Verteile Revisionskontrolle mit GIT

Christian Thäter

ct@pipapo.org

25. Juni 2007

Über diesen Vortrag

1. Was ist Revisionskontrolle?
2. Wie funktioniert GIT?
3. GIT Workshop

Fragen werden nach jedem Abschnitt beantwortet!

Was ist Revisionskontrolle?

Verwaltung und Historie eines Projekts

- Was wurde geändert
- Wer hat was geändert
- Wann wurde die Änderung vorgenommen
- Warum wurde die Änderung gemacht

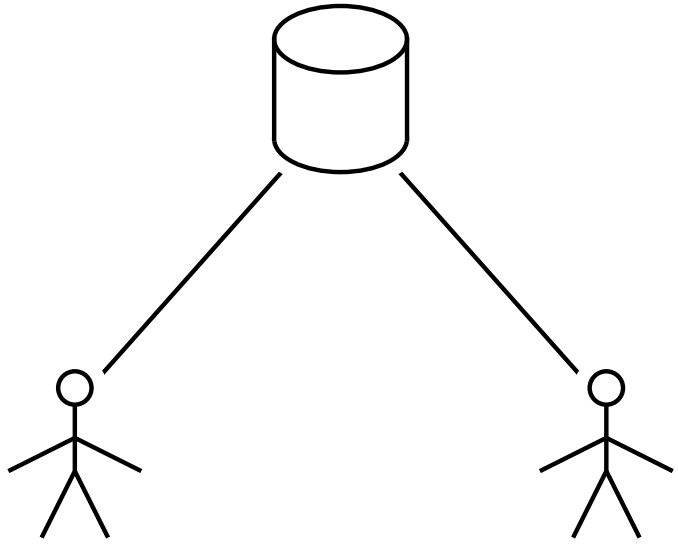
Was bietet Revisionskontrolle?

- Unlimitiertes Undo
- Dokumentation (der Historie)
- Hilfe bei Fehlersuche und Behebung
- Mehrere Varianten eines Projekts parallel pflegen
- Austausch von Änderungen mit Anderen

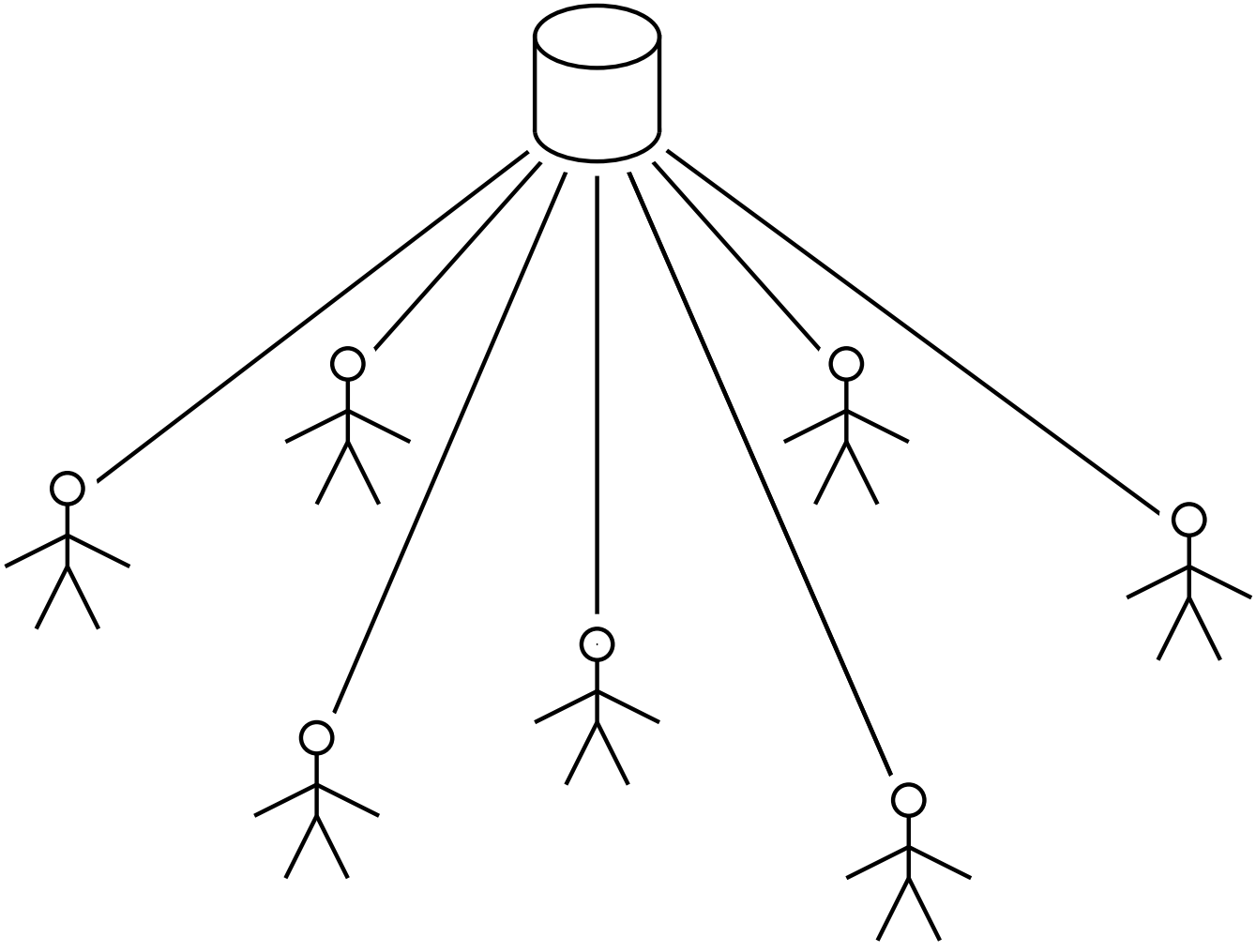
Sachen die man verwalten kann:

- Sourcecode
- Konfigurationsfiles
- Aufsätze, Diplomarbeit oder ähnliches
- "LaTeX-beamer" Präsentationen
- Grundsätzlich:
  - Hauptsächlich Text
  - Binäre Dateien die sich eher selten ändern
  - Eigenes Verzeichnis

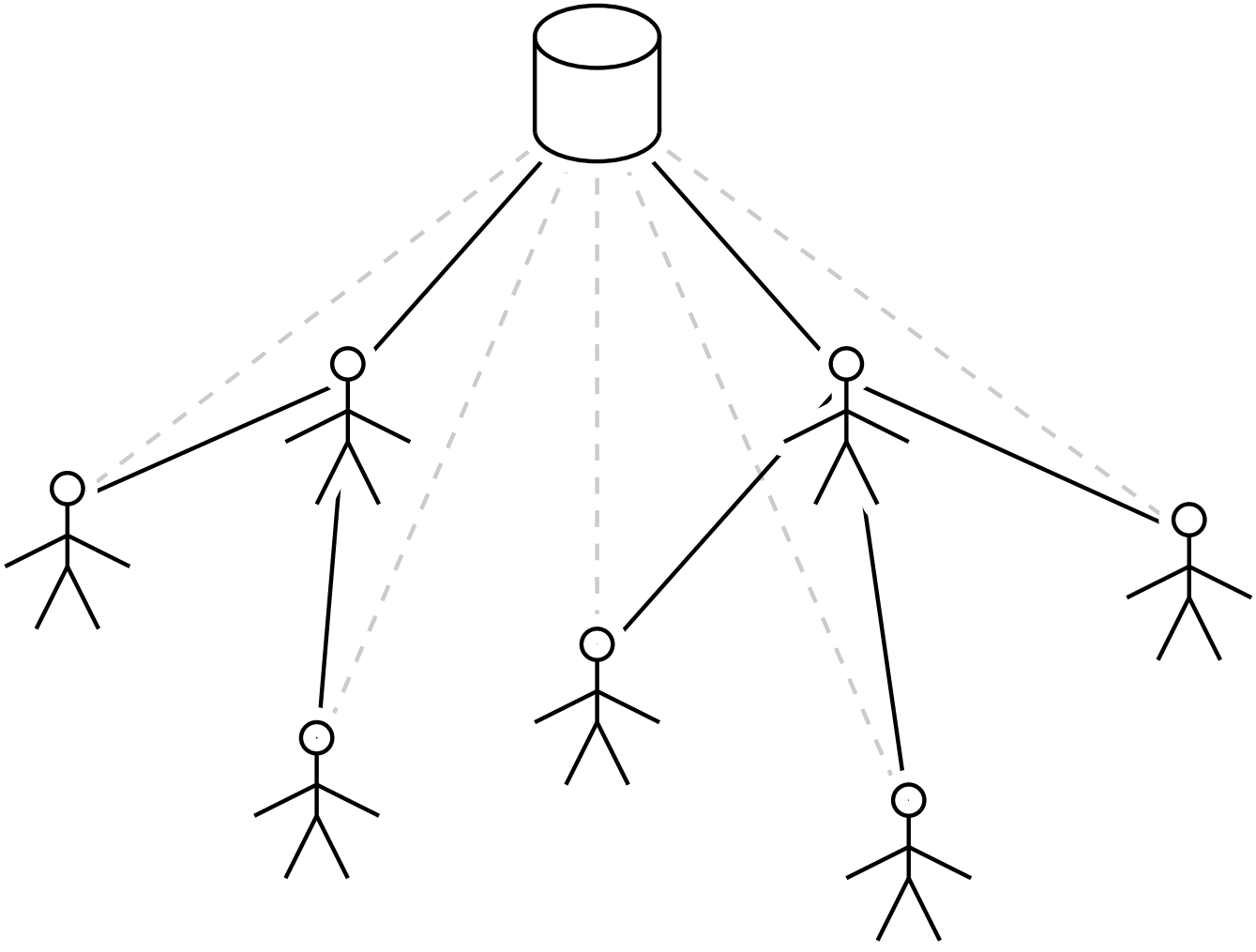
Klassische Revisionskontroll Systeme



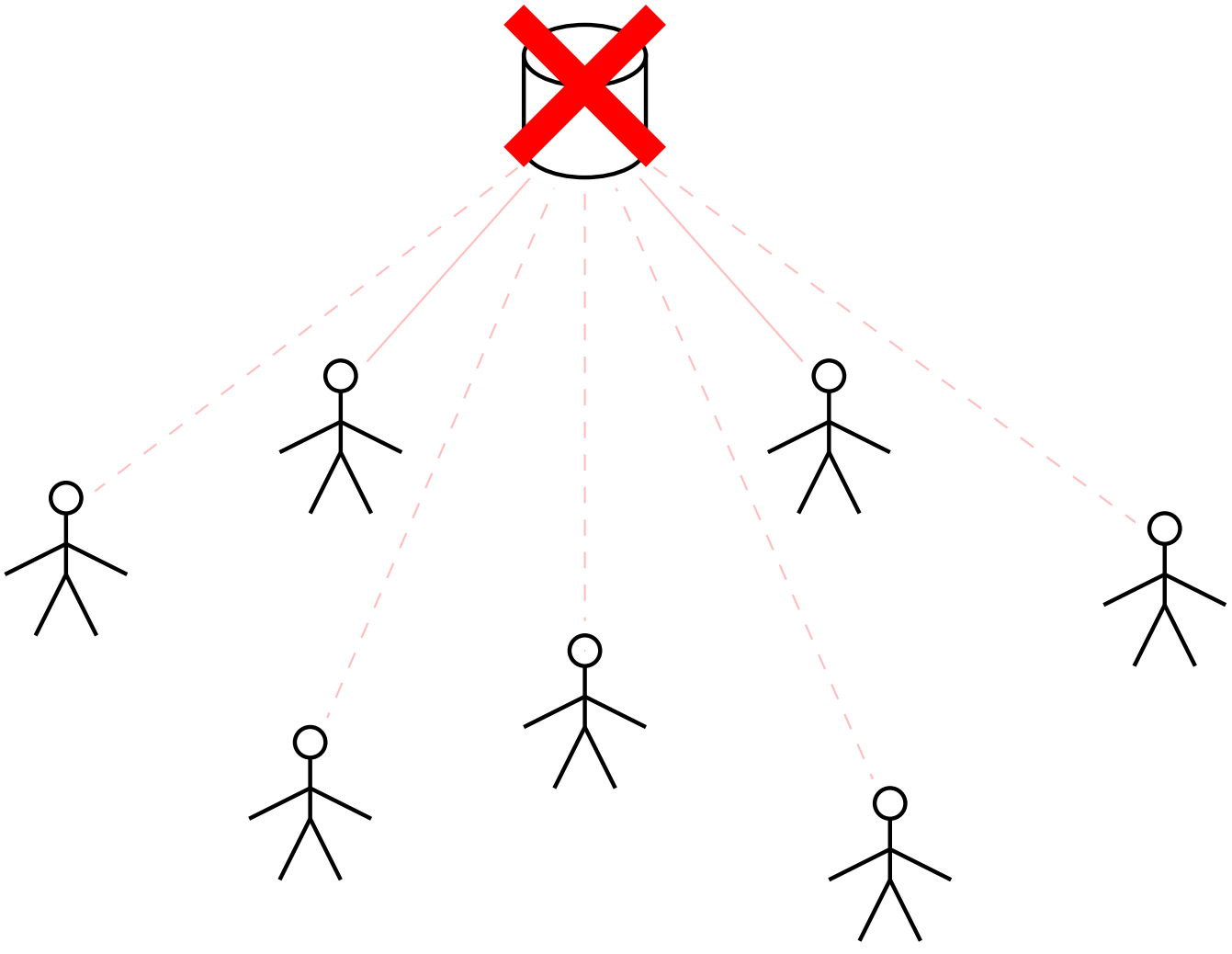
Klassische Revisionskontroll Systeme



Klassische Revisionskontroll Systeme



Klassische Revisionskontroll Systeme



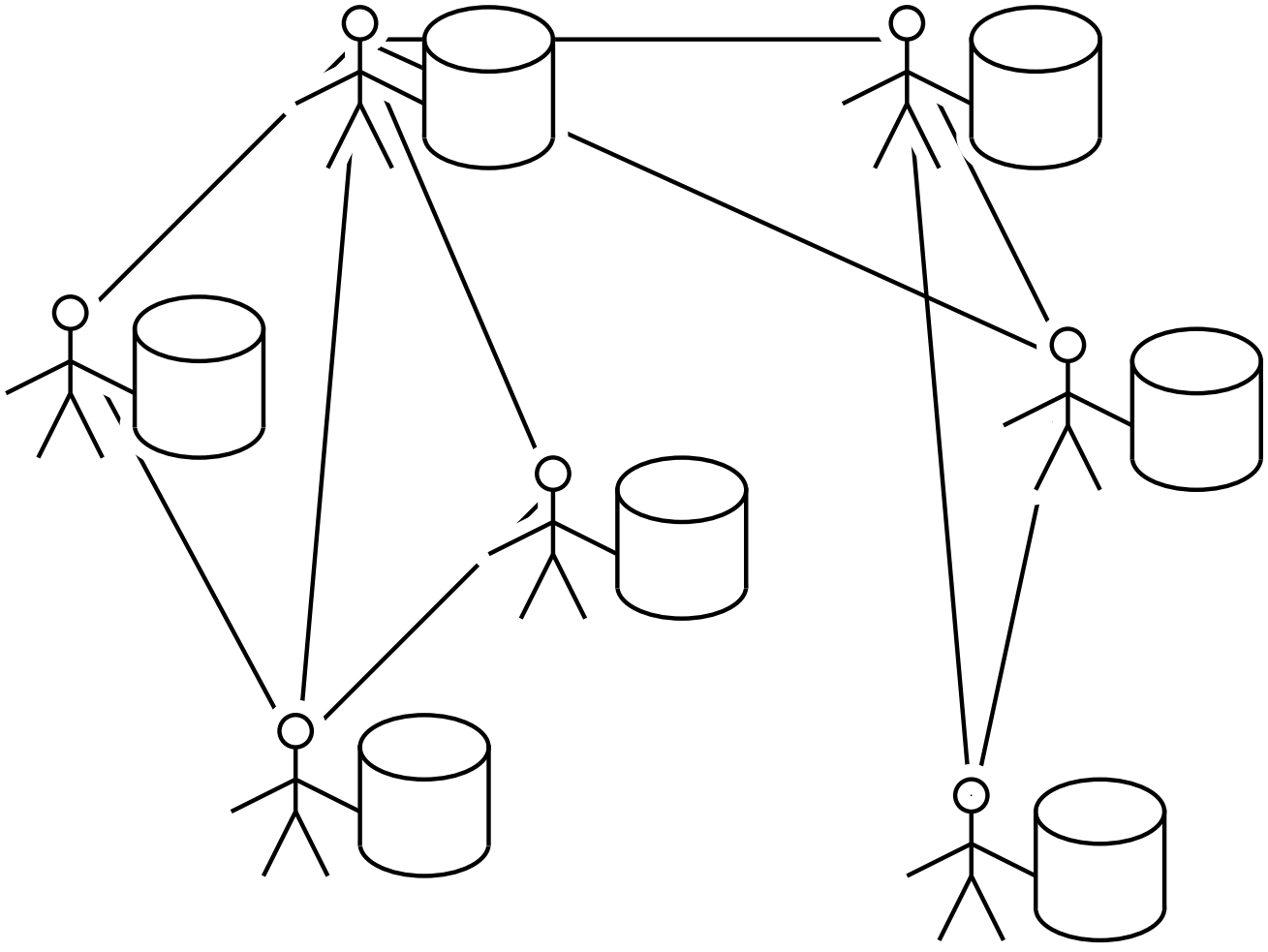
Klassische Revisionskontroll Systeme

- Latenz ist schlecht für die Performanz
- Der Server hat absolute Kontrolle
- Funktioniert nur wenn Server erreichbar
- Zu viele Committer
- Branchen ist oft nicht einfach

Verteilte Revisionskontroll Systeme

- Repository ist lokal
- Volle Historie
- Extrem schnell
- Nur der Besitzer selber braucht Schreibzugriff
- Branchen is extrem einfach und effizient
- Gespiegelte Repositories (Backup)
- Simple/clean Commits

Verteilte Revisionskontroll Systeme



Geschichte

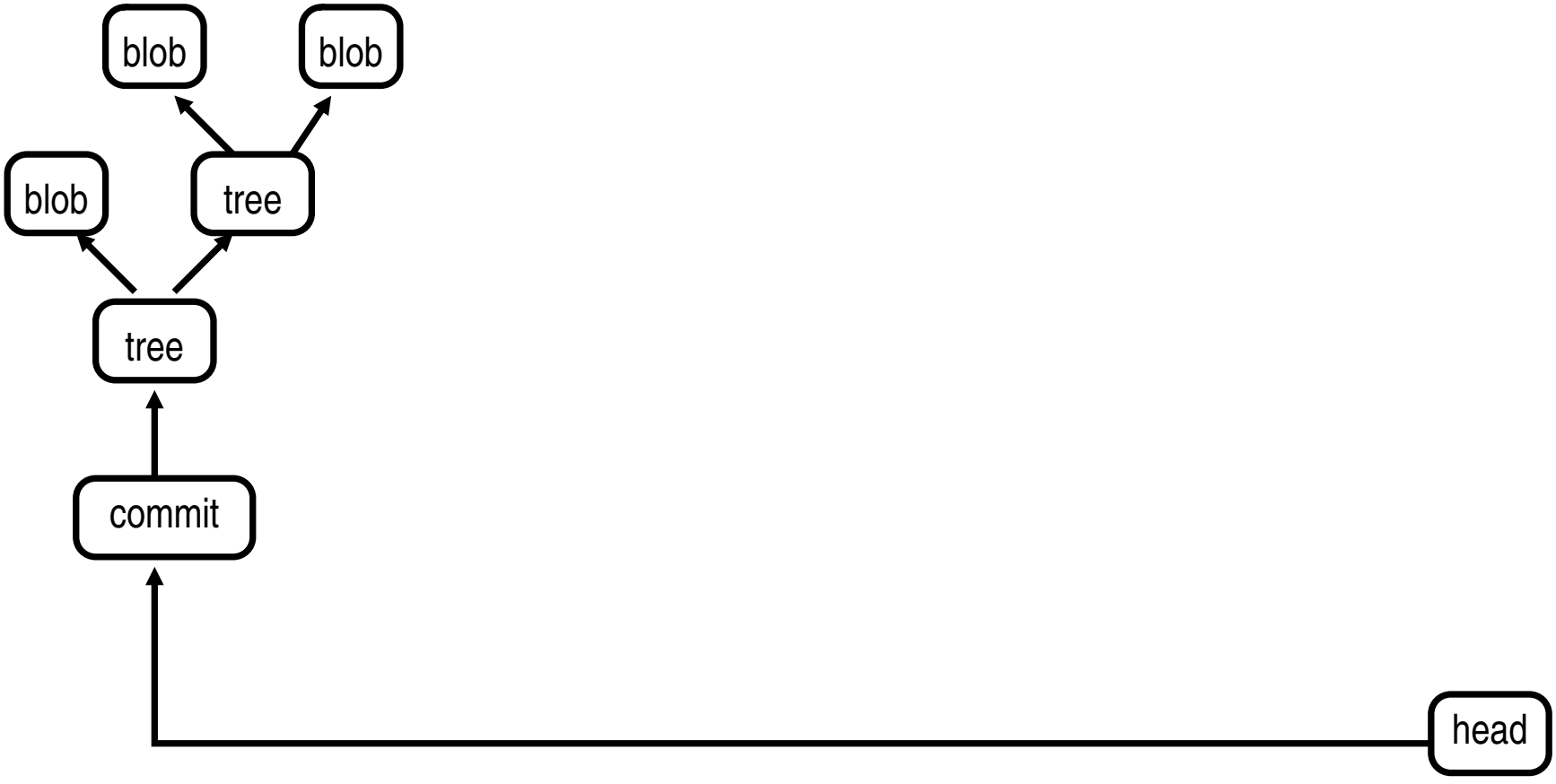
- Wurde von Linus für den Kernel entwickelt
- GIT heisst:
  - Drei Buchstaben die man aussprechen kann
  - “stupid. contemptible and despicable. simple”
  - “global information tracker”
  - “goddamn idiotic truckload of sh*t”
- Anfangs nur sehr simpler *“Content Tracker”*
- “plumbing” sind interne Befehle
- “porcelain” ist das Benutzer Interface

Aufbau

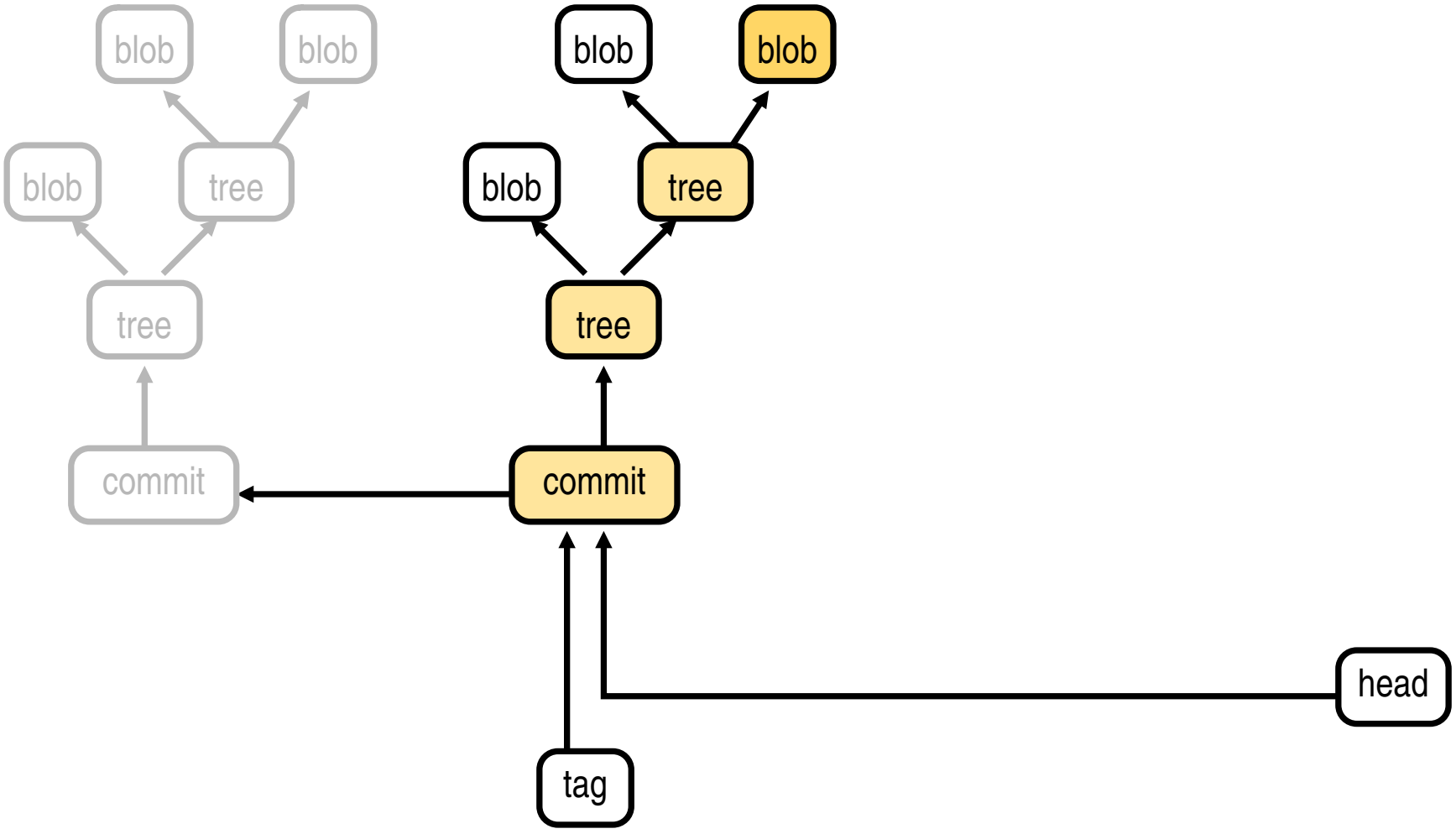
- Objektdatenbank
  1. blob : Datenobjekt
  2. tree : Verzeichnis
  3. commit : zeigt auf “tree”, parent “commits”
  4. tag : zeigt auf einen “commit”

  Alle Objekte sind mit ihrem SHA1 Hash addressiert
- Index : Status/Cache
- Heads : Referenzen auf “commits” oder “tags”
- Checkout : Arbeitskopie an der man arbeitet

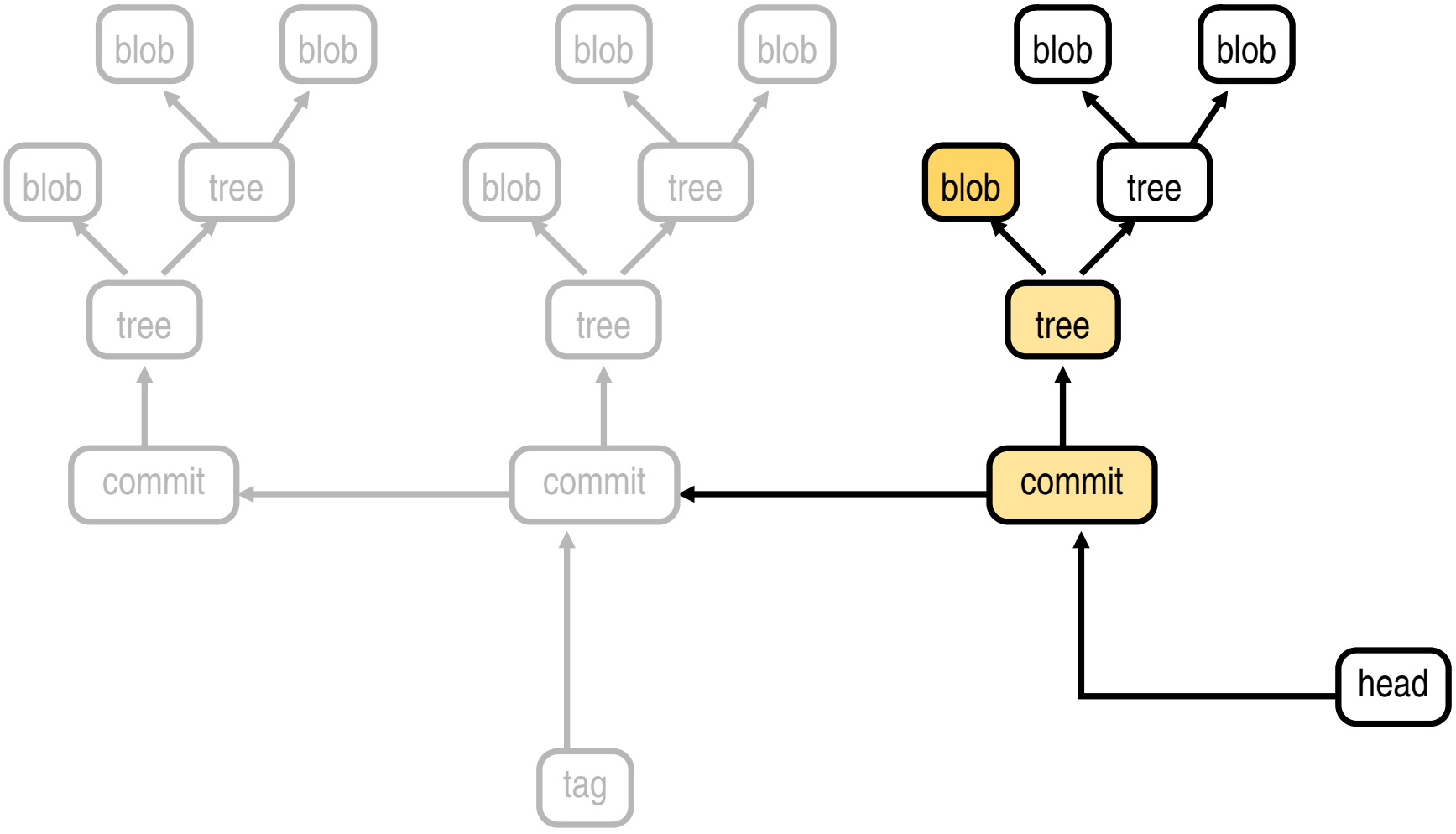
Aufbau



Aufbau


blob
blob
blob
tree
tree
commit
blob
blob
blob
tree
tree
commit
head
tag

Aufbau


blob
blob
blob
tree
tree
commit
blob
blob
blob
tree
tree
commit
tag
blob
blob
blob
tree
tree
commit
head

GIT ist platzsparend

Beispiel YAP Prolog:

GIT ist platzsparend

Beispiel YAP Prolog:

- Seit 2001 CVS; 1848 Revisionen

GIT ist platzsparend

Beispiel YAP Prolog:

- Seit 2001 CVS; 1848 Revisionen
- 18MB Sourcecode; 1547 Files

GIT ist platzsparend

Beispiel YAP Prolog:

- Seit 2001 CVS; 1848 Revisionen
- 18MB Sourcecode; 1547 Files
- 47MB CVS Repository

GIT ist platzsparend

Beispiel YAP Prolog:

- Seit 2001 CVS; 1848 Revisionen
- 18MB Sourcecode; 1547 Files
- 47MB CVS Repository
- 12MB GIT Repository

GIT ist platzsparend

Beispiel YAP Prolog:

- Seit 2001 CVS; 1848 Revisionen
- 18MB Sourcecode; 1547 Files
- 47MB CVS Repository
- 12MB GIT Repository
- 199MB GIT Repository (unpacked); 28584 Files

GIT ist platzsparend

Beispiel YAP Prolog:

- Seit 2001 CVS; 1848 Revisionen
- 18MB Sourcecode; 1547 Files
- 47MB CVS Repository
- 12MB GIT Repository
- 199MB GIT Repository (unpacked); 28584 Files

> *<hharrison> looks like the whole gcc repo*
> *should be around 450MB packed, not bad*
> *from 11GB svn*

GIT 1.5.x Installieren

```
$ git
Display all 144 possibilities? (y or n)
git                         git-gui                     git-reflog
git-add                     git-hash-object             git-relink
git-add--interactive        git-http-fetch              git-remote
git-am                      git-http-push               git-repack
git-annotate                git-imap-send               git-repo-config
git-apply                   git-index-pack              git-request-pull
git-applymbox               git-init                    git-rerere
git-applypatch              git-init-db                 git-reset
git-archimport              git-instaweb                git-resolve
git-archive                 git-local-fetch             git-rev-list
git-bisect                  git-log                     git-rev-parse
git-blame                   git-lost-found              git-revert
git-branch                  git-ls-files                git-rm
git-cat-file                git-ls-remote               git-runstatus
git-check-ref-format        git-ls-tree                 git-scm
--More--
```

‘git‘ genügt

- `git befehl` == `git-befehl`
- Kurze Hilfe: `git befehl -h`
- Manpage: `git befehl --help`
- Eine Handvoll Befehle kennen reicht

Jetzt gehts los ...

- Sich GIT vorstellen

```
$ git config --global user.name "Dein Name"
$ git config --global user.email deine@e.mail
```

legt `~/.gitconfig` an.

Repository anlegen

- Neues Repository erzeugen:

  ```
  cd mysourcedir
  git init
  ```

- Vorhandenes Repository clonen:

  `git clone` *`giturl`* `[dir]`

  - http://host/repository
  - git://host/repository
  - ssh://host/repository
  - user@host:repository
  - /verzeichnis

git clone optimieren

- Objektdatenbank hardlinken

  `git clone -l /verzeichnis dir`

- Objektdatenbank referenzieren

  `git clone -s /verzeichnis dir`

- Vorhandene Daten aus Objektdatenbank mitbenutzen

  `git clone --reference /verzeichnis` *giturl*

Commit vorbereiten

`git status`

`git diff`

- Status einzelner Files
- detailierte Änderungen

`git add` *name*

- Verzeichnisse werden recursiv hinzugefügt
- Änderungen müssen mit git-add hinzugefügt werden

Commit

- Vorher mit ‘git add‘ alles hinzugefügt:

  ```
  git commit -m 'message'
  ```

- File Liste mit angeben:

  ```
  git commit -m 'message' -- files..
  ```

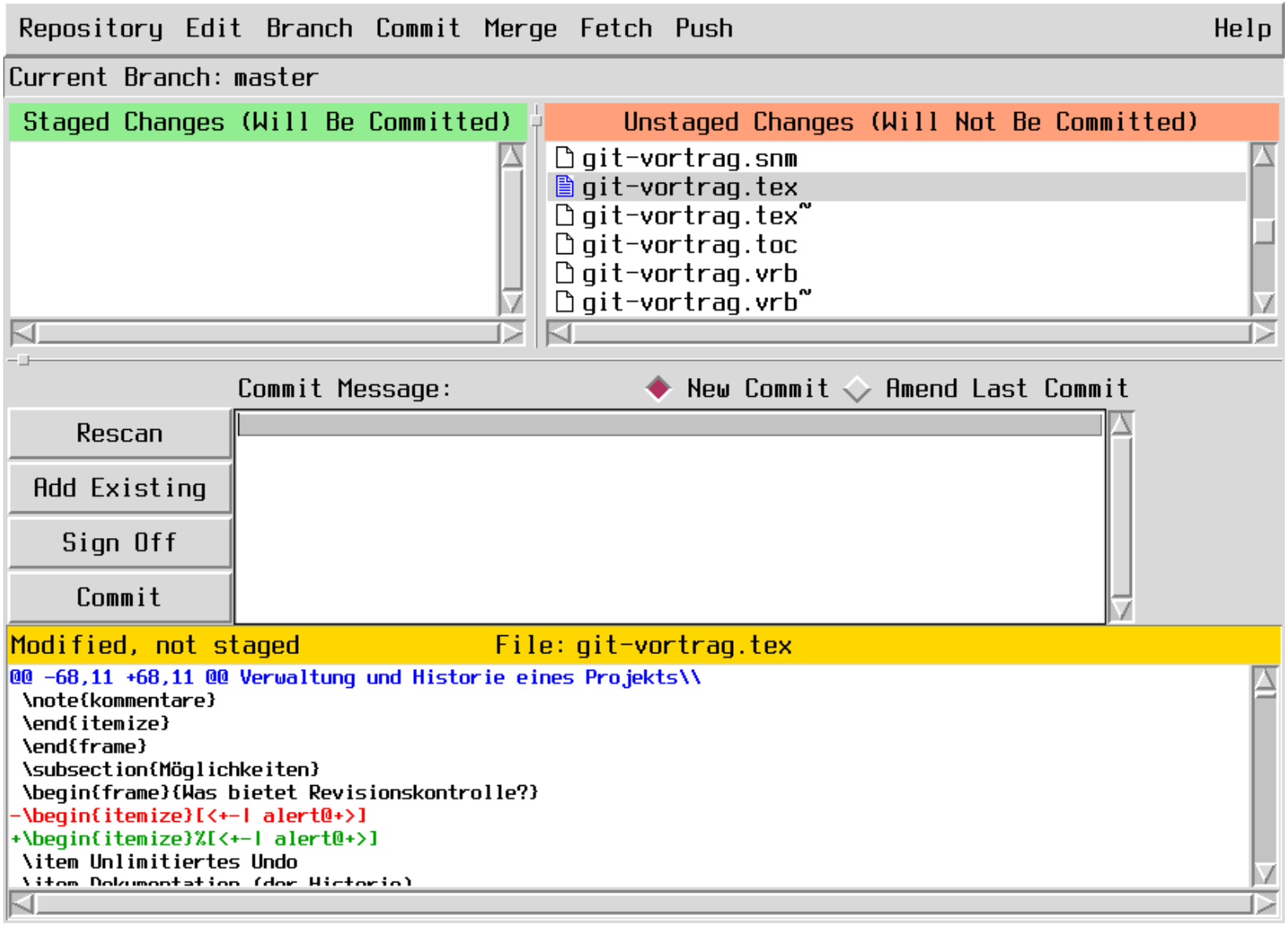
- Alle Änderungen (wie CVS):

  ```
  git commit -a -m 'message'
  ```

- Menugeführt, auch einzelne Hunks:

  ```
  git commit --interactive -m 'message'
  ```

... oder mit ‘git gui’



Branchen

- Neuen Branch anlegen und Tree auschecken:
  `git checkout [-m] -b newbranch [head]`
  - `HEAD` ist default
  - `HEAD^` voriger commit
  - `HEAD~2` vor-voriger commit usw...
  - "head" kann alles sein was zu einem commit auflöst
  - mit Zeitangabe: `...@{2 days ago}`
- Branches auflisten: `git branch`
- Branch wechseln: `git checkout branch`
- `git branch -d name` löscht Branch

Andere Repositories

- Andere Repositories registriern:

  ```
  git remote add name giturl
  ```

- Registrierte Remotes auflisten:

  ```
  git remote
  ```

- Remote Branches auflisten (local):

  ```
  git branch -r
  ```

- Remote Heads listen:

  ```
  git ls-remote name
  ```

Updates und Mergen

- Änderungen von *name* ins Repository holen:

  `git fetch [name] [refspec]`

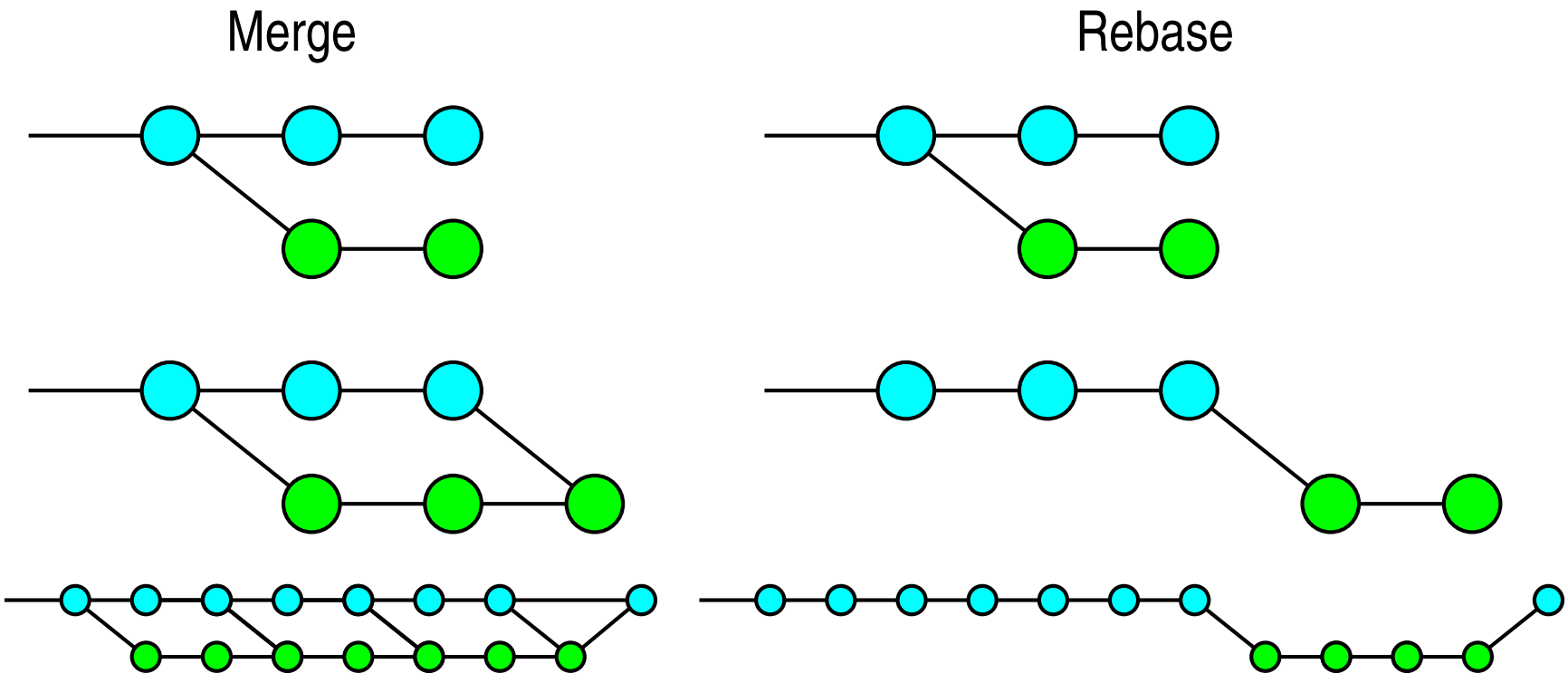
  Ändert nichts an der Arbeitskopie!

  refspec: `+srcbranch:dstbranch`
- Änderungen mergen:

  `git pull [name] [refspec]`

  macht erst "fetch" und dann "merge"
- `git merge` braucht man nur für Sonderfälle
- Vorm Mergen committen!

git rebase



- Rebase ändert Historie und SHA1 Hashes!

Konflikte lösen

Konflikte ansehen:

```
git status
git diff
```

“git diff” hat eine spezielles Format:

```
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:file.txt
 +Hello world
++=======
+ Goodbye
++>>>>>>> 77976...:file.txt
```

Konflikte lösen

1. Konflikt lösen

```
@@@ -1,1 -1,1 +1,1 @@@
- Hello world
 -Goodbye
++Goodbye world
```

2. Files adden `git add ...`
3. Commit `git commit ...`

Kein commit bei unaufgelösten Konflikten

... oder "git gui"

Änderungen veröffentlichen

- Per email

  `git format-patch`

  `git send-email` wird dann mit `git am` eingespielt
- Git Server `git daemon` von inetd starten
- Webserver per http:// (suboptimal)
- Öffentlich auf http://repo.or.cz/
- Zum Server senden mit `git push`
- Öffentliche Repositories sind “bare”, kein Checkout

Repository Wartung

- Konsistenz des Repositories überprüfen:

  ```
  git fsck
  ```

- Repository komprimieren:

  ```
  git gc [--prune]
  ```

Was nicht gesagt wurde:

- GPG signierte Tags
- Eigene merge und diff Engines
- Bugs mit “git bisect” halbautomatisch finden
- Recovery mit “revlog”
- “gitk” und “gitweb”
- Mob Software, anonyme commits

- ... viele andere Features

Danke

Auf Wiedersehn