An Amazing Source Code Control Management System

Hagen Paul Pfeifer
hagen@jauu.net

Agenda

00 - Git Introduction

01 - Git 101

02 - Understanding Git

03 - Working In Teams

04 - Limitations

Practical Session

Questions? feel free to interrupt me at any time!

License

Git Introduction

What is Git?

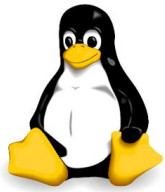
Source Code Management (SCM) System*

- Distributed - no central repository
- Strong support for non-linear development
- Fast and small
- Cryptographic authentication of history
- Open Source

But Git is more: Git initiated a change in the open source development process. Code changes became more transparent and thus software projects in general. Collaboration became more easy compared to the stone aged sourceforge days (cvs, svn).

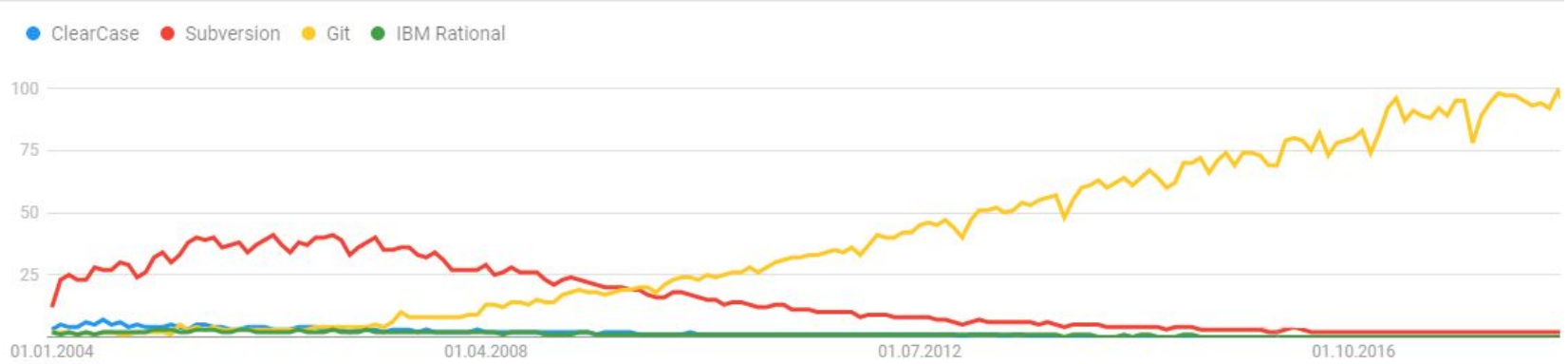
*this is what Wikipedia says about Git

Who is Using Git?



… and thousands companies more

Trend:



Git is already the industry standard - and loved by users!

Who?

Invented by ***Linus Torvalds***

“Nobody actually creates perfect code the first time around, except me. But there's only one of me."

Git - The Story Begins

Back in the days the Linux Kernel was developed by sending **diffs** to a **mailing list** and the developer apply **patches** to the local kernel directory

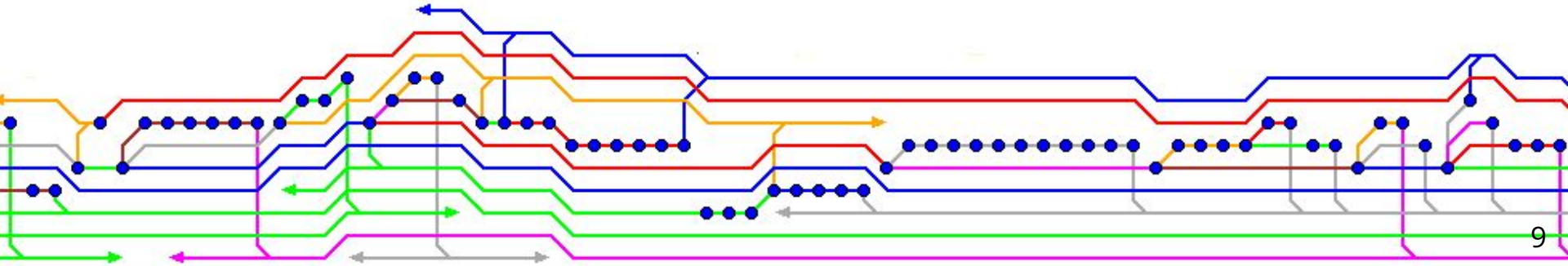
Starting with **2002** Linus switched to **BitKeeper**. A proprietary and distributed SCM Tool. Especially the **proprietary** aspects raised discussions

In 2005 Linus began to write a **git** - focusing on the kernel development process

The rest is history

Linus Goals

- Simple Design
- Speed - Everything is local
- Thousands of parallel branches - non-linear development
- Fully Distributed
- Large Repositories



Git Distribution

Git in the **beginning** was a set of **C programs** and **shell scripts** targeted for Linux. Nowadays a universal C library is also available used by upper level libraries for many languages and stand alone applications

The “official git client” is available for **Linux, macOS, Microsoft Windows, BSD** Distributions and **Solaris**

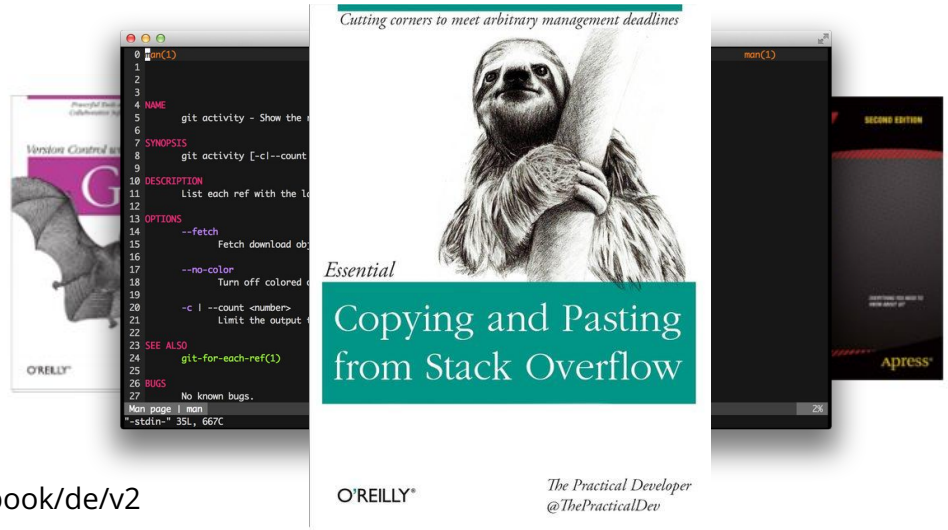
Clients also available for **Android, iOS** and other Operating Systems

Git Documentation

Before we Start

We probably cannot discuss all questions. But one advantage of Git is the great documentation:

- Many (free) books available (e.g. ProGit2)
- Git comes with high quality, up-to-date man pages
- Internet: extensive blog posts, youtube to stack overflow



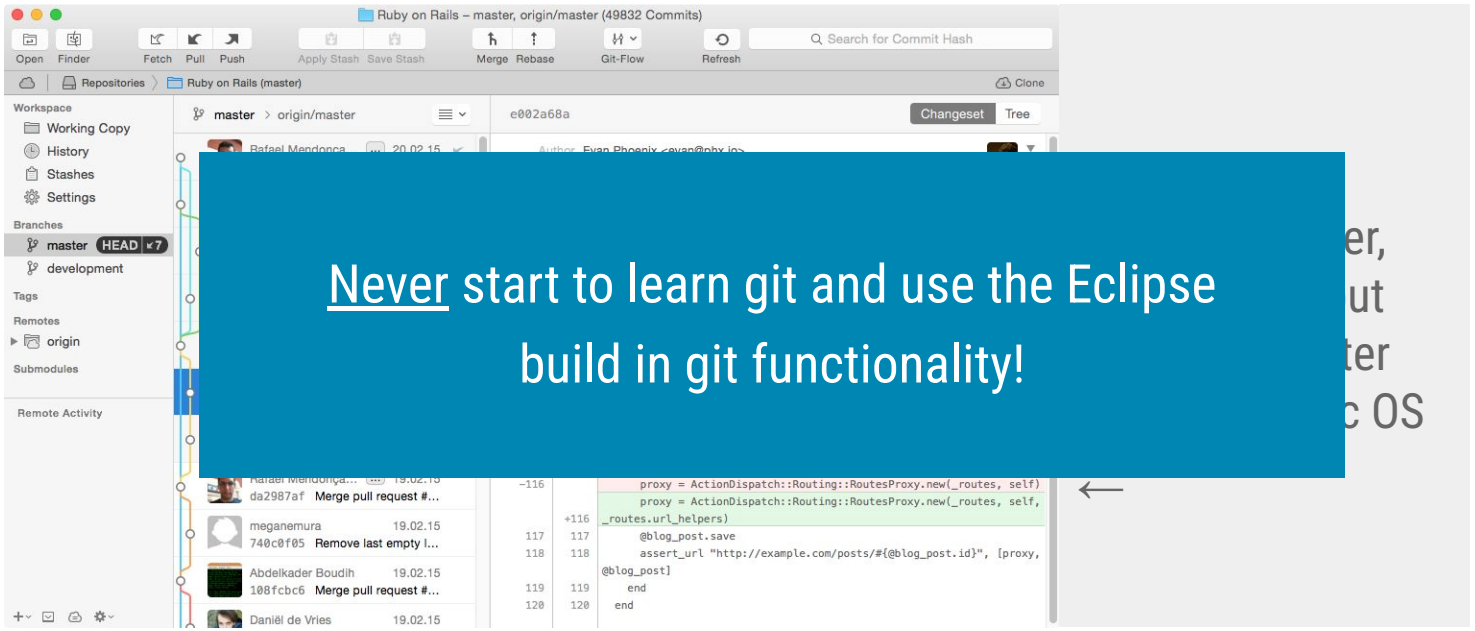
ProGit2 https://git-scm.com/book/de/v2

Git Command Line Interface

Before we Start

There are plenty of graphical user interfaces available.

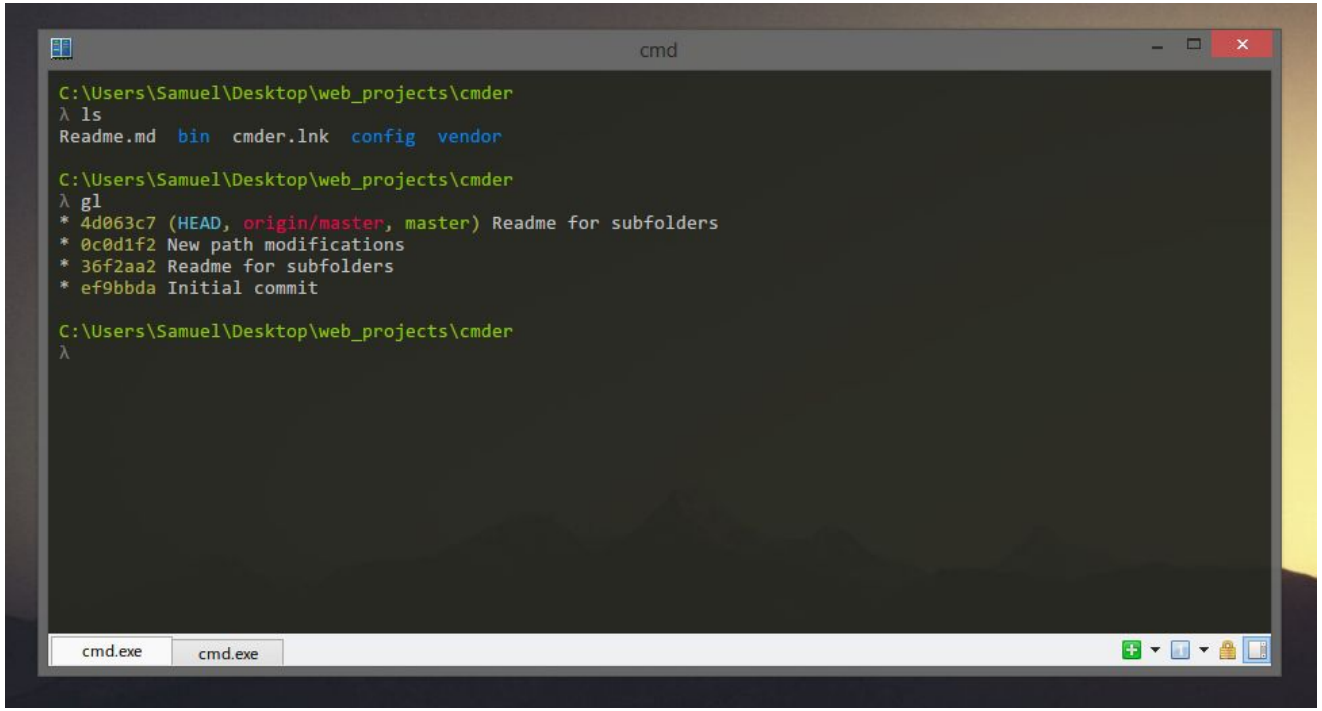
Beware: these GUI's simple wrap the command line functionality and this abstraction may hide information and lead to confusion - especially when you start using Git



Windows Terminal

Before we Start

One tip for Windows users: install **cmder**! You get a **resizable terminal** with UNIX **bash** support and a working **git** installation - installed in **2 minutes** without any hassles for **free**!



Before we Start

Your Git Client - Suggestions*

Windows	git (Cmder), SourceTree, TortoiseGit, Visual Studio
Linux	git, tig (curse), gitg
MacOS	git, GitTower

The great thing about Git: you are not committed to use one client - try some clients and **use the client that fits to your workflow**! Or use several clients simultaneously - if you are a crazy scientist why not!?

If in doubt: use the standard Git client - he is by the way

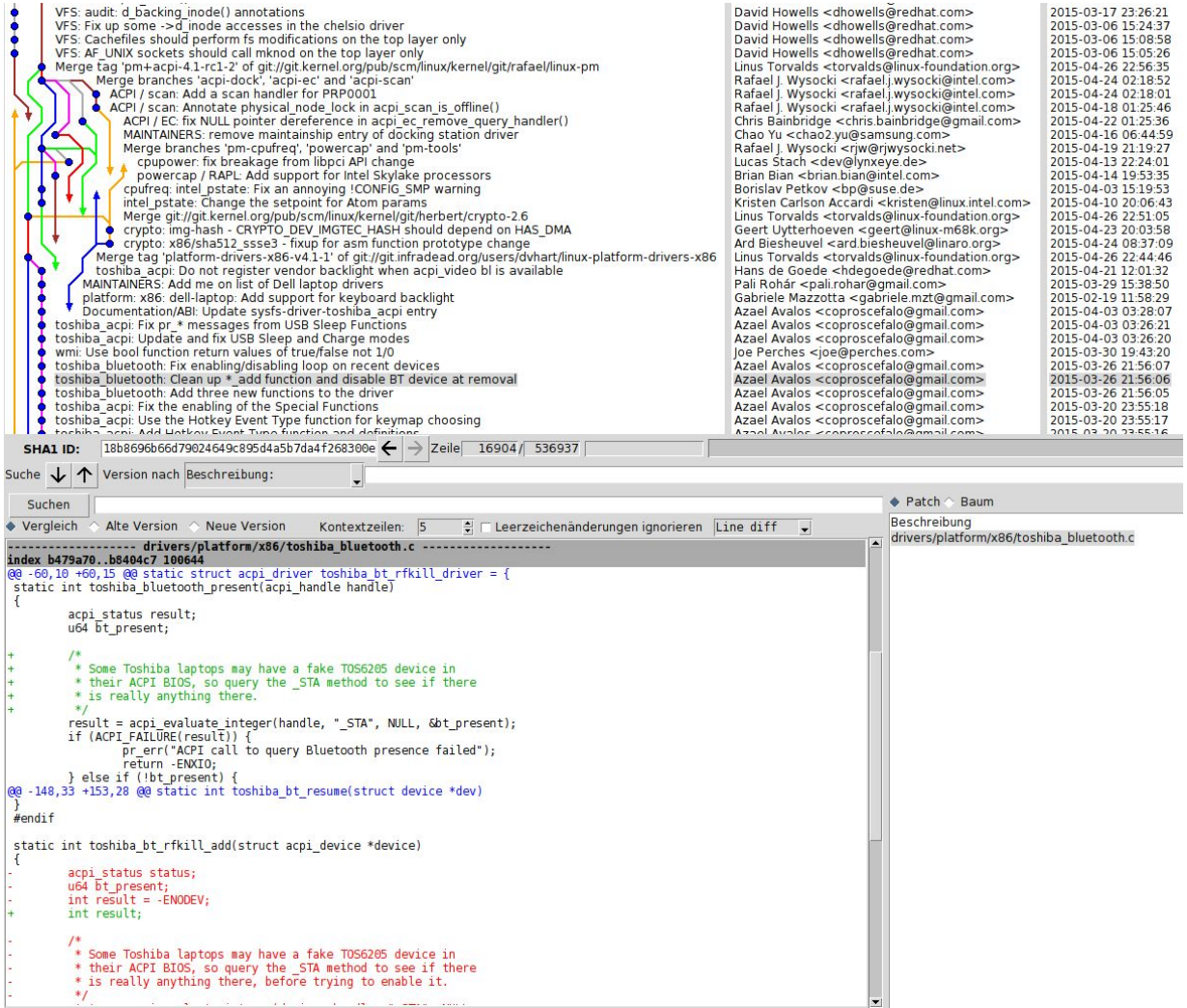
the most feature-rich client

*highly subjective, ignore me if you want - sorry if I blamed your beloved client!

Before we Start

Git GUI

Git comes already bundled with a graphical user interface: **gitk**



Git 101

101 → US American university course numbering systems used for an introductory course at a beginner's level

Don’t be a Nobody


```
$ git config --global user.name “John Doe”
$ git config --global user.email “johndoe@example.com”
```

Git Configuration

Edit Configuration

Plain Ini Formatted Text Files located in

- UNIX: `~/.gitconfig`
- Windows: `C:\Users\MyLogin\.gitconfig`

More comfortable:

```
$ git config --global --edit
```

Git Configuration

`git config --edit` will open your $editor, often `vim`

You can change the default editor by setting `core.editor`❤️:

```
$ git config --global core.editor "subl -n -w" → Sublime
$ git config --global core.editor "mate -w"    → Text Mate
$ git config --global core.editor "emacs"      → Emacs
```

❤️ works also for: atom, ultra edit, notepad++, bbedit, visual studio code, ...

Git Configuration

Notepad++ as 64 bit application for Windows users:

```
$ git config --global core.editor "'C:/Program
Files/Notepad++/notepad++.exe' -multiInst -notabbar
-nosession -noPlugin"     ← take care about all the quotes!
```

Git Configuration

Show actual Git configuration:

```
$ git config --global --list
user.name=John Doe
user.email=johndoe@example.com
core.editor=vim
color.ui=true                          ← command line color
```

We will come back later and show how you can tweak your Git configuration

Git Configuration

Proxy Configuration

To reach repositories behind a proxy:

```
$ git config --global http.proxy "http://U:P@example.com"
```

The First Repository

Two possibilities:

- Create a new repository
- Clone an existing repository

For now we create a new empty repository (within empty dir):

```
$ git init
Initialized empty Git repository in /tmp/foobar/.git/
```

ClearCase users: a repository is Git's VOB pendant

Git Working Stages

Overview

As with Subversion, CVS and other SCM tools you commit and that's it - with git you have a layer between your local changes and the final committed stage - the index

Work Space	Index	Repository
This is the place where file manipulations are recognized	A temporary scratch place for modifications shortly before commit (→)	**The** repository

Git Working Stages

The Index

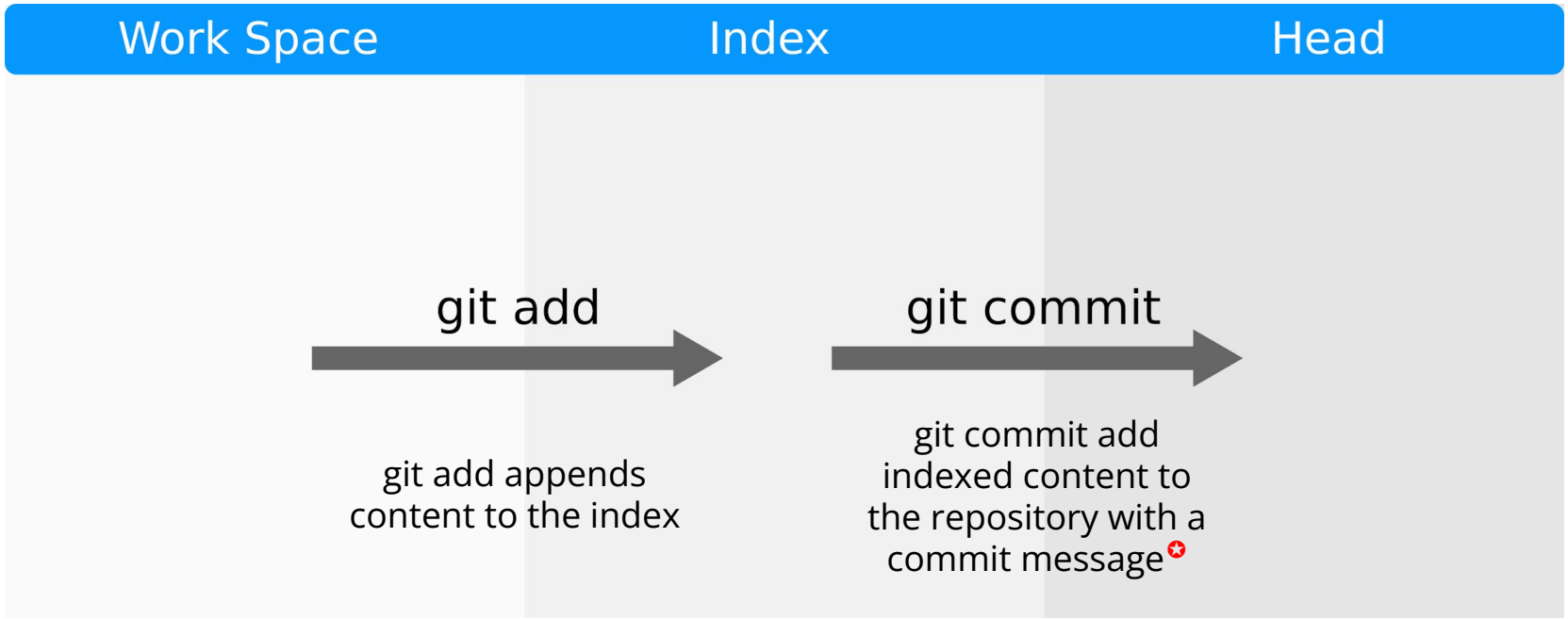
Consider the index as a **scratch place** where you can pimp your commit

The index helps you to **structure commits logically** - **not chronologically**

By the way: no need to checkout files a priori as required with other SCM systems: you decide what changes are going in (`git add`)

Git Working Stages

Traverse Stages



a commit message is required - but can be empty

Make the First Commit

Now add a simple file to the working directory, add it to the index and finally commit the file:

```
echo “Howdy World” > README
git add README
git commit -m “initial commit”
```

Git add is created for **selective commits** - group commits **semantically**, never group commits based on time (i.e. end of working day)

Depending on your work you will probably commit up to several times per hour

Commit Messages

Take your time - commit messages will help you in the future! Be **precise** and **detailed**: describe the **why** - **not how**!

Present tense! **Describe** the changes **for others** - not yourself! Possible side effects? ...

Formatting tip:

<topic>: summary

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body. The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by
  a single space, with blank lines in between, but conventions vary
  here
- Use a hanging indent
```

https://chris.beams.io/posts/git-commit/

Git Working Stages

Workspace and Index Status

Show what files in workspace are new, modified, deleted and what files are already indexed[1]:

```
$ git status
```

[1] this will be one of your most used git commands

Status Between Stages

git status

```
On branch master
Initial commit
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)
```

```
echo “Howdy World” > README
git add README
git commit -m “initial commit”
```

```
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: README
```

One More Time

```
$ git status --short                ← nothing, no output
$ echo “BarFoo” >> README           ← file is modified
$ echo “int main(){}”  > 1.c        ← file is created
$ echo “print(8**999)” > 2.py       ← file is created
$ git status --short
 M README                           ← modified, correct
?? 1.c                              ← unknown to git, not in index
?? 2.py                             ← unknown to git, not in index
$ git add README 1.c                ← add all, except 2.py
$ git status --short
M  README                           ← modification in index
A  1.c                              ← addition in index
?? 2.py                             ← still unknown to git
```

One More Time

```
$ echo “qux” >> README      ← modify again
$ git status --short
MM  README                  ← modification in index & working dir
A   1.c                     ← no change
??  2.py                    ← same same
$ git commit -m “commit #2”
$ git status --short
 M  README                  ← index change committed, rest
remains
??  2.py                    ← still unknown
$ git add README 2.py
$ git commit -m “commit #3”
$ git status --short        ← nothing
```

Repository Log

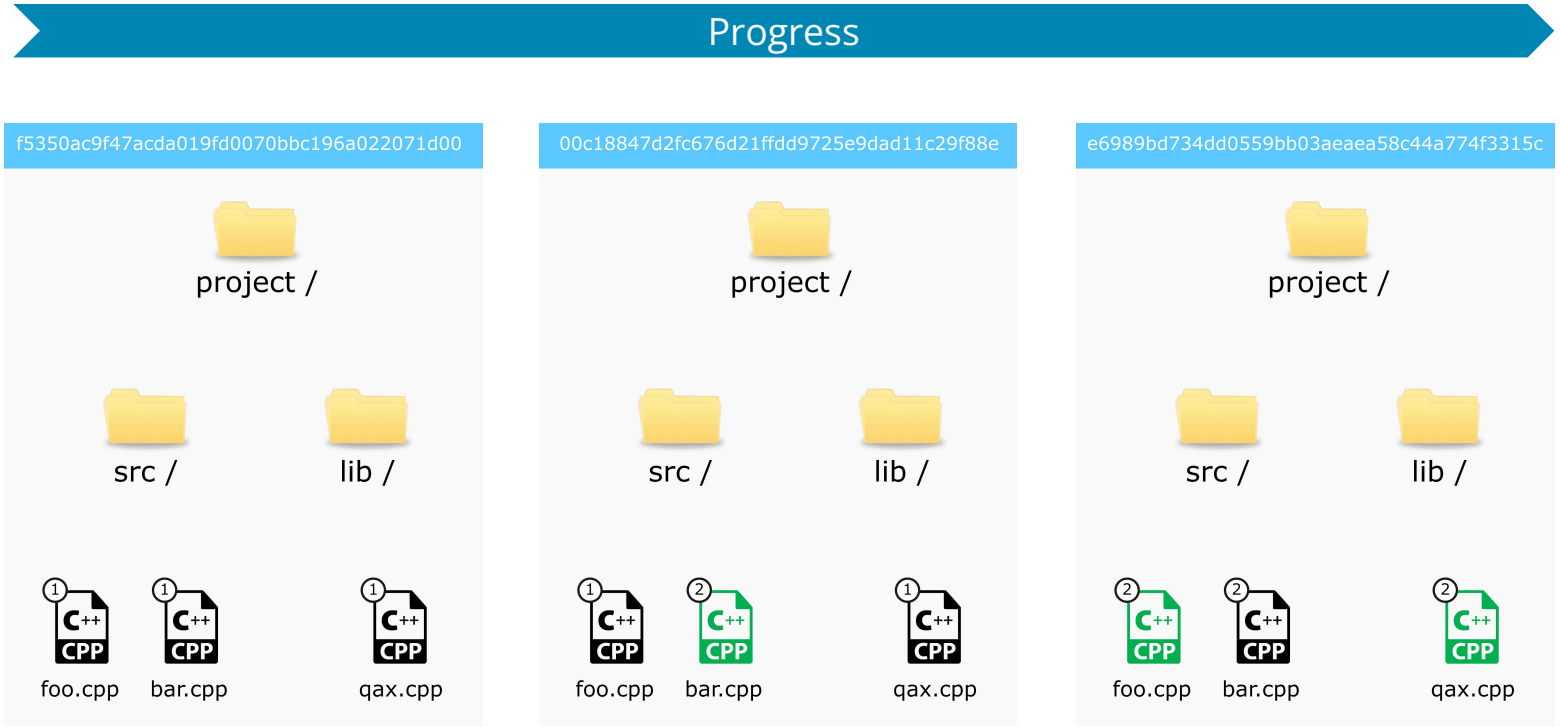
We did the first commit - where is it? `git status` does not show anything!

This is where `git log` comes into play:

```
commit be646c5781f9a0d877a89ba40a85dd7001198554
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:01:32 2016 +0100

    commit #3

commit 208e6ba0559e6a2cc5dd45acc55d973a48b436ec
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:01:32 2016 +0100

    commit #2

commit a8c52c86b6e10c4253c72f9953ba242f6b50621c
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:01:32 2016 +0100

    initial commit
```

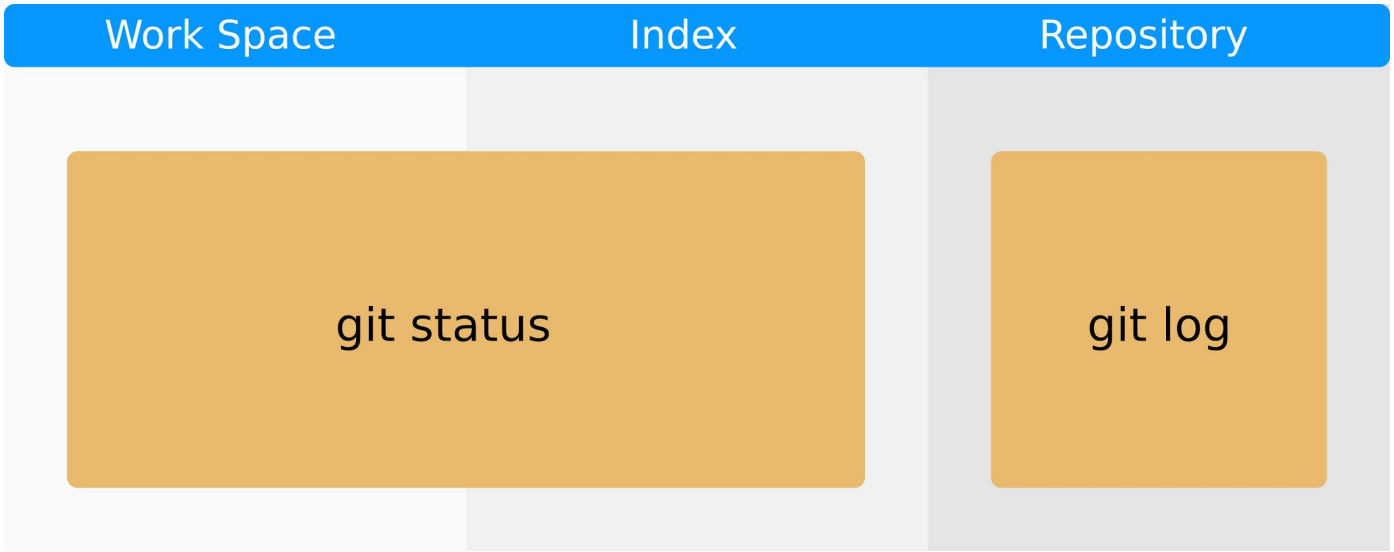
Git Commit ID

Each Commit is a unique Tag/Label for the whole repository



Repository Log

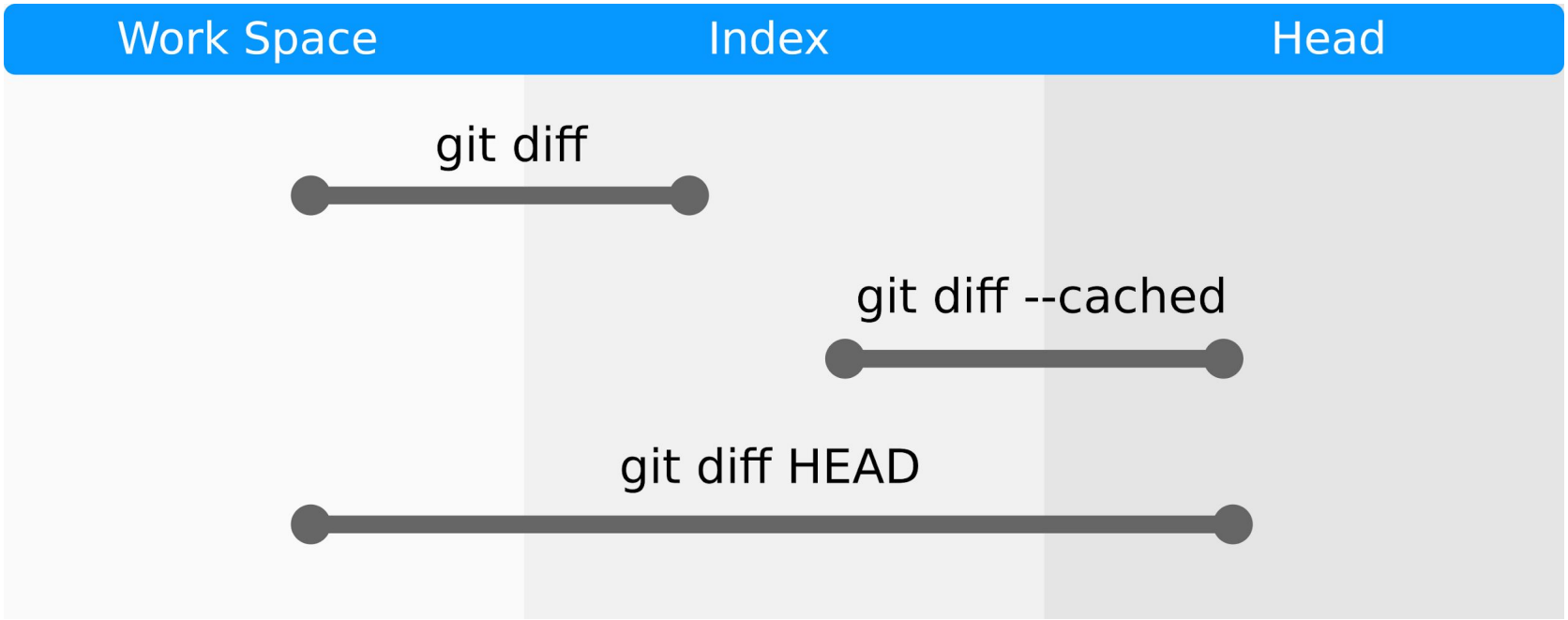
Status vs. Log



Git Working Stages

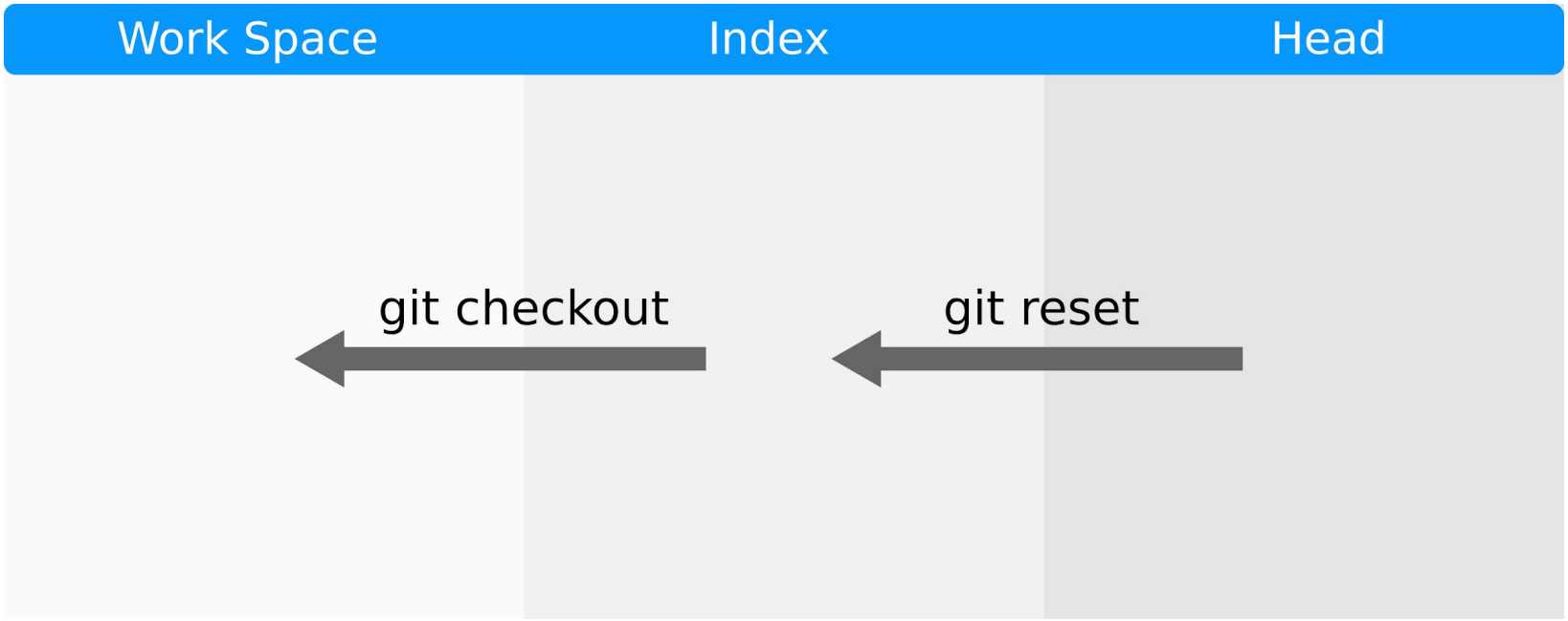
Diff Stages

Show line-by-line diff equal to `diff(1)`



Git Working Stages

Traverse Stages - Яeverse Direction



Summary - Git Starter Kit

`git init` to create repository

`git status` to show stage information (working dir, index)

`git diff` to see line-by-line difference between stages

`git add` to add files to the index

`git commit` to transfer indexed data into the repository

`git log` to browse repository information

Fingers crossed: all commands are more powerful and flexible - we will come back later to each command in detail

The Beauty of Git - The Core

The Beauty of Git - The Core

After this section you will be able to **write Git itself**. If you are an average Python Programmer it takes you **less then a day**. To "*fix the remaining 5%*" you will probably need 101,4 person years.

Let's go!

```
ansic:       169299 (44.92%)
sh:          148094 (39.30%)
perl:         29428 (7.81%)
tcl:          21431 (5.69%)
python:        6562 (1.74%)
lisp:          1786 (0.47%)
php:            120 (0.03%)
asm:             98 (0.03%)
csh:             45 (0.01%)
```

The Beauty of Git - The Core

```
$ git init
$ tree .git                                    (ignoring .git/hooks directory)
.git
├── branches
├── config
├── description
├── HEAD
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

The Beauty of Git - The Core

We add a file to the repository and make the file known to Git

```
$ echo "foo" > bar.txt
$ git add bar.txt
```

What happens to the `.git` directory?

The Beauty of Git - The Core

```
$ tree
bar.txt
.git
├── branches
├── config
├── description
├── HEAD
├── index
├── info
│   └── exclude
├── objects
│   ├── 25
│   │   └── 7cc5642cb1a054f08cc83f2d943e56fd3ebe99
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

257cc56.. is the SHA1 sum of the file content

`git hash-object -w bar.txt` → **257cc56...**

The Beauty of Git - The Core

What is in the file?

```
$ cat .git/objects/25/7cc5642cb1a054f08cc83f2d943e56fd3ebe99
xKÊÉOR0aHËÏçUB%
```

It’s simple compressed, lets uncompress the file:

```
$ alias deflate="perl -MCompress::Zlib -e 'undef $/; print uncompress(<>)'"
$ deflate .git/objects/25/7cc5642cb1a054f08cc83f2d943e56fd3ebe99
blob 4foo
```

blob: type of data

4: size of data in bytes

foo: the content itself

The Beauty of Git - The Core

Now commit the file and look how a commit is handled:

```
$ git commit -m "initial commit"
bar.txt
.git
├── branches
├── config
├── description
├── HEAD
├── index
├── info
│   └── exclude
├── objects
│   ├── 25
│   │   └── 7cc5642cb1a054f08cc83f2d943e56fd3ebe99     ← the file content (blob)
│   ├── 4c
│   │   └── 74dfe6753b8290f893fec1ef337b9281211c33     ← ?
│   ├── ee
│   │   └── ec1327b15c7313c2a98a2743c56ec5858364d7     ← ?
│   ├── info
│   └── pack
[...]
```

The Beauty of Git - The Core

Object Types

Of which type is a Git object?

```
$ git cat-file -t 257cc..
blob ← we already knew this
```

Let’s look at the two new objects:

```
$ git cat-file -t eeec1..
Tree
$ git cat-file -t 4c74d..
Commit
```

The Beauty of Git - The Core

Objects Content

What content has a git object?

```
$ git cat-file -p 257cc..
foo
```

Let's look at the two new objects:

```
$ git cat-file -p eeec1..
100644 blob 257cc5642cb1a054f08cc83f2d943e56fd3ebe99   bar.txt

$ git cat-file -p 4c74d..
tree eeec1327b15c7313c2a98a2743c56ec5858364d7
author Hagen Paul Pfeifer <hagen@jauu.net> 1457792159 +0100
committer Hagen Paul Pfeifer <hagen@jauu.net> 1457792159 +0100

initial commit
```

The Beauty of Git - The Core

One More Time

Adding more files, add, show ID and commit:

```
$ echo "alice" > 1.txt
$ echo "bob"   > 2.txt
$ git add 1.txt 2.txt
$ echo "alice" | git hash-object -w --stdin
c9fc40bfbf49f914afa3dd2326af68f58e484948
$ echo "bob" | git hash-object -w --stdin
696fb6baa5ce30099c89066294e5973ee42a1899
$ git commit -m "second commit"
```

The Beauty of Git - The Core

```
$ git commit -m "second commit"

1.txt
2.txt
bar.txt
.git
├── branches
├── config
├── description
├── index
├── info
│   └── exclude
├── objects
│   ├── 25
│   │   ├── 292dac2fbfc2631c8ae2f50c516404eb611146    ?
│   │   └── 7cc5642cb1a054f08cc83f2d943e56fd3ebe99    ← file content (first commit)
│   ├── 4c
│   │   └── 74dfe6753b8290f893fec1ef337b9281211c33    ← first commit
│   ├── 69
│   │   └── 6fb6baa5ce30099c89066294e5973ee42a1899    ← new file content ("bob")
│   ├── c9
│   │   └── fc40bfbf49f914afa3dd2326af68f58e484948    ← new file content ("alice")
│   ├── ee
│   │   └── ec1327b15c7313c2a98a2743c56ec5858364d7    ← tree object to first commit
│   ├── f8
│   │   └── 0c8b8b833ea94366a487ffc59ceff5fc59c47d    ?
```

The Beauty of Git - The Core

Nesting Commit Objects

```
$ git cat-file -p f80c8b8..
100644 blob c9fc40bfbf49f914afa3dd2326af68f58e484948   1.txt
100644 blob 696fb6baa5ce30099c89066294e5973ee42a1899   2.txt
100644 blob 257cc5642cb1a054f08cc83f2d943e56fd3ebe99   bar.txt
```

```
$ git cat-file -p 25292d..
tree f80c8b8b833ea94366a487ffc59ceff5fc59c47d
parent 4c74dfe6753b8290f893fec1ef337b9281211c33
author Hagen Paul Pfeifer <hagen@jauu.net> 1457793944 +0100
committer Hagen Paul Pfeifer <hagen@jauu.net> 1457793944 +0100

second commit
```

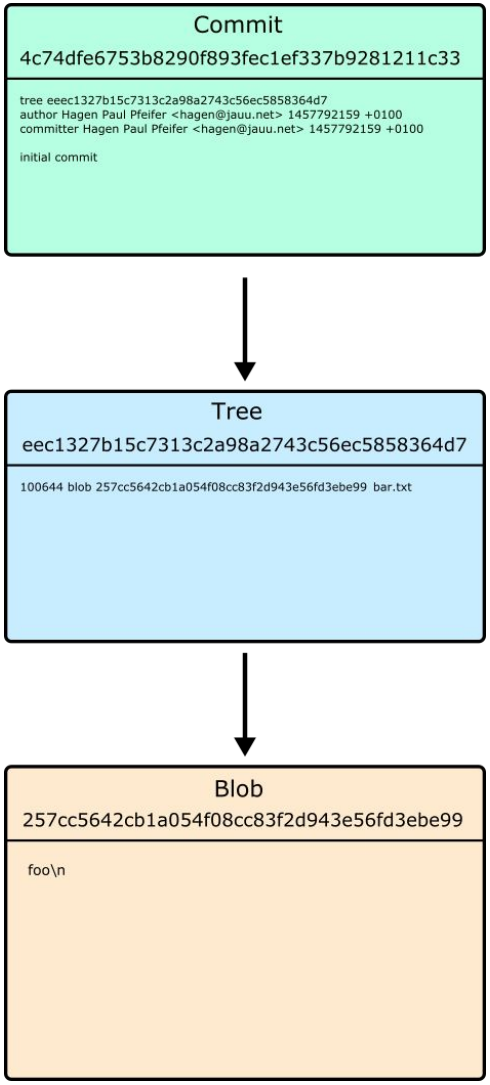
The Beauty of Git - The Core

A **merge** is simple a **commit** with **more** than one **parent**

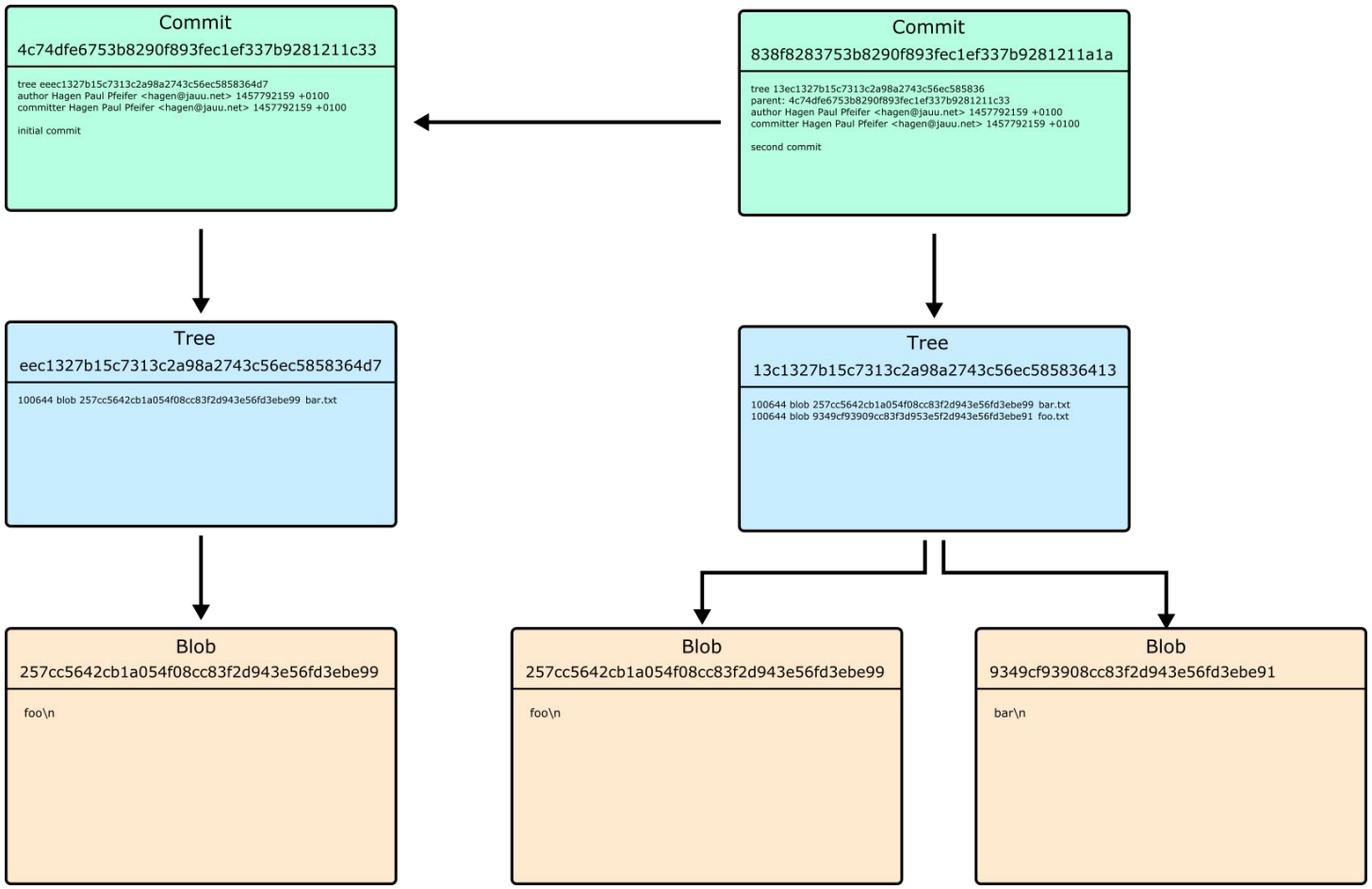
Files in **subdirectories** are represented by a **tree object**:

```
$ cd /usr/src/linux; git cat-file -p HEAD\^{tree}
100644 blob 4ad4a98b884b9c857166d993d167fe4953210201   .gitignore
100644 blob b4091b7a78fe11ccd0e5f44f0703ace69dc09707   .mailmap
100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c   COPYING
100644 blob 1d616640bbf64a960a881c56ed6f5b0a5bd17b28   CREDITS
040000 tree 9ee038f3d013db460cc4d0e26501116e82d73800   Documentation
100644 blob f55cefd9bf29a2fa2746f7039f2481dfdceadfc7   Kbuild
100644 blob c13f48d65898487105f0193667648382c90d0eda   Kconfig
100644 blob a9ae6c105520011994801168a7841b4d713b716e   MAINTAINERS
100644 blob afabc44a349b7b31a2028660e7e61b134556a675   Makefile
100644 blob 69c68fb4a10908a4f8d1986c6fdad2ec7b3c397b   README
100644 blob 0cb8cdfa63bcf0837582d5db2a443d511e5ddf48   REPORTING-BUGS
040000 tree 3a980fb70ab743799b20a955b1b144810f6f16d1   arch
040000 tree 64d25fbccebbaf1ab7552210ae7fcaa6c3d245fa   block
```

The Beauty of Git - The Core

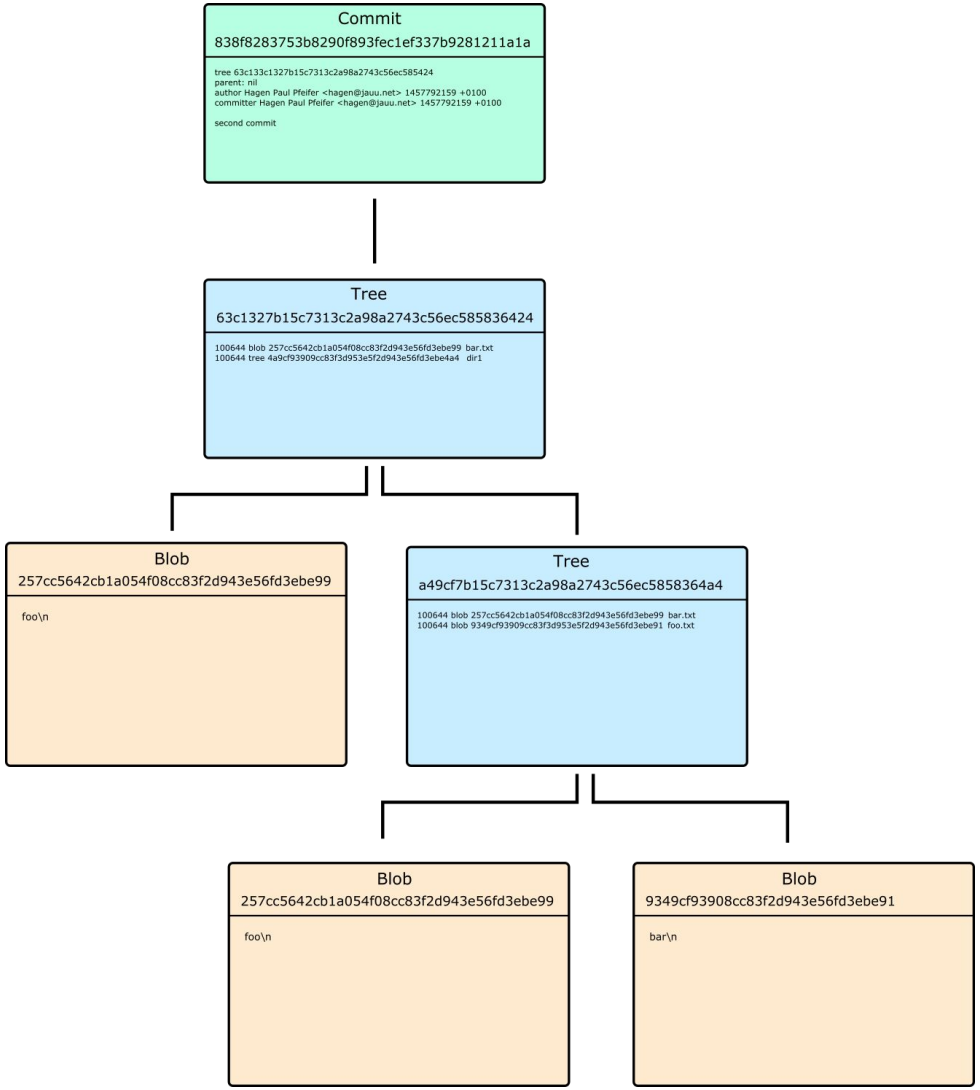


The Beauty of Git - The Core



The Beauty of Git - The Core

Files in subdirectories are represented by a tree object:



The Beauty of Git - The Core

A **branch** is just a simple file, the string content is a **reference** to **a commit id**! They are really lightweight!

```
$ cat .git/HEAD
ref: refs/heads/master
$ cat .git/refs/heads/master
25292dac2fbfc2631c8ae2f50c516404eb611146
```

At the end: branch management is just pointer management - pointing branch names to commit id’s

The Beauty of Git - The Core

The Offspring - Firsts Days of Git

You can still checkout the first days of Git development - as of commit `e83c51633` Git hosts itself!

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ git checkout -b day0 e83c51633
$ ls
cache.h cat-file.c commit-tree.c init-db.c Makefile read-cache.c
README read-tree.c show-diff.c update-cache.c write-tree.c
$ cat *.{c,h} | wc -l
1036
```

Practical Session - I

Credentials: add name and email to your git configuration file

Create git directory (all repositories will be placed here for now):

```
mkdir git;cd git
```

Create pseudo project with example files:

```
mkdir project; cd project; touch a.c b.c c.c
```

Create repository

```
git init
```

Show what happen in the git repository, do it periodically:

```
git status
```

Add files to index with

```
git add <filename> <filename-2> ..
```

Commit index to repository

```
git commit
```

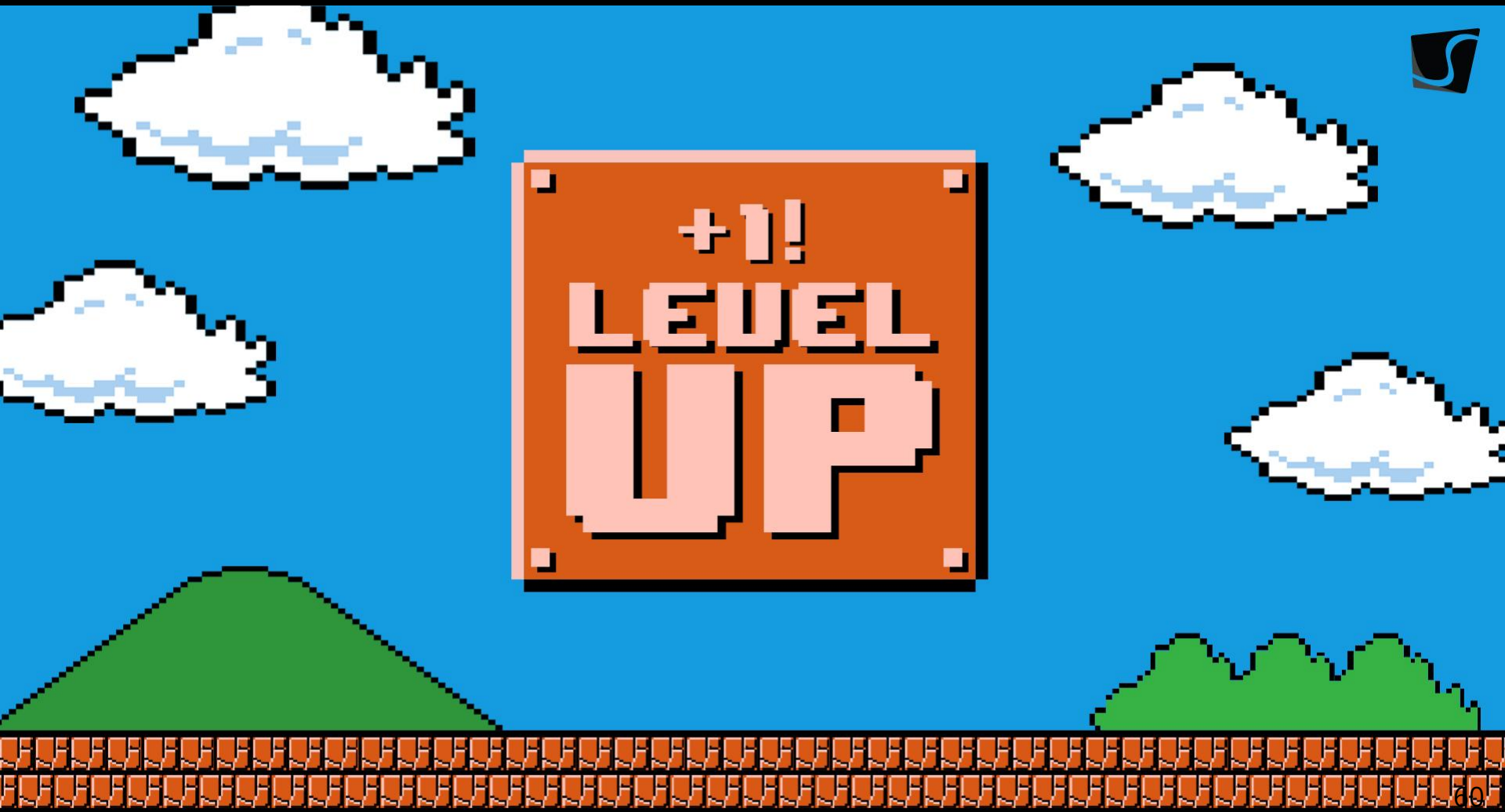
Modify files, show difference, add to index and commit again

```
echo foo >> a.c
git diff
git add a.c
git commit
```

Show history

```
git log
```

Understanding Git



Git Concepts

Before we go on we discuss one fundamental concept of Git:

- **Commit ID**

Git Commit ID

The commit ID:

2482abb93ebf7bfbf85965ca907f0058ff968c59

5bacd7805ab4f07a69c7ef4b1d45ce553d2b1c3a

e2e9b6541dd4b31848079da80fe2253daaafb549

Describes a particular commit **exactly!**

- Tree, Date, Committer, History, ..

Git does **not** work with **revision numbers**, incrementing with each commit known from CVS or Subversion

Git uses a **secure** (cryptographic) hash scheme based on **content** and **history**

Git Commit ID

Some interesting characteristic can be derived from the circumstance how the ID is calculated:

- If the content changes - one bit is already sufficient - the ID will be **completely different**. For the affected and all subsequent commits

- If a person in a parallel universe writes the **same code**, at the same time with an identical name: the **commit IDs** will be **100% identical** - there will be **no difference** in Git - because there is no difference!

Git Commit ID

Secure Hash Algorithm 1

Commit ID: SHA1 sum, 40 hex characters, 20 bytes

Hash Collisions:

- 10^{48} different hashes possible
- Birthday attack (50% collision probability): 2^{80} tries. More than 1000 times the number of grain of sand on earth!

SHA1 is a secure hash function - it is practical **impossible to recreate the input data** from the hash value

Shorten the **ID** is legitimate and safe:

`d4fc1a460f3017e958e6a8ea560ea0afd91bf6fe` → `d4fc1a460f30`

Mar 20, 2017 GitHub.com implements SHA-1 collision detection algorithm

Cloning

Clone Repositories

git clone

git init to create new empty repositories

Often you join an existing project: you will **clone** the repository:

```
$ git clone git@git.jauu.net:repos/foobar.git
$ git clone https://github.com/Microsoft/vscode.git
$ git clone git://git.kernel.org/linux/kernel/git/torvalds/linux.git
```

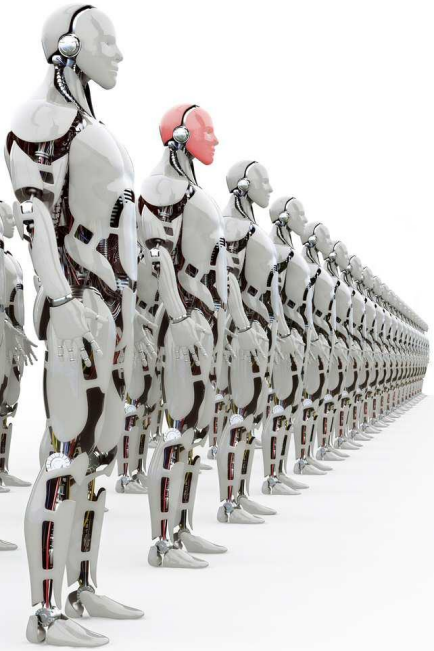
No matter what protocol is used - at the end a new directory is presented

Clone Repositories

Local Clones

The repository is **local**. If you clone a repository you download the **complete repository** - not just the last snapshot as with Subversion, CVS and others

You can work from home or on a plane without problems. **No network connectivity is required** - for all* operations!



Server outage in your central IT department? Relax! Even if things start to burn in the datacenter - your project is save! **Every developer has a pixel-perfect copy of the whole repository**!

* with the exception of sync operations (pull, push)

Clone Repositories

Disk Consumption

Storing the full history at each repository consumes memory - no doubt

Doing this in this way was a willfully **design decision**! Even more: Git does **not** store **deltas** between changes!

Git uses a **powerful compression** mechanism to reduce the amount of space to a minimum

There are many strategies to cope with huge repositories: split in several independent repositories, use Git LFS, ...

Linux Kernel 6.16: 5.9 GiB

Clone Repositories

Protocols

Git supports a lot of **protocols**: git, ssh, file, http and https

In the beginning HTTP(s) was slow and pushing not possible - today **http** is probably the **best choice**

- Less problems in **proxied, firewalled environments** (company, hotel, ...)
- Prefer HTTP**S** for private sensitive repositories

File scheme useful if you clone from network file systems (e.g. FTP, CIFS) or from local storage (avoids starting a local server)

```
$ git clone /home/pfeifer/src/foobar.git
```

Repositories via HTTPS

Self Signed Certificates

Self signed certificates are not trusted by default (surprise, surprise):

```
$ git push -u origin master
fatal: unable to access 'https://codedev.rsint.net/training/git.git/':
SSL certificate problem: unable to get local issuer certificate
```

"Workaround": disable SSL verification on repository basis (local) or for all repositories (global)

```
$ git config --global http.sslVerify false
```

Branching

Branching

Overview

We are now in the ability to **add content** to the repository - linear life is great!

But how to manage **multiple features** at the same time? Subversion users normally do not care: every feature is dumped to the one branch. Sometimes for larger features you maintain two or more branches.

Git is different - you use **topic branches**! You can drive your SVN style development - but the intuitive way is to differentiate on a branch basis.

Branches are an intuitive way to **differentiate different topics** like features, fixes or whatever.

Branching

Overview

Branches are really **lightweight**! It's **easy** to **create** branches, it is **easy** to **switch** between branches and it is **easy** to reorder and **merge** branches!

If you create a repository and commit objects Git already created a branch: the `master` branch (which is not special in any way)!

```
$ git branch
* master
```

Branching

Create a new Branch

```
$ git branch fix-python-example
$ git branch
  fix-python-example
* master
```

`git branch` creates a new branch, the active branch is still the `master` branch!

Branching

Switch branches

Switch between branches

```
$ git checkout fix-python-example
Switched to branch 'fix-python-example'
$ git branch
* fix-python-example
  master
```

Shortcut for create and checkout the new branch:

```
$ git checkout -b fix-python-example
```

Branching

Faster Switch Branches

Sometimes you **switch** between the **same branches** several times - Git provides a shortcut to reduce typing:

```
$ git checkout -
Switched to branch 'fix-python-example'
$ git checkout -
Switched to branch 'master'
$ git checkout -
Switched to branch 'fix-python-example'
[...]
```

Branching

Where am I

Newly created branches start at the exact position you branched!

How to see where my branch points to?

```
$ git log --decorate
commit 7ca490bbc024626ef28a184f88965b59bed8 (HEAD -> fix-python-example, master)
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:04:09 2016 +0100

    commit #3

commit 67e36030952d0cbe9bddfa4a1af0fe6f04d0
Author: Hagen Paul Pfeifer <hagen@jauu.net>
[...]
```

With Git 2.13 (Q2 2017), no need for configuration: `--decorate` is the default!

Branching

Branch Origin

Create two commits on our new branch

```
$ echo “print(1**666)” > 2.py
$ git add 2.py
$ git commit -m “commit #4”
$ echo “for i in range(51):print(2*i)” > 3.py
$ git add 3.py
$ git commit -m “commit #5”
```

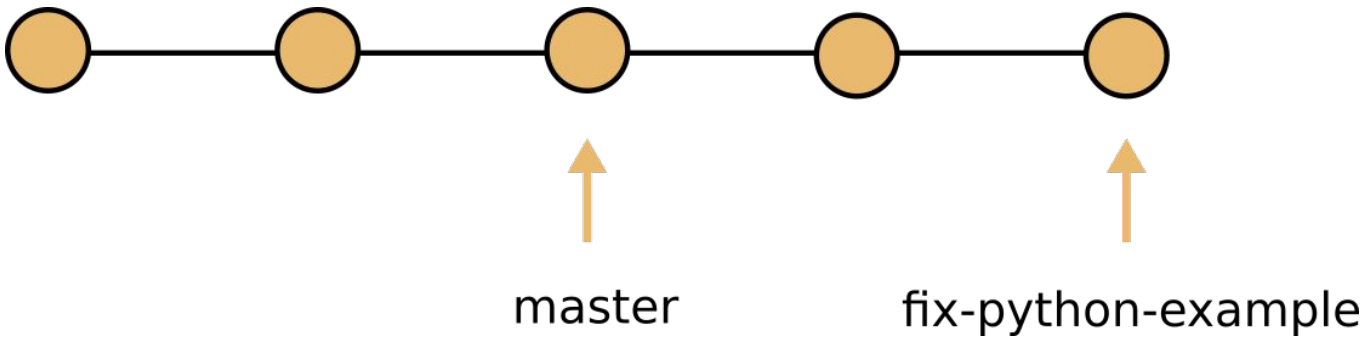
Branching

Branch Origin

```
$ git log --decorate
commit 7b0652211e15810e82d6d45a509f56ebbed73383 (HEAD -> fix-python-example)
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 22:02:37 2016 +0100

    commit #5

commit fd1b245ca3b38eebd08095a664ac6d6454099b5d
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 22:02:37 2016 +0100

    commit #4

commit 7ca490bbc024626ef28a184f88965b59be6bbed8 (master)
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:04:09 2016 +0100

    commit #3
[...]
```

Branching

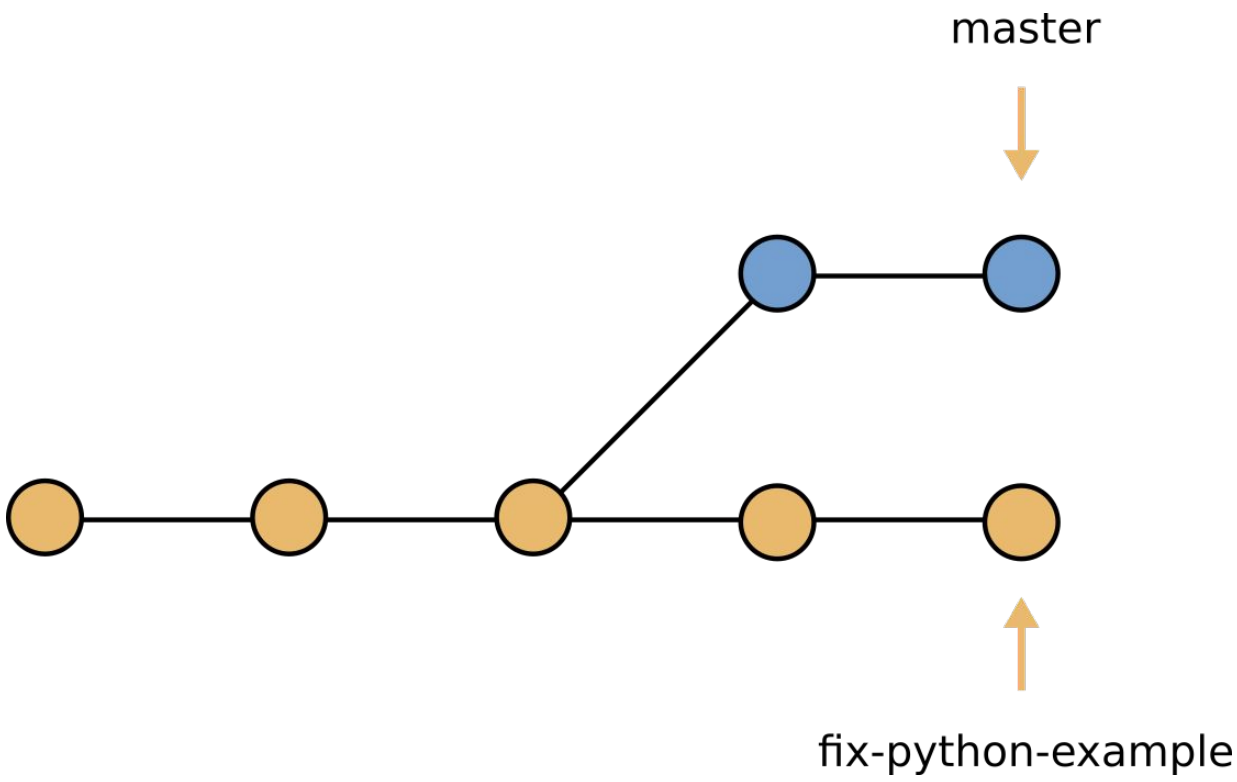
Overview



Branching

Commit to an Existing Branch

Checkout **master** and **add two commits** to the master branch:



Branching

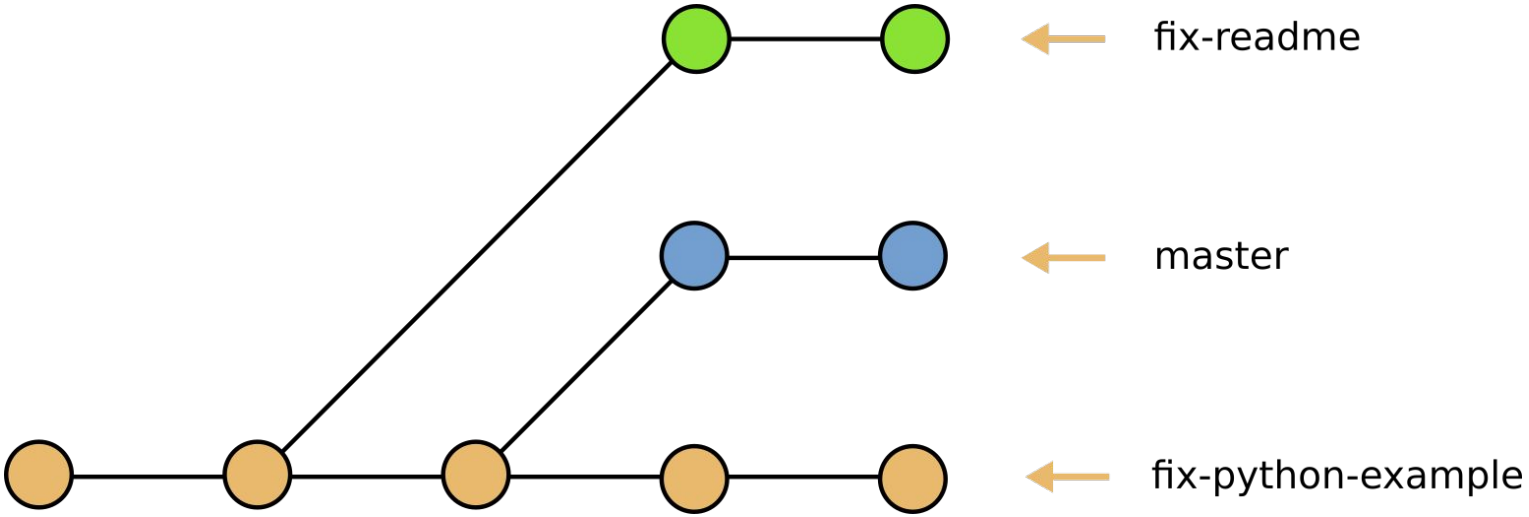
Create a Branch

Now create a new branch, starting at second commit and add two commits:

```
$ git checkout -b fix-readme 70b0ed2
$ git branch -v
  fix-python-example 7b06522 commit #5
* fix-readme         70b0ed2 commit #2
  master             697b467 commit #3
$ echo “Foo” >> README
$ git commit --all --message "commit #6"
$ echo “Bar” >> README
$ git commit -a -m "commit #7"
```

Branching

Branch Origin



Branching

Delete a Branch

Delete[1] a branch:

```
$ git branch --delete fix-readme
$ git branch -d fix-readme                    ← shorter version
```

Git will not delete a branch if the commits are not merged! To delete the branch anyway:

```
$ git branch --delete --force fix-readme
$ git branch -D fix-readme                  ← shorter version
```

[1] in section *Advanced Git* we will see how we can recover deleted branches

Branching

HEAD

HEAD is a canonical name of the active branch

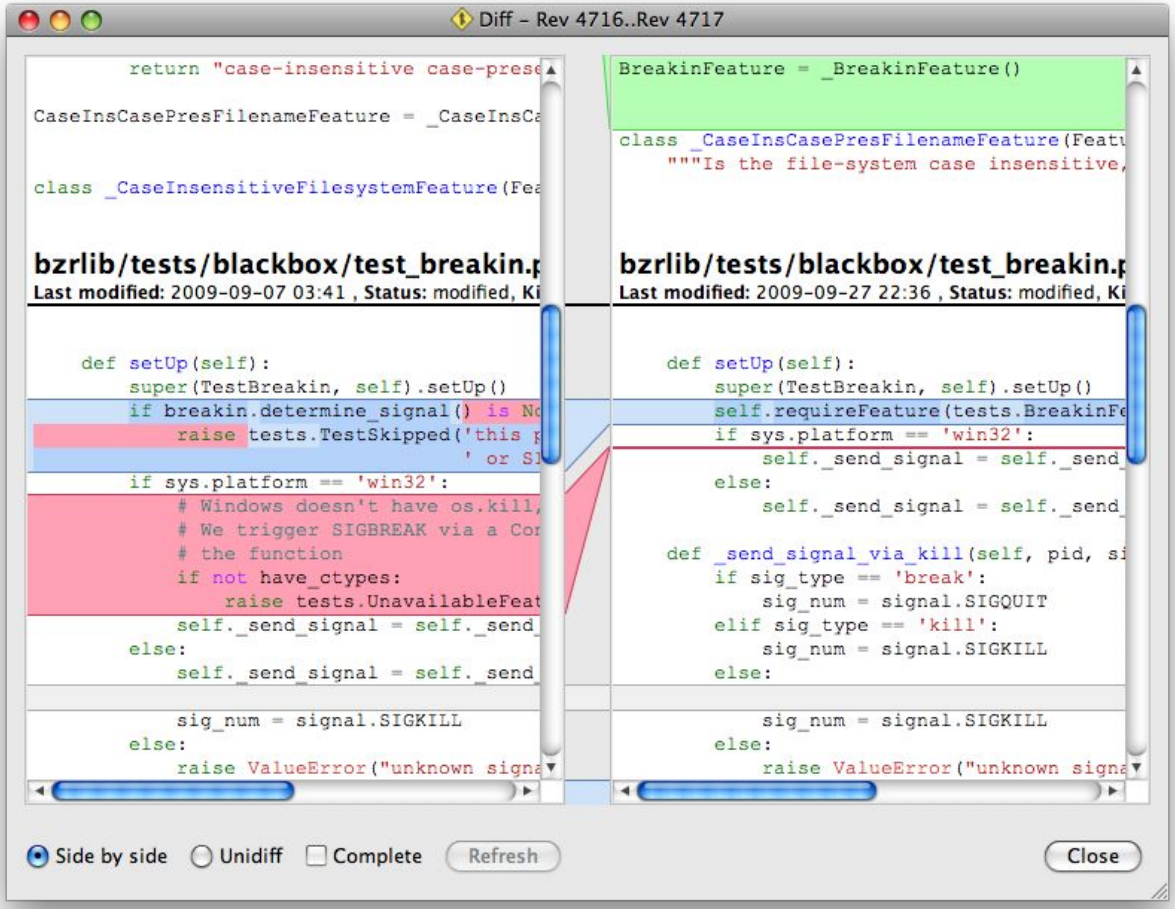
```
$ cat .git/HEAD
ref: refs/heads/fix-readme
```

Some common and recurring operations don't require the exact branch name. For example:

- You want to know the previous commit, `HEAD~1`

Starting with Git 1.8.5 there is an alias to `HEAD`: `@`

Differences



Differences

Between Branches - Range Notation

Show the diff between the tips of the two branches:

```
$ git diff branch-a..branch-b
```

```
$ git diff fix-readme..master
diff --git a/2.py b/2.py
new file mode 100644
index 0000000..e6d5752
--- /dev/null
+++ b/2.py
@@ -0,0 +1 @@
+print(8**999)
diff --git a/README b/README
index 0579475..d425b7d 100644
--- a/README
+++ b/README
@@ -1,3 +1,3 @@
 Howdy World
 BarFoo
-foo
+qux
```

```
$ git diff master..fix-readme
diff --git a/2.py b/2.py
deleted file mode 100644
index e6d5752..0000000
--- a/2.py
+++ /dev/null
@@ -1 +0,0 @@
-print(8**999)
diff --git a/README b/README
index d425b7d..0579475 100644
--- a/README
+++ b/README
@@ -1,3 +1,3 @@
 Howdy World
 BarFoo
-qux
+foo
```

Differences

Between Branches - Range Notation

Show required changes to master to get fix-readme content

```
$ git diff master..fix-readme    # identical: git diff master fix-readme
```

Show required changes from previous master to current master (e.g. last commit) in relative notation:

```
$ git diff master~1..master
```

Identical operation - absolute notation

```
$ git diff 74a3dcfbc..697b4674b
```

Differences

Between Branches - Simplified

Required changes to branch fix-readme to become HEAD

```
$ git diff fix-readme          # similar to git diff fix-readme..HEAD
```

Simplified reverse operation: “what happened on branch `fix-readme`, what is added, removed or changed”

```
$ git diff ..fix-readme        # identical to git diff HEAD..fix-readme
```

Review Tip:

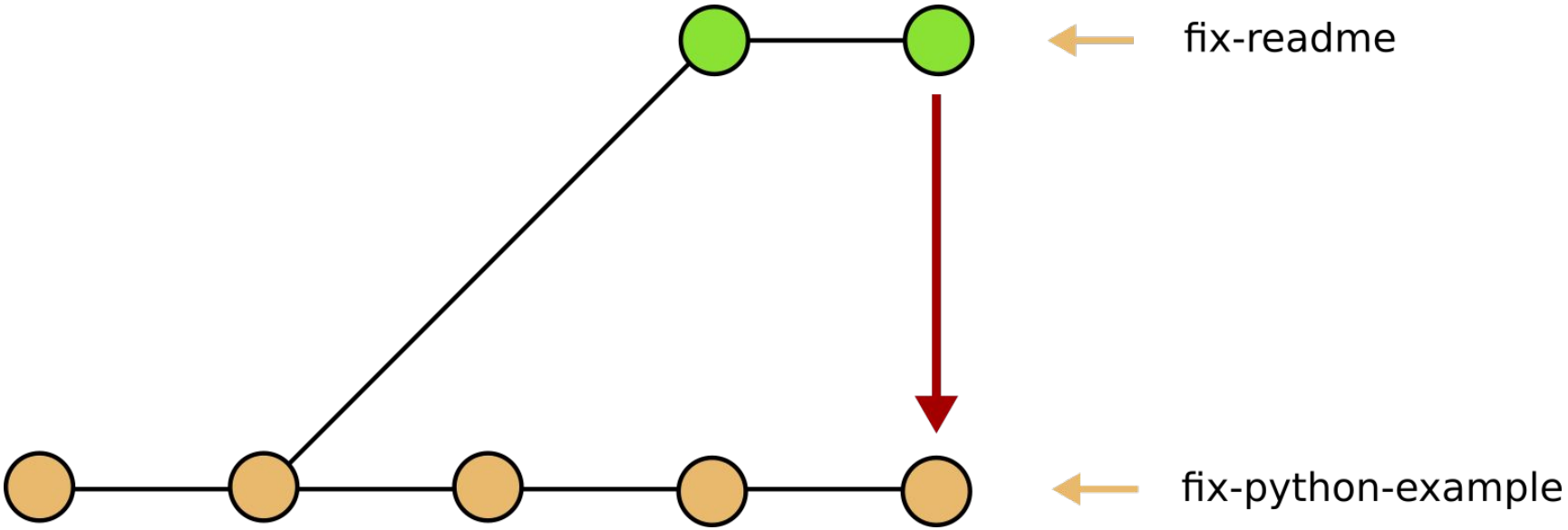
if your active branch is the review branch: `git diff master`

if your active branch is master: `git diff ..branch-name`

Differences

Between Branches - Graphical

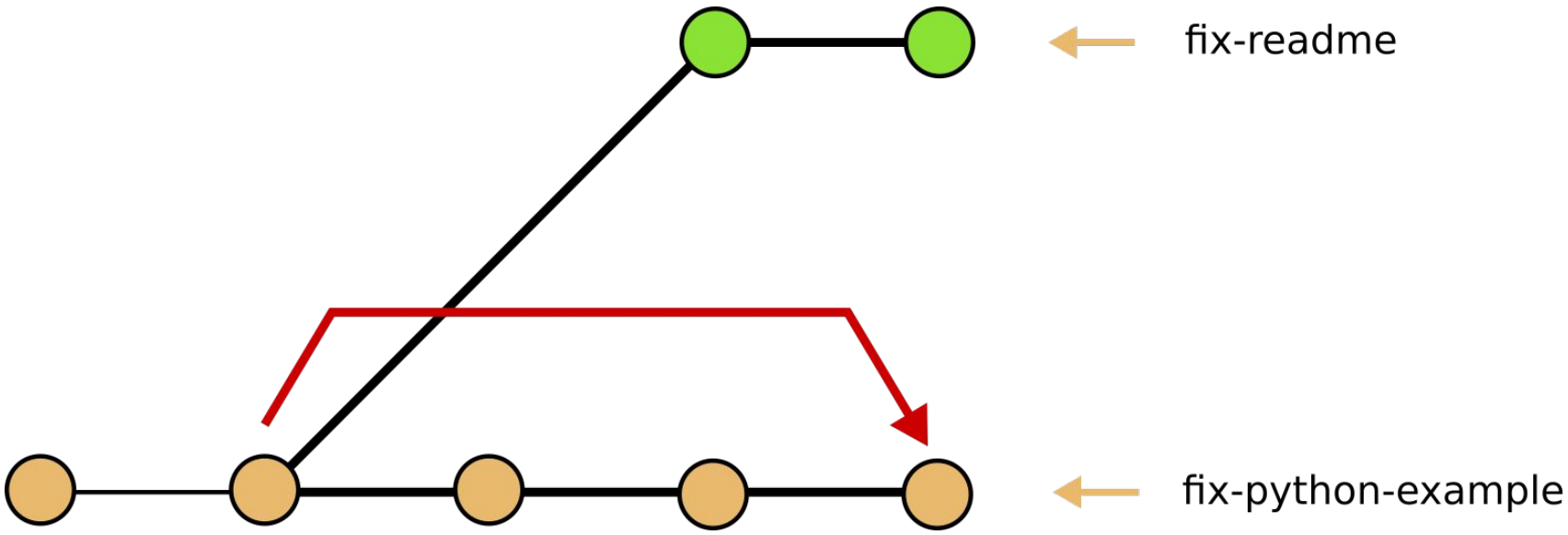
```
$ git diff fix-readme..fix-python-example
```



Differences

Triple Dot Range

```
$ git diff fix-readme...fix-python-example
```



Identical to:

```
git diff $(git merge-base fix-readme fix-python-example) fix-python-example
```

Differences

Diff Meta Data

Show difference limited to directory or file:

```
$ git diff ref-1:path ref-2:path
$ git diff ref-1 ref-2 path
```

Show difference, except that particular directory:

```
$ git diff davem-net-master davem-net-next-master -- . ':!Documentation'
```

Difference

External Diff Tool

Switch to alternative diff tool:

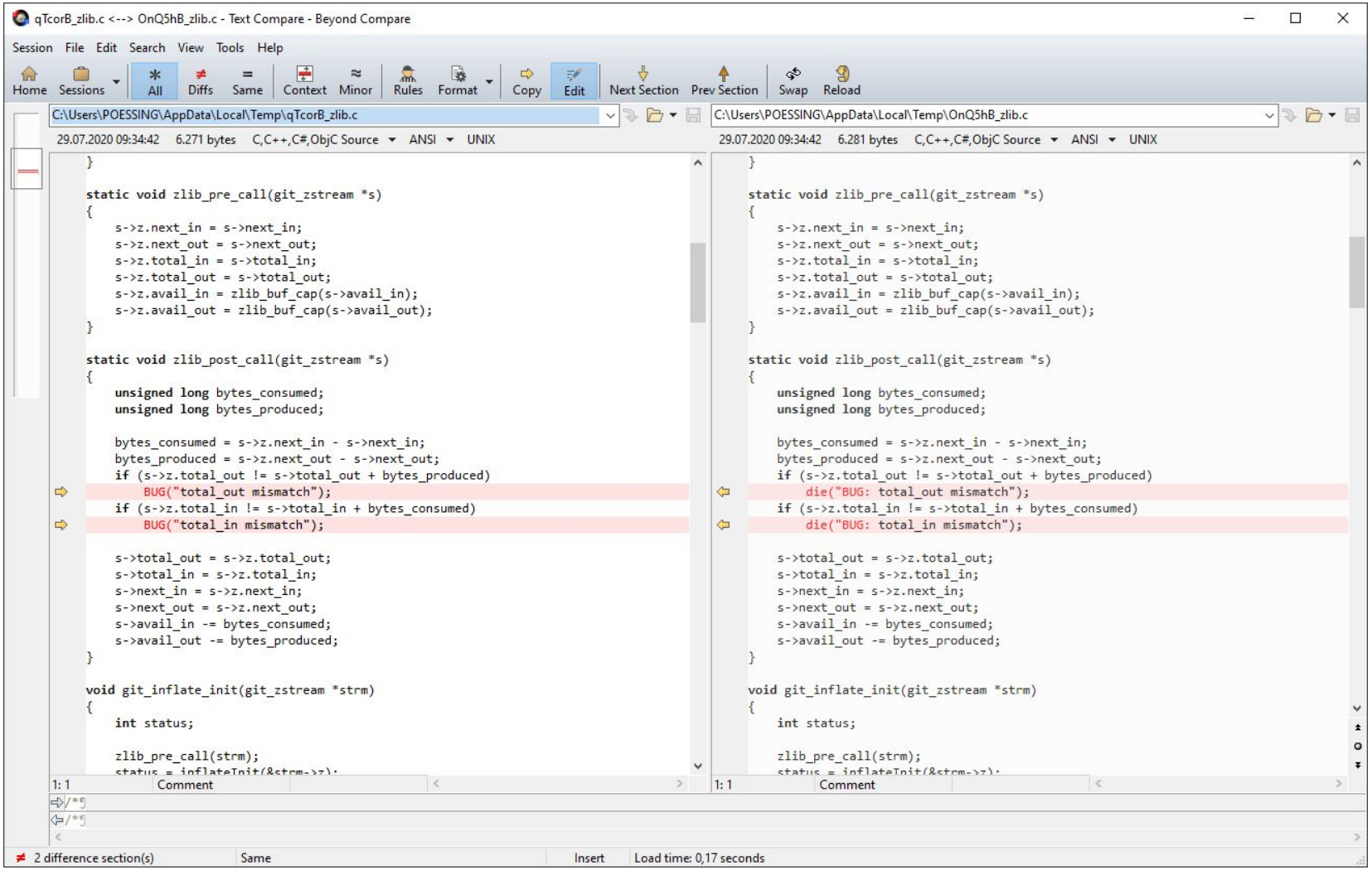
```
$ git config diff.tool winmerge
$ git config diff.tool meld
```

```
$ git difftool branch1:net/socket.c branch2:net/socket.c
```

defaults to vimdiff

Difference

Beyond Compare




```
$ git difftool 033abf97fc 9a6f1287fb -- zlib.c
```

Repository Logs

Commit History

Git log show the history of the repository including branch information

```
$ git log --decorate

commit be646c5781f9a0d877a89ba40a85dd7001198554 (HEAD -> master)
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:01:32 2016 +0100

    commit #3

commit 208e6ba0559e6a2cc5dd45acc55d973a48b436ec
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:01:32 2016 +0100

    commit #2

commit a8c52c86b6e10c4253c72f9953ba242f6b50621c
Author: Hagen Paul Pfeifer <hagen@jauu.net>
Date:   Tue Mar 15 21:01:32 2016 +0100

    initial commit
```

Commit History

Show log with branch names, for all branches, in graphed short format:

```
$ git log --graph --decorate --oneline --all

* 8de98b0 (fix-readme) commit #6
| * 7b06522 (HEAD -> fix-python-example) commit #5
| * fd1b245 commit #4
| * 697b467 (master) commit #3
|/
* 70b0ed2 commit #2
* 74a3dcf initial commit
```

Commit History

Formatting - Useful Options

Show commit file changes in diff format:

```
$ git log -p
```

Show statistic for each commit (file changed, lines added & removed)

```
$ git log --stat
```

User formatted info (id, name, relative date, summary)

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

Commit History

Limiting - Useful Options I

Limit to 23 commits

```
$ git log -23
```

Show log for specific period (can be absolute, e.g. 2008-11-01):

```
$ git log --since=4.weeks --before='2 weeks 3 days 2 hours 30 minutes 59 seconds ago'
```

Show only commits where blob (function) was changed (added, removed)

```
$ git log -Sbpf_init_net -p --all
```

Only show commits with specific commit comment

```
$ git log --grep "Fixes: 3759824da87b"
```

Commit History

Two more goodies - I

Commit information for specific file beyond renames

```
$ git log --name-status --follow --oneline -- include/asm-generic/barrier.h
1077fa3 arch: Add lightweight memory barriers dma_rmb() and dma_wmb()
M       include/asm-generic/barrier.h
febdbfe arch: Prepare for smp_mb__{before,after}_atomic()
M       include/asm-generic/barrier.h
47933ad arch: Introduce smp_load_acquire(), smp_store_release()
M       include/asm-generic/barrier.h
93ea02b arch: Clean up asm/barrier.h implementations using asm-generic/barrier.h
M       include/asm-generic/barrier.h
885df91 Create asm-generic/barrier.h
C061    arch/mn10300/include/asm/barrier.h      include/asm-generic/barrier.h
1c80f22 Disintegrate asm/system.h for MN10300
A       arch/mn10300/include/asm/barrier.h
```

find renames: `git diff --summary --find-renames=90 'HEAD~50' HEAD | grep rename`

Commit History

Two more goodies - II

See how function ipv6_add_dev in addrconf.c **evolved** over time:

```
$ git log -L '/*ipv6_add_dev(/','/^}/':net/ipv6/addrconf.c
```

For **every modification** in the range **show** the **log** entry inclusive diff

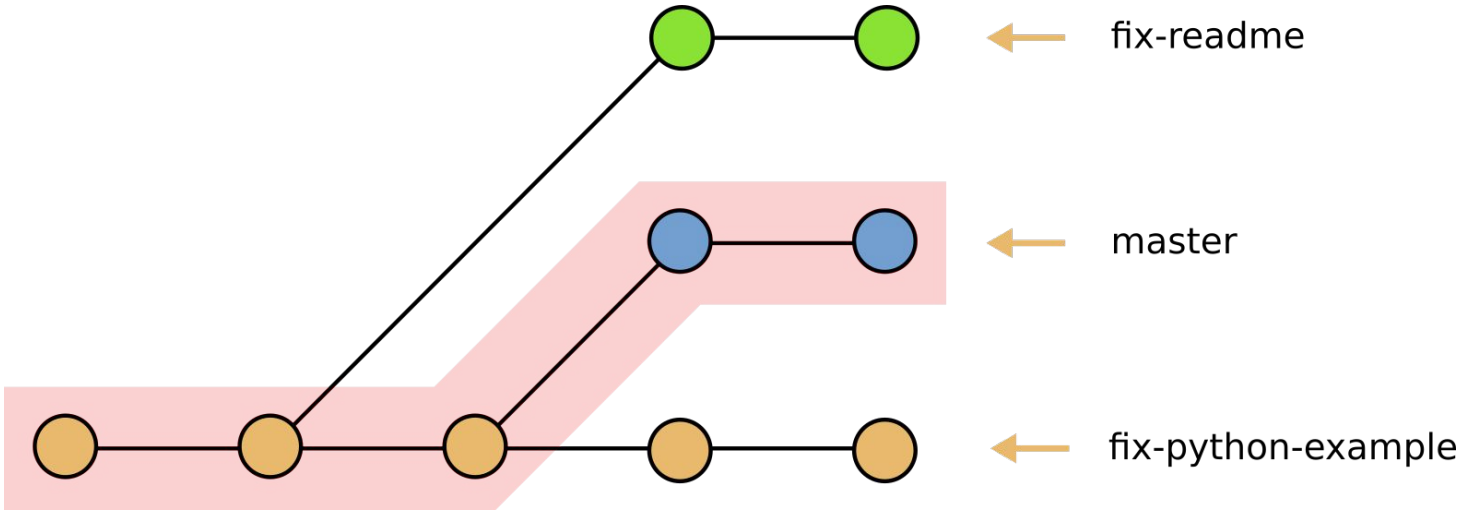
Regular Expression Range:

- Start: `*ipv6_add_dev(`
- End: `^}`

Commit History

Branch Specifics - Ranges

```
$ git checkout master
$ git log
```

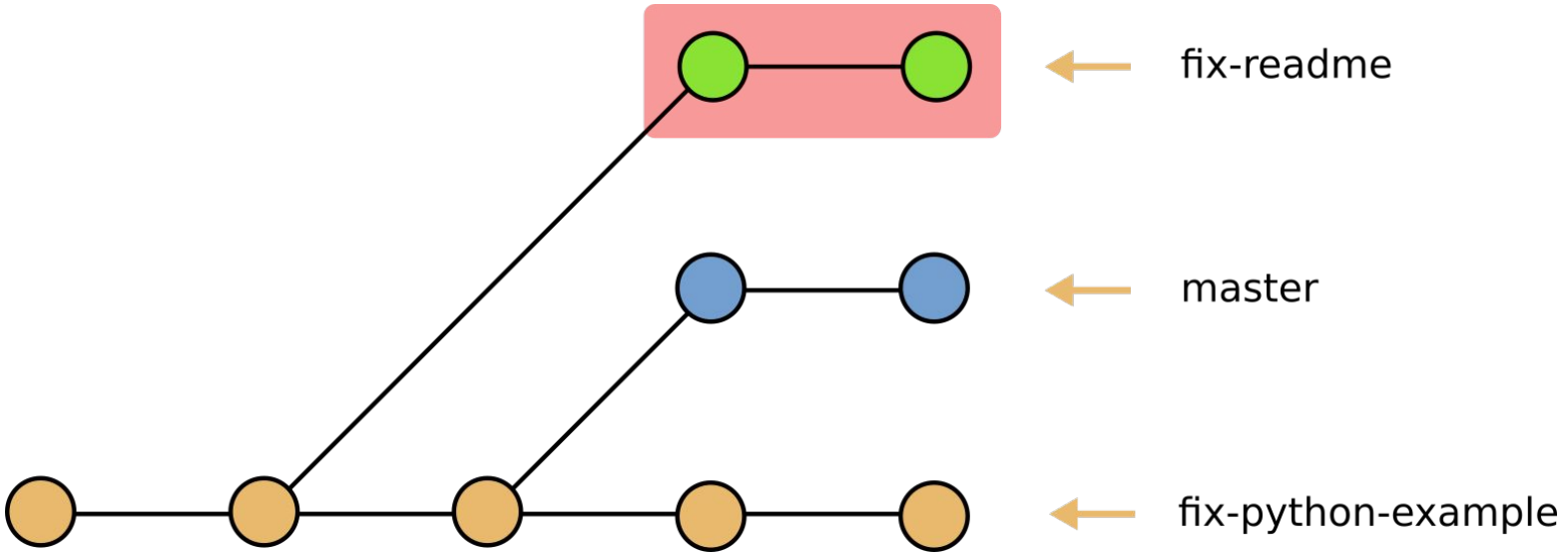


Commit History

Branch Specifics - Ranges

Commits in fix-readme but not in master

```
$ git log master..fix-readme
```

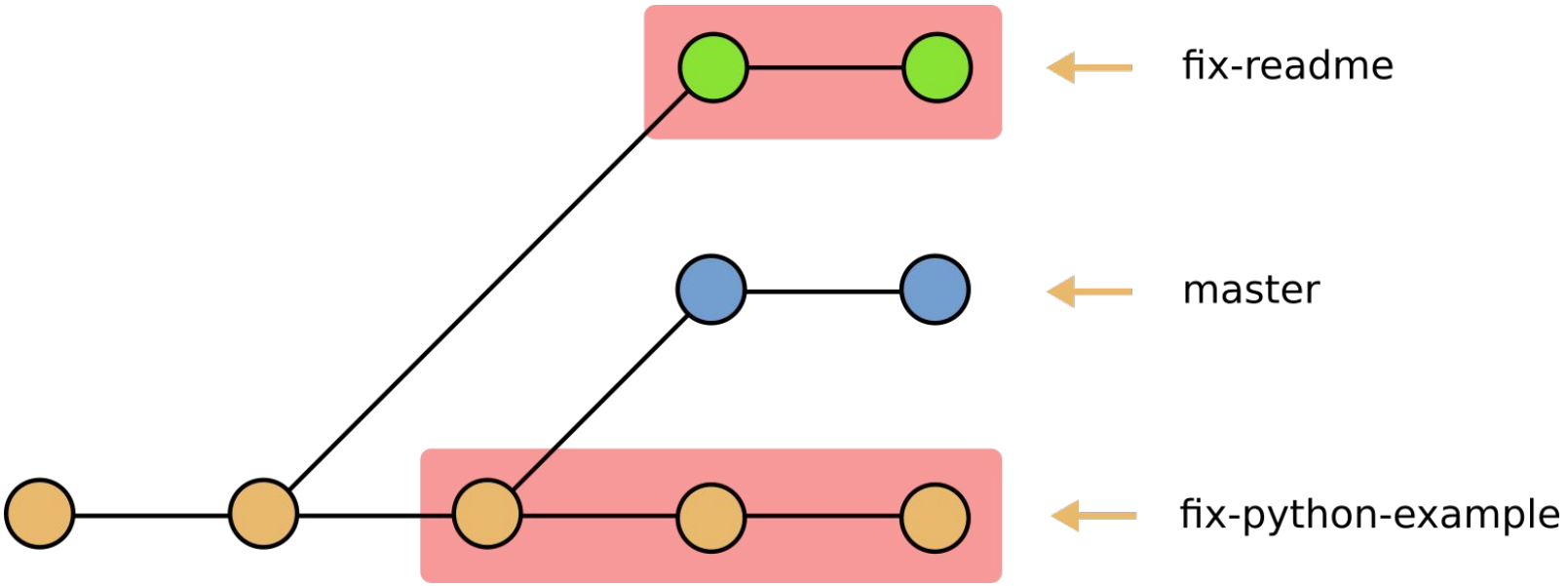


Commit History

Branch Specifics - Ranges

Commits only in fix-readme and fix-python-example - no shared commits

```
$ git log fix-python-example...fix-readme
```

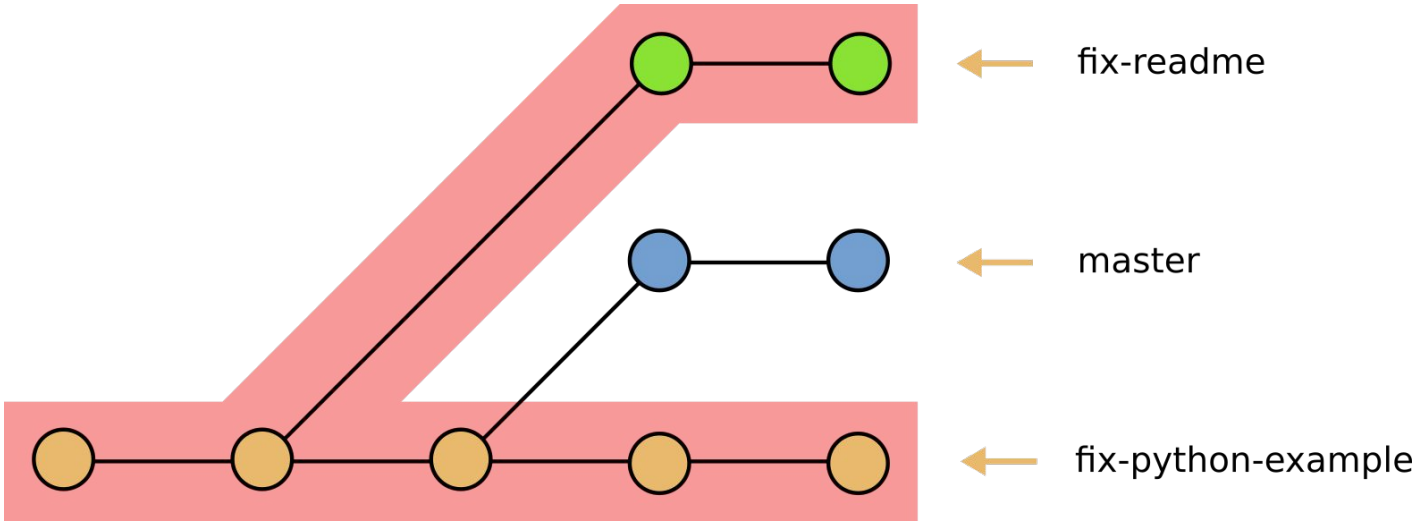


Commit History

Branch Specifics - Ranges

Commits in fix-python-example **and** fix-readme

```
$ git log fix-python-example fix-readme
```



Merging

“It Does Not Matter How Easy It Is To Branch, Only How Easy It Is to Merge”

-- Linus Torvalds

Merging
Background

After branching and developing merging is the subsequent operation - **bringing things together**!

Git do a terrible good job of simplify the merge process

- Git detects renames
- Knows what is the origin of a branch

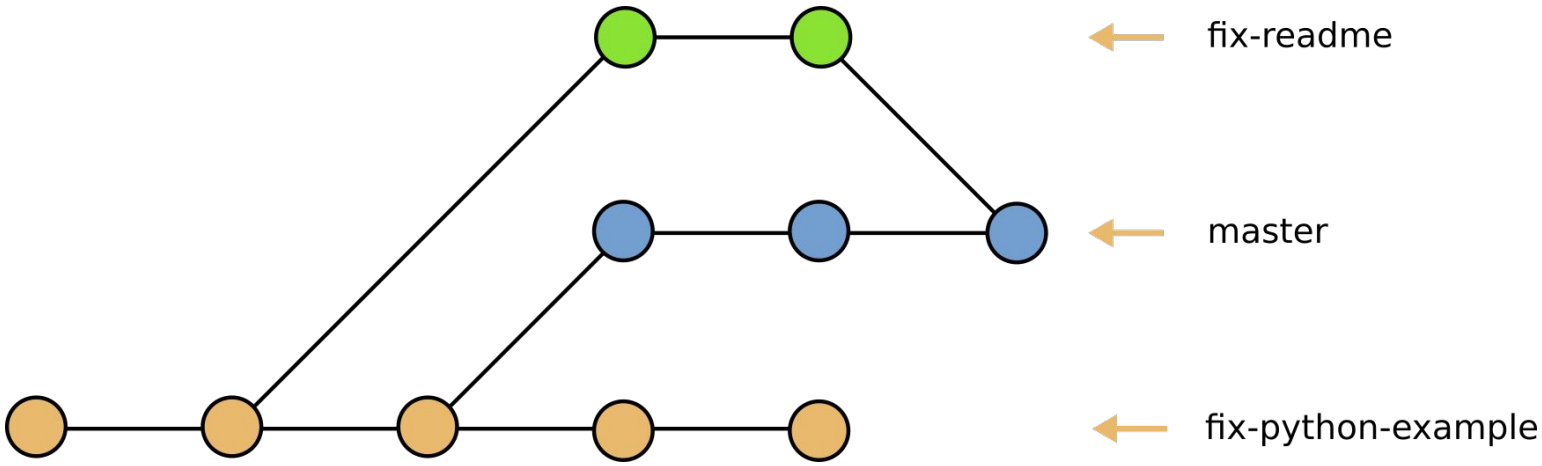
In Advanced Git we will see the background:

- What's merging at its lowest level
- What implement other SCMs
- How can I tweak the merge algorithm

Merging

Merge content from branches:

```
$ git checkout master
$ git merge fix-readme
Merge made by the 'recursive' strategy.
3.py   |    1 +
1 file changed, 1 insertion(+)
```

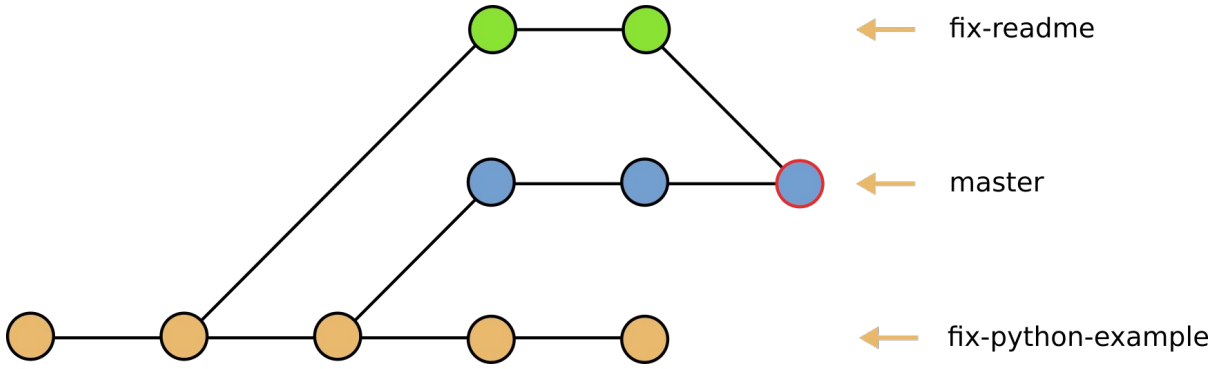


Merging

Merge Commit

The red highlighted circle is a merge commit:

- Who merged at what time which branches
- Conflicted merges will be resolved in these commits



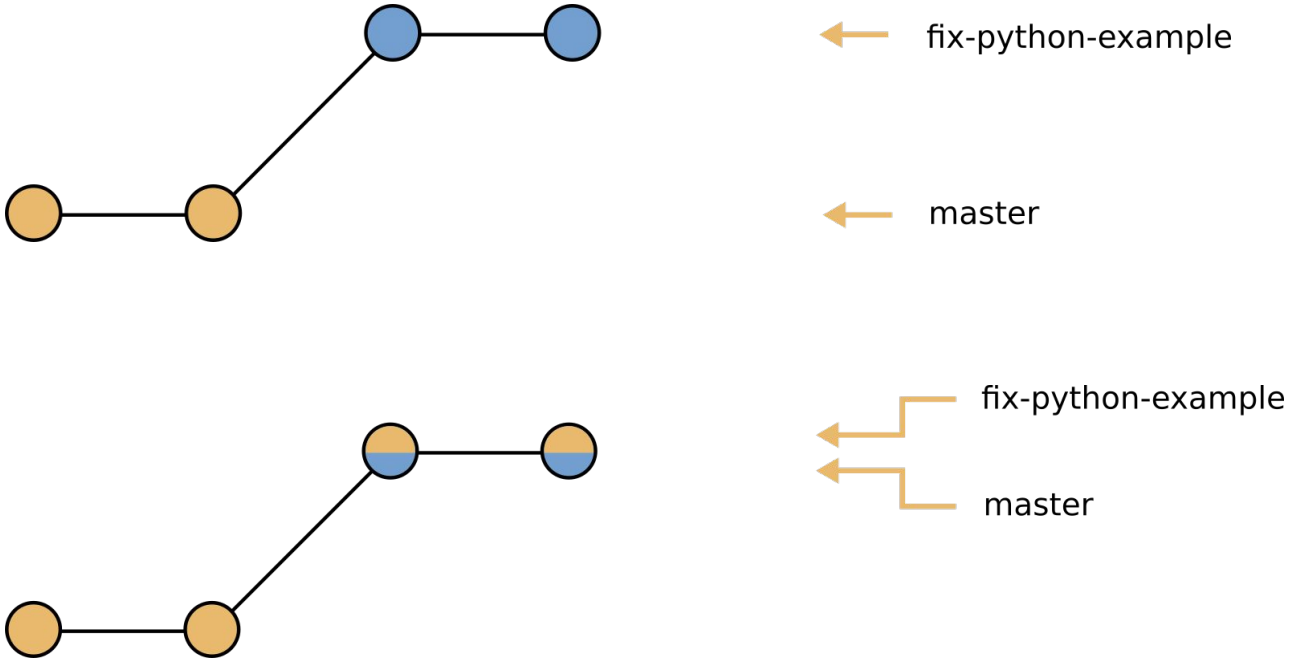
Show all merge commits:

```
$ git log --merges
```

Merging

Fast Forward Merges

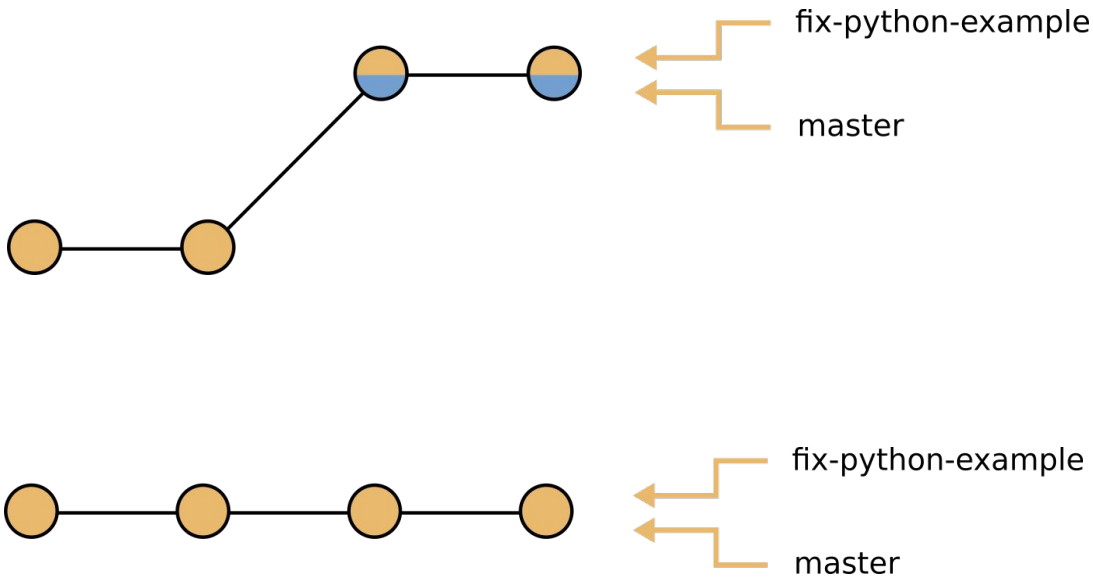
Fast forward merges are nice locally. Ask yourself: “is the branch history required?”



Merging

Fast Forward Merges

With fast forward you probably lose branch information

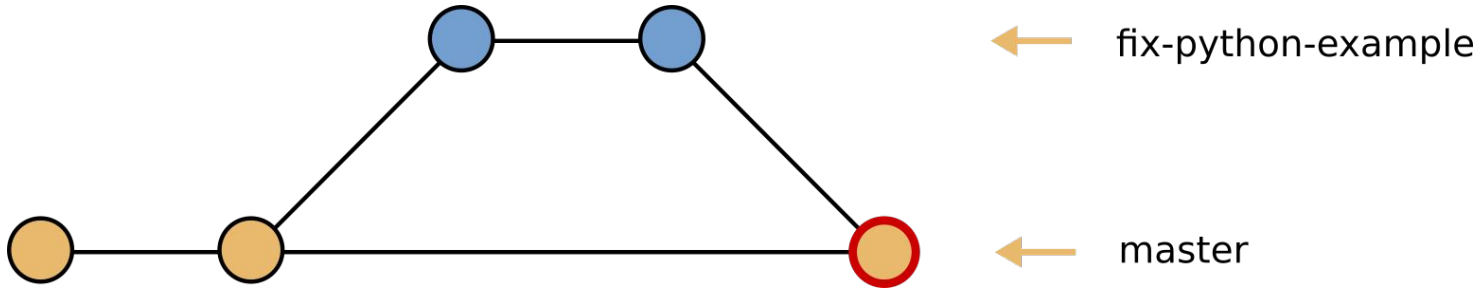


Merging

Avoid Fast Forward - Enforce Merge Commit

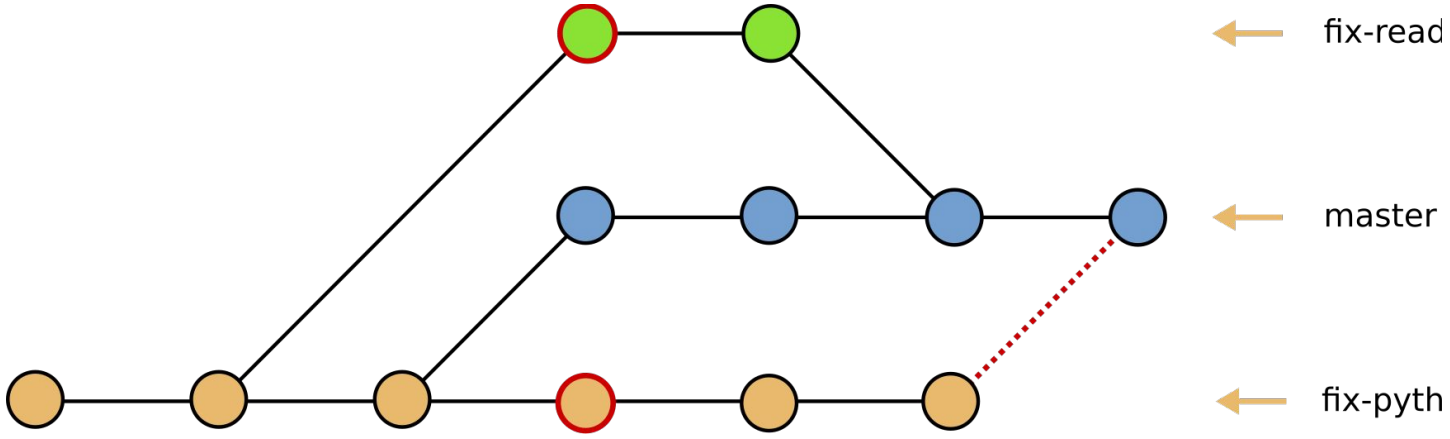
Enforce a merge commit:

```
$ git merge --no-ff <branch>
```



Merging
Conflicts

If two branches changes the same content from the identical file a merge will result in a merge conflict - this cannot be resolved in an automatic way



Merging
Detect Conflicts

```
$ git merge fix-python-example
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

Showing conflicted files:

```
$ git status --short
UU README
```

Merging
Resolve Conflicts

Two approaches to resolve the conflict:

- Edit the file(s) with your editor, conflict markers are presented
- Call git mergetool to start a merge tool and help you

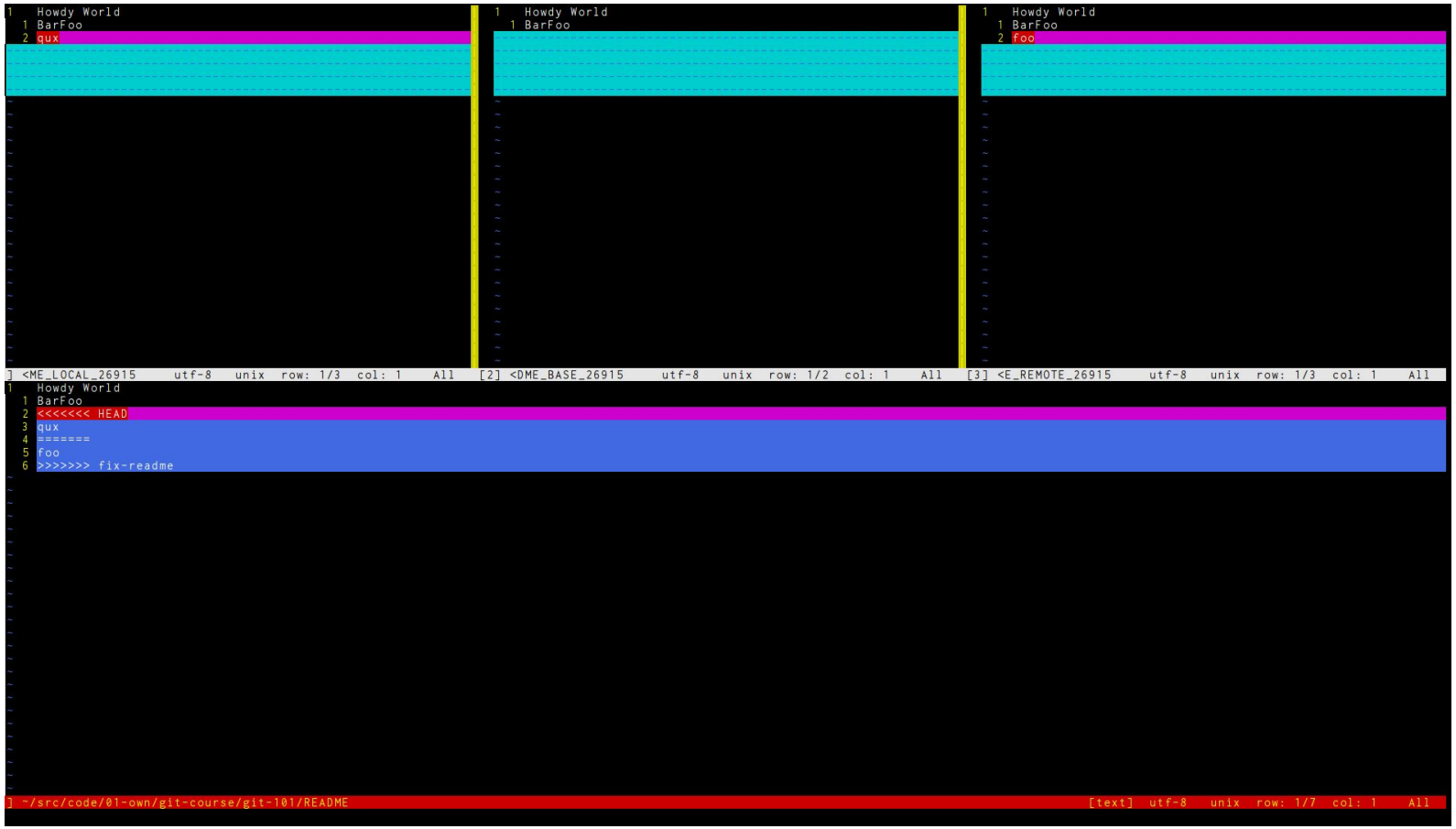
```
$ vim README
Howdy World
BarFoo
<<<<<<< HEAD
qux
=======
foo
>>>>>>> fix-python-example
```

1) remove all markers 2) there may be multiple conflict sections per file

Merging

Resolve Conflicts - II

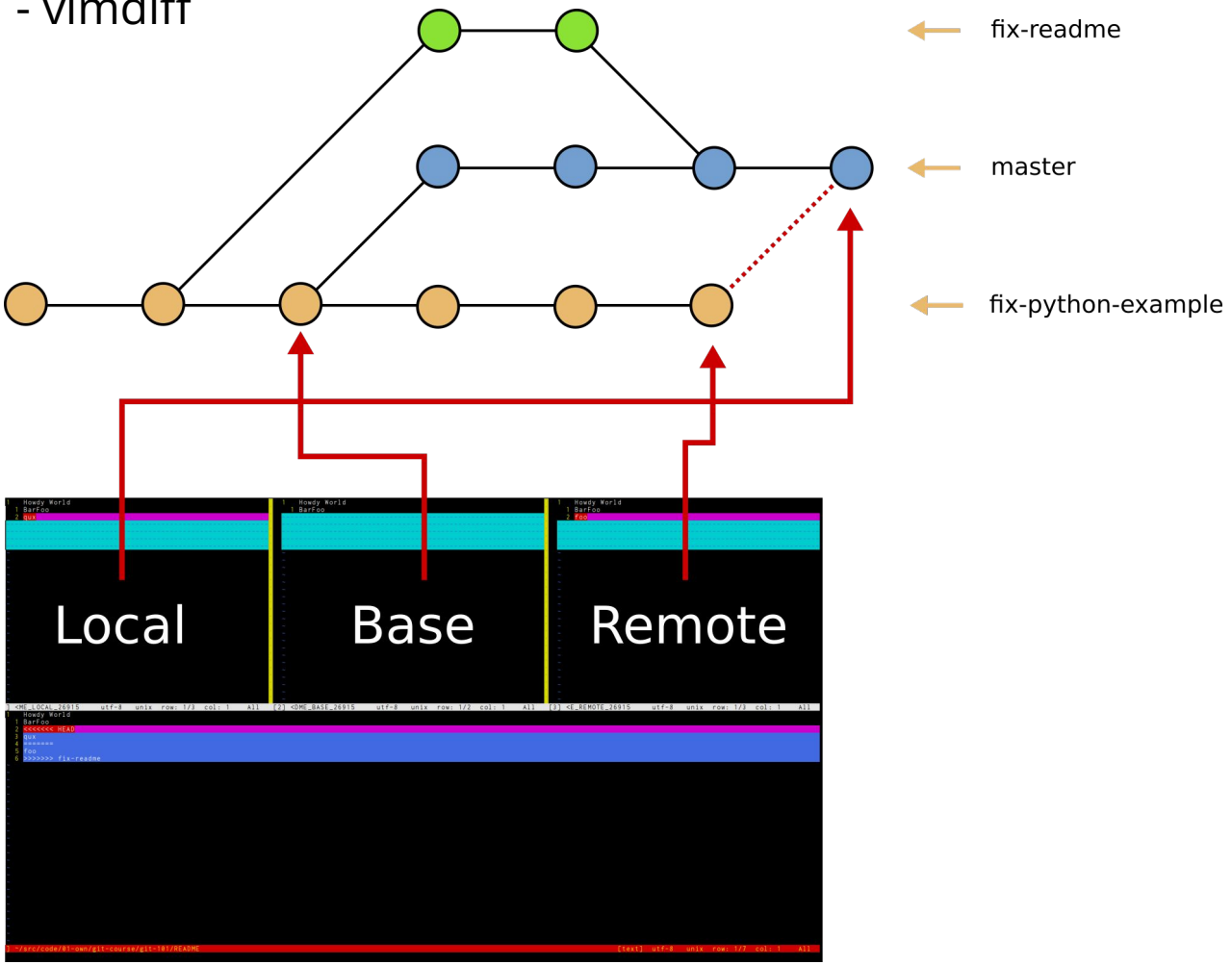
Git Mergetool → vimdiff



Merging

Resolve Conflicts - II

Git Mergetool - vimdiff



Merging
Finish Conflicted Merge

After resolving all conflicts (testing is fine) the merge should be finished:

```
$ git add README
$ git commit
```

Merging

Binary and Unmergeable Files

Sometimes you don't **want** or **can't resolve conflicts** by juggling lines

Favor changes in **case of conflict**:

```
$ git merge -Xours <branch-name>
```

Instead of ours, **theirs** is also possible

Beware: no conflicts will raise up. This is probably not what you want and this option should by no means be a default option

only sections between conflict markers are considered - non conflicted areas are normally merged

Merging

Binary and Unmergeable Files

Checkout the wanted file version **after** a conflict occurs

```
$ git checkout --ours <conflicted-filename>
```

This approach is more explicit and safer!

Tip: you can checkout any file from any branch at any time:

```
$ git show <branch>:<conflicted-filename> > <conflicted-filename>
```

Merging
Merge Strategies

Without option git uses a so called **recursive** merge **strategy**

`--Xours` and `--Xtheirs` are **merge options**

Useful option: `--ignore-all-space`
Ignores whitespace at line end, and considers all other sequences
of one or more whitespace characters to be equivalent.

Other available merge strategies (subset):

- *resolve*
- *octopus*
- *ours* (not identical to the option! Basically ignore the other branch)

Merging
Merge Strategies

Via .gitattributes you can favor a union style in the case of conflicts for specific file types:

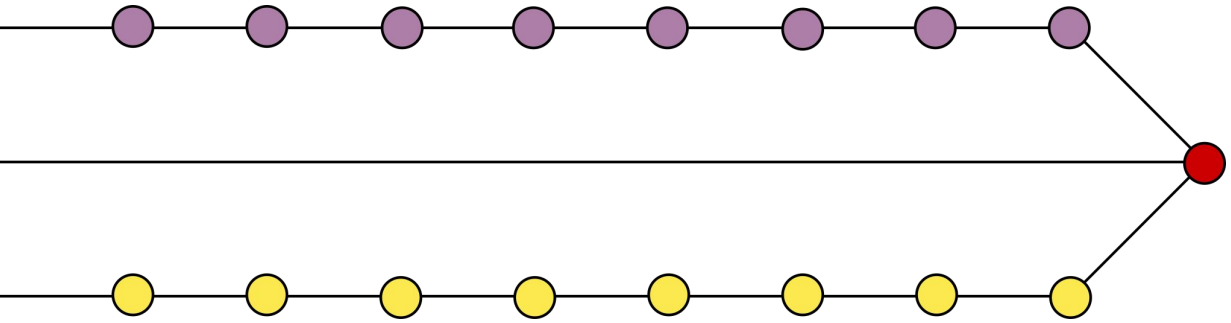
```
$ cat .gitattributes
Changelog merge=union
```

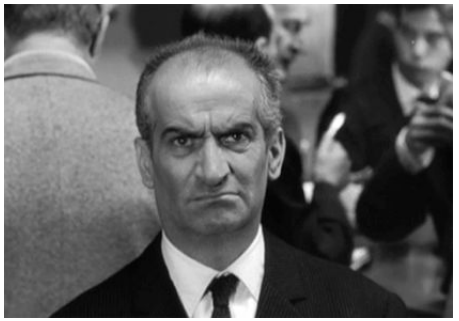
For conflicts in the file Changelog both changes are merged

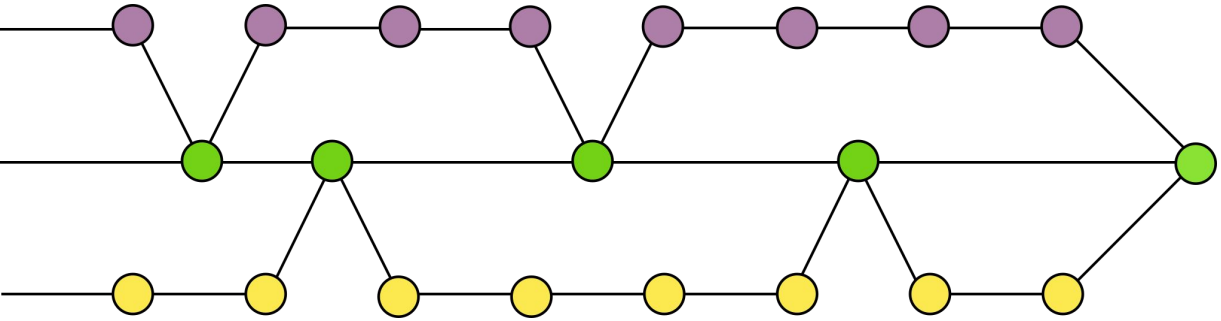
Merging
Reducing Conflicts

Conflicts may happen. To reduce merge conflicts effectively:

Commit Early - Commit Often







Merging
Final Words

Show all branches where a particular commit is included:

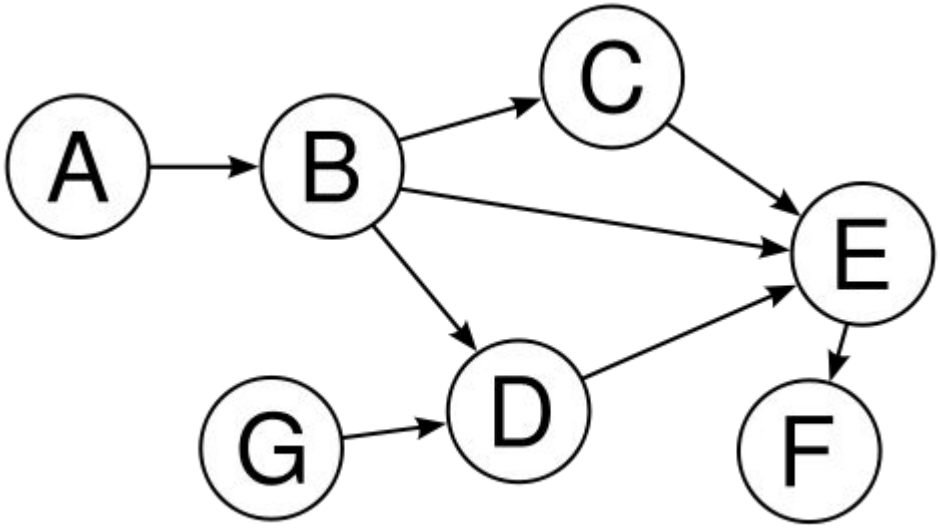
```
$ git branch --remotes --contains <commit>
```

Show all commits recently merged as a handy new command (alias):

```
$ git config --global alias.merge-log = "!f() { git log --stat "$1^..$1"; }; f"
```

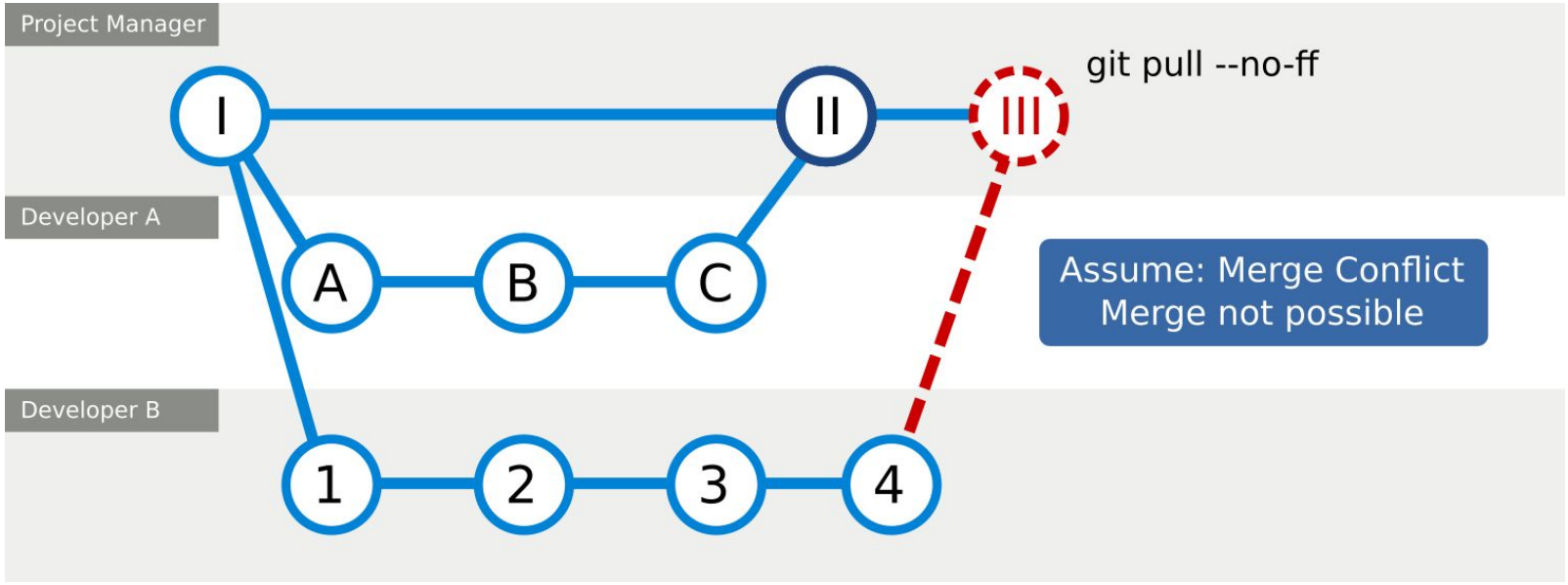
Git Rebase

Powertool to change & modify the directed acyclic graph to re-wire edges



Rebasing

Preface - Example I



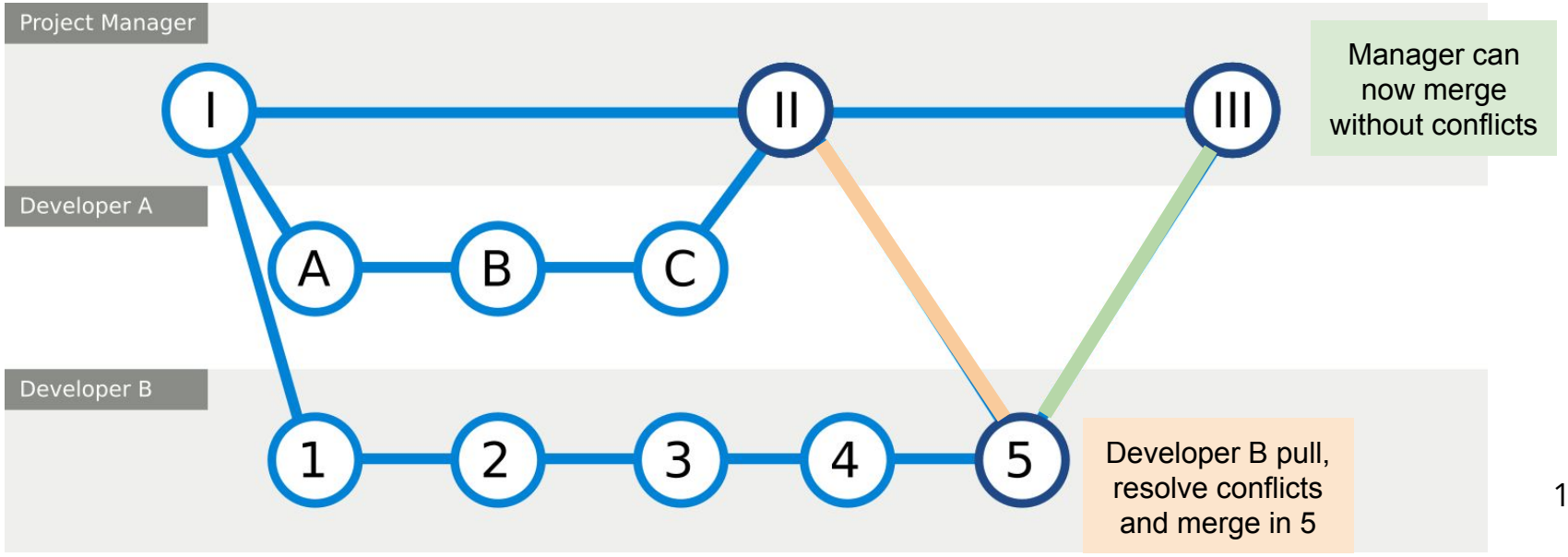
Note: conflicts are normally resolved at "producer" side. Merge manager (project manager, software maintainer) often have no clue how to fix the conflict. Here developer B knows probably best how to fix the conflict. If not she/he can contact developer A to discuss this

Rebasing

Preface - Example II

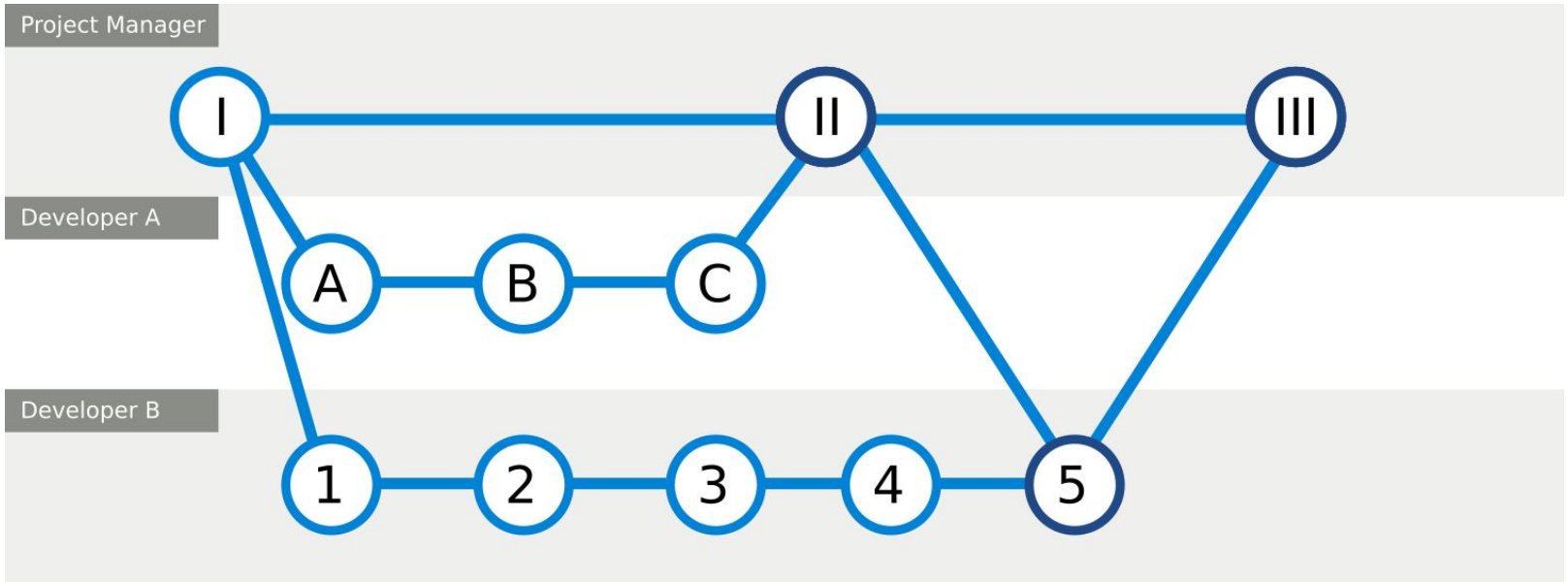
Developer B is enforced to **merge** upstream work and **resolve conflicts** locally

The maintainer can now merge developer B commits without any trouble



Rebasing

Clarity of History



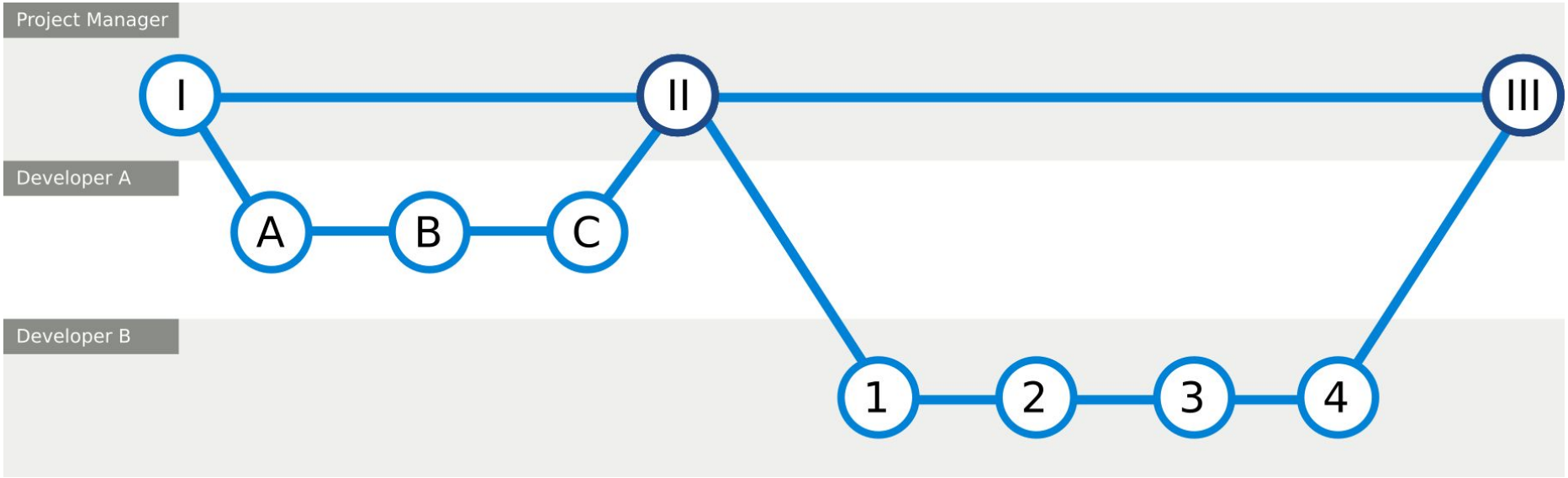
Big **downside** of reverse merges: the **graph** is **cluttered** with non project relevant merges - we need somethings else!

Git **rebase** will help us here (and other areas)

Rebasing

Preface - Example II

Ideal graph structure:



Rebasing

Preface - Example III

Tip: if you **know** the code you are working on is **highly modified** by **others** try to **integrate** upstream work **often**!
(if you start to integrate after e.g. 3 months it is probably **hard** to resolve conflicts introduced **by your** code!)

Side effect: you will get early feedback how your modifications behave with the latest upstream code.

Rebasing

Relocate branches

Rebase is the process to **relocate** (move) a branch on **another** branch/commit

It **preserves all** the work, all commits - but **changes** the **location**

Rebase is **destructive** operation! All subsequent **commit ID**s will **change** (remember: the commit id is composed of the changeset and the history of all commits)

Rebase operations must be **limited** to **local** branches. Never rebase published structures

Rebasing
Under the Hood

Will **iteratively** work over **all** commits that happened on the feature branch and **replay** them as if they were written just in time starting with the oldest commit

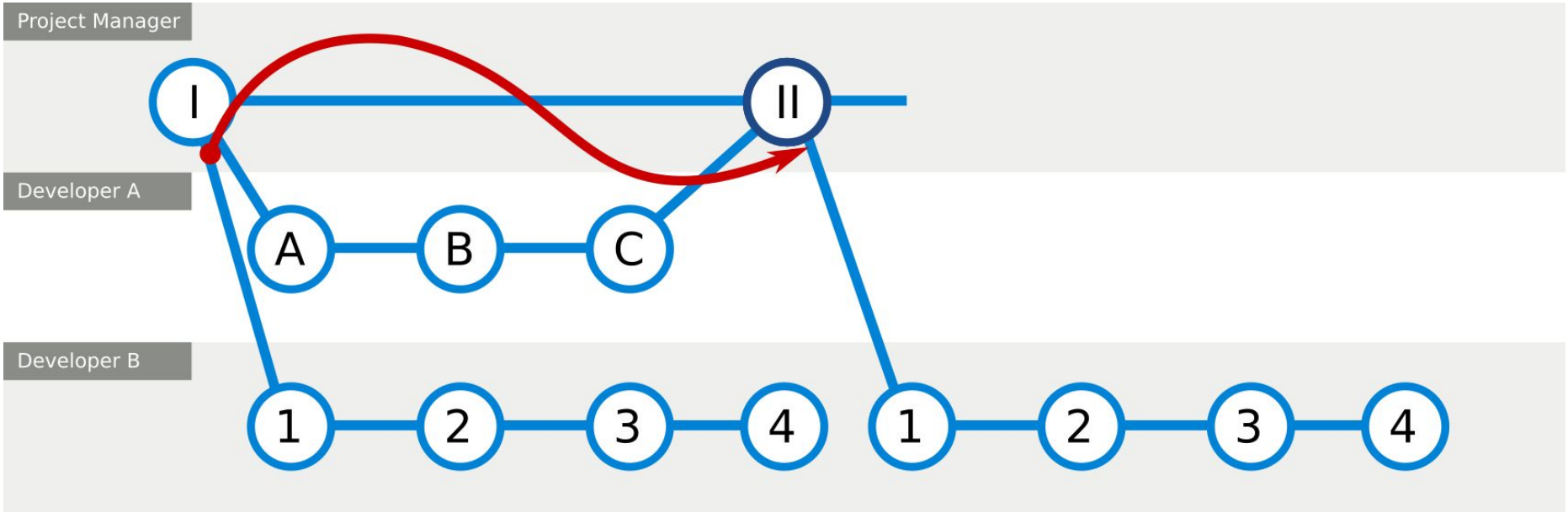
Conflicts may happen at **every** commit!

Commit **IDs** are **re-calculated!**

Rebasing

Relocate branches

Rebase is the process to relocate (move) a branch on another branch - visualized

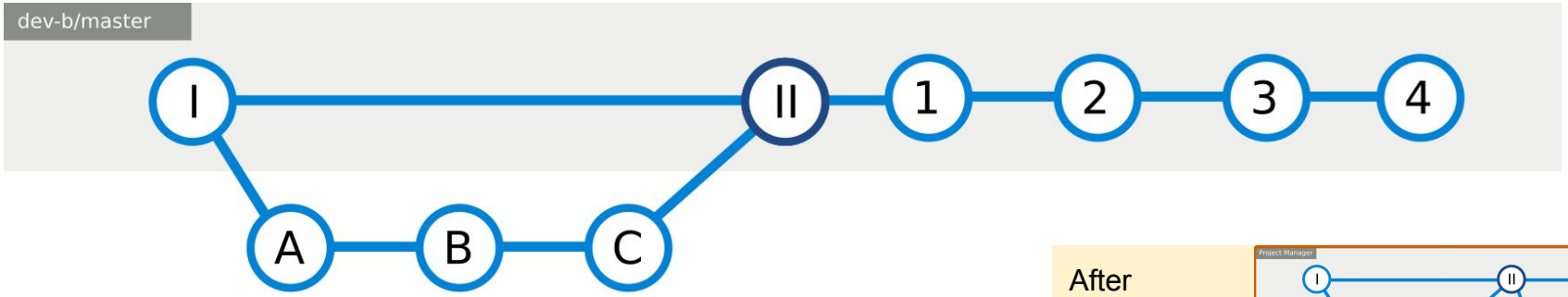


Rebasing

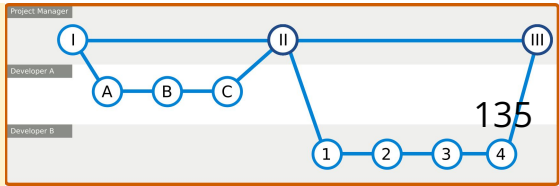
Hands on

Rebase (on) master (upstream) into master branch

```
$ whoami
Developer-b

$ git checkout upstream && git pull                    ← update upstream
$ git checkout master
$ git rebase upstream
```



After maintainer merge:



Rebasing

Conflict Resolve

Compared to merge conflicts, rebase conflicts may arise at each commit!

Unlike merges there is no merge commit (because it is no merge, just reordering/copy of work done on another branch

Conflicts: resolve conflict via editor or mergetool and continue rebasing

```
$ git checkout fix-python-example
$ git rebase master
[conflict message]
$ git mergetool
$ git add <conflicted-files>
$ git rebase --continue
```

Rebasing

When to Rebase and when not

Rebasing is a **destructive** operation - never do it on published branches!

Rebasing is **perfect** to **cleanup** and **prepare** your **topic branches** before splice commits into your public master branch!

→ **Rebase a topic branch before completing a Merge Request**

Rebase workflows exhibit a more clean history (`git log`) and **bisecting** is also simplified

Rebasing

When to Rebase and when not

Rebasing is much more powerful as "move topic branches on top of other branches". Rebasing support:

- **Modify** existing commits
- **Squash** several commits into one
- **Edit** the **commit message**
- **Delete** individual commits

Remember:

“c*lean your own stuff*" **vs** "*don't clean other people's stuff*"

Git Cherry-Pick

Picture by Christian Engvall with permission to reuse

Git Cherry-Pick

Re-**apply changes** introduced by an **existing commit** on the branch you are currently on, supposing you have a clean working tree.

With `git cherry-pick` maintaining a software project became much easier.

Bugfixes can now be back/forward-ported with a single command.

```
$ git cherry-pick <ID>
```

Git Cherry-Pick

Howto

Check for clean working tree:

```
$ git status --short        # check for clean working tree
```

Cherry pick commit id 29e406e

```
$ git cherry-pick 29e406e
[release-1 724574e] README: fix typo
 Date: Sat Jun 21 10:54:47 2025 +0200
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git Cherry-Pick

Example

```
$ git cherry-pick 29e406e
[release-1 724574e] README: fix typo
 Date: Sat Jun 21 10:54:47 2025 +0200
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
$ git log --oneline --decorate --graph
* 724574e (HEAD -> release-1) README: fix typo
| * 29e406e (master) README: fix typo
| * 67a2fd8 ChangeLog
| * fb67376 Added .gitignore
|/
* ab75e3a Makefile
* ff5a0f4 Add README.md
* 920dc34 Third Commit
* 622e27c Second Commit
* 886232a Initial Commit
```

Git Cherry-Pick

Example Continued

```
$ git log --decorate -1 29e406e
commit 29e406e98de9e1d5f28d982506a5f01a94c0118b (master)
Author: Christian Breunig <christian@breunig.cc>
Date:   Sat Jun 21 10:54:47 2025 +0200

    README: fix typo
```

```
$ git log --pretty=fuller --decorate -1 724574e
commit 724574ebd188e4d14e6db05b7e8930d74d5b80ea (HEAD -> release-1)
Author:     Christian Breunig <christian@breunig.cc>
AuthorDate: Sat Jun 21 10:54:47 2025 +0200
Commit:     Christian Breunig <christian@breunig.cc>
CommitDate: Sat Jun 21 10:55:32 2025 +0200 # see updated timestamp

    README: fix typo
```

Git Cherry-Pick

Options

Cherry-Pick option `-x` adds a back reference to the original commit to the commit message of the cherry-pick.

```
$ git log --pretty=fuller --decorate -1 4ed5f6c
commit 4ed5f6c36f09a687879fe5855275ca3f0109d322 (HEAD -> release-2)
Author:     Christian Breunig <christian@breunig.cc>
AuthorDate: Sat Jun 21 10:54:47 2025 +0200
Commit:     Christian Breunig <christian@breunig.cc>
CommitDate: Sat Jun 21 10:54:47 2025 +0200

    README: fix typo

    (cherry picked from commit 29e406e98de9e1d5f28d982506a5f01a94c0118b)
```

Git Aliases

Git Aliases

Some command for Git might feel very much unhandy

With aliases you can **shorten** your most beloved commands to a **bare minimum** so you do not need to type the entire command over and over again

All aliases are stored in `.gitconfig`

Git Alias

Example I

Create an alias for `git status --short` which is invoked by calling `git st`

```
$ git config --global alias.st “status --short --branch”
```

Alias for a nice looking graph which only shows the commits headline, called `git lg`.

```
$ git config --global alias.lg “log --color --graph
--pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
%C(bold blue)<%an>%Creset' --abbrev-commit”
```

Git Alias

Example II

You are not bound to alias only build-in git commands, you can also invoke external tools

This brief example will show how the `visual` alias is used to execute the gitk binary. You can call any arbitrary binary with its arguments here!

External programs must be prefixed with an exclamation mark.

```
$ git config --global alias.visual ‘!gitk’
```

Imagine an alias which opens Beyond Compare showing you the latest diff on a file

Git Bisect

Git Bisect

A **different way to find a regression**!

With `git bisect` you are able to **track down** the particular **commit** which introduced a regression. **Precondition**: you are **able to differentiate** a **good** from a **bad** commit

Helpful when you know how to **reproduce** the failure

Even better: you have a **test** which can be **executed automatically**

git bisect will change the way how to debug regressions! It is a different way to deal with errors

Git Bisect

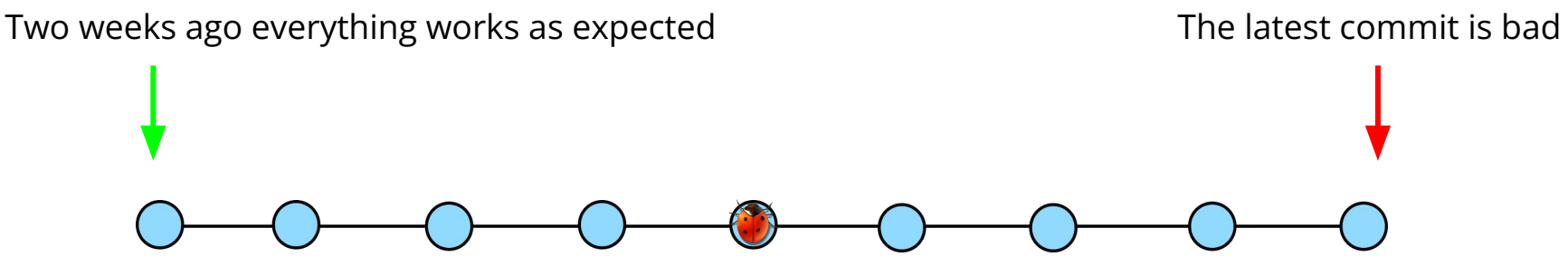
Example

Imagine: during development a **memory corruption bug** was introduced. Wrong pointer arithmetic (e.g. the day after Oktoberfest)

Bug is **reproducible** via software tests

You have **no clue which component misbehaves**! Seems the pointer is randomly pointing to other areas and corrupts memory - your "guess"

But **you know** the following:



Git Bisect

Mechanism

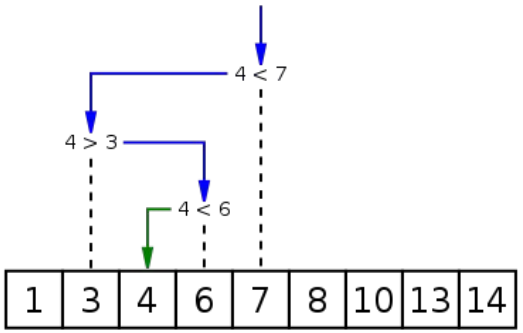
Git bisect does a **binary search** over a particular range to track down to a particular commit

Git bisect is another cause why it is **important** to do **tiny**, **coherent, buildable commits** - changing one thing at a time. Don't change multiple things in one commit (→ commit hook to check quality?)

With git bisect even non programmers or colleagues not familiar with the particular code are able to point you to the bad commit!

O(log2 n):

```
100 revisions: 7
1000 revisions: 10
10000 revisions: 14
100000 revisions: 17
```



Git Bisect

Howto

Start bisect machinery:

```
$ git bisect start
```

Mark the bad commit, assume HEAD is bad:

```
$ git bisect bad
```

Mark a good commit

```
$ git bisect good v5.4
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[93ed8405beb387ec86874d951cf630de2c4fd927] Feature 10
```

Git Bisect

Howto

After `git bisect good`, git will checkout a commit in between. Now compile and run your tests. After test you mark the particular commit as bad or good

```
$ git bisect good || bad
```

If it is not possible to compile and/or test the commit you can even skip this commit:

```
$ git bisect skip
```

Git Bisect

Howto

After n iterations git will show you the first bad commit:

```
$ git bisect good
41a5ddd4dea219d826a15f7a085e412c29333b10 is the first bad commit
commit 41a5ddd4dea219d826a15f7a085e412c29333b10
Author: Hagen Paul Pfeifer
Date:   Wed Mar 19 19:43:40 2014 +0100
[...]
```

If it is a trivial commit you probably can revert it instantly:

```
$ git revert 41a5ddd4dea219d8
```

Git Bisect

Howto

After finishing bisecting - bisecting must be exited

```
$ git bisect reset
```

Git Bisect

Automate Bisecting

Bisecting 17 commits (100.000 commits!) is annoying

Bisecting can be automated:

```
$ git bisect run test-script.sh
```

Script exit code for success: 0, for failure: 1 - if the commit should be skipped the script must return 125

Git Bisect

One more Thing

Redefine terms:

```
$ git bisect start --term-old fast --term-new slow
```

Check remaining suspects:

```
$ git bisect visualize
```

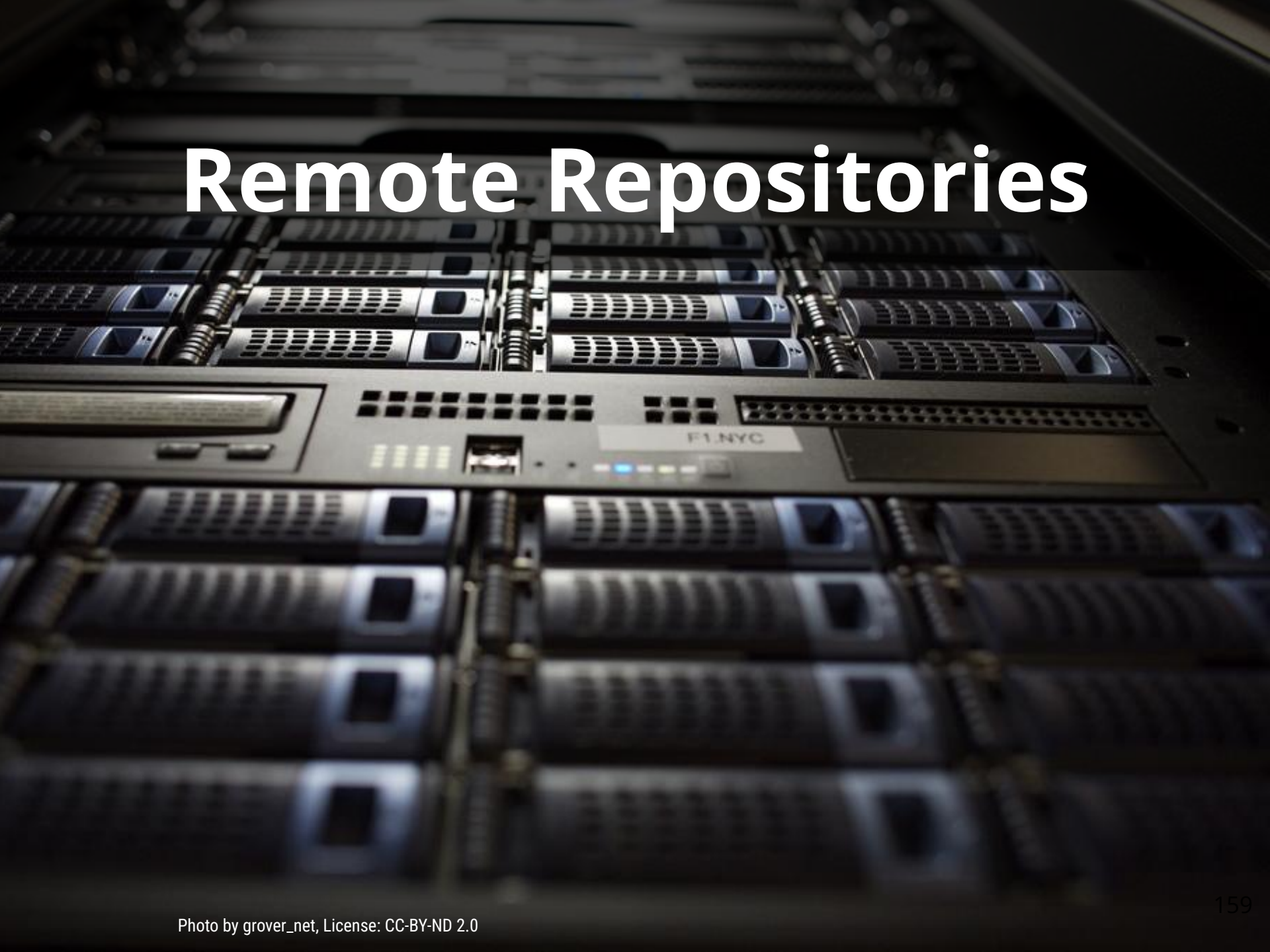
Show what you have done so far:

```
$ git bisect log
```

If you know the subsystem, reduce the commits even further

```
$ git bisect start -- dir-1/subdir-a dir-2/subdir-2
```

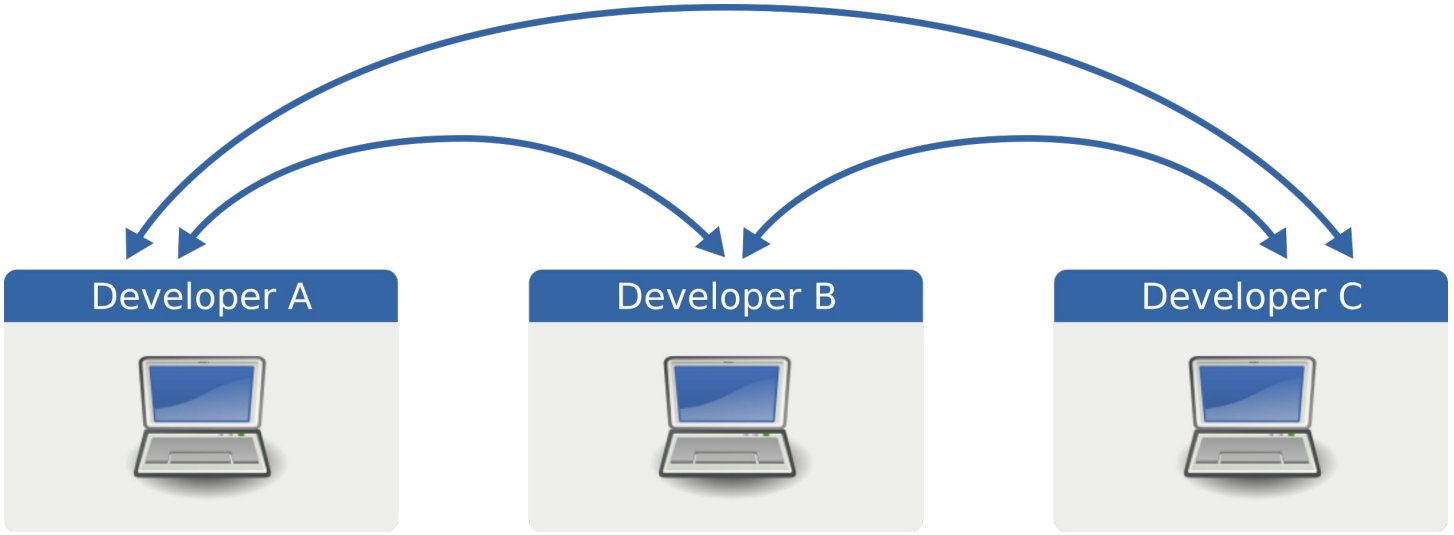
Remote Repositories



Distributed Workflow

Developer Interplay

From an abstract point of view distributed development looks like the following illustration

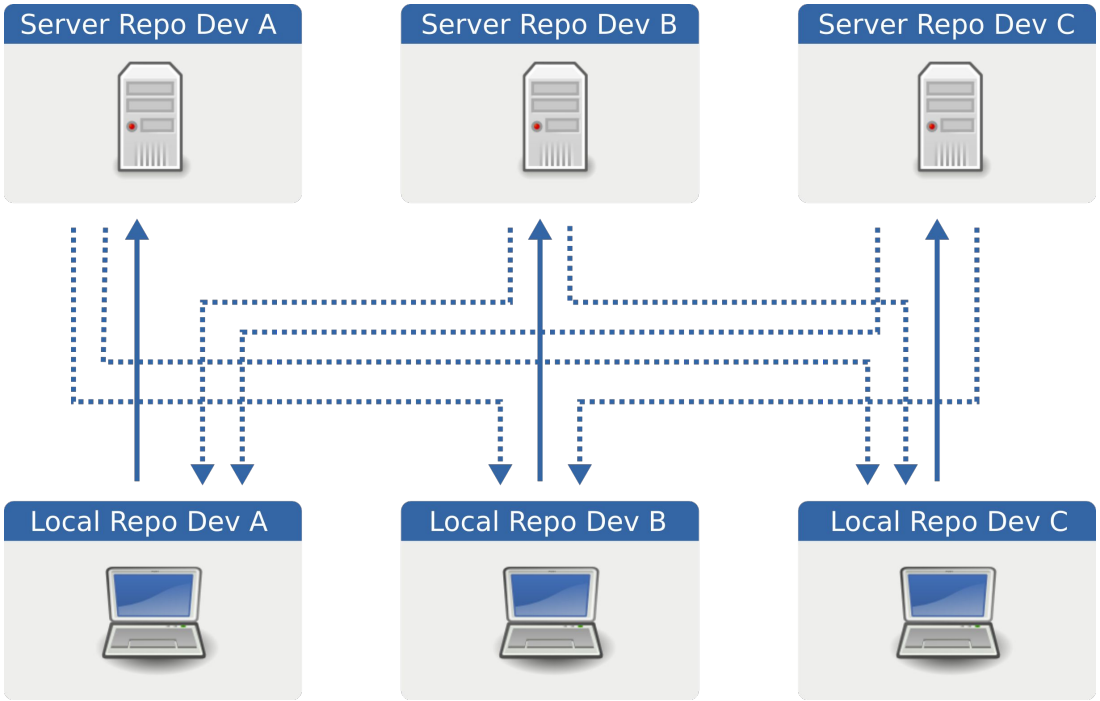


There is one practical problem: developer cannot access the hard disk of other developers directly! 😉

Distributed Workflow

Server Infrastructure

As common in system design: there is no problem that cannot be solved by adding a level of indirection



multiple repositories can be served on one server

Rethink Cloning

Origin

Remember cloning a repository via git clone?

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
[...]
$ git remote -v
origin    git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git (fetch)
origin    git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git (push)
```

Git will setup remote information when we clone a repository

The local repository is connected with the server remote repository “somehow”

origin is the default name, can be renamed and is not special in any way

Rethink Cloning

Branches

Take a look at branch structure

```
$ git branch --all --verbose --verbose
* master                f55532a [origin/master] Linux 4.6-rc1
  remotes/origin/HEAD   -> origin/master
  remotes/origin/master f55532a Linux 4.6-rc1
```

When we clone a repository git will checkout **origin/master** by default and name the branch master, if not specified other by `--branch`

master is a **tracking branch** - connected with upstream *origin/master*

Prepare Cooperation Infrastructure

Upstream Branches

For our project we need a stable branch (called *master*) and a developer branch where all new features are merged

For our vanilla repo we add a new remote information, also call it origin

```
$ git remote add origin git@github.com:hgn/foobar.git
```

Master branch already created, only add new local branch called *dev*

```
$ git branch dev
```

Now push *master* and *dev* to *origin* - after this local and remote repositories are identical

```
$ git push --set-upstream origin master:master
$ git push --set-upstream origin dev:dev  # ← this creates a new branch at server!
```

Prepare Cooperation Infrastructure

Upstream Branches - II

Push semantic:

```
$ git push origin local-branch:remote-branch      # ← like cp(1)
```

If branch with name bar is not available at origin it will **create** a **new branch**

```
$ git push origin foo:bar
```

Often local and remote branch name is **identical**

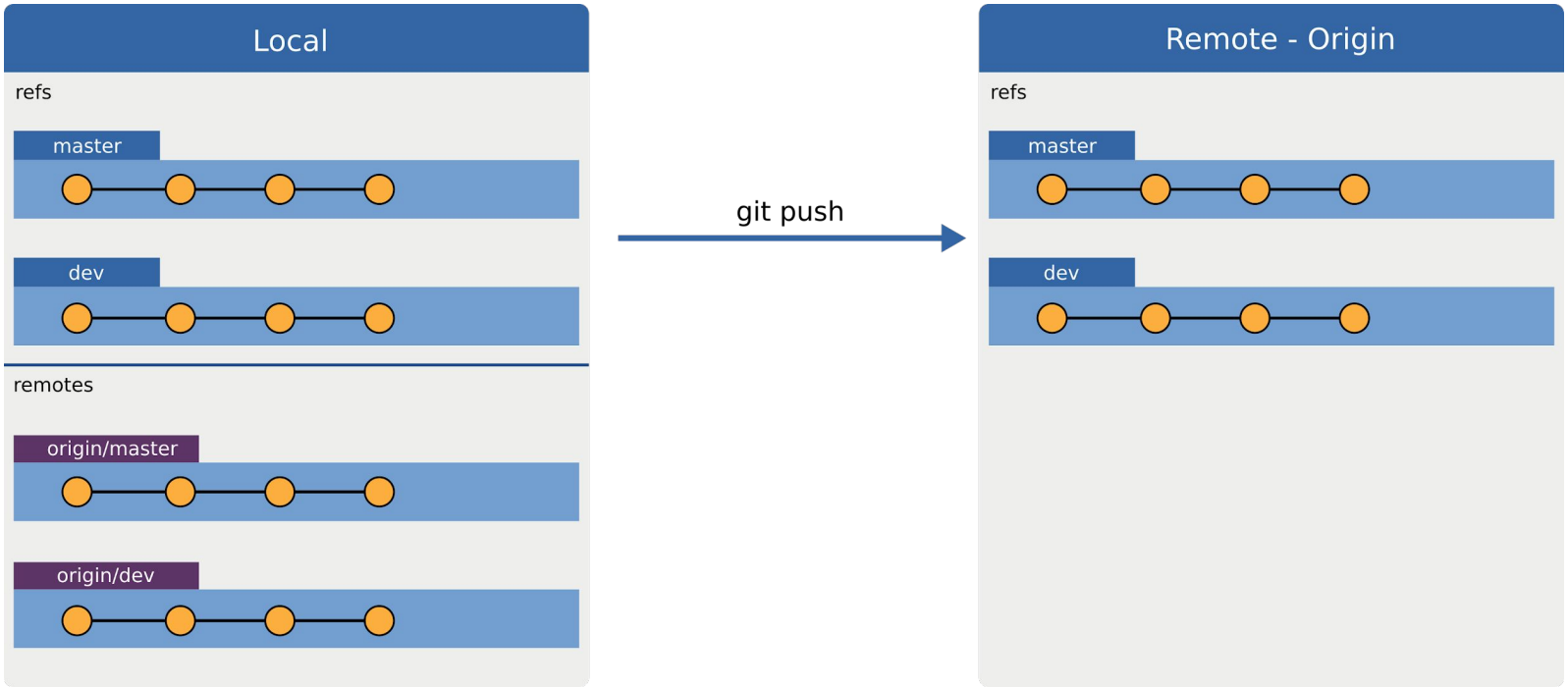
```
$ git push origin foo:foo
```

Delete a **remote** branch foo (i.e. "overwrite with null"):

```
$ git push origin :foo
```

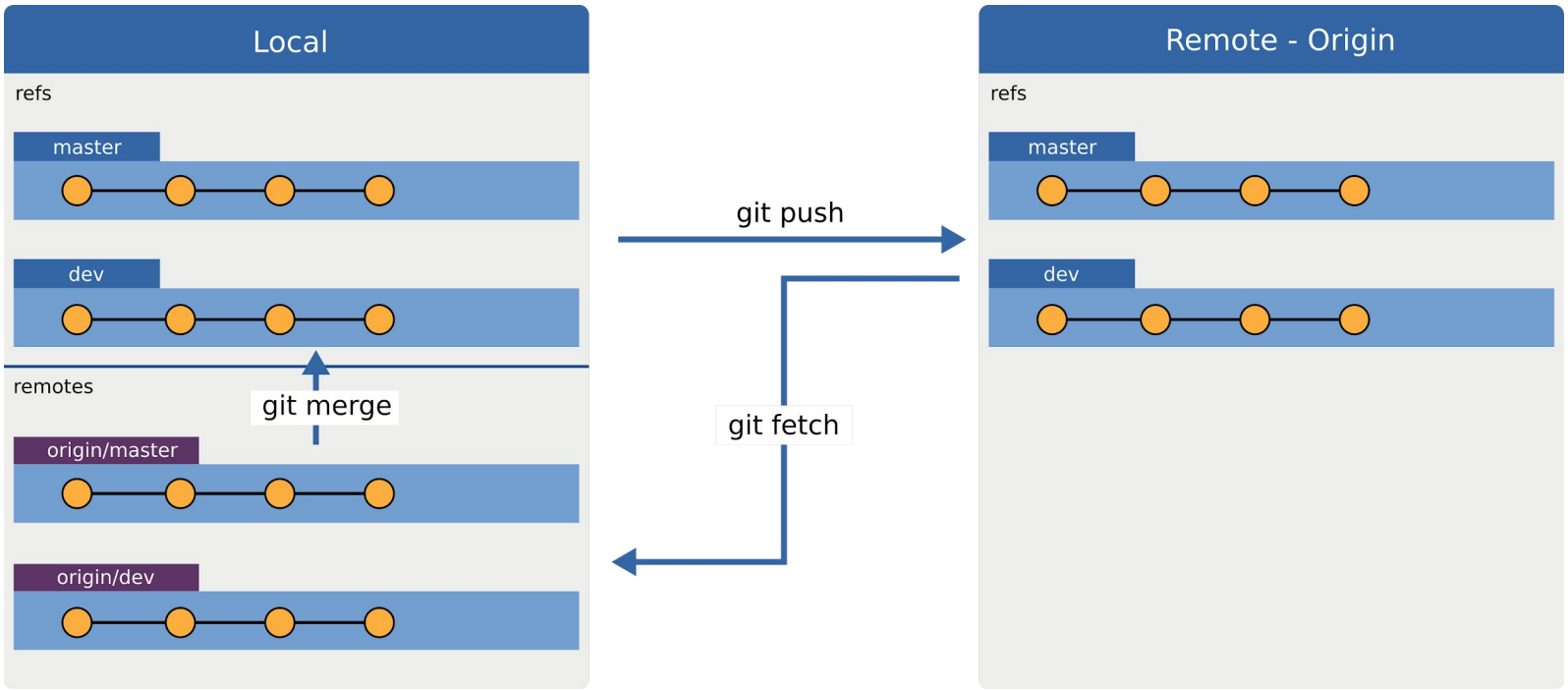
Prepare Cooperation Infrastructure

Upstream Branches



Prepare Cooperation Infrastructure

The Three Sync Operations

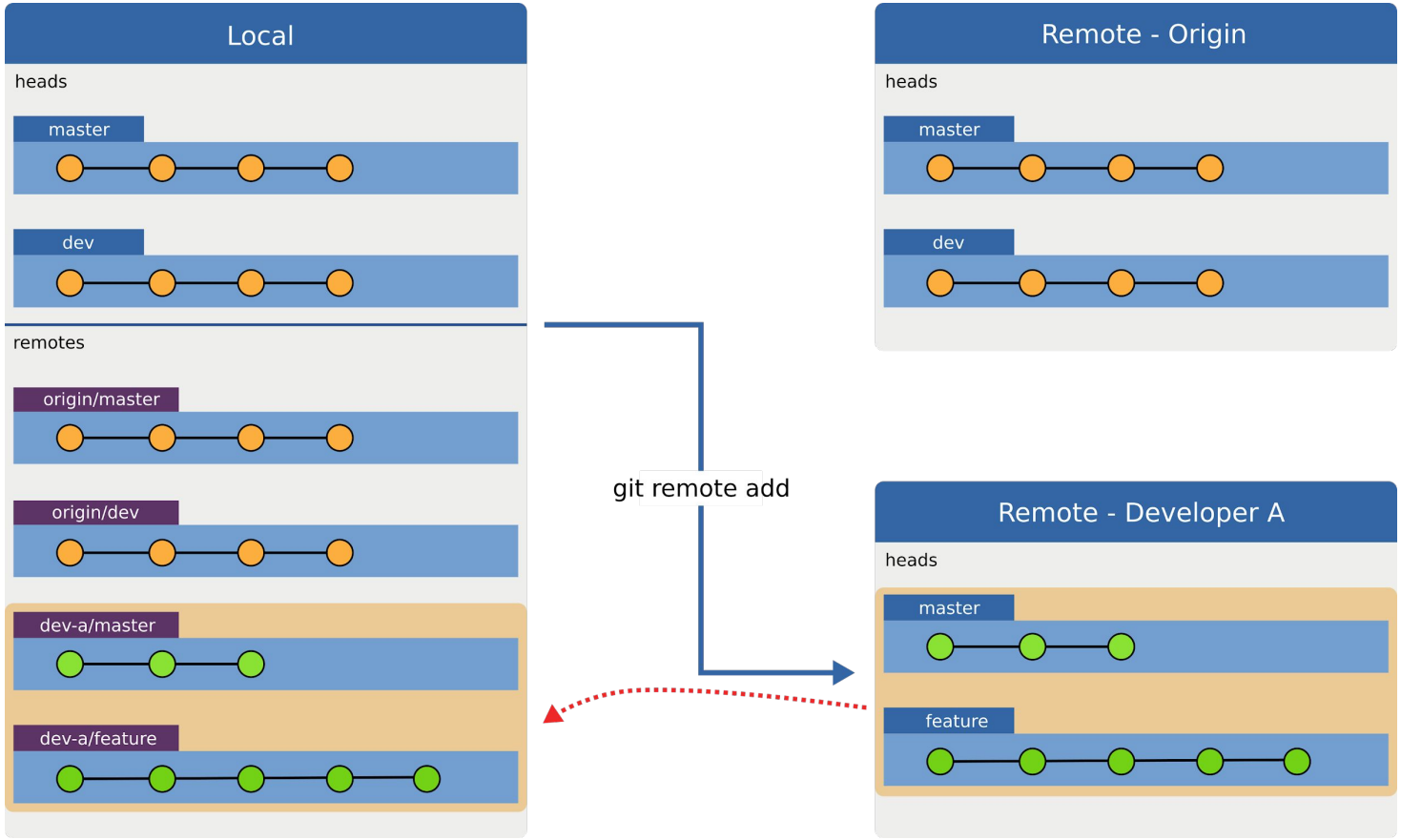


git pull = git fetch + git merge

git clone = git init + git remote add origin <url> + git fetch + git checkout master origin/master

Prepare Cooperation Infrastructure

Adding more Remote Repositories



Prepare Cooperation Infrastructure

Deal with Remote Branches

Update (fetch) new data from remote repositories:

```
$ git remote add dev-a http://git.example.com/project.git
$ git remote update
Fetching origin
Fetching dev-a
```

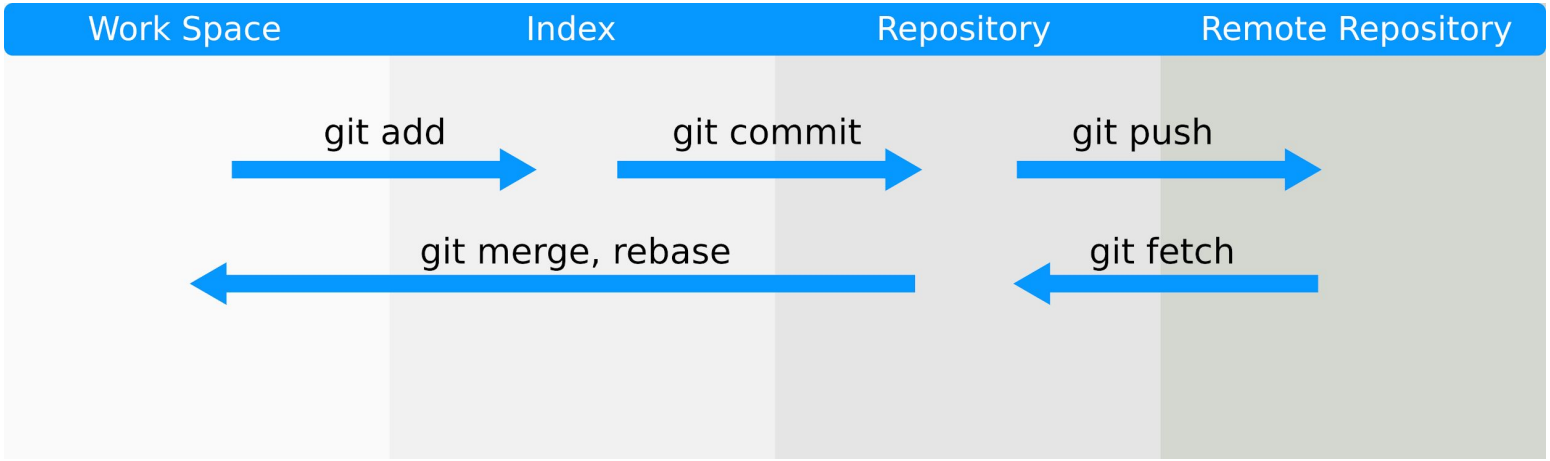
Show remote branches:

```
$ git branch -r
  dev-a/feature
  dev-a/master
  origin/HEAD -> origin/master
  origin/dev
  origin/master
```

equivalent to `git fetch --all`

Prepare Cooperation Infrastructure

Operations



Repository Safety

Local repositories are **not backuped**!

Push your **local branches to a remote** repository regularly - just to have a copy of it

If you follow the "*Commit early, commit often*" paradigm you get this for free!

Merge Process

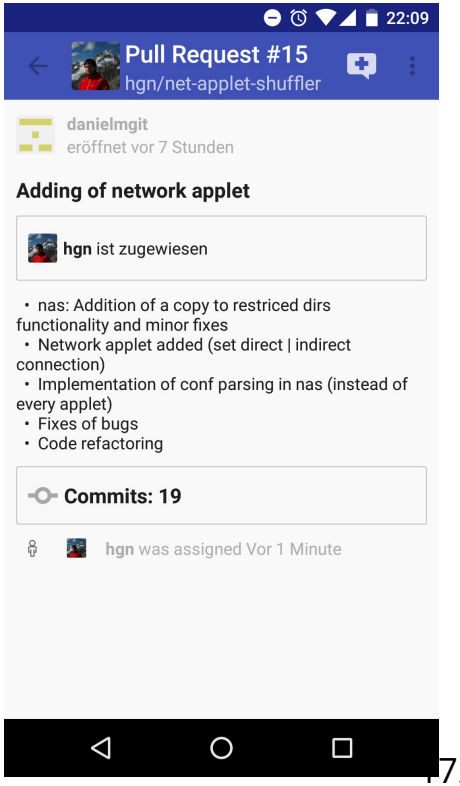
How to know when to merge?

A repository can point to several other repositories but when to merge?

The developer raise his hands using:

- GitHub, GitLab: **Pull/Merge Requests** are opened online
- Email: “new feature ready, please merge”
- Webhooks
- Phone call
- …

The integrator can also proactively fetch a remote repository!



Merge Process

Merge Remote Repository

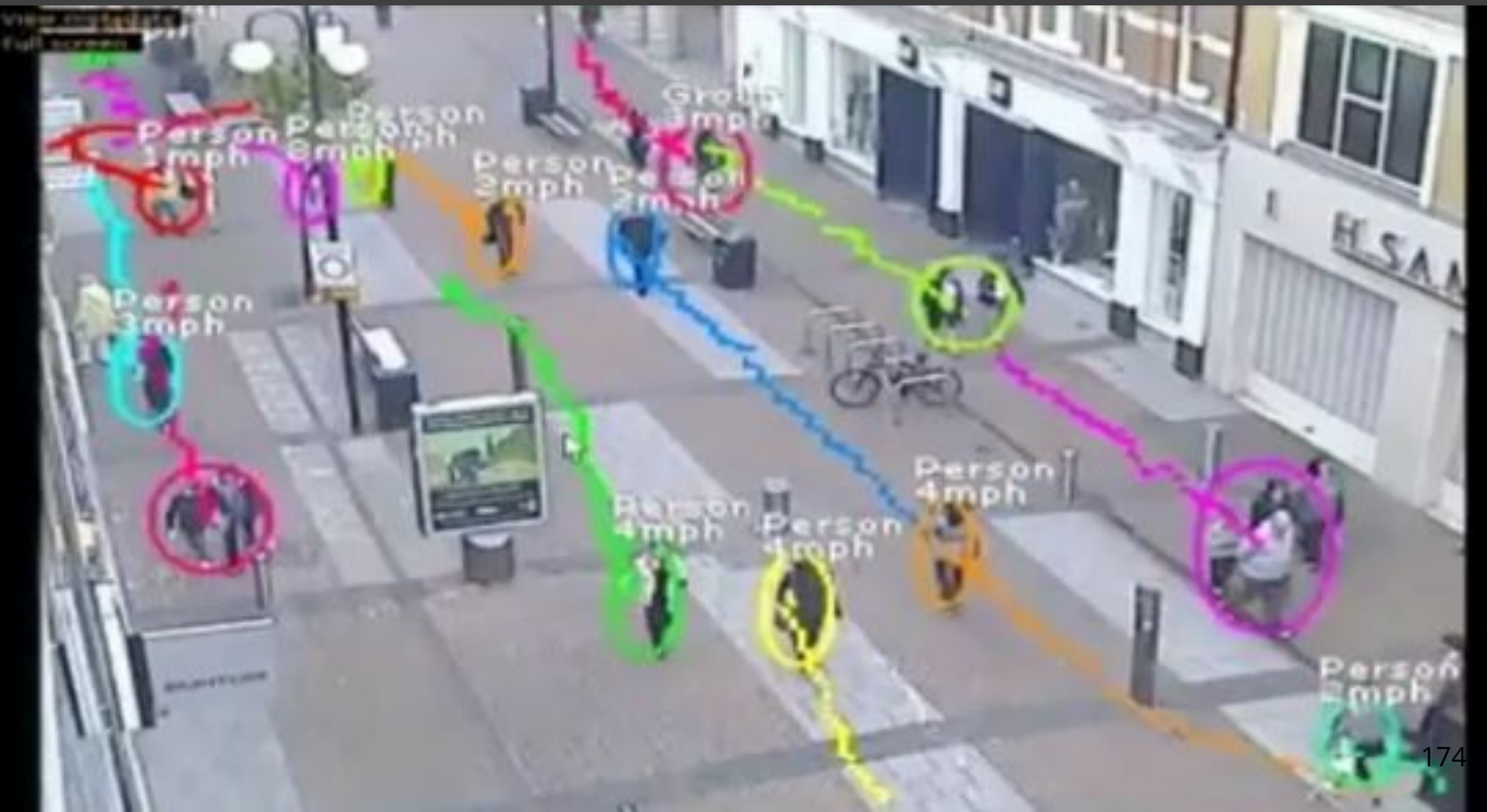
Fetch remote commits into local storage

```
$ git remote update
```

Merge branch **feature** from **dev-a** repository in our master branch

```
$ git checkout master
$ git merge --no-ff dev-a/feature
```

Tracking Branches



Tracking Branches

Make your Life more Comfortable

Local branches can have a **relationship** to a **remote branch** - if they have, the remote branch is called a **tracking branch**

Git operations like **pull** know exactly **where to pull from**

When you **clone** a repository it **creates** a master branch which **tracks origin/master**

Tracking Branches

Example

Let's create a bare repository and setup two remote repositories

```
$ mkdir kernel; cd kernel
```

```
$ git init
Initialized empty Git repository in /tmp/kernel/.git/
```

```
$ git remote add torvalds git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
$ git remote add miller git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git
$ git remote update
```

Tracking Branches

Example

Now we have the **content fetched**, including all objects, tags, remote branches

We still do **not have local branches** or a checkout working directory

```
$ git branch   ← show nothing
$ git branch -a
remotes/miller/master
remotes/torvalds/master
```

Checkout local branch and track upstream branch

```
$ git checkout -b upstream-miller-master miller/master
Checking out files: 100% (54444/54444), done.
Branch upstream-miller-master set up to track remote branch master from miller.
Switched to a new branch 'upstream-miller-master'
```

Note: checkout of remote master branches automatically creates a tracking branch!

Tracking Branches

Example

Just branch without modifying working directory

```
$ git branch upstream-torvalds-master remotes/torvalds/master
```

Tracked branches are recorded in local git configuration file:

```
$ tail -n 6 .git/config
[branch "upstream-miller-master"]
     remote = miller
     merge = refs/heads/master
[branch "upstream-torvalds-master"]
     remote = torvalds
     merge = refs/heads/master
```

Read: branch u*pstream-miller-master* is local, tracking remote branch refs/heads/master on remote miller (search for remote miller in config to find server url).

Resolving Others Merge Conflicts

Resolving Merge Conflicts of Others

Best approach: the developer who introduced the conflict resolve the conflict

Sometimes the developer is busy, in holiday and the particular branch must be merged. The integrator must resolve the conflict ...

This is addressed here!

Resolving Merge Conflicts of Others

Fetch the branch of the developer locally, `FETCH_HEAD` is a canonical name always pointing to the last fetched commit

```
$ git fetch https://codequal.rsint.net/alice/crypto-simulation.git master
```

Checkout branch you want to merge the changes in

```
$ git checkout master; git pull
$ git merge --no-ff FETCH_HEAD
Auto-merging mac/core/modules/slave.cc
Auto-merging mac/core/common/ticker.h
Auto-merging mac/core/common/AdapterPhyApi.h
CONFLICT (content): Merge conflict in mac/core/common/AdapterPhyApi.h
Automatic merge failed; fix conflicts and then commit the result.
```

Resolving Merge Conflicts of Others

Resolve any conflict and merge as usual

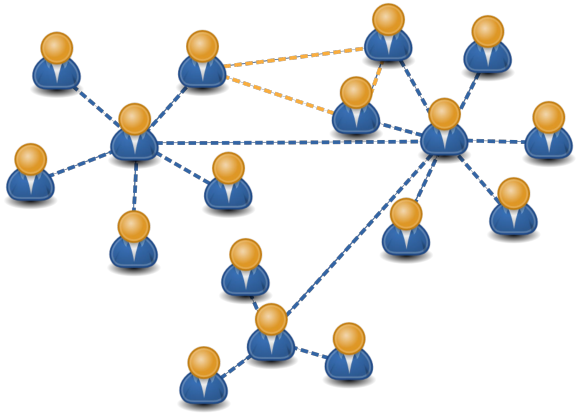
```
$ git mergetool
$ git add files
$ git commit
```

Branch and Developing Models

Branching Models

Introduction

Git is **distributed** by nature, but you **can use** it like a **centralized** SCM - git **does not enforce** how you develop!



Take your time: start with a simple model, reevaluate it and change if required

The concrete branching model is not incorporated into git - it is up to the developers, it is a **convention**, a policy that one repository is kind of special or two repositories, or the hierarchy, ...

Branching Models

Overview

Dictator model: there is one maintainer (integrator) who merges or refuse to merge particular commits.

Consensus driven model: every developer (or one person of a group, component maintainer) is able to merge but only if other developers acknowledge the functionality.

Component model: each developer has his access domain where only the developer has unrestricted push access. Other developers have to be granted permission for particular commits.

Test driven model: each developer can push changes if all tests are passed.

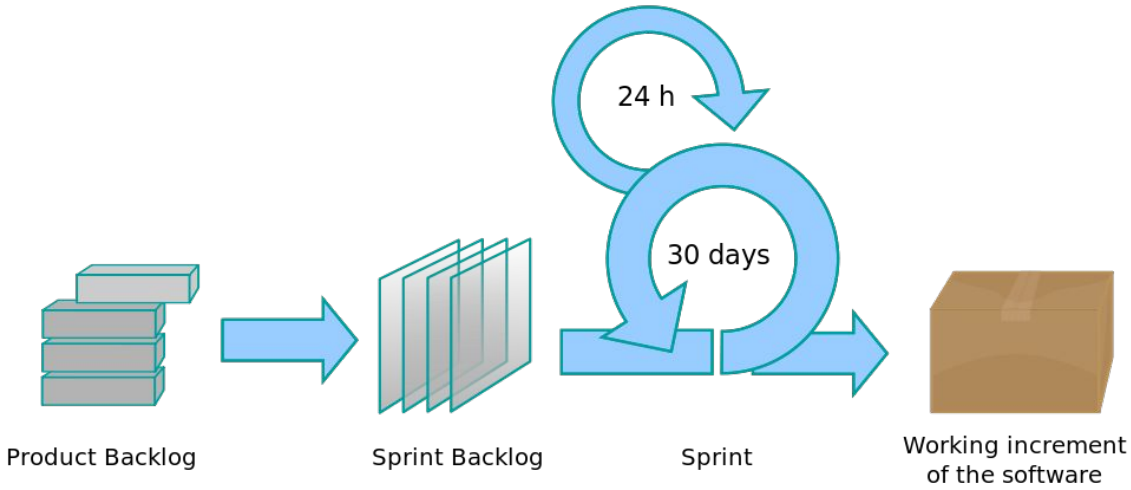
Or a mixture of some or all models!

Agile Software Development

We are not focussing on Scrum, Kanban or whatever

The upcoming branching models can be used for the majority of software development processes

One branching model doesn't fit every time - you can and should modify the presented model for your workflow



Branching Models

Introduction - II

Upcoming slides present two models:

- **Centralized** workflow
- **Decentralized** for agile software development

Branching Models

Introduction - III

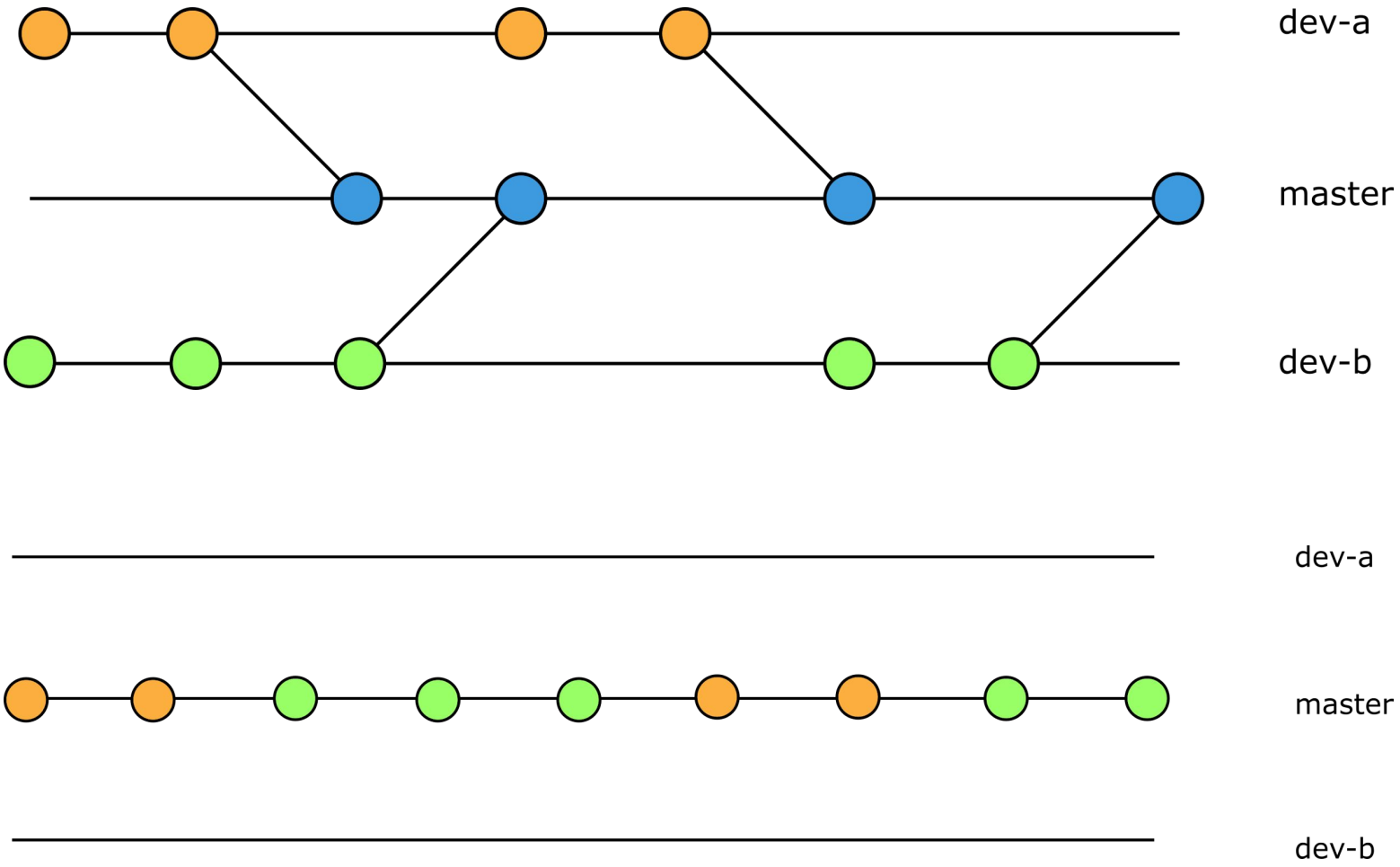
Important: the **branching model** and the **responsibility model** are two fully **distinct** dimensions

For example:

- You can drive the "only one master branch model" (aka SVN style model) and **every user can push** to this repo without limitations

- You can use the same one master model but **only the integration** manager can **push** to to the master branch!

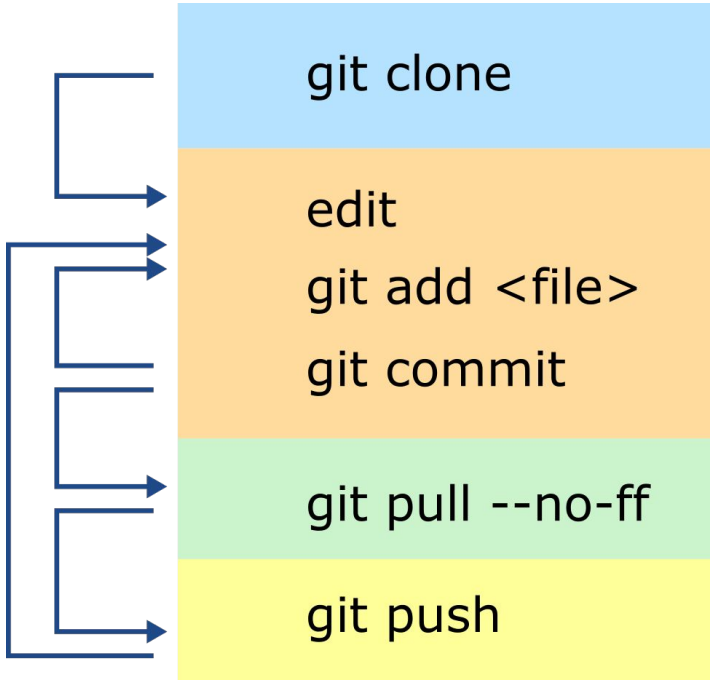
Garage Project Workflow



Garage Project Workflow

Consensus & Dictator Driven

Everybody can merge and change the master

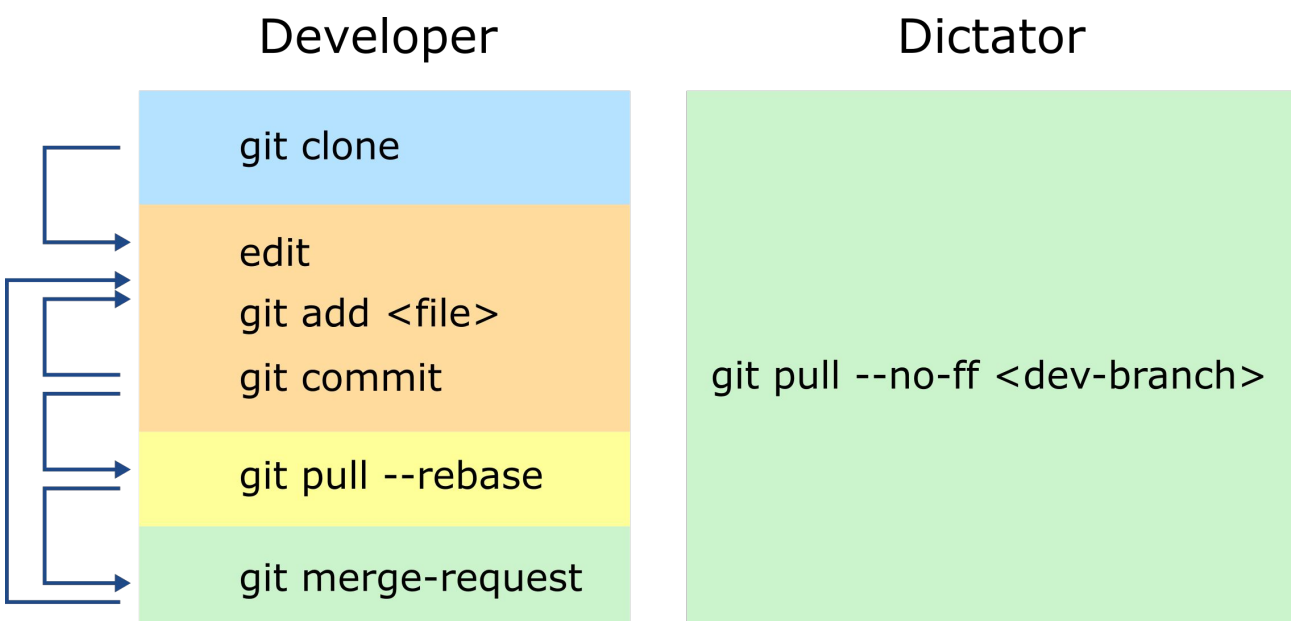


Garage Project Workflow

Consensus & **Dictator Driven**

Developers can request a merge, only integrator/manager can merge into master branch

Integrator/manager can be a group of people (sub-manager)



Centralized Workflow

Summary

Work is done on master branch

Suitable for small projects

With and without review

Enterprise Workflow

Multiple Releases supported (backporting required)

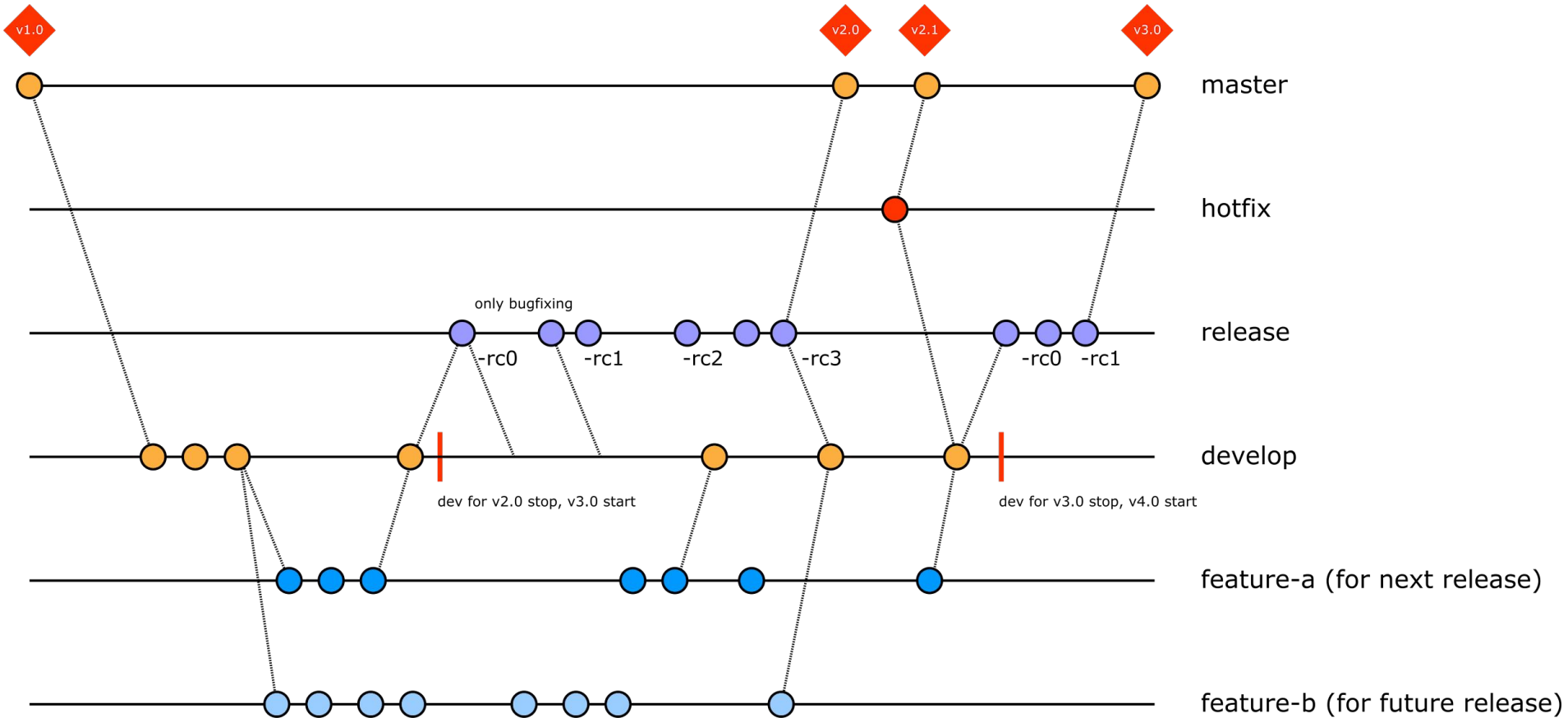
Several features are developed simultaneously by several groups

Hotfix required for older releases

Integration team involved

→ one development branch is not enough

Enterprise Workflow



Enterprise Workflow

Develop branch can be called **integration branch** - this is where nightly builds are generated from

Developer contributions are directly merge into **release**, **develop, hotfix** and **feature** branches

Feature branches merged into develop branch

Merges to master only via release or hotfix

Parallel development possible

Maintenance branches for stable releases possible

Practical Session - II

Log into Gitlab server

https://codequal.rsint.net

Ignore any "Upload SSH Key" messages - just upload via HTTPS!

Go to the example project page:

https://codequal.rsint.net/training/git

and fork this project (search for "Fork")

Clone this project:

```
git clone https://codequal.rsint.net/USERNAME/git.git
```

Add pointer to the upstream repo:

```
cd git
git remote add world https://codequal.rsint.net/training/git.git
git pull world master:master
git log --graph -p --decorate --all
```

Create a new topic branch and change to this repo:

```
git checkout -b <branchname>
```

Modify a random file, add to index and commit:

```
$EDITOR some/random/file.c
git add some/random/file.c
git commit -v
```

Push new commits to your newly created branch:

```
git push --set-upstream origin <branchname>:<branchname>
```

Online create a new merge request towards branch "world/master"

```
https://codequal.rsint.net/USERNAME/git
```

Periodically update your master branch by pulling world/master into your local master and rebase new changes into your local branch:

```
git pull world master:master
git rebase master
```

Undo Failures

Undo

Imagine

- You **deleted** a **branch** mistakenly and your work go the way of the dodo
- Wrong rebased mangled commits
- ...

Result: you lost hours of work!

Is there a way to recover lost commit and branches? -> Yes

Recovering

The Reflog

If you **delete a branch**, the branch **content** is **not deleted** immediately, rather the **reference pointer** in the `.git/refs` folder is deleted - not the commits itself!

For recovering one git construct is required: the **reflog**

Each time the tip of a branch or **references** are **updated** an entry describing the modification is **recorded** in the reflog

The reflog is ordered **chronologically**, last modification on top

Recovering

The Reflog

```
$ git reflog
f1f709d HEAD@{0}: pull: Fast-forward
0d99b7c HEAD@{1}: checkout: moving from master to upstream-lukas-master
0d99b7c HEAD@{2}: rebase finished: returning to refs/heads/master
0d99b7c HEAD@{3}: rebase: checkout upstream-lukas-master
2651a19 HEAD@{4}: checkout: moving from rename to master
00688e2 HEAD@{5}: checkout: moving from master to rename
2651a19 HEAD@{6}: checkout: moving from rename to master
00688e2 HEAD@{7}: commit: README: rename omnetta to zaytuna, rework text
```

Recovering

The Reflog

Search for the deleted branch, git show <ID> will show you the content of each commit

To **restore** a **branch** simple **checkout** the deleted <ID> again:

```
$ git checkout -b recovered-branch <ID>
```

Recovering

The Reflog

How to remove the last reflog pointers too?

```
$ git reflog expire --expire=1.minute refs/heads/master
```

Remove all reflog entries of the master branch older then one minute

Now you can call the **garbage collector** and all unreferenced commits will be deleted

Recovering
The Reflog

More unlikely, but still thinkable: you deleted branches manually (or a third party tool) and no reflog is written?

No panic: you can **list** the **raw database objects**, unreferenced by any branch and not referenced in the reflog:

```
$ git fsck --full --unreachable --lost-found
unreachable blob e3803017d0be9601ebb8fd694a30ad4ba5ab6b20
unreachable blob fa80794e22e8f291d6c6f5e4fadeb2d198c3dd08
unreachable commit f70145060d3ab822bef4a3eb2cbf758f9258d39e
unreachable blob 290270fce2b936307dd40dbb9465f1a1bdece852
unreachable commit a4028fd8f4cf80d20b406b6ce961b4095dd169d0
unreachable blob 4b84b6d6ad7a06905215b12aba4aa7e476e36be5
[...]
```

Splitting Work

Splitting Work

Structure Better Commits

Git **index** is a crucial component to **structure commits** based on context:

- **Bundle feature** A as commit 1, bundle feature B as commit 2, ...
- Each commit should be clean (compilable, testable, ...)

For example: extending the API and adding a test should go into two commits - at least in two commits, probably more

To sum up: **commited changes should be related**

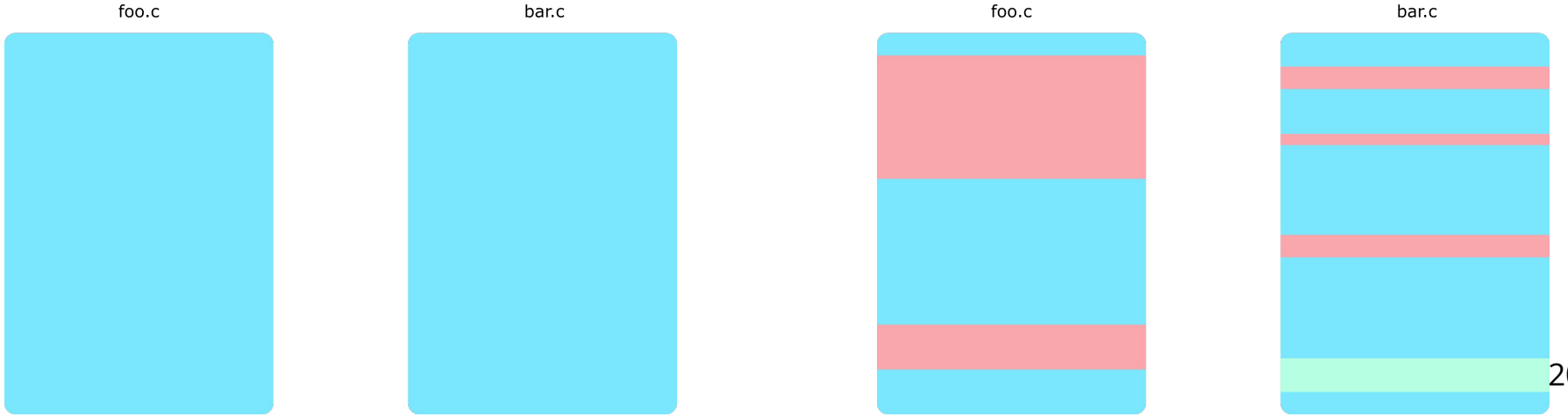
Splitting Work

Patch Add

But: sometimes you **forgot to structure** your work **chronologically** and two or more non-related changes are written to disk

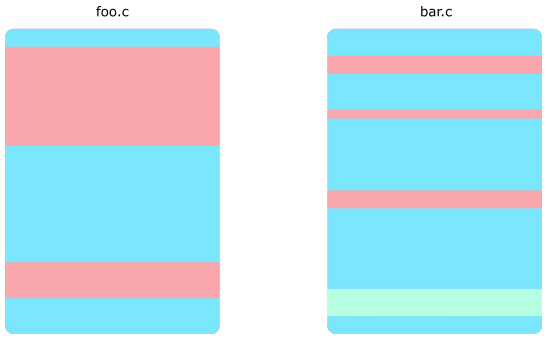
If changes are done in **separate files**: **no problem**! Just call git add and commit consecutively!

How to handle if two or **more logically** disjunct **changes** are in **one file**?



Splitting Work

Patch Add



Goal: commit red changes first, second commit remaining

```
$ git add foo.c                ← add all changes in foo.c to index
$ git add --patch bar.c        ← you select which parts go in for bar.c
```

Git will iterate of each hunk and let you decide if the part should be added to the index or not:

```
diff --git a/foo.c b/foo.c
-text before
+text after
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

Ignoring Files

Ignore Files

Generated object files, generated executables, temporary editor files, trace files et cetera: **all not tracked files will clutter your status** - not fine

To **hide** these **files** - git provides a mechanism to **ignore files**

.gitignore files are **versioned files**: if you check in these files all users will benefit of it

```
# general section
.*
*.o

# Editor files
*.orig
*~

# CScope files
cscope.*

# generated include files
include/config
```

Ignore Files

Directory Specific Ignore Files

Prefer to **add .gitignore** files for **last-level directory** - where the file to be ignored is located:

- If you move directories the .gitignore is also moved - no adjustment of root .gitignore file because path changes
- No overwhelming .gitignore files
- Subsystem maintainers can ignore their subsystem specific stuff

Place in the project **root directory** common file pattern which should be **excluded everywhere** (editor backup files, object files, etc)

Ignore Files

Delete all Ignored Files

After **removing** all **ignored files** and deleting all untracked files you have an environment identical to the **state after** initial **clone** of your repository

Great to **make sure everything** is **clean** - no artifacts will be left

Helps you if your build environment is not 100% reliable (i.e. C++ build dependencies)

```
$ git clean --force -d -x
```

Ignore Files

Generator

Gitignore.io provides **templates** for many programming languages, IDEs and other environments

Select one or multiple templates and save as **.gitignore** file

Submodules

Submodules

Introduction

Git submodules allows to **include other** git **repositories** - keeping history separated but synchronized with the container repository

Use cases:

- Shared content (e.g. C library used in several projects)
- Bundle several independent repositories in one large repository
- Repository management by someone else (you will probably upgrade from time to time, e.g. security fixes)
- Store sensitive data in separate repositories

Submodules

Concepts

Submodules are **tracked** by the **Commit ID** - not branches. The commit is **stored** in the **Container Repository**

Submodule repository and Container repository are really **loosely coupled**! If you change things in the submodule, the container repository is **not automatically updated**

Finally: using submodules add **additional complexity**! Coworkers must deal with it and as usual: **more commands, more cry!** :-)

Thinking outside the box: you can use a build tool or custom script to bundle several repositories! Think about how Linux distributions, Docker, ... solve dependency issues!

Submodules

Howdy World

Let's start with an example!

We first create a library repository (lib with two commit) which will later be integrated in a larger container project:

```
$ mkdir library-foo; cd library-foo
$ echo "int uberrand(void) { return 42; }" > librand.c
$ git init; git add .; git commit -m "initial commit"
$ echo "all: " > Makefile
$ echo -e "all:\n\tgcc -c librand.c\n\tar rcs librand.a librand.o" > Makefile
$ git add Makefile; git commit -m "provide helper makefile"
$ git remote add origin git@github.com:hgn/librand.git
$ git push -u origin master
```

Submodules

Howdy World

Now create the container project and include the library as a submodule:

```
$ mkdir strongswan; cd strongswan
$ echo "int main(void) { return uberrand(); }" > strongswan.c
$ git init; git add .; git commit -m "initial commit"
$ git submodule add git@github.com:hgn/librand.git librand
$ git status -s
A .gitmodules
A librand
$ cat .gitmodules
[submodule "librand"]
    path = librand
    url = git@github.com:hgn/librand.git
$ git commit -am "add external random library as submodule"
```

Submodules

Obstacles - Cloning

If you **clone** a repository with submodules the content is **not downloaded by default!**

```
$ git clone git@github.com:hgn/strongswan.git
$ ls strongswan/librand                                ← nothing, empty dir
```

The way to go:

```
$ git clone --recursive git@github.com:hgn/strongswan.git
```

Verbose version:

```
$ git submodule init
$ git submodule update
```

Submodules

Obstacles - Updating

Now try to update the rand library

```
$ cd librand
$ git fetch
$ git merge origin/master
```

Check what happens in the container project:

```
$ cd ..
$ git diff
--- i/librand
+++ w/librand
@@ -1 +1 @@
-Subproject commit 5ac8267fe563470a16e920bbe47e32860d103648
+Subproject commit 3de6526f8b792fab1304bd1d3a4034f1095e4502
```

Submodules

Obstacles - Updating

Instead of fetch & merge you can simple do

```
$ git submodule update --remote
```

Submodules

Cheat Sheet

"User of repo" - update all to the latest superproject snapshot:

```
$ git pull
$ git submodule update --init --recursive
```

"Maintainer of repo" - bump superproject to the latest sub-projects snapshot:

```
$ git submodule update --init --recursive --remote
$ git add subproject-foo subproject-bar
$ git commit -m "bump subproject to release major.minor.patchlevel"
```

Submodules

Obstacles - Modify Submodule In-place

You can **modify** the **submodule** and commit changes upstream. But this is more **complicated and error prone**!

I **suggest** to **update** the submodule repository in the **original repository**

If interested in updating repositories in-place:

https://git-scm.com/book/en/v2/Git-Tools-Submodules

Subtrees

The Newcomer

Subtrees are **similar** to submodules: let you include repositories within your container repository

Difference I - User Point of View

- No .gitmodule files
- "Enduser" do not even know anything about subtrees - they are 100% transparent for container repository users

Difference II - Implementation Point of View

- Submodules are links, subtrees are copies
- Submodule usage results in smaller overall repositories, subtrees are larger - with full history

Subtree support since May 2012 – 1.7.11

Temporary Safety - Stash

Git Stash

Imagine:

- you are in the **middle** of **adding a new feature**. E.g. "CO_2 reducer"
- Your manager called and you should **instantly fix** a **bug** on the **release branch**
- You are **far away** to **commit** files even you have no idea what to write into the commit message - you **cannot commit yet**

Preferable: temporarily interrupt your work, save filesystem state without much overhead, commit fix and continue work

This can be done with **git stash!**

Git Stash

Stash away every uncommited files:

```
$ git stash
```

Do whatever you want now, edit, cherry-pick, switch branches, ...

Restore the state before you typed git stash (make sure you are on the branch you want to apply the stash)

```
$ git stash pop
```

Git Stash

More Features

Git Stash can store **more than one snapshot**

They are **ordered** in a Stack Order (**last in, first out**). But picking specific stashes by number is possible

To **show** all saved **stashes**:

```
$ git stash list
```

Git Stash

More Features

To fetch the last element and delete from the stash stack:

```
$ git stash pop
```

To show the content of a specific stash:

```
$ git stash show <STASH>
```

Interactive Rebase

Interactive Rebasing

You want to **change/alter** the **history** of your commits - interactive rebasing is the way to go!

Use cases:

- You pushed 20 commits, a merge request is stalled because Commit 10 contains a Bug which must be fixed
- Commit 10 has a faulty commit message and must be reworded
- Commit 10 must be divided in two separate commits
- Commit 10 and 11 must be squashed into one commit

Interactive Rebase

```
$ git log --oneline
8003557 core: get rid of py float conversation
f1e3854 merge todo section
30847f9 function: avoid divide by zero error
f49c14a add percent output
1b88da5 function: add function alignment functionality
22c0313 core: add debug hints
3191d95 instruction: add more instruction to database
6e320e1 kernel-build: config have are probed ones -> remove
29dc609 core: raise warning if pygal is not installed
7bcf8e6 core: add more instruction to the database
```

Interactive Rebase

```
$ git rebase --interactive HEAD~5
pick 1b88da5 function: add function alignment functionality
pick f49c14a [function] add percent output
edit 30847f9 function: avoid divide by zero error
pick f1e3854 merge todo section
pick 8003557 core: get rid of py float conversation

# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
# If you remove a line here THAT COMMIT WILL BE LOST.
```

Interactive Rebase

```
Stopped at 30847f9... function: avoid divide by zero error
You can amend the commit now, with
        git commit --amend
Once you are satisfied with your changes, run
        git rebase --continue


$ vim foobar.c ← fix things
$ git add foobar.c
$ git commit --amend
$ git rebase --continue
Successfully rebased and updated refs/heads/fix.
```

Searching & Blaming

Search the Repository

Git provides a build-in **grep** command, advantages to grep(1):

- Comes bundled with git for Win, Linux, Mac, ...
- Search by default only **registered files**
- Search in parallel (number of cores) → **fast**
- ... implement a **lot of features** useful for programmers! ;-)

Search the Repository

Search for foo and bar in one file but not necessary on the same line:

```
$ git grep -l --all-match -e foo -e bar
```

Show “break” (newlines) between matches, show function signature, show 10 lines before, 5 after match, show filename and line number:

```
$ git grep --break --show-function \
  --before-context 10 --after-context 5 --heading --line-number
```

Show the whole C/C++ function surrounded where time_t is used in all C and Header files:

```
$ git grep --break --function-context 'time_t' -- '*.[ch]'
```

Git Blame

Annotates each **line** in the given file with **information** from the revision which **last modified** the line

You are interested in one particular line: git blame show **who**, **when** and **why**™ the code was introduced

Usages:

- Find a Bug
- Want to understand code, why was function changed

Commits starting with **^ID** means this line was in the first commit (committer info probably incorrect)

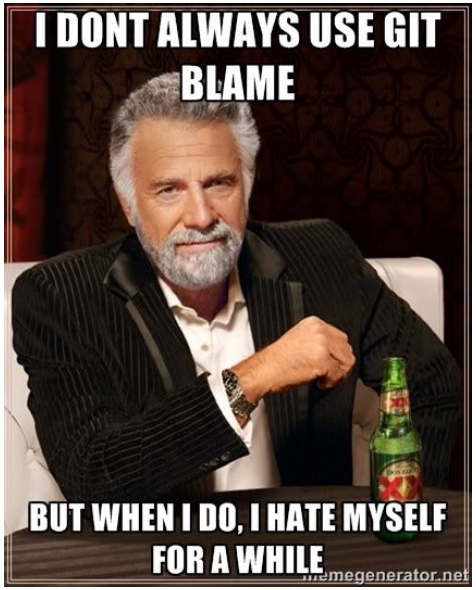
-C resolve **renames**. If lines come from different files (e.g. refactoring) you find where line come from - really helpfull

Git Blame

Annotates each line in the given file with information from the revision which last modified the line

```
$ git blame <filename>
```

```
2009-01-16 14:27 Jonas Fonseca     4bb9536          INPUT_CANCEL
2009-01-16 14:27 Jonas Fonseca     4bb9536      };
2009-01-16 14:27 Jonas Fonseca     4bb9536
2009-01-16 14:27 Jonas Fonseca     4bb9536  165 typedef enum input_status (*i~
2009-01-16 14:27 Jonas Fonseca     4bb9536
2009-01-16 14:27 Jonas Fonseca     4bb9536      static char *prompt_input(con~
2009-01-16 14:27 Jonas Fonseca     4bb9536      static bool prompt_yesno(cons~
2006-05-15 03:50 Jonas Fonseca     6706b2b
2009-02-22 01:19 Jonas Fonseca     91e8041  170 struct menu_item {
2009-02-22 01:19 Jonas Fonseca     91e8041          int hotkey;
2009-02-22 01:19 Jonas Fonseca     91e8041          const char *text;
2009-02-22 01:19 Jonas Fonseca     91e8041          void *data;
2009-02-22 01:19 Jonas Fonseca     91e8041      };
2009-02-22 01:19 Jonas Fonseca     91e8041  175
2009-02-22 01:19 Jonas Fonseca     91e8041      static bool prompt_menu(const~
2009-02-22 01:19 Jonas Fonseca     91e8041
2006-05-10 22:16 Jonas Fonseca     03a93db      /*
2009-02-18 11:47 Jonas Fonseca     b2ff9a4       * Allocation helpers ... Ent~
2009-02-18 11:47 Jonas Fonseca     b2ff9a4  180  */
2009-02-18 11:47 Jonas Fonseca     b2ff9a4
2009-02-18 11:47 Jonas Fonseca     b2ff9a4      #define DEFINE_ALLOCATOR(name~
2009-02-18 11:47 Jonas Fonseca     b2ff9a4      static type *
[blame] tig.c - line 172 of 7494 (2%)
```



Pointing at Things

Pointing at Things

One Commit

A specific commit:

```
90c337da152
```

One commit before a specific commit:

```
90c337da152^
```

Two commit before a specific commit:

```
90c337da152^^
```

23 commits before a specific commit:

```
90c337da152~23
```

Pointing at Things

One Commit

The most recent commit on the actual branch:

```
HEAD
```

One commit before the most recent commit on the actual branch:

```
HEAD^
```

Two commits before the most recent commit on the actual branch:

```
HEAD^^
```

23 commits before the most recent commit on the actual branch:

```
HEAD~23
```

Pointing at Things

The commit 5 minutes ago on the actual branch:

```
HEAD@{5 minutes ago}
```

One commit from yesterday on the actual branch:

```
HEAD@{yesterday}
```

Commit 1 year and 6 months ago on the master branch:

```
master@{1 year 6 months ago}
```

Pointing at Things

Naming Things

Make the commit ID more human friendly:

```
$ git describe c8507fb23
v4.2-rc3-181-gc8507fb235
```

Based on the ID the **nearest annotated tag** followed by the **number of commits** until described **commit id**

Reverse operation: human readable to exact commit id:

```
$ git rev-parse c8507fb23
c8507fb235bea3314a02a67ddda0d4e6cf01fa78
```

Naming Things

Tags

Sometimes your particular software state must be **labeled/marked** in some fashion - because it is **special** in some way: this is where tags come into play

Technically you need no tags - each commit ID is exact and unambiguous

Tags help to differentiate a state at a higher level. E.g. because the commit is a release commit. Tags are a kind of communication

Git support two kinds of tags:

- **Reference tags**
- **Annotated tags**

Tags

Annotated Tags

Create annotated tag:

```
$ git tag -a -m "v6.0" v6.0
```

Show tags

```
$ git tag
```

Tags must be pushed explicitly:

```
$ git push --tags
```

Tags

Signed Annotated Tags

Commits (thus the content, the history) are **cryptographically secured**. But how can you **trust** that a given **repository** is the desired repository? What about MiTM attacks?

Git tag provides possibility to **sign** a **tag** cryptographically by using **GnuPG** (web of trust)

Sign with your default key (`user.signingkey`)

```
$ git tag -s -m "release 23" v23
```

With explicit GnuPG key:

```
$ git tag -u 'key-id' -m "release 23" v23
```

Tags

Verify Signed Annotated Tags

To verify a signed tag:

```
$ git tag -v v23
```

Public key cryptography: you need the public key of the signer to verify the signature. And as usually you should trust the public key blindly: a) you personally verified the key or b) your web of trust rank the key as trusted

Tags

Signed Everything

Signing tags make sure a specific (trusted) person signed exactly **that tag**. There is no trust guarantee that commits are from a trusted person

Git allows you to sign each individual commit

```
$ git commit -S -m "implement feature foo"
```

Signing every commit (crazy mode)

```
$ git config --global commit.gpgSign true
```

Since Git 1.7.9

Tags
Final Words

Show all Git tags where a particular commit is included:

```
$ git tag --contains <commit>
```

Archive

Export Repository

Extract files for e.g. make it available for download

```
$ git archive --prefix="$(basename $(pwd))-$(git describe)"
--format tar.gz HEAD -o "$(basename $(pwd))-$(git describe).tar.gz"
```

Generates

```
→  command-control-rest-daemon-v1.0.tar.gz
```

Repo & Branch Naming Convention

General Naming

Branch names are normally **lowercase**, names are **separated** by **dash** (-):

stable-swift-3.0-merge-branch

migrate-user-data-dir

Git repositories follow this name convention:

net-next

protobuf-gradle-plugin

Repo & Branch Naming Convention

Branch Naming Pattern

master

release/{4.2, 4.3, 4.4, 4.5, ...}
users/{foo, bar, ...}

feature/{foo, bar, ...}

test/2017-05-14/foo

Do not use plural for category (e.g. "releases")

Large Files

Large Repositories

Larger repositories will **slow down** git **operations**

Git has **no artificial** size **limitation**. Repos larger than GByte are possible as well as millions of objects

Due to the inner working, operations becomes **slow** depending on **file size**, **number of files** and **commits**

To reduce the number of files and/or commits **splitting** in several repositories is a recommendation

Large Repositories

Microsoft Windows

Microsoft source code size: 270 GiB (Linux kernel 5.9: 2.1 GiB)

Microsoft Windows versioned with Git

```
Clone:    12 hrs
Checkout:  3 hrs
Status:    8 min
Commit:   30 min
```

Large Repositories - II

Git Internal Show Stopper

What makes git slow?

- **No secondary** path-to-id **index** (reverse index). **git blame** for example needs to get over all commits to see which id modifies a file

- Git uses **lstat**() to check if a file was modified. For 1000000 files this is a lot of IO. The inode cache helps when warm

- **Index file** is also a problem to some extend. It is large (e.g. 100MB) and must be rewritten at each modified repository access

Large Repositories - III

Sometimes projects contain **huge files**, and a **repository split** is **not** an **option**

- **Large texture files** (PSD), i.e. game development
- **FPGA images** must be treated as code (non-reproducible due to non-deterministic synthesis process)
- **Binary blobs** like file database (SQLite)

Git LFS

A way to Cope with Huge Files

Git Large File Storage (LFS) - an git **extension**

Several **predecessors** (git annex and others)

Installation:

Packages for *nix and M$ available at
https://git-lfs.github.com/

Setup git-lfs once:

```
$ git lfs install
```

Git LFS

Licence and Server Backend

Git LFS client is **open source** (MIT licence), server reference implementation available too

GitHub, GitLab support LFS out of the box. Third party backends available too, e.g. Artifactory.

Git LFS

Working with large files

Track large files

```
$ git lfs track '*.bin'
$ git add .gitattributes
$ git commit -m "handle bin file by LFS"
```

This must be done explicitly for every file or file extension that should be handled by git lfs

Remaining git flow is untouched:

```
$ git add image-foo.bin
$ git commit -m "add large image to repo"
```

Git LFS

Sidenote: Exclusive Locks

Resolving merge **conflicts** for **binary files** is not possible. The usual way for centralized SCM systems is to lock files ("ClearCase reserved checkout")

In a vanilla **distributed** environment **exclusive, global locks** are not possible

Try to **avoid** a workflow where locks are required. A locked environment **does not scale** for >=2 developers working simultaneously on one piece of code

If you maintain several branches merge capability is often a must

Git LFS

LFS Feature: Exclusive Locks

LFS starting with version 2.0 implement file locking!

Usage is straightforward:

```
$ git lfs track "*.jpg" --lockable
$ git lfs lock images/foo.jpg
$ git lfs unlock images/foo.jpg
```

Show locked files:

```
$ git lfs locks
```

Git LFS

Under the Hood

LFS is based on core git functionality: **git filters**

- Two commands: **clean** and **smudge** is called when you add objects to repository or check files out

Git LFS stores **pointer files**, not objects directly

- You fetch only the objects when needed: checkout a particular version, etc. A git pull **do not fetch all** LFS **blobs**. Just the checked out revision

Git LFS objects are stored in a separate DB server

"Pointer files" content:

```
version https://git-lfs.github.com/spec/v1
oid     sha256:48129f938184..
size    8428238932
```

Git LFS

Under the Hood - II

```
$ cat .gitattibutes
*.bin filter=lfs diff=lfs merge=lfs -text
```

A new **folder** within .git is created: .git/**lfs**

- This is where LFS objects are stored and cached locally

A **pre-push hook** is activated to **check** that git-lfs is installed

Git LFS

Tips

Following command will download all objects younger than 7 days for all local branches. If you work in a team this command is handy.

```
$ git lfs fetch --recent
```

You can configure the LFS download characteristics manually

```
$ git config lfs.fetchrecentalways "true"
$ git config lfs.fetchrecentrefsdays 7
```

The build server may not need every object, excluding may become handy

```
$ git lfs fetch --include images/** --exclude videos/**
```

Git LFS

Converting existing Repository

Already existing repositories may be **migrated** too - but keep in mind that you **kill the history** to some degree!

`git-filter-branch` can be used to do such modifications - but, it's **hard** to get it right

Use the **BFG repository cleaner** to modify the repository:

```
$ bfg --strip-blobs-larger-than 100M ...
```

VFS (Virtual File System) for Git

Microsoft Way to Handle Large Repos

VFS for Git virtualizes the file system beneath your git repo so that git and all tools see what appears to be a normal repo, but VFS for Git only downloads objects as they are needed. VFS for Git also manages the files that git will consider, to ensure that git operations like status, checkout, etc., can be as quick as possible because they will only consider the files that the user has accessed, not all files in the repo.

Even for huge repositories git operations are quite fast!

E.g. `git status` on Windows repo: 4 seconds

Project was formerly known as GVFS (Git Virtual File System)

GitCare

Health Checks for your Repo

Database Corruption

Unpacked Objects

- Disks are not bulletproof - bitrot is possible
- A malicious attack can change code

Git does **not regularly check** your repository for inconsistency - which requires recalculating the SHA1 for every object. But you can do this **manually** or **automated** on the server at night using a cron job or systemd timer event

```
$ git fsck --full --strict
error: sha1 mismatch 32854a8a3dc183fe85e4be4fcf338852c8fa53bd
error: 32854a8a3dc183fe85e4be4fcf338852c8fa53bd: object corrupt or missing
Checking object directories: 100% (256/256), done.
missing blob 32854a8a3dc183fe85e4be4fcf338852c8fa53bd
```

Database Corruption

Unpacked Objects

For **corrupted, unpacked objects** simple ask a coworker or take the non-corrupted blob from your backup and **copy it** to `.git/objects/[corrupted-object]`

That's all!

Database Corruption

Packed Objects

If bits are rotted in the pack file errors will show up differently:

```
$ git status
error: packfile .git/objects/pack/pack-cc820c084.pack does not match index
error: packfile .git/objects/pack/pack-cc820c084.pack does not match index
error: packfile .git/objects/pack/pack-cc820c084.pack does not match index

$ git verify-pack -v .git/objects/pack/pack-cc820c084.idx
[...]
.git/objects/pack/pack-cc820c084.pack: ok
```

Database Corruption

Packed Objects

For packed file corruption:

```
$ rm .git/objects/pack/pack-cc820c084.pack
$ cp $coworker/pack-cc820c084.pack .
$ git unpack-objects -r < pack-cc820c084.pack
$ git fsck
```

Pro Tip: don't forget to run `memtest86(1)` and check the status of your disks via `smartctl(1)` ;-)

Garbage Collection

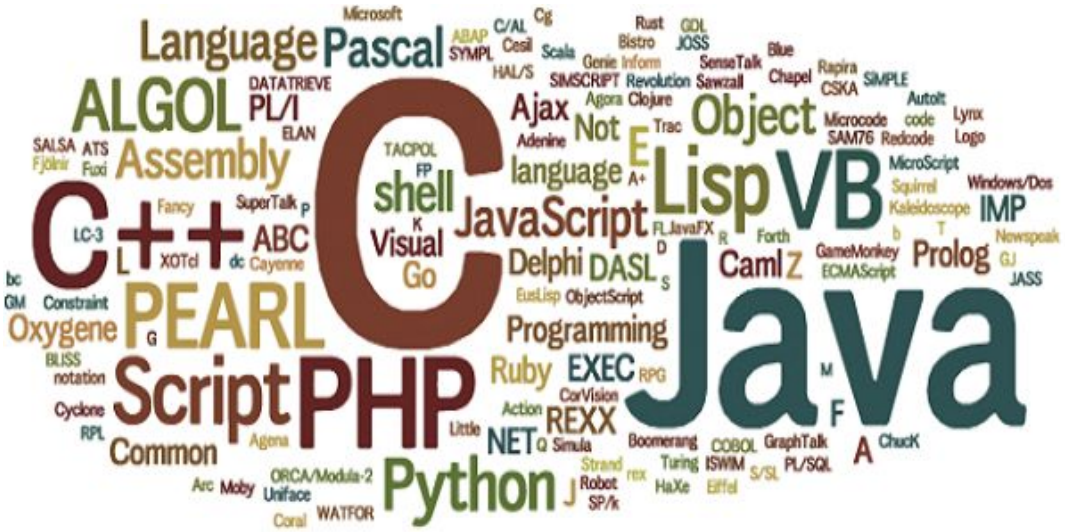
It can happen that you have objects in the repository but it is not reachable, either

- No tags pointing to the commits
- No branch pointing to the commits

`git gc` runs over all objects and look if it is **referenced**, **if not** the object is **deleted**

```
$ git gc
```

Language Bindings and Libgit2



Language Bindings

Libgit2

In the early days **Git** was just a **client application**. Written in **C** and a couple of **shell scripts**. This was ok for a while but this model had some **problems**:

- GUI developer had to code their own git fundament (heavy) or to **execute** (subprocess) the reference client and **parse the output**
- **Porting** git to other architectures **was complicated**
- More problems exists

After a while and some tries an independent low level C library is available:

libgit2

https://github.com/libgit2/libgit2

Language Bindings

Libgit2

Features includes:

- **Cross platform:** Linux, *BSD, Mac OS, iOS, Amiga, MinGW and native Windows
- **C89**: build with gcc, clang and MSVC
- **Thread Safe**
- **Zero dependencies** - no crypto library, nothing
- **GPL2 licensed** with Link Exception (“less” restrictions compared to LGPL)

Supported languages:

- Python, Ruby, Objective-C, Swift, Lua, Perl, Node.js, ParrotVM, Go, C++QT, Erlang, D, Java, PHP, .NET, Rust, ..

A rich ecosystem waiting to develop a broad infrastructure of tools

Language Bindings
Examples

- GUI Tools
- Server Backend Application
- Continuous Integration Tools
- Coding Guideline checker
- Repository statistics
- ...

Language Bindings

Python Example

```
import json
import sys
from datetime import datetime
import pygit2

def main(repository):
    repo = pygit2.Repository(repository)

    commits = []
    for commit in repo.walk(repo.head.oid, pygit2.GIT_SORT_TIME):
        commits.append({
            'hash':                 commit.hex,
            'message':              commit.message,
            'commit_date': datetime.utcfromtimestamp(
                                    commit.commit_time).strftime('%Y-%m-%dT%H:%M:%SZ'),
            'author_name':          commit.author.name,
            'author_email':         commit.author.email,
            'parents': [c.hex for c in commit.parents],
        })
    print(json.dumps(commits, indent=2))


if __name__ == '__main__':
    main(sys.argv[1])
```

Email Workflow

Email Workflow

So far we talked about a **HTTP server** exchange **workflow**:

- Every commit is **pushed to a server** and coworkers **pulled** the work **into** local **branch**
- **GitHub/GitLab** workflow is more specialized because you can even do **some operation** on the **Web frontend** (like merging branches by pushing fancy buttons)

But **HTTP** is **not** the **only way** to exchange code!

The Linux kernel, Git itself and many other projects still use a more antiquated workflow by using plain **Emails** as the exchange medium:

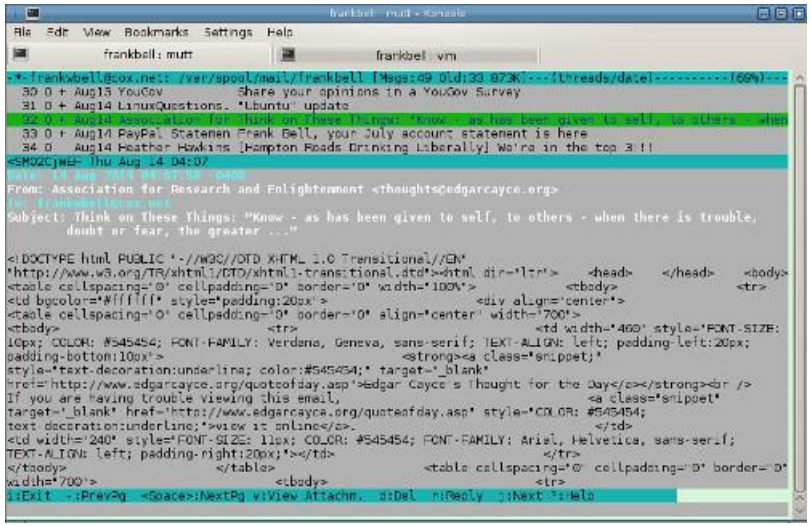
- The commit message is the Email body itself
- The patch is an Email attachment

Setup Email

Preparation

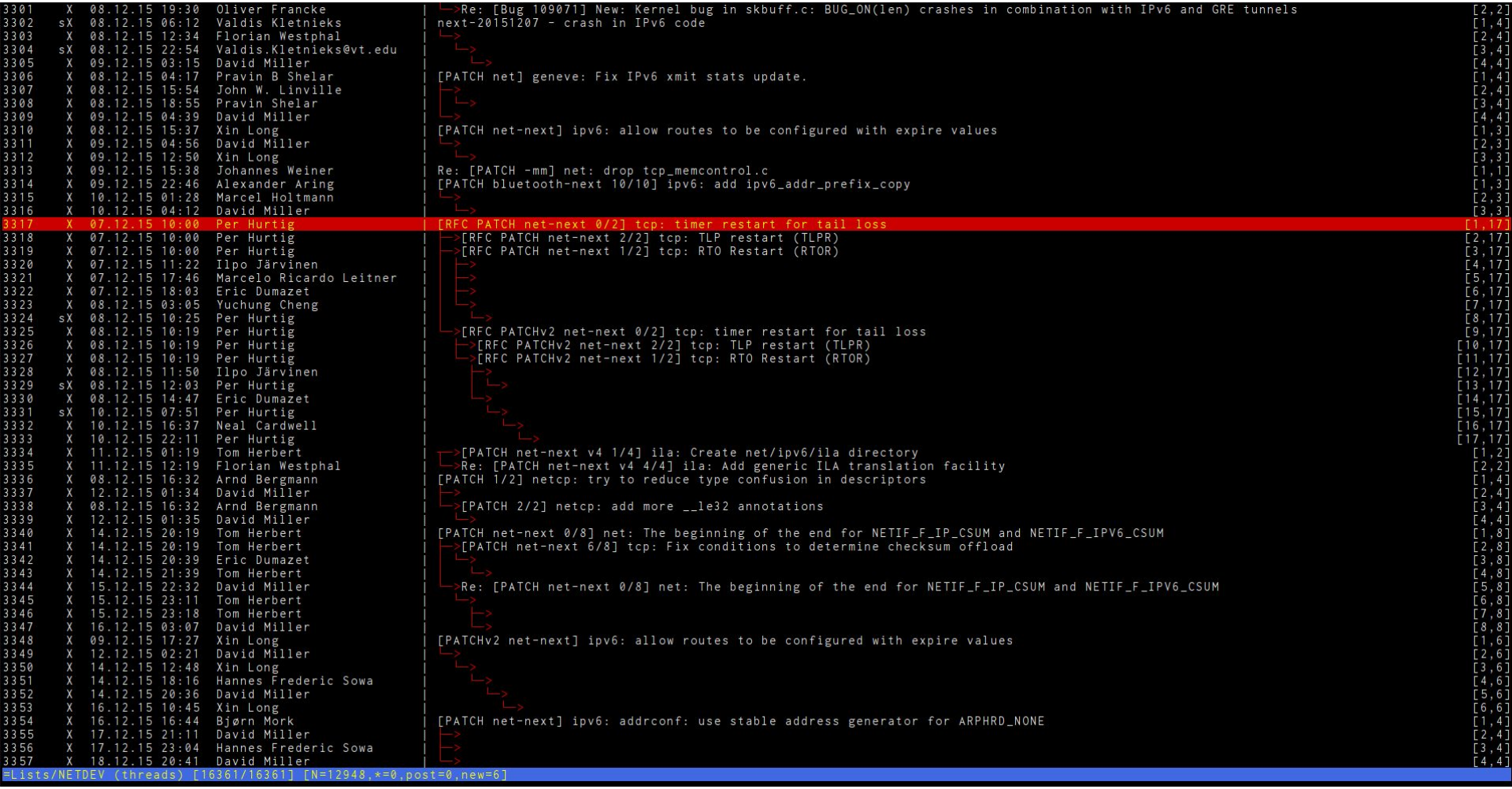
- **RX**: your email client should be able to save an email (mbox, maildir format)
- **TX**: ordinary SMTP support (if you use Thunderbird, Apple Mail everything should be fine)

Be aware of the following problem: some email clients mangle attachments



Linux Kernel Development

Exemple: Netdev Mailing List



Used mail user agent: mutt

Prepare Emails

Create patches (here: last 3 commits) and save in outgoing dir:

```
$ mkdir outgoing
$ git format-patch -o outgoing -3
```

Different prefix:

```
$ git format-patch --subject-prefix="RFC" -o outgoing -3
```

With a special cover letter for larger commits

```
$ git format-patch --cover-letter -o outgoing -3
```

Send Emails

SMTP Credentials

Configure Git to know you SMTP host and configuration, here for Gmail

```
$ git config --global sendemail.smtpserver smtp.gmail.com
$ git config --global sendemail.smtpserverport 587
$ git config --global sendemail.smtpencryption tls
$ git config --global sendemail.smtpuser your_email@gmail.com
```

Send Emails

Send all commits saved in directory outgoing:

```
$ git send-email --to <email> outgoing/*
```

For password protected SMTP server:

```
git send-email --smtp-pass 'PASS' --annotate --from "Daniel Metz
<dmetz@mytum.de>" --to netdev@vger.kernel.org outgoing/*
```

For SSL/TTL SMTP transport: additional Perl packages are required: community/perl-authen-sasl community/perl-net-smtp-ssl

Resend Patches

You can change the prefix too:

```
$ --subject-prefix "PATCH v2"
```

Reword the patch before sending it:

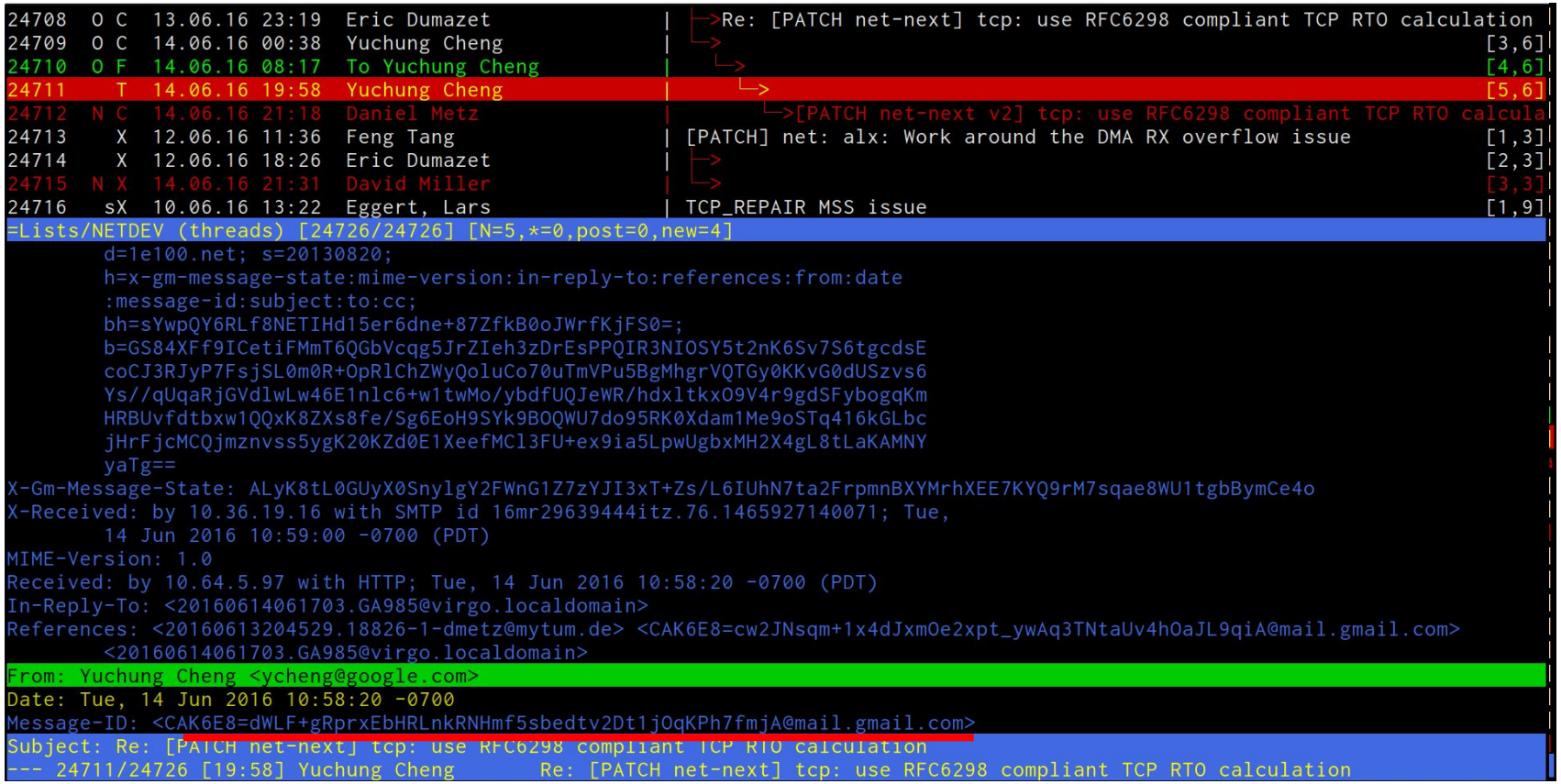
```
$ --annotate
```

And write what you changes in version v2 under the three dashes:

```
Changes v1 -> v2:
 - removed foobar
 - added zlib-dev
```

Resend Patches

Anchor at Specific Position



Anchor to specific threat

```
--in-reply-to='CAK6E8=dWLF+gRprxEbHRLnkRNHmf5sbedtv2Dt1jOqKPh7fmjA@mail.gmail.com'
```

Receiving Patches

Receiving patches via Email is simple:

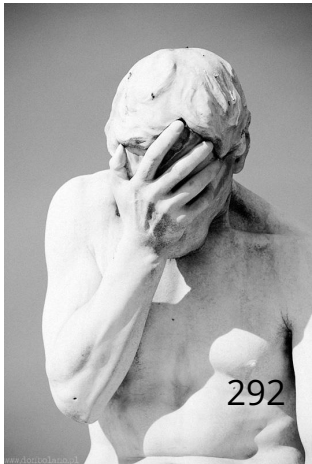
- Save particular Email as mbox/mailbox
- `git am <file | directory>`

SMTP/IMAP

`git send-email` works with **Gmail** too! Linus Torvalds is using Gmail

Many email clients are capable to **deal with Git**. See `linux/Documentation/email-clients.txt` if your MUA is supported

Same document, **Lotus Notes** text: "**Run away from it**"



Switching to Git

Rules of Thumb

There is no master plan nor automated scripts converting from tool $Foo to Git

Take your time!

- Get **familiar** with Git, use it for a while, try it, get comfortable
- Want to use **submodules**: discover dos and don'ts

Take the **chance** to **restructure** your source code

- **Avoid** huge repositories, **split** at certain level. Consider: access rights, library boundaries, are components reused in several projects? Submodule candidate?

Rules of Thumb - II

Rethink build system and versioned files

- Versioned binary artifacts?
- Versioned generated files?
- → clean up - now or never

That's It!

Any Questions?

hagen@jauu.net

More Practice Required?

GitHub provides an online tutorial to train the basic commands

https://try.github.io

Appendix
Vim Keys

Switching between modes:

```
i   → insert mode (for text editing)
ESC → command mode (for commando operations)
```

In command mode:

```
:w  → write file
:q  → quit vim
u   → undo
/   → search forward
```