Git and GitHub Workflow

e-Yantra Team

Embedded Real-Time Systems (ERTS) Lab
Indian Institute of Technology, Bombay

May 14, 2020

Agenda for Discussion

1. Version Control System (VCS)
   - Motivation
   - Need for VCS
2. Git
   - What is Git?
   - Features
   - Working Principle
   - Getting Started
3. GitHub
   - Remote Repositories
   - Branching
   - Merge Branches, Resolving Conflicts
   - Collaboration using GitHub
   - Git Reset, Revert, Rebase

Motivation

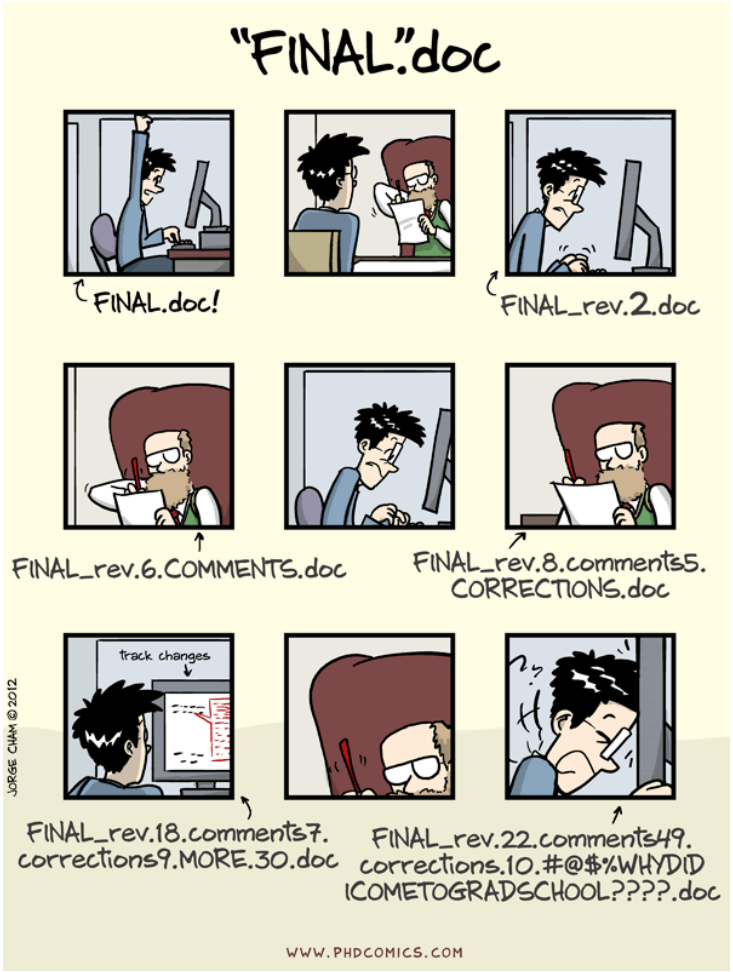

Figure 1: Multiple versions [1]

Need for VCS

Need for VCS

- Records your work

Need for VCS

- Records your work
- Keeps track of every change over time

Need for VCS

- Records your work
- Keeps track of every change over time
- Compare earlier versions

Need for VCS

- Records your work
- Keeps track of every change over time
- Compare earlier versions
- Share your project with others

Need for VCS

- Records your work
- Keeps track of every change over time
- Compare earlier versions
- Share your project with others
- Collaborate with team members

Need for VCS

- Records your work
- Keeps track of every change over time
- Compare earlier versions
- Share your project with others
- Collaborate with team members
- Attain peace of mind ! :)

Git vs Other VCSs

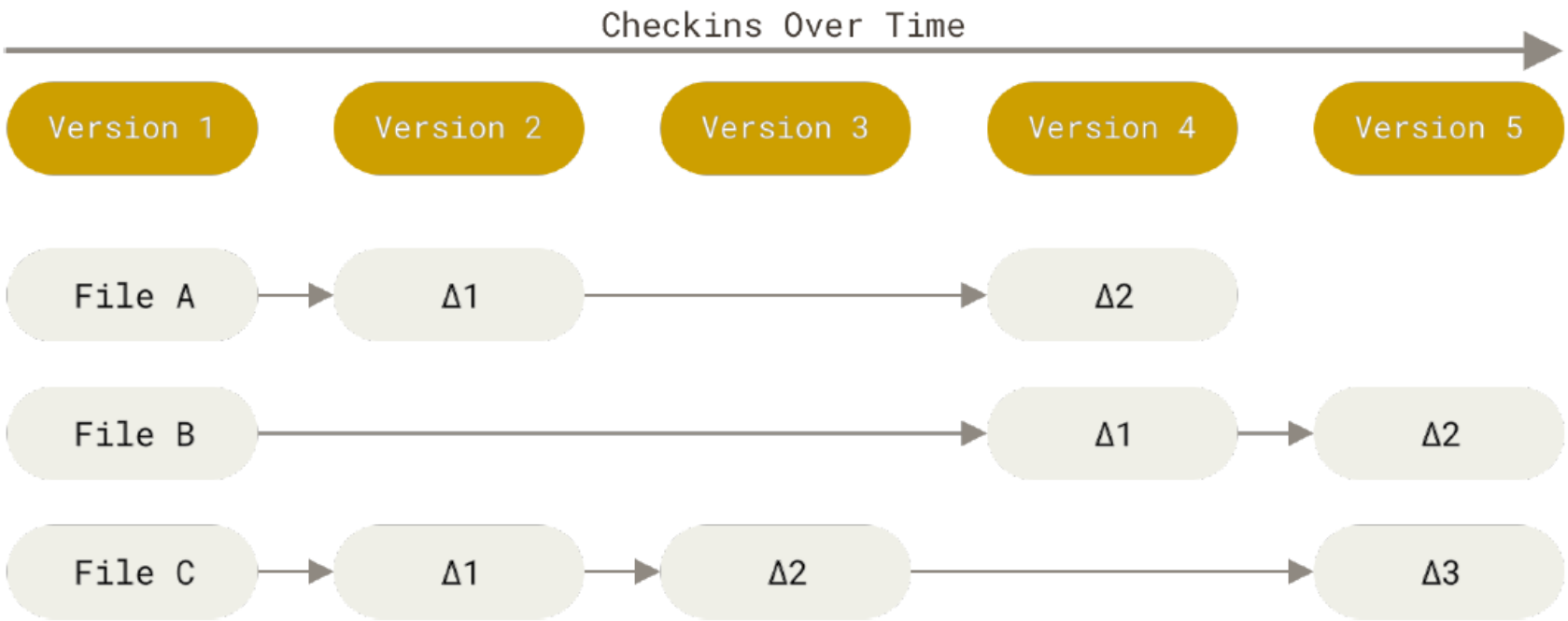

Figure 2: Storing data as changes to base version of each file [2]

Git Introduction

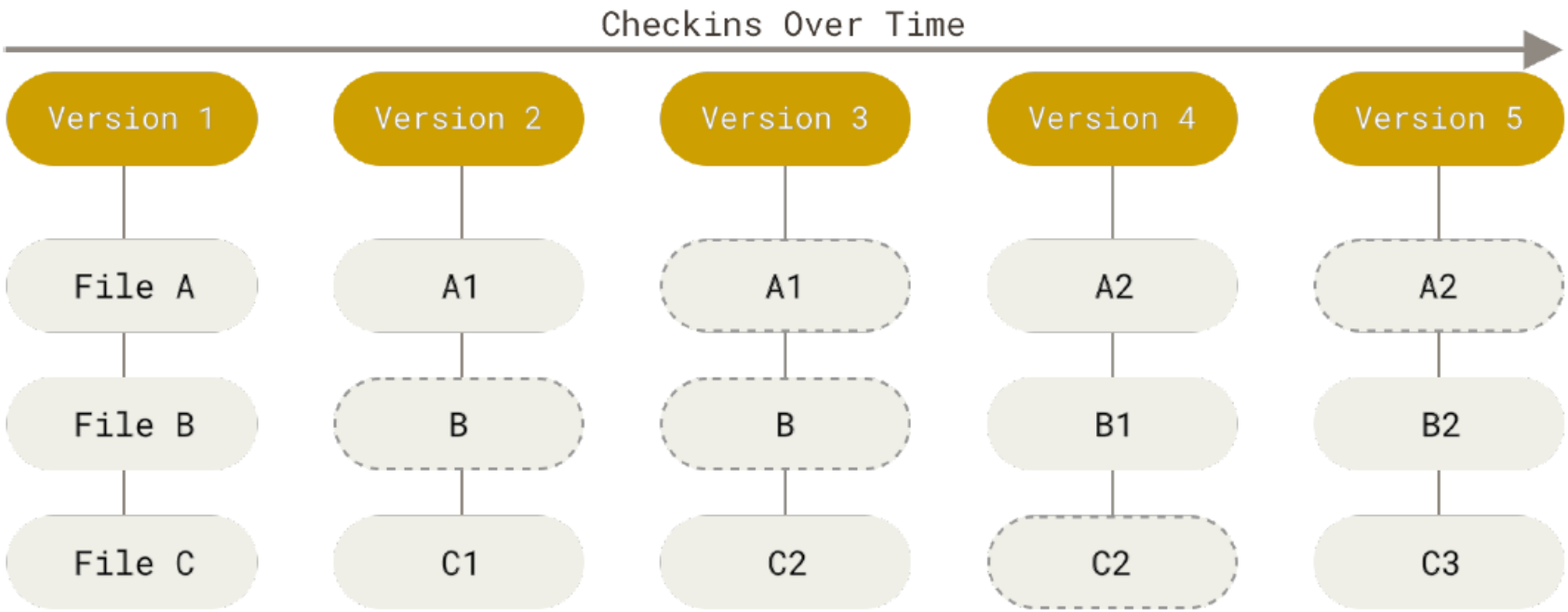

Figure 3: Storing data as snapshots of project over time [2]

Features

Features

- Every operation is Local

Features

- Every operation is Local
  - ⋆ entire project history on local disk
  - ⋆ local difference calculation of versions
  - ⋆ not much issue if no network connection

Features

- Every operation is Local
  - ⋆ entire project history on local disk
  - ⋆ local difference calculation of versions
  - ⋆ not much issue if no network connection
- Has integrity

Features

- Every operation is Local
  - ⋆ entire project history on local disk
  - ⋆ local difference calculation of versions
  - ⋆ not much issue if no network connection
- Has integrity
  - ⋆ before storing everything is check-summed
  - ⋆ storing by the hash value of content and not file names
  - ⋆ impossible to change contents w/o Git knowing about it

Features

- Every operation is Local
  - ⋆ entire project history on local disk
  - ⋆ local difference calculation of versions
  - ⋆ not much issue if no network connection
- Has integrity
  - ⋆ before storing everything is check-summed
  - ⋆ storing by the hash value of content and not file names
  - ⋆ impossible to change contents w/o Git knowing about it
- Only adds data

Features

- Every operation is Local
  - ⋆ entire project history on local disk
  - ⋆ local difference calculation of versions
  - ⋆ not much issue if no network connection
- Has integrity
  - ⋆ before storing everything is check-summed
  - ⋆ storing by the hash value of content and not file names
  - ⋆ impossible to change contents w/o Git knowing about it
- Only adds data
  - ⋆ nearly all actions only *add* data to database
  - ⋆ lose of data that is not yet committed

Features

- Every operation is Local
  - ⋆ entire project history on local disk
  - ⋆ local difference calculation of versions
  - ⋆ not much issue if no network connection
- Has integrity
  - ⋆ before storing everything is check-summed
  - ⋆ storing by the hash value of content and not file names
  - ⋆ impossible to change contents w/o Git knowing about it
- Only adds data
  - ⋆ nearly all actions only *add* data to database
  - ⋆ lose of data that is not yet committed

So, Git essentially stores a snapshot of the current state of the project along with a unique key (hash value) to address it when needed.

Working Principle

Working Principle

Three main states where your files reside in Git:

Working Principle

Three main states where your files reside in Git:

Modified **Staged** **Committed**

Working Principle

Three main states where your files reside in Git:

Modified **Staged** **Committed**

Three main sections of Git project:

Working Principle

Three main states where your files reside in Git:

Modified **Staged** **Committed**

Three main sections of Git project:

Working Tree **Staging Area** **Git Directory**

Working Principle

Three main states where your files reside in Git:

Modified **Staged** **Committed**

Three main sections of Git project:

Working Tree **Staging Area** **Git Directory**

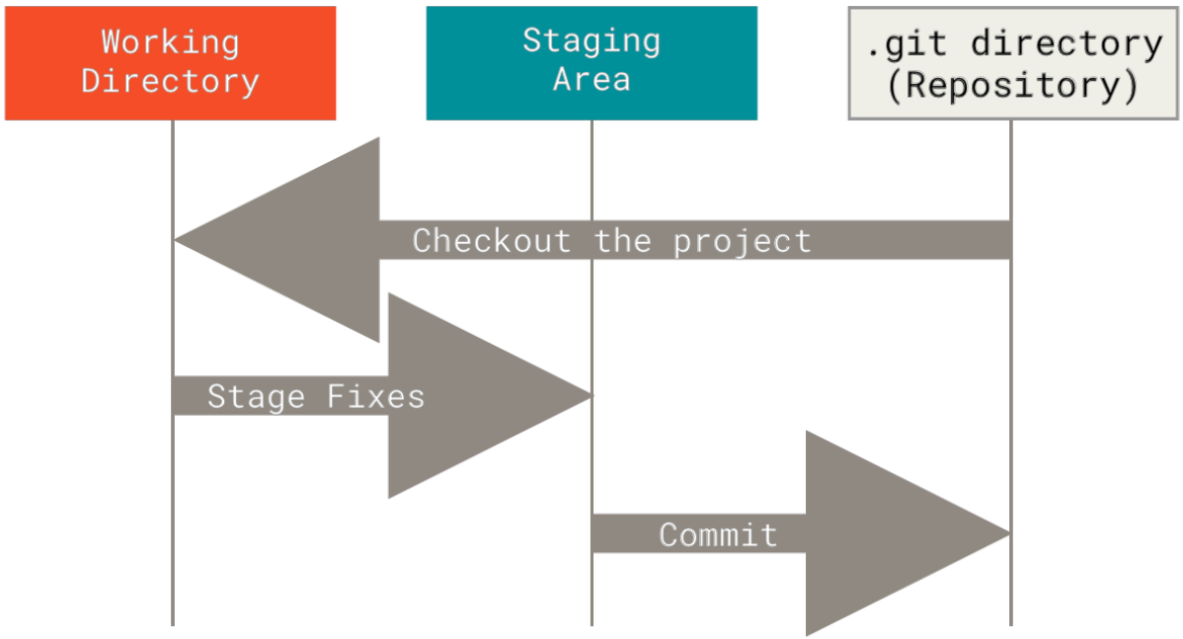

Figure 4: Three sections of Git project [2]

Git Workflow

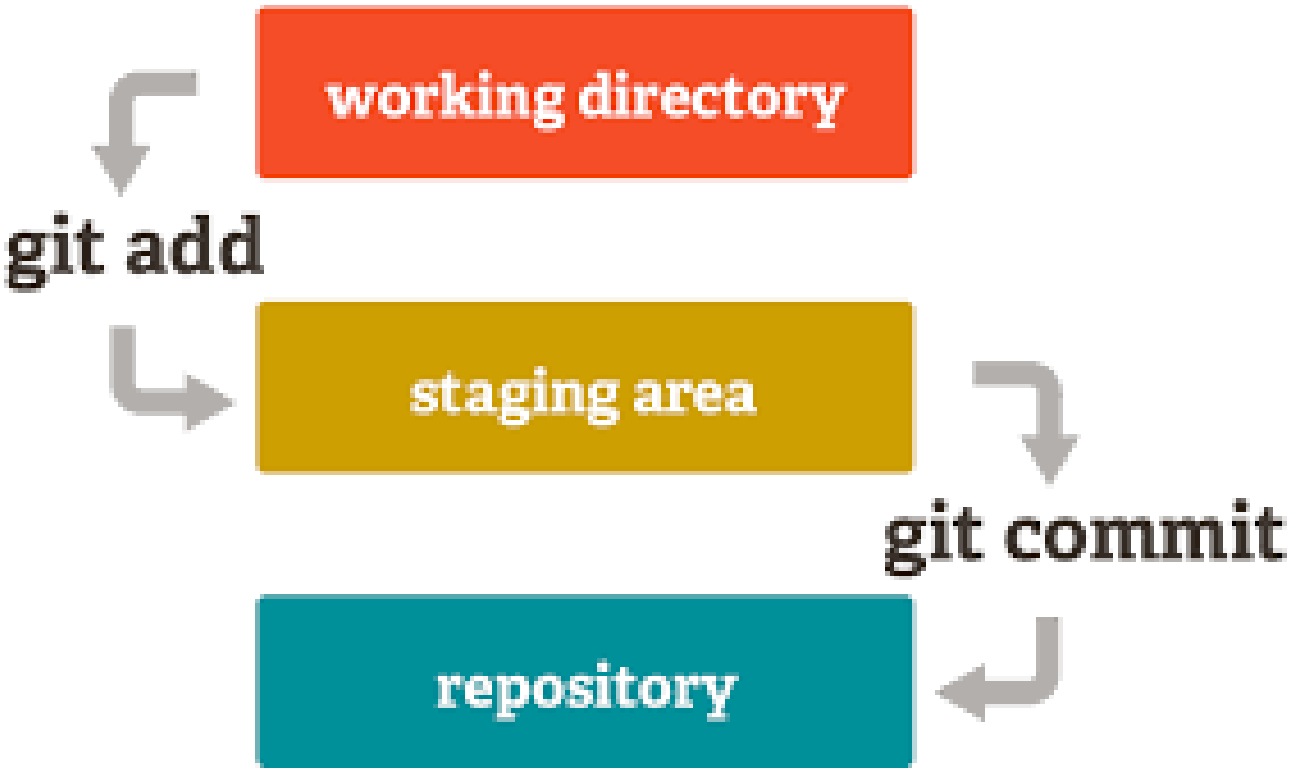

Figure 5: Git Workflow

Getting Started

Getting Started

Create local Git repository

Getting Started

Create local Git repository

```
mkdir my_eysip_project
git init
```

Getting Started

Create local Git repository

```
mkdir my_eysip_project
git init
```

Setup your identity

Getting Started

Create local Git repository

```
mkdir my_eysip_project
git init
```

Setup your identity

```
git config --global user.name <registered_username>
git config --global user.email <registered_email_address>
```

Getting Started

Create local Git repository

```
mkdir my_eysip_project
git init
```

Setup your identity

```
git config --global user.name <registered_username>
git config --global user.email <registered_email_address>
```

Create and add a file to local repository

Getting Started

Create local Git repository

```
mkdir my_eysip_project
git init
```

Setup your identity

```
git config --global user.name <registered_username>
git config --global user.email <registered_email_address>
```

Create and add a file to local repository

```
echo "Hey Git!" > README
git add README
```

Getting Started (continued)

Getting Started (continued)

Checking status

Getting Started (continued)

Checking status

```
git status
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

View commit history

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

View commit history

```
git log
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

View commit history

```
git log
git log <filename>
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

View commit history

```
git log
git log <filename>
git log -p (OR) git log --patch
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

View commit history

```
git log
git log <filename>
git log -p (OR) git log --patch
git log --decorate --oneline (OR) git log --pretty=oneline
```

Getting Started (continued)

Checking status

```
git status
```

Commit your changes

```
git commit -m "<my_commit_message>"
git commit -am "<my_commit_message>"
```

View commit history

```
git log
git log <filename>
git log -p (OR) git log --patch
git log --decorate --oneline (OR) git log --pretty=oneline
git log --pretty=format:"%h %s" --graph
```

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

`HEAD`

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

`HEAD`

`git diff`

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
```

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
git help <command>
```

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
git help <command>
```

Removing files:

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
git help <command>
```

Removing files:

```
git rm <filename>
```

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
git help <command>
```

Removing files:

```
git rm <filename>
git rm --cached <filename>
```

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
git help <command>
```

Removing files:

```
git rm <filename>
git rm --cached <filename>
```

Moving or Renaming files:

More details

Git takes snapshots of the tracked files in the working directory and stores a compressed version in its database.

Some important commands/terms:

```
HEAD
git diff
git diff --staged
git help <command>
```

Removing files:

```
git rm <filename>
git rm --cached <filename>
```

Moving or Renaming files:

```
git mv <file_from> <file_to>
```

Ignoring Files

Ignoring Files

.gitignore

Ignoring Files

.gitignore

```
touch .gitignore
cat .gitignore
```

Ignoring Files

.gitignore

```
touch .gitignore
cat .gitignore

# ignore all .o files
*.o
```

Ignoring Files

.gitignore

```
touch .gitignore
cat .gitignore

# ignore all .o files
*.o

# but keep track of main.o, even though ignoring
# .o files above
!main.o
```

Ignoring Files

.gitignore

```
touch .gitignore
cat .gitignore

# ignore all .o files
*.o

# but keep track of main.o, even though ignoring
# .o files above
!main.o

# only ignore TODO file in the current directory,
# not subdir/TODO
/TODO
```

Ignoring Files

.gitignore

Ignoring Files

.gitignore

```
# ignore all files in any directory named build
build/
```

Ignoring Files

.gitignore

```
# ignore all files in any directory named build
build/

# ignore docs/dummy_notes.md but not
# docs/final_notes/arm_arch.md
docs/*.md
```

Ignoring Files

.gitignore

```
# ignore all files in any directory named build
build/

# ignore docs/dummy_notes.md but not
# docs/final_notes/arm_arch.md
docs/*.md

# ignore all .html files in the docs/ directory
# and any of its sub-directories
docs/**/*.html
```

Ignoring Files

.gitignore

```
# ignore all files in any directory named build
build/

# ignore docs/dummy_notes.md but not
# docs/final_notes/arm_arch.md
docs/*.md

# ignore all .html files in the docs/ directory
# and any of its sub-directories
docs/**/*.html
```

This file is in root directory usually, but it is also possible to have additional *.gitignore* files in sub-directories.

Undoing Things

Undoing Things

Undoing the commit:

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
```

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
```

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
git commit --amend
```

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
git commit --amend
```

Un-staging a Staged file:

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
git commit --amend
```

Un-staging a Staged file:

```
git add dummy_notes.md
```

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
git commit --amend
```

Un-staging a Staged file:

```
git add dummy_notes.md
git reset HEAD dummy_notes.md
```

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
git commit --amend
```

Un-staging a Staged file:

```
git add dummy_notes.md
git reset HEAD dummy_notes.md
```

Un-modifying a Modified file:

Undoing Things

Undoing the commit:

```
git commit -m "Initial commit"
git add forgotten_file
git commit --amend
```

Un-staging a Staged file:

```
git add dummy_notes.md
git reset HEAD dummy_notes.md
```

Un-modifying a Modified file:

```
git checkout -- dummy_notes.md
```

Add, Fetch, Merge Remote Repos

Add, Fetch, Merge Remote Repos

Adding a remote repo explicitly:

Add, Fetch, Merge Remote Repos

Adding a remote repo explicitly:

```
git remote add <shortname> <.git url>
git remote -v
```

Add, Fetch, Merge Remote Repos

Adding a remote repo explicitly:

```
git remote add <shortname> <.git url>
git remote -v
```

Fetching and Merging all information from remote repo:

Add, Fetch, Merge Remote Repos

Adding a remote repo explicitly:

```
git remote add <shortname> <.git url>
git remote -v
```

Fetching and Merging all information from remote repo:

```
git fetch <shortname>
git merge <shortname>/master
git merge <shortname>/master --allow-unrelated-histories
```

Pull from, Push to, Inspect, Clone Remote Repos

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pushing to the remote repo:

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pushing to the remote repo:

```
git push <remote> <branch>
git push origin master
```

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pushing to the remote repo:

```
git push <remote> <branch>
git push origin master
```

Inspecting a remote repo:

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pushing to the remote repo:

```
git push <remote> <branch>
git push origin master
```

Inspecting a remote repo:

```
git remote show origin
```

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pushing to the remote repo:

```
git push <remote> <branch>
git push origin master
```

Inspecting a remote repo:

```
git remote show origin
```

Cloning a remote repo:

Pull from, Push to, Inspect, Clone Remote Repos

Pulling all information from remote repo*:

```
git pull <shortname> master
git pull <shortname> master --allow-unrelated-histories
```

* *May encounter merge conflict error* - will see later

Pushing to the remote repo:

```
git push <remote> <branch>
git push origin master
```

Inspecting a remote repo:

```
git remote show origin
```

Cloning a remote repo:

```
git clone <.git url>
```

Git Branching

- As stated before, Git stores snapshots of the working directory's staged contents on each commit.

Git Branching

- As stated before, Git stores snapshots of the working directory's staged contents on each commit.
- Each commit id acts as a key to the respective snapshot of the staged content.

Git Branching

- As stated before, Git stores snapshots of the working directory's staged contents on each commit.
- Each commit id acts as a key to the respective snapshot of the staged content.

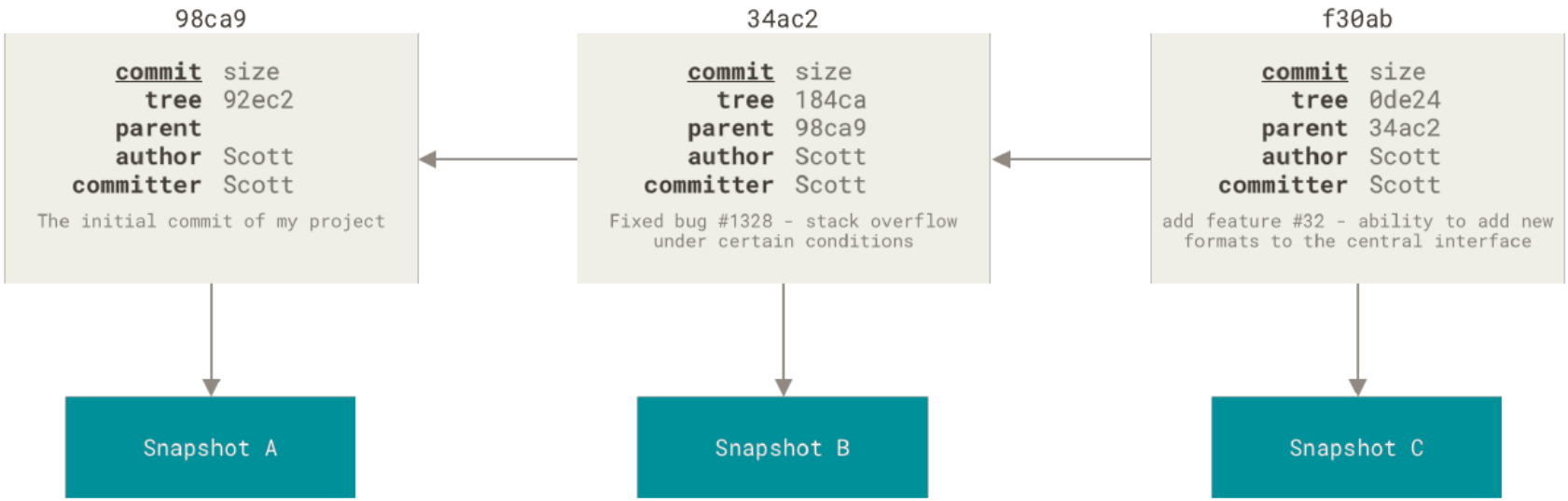

Figure 6: Commits and their parents example [2]

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Creating a New Branch:

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Creating a New Branch:

```
git branch testing
```

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Creating a New Branch:

```
git branch testing
```

Switching Branches:

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Creating a New Branch:

```
git branch testing
```

Switching Branches:

```
git checkout testing
git checkout master
```

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Creating a New Branch:

```
git branch testing
```

Switching Branches:

```
git checkout testing
git checkout master
```

Creating and Switching to New Branch at once:

Git Branching

- A Git branch is simply a movable pointer associated with a series of such sequential commits.
- Default branch name in Git is **master**.

Creating a New Branch:

```
git branch testing
```

Switching Branches:

```
git checkout testing
git checkout master
```

Creating and Switching to New Branch at once:

```
git checkout -b testing
```

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Make changes to new branch and merge:

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Make changes to new branch and merge:

```
git checkout master
git merge testing / git mergetool
git log --oneline
git branch / git branch --merged <branch>
git branch --no-merged <branch>
```

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Make changes to new branch and merge:

```
git checkout master
git merge testing / git mergetool
git log --oneline
git branch / git branch --merged <branch>
git branch --no-merged <branch>
```

Push contents of branches to the remote repo:

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Make changes to new branch and merge:

```
git checkout master
git merge testing / git mergetool
git log --oneline
git branch / git branch --merged <branch>
git branch --no-merged <branch>
```

Push contents of branches to the remote repo:

```
git push origin testing
git push origin master
git log --oneline master / testing
```

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Make changes to new branch and merge:

```
git checkout master
git merge testing / git mergetool
git log --oneline
git branch / git branch --merged <branch>
git branch --no-merged <branch>
```

Push contents of branches to the remote repo:

```
git push origin testing
git push origin master
git log --oneline master / testing
```

More Branching commands:

Merge Branches, Resolving Conflicts

- After creating and switching to a new branch, the HEAD points to the latest commit on this branch.

Make changes to new branch and merge:

```
git checkout master
git merge testing / git mergetool
git log --oneline
git branch / git branch --merged <branch>
git branch --no-merged <branch>
```

Push contents of branches to the remote repo:

```
git push origin testing
git push origin master
git log --oneline master / testing
```

More Branching commands:

```
gitk
gitk --all
git branch -d testing / git push origin --delete testing
```

Working on different branches

Two users working on two different branches of the same remote repository.

Working on different branches

Two users working on two different branches of the same remote repository.

User 1:

Working on different branches

Two users working on two different branches of the same remote repository.

User 1:

```
git clone <.git url> OR git pull (if already cloned)
git checkout -b user1_branch
<make changes>
git commit -am "user1 commit message"
git push origin user1_branch
```

Working on different branches

Two users working on two different branches of the same remote repository.

User 1:

```
git clone <.git url> OR git pull (if already cloned)
git checkout -b user1_branch
<make changes>
git commit -am "user1 commit message"
git push origin user1_branch
```

User 2:

Working on different branches

Two users working on two different branches of the same remote repository.

User 1:

```
git clone <.git url> OR git pull (if already cloned)
git checkout -b user1_branch
<make changes>
git commit -am "user1 commit message"
git push origin user1_branch
```

User 2:

```
git clone <.git url> OR git pull (if already cloned)
git checkout -b user2_branch
<make changes>
git commit -am "user2 commit message"
git push origin user2_branch
```

Working on common master branch

Both users working on same **master** branch of the remote repo.

Working on common master branch

Both users working on same **master** branch of the remote repo.

- *User1* and *User2* - before starting to work,

  ```
  git pull origin master
  ```

  to get all changes from remote repo

Working on common master branch

Both users working on same **master** branch of the remote repo.

- *User1* and *User2* - before starting to work,

  ```
  git pull origin master
  ```

  to get all changes from remote repo
- *User1* - makes changes, commit them to your local repo

Working on common master branch

Both users working on same **master** branch of the remote repo.

- *User1* and *User2* - before starting to work,

  ```
  git pull origin master
  ```

  to get all changes from remote repo
- *User1* - makes changes, commit them to your local repo
- *User1* - decides a fixed time and informs *User2* before pushing to repo, perform

  ```
  git push origin master
  ```

  to remote repo

Working on common master branch

Both users working on same **master** branch of the remote repo.

- *User1* and *User2* - before starting to work,

  ```
  git pull origin master
  ```

  to get all changes from remote repo
- *User1* - makes changes, commit them to your local repo
- *User1* - decides a fixed time and informs *User2* before pushing to repo, perform

  ```
  git push origin master
  ```

  to remote repo
- *User2* -

  ```
  git pull --rebase origin master
  ```

  again to get changes made by *User1*, resolve merge conflicts arising on User2's work manually if it fails during merge **OR** by **Git Stashing** if it fails while starting to merge.

Git Stashing

Git Stashing

- Git Pull or Merge fails as there are changes in working directory OR staging area that could be overwritten

Git Stashing

- Git Pull or Merge fails as there are changes in working directory OR staging area that could be overwritten
- Want to switch branches, but don't wish to commit half-done work

Git Stashing

- Git Pull or Merge fails as there are changes in working directory OR staging area that could be overwritten
- Want to switch branches, but don't wish to commit half-done work

Stashing takes the dirty state of working directory, saves it on a stack of unfinished changes that can be reapplied any time even on different branches.

Stashing your work:

Git Stashing

- Git Pull or Merge fails as there are changes in working directory OR staging area that could be overwritten
- Want to switch branches, but don't wish to commit half-done work

Stashing takes the dirty state of working directory, saves it on a stack of unfinished changes that can be reapplied any time even on different branches.

Stashing your work:

```
git status
git stash push
git status -s
git stash list
git stash apply / git stash apply --index
git status -s
git stash drop / git stash drop stash@{0}
git stash pop
git stash push -u / git stash push --patch
```

Git Reset

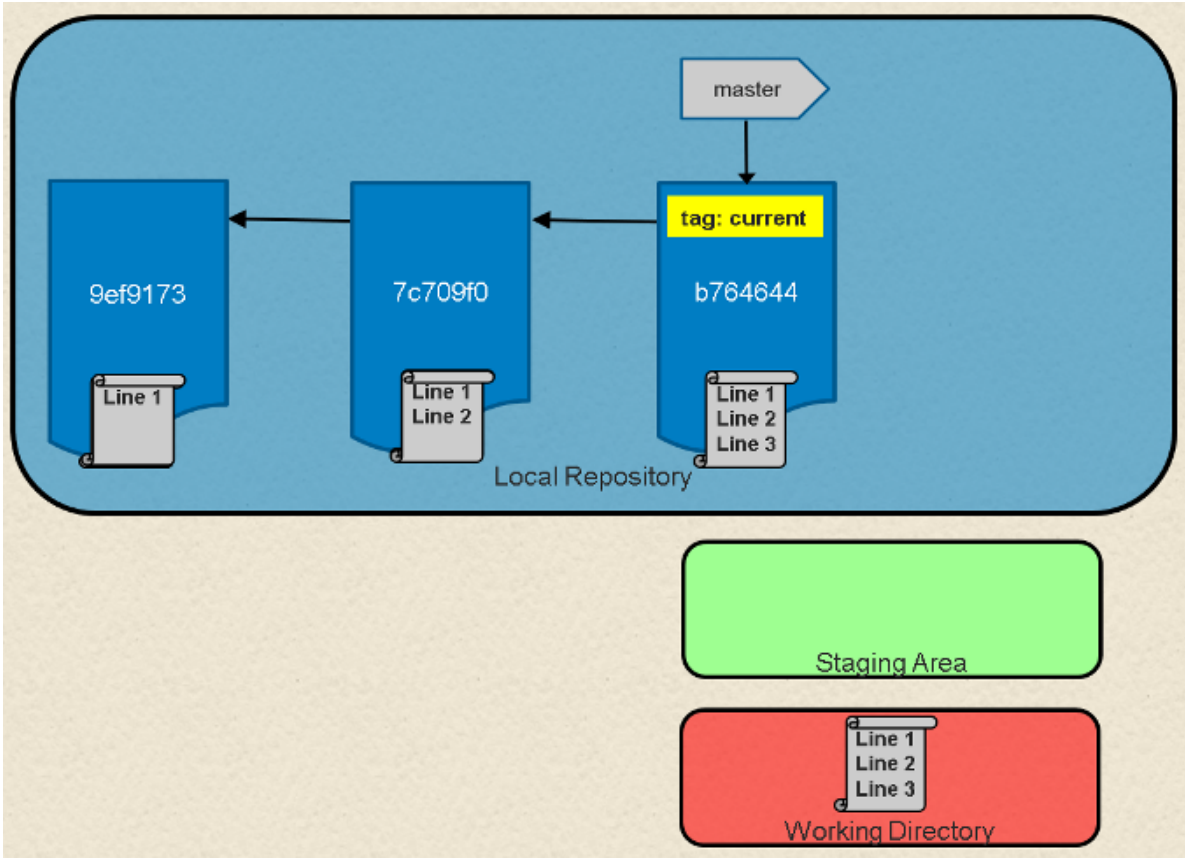

Figure 7: Local Git repository example [4]

Git Reset

Reset

```
git log --oneline
git reset 9ef9173 / git reset HEAD~2
git log --oneline
cat .git/ORIG_HEAD
git reset <sha1-id>
```

Git Reset

Reset

```
git log --oneline
git reset 9ef9173 / git reset HEAD~2
git log --oneline
cat .git/ORIG_HEAD
git reset <sha1-id>
```

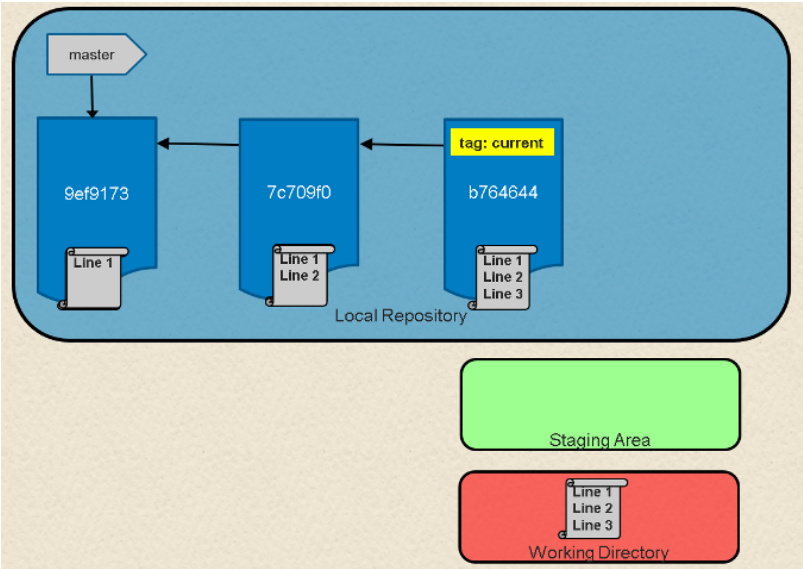

Figure 8: After git reset HEAD 2 [4]

Git Revert

Revert

```
git log --oneline
git revert HEAD
git log --oneline
cat .git/ORIG_HEAD
git reset <sha1-id>
```

Git Revert

Revert

```
git log --oneline
git revert HEAD
git log --oneline
cat .git/ORIG_HEAD
git reset <sha1-id>
```

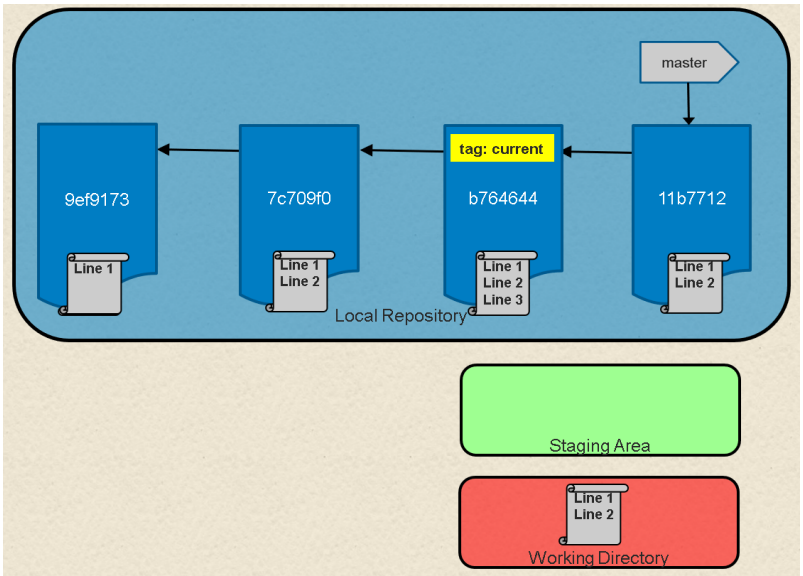

Figure 9: After git revert HEAD [4]

Git Rebase

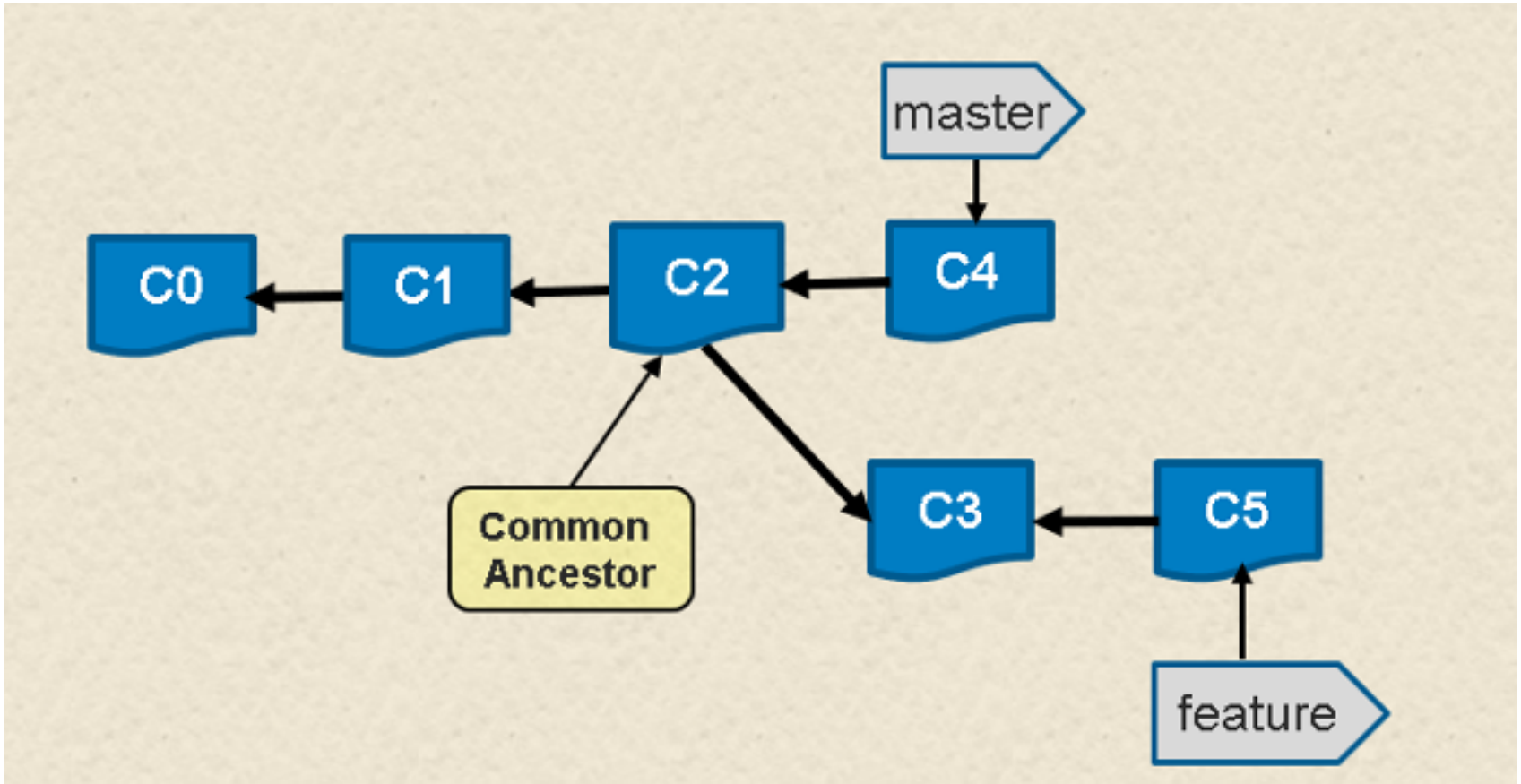

Figure 10: Chain of commits for master and feature branches [4]

Git Rebase

Rebase

```
git log --oneline master
git log --oneline feature
git checkout feature
git rebase master
git add <some-file>
git rebase --continue / git rebase --abort
git log --oneline feature
```

Git Rebase

Rebase

```
git log --oneline master
git log --oneline feature
git checkout feature
git rebase master
git add <some-file>
git rebase --continue / git rebase --abort
git log --oneline feature
```

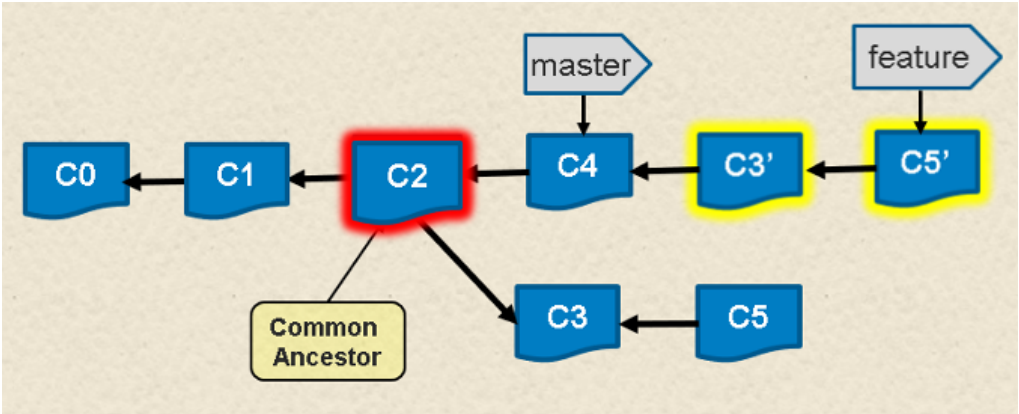

Figure 11: After git rebase master [4]

Git Rebase

Rebase

```
cat .git/ORIG_HEAD
git reset <sha1-id>
git log --oneline feature
git reflog
git reset HEAD@{1}
```

Git Rebase

Rebase

```
cat .git/ORIG_HEAD
git reset <sha1-id>
git log --oneline feature
git reflog
git reset HEAD@{1}
```

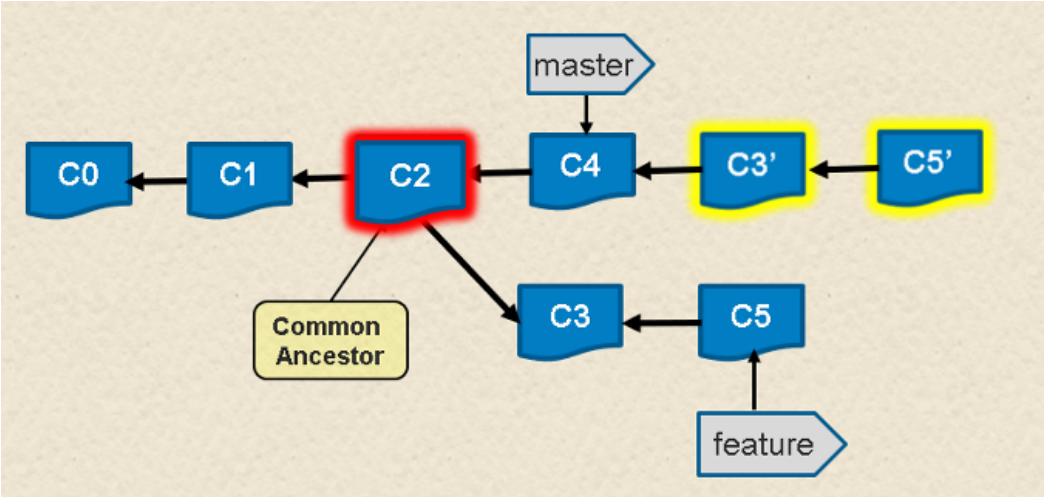

Figure 12: After undo of git rebase master [4]

References

[1] PhD Comics
[2] Pro Git Book, 2nd Edition
[3] Resolving Merge Conflicts in Git
[4] How to reset, revert and return to previous state in Git
[5] Comparing Git Workflows

Thank You!

Author name: e-Yantra Team