Open Source Software Development Process

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.

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.

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) **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.
4) **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.
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.

A. Participating and Developing

1) Starting
2) Discussion
   - Subscribe Mailing List
   - Take part in News Group
   - Participate in Conference
3) **Programming and Debugging**
   - Consummate documents
   - Running test codes
   - Report Bugs
   - Submit patch
4) Improving

B. Contributing and Maintaining

1) Creating a Repository

2) Making Changes
    - Adding Files
    - Committing Changes
    - Files Status and Differences
    - Managing Files
3) **Maintaining and Branching**
    - Creating Branches
    - Merging Branches
    - Handling Conflicts
    - Deleting and renaming branches
4) Handling Releases

II. Programming and Debugging

For example, how are crash reports handled? How are bug reports handled? How are bugs classified and confirmed?

Good programmers know that they spend as much time debugging as writing so they try to learn from their mistakes. Every bug you find can teach you how to prevent a similar bug from happening again or to recognize it if it does.

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.

A major force in the evolution of programming languages has been the attempt to prevent bugs through language features. Some features make classes of errors less likely: range checking on subscripts. restricted pointers or no pointers at all, garbage collection, string data types and strong type-checking. On the opposite side of the coin, some features are prone to error, 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.

A. Debuggers

A debugger can be invoked directly when a problem is known to exist. Some debuggers take over automatically when something unexpectedly goes wrong during program execution. It's usually easy to find out where the program was executing when it died, examine the sequence of functions that were active (the stack trace), and display the values of local and global variables. That much information may be sufficient to identify a bug. If not, breakpoints and stepping make it possible to re-run a failing program one step at a time to find the first place where something goes wrong.[4]

Once a bug has been seen, the first thing to do is to think hard about the clues it presents. How could it have come about? Is it something familiar? Was something just changed in the program? Is there something special about the input data that provoked it? A few well-chosen test cases and a few print statements in the code may be enough.

If there aren't good clues, hard thinking is still the best first step, to be followed by systematic attempts to narrow down the location of the problem. One step is cutting down the input data to make a small input that fails; another is cutting out code to eliminate regions that can't be related. It's 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.

Use other aids as well. Explaining your code to someone else (even a teddy bear) is wonderfully effective. Use a debugger to get a stack trace. Use some of the commercial tools that check for memory leaks, array bounds violations, suspect code, and the like. Step through your program when it has become clear that you have the wrong mental picture of how the code works.

Know yourself, and the kinds of errors you make. Once you have found and fixed a bug, make sure that you eliminate other bugs that might be similar. Think about what happened so you can avoid making that kind of mistake again.

B. Basic Debugging

1) **Good Clues, Easy Bugs:** Something impossible occurred, and the only solid information is that it really did occur. So we must think backwards from the result to discover the reasons. Once we have a full explanation, we'll know what to fix and, along the way, likely discover a few other things we hadn't expected.

- *Look for familiar patterns.* Ask yourself whether this is a familiar pattern. ”I’ve seen that before” is often the beginning of understanding, or even the whole answer.
- *Examine the most recent change.* If you’re changing only one thing at a time as a program evolves, the bug most likely is either in the new code or has been exposed by it.
- *Don’t make the same mistake twice.* After you fix a bug, ask whether you might have made the same mistake somewhere else.
- *Debug it now, not later.* Being in too much of a hurry can hurt in other situations as well. Don’t ignore a crash when it happens; 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 their 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.
- *Explain your code to someone else.* Another effective technique is to explain your code to somcone else. This will often cause you to explain the bug to yourself.

2) **No Clues, Hard Bugs**
   - *Make the bug reproducible.* The first step is to make sure you can make the bug appear on demand. It’s frustrating to chase down a bug that doesn’t 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.* Studying the patterns of numbers related to the failure pointed us right at the bug.
   - *Display output to localize your search.* If you don’t understand what the program is doing, adding statements to display more information can be the easiest, most effective way to find out.
   - *Write self-checking code.* If more information is needed, you can write your own check function to test a condition, dump relevant variables.
   - *Write a log file.* Another tactic is to 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.

- *Draw a picture.* Sometimes pictures are more effective than text for testing and debugging.
- *Use tools.* Make good use of the facilities of the environment where you are debugging. Use shell scripts and other tools to automate the processing of the output from debugging runs.
- *Keep records.* If you record your tests and results, you are less likely to overlook something or to think hat you have checked some possibility when you haven't.

3) **Non-reproducible Bugs:** Bugs that won't stand still are the most difficult to deal with, and usually the problem isn't as obvious as failing hardware. The very fact that the behavior is nondeterministic is itself information, however; it means that the error is not likely to be a flaw in your algorithm but that in some way your 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 behavior or even disappears when debugging code is added. it may be a memory allocation error-somewhere you have written 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 site seems far away from anything that could be wrong, the most likely problem is overwriting memory by storing into a memory location that isn't used until much later.
   - When a program works for one person but fails for another, something must depend on the external environment of the program. This might include files read by the program, file permissions, environment variables, search path for commands, defaults, or startup files.

C. Functional Debugging

III. Maintaining and Branching

How are the assignments to individual developers made? How to merge code changes in Git? How are code inconsistency handled? In each step of the process, have you identified any software engineering issues which have rooms for improvements?

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

1) active contributor size: How many users are actively contributing code to this project, and how often? In many instances, youll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects. For really large companies or projects, the number of developers could be in the thousands, with dozens or even hundreds of patches coming in each day. This is important because with more and more developers, you run into more issues with making sure your code applies cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by work that is merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your patches valid?
2) workflow in use: Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?
3) commit access: The workflow required in order to contribute to a project is much different if you have write access to the project than if you dont. If you dont have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to you. Ill cover aspects of each of these in a series of use cases, moving from simple to more complex; you should be able to construct the specific workflows you need in practice from these examples.

A. Maintaining

In addition to knowing how to effectively contribute to a project, youll likely need to know how to maintain one. This can consist of accepting and applying patches generated via format-patch and e-mailed to you, or integrating changes in remote branches for repositories youve added as remotes to your project. Whether you maintain a canonical repository or want to help by verifying or approving patches, you need to know how to accept work in a way that is clearest for other contributors and sustainable by you over the long run.

1) Working in Topic Branches: When youre thinking of integrating new work, its generally a good idea to try it out in a topic branch a temporary branch specifically made to try out that new work. This way, its easy to tweak a patch individually and leave it if its not working until you have time to come back to it.
2) Applying Patches from E-mail: If you receive a patch over e-mail that you need to integrate into your project, you need to apply the patch in your topic branch to evaluate it. There are two ways to apply an e-mailed patch: with *git apply* or with *git am*.
3) Checking Out Remote Branches: If your contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, you can add them as a remote and do merges locally.
4) Determining What Is Introduced: Now you have a topic branch that contains contributed work. At this point, you can determine what youd like to do with it. This section revisits a couple of commands so you can see how you can use them to review exactly what youll be introducing if you merge this into your main branch.
5) Integrating Contributed Work: When all the work in your topic branch is ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall workflow do you want to use to maintain your project? You have a number of choices, so Ill cover a few of them.
6) Tagging Your Releases: When youve decided to cut a release, youll probably want to drop a tag so you can re-create that release at any point going forward.
7) Generating a Build Number: Because Git doesnt have monotonically increasing numbers like v123 or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run git describe on that commit.
8) Preparing a Release: Now you want to release a build. One of the things youll want to do is create an archive of the latest snapshot of your code for those poor souls who dont use

Git.

9) The Shortlog: Its time to e-mail your mailing list of people who want to know whats happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or e-mail is to use the git shortlog command.

B. Branching

Branching means you 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. As you initially make commits, youre given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.

1) **Creating Branching:** To create a branch and switch to it at the same time, you can run:

   ```
   git checkout -b example
   ```

   Then you can do the following work in the branch:

   - Test Changing
   - Add new functionality
   - Fix bugs

   At this stage, youll receive a call that another issue is critical and you need a hotfix. Youll do the following:

   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.

2) **Merging Branching:** Combine directory and file contents from separate sources to yield one combined result. Suppose youve decided that your issue is complete and ready to be merged into your master branch. All you have to do is check out the branch you wish to merge into and then run the git merge command:

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

- Sources for merges are local branches
- Merges always occur in the current, checked-out branch
- A complete merge ends with a new commit

Merge heuristics:

- Several merge strategies: resolve, recursive, octopus, ours
- Techniques: fastforward, threeway

Merge types

- straight merge
- squashed commits
- cherry picking

3) **Handling Conflicts:** Occasionally, this process doesnt go smoothly. If you changed the same part of the same file differently in the two branches youre merging together, Git wont be able to merge them cleanly. Youll get a merge conflict information. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

```
git status
```

You can handle this conflict by manual or other tools.

a) Manual: Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts.
b) Tools: If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
git mergetool
```

4) **Deleting and renaming branches:** Now that your work is merged in, you have no further need for the example branch. You can delete it and then manually close the ticket in your tracking system:

```
git branch -d example
```

C. *Other Issues*

1) **Pushing:** When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches arent automatically synchronized to the remotes you write to you have to explicitly push the branches you want to share. That way, you can use private branches for work you dont want to share, and push up only the topic branches you want to collaborate on.
2) **Tracking:** Checking out a local branch from a remote branch automatically creates what is called a tracking branch. Tracking branches are local branches that have a direct relationship to a remote branch. If youre on a tracking branch and type git push, Git automatically knows which server and branch to push to. Also, running git pull while on one of these branches fetches all the remote references and then automatically merges in the corresponding remote branch.
3) **Rebasing:** In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.

D. *Sending Changes Upstream*

- Generate and send patches via email
  - Most developers send patches to a maintainer or list
  - Highly visible public review of patches on mail list
- Maintainer pulls updates from a downstream developer
  - Maintainer can directly pull from your published repository
  - Initiated by upstream maintainer
- Developer pushes updates to an upstream maintainer
  - Some developers have write permissions on an upstream repository
  - Initiated by downstream developer

IV. Issues on Cloud Programming

Aaron Koblin: Artfully visualizing our humanity

Why free software has poor usability, and how to improve it

A. *modeling, examination, investigation*

1) individuals
2) groups
3) organizations

1) operate systems
2) web
3) application
4) network

1) contribute:
2) process: stable, scalable
3) acquire: software, individuals, groups

References

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

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

[3] G. Hertel, S. Niedner, and S. Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research Policy*, 32(7):1159 – 1177, 2003.

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

[5] B. Kogut and A. Metiu. Opensource software development and distributed innovation. *Oxford Review of Economic Policy*, 17(2):248–264, 2001.

[6] 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.

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

[8] C. Yilmaz, A. M. Memon, A. Porter, A. S. Krishna, D. C. Schmidt, and A. Gokhale. Techniques and processes for improving the quality and performance of open-source software. In *Software Process: Improvement and Practice*. John Wiley & Sons, 2006.