Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Florian Franzmann, Martin Hoffmann, Isabella Stilkerich

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

24. April 2013

Überblick

Zwei Prinzipien für die Übung

KISS – Keep it Small and Simple!

- Kleine Softwaremodule mit geringer Kopplung
- **Eine** (C-)Funktion löst **eine** Aufgabe
- ☞ Bessere Wartbarkeit, Testbarkeit, Verifizierbarkeit

DRY – Don‘t repeat yourself!

- Code nicht unnötig duplizieren
- Oft benutzten (getesteten) Code wiederverwenden
- ☞ Einsatz von Bibliotheken
- Ein Beispiel: libmathe16

Verzeichnisstruktur in der Übung

- Quellverzeichnis (source)
- Hier liegen die Quelldateien
  - include ← Schnittstellenbeschreibungen (.h)
  - src ← Implementierung (.c)
  - tests ← Testfallimplementierungen (.c)
  - (cmake) ← (eigene CMake Erweiterungen)
- Binärverzeichnis (binary)
- Hier landen ausschließlich(!) generierte Dateien
  - Objektdateien (.o)
  - Bibliotheken (.a)
  - Ausführbare Dateien
- ☞ „Out-of-Source Build“
- ☞ Beispiel

Buildsystem

DRY: Befehle (gcc/ar/...) nicht unnötig händisch wiederholen

- Stupides Wiederholen von Befehlen ist fehlerträchtig!
- Lösung: Buildsystem
  - Automatisiertes Bauen
  - Automatisches Auflösen von Abhängigkeiten
  - Viele existierende Lösungen: make, ANT, Maven, u.v.m.
- Wir nutzen **CMake**

Inhaltsverzeichnis

Was ist CMake?

- Ein Meta-Buildsystem!
  - Erzeugt Buildsystemdateien
    - **Makefiles** (GNU, NMake, ...)
    - Projektdateien (KDevelop, Eclipse, Visual Studio, Xcode)
  - Einfache, skriptähnliche Sprache
  - Plattform-/Betriebssystemunabhängig
  - Ermöglicht „Out-of-Source Builds“
- Weit verbreitet
  - KDE, MySQL, LLVM, u.v.m.

CMake in der Übung

- Konfigurationsdatei(en): CMakeLists.txt
- Separat in jedem Unterverzeichnis
  - Ausgehend vom Basisverzeichnis → `add_subdirectory(...)`
- Definition von sog. „Targets“
  - `add_executable(<Targetname> <Quelldatei1.c> <Quelldatei2.c>)`
  - `add_library(<Libraryname> <Quelldatei1.c> <Quelldatei2.c>)`
- Hinzubinden von Bibliotheken
  - `target_link_libraries(<Targetname> <Libraryname>)`
  - Abhängigkeiten werden automatisch erkannt
- Manuelle Festlegung von Abhängigkeiten
  - `add_dependency(<Targetname1> <Targename2>)`
- ☞ Beispiel

Erzeugen von Makefiles

- **Außerhalb** des Quellverzeichnisses
- `% cmake <Pfad zum Quellverzeichnis>`
- `% make help` zeigt alle möglichen Targets

☞ Beispiel

Inhaltsverzeichnis

Testfallintegration mit CMake

- CMake unterstützt die Integration von Tests im Softwareprojekt
- Automatisierte Ausführung und Auswertung von Testläufen
- Konfigurationsdatei: tests/CMakeLists.txt
  - Ausführbares Target:

    ```
    add_executable(plus_test plus_test.c)
    ```
  - Hinzubinden der zu testenden Bibliothek:

    ```
    target_link_libraries(plus_test mathe)
    ```
  - Bekanntmachen als Testfall:

    ```
    add_test(MatheTest_PLUS plus_test)
    ```
- `make test` führt Tests aus
- Automatische Testauswertung
  - Anhand Rückgabewert (0 → OK, -1 → Fehler)
  - Parsen von Ausgaben

☞ Beispiel

Verzeichnisstruktur

- Quellverzeichnis

```
% tree ~/source
~/source
|-- CMakeLists.txt
|-- include
|   `-- mathe.h
|-- src
|   |-- CMakeLists.txt
|   |-- abs.c
|   `-- plusminus.c
`-- tests
    |-- CMakeLists.txt
    |-- abs_test.c
    `-- plus_test.c
```

- Binärverzeichnis

```
% cd ~/binary
% cmake ../source
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Checking whether C compiler has -isysroot
...
-- Configuring done
-- Generating done
-- Build files have been written to: ~/build
% make
[ 20%] Building C object src/CMakeFiles/mathe.dir/plusminus.c.o
[ 40%] Building C object src/CMakeFiles/mathe.dir/abs.c.o
Linking C static library libmathe.a
[ 60%] Built target mathe
Scanning dependencies of target abs_test
[ 80%] Building C object tests/CMakeFiles/abs_test.dir/abs_test.c.o
Linking C executable abs_test
[ 80%] Built target abs_test
Scanning dependencies of target plus_test
[100%] Building C object tests/CMakeFiles/plus_test.dir/plus_test.c.o
Linking C executable plus_test
[100%] Built target plus_test
% make test
Running tests...
Test project ~/build
    Start 1: MatheTest_PLUS
1/2 Test #1: MatheTest_PLUS .......   Passed    0.00 sec
    Start 2: MatheTest_ABS
2/2 Test #2: MatheTest_ABS  .........***Failed    0.00 sec

50% tests passed, 1 tests failed out of 2

Total Test time (real) =   0.02 sec

The following tests FAILED:
     2 - MatheTest_ABS (Failed)
Errors while running CTest
```

Testen

- Erste Grundregeln:
  - Möglichst feingranular testen
  - Einzelne Testfälle für einzelne Funktionen!
  - Beachte die Grenzen der Datenypen! → `INT16_MAX, INT16_MIN`
- Testfälle müssen **zumindest** den gesamten erreichbaren Code abdecken.
- Hilfsmittel: Code Coverage Analysewerkzeug

Achtung

Testfälle können nur die Anwesenheit von Fehlern zeigen, nicht deren Abwesenheit! (→ vgl. Verifikation)

Codeüberdeckung mit gcov/lcov

- Werkzeug aus der **gcc** Toolchain
- Instrumentierung des Binärcodes
- Protokollieren der Programmausführung
  - Wie oft wird jede Codezeile ausgeführt?
  - Welche Zeilen werden überhaupt ausgeführt?
  - Welche Verzweigungen wurden genommen?
- HTML Ausgabe: lcov

→ Tests solange verfeinern, bis alles überdeckt ist!

Inhaltsverzeichnis

Anforderungen

Typische Aufgaben eines Versionsverwaltungssystems sind:

- sichern alter Zustände (⇒ commits)
- Zusammenführung paralleler Entwicklung
- Transportmedium

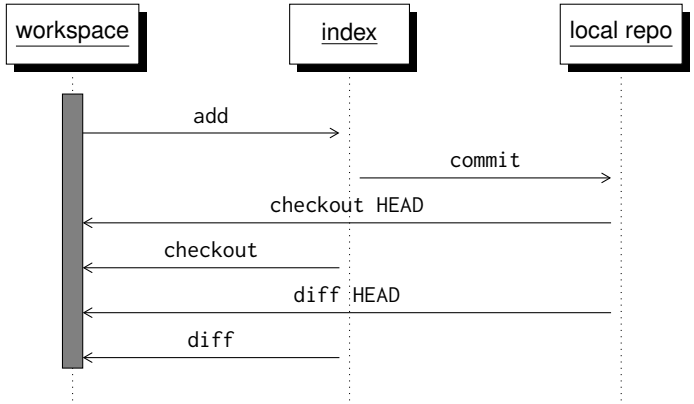
Idealerweise zusätzlich:

- Unabhängige Entwicklung ohne zentrale Infrastruktur

git

- wir werden in VEZS `git` verwenden
- 2005 von Linus Torvalds für den Linux-Kernel geschrieben
- Konsequenz der Erfahrungen mit *bitkeeper*
- Eigenschaften:
  - dezentrale, parallele Entwicklung
  - Koordinierung hunderter Entwickler
  - Visualisierung von Entwicklungszweigen

`git`-Arbeitsschritte – lokal



`git`-Arbeitsschritte

- initial Repository herunterladen:
  ```
  % git clone <URL>
  ```
- oder anlegen:
  ```
  % git init
  ```
- Commit im Index zusammenbauen (⇒ „Verladerampe“):
  ```
  % git add <Datei1>
  % git add <Datei2>
  % ...
  ```
- anschauen was bei `git commit` passieren würde:
  ```
  % git status
  ```
  oder
  ```
  % git diff --cached
  ```
- anschließend Index an das Repository übergeben:
  ```
  % git commit
  ```

`git`-Kommandos: Lokale Quellcodeverwaltung I

- Repository erstellen:
  ```
  % git init
  ```
- Änderung hinzufügen:
  ```
  % git add <Datei>
  ```
- oder interaktiv:
  ```
  % git add -i
  ```
- feingranulares hinzufügen:
  ```
  % git add -p
  ```
- Änderungen einchecken:
  ```
  % git commit -i <Datei1> <Datei2> ...
  ```

git-Kommandos: Lokale Quellcodeverwaltung II

- alles was nicht im git ist löschen:
  `% git clean -d <Pfad>`
  nur anzeigen, was gelöscht werden würde:
  `% git clean -n -d <Pfad>`
- herausfinden was beim nächsten Commit verändert wird:
  `% git diff --cached`
- oder als Kurzzusammenfassung:
  `% git status`
- geänderte aber noch nicht eingecheckte Datei zurücksetzen:
  `% git checkout -- <Datei>`

git-Kommandos: Lokale Quellcodeverwaltung III

- das Log anschauen:
  `% git log`
  mit Graph:
  `% git log --graph`
- herausfinden, was im letzten Commit verändert wurde:
  `% git whatchanged`
- einen Commit rückgängig machen:
  `% git revert <commit-id>`
- Änderungen sichern, aber noch nicht einchecken:
  `% git add ...`
  `% git stash`

git-Kommandos: Lokale Quellcodeverwaltung IV

- gesicherte Änderungen wieder hervorholen:
  `% git stash apply`
- Stashinhalt anzeigen:
  `% git stash list`
- Stash-Element löschen:
  `% git drop <id>`
- einen Branch anlegen:
  `% git branch <Name>`
- alle registrierten Branches anzeigen:
  `% git branch -a`
- zu einem Branch wechseln:
  `% git checkout <Name>`

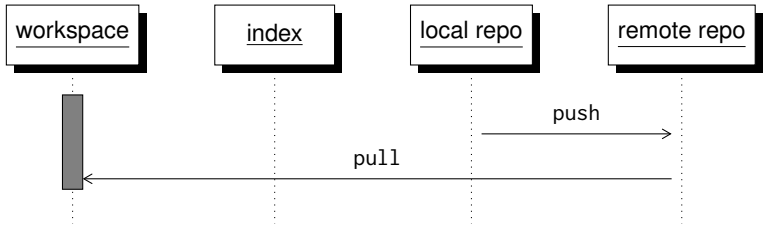
git-Kommandos: Lokale Quellcodeverwaltung V

- menügeführt das Repository befragen:
  `% tig`

Lesenswertes zu git

- http://gitready.com
- http://book.git-scm.com/
- http://gitcasts.com
- http://eagain.net/articles/git-for-computer-scientists/
- http://justinhileman.info/article/git-pretty/

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

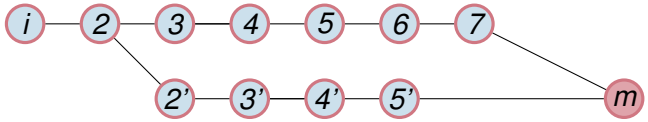
Commits sollen { atomar, gut dokumentiert, linear, unveränderlich } sein

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)
```

Vorübergehendes Sichern von Änderungen

- `git stash` und `git stash pop`

⇝ sichert Änderungen an der Working Copy auf Stapel

- `rebase` braucht saubere Working Copy

⇒ vorher `git stash`

Im Feature-Branch

```
% git stash

Saved working directory and index state WIP on master: 81c0895 cmake
HEAD is now at 81c0895 cmake, git ...

% ...
% git stash pop
```

Arbeitsablauf für größere Änderungen

- `git rebase`

⇝ wendet Commits auf einen anderen Branch an

⇝ schreibt Geschichte um

Im Feature-Branch

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

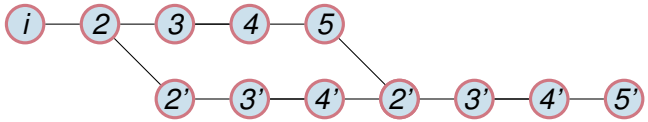
`pick` ⇝ übernimmt Commit

```
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

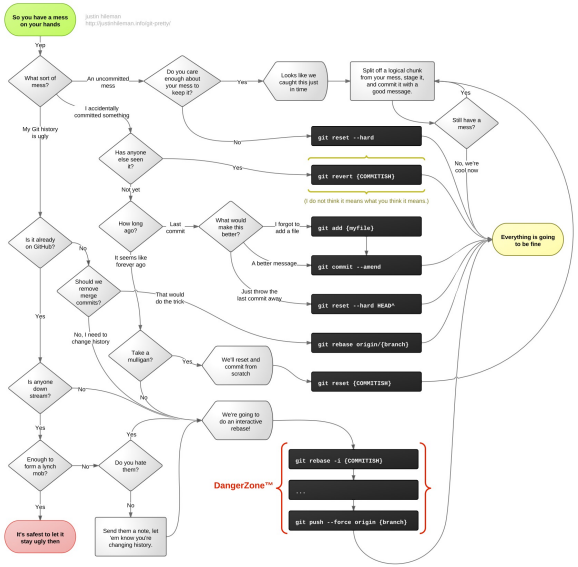
Wenn der Feature-Branch im Chaos versinkt?

⇝ 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

Geschichte neuschreiben



`git push --force`

Nützliche Aliase

.bashrc

```
function git_current_branch() {
  git symbolic-ref HEAD 2> /dev/null | sed -e 's/refs\/heads\///'
}

# git push ohne tracking
alias gpthis='git push origin HEAD:$(git_current_branch)'
# alle branches holen und dann rebase
alias gup='git fetch origin && git rebase -p origin/$(git_current_branch)'
```

⇝ https://gist.github.com/geelen/590895

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/
- http://blog.sensible.io/post/33223472163/git-to-force-push-or-not-to-force-push

Fragen?