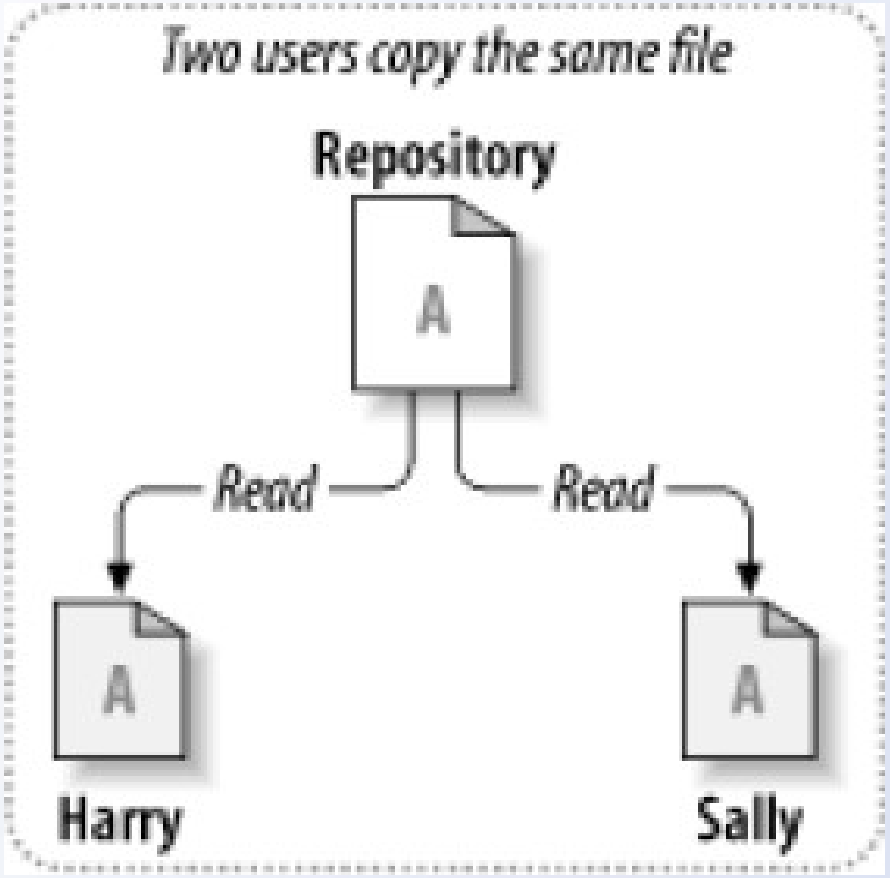
GIT

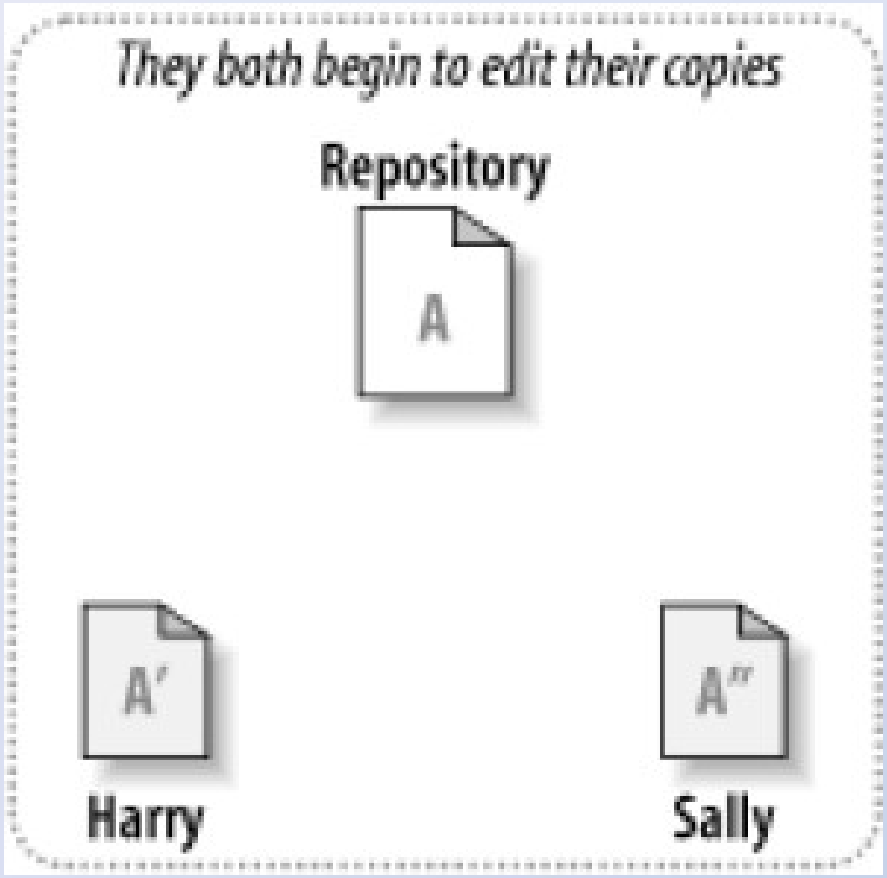
Problem

How will the system allow file sharing, and also prevent users from conflicts with each other ?

What about locks ?

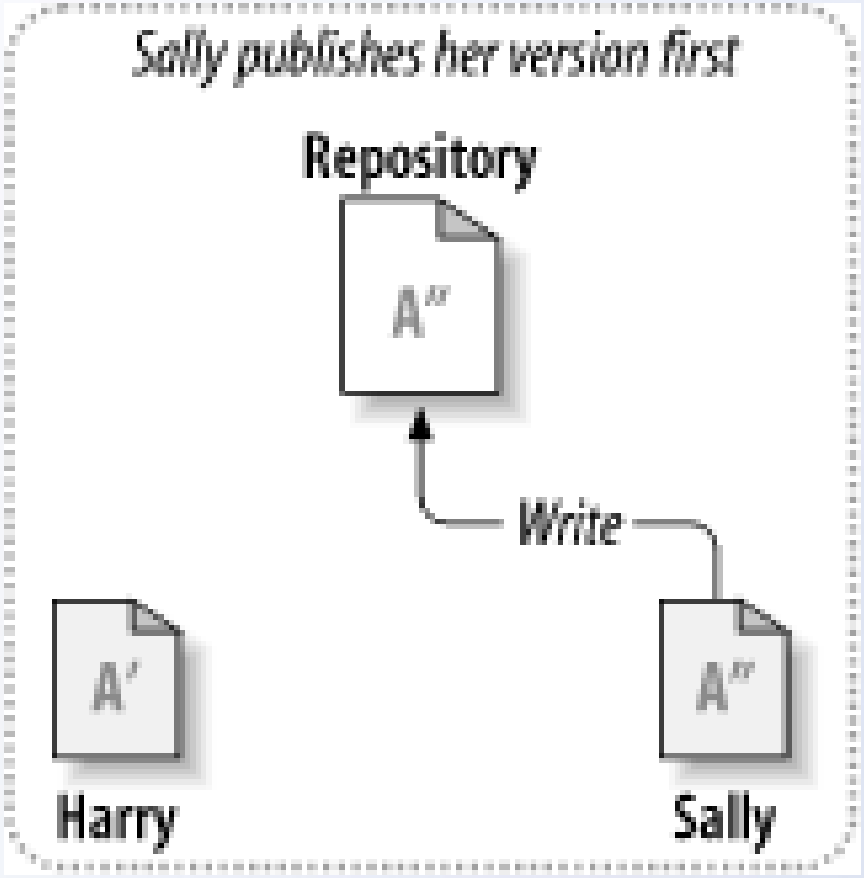
Copy-Modify-Merge Solution

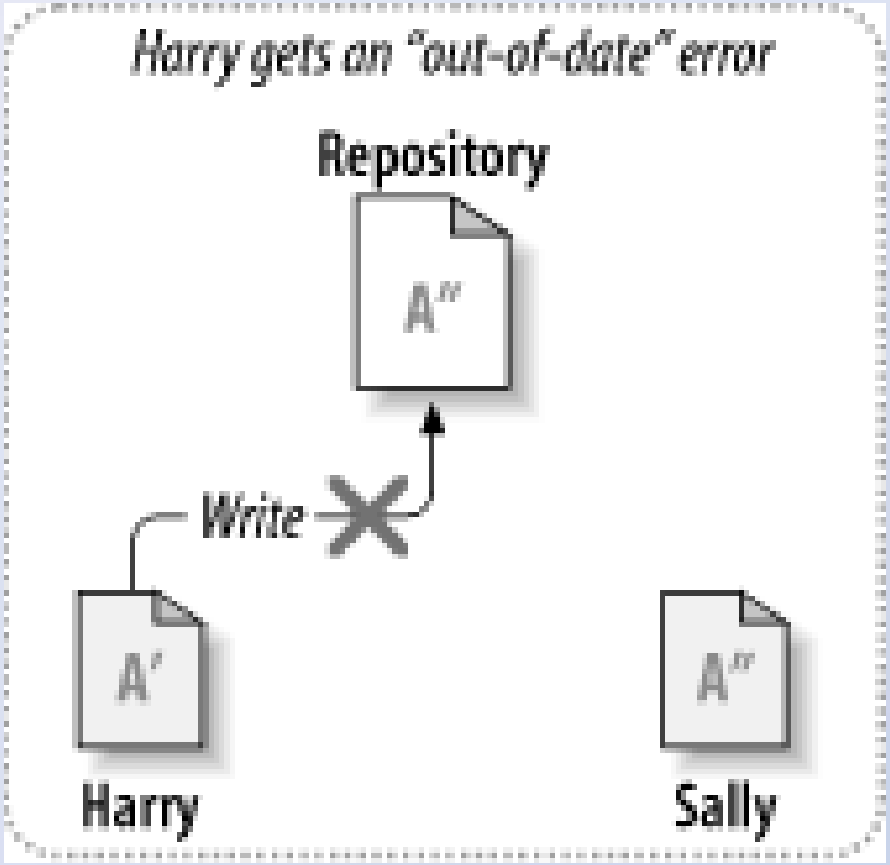




IT TALENTS
Training Camp

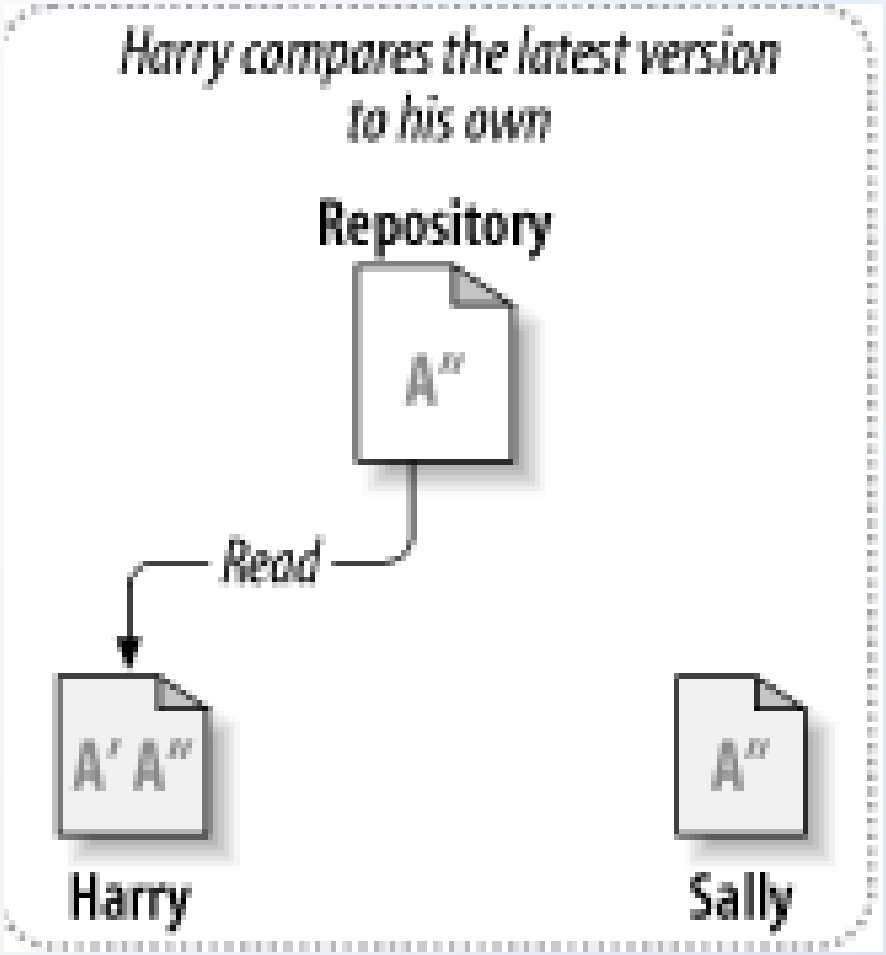
Copy-Modify-Merge Solution

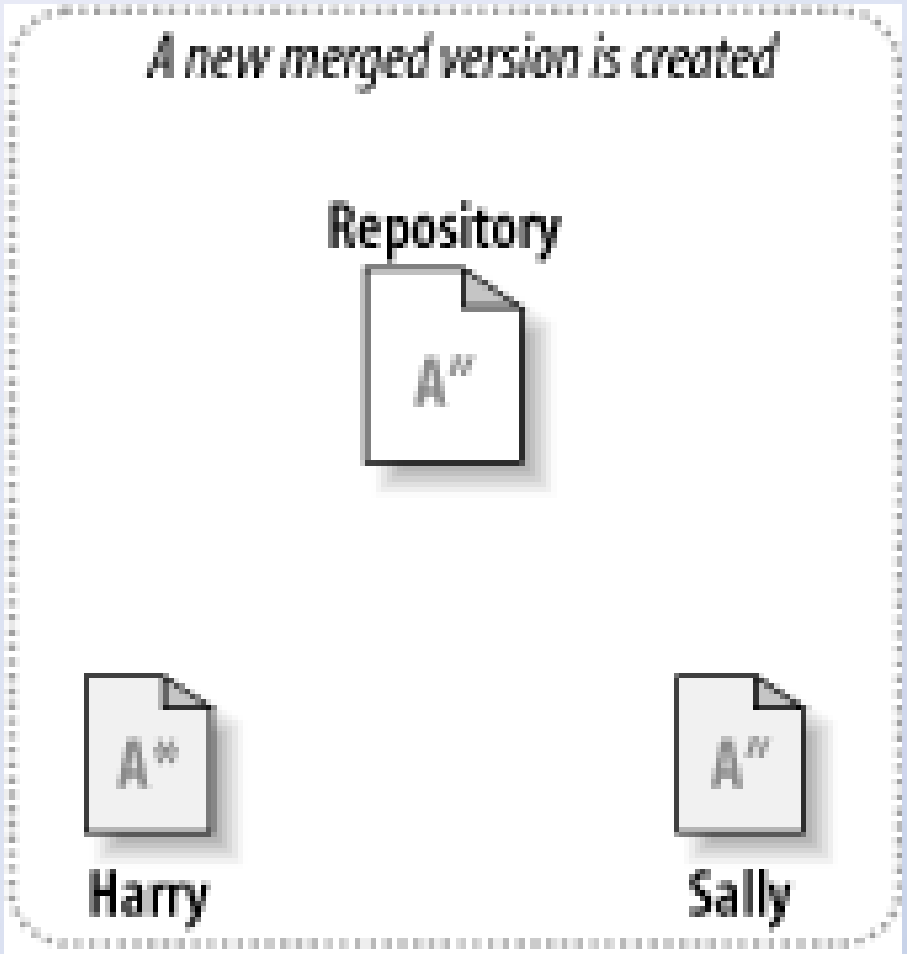




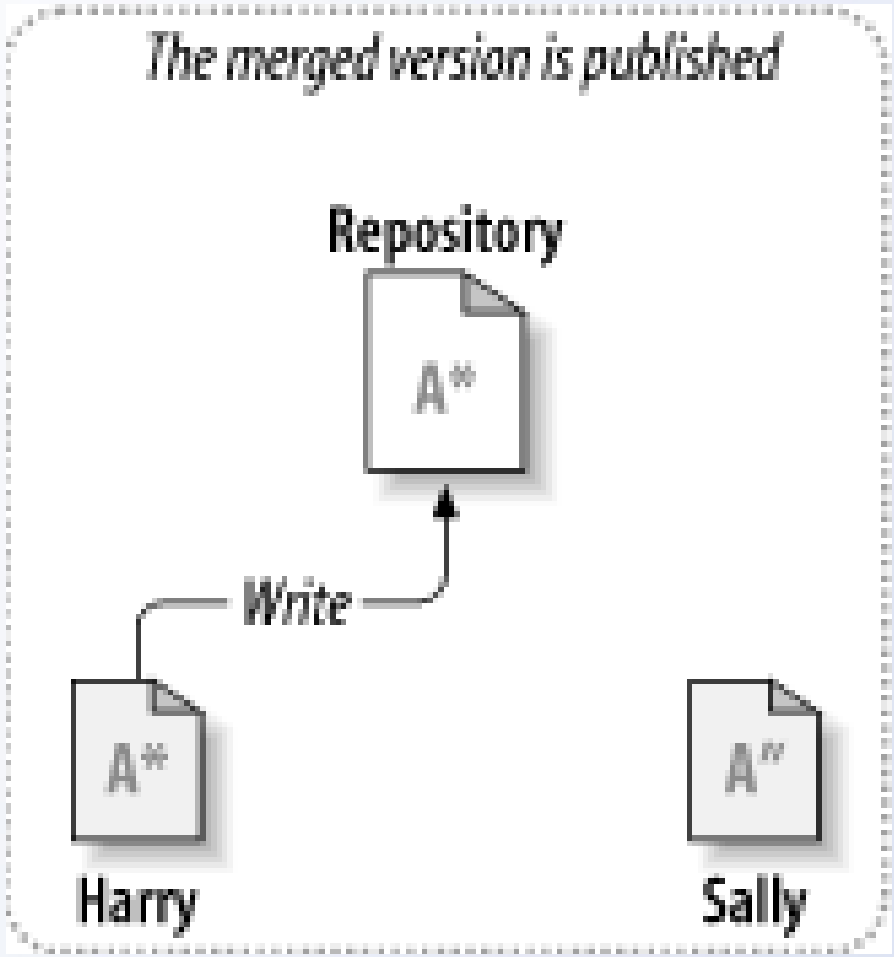
IT TALENTS
Training Camp

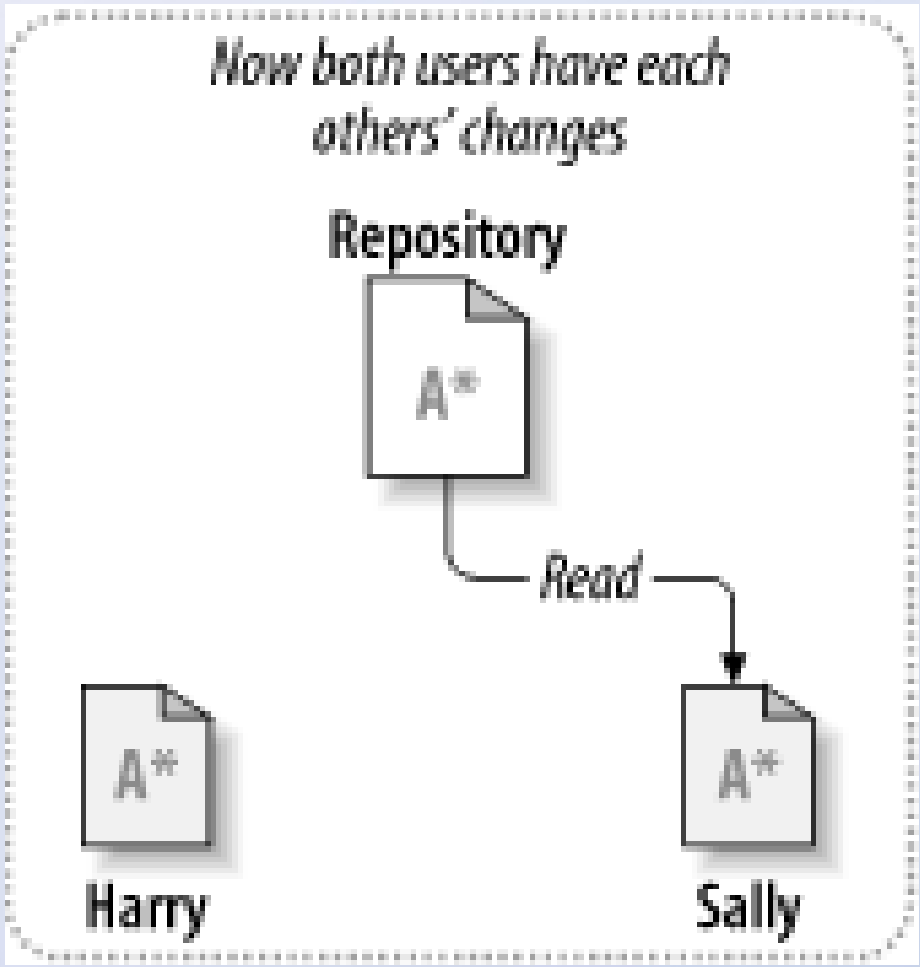
Copy-Modify-Merge Solution





Copy-Modify-Merge Solution



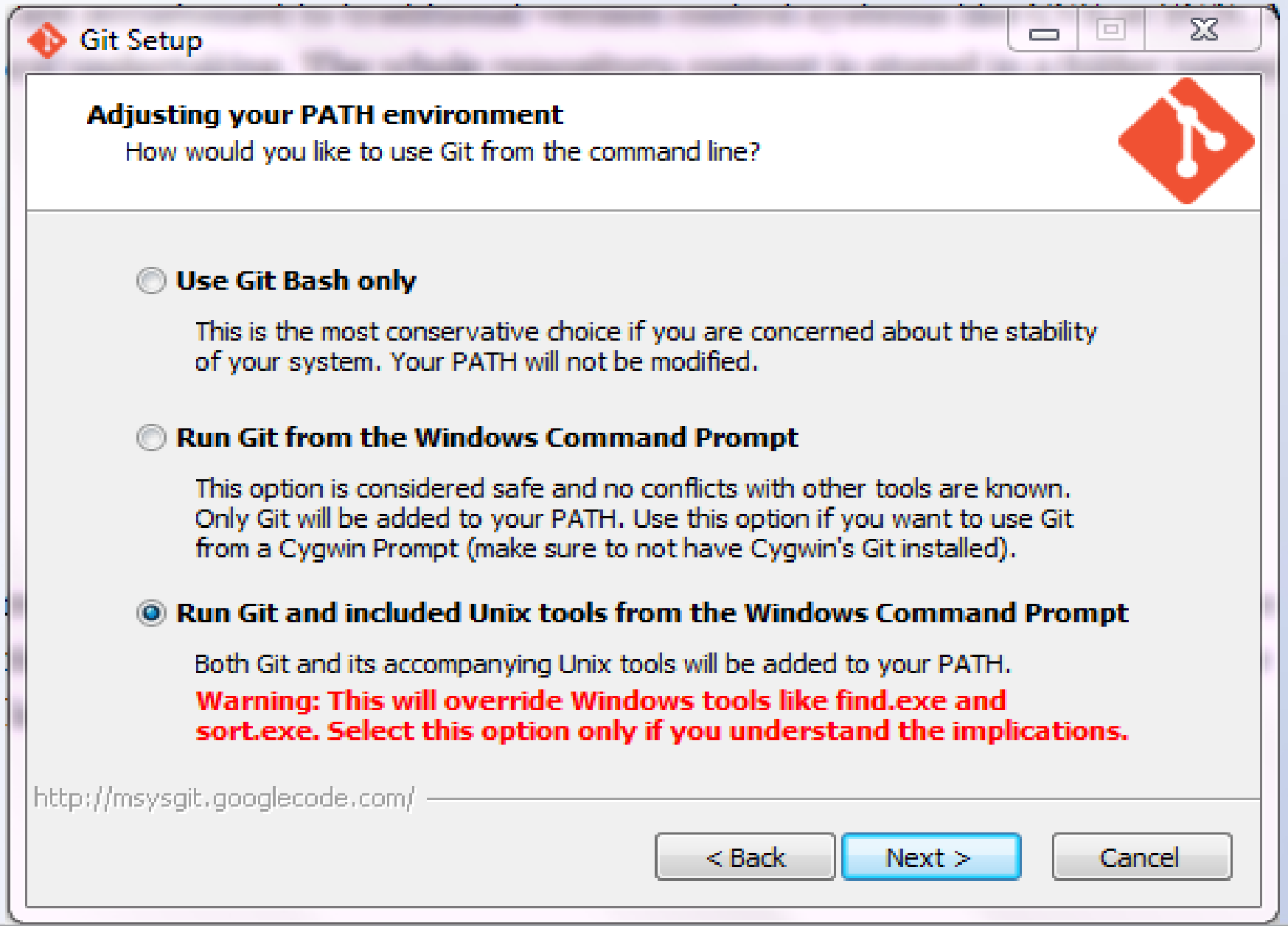


IT TALENTS
Training Camp

Installation

- Windows
  - http://git-scm.com/download/win
- Linux
  - apt-get install git

Installation



Distributed Version Control System

This kind of VC system does not necessarily has a central server that stores data.

You copy an existing repository from somewhere. That's called **cloning.**

Typically there is central repository and some copies of it.

They act exactly like the main(remote) repository

What is GIT

GIT is distributed version control system

It originates from the Linux kernel and is used by many open-source projects and commercial organizations.

The core runs on C and it very fast and error-prone.

So How Can I Create One ?

You may create your own repo in your machine!

Just make a directory and execute **git init.**

That's it ! You have your own repository with version control, conflict resolving, branching and tagging abilities, etc., which you may **share** with others !

Who is here ?

Next, configure who is working on this repository :

```
git config --global user.email "ivo.ivov@ivomail.bg"
git config --global user.name"ivo"
```

Configures for all git repositories
on this machine.
Remove –*global* to setup these only
for this project.

The staging index

Git internally holds a thing called the **index**, which is a snapshot of your project files.

After you have just created an empty repository, the index will be empty.

You must manually stage the files from your working tree to the index using **git add**:

```
git add test.txt
```

The staging index

git add works recursively, so you can add whole folders as well.

Important : The same applies if you change a file in your working tree - you have to add this change to the **index** with git add !

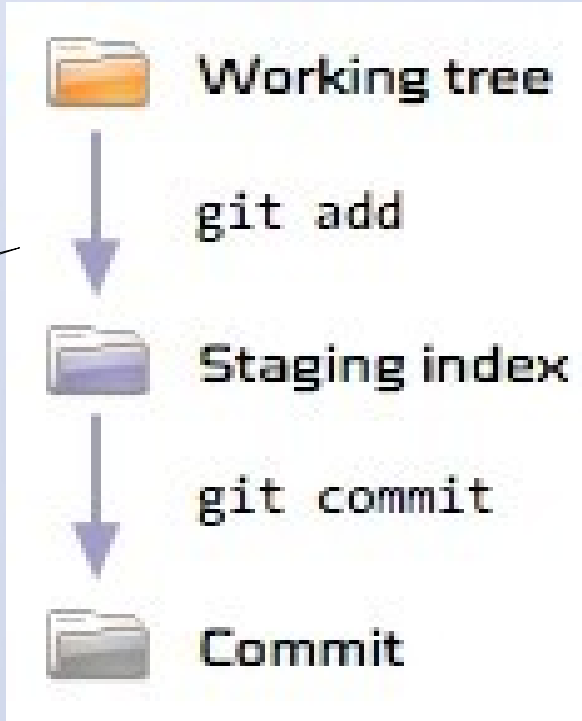
edit test.txt
git add test.txt

Commiting

git commit takes the contents of the **index** and creates a new commit:

git commit -m "I commit in Git !"

Don't forget to add your changes to the index!



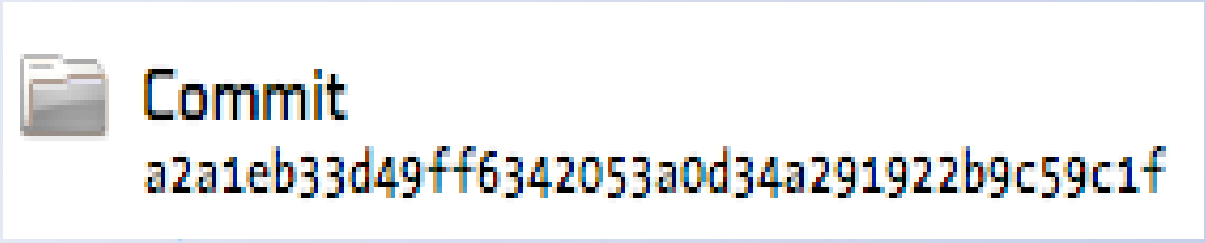
IT TALENTS Training Camp

Commiting

Similiar to the index a commit is a full snapshot of your project files.

But they are not numbered !

Instead, a commit gets assigned a SHA-1 hash of the snapshot contents:



Commit History

The workflow for editing files in a git repository looks like this:

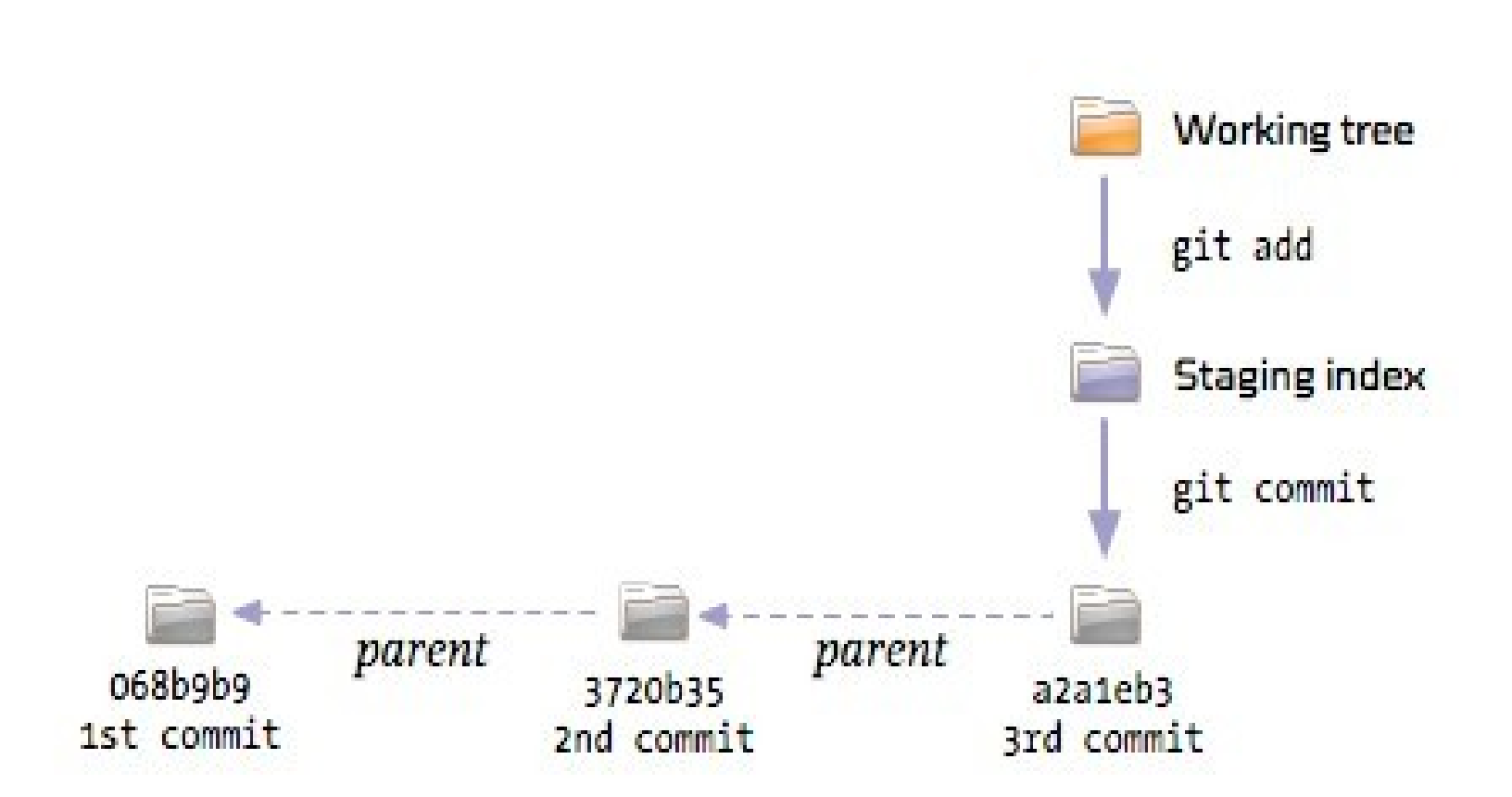
You make changes to the working tree files.

You add these changes to the index using git add.

You create a new commit from the index using git commit.

Commit History

As you do this repeatedly, you will create a new commit each time, pointing back to the previous commit:



Commit History

This is how git keeps track of the project history.

It stores snapshots of the project files as commits. These commits point back to the commit they were created from.

All these snapshots are saved in a very efficient manner.

You can see the history using **git log**.

DEMO: adding, commiting and showing history.

What will I commit ?

git status shows you how the working tree is different from the index and how the index is different from the last commit.

git status

```
# Changed but not updated:
#
#   modified:   my_file.txt
#
# Untracked files:
#
#   my_new_file.txt
```

Changed and added to the index.

Changed, but not added to the index.

What will I commit ?

Well, adding every time before committing may be a little overhead.

If you're sure what you commit you may use :

```
git commit -a -m "commit message"
```

This will add all changed (but not new!) files to the index before committing.

Throwing changes away

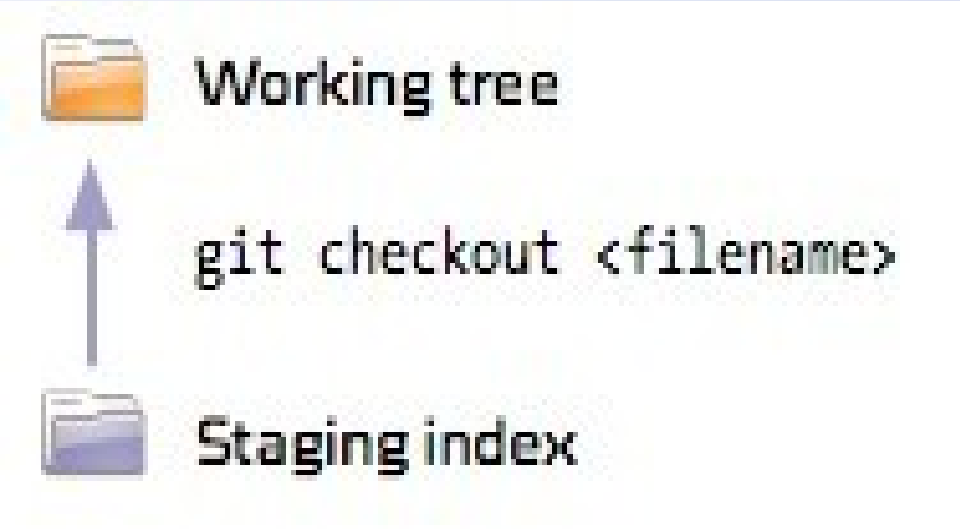
Shit happens.
What if you want to return to the original state?

This depends on where the changes are...

If you have not added them to the **index** yet, you can restore them from the index using :
git checkout <filename>

Throwing changes away

This restores a file or a folder as it is stored in the index to your *working tree*:



IT TALENTS
Training Camp

Throwing changes away

What if you have already added the changes to the index?

You can restore the index to the last commit using *git reset*:

```
git reset HEAD myfile.txt
```

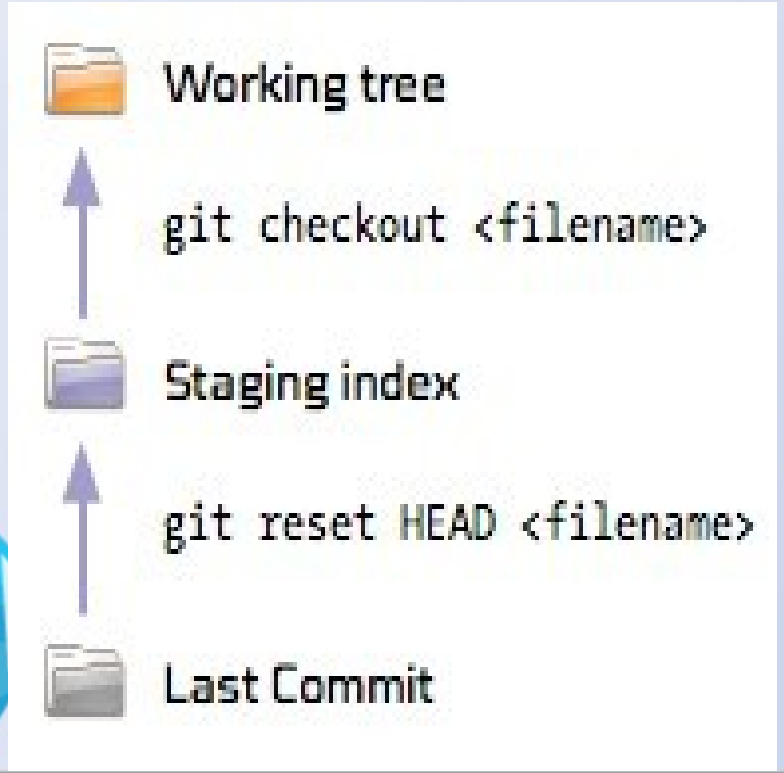
You can also restore the whole index:

```
git reset HEAD
```

Throwing changes away

HEAD always refers to the last commit which has been made.

Using this, the index is restored to the contents of the last commit - you can then use *git checkout* to restore your working tree as well.



Throwing changes away

If you have already committed your changes, you can make use of “undo” command called *git revert <commit>:*

git revert 068b9b9

This will create a second commit which undoes the changes of the given commit.

Seeing the diff between commits

To see the difference from one commit compared to its parent, use *git show <commit>*:

git show abb9180a

To compare two specific commits, use
git diff <commit_from>..<commit_to>:

git diff y5g6eb9..24arphy

To see the diffs for the complete history, use *git log -p.*

Tagging commits

These parts of hash code(a7hda8,m3203a7d,3s85md8,etc.) are useful but are ugly.

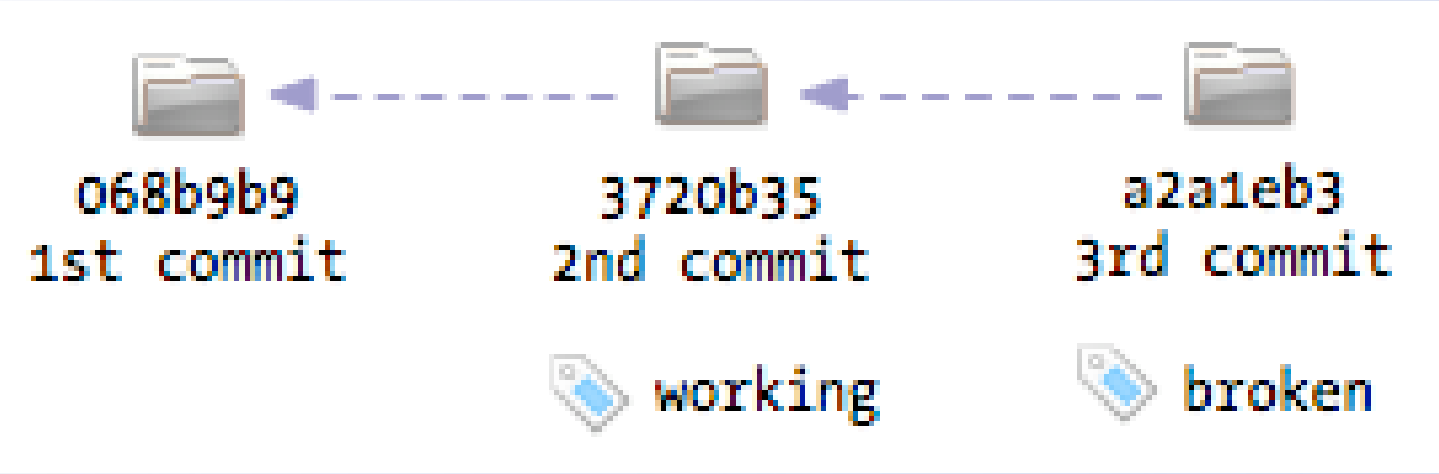
You may assign a **tag** to a specific commit.

git tag <*name*> <*commit*> assigns a tag to a commit.
If <commit> is omitted, the last commit gets tagged:

git tag beta-release 1729b3h
git tag release

Tagging commits

A tag is nothing more than a label that can be used to refer to the tagged commit:



Tags can be used everywhere where you can use the commit hash, for example in *git diff*:

```
git diff working..broken
```

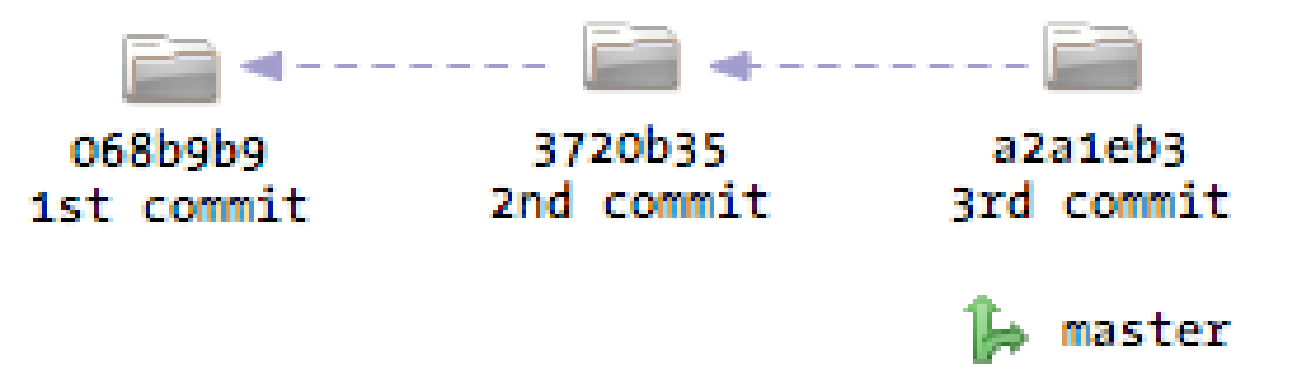
IT TALENTS
Training Camp

Branching

All the time, we were already working in a branch called *master*.This branch was created automatically when we created the repository.

You can see all branches in the repository using git branch:

```
git branch
* master
```



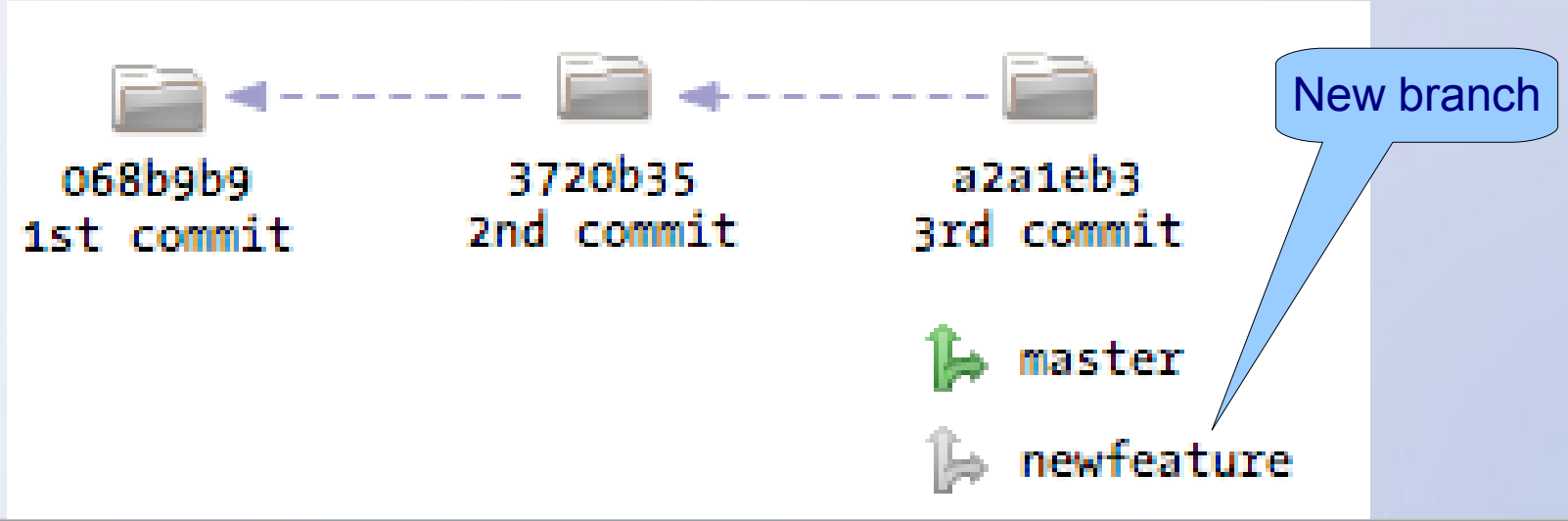
At all times, a branch points to some commit, in our case to the latest commit we made.

IT ALENTS
Training Camp

Branching

To track changes in a separate branch, we have to create a new one using *git branch <name> <commit>*.
The specified commit will be the starting point for the new branch - if you omit it, the latest commit will be used:
git branch newfeature

This will create a new branch called newfeature based on the latest commit in *master branch*:



Branching

At all times, there is one specific active branch.

If you call *git branch*, you will see that a new branch named *newfeature* was created, but master is still the active one:

```
git branch

* master
  newfeature
```

Branching

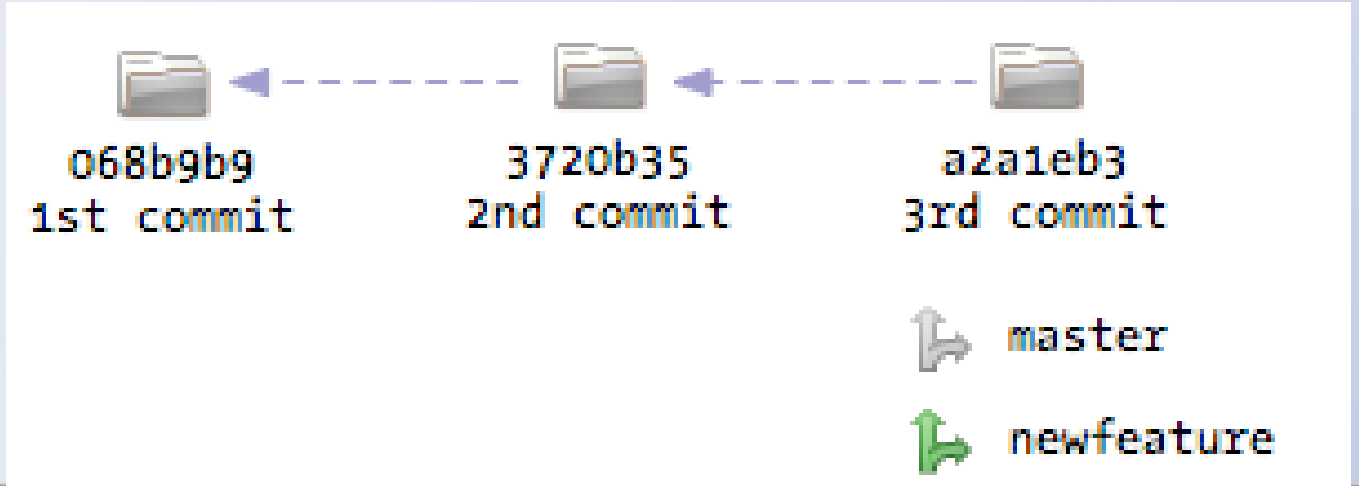
You can switch between branches using *git checkout <branchname>.*
This is the same command we used to get files from the staging index to the working tree.

git checkout newfeature

Now the active branch in our repository is *newfeature*:

git branch

```
  master
* newfeature
```

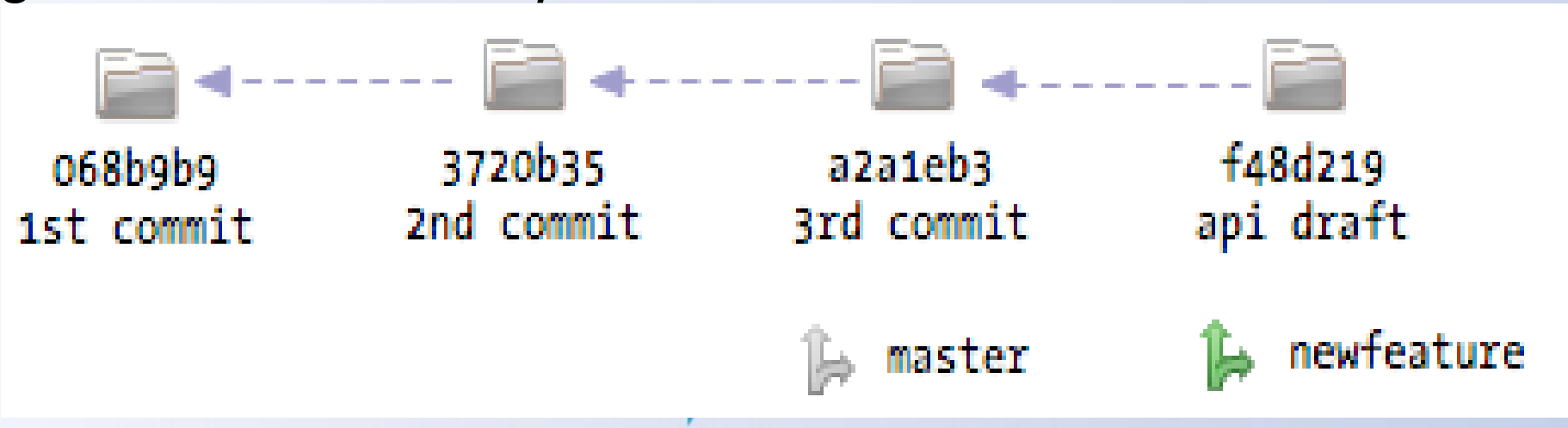


Branching

We can start working in this branch using commands git add, git commit, etc.

Let’s see what happens when we create a new commit in the branch:

edit somefile.txt
git commit -a -m "api draft"



Branching

Let's say we are finished with working on our new feature for the moment and want to continue working on the *master* branch.

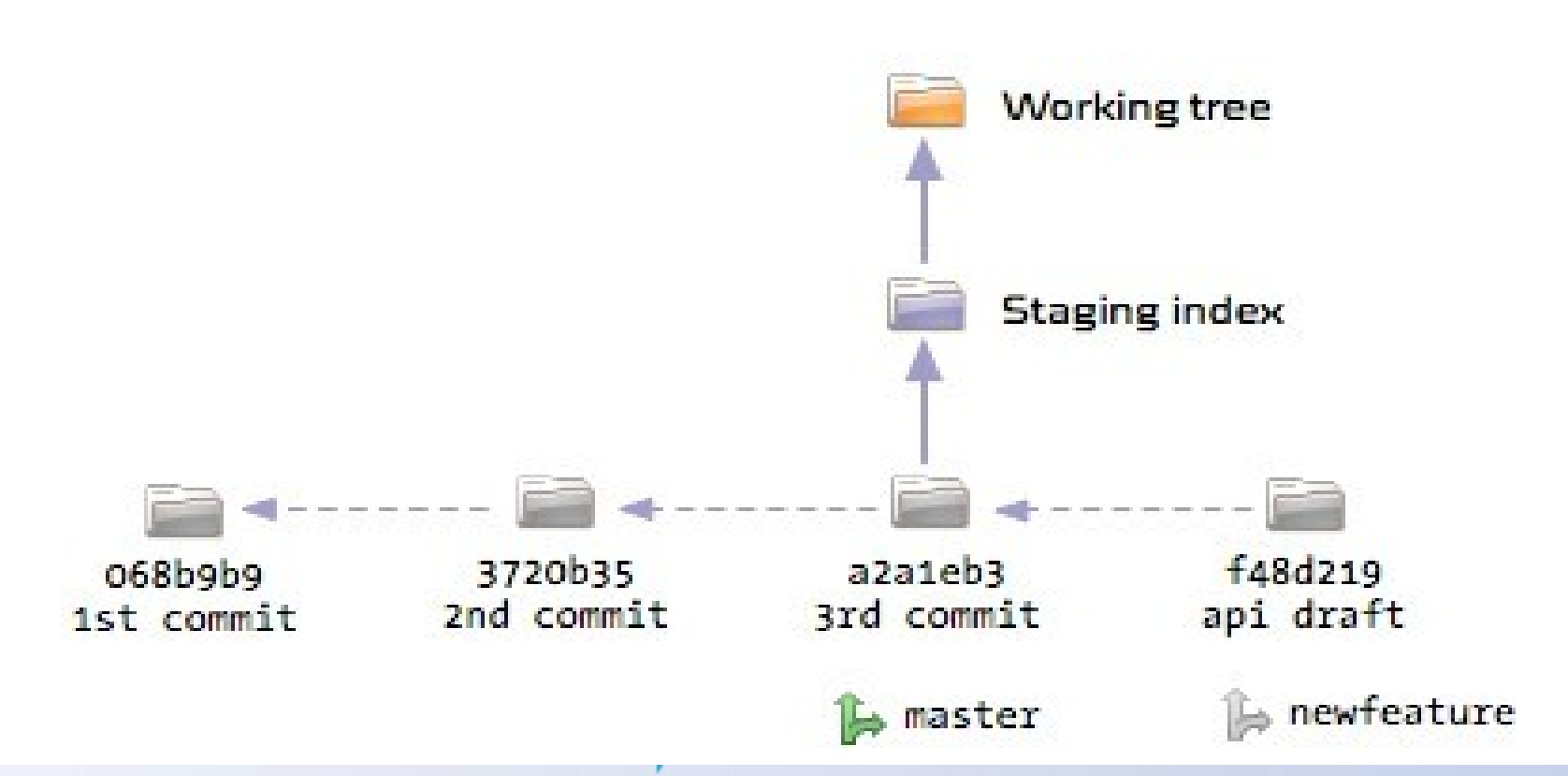
That is easy - we just switch back to the master branch:

```
git checkout master
```

Branching

This will set the active branch back to *master*.

It will also reset the index and your working tree to the contents of the last commit in master:



Branching

So you will see the project in the exact same state it was in when we forked off the new branch.

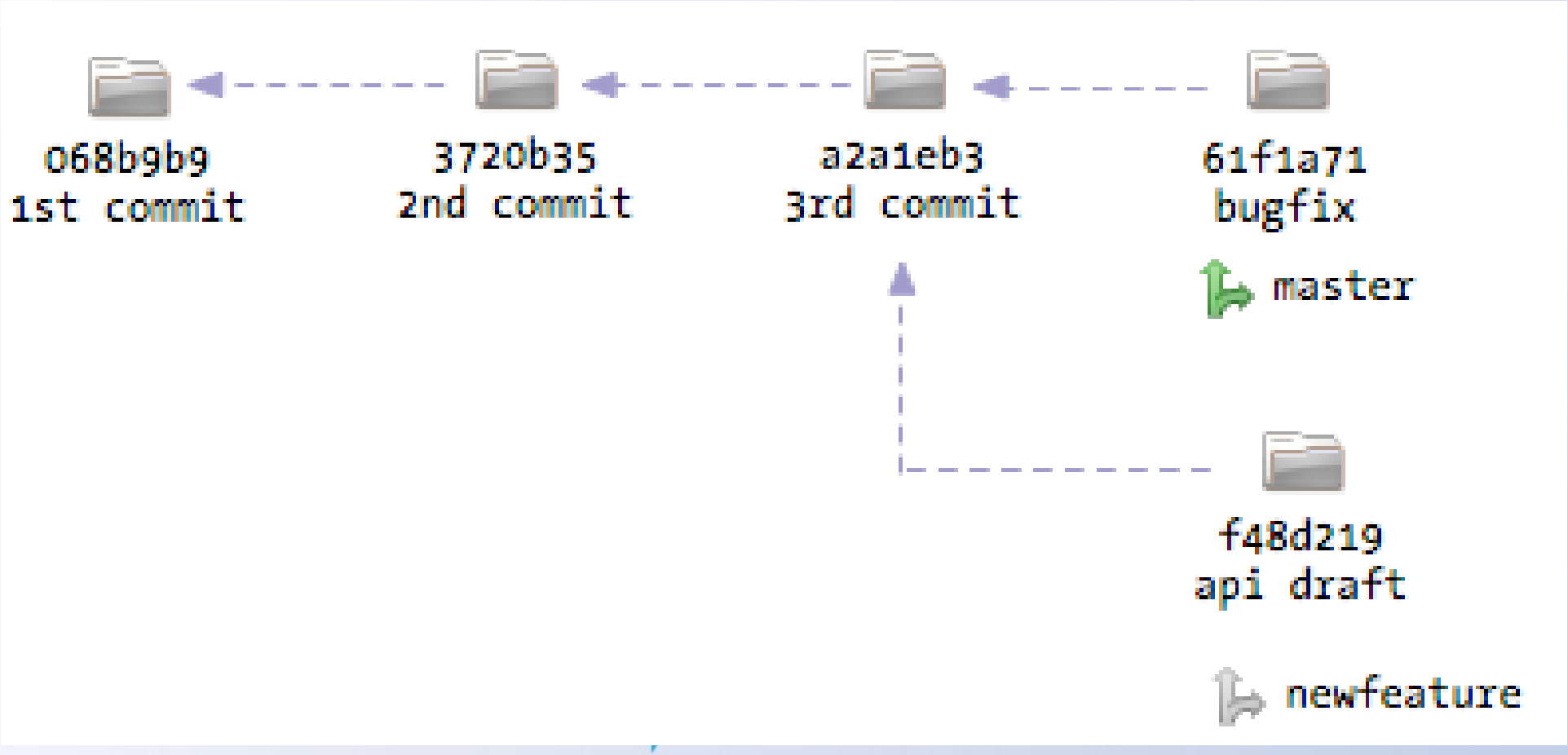
There will be no trace of the changes of our *newfeature* branch.

What happens if we add some changes and commit these?

edit UIElement.java
git commit -a -m "emergent bugfix"

Branching

The same as before: A new commit will be created based on the latest commit in *master* and *master* will be pointing to the new commit after that :



Merging

Let's say we have completed the new feature and want to get it back into the *master branch*.

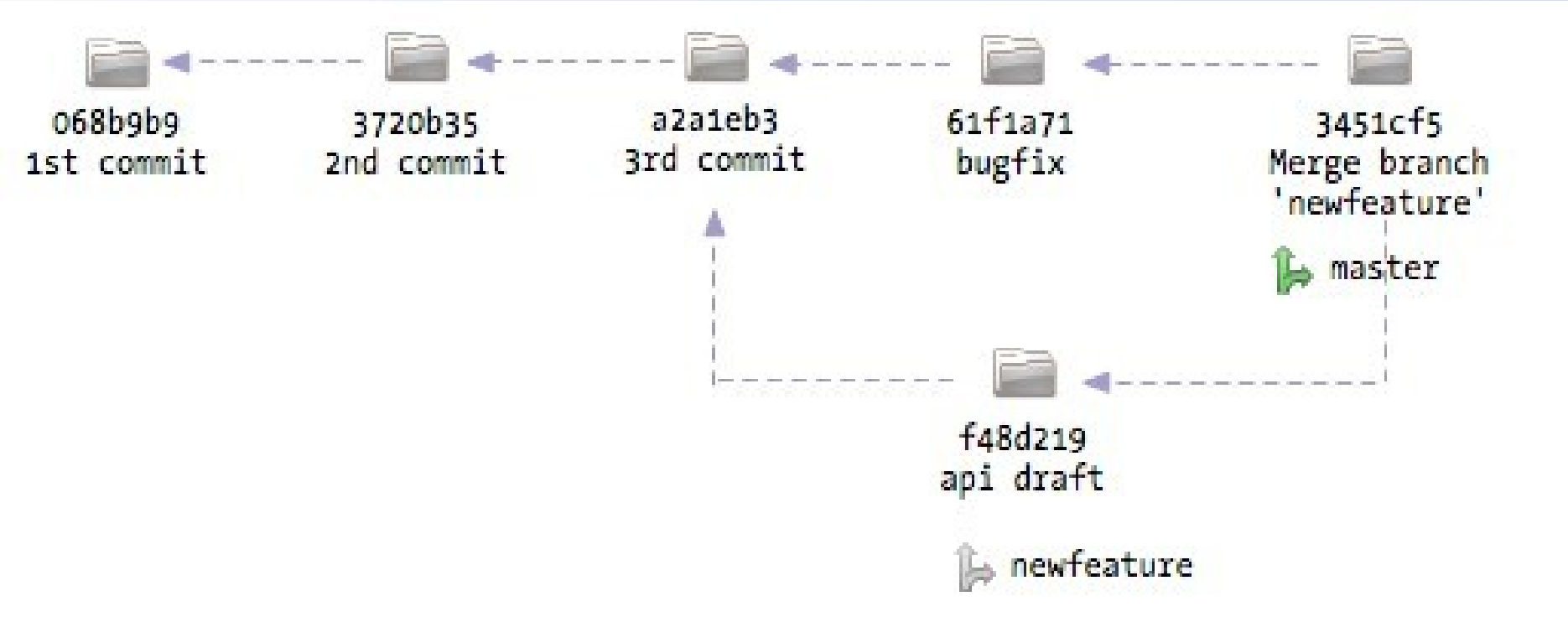
This can be achieved with the git merge command while having the *master branch* selected as the active branch:

```
git merge newfeature
```

Merge current and the *newfeature* branches.

Merging

If all goes well there will be no conflicts and git will just create a new commit containing the changes which have been made in both branches:



Merging

However, if the same contents were changed in both branches, a conflict will arise.To resolve them, look at the file with the conflict and you will see the respective lines marked. On the top you will see the version from the active branch, below you will see the version from the merged branch:

```
<<<<<<< HEAD:testfile.txt
this change was done in master
=======
this change was done in newfeature
>>>>>>> newfeature:testfile.txt
```

Merging

You have to resolve this conflict and remove the markers. After that you add the file to the index and commit the result:

```
git add testfile.txt
git commit
```

Deleting a branch

After you have merged the branch, you can delete it should you not need it anymore:

```
git branch -d newfeature
```

What about the remote repository?

Create a remote repository : www.github.com

Create a remote pointing at your GitHub repository :

git remote add origin
https://github.com/<username>/<project>.git

Send your commits in the "master" branch to GitHub :
git push origin master

What about the remote repository?

Transfering existing remote repository locally:
git clone <URL>

Pushing local commits to your remote repository stored on GitHub :
git push

Updating from remote repository :
git pull

Task

- Create a repository (bitbucket.org)
- Initialize your working tree
- Commit some text files (test1.txt, test2.txt)
- Pull in the repository
- Create two branches
- Make changes in both of them and commit
- Tag some commit
- Try to synchronize one of them with the master
- *Transfer branches on remote repository

Further reading

- http://git-scm.com/doc
- http://www.vogella.com/articles/Git/article.html
- Adding and removing remote branches :
  - http://www.gitguys.com/topics/adding-and-removing-remote-branches/