Open Source Software Development Process: Debugging and Maintaining

Yongsen MA
Shanghai Jiao Tong University
E-mail: mayongsen@gmail.com

I. Introduction

Open source software development projects are generally Internet-based networks or communities of software developers. The software and codes are made freely available to all that adhere to the licensing terms. Open source software projects and development processes have spread rapidly and widely. The number of developers participating in each project ranges from a few to many thousands, and so does the number of users [6].

In sharp contrast with the traditional software development, all the developers are offered free access to the source code of open source software. This means that anyone with the proper programming knowledge and motivations can use, study, and modify any open source software. Currently, the rapid technological advances in computer hardware, software and networking have made it much easier to create and sustain a communal development style at ever-larger scales.

Through the experience of participating, programming and contributing, developers can find the opportunity to learn, practice and share. For every individuals, project groups, and organizations that are associated with the open source software, they can improve or adapt their processes and practices more effectively. Also, the developers appear to really enjoy their work, and to be recognized as trustworthy and reputable contributors. They can also self-select the technical roles they will take on as part of their participation in a project, rather than be assigned to a role in a traditionally managed software engineering project. Moreover, many developers participate in and contribute to multiple projects, and most participants typically contribute to just a single

module. Though a small minority of modules may include patches or modifications contributed by hundreds of contributors both individually and collaboratively.

An open source software development project is typically initiated by an individual or a small group with an idea for personal or business reason. The project initiators also generally become the project owners or maintainers who take on responsibility for project management. The individual or group may develop an initial, rough version of the code that outlines the functionality envisioned. The source code for this first version is then made freely available to all via downloading from the Internet or Ftp server established by the project. The project founders also set up mailing lists for the project. Those interested in using or further developing the code can consult this list in order to get help running the software or writing code. The list can also be used to provide information or distribute new open source code for others to discuss and test further.

A complete open source software system is typically composed of the following components:

1) **Home Page:** the gateway to the open source project. It is mainly composed of the project documents, available versions, and links to related resources. It plays a guide role, through which both developers and users can quickly find the information they need.
2) **Code Repository:** the core of an open source project. All the development is carried out around the code repository. The code repository is governed by a community consisting of developers who can commit code to the authorized version of the software.
3) **Bug Tracking System:** the system for tracking software bugs. It helps developers to manage software defects and improve the quality of software. It also provides a simple way of collecting feedback information from the users.
4) **Mailing List:** the platform for problem discussion and information sharing. It is widely used as the official communication method in open source projects, or even utilized as bug tracking for the open source projects that have no bug tracking system.
5) **Wiki:** the pages written in simple markup language. Because of its open and collaborative characteristics of document writing mode, it is widely used in open source project as a primary method of document preparation.

For the newcomers who are interested in open source software, they can participate in the development progress and gradually contribute to more complex and technically difficult work both individually and collaboratively. In the case that some projects are successful in attracting

people to participate in the process, some of the developers do create new and modified code based on their own interests. New and modified code that is deemed to be of sufficient quality and of general appeal by the project maintainers is then added to the authorized version of the software. In many projects the privilege of adding to the authorized code is restricted to only a few trusted developers who then become part of a community. Most contributors are experienced, professional programmers. Some act as independent individuals volunteering to develop code, others are employees of organizations that support their participation.

In summary, the open source software development process is mainly composed of the **individual participating and developing** and the **collaborative contributing and maintaining**.

A. Participating and Developing

1) Starting: know enough about the open source project, familiar with the functions and features of the project, find the part that you are interested in.
2) Discussion: the most important decision are usually made by technical discussions. To contribute to community work, you should actively participate in the following discussions:
   - Subscribe Mailing List
   - Take Part in News Group
   - Participate in Conference
3) **Programming and Debugging:** based on the understanding of the open source project and the discussion on specific topic, then be concentrate on the source code and other related issues:
   - Maintain Website
   - Consummate Documents
   - Run Test Codes
   - Report Bugs
   - Submit Patches
4) Improving: gain the experience and programming ability by participating and developing.

B. Contributing and Maintaining

1) Creating a Repository

2) Making Changes
   - Add Files
   - Commit Changes
   - Maintain Files Status and Differences
   - Manage Files
3) **Maintaining and Branching**
   - Create Branches
   - Merge Branches
   - Handle Conflicts
   - Delete and Renaming Branches
4) Handle Releases

The core of the open source project is the **development** and **maintenance** of the source code. In the following, this report will give a detailed investigation on the debugging and maintaining issues in open source software development process.

II. Debugging

It may spend as much time debugging as coding for programmers, so they need to learn from their mistakes. Every bug can teach developers how to prevent a similar bug from happening again or to recognize it if it does. Generally, the debugs can be classified into basic bugs and functional bugs. The basic bugs is closely related to the syntax and grammar rules of the program languages. In some other cases, the functional bugs will occur when there are conflicts related to hardware or software.

A. Basic Debugging

As is presented in [4], the bugs can be divided into three kinds, i.e., easy bugs in good clues, hard bugs in bad clues and non-reproducible bugs. Developers can adopt different methods to debug it based on the bugs types.

Once a bug has been seen, the first thing to do is to think about the clues it presents. Check about if it is something familiar or just because something is just changed in the program. A few test lines and a few display statements in the code may be enough to resolve it. If there are

no good clues, try to cut down the input data to make a small input that fails. Another method is cutting out code to eliminate regions that can not be related. It is also possible to insert checking code that gets turned on only after the program has executed some number of steps, again to try to localize the problem. Furthermore, breakpoints and stepping make it possible to rerun a failing program one step at a time to find the first place where something goes wrong. Try to use debugger tools to check for memory leaks, array bounds violations, suspect code.

1) **Good Clues, Easy Bugs:** When some bugs occur, first think backwards to discover the reasons. Once we have a full explanation, we'll know what to fix and discover other things.
   - *Look for familiar patterns.* If this is a familiar pattern, the bugs will be better understood and the answer can be easily found.
   - *Examine the most recent change.* If only one thing is changed at a time as a program evolves, the bug most likely is in the new code.
   - *Don't make the same mistake.* After a bug is fixed, remember not to make the same mistake somewhere else.
   - *Debug it timely.* Don not ignore a crash when it happens. Try to track it down right away, since it may not happen again until it's too late.
   - *Get a stack trace.* Although debuggers can probe running programs, one of the most common uses is to examine the state of a program after death.
   - *Read before typing.* One effective but under-appreciated debugging technique is to read the code very carefully and think about it for a while without making changes.
2) **No Clues, Hard Bugs**
   - *Make the bug reproducible.* The first step is to make sure the bug can appear on demand. It is difficult to chase down a bug that does not happen every time.
   - *Divide and conquer.* Each test case should aim at a definitive outcome that confirms or denies a specific hypothesis about what is wrong.
   - *Study the numerology of failures.* Try to study the patterns of numbers related to the failure pointed us right at the bug.
   - *Display output.* Add some useful statements to display more information. It can be the easiest and most effective way to find out the problem.
   - *Write self-checking code.* If more information is needed, write check function to test

a condition, dump relevant variables.

- *Write a log file.* Write a log file containing a fixed-format stream of debugging output. When a crash occurs, the log records what happened just before the crash.
- *Use debug tools.* Make good use of the facilities of the environment as debugging. Use shell scripts and other tools to automate the processing of the output from debugging.
- *Keep records.* Keep the record of tests and results, it is less likely to overlook something or check some possibility unnecessarily.

3) **Non-reproducible Bugs:** Bugs that won't stand still are the most difficult to deal with. It means that the error is not likely to be a flaw in the algorithm but that in some way the code is using information that changes each time the program runs.
   - Check whether all variables have been initialized. Local variables of functions and memory obtained from allocators are the most likely culprits in C and C++.
   - If the bug changes or even disappears when debugging code is added, it may be a memory allocation error. The bug is outside of allocated memory, and the addition of debugging code changes the layout of storage enough to change the effect of the bug.
   - If the crash seems far away from anything that could be wrong, the most likely problem is overwriting memory by storing into a memory location that is not used later.

In summary, debugging is hard and can take long and unpredictable amounts of time, so the goal is to avoid having to do much of it. Techniques that help reduce debugging time include good design, good style, boundary condition tests, assertions and sanity checks in the code, defensive programming, well designed interfaces, limited global data, and checking tools. On the other hand, some features are prone to error under certain cases, like *goto* statements, global variables, unrestricted pointers, and automatic type conversions. Programmers should know the potentially risky bits of their languages and take extra care when using them. They should also enable all compiler checks and heed the warnings.

B. Functional Debugging

For a realistic system, engineers first construct a functional model that describes the system, then code is generated from this model. A functional debugger is a tool for locating errors in functional models. Such a tool should help the engineer understand erroneous behavior in terms of the functional model.

1) Try to use other stable software to make sure the hardware, devices, operating system and software are in good condition. Check if the software can work normally on other systems.
2) Examine carefully about which part is responsible for the bug, especially the code that changed most recently, to determine whether the bug is caused by the configurations of hardware, device, operating system.
3) When a program works for one person but fails for another, something must depend on the external environment. This might include files read by the program, file permissions, environment variables, search path for commands, defaults, or startup files.
4) Show the intermediate variables if necessary. Make sure the variables is in consistent with the theory, algorithm and protocol.
5) Send bug reports and submissions to the mailing list of the open source project.

III. MAINTAINING

There are two important factors for a successful open source project: a widely accepted leadership setting the project guidelines and driving the decision process, and an effective coordination mechanism among the developers based on shared communication protocols [2]. The leadership deeply influences the evolution of the project by selecting the best fitting solution among the ones that different contributors propose for the same problem. The coordination mechanism shared by developers is in most cases able to produce a well structured flow of contributions. In Git, the contributing and maintaining are carried out by the concept of branching. Some people refer to the branching model in Git as its killer feature, and it certainly sets Git apart in the other vision control system community [3].

A. Branching

Branching means developers can diverge from the main line of development and continue to do work without messing with that main line. The way Git branches is incredibly lightweight, making branching operations nearly instantaneous and switching back and forth between branches generally just as fast. A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. When the commits are initially made, a master branch will point to the last commit. If new commit is add, the pointer will move forward automatically. The branching is generally composed of the following procedures:

1) **Creating:** To create a branch and switch to it at the same time:

   ```
   git checkout -b example
   ```

   Then do the following work in the branch:

   - Test changing
   - Add new functionality
   - Fix bugs

2) **Patching:** If another issue is critical at this stage, then do the following to add a hotfix:

   a) Revert back to your production branch.
   b) Create a branch to add the hotfix.
   c) After its tested, merge the hotfix branch, and push to production.
   d) Switch back to your original story and continue working.

3) **Merging:** Combine directory and file contents from separate sources to yield one combined result. If the issue is complete and ready to be merged into the master branch, check out the branch to be merged and then run the git merge command:

   ```
   git checkout master
   git merge example
   ```

4) **Handling Conflicts:** If the same part of the same file differently in the two branches that will be merged together is changed, Git will not be able to merge them cleanly. Then there will be a merge conflict information. The conflict can be handled by manual or other tools.

   a) Manual: Git adds standard conflict-resolution markers to the files that have conflicts, so open them manually and resolve those conflicts.
   b) Tools: Git also provides graphical tools to resolve these issues, which fires up an appropriate visual merge tool handle the conflicts:

      ```
      git mergetool
      ```

   It has paused the process while the conflict is under resolving. To see which files are unmerged at any point after a merge conflict:

   ```
   git status
   ```

5) **Deleting and Renaming:** When the work is merged in, there is no further need for the example branch. Delete it and then manually close the ticket in the tracking system:

   ```
   git branch -d example
   ```

B. Contributing

When developers want to share a branch with the world, they need to push it up to a remote one that have write access to. The local branches are not automatically synchronized to the remote servers. Also, developers can use private branches for work if they do not want to share, and push up only the topic branches that they want to collaborate on. In general, this can be described as follows:

1) Developers generate and send patches via E-mail
    - Most developers send patches to a maintainer or mailing list
    - Highly visible public review of patches on mailing list
2) Developers push updates to an upstream maintainer
    - Some developers have write permissions on an upstream repository
    - Initiated by downstream developers
3) Maintainers pull updates from a downstream developer
    - Maintainers can directly pull from the published repository
    - Initiated by upstream maintainers

Because Git is very flexible, people can and do work together many ways. It is problematic to describe how to contribute to a project, since every project is a bit different. Some of the variables involved are active contributor size, chosen workflow, commit access, and possibly the external contribution method.

- **Active contributor size:** This is important because with more developers, the project will run into more issues. The submitted changes may be rendered obsolete or severely broken by work that is merged while the changes were waiting to be approved or applied.
- **Workflow in use:** If it is centralized that each developer having equal write access to the main code line, or the project have a maintainer or integration manager who checks all the patches, the maintenance of workflow is more easy.
- **Commit access:** The workflow required in order to contribute to a project is much different if developers have write access to the project than if they do not.

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to developers. There are several common patterns for contributing to a project, and there are a huge number of variations on how its done.

C. Maintaining

The open source project maintaining can be consist of accepting and applying patches generated via patch and E-mail, or integrating changes in remote branches for repositories that have been added as remotes to the project. Whether developers maintain a canonical repository or want to help by verifying or approving patches, they need to know how to accept work in a way that is clearest for other contributors and sustainable over the long run.

1) *Working in Topic Branches:* When developers are thinking of integrating new work, its generally a good idea to try it out in a topic branch, i.e., a temporary branch specifically made to try out that new work. It is easy to tweak a patch individually and leave it if it is not working until developers have time to come back to it.
2) *Applying Patches from E-mail:* If a patch is received over e-mail that need to be integrated into the project, maintainers need to apply the patch in the topic branch to evaluate it. There are two ways to apply an e-mailed patch: with `git apply` or `git am`.
3) *Checking Out Remote Branches:* If the contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent the URL to the repository, maintainers can add them as a remote and do merges locally.
4) *Determining What Is Introduced:* Now there is a topic branch that contains contributed work, maintainers should use the right commands to review exactly what should be introduced if the patch is merged into the main branch.
5) *Integrating Contributed Work:* If all the work in the topic branch is ready to be integrated into a more mainline one, determine the workflow that can be used to maintain the project.
6) *Tagging the Releases:* When developers have decided to cut a release, they will probably want to drop a tag so that the release can be recreated at any point going forward.
7) *Generating a Build Number:* If developers want to have a human-readable name to go with a commit, they can run `git describe` on that commit.
8) *Preparing a Release:* When developers want to release a build, one thing is to create an archive of the latest snapshot of the code for those poor souls who do not use Git.
9) *The Shortlog:* It is time to e-mail the mailing list of people who want to know what is happening in the project. A nice way of quickly getting a sort of changelog of what has been added since the last release or e-mail is to use the `git shortlog` command.

IV. Cloud Programming

The cloud interfaces are convenient for launching multiple independent instances of traditional single-node services, but writing truly distributed software remains a significant challenge. Distributed applications still require a developer to orchestrate concurrent computation and communication across machines, in a manner that is robust to delays and failures. Writing and debugging such code is difficult even for experienced infrastructure programmers, and drives away many creative software designers who might otherwise have innovative uses for cloud computing platforms [1].

A cloud programming system should include the editing, debugging, testing, running procedures. In the lower layer, it can record the state of debugging, which means the editing (say vi, Emacs) and debugging (say GDB, Code::Blocks) tools should be connected to the vision control system (say Git). In the upper layer, it should provide feedback information, which indicates it can make adaption according to the state of developers (say commits, code) and end users' requirements (say QoS, traffic loads).

A. Cloud Debugging

Git is more likely a vision control system, which is not connected to the programming and debugging process directly. The vision control function actually provides backup for the code editing: developers write code lines by vi or Emacs, then update the overall code by Git, in which the changes can be recorded for software evaluation and reversion. In the same way, it can also track the debugging process: the system records the differences after the debugging, especially when the bugs first appear and when they are solved. It means that the debugging is set *in the cloud but not just locally*. This can be helpful for both the debugging ability of every individual developers and the maintaining and bug tracking of the overall open source project:

1) Remind developers about the bugs they had made, which are usually issued by memory. It can also collect the bugs statistics which can provide advice about the debugging issues.
2) Maintain the bug differences between contributors, which can improve the performance of bug tracking system of the open source project.
3) The bug patterns can be classified based on the collected statistics, which can help to find potential problems and promote the software evaluation.

B. Process Modeling

Git is a vision control system, which is designed for general open source software development. However, some functions should be tailored for some specific applications. The most popular repositories on Github are web applications(html5-boilerplate, rails, bootstrap), or written in high-level languages(JavaScript, Ruby). On the other hand, other projects mainly focus on operating system or debugging environment(Linux kernel, Eclipse), or written in low-level languages(C, C++, Java). On the other hand, the open source platform is self-organized, and the participating and contributing are usually motivated by the developers' own interest. Some are interested in the software engineering, while others may focus on wireless networking. So many problems are solved from certain perspectives but not the same case from another aspect [5]. It means that the programming languages, cooperation mechanisms and target end users of different projects are *highly heterogeneous*. So it is necessary to make *multi-stage modeling* to maintain it [7]. If the communication patterns, contribution levels, evaluation speed and user needs are classified and modeled, it can improve the efficiency according to the specify requirements and characters.

For example, a teaching system for programming classes should be centralized so that assignments can be made. The commits and bugging trace of every students can be issued effectively. Furthermore, some work should be done by group that certain individuals should be connected with each other. So it is something like a mesh topology in wireless network. Another instance is that the weighted sequence $\{1/2,1/4,1/8,1/16\}$ is more effective than $\{1/3,1/6,1/9,1/12\}$ in exponentially weighted moving average from the programming perspective. A cloud programming system should provide a platform for the communication and cooperation of programming, computing and networking in system level.

REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. EuroSys '10, pages 223–236, New York, NY, USA, 2010. ACM.

[2] A. Bonaccorsi and C. Rossi. Why open source software can succeed. *Research Policy*, 32(7):1243 – 1258, 2003.

[3] S. Chacon, J. Hamano, and S. Pearce. *Pro Git*, volume 288. Apress, 2009.

[4] B. Kernighan and R. Pike. *The practice of programming*. Addison-Wesley Professional, 1999.

[5] A. Koblin. Artfully visualizing our humanity. http://www.ted.com/talks/aaron_koblin.html.

[6] G. Krogh and E. Hippel. Special issue on open source software development. *Research Policy*, 32(7):1149 – 1157, 2003.

[7] W. Scacchi, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani. Understanding free/open source software development processes. *Software Process: Improvement and Practice*, 11(2):95–105, 2006.