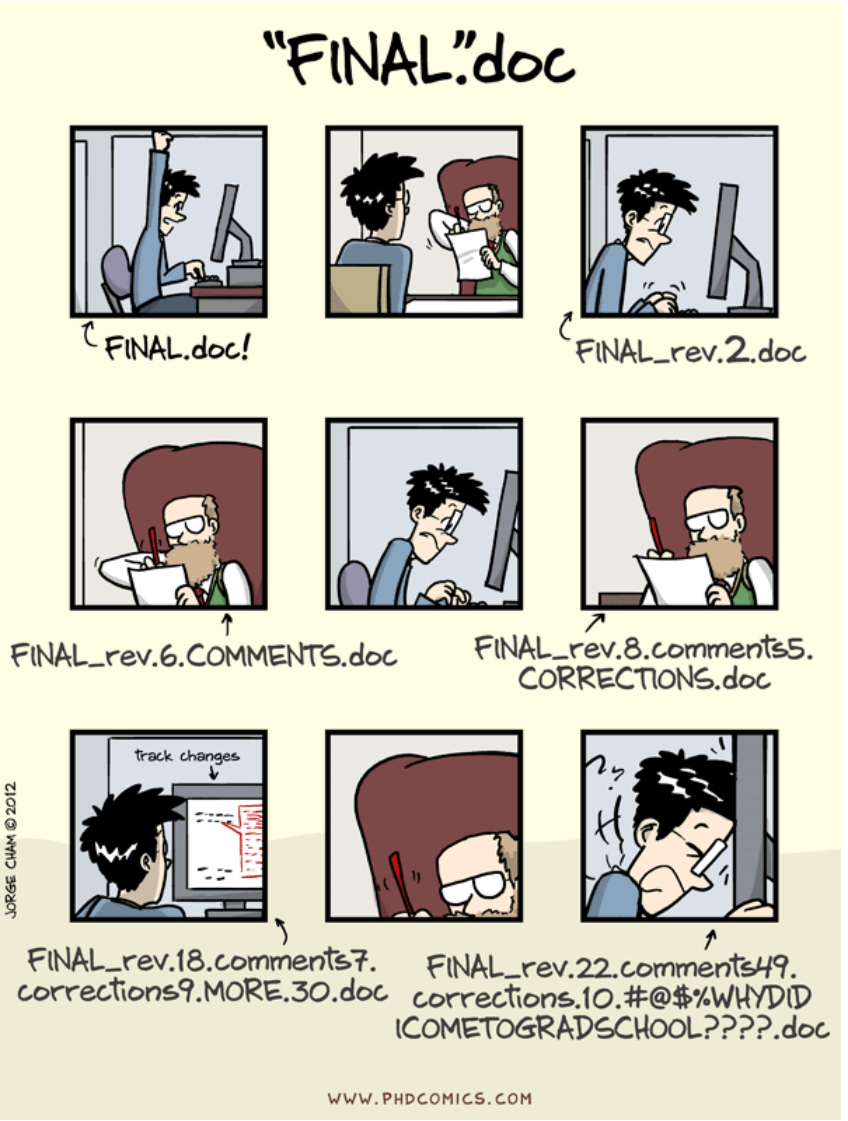
How to work with git

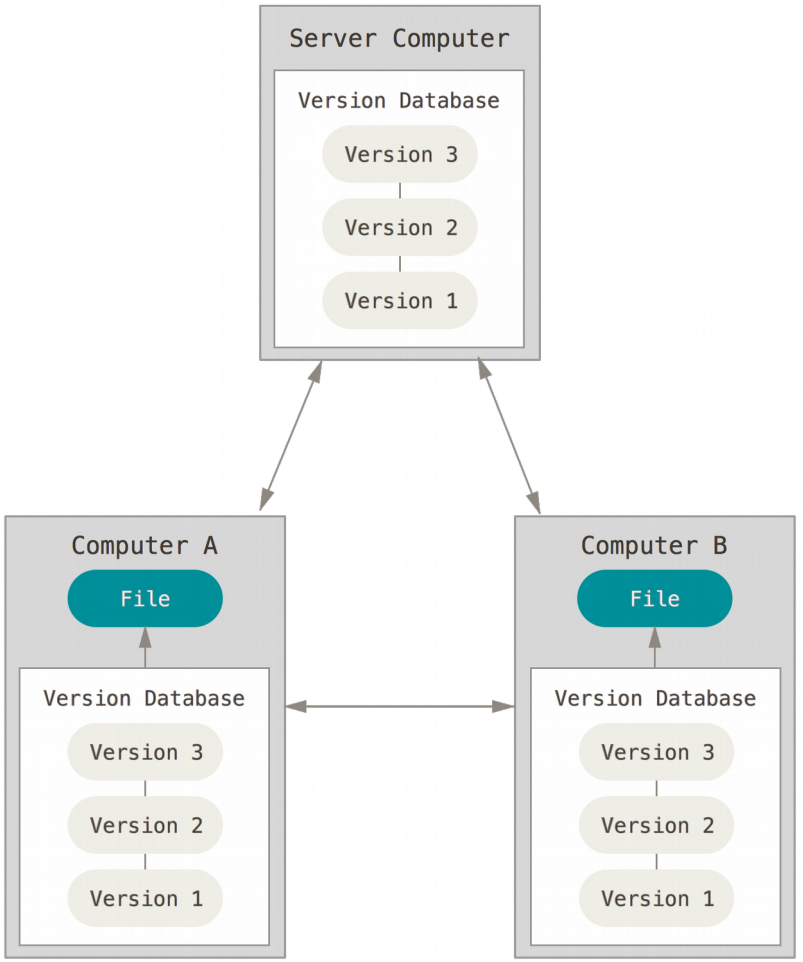
Sebastian Illing

Why use a Version Control System?

- Storing versions (properly)
- Restoring Previous Versions
- Understanding What Happened
- Collaboration
- Backup



Git is a Distributed Version Control System



- Clients mirror whole repository
- No single point of failure
- Most work is local
- Remote is optional

Getting a Git Repository

Initializing a Repository in an Existing Directory

```
> cd existing_directory
> git init
```

- Creates subdirectory *.git*
- But nothing in the directory is tracked yet!

```
> git add <files>  # track files
> git commit -m 'initial project version'
```

Cloning an Existing Repository

```
> git clone https://github.com/some_repo.git
```

- Creates directory *some_repo*
- Initializes a *.git* directory
- Pulls down all data of the repository

Basic Workflow

1. Start by pulling down the latest changes from the server

```
> git pull
```

2. Work as usual

3. Wrap up your changes in a commit

```
# See what has changed
> git status
> git diff <filename>

# Choose files to commit
> git add <file1> <file2>

# Finally commit the changes
> git commit
```

4. Push your changes to the server

```
> git push
```

Basic Workflow

1. Start by pulling down the latest changes from the server

```
> git pull
```

2. Work as usual

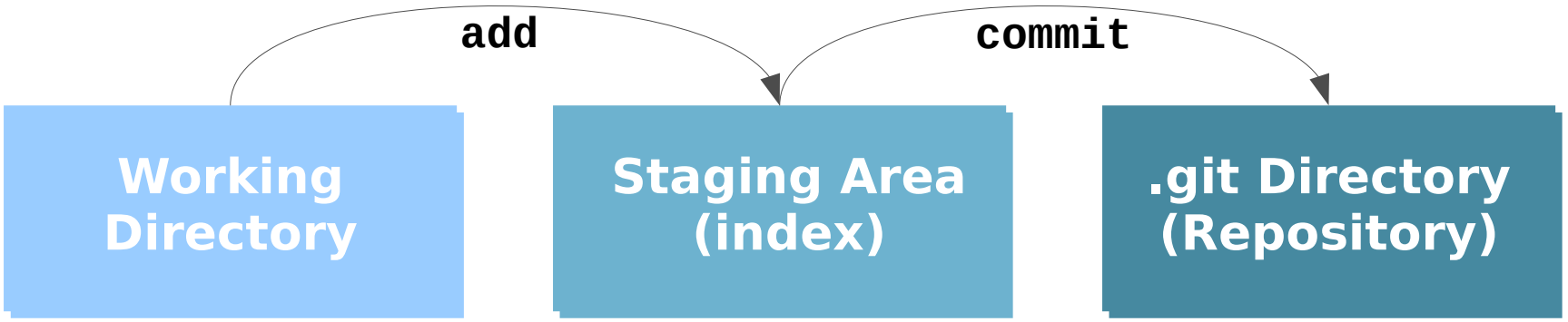
3. Wrap up your changes in a commit

```
# See what has changed
> git status
> git diff <filename>

# Choose files to commit
> git add <file1> <file2>

# Finally commit the changes
> git commit
```

4. Push your changes to the server

```
> git push
```

The Three States



Modify your files

Stage files for next commit

Commit stores staged files permanently

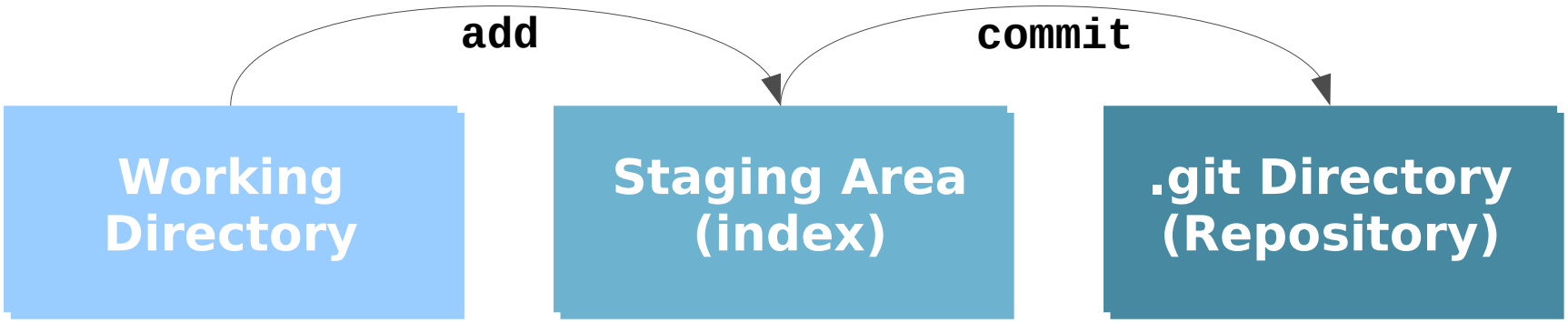
How to add files to staging area?

```
> git add <file_names>
```

How to commit staged files?

```
> git commit
```

Why is the staging area useful?



- Split work into separate commits
  → Full control what you want to commit

```
> git add -e <file>
```

- Allows / forces you to review changes

```
# Bypass the staging area
> git commit -a
```

Checking the Status of your Files

```
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

```
> echo 'My Project' > README
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will
be committed)

    README

nothing added to commit but untracked files
present (use "git add" to track)
```

Checking the Status of your Files

```
> git add README
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

Checking the Status of your Files

```
> vim CONTRIBUTING.md  # change some lines
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be
committed)
  (use "git checkout -- <file>..." to discard
changes in working directory)

    modified:   CONTRIBUTING.md
```

Viewing your changes

How to see your unstaged changes?

```
> git diff <optional-filename>
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 54fe737..d5878d7 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -1,3 +1,3 @@
 This is a testfile
-Here is a tpyo
+Here is a typo
```

How to see your staged changes?

```
> git diff --staged <optional-filename>
```

How to see difference between two commits?

```
> git diff <commit> <commit>
```

Basic Workflow

1. Start by pulling down the latest changes from the server

```
> git pull
```

2. Work as usual

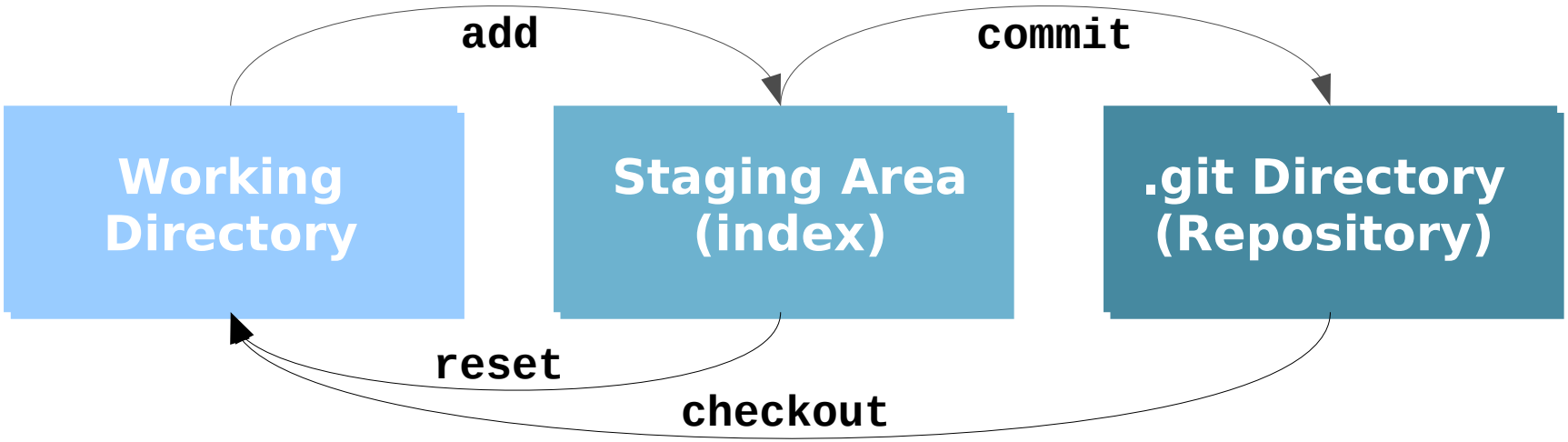
3. Wrap up your changes in a commit

```
# See what has changed
> git status
> git diff <filename>

# Choose files to commit
> git add <file1> <file2>

# Finally commit the changes
> git commit
```

4. Push your changes to the server

```
> git push
```

Undoing things




```
# Unstaging a Staged File
> git reset <file>
```

```
# Unmodifying a Modified File
> git checkout <file>
```

What if you commited too early or messed up your commit message?

```
# Adds staging area to last commit
> git commit --amend
```

Viewing the Commit History

```
> git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Illing <sebastian.illing@met.fu-berlin.de>
Date:   Thu Feb 2 16:52:11 2017 +0000

    change the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Illing <sebastian.illing@met.fu-berlin.de>
Date:   Tue Jan 31 16:40:33 2017 +0000

    add README

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Illing <sebastian.illing@met.fu-berlin.de>
Date:   Mon Jan 30 10:31:28 2017 +0000

    first commit
```

Basic Workflow

1. Start by pulling down the latest changes from the server

```
> git pull
```

2. Work as usual

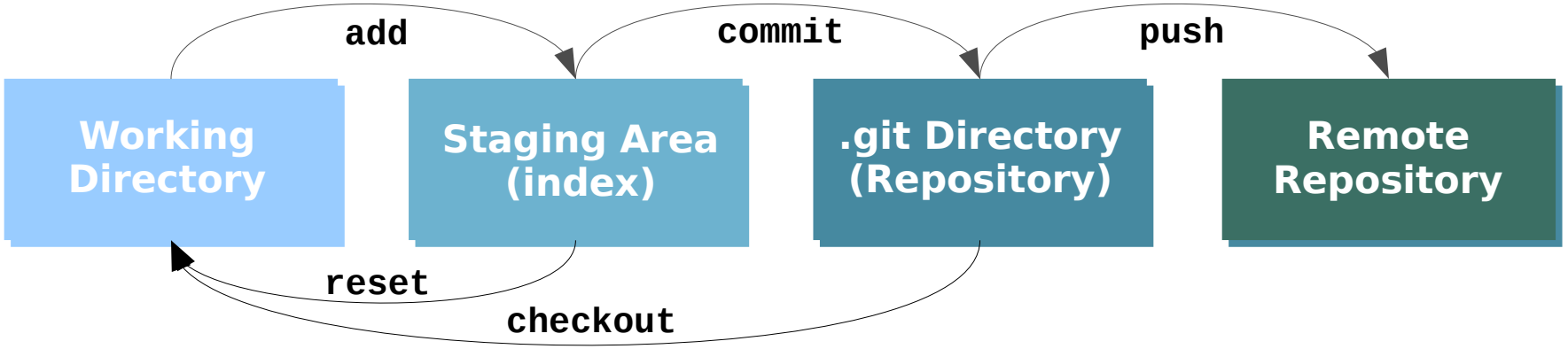
3. Wrap up your changes in a commit

```
# See what has changed
> git status
> git diff <filename>

# Choose files to commit
> git add <file1> <file2>

# Finally commit the changes
> git commit
```

4. Push your changes to the server

```
> git push
```

Working with remotes

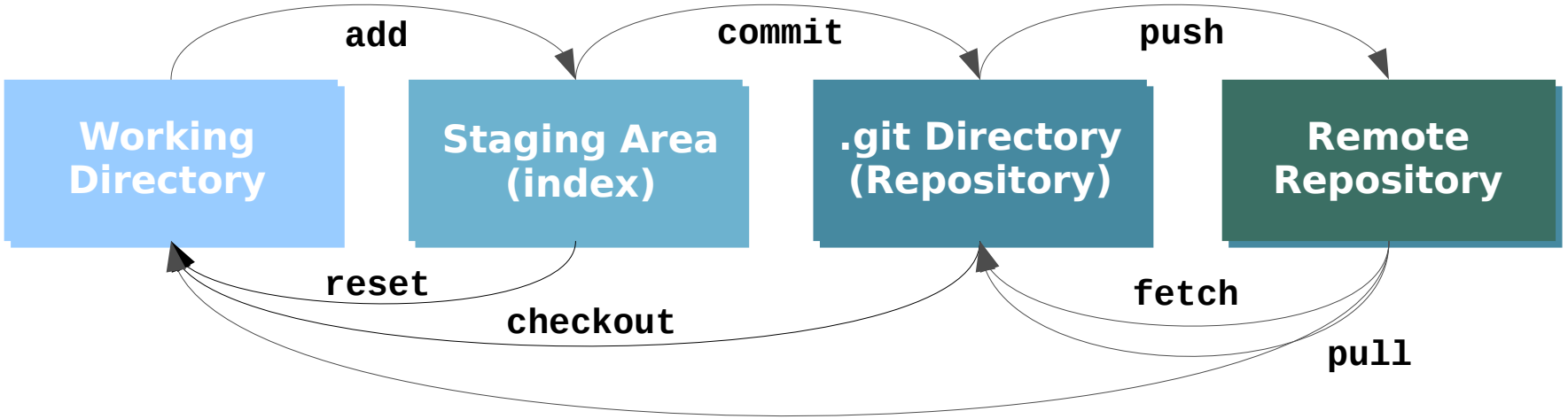



```
# add a remote repository
> git remote add <url>
> git branch --set-upstream master origin/master
```

How to **push** changes to remote?

```
> git push
```

Working with remotes



How to **pull** latest changes from server?

```
# Fetch changes
> git fetch
> git merge origin master
```

```
# Fetch and merge combined
> git pull
```

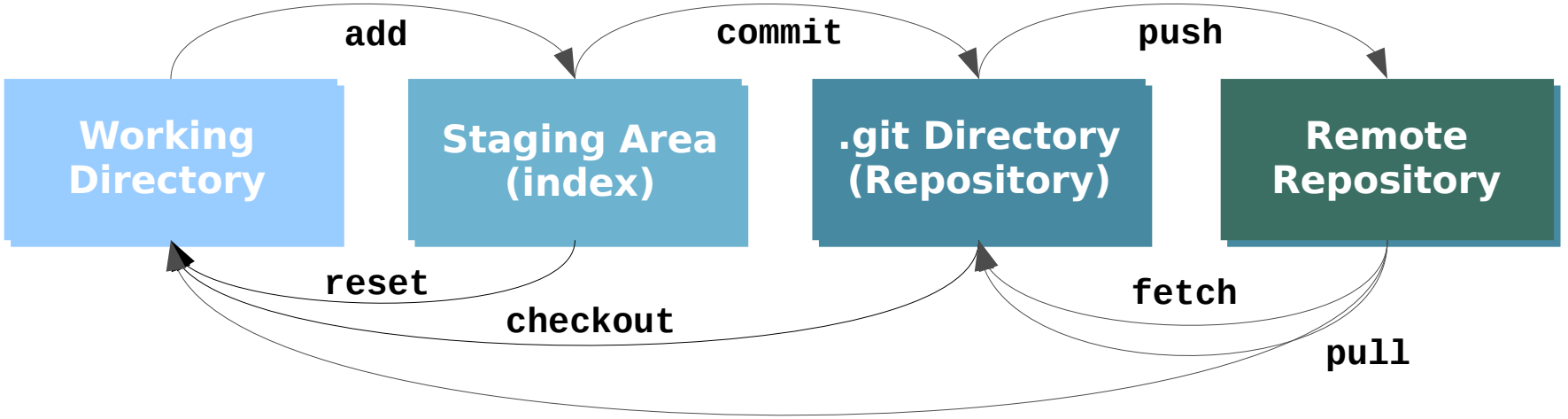
Basic Merge Conflicts

```
> git pull or git merge origin master
Auto-merging calc.py
CONFLICT (content): Merge conflict in calc.py
Automatic merge failed;
fix conflicts and then commit the result.
```

```
<<<<<<< HEAD:calc.py
contact : email.support@freva.de
=======
please contact us at support@freva.de
>>>>>>> origin:calc.py
```

```
please contact us at email.support@github.com
```

```
# commit resolved merge conflict
> git add calc.py
> git commit
```




```
# Get changes from remote
> git pull
# Do your usual work
# See what has changed
> git status
> git diff <filename>
# Choose files to commit
> git add <file1> <file2>
# Finally commit the changes
> git commit
# Push commits to remote
> git push
```

GitLab

GitLab is a graphical interface for your remote repositories

https://gitlab.met.fu-berlin.de/

Login using your ldap account

Additional Resources

Git cheat sheet:
https://www.git-tower.com/blog/git-cheat-sheet

Video course (about 4h):
https://de.udacity.com/course/how-to-use-git-and-github--ud775/

Very good ebook:
https://git-scm.com/book/en/v2

And of course:
git <command> --help

Working in Contexts

In real projects, work happens in multiple **contexts** in parallel:

- You have a stable version of your software (context 1)
- You are writing on the documentation (context 2)
- You're implementing a new feature (context 3)
- You're also trying to fix an annoying bug (context 4)

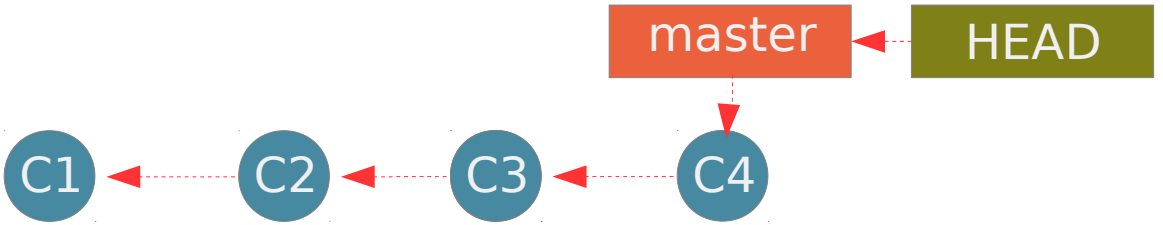
Not working in clearly separated contexts will cause **problems**:

- You have developed a cool plugin for the Freva system
- You're working on a new feature and made some commits
- Someone discovers a critical bug

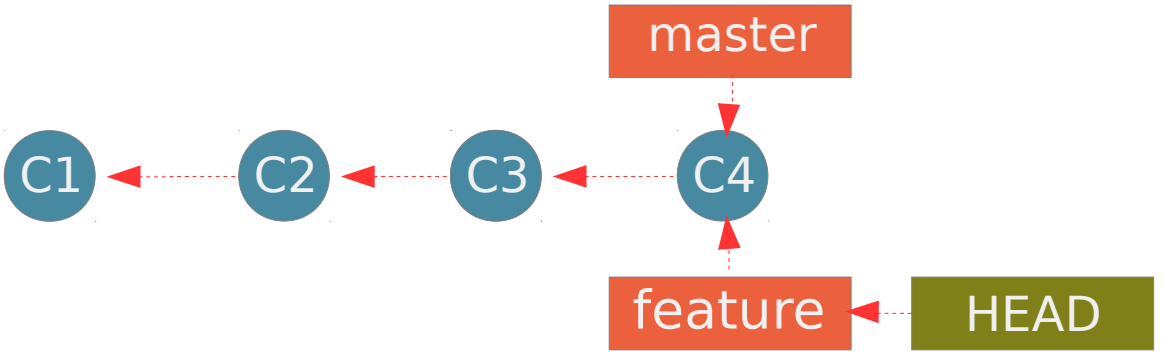
What can you do?

Working with Branches

A Branch represents a context in a project and helps you keep it separate from all other contexts.

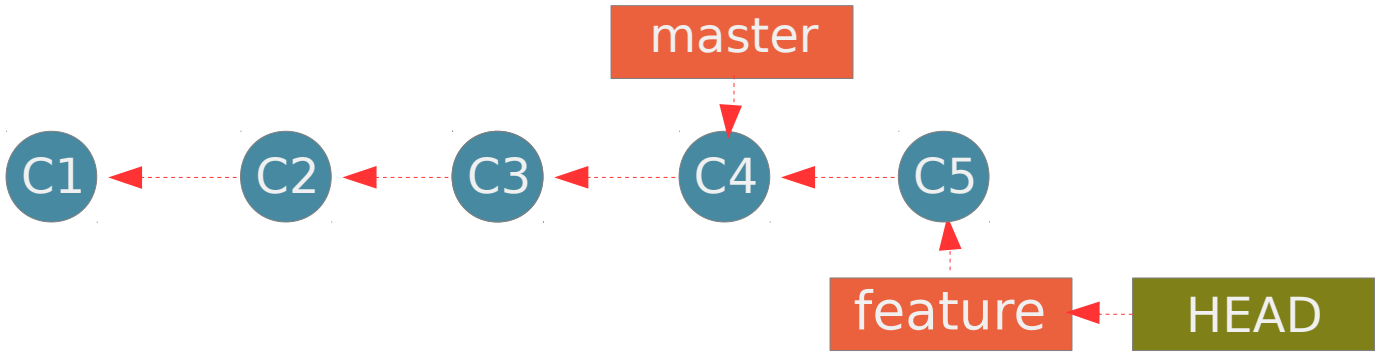



```
# create a new branch
> git branch feature
# switch context
> git checkout feature
```



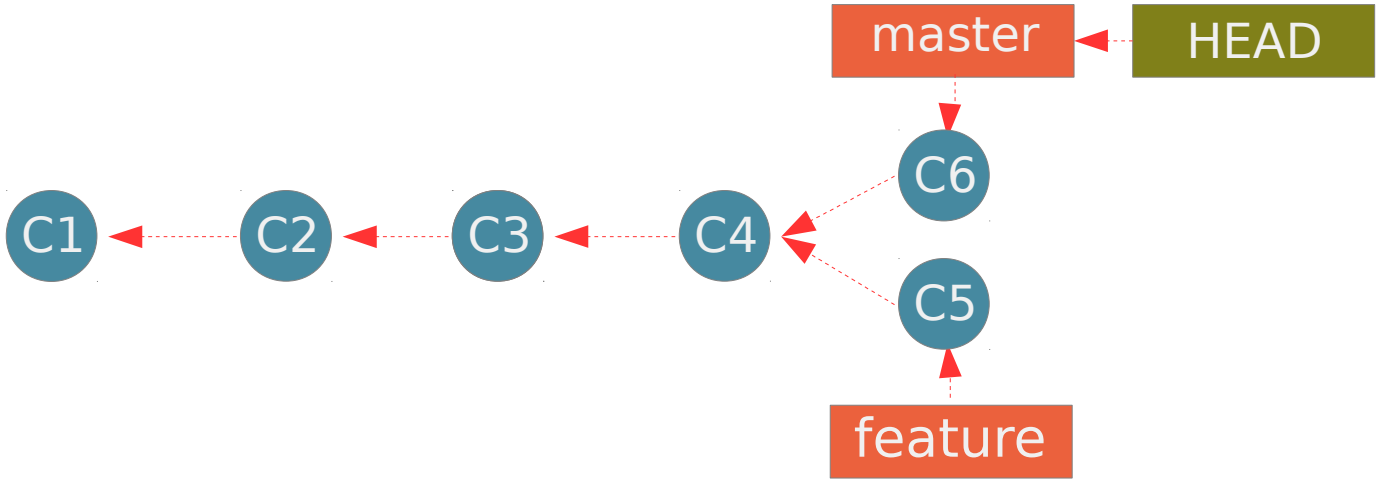
Working with Branches

```
# do some work
> vim test.py
> git add test.py
> git commit -m 'made a change'
```



Working with Branches

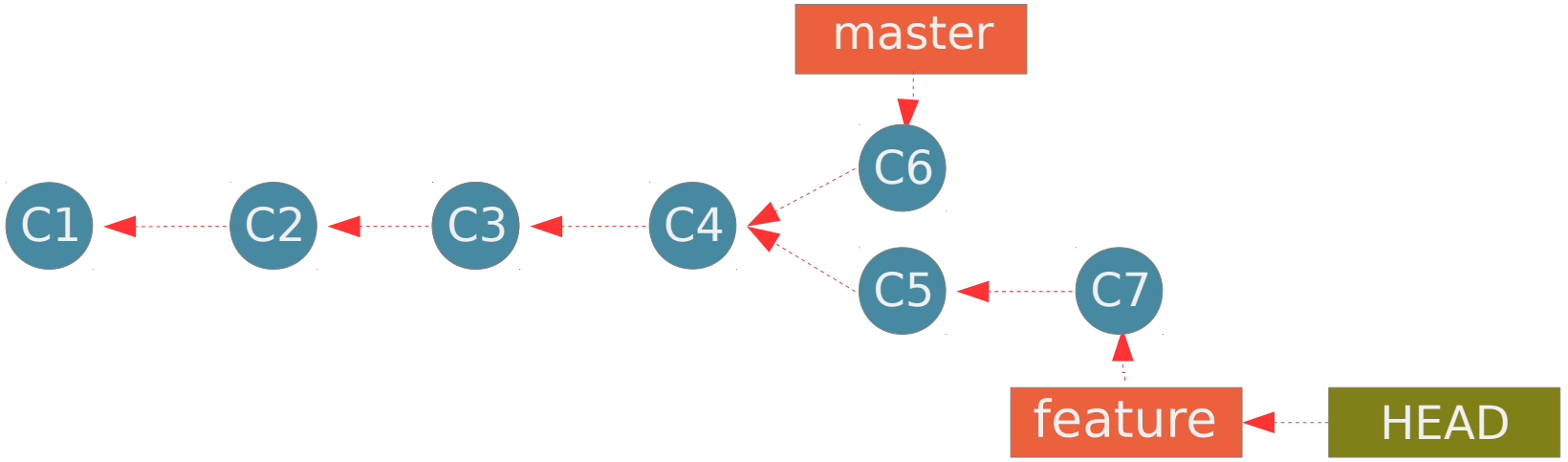
```
# Bug discovered
> git checkout master  # moves HEAD to master
> vim calc.py         # fix the bug
> git add calc.py
> git commit -m 'fix bug'
```



The bug is fixed and we can work on the new feature again!

Working with Branches

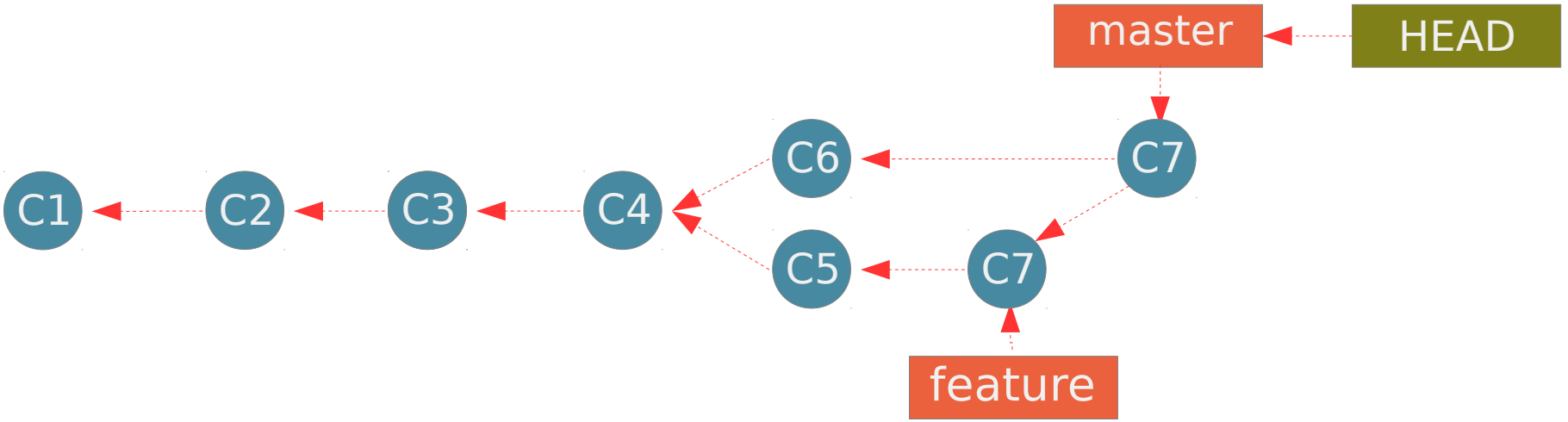
```
# Finish the new feature
> git checkout feature  # moves HEAD to feature
> vim test.py           # finish the feature
> git add test.py
> git commit -m 'Add new feature'
```



How can we get the new feature to the master branch?

Basic Merging

```
# Merge feature into master
> git checkout master
> git merge feature
Merge made by the 'recursive' strategy.
test.py |    3 +--
1 file changed, 1 insertion(+), 3 deletions(-)
> git branch -d feature   # delete branch
```



Basic Merge Conflicts

```
> git merge feature
Auto-merging calc.py
CONFLICT (content): Merge conflict in calc.py
Automatic merge failed;
fix conflicts and then commit the result.
```

```
<<<<<<< HEAD:calc.py
contact : email.support@freva.de
=======
please contact us at support@freva.de
>>>>>>> feature:calc.py
```

```
please contact us at email.support@github.com
```

```
# commit resolved merge conflict
> git commit
```