Source Control Management

Source control management is a critical component of any software engineering project. Software Engineering is a team activity which means that multiple people are working on developing the code at the same time. As a result, problems can occur resulting in lost productivity and possibly lost code. A proper source control management process can greatly reduce the likelihood of problems, and off a path to the solution when problems do occur.

In this reading you will learn about:

- What is Source Control Management (SCM)?
- Why is SCM important?
- What are the key concepts of Git?
- What are best practices for SCM?
- What Git workflow should your team use?

What is Source Control Management (SCM) and why is SCM important?

Up till this point you have primarily worked on software projects as an individual. Have you ever done one of the following:

- Broke your own code after making a bunch of new changes, and you just want to get back to your last working version?
- Accidentally deleted a file, and lost hours of work?
- Somehow messed up the file structure of your project, and you can't remember how you had the files arranged when the code worked?

For those of you who have worked on team projects, there may have been other questions that came up such as:

- Whose computer stores the master copy of the project?
- How do you update the rest of the team with your changes?
- What happens when multiple team members try to update the same file?
- How do you know what the rest of the team is working on?

Source Control Management (SCM) a.k.a. Version Control System (VCS) can provide a solution to most of

these problems. A VCS provides a way to track and maintain changes to files. When working on in a team setting there are the additional benefits of knowing who made the changes to the files, why they made the changes, and when they made them. A system like this allows each member of the team to work together confident in the fact that no one person (including themselves) can bring down the whole team, because you can always revert back to working code. Think of a VCS as a very advanced form of the the control-z feature of most text editors.

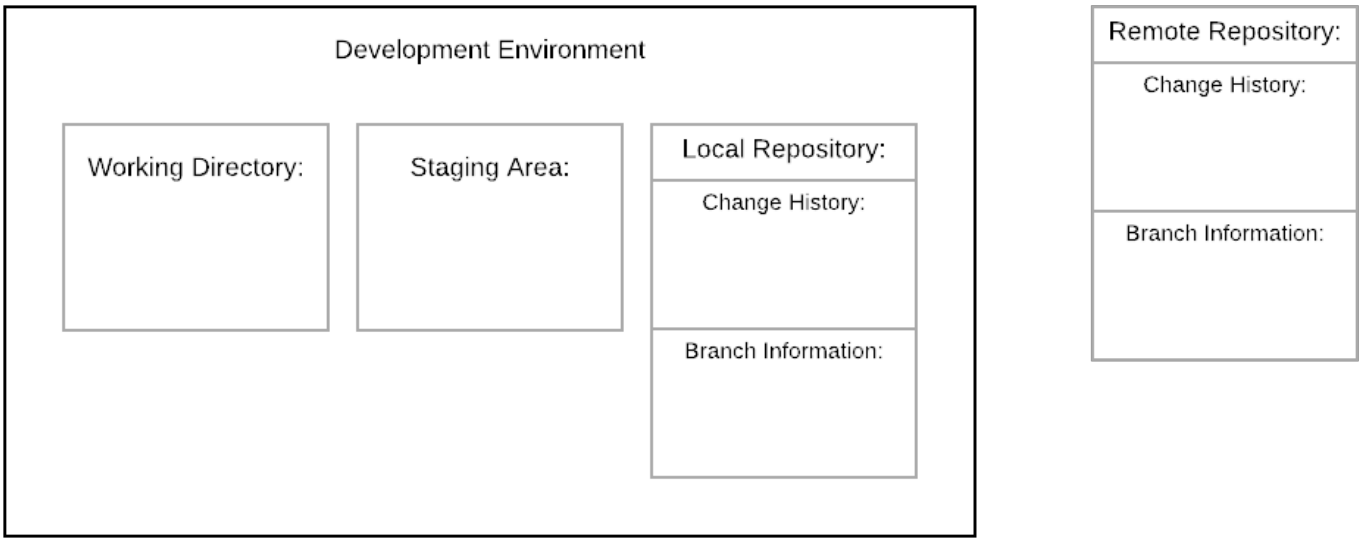
There have been several popular software version control systems over the years including CVS, SVN, Git, and Mercurial to name a few. What all these systems have in common is that they rely on a repository for the code which: * Contains a shared copy of all the files * Maintains a version history of all the files as well as a backup of past versions

Where these different version control systems differ is in how many repositories there are and where the repositories are stored. Earlier versions of VCS relied on a single centralized repository which existed on a server which was accessible to the whole team. Newer and now much more popular VCSs rely on a distributed repository model in which each member of the team has a local copy of the repository in their development environment and there exists a central remote repository for sharing files among team members.

Key concepts of Git

To understand the key concepts of Git you must first understand the notions of the local development environment and remote repository. The local development environment is where you do work on your local machine. In Git this area is broken into three parts.

1. Working Directory (aka Working Tree)
2. Staging Area (aka index)
3. Local Repository



The Working Directory is where you keep your files for the project. There are two categories of files in the Working Directory.

1. Files that git knows about called *Tracked Files*.
2. Files that git doesn't know about yet called *Untracked Files*.

The command `git status` will tell you the current (tracked/untracked) status of all files in your working directory along with other useful information that we will cover as needed.

Only once files have been *added* to the *staging area* do they become *tracked*. Once you have *added* all the files that you want to store in your local repository to the *staging area* then you can *commit* those files which stores a snapshot of your files permanently to your *local repository*.

If you are working alone the local development environment will be everything you need to keep a revision history of your code and be able to revert to past versions easily.

However, if you ever want to share your code with others or you want to back up a copy of your repository off your local machine then you will want to utilize a Remote Repository. This repo will likely be stored on a server at a Git repository hosting site such as GitHub or Bitbucket.

To interact with the Remote Repository, we primarily use three commands:

1. Push
2. Pull
3. Fetch

When you want to share the commits in your local repository with the remote repository you can use the *push*

to command which sends the changes from the Local Repository to the Remote Repository.

Provided there are no modified files in the working directory and no staged files a `git pull` command will get any changes in the Remote Repository and update the Local Repository and the Working Directory.

Another way to get the changes from the Remote Repository without updating the Working Directory or the Staging Area is to use the `git fetch` command.

There is much more to Git as you will see in the suggested work flow, but these understanding these key concepts are imperative to the more advanced concepts needed for effective team use of git. If you have uncertainty about the key concepts you should consider creating your own repository and practicing these concepts outside of your team repository.

What are best practices for SCM?

Commit Often

Think of commits a an advanced form of saving your files. Just like when you first learned to code you were told to save often to avoid losing work the same logic applies to committing often. By frequently committing your changes it is easier to revert (undo) to earlier versions in a more granular manner.

Pull Before You Work

Given that you are working on a team and other people are concurrently contributing to the codebase you should get in the habit of always pulling from the remote repository before starting to work on anything new. This will ensure that you are always starting to work on the latest version of the code, and it will cut down on conflicts when you got to merge your code later.

Good Commit Messages

Every time you make a commit you will also write a corresponding commit message that goes into the history. Focus the topic of the message on what was changed / added and why it was done. Don't get lazy and include short non-descriptive messages, and don't try to be clever and leave joke messages. These messages can and will be read by others and go into the history of the project.

Git Status and Git Diff Are Your Friends

Once you make a commit there is a permanent record of your changes added to the repository. You should utilize the staging area to double check exactly what you are about to commit and where you are about to commit it before taking action. By using `git status` and `git diff` commands you will have a clearer

picture of the situation.

Use Branches

Branches are easy to create and easy to delete once you don't need them anymore. By using branches, you will be able to work in parallel with other team members on different features. Upon completing the features, you can each merge your branches back into the common branch to combine your work.

Use Common Workflow

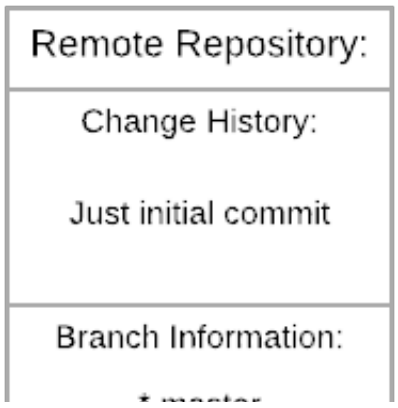
In this reading we are covering on particular workflow that we suggest for this class project. In reality there are many ways to be successful with SCM. However, it is extremely important that your team establishes a common process for creating and merging branches. Without which a greater amount of time will be wasted in dealing with merge conflicts.

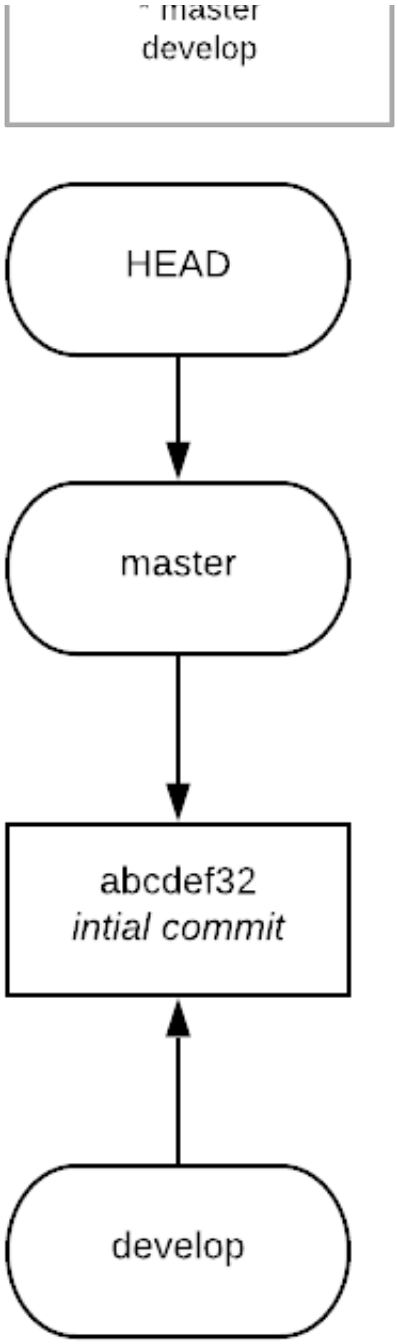
Suggested Workflow

For this class we suggest following a workflow similar to the Gitflow Workflow that was first published and made popular by Vincent Driessen at nvie. More information about this workflow and other options can be found here.

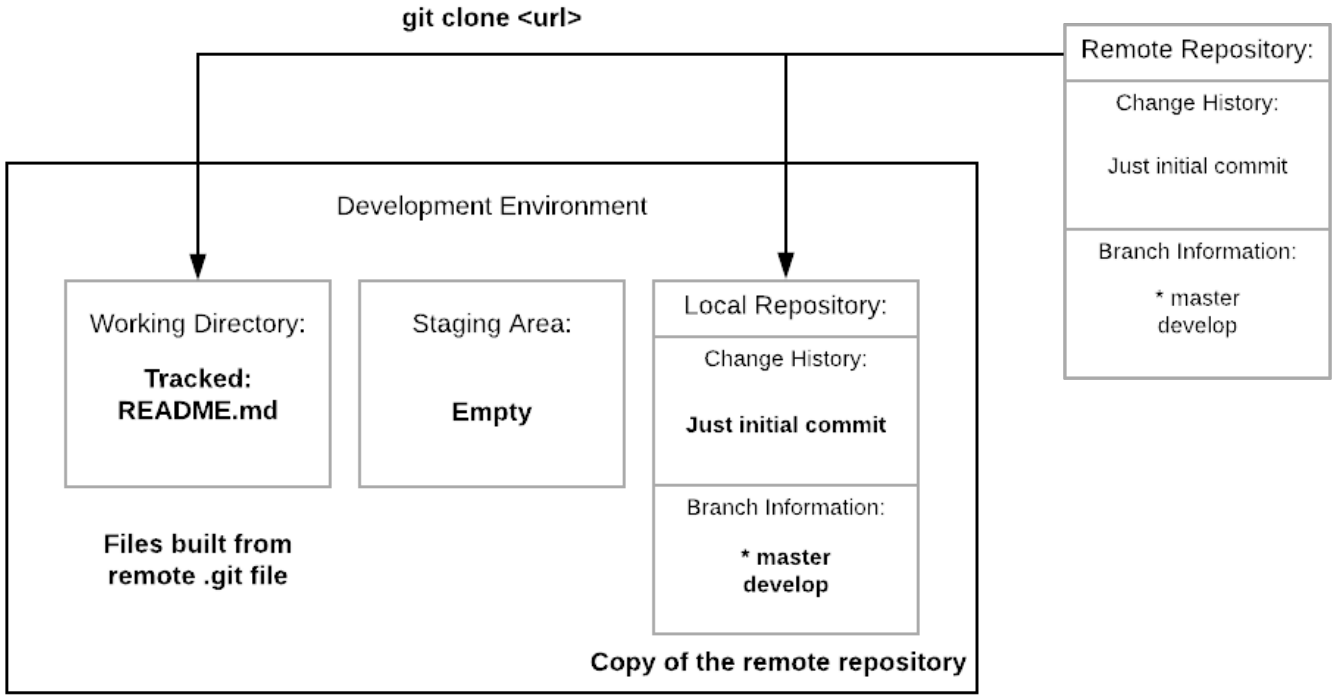
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

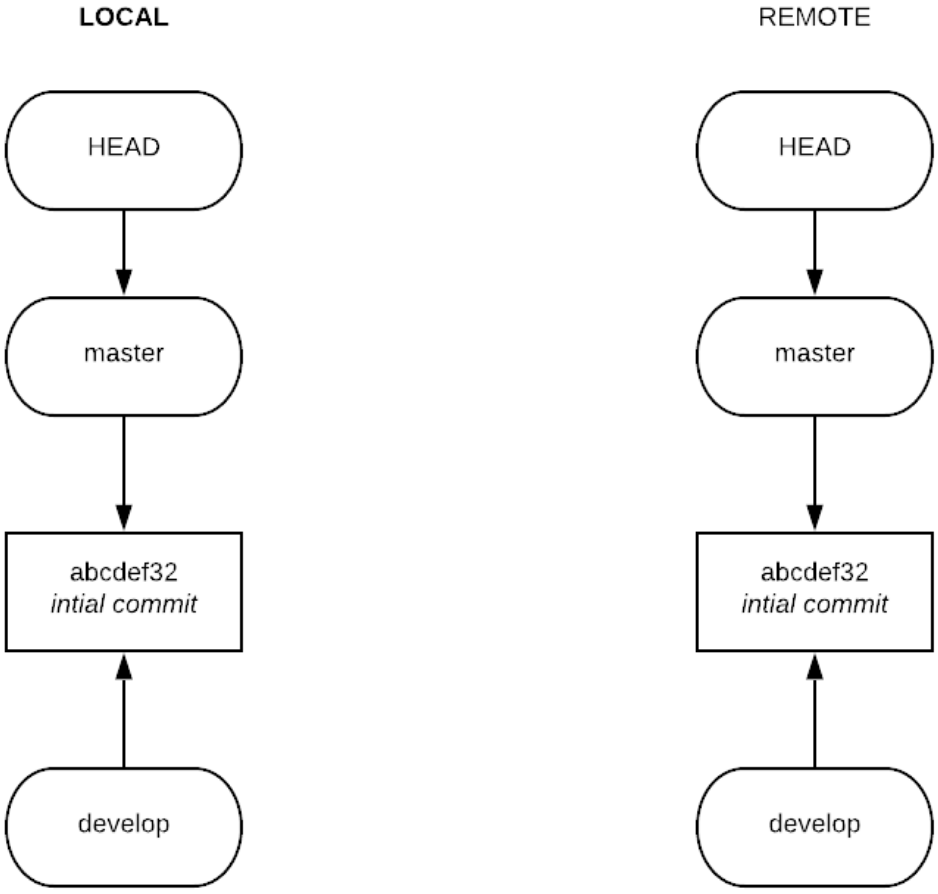
In this workflow you will use main branches to keep track of the history of your project, master and develop. The master branch will be *protected* and only be used to store release history for each Sprint. The develop branch will act as the branch where your team will integrate all the features developed by the team. Although this isn't a requirement of using git we recommend that your first step will be to create your remote repository on GitHub and invite the rest of the team as collaborators. You will also create a develop branch during this initial process.





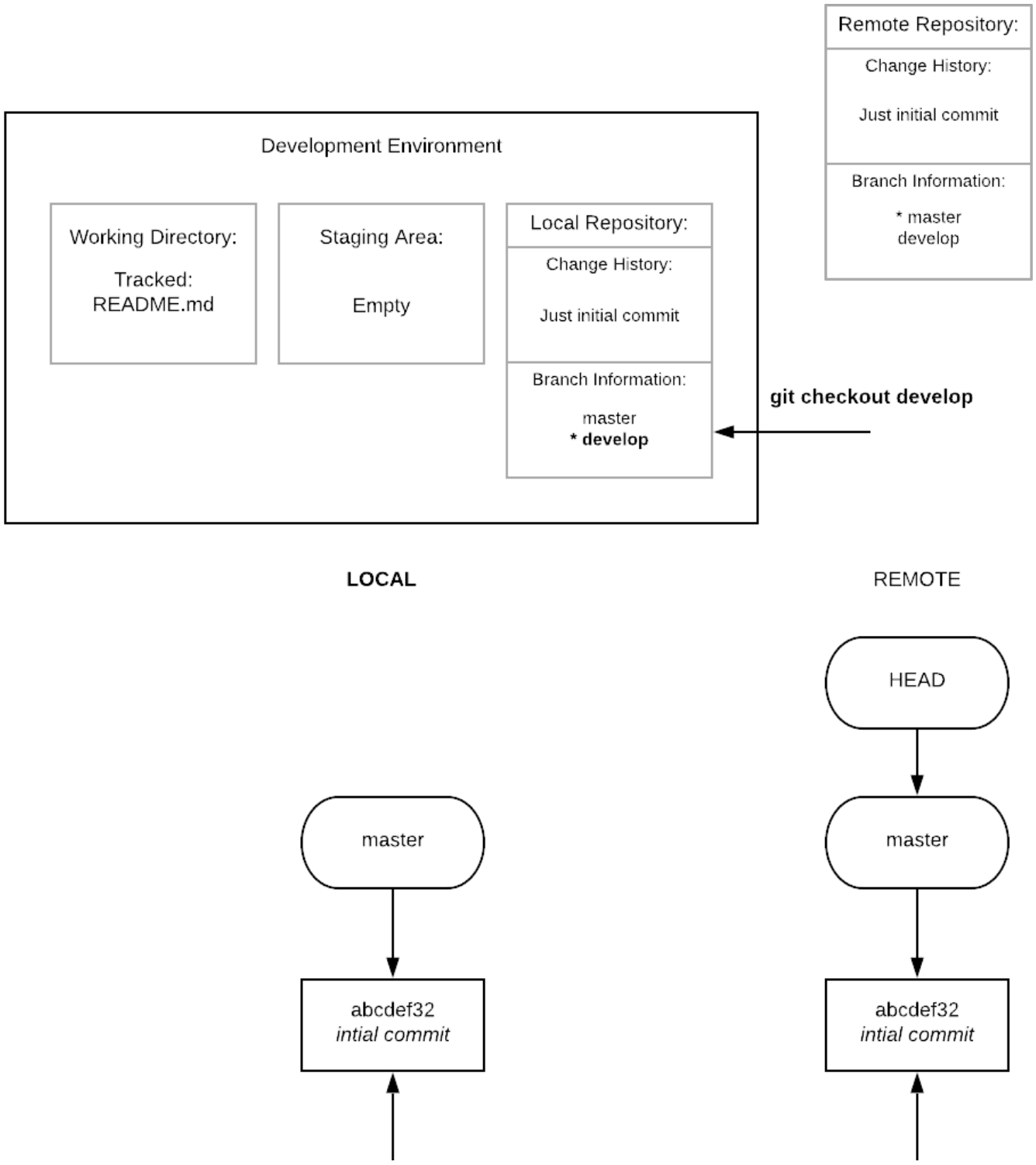
Then to create a copy of the remote repository on your local machine you should first choose where you want to be your local development environment. This is where you will put the repository. You do not have to create a folder for your repository. Git will do that for you. Use the command `git clone <url copied from GitHub>` to copy the remote repository (.git file) to become the local repository. Git uses the local repository to populate the working directory with the files you see on GitHub.


git clone <url>
Remote Repository:
Change History:
Just initial commit
Branch Information:
* master
develop
Development Environment
Working Directory:
Tracked:
README.md
Staging Area:
Empty
Local Repository:
Change History:
Just initial commit
Branch Information:
* master
develop
Files built from
remote .git file
Copy of the remote repository

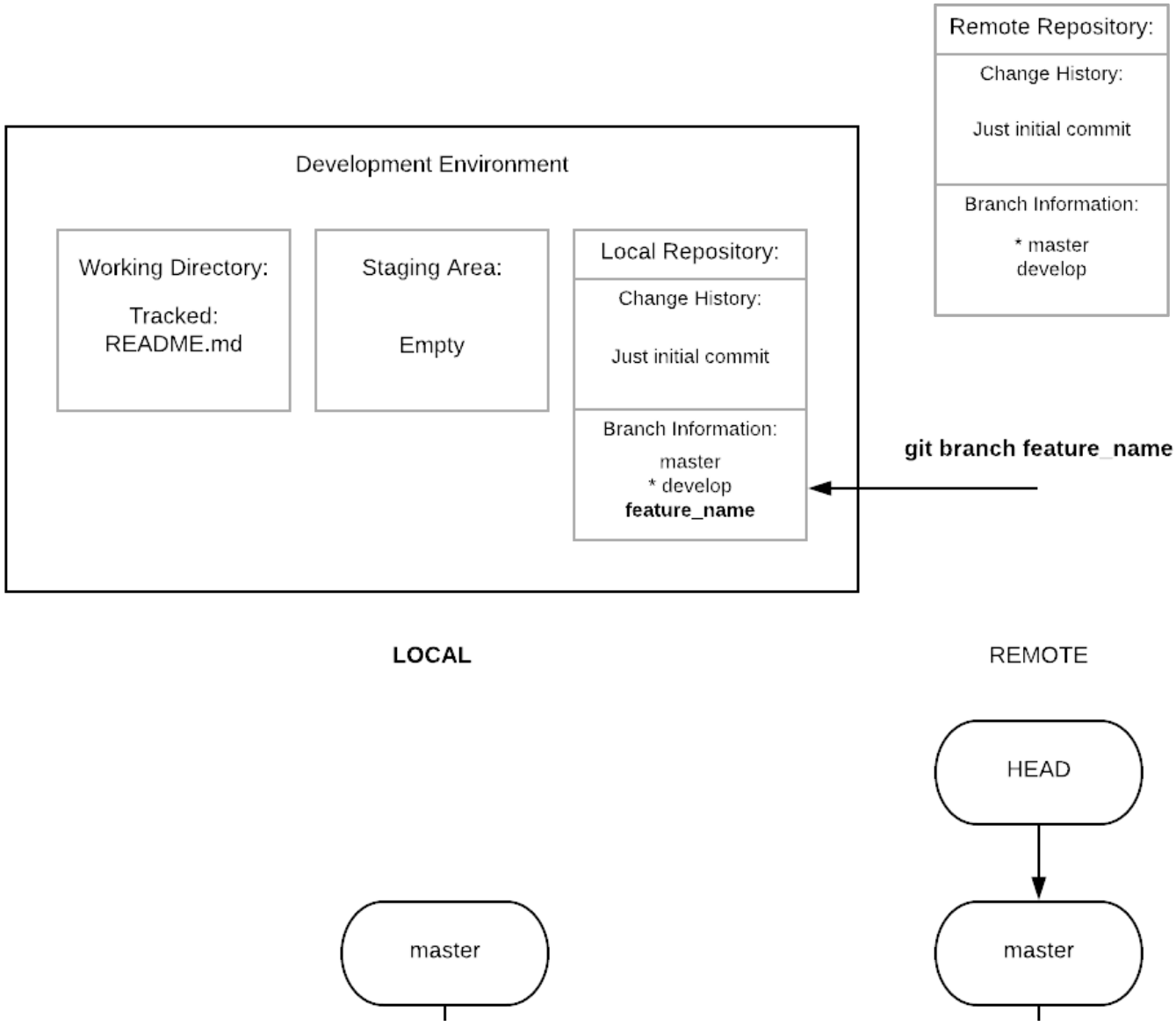

LOCAL
REMOTE
HEAD
master
abcdef32
intial commit
develop
HEAD
master
abcdef32
intial commit
develop

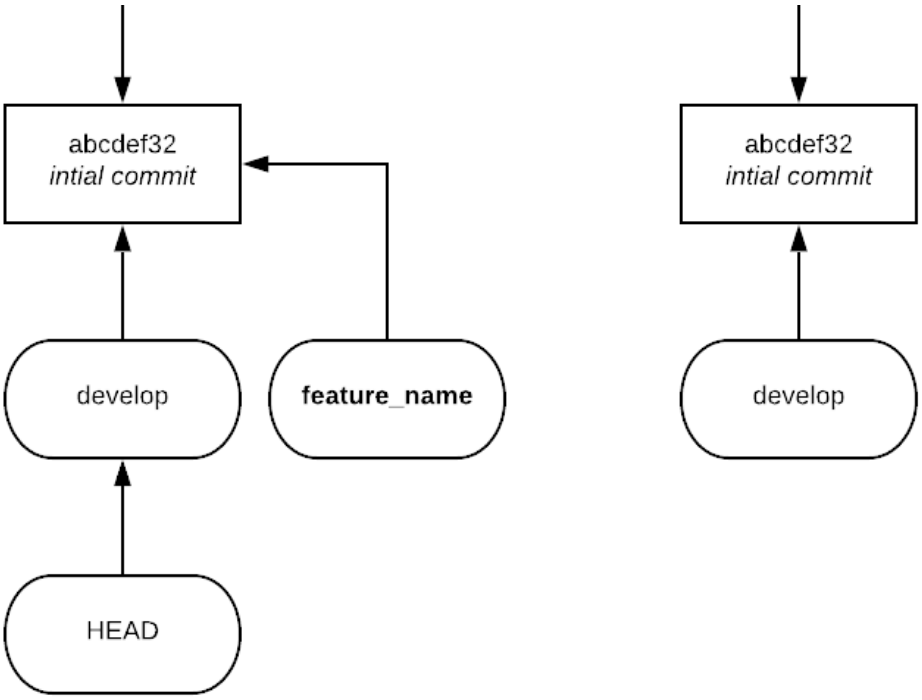
Now your local repository will match the remote repository exactly. You will have two branches in each repository, *develop* and *master*, and you will have one tracked files in the working directory of your local development environment. You are currently on the *master* branch as indicated by HEAD pointing to master.

At this point you should switch to the develop branch using the command `git checkout develop`. This will point the HEAD to the develop branch.

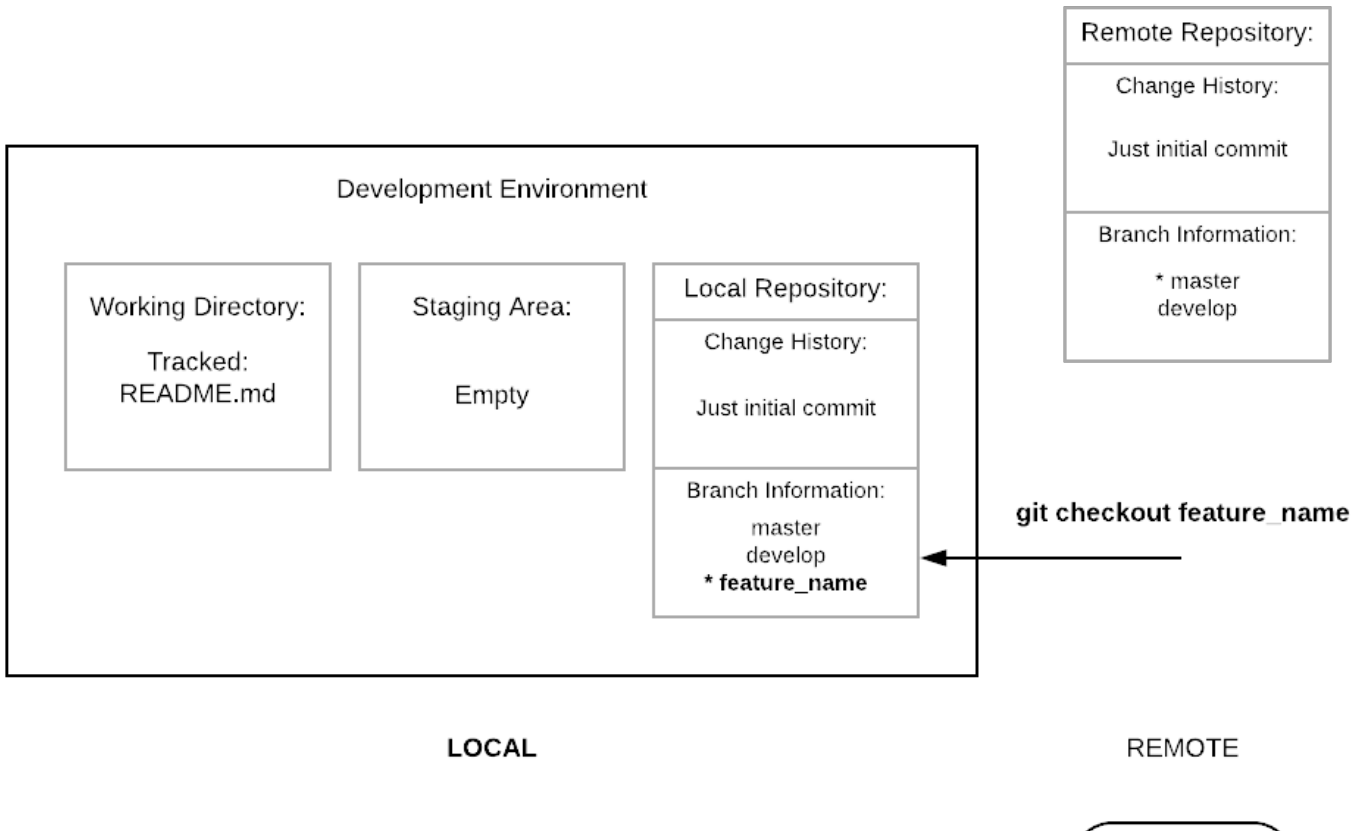


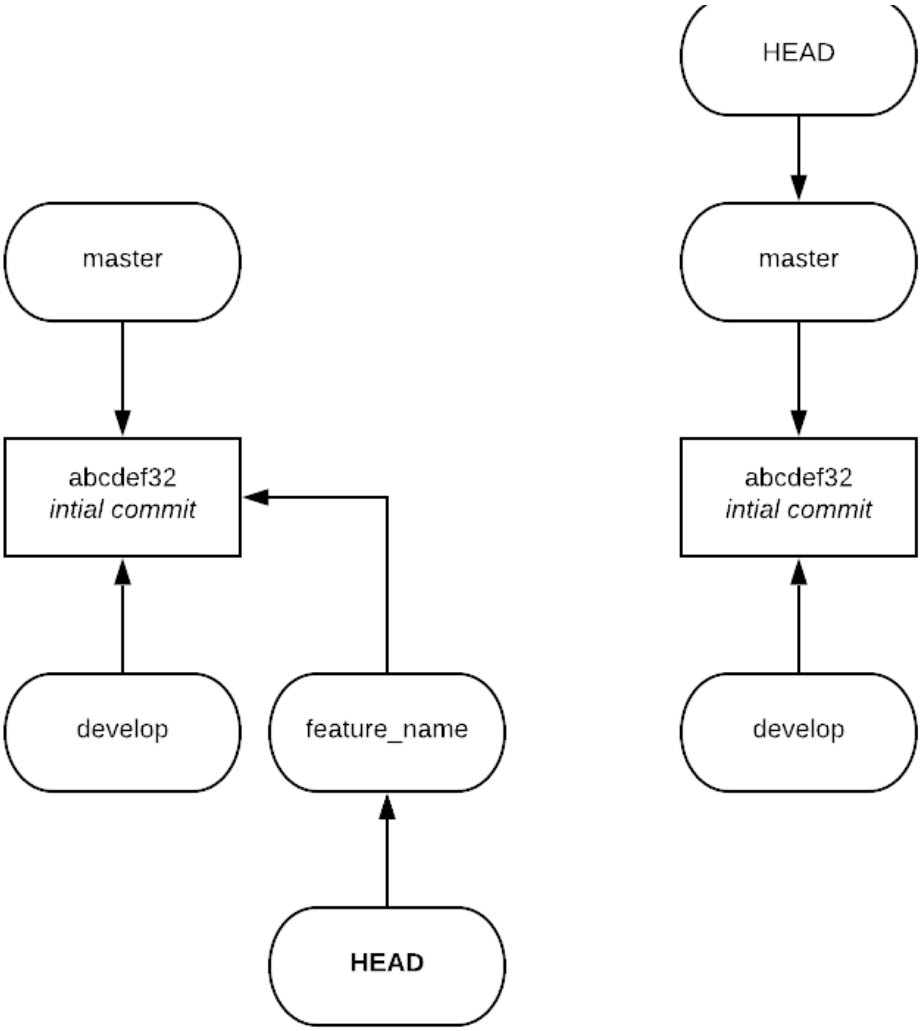
To work on a new feature you should create a new branch named after your feature. This can be done using the command `git branch feature_name`. Notice that you do not switch automatically to this new branch.



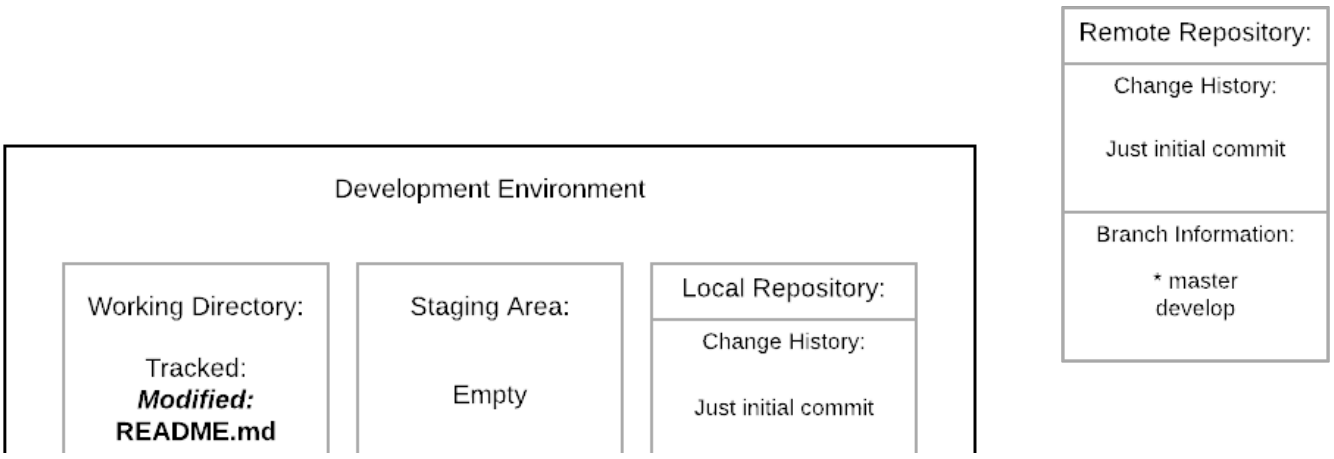


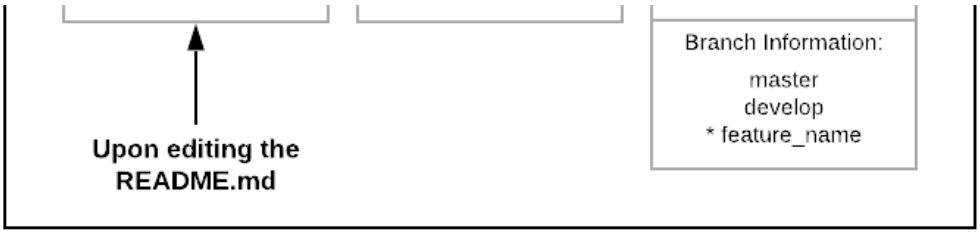
To switch to the new branch you can use the command `git checkout feature_name`. Notice now that HEAD is pointed at the branch new*feature*name. A way to check that this worked is the command `git branch`. At this point you should see a * beside new*feature*name.

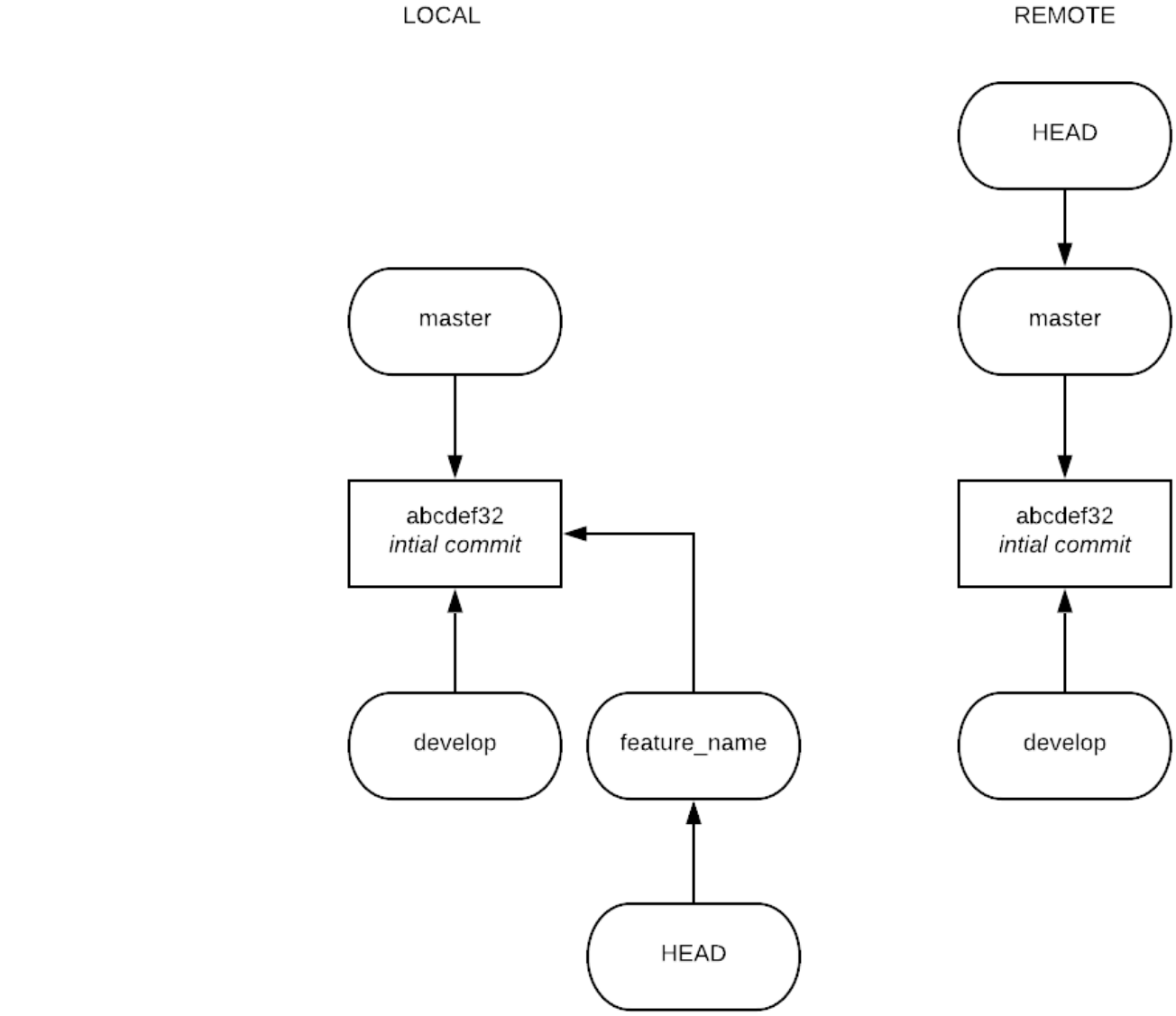




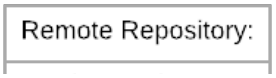
At this point you can either create new files or edit existing files with the confidence of knowing that you are working in the isolation of your own branch. Once you start editing an existing file which is already tracked such as the README.md the status of that file changes to MODIFIED. This can be seen by using the command `git status`.

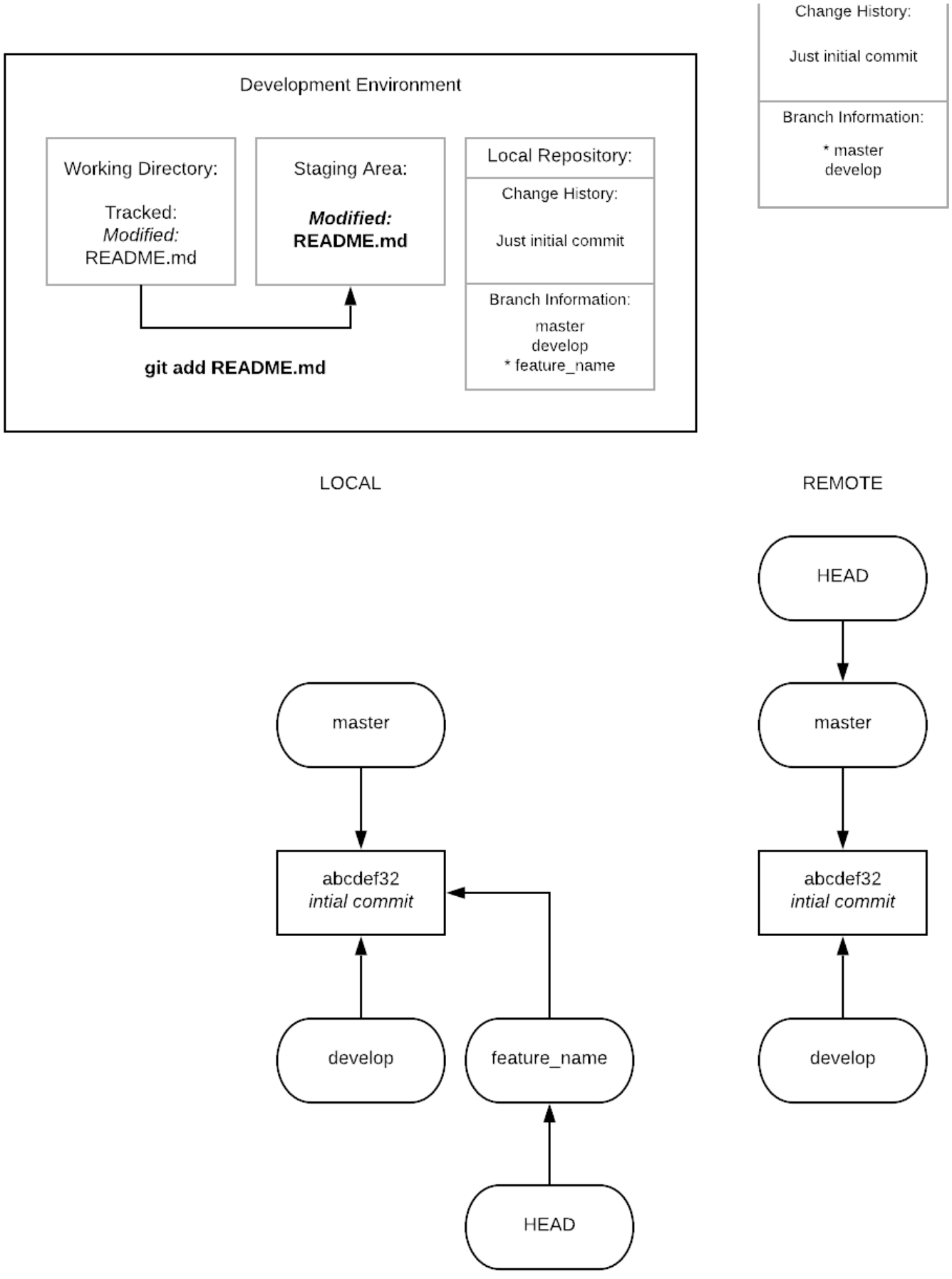






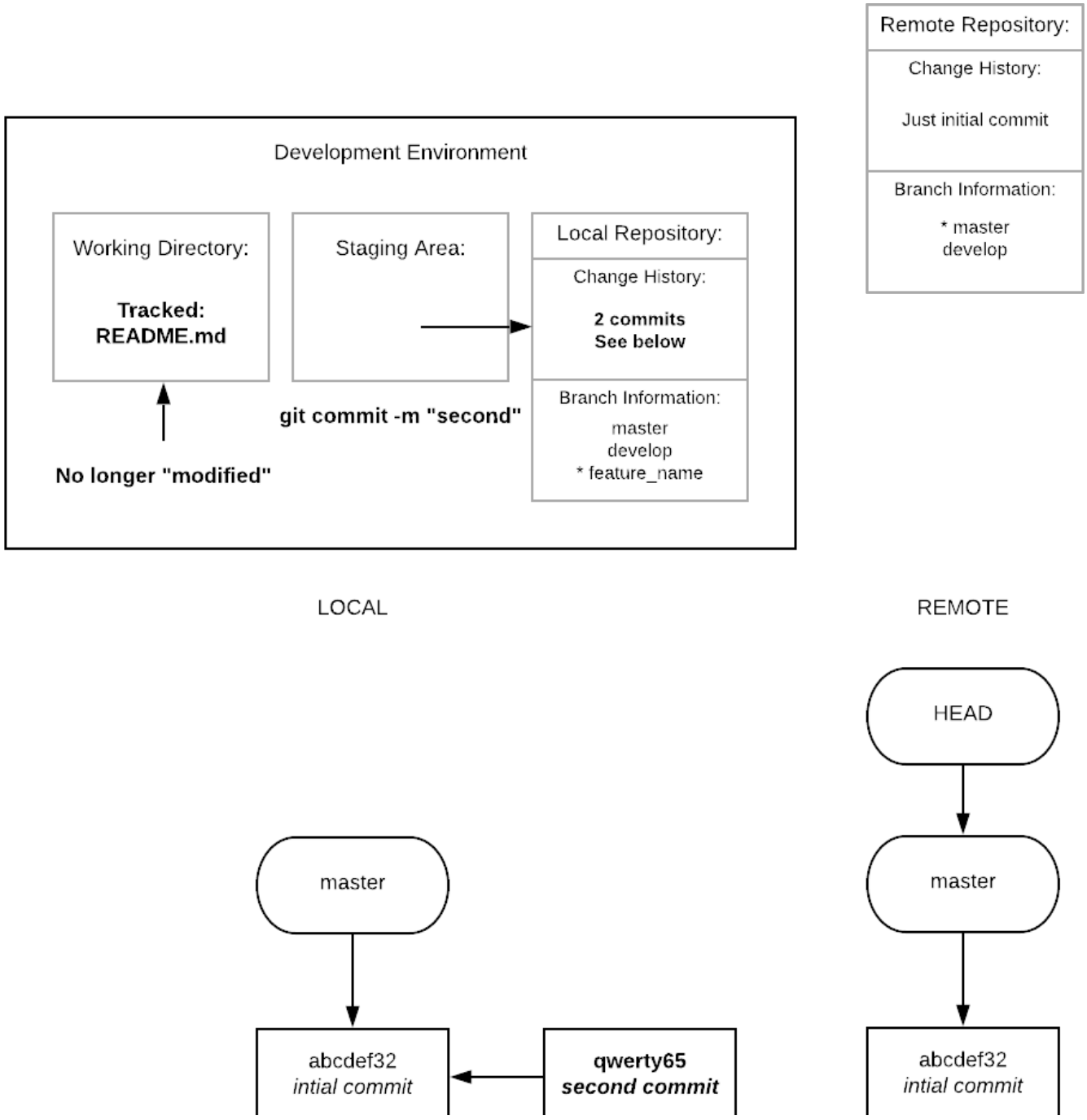
Once a file has been modified it will need to be added to the staging area again in order for it to potentially be committed to the local repository. The command to perform this operation on the README.md file is `git add README.md`. Note that we could also add a newly created file to the staging area as well. For example, `git add nick.txt`.

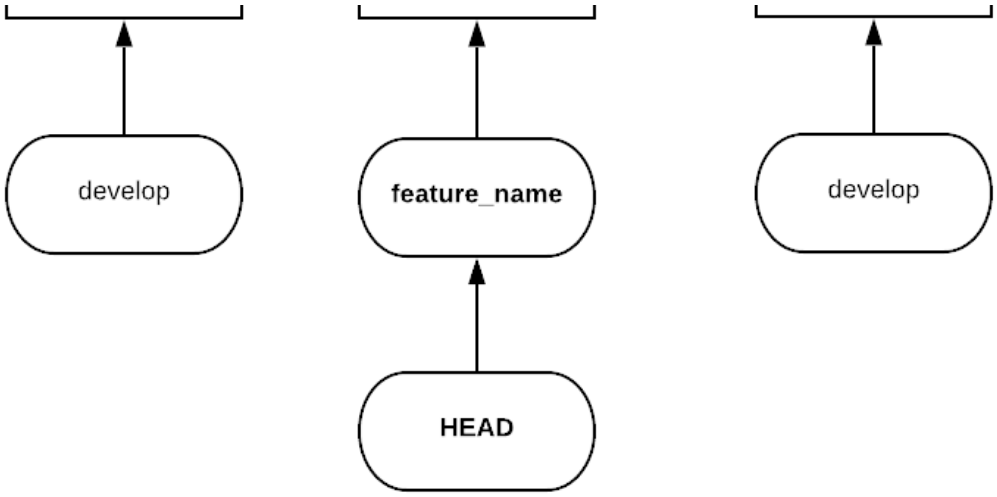



Development Environment
Working Directory:
Tracked:
Modified:
README.md
Staging Area:
Modified:
README.md
Local Repository:
Change History:
Just initial commit
Branch Information:
master
develop
* feature_name
git add README.md
Change History:
Just initial commit
Branch Information:
* master
develop
LOCAL
REMOTE
HEAD
master
abcdef32
intial commit
develop
feature_name
HEAD
master
abcdef32
intial commit
develop

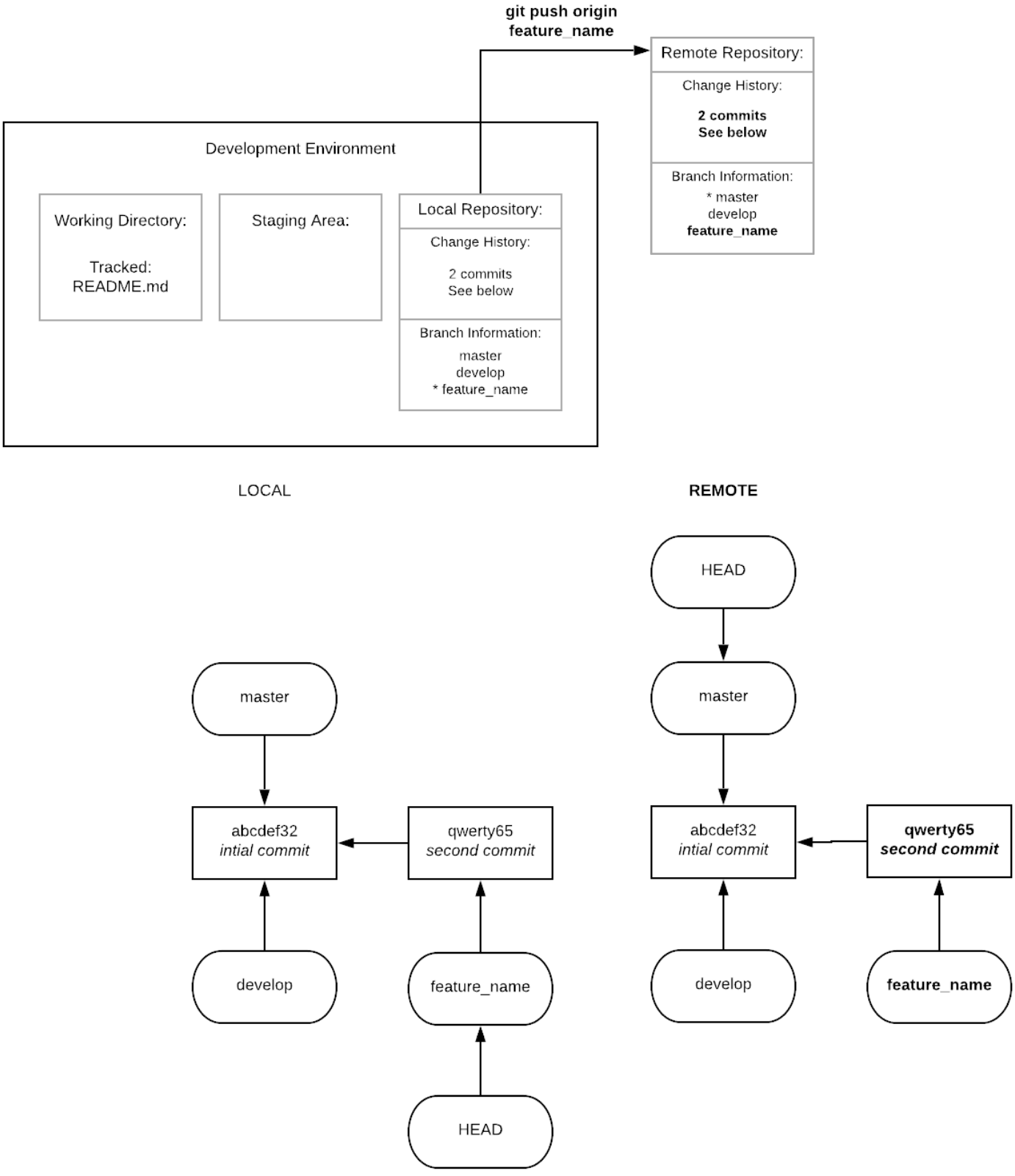
Note that once you have added a file to the staging area you can go straight ahead and commit the file to the local repository. However, you may realize you need to modify the file further before committing. If you modify the file after having added it to the staging area you will need to issue the `git add filename` command again to catch the changes before committing.

At this point you can commit your changes to the local repository using the command `git commit -m "Detailed comment about commit."`. At this point our feature branch will be one commit ahead of both the develop and master branches.





Now that you have committed your changes to your *local repository*, you may want to share these changes with your teammates for review and integration into the product. To do this you will need to *push* your updates to the *remote repository*. However, at this point there is no *feature_name* branch on the remote repository. To push your changes to the remote repository on the branch *feature_name* you can use the command `git push origin feature_name`.



After this command you can see the commit history has changed and the feature branch is one commit ahead of the develop and master. The next step in order to integrate you changes into the develop branch is to merge

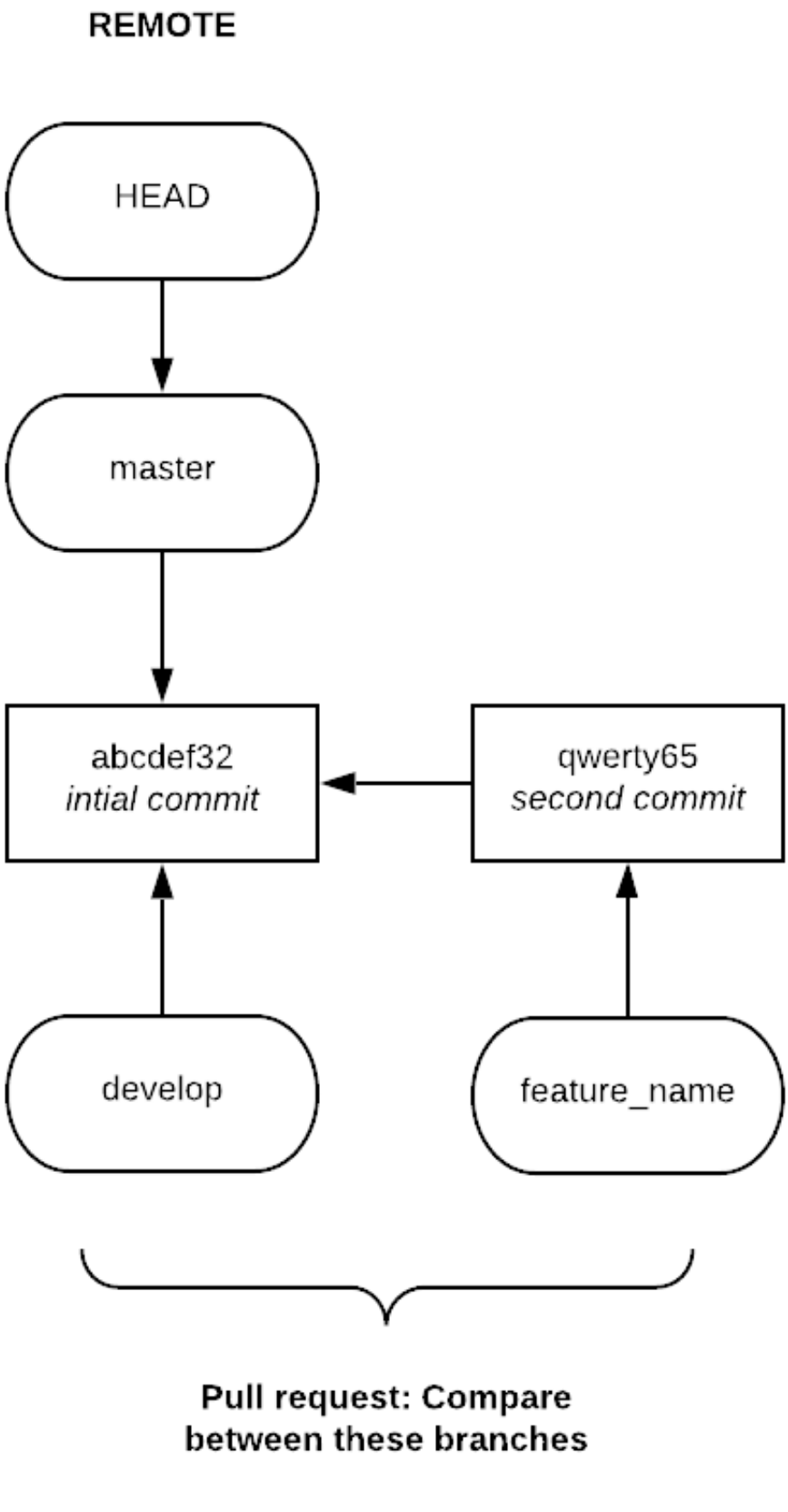
the feature_name into develop.

- What could go wrong?
- Should this code be reviewed by your team first?
- Should the code be tested by another team member first?

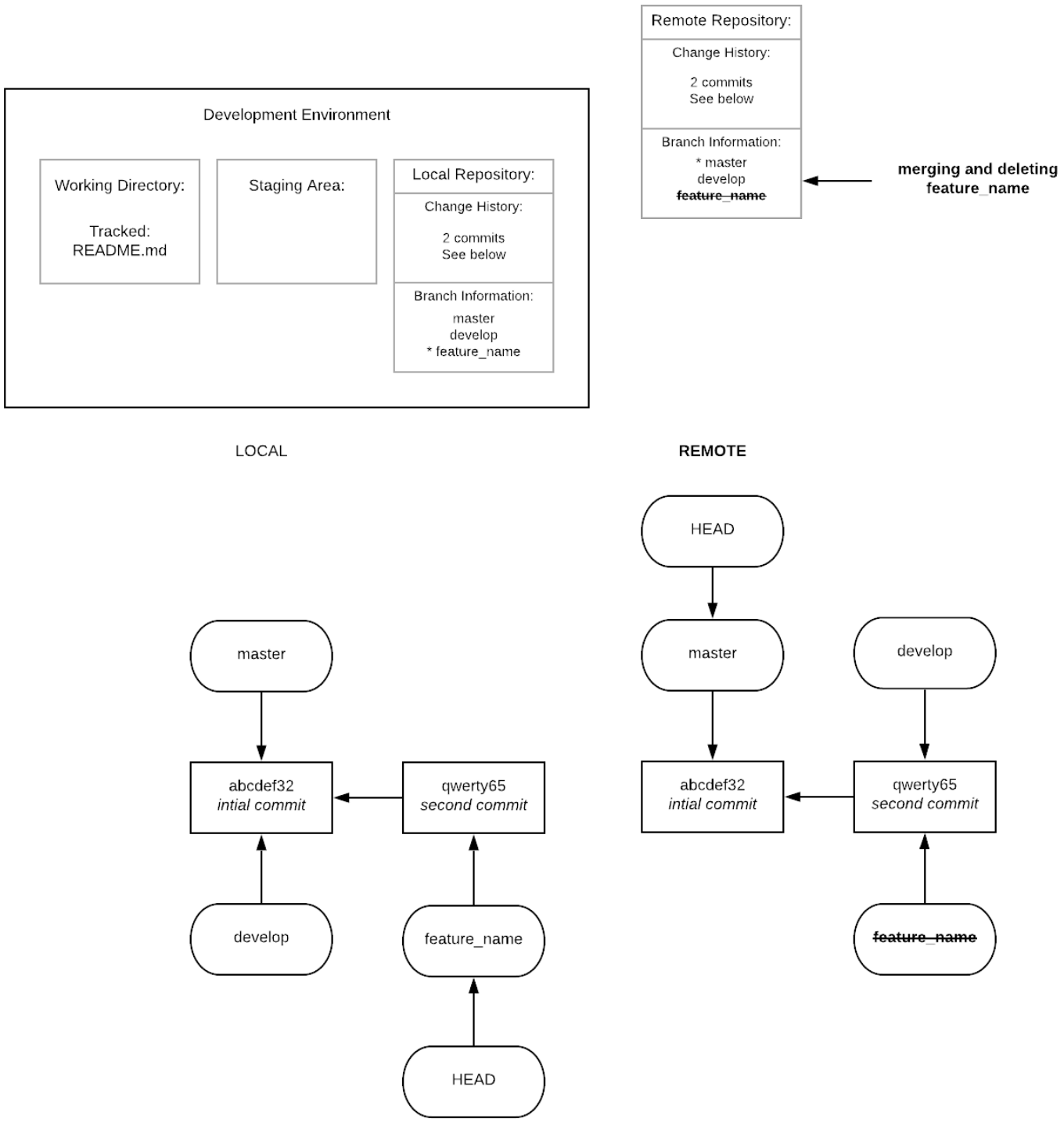
In industry this is where the term called *Definition of Done* comes into play. Different teams will have different level of scrutiny that a code change must pass before being accepted into the shared team branches of develop and master.

For this class we have required you to protect both your master and develop branches which means that you can not push directly to those branches. Instead you will need to have the commit reviewed by another teammate and approved for merge. The mechanism for doing this is generating a pull request which can be done through the GitHub web interface. Once the pull request has been approved then anyone on the team can merge this code into the develop branch.

At the end of each Sprint when the team is preparing for the Sprint Demo you will need to add another pull request to merge the develop branch into master. Be careful with this, and only perform this action once you are 100% certain the code is ready for demo.


REMOTE
HEAD
master
abcdef32
intial commit
qwerty65
second commit
develop
feature_name
Pull request: Compare
between these branches

Once you are sure the the merge was successful, the *feature_name* branch is no longer needed and can be deleted through the GitHub web interface. Notice that this does not delete said branch on local repositories.



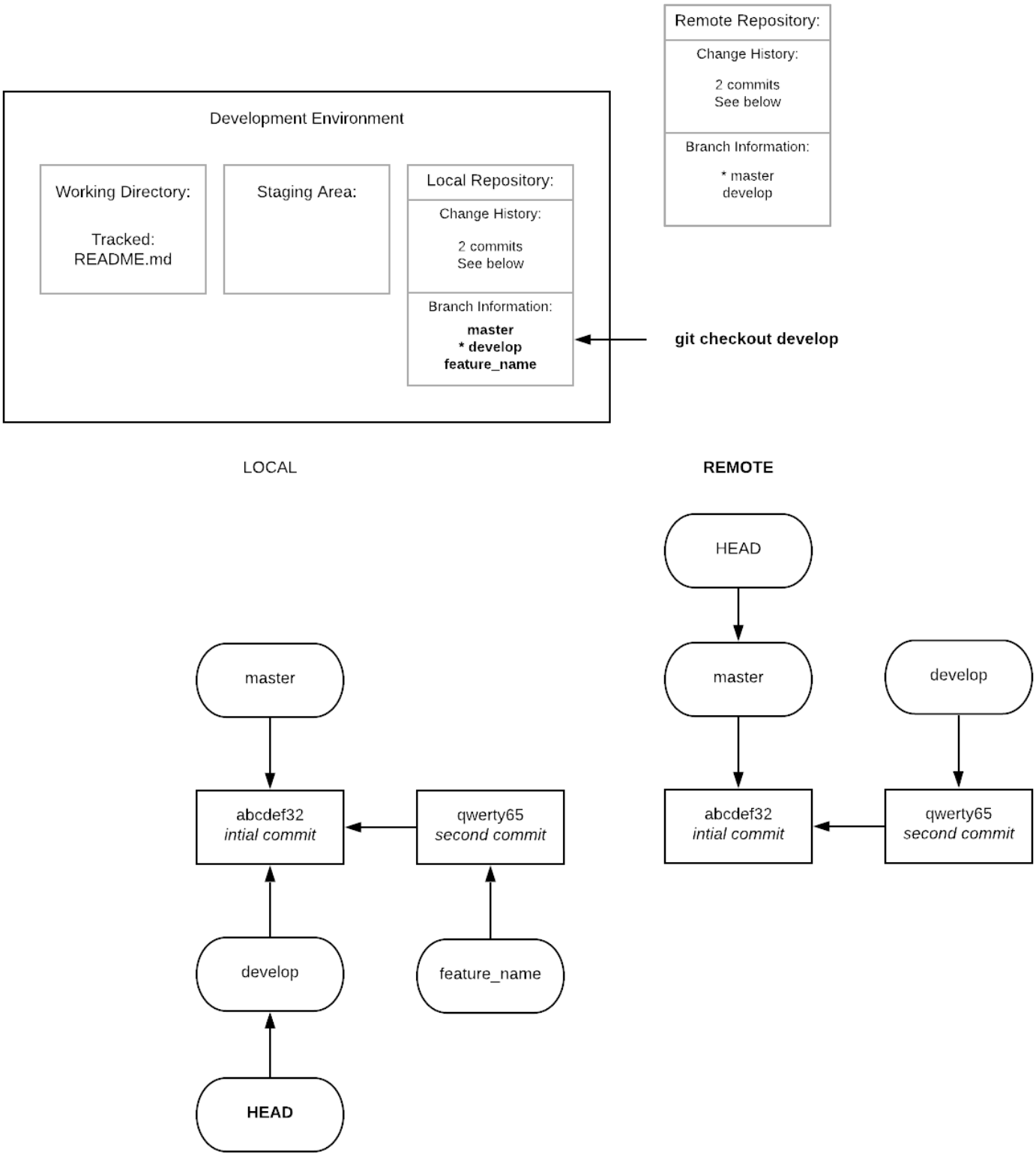
Now you are ready to move on to your next feature. What should you do? Start by asking yourself the following questions:

- Which branch are on in your local repository? (Another way to ask this is where is HEAD pointing in your local repo?)
- Do you have the latest version of the code on your local machine?
- Did I correctly switch to the new feature branch before starting to work?

The command `git status` can help you answer all of those questions.

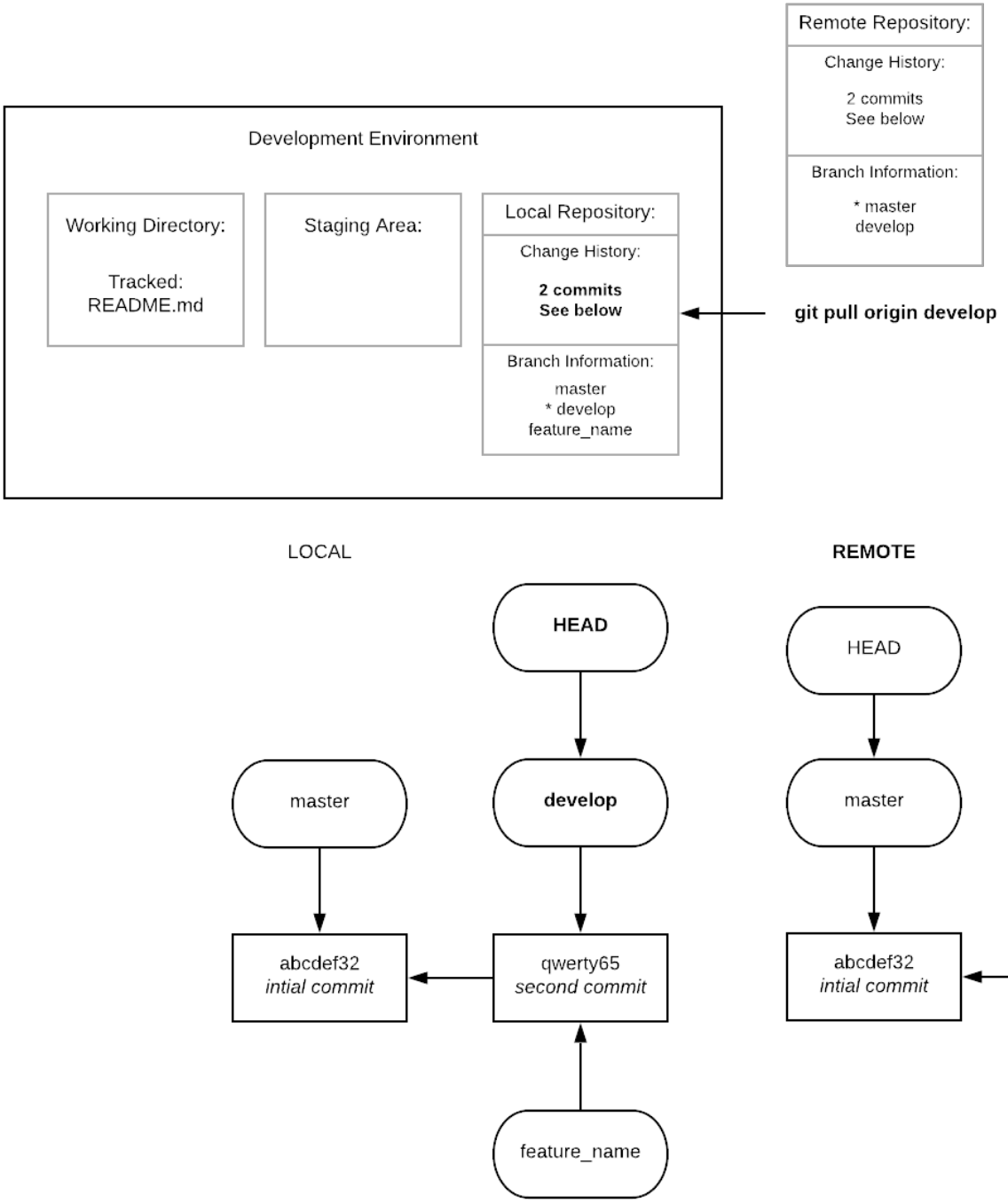
At this point you are probably still on the branch *feature_name* on your local repository. You will need to switch to the *develop* branch.

```
git checkout develop
```



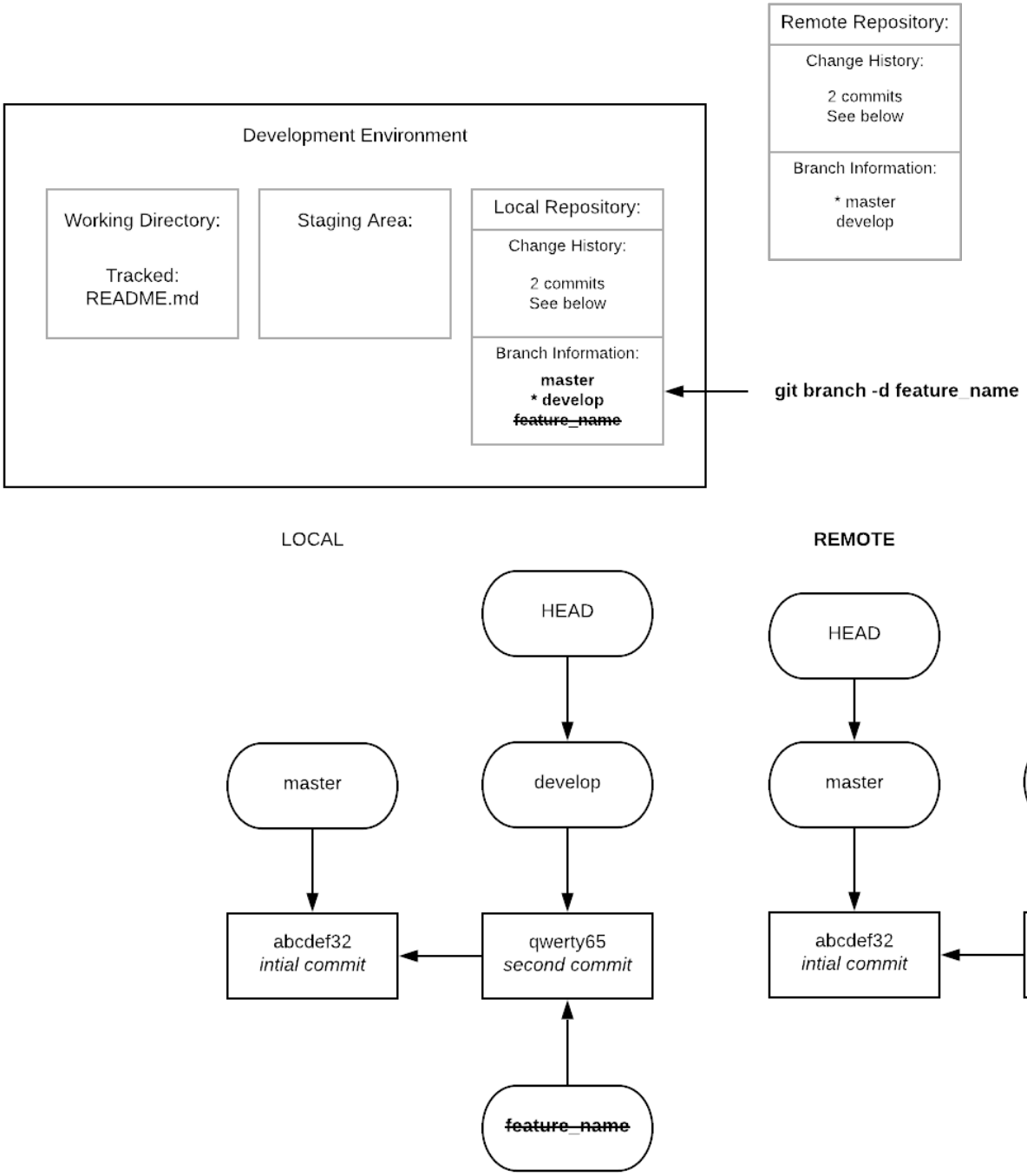
Then you will need to *pull* to get your local *develop* branch to match what is on the remote repository.

```
git pull origin develop
```



At this point it is probably a good idea to go ahead and delete your local *feature_name* branch that you used for your previous feature. To do this use the command:

```
git branch -d new_feature_name
```



Now you are ready to loop back through this set of instructions again for the new feature starting at the command `git branch feature_name` where now *feature_name* should be the name of the second feature you are going to work on.

But wait. What branch are you on? Thats right you are on *develop*. Don't forget to checkout the feature branch.

Summary

In this lecture we have discussed the source control management and why it is important. We introduced the key entities of Git which are :

- Working Directory
- Staging Area
- Local Repository
- Remote Repository

Additionally we covered how the commands `pull`, `add`, `commit`, and `push` alter each of these.

We have also gone into detail on common workflow that our team should use for interacting with git as well as a discussion of the best practices of using git..

In the next reading we will go into detail on the User Interface and User Experience design process and best practices for developing applications that look good and are easy to use.