GIT

- Introduction
- Architecture
- Basic Workflows
- Useful commands
- Discussion

References

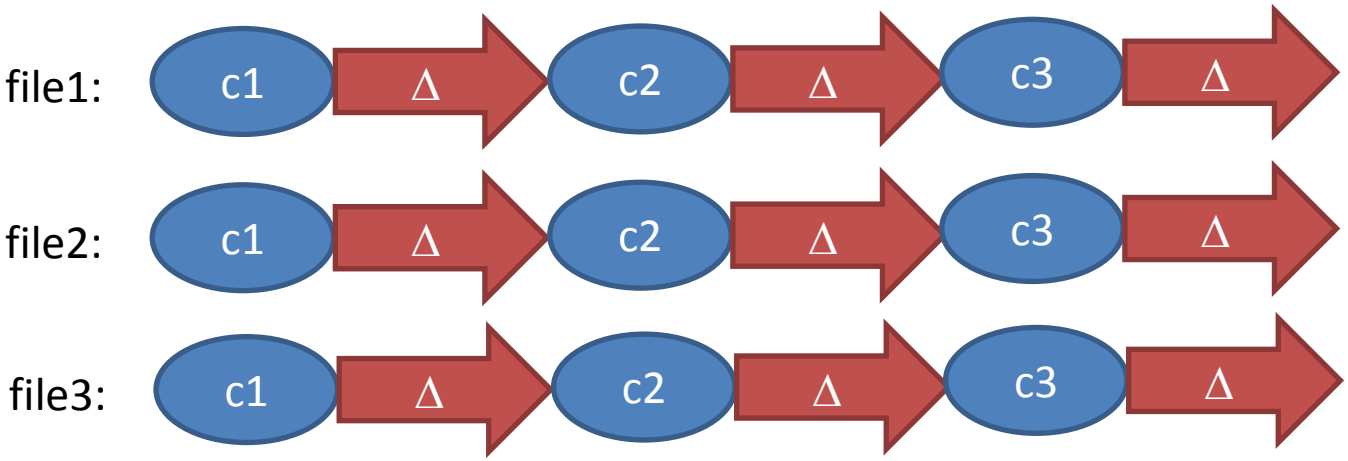
- John Wiegley: *Git from the bottom up*
  a lot is taken from that source. PDF available online.
- Scott Chacon: *Pro Git*
  PDF book available online. Complete reference to (almost) all aspects of GIT
- A lot of Git cheat sheets are available online

GIT Advantages

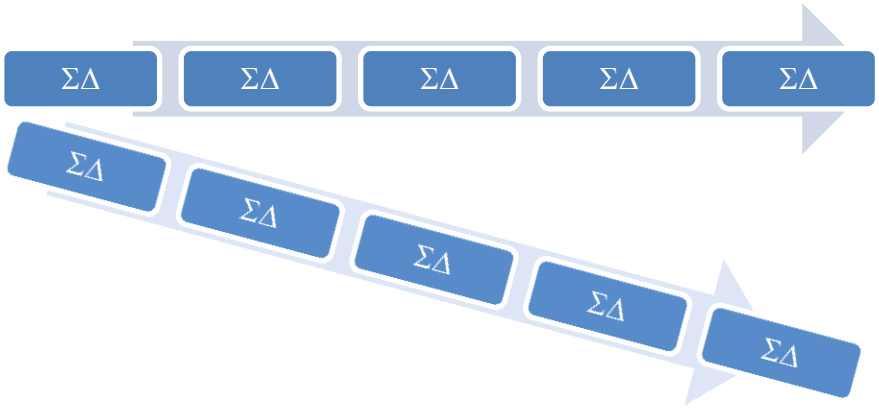
- all metadata in one directory (usualy .git), not one meta-directory in each directory
- simple, consistent, powerful model
- fast, robust
- supports much more workflows than traditional vcs (CVS, SVN)

Well Known: CVS/SVN

- idea: sequence of deltas of a file

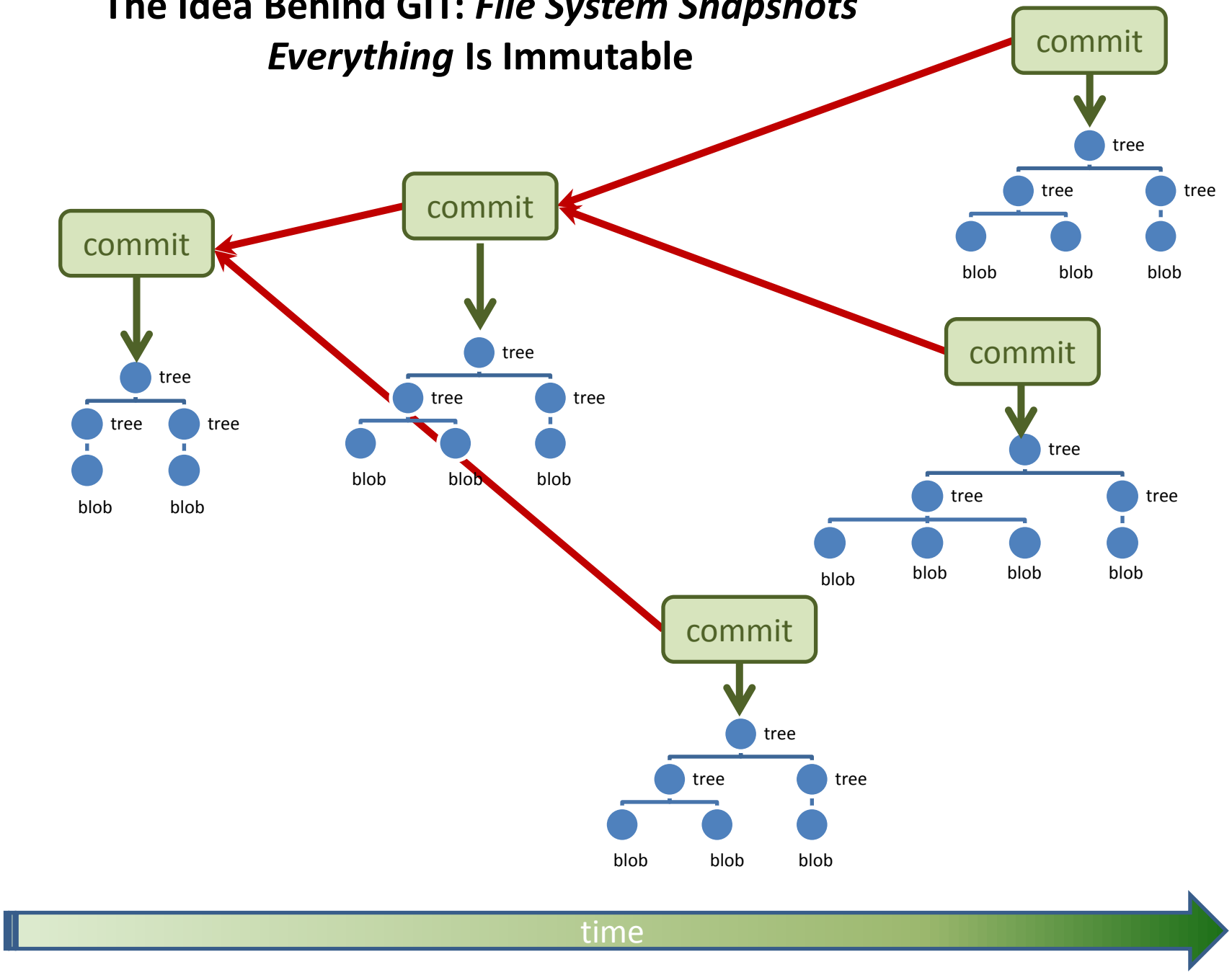


- branches: a complex other construct



The Idea Behind GIT: *File System Snapshots*

Everything Is Immutable



GIT Terminology 1

Recursive definitions …

working tree:

a directory to which a repository is associated with (usually the sub dir .git).

repository:

a collection of commits. It defines HEAD.

It contains branches and tags, which give names to commits.

commit:

a snapshot of the working tree at some point in time. The commit called HEAD, when the commit was made, becomes the commit‘s parent. Thism defines the history of this commit.

HEAD:

the commit the working tree refers to at this moment

GIT Terminology 2

Further definitions …

branch:

a name for a reference to a commit. The parents define the history of the branch: a „branch of development“.

tag:

a name for a commit. Thus always names the same commit.

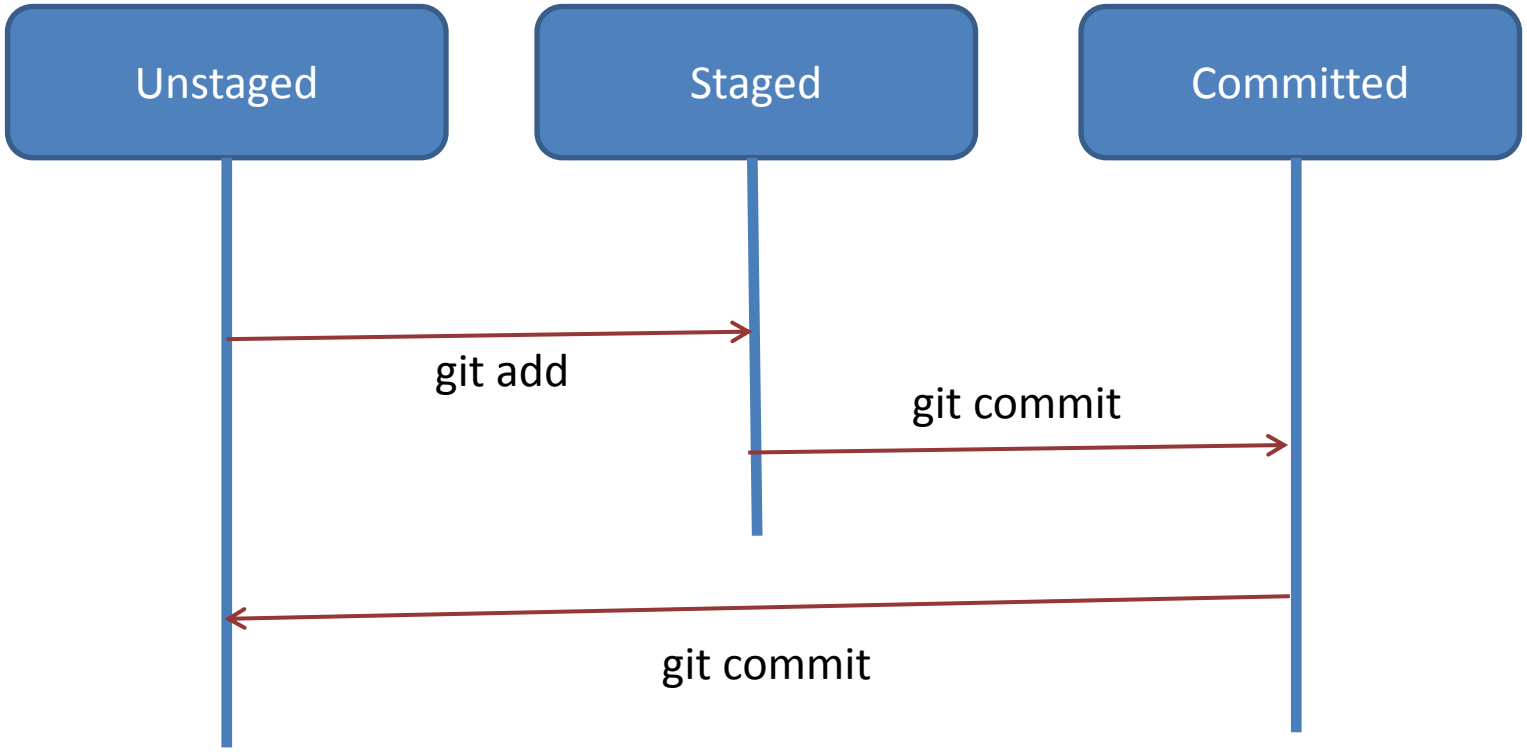
master:

by convention the name of a branch, in which the mainline of development is done. In no way special.

index:

often called „staging area“. GIT commits changes not directly from the working tree, but from the index. Files changed have to be added to the index before they can be committed.

GIT File Workflow

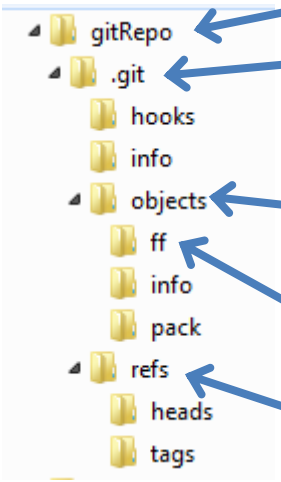


File Content Is Stored In *Blobs*

- file content is stored in *blobs*
- blobs are the *leaves* of the (file system like) tree
- blobs store *no meta-data*. That‘s the responsibility of a tree
- blobs are *immutable*
- the *name* of a blob is the *SHA-1 hash* of (its size concatenated with) its content
- blobs with the same content have the same name – *everywhere* in the world
- files with *identical* content stored in different trees refer to *one* blob only
- GIT uses *content based* adressing

```
$ mkdir gitRepo; cd gitRepo
$ git init
$ echo "Creasy" > names
$ git hash-object –w names
ffef955277e4ad7972740fe9c1fe1fff60e60a27
$ git cat-file –t ffef95
blob
$ git cat-file blob ffef95
Creasy
```

The Structure Of A GIT Repository



the working tree

its repository

content based store (name: SHA-1 hashes of the content, e.g. blobs)

file "names" is here

names for commits (wait some minutes)

```
$ find .git -type f -and -regex '.*/objects/../.*'
.git/objects/ff/ef955277e4ad7972740fe9c1fe1fff60e60a27
```

Directories Are Stored In *Trees*

- blobs have only content (+ their SHA-1 hash name)
- structure and naming is represented in a *tree*
- a tree assembles *(sub-) trees and blobs*
- a tree object is *immutable* and its name is the SHA-1 hash of its content (like blobs)
- trees are created from the files actually in the index ("staged files")

```
$ mkdir descr; cd descr
$ echo "repository for cavy names" > README; cd ..
$ git hash-object -w descr/README
af0ceb86afde8b261a20c32a2d9f82a7e2352bd0
$ find .git -type f -and -regex '.*/objects/../.*'
.git/objects/af/0ceb86afde8b261a20c32a2d9f82a7e2352bd0
.git/objects/ff/ef955277e4ad7972740fe9c1fe1fff60e60a27
$ git ls-files –stage # outputs nothing
$ git add .           # adds all
$ git ls-files –stage # warning: LF CRLF problems --- ignore
100644 af0ceb86afde8b261a20c32a2d9f82a7e2352bd0 0       descr/README
100644 ffef955277e4ad7972740fe9c1fe1fff60e60a27 0       names
```

Directories Are Stored In *Trees* (continued)

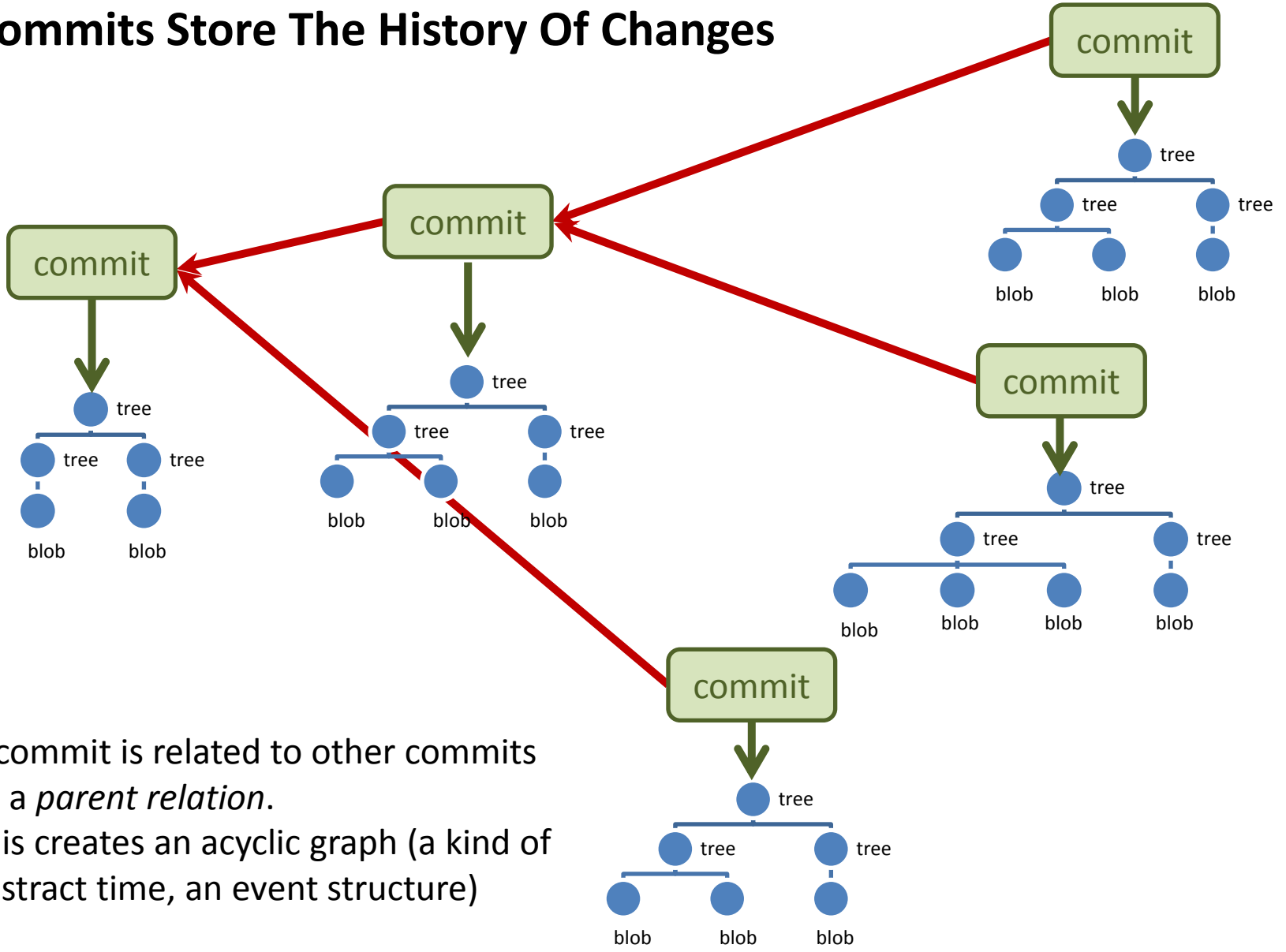
```
$ git write-tree
3cd53c9c80254073be17b2bcf06edb4ac4a85803
$ find .git -type f -and -regex '.*/objects/../.*'
.git/objects/3c/d53c9c80254073be17b2bcf06edb4ac4a85803 # tree
.git/objects/50/ae7f1e28fd1ecf40bfc55c2e92410a49063fed # tree
.git/objects/af/0ceb86afde8b261a20c32a2d9f82a7e2352bd0 # blob
.git/objects/ff/ef955277e4ad7972740fe9c1fe1fff60e60a27 # blob

$ git ls-tree 3cd5
040000 tree 50ae7f1e28fd1ecf40bfc55c2e92410a49063fed    descr
100644 blob ffef955277e4ad7972740fe9c1fe1fff60e60a27    names

$ git ls-tree 50ae
100644 blob 9190600164fe4b0818d230f825efb7bf0bcac269    README
```

Trees are *shared* based on their SHA-1 hashes

The same tree/blob structure (+ 1 commit object) would have been created by:

```
git commit –a –m "initial commit" # commit coming soon :-)
```

Commits Store The History Of Changes



A commit is related to other commits by a *parent relation*.
This creates an acyclic graph (a kind of abstract time, an event structure)

time

Creating A Commit

- a commit says,
  - *who* committed
  - *when*
  - *what* tree and
  - *why* and
  - from what *commit(-s)* the change was derived (the immediate past, the *parent(s)*)
- *each* commit represents the *whole history* of previous changes (follow its parents

```
$ git commit-tree -m "the first name" 3cd5 # afterwards 5 objects are stored
3cdb74ef7a0a2b847d21d46c251bbb5c71b19ef9
$ git cat-file -t 3cdb
commit
$ git cat-file commit 3cdb
tree 3cd53c9c80254073be17b2bcf06edb4ac4a85803
author Reinhard Budde <reinhard.budde@iais.fraunhofer.de> 1360330438 +0100
committer Reinhard Budde <reinhard.budde@iais.fraunhofer.de> 1360330438 +0100

the first name
```

A Second Commit

```
$ echo "Pid" >> names
$ echo "Mucky" >> names
$ git add names
$ git ls-files --stage
100644 af0ceb86afde8b261a20c32a2d9f82a7e2352bd0 0       descr/README
100644 fd867e53ec91b9b48204214e4d67740a505aea2f 0       names       # neuer SHA-1 hash!!
$ git write-tree
1be52efc1e1daed1c706b60588400487db083fe4
$ git commit-tree -m "2 further names"
  -p 3cdb74ef7a0a2b847d21d46c251bbb5c71b19ef9 1be52
4335220dd169d637b3299363033307f3ef31fc1d
$  git cat-file commit 433522
tree 1be52efc1e1daed1c706b60588400487db083fe4
parent 3cdb74ef7a0a2b847d21d46c251bbb5c71b19ef9
author Reinhard Budde <reinhard.budde@iais.fraunhofer.de> 1360334547 +0100
committer Reinhard Budde <reinhard.budde@iais.fraunhofer.de> 1360334547 +0100

2 further names
```

A Bigger Picture

- you have 1-n *branches* (~~ lines of development) in your repository
- a *branch* has a *name*. That name refers to a SHA-1 hash of a *commit* (see below)
- this commit is the most recent commit of this branch, the *head* of that branch
- this commit contains the whole history of this branch
- ***NOTHING ELSE***
- adding a new branch (low level) to a repository:

```
echo "4335220dd169d637b3299363033307f3ef31fc1d" > .git/refs/heads/branch1
```

- by *convention* you should have the branch master:

```
echo "4335220dd169d637b3299363033307f3ef31fc1d" > .git/refs/heads/master
```

- the head of the branch actually checked out in your working tree is referenced symbolically by HEAD (a top-level file in .git):

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

- this content has been created by git init and is changed by checking out another branch:

```
git checkout branch1
$ git checkout branch1
Switched to branch 'branch1'
$ cat .git/HEAD
ref: refs/heads/branch1
```

Fom Plumbing To Porcelain

- working with a repository using low level commands gives insight into GIT, but is boring
- low-level commands: plumbing
- user-level commands: porcelain
- working with porcelain commands:

```
$ git add …
$ git status
$ git branch –v
$ git log
$ git log --oneline
$ git checkout …
$ git commit –m "…"
```

- … more later …

Don't forget: if you understand commits, you master version control

Names of commits …

Name	Meaning
branchname	*refers* to the *most recent* commit of that branch (.git/refs/heads/branchname)
tagname	*refers* to the a commit, never changed (.git/refs/tags/tagname)
HEAD	the currently checked out commit
ffef955277e4ad… ffef9552 ffef	a commit, full name is the 40 char SHA-1 hash
name^	the parent of the commit
name^2	the second parent (e.g. a merge commit)
name~10	the 10th parent (name~1 == name^, name~3 == name^^^)
name:path	file of the tree of the commit
name^{tree}	the tree of the commit

Don't forget: if you understand commits, you master version control

Names of commit ranges …

Name	Meaning
name1..name2	all commits from name2 back to name1, but excluding name1
name1…name2	all commits referenced by name2 OR name1, but not by both
master..	short for: master..HEAD; show changes made to the current branch
..master	short for: HEAD..master; useful after fetch: what occured after laster rebase/merge
--since="2 weeks ago"	all commits since a date
--until="2 weeks ago"	all commits up to a date
--grep="pattern"	all commits whose message match the pattern
--committer="pattern"	all commits whose committer name match the pattern
--author="pattern"	all commits whose author name match the pattern (for local repos as above, if patches are mailed, author + committer differ)
--no-merges	all commits that have only one parent

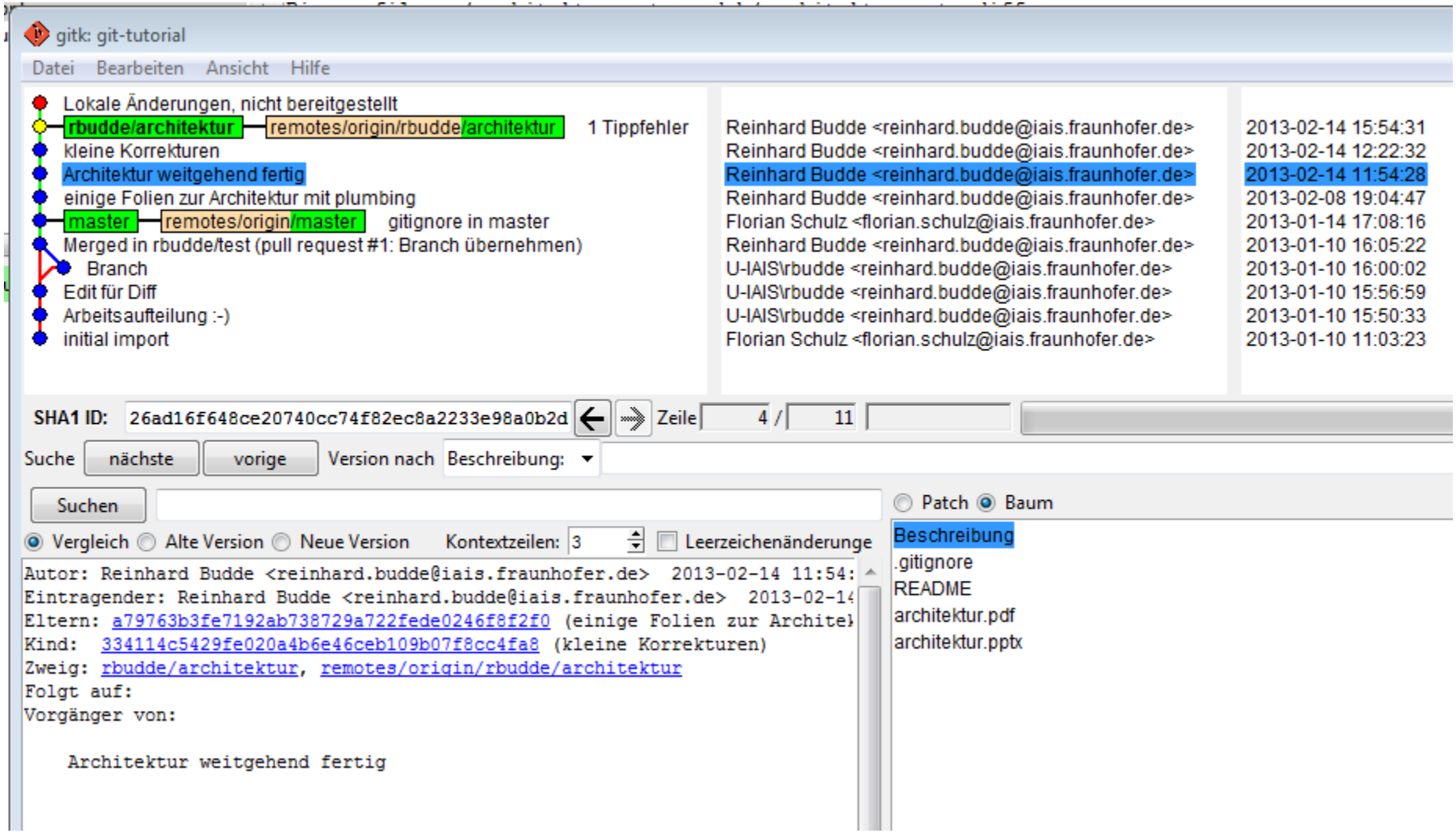
```
git log deploy-2012-12-14..HEAD --since="10 days ago" --grep="jira-3212" --no-merges
```

Useful Commands For Achieving A Good Overview

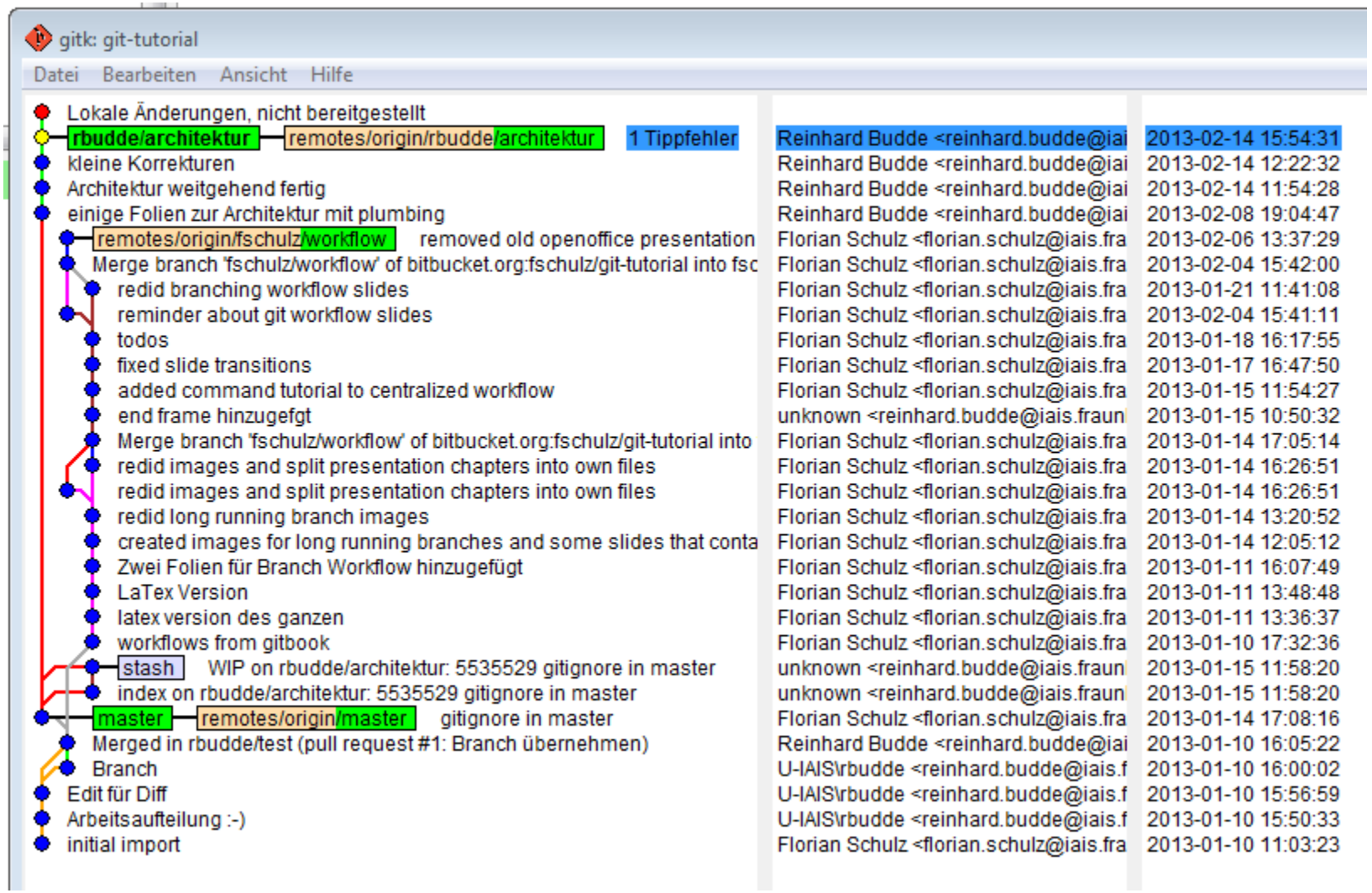
Command	Meaning
git remote -v	show the remote tracked repositories
git branch -vv [-a]	show all local branches and the remote branches, they are connected to
git config --global push.default simple	enforce that you can only *push* to *remote* branches of the *same* name
git branch name origin/name	get remote branch name. *Advice*: use the *same* name locally
git log HEAD~7.. --oneline	the last 7 (8???) commits in short notation
git log --graph --oneline	show as a graph
git status [-s]	working tree status [-s short format, but easy to read]

- all examples use the command line interpreter **`git bash`**
- git gui is nice, too. See next slides.
- egit in eclipse is fully compatible with command line git. See next slides.

A Branch In gitk (git gui)



All Branches In gitk (git gui)



- usually you have to filter out commits to control the amount of information.
- use "Ansicht bearbeiten" (F4).
- note some strange wordings, e.g. checkout is translated to "Umstellen …"
  or commit "abzeichnen"

GIT And Eclipse

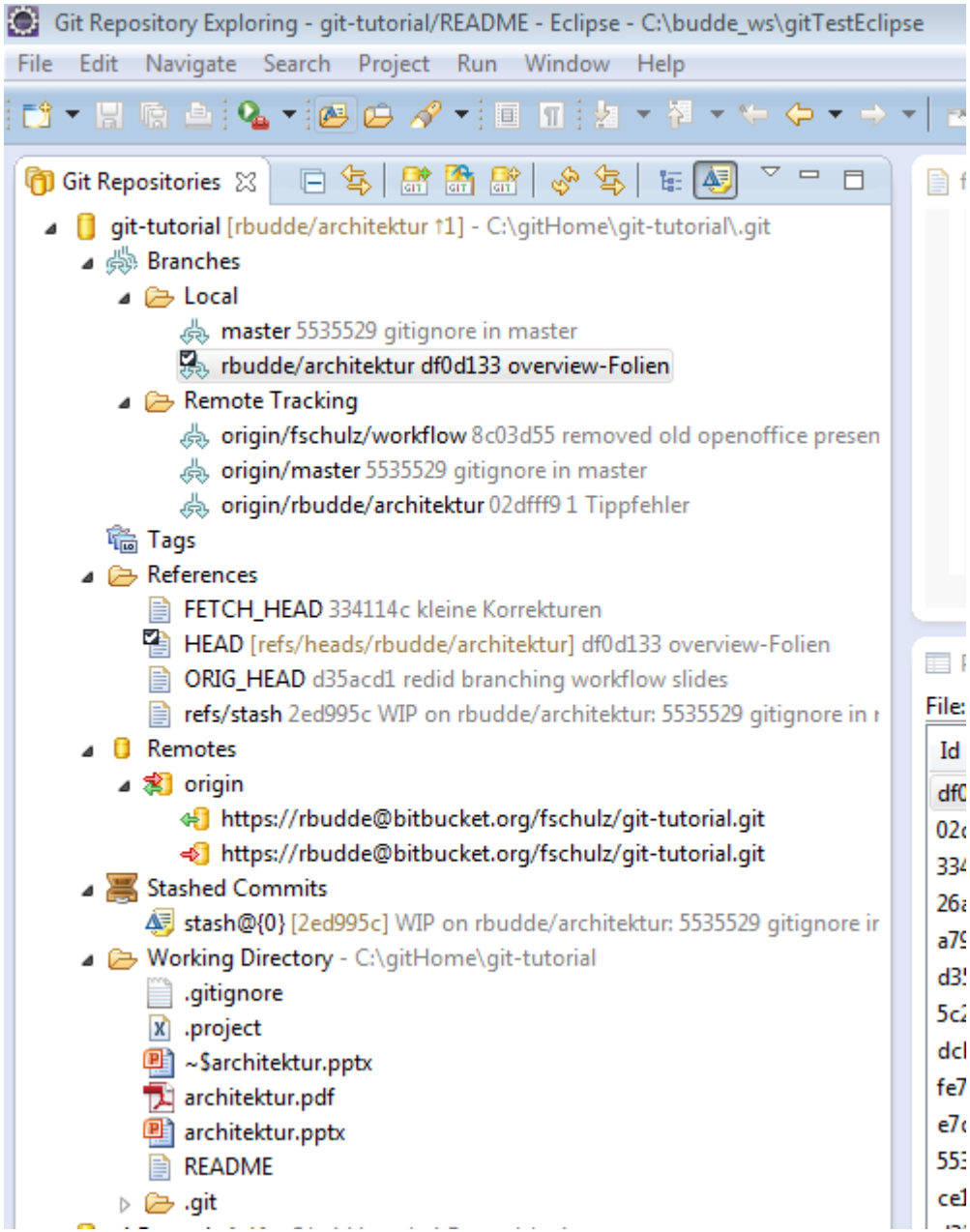
the GIT exploring perspective shows

- local and remote branches
- tags, references (HEADs) and remote repositories
- stashed commits

You may

- create and delete branches
- checkout branches
- push and pull branches
- merge and rebase

You may even browse the local repository `.git`



GIT And Eclipse

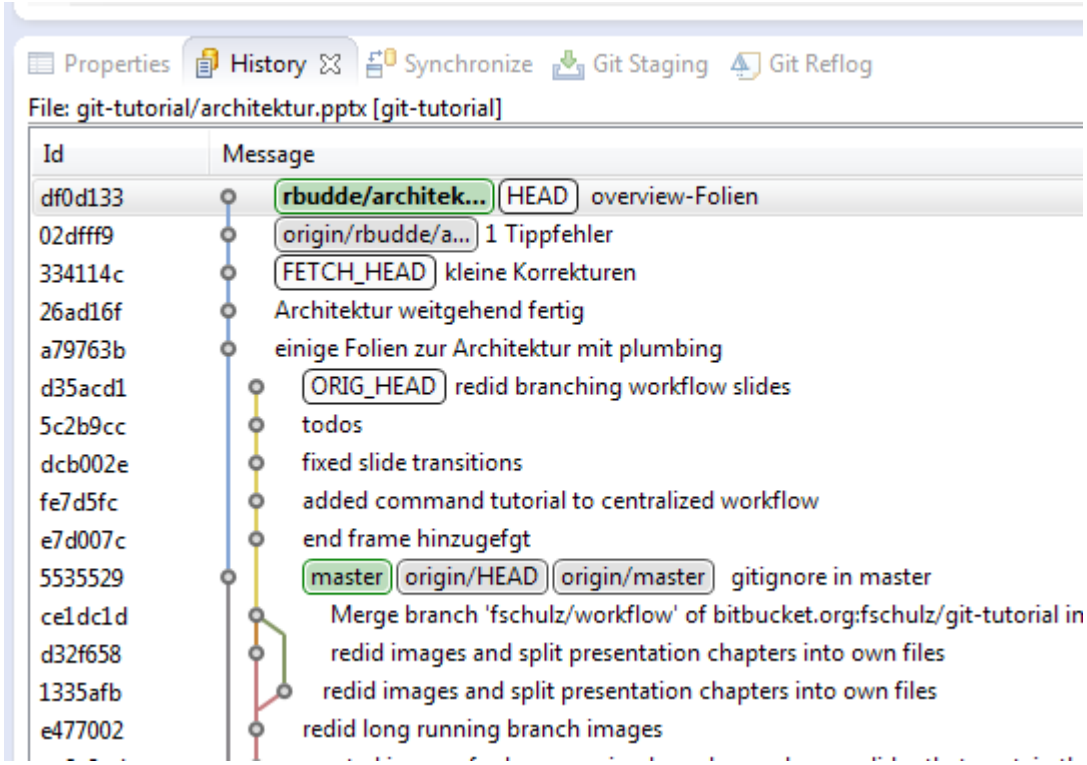
the GIT staging view

- shows the staged and unstaged changes from the working tree and
- allows to edit a commit message and commit changed

the history view shows

- the commit history of a selected resource (as a graph), but also
- the commit history of a project or a repository

Select a resource, drag it to the view and use the menu buttons to change the scope of the view



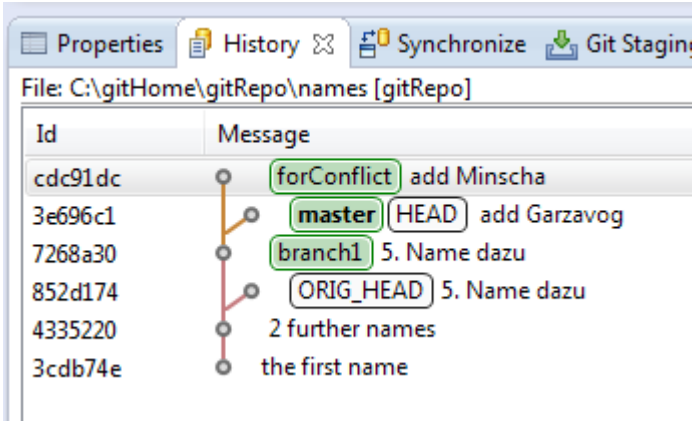
Solving A Merge Conflict With Eclipse -1-

1. create an Eclipse project from gitRepo -- now we use the cavy-repository again
   import…
   select project from GIT
   select local repo
   add the repo
   import as general project

2. create a conflict :-)

```
$ git branch forConflict
$ cat names
Creasy
Pid
Mucky
Greta
$ echo "Garzavog" >> names
$ git commit -a -m "add Garzavog"
$ git checkout forConflict
$ echo "Minscha" >> names
$ git commit -a -m "add Minscha"
```

Solving A Merge Conflict With Eclipse -2-

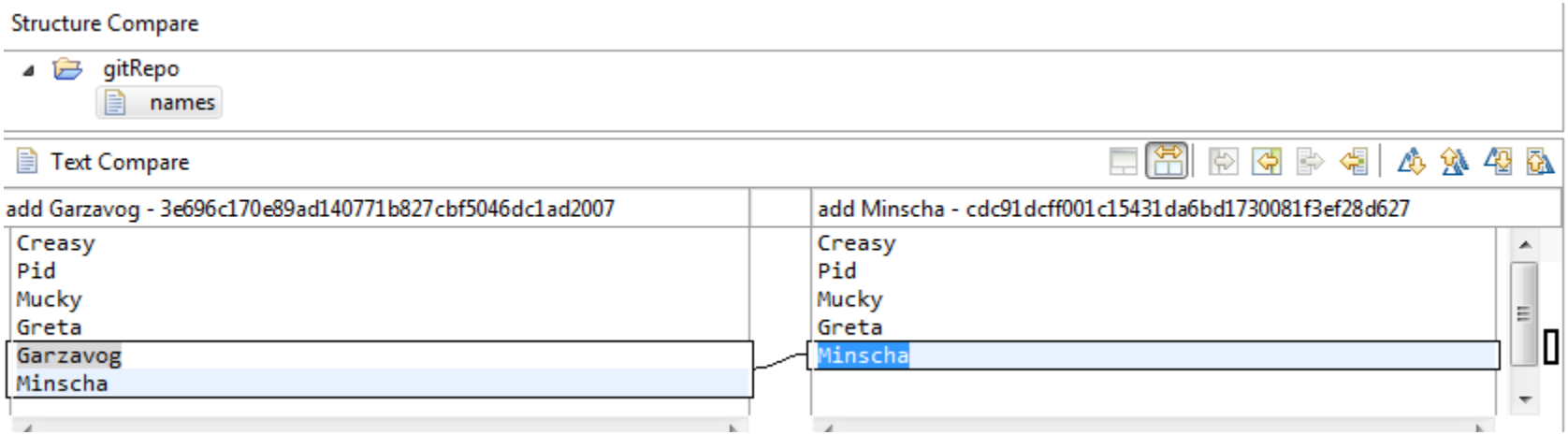


3. merge into master the branch forConflict

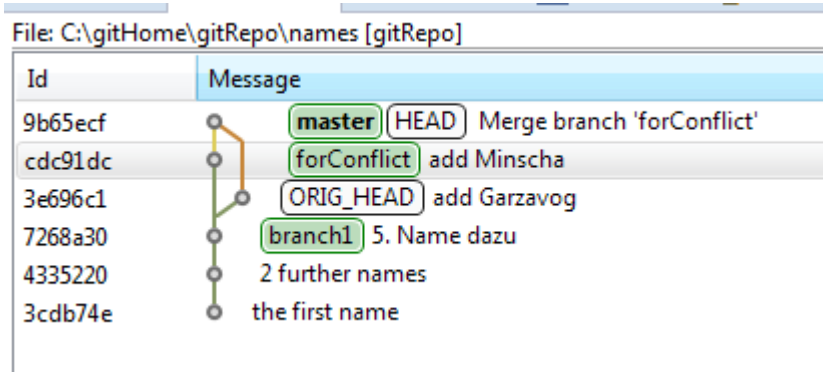
```
$ git checkout master
$ git merge forConflict
Auto-merging names
CONFLICT (content): Merge conflict in names
Automatic merge failed; fix conflicts and then commit the result.
```

Solving A Merge Conflict With Eclipse -3-

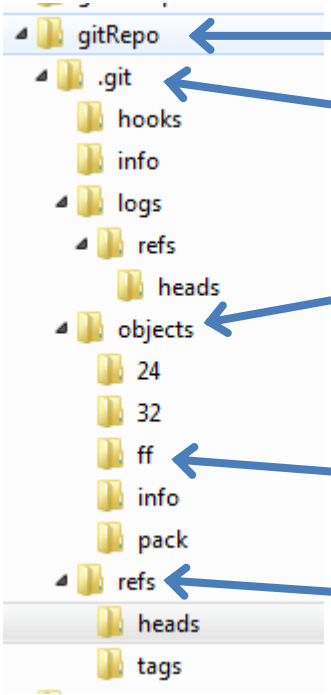
4. switch to eclipse
   - in the GIT *staging view* the conflict is marked red
   - from the context menu start the merge tool
   - !!! select merge mode **`Use HEAD of conflicting files`** , click OK !!!
   - solve the conflict, save the file



   - in the *staging view*, move the file to staged and commit it
   - this will finish the merge, as the history view shows …



The Structure Of A GIT Repository



the working tree

its repository

content based store (name: SHA-1 hashes of the content, e.g. blobs)

file content of "names" is found here

names for commits (wait some minutes)

```
$ find . -type f -and ! -name '*.sample'
./COMMIT_EDITMSG
./config
./description
./HEAD
./index
./info/exclude
./logs/HEAD
./logs/refs/heads/master
./objects/24/6bf44b5432c7a982e9fd0f44f1b9a480c1495f
./objects/32/f81177700a7d16640b019d5ffbe52563643090
./objects/ff/ef955277e4ad7972740fe9c1fe1fff60e60a27
./refs/heads/master

rbudde@ILYA /c/gitHome/gitRepo/.git (GIT_DIR!)
```

configuration

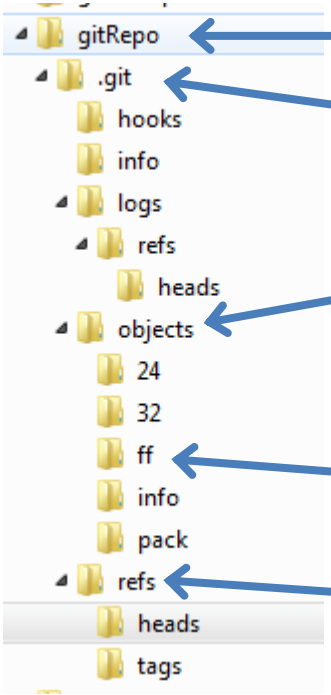
the commit checked out

working tree image (added files!)

the objects stored

the last commit of the branch "master"

The Structure Of A GIT Repository



the working tree

its repository

content based store (name: SHA-1 hashes of the content, e.g. blobs)

file "names" is here

names for commits (wait some minutes)

```
--- cd to .git and then:
$ find . -type f -and -regex './objects/../.*'
./objects/24/6bf44b5432c7a982e9fd0f44f1b9a480c1495f # hash is 246bf.....
./objects/25/ae680dccc76c1e67ceb52886cdcabaf94186b1
./objects/32/f81177700a7d16640b019d5ffbe52563643090
./objects/48/33b56499eae2332773f2fce8784861dcb91e69
./objects/8f/38871a7d5599bdd54d8ec6ccd733afa94b0a16
./objects/91/90600164fe4b0818d230f825efb7bf0bcac269
./objects/ff/ef955277e4ad7972740fe9c1fe1fff60e60a27
```

File Content Is Stored In *Blobs*

- file content is stored in *blobs*
- blobs are the *leaves* of the (file system like) tree
- blobs store *no meta-data*. That‘s the responsibility of a tree
- blobs are *immutable*
- the *name* of a blob is the *SHA-1 hash* of (its size concatenated with) its content
- blobs with the same content have the same name – *everywhere* in the world
- files with *identical* content stored in different trees refer to *one* blob only
- GIT uses *content based* adressing

```
$ mkdir gitRepo; cd gitRepo
$ git init
$ echo "Creasy" > names
$ git hash-object names
ffef955277e4ad7972740fe9c1fe1fff60e60a27
$ git add names
$ git commit –m "first of 6 names"
$ git cat-file –t ffef95
blob
$ git cat-file blob ffef95
Creasy
```