ECSE 437 Study guide

Francis Piché

October 2, 2019

Contents

Part I
Preliminaries

1 License Information

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*

$$master \rightarrow 6cabc07$$

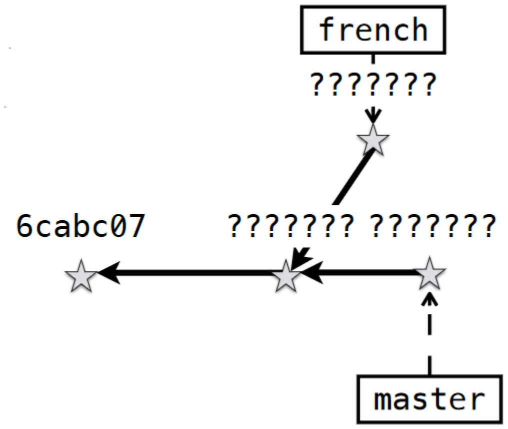
now suppose we add a new commit which is a child of our previous commit (is based on those changes).

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

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 Servers

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**.

Clients may interact with the git server over **http, https, ssh, git** protocols, or using the local filesystem if the server is on the same host as the client.

4.3 Git Internals

Each git repository has a **.git** directory at the root of the repository containing information that git uses to operate. We can go in and lookat / modify this to get new interesting behaviors out of git.

4.3.1 Object Database and Refs

Under the **objects/** directory, we have a key-value store. This database is the core of how Git works. There are 3 types of values stored at these keys:

- Commits: A reference to a committed change tracked by Git. This commit refers to a *tree* object which describes the state of the repository at the time of that commit.
- Tree: A directory being tracked by Git. Contains blobs, and subtrees
- Blob: A file being tracked by git.

So basically, commits are references to a root-level tree object, trees contain (possibly) subtrees and blobs.

The **git cat-file <type>** command can be used to see the content of an object. The <type> parameter can be skipped if -p or -t is used to find the object type. If you use this command on a commit reference, you will see the object hash of the tree, this commits parent, the author, date, and message. You could then cat-file the tree to see the individual blob hashes.

The **refs/** folder contains all the aliases for your references. This includes branches (heads), tags and remotes. This is how git keeps track of what points to where. If you open one of these ref files (master) in heads/ for example: you'll see the hash for that object.

4.3.2 Hooks

Git hooks are stored in the **hooks/** folder. These are scripts that are executed when certain git actions are performed. There are client-side hooks and server-side hooks.

Client side hooks (non-exhaustive list):

- **pre-commit** Before commit has been executed. Useful for enforcing code-style stuff or running tests before committing.
- **prepare-commit-message**. Is passed $1: Path to the file containing the message, $2 Commit type $3 SHA if the commit type is amended. This is run before the commit message editor has been launched, after the template has been prepared. This is used mostly for formatting templated commit messages (merges, reverts etc).
- **commit-msg** $1: The path to the file containing the message. This runs after the commit message is saved, but before the commit is saved to the database. This is mostly used to enforce commit message formats on the client-side.
- **post-commit** After the commit has been written to the object database tis script runs. Its mostly used for notifications.

The problem with client side hooks is that every client has to enable it, so its pretty easy for someone to forget, modify or otherwise throw a wrench in your system. For this reason client-side hooks are best used as "personal convenience" for developers, rather than enforcing team standards. But if everyone acts in the best interest of the team then they can work for that as well.

Server Side hooks:

- **pre-receive** Runs when handling a push from a client. Reads arguments from stdin. If non-zero exit status, non of these refs are accepted. This can be used to ensure protected files are not modified, or that everyone rebases their pushes (by checking that each updated ref is a fast-forward)
- **update** Arguments: $1 <old-ref> $2 <new-ref> $3 <ref-name>. Similar to pre-recieve, but runs once for each branch the pusher is updating. If there are multiple branches being updated pre-receive runs only once, wheras this will run once for each.
- **post-receive** This runs after the whole process is completed and can be used to update other services or send notifications. This script cannot stop pushes, but the client wont disconnect until this script completes.

Part IV
Build Systems

A build system is one that turns source code into a deliverable product. For a Java project for example, this might mean turning .java files into .class files through compilation, then bundling these to an executable .jar file. Documentation into a .html file through static-code analysis of source code, and maybe the .jar files and .html files get compressed into a .zip file which is the final deliverable. This is just one example of a "build".

The importance of build systems is very apparent when it comes to C programs. Without build systems, we would need to compile each .c file, link the .o files into the actual exectuable. But if there is a very large number of these files, we would need to remember or keep track of a huge process!

That process would need to be repeated very often, since as developers, we often go through the "Think → code → build → run" cycle often in the development process.

5 Paper: A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance

The build process is broken down into a few steps:

1. Configuration: User chooses which parts of system to change
2. Construction: Compilers etc in an order satisfying dependencies
3. Certification: Run tests to ensure nothing broke
4. Packaging: Bundle deliverables together with libraries, documentation and data files.

Low level build systems are those which explicitly define dependencies between input and output files, and specifies scripts on how to resolve these dependencies. **Make** and **Ant** are examples of low-level systems.

Abstraction Based build systems automatically generate low-level specifications based on high-level criteria. **CMake** is an example of such a tool. It will take care of platform-specific configuration by detecting the platform etc.

Framework-Driven build systems use *convention* rather than explicit *configuration.* **Maven** is an example of one of these systems. It assumes your code is organized in a specific structure. If convention is assumed to be followed, the framework-driven tool can infer build requirements automatically.

Dependency Management tools find and acquire required libraries and tools for a project automatically. For example, Maven has a way to state which packages you need, and (possibly) the version. If the requirement is missing it will pull it from a repository.

6 Low-Level Build Systems

6.1 Make

Using a low-level build system can be broken down into 2 steps:

1. Expressing dependencies
2. Writing Recipes

For make, this is done with a **Makefile**. This file expresses a dependency graph structure and a set of scripts to execute to resolve these dependencies if: they are missing, or if a dependent file was modified.

An example makefile:

```
# program is the target (output file), and the others are files which the target
    depends on (dependencies)
program : source1.o source2.o source3.o main.o
   # The next command is the recipe dictating how to generate the target if it is
       missing or one of its dependencies was modified more recently than the
       target.
   gcc -o program random.o input.o main.o

# Equivalent syntax:
program : source1.o
program : source2.o
program : source3.o
program : main.o
program :
   gcc -o program random.o input.o main.o

# Dependencies of one target can be its own target with its own dependencies
source1.o : source1.c
   gcc -c source1.c

# Equivalent syntax:
source1.o : source1.c ; gcc -c source1.c

# We can also use variables:
compile = "gcc -c source2.c"
source2.o : source2.c
   $(compile)

# Warning! Variables use lazy evaluation
filename = "source2.c"
command = "gcc -c $(filename)"
filename = "something_else.c"
source2.o : source2.c
   $(command) # Will run gcc -c something_else.c

# Other .o files would have similar target-recipe pairs
```

Running the **make** command will look for the Makefile and run the first target in the file. For each missing dependency in the dependencies it will look for a target matching the dependency name. It will then attempt to resolve this target the same way. This essentially is a tree traversal of dependencies. Once a leaf is reached (a target where all dependencies exist) it will be able to resolve all the way back up to the root (original target).

We can also pass specific targets as arguments to the command to be able to build just parts of the system.

Note that only out-of-date targets are built! A target is out-of-date when it is older than one of its dependencies.

Errors are raised if a dependency is a target of another rule, but that rule has dependencies that do not exist.

6.1.1 Problems with Make

The issue with make is that the scripts are platform specific. This means you would often have to write multiple makefiles so that people on different platforms can run the build. Also, the since make cannot handle dependencies across directories, recursive calls to make are made for each subdirectory, but then each invokation has no knowledge of the global dependency graph, so multiple passes must be made (slow and unreliable).

Make also doesn't jive well with Java. Since the Java compiler can resolve dependencies on its own, this interferes with the explicit nature of the Makefile, since make can't know which output files are generated for a given input file. For example, if we have a java file which imports other java files, running javac compiler on the one file will create .class files for all of the imported files.

6.2 Ant

Ant is meant to be a replacement for *make* in Java systems. Ant uses a task-based execution model, where tasks are executed in java code, making them more portable.

Ant also doesn't track dependencies at the file level, and is aware for the potential for multiple output files from a single input file, and can update its dependencies in consequence.

6.2.1 Ant Concepts

- Properties: These are basically variables for storing and using commonly used elements.
- Tasks: These are command invokations, either given by the Ant API or written yourself.
- Targets: A grouping of tasks used to accomplish some broader goal. Like how you group statements into a function in programming

Ant specifications are written in XML format in a **build.xml** file. Example:

```
<project name="DiscountStrategy" default="compile">
  <property name="blddir" location="build" />
  <property name="classdir" location="${blddir}/classes" />

  <target name="init">
    <mkdir dir="${blddir}" />
    <mkdir dir="${classdir}" />
  </target>
  <target name="compile" depends="init">
    <javac srcdir="src" destdir="${classdir}" />
  </target>
  <target name="clean">
    <delete dir="${blddir}" verbose="true" />
  </target>
</project>
```

Here, the **project** node wraps the whole project and specifies the name of the project as well as the default target to run if not otherwise specified in the **ant** call.