Introduction to Software Engineering Principles

In this chapter we will briefly cover a few topics we feel it is important for all software engineers to be familiar with. We strongly encourage you to do more research on these various topics on your own, however we hope that this chapter will provide a strong starting point for your own endeavors.

Source Control

By far one of the most important topics we will cover, and the one we will spend the most time on is source control. Professional programmers use source control, sometimes called version control, in nearly all aspects of a project. It is used to track changes made by team members, keep all files synced between each member, to provide a way to return to previous revisions of a project, and much more. Source control can even be used when working on a project by yourself to provide an easy way to access your work from any machine and sync between them, a way to restore to a previous version if something goes horribly wrong, and an easily viewable history of your work to this point. You can see a list of uses for source control below, after which we will look at a few more in-depth. Common source control systems are GIT, Mercurial, SVN, CVS, or Perforce. Nearly all types of source control have the same basic features however it is worth looking more in-depth at your different options to select the best one for your team and project. We will stay fairly agnostic of any specific version and focus on features included in nearly every type of source control.

Uses of Source Control

- Keep Track of Changes of multiple programmers.
- Makes merges of multiple programmers easier.
- You can go back in time.
- You can query who modified a file and when.
- You can learn what files need to be changed to implement a feature.
- You can find out what changes broke the daily build.
- You can evaluate at what level people contributed to a project.
- You can peer review changes
- You have a backup of your sources

The most commonly used feature of source control is the ability to keep all the files of a project synced between all team members or multiple machines easily. You can “checkout” or “clone” files from the repository to keep your files synced with everyone elses’ changes. The repository is all of the files in the project that are under source control. Checking out the files means essentially downloading the latest version of the files to your machine to review or work on. As you make changes the source control system will track your changes in the background so it will know what to change in the repository.

Once you have made multiple changes to the project it is time to “commit” your changes to the repository. You will likely need to “add” any new files you created and then tell the source control system to commit all the changes you have been making to the repository. Now anyone who

Draft

checks out the files next will see the changes you made reflected in their copy of the project. However, what if there is a “conflict” where the source control system doesn’t know which copy of the file to use after two people commit without checking out in between? In this case, most source control systems will allow you to view the files in conflict and determine the best course of action to resolve the issue, either by making a change to merge the files manually, or simply use one version over the other.

Another main feature of source control is the ability to “branch” from the main repository or trunk. When you branch from the main repository you get a copy of the project to work on that will not interfere or change the trunk that other people may still be working on. This allows you to make changes without impacting others’ work. Once you’ve made enough changes, you can “merge” your code back into the main repository. Since the source control tracks every change you made to the branch, it will do its best to combine all the changes to the files without causing any errors, but it may raise a conflict as noted above.

Various Source Control Programs

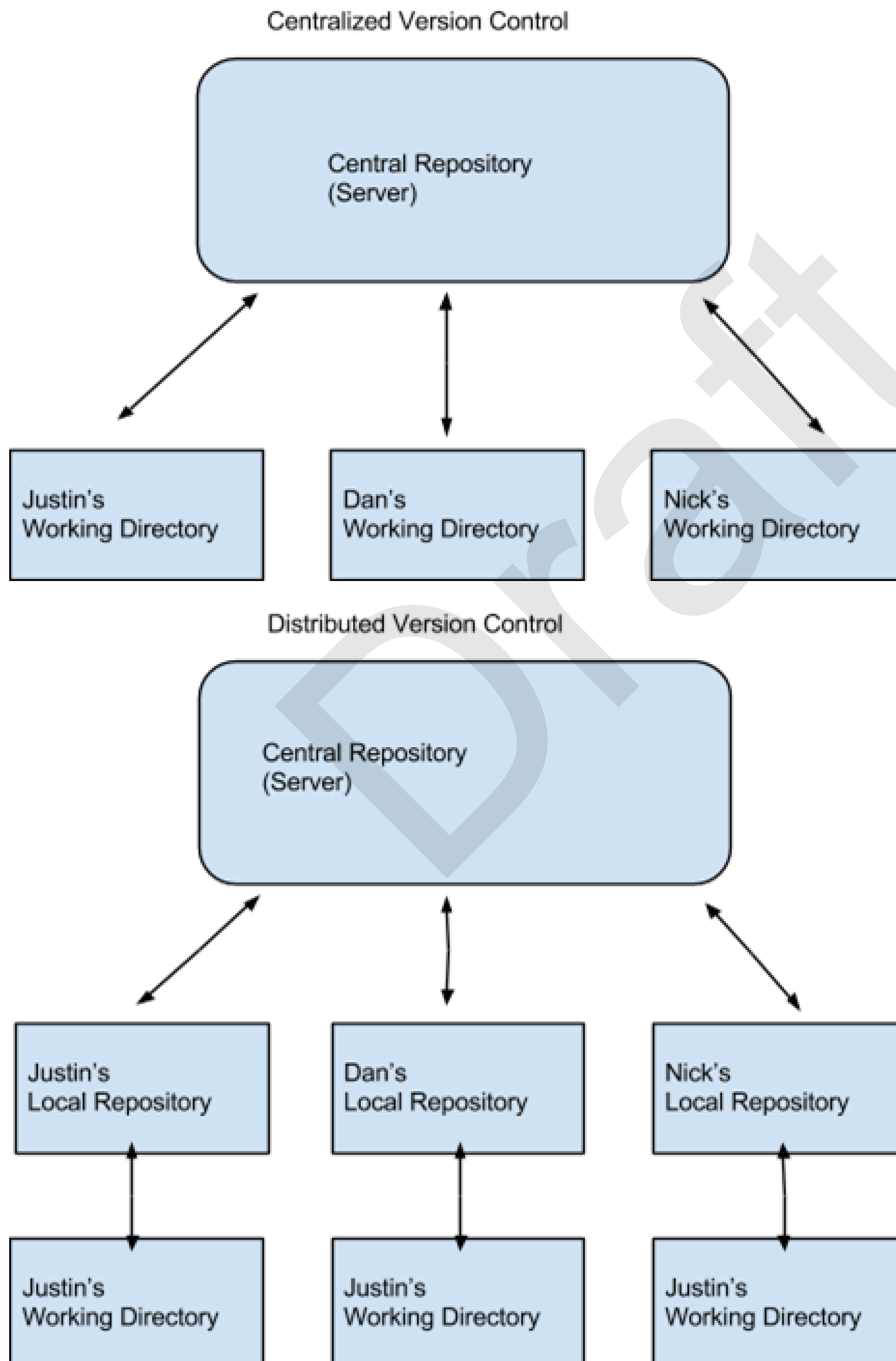
- CVS –
  - The oldest of all source controls.
- SVN –
  - Rewrite of CVS.
  - Centralized repository. Very popular.
- Perforce –
  - Also very popular
  - Not Free
- Mercurial –
  - Distributed Source Control.
  - No need for one centralized server.
  - Uses a local repository.
- GIT –
  - Also distributed source control
  - Fast
  - Written by Linus Torvalds for the Linux Kernel

Distributed vs Centralized Source Control

There are two main general types of source control: distributed and centralized. The most common centralized source control is SVN (see below figures). All files are accessed from a single central repository and works just like a client-server relationship. All of your commits and checkouts will be done from this one central repository. This type of source control strongest asset is that all the files are accessible from a centralized location and everyone always has exactly the same copies when they start working. However, this is also its biggest drawback as it requires maintenance of the server, and every member needs to have access to the server to make changes to the files. Examples of distributed source control systems are GIT and

Draft

Mercurial. In this case, every user has their own copy of the repository on their machine as well as the history of changes (see below figures). In many cases this allows for faster access to files, more in-depth tracking, and an easier ability to branch and merge. The main advantage is that you only need to connect to other team members when sharing changes rather than using a central repository. However this also creates the drawback that it is easier to make mistakes and makes conflicts more common.



© 2014 Gustavo Rodriguez-Rivera and Justin Ennen,Introduction to Systems Programming: *a Hands-on Approach*
(V2014-10-27) (systemsprogrammingbook.com)

Software Development Methods - Scrum and XP Programming

The first and arguably one of the most important topics in software engineering is the development method your team uses in projects. We will look at two popular programming methodologies, Scrum and XP Programming. They share some traits but also a few important differences that we will examine briefly.

Scrum is a programming methodology that uses agile programming techniques which focuses largely on project management when there is difficulty in planning ahead on the project. In a scrum team, the team is broken up into the scrum master, product owner, and team members. The scrum master is responsible for handling conflicts and barriers between the team members and product manager. The product manager is responsible for representing customer interests and prioritizing features or requirements. Scrum teams typically use what is called a “burndown chart” in order to show what is left for the team and the progress they have made.

XP Programming is a programming methodology that is typically used when there is some risk of completion for the project, such as the design changing continuously, or if the customer is unsure of what exactly they want. XP Programming uses a flat management structure and involves many checkpoints where the team and customer act upon customer requests or changes. Another trademark of XP Programming is that it often uses pair programming wherein you would have two programmers at a single computer working on the same module. The team typically constantly builds the project to ensure functionality at all times and leaves optimization until the final steps.

However there are ways in which XP and Scrum have some similarities. For example, they both function iteratively, that is they work on specific components for set amounts of time. These are often called “sprints”. In XP these typically last only 1-2 weeks, but for Scrum these sprints normally last two weeks to a month long. In addition, in Scrum changes made to the requirements and features must wait until the end of the sprint, but in XP as long as the team hasn’t started on the feature changes can be made. Additionally, in XP Programming, typically the customer decides on feature priority, whereas in Scrum choosing priorities becomes the responsibility of the product owner and the team then decides the specific order.

Internal Development Website

One tool you will find exceedingly valuable when working in a team on a large project, is an internal development website. This website would allow you to provide links to sources, either for your own code, or libraries used in the project. It should detail how to build the system and the results of previous builds. It will likely also need to feature a bug tracking system to allow easy discussion of them among team members. The website also provides a place to store design documents, making them easily accessible to all team members at any time, and finally it could contain a directory with contact listing for each team member to make it easy to find whoever you need to get in touch with.

Draft

Given all these uses for a website, it easy to see how helpful it can be to have one for your project, and indeed there are many services that provide such types of websites. The website could be as simple a private wiki that only your team members can access and edit the pages of. This type of website is very simple and fast to set up and can be used even on smaller projects. For larger projects, you may require a more robust website that supports an htdocs directory to easily add files to it. General rules of thumb for maintaining an internal development website is that you should not leave information in the heads of the programmers or designers, if it is relevant to the project it should be somewhere visible to team members for review. Anything important should be documented somewhere for the sake of current, and future development or maintenance.

Testing

Testing is an integral part of software engineering and development. When we say testing, we do not mean just providing a single correct input to our program and testing for seemingly correct output. When you test, you must test for edge cases, incorrect input, and you must always, always have specified exactly expected output from the program to test against for your input. The most practical way to improve the quality of an existing program is through testing. As well, the best way to test your program, is to write automated tests rather than manually test them, as you can run many more tests in a much shorter time when they are automated thus saving many human hours.

However, who is responsible for writing the tests? Typically the programmer is responsible for ensuring the software works correctly. The tests written by the programmer should be included in source control, as they are equally important as the rest of the code base. This type of coding could be considered white box testing as the development team of programmers has knowledge of the internals of the program. In some cases though, on larger projects, there may be an entire Quality Assurance department (QA) responsible for testing the software. This team is typically independent of the development team, and they perform what is called black-box testing as they have no internal knowledge of the program and are simply testing for correct input and output. It is important that a team independent of the development test the software, not only because it removes ego, but it also ensures that the program functions as expected to people outside the development team.

There are four main kinds of tests which are outlined below:

- Unit tests
  - A group of tests that test a specific class.
  - Every class should have unit tests.
  - Written by programmers
- System Tests
  - Test a specific subsystem of the product. Test multiple classes involved in a specific feature.
  - Written by QA and programmers
- Regression Tests
  - Test written to reproduce a bug and then it is used to verify that the bug is fixed.

Draft

        - If no regression test is written, there is a risk that the bug will be reintroduced.
        - Written by programmers and QA.
- Acceptance Tests
    - Evaluate the quality of the software before a release.
    - The tests tell if the product is ready for prime time or not.
    - Written by QA.

One last thing to consider is when you should test. The real answer is, all the time. You should test your code after writing it and before submitting to source control. Every day during the build automated tests should run on the entire code base and some systems even trigger tests whenever someone commits to source control automatically.

Bug Tracking

Once you have a system in place for testing, it is important to keep a database of the existing bugs. Sometimes you can even track your bugs in the same way you track features and even use the same database. You can use the bug/feature cycle in order to manage your found bugs and the process of fixing them. The bug/feature cycle is as follows:

1. Create a bug/feature report.
2. The Product Manager assigns a priority and severity.
    a. Priority – Importance for the organization.
        i. 1 –Finish it as soon as possible.
        ii. 2 – Make sure that it is in next release
        iii. 3 – Try to put it in next release.
    b. Severity – How it impacts the user.
        i. 1 – User absolutely cannot use the product without fixing this bug.
        ii. 2 – After a workaround the user can use product.
        iii. 3 – User can use product but bug makes use somehow difficult.
3. Assign the bug to a programmer.
4. The programmer analyzes the bug and puts an estimate of time.
5. The programmer fixes the bug and creates a regression test.
6. The programmer submits the fix to code review.
7. Once accepted, the fix is committed to source control and the bug report is passed to QA to verify that the bug has been fixed.
8. The bug report is closed. Closed bugs are reported in the release notes.

Following this process can help streamline how your team handles bugs, and prevents duplicating information about them.

Refactoring

Rarely will you ever work on a project completely from scratch. Often even your own code will be re-used long past your expected time table. For this reason it is important to return to your code after completing the functionality to improve its maintainability. This could include

Draft

making it more readable, or reducing the complexity of an algorithm. This is called refactoring, and is only a few examples of the way you can improve the maintainability of your code base. Below we will outline specific cases that can improve the quality of the code you write, so that the next guy has an easier time. Keep in mind that this is not an exhaustive list of ways to improve your code, but rather a starting point. As well, it is important to remember that you will also be the "next guy" once you start working on the next project with an existing code base. This makes the best rule of thumb to simply do what you would want someone to do for you.

- Comments
  - Should only be used to clarify "why" not "what".
  - Comments are important.
  - Use Javadocs or other ways to combine comments and documentation.
- Long Method
  - The longer the method the harder it is to see what it is doing.
- Long Parameter List
  - Don't pass in everything the method needs; pass in enough so that the method can get to everything it needs.
  - Use a parameter object that contains multiple parameters.
- Duplicated Code
  - Use functions to cut down on the amount of code that simply repeats the same process.
- Large Class
  - A class that is trying to do too much can usually be identified by looking at how many instance variables it has. When a class has too many instance variables, duplicated code cannot be far behind.
- Type Embedded in Name
  - Avoid redundancy in naming. Prefer schedule.add(course) to schedule.addCourse(course)
- Uncommunicative Name
  - Choose names that communicate intent (pick the best name for the time, change it later if necessary).
- Inconsistent Naming Schemes
- Dead Code
  - A variable, parameter, method, code fragment, class, etc is not used anywhere (perhaps other than in tests).
  - Delete the code.
- Speculative Generality
  - Don't over-generalize your code in an attempt to predict future needs.
  - If you have abstract classes that aren't doing much use Collapse Hierarchy
    Remove unnecessary delegation with Inline Class
    Methods with unused parameters - Remove Parameter
    Methods named with odd abstract names should be brought down to earth with by renaming them to something more constructive.

Draft

Chapter Summary Questions

1. Name a popular centralized source control system, and a popular distributed source control system. Describe the difference between centralized and distributed source control.
2. Briefly describe the similarities between scrum and XP programming and describe which one you personally would prefer and why.
3. List the types of tests and when to use each.
4. Pick a few of the refactoring guidelines and look back at an old project you have worked on and correct it to match the guidelines or other guidelines you come up with on your own.

Draft