ECSE 437 Study guide

Francis Piché

October 1, 2019

Contents

Part I
Preliminaries

1 License Information

These notes are curated from Professor Shane McIntosh's lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

Part II
Introduction

2 Release Engineering

2.1 Paper: Firefox Release Engineering

This paper outlined the project ot improve releases of the browser Firefox at Mozilla. The ultimate goal is to be able to "push a button and walk away" and this project is one step towards that.

- Project based on premise that Firefox would become increasingly larger target for security exploits, and therefore security patches must be able to be pushed very quickly in what is known as a "*chem-spill release*" This has 3 consequences:
    1. Post-mortem performed after every release to see if anything can be improved
    2. Chemspill releases are much less stressful.
    3. Go-to-build process for non-chemspill releases, with full-time person keeping track of where the build is at, in case of chemspill to speed up release safely.
- One person has responsibility of coordinating entire release.
    1. Attend triage meetings
    2. Understand background context
    3. Make backout decisions
    4. Centralized point of communications on release day

- A ”go to build” email is sent as the initiator for the release. Contains exact changeset to be used in the build, and the ”severity” or ”pace” of the release.
- Release coordinator decides and communicates the urgency of the release to keep cohesion amongst all groups.
- Many source control repositories are used in a single release, so keeping track of which version to use in each is essential. First step in release is tagging all the correct versions of code with the release tag.
- Tagging allows subsequent builds to reuse snapshots of code from previous releases if needed (useful for retrying)
- Desktop and mobile tagging done in parallel each taking 10-20 minutes

Once a US English desktop build is ready, a process unpacks it and replaces all US English strings with ones for each different locale then repackage. Done in parallel on 6 machines since there are a lot of locales to do this for.

Digital signatures are then added to ensure the customer receives an unmodified build. Humans are no longer required for this process. In the future they would like this to take place at the build/repack stage.

Updates are generated in a special file format. Client applications regularly check for new update files on a remote server. Major updates result in the user being prompted to update their client, and all files are re-downloaded and installed. Minor or partial updates result in only a few files to be updates via a diff file.

A community mirror network is the first place after QA that the new version is released to. This is then monitored to ensure it will be able to handle several million downloads over the following days. Once this check is complete the release is official, and the coordinator sends a ”go live” email.

2.1.1 Conclusion

The final points and ”lessons learned”:

- Stakeholders must understand the justification for spending on a release system
- Groups other than release engineering are also responsible for the release success
- Clear handoffs must be established
- Ensuring less employee turnover meant more knowledge stayed in the company
- Lots of small continuous improvements is easier to manage change than big complex releases.

3 Releases and Release Pipelines

In the past releases were very rare, large, expensive, intrusive and extremely risky.

Today web technologies have extremely automated auto-rollout releases and companies can ship their code multiple times per day resulting in many small changes continuously integrating into the product. This means that releases are cheaper and less intrusive to install.

A **release pipeline** can be broken down into a few main phases of a software deployment:

- Integration: developers write code and *integrate* their changes into the central repository.
- Build: The build phase is built up of 4 steps.
  - Configure decide which pieces do we include in the output
  - Construction: compile, link etc
  - Certification: test
  - Packaging: bundle everything into one package, jars, dll's, ect.
- Deploy: Get the package out to customers
- Monitor: Keep an eye on the usage of the system, run health checks and make sure everything is good. Check UA to inform future decisions / releases.

Part III
Version Control

Defining version control:

Version control allows you to manage multiple revisions of the same documents. Often it allows multiple users to share and work on the same documents concurrently through a centralized repository, and the VCS (version control system) will sort out which lines changed and handle merging the versions together.

The VCS will allow you to save disk space by reducing copies saved on client and recover from accidental deletes/edits.

4 Git

Git is a *Distributed Version Control System* (DVCS). This means that you can make commits and changes on the client without needing a connection to a centralized server as you would need in a (CVCS). In a CVCS, branch and merge operations must be created and handled by the server. However DVCS allow users to create and merge branches as simple lightweight operations on their own machines.

4.1 Git Client Operations

A **git clone** operation creates a working copy of the remote repository on your local machine. Your machine now has its own version of the remote repository. You can make changes, look at the history and do basically whatever you want, without affecting the remote repository.

A few interesting commands for exploring a repository:

- **git log** to see the commit history. Note that commits are identified with commit hashes. Each log will contain this hash, the author, date and message for the commit.
- **git branch -a** lists all the branches, and **git tag** shows the tag history. A **tag** is just an alias for a commit hash.
- **git checkout ¡commit¿** lets you set your local workspace to match that of a given commit. (Possibly traveling in time)

Making changes is easy:

- **git add ¡files¿** to stage files you've changed
- **git commit -m "message"** to commit all staged files with a message to the currently checked out branch.

4.1.1 Commits

Remember that a branch is really just a pointer to a specific commit. So when "making a commit onto a branch" we're really just moving this pointer to the new commit. Suppose initially we only have one commit, $6cabc07$ on a branch called *master*

$$master \rightarrow 6cabc07$$

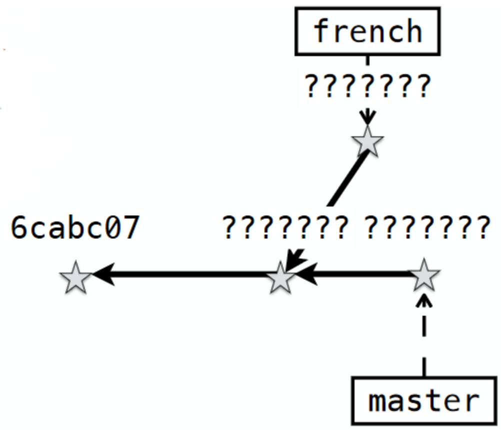
now suppose we add a new commit which is a child of our previous commit (is based on those changes).

$$6cabc07 \leftarrow 3aa8f7b \leftarrow master$$

So now master points to our new commit, and the new commit points to its parent, which is our old commit.

4.1.2 Merges and Rebasing

Now suppose you have multiple branches to organize your work. Say you were working on the master branch for a while, making commits, then decide to translate your app to French. Then you create a *french* branch and make commits onto there. What you'll end up with is a *tree* structure something like this:



(Image taken from Prof Shane McIntosh's slides.)

Also notice that master continued to have new commits that do not affect the french branch. Say we want to merge the french changes into the master branch, we can do so by doing a **git merge** command, which creates a new commit that has two parent commits. These parent commits are the heads of the two branches involved in the merge.

Problems arise when the same parts of a file were modified on both the target branch and source branch of a commit. These are called *merge conflicts*. A common complaint with merging is that it throws all the conflicts from all the commits at you at once, and it can be difficult to resolve.

git rebase alleviates this pain by performing an equivalent (but different) operation to a merge. Often, this operation is performed *before* pushing changes to be merged to a master branch, to ensure it will proceed smoothly. Rebasing works by *rewinding* the changes on your branch, by pushing them to a stack in the order of following parent pointers until a common commit between your branch and the target branch is found. Then, it will pop each commit off of the stack and attempt to apply it to the head of the target branch. If there are conflicts you deal with them interactively at each pop of the stack. The end result is that your feature branch is now fully up to date with the target branch.

Following a rebase, a merge is trivial, since all it needs to do is move the pointer for the master branch (also called *fast-forwarding*).

4.1.3 Undos

There are several types of undo operations:

- **git reset <paths>** is the opposite of **git add <paths>**

- **git reset -p <paths>** will interactively select portions of code in the diff for the selected <paths> and chosen hunks of code will be unstaged. This is the opposite of **git add -p <paths>**
- **git reset –hard <commit>** Reset your HEAD pointer to a previous commit and discard all changes since then.
- **git reset –soft <commit>** Reset HEAD pointer to a previous commit but do not discard all changes. They will show up as ”changes to be commited” in a **git status** call.
- **git revert <commit>** apply a new commit which exactly undoes the changes of the specified <commit>. Note that reverting any other commit than the most recent may have consequences since other commits may depend on the reverted changes.

4.1.4 Pushing and Pulling

To integrate your changes to the remote server, you must **git push <remote> <branch>** to publish your local copy of <branch> to the specified <remote> repository (remember that git is a DVCS so there may be several remote repositories).

To get download changes from the remote server, you can use **git fetch <remote>**. Note that this does not integrate the changes into your HEAD working copy. To do this you must **git pull <remote> <branch>**. Note that **pull** performs a **fetch**, so it is not necessary to call fetch. Fetch is really just for if you want to double check what the remote changes are before deciding to integrate them with your local stuff.

4.2 Git Server Operations

Each Git repository can act as a server or a client. But typically, a server-only repository is a *bare* repository named with the *.git* extension. To create a bare repository, use **git init –bare**.