcations on Software Engineering and Methodology*, vol 11, number 3, pp.309-346, July 2002.

[13] J. Y. Moon and L. Sproull, "Essence of Distributed Work: The Case of the Linux Kernel", *First Monday*, vol 5, number 11, Nov. 2000.

[14] J. Kontio, "OTSO: A Systematic Process for Reusable Software Component Selection", technical report, CS-TR-3478, University of Maryland, December 1995.

[15] J. Kontio, "A Case Study in Applying a Systematic Method for (COTS) Selection", *Proc.18th International Conference on Software Engineering (ICSE'96)*, pp.201-209, March 1996.

[16] C. Alves and J. Castro, "CRE: A Systematic Method for COTS Components Selection", *XV Brazilian Symposium on Software Engineering*, pp.193-207, Oct. 2001.

[17] N. A. Maiden and C. Ncube, "Acquiring COTS Software Selection Requirements", *IEEE Software*, vol 15, issue 2, ISSN: 0740-7459, pp.46-56, March/April 1998.

[18] M. Krystkowiak, V. Betry, E. Dubois, “Efficient COTS Selection with OPAL Tool”, *Proc. International Workshop on Models and Processes for the Evaluation of COTS Component (MPEC 2004) W7S Workshop - the 26th International Conference on Software Engineering*, pp.23-26, May 2004.

[19] C. Murphy, “The Open Source Maturity Model is a Vital Tool for Planning Open Source Success”, www.navicasoft.com/pages/osmm.htm, accessed on April 1, 2007.

[20] B. Golden, “Make Open Source Software Ready for the Enterprise: The Open Source Maturity Model, Extracted from Succeeding with Open Source”, Addison-Wesley Professional, Oct 2005.

[21] F. Duijnhouwer and C. Widdows, “Open Source Maturity Model, Capgemini Expert Letter”, http://www.seriouslyopen.org/nuke/html/modules/Downloads/osmm/GB_Expert_Letter_Open_Source_Maturity_Model_1.5.3.pdf, accessed on April 1, 2007, August 2003.

[22] “Business Readiness Rating for Open Source-A Proposed Open Standard to Facilitate Assessment and Adoption of Open Source Software”, www.openbrr.org, accessed on April 1, 2007, BRR 2005-RFC1.

[23] K. v. d. Berg, “Finding Open options, An Open Source Software Evaluation Model with a Case Study on Course Management Systems”, Master thesis, Tilburg University, August 2005.

[24] D. Wheeler, "How to Evaluate Open Source Software/Free Software (OSS/FS) Programs", http://www.dwheeler.com/oss_fs_eval.html, accessed on April 1, 2007, Jan 2006.

[25] E. Forman, DSc. *Decision by Objectives*, World Scientific Publishing Company, 1st edition, Feb 15 2002.

[26] L. Santillo, "Early & Quick COSMIC-FFP Analysis Using Analytic Hierarchy Process", *Lecture Notes in Computer Science*, vol. 2006, *Proc. the 10th International Workshop on New Approach in Software Measurement*, Springer-Verlag, pp.147-160, Oct 2000.

[27] T. L. Satty, *The Analytic Hierarchy Process*, McGraw-Hill Inc., 1980.

[28] J. McCaffrey, "The Analytic Hierarchy Process", *MSDN Magazine*, vol 20, No 6, pp.139-144, June 2005.

[29] G.A. Miller, "The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *Psychological Review*, vol. 63, pp81-97, Mar 1956.

[30] "Document E09, SourceForge.net: Software Map", http://sourceforge.net/docs/B09#topic_suggestion, accessed on April 1, 2007.

[31] S. Mendenhall, W. Mendenhall and L. Ott, *Elementary Survey Sampling (third edition)*, Duxbury Press, Boston.

[32] A. C. Tamhane and D. D. Dunlop, *Statistics and Data Analysis from Elementary to Intermediate*, Prentice Hall, Oct 1999.

[33] "Document B06, SourceForge.net: Search", http://sourceforge.net/docman/display_doc.php?docid=32777&group_id=1, accessed on April 1, 2007.

[34] M. Sturmer, "Open Source Community Building", PhD thesis, University of Bern, March 2005.

[35] J. E. Robbins, "Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools", in *Making Sense of the Bazaar: Perspectives on Open Source and Free Software,* J. Feller, B Fitzgerald, S. Hissam, and K. Lakhani, MIT Press, 2004.

[36] L. Rosen, *Open Source Licensing Software Freedom and Intellectual Property Law*, Prentice Hall, 2004.

[37] *The Copyright Law of the United States*, June 2003 Edition of Circular 92.

[38] *U.S. Patent Act* -- 35 USCS Sects. 1 – 376.

[39] Daniel A. Tysver, "Rights Granted Under U.S. Patent Law", www.bitlaw.com, accessed on April 1, 2007.

[40] "Copyright vs. Trademark vs. Patent", www.lawmart.com, accessed on April 1, 2007.

[41] "Open Source License Overview", http://sourceforge.net/docman/display_doc.php?docid=778&group_id=1, accessed on April 1, 2007.

[42] "Report of License Proliferation Committee and draft FAQ", www.opensource.org, accessed on April 1, 2007.

[43] "Copyright Registration for Derivative Works, Circular 14", http://www.copyright.gov/circs/circ14.pdf, accessed on April 1, 2007.

[44] "Free and Open Source Software Licensing White Paper", http://www.opensolaris.org/, accessed on April 1, 2007, April 2006.

[45] S. J. Davidson, "A Primer on Open Source Software for Business People and Lawyers", World Intellectual Property Organization, http://www.wipo.int/sme/en/documents/opensource_software_primer.htm, accessed on April 1, 2007.

[46] J. Lerner and J. Tirole, "The Scope of Open Source Licensing", *Journal of Law, Economics, and Organization*, vol. 21(1), pp.20-56, April 2005.

[47] "What Is CopyLeft", http://www.gnu.org/copyleft/, accessed on April 1, 2007.

[48] "Various Licenses and Comments about Them", http://www.fsf.org/licensing/licenses/, accessed on April 1, 2007).

[49] "The MIT License", http://www.opensource.org/licenses/mit-license.php, accessed on April 1, 2007.

[50] "The BSD License", http://www.opensource.org/licenses/bsd-license.php, accessed on April 1, 2007.

[51] "Apache License, Version 2.0", http://www.opensource.org/licenses/apache2.0.php, accessed on April 1, 2007.

[52] "The GNU General Public License (GPL), Version 2", http://www.opensource.org/licenses/gpl-license.php, accessed on April 1, 2007, June 1991.

[53] "GNU Lesser General Public License, Version 2.1", http://www.opensource.org/licenses/lgpl-license.php, accessed on April 1, 2007, February 1999.

[54] "The Mozilla Public License 1.1 (MPL 1.1)", http://www.opensource.org/licenses/mozilla1.1.php, accessed on April 1, 2007.

[55] R. Wilson, "The Mozilla Public License – An Overview", http://www.oss-watch.ac.uk/resources/mpl.xml, accessed on April 1, 2007, Nov 2005.

[56] "Common Development and Distribution License (CDDL) Description and High-Level Summary of Changes", http://www.sun.com/cddl/CDDL_why_summary.html, accessed on April 1, 2007.

[57] "Common Development and Distribution License (CDDL), Version 1.0", http://www.opensource.org/licenses/cddl1.php, accessed on April 1, 2007.

[58] "Common Public License Version 1.0", http://www.opensource.org/licenses/cpl1.0.php, accessed on April 1, 2007.

[59] "Eclipse Public License – v 1.0", http://www.opensource.org/licenses/eclipse-1.0.php, accessed on April 1, 2007.

[60] International Organization for Standardization, "Information Technology-Software Product Evaluation: Quality Characteristics and Guidelines for Their Use", *International Organization for Standardization/International Electrotechnical Commission 9126*, March 1991.

[61] T. Punter, R. v. Solingen, J. Trienekens, "Software Product Evaluation – Current Status and Future Needs for Customers and Industry", *Proc. 4th Conference on Evaluation of Information Technology*, Oct 1997.

[62] S. Hildreth, “Software QA 101: The Basics of Testing”, http://www.informit.com/articles/article.asp?p=333473&rl=1, accessed on April 1, 2007, Sept 3, 2004.

[63] A. Abran, “Function Points: A Study of Their Measurement Processes and Scale Transformations”, *The Journal of Systems and Software*, vol 25, pp 171-184, May 1994.

[64] B. A. Kitchenham and L. Jones, “Evaluating Software Engineering Methods and Tool Part 7: Planning Feature Analysis Evaluation”, *ACM SIGSOFT Software Engineering Notes*, vol 22, issue 4, pp. 21-24, July 1997.

[65] T. Euler, “An Adaptable Software Product Evaluation Metric”, *Proc. Software Engineering and Applications*, Nov. 2005.

[66] J. Kontio, S. Chen, K. Limperos, R. Tesoriero, G. Caldiera, and M. Deutsch, “A COTS Selection Method and Experiences of Its Use”, in *Proc. The Twentieth Annual Software Engineering Workshop*, Nov 1995.

[67] L. Holmes, “Evaluating COTS Using Function Fit Analysis”, http://www.qpmg.com/evaluating_cots.htm, accessed on April 1 2007.

[68] A. Cechich, M. Piattini, “Early Detection of COTS Component Functional Suitability”, *Information and Software Technology*, vol 49, issue 2, pp. 108-21, Feb 2007.

[69] W. Scacchi, “Understanding the Requirements for Developing Open Source Software Systems”, *IEEE Proceedings – Software*, vol 149, issue 1, pp. 24-39, Feb 2002.

[70] D. German, A. Mockus, "Automating the Measurement of Open Source Projects", *Proc. Taking Stock of the Bazaar: 3rd Workshop on Open Source Software Engineering, International Conference on Software Engineering*, pp. 63-68, May 2003.

[71] M M Lehman, "Laws of Software Evolution Revisited", Lecture Notes in Computer Science vol. 1149, *Proc. the 5th European Workshop on Software Process Technology*, pp. 108-124, 1996.

[72] R. T. Alexander, M. R. Blackburn, "Component Assessment Using Specification-Based Analysis and Testing," *Technical Report, SPC-98095-CMC, Software Productivity Consortium*, May 1999.

[73] "Measurement Manual (The COSMIC Implementation Guide for ISO/IEC 19761:2003) Version 2.2", http://www.cosmicon.com/, accessed on April 1 2007, Jan 2003.

[74] M.W.Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study", *Proc. International Conference of Software Maintenance (ICSM-00)*, pp. 131-142, Oct 2000.

[75] S. Henry,and K. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, vol 7, pp.510-518, Sept. 1981.

[76] I. Stamelos, L. Angelis, A. Oikonomou and G. L. Bleris, "Code Quality Analysis in Open Source Development", *Information Systems Journal*, vol 12, issue 1, pp. 43-60, 2002.

[77] “How Open-Source and Commercial Software Compare: A Quantitative Analysis of TCP/IP Implementations in Commercial Software and in the Linux Kernel”, www.reasoning.com, accessed on April 1, 2007, 2003.

[78] T. Littlefair B.Sc, “An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment”, PhD Thesis of Edith Cowan University, June 2001.

[79] B. W. Boehm, *Software Engineering Economics*, Prentice Hall PTR, Oct 1981.

[80] T. J. McCabe, “A Complexity Measure”, *IEEE Transactions on Software Engineering*, vol 2, pp. 308-320, Dec 1976.

[81] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, vol 20, number 6, pp.476-494, June 1994.

[82] “Integrating Open Source into Your Business”, ftp://ftp.software.ibm.com/linux/pdfs/integratingOS21Aug06.pdf, accessed on April 1, 2007.

[83] D. Woods and G. Guliani, *Open Source for the Enterprise*, O’Reilly & Associate, July 2005.

[84] R. Zachary, “Six Options for Open-Source Support”, *Information Week’s Optimize Magazine*, Issue 53, March 2006.

[85] E. Lai, “Panel: Lack of Support Hinders Open-Source Popularity”, http://www.computerworld.com/softwaretopics/software/story/0,10801,108740,00.html, accessed on April 1 2007, Feb 16, 2006.

[86] K. R. Lakhani and E. v. Hippel, "How Open Source Software Works: "Free" User-to-User Assistance", *Elsevier Science B.V. Research Policy*, vol 32, pp. 923-943, 2003.

[87] "Tracker: Bug Reporting, Support Requests, Feature Requests, Patches", http://sourceforge.net/docman/display_doc.php?docid=24202&group_id=1#purpose, accessed on April 1, 2007.

[88] "Document D04: Statistics", http://sourceforge.net/docman/display_doc.php?docid=14040&group_id=1, accessed on April 1, 2007.

[89] M. Michlmayr, "Software Process Maturity and Success of Free Software Projects", In *Zielnski, K., Szmuc, T. (Eds.), Software Engineering: Evolution and Emerging Technologies*, pp. 3-14, 2005.

[90] K. Crowston, H. Annabi and J. Howison, "Defining Open Source Software Project Success", *Proc. International Conference on Information System (ICIS)*, Dec. 2003.

[91] J. Howison and K. Crowston, "The Perils and Pitfalls of Mining SourceForge", *Proc. the Workshop on Mining Software Repositories at the 26th International Conference on Software Engineering*, 2004.

[92] D. Weiss, "Measuring Success of Open Source Projects Using Web Search Engines", *Proc. the First International Conference on Open Source Systems (OSS 2005)*, pp.93-99, 2005.

[93] "Computerworld Development Survey Gives Nod to C#", http://www.computerworld.com/developmenttopics/development/story/0,10801,100542,00.html, accessed on April 1, 2007, March 2005.

[94] G. McLeod, "Extending UML for Enterprise and Business Process Modeling", *Proc. International Workshop UML 98*, pp. 195-204, June 1998.

[95] R. J. Muller, "Database Design for Smarties: Using UML for Data Modeling", *Morgan Kaufmann*, Feb. 1999.

[96] Y.G.Kim, H.S.Hong, SM.Cho, D.H.Bae and S.D. Cha, "Test Cases Generation from UML State Diagrams", *IEEE Proceedings: Software*, vol. 146, no. 4, pp. 187-192, Aug 1999.

[97] J. Siegel, "Introduction to OMG UML", http://www.omg.org/gettingstarted/what_is_uml.htm, accessed on April 1, 2007, July 2005.

[98] Object Management Group (OMG), "Unified Modeling Languages: Superstructure, version 2.0, formal/05-07-04", August 2005.

[99] C. Kobryn, "Object Modeling with UML: Introduction to UML", www.omg.org/docs/omg/01-03-02.ppt, accessed on April 1, 2007.

[100] S. W. Ambler, "Agile Models Distilled: Potential Artifacts for Agile Modeling", http://www.agilemodeling.com/artifacts/, accessed on April 1, 2007, April 2006.

[101] Object Management Group (OMG), "XML Metadata Interchange (XMI) Specification, v2.0, formal/03-05-02", May 2003.

[102] J. Mustajoki, R. Hmlinen, "Web-HIPRE: Global Decision Support by Value Tree and AHP Analysis", *INFOR Journal*, vol. 38, no. 3, pp.208-220, 2000.

VITA

Guobin He received a B.S. degree in Computer Engineering from National University of Defense Technology, China, in 1994 and an M.S. degree in Computer Science at Beijing University of Aeronautics and Astronautics in 1999. He began to pursue his Ph.D. degree in Computer Science at Texas A&M University in 2001, focusing on parallel computing and compilers. In 2006, he switched to the Doctor of Engineering program, concentrating on software engineering. He can be reached via email at guobinhe@gmail.com. The mailing address is Dr. Dick B. Simmons, Department of Computer Science, Texas A&M University, College Station, TX 77843.