Empirical Study of The Open Source Software Evolution using JUnit

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology
in
Computer Science and Applications

Submitted By
Seema Singh
(Roll No. 601103021)

Under the supervision of
Vineeta Bassi
Assistant Professor
(SMCA)

SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS
THAPAR UNIVERSITY
PATIALA – 147004

June 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Empirical Study of The Open Source Software Evolution using JUnit",** in partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Applications submitted in School of Mathematics and Computer Applications, Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Vineeta Bassi** and refers other researcher's work which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for award of any other degree of this or any other University.

(Seema Singh)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Vineeta Bassi)

Assistant Professor

SMCA

Countersigned by:

(Dr. Rajesh Kumar)

Head

School of Mathematics and Computer Applications

Thapar University

Patiala

(Dr. S. K. Mohapatra)

Dean (Academic Affairs)

Thapar University

Patiala

ABSTRACT

Open Source Software (OSS) is software products available to the public use, with its source code to study, change, and improves its quality. Open Source Software Development (OSSD) is the process by which open source software is developed within the process of software engineering life-cycle methods. However when open source software used for commercial purpose, then an open source license is required. Open source software is mostly developed in a public and collaborative manner. Open source software development movement has been getting focus on both the area of development and academics. Despite the fact that the open source software developments have seen remarkable successful in recent years, there are a number of product quality issues and challenges facing the open source software development model. The data of Open Source Software (OSS) in the repositories is available for most large software projects and represents a detailed and rich record of the historical development of software systems. Until recently these repositories were used primarily for their intended activities such as maintaining versions of the source code or tracking the status of a defect. Software practitioners and researchers are beginning to recognize the potential benefit of mining this information for other purposes. This study thus has been conducted focusing on the evolution of the open source software. The idea in this thesis is to study the evolution of the open source software under the effects of the data mined from the version control system.

The laws of software evolution and their development as the basis for a theory of software evolution represents a major intellectual contribution and challenge to the software engineering research community, and to the broader community of computer science.

In this thesis an open source unit testing framework for the Java programming language is studied on its different parameters. The approach of this thesis is study of software evolution in open source software environment by using differ.ent software resources to support Lehman's law.

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life. This work would not have been possible without the encouragement and able guidance of my supervisor **Vineeta Bassi**. I thank my supervisor for their time, patience, discussions and valuable comments. Their enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to **Dr. Rajesh Kumar**, Associate Professor and Head, School of Mathematics and Computer Applications, for motivation and inspiration that triggered me for the thesis work.

I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra,** Senior Professor and Dean of Academic Affairs the University, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of School of Mathematics and Computer Applications for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my brother, since he insisted that I should do so. I would also like to thank my close friends for their constant support.

List of Contents Page No.

List of Figures Page No.

List of Table Page No.

List of Abbreviations

ASPM	Average Statement Per Method
BSD	Berkley Software Distribution
C&CLOC	Lines of Code and Comments
CLOC	Comment Lines of Code
CPL	Community Public License
CPM	Counting Practice Manual
CVS	Concurrent Version System
FS	Free Software
FOSS	Free Open Source Software
DSEE	Domain Software Engineering Environment
GCC	GNU Compiler Collection
GDB	GNU Symbolic Debugger
GPL	GNU General Public License
IFPUG	International Function Point User Group
LGPL	GNU Lesser General Public License
LOC	Lines of Code
MPL	Mozilla Public License
MSR	Mining Software Repositories
MVG	McCabe Cyclomatic Complexity
PBS	Percentage Branch Statements
OSD	Open Source Development
OSP	Open Source Projects
OSS	Open Source Software
OSSD	Open Source Software Development
OSSDP	Open Source Software Development Process
RCS	Revision Control System
SCCS	Source Code Control System
SLOC	Source Lines of Code
SVN	Subversion
VC	Version Control
VCS	Version Control System
VCSes	Version Control Systems

Chapter 1
INTRODUCTION

This chapter will cover all basic information about the used concept in thesis work. It contain introduction part of OSS and its development process, what is software evolution and what the basic process steps used in evolution process. Then for software evolution needs different version of selected OSS by using SVN repositories.

The world of software changes rapidly. New technologies and with them, new opportunities, come and go at an even increasing speed. The Free and Open Source Software (FOSS) movement is one such development that is playing out before us today [1]. Open Source Software Development (OSSD) combines features found in traditional software processes with other features in a unique way that can potentially produce high-quality software, faster and cheaper within the rapidly changing Internet environment. Although OSSD is not a faultless solution, it provides potential benefits and opportunities to the system development process. The most obvious benefit is the reduced cost that open source provides.

In recent years, the research on free and open source software focused on motivation, intellectual property and business value, lacked of all-around estimation and development prospect analysis. Furthermore, most researches centralized on open source software and ignored free software which is the pioneer of the free and open source software movement. Richard Stallman said: "The Free Software movement and the Open Source movement are like two political camps within the free software community" [1].

This thesis is focusing on the evolution of the Open Source Software. The goal of the thesis is Open Source Software and its evolution under the effects of Version Control System. The work includes examining the observed differences in all the versions of an Open Source Software and then comparing these series of versions, and also tracking the changes in projects as they evolve, this thesis also proved the Lehman’s Laws of Evolution and provides a body of original empirical evidence to support it.

1.1 Open Source Software

Open Source Software (OSS) allows users to have access to the source code of the software, the freedom to use the software as they see fit, modify the software to create derived works, and redistribute the derivative software for free or at a charge. OSS allows engagement, interaction, feedback and sharing of content (software) at the user level and flourishes as a result of this.

In 1984, Richard Stallman founded the Free Software Foundation (http://www.fsf.org/fsf/fsf.html), a tax-exempt charity that raises funds for work on the GNU Project (http://www.gnu.org/gnu/thegnuproject.html). GNU is a recursive acronym for "GNU's Not Unix" and a homophone for "new." The GNU Project seeks to develop Unix-compatible software and return software to a state of freedom. Stallman is both an open source evangelist and a major open source contributor as the principal author of the GNU C Compiler (GCC), GNU symbolic debugger (GDB), GNU Emacs, and more. All these packages provide essential tools for GNU/Linux.

The purpose of the Free Software Foundation is not to ensure distributing software to the end user without cost, but to ensure that the end user can use the software freely. From the Free Software Foundation's perspective, the term "free software" had nothing to do with price: A program is free software if user has the freedom to run the program, modify it to suit their needs, redistribute copies either gratis or for a fee, and distribute modified versions of the program so that the community can benefit from its improvements. Because free refers to freedom, not to price, it is not contradictory to say that software can be both for sale and free simultaneously. According to the Free Software Foundation, the freedom to sell copies is crucial: Selling collections of free software on CD-ROM raises funds for free software development. Therefore, according to the open source definition of the term "free," a program that people cannot freely include on these collections does not qualify as free software.

The Copyleft and General Public License are designed to guarantee this freedom. Copylefts are, in essence, copyrights with GPL regulations. Open source software, essentially a superset of free software, exists in almost countless varieties today, each with its own unique history. Linux, perhaps the best-known open source software package, began modestly in 1991, seven years after the founding of the Free Software Foundation. Linus Torvalds, at the time a graduate student at Helsinki University in

Finland, wrote a Unix-compatible operating system and posted it on the comp.os.minix newsgroup, single-handedly starting the Linux revolution.

Torvalds handed on the kernel maintenance to Alan Cox in 1994 but continued monitoring each kernel version to determine what should be left in and left out. Since 1994, Torvalds has let others deal with user space issues like libraries, compilers, and the many utilities and applications that go into every Linux distribution. By doing so, Torvalds gives users and vendors the freedom to customize his work [2].

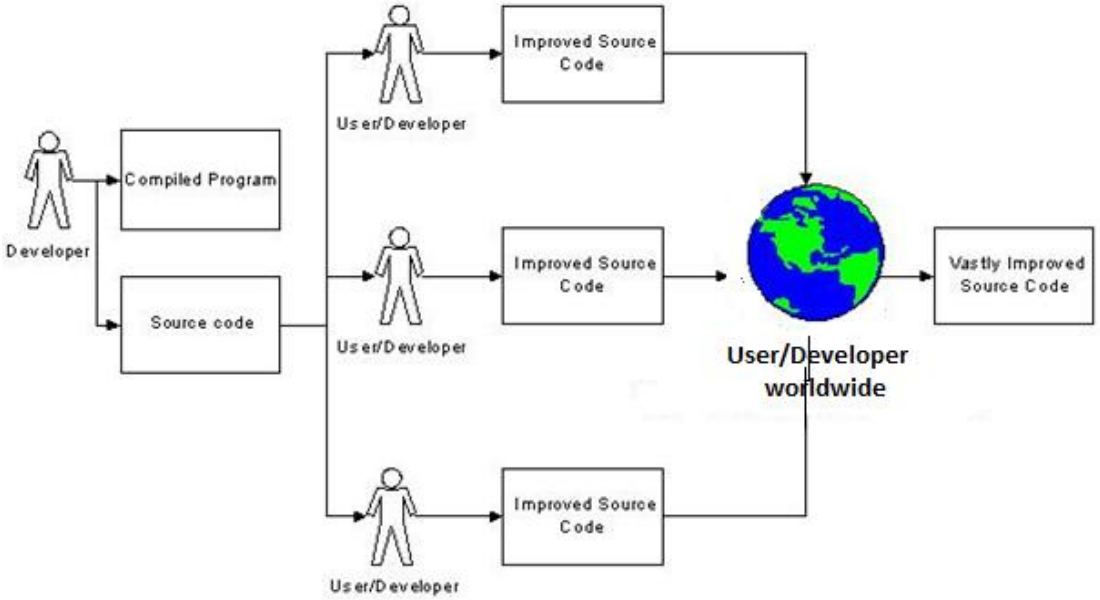


Figure1.1 Open Source Software Concept [3]

This Figure1.1 shows that a single developer who had build a new application, compile that one and provide it's on the open source software environment. Once it available publically each and every user or developer has accessed that code and makes the change if applicable. This kind of approach produced a good quality of code.

Some examples of Open Source Software

		Technical							Managerial			
		Technical support	Backward compatibility	Standard compatibility	Binary availability	Integration with commercial SW	Commercial adoption	Open source dependency	Software license	Current development status	Commercial substitutes	Notes
Operating system	BSD	+	++	OSF	Y	+	+	n/a	BSD	Stable	Y	freebsd.org
	Linux	++	+	OSF	Y	+	++	n/a	GPL	Stable	Y	linux.org
	Macintosh OS X	++	–	OSF	Y (binary only)	–	–	n/a	CPL	Commercial release	Y	apple.com
	Solaris (announced)	++	++	OSF	Y (binary only)	++	++	n/a	CPL	Commercial release	n/a	sun.com
Application environment	Bind	+	++	DNS	Y	+	++	Unix	BSD	Stable	Y	isc.org/bind
	Gnome	+	–	n/a	Y	–	+	Linux, BSD	GPL	Stable	Y	gnome.org
	GNU CC	++	++	ANSI	Y	++	+	Open platform	GPL	Stable	Y	gnu.org
	GNU Emacs	++	+	n/a	Y	+	+	Unix	GPL	Stable	Y	gnu.org
	GNU Make	++	++	n/a	Y	+	+	Unix	GPL	Stable	Y	gnu.org
	Java	++	+	n/a	Y (binary only)	++	++	Open platform	CPL	Stable	n/a	javasoft.com
	KDE	–	–	n/a	Y	–	–	Linux, BSD	BSD	Dev. release	Y	kde.org
	Perl	++	+	n/a	Y	++	++	Open platform	BSD	Stable	N	perl.org
	Sendmail	++	++	SMTP	Y	+	++	Unix (OS version)	BSD	Stable	Y	sendmail.com
	Tk/Tcl	++	+	n/a	Y	+	–	Open platform	BSD	Stable	N	scriptics.com
	X-Windows	++ (vendor)	++	X11	Y (vendor supplied)	++	++	Unix	BSD (X)	Stable	N	x.org
Development library	Gimp	–	–	n/a	Y	–	–	Unix	GPL	Dev. release	Y	gimp.org
	JDK	++	++	n/a	Y (binary only)	++	++	Open platform	CPL	Stable	n/a	Javasoft.com
	LDAP	–	n/a	LDAP	N	+	–	Unix	BSD	Disc.	Y (Netscape)	Defunct
	OpenLDAP	–	–	LDAP	N	–	–	Open	BSD	Dev. release	Y	openldap.org
	OpenSSL	–	+	SSL	N	–	–	Open	BSD	Dev. release	Y	operssl.org
	SSLeay	—	+	SSL	N	–	–	Unix	BSD	Disc.	Y	mozilla-crypto. ssleay.org
Application	Apache	+	++	HTTP	Y	+	++	Open	BSD	Stable	Y	apache.org
	Mozilla	+	+	HTML	Y	++	–	Open platform	CPL	Stable release	Y (Netscape)	mozilla.org
	MySQL	+	+	SQL	Y	–	–	Unix	CPL (Recent:GPL)	Stable	Y	mysql.org
	PHP	+	+	n/a	Y	–	–	Open platform	BSD	Stable	Y	php.net
	Pine	+	+	SMTP, MIME	Y	–	+	Unix	BSD	Stable	Y	pine.org

Table1.1 Examples of Open Source Software [4]

1.1.1 Open Source Development Process

There is also research work describing the development process enacted in some popular OSS projects. Yet there is no internationally accepted OSS development process model defining how OSS is developed in practice. A process description is important for coordinating all the software development activities, including both people and technology. Coordination is enabled by giving process engineers the

chance to collectively discuss and administer the dependencies between people, processes and technologies [2].

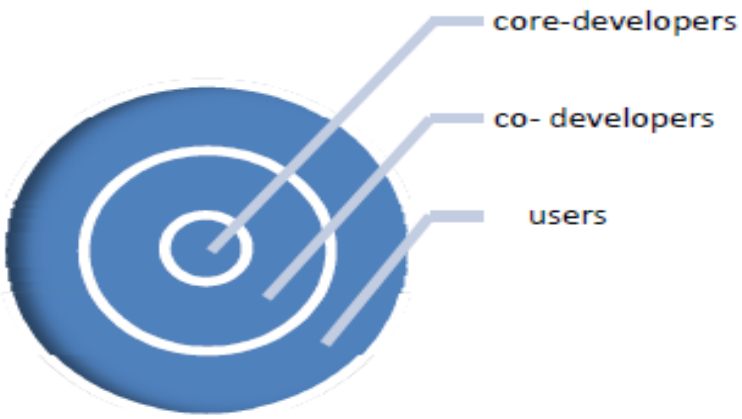


Figure1.2 Structure of developers and users in OSSD model [5]

Open source software development is a kind of distributed software development that has a large amount of contributors and because of using internet and make sharing freely it is so successful and useful that developers can communicate over the distance. Due to having variety of contributors in OSS projects and the story of knowledge sharing among them the project can be more powerful and this might lead to improve even the position of contributors. For example (Figure1.2) they can move to the developers group from the users or even in developers group move to the core developers. Co-developers that are the people who have directly impact on software development in the project, also the effect on code base and they can find issues on licensing. Figure 1.3 shows the general open source system development cycle that allows multiple participants to contribute to software development process requires a massive coordination effort.

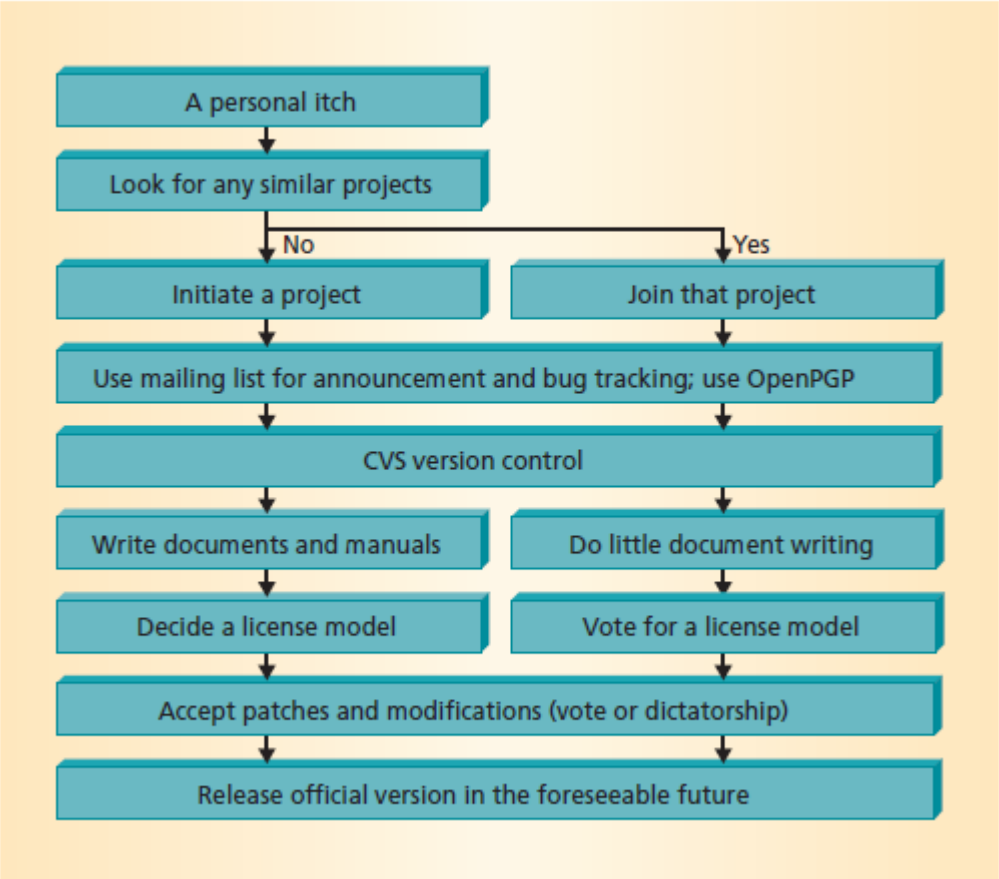


Figure1.3 The general open source system development cycle. Allowing multiple participants to contribute to the software development process requires a massive coordination effort [2]

The open source software development cycle, as the flow chart in Figure 1.3 allows literally anyone to participate in the process, but having multiple participants means a massive coordination effort. Developers can use several different models to coordinate these large-scale efforts, from standardizing software due to get different kind of approach each participant merge n new versatility by that good quality software will introduce.

1.1.2 Open Source Software Licenses

An OSS license defines the privileges and restrictions that a user must follow in order to use and modify software. If a developer wants to publish a program as OSS, he/she can distribute the program as an un-copyrighted product. The users of the program can read, copy, modify, and redistribute the program. However, it is also possible for someone to make the program copyrighted by modifying the original program. Consequently, the modified, copyright-protected program becomes a personal

property, and is not OSS any more. To prevent this situation, most of OSS licenses implement “copy left” concept: anyone who redistributes the software, with or without changes, must pass along the freedom to further copy and change it [6].

Open Source Software Licenses and Their Effects

	Can be mixed with nonfree software	Proprietary modifications can be made private	Can be relicensed	Allows proprietary licensing
GNU Public License	Y	N	N	N
Library GPL	Y	Y	N	N
Berkeley Software Development	Y	Y	N	N
Community Public License	Y	N	N	Y
Commercial	Y	Y	N	N

Table1.2 OSS Licences and their effect [4]

Adopting OSS is not free from the terms set forth by software licenses. OSS products have several different types of license, each of which imposes a different set of restrictions that could potentially impede critical project capabilities such as internal reuse, proprietary custom extensions, and resale.

Table1.2 lists the following common types of OSS license: GPL (GNU Public License), perhaps the most common one; LGPL (Library GPL), a modified version of GPL applying specifically to software libraries; BSD (Berkeley Software Development), applying mostly to derivatives and variants of BSD Unix; and CPL (Community Public License), a type of license typically found in community versions of commercial software. The licensing terms of your chosen software will affect your current and future project scope, such as internal use versus resale [4].

Basically source license models fall into three general categories: free the program can be freely modified and redistributed; copy left the owner gives up intellectual property and private licensing; and GPL-compatible licenses are legally linked to the GPL licensing structure.

In addition to open source licensing models, developers use hundreds of other licensing models for the many kinds of software they market, ranging from shareware to giftware to proprietary agreements, or anything in between. Each of these models contributes to the general confusion surrounding licensing arrangements and the terminology that describes them, because ordinary users seldom read software licenses in detail as also shown in Table1.3.

Some common open source license models include:

• **General Public License**. This free software licensing uses the copyleft model, a self-perpetuating spiral model that strictly ensures distribution of any derivative work under the same license model.

• **Lesser GPL.** Once known as library GPL, LGPL lets users extend the source with proprietary modules.

• **Berkeley Software Distribution.** The BSD model offers free code distributions and allows covering derivative works under different terms as long as the necessary credit is given. Examples of BSD licensees include Apache, BSD-related OSs, and free versions of Sendmail.

• **Mozilla Public License**. MPL requires distributing derivative works under MPL, which means that derivative work loses patent rights but still can enjoy private licensing. However, a module that MPL covers cannot legally be linked together with a module that GPL covers.

• **Netscape Public License.** This MPL extension permits Netscape to use your added code even in its proprietary versions of the program.

• **Qt Public License**. A noncopyleft free software license, QPL requires distributing any modified source distributions only as patches.

• **Artistic License**. Nearly identical to the GPL model, AL doesn't require distributing derivative works under the same terms when a company uses them internally [2].

Licensing model	Free software	Open source	Copyleft	GPL-compatible	Examples
GPL	Yes	Yes	Yes	Yes	CVS
LGPL	Yes	Yes	Partial	Yes	GNU C library
X11	Yes	Yes	No	Yes	XFree86
Python	Yes	Yes	No	Yes	Python
BSD	Yes	Yes	No	No	Apache, Sendmail
MPL/NPL	Yes	Yes	No	No	Mozilla
QPL	Yes	Yes	No	No	Qt
Sun Industry Standard Source License (SISSL)	Yes	Yes	No	No	Commercial-version StarOffice
Artistic License (AL)	No	Yes	No	No	Perl
Apple Public Source License (APSL)	No	Yes	No	No	Darwin

Table1.3 Open Source Licensing Models [2]

We've listed several of these licensing models in Table 1.3 for easy cross-reference. Among open source licensing structures, although the GPL license calls for the

strictest regulation, complaints and public scorn currently provide the main methods for opposing GPL violations. Despite the absence of harsher sanctions, most companies are willing to correct licensing problems and release the modified version of their software to avoid a damaged reputation.

For example, nVidia modified the XFree86 driver for use in its graphics drivers, but did not release the code. Because part of the drivers' code falls under the GPL model, nVidia had to remove all GPL code, then re-release the drivers. In a similar case with a different outcome, Microsoft bought Software Systems, makers of GPL-regulated software, and repackaged its products as Microsoft Interix to provide a UNIX environment within Windows. By doing so, Microsoft could claim Interix as its own work, thereby skirting the GPL regulation.

1.2 Software Evolution, Its Requirements and Evolution Law

The evolution of open source software conforms to the laws of software evolution that have been in development for more than 30 years. The laws of software evolution and their development as the basis for a theory of software evolution represents a major intellectual contribution and challenge to the software engineering research community, and to the broader community of computer science. The principal developer and advocate for the laws of software evolution is Professor M.M. (Manny) Lehman from Imperial College in London, and over the years his work has been expanded and refined by a growing list of students and scholars of software evolution. However, the emerging trend of free or open source software (F/OSS) development, often focusing attention to popular software systems like the GNU/Linux operating system, the Apache Web server, the Mozilla Web browser, and many others, raises the question as to whether F/OSS conforms to or breaks the laws of software evolution as currently formulated. Finding conformance would be reassuring to the current outstanding effort representing decades of study, whereas finding breakdowns, inconsistencies, or failures might point to refutations of the laws/theory, or at least the need to rethink, refine, and reformulate the laws/theory to account for the evolution of F/OSS [7].

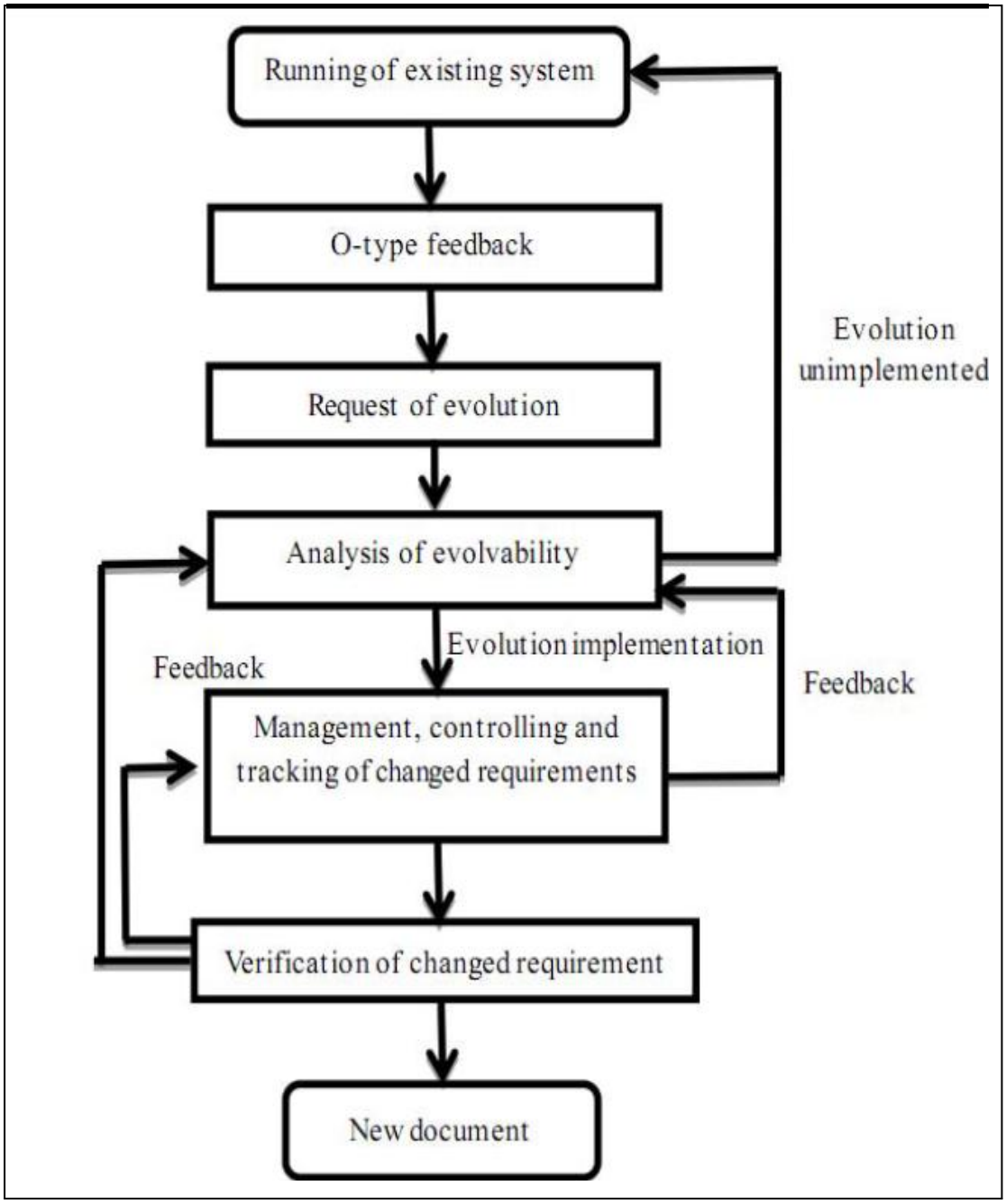


Figure1.4 Process of software evolution requirement [7]

Software evolution is a process of continuous feedback and iteration, so is the requirement process in software evolution. The whole process of evolution requirement is shown in Figure 1.7

The figure is described as the following:

Step 1: Running of existing system. Only when the system is running, can the faults and errors be found and the users know their system and their demands better.

Step 2: The O-type feedback. This feedback comes from the system running process and the purpose of this step is to discover and obtain the evolution requirements. As the reason of driving the evolution, this type of feedback can be sort into user feedback and system feedback. After the system was delivered, with the time passing by, users are getting used to their new system and will change their minds about the system. Meanwhile, external environment is changing and under the influence of elements as laws and regulations' changing, users may demand more towards the system.

Step 3: The request of evolution. After the O-type feedback, request of evolution should be proposed. During this stage, requirement engineers summarize the problems proposed before, and make the evolution plans. It is necessary in evolution requirement management to make evolution requirement plans.

Step 4: Analysis of evolvability of the running system. After the plan is made, time comes to understand and analyse the running system. If the original requirement documents exit, the documents should be referred to for a better understanding of the system and a more accurate judgement to the system. If the documents are lost or not complete, to get a correct assessment report of evolution requests, reverse engineering is needed. According to the report, either continuation or stop of the evolution process will be executed during the next stage.

Step 5: Management, controlling and tracking of changed requirements. System requirements are always changing, especially for the complex systems. To keep the consistency of requirements, every possible requirement change should go through requirement changing managing process which should be realized in formalized ways. Under this way, the changes to requirement documents can be under control. To be specific, there are three activities in requirement changing management that are analysis of the requirement problems, analysis of change and the execution of the change.

Step 6: Verification of evolution requirement. Testing is the way to find the problems as early during the evolution requirement process.

Step 7: Produce new requirement document and begin the evolution process [7].

The following table shows the Lehman's Law of Evolution.

1974	1.Continuing Change	Software systems should continuously be changed in order to be used for longer period or they become less satisfactory.
1974	2. Increasing Complexity	The complexity of a software system increases unless some preventive maintenance is done to control it. Increase in complexity may arise due to addition of more functionalities leading to more interaction.
1974	3. Self-Regulation	Evolution process of software systems is self-regulatory. This means that growth rate is regulated by the maintenance process.
1978	4. Conservation of Organizational Stability	The average global effective rate in evolving software tends to remain constant over product lifetime. This means it is hard to change the staffs who have been working on evolving software.
1978	5. Conservation of Familiarity	The average incremental growth remains constant as the software evolves. A huge change that might cause lack of familiarity of staff members is avoided.
1991	6. Continuing Growth	The functional content of software systems must be continually enhanced in response to user feature request in order to maintain user satisfaction over its life period.

1996	7. Declining Quality	The quality of software systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
1996	8. Feedback System	Evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement for other than the most primitive processes.

Table1.4 Lehman's Laws of Software Evolution [8]

1.3 Version Control System

Version control systems have become indispensable software engineering tools. They enable a developer to work on a private copy of a jointly developed project and provide support for merging her changes with the contributions of other team members later on. As these systems track the history of a project, they contain a wealth of information that can be leveraged by other software engineering tools.

A version control system is a piece of software that helps the developers on a software team work together and also archives a complete history of their work [9].

1.3.1 Brief History of Version Control System (VCS)

The following is not an exhaustive history; there have been dozens of VCSes built since the concept was invented, most obscure and proprietary. We'll concentrate here on the open-source VCSes that have led up to today's leading-edge systems, mostly ignoring those which in retrospect turned out to be false starts or dead ends. We will cover, though not necessarily in as much depth, some of the more influential closed-source systems as well. Some names of VCSes are normally capitalized and others are all-caps acronyms, several are by convention normally left in all lower case

The First Generation: The common trait of first generation of VCSes was that they were all file-oriented and centralized. Most were locking-based, with merging-based systems arriving towards the end of that era.

- **Prehistory of version control:** Version-control systems are loosely derived from change-management and logging systems used on mainframes in the 1960s; these connections are sketched in Walter Tichy's http://agave.garden.org/~aaronh/rcs/tichy1985rcs/html. It is worth noting that the very first VCS, while it later became closely associated with the Unix operating system, was originally written on an IBM System/370 computer running OS/MVT.
- **Source Code Control System (SCCS):** In the beginning was SCCS, the Source Code Control System. It was written in 1972 by Marc Rochkind, one of the early Unix developers at Bell Labs. SCCS was locking, file-oriented, and centralized; in fact, it had no support for remote (networked) repositories, a capability that was not even really imagined until the post-1982 rise of the Internet. Early SCCS was, like Unix itself, officially proprietary but de-facto open source until about 1985. A direct descendant of the original Bell Labs code is now open source in OpenSolaris; a slightly cleaned-up version of the Solaris code is available as this standalone project. There is one open-source clone, the written-from-scratch CSSC (Compatibly Stupid Source Control). CSSC is mainly intended to help SCCS users migrate to newer systems.
- **Revision Control System (RCS):** RCS, the Revision Control System, was the second VCS to be built and the oldest still in common use. It was written by Walter F. Tichy at Purdue University during the early 1980s. (The sources of rcs-5.7 include a note that RCS version 3 was released in 1983; its earlier history remains obscure.) Like SCCS, to which it was a direct response, RCS is locking, file-based, and centralized with no capability for network access. Unlike SCCS, RCS is still in fairly widespread use in 2008. RCS is lightweight and low-overhead compared to more recent and more capable but also more elaborate VCSes. The command interface, while primitive, is cleaner than that of SCCS. The document source for this survey itself was kept under RCS before being split into multiple files moved to Mercurial, and that is typical of its modern use cases; programmers often rely on it to keep

histories of single documents or small programs that they are maintaining single-handed.

- **DSEE/ClearCase:** DSEE (Domain Software Engineering Environment) was a VCS with significant SCM features including a build engine and a task tracker, first released in 1984 on Apollo Domain workstations. After Apollo was acquired by HP in 1989, DSEE continued to be used extensively within HP. The DSEE team was eventually allowed to spin off and became Atria; DSEE was rebranded as Clear Case. Later, Atria were acquired by IBM and continue to be in wide use in 2008 under the name Rational Clear Case. While DSEE/ClearCase are proprietary and closed source, they are interesting to examine because they represent a sort of apex of first-generation VCS design [10].

The Second Generation: In this generation main focus was on CVS and SVN.

- **Concurrent Versions System (CVS):** RCS predated ubiquitous networking and did not scale up well to collaborative development. These were becoming important around 1985 just as Tichy's RCS paper and production releases were becoming well-known. Dick Grune, a professor at the Free University of Amsterdam, wrote some script wrappers around RCS in 1984-1985 to address these problems. Two years later Brian Berliner turned these scripts into C programs which became the basis for later versions of what became known as the Concurrent Versions System.CVS began to see wide use around 1989-1990, but still in a local mode (client and repository on the same machine). It did not become really ubiquitous until after Jim Blandy and Karl Fogel (later two principals of the Subversion project) arranged the release of some patches developed at Cygnus Software by Jim Kingdon and others to make the CVS client software usable on the far end of a TCP/IP connection from the repository. This was in late 1994, just after the first big wave of Internet build out. Thereafter, CVS reigned more or less unchallenged in open-source development for about ten years, until Subversion 1.0 shipped in 2004.

CVS was file-oriented and centralized, like its predecessors, but broke new ground by being designed for collaborative development using a merging rather than locking-

based approach. Almost all subsequent systems have followed this lead, adopting CVS terminology and conventions and even the style of its command-line interface.

- **Subversion (SVN):** There were several attempts to rescue the CVS codebase and its basic architecture beginning in the late 1990s: Meta-CVS and CVSNT were the best known. All of these failed, neither solving CVS's architectural problems in a convincing way nor achieving significant mind-share among developers. A new approach was needed. In 2000, a group including some former CVS core developers launched Subversion, a project explicitly intended to improve on and supersede CVS. Subversion is also known as SVN, after the directory name its repositories conventionally use. While their first official production release did not arrive till 2004, betas from about 2002 onward were usable; in fact, it became clear the project was going to succeed at its major goals well before the 1.0 version shipped.

At time of writing in early 2008 there are actually more active SVN than CVS projects, and it is generally understood that CVS's days are numbered. However, in the longer term so are Subversion's. It is clearly the best of the centralized VCSes, but just as clearly the last of its kind. As dispersed, Internet-mediated software development becomes the norm rather than the exception; the centralized model of version control is running out of steam as visibly as the file-centric one did with CVS in the 1990s [10].

The Third Generation: Third-generation VCSes are natively decentralized. At time of writing there are many competing projects in this space; BitKeeper, git, monotone, darcs, Mercurial, and bzr, are the best known. All but BitKeeper are open-source.

- **The Arch family:** Arch (with BitKeeper) pioneered the diagnostic traits of third-generation VCSes. It was merging-based, change set-based, and fully decentralized. Tom Lord, the author of Arch, had a firm grasp on the concept of a “changeset” in 2002, and may have been the person who popularized the term (though BitKeeper had used it years earlier).Unfortunately, the Arch design became notorious for both brilliant ideas and a user interface that was poorly documented, complex, and difficult to work with. Even more unfortunately, Tom Lord shared all of these traits, both positive and negative.

Early sources were publicly available in January 2002, when Tom Lord responded to notice of his original implementation in shell (larch) on the Subversion mailing list. During the next three years, Lord's specification for the behavior of his system ("Arch 1.0") was re-implemented in a number of different languages. But by 2005, development on GNU Arch itself had effectively stalled out [10].

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before Commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before Merge	Bazaar, Git, Mercurial

Table1.5 Three Generations of Version Control [9]

Table1.5 shows the differences that exist in all three generations of version control system that is summarized as above.

- **Way of Showing Subversions**

The version control tree represents the software evolution history of the project development. Generally, version evolution has two ways: Serial evolution and Parallel evolution [11].

Every new version that made by the Serial evolution may be evolved from the present latest version. So, because of the evolution process of different versions to forms a simple linear chain, known as the version chain. In this way, version evolution is forward based on one to one mapping relations, in order to compensate for deficiencies, improve performance and adapt to the environment. Parallel evolution uses one to many mapping relation.

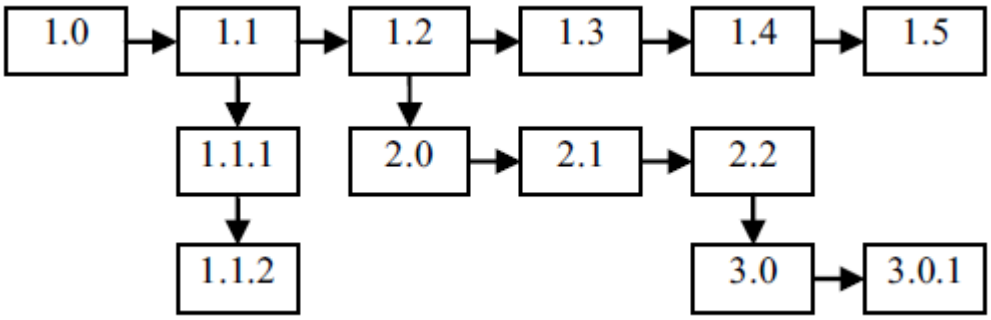


Figure1.5 Version Tree Sample Graph [11]

By work on both versions evolution way are usually related with each other, form the more common version map with branches, known as version tree. The version tree shows the evolution history of the project development.

In Figure 1.5, a total of three version chains, they are "1.0-1.1-1.2-1.3-1.4-1.5", "1.0-1.1-1.1.1-1.1.2" and "1.0-1.1-1.2-2.0-2.1-2.2-3.0-3.0.1." "1.0-1.1-1.2-1.3-1.4-1.5" is the serial evolution."1.1-1.2" and "1.1-1.1.1" is the parallel evolution.

1.3.2 Benefits of Version Control System

The following list presents some of the highlighted benefits of Version Control systems which have become a necessary condition for manage the OSS.

- **Single Centralized for all the code:** Version Control system is a central place to all related files of OSS project and these files are accessible for all team members involved in the project.
- **History and Author Tracking:** It contains all history a project which is under version control then anyone can have the possibility to look back to any of the older versions of project and make changes that have been made and identify each line of code by name of the author.
- **Management of Different Development and Update Lines:** It maintains all development and updating files of the projects under version control, e.g. different product versions or versions for different operating systems, then Version Control system organizes and helps in managing these different lines for the programmers, and allows them to include code parts from one line of development to another. It contains all require information of developers and its updating work.
- **Allow Parallel Development:** Many version control systems allow the feature of parallel development. Thus it support two or more than two developers to work on

the same code at the same time and finally changes have been merged automatically by the VCS with minimum efforts from the developers involved.

- **Less Network Usage:** Many Version Control Systems do not send the complete project files on the network, but only small repository files, which are then used to redeveloped the particular version at the user's side. This saves bandwidth and makes the work with such systems faster if one deals with many or large files of several megabytes [12].

1.4 Objective of Thesis

OSS is threat to closed source software development for commercial use because if same kind of software available in free of cost why person expend their money to bought. OSS has become the part of both research and industry field. The communication is necessary part in development of OSS. This becomes the plus point in development process because developers are free to work hence source code is written with more attention and addition of new features. Hence OSS movement has motivated to research work and for develop new kind of applications. This thesis work is use to increase the research work and merged the resulted information of empirical study that are focused on a relatively narrow set of open source software properties. The objective of this thesis work is to show an area of measuring the open source software evolution with the help of the version control system data. The version control system is use to measure the undertaken data and then considers understanding the process of work that is completed in terms of development process and maintain that process also.

The main aim of this thesis is on the measuring the OSS project outcomes and consider the differences occur in the each version of project process. OSS Tools that is available on internet and method have been used in the measurement of the source code across different versions of the software project. The scope of this thesis is to study the selected open source software evolution on the behalf of some set of software metrics by using version control system and resulted work of study satisfied the Lehman’s law of evolution in open source software field.

Chapter 2
LITERATURE REVIEW

This chapter includes all the description of theoretical aspect of this thesis study. This chapter shows all the information which has collected from various research papers related to this topic that contains methods, law and the information about the technical supported data.

Open Source Software (OSS) has recently become the subject of scrutiny and debate, not least for the promise it holds to solve systems development challenges. Over the past thirty years, researchers have proposed hypotheses on how software changes, and provided evidence that both supports and refutes these hypotheses [13]. In the words of Lehman, "Evolution is an essential property of real-world software" and "As your needs change, your criteria for satisfaction changes". Over time requirements will change, and software must evolve to continue to satisfy these new requirements. If the environment is the one that drives the evolution, programmers are the ones who evolve the program [14].

This literature review is use to determine the software evolution area in the field of Open Source Software Development and its result support the need for the making some more related metrics to measure the evolution of OSS [15].

2.1 Open Source Software

Free software (FS), term given by Richard Stallman, introduced in 1984, can be obtained at zero cost i.e. software which gives the user certain freedoms. Open Source Software (OSS), term coined by Eric Raymond, in 1998, is software for which the source code is freely and publicly available, though the specific licensing agreements vary as to what one is allowed to do with that code . In the case of FS, only executable file is made available to the end user, through public domain and end user is free to use that executable software in any way, but the user is not free to modify that software. The alternative term Free/Libre and Open Source Software (FLOSS) refers to software whose licenses give users four essential 'freedoms':

- To run the program for any purpose,
- To study the workings of the program, and modify the program to suit specific needs,
- To redistribute copies of the program at no charge or for a fee, and
- To improve the program, and release the improved, modified version (Perens, 1999; 2004) [16].

The Free and Open Source Software (FOSS) movement is one such development that is playing out before us today. As an innovation force, the representative of the free and open source software, Linux challenges Windows which represents proprietary software. Free and open source movement expedites technology and knowledge development in public. In addition, it is a reliable force to fight against monopolization as well as inspire creativity in software industry [1]. In recent years, open source development has become an important force in the software engineering discipline. Software engineers are increasingly expected to be able to evaluate open source solutions, integrate with open source products or libraries, and contribute fixes and enhancements to open source projects [7]. The basic attraction of open source software development area is open source code which is freely available to read, modify or redistribute by any of the interested user. For motivation of an individual in open source software the adopted license has crucial role [17].

There are a lot of definitions for the concept of open source software development, according to OSD software which has ability to distribute freely with available source code through the Internet and using unpaid people that can modify the code freely, is open source software.

Some major concepts to identify open source software are as follow:

- Distributed software.
- Free software.
- Available source code.
- Communicate through internet.
- Developers are users.
- Unpaid and large amount volunteers [5].

2.1.1 Benefits of Open Source Software

The benefits with OSS (Feller & Fitzgerald, 2001; FLOSS Project Report, 2002) are as follows:

- Collaborative, parallel development involving source code sharing and reuse
- Collaborative approach to problem solving through constant feedback and peer review
- Large pool of globally dispersed, highly talented, motivated professionals
- Extremely rapid release times
- Increased user involvement as users are viewed as co-developers
- Quality software
- Access to existing code [15].

2.1.2 Disadvantages of Open Source Software

- In the rapid development environment, the result could be slower, given the absence of formal management structures (Bezroukov, 1999; Levesque, 2004; Valloppillil, 1998).
- Strong user involvement and participation throughout a project is becoming problematic as users tend to create bureaucracies which hamper development (Bezroukov, 1999).
- OSS is premised on rapid releases and typically has many more iterations than commercial software. This creates a management problem as a new release needs to be implemented in order for an organization to receive the full benefit (Farber, 2004).
- The user interfaces of open source products are not very intuitive (Levesque, 2004; Valloppillil, 1998; Wheatley, 2004).
- As there is no single source of information as well as no help desk therefore no ‘definitive’ answers to problems can be found (Bezroukov, 1999; Levesque, 2004).
- System deployment and training is often more expensive with OSS as it is less intuitive and does not have the usability advantages of proprietary software [15].

Open Source Software Development

Open source life cycle for OSSD paradigm demonstrates several common attributes like parallel development and peer review, prompt feedback to user and developer contributions, highly talented developers, parallel debugging, user involvement, and rapid release times [15].

Vixie (1999) holds that an open source project can include all the elements of a traditional SDLC. Classic OSS projects such as BSD, BIND and SendMail are evidence that open source projects utilize standard software engineering processes of analysis, design, implementation and support.

Mockus, Fielding & Herbsleb (2000) describe a life cycle that combines a decision-making framework with task-related project phases. The model comprises six phases like roles and responsibilities, identifying work to be done, assigning and performing development work, pre-release testing, inspections, and managing releases.

Jorgensen (2001) provides a more detailed description of specific product related activities that support the OSSD process. This model explains the life cycle for changes that occurred within the Free BSD project [15].

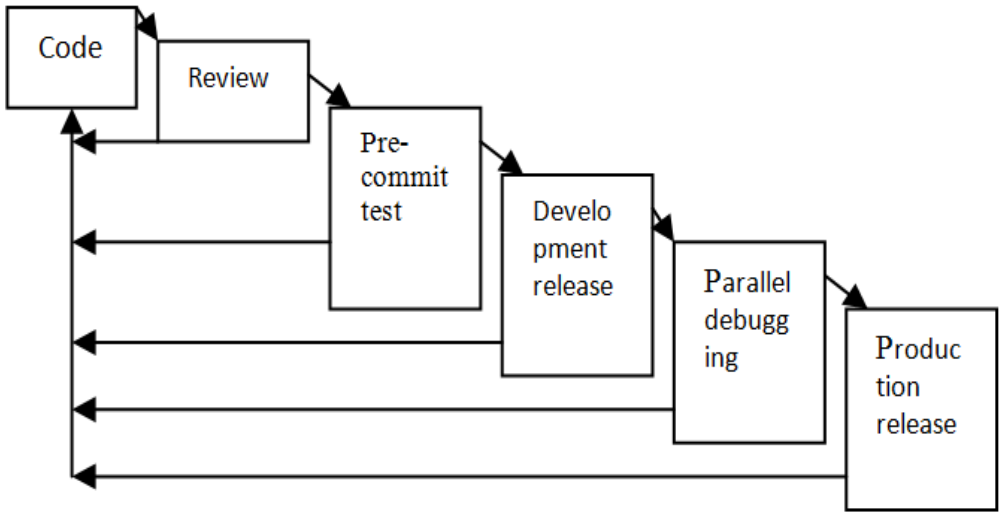


Figure2.1 Jorgensen Life-Cycle, 2001 [15]

Jorgensen's model is widely accepted **(Feller & Fitzgerald, 2001; FLOSS Project Report, 2002)** as a framework for the OSSD process, on both macro (project) and micro (component or code segment) levels. However, flaws remain. When applied to

an OSS project, the model does not adequately explain where or how the processes of planning, analysis and design take place.

Schweik and Semenov (2003) propose an OSSD project life cycle comprising three phases: project initiation; going 'open'; and project growth, stability or decline. Each phase is characterized by a distinct set of activities.

Wynn (2004) proposes a similar open source life cycle but introduces a maturity phase in which a project reaches critical mass in terms of the numbers of users and developers it can support due to administrative constraints and the size of the project itself.

Roets, et al. (2007) expands Jorgensen life-cycle model and incorporates aspects of previous models, particularly that of **Schweik and Semenov (2003)**. In addition, this model attempts to encapsulate the phases of the traditional SDLC (Figure 2.2). This model facilitates OSS development in terms of improved programming skills, availability of expertise and model code as well as software cost reduction [15].

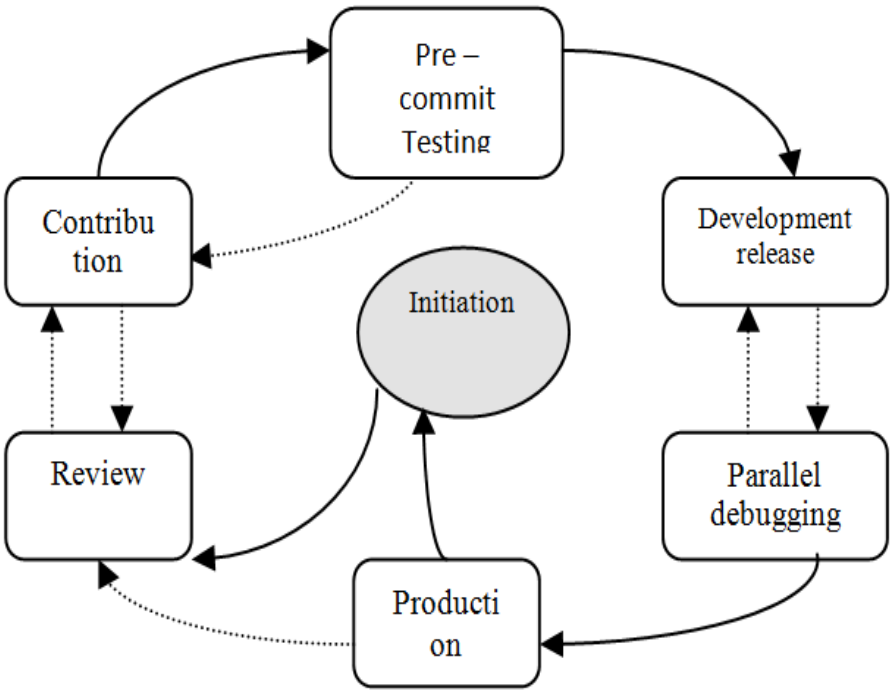


Figure2.2: Roets, et al. life-cycle model of OSSD projects, 2007 [20]

2.2 Software Evolution

Software evolution is the term used in software engineering (specifically software maintenance) to refer to the process of developing software initially, then repeatedly updating it for various reasons. The role of software process improvement, innovation and technology transfer may also be considered as factors that may contribute to software evolution [6].

The practice of software maintenance requires changing the design of the system: by fixing bugs so that the design is correct, by adapting the system for use in new environments, by adding new features, or by improving the internal design so that the system is easier to manage and change. It is innovation, not preservation that drives software change: a new system, better adapted to its environment, evolves from the old one. It shows software evolution process [18]. As Lehman pointed out, their work represents that of a small group that views evolution as a noun and focuses on the nature of software evolution and the properties of the evolution phenomenon. Other groups of researchers view evolution as a verb. They focus on methods and tools to facilitate software evolution and the tasks that implement it [12].

E-type programs that is, for systems actively used and embedded in a real world domain. Once such systems are operational, they are judged by the results they deliver. Their properties include loosely expressed expectations that, at least for the moment, stakeholders are satisfied with the system as is. In addition to functionality, factors such as quality (however defined), behaviour in execution, performance, ease of use, changeability and so on will be of concern. In this they differ from S-type programs where the sole criterion of acceptability is correctness, in the mathematical sense. Only that which is explicitly included in the specification or follows from what is so included is of concern in assessing (and accepting) the program and the results of its execution. An S-type program is completely defined by and is required to be correct with respect to a fixed and consistent specification [19].

Lee, Yang and Auburn (2007) provide an overview of open source software evolution with software metrics, and present various metrics that can be used during the software evolution process to understand the evolution behaviour [20]. Israel Herraiz (2009) suggests using SLOC, complexity metrics to characterize a software product and its evolution. Breivold, Chauhan and Ali Babar (2010) examined the metrics used for assessing OSS evolution, and defined various type of metrics [21].

2.3 Software Metric and its Classification

Software metric is a measure of some property of a piece of software or its specifications. Software metrics measure different aspects of software complexity and therefore play an important role in analyzing and improving software quality [22]. Software Metrics are standards to determine the size of an attribute of a software product and a way to evaluate it [19].

Advantages of Software Metrics:

- In Comparative study of various design methodology of software systems.
- For analysis, comparison and critical study of various programming language with respect to their characteristics
- In comparing and evaluating capabilities and productivity of people involved in software development.
- In the preparation of software quality specifications.
- In the verification of compliance of software systems requirements and specifications.
- In making inference about the effort to be put in the design and development of the software systems.
- In getting an idea about the complexity of the code.
- In taking decisions regarding further division of complex module is to be done or not.
- In providing guidance to resource manager for their proper utilization.
- In comparison and making design tradeoffs between software development and maintenance cost.
- In providing feedback to software managers about the progress and quality during various phases of software development life cycle.
- In allocation of testing resources for testing the code [22].

Limitations of Software Metrics:

- The application of software metrics is not always easy and in some cases it is difficult and costly.

- The verification and justification of software metrics is based on historical/empirical data whose validity is difficult to verify.
- These are useful for managing the software products but not for evaluating performance of the technical staff.
- The definition and derivation of Software metrics is generally based on assuming which are not standardized and may depend upon tools available and working environment.
- Most of the predictive models rely on estimates of certain variables which are often not known exactly [22].

Classification of Software Metrics

Process Metrics: Process metrics are known as management metrics and used to measure the properties of the process which is used to obtain the software. Process metrics include the cost metrics, efforts metrics, and advancement metrics and reuse metrics. Process metrics help in predicting the size of final system & determining whether a project on running according to the schedule.

Products Metrics: Product metrics are also known as quality metrics and is used to measure the properties of the software. Product metrics includes product non reliability metrics, functionality metrics, performance metrics, usability metrics, cost metrics, size metrics, complexity metrics and style metrics. Products metrics help in improving the quality of different system component & comparisons between existing systems [22].

Examples of Software Metrics

1. Size Metrics

Some of metrics which are used to attempt the quantify software size. :

Lines of Code. LOC is the most frequently used size metric for size of source code. It can be easily and precisely describe. The definition of LOC is to count any line that is not a blank or comment line, regardless of the number of statements per line in program.

Function Points. Function points are used to be a measure the size of written program and thus, define the effort required for development [23].

2. Complexity Metrics

Different metrics have been suggested for measuring complexity of source code [23]. are as followed below.

Cyclomatic Complexity. McCabe (1976) proposed a complexity metric known as Cyclomatic Complexity. According to McCabe Cyclomatic Complexity can be used as a measure the complexity of program and thus, represent a way to program development and testing.

Information Flow - The information flow within a program structure may also be used as software metric for complexity of program.

3. Quality Metrics

Some of the quality Metrics by Boehm, McCall and others are discussed below: [24].

Defect Metrics. Defects metrics refers to measures in a software product is a metric of software quality.

Reliability Metrics. Reliability shows the probability of software failure, or the rate at which the software failure will happen. Mean Time to Failure (MTTF) is reliability metric.

Maintainability Metrics. According to empirical study done by Rombach, software complexity metrics can be used to define the maintainability of software.

2.4 Description of Software Resources Used

Version Control System

Version control systems have become indispensable software engineering tools. They enable a developer to work on a private copy of a jointly developed project and provide support for merging her changes with the contributions of other team members later on. As these systems track the history of a project, they contain a wealth of information that can be leveraged by other software engineering tools. Version control is the important functions of software configuration management, is responsible for all the elements in the configuration database automatically assigned version of the logo, and to ensure that version of the name of the uniqueness [25].

Version of a system is a system example, in a way different from the practical examples of other systems. The new version of the system may have different functions, performance, may modify the system error. Some versions may have no difference in function, just design for different hardware or software configuration. If only slight differences between versions, and sometimes a version is called a variant of another version.

A system release version is the version to be distributed to customers. Each system release version should include some new functions, or for different hardware platforms. Versions of a system are far more than the release versions, because the institution's internal versions are created for internal development or testing, and some of the internal versions will not be released to the customer [25].

The Repository: At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a file system tree a hierarchy of files and directories. Any number of clients connects to the repository, and then read or writes to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others. Figure 2.3, "A typical client/server system" illustrates this.

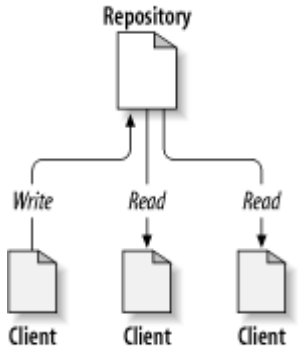


Figure2.3 A typical client/server system [26]

When a client reads data from the repository, it normally sees only the latest version of the file system tree [26].

Subversion Repositories: Subversion implements the concept of a version control repository much as any other modern version control system would. Unlike a working copy, a Subversion repository is an abstract entity, able to be operated upon almost exclusively by Subversion's own libraries and tools.

Revisions: A Subversion client commits (that is, communicates the changes made to) any number of files and directories as a single atomic transaction. By atomic transaction, we mean simply this: either all of the changes are accepted into the

repository, or none of them is. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions [26].

Each time the repository accepts a commit; this creates a new state of the file system tree, called a revision. Each revision is assigned a unique natural number, one greater than the number assigned to the previous revision. The initial revision of a freshly created repository is numbered 0 and consists of nothing but an empty root directory.

Figure 2.4, "Tree changes over time" illustrates a nice way to visualize the repository. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a file system tree hanging below it, and each tree is a "snapshot" of the way the repository looked after a commit.

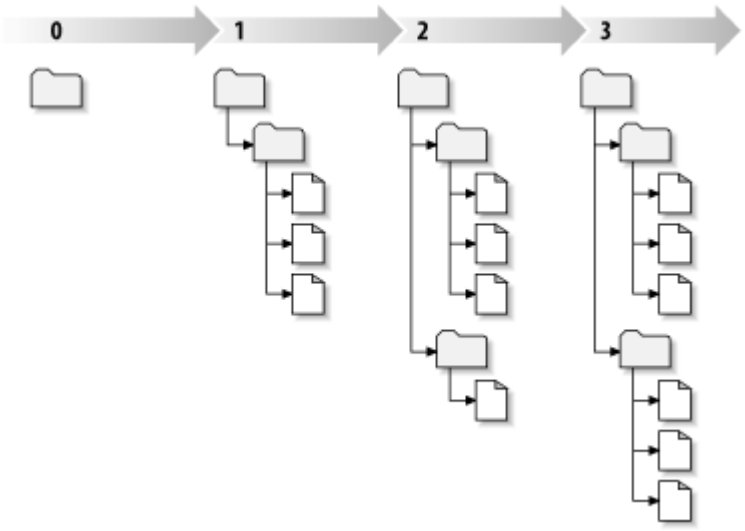


Figure2.4 Tree changes over time [26]

Source Code: The certain aspects of open source development have pedagogical benefit in a computer science or software engineering program. The study of open source projects exposes students to large bodies of code and tools for build automation, version control, and project management. Most importantly, students experience the social dynamics of working on a large project [7].

This kind of study will become more beneficial more researchers. The source code analysis method also helps to derive the evaluation of complexity of program [23].

In open source software environment source code is easily available with required related files.

2.5 Description of Software Tools Used

To support the software artefacts which are included in software tools that are used to done the process work due to the heavy volume of data processing, and there are such open source software tools available of both kind free and non-free forms. Small subsets of that kind of tools which are used in this thesis work are described here.

1) Function Point Modeler
2) SourceMonitor

1) Function Point Modeler™ Enterprise Edition is the only product on the market today which not only counts or estimates software but also manages the whole IT-Metrics (Project, Product and Process Metrics) centralized in used company.

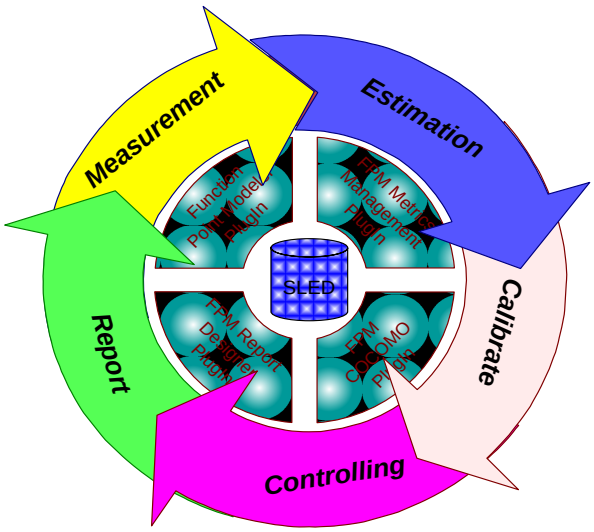


Figure2.5 Function Point Modeler [27]

The elements of Figure2.5 are as follows:

- **Function Point Modeler™ (FPM)** conforms to the IFPUG Counting Practices Manual (CPM),
- **FPM Metrics Management™** is a metrics management tool with Software Life Cycle Experience Database (SLED) to manage whole metrics in the used company,

- **Function Point Modeler COCOMO II ™** conforms to the COCOMO II. Project estimates and factor calibrations are based on input data in the SLED.
- **Function Point Modeler Report Designer™** is a powerful report designer tool which allows creating very complex reports from the SLED to meet nearly all business requirements in used company.

These four inner major parts of Function Point Modeler are help in measurement, estimation, calibrate the measured data, controlling and reporting to the company for its project calculation.

Function Point Modeler™ conforms to the IFPUG CPM. **Function Point Modeler™** is designed by Certified Function Point Specialists to meet all project function point measurement requirements of a Function Point Specialists.

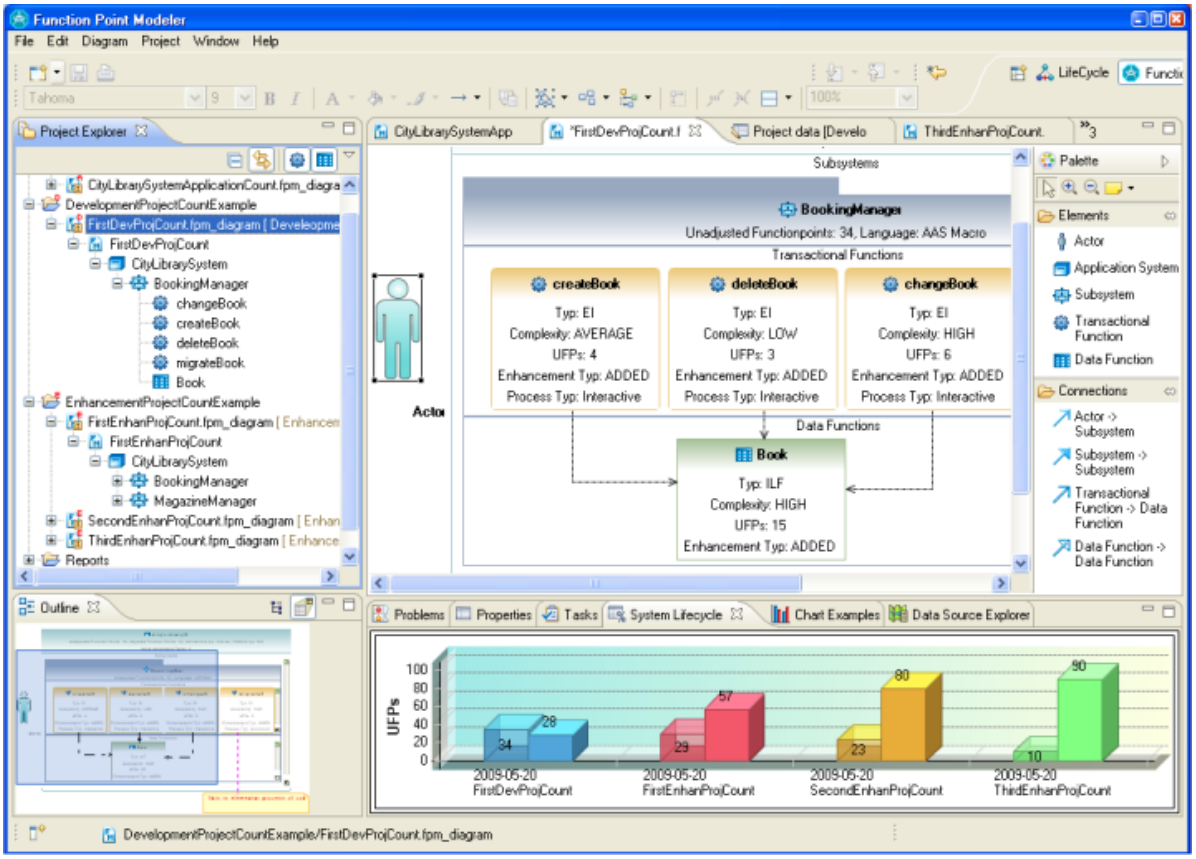
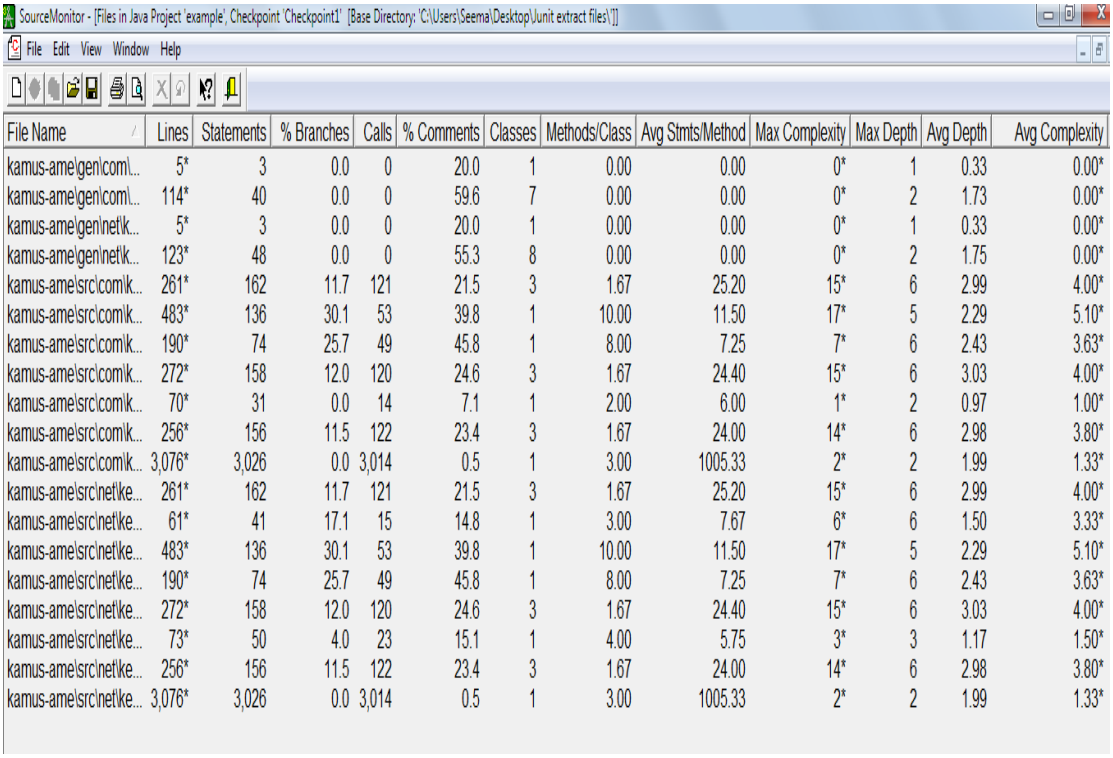


Figure2.6 Function Point Modeler Perspective [27]

This Figure2.6 shows that a developer who is showing as actor develop a application Booking Manager and below there are in its subsystem some transactional function such as change book, delete book etc. In the subsystem there is also data function called book which consist inter logical file. It also shows the unadjusted function point in the graphical form.

2) Source Monitor

The free available SourceMonitor is used to look inside the source code of software product for identify the relative complexity of input modules. SourceMonitor, available in C++, that runs through input code at well speed. SourceMonitor provides software metrics in a fast, single pass through source files. It also measures metrics according to input kind of source code written in C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6) or HTML. It also saves measured metrics in checkpoints for comparison during different software development projects. The program displays and prints metrics in tables and charts and operates within a standard Windows GUI or inside your scripts using XML command files. It also exports metrics to XML or CSV files for further processing with other tools [27].



File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complexity
kamus-ame\gen\com\...	5*	3	0.0	0	20.0	1	0.00	0.00	0*	1	0.33	0.00*
kamus-ame\gen\com\...	114*	40	0.0	0	59.6	7	0.00	0.00	0*	2	1.73	0.00*
kamus-ame\gen\net\k...	5*	3	0.0	0	20.0	1	0.00	0.00	0*	1	0.33	0.00*
kamus-ame\gen\net\k...	123*	48	0.0	0	55.3	8	0.00	0.00	0*	2	1.75	0.00*
kamus-ame\src\com\k...	261*	162	11.7	121	21.5	3	1.67	25.20	15*	6	2.99	4.00*
kamus-ame\src\com\k...	483*	136	30.1	53	39.8	1	10.00	11.50	17*	5	2.29	5.10*
kamus-ame\src\com\k...	190*	74	25.7	49	45.8	1	8.00	7.25	7*	6	2.43	3.63*
kamus-ame\src\com\k...	272*	158	12.0	120	24.6	3	1.67	24.40	15*	6	3.03	4.00*
kamus-ame\src\com\k...	70*	31	0.0	14	7.1	1	2.00	6.00	1*	2	0.97	1.00*
kamus-ame\src\com\k...	256*	156	11.5	122	23.4	3	1.67	24.00	14*	6	2.98	3.80*
kamus-ame\src\com\k...	3,076*	3,026	0.0	3,014	0.5	1	3.00	1005.33	2*	2	1.99	1.33*
kamus-ame\src\net\ke...	261*	162	11.7	121	21.5	3	1.67	25.20	15*	6	2.99	4.00*
kamus-ame\src\net\ke...	61*	41	17.1	15	14.8	1	3.00	7.67	6*	6	1.50	3.33*
kamus-ame\src\net\ke...	483*	136	30.1	53	39.8	1	10.00	11.50	17*	5	2.29	5.10*
kamus-ame\src\net\ke...	190*	74	25.7	49	45.8	1	8.00	7.25	7*	6	2.43	3.63*
kamus-ame\src\net\ke...	272*	158	12.0	120	24.6	3	1.67	24.40	15*	6	3.03	4.00*
kamus-ame\src\net\ke...	73*	50	4.0	23	15.1	1	4.00	5.75	3*	3	1.17	1.50*
kamus-ame\src\net\ke...	256*	156	11.5	122	23.4	3	1.67	24.00	14*	6	2.98	3.80*
kamus-ame\src\net\ke...	3,076*	3,026	0.0	3,014	0.5	1	3.00	1005.33	2*	2	1.99	1.33*

Figure2.7 SourceMonitor showing metrics value of a program.

Figure2.7 is showing the calculated values of different metrics by analyzing a program. This calculation done on the basis of input source code then SourceMonitor itself find software metrics such as lines, statement, percentage of branches etc.

Chapter 3

PROBLEM STATEMENT

This chapter contains problem statement of this thesis which is much in discussion currently in open source software field.

3.1 Problem Definition

The availability of source code increases the development as well as evolution of Open Source Software (OSS). OSS motivates the programmer's ability for increase the quality of original Software.

This thesis work included the study of changes in the values of software metrics over the different versions of selected open source software. By using software evolution discipline here measuring the metrics has been reviewed. The important work done by Lehman and others deliver to describe some fundamental governing principles of software evolution, but it is not widely acceptable, in the opinion of some other authors, software evolution has been the victim of a general paucity of strongly empirical studies. In this thesis the literature review has find out that software evolution research has distributed and that studies more recently have classified into measuring the subject from the different perspective of the software development paradigm. These viewpoints consider the different open source software examples, and when this above idea moved to works on open source software evolution, it was presented how this particular software example contains evolutionary characteristics specific to itself. It has therefore been demonstrated that open source software evolution is a viable and promising way of investigation, and furthermore that the reports of past authors suggest that studies should consider a suitable realistic projects from which to draw their observations. The software evolution process can be well described using the different kind of Metrics and visualizing these metrics approach to get the relevant result. The problem here is to find the subset of metrics to measure size and complexity of the realistic software project. This process supports Lehman's laws of software evolution by using the measurement of the open source software metrics.

The research of open source software evaluation has been majorly completed on the basis of development size and maintainability examples but here in this thesis work

the evolution is studied on various metrics examples like size, effort and complexity metrics. Software metrics have been studied with great attention and examined for several different features of the Open Source System.

By the literature survey it found that software evolution is mostly done in the closed source software as compared to open source software hence here software evolution will do on open source software. This thesis study suggested different number of software evaluation of size and effort in development metrics for open source system and subsystem for a real time open source project called JUnit. For achieving the result some kind of technical support is required. This technical support depends on the repositories data and open source software tools which are required for software evolution process.

Chapter 4

Software Evolution Process in Open Source Software

This chapter includes case study of selected open source software and calculate the software metrics of different version of that software with the help of different software resources.

4.1Software Evolution Process

The Open Source Software evolution components are depend on the ability of users or programmers to freely access the source code to make changes and enhance the quality of that original software source code.

4.1.1 Technical Software Support in Evolution

This thesis working process is based on open source software evolution. First select open source software then with the help of version control system collect all required data for evolution. Version control and bug tracking systems contain large amounts of historical information that can give deep insight into the evolution of a software project. Version information may be enhanced with data from bug tracking systems that report about past maintenance activities. Both information sources together lead to an extensive database that enables reasoning about the past and anticipating future evolution of software projects.

The SVN repository play most important role in software evolution because all necessary data is present in the subversion repositories. Internet is the basis of this SVN repositories creation and it is also an interface between the user and the virtual resource of SVN repository system. Figure4.1 shows the way of displaying subversion data of selected software JUnit on internet. It's contained all related information such as last update date, size and downloads files. This repository data is help a lot in software evolution process.

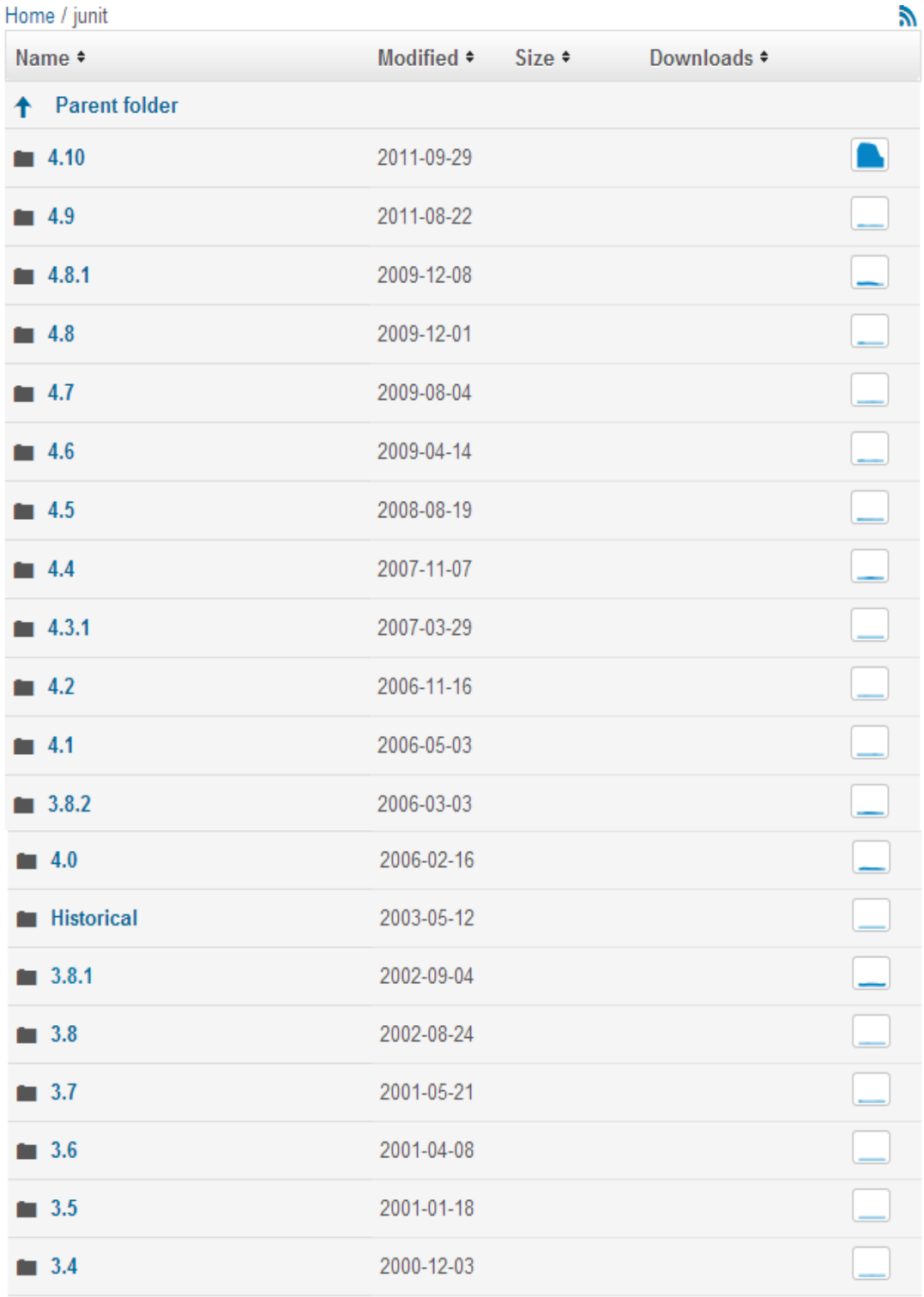


Figure4.1 Image of 'Sourceforge' site for version of JUnit [28]

The core tools used to study the data in the SVN repository are the following:

- **Function Point Modeler**

Function Point Modeler is a new generation tool for Function Point Analysis to measure Software. It is an International Function Point User Group (IFPUG)

Counting Practice Manual (CPM) conform tool, designed by Certified Function Point Specialists to meet all requirements in our daily work as Function Point Specialists. It supports the life-cycle of elementary processes and application systems and tracks its functional changes.

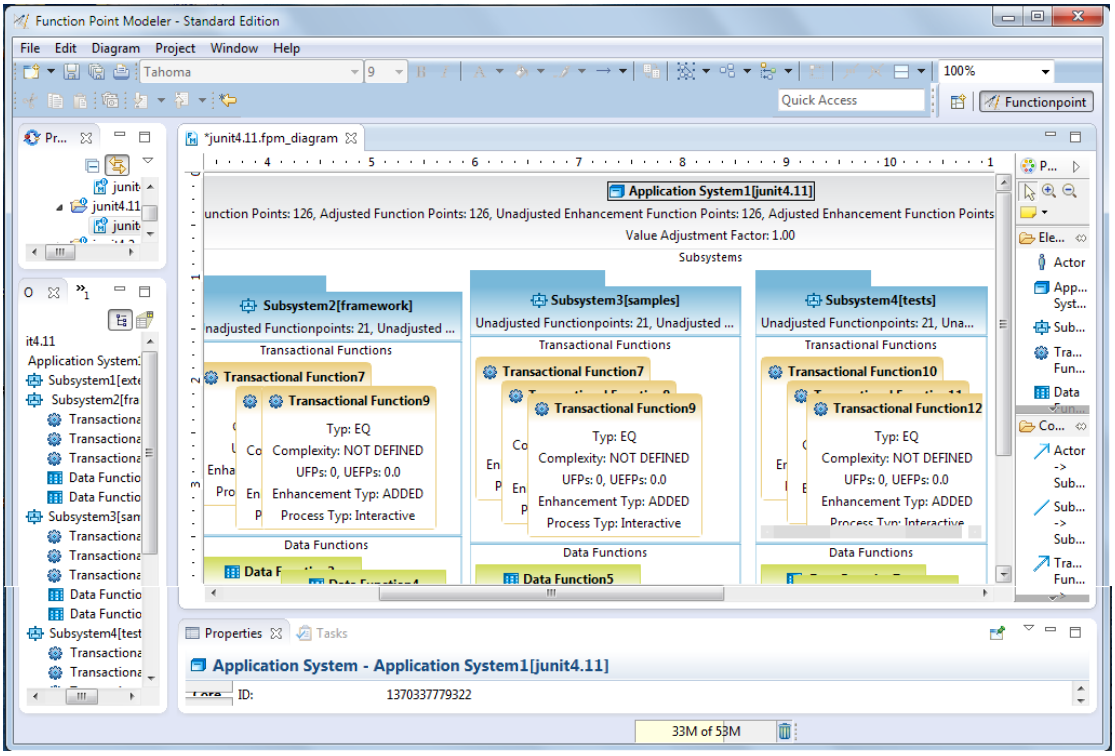
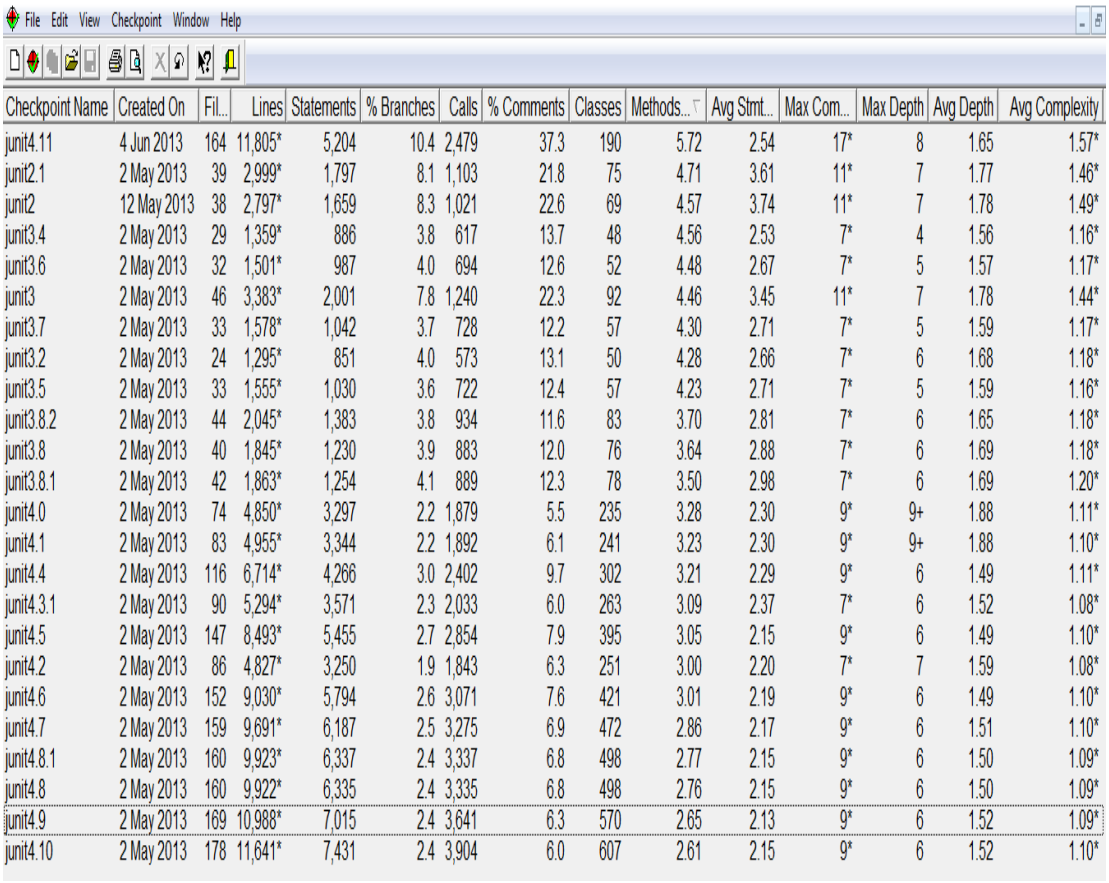


Figure4.2 Shows calculated value JUnit by Function Point Modeler

This Figure4.2 shows the output of JUnit4.11 by Function Point Modeler. It shows the main application system of JUnit4.11 then its subsystems which contain different number of transactional functions and data functions, that is used to calculate unadjusted function point and adjusted function point.

- **SourceMonitor**

Source Monitor measures metrics in a fast, single pass manner through source files. It also examine metrics for source code written in C++, C, C #, VB.NET, Java, Delphi, Visual Basic (VB6) or HTML.



Checkpoint Name	Created On	Fil...	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods...	Avg Stmt...	Max Com...	Max Depth	Avg Depth	Avg Complexity
junit4.11	4 Jun 2013	164	11,805*	5,204	10.4	2,479	37.3	190	5.72	2.54	17*	8	1.65	1.57*
junit2.1	2 May 2013	39	2,999*	1,797	8.1	1,103	21.8	75	4.71	3.61	11*	7	1.77	1.46*
junit2	12 May 2013	38	2,797*	1,659	8.3	1,021	22.6	69	4.57	3.74	11*	7	1.78	1.49*
junit3.4	2 May 2013	29	1,359*	886	3.8	617	13.7	48	4.56	2.53	7*	4	1.56	1.16*
junit3.6	2 May 2013	32	1,501*	987	4.0	694	12.6	52	4.48	2.67	7*	5	1.57	1.17*
junit3	2 May 2013	46	3,383*	2,001	7.8	1,240	22.3	92	4.46	3.45	11*	7	1.78	1.44*
junit3.7	2 May 2013	33	1,578*	1,042	3.7	728	12.2	57	4.30	2.71	7*	5	1.59	1.17*
junit3.2	2 May 2013	24	1,295*	851	4.0	573	13.1	50	4.28	2.66	7*	6	1.68	1.18*
junit3.5	2 May 2013	33	1,555*	1,030	3.6	722	12.4	57	4.23	2.71	7*	5	1.59	1.16*
junit3.8.2	2 May 2013	44	2,045*	1,383	3.8	934	11.6	83	3.70	2.81	7*	6	1.65	1.18*
junit3.8	2 May 2013	40	1,845*	1,230	3.9	883	12.0	76	3.64	2.88	7*	6	1.69	1.18*
junit3.8.1	2 May 2013	42	1,863*	1,254	4.1	889	12.3	78	3.50	2.98	7*	6	1.69	1.20*
junit4.0	2 May 2013	74	4,850*	3,297	2.2	1,879	5.5	235	3.28	2.30	9*	9+	1.88	1.11*
junit4.1	2 May 2013	83	4,955*	3,344	2.2	1,892	6.1	241	3.23	2.30	9*	9+	1.88	1.10*
junit4.4	2 May 2013	116	6,714*	4,266	3.0	2,402	9.7	302	3.21	2.29	9*	6	1.49	1.11*
junit4.3.1	2 May 2013	90	5,294*	3,571	2.3	2,033	6.0	263	3.09	2.37	7*	6	1.52	1.08*
junit4.5	2 May 2013	147	8,493*	5,455	2.7	2,854	7.9	395	3.05	2.15	9*	6	1.49	1.10*
junit4.2	2 May 2013	86	4,827*	3,250	1.9	1,843	6.3	251	3.00	2.20	7*	7	1.59	1.08*
junit4.6	2 May 2013	152	9,030*	5,794	2.6	3,071	7.6	421	3.01	2.19	9*	6	1.49	1.10*
junit4.7	2 May 2013	159	9,691*	6,187	2.5	3,275	6.9	472	2.86	2.17	9*	6	1.51	1.10*
junit4.8.1	2 May 2013	160	9,923*	6,337	2.4	3,337	6.8	498	2.77	2.15	9*	6	1.50	1.09*
junit4.8	2 May 2013	160	9,922*	6,335	2.4	3,335	6.8	498	2.76	2.15	9*	6	1.50	1.09*
junit4.9	2 May 2013	169	10,988*	7,015	2.4	3,641	6.3	570	2.65	2.13	9*	6	1.52	1.09*
junit4.10	2 May 2013	178	11,641*	7,431	2.4	3,904	6.0	607	2.61	2.15	9*	6	1.52	1.10*

Figure4.3 Calculated value of JUnit by SourceMonitor

Figure4.3 showing the value of different metrics of JUnit like as number of files, number of lines, number of statements etc.

4.1.2 Selected Metrics

This part of thesis is use to demonstrating the definition of various size metrics used in the study. The metrics used to study the data under the SVN repository are the following:

Statements: In Java, computational statements are terminated with a semicolon character. Branches such as if, for and while are also counted as statements. All attributes are counted as statements as well, though calls inside attributes are ignored. The exception controls statements try, catch, and finally are also counted as statements.

Percent Branch Statements: Statements that cause a break in the sequential execution of statements are counted separately. These are the following: if, else, for,

do, while, break, continue, switch, case and default. The exception block statements try, catch and finally are also counted as branch statements, as are throw statements.

Method Call Statements: All method calls are counted, in statements as well as in logical expressions because in method calls there are some statements are present and these statements may contain some logical expressions.

Average Statements per Method: The total number of statements found inside of methods found in a program.

Function Point Count:

• The size of a large software product can be estimated in better way through a larger unit called module. A module can be defined as segment of code which may be compiled independently.

• For example, let a software product require n modules. It is generally agreed that the size of module should be about 50-60 line of code. Therefore size estimate of this Software product is about n x 60 line of code where n is number of modules.

Number OF Files: Number of Files (NOF) is used to show that how many source files are required to build the software.

4.1.3 Steps of Software Evolution Study

Software evolution process can understand as the empirical study of changes of software system on the behalf of selected software metrics.

1. First bound the study area

The OSS is a very wide area for study work; it can make some kind of confusion. Thus first define the field in which the evolution study will processed. In this thesis the study area is size of different version and subversion of real time open source software.

2. Decide the metrics used in study

Once the software evaluation field has decided one should select the related software metrics of that field. That metrics further used to evaluate the software and result is completely depend on these selected metrics. In this thesis study work the some set of size metrics have decided.

3. Find realistic open source software product

After decided the set of metrics now it’s time to select the real time open source software that will evaluate. In this thesis after exploring a lot of open source software

the study finalized the JUnit. JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which are collectively known as xUnit that originated with SUnit.

4. Use the all required set of software tools.

For software evolution process there is requirement of technical support that is available in the form of software tools. These tools must be related to decided metrics because these tools give the evaluated result. In this thesis the function point modeler, Source monitor and some SVN repositories have used.

5. Show the results in well understanding manner.

This section include that how the resulted data will show by that it will easily understand by reader. Thus in thesis result are shown in the form of graphs [15].

4.2 Study of Software Evolution on JUnit

JUnit is a unit testing framework for the Java Programming Language. It is important in the test driven development, and is one of a family of unit testing frameworks collectively known as xUnit.

4.2.1 Introduction of JUnit

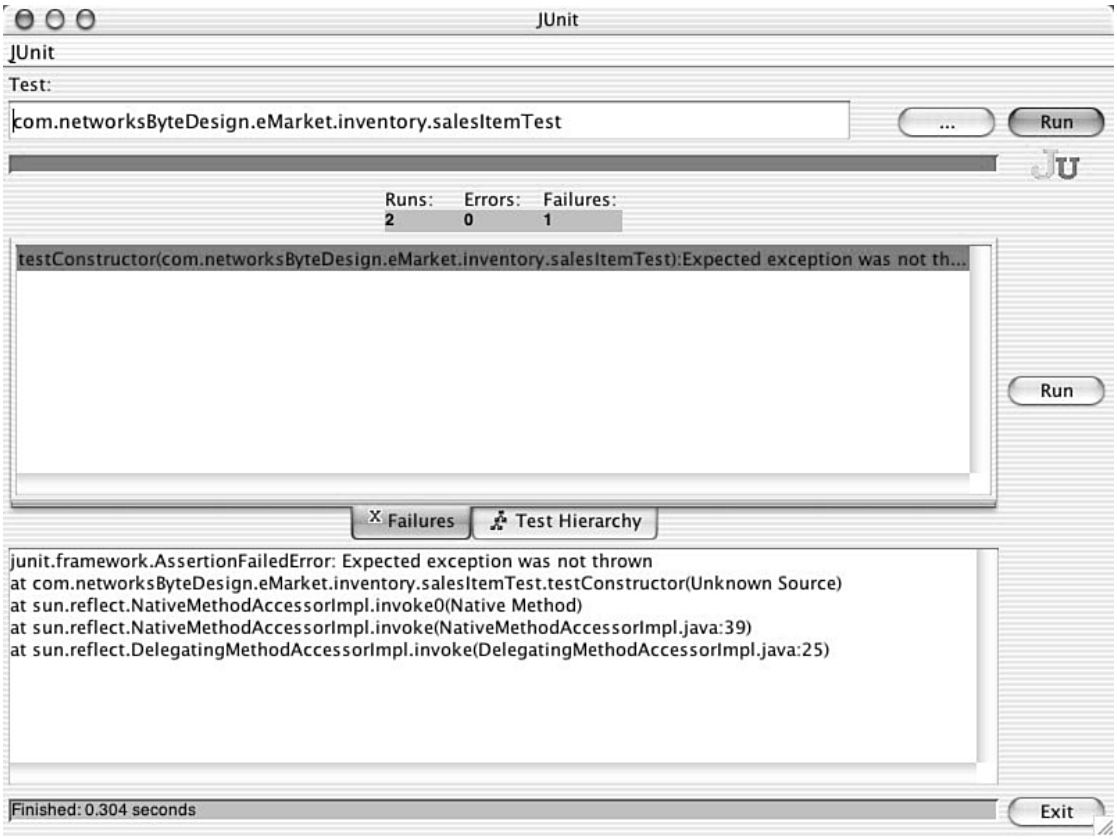


Figure4.4 JUnit software [28]

JUnit promotes the idea of "first testing then coding", which emphasis on setting up the test data for a piece of code which can be tested first and then can be implemented. This approach is like "test a little, code a little, test a little, code a little..." which increases programmer productivity and stability of program code that reduces programmer stress and the time spent on debugging [29].

Features

Some basic feature of JUnit are define in these steps:

- JUnit is an open source framework which is used for writing & running tests.
- Provides Annotation to identify the test methods.
- Provides Assertions for testing expected results.
- Provides Test runners for running tests.
- JUnit tests allow to write code faster which increasing quality
- JUnit is elegantly simple. It is less complex & takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually combine through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if test is going fine and it turns red when a test fails [29].

Functionality. JUnit is a Unit Testing Framework used by developers to implement unit testing in Java and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the followings:

- Eclipse
- Ant
- Maven

Release Activity. The last stable release of JUnit under study is JUnit 4.11 and release date is 15th Nov, 2012.

Number of Versions. The JUnit under study have more than 30 releases over a long period of time. This thesis studies is over the 24 versions of the JUnit.

4.2.2 Study Version Control data of JUnit for evolution

This part of Thesis shows an analysis of an open source software system evolution, JUnit, which is an open source test framework, based on its size, complexity, and modularity metrics. The JUnit open source software is under Common Public License (CPL) license and whose source code can be read by any user required to study it. The CPL (Common Public License) is a free software / open-source software license published by IBM.

The version information of JUnit from repositories

XUnit is the family name given to bunch of testing frameworks that have become widely known amongst software developers. The name is a derivation of JUnit the first of these to be widely known. The origins of these frameworks actually started in Smalltalk. Kent Beck was a big fan of automated testing at the heart of software development. The focus was on making it easy for programmers to define the tests using their regular smalltalk environment, and then to run either a subset or a full set of tests quickly. Kent and his followers would run unit tests after every change to the system going through a rapid edit and test cycle in the Smalltalk IDE [30].

JUnit was born on a flight from Zurich to the 1997 OOPSLA in Atlanta. Kent was flying with Erich Gamma, and what else were two geeks to do on a long flight but program? The first version of JUnit was built there, pair programmed, and done test first (a pleasing form of meta-circular geekery). JUnit took off like a rocket - and was essential to supporting the growing movement of Extreme Programming and Test Driven Development. Tons followed; nearly every language has at least one JUnit port. It was inevitable that it was also 'ported' back to Smalltalk as a real framework. It is rather difficult to think of any other piece of Java software that has made a greater contribution to Java software quality. Kent Beck has done an equally noble service to this field by releasing the JUnit source code as it was written, thereby bequeathing us a permanent historical overview of its development.

This thesis examined those releases most readily-available, from versions 3.7 up to 4.11.

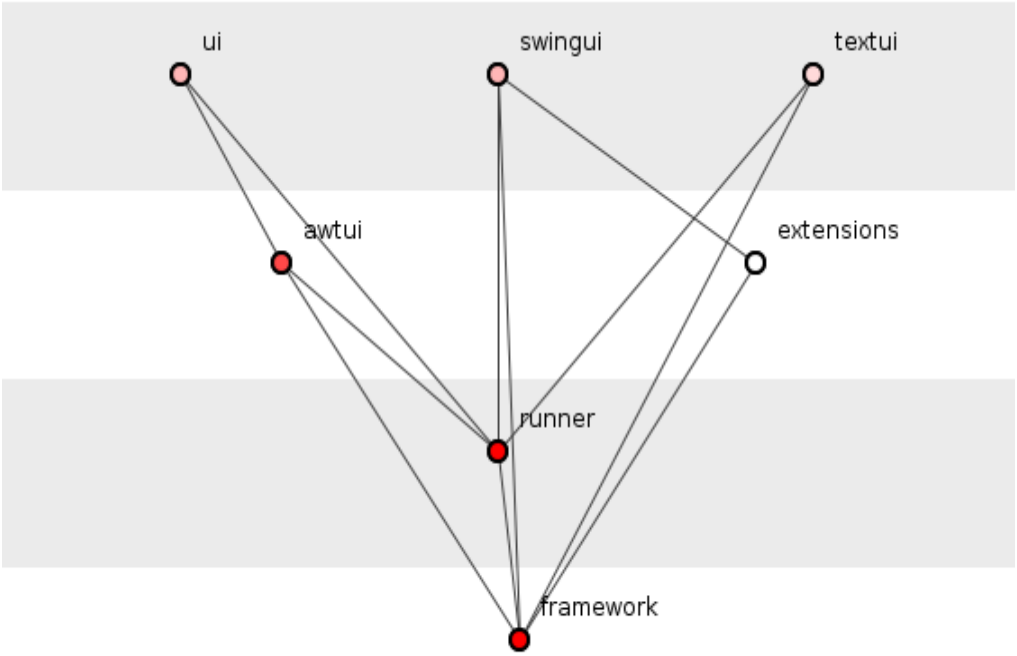


Figure4.5 JUnit 3.7 [31]

This Figure4.5 shows version 3.7 and perhaps one aspect seems most striking. Structure being a set of elements and their dependencies, most would conclude that JUnit version 3.7 is a well-structured work [31].

Here in Figure4.5 and all these kind of figures contains the package information of JUnit framework packages are such as textui, swingui, runner, awtui etc and lines shows how these packages calls each other.

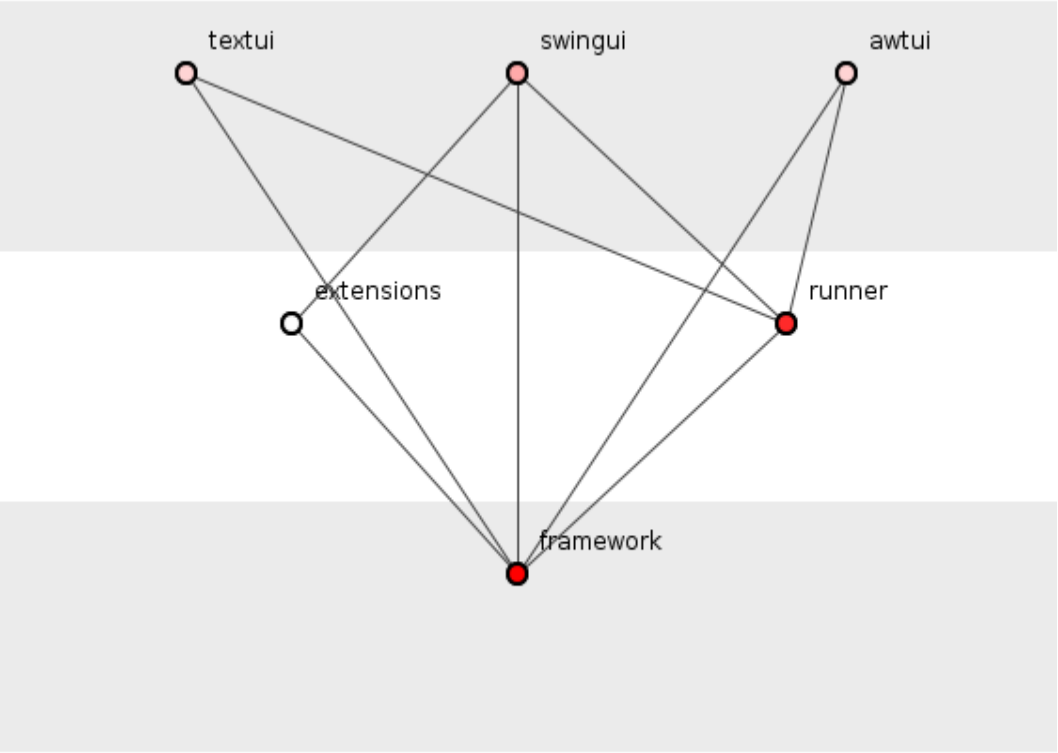


Figure4.6 JUnit3.8 [31]

In version 3.8 the ui package has packed its bags but otherwise the structure is unchanged. Package ui is not present in JUnit3.8 but its functionality is borrowed by other packages.

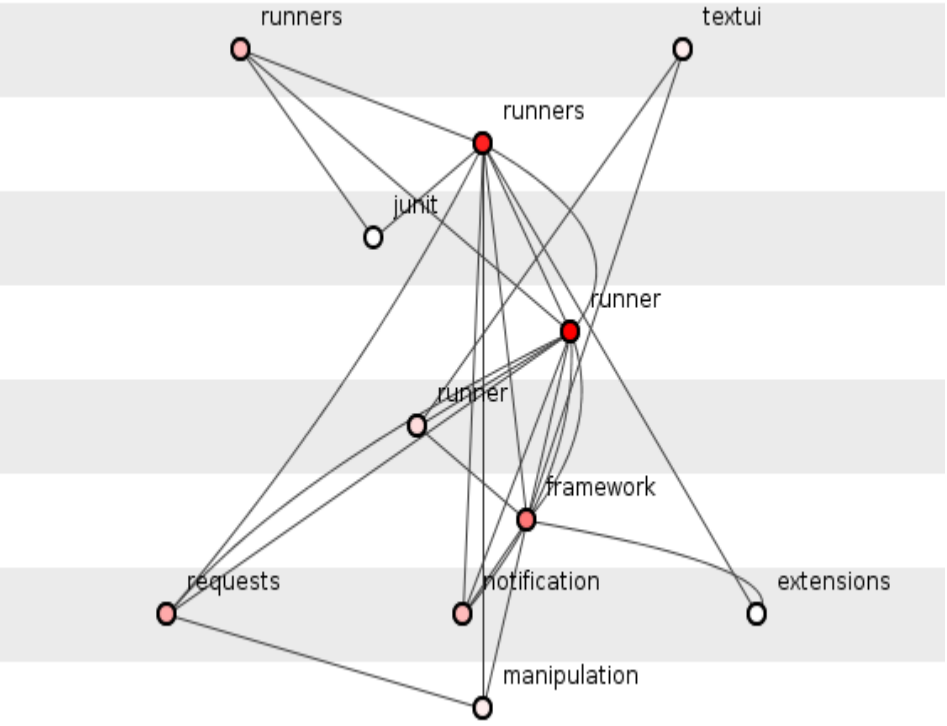


Figure4.7 JUnit4.0 [31]

The diagram is still relatively clean-looking despite the number of packages having almost doubled from six to eleven. A nagging doubt has surfaced, however. In some qualitative way the elegance of version 3.8 has, if not been lost, at least suffered some tarnishing.

Curved lines have appeared indicating our first dependencies going up the page. In itself, this is insignificant; Spoiklin's algorithm produces such curves as artifacts rather than damning moral judgments. According Spoiklin diagram in which a circle will represent a package, straight lines will represent dependencies from packages drawn above to those below and curved lines will represent dependencies from packages drawn below to those above. The colour of a package will indicate the relative number dependency tuples (that is, transitive dependencies) in which it partakes: the more red, the more dependency tuples [31].

A curved line only raises concern where it forms the infamous, "Bow," pattern of mutually-dependent elements and such, alas, is the case here. Mutually-dependent packages invite suspicion. Mutually-dependent packages suggest exposure to one another's ripple-effect changes and this troubles programmers. The package-structure looks manageable and concerns are allayed with release 4.1 whose structure is identical with that of version 4.0. Where in version 3.7 and 3.8 have less number of packages as well as functions also but in version 4.0 number of packages has increased by 5 more packages than version 3.8. Thus some new functionality added in version 4.0 than the previous one.

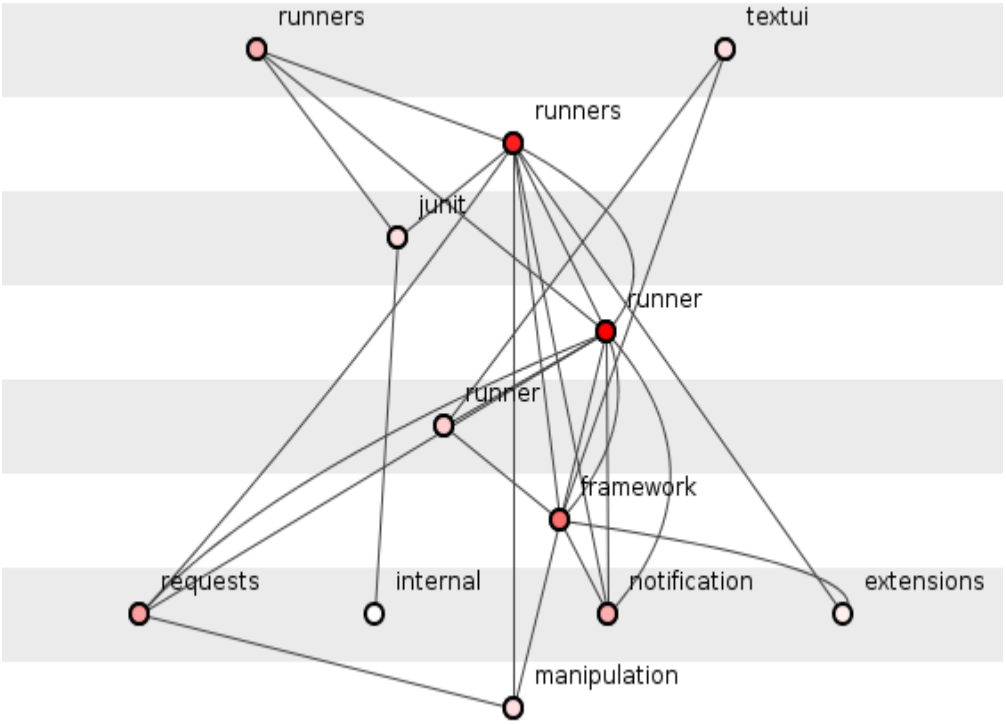


Figure4.8 JUnit4.2 [31]

Version 4.2 is also similar to 4.0; a single new package is added, internal. Version 4.3, however, is growling around the corner.

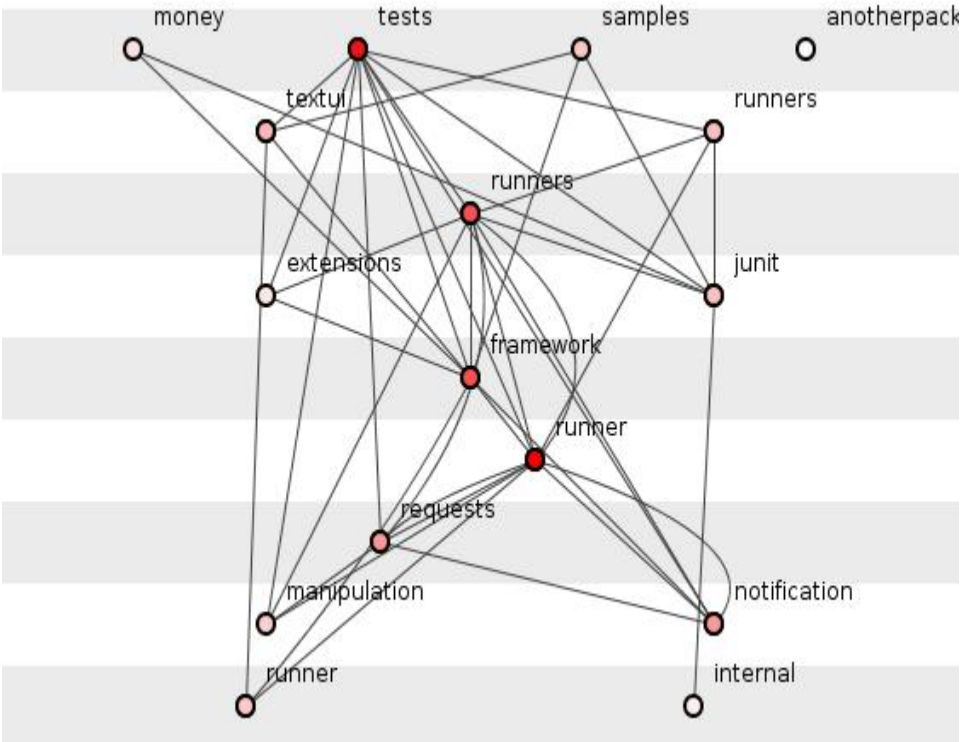


Figure4.9 JUnit4.3 [31]

Version 4.3 represents by far the largest structural change in the history studied here. Beneath the surface, the number of functions leaps from 564 in version 4.2 to 1309 in version 4.3 (it will fall almost as dramatically in the next release), though the number of packages rises from eleven to just sixteen. (This non-commensurate rise in the number of packages may help explain the fall in configuration efficiency from 31% to just 18%.) [31].

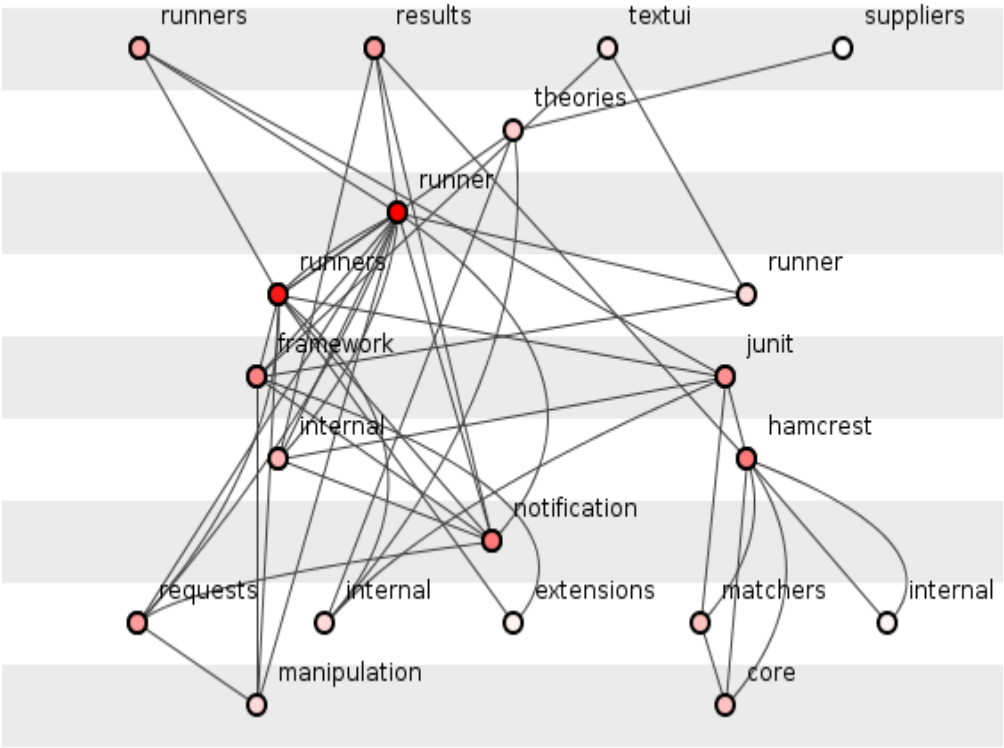


Figure4.10 JUnit4.4 [31]

This Figure4.10 shows version 4.4 and tests leaves us before we really got to know it. The number of functions falls from 1309 to 853; the number of packages rises to 19.

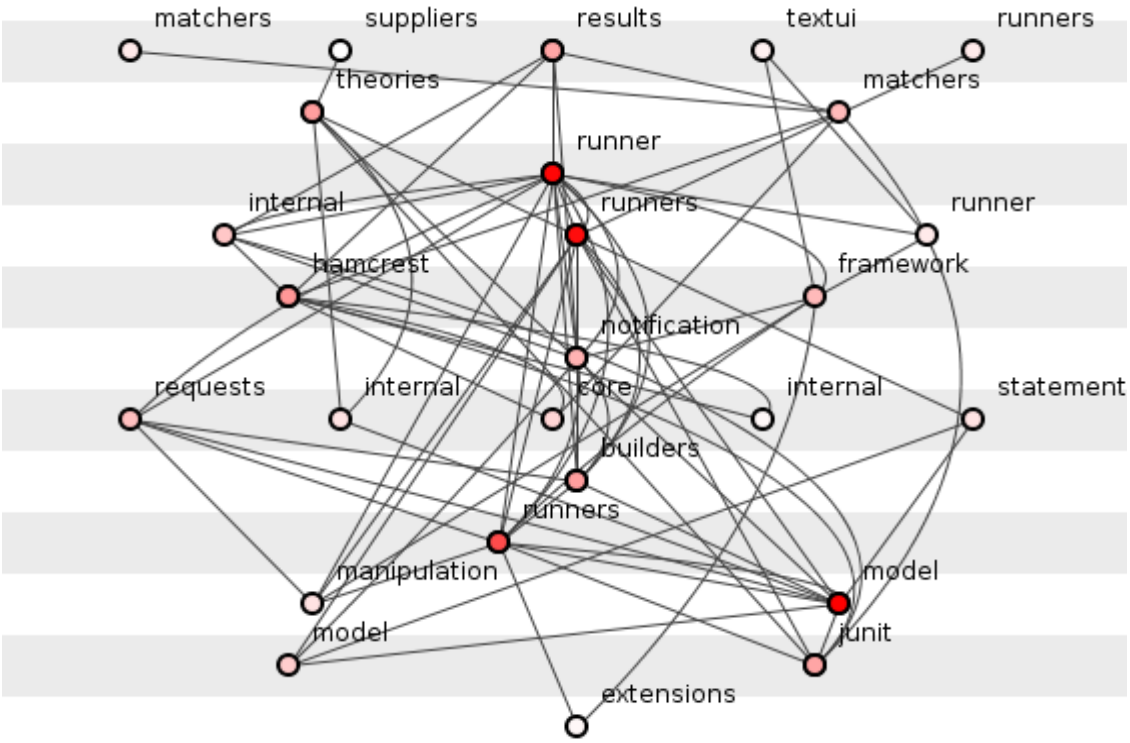


Figure4.11 JUnit4.5 [31]

A difficulty in tracing transitive dependencies characterizes the late phase of JUnit's package-structure. This Figure4.11 shows version 4.5. 150 functions more than version 4.4, the system boasts a configuration efficiency risen from 29% to 34%. Its twenty-six packages, however, have become embroiled.

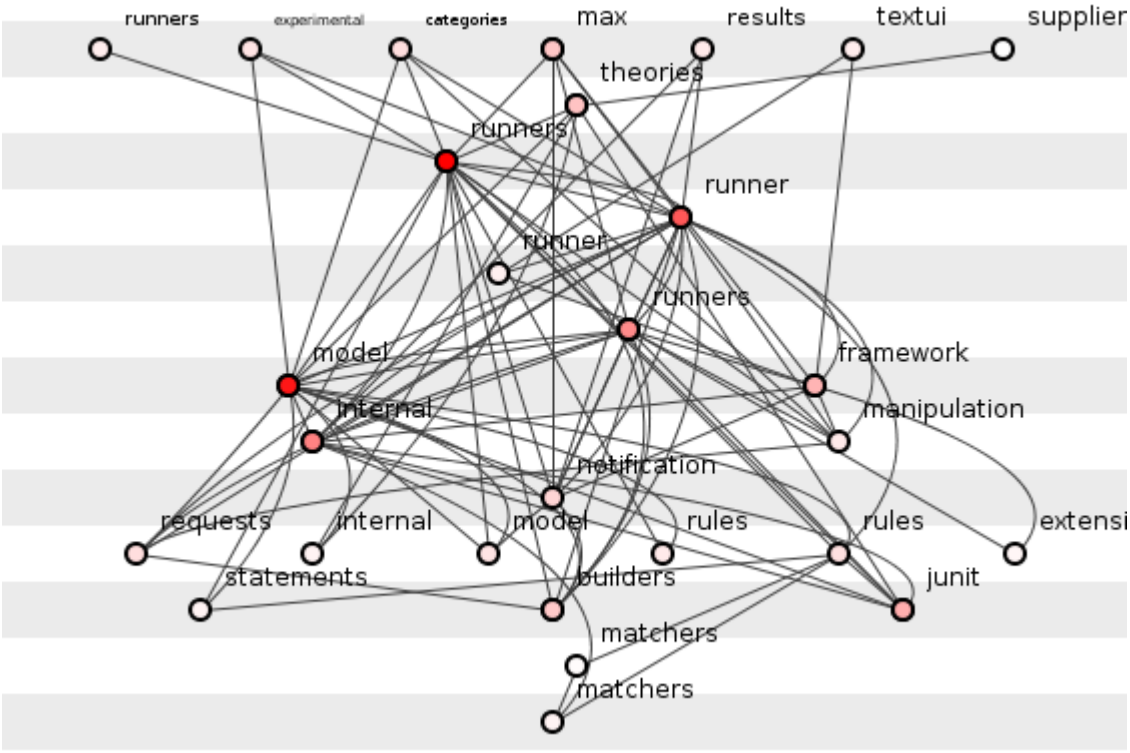


Figure4.12 JUnit4.11 [31]

By version 4.11, transitive dependencies have proliferated seemingly unchecked. We are far from the short dependency-chains and few cyclic-dependencies of good structure [31].

Metrics calculation using version control data

The selected set of metrics has been calculated for all the twenty seven versions of the selected open source software called JUnit. The metrics are classified into two groups called the size metrics and complexity metric.

- **Size metrics**

This thesis study involves SourceMonitor as tool is used here to calculate the given metric as Statements, Average Statements per Method, Method Call Statements and Percent Branch Statements. All the twenty four versions at the package level is analyzed and then collectively added to get the metrics at full version level.

Versions	Statements	Percent Branch Statements	Method Call Statements	Average Statements per Method
junit2	1,659	8.3	1,021	3.74
junit2.1	1,797	8.1	1,103	3.61
junit3	2,001	7.8	1,240	3.45
junit3.2	851	4	573	2.66
junit3.4	886	3.8	617	2.53
junit3.5	1,030	3.6	722	2.71
junit3.6	987	4	694	2.67
junit3.7	1,042	3.7	728	2.71
junit3.8	1,230	3.9	883	2.88
junit3.8.1	1,254	4.1	889	2.98
junit3.8.2	1,383	3.8	934	2.81
junit4.0	3,297	2.2	1,879	2.3
junit4.1	3,310	2.5	1,885	2.29
junit4.2	3,250	1.9	1843	2.2
junit4.3.1	3,571	2.3	2033	2.37
junit4.4	4,266	3	2402	2.29
junit4.5	5,455	2.7	2854	2.15
junit4.6	5,794	2.6	3071	2.19
junit4.7	6,187	2.5	3275	2.17
junit4.8	6,335	2.4	3335	2.15
junit4.8.1	6,337	2.4	3337	2.15
junit4.9	7,015	2.4	3641	2.13
junit4.10	7,413	2.4	3904	2.15
junit4.11	5,204	10.4	2,479	2.54

Table4.1 Size metrics over JUnit

- **Funtion Point Count**

This thesis work involved in measure of Function point analysis with the help of Function Point Modeler as a tool. After analysis the JUnit source code, below given detail is gathered. In this table value of Unadjusted Function Point (UFP) and Adjusted Function Point (AFP) is calculated by Function Point Modeler software tool.

version	Number of subsystem	Number of files	UFP	AFP	FP
junit2	6	38	42	27	1134
junit2.1	6	39	42	42	1764
junit3	6	46	53	53	2809
junit3.2	8	24	63	63	3969
junit3.4	9	29	76	76	5776
junit3.5	9	33	79	79	6241
junit3.6	9	32	98	98	9604
junit3.7	9	33	104	104	10816
junit3.8	8	40	107	107	11449
junit3.8.1	8	42	107	107	11449
junit3.8.2	8	44	88	88	7744
junit4.0	6	74	68	68	4624
junit4.1	9	80	109	109	11881
junit4.2	6	86	110	110	12100
junit4.3.1	6	90	110	110	12100
junit4.4	6	116	111	111	12321
junit4.5	6	147	115	115	13225
junit4.6	6	152	112	112	12544
junit4.7	6	159	117	117	13689
junit4.8	6	160	120	120	14400
junit4.8.1	6	160	122	122	14884
junit4.9	6	169	123	123	15129
junit4.10	6	178	125	125	15625
junit4.11	6	164	126	126	15876

Table4.2 Function point metrics over JUnit

Graphical representation

The results can be observed from graph shown below. The growth pattern of JUnit which helps in defining the Software Evolution for that particular selected software product, are evaluated in terms of metrics value shown in graph. In this representation x-axis show the number of version and y-axis shows its metrics value. The system has a high growing rate: it consisted of 1600 NOS initially and of 5000 at the end. This means that the total size of the JUnit increased by over 32 percent which is a growth rate. There are some major releases between 3.2 and 3.8.2 and after that the growth of the system is stable. The time span between major releases is more than the minor releases. After junit3.8.2 there is continuous grow in number of statements. Where in Figure4.13 the Junit4.9 contain highest value. Figure4.14 shows that Percentage of

braches statements contains various ups and down between JUnit version 2 to 4.10 but there is rapidly hike shows in version 4.11.

- **States of Statements in twenty four official versions of JUnit**

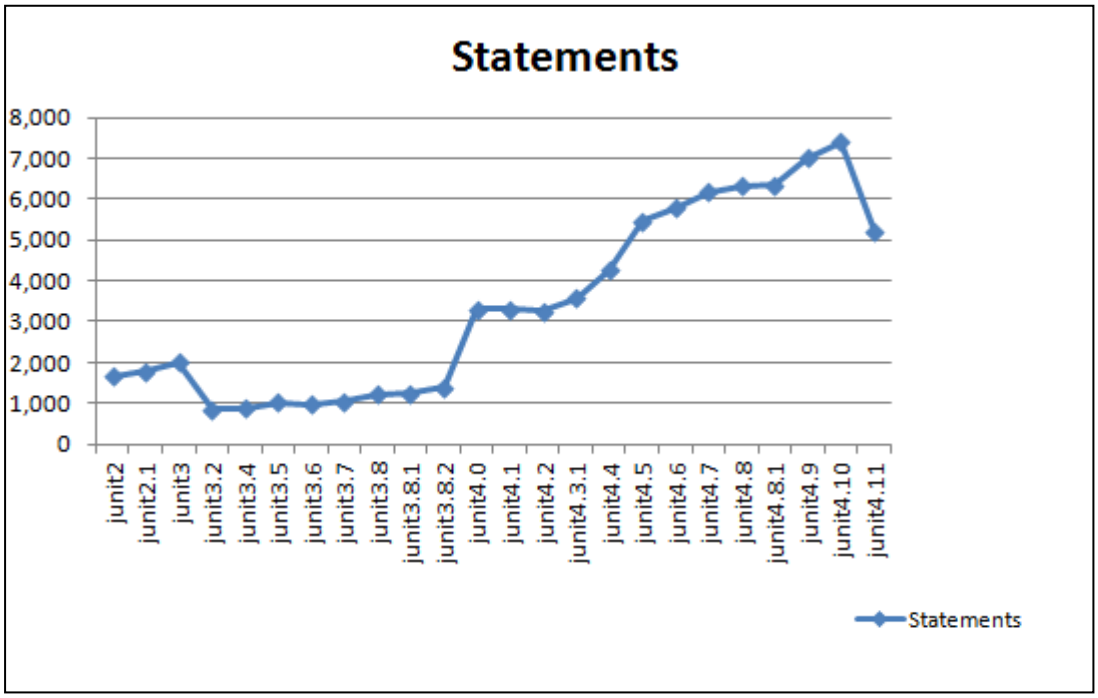


Figure4.13 Growth of Statements

- **States of PBS in twenty four official versions of JUnit**

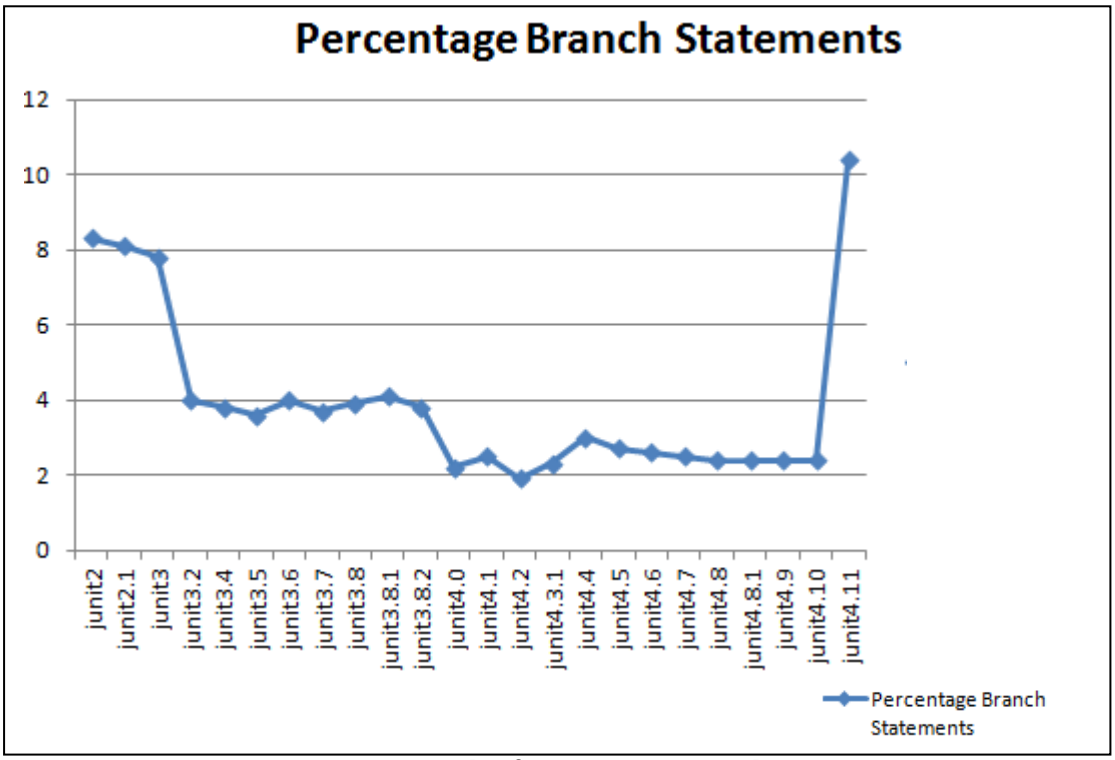


Figure4.14 Growth of Percentage Branches

- **States of Method call statement in twenty four official versions of JUnit**

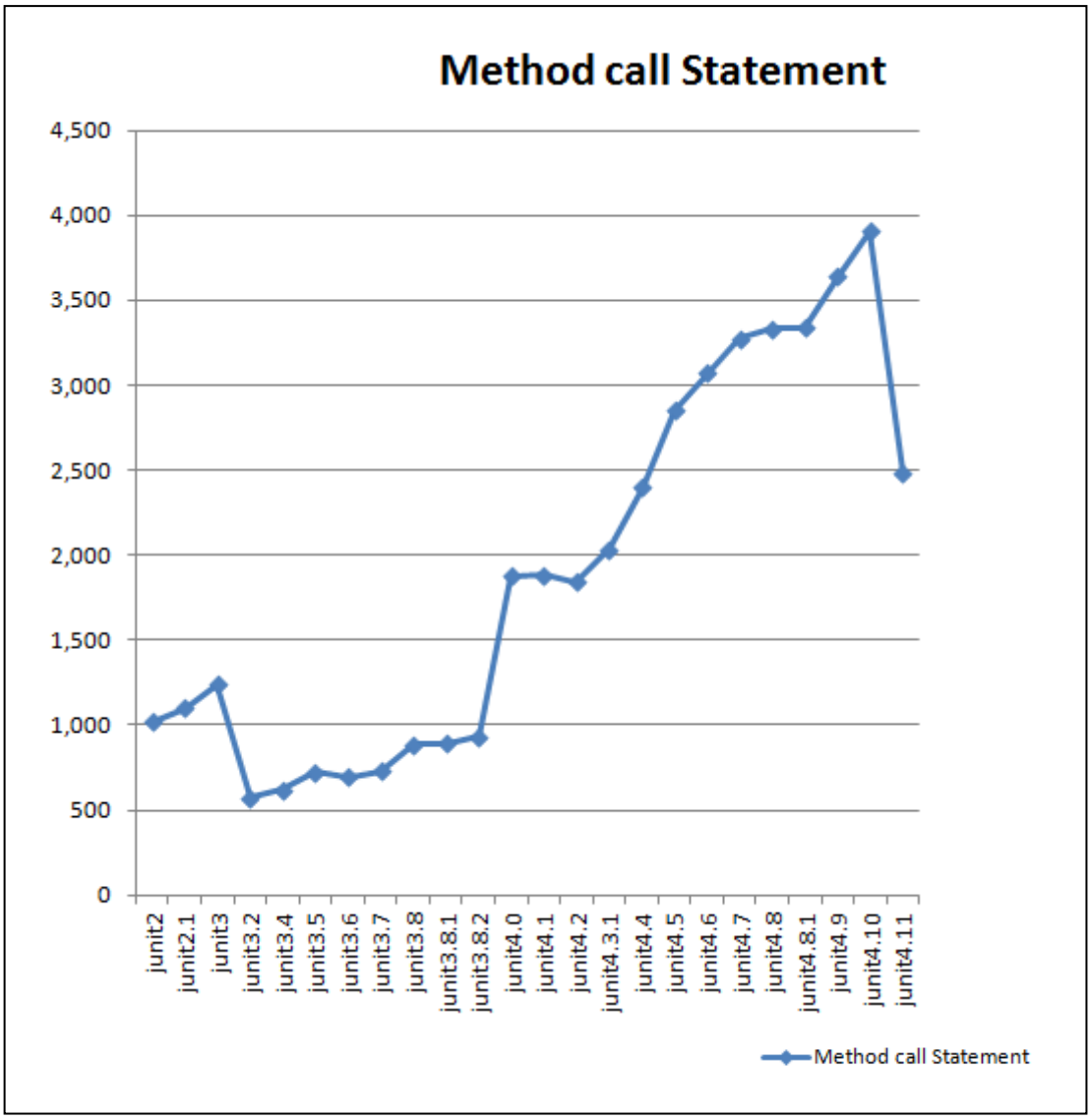


Figure4.15 Growth of Method Call Statement

Figure4.15 shows the growth of method call statements, version 2 to 3 there is linear increments then from version 3.2 to 4.0 there are minor differences and then after release of version 4.0 again there is a continuous increment in this metrics.

- **States of ASPM in twenty four official versions of JUnit**

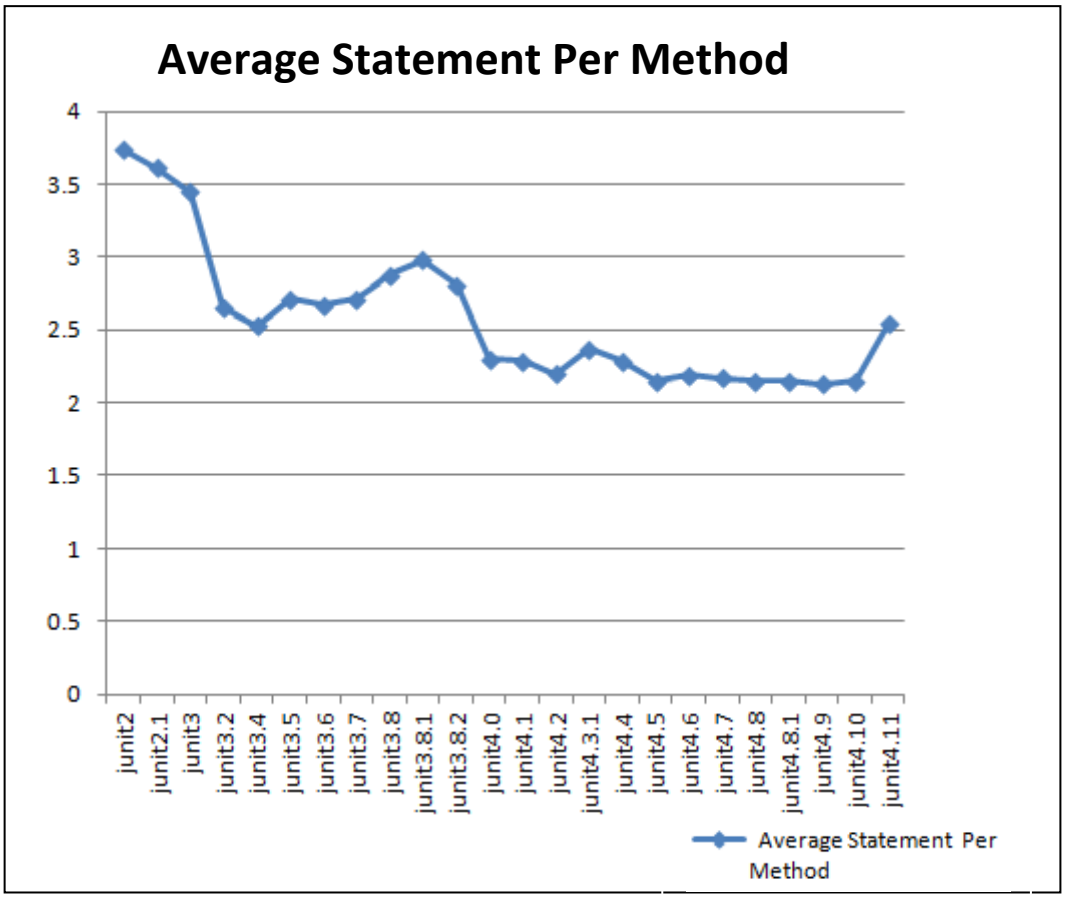


Figure4.16 Growth of Average Statement per Method

In Figure4.16 the average of statement method is varied version by version but there is not so much difference in average hence complexity of source code has maintained.

4.2.3 Lehman's Laws with Results

In this part of thesis work is now moving to define the Lehman's laws of software evolution using the measures some metrics of the 24 versions of JUnit.

Law I: Law of Continuing Change

The first law postulates that a program must continually adapt to its environment, otherwise it becomes progressively less useful. According to this law, evolving software product must be continually adapted the changes else they become progressively less satisfactory this law or the evolving software product must be adapted to the changing environment. The changes in usage environment may involve in some addition to statements which will increase the size of software leading to

growth. In this case study of the JUnit it can easily be recognized that the gradually changes has occurred in each versions as shown in Figure5.1 and 5.2.

Law II: Law of Increasing Complexity

The second law postulates that as a program evolves, its complexity increases, unless proactive measures are taken to reduce or stabilize the complexity. This law define that software complexity must be increase in several versions. In this case study of the JUnit it can easily be recognized that the complexity increases gradually as the new version of the software. In Table4.2 the number of FP is increasing as per version of JUnit this will shows increment in complexity. The Figure4.3 visualized that the average complexity is varied over the different versions which states that JUnit supports the second Lehman's law of software evolution and JUnit4.11 contains highest value of average complexity.

Law III: Law of Self Regulation

This law suggests that the evolution of large software systems is a self-regulating process, that is, the system will adjust its size throughout its lifetime. The observations made from the Subfiles (test, samples) of JUnit by Function point modeler shows that there is steady increase in size of the software following the Lehman's 3rd law of software evolution and also shown in Figure4.5 to Figure4.12.

Law IV: Conservation of organizational stability

The average global effective rate in evolving software tends to remain constant over product lifetime. This means it is hard to change the staffs who have been working on evolving software. This law, also known as "invariant work rate", stipulates that the rate of productive output tends to stay constant throughout a program's life time. This study has no proof to support this law.

Law V: Conservation of Familiarity

The average incremental growth remains constant as the software evolves. A huge change that might cause lack of familiarity of staff members is avoided. The Figure 5.1 and 5.2 are support this law because the basics functions are linear in showing graphs these functions such as statements and method call statements are consist the basic familiarity of JUnit.

Law VI: Law of Continuing Growth

This law suggests that incremental system growth tends to remain constant (statistically invariant) or to decline, because developers need to understand the program's source code and behaviour. The functionality provided by the software

should continually grow so that the users get satisfied over a long period of time which is an advantage of the evolving systems. In this case study of the JUnit shows the continuous growth of the software project as shown in Figure4.13 and 4.15 continuous growths.

Law VII: Law of Declining Quality

This law stipulates that over time, software quality appears to be declining, unless proactive measures are taken to adapt the software to its operational environment. The decrease in quality can be predicted by increase in the maximum complexity as increase in complexity itself is an indicator of declining quality. In case of JUnit, the law is supported with some exception because as shown in Figure4.3 where average matrix value of JUnit4.11 is highest.

Law VIII: Law of Feedback system

This brief outline of the Laws of Software Evolution has included references to the role of feedback in the process. These remarks may be generalised with the observation that global software system evolution processes constitute complex multi-loop, multi-level, multi-agency feedback systems. The existence of feedback system is true for JUnit since there is the user mailing system in the website hosting JUnit development. Total number of bugs reported till April 2013 is 968 which mean a proper feedback system is there to support the JUnit development. User mailing list also defines that the request of new features and changes in existing features is also present in this software.

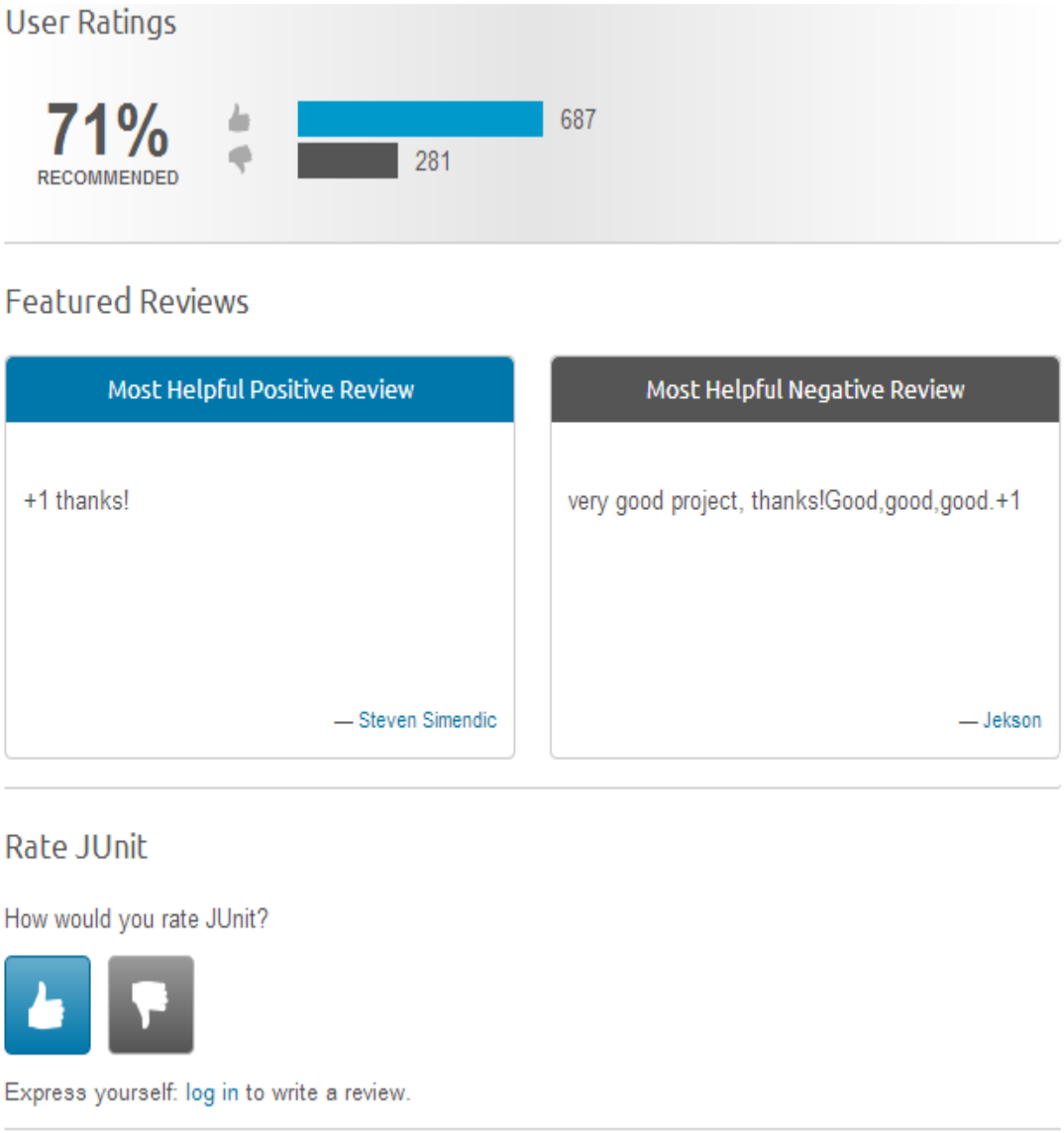


Figure4.17 Feedback data by Sourceforge site [32]

Figure4.15 shows the feedback reviews of JUnit framework online on sourceforge site.

Chapter 5

Evolution Result of Unit

This chapter summarize the result of work that was done in previous chapter and shows the supported Lehman's law.

5.1 Results

Using evaluated information which has shown in the previous chapter of JUnit is discussed here. As mentioned before that this thesis show the result in visualized manner so here is the graph of results which shows the variation of selected size metrics in different 24 version of JUnit open source software.

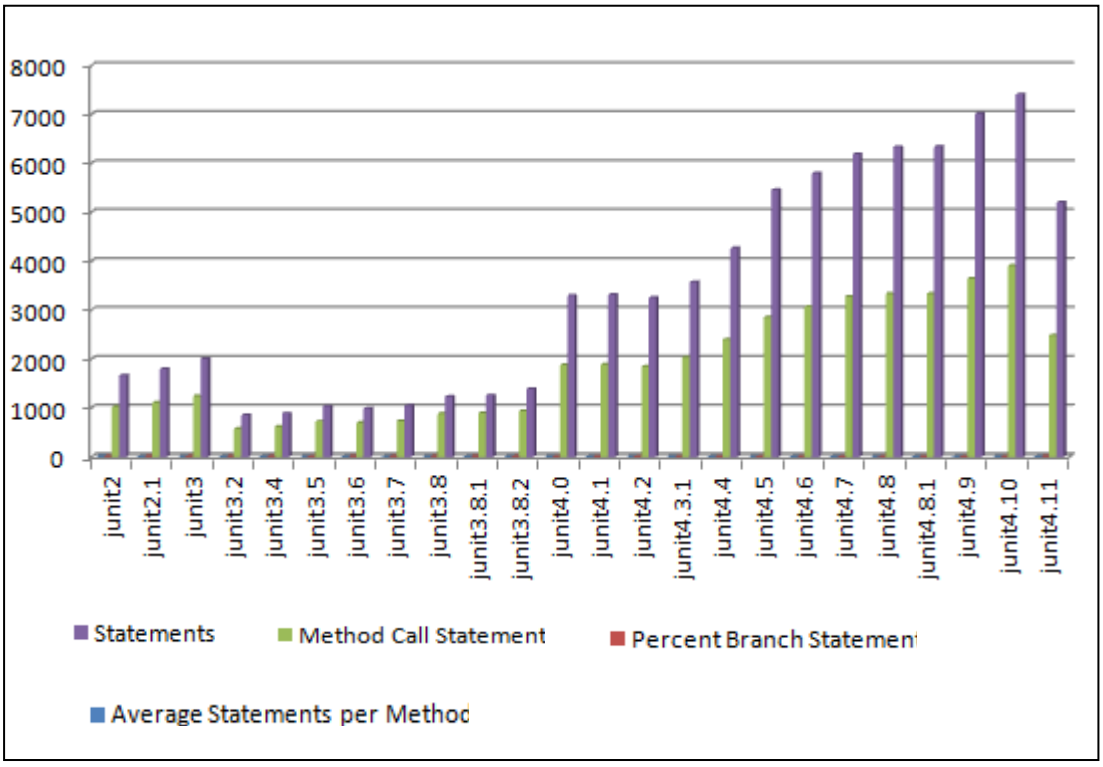


Figure 5.1 Growth of all evaluated metrics by SourceMonitor tool

By analyzing these results, size metrics of the JUnit software system is increasing. Figure 5.1 explains the growth of different size metrics of different version of JUnit. From releases 4.1 to 4.10 version of JUnit shows major differences in its source code. Figure 5.1 shows the significant changes between versions but from version 3.4 to 3.7 fewer changes occur if compared to other versions. The calls and statements

coevolves, both are changed in the same revision: most of the statements changes are done in the same revision as the associated source code change.

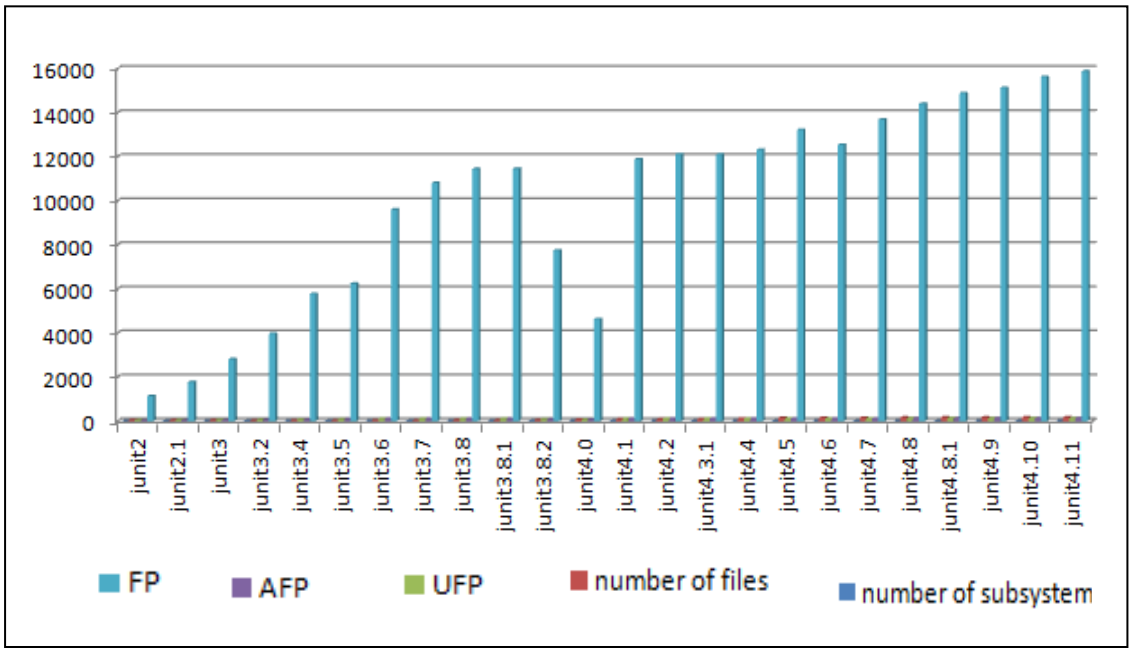


Figure5.2 Growth of Function Point Size Metrics by Function Point Modeler

In general the Growing rate of the whole system also increases. Figure 5.2 shows that the growing rate of function point of the system. The starting from JUnit2 to JUnit3.8 function point has increased but from JUnit3.8.2 to JUnit4.0 it decreases and after that there is linearly increasing rate has shown here till JUnit4.11. This behaviour shows that function point has varied from version by version that mean for each function points in each version efforts have also varied. In the last examined releases only a few new files has been added and deleted which is shown by the growing rate for JUnit4.11. The version JUnit4.11 is a stable release from 2012.

This graph shows that number of subsystem is not growing so well but the number of files is growing at well rate. For calculate the function point of the system, the modelling approach is used as like JUnit is the model and its main source code files are the sub models.

5.2 Results to Support Lehman's Law of Evolution

Following table shows the result of thesis work to satisfied Lehman's law of software evolution in the area of open source software with the proof of support.

	Brief Name	**Support**	**Indicator**
1	Continuing Change	Yes	Figure5.1 and 5.2 demonstrate continuous change.
2	Increasing complexity	Yes	Proved in Table4.2 by increases number of FP and also visualized by Figure4.3.
3	Self Regulation	Yes	Shown in Figure4.5 to 4.12.
4	Conservation of Organizational Stability	No	No data that provides evidence for or against the law is applicable.
5	Conservation of Familiarity	Yes	Figure5.1 and 5.2, proof this law
6	Continuing Growth	Yes	Figure4.13 and 4.15 support this law.
7	Declining Quality	Yes	Figure4.3 proved this law(with exception)
8	Feedback System	Yes	Email system is used as feedback system and total of 968 emails has been send by user in Figure4.17.

Table 5.1: Support to Lehman's Laws of Software Evolution

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

The realistic open source software known as JUnit has observed to study the software evolution by measuring some set of size metrics. The Software evolution way in the open source software field are studied with the help of the version control system data by using the open source software repositories. These SVN repositories are based on concept of version control system. For working on software evolution these repositories plays the most important role because all the required data are collected from these repositories. Then this available data evaluated on the basis of software metrics. After the evaluation analyze the resulted information for check that this data is support the Lehman's law of evaluation or not. This thesis presents a way to study the software evolution process of open source software product with the help of data that is present on version control system. This kind of study is become more interesting because this empirical study shows the development facts of that software. In the application of Lehman's law 1, law 2, law 3, law 5and law 6 was easy to find by using selected set of metrics and the law 4, law 7 was hard to determine for open source software because of its unlimited resources and less reliable resources in open source software environment.

Hence the study work of this thesis shows the evolution characteristics of open source software called JUnit. This thesis result shows that selected size metrics varied from version to version and support Lehman's 1^{st} , 2^{nd} , 3^{rd} , 5^{th},6^{th} ,7^{th} , 8^{th} laws of software evolution. This study work help to understand the way of open source software evolution methodology for some addition research work.

6.2 Future Scope

OSS field is very wide for research work, in this thesis some selected set of metrics are used to study the measurement methods for the software evolution in open source software environment. Following Lehman's law and expanding these studies into the world of open source could yield more ideas for future work. Commercial software has more of a unilateral direction where design, development, and testing occur in a controlled environment. Open source software has a distributed structure where developers start a project to "scratch an itch". These projects "release often" and testing is done by users and consumers of the software. Thus the feedback system model should be expanded and updated to account for this different open source software and also the more evolution methods are needed for maintain its quality. This study can also lead to more different software evolution studies based on some other software metrics which are still not covered such as reliability, maintainability and other quality metrics.

References

[1] Jing Yang and Jiang Wang, "*Review on Free and Open Source Software*", in *Proc. IEEE International Conference on Service Operations and Logistics, and Informatics,* vol.1, pp. 1044- 1049, 2008.

[2] Ming-Wei Wu and Ying-Dar Lin, "Open Source Software Development: An Overview", *IEEE Journal of Computer Science*, vol. 34, issue 6, pp. 33-38, 2001.

[3] OSS [Online]: http://www.cippic.ca/open-source.

[4] Huaiqing Wang *et al.*, "Open Source Software Adoption: A Status Report", *IEEE Journal of Software***,** vol 18, issue 2, pp. 90-95, April 2001.

[5] Atieh Khanjani and Riza Sulaiman, "The Process of Quality Assurance under Open Source Software Development", in *Proc. IEEE Symposium on Computers and Informatics*, pp. 548-552, 2011.

[6] A. Beard, H. Kim, "A survey on open source software licenses", *Journal of Computing Sciences in Colleges*, vol. 22 , no. 4, April 2007.

[7] Tilei Gao, *et al.*, "A Process Model of Software Evolution Requirement Based on Feedback", in *Proc. International Conference of Information Technology, Computer Engineering and Management Sciences*, vol. 2, pp. 171-174, 2011.

[8] Y. Lee, J. Yang, K. H. Chang, "Metrics and Evolution in Open Source Software", *Seventh International Conference on Quality Software*, pp. 191 – 197, 2007.

[9] E. Sink, "Version Control by Example*",* Pyrenean Gold Press, July 2011.

[10] version control system[Online]: http://www.catb.org/esr/writings/version-control/version-control.html.

[11] Y. Ren, T. *Xing*, Q. Quan, Y. Zhao, "Software Configuration Management of Version Control Study Based on Baseline", in *Proc. 3rd International Conference on Information Management, Innovation Management and Industrial Engineering*, 2010.

[12] Stephen Cook "Software Evolution and Software Evolvability", *University of Reading, UK,* 2000.

[13] Guowu Xie et al. "Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software" in *Proc. IEEE International Conference on Science Maintenance, Edmonton,* 2009.

[14] Michael W. Godfrey and Daniel M. German "On the Evolution of Lehman's Laws" 2013

[15] Nindya Kotwal, "Measuring the Open Source Software using Version Control System to study the Software Evolution", Thapar University, Patiala, June 2012.

[16] Parminder Kaur and Hardeep Singh "Measurement of Processes in Open Source Software Development" *in Proc. Trends in Information Management*, vol. 7, No 2 Dec 2011.

[17] K. Crowston, Kangningwei, J. Howison, A. Wigggins, "Free/Libre Open Source Software Development What We Know and What We Do Not Know", *ACM Computing and Surveys*, vol. 44, edition 2, 2012.

[18] Michael W. Godfrey , *"The* Past, Present, and Future of Software Evolution*" International Conference on Software Maintenance* , 2008.

[19] M M Lehman "Rules and Tools for Software Evolution Planning and Management" Department of Computing Imperial College. 2013.

[20] Hilda B. Klasky "A Study of Software Metrics", Graduate School-New Brunswick Rutgers, The State University of New Jersey, May, 2003.

[21] H. Pei *et al.*, "A Systematic Review of Studies of Open Source Software Evolution", in *Proc. 17th Asia Pacific, IEEE Software Engineering Conference,* pp. 356-365, 2010.

[22] Gurdev Singh and Vikram Singh, "A Study of Software Metrics", *International Journal of Computational Engineering & Management*, vol. 11, January 2011.

[23] Mohammed Abdullah Al-Hajri *et al.*, "Modification of standard Function Point complexity weights system" , *The Journal of Systems and Software* , pp. 195-206, vol. 74, 2005.

[24] Barbara Kitchenham, "Software Quality: The Elusive Target", *National Computing Centre, IEEE Software,* vol.13, January 1996.

[25] Andy Kellens *et al.*, "Reasoning over the Evolution of Source Code using Quantified Regular Path Expressions", in *Proc. 18th Working Conference on Reverse Engineering* , 2011.

[26] Yongchang Ren *et al.* "Software Configuration Management of Version Control Study Based on Baseline", in *Proc. 3rd IEEE International Conference on Information Management, Innovation Management and Industrial Engineering,* 2010.

[27] CFPS M.E. "*Function Point Modeler*" Function Point Modeler Inc. Germany 01 March 2011.

[28] Version control system book [Online]: *http://svnbook.red-bean.com/en/1.7/svn-book.html.*

[29] Case study of JUnit [Online]: *http://edmundkirwan.com/general/junit.html*.

[30] JUnit Tutorial [Online]: *http://tutoiralspoint.com.*

[31] Version information of JUnit [Online]: *http://www.martinfowler.com /bliki/Xunit.html.*

[32] SourceMonitor [Online]: *http://www.campwoodsw.com/sourcemonitor.html*.