Life cycle of Defects in Open Source Software Projects

Timo Koponen
1 Department of Computer Science, University of Kuopio
P.O.B 1627, FIN-70211 Kuopio, Finland
timo.koponen@uku.fi

Abstract. We studied the maintenance process from the viewpoint of defect management and the defect life cycle. First, we outline a model for the defect life cycle based on ISO/IEC standards, the Framework for Open Source maintenance process, and the Bugzilla defect management system. Thereafter, we analyze defects from two Open Source software projects. The aim of the study was support the maintenance reliability. However, we found that most of the defects did not follow the life-cycle model. Defects were usually directly resolved from initial state without being assigned.

1 Introduction

Most Open Source software users are not software developers or programmers and they are rarely able to modify or repair software. So it is hard to imagine that software will be adopted if users do not have confidence in the software itself and in the maintenance provided Open Source users are often encouraged to report defects and request enhancements, and for this they need a channel to communicate with developers. Many projects use dedicated systems such as Bugzilla [1] for defect reporting and management. These systems provide a communication channel and a system for maintenance process management.

Nowadays, the reliability of the maintenance process is based on a well-described process. A standard model of the maintenance process is presented in the Software Engineering Body of Knowledge (SWEBOK), ISO/IEC 12207 and ISO/IEC 14764 standards [2], [5]. These ISO/IEC Maintenance standards describe the activities required and their inputs and outputs [2], [5], but it is not known if the standard model is applicable for Open Source maintenance. In our earlier studies, we described a framework for the Open Source maintenance process [4] and found it similar to the ISO/IEC Maintenance standard.

However, a well-defined process does not provide reliability if it is not followed. In fact, it is not known if the defects in Open Source projects follow the described process. This study explored defect management and the life cycle of defects in Open Source projects. Our first objective was to define a model for the life cycle of the defect. The second objective was to find the most common life cycles from the case studies and compare them with the life-cycle model. The third objective was to evaluate reliability of the maintenance process by analyzing life cycles. The rest of the article is organized in the following way. Section 2 presents background of the study. Section 3 introduces the case studies and data. Section 4 explains and analyzes

Please use the following format when citing this chapter:

Koponen, T., 2006, in IFIP International Federation for Information Processing, Volume 203, Open Source Systems, eds. Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G., (Boston: Springer), pp. 195-200

the results of the case studies. Section 5 presents related work and Section provides a brief conclusion.

2 Background

Defect management systems (DMS) allow users to report problems, bugs or enhancement requests as a defect. They also provide flexible possibilities to track, control, and assign defects. These features allow the maintenance process to be managed. Defect management systems present defects as defect reports.

A defect report contains many attributes but we focused on analyzing the attributes *state* and *resolution of the defect*. The *state* describes the defect's condition, such as new or resolved. In the Bugzilla defect management system defects can be in the seven states presented in Table 1. It is not allowed to transit between all states directly: for example, it is not allowed to transit from *closed* to *new*. To illustrate the allowed state changes we drew a state transition diagram (Figure 2), which presents the allowed state transitions in the Bugzilla defect management system.

Table 1. States of the defect in the Bugzilla defect management system

State	Explanation
Unconfirmed	Defect has been recently added and it is not confirmed yet.
New	Defect has been recently added and others have confirmed it.
Assigned	Defect has been assigned to proper person.
Resolved	Defect has been resolved but it is in quality assurance.
Verified	Defect has been resolved and accepted by quality assurance.
Closed	Defect has been resolved, verified and closed
Reopened	Defect was resolved but now it has been reopened for some reason.

A defect should be resolved even it does not lead to changes or modification of software. *State* does not describe the outcome of the defect so *resolution* is needed to express this . Earlier studies have shown that many of the resolved defects do not cause changes to software [3]. Table 2 shows the *resolutions* that are possible in the Bugzilla system. Figure 2 and Framework for Open Source Maintenance process [4] show that the most common defect life-cycles should be similar to those presented in Table 3. Some of the defects can be classified as duplicate or invalid immediately and they can be resolved without assignment. On the other hand, a defect that leads to changes in the source code should always be assigned.

Table 2. Resolutions of defects in the Bugzilla defect management system

Resolution	Explanation
empty	Defect does not have resolution yet
Fixed	Defect is fixed and changes have been made
Works for me	Defect does not occur in other users' systems
Won't fix	Defect is not a fault or real problem; or it is a feature
Invalid	Defect is invalidly reported or information is missing
Duplicate	Defect is a duplicate

Table 3. Expected defect life-cycles

Resolution	Life-cycle
Fixed	Unconfirmed→New→Assigned→Resolved→Verified→Closed
Other	Unconfirmed→Resolved→(Verified)→Closed

3 Case studies

To study the life cycles of defects in the real world, we collected and analyzed defects of the Apache HTTP Server and Mozilla Firefox. These are widely used and their quality is highly appreciated so they are representative case studies. We selected a two-year time-period for analysis, and all the defects reported between September 2003 and September 2005 were analyzed. This sampling produced 1266 defects from Apache and 27681 from Mozilla. The resolutions of the analyzed defects are presented in Table 4.

Table 4. Resolution of the defect in Apache and Mozilla

Resolution	Duplicate	Fixed	Invalid	Won't fix	Works for me	Later	Remind	Not resolved
Apache	162	288	370	84	33	5	1	323
Mozilla	10038	2414	3404	714	3730	0	0	7381

Table 4 shows that Apache had 943 resolved defects and Mozilla had 20 300 resolved defects. However, not all of the resolved defects led to a change or modification of software. In the case of Apache, 288 defects (less than 31 percent of all resolved defects) ended up *fixed,* and in the case of Mozilla 2414 defects (less than 12 per cent of resolved defects) ended up *fixed.* Furthermore, there were also two additional resolutions, Remind and Later, in the Apache. Those states were rarely used.

However, the final state and resolution does not explain defect processing and the defect management process so we analyzed the life cycles of the defects in both case studies. Table 5 presents the most common defect life-cycles in the Apache and Mozilla projects.

Table 5. Two most common defect life-cycles in the Apache and Mozilla projects

Apache		Mozilla	
247	New→Resolved	3764	Unconfirmed→Resolved→Verified
511	New→Resolved→Closed	11133	Unconfirmed→Resolved

A direct transition from the state *new* to the state *resolved* is the most common life-cycle of defects in the Apache project. There was no significant use of the state *unconfirmed.* However, according to Bugzilla [1] *unconfirmed* should be the initial state of the defect. Furthermore, there was also a new state, *needinfo,* meaning that the defect report did not contain all the necessary information.

A direct transition to the state *resolved* is also very common in Mozilla. However, in this case, it was usually from *unconfirmed* to *resolved.* In addition, it seems to be

very uncommon to close a defect, so most defects end up *resolved*. Furthermore, we found state transitions that were not allowed, such as a transition from *verified* or *resolved* to *unconfirmed*. However, all state transitions were allowed in the Apache project.

4 Results

As the cases in the previous section show, the defect life cycles do not correlate with the life-cycle model and the state of the defect transits almost directly to the state *resolved*. Furthermore, Table 4 also shows that most of the defects did not lead to a change or modification of software. In case of the Apache, over 800 of 943 defects transited directly from the state *new* to the state *resolved*. The most common life cycle in the Apache project is presented with bold black line in Figure 2.

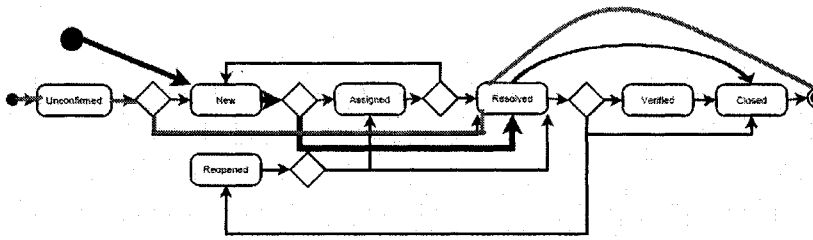

Figure 2. The most common defect life cycle in the Apache (Black bold line) and in the Mozilla (Gray bold line)

According to the life cycle model it means that those defects did not lead to changes in the source code. The resolutions of the defects in the Apache project are presented in Table 7. It shows that there were also defects that led to change or modification.

Table 7. Resolution of defects with most common life cycle

Project	Duplicate	Fixed	Invalid	Later	Won't fix	Works for me
Apache	65	142	236	2	47	0
Mozilla	5409	246	2443	0	326	2709

As we see in Table 4, only about 290 defects, which is about 30 per cent, lead to fixes. However, these defects should have followed the life cycle model. Surprisingly, 142 of 288 defects that led to source code changes were changes directly from the state *new* to *resolved*. Furthermore, there were other almost direct jumps to the state *resolved*, which together covered 237 of 288 fixed defects (82 per cent). Thus, only 51 of 288 defects (18 per cent) that were fixed followed the life cycle model and went through the states *new*, *unconfirmed*, *assigned* and *resolved*. If we then look at Mozilla, we can see in Table 6 that over 16 000 of 20 300 defects jumped directly

from the state *unconfirmed* or *new* to *resolved.* The most common life cycle in the Mozilla project is presented with bold gray line in Figure 2.

According to the life cycle model, those defects did not cause changes in the source code. The resolutions of the analyzed defects from the Mozilla project are presented in Table 4: only about 2400 defects were fixed, which is less than 12 per cent. However, these defects should have followed the expected life cycle and others should have jumped more or less directly to *resolved.* Surprisingly only 246 of 11 133 defects that jumped directly from *unconfirmed* to *resolved* were fixed. However, there were other almost direct jumps to *resolved*, such as from *unconfirmed* to *resolved* via *new*, which together covered 1652 of 2414 fixed defects (68 per cent). Thus, 748 of 2414 defects (31 per cent) that led to fixing followed the expected life cycle or went through at least the states *new* or *unconfirmed, assigned* and *resolved.* There were also 14 defects that could not be classified into either group because they had so many state changes.

Despite the number of defects, both cases have similar characteristics. The proportion of defects that led to changes (fix) was relatively small. Most of the defects transited directly to the state *resolved* and it was possible to have a *resolution.* It seems that developers just pick up a defect and resolve it without assigning, and they did not update the state of the defect before it was resolved. However, the state of the defect does not tell the whole truth since defect management systems allow users to leave comments without changing the status of the defect. There were also discussions in the mailing lists, which we did not analyze in this study.

5 Related Work

To our knowledge, this is the first work that studies a defect life cycle in Open Source projects. However, the defects and defect management systems have been previously studied from other viewpoints. Mockus et al. [6] has studied defects and changes of the source code in the Apache and Mozilla projects. They compare the numbers of changes and defects per developer in commercial projects. Furthermore, they measure defect density in the projects and compare it with the size of the source code. Huntley [7] has studied the defects of the Apache and Mozilla projects from the viewpoint of Organizational learning. Fisher et al. have combined version control and defect information in their studies [8] creating a release history from the version control system and bug tracking system. They describe the changes of the source code and defects from the release history viewpoint.

5 Conclusion

We studied maintenance process and expected that the life cycle of defects would be similar to the maintenance process, with several states during the process. However, the study shows that the defect life cycle in two well-known Open Source Software

projects was much more straightforward. The state of the defect was set to *resolved* directly after the initial state. More surprisingly, the outcome of the defect did not seem to have any relation with its life cycle: even the defects led to changes they were not assigned. The states of the defect could be simplified to *open* and *closed.* These two states are enough to cover 84 per cent of all defects in the Apache project and 79 per cent in the Mozilla project. So, the usage of a defect management system does not seem to be efficient in the Open Source projects studied. It is generally claimed that defect management is a crucial part of maintenance, leading to the assumption that users cannot rely on the maintenance of Open Source Software.

To improve maintenance reliability from the user's viewpoint, these software projects should use defect management more intensively or publish a document explaining the procedures in use. At least, developers should assign a defect when they start working with it so that users and other developers could see that the defect is being dealt with. Unfortunately, similar data have not been published concerning a proprietary project. In our experience, proprietary projects have a similar type of defect life cycle.

7 References

1. Bugzilla.org, 2005. http://www.bugzilla.org
2. ISO/IEC. ISO/IEC 12207:1995/Amd 2002: Software Engineering: Software life cycle processes. ISO/IEC, 2002.
3. T. Koponen and V. Hotti. Defects in open source software maintenance - two case studies - apache and mozilla. In *Proceedings of The 2005 International MultiConference in Computer Science and Computer Engineering*, Las Vegas, NV, USA, 2005. CSREA Press.
4. T. Koponen and V. Hotti. Open source software maintenance process framework. In *5-WOSSE: Proceedings of the fifth workshop on Open source software engineering*, pp. 1-5, New York, NY, USA, 2005. ACM Press.
5. IEEE Computer society. *Guide to the Software Engineering Body of Knowledge (SWEBOK).* IEEE Computer society, Los Alamitos, California, USA, 2001.
6. A. Mockus, R. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309-346, 2002
7. C. Huntley. Organizational learning in open source software projects: An analysis of debugging data. *IEEE Transactions on Engineering Management*, 50(4), 2004.
8. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the Inter- national Conference on Software Maintenance*, pp. 23, Washington, DC, USA, 2003. IEEE Computer Society.