

## CHAPTER 3

### CHANGE MANAGEMENT

#### 3.1 Why change management

Assume that we have 2 people working on the same code base, as shown below:

Alice	Master	Bob
<pre>def say_hello():     print("hello")  def say_world():     print("world")  def say_hello_world():     say_hello()     print(" ")     say_world()</pre>	<pre>def say_hello():     print("hello")  def say_world():     print("world")  def say_hello_world():     say_hello()     print(" ")     say_world()</pre>	<pre>def say_hello():     print("hello")  def say_world():     print("world")  def say_hello_world():     say_hello()     print(" ")     say_world()</pre>

The master version of the code is the final, centralised version that combines both Alice and Bob's work. Then, Alice refactors code (makes the code simpler), and Bob extends code.

Alice	Master	Bob
<pre>def say_hello_world():     print("hello world")</pre>	<pre>def say_hello():     print("hello")  def say_world():     print("world")  def say_hello_world():     say_hello()     print(" ")     say_world()</pre>	<pre>def say_hello():     print("hello")  def say_world():     print("world")  def say_hello_world():     say_hello()     print(" ")     say_world()  def say_goodbye_world():     print("goodbye")     print(" ")     say_world()  def say_hello_and_goodbye():     say_hello_world()     say_goodbye_world()</pre>

Now, Alice and Bob both want to update the master version of the code. If Alice copies first to master then Bob, we end up at the following state.

Alice	Master	Bob
		<pre>def say_hello():     print("hello")  def say_world():     print("world")</pre>
<pre>def say_hello_world():     print("hello world")</pre>	<pre>def say_hello_world():     say_hello()     print(" ")     say_world()  def say_goodbye_world():     say_goodbye()     print(" ")     say_world()  def say_hello_and_goodbye():     say_hello_world()     say_goodbye_world()</pre>	<pre>def say_hello_world():     say_hello()     print(" ")     say_world()  def say_goodbye_world():     say_goodbye()     print(" ")     say_world()  def say_hello_and_goodbye():     say_hello_world()     say_goodbye_world()</pre>

In this case, Alice has wiped the first file, but Bob overrides the second file- the function `say_hello_world` does not work since `say_hello` and `say_world` have gone missing. So, the code is broken.

Instead, if Bob copies first to master then Alice, we end up at the following state.

Alice	Master	Bob
		<pre>def say_hello():     print("hello")  def say_world():     print("world")</pre>
<pre>def say_hello_world():     print("hello world")</pre>	<pre>def say_hello_world():     print("hello world")  def say_hello_and_goodbye():     say_hello_world()     say_goodbye_world()</pre>	<pre>def say_hello_world():     say_hello()     print(" ")     say_world()  def say_goodbye_world():     print("goodbye")     print(" ")     say_world()  def say_hello_and_goodbye():     say_hello_world()     say_goodbye_world()</pre>

In this case as well, Alice still wipes the first file, meaning that the new file doesn't compile- the function `say_hello_and_goodbye` does not work since `say_goodbye_world` has gone missing. So, the code is broken.

So, it doesn't matter who compiles first- the code still breaks after the changes get made. This is why we need a change management system. Without careful change management, there is no way of knowing what version of the constantly changing code base has been used during the process.

For example, the results from code testing is useless without code management. It is not possible to tell what version the report applies to. Moreover, the code might have already been changed/bugs fixed since it was test. This also applies to:

- compilation,

- dependency management,
- code reviews,
- static analysis,
- refactoring,
- deployment and rollback,
- bug reporting and triage.

### 3.2 What are change control items

Any item that is directly edited by the team is a change control item, e.g.

- source code files,
- build scripts or configuration files,
- default application configuration files,
- user documentation,
- requirements specification,
- version control tool configurations, etc.

Anything indirectly generated by the manually edited files are not change control items, and should not be committed to the version control system, e.g.

- compiled binaries,
- third party libraries,
- compiled document formats, e.g. PDF, PS,
- auto-generated source code files,
- client side IDE configuration,
- log files, etc.

Some people argue that we should store non-control items in a SCM (source code management), since:

- the source code takes a long time to compile- storing cached binaries will make this faster.
- compilations cannot be automated- someone has to press a button on the GUI.
- third party libraries aren't available in a public release repository.

These are bad reasons to store non-control items in the repository. It is not a file storage system. In fact, we can solve these issues using release and dependency management. For example,

- we can break code into smaller modules with separate compilation pipelines. Each module can be compiled easily and in parallel.
- agree and configure a release management process for each module using a release repository. This can depend on libraries provided by others, and also publish the ones developed by the team.

### 3.3 Version Control System

A version control system (e.g. git) is a centralised repository that the team can access to and make changes. This allows us to control change concurrently. The team can make changes to a working copy of the code from the centralised repository. When a member is ready to commit, they check if the repository has been updated. They receive changes from the repository and resolve any conflicts that emerged. They repeat this step until there is no change made to the repository since the last pull. At that point, they commit/push the change to the repository. A simple representation of this is given below.

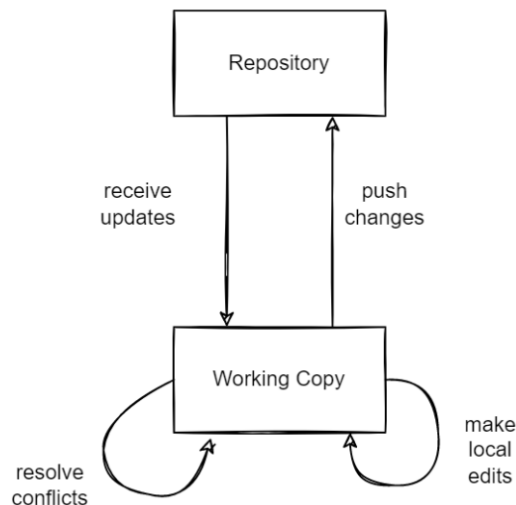


Figure 3.1: The workflow in a version control system

A change control system can be centralised or distributed:

- In a centralised version control system, there is one single centralised repository on the server. Everyone has working copies on their machine, which they can use to push their code to the repository. The following figure summarises this.

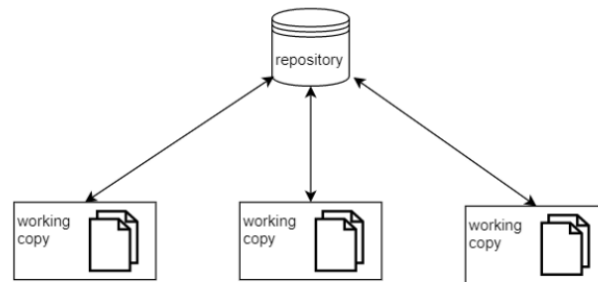


Figure 3.2: A centralised version controlled system

- In a distributed version control system, there are multiple repositories. Every developer has a local repository. They can check out a working copy and work on it. Changes from the local repository can be pushed/pulled from other repositories amongst the team (like peer-to-peer distribution). So, each member has a local repository and they can push from each other's repository across the network. The following figure summarises this.

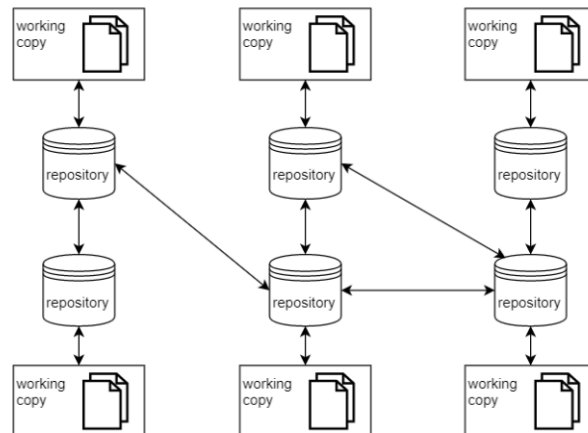


Figure 3.3: A distributed version controlled system

In practice, it is quite difficult to support distributed version control system. Instead, most version control systems are hybrid- they have a centralised arrangement in a distributed fashion. So, each member has a local repository, and there is a central external repository. The team works on a local copy of the repository, and synchronise with the centralised, external repository. The following figure summarises this.

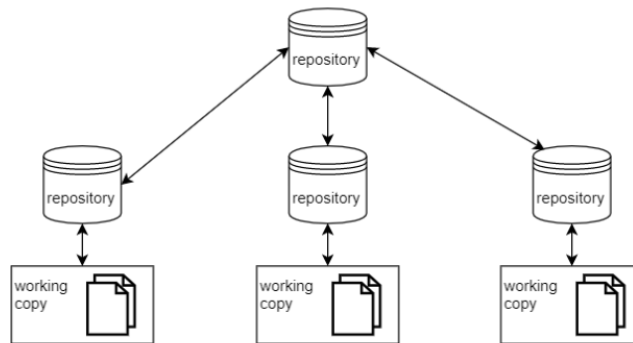


Figure 3.4: A hybrid version controlled system

### 3.4 Commits

Commits can be thought of as either a package of information needed to describe the change since the previous commit, or the complete snapshot of source code at a time. Both views are equivalent.

A commit has:

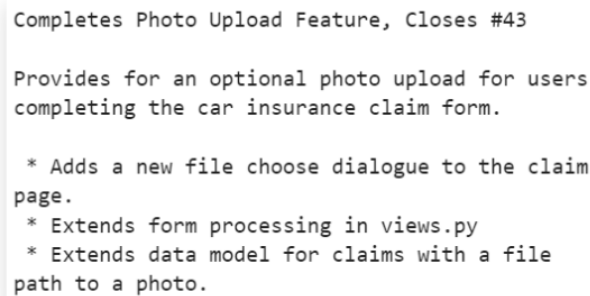
- a unique id;
- the id of the parent commit(s), to identify what code has been changed;
- changes to existing control items;
- any added/removed change control items;
- metadata, e.g. author, timestamp, and log message.

To produce the commit id, we can use a counter in a centralised version control system. However, this is not possible in distributed systems. Instead, they use a one-way hash function that combines all the information in a commit (i.e. previous commit ids, metadata, etc.) to produce a fixed-length id. There is a very low probability that two commits will get the same id.

The commit log message should:

- have a short, meaningful title;
- describe the intent of the commit;
- link to the issue in the project tracker or the merge request;
- explain how the commit addresses the issue.

The message does not need to explain what code was changed- we can see that from the files changed. It does however need to explain why and how the code was changed. An example of a commit message is given below.



```
Completes Photo Upload Feature, Closes #43

Provides for an optional photo upload for users
completing the car insurance claim form.

* Adds a new file choose dialogue to the claim
page.
* Extends form processing in views.py
* Extends data model for claims with a file
path to a photo.
```

Figure 3.5: A commit message.

### 3.5 Branches

Branching is the practice of maintaining multiple development lines based on a common history within a single version control repository. A commit has one parent; but the parent can have multiple children commits, all of which will be part of different branches. We can have multiple branches of concurrent development in parallel versions of the repository. It is useful for:

- experimenting with implementation of new features- it may take several commits to implement this feature, and it may not work the first time round.
- undertaking a substantial reorganisation- experiment a big reorganisation of the repository without disrupting others' work. This version of the repository might be adopted later.
- supporting a specialised variant to an application- some version of the code may always be different. We maintain this variant out of the main branch.
- creation of a release- this is a branch where no more features are added, and we only resolve bugs. This allows us to keep adding features to the main branch (which will be part of a future release).

Different branching practices can be combined into a branching strategy for a project/team.

There are 3 main types of branching:

- Trunk-based development. In this case, there is only one branch (called main, master, or trunk). The developers commit to that single branch.

This is summarised in the image below.

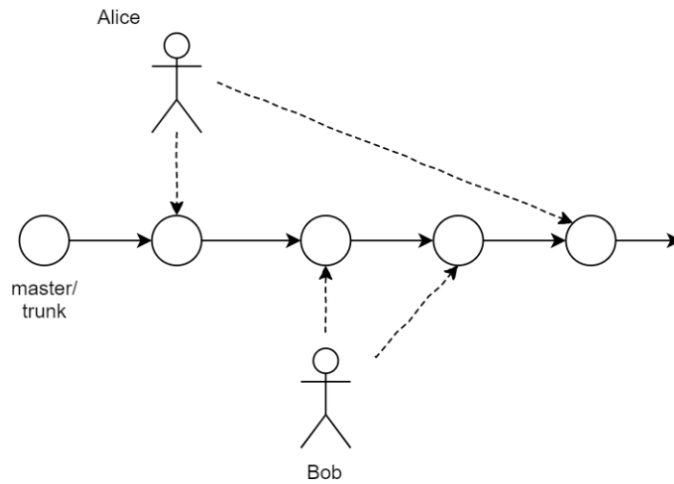


Figure 3.6: Trunk-based development

The advantages of this approach is:

- the workflow is simple;
- everyone can see the current state of the project easily;
- it forces developers to break changes into small increments of change; and
- it reduces the risk of merge conflicts.

A disadvantage of this approach is there is a higher risk of broken code—a commit can have broken tests/broken continuous integration pipelines.

- Feature branching. In this case, there is one branch for each feature. There is a master branch. The team works on different features over different periods in different branches. This allows for concurrent development with the master branch. So, there are more frequent commits to the features branches. When features are ready, we first merge the master branch to the feature branch, and then merge the feature branch to the master branch.

The advantages of this approach is:

- the team can develop features in isolation; and
- it is more convenient to perform code reviews.

A disadvantage of this approach is that this can encourage longer periods between integration.

This is summarised in the image below.



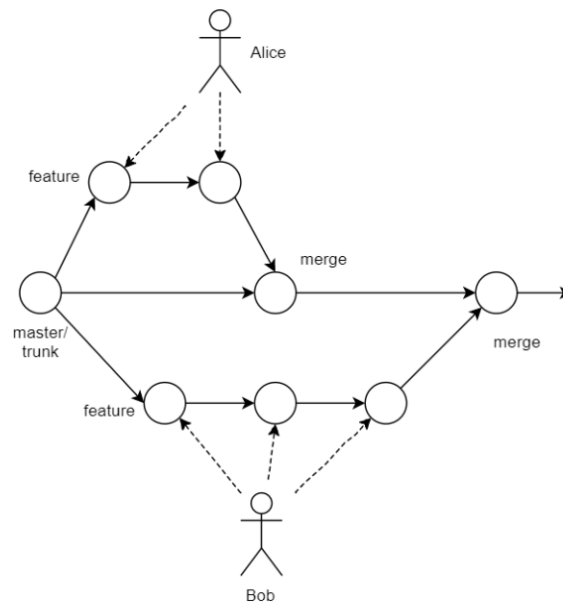


Figure 3.7: Feature branching development

- Staging branching. In this case, there is a branch that runs parallel to the master branch. This is different to feature branches, which are temporary. This allows changes that have been committed to spend some time in a UAT (user-acceptance testing) environment. Then, it gets merged to the master branch after user testing. In contemporary software development, some teams configure UAT environment for each feature branch- this combines both staging and feature branching. The following image summarises this approach.

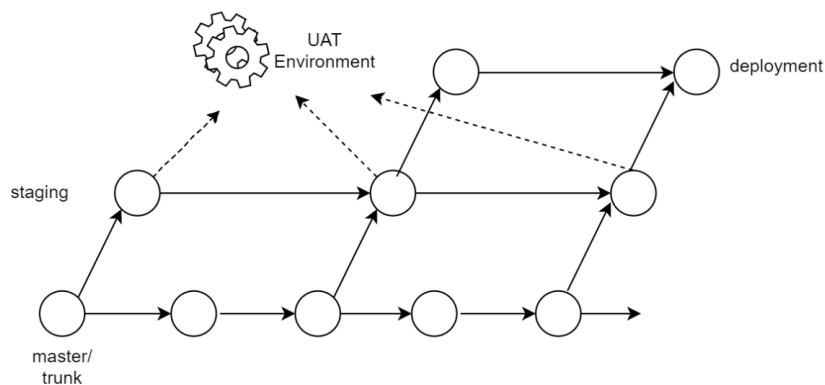


Figure 3.8: Staging branching development

When naming branches, we should following conventions to enhance project

maintainability. For example, the main branch of the repository is called the master/trunk branch. It contains the primary version of the project. The user acceptance testing branch can be called staging, UAT or preprod(uction). Similarly, the production branch can be called deploy(ment) or prod(uction). When naming feature branches, we can follow a hierarchical approach, e.g. feature/webapp/splash.

In a software development project, a team follows a collection of (often customised) branching practices. There are many standard published branching strategies, e.g. Gitflow, Github flow. Most teams tailor their strategy to suit their circumstances.

It is common to manage branch growth. It is a good idea to delete branches after there is no further use to it. Also, commits can be squashed- we can merge all commits on a feature branch into one commit before merging the branch itself to the main branch.

In summary, disciplined change management, supported by an automated version control system, is fundamental to software engineering.