Chapter 5

RESULTS AND DISCUSSION

5.1 Software Quality

As McConnell [96] pointed out, successful OSS projects should not be compared with the average proprietary source project. They should be compared with the software development effectiveness achieved by leading edge organizations that use a combination of practices to produce better quality software and keep costs and schedules down. This assertion suggests that the quality of the delivered product should be compared with the quality levels pursued by the modern software industry.

But first it is necessary to introduce the notion of software quality. Just as any other product, for example a building, a car or a commercial electronic device, has a level of quality, software products have some kind of quality as well. By quality we mean whether or not the product conforms to a set of standards posed by someone, either by the manufacturer or by the customer. For example: Does the software do what the user wants it to do? Or is it well designed, well coded, well tested, error free, so that it will function properly? For a more detailed discussion on “What is software quality?” [97].

In order to assess software quality, it is usual to construct models which combine different aspects of quality to produce a final result about the level of a software product quality. There exist several models of software quality, which suggest various ways to bring together different quality attributes. Each one of these models tries to aggregate these several attributes of quality in order to give an overall view of the quality of software. One of such models is the ISO 9126 model [100].

The ISO 9126 quality model was suggested in 1991 in order to combine the several views of software quality models that existed during that time. ISO 9126 is a hierarchical model consisting of six major attributes contributing to software quality. The attributes and what they represent are:

- Functionality
- Reliability
- Usability
- Maintainability
- Portability
- Efficiency
As with other models, these attributes do not necessarily lead to direct measurement, but to other indirect measurements, which can be further divided into other quality factors. Some of the above attributes, which can be measured using direct measurement, are usually expressed as some kind of equation, which has inputs of various source code metrics such as the McCabe's cyclomatic [98] or Halstead Effort Metrics. An example of such equation is the Maintainability Index proposed by Carnegie Mellon's Software Engineering Institute [138]. An extensive discussion and reference to software metrics can be found in [99].

It is worth mentioning here that software quality models and quality in general, is a field of a continuous debate on the correctness, for example, of the models or the attributes/factors/metrics they use to measure quality. In this chapter, we preferred to include security in the reliability section. These quality factors have been examined in varying degree in the OSS literature. The rest of the article is structured according to the quality factors mentioned above. Hereto, we have to mention here that it would be ideal to use the same standard on some commercial systems in order to compare their quality versus the quality of F/OSS projects. However, it is difficult to find such assessments and quantitative studies, since many of them are hard to find and even they are confidential.

5.2 Functionality

The first quality factor we examine in this chapter is functionality. According to the ISO 9126 standard, functionality is “A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs” [100]. The standard further noted that functionality is a set of attributes and “This set of attributes characterizes what the software does to fulfill needs, whereas the other sets mainly characterizes when and how it does so” [100]. The rest of the section is based on the latter concept and explores the extend OSS fulfilling this prerequisite. OSS has produced products that in certain cases they are a kind of “monopoly” in their application domain. Such a “category killer” is BIND (Berkeley Internet Name Domain) Server, a critical application for Internet's infrastructure. The BIND project was initiated as an academic project and soon became the dominant Domain Name System (DNS). Now it is being maintained by the Internet Software Consortium and its dominance and long time usage has proved
that it implements the functionality of a DNS very well. Furthermore, the well-known Apache Web Server is used in 62.7% of the cases [101], implying that the functionality offered satisfies the majority of the web server administrators and hosting companies. There are numerous other examples of OSS products, which are being used by developers and users instead of closed source software, which performs the same functionality. Such examples are relational database management systems, programming tools and much more. Table 5.1 shows a summary of the major OSS products.

Table 5.1 Major OSS products

<table>
<thead>
<tr>
<th>Application Type</th>
<th>Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Systems</td>
<td>• GNU/Linux</td>
</tr>
<tr>
<td></td>
<td>• Free/Net/OpenBSD</td>
</tr>
<tr>
<td>Graphical User Interfaces</td>
<td>• GNOME</td>
</tr>
<tr>
<td></td>
<td>• KDE</td>
</tr>
<tr>
<td></td>
<td>• Ximian</td>
</tr>
<tr>
<td>Office Suites, Graphics</td>
<td>• OpenOffice.org</td>
</tr>
<tr>
<td></td>
<td>• Koffice</td>
</tr>
<tr>
<td></td>
<td>• GNOME Office</td>
</tr>
<tr>
<td></td>
<td>• Gimp</td>
</tr>
<tr>
<td>Network Applications</td>
<td>• Apache</td>
</tr>
<tr>
<td></td>
<td>• BIND</td>
</tr>
<tr>
<td></td>
<td>• Sendmail</td>
</tr>
<tr>
<td></td>
<td>• Mozilla</td>
</tr>
<tr>
<td></td>
<td>• Samba</td>
</tr>
<tr>
<td>Programming Languages/Compilers</td>
<td>• GCC</td>
</tr>
<tr>
<td></td>
<td>• Perl</td>
</tr>
<tr>
<td></td>
<td>• Python</td>
</tr>
<tr>
<td></td>
<td>• TCL/TK</td>
</tr>
<tr>
<td></td>
<td>• PHP</td>
</tr>
</tbody>
</table>

The majority of the OSS applications lie in the area of Internet or system applications. This is something that has been identified by some of the pioneers of OSS like Brian Behlendorf and Eric Raymond. They identified that open source projects were involved mainly in projects that had to do with networks or operating systems applications [102] and [103]. The majority of the successful cases of OSS are coming from the areas mentioned above and not from the desktop or GUI applications category. Figure 5.1 shows the number of the projects of the top categories, hosted at
Sourceforge.net (a major host provider for OSS projects). Behlendorf suggests that OSS development tends to be more effective in projects where incremental change is rewarded and he states that this kind of development applies to back-end systems rather than front ends. An answer to the question why developers spend more time on application from the network and system software category can be given if someone takes into account the “egoboo” argument by [104]. Raymond states that one of the main reasons that someone voluntarily contributes to an OSS project is the “egoboosting” or the enhancement of his/her reputation among others. Network and system applications are software that everyone uses, so in order to gain reputation a contributor to an OSS project prefers this kind of projects.

Moreover, the existence of a large variety of OSS in the areas mentioned (network and system applications, programming tools and operating systems), reveals that this kind of development is also suitable (suitability is another factor of a software functionality) for applications with well defined (pre-defined) requirements. It can be said that OSS seems better suited to horizontal requirements, addressing the needs of broad user categories [105]. McConnell explained this lack of requirements by saying: “By the time Linux came around, requirements and architecture defects had already
been flushed out during the development of many previous generations of UNIX”.

Whether OSS is appropriate to vertical requirements, such as the Enterprise Resource Planning and/or Customer Relationship Management (ERP/CRM) systems that require a lot of customization remains to be seen. However, such systems exist and one good example is Compiere [http://www.compiere.org] with more than 460,000 downloads and frequently found in the top ten downloads list of Sourceforge.net hosting service [http://sourceforge.net].

It still remains an issue whether this project will reach the success level of the commercial ERP systems. In addition to the above, OSS tries to be compliant with many standards that promote functionality and interoperability. The OSS movement sets as one of its first priorities compliance to open standards. By open standards we do not mean only standards that were developed in a democratic way, with anyone contributing and suggesting things. On the contrary, open standards mean that someone who wants to implement such standards does not have to pay for any patent or to sign any license that poses various limitations on the usage and the implementation of the standard. These limitations, that sometimes software companies pose, lead to incompatibility between different software products and raise interoperability issues. For example, UNISYS has posed some patent matters about the GIF image format which implements the LZW algorithm for image compression. This algorithm was patented after UNISYS and many products that implement this algorithm now have legal problems with UNISYS5. In fact that was the reason for the creation of, well known in the UNIX world, the GNU gzip: The Free Software Foundation wanted an alternative for the compress utility, which also implemented the LZW algorithm by Unisys. An interesting article about open standards and patents can be found at Coverpages, a resource center by OASIS [106].

In addition it is very difficult to implement some standards because of copyright reasons, some standards are hard to implement because they are not available to the public in an easy form and workable format, like on a web site. As Freericks [107] suggests, open access to a standard sometimes is not enough. Standards should be available to the public in different workable formats, in order to facilitate their usage by the developers’ community with the support of the appropriate tools and methodology. The openness of a standard also offers more intensive participation and
feedback of the users, which at the end will force the standard to be more flexible and optimized. By this way, the open source approach might lead to a faster diffusion of a standard. As mentioned above, the existence of open standards is very important for the interoperability of an application and lot of organizations are supporting open standardship. Interoperability is a critical quality attribute related to functionality. Such examples are the World Wide Web Consortium, the Internet Engineering Task Force, OMG, the OASIS Group and the Open Group.

OSS tries to stick to these standards when it implements an application that takes advantage of them. Indeed, OSS developers try to stick as much as possible to standards, open standards, and in many cases this has been the ultimate goal. Behlendorf [108] recounted an incident, which happened in the early days of the Apache server, in which the America On Line's (AOL) proxy server responded with an error message under certain circumstances when functioning with the Apache. The problem was tracked back to an anomaly in the implementation of the HTTP/1.1 protocol on the side of the AOL's product. The Apache group explained that their software implemented the protocol properly and the problem was a misimplementation of the AOL's proxy. AOL fixed the problem and everything went fine. Similar problems of interoperability are numerous in the case of the implementation of the HTML definition, where many web browsers and servers implement their own “customized” version of the HTML and do not perform properly with other products.

It is worth mentioning here that, for the Linux7 kernel case interoperability is achieved with the help of various modules (which mainly facilitate porting as mentioned below) that can be loaded into the kernel in order to achieve communication with other systems. An important OSS product that can offer invisible file and printing sharing among various platforms is samba [http://www.samba.org]. Samba makes it possible for machines with different file systems and network protocols to share files and printing services. This makes working in a multi-operating system environment easy and seamless.
5.3 Reliability

The current status of the global IT market makes reliability one of the most important factors of software quality. The ISO 9126 [100] model defines reliability as “A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time”. With the presence of critical applications and applications that run over the Internet for a long period of time, it is crucial for organizations to have software that ensures all around the clock reliability. Examples of such large applications are online shops, monitoring applications that use the Internet as means of communication, banking applications and so on. Proponents of OSS claim that OSS satisfies the condition that the ISO 9126 model poses and there is enough evidence to support that OSS is more reliable than proprietary software. An important factor that directly affects reliability is the frequency of bug discovering. In this case, the status of OSS in bug discovering is described by [96] postulation: “Given enough eyeballs, all bugs are shallow” (otherwise known as “Linus Law”). Indeed, feedback about bugs and misimplementations in OSS is direct and immediate: Once a version of software is released, it is a matter of days, hours or even minutes before the first bug is reported and the official fix is announced [109].

An example of such immediate response is the case of the Teardrop attack described by [110]. In 1997 a denial-of-service (DoS) attack against the Linux operating system was reported. A flaw in the Linux kernel's IP stack caused the system to crash, when a special IP packet was received. What made this attack more interesting is that it was also affecting another well known closed source operating system, making Teardrop a case of a heterogeneous attack. For the Linux case, the vulnerability was fixed within few hours. Moreover, the patch which fixed the bug contained a fix for another bug, which developers observed, but attacker didn't. The patch, which was originally aimed at the first bug, immunized the system against the second bug as well. This rendered Linux resisting future attacks of not only Teardrop itself, but of other invariants of it as well. The patch for the other closed source operating system took a longer time to be published and successive patches were needed for the other kind of attacks to be resolved. Another fact that makes F/OSS more recoverable is that these patches are immediately put on line and are included in the code stored in the CVS repository of...
the product. It is not strange to see a new version of software, one day after the previous one was released. In fact, during the early days of Linux, this was the case. In F/OSS there is no need for someone to wait for the company to publish a fix. In fact, even if a particular F/OSS project is abandoned by its original coordinators, there is always the possibility for someone else to take over, allowing people to continue using the software.

The strength of OSS comes to a large extent out of the principle mentioned above, namely fast bug fixing. But for this to become a reality, OSS depends heavily on the Internet and tools that operate over it, in order to make communication between developers fast and effective [111]. Effective communication is needed not only between developers, but also between developers and users and between users themselves. It is important for the day-to-day users to have an organized way, a well organized set of tools for communicating with the coordinators of a project in order to submit bug reports and several other problems associated with the project. As Feller and Fitzgerald [111] noted, “These tools are backed up with an ethos which recognizes quality contributions from any source and treats users as co-developers”.

These two points operate as a motivation factor for users and developers to submit either new functionality, bug reports and/or bug fixes. In the same book it is mentioned that feedback about the software is rather different from that of proprietary software, where the tight schedules and low budgets often limit testing and bug discovery. OSS users and developers, (in many cases a user is also a developer) constitute a large pool of beta testers, who try the software, report bugs and fix them, not only because it is their job to do so (which is the case in proprietary software), but because they use the software for their own needs.

Mockus, Fielding and Herbsleb [112] conducted an extensive empirical study on the development of two open source projects, the renowned Apache Web Server and the Mozilla browser. They processed the mail archives, the CVS repository, the BUGDB and other sources in a period of three years in order to test a group of questions and formulate a group of hypotheses about OSS. In that study, the authors reported that in the Apache case, 458 people reported 591 problems that resulted in code change and 182 people did 695 fixes. For the Mozilla case, things were different with 113 people reporting 50% of the bugs. However 46 of these 113 people did not contribute any
code for the fix. A good study that present errors found in the Linux kernel by automatic, static, compiler analysis is presented by [113].

The authors in this study examined the bugs found in 21 versions of the Linux kernel and compared the results against a version of the OpenBSD. They found that the part of the kernel were most of the bugs are concentrated, is the device drivers section, which has error rates up to three to seven times higher than the rest parts of the kernel. Furthermore they showed that the bug distribution among source code files follows a logarithmic distribution and the clustering of the errors pointed out that for the most heavily clustered error type, less than 10% of the files contained those errors. What is more is that, the average lifespan for certain types of the bugs across the 21 versions of the Linux kernel were found to be about 1.8 years. The comparison of the error rates of the OpenBSD against the Linux kernel showed that the OpenBSD has a higher error rate, from 1.2 to 6 times higher. In F/OSS, the time to resolve bugs is also short. In the study by Mockus et al. mentioned before, authors questioned the time it took to resolve problems in these projects, whether high priority problems were resolved faster than low priority problems and whether the resolution interval had increased over time. Results on the Apache project showed a cumulative 50% of the problems were fixed within one day, 75% within an interval of 42 and 90% within 140 days. Concerning priority, there was prioritization of the problems according to their nature. For example, problems that fell into the “Core” and the “Most sites” category were resolved as quickly as possible. Regarding resolution interval, it was found that there was indeed a reduction and the interval was significantly shorter in the most recent periods of the software as studied. As the authors point out, “this indicates that important aspect of customer support improved over time, despite dramatic increase in the number of users”. For the Mozilla project the median resolution interval is much longer than for Apache. This happens mainly because in Mozilla the interval between the fix submission and fix verification by the project coordinators is much longer.

One additional feature, which plays a critical role in software product quality, is defect density. Again Mockus et al. present an extensive study on defect density for the Apache and the Mozilla projects, and compare it against four other commercial products. In order to measure the defect density they used, as a metric, the defects per
thousand lines of code added (KLOCA). They found that “Although the user-perceived defect density of the Apache project is inferior to that of the commercial products, the defect density of the code before system test is much lower”. This means that the development process used for the Apache assures that fewer bugs are inserted into the source code. In addition, inspections and tests conducted by the group of the developers are more efficient in finding bugs. The same results were found to be true for the Mozilla project.

Reasoning, a company that provides automated software inspection services for many IT companies, has made some recent reports that compare the defect density between OSS products and closed source software publicly available. The first one of these studies [114], analyzed the code of the Linux kernel component that handles the TCP/IP networking and compared it against a sample of 160 projects (22 million lines of code). The second study measured Apache2.1-dev, which was compared against a sample of 200 projects (35 million lines of code). Results for both studies are presented in Table 5.2. The last study conducted by Reasoning was on Tomcat and it was found to have a defect density of 0.24 defects/KSLC (Kilo Source Lines of Code). All the above results indicate these; industrial sized, OSS applications achieve a low defect density.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Application Average (defects/KLSC)</th>
<th>Industry Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 2.4.19 Networking Sample</td>
<td>0.10</td>
<td>&lt; 0.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.15 – 0.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 0.35</td>
</tr>
<tr>
<td>Apache 2.1-dev</td>
<td>0.53</td>
<td>&lt; 0.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.36 – 0.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 0.71</td>
</tr>
</tbody>
</table>

From the above, it can be conjectured that one of the strongest points of OSS is indeed its effectiveness in bug fixing. One of the main reasons for this effectiveness is the rapid release of the product along with its source code. This helps people to inspect and debug the code at the time they discover a bug, if they are capable of doing so, and, if they are lucky, to fix the software and send the correct version to the coordinator and from there to the whole community. Along with the availability of the source code, debugging is made effective because of the lack of any formal
debugging/testing process and most of it is done in operational mode of the system. As Vixie [115] points out, developers are friendlier when they are doing something that they are not paid for or not responsible for. The lack of pressure makes them more willing to read and fix the software and share that fix with the rest of the community. Additionally the effectiveness of debugging in OSS is achieved through the true peer review of the code. Without having to meet each other, the developers do not have any reason not to support the peer review process, thus making it more reliable [111] and more effective. One important question raised is whether OSS is more secure than proprietary software. This topic is one of most frequent subjects on debates about OSS. Indeed many people support the idea of code openness, which gives people the opportunity to inspect the code for potential flaws [116]. It is not only the algorithm that has to be open (especially cryptographic algorithms), but the code, too. It is not difficult for someone to assert that it is not only the developer that has access to the code, but the potential attacker, too. So what is the benefit? Witten Landwehr and Caloyannides [117] suggested that having closed source software, someone cannot be sure about the content of it. It is not only the source code itself, but the compiler that was used to make it executable. It is possible for the compiler to insert various malicious parts in the software for later use. Open source software let the users review their code and investigate the existence of such back doors and other kinds of flaws in software. The “Fuzz” report of the University of Wisconsin [118] showed that the reliability of the GNU and Linux software was better than that of commercial UNIX products. From all the above, the conclusion is that it is not certain in all cases that OSS is more secure than proprietary software, but the openness of the code is definitely a positive aspect.

The question of whether or not OSS is reliable was the cause for an interesting project on dependability, in the area of OSS. The project was conducted by DIRC (Interdisciplinary Research Collaboration in Dependability in Computer-Based Systems) in the United Kingdom. A definition of dependability exists on their web site: “Dependability is a systems (or "weakest-link") issue since virtually any aspect of a system, and of the means by which it was specified and designed, can cause a computer-based system to malfunction” [119]. The group that was appointed with the task of studying OSS organized an International Workshop on Open Source Development in order to gain better understanding of the OSS phenomenon. The
outcomes of that study were that “there is much variation among OSS products just as there is among proprietary software” and “that the general issues of OSS worth no further research, but the OSS development methodologies in constructing software should not be avoided” [120]. In the workshop mentioned above, [121] presented a first attempt to develop a model for the reliability improvement in OSS projects. In that study, they initially used a simple probabilistic model to understand better the relationship between software processes and software dependability. The model takes into account the probability of a user to report a bug and to fix it at the same time. Although the model looks simple (its authors admit so), it is a first step towards understanding the process of bug reporting and fixing in OSS.

5.4 Usability

Usability as a computer term means the ease with which a user can learn to operate a machine. It represents the ease with which a user can input data to a machine (a desktop computer for example), operate the machine in order to process the data and understand the output of the machine to obtain some useful result. One of the most respectful researchers in usability, [122], describes “usability as the ease of learning, efficiency of use, memorability, error frequency and severity”. In the ISO 9126 [100] model Usability is defined as “A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users”. Usability regarding OSS is another research issue and has been a matter of interest not only for the academic community, but also for the OSS movement itself. Many people within the OSS community realize that usability must be a core issue for OSS if it wants to attract a critical mass. Raymond [103] makes it clear that in the desktop market, what counts is the ergonomic design of the product and the user interface, crucial aspects for the average desktop user.

In the same vein, another OSS pioneer raises usability as one of the important issues in OSS [108]. Feller and Fitzgerald [123] state that the non-developers will focus not only on the availability of the source code, but mainly on the quality and the support of the product. OSS usability seems to be no better than that of the proprietary software and suffers from the same problems that the commercial software does. These problems stem from the inherent difference between developers and users. The
community of OSS does not pay so much attention to the usability issues that they themselves do not experience [129]. Indeed there is substantial evidence that many of the developers that drive OSS development are in the upper class of the programming profession [111]. These people cannot imagine how a system looks like for a novice user and how easy it is to operate it. Behlendorf [108] identifies that end user applications which involve GUIs, like a desktop environment, are hard to write. He states that the main causes for that are the constant changes of a program concerning GUIs, its software is rather complex and most notably, because most programmers are not so good GUIs designers.

Moreover, the tailored nature of OSS development is an obstacle to taking usability into account. Indeed OSS projects start as “a developers' personal itch” [104], i.e. developers make software to meet their own needs and other users’ needs. For example, at the time people wanted a web server and Apache was made. Developers and users in that case are essentially the same and the software depicts their own needs. Usually parties involved care for the software to meet their own demands of usability and do not take into consideration other users. This leads to a sophisticated product with a concrete and specific user interface, which is suitable for people with their own special skills. For example, many utilities concerning the Linux system do not have a modern user interface, they are command line tools, with many command line parameters to determine functionality or have a text/console based front end. Nowadays many graphical front ends have made their appearance, but for fine-tuning purposes the use of the command line in the Linux case is arduous but inevitable.

The two major Linux desktop environments, GNOME and KDE, have identified the problem of usability. Both have launched “Usability Projects” in order to investigate and improve the usability of their products. The GNOME Usability Project [http://developer.gnome.org/projects/gup/] has come up with a study [124], which was conducted in March 2001, by the usability experts of SUN's Human Computer Interaction staff. The study involved twelve people with experience on the use of computers, but with no computer science background. They used the GNOME 1.2.2 in order to perform various day-to-day user tasks, such as logging in, exploring the desktop, file management tasks, customization tasks and logging out, according to a scenario. The group of experts came up with various design recommendations to the
GNOME people about their project. Additionally, recognizing the importance of the user interface of the applications that use the GNOME environment, GNOME has issued a full text of guidelines that the applications in the GNOME environment should follow [125]. KDE and OpenOffice.org have also launched usability projects, with similar goals to those of the GNOME. Ecklund, [126] conducted another comparative usability study about F/OSS and proprietary office application. The study compared the usability of Sun StarOffice's Calc11 with Microsoft Office's 2000 Excel. The study was conducted for a course at Berkeley University. Their findings revealed that even though Excel outperformed Calc -with small differences, often statistically insignificant- 92% of the people expressed their willingness to switch to Calc, taking into account the fact that it is free.

Everitt and Lederer [127] conducted a similar study for the same course at Berkeley, but with Microsoft Word and StarOffice's Writer. Again, the results were comparative for both of the products, but the users found Word easier to use than Writer. Both of these studies are only a small evidence of assessment of the usability of F/OSS since, as mentioned, StarOffice has been developed by Sun as closed source software. Another recent usability study, which directly compares OSS GUI and Windows XP, was conducted by a commercial company, relevantive AG [128]. In that study 60 test participants performed typical tasks, regarding desktop usage, on a Linux system equipped with the KDE 3.1.2 desktop. Another group of 20 participants performed identical tasks on a Windows XP system. The main result of this study was that the usability of Linux as a desktop system has been evaluated by users as nearly equal to Windows XP. A difficulty comparison between the Windows XP and the KDE is presented in Figure 5.2.
The average time for a user to complete a specific task was in average only little behind Windows XP. The time needed by the users to complete a task in each one of the two systems is presented in Figure 5.3. The study also describes certain significant problems of the Linux system, such as partly missing clarity and structure of the desktop interface. However, the majority of the participants quite liked the system and needed a maximum of one week to acquire their previous level of competence on this system, thus making a transition to Linux look easier.
A good survey on usability of OSS is that of [129]. In addition to the OSS usability findings already discussed, the authors try to find the causes of the problems in OSS usability. They argued that it is not only the OSS developers nature, which makes them to ignore usability, but it is also the lack of usability experts in the OSS development process, a fact that inhibits OSS from having a better usability than closed source software. It seems a foregone conclusion that the nature of OSS development is not suitable for designing for usability. Usability takes place before any actual coding starts and certainly this is not the case in OSS. Usability is a quality characteristic that needs to be planned from the first stages of the software development cycle [130], [131]. In particular [126] “Usability Engineering defines the target usability level in advance and ensures that the software developed reaches that level”. High usability levels may be pursued through various usability engineering techniques, e.g. by including a user analysis task in the requirements analysis phase, by allowing for user-centered design or by building prototypes. There exists an obvious cause-effect relationship between usability-oriented techniques and perceived usability after product development. Usually, as mentioned earlier, in the case of the proprietary software, a prototype is made in order to explore its usability and the main functionality is added later. This aspect of software development is not within the realms of OSS, a fact which leads OSS to follow ideas derived from proprietary software, regardless of whether they are the best or not. In addition, as it is underlined by Nichols, usability problems are harder to specify and distribute than functionality problems. Users rarely post usability problems and it is even more difficult for someone to take over these problems, for the coordinators to assign such problems to someone and even to modularize them.

At this point it is interesting to point out that Microsoft uses a development model that at some points resembles that of the OSS case. As [132] described, Microsoft uses, as the authors call it, the “sync-and-stabilize” development method. In the ‘Microsoft model’, instead of having a large team working together in a sequential manner (like the “waterfall” development model), developers split into many small parallel teams (three to eight developers each). These small teams work together in order to build large products, while they still have the freedom to evolve their designs and operate nearly autonomously. In addition, they continually synchronize what they are doing as autonomous teams and periodically stabilize the product in increments (milestones)
rather than once at the end. These teams start with an initial set of features which they change during the development cycles. They also test features of the product as they build them, including bringing customers to try prototypes in the usability lab. The team managers monitor the whole process prioritizing features to be implemented during the next iteration, according to marketing needs, user satisfaction and the time it remains to deliver the final product. Heretofore, this style of development resembles that of OSS. It could as well be argued that this “sync-and-stabilize” approach is a kind of an agile method, but there are significant differences. One obvious difference is that Microsoft has certain schedules presupposing that the incremental versions of the product will be stabilized. For OSS this is not the case, since there are no strict release schedules. Furthermore, Microsoft uses its own usability labs; something that OSS has not the ease to do. Furthermore, usability is something that demands careful design before coding, which is rather difficult for OSS development.

One of the most important aspects of usability is localization or internationalization. The purpose of localization is to make software accessible to users who are not fluent in the language that the system uses for interaction. Software systems in most cases use English as their interaction language, discouraging non English fluent users from using that software. This part of usability is rather important for desktop application like GNOME, KDE or Mozilla. These projects have raised localization as one of their major issues and many people are working on the translation of their user interfaces and documentation. The localization of the application is an important factor for the dissemination of the OSS, a fact that has been acknowledged by many government bodies, which promote the localization of many OSS applications (for example the Greek General Secretariat for Research and Technology).

Another important factor of usability in OSS is the difficulty of installing some OSS products. Indeed many users find it difficult to migrate to Linux and the whole process of installation is sometimes rather non-user friendly [129]. Many Linux distributions expect the user to answer many technical questions and to pass strange parameters to the system. However, we have to mention that many companies have made huge efforts to simplify the whole installation process and they are continuously trying to make their product better. The lack of a standard installation process of the various OSS products is another problem and sometimes the existence of various packaging systems is confusing for novice users. Luckily, while typically there is no
formal process, there exist enough documentation on the Internet about OSS products and there are hundreds of mailing lists, in which the participants try to help each other.

5.5 Maintainability

Maintainability has to do with the easiness of code modification. According to the ISO 9126 [100] model, maintainability is “A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adoptions of software to environmental changes and changes in the requirements and functional specifications)”. Maintainability of OSS projects is a factor that was one of the first to be investigated by the OSS literature. This was done mainly because OSS development emphasizes on the maintainability of the software released. Making software source code available over the Internet allows developers from all over the world to contribute code, adding new functionality (parallel development) or improving present one and submitting bug fixes to the present release (parallel debugging). A part of these contributions are incorporated into the next release and the loop of release, code submission/bug fixing, incorporation of the submitted code into the current and new release is continued. This circular manner of OSS development implies essentially a series of frequent maintenance efforts for debugging existing functionality and adding new one to the system. These two forms of maintenance are known as corrective and perfective maintenance respectively. From the above comes the fact that maintainability is a crucial part of OSS development and thus has become a research topic in various studies.

One of these studies [133] measured 100 C programs – a total of 606095 lines of code found in a Linux distribution, using a commercial code analysis tool, Logiscope® by Telelogic Tau. The tool examines the code according to various attributes affecting maintainability, such as testability, simplicity, readability and self-descriptiveness. For each one of these factors, a number is computed taking account of several metrics such as lines of code, McCabe's Cyclomatic Complexity [134] and Halstead metrics [135]. The results are shown in Table 5.3. These results sound like the proverb of the half full, half empty glass of water. The fact that half of the modules were acceptable shows the high maintainability of the OSS. Conversely, the other 50 percent might show the bad maintainability aspect of OSS. Taken into
account the massive parallel development and the peer review of the OSS development, these findings imply that further inspection is needed for these modules.

Table 5.3: Component allocation (in %) in modules characterization

<table>
<thead>
<tr>
<th>Modules characterization</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable as is</td>
<td>50.18%</td>
<td>18.65%</td>
<td>48.37%</td>
</tr>
<tr>
<td>Require Comments</td>
<td>30.95%</td>
<td>14.09%</td>
<td>31.83%</td>
</tr>
<tr>
<td>Require further inspection</td>
<td>8.55%</td>
<td>8.50%</td>
<td>7.65%</td>
</tr>
<tr>
<td>Require further testing</td>
<td>4.31%</td>
<td>4.14%</td>
<td>3.55%</td>
</tr>
<tr>
<td>Completely rewritten</td>
<td>5.57%</td>
<td>10.73%</td>
<td>3.20%</td>
</tr>
</tbody>
</table>

Schach et al. [136] studied 365 versions of the Linux kernel. For every version of the kernel they measured the number of instances of common (global) coupling. The term coupling refers to the situation when two mutually dependent software modules interact with each other. Fewer interactions and dependences between the modules of the software can reduce the risk of a single module's fault badly affecting the others. Although there are various types of coupling, the authors decided to investigate global coupling, which happens when two modules share reference to the same global variable. As the authors underlined, it has been shown that coupling is related to fault proneness but has not been explicitly shown to be related to maintainability.

However, they argue that, if a module is fault-prone then it will facilitate maintainability actions, which directly affect software maintainability. While their measurement shown that the number of the total lines of the Linux kernel grows linearly with the kernel's version number, the number of instances of common coupling grows exponentially with version number. The latter made the authors to come to the conclusion that if this situation continues, there will be a heavy dependency among Linux kernel's modules, and therefore it will be difficult to maintain it. If this is the case with other OSS products and OSS depends heavily on common coupling, it will prove to be a negative aspect of OSS. In another study, the maintainability of the source of an open source product is compared directly with a closed source one [137]. In that study, the Maintainability Index (MI) proposed by Carnegie Mellon's SEI [138] was used in order to gain insight into the evolution of the maintainability of five OSS systems. The reason behind the use of the MI stem from
concurrency by both the authors and McConnell’s that OSS should conform to such standards and metrics. The results of the study show that OSS maintainability is no worse than that of closed source software. In direct comparison, OSS was found to be better than closed source software. A summary of the results of this study are shown in Figure 5.4.

![Fig. 5.4: MI figures for three OSS products](image)

The same authors made another study of the maintainability of an open source ERP/CRM system, using a commercial tool [137]. This system was written in Java and includes large portions of code from other OSS systems. The results, according to the quality standard denoted by the commercial tool (Logiscope®), are shown in Table 5.4. The table depicts the percentile of the application classes. It is obvious that these results suggest that the quality of the product is higher than the average. To a large extent, it is worth mentioning here that the maintainability or the code quality of the product itself is not a sufficient reason to suggest that the specific ERP/CRM system outperforms others, because functionality, which is a critical parameter in such systems, was not considered in the design of the study.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Excellent</th>
<th>Good</th>
<th>Fair</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>13%</td>
<td>51%</td>
<td>35%</td>
<td>1%</td>
</tr>
<tr>
<td>Reliability</td>
<td>75%</td>
<td>25%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Changeability</td>
<td>47%</td>
<td>21%</td>
<td>0%</td>
<td>32%</td>
</tr>
<tr>
<td>Stability</td>
<td>80%</td>
<td>17%</td>
<td>0%</td>
<td>4%</td>
</tr>
<tr>
<td>Testability</td>
<td>17%</td>
<td>47%</td>
<td>32%</td>
<td>3%</td>
</tr>
</tbody>
</table>
5.6 Portability

Portability in software is an attribute that has to do mainly with platforms and machine dependence. It expresses the ease of transferring an existing system, running on a specific machine, to another machine with a different configuration. ISO/IEC 9126 [100] describes portability as “A set of attributes that bear on the ability of software to be transferred from one environment to another (including the organizational, hardware, or software environment)”. From its early days, portability has been a central issue in OSS development. Various OSS systems have as first priority the ability of their software to be used on platforms with different architectures. Here, we have to stress one important fact, which originates from the nature of OSS, and helps portability, namely the availability of the source code of the destination software. If the source code is available, then it is possible for the potential developer to port an existing OSS application to a different platform than the one it was originally designed for. Perhaps the most famous OSS, the Linux kernel, has been ported to various CPU architectures other than its original one, the x86. Linux kernel based systems exist for PowerPCs, Alphas, SPARCs along with some other, sometimes “exotic” architectures [139]. What helps Linux to be so portable is mainly the fact that, like many other UNIX like systems, it is written in C, a third generation language, and not in a machine specific machine language. This means that, with some effort, Linux is portable to those systems where a C compiler exists. The compiler that Linux uses is the GNU gcc. This leads to the conclusion that for every machine there exists a port of the gcc compiler, Linux is portable [140].

This portability feature is stated on the main kernel site (http://www.kernel.org) as “Linux is easily portable to most general-purpose 32- or 64-bit architectures, as long as they have a paged memory management unit (PMMU) and a port of the GNU C compiler (gcc)”. Moreover, the C compiler that the kernel uses, belongs to the gcc family, the GNU Compiler Collection. This collection, which is free software, contains front ends for numerous languages, including C, C++, Java, and Fortran and can be ported to different architectures. F/OSS applications that utilize this compiler collection can be ported to different platforms with little or no effort. Another factor that made Linux portable is its own design. The creator/creators of Linux have tried to make its design as clean as possible in order to make it much portable. This was
achieved with the usage of loadable kernel modules. The existence of loadable kernel modules allows much of hardware specific functionality to be separate of the main kernel making the system more modular. These modules make it possible for the system to be installed on a huge variety of machines, with different hardware configuration, as long as a module for that hardware exists. For example, if someone tries to compile the Linux kernel, he will have to configure the kernel modules for the specific hardware that the compilation is going to take place. In the configuration, he finds a vast selection of modules for every kind of hardware, file systems, code pages and so on. For example, he may configure which file systems modules he wants to be accessed by his system (there are modules for EXT2, FAT32, NTFS, etc). Two factors about the Linux kernel portability, namely, the existence of a port of the gcc and the ability to load kernel modules resulted in the presence of Linux based machines on different platforms. For example, the Debian GNU/Linux is released for ten different ports, along with other ports without the Linux kernel.

Another aspect that shows the importance of portability in the F/OSS world is the adherence of many F/OSS software to well known standards related to portability. One of the most popular standards of that kind is the IEEE POSIX (Portable Operating System Interface) standard. Indeed, Linux kernel tries to stick as much as possible to that standard in order to allow other applications that utilize it to be portable. Developing an application, following the POSIX API standard, it is possible to compile its source code, and to run it, on different platforms. In addition, the Linux kernel includes much of the functionality of the Single UNIX Specification, Version 2 (The Open Group, 2002). The Single UNIX Specification is an attempt to integrate various standards, including the IEEE POSIX and ISO C, in order to provide a unified application programming interface. UNIX, from which Linux semantics was derived, was designed to be portable and this fact helps Linux itself to be portable. It is obvious that Linux tries to follow this specification in favor of the portability of the applications written for its system. One other important category of free operating systems, namely the BSD family (Free/Open/NetBSD), also follows the majority of the standards that Linux uses, thus making porting of the applications among Linux and the BSD family easy. The portability among the various Linux distributions is also another aspect of portability. Unfortunately many distributions add some features that prevent sometimes applications statically
compiled for one distribution to run on another. This leads to the creation of the Linux Standard Base [http://www.linuxbase.org], an initiative that has as a goal to promote portability among the various Linux distributions.

5.7 Evaluation of Open Source Software

Open Source Software (OSS) in general refers to any software whose source code is freely available for distribution. The success and benefits of OSS can be attributed to many factors such as code modification by any party as the needs arise, promotion of software reliability and quality due to peer review and collaboration among many volunteer programmers from different organizations, and the fact that the knowledge-base is not bound to a particular organization, which allows for faster development and the likelihood of the software to be available for different platforms. Eric Raymond in [17] states that “with enough eye balls, all bugs are shallow”, which suggests that there exists a positive relationship between the number of people involved, bug numbers, and software quality. Some examples of successful OSS products are Apache HTTP server and the Mozilla Firefox internet browser.

As software products have become increasingly complex, software reliability is a growing concern, which is defined as the probability of failure free operation of a computer program in a specified environment for a specified period of time [78], [16]. Reliability growth modeling has been one approach to address software reliability concern, which dates back to early 1970’s [3], [7], [19]. Reliability modeling enables the measurement and prediction of software behaviors such as Mean Time To Failure (MTTF), future product reliability, testing period, and planning for product release time. Software reliability growth models generally fall into two major classes: time between failures and fault count models. The main input parameter to the “time between failures” models is the intervals of successful operations. As the failures occur and fixed, it is expected that these intervals to increase. The pattern of these intervals is reflected by a probability distribution model whose parameters are estimated from simulation, testing, and operation profiles. Some examples that belong to this class of reliability modeling are Jelinski-Moranda and Littlewood models.

The “fault count” class labeled as such because the input parameter of study is the number of faults in a specified period of time rather than the times between failures.
Normally the failure rate, defined as the number of failures per hour, is used as the parameter of a Probability Distribution Function (PDF). Like the first class, as the fault counts drop, the reliability is expected to increase [8]. Examples of this class are Goel-Okumoto and Musa-Okumoto models [82].

This study is concerned with the “fault count” of five popular OSS products: Eclipse, Apache 2, Firefox, MPlayer OS X, and ClamWin Free Antivirus. These projects are selected because of their high number of downloads, length of project operation, and sufficient number of bug reports. MPlayer OS X and ClamWin Free Antivirus are two projects, which can be found in sourceforge.net [20]. MPlayer OS X, launched in 2002, is a project based on MPlayer, which is a movie player for Linux with more than six million downloads. ClamWin Free Antivirus was launched in 2004 that has had more than 19 million downloads. Eclipse, Apache 2, and Firefox are the other three OSS projects, which use Bugzilla as their bug repository system. Bugzilla is a popular bug-repository system that allows users to send information about a detected bug such as bug description, severity, and reporting time.

Additionally, these projects are well-known and well-established, which have been in operation for more than four years. For software projects that have not been in operation long enough, the failure data collected may not be sufficient to provide a decent picture of software quality, which may lead to anomalous reliability estimates [12].

The Weibull distribution function is employed for the reliability analysis of the aforementioned OSS projects. Weibull distribution is widely utilized in lifetime data analysis because of its flexibility in modeling different phases of bathtub reliability, i.e., decreasing, constant, and increasing failure rates. The function has been particularly valuable for situations for which the data samples are relatively small, such as in maintenance studies [13].

White-box and black-box models are two approaches for predication of software reliability. The white-box models attempt to measure the quality of a software system based on its structure that is normally architected during the specification and design of the product. Relationship of software components and their correlation are thus the focus for software reliability measurement [2], [5]. In the black-box approach, the
entire software system is treated as a single entity, thus ignoring software structures and components interdependencies. These models tend to measure and predict software quality in the later phases of software development, such as testing or operation phase. The models rely on the testing data collected over an observed time period. Some popular examples are: [79], [85], [11], [78]. This study is concentrated on the black-box reliability approach to measure and compare the reliability of the selected OSS projects.

A fault or bug is a defect in software that has the potential to cause the software to fail. An error is a measured value or condition that deviates from the correct state of software during operation. A failure is the inability of the software product to deliver one of its services. Therefore, a fault is the cause for an error, and software that has a bug may not encounter an error that leads to a failure. Failure behavior can be reflected in various ways such as Probability Density Function (PDF) and Cumulative Distribution Function (CDF). PDF, denoted as f(t), shows the relative concentration of data samples at different points of measurement scale, such that the area under the graph is unity. CDF, denoted as F(t), is another way to present the pattern of observed data under study. CDF describes the probability distribution of the random variable, T, i.e. the probability that the random variable T assumes a value less than or equal to the specified value t. In other words,

\[
F(t) = P(T \leq t) = \int_{-\infty}^{t} f(x) \, dx \Rightarrow f(t) = F'(t)
\]

Therefore, f(t) is the rate of change of F(t). If the random variable T denotes the failure time, F(t), or unreliability, is the probability that the system will fail by time t. Consequently, the reliability R(t) is the probability that the system will not fail by time t [16], i.e.:

\[
R(t) = P(T > t) = \int_{t}^{\infty} f(x) \, dx \Rightarrow R(t) = 1 - F(t)
\]

This chapter will show that the relative frequency of failure times follows the Weibull PDF:

\[
F(t) = \frac{\beta}{\alpha} t^{\beta-1} e^{-\left(\frac{t}{\alpha}\right)^\beta}
\]

Where \( \alpha \) is the scale parameter and \( \beta \) represents the shape parameter of the distribution. The effect of the scale parameter is to squeeze or stretch the distribution.
The Weibull PDF is monotone decreasing if $\beta \leq 1$. The smaller $\beta$, the more rapid the decrease is. It becomes bell shaped when $\beta > 1$, and the larger $\beta$, the steeper the bell shape will be. Furthermore, it becomes the Rayleigh distribution function when $\beta = 2$ and reduces to the exponential distribution function when $\beta = 1$. Figure 5.5 shows the Weibull PDF for several values of the shape parameter when $\alpha = 1$ [8].

![Weibull PDF for several shape values when $\alpha = 1$.](image)

The reliability function of Weibull distribution is [15]:

$$ R(t) = e^{-\left(t/\alpha\right)^\beta} $$

(2)

### 5.8 Results

This section will highlight the implication of power exponential model over the analysis of the software reliability using the case study of OFBiz. The approach to the reliability estimates of the selected OFBiz projects consists of three steps: bug-gathering, bug filtering, and bug-analysis. In the bug-gathering step, the online bug-repository systems are used to collect the failure data. Quality estimation of an OFBiz product depends on sufficient error reports and the accuracy of reports provided by the customers using the product. Although the collected reports may differ among bug-repository systems, the following fields are considered:

- **Bug-ID** - A unique identification for each bug reported.
- **Bug-Time** - The actual time of bug reported.
- **Bug-Resolution** – The state of a bug determined by the organization, such as whether the bug reported is valid, deleted, or fixed.
- **Bug-Reporter** – Information about the user submitting the bug report.
- **Product** – The specific product name or component in which a bug is detected.
Bugs reported may be duplicates, provide incomplete information, or may not represent real defects. Therefore, during the bug-filtering, such noises are removed from the bugs gathered in the first step. Finally, in the third step, the filtered data is organized into bug-frequencies for fixed time periods, which are set at two weeks. Observing the volume of bug-reports and the varying years of operations among the products, the choice of two-week periods provides sufficient number of failure frequencies that captures the pattern of bug reports. The bug-frequencies are then plotted against the time periods for further analysis in terms of graph fitting, comparison among the products, and reliability.

A. Bug Gathering and filtering

The duration for which the failure data is collected for the six features of OFBiz project are listed in Table 5.5.

<table>
<thead>
<tr>
<th>Project name</th>
<th>Start date</th>
<th>End date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Parameters</td>
<td>2009</td>
<td>2010</td>
</tr>
<tr>
<td>Service Security and Access Control</td>
<td>2010</td>
<td>2010</td>
</tr>
<tr>
<td>Job Scheduler</td>
<td>2010</td>
<td>2010</td>
</tr>
<tr>
<td>Service Reference</td>
<td>2011</td>
<td>2011</td>
</tr>
<tr>
<td>Event Condition Action</td>
<td>2011</td>
<td>2012</td>
</tr>
</tbody>
</table>

Table 5.6 illustrates a portion of a bug report collected using FindBug stored. In the table, “Request ID” is Bug-Id; “Summary” is a short description of the problem reported; “Open Date” is the same as Bug-Time; “Priority” shows the level of bug severity, which is a value in the inclusive range of 1 – 9; “Status” is Bug-Resolution; and “Assigned To” is the person or the team that will be assigned to look into this specific bug. The “Status” field may carry different values such as “Open”, “Closed”, or “Deleted”. The “Open” value is an indication that the bug has not been fixed yet; “Closed” reflects that the bug has been fixed, and “Deleted” means that the bug reported is not a valid one.
Table 5.6: A portion of a bug report using FindBug

<table>
<thead>
<tr>
<th>Request ID</th>
<th>Summary</th>
<th>Open date</th>
<th>Priority</th>
<th>Status</th>
<th>Assigned to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1570891</td>
<td>Catalogue management is difficult to trace</td>
<td>2\textsuperscript{nd} Feb, 2010</td>
<td>5</td>
<td>Deleted</td>
<td>Self</td>
</tr>
<tr>
<td>1565875</td>
<td>Pricing management application not referenced</td>
<td>14\textsuperscript{th} April, 2010</td>
<td>5</td>
<td>Closed</td>
<td>Self</td>
</tr>
<tr>
<td>1565868</td>
<td>Invoice, payment and billing accounts, fixed assets found iterative data in database</td>
<td>21\textsuperscript{st} Sep, 2011</td>
<td>5</td>
<td>Closed</td>
<td>Self</td>
</tr>
<tr>
<td>1564167</td>
<td>Content management with faulty customization</td>
<td>28\textsuperscript{th} Dec, 2011</td>
<td>5</td>
<td>Closed</td>
<td>Self</td>
</tr>
<tr>
<td>1531952</td>
<td>Difficult to initiate service call from heterogeneous OS</td>
<td>19\textsuperscript{th} April, 2012</td>
<td>5</td>
<td>Open</td>
<td>Self</td>
</tr>
</tbody>
</table>

The bug reports extracted from FindBug are initially in XML format. A Java program is developed to gather the relevant data from the XML format for further data filtering and analysis. Figure 5.6 shows a portion of a bug report in XML stored at FindBug.

```xml
<bug>
  <bug_id>366101</bug_id>
  <creation_ts>2011-01-05 16:41 PST</creation_ts>
  <short_desc>nsIFile.initWithPath should accept "c:/mozilla" as native path (forward slashes should be treated as backslashes)</short_desc>
  <delta_ts>2011-01-05 16:57:22 PST</delta_ts>
  <reporter_accessible>1</reporter_accessible>
  <cclist_accessible>1</cclist_accessible>
  <classification_id>3</classification_id>
  <classification>Components</classification>
  <product>Core</product>
  <component>XPCOM</component>
  <version>Trunk</version>
  <rep_platform>PC</rep_platform>
  <op_sys>Windows XP</op_sys>
  <bug_status>NEW</bug_status>
  <priority>--</priority>
  <bug_severity>normal</bug_severity>
  …
  <who name="Awtar">awtar@cbkinfotech.com</who>
  <bug_when>2011-04-13 16:16:07 PST</bug_when>
  …
  <bug_status>VERIFIED</bug_status>
  <resolution>WORKFORME</resolution>
</bug>
```

Fig. 5.6: A portion of a bug report in XML at FindBug
The fields in Figure 5.6 are self-explanatory, like “Bug_id” is Bug-ID and “Who_name” is Bug-Reporter. The “Resolution” field is Bug-Resolution, which may hold different values:

- **FIXED** – This bug is a valid one and it is fixed.
- **INVALID** – This bug was not a valid one.
- **WONTFIX** – This bug will not be fixed.
- **WORKSFORME** – This bug cannot be reproduced.
- **INCOMPLETE** – Information provided is not sufficient for producing the bug.
- **LATER** – This bug will not be fixed in this version.
- **REMINDE** – This bug will not probably be fixed in this version.
- **DUPLICATE** – This is a duplicate bug reported earlier.

The bug reports collected are then filtered out. For the two projects from Sourceforge.net, the bug reports with the status value of “Deleted” are discarded. For the failure data collected from Bugzilla, those bug reports with the following “Resolution” values are accepted and the rest are discarded: FIXED, WONTFIX, LATER, and REMIND.

### 5.9 Error Analysis using FindBug

The proposed research is experimented on OFBiz, which is a general purpose framework for building software that controls common business processes. Unlike many systems which provide abstract software structures, OFBiz attempts to supply a set of data structures and software libraries for managing more concrete business elements such as orders, employees, contracts, and manufacturing processes. In the bug-analysis step, the frequency of bugs in two week periods is estimated. Therefore, the x-axis and y-axis represent the biweekly time and the corresponding bug frequency, respectively. All the Figures show the bug frequency plots for the error testing of OFBiz projects.
The various field of OFBiz evaluated under FindBug for Catalogue Management are product catalogue ID, Catalogue Name, Use Quick Add, Style sheet, Header Logo, Content Path Prefix, Template Path Prefix. The analysis shows that faulty fields of Template Path Prefix as well as Content Path Prefix. The prime cause for this is unstructured class files found with similar interfaces that need to be highly customizable depending on the access and administrative rights of any operating system. Another set of the bug found in this section is contemporary catalogue, where it was found that user cannot retrieve the previously explored results. The prime cause found here was defective usage of java interfaces for which the application loads properly but fails to work appropriately due to breakdown in service calls.

Fig. 5.7: Evaluated bug frequency for Catalogue Management

Fig. 5.8: Evaluated bug frequency for Pricing Management
The above case was found mainly in accounting manager application. Various fields under evaluation by FindBug are Agreement ID, Product ID, Party ID (from/to), Agreement ID type, Dates, and Description. According to analysis of FindBug, the error captured here are mainly the Agreement ID and its confiscated Description. Due to dual usage of AgreementID as java class, however, the error was found. But the most critical error found here was mainly the description that was mainly using all string characters while using service initiation. However, the logical parameters that are in use in this application are working fine. The bugs are comparatively less in number but debugging basically consumes lots of time as this module is basically connected with all the major modules like invoice, payments, fixed assets etc in OFBiz.

![Graph](image)

**Fig. 5.9: Evaluated bug frequency for Content management**

The major fields used for evaluation by FindBug for content management are Data resources, sub-content pattern, decorator templates, access content information, and controller.xml file. The cumulative bugs found in this part of evaluation are very high in number (801). The content management application is basically connected with product management class. Although the products management is non-sophisticated for any user, but massive bugs are found in content management due to constant upgradation and version control mechanism in Java. Whenever a tester performs testing and debugging on any other part of the application that are connected to this URI, the errors shoots up thereby showing main errors caused due to faulty reference calls.
Fig. 5.10: Evaluated bug frequency for version

This set of evaluation is for understanding the errors trapped in various versions of OFBiz. For the purpose of evaluation, the initial version released on 2010 (ver-1), and 2012 (ver-2) is used; implication of the model generates a new version that is user specific application (ver-3). The initial version was found with maximum level of bugs, although it does not seem to diminish even in the current version. However, implication of the model seems to greatly reduce the defects that are encountered in the entire previous version. In the above figure 5.10, ver-1 is evaluated for 0-20 days, ver-2 from 21st to 40th day, and ver-3 from 41st to 60th day.

Fig. 5.11: Evaluated bug frequency for Billing Accounts

This is one of the prime defects found in both the versions of OFBiz using FindBug. The prime bug found was that OFBiz application don’t stop user from taking an order for value more than available line of credit. Another prime flaw observed is that even after payment is done for the billing account; it does not increase line of credit. The
major root cause explored in this bug is highly hierarchical data structure with maximum assumptions unknown to the user. Although initial level of the application (keeping track of credit, payments, multiple authorization) are working fine. But, the core billing account seems to show defective and unreliable values that are normally not traced by the user in initial usage period. The complexity is it increases with time showing faulty loop structure used in programming. One of the critical application point of OFBiz is therefore found with some major bugs that need to be instantly evaluated and repaired.

5.10 Error Filtered after using proposed Model

![Fig. 5.12: A curve fitted onto bug frequencies for OFBiz](image)

This pattern is supported by large body of empirical studies in that software projects follow a life cycle pattern described by Rayleigh distribution function, a special kind of Weibull distribution with shape parameter $\beta = 2$. This is considered a desirable pattern since the bug arrival rate stabilizes at a very low level. In closed source software, the stabilizing behavior is usually an indicator of ending test effort and releasing the software to the field. This pattern also is supported by Musa-Okumoto model in that the simple bugs are caught easily at the beginning of testing phase. The remaining bugs tend to be more difficult to detect because, for example, they are not exercised frequently. Therefore, the rate of undetected bugs drops exponentially as testing continues.

On a quick glance at Figure 5.10, various versions does not seem to follow this pattern. When bug reports are scrutinized in more detail, it is found that the versions separately follow a similar pattern. Figure 5.13 illustrates the bug frequencies for
individual constituents of OFBiz (Service Parameters, Service Security and Access Control, Job Scheduler, Service Reference, Event Condition Action) releases superimposed in one diagram, rather than lumping the bug frequency for all versions as in Figure 5.10.

**Fig. 5.13: Filtered bug frequencies of OFBiz for different constituents**

In this figure, the peak in each constituent shows the diminished bug after implying the proposed model. In reality, the users started using the unofficial (beta) version and reporting the bugs before the official release of the version. For example, OFBiz initial version is officially released in 2010, but customers started downloading it after that. In other words, the bug reports for current version are mixed with those of an earlier version. When the bug reports are correctly extracted based on the version numbers, each version seems to follow the same pattern as those of constituents e.g. Service Parameters, Service Security and Access Control, Job Scheduler, Service Reference, Event Condition Action.

Figure 5.14 shows the bug frequencies for OFBiz extracted from Figure 5.13. It is this version that will be used in reliability analysis of OFBiz because of its high bug reports in comparison to other versions. However, all versions of OFBiz will later be compared against each other.
The OFBiz Project is a freely available package that is used for a wide variety of statistical computing and graphics techniques. OFBiz is able to apply the Maximum Likelihood Estimation (MLE) technique for estimating the parameters of Weibull distribution. Since OFBiz requires time domain data, the relative frequency of bug reports needs to be converted to occurrence times of failure. Therefore, each bug report is mapped to its corresponding biweekly period. For example, 4 bugs reported in the 1st biweekly and 3 bugs reported in the 2nd biweekly periods are converted to: 1,1,1,1,2,2,2. This further illustrates that the total number of failures at the k\textsuperscript{th} position in the list, which implies that the input provided to OFBiz is cumulative.

The computed shape and scale for each OSS features are listed in Table 5.7. As indicated previously, the effect of the scale parameter is to squeeze or stretch the PDF graph. The greater the value, the greater the stretching. Therefore, the larger the value, the flatter the curve of PDF will be, which implies lower rate of failure. The correlation coefficient is a measure of strength of the match between the fitted curve and the bug frequencies pattern. The closer the coefficient value is to one, the stronger the match is.

<table>
<thead>
<tr>
<th>Project name</th>
<th>Scale</th>
<th>Shape</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-commerce</td>
<td>67.93</td>
<td>1.17</td>
<td>0.95</td>
</tr>
<tr>
<td>Catalogue management</td>
<td>172.68</td>
<td>10.77</td>
<td>0.99</td>
</tr>
<tr>
<td>Pricing Management</td>
<td>31.5</td>
<td>1.33</td>
<td>0.99</td>
</tr>
<tr>
<td>Order management</td>
<td>31.36</td>
<td>1.23</td>
<td>0.99</td>
</tr>
<tr>
<td>Customer management</td>
<td>49.88</td>
<td>0.94</td>
<td>0.90</td>
</tr>
<tr>
<td>Warehouse management</td>
<td>61.06</td>
<td>1.72</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 5.7: Parameter Estimates for Selected Features
<table>
<thead>
<tr>
<th>Project name</th>
<th>Scale</th>
<th>Shape</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>76.82</td>
<td>3.64</td>
<td>0.91</td>
</tr>
<tr>
<td>Content management</td>
<td>91.14</td>
<td>6.14</td>
<td>0.93</td>
</tr>
<tr>
<td>Billing account</td>
<td>120.19</td>
<td>8.99</td>
<td>0.96</td>
</tr>
<tr>
<td>Invoice and payments</td>
<td>144.82</td>
<td>15.74</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Figure 5.15 shows the fitted graphs for all the major features in OFBiz. From the figure, the estimated PDFs are good representations of failure behavior. Among these, OFBiz versions have the lowest coefficient values. After some experimental analysis, the reason is due to a sharp increase of bug reports over a short period of time (about 10 biweeklies) in comparison to the measurement scale, which is about 160 biweekly periods. Since the increase and span of failures are correspondent to the shape and scale parameters, respectively, the estimates are favored toward the scale to cover a larger span of time. This implies the peak of the fitted graphs was pushed down, cutting off a portion of bug reports. The precise use of maximum likelihood using neural network technique and Markov chain Monte Carlo approaches are basically used for the evaluation of the exponential power framework. All the bugs that were gathered are basically evaluated using FindBug and then proposed model is used to analyze the intensity of software reliability. All errors were redesigned using new functions in order to study various statistical characteristics of the features in OFBiz.

Fig. 5.15a: Estimated PDF for Catalogue management.
Fig. 5.15b: Estimated PDF estimation for pricing management.

Fig. 5.15c: Estimated PDF estimation for Billing Account.
Reliability of the OFBiz can be calculated by inserting the shape and scale parameters from Table 5.7 into the Weibull reliability function. Figure 5.16 exhibits the reliability graphs for the five OFBiz Constituent (Service Parameters, Service Security and Access Control, Job Scheduler, Service Reference, and Event Condition Action). They all show a reliability of 0.90 and 0.80 over 4 and 8 biweekly periods, which are about 2 and 4 months of operation, respectively. Among these products, Service
Security and Access Control has the highest reliability. On the other hand, the figure shows that Service Reference, which almost completely overlaps Job Scheduler, has the least reliability. After Service Reference and Job Scheduler, the graph of OFBiz shows the next lowest reliability. OFBiz could have performed worse if it were not for the future releases. More specifically, with a new release, the customers opt not to use the older versions. Hence, the number of downloads and bug reports would be reduced. Although this may not change the shape of the graph, it would likely reduce the steepness of the graph.

![Graph showing software reliability for different features of OFBiz](image)

**Fig. 5.16: The Software reliability for the five OFBiz Constituent**

Table 5.8 shows the individual software reliabilities for different time periods for each OFBiz constituents.

<table>
<thead>
<tr>
<th>Project / Biweekly periods</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalogue Management</td>
<td>0.95</td>
<td>0.90</td>
<td>0.84</td>
<td>0.79</td>
<td>0.73</td>
</tr>
<tr>
<td>Content Management</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>Pricing Management</td>
<td>0.92</td>
<td>0.80</td>
<td>0.69</td>
<td>0.58</td>
<td>0.48</td>
</tr>
<tr>
<td>Billing Accounts</td>
<td>0.90</td>
<td>0.78</td>
<td>0.67</td>
<td>0.56</td>
<td>0.47</td>
</tr>
<tr>
<td>Invoice and Payments</td>
<td>0.89</td>
<td>0.80</td>
<td>0.72</td>
<td>0.65</td>
<td>0.59</td>
</tr>
</tbody>
</table>

Figure 5.17 compares the reliabilities for different features of OFBiz (Catalogue management, Pricing Management, Order management, Customer management, Accounting, and Billing account). As expected, as billing accounts and their reliabilities are improved. The latter three features show a reliability of at least 0.90
for up to 60 biweekly periods, which is over two years of operation. The figure further indicates that the Billing Account has a reliability of 0.99 for 108 biweekly periods.

Bug reports are the source for deeper understanding of failure distribution, classifying failures, and building accurate dependability models. The quality of bug analysis heavily depends on comprehensive and accurate recording of bug reports. The lack of a commonly accepted data format for archiving bug reports and efficient algorithms for data filtering adds to the complexity of failure data analysis.

In this experiment, two different trends of bug reports for the five OSS projects are visually discernible. The first group of products, i.e. Service Parameters, Service Security and Access Control, Job Scheduler, Service Reference, Event Condition Action show the trend of a peak followed by a long decreasing tail, which is a good candidate to be fitted by the Weibull distribution function. The second group, which encompasses Job Scheduler and Service Reference, shows multiple peaks instead of one particular peak.

![Image](image.png)

**Fig. 5.17: The reliability for different features of OFBiz**

These products are more complex to analyze and to be modeled. The complexity can be explained by different factors such as coinciding of multiple versions and enclosure of different components or partial products in the main product. Although,
it is likely that a PDF can be fitted, the chances for lower coefficient values for such products are higher, and the reliability growth estimates can be obscured. On the other hand, estimating reliability with least overlap among versions can provide a good sense of software quality over a period of time.

Although higher reliability may be provided by later versions, care must be exercised when comparing them against products started from scratch. This is due to the fact that the reliability of later versions could be bloated as the result of borrowing many components from the earlier versions with detected and fixes of many prior defects. When analyzing OFBiz, the bug reports include the version numbers so that their associated bug reports can be extracted. But the same could not be said about any other open source ERP package.

Because of time invested in collecting and mining the failure data, one avenue of future research is to investigate the reliability growth of these products with respect to some models of Non-Homogeneous Poisson Process (NHPP). The main issue in NHPP models is the determination of an expected value for the number of failures up to a point in time [6], [16].

**Fig. 5.18: Variation of software reliability with time**
Figure 5.18 shows the variation of reliability with time for the hardware half and software half separately and additionally the reliability values when the software and hardware are considered about along. It is apparent that if hardware and software don't seem to be treated along within the reliability analysis, the anticipated values are an overestimate and can be removed from realistic. A live of the error concerned within the reliability calculations can even be obtained by comparison. Figure 5.19 may be a comparison of the software reliability obtained using the developed model and also the standard Musa model. It is seen that the variation is nearly nil.

5.11 Comparison with other ERP Open source

This section evaluates the most frequently used Open source ERP named as “Seam” and OFBiz frameworks.

A. ERP-Specific Criteria

1) Technology complexity: We find that JBoss Seam requires developers to clearly understand Java EE technology, but OFBiz does not require deep knowledge of any one technology. Seam developers need skills in JSF, EJB3, and JPA. In contrast, for OFBiz, a basic knowledge of the architecture of the framework is sufficient for getting started. In the data source layer in particular, OFBiz’s Entity Engine is straightforward for any developer familiar with relational database
technology, but JPA requires in-depth knowledge of object technology and object-relational mapping. Although Seam’s complexity may be necessary for a large project, OFBiz technologies are easier to understand and well tailored to smaller ERP projects.

2) **Ease of service exposure:** In our prototype, we found that OFBiz provides simpler means for exposing business logic as SOAP Web services. As already mentioned, OFBiz forces developers to adopt a service-oriented interface to the business logic layer. Once an OFBiz service has been defined, it is a simple matter of adding the attribute export="true" to the XML service definition:

```xml
<service engine="java" name="calculateDepre"
    location="org.ofbiz.ams.DepreciationServices"
    invoke="calculateDepre"
    default-entity-name="AssetExts"
    auth="false" export="true">
</service>
```

With the export attribute turned on, OFBiz automatically creates a WSDL specification for the service. In Seam, to export existing session beans’ methods as Web service operations, developers must create a separate Java interface declaring the methods that should be exported then add two annotations: @Remote and @WebService(name = "...", serviceName = "..."). Like OFBiz, Seam automatically creates the appropriate WSDL specification. We conclude that although the Seam solution would be better when developers want a coarse-grained remote interface and a fine-grained local interface to the same business logic, the OFBiz approach is simpler.

3) **Ease of administration:** Both frameworks require the Java Development Kit (JDK). OFBiz ships with an embedded version of the Tomcat application server, but Seam requires separate installation of the application server and some additional configuration to notify the server the Seam should be loaded before it is ready to run. At application deployment time, OFBiz is capable of dynamically loading controllers, screens, widgets, forms, and business logic implemented by
scripts. This saves development time and deployment time. OFBiz makes installation and deployment easier.

4) **Resource utilization**: In the standard setup, Seam runs on the JBoss application server, which provides complete EJB3 support, whereas OFBiz, by default, runs on the more lightweight Tomcat application server. In our case study, we found that the Seam installation required 335.2 MB of disk space and 85808 KB of RAM at runtime, whereas the OFBiz installation required 193.2 MB of disk space and 75820 KB of RAM. OFBiz is therefore more lightweight than Seam in terms of resource utilization.

5) **Ease of presentation implementation**: Both Seam and OFBiz utilize the Model-View-Controller (MVC) pattern in the presentation layer. For Seam, developers must be familiar with JSF and EJB to implement presentation layer components. OFBiz developers need only know how to use the Entity Engine, Service Engine, screens, form widgets, and the built in templates. In developing our prototype AMS systems, we found that the learning for JSF is steeper than that for OFBiz. However, experienced Java developers already familiar with JSF but not with the OFBiz framework would presumably find the OFBiz learning curve steeper.

6) **Ease of business logic implementation**: OFBiz makes business logic layer services available to many clients automatically, leaving the developer with the relatively straightforward task of plugging in small segments of reusable business logic code in Java or a scripting language. In our case study, we found that Seam, on the other hand, sometimes requires more hand coding than OFBiz for the same task.

7) **Ease of database administration**: Both Seam and OFBiz provide means to connect with several different database management systems. In both frameworks, changing the database is a simple matter of modifying a configuration file to point to the correct data source. However, the OFBiz framework does have the advantage that developers can create, import, export, and seed data in an XML file format using a simple Web interface.
B. General Criteria

1) **Availability:** We have not been able to find any data on the relative failure rates of the Tomcat and JBoss application servers. However, we do find that JBoss requires server restarts more frequently than OFBiz when deploying new versions of the code. This means that OFBiz provides slightly more availability than Seam.

2) **Customizability:** JBoss Seam is in a sense more customizable than OFBiz in that it does not restrict the developer in any way. OFBiz developers must adapt to its style. On the other hand, OFBiz provides many customizable features, for messaging, database vendor, or work flow with XPDL standard. OFBiz has many common services that can be invoked in any application. The developer only needs to change a simple XML specification to switch the service endpoint to a different component. Therefore, it is very easy to customize OFBiz if what the developer wants to do fits into the OFBiz framework. Since OFBiz is optimized for the common cases in ERP development, in most cases there will be a fit.

3) **Modifiability:** In Seam, the application architecture is up to the developer. This means that the modifiability of the final system depends on the developer’s skill. OFBiz, on the other hand, requires developers to use a service-based organization that forces them to build a loosely-coupled system. Since OFBiz is loosely coupled by design, it encourages modifiability more than Seam does.

4) **Integrability:** As previously explained, OFBiz makes it simple to expose any business logic service as a SOAP Web service endpoint. In this simple case study scenario, OFBiz

5) **Exhibited more integrability than Seam.** However, we should note that although OFBiz is the better framework according to this test, we have only tested one small aspect of integration. Other types of integration requiring standard Java APIs like JMS would not be advantageous for OFBiz, and integration tasks that benefit from JBoss’ special features, e.g. jBPM, would put OFBiz at a disadvantage.
6) **Flexibility**: OFBiz is more flexible than Seam framework because it has many management and support tools for tasks such as user interface development, web service exposure and database setup. With the OFBiz framework, we have more choices of scripting languages.

### 5.12 Summary

The success of OFBiz has been mostly attributed to the speed of development and the reliability, portability, and scalability of the resulting software. In turn, these qualities are attributed to three main issues, namely the fact that developers are usually also users of the software, the public availability of the source code, and the fact that developers are members of a community of developers. First, OFBiz projects often originate from a personal/corporate need. Such needs attract the attention of other user-developers and inspire them to contribute to the project. This approach to software offers some real benefits in the design process. Since developers are users of the software, they understand the requirements in a deep way. As a result, the ambiguity that often characterizes the identification of user needs or requests for improvement in the traditional software development process is eliminated: programmers know their own needs.

Second, in OFBiz projects, the source code is open to inspection by and contributions from any interested individual. Therefore, users can also be developers. If they find bugs, they can fix them themselves rather than having to wait for the developers to do so; if a specialized feature is needed, it can be added, even if it is not one that the developers feel is cost-justified. As a result, OFBiz bugs can be fixed and features evolved more quickly.

Finally, developers are part of a community. The OFBiz community represents a nexus of exchanges in which people report bugs expecting that other members will fix them. Similarly those who fix bugs expect other developers to contribute to other parts of the project. Reputation is another important aspect—the community is in fact frequently described as being based on peer recognition and in some cases on a “cult of the personality”. In particular, peer recognition is a value for the community that can sometimes lead to employment opportunities or access to venture capital. In such
an environment, developers may be motivated to do the best work they can, rather than anonymously finishing code so it can be shipped.

The main implication of the three characteristics described above is that OFBiz software engineering processes have evolved to develop software that meets developers’ needs. On the other hand, OFBiz, with its reliance on self-interested developers, may be less well suited for developing applications that address problems that developers tend not to face. We see very good OFBiz tools for software development and good end-user tools for issues faced by developers (e.g., email, word processing), for example, but would expect to see few OFBiz applications for problems developers rarely face (e.g., accounting, textual analysis).

There is some empirical support for this limitation to the OFBiz software engineering process. In our analysis of projects supported by SourceForge, for example, we found fewer projects for business and specialized topics. Furthermore, these projects tended to be in earlier stages of development and less used. Therefore, for the OFBiz model to work for a broad class of applications, projects need mechanisms to address the potential divide between developers and non-developers.