Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Florian Franzmann, Martin Hoffmann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

25. Mai 2012

Überblick

Nachtrag feingranulares Testen

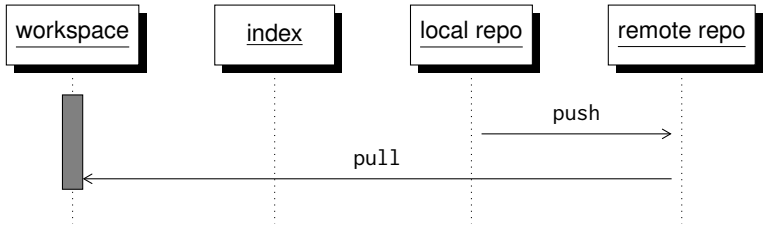
- gängiger Fehler: alle Testfälle in einer Datei
- schlägt ein Testfall fehl, muß der Tester erst herausfinden warum

⇒ immer eine `main` pro getestetem Kriterium!

- mit möglichst wenigen Ausstiegspunkten
  - einer für Fehlerfall
  - einer für Erfolgsfall

Table of Contents

git-Arbeitsschritte – entfernt I



git push [<remote> [<branch>]]

- schiebt Änderungen nach Remote in den ausgewählten Branch
- dies geht nur, wenn lokales Repo auf dem aktuellen Stand ist!
- sonst beschwert sich git:

```
% git push origin master
To /tmp/test.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '/tmp/test.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

⇝ wir müssen das Repository erst auf den aktuellen Stand bringen

git pull [<remote> [<branch>]]

- holt Änderungen aus dem ausgewählten Remote in den aktuellen Branch
- verschmilzt aktuellen Branch mit geholten Änderungen
- gleicher Effekt wie `% git fetch && git merge FETCH_HEAD`
-

```
% git pull origin

remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /tmp/test
   38b95cb..8ec6e93  master     -> origin/master
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- jemand hat in der Zwischenzeit die gleiche Stelle der Datei verändert

⇝ Konflikte müssen von Hand behoben werden

Konflikt beheben

-

```
% cat test.txt
hallo
<<<<<<< HEAD
welT!     meine Version
=======
Welt!     Version in origin/master
>>>>>>> 8ec6e9309fa37677e2e7ffcf9553a6bebf8827d6
```

⇝ sich für eine von beiden Versionen entscheiden

⇝ die andere beseitigen

- Konflikt auflösen:

```
% git commit -a
[master 4d21871] Merge branch 'master' of /tmp/test
```

```
% git push origin master
Counting objects: 5, done.
Writing objects: 100% (3/3), 265 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/test.git
   8ec6e93..278c740  master -> master
```

Arbeitsablauf mit Branches

In den meisten Versionsverwaltungssystemen

1. Featurebranch anlegen
2. Feature im Branch implementieren, testen
3. Featurebranch mit `master` veschmelzen
4. ggf. Featurebranch löschen

Naiver Ansatz

⇝ skaliert nicht!

Warum `branch/edit/merge` nicht skaliert

Aufgaben von Versionsverwaltung

1. Codeschreiben unterstützen
2. Konfigurationsmanagment/Branches
   ⇝ z. B. Release-Version, HEAD-Version ...

⇝ Konflikt

1. braucht Checkpoint-Commits
   - möglichst oft einchecken

   ⇝ skaliert nicht
2. braucht Stable-Commits
   - nur einchecken, wenn Commit perfekt

   ⇝ nicht praktikabel

Lösung mit `git`: öffentlicher vs. privater Branch

Öffentlicher Branch ⇝ verbindliche Geschichte

Commits sollen { atomar, gut dokumentiert, linear, unveränderlich } sein

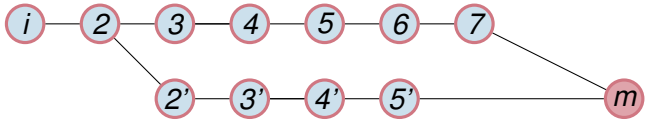
Privater Branch ⇝ Schmierpapier

- für einzelnen Entwickler
- möglichst lokal
- wenn im zentralen Repo ⇝ sich auf Privatheit einigen

Aufräumen

- verschmelze nie direkt privaten mit öffentlichem Branch
  - Historie wird sonst unübersichtlich

  ⇝ nicht einfach `git merge` im `master` machen



- vorher immer erst `git`
  - `rebase` ⇝ Commits auf Branch anwenden
  - `merge --squash` ⇝ einzelnen Commit aus Branch-Commits
  - `commit --amend` ⇝ Commit-Nachricht nachbearbeiten
- Ziel: öffentlicher Commit ≡ Kapitel eines Buches

Michael Crichton

Great books aren't written – they're rewritten.

Arbeitsablauf für kleinere Änderungen

- `git merge --squash`

⇝ zieht Änderungen aus einem Branch in den aktuellen Index

Branch

```
% git checkout -b private_feature_branch (Branch anlegen)
% touch file1.txt
% git add file1.txt
% git commit -am "WIP" (Änderungen in Branch einchecken)
```

Merge

```
% git checkout master (nach master wechseln)
% git merge --squash private_feature_branch
(Änderungen auf Index von master anwenden)
% git commit -v (Änderungen einchecken)
```

Arbeitsablauf für größere Änderungen

- `git rebase`

⇝ wendet Commits auf einen anderen Branch an

Im Feature-Branch

```
% git rebase --interactive master
```

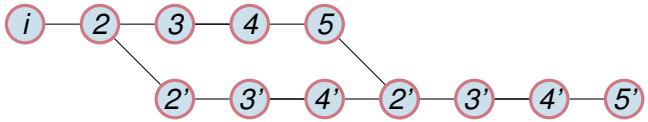
`pick` ⇝ übernimmt Commit

```
pick ccd6e62 Work on back button
pick 1c83feb Bug fixes
pick f9d0c33 Start work on toolbar
```

`squash` ⇝ verschmilzt Commit mit Vorgänger

```
pick ccd6e62 Work on back button
squash 1c83feb Bug fixes # mit Vorgaenger verschmelzen
pick f9d0c33 Start work on toolbar
```

Aufsetzen auf bestehenden Zweigen (rebase)



- Patches aus dem “unterem” Zweig werden auf den “oberen” aufgespielt
- Die Historie ist nun linear
- Linearisierte Änderungen lassen sich häufig einfacher bewerten
- Vorsicht!
  - Verzweigungen vom alten Zweig können nun nicht mehr zusammengeführt werden
  - Keine gemeinsamen Vorgänger mehr
  - Visualisierung der Historie ist nun bestenfalls verwirrend

Wenn der Feature-Branch im Chaos versinkt?

⇝ aufgeräumten Branch anlegen

1. auf Branch `master` wechseln
   `% git checkout master`
2. Branch aus `master` erzeugen
   `% git checkout -b cleaned_up_branch`
3. Branch-Änderungen in den Index und die Working Copy ziehen
   `% git merge --squash private_feature_branch`
4. Index zurücksetzen
   `% git reset`

- danach Commits neu zusammenbauen

git-Kommandos: Austausch von Quellcode I

- initiales *Klonen*:
  `% git clone http://www4.cs.fau.de/...`
- Einspielen entfernter Änderungen:
  `% git pull`
  ⇒ äquivalent zu
  `% git fetch && git merge`
- Mehrere Repositories registrieren:
  `% git remote add 32-stable git://git.kernel.org/.../...`
- registrierte Remotes untersuchen:
  `% git remote -v`

git-Kommandos: Austausch von Quellcode II

- alle Remotes nachladen (aktueller Branch wird nicht verändert)
  `% git remote update`
- lokalen Branch aus dem neuen ‘Remote“ anlegen:
  `% git checkout -b work 32-stable/master`
- Unterschiede zwischen lokalem und entferntem Branch untersuchen:
  `% git log ..origin/master`
- aktuelle Änderungen auf dem entfernten Branch neu aufspielen:
  `% git pull --rebase`
- die neuste Änderung untersuchen:
  `% git show`

git-Kommandos: Austausch von Quellcode III

- herausfinden wer für welche Zeilen einer Datei verantwortlich ist:
  `% git blame`
- die letzten drei Änderungen als Patch:
  `% git format-patch HEAD~~`
- Sendeziel für Patchversand per E-Mail vorgeben:
  `% git config sendemail.to=...@...`
- Patchset letzten drei Änderungen per E-Mail senden:
  `% git send-email --compose HEAD~~`
- einen Patch aus einer Mailbox anwenden:
  `% git am < <Datei>`

Lesenswertes zu git

- `http://gitready.com`
- `http://book.git-scm.com/`
- `http://gitcasts.com`
- `http://eagain.net/articles/git-for-computer-scientists/`
- `http://sandofsky.com/blog/git-workflow.html`
- `http://365git.tumblr.com/`

Table of Contents

Gerrit

- Infrastruktur für Softwareprojekte
- verwaltet `git`-Repositories
- unterstützt Code-Reviews ⇝ dazu ein anderes Mal mehr
- für die Anmeldung notwendig:
  - OpenID eines beliebigen Anbieters, z. B.
    - RRZE ⇝ `http://openid.fau.de/<login>`
    - Facebook
    - Google

    ⇝ `http://openid.net/get-an-openid/`
  - öffentlicher SSH-Schlüssel

Anmeldung an Gerrit I

1. mit OpenID anmelden
   `http://vamos.cs.fau.de/gerrit`
   ⇝ als Benutzernamen den CIP-Login angeben
2. eine Mail mit den Namen der Gruppenmitglieder an `i4ezs@lists.cs.fau.de` schreiben
3. ggf. SSH-Schlüssel erstellen
   `% ssh-keygen -t rsa -f i4gerrit`
4. und verwenden
   - Datei `i4gerrit` nach `~/.ssh` verschieben
   - In Gerrit in *Settings* → *SSH Public Keys*: öffentlichen Schlüssel hinzufügen ⇝ `i4gerrit.pub`
   - auf dem Arbeitsplatzrechner Datei `~/.ssh/config` anpassen:

Anmeldung an Gerrit II

```
Host i4gerrit
  HostName vamos.cs.fau.de
  Port 29418
  User <login>
  IdentityFile ~/.ssh/i4gerrit
  ForwardAgent no
  ForwardX11 no
```

Anmeldung an Gerrit III

5. Rechte anpassen

Anmeldung an Gerrit IV

6. **sicherstellen, dass der Quellcode lokal eingecheckt ist!**
7. sich für eine der Lösungen für Aufgabe 1 entscheiden und diese hochladen
   ⇝ im entsprechenden lokalen Repository
   - `master`-Branch ins Gerrit hochladen
     ```
     % git push ssh://i4gerrit/vezs/gruppe<X> master:master
     ```
   - Branch für Aufgabe 1 anlegen und hochladen:
     ```
     % git checkout -b aufgabe1
     % git push ssh://i4gerrit/vezs/gruppe<X> aufgabe1:aufgabe1
     % git checkout master
     ```
   - `remote origin` in `.git/config` konfigurieren:

Anmeldung an Gerrit V

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = ssh://i4gerrit/vezs/gruppe<X>
```

8. wir werden in Zukunft mit zwei Remote-Repositories arbeiten
   `upstream` das Vorgabe-Repository
   `origin` das Gruppen-Repository
   - `remote upstream` konfigurieren
     ```
     % git remote add upstream \
           <login>@faui0sr0.cs.fau.de:/<...>/vezs-uebung.git
     ```
   - Alias für *pull von upstream* in `.git/config` konfigurieren:

Anmeldung an Gerrit VI

```
% git config alias.pu '!git fetch origin -v; \
                       git fetch upstream -v; \
                       git merge upstream/master'
```

⇝ `git pu` zieht von `origin` und `upstream` und macht `merge`

Fragen?