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

23. April 2012

Überblick

Zuverlässig Software entwickeln

CMake – Ein Meta-Buildsystem

Testen

Versionsverwaltung mit git

Aufgabenstellung: Ampel, Teil 1

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

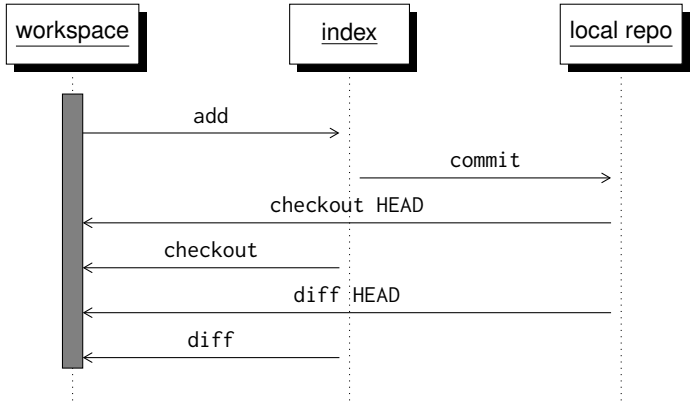
Idealerweise zusätzlich:

- 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`

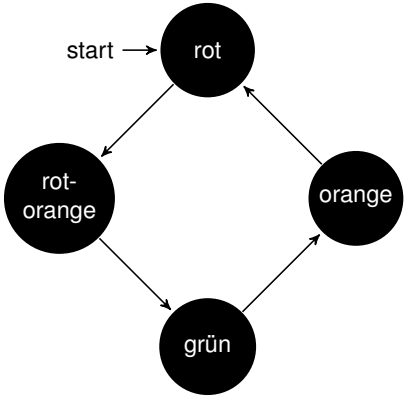
Lesenswertes zu `git`

- `http://gitready.com`
- `http://book.git-scm.com/`
- `http://gitcasts.com`
- `http://eagain.net/articles/git-for-computer-scientists/`

Inhaltsverzeichnis

Ampelautomat

- Ampel hat genau vier Zustände
- jeder Zustand genau einen Folgezustand



Aufgabenstellung

- Implementierung der Zustandsübergangsfunktion
  `tlight_set_next_phase(TLight *l)`
- Implementierung von Hilfsfunktionen
  `tlight_set_red(TLight *l), ...`
- Entwurf feingranularer Testfälle
  $\Rightarrow$ jede Funktion auf fehlerhaftes Verhalten prüfen
- Implementierung der Testfälle
- *Ziel:* mindestens vollständige Überdeckung

Fragen?