

GIT

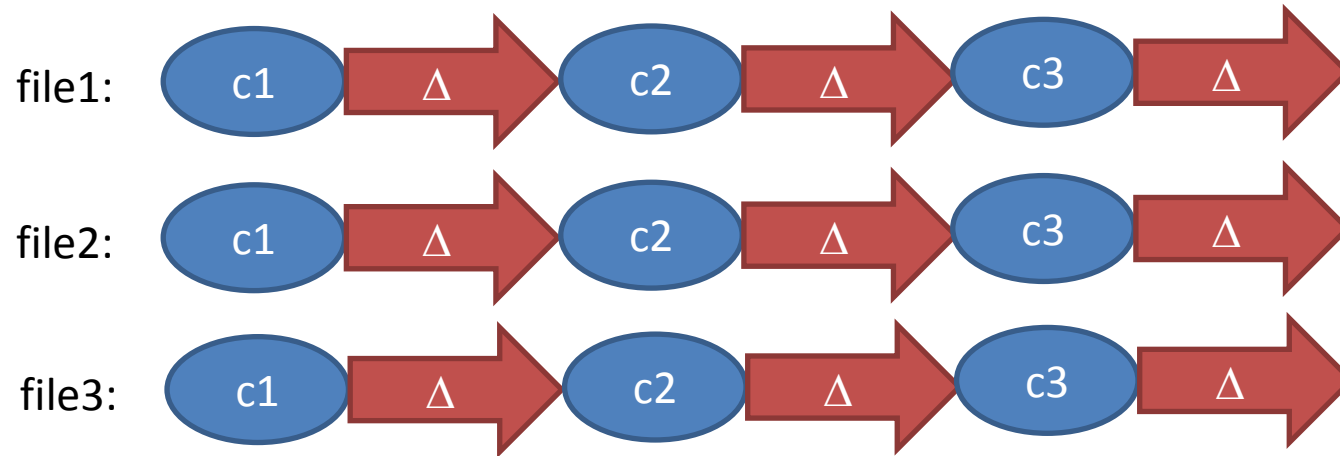
- Introduction
- Architecture
- Basic Workflows
- Useful commands
- Discussion

References

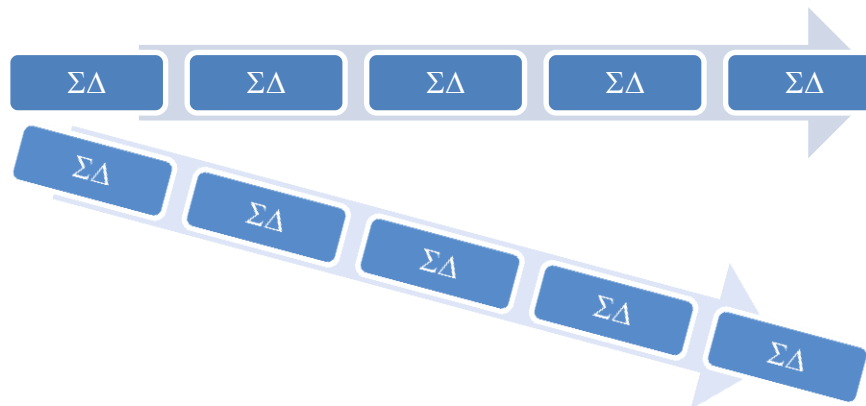
- **John Wiegley:** *Git from the bottom up* (Dec 2009)
<http://newartisans.com/2008/04/git-from-the-bottom-up/>
a lot is taken from that source.
- **Scott Chacon:** *Pro Git*
<http://git-scm.com/book/de>
HTML + PDF book available online.
Complete reference to (almost) all aspects of GIT
- <http://www.kernel.org/pub/software/scm/git/docs/everyday.html> explains the usage of GIT by example for different development styles(
- A lot of Git **cheat sheets** are available online
- **git help** <command> is a valuable resource: explanation and examples
- <http://nvie.com/posts/a-successful-git-branching-model/> describes a realistic and complete development model for enterprise software based on GIT, compact and good
- <http://stackoverflow.com> has a discussion about almost all aspects of using GIT

Well Known: CVS/SVN

- idea: sequence of deltas of a file

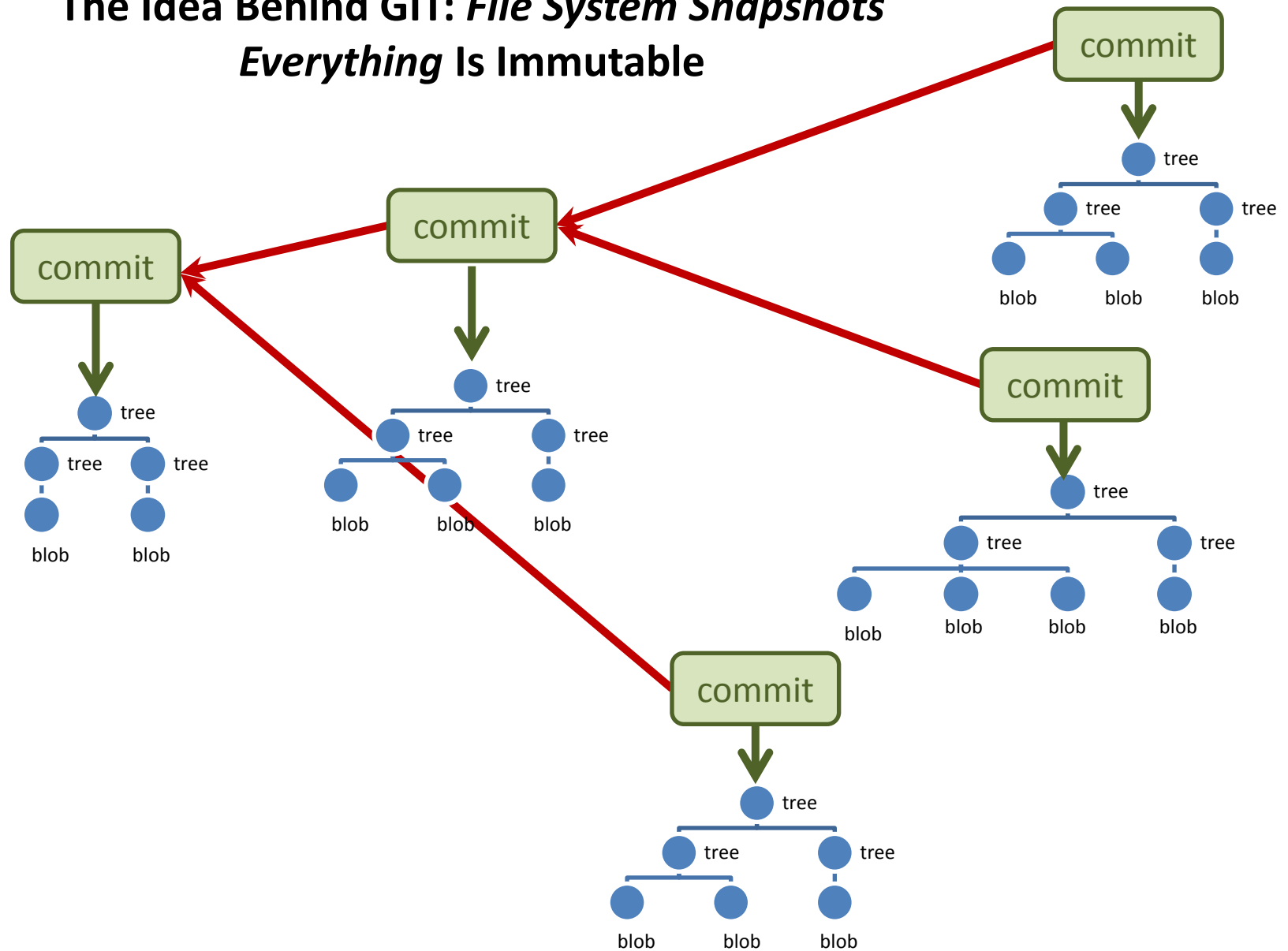


- branches: a complex other construct



The Idea Behind GIT: *File System Snapshots*

Everything Is Immutable



time

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. This 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.

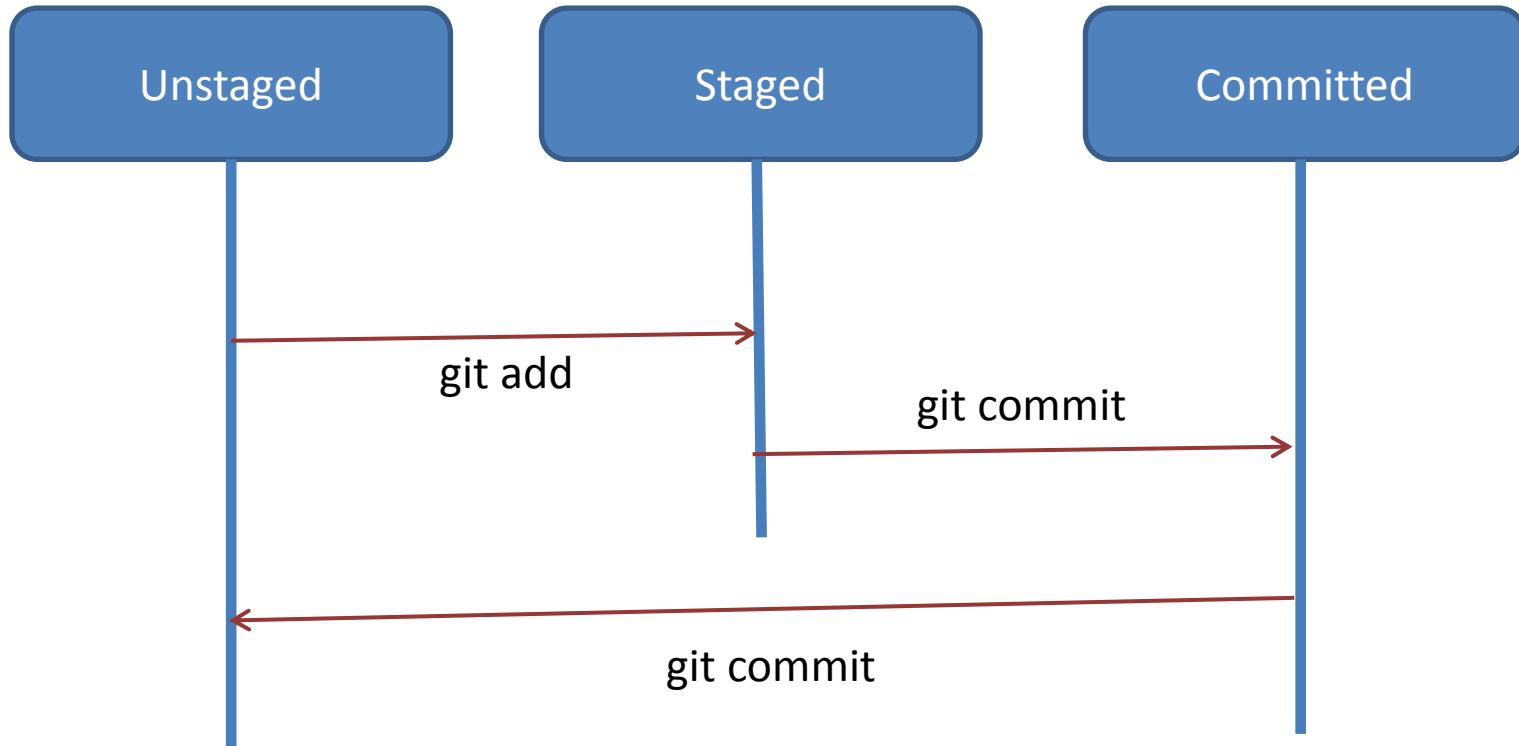
master:

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



```
$ mkdir repo; cd repo
$ git init
$ gvim explain.txt
$ git add ex*
$ git commit -m "added explain"
```

```
$ gvim explain.txt # next change
$ git add ex*
$ git commit -m "typos removed"
$ git log --oneline
$ git diff HEAD HEAD~ # what changed?
```

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* addressing

```
$ mkdir gitRepo; cd gitRepo
```

```
$ git init
```

```
$ echo "Creasy" > names
```

```
$ git hash-object -w names
```

```
ffef955277e4ad7972740fe9c1fe1fff60e60a27
```

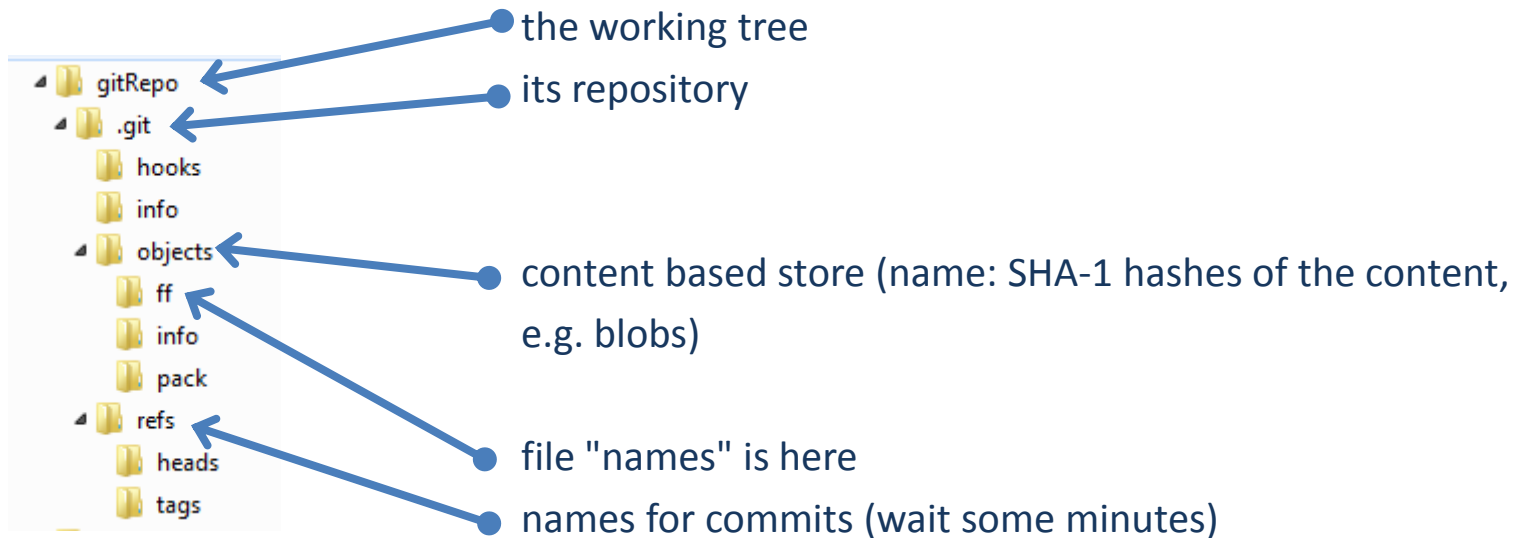
```
$ git cat-file -t ffef95
```

```
blob
```

```
$ git cat-file blob ffef95
```

```
Creasy
```


The Structure Of A GIT Repository



```
$ 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

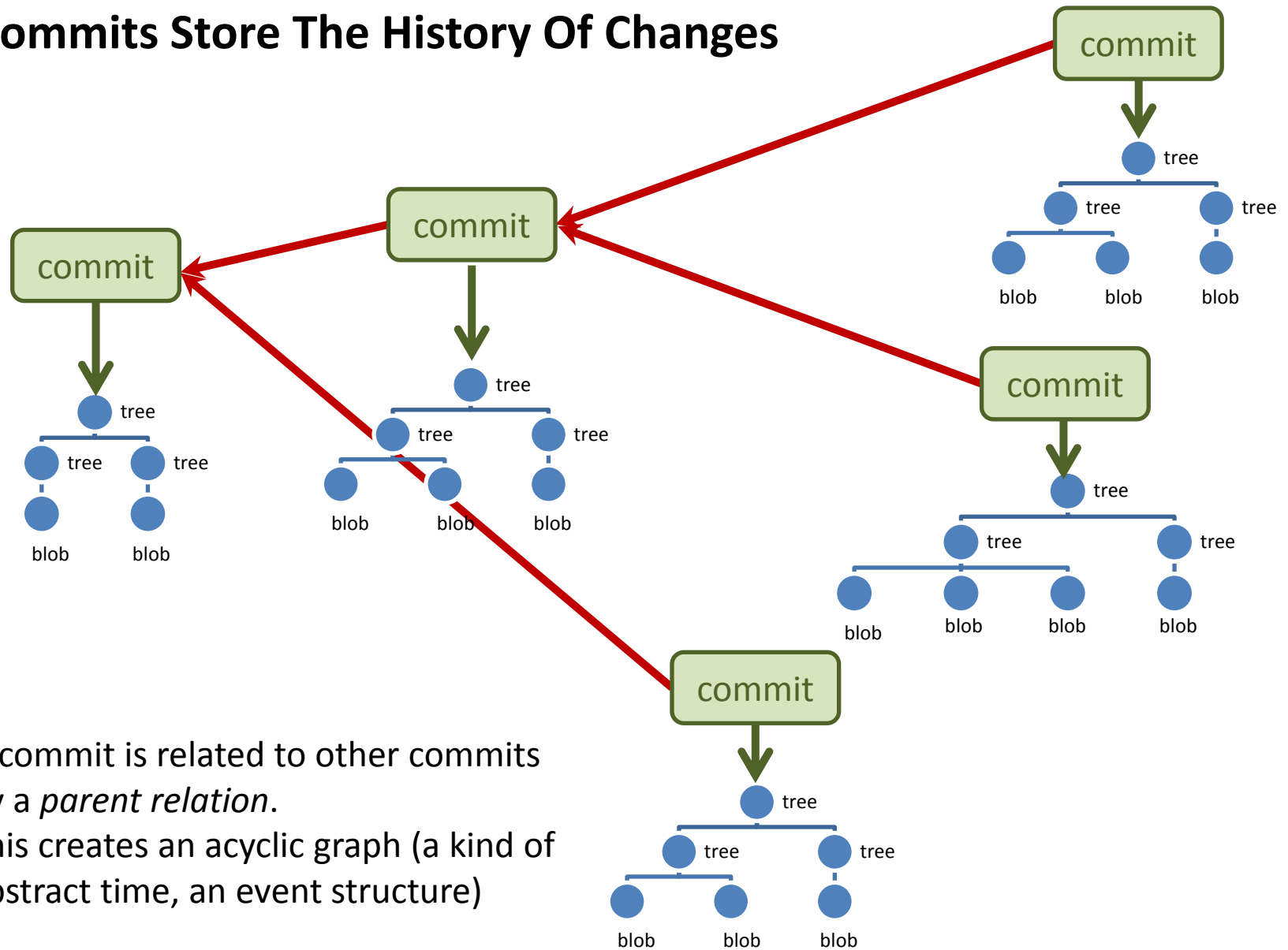
$ git ls-tree 50ae
100644 blob 9190600164fe4b0818d230f825efb7bf0bcac269      README
```

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)



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 # 5 objects are stored
3cdb74ef7a0a2b847d21d46c251bbb5c71b19ef9
$ git cat-file -t 3cdb
commit
$ git cat-file commit 3cdb
tree 3cd53c9c80254073be17b2bcf06edb4ac4a85803
author Reinhard Budde <reinhard.budde@iaais.fraunhofer.de> 1360330438 +0100
committer Reinhard Budde <reinhard.budde@iaais.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
```

- HEAD is created by git init and changed by checking out another branch:

```
$ 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 "..."
```

- ... next slides ...

GIT

- distributed work, working offline is simple
 - all metadata in one directory (usually .git), not one meta-directory in each directory
 - simple, consistent, powerful model
 - fast, robust
 - supports much more workflows than traditional vcs (CVS, SVN)
 - experiments in separate branches, undoing changes, merging if changes are successful: all this is simple and cheap
-
- takes definitively some time to become familiar with
 - for beginners sometimes irritating
 - affine to the command line :-), overwhelming number of parameter for commands (but very good help)

GIT For A Standalone Developer (1)

A standalone individual developer does not exchange patches with other people, and works alone in a single repository, using the following commands.

[git-init\(1\)](#) to create a new repository.

[git-show-branch\(1\)](#) to see where you are.

[git-log\(1\)](#) to see what happened.

[git-checkout\(1\)](#) and [git-branch\(1\)](#) to switch branches.

[git-add\(1\)](#) to manage the index file.

[git-diff\(1\)](#) and [git-status\(1\)](#) to see what you are in the middle of doing.

[git-commit\(1\)](#) to advance the current branch.

[git-reset\(1\)](#) and [git-checkout\(1\)](#) (with pathname parameters) to undo changes.

[git-merge\(1\)](#) to merge between local branches.

[git-rebase\(1\)](#) to maintain topic branches.

[git-tag\(1\)](#) to mark known point.

GIT For A Standalone Developer (2)

Start individual development

```
$ tar xzf frotz.tar.gz; cd frotz; git init
$ git add .      # add everything under the current directory.
$ git commit -m "import of frotz source tree."
$ git tag v2.43 # make a lightweight, unannotated tag.
```

Create a topic branch and develop.

```
$ git branch alsa-audio # create a new topic branch.
$ git checkout alsa-audio # checkout (abbr: git checkout -b alsa-audio)
$ edit/compile/test
$ git diff HEAD          # to see what changes you are committing.
$ git commit -a -m "first changes"
$ edit/compile/test
$ git reset --soft HEAD^ # take last commit back, keep what is in the working tree.
$ edit/compile/test
$ git diff ORIG_HEAD     # view the changes since the premature commit we took back.
$ git commit -a -c ORIG_HEAD # redo the commit undone, use message you originally wrote.
$ git checkout master    # switch to the master branch.
$ git merge alsa-audio   # merge a topic branch into your master branch.
$ git log --since='3 days ago' # review commit logs
$ git log v2.43.. curses/ # view changes that touch the curses directory, since v2.43 tag.
```

A Branch In gitk (git gui)

The screenshot shows the gitk: git-tutorial window. The top menu bar includes 'Datei', 'Bearbeiten', 'Ansicht', and 'Hilfe'. The left pane displays a branch diagram with the following nodes and actions:

- Lokale Änderungen, nicht bereitgestellt
- rbudde/architektur** (highlighted in green) — **remotes/origin/rbudde/architektur** (highlighted in orange) 1 Tippfehler
- kleine Korrekturen
- Architektur weitgehend fertig** (highlighted in blue)
- einige Folien zur Architektur mit plumbing
- master** (highlighted in green) — **remotes/origin/master** (highlighted in orange) gitignore in master
- Merged in rbudde/test (pull request #1: Branch übernehmen)
- Branch
- Edit für Diff
- Arbeitsaufteilung :-)
- initial import

The right pane shows a list of commits with their authors and timestamps:

Author	Timestamp
Reinhard Budde <reinhard.budde@iais.fraunhofer.de>	2013-02-14 15:54:31
Reinhard Budde <reinhard.budde@iais.fraunhofer.de>	2013-02-14 12:22:32
Reinhard Budde <reinhard.budde@iais.fraunhofer.de> (highlighted in blue)	2013-02-14 11:54:28 (highlighted in blue)
Reinhard Budde <reinhard.budde@iais.fraunhofer.de>	2013-02-08 19:04:47
Florian Schulz <florian.schulz@iais.fraunhofer.de>	2013-01-14 17:08:16
Reinhard Budde <reinhard.budde@iais.fraunhofer.de>	2013-01-10 16:05:22
U-IAIS\rbudde <reinhard.budde@iais.fraunhofer.de>	2013-01-10 16:00:02
U-IAIS\rbudde <reinhard.budde@iais.fraunhofer.de>	2013-01-10 15:56:59
U-IAIS\rbudde <reinhard.budde@iais.fraunhofer.de>	2013-01-10 15:50:33
Florian Schulz <florian.schulz@iais.fraunhofer.de>	2013-01-10 11:03:23

The bottom section shows the SHA1 ID: 26ad16f648ce20740cc74f82ec8a2233e98a0b2d. Below this are buttons for 'Suche', 'nächste', 'vorige', and 'Version nach'. A search bar with the text 'Suchen' is also present. The bottom left pane shows the commit details for the selected commit:

Autor: Reinhard Budde <reinhard.budde@iais.fraunhofer.de> 2013-02-14 11:54:
Eintragender: Reinhard Budde <reinhard.budde@iais.fraunhofer.de> 2013-02-14
Eltern: [a79763b3fe7192ab738729a722fede0246f8f2f0](#) (einige Folien zur Archite)
Kind: [334114c5429fe020a4b6e46ceb109b07f8cc4fa8](#) (kleine Korrekturen)
Zweig: [rbudde/architektur](#), [remotes/origin/rbudde/architektur](#)
Folgt auf:
Vorgänger von:

The bottom right pane shows the file list for the selected commit:

- Beschreibung** (highlighted in blue)
- .gitignore
- README
- architektur.pdf
- architektur.pptx

The bottom status bar shows the text 'Architektur weitgehend fertig'.

All Branches In gitk (git gui)

The screenshot shows the gitk GUI for a repository named 'gitk: git-tutorial'. The interface includes a menu bar (Datei, Bearbeiten, Ansicht, Hilfe) and a commit history graph on the left. The graph shows a main branch 'master' with a 'stash' and a 'branch' 'rbdude/architektur'. A commit 'remotes/origin/rbdude/architektur' is highlighted. The right pane displays a list of commits with their authors, email addresses, and timestamps.

Commit Message	Author	Timestamp
Reinhard Budde <reinhard.budde@iai	2013-02-14 15:54:31	
Reinhard Budde <reinhard.budde@iai	2013-02-14 12:22:32	
Reinhard Budde <reinhard.budde@iai	2013-02-14 11:54:28	
Reinhard Budde <reinhard.budde@iai	2013-02-08 19:04:47	
Florian Schulz <florian.schulz@iais.fra	2013-02-06 13:37:29	
Florian Schulz <florian.schulz@iais.fra	2013-02-04 15:42:00	
Florian Schulz <florian.schulz@iais.fra	2013-01-21 11:41:08	
Florian Schulz <florian.schulz@iais.fra	2013-02-04 15:41:11	
Florian Schulz <florian.schulz@iais.fra	2013-01-18 16:17:55	
Florian Schulz <florian.schulz@iais.fra	2013-01-17 16:47:50	
Florian Schulz <florian.schulz@iais.fra	2013-01-15 11:54:27	
unknown <reinhard.budde@iais.fraun	2013-01-15 10:50:32	
Florian Schulz <florian.schulz@iais.fra	2013-01-14 17:05:14	
Florian Schulz <florian.schulz@iais.fra	2013-01-14 16:26:51	
Florian Schulz <florian.schulz@iais.fra	2013-01-14 16:26:51	
Florian Schulz <florian.schulz@iais.fra	2013-01-14 13:20:52	
Florian Schulz <florian.schulz@iais.fra	2013-01-14 12:05:12	
Florian Schulz <florian.schulz@iais.fra	2013-01-11 16:07:49	
Florian Schulz <florian.schulz@iais.fra	2013-01-11 13:48:48	
Florian Schulz <florian.schulz@iais.fra	2013-01-11 13:36:37	
Florian Schulz <florian.schulz@iais.fra	2013-01-10 17:32:36	
unknown <reinhard.budde@iais.fraun	2013-01-15 11:58:20	
unknown <reinhard.budde@iais.fraun	2013-01-15 11:58:20	
Florian Schulz <florian.schulz@iais.fra	2013-01-14 17:08:16	
Reinhard Budde <reinhard.budde@iai	2013-01-10 16:05:22	
U-IAIS\rbdude <reinhard.budde@iais.f	2013-01-10 16:00:02	
U-IAIS\rbdude <reinhard.budde@iais.f	2013-01-10 15:56:59	
U-IAIS\rbdude <reinhard.budde@iais.f	2013-01-10 15:50:33	
Florian Schulz <florian.schulz@iais.fra	2013-01-10 11:03:23	

- 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

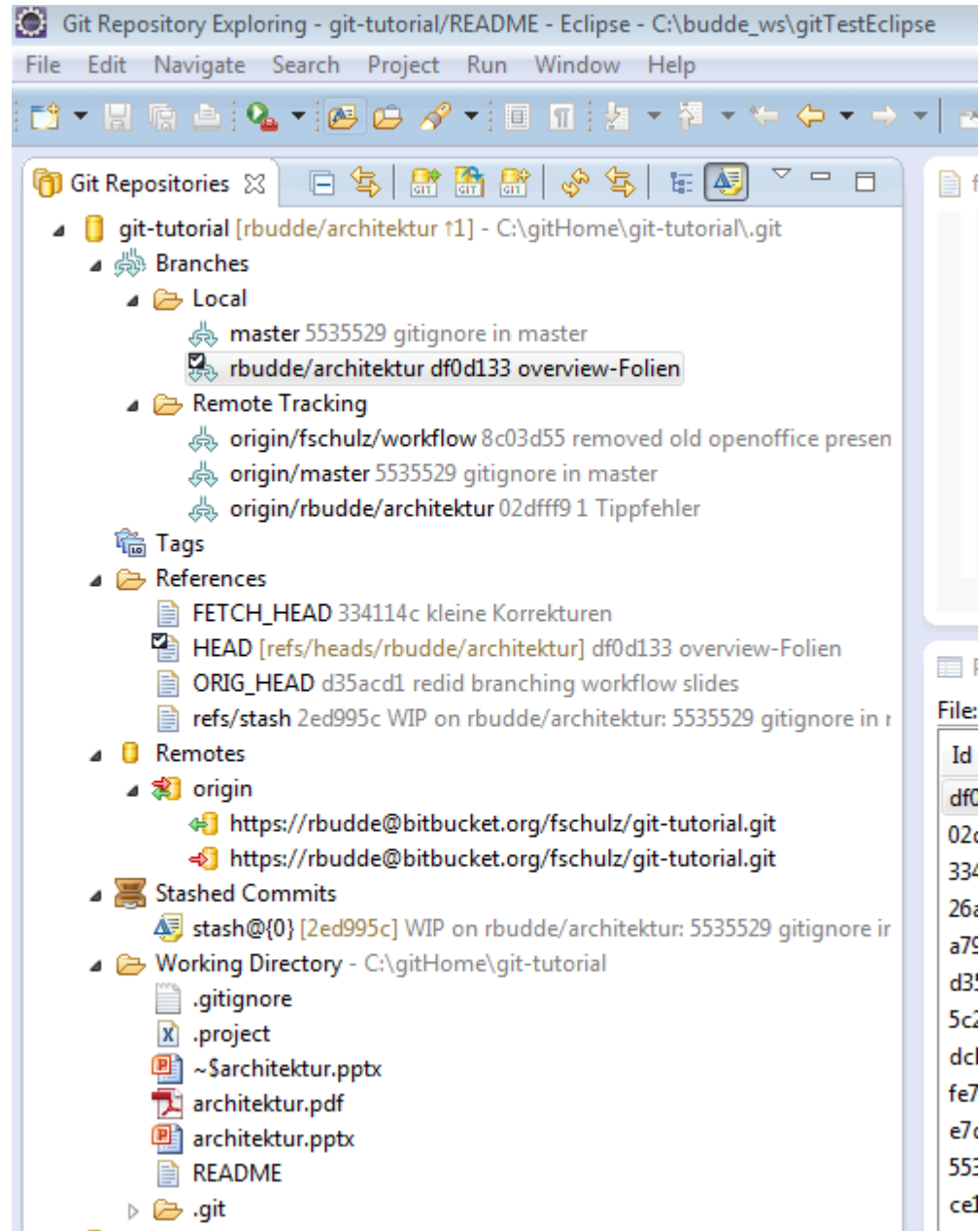
the GIT exploring perspective shows

- 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

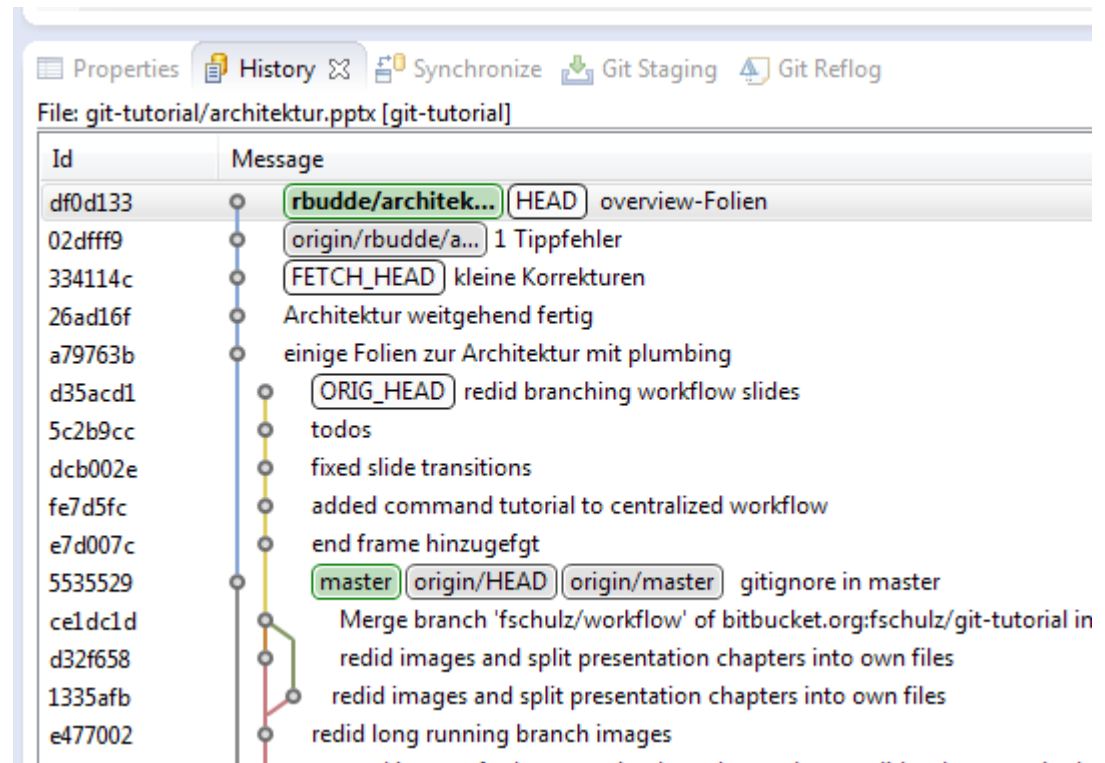
the GIT staging view

- 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

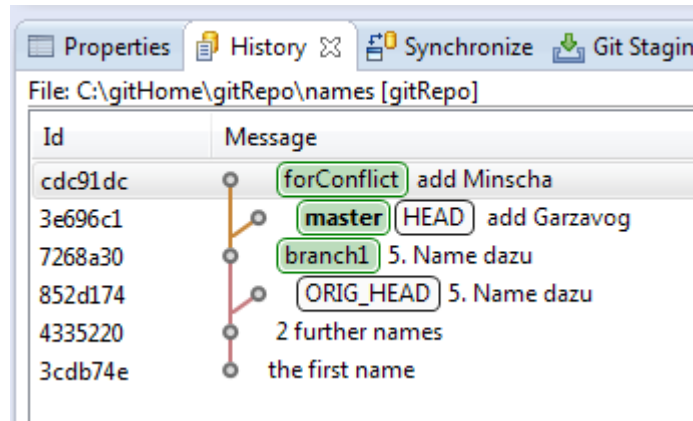
Select a resource,
drag it to the view and
use the menu buttons
to change the scope
of the view



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-



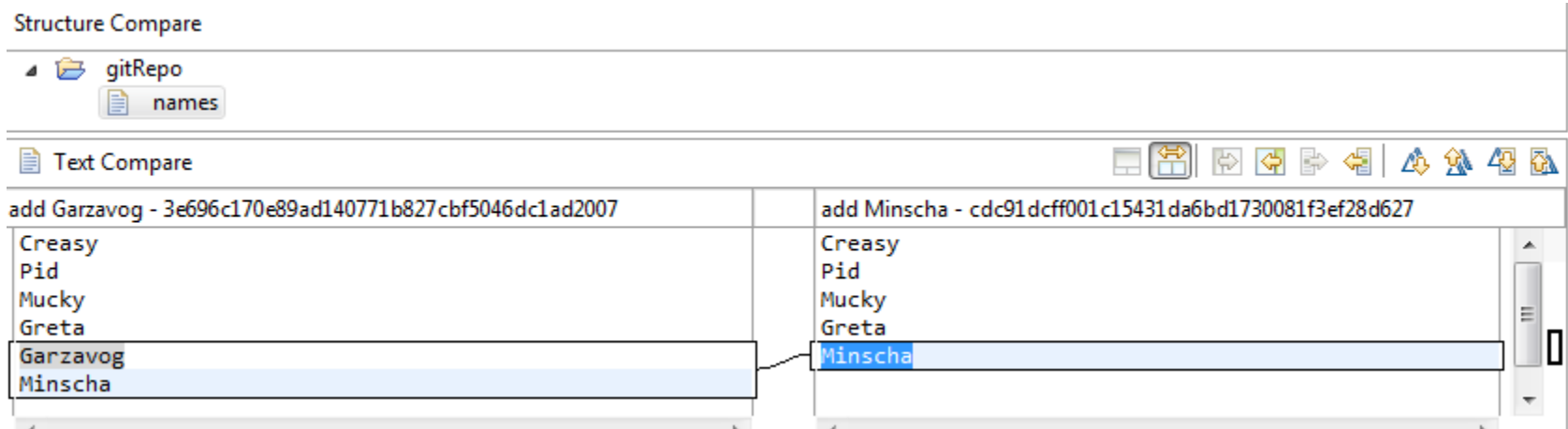
3. merge into master the branch forConflict

```
$ 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.
```

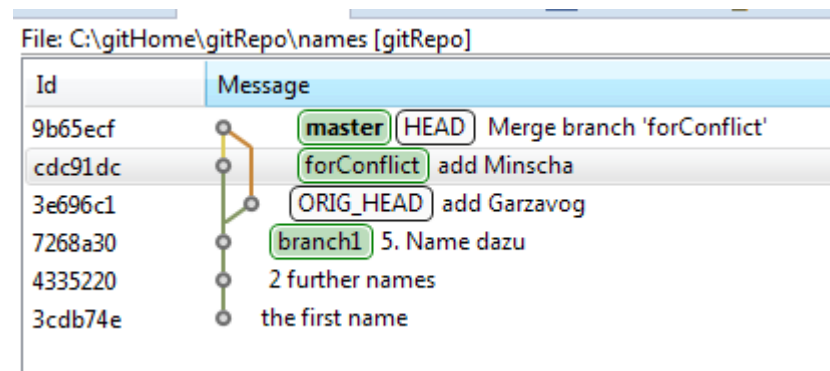
Solving A Merge Conflict With Eclipse -3-

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 ...



Configuration

- environment variables (avoid that)
- configuration files
 - user (~/.gitconfig)
 - system (<git-install-dir>/etc/gitconfig)
 - repository (<working-dir>/.git/config)
- in eclipse preferences open one of the configuration files as a *text file* and modify it ... or ...
- use `git config`

- the user conf file should define user data:

```
[user]
  email = reinhard.budde@iaais.fraunhofer.de
  name = Reinhard Budde
```

- the system conf file should define general data:

```
[core]
  symlinks = false
  autocrlf = true
  filemode = false
[http]
  sslverify = false
[alias]
  b = branch
  co = checkout
```

Reverting / Undoing Changes --- **Be careful**

the best choice:

- checkout a previous commit
- create a branch starting from there

these commands leave the working tree unchanged:

- `git reset HEAD`
delete changes staged in the index
- `git reset --soft HEAD~`
ignore the last commit; backup to the parent of HEAD

this commands resets the working tree (BE VERY CAREFUL):

- `git reset --hard HEAD~`
backup to the parent of HEAD; set the working tree to that commit

if the index is clean, the following commands are equivalent:

- `git reset --hard HEAD~2`
- `git reset --soft HEAD~2`
`git reset --hard`
- `git checkout -b twoCommitsBack HEAD~2`
- *why don't you use the last command?*

Reflog And Stash

the reflog stores all your changes in the form of commits:

- for 30 days (you can configure that)
- independant from branches, push, pulls, reset
- you may inspect it using `git reflog [<commit>]`
- or use the Eclipse view to browse
- names are `HEAD@{0}` ...

the stash stores the working tree and the index as a new commit:

- `git stash`
names are `stash@{0}` ...
- `git stash apply`
applies the stash and keeps it; `git stash pop` applies and removes

typical use:

- `# ... hack hack hack ...`
- `$ git stash`
- `$ edit emergency fix`
- `$ git commit -a -m "Fix in a hurry"`
- `$ git stash pop`
- `# ... continue hacking ...`

Rewriting The History

never rewrite the history when somebody has seen it (i.e. after a push ...)

there are three relevant cases:

- rewrite the last commit when committing the next one (amend)
`git commit --amend` # you may edit the last commit message
- replace a sequence of commits by a single one
`git rebase -i HEAD~5` # combine the last 5 commits in interactive mode
- rebase a branch based on another branch

```
$ git checkout topic
# edit, commit, ... , now integrate the work of others
$ git checkout master
$ git pull
$ git checkout topic
$ git rebase master # reintegrate master to minimize conflicts when done
```

never rewrite the history when somebody has seen it (i.e. after a push ...)

Understanding Commits => Mastering Version Control

Names of commits ...

Name	Meaning
branchname	<i>refers</i> to the <i>most recent</i> commit of that branch (<code>.git/refs/heads/branchname</code>)
tagname	<i>refers</i> to the a commit, never changed (<code>.git/refs/tags/tagname</code>)
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 (<code>name~1 == name^</code> , <code>name~3 == name^^</code>)
name:path	file of the tree of the commit
name^{tree}	the tree of the commit

Understanding Commits => Mastering 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 occurred 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

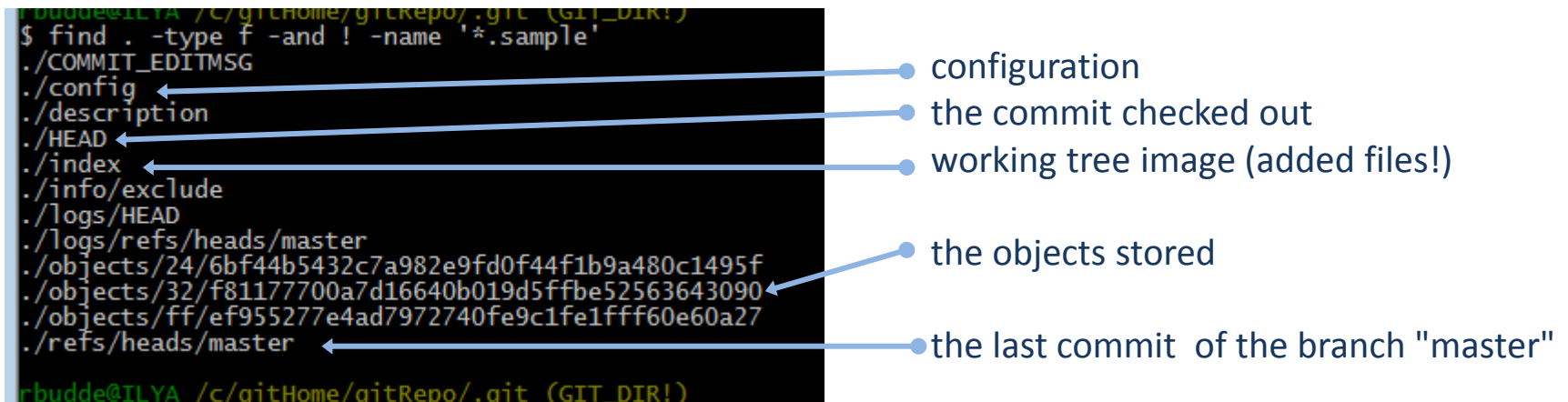
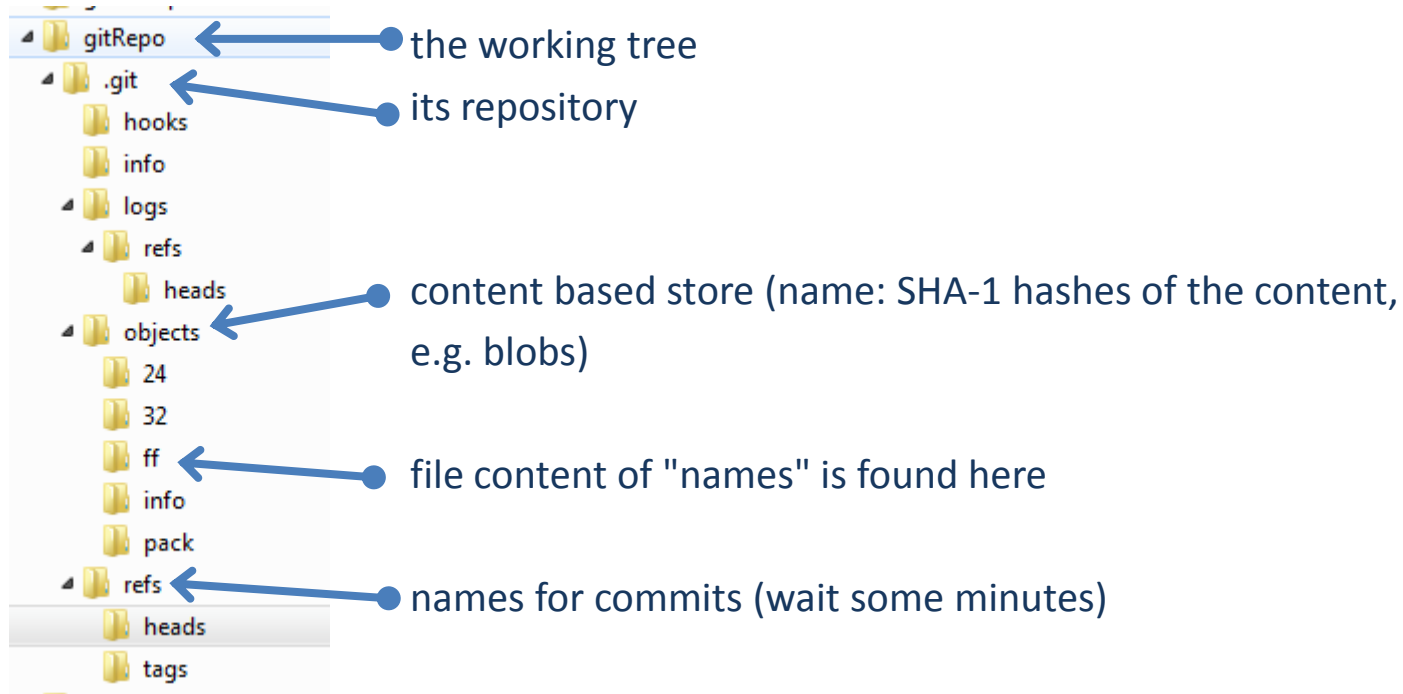
```
git log deploy-2012-12-14..HEAD --since="10 days ago" --grep="jira-3212" --no-merges
```


Useful Commands When Exploring A Repository

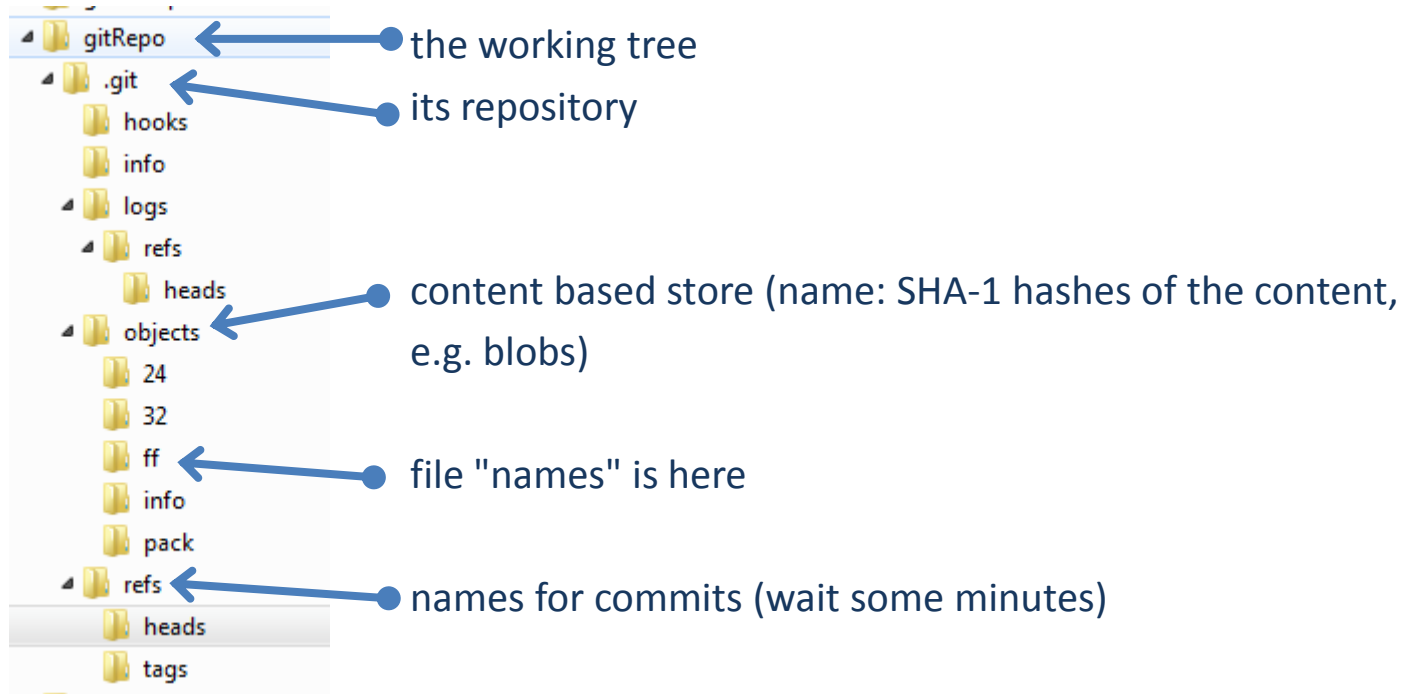
Command	Meaning
<code>git remote -v</code>	show the remote tracked repositories
<code>git branch -vv [-a]</code>	show all local branches and the remote branches they are connected to
<code>git config --global push.default simple</code>	enforce that you can only <i>push</i> to <i>remote</i> branches of the <i>same</i> name
<code>git branch name origin/name</code>	get remote branch name. <i>Advice</i> : use the <i>same</i> name locally
<code>git log HEAD~7.. --oneline</code>	the last 7 commits in short notation
<code>git log --graph --oneline</code>	show as a graph
<code>git status [-s]</code>	working tree status [-s short format, but easy to read]
<code>git blame <file></code>	show who changed which line and at what commit
<code>git diff [<file>]</code>	diff working tree and HEAD
<code>git diff --staged [<file>]</code>	diff index (staging area) and HEAD

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

The Structure Of A GIT Repository



The Structure Of A GIT Repository



--- 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* addressing

```
$ 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
```