



we engineer

Dr. Dirk Leinenbach

Test Driven Development

Überblick

- **Testen von Software: Zielsetzung, Ansätze**
- **Unit Tests**
- **Test Driven Development (TDD)**
 - Idee - Zyklus: **Red**, **Green**, Refactor
- **Best Practices**
 - Bug Reports
 - Blackbox vs. Whitebox
 - Legacy Projekte
- **Live-Session: gemeinsame TDD-Entwicklungssession**

Ziele

- **Korrekte Software ausliefern**
 - Höhere Kundenzufriedenheit
 - Geringere Support-Aufwände
 - Entwickler sind stolz auf ihre Arbeit → Motivation!
- **Änderbarkeit und Wartbarkeit**
 - Anforderungen ändern sich (ständig)
 - Spätere Weiterentwicklung
 - Innere Qualität vs. technische Schuld
 - Unterstützung durch Softwareentwicklungsmethoden
- **Fehler früh finden**
 - Späte Fehlerbehebung wird exponentiell teurer
(Design, Implementierung, Testabteilung, nach Auslieferung)

Wie kann man Testen?

Manuell - Wahllos

Wahlloses Ausprobieren

- Was? Komplette Anwendung
- Wie? „mal schauen ob es geht...“
- Wer? Entwickler, Tester
 - Vorteile:
 - + Direkt loslegen - Keine Planung notwendig
 - Nachteile:
 - Nicht wiederholbar, keine Dokumentation des gewünschten Verhaltens
 - Manueller Aufwand → seltenes Testen
 - Keine Entdeckung von Regression-Fehlern
 - Seltene Tests „alten“ Codes („der funktioniert ja...“)

Wie kann man Testen?

Manuell - Strukturiert

Strukturiertes „Ausprobieren“ mit Testplänen

- **Was?** Komplette Anwendung
- **Wie?** Erstellen und Abarbeiten von Testplänen, Dokumentieren der Ergebnisse
- **Wer?** Entwickler, Tester
 - **Vorteile:**
 - + Ende-zu-Ende: Verhalten der kompletten Applikation in realer Umgebung
 - **Nachteile:**
 - Wiederholter manueller Aufwand
 - Seltenes Testen (nur vor Auslieferung)
 - Späte Fehlerentdeckung: hohe Kosten, langsame Feedbackschleife
 - Hohe Test-Aufwände
 - Hoher Aufwand für Test-Setup (Datenbanken, Systemumgebung, ...)

Wie kann man Testen?

Automatisiert

➤ Automatisierte Tests

- Einmal schreiben, häufig ausführen
- Ergebnis wird durch den Computer automatisch überprüft
- Code-nahe Tests werden durch die Entwickler selbst geschrieben!
- Ausführbare Spezifikation und Dokumentation!

➤ Wer testet? Alle!

- Ursprünglichen Entwickler, Team-Mitglieder
- Tester
- Continuous Integration Systeme → Ausführung auf verschiedenen Plattformen

➤ Tests auf mehreren Ebenen

- Akzeptanztests, GUI-Tests, Integrationstests, **Unit-Tests**
- Automatisierung: Voraussetzung für agile Entwicklungsmethoden

Der Rest der Welt

- **Externe Komponenten**
 - Datenbanken, Dateien, Backendsysteme, externe Prozesse
- **Echte Integration in Tests schwierig und aufwendig**
 - Separate Instanzen für jeden Entwickler / Tester
 - Komplexes Setup
 - Langsame Testausführung
 - Sauberes Zurückversetzen in Startzustand **nach** Tests notwendig (bzw. Initialisierung **vor** den Tests)
- **Alternative: Simulation, Mock-Objekte**
 - Externe Komponenten und Objekte simulieren (so weit wie nötig)
 - Deutlich vereinfachte Tests von Fehlersituationen (Netzwerkfehler, fehlende Berechtigungen, Festplatte voll, ...)
 - Schnellere Testausführung

Unit Tests

➤ Was wird getestet?

- Möglichst kleine und unabhängige Code-Teile
- Objekte, Funktionen
- Auch Fehlverhalten + Error-Handling!

➤ Eigenschaften

- Möglichst automatisiert
- Isolierte Tests einzelner Module
- Kurze Laufzeit, häufige Ausführung

➤ Grenzen

- Keine garantierte Korrektheit, aber stetige Verbesserung der Code-Qualität
- GUI bleibt ungetestet (aber auch hierfür gibt es automatisierte Lösungen)
- Gefahr von Blind Spots: Entwickler schreibt die Tests selbst
- Kein Ersatz für Akzeptanz- bzw. Anwender-Tests

Frameworks für Unit-Tests

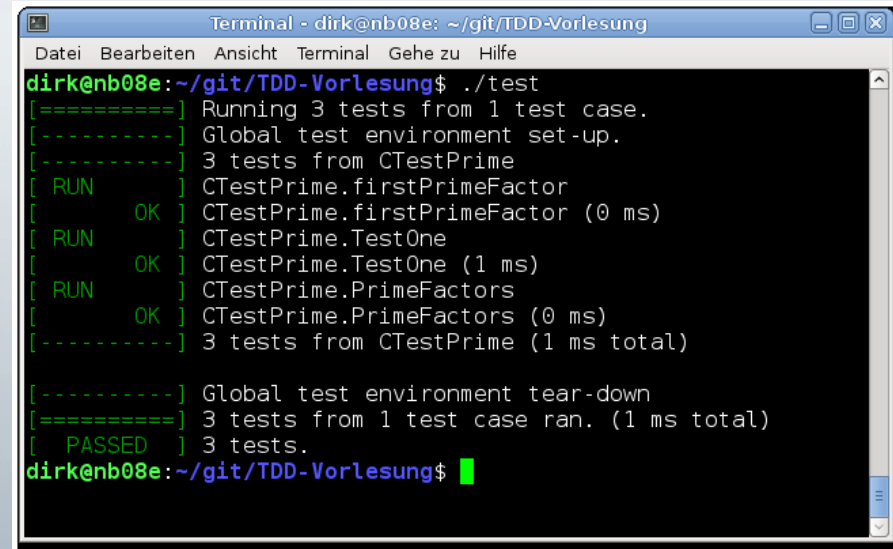
➤ Einfache Test-Definition

- Framework übernimmt Ausführung und Auswertung
- Detaillierte Analyse mit 3rd-Party Tools: zeitliche Entwicklung, Testabdeckung, ...

➤ Frameworks für viele Sprachen verfügbar

- Java: JUnit
- C/C++: cppunit, googletest, libtap
- Javascript: Test.More, JSUnit
- Python: PyUnit
- Ruby: Test::Unit
- PHP: PHPUnit, SimpleTest
- Perl: Test::More
- ...

➤ <http://opensourcetesting.org/>



```
Terminal - dirk@nb08e: ~/git/TDD-Vorlesung
Datei Bearbeiten Ansicht Terminal Gehe zu Hilfe
dirk@nb08e:~/git/TDD-Vorlesung$ ./test
[=====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 3 tests from CTestPrime
[ RUN      ] CTestPrime.firstPrimeFactor
[      OK   ] CTestPrime.firstPrimeFactor (0 ms)
[ RUN      ] CTestPrime.TestOne
[      OK   ] CTestPrime.TestOne (1 ms)
[ RUN      ] CTestPrime.PrimeFactors
[      OK   ] CTestPrime.PrimeFactors (0 ms)
[-----] 3 tests from CTestPrime (1 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test case ran. (1 ms total)
[ PASSED   ] 3 tests.
dirk@nb08e:~/git/TDD-Vorlesung$
```

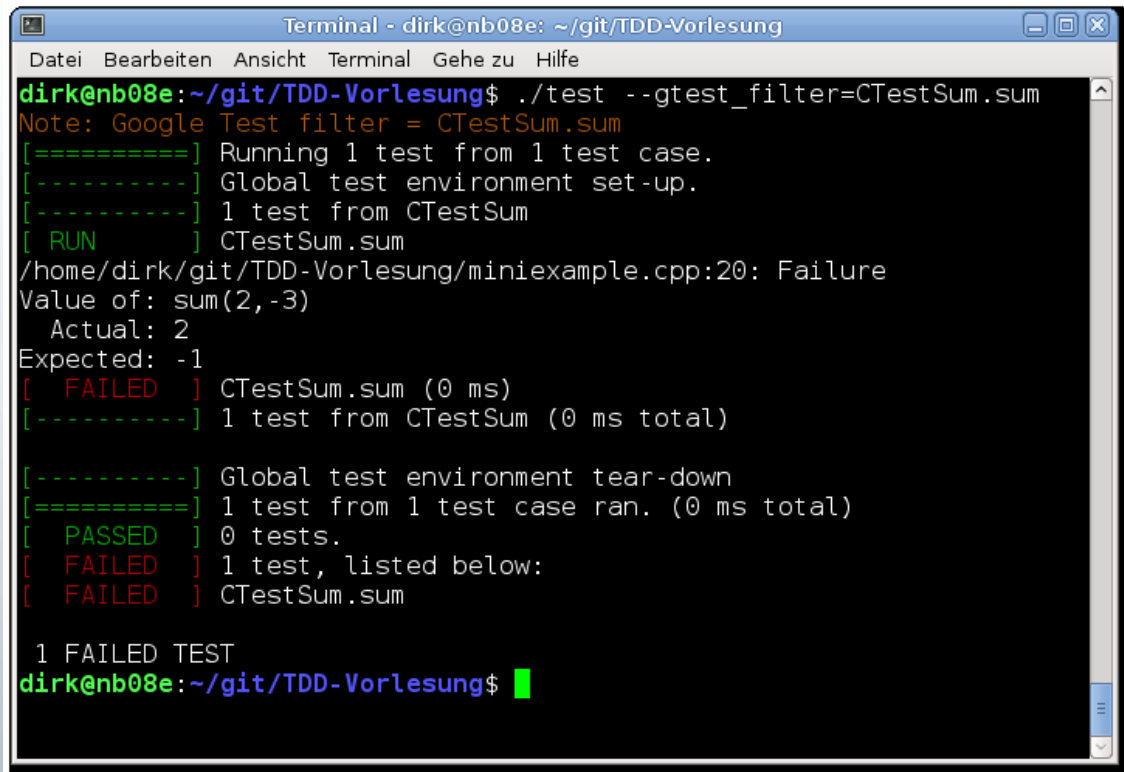
Beispiel googletest

```
#include <gtest/gtest.h>
```

```
class CTestSum : public ::testing::Test {  
};
```

```
int sum(int a, int b) {  
    int j=b;  
    int sum = a;  
    while (j>0) {  
        sum++;  
        j--;  
    }  
    return sum;  
}
```

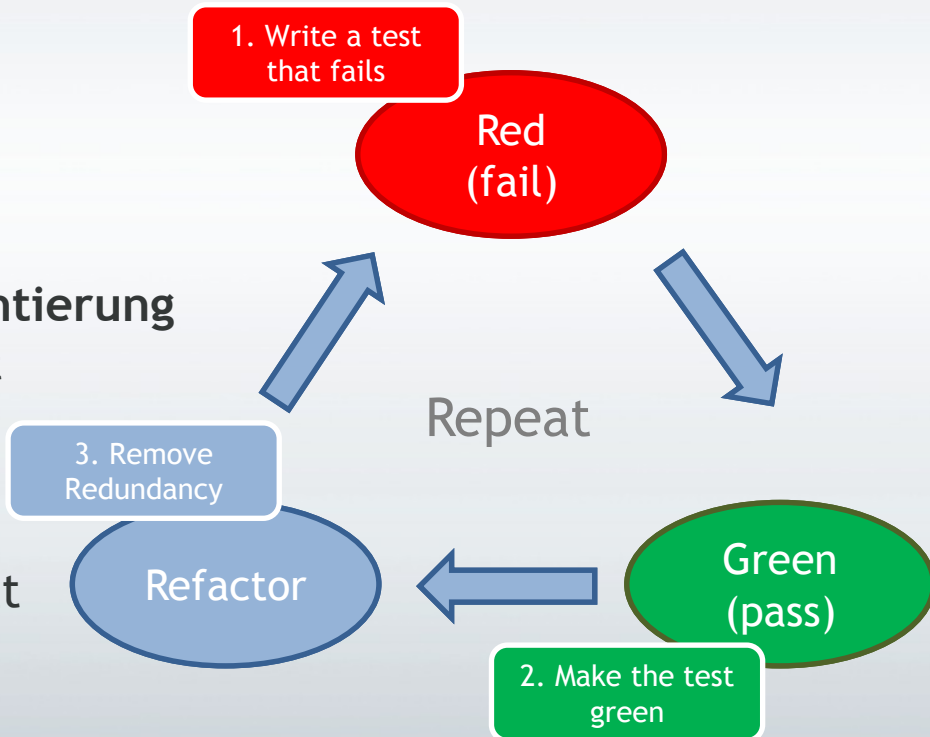
```
TEST_F(CTestSum, sum) {  
    ASSERT_EQ(5, sum(2,3));  
    ASSERT_EQ(-1, sum(2,-3));  
}
```



```
Terminal - dirk@nb08e: ~/git/TDD-Vorlesung  
Datei Bearbeiten Ansicht Terminal Gehe zu Hilfe  
dirk@nb08e:~/git/TDD-Vorlesung$ ./test --gtest_filter=CTestSum.sum  
Note: Google Test filter = CTestSum.sum  
[=====] Running 1 test from 1 test case.  
[-----] Global test environment set-up.  
[-----] 1 test from CTestSum  
[ RUN ] CTestSum.sum  
/home/dirk/git/TDD-Vorlesung/miniexample.cpp:20: Failure  
Value of: sum(2,-3)  
    Actual: 2  
Expected: -1  
[  FAILED  ] CTestSum.sum (0 ms)  
[-----] 1 test from CTestSum (0 ms total)  
  
[-----] Global test environment tear-down  
[=====] 1 test from 1 test case ran. (0 ms total)  
[  PASSED  ] 0 tests.  
[  FAILED  ] 1 test, listed below:  
[  FAILED  ] CTestSum.sum  
  
1 FAILED TEST  
dirk@nb08e:~/git/TDD-Vorlesung$
```

Test Driven Development TDD

- **Erst der Test, dann die Implementierung**
 - Keine Implementierung ohne Test
 - Nur so viel Implementierung wie nötig (für grünen Test)
- **Arbeiten in Zyklen: TDD-Mantra**
Red → **Green** → Refactor → Repeat
- **Refactorings zur Erhaltung der Code-Qualität**
 - In kleinsten Schritten
 - Gefahrlos, da durch automatisierte Tests abgedeckt
 - Evolutionäres Design statt Design bzw. Optimierung auf Vorrat
- **Kurze Feedbackschleife**



TDD-Zyklus

1. Red

- Feature / Test auswählen: was wollen wir als nächstes tun?
- Schreibe einen Test, der die Anforderung überprüft
- Prüfe, dass der Test fehlschlägt

2. Green

- Implementiere das Feature
- So schnell wie möglich - gerne auch (sehr) hässlich!
- Primäres Ziel: grüner Test!

3. Refactor

- Verbessere die Implementierung ohne dass der Test rot wird
- Schrittweise Design-Verfeinerung: Entfernung von Redundanz
- Redundanz oft auch zwischen Implementierung und Test
- Alle Tests als Sicherheitsnetz → Mut zur Änderung

TDD-Zyklus

- **Dauer typischerweise nur wenige Minuten!**
 - Ständige Anpassung der Schrittgröße
 - Schwieriges Terrain: Mini-Trippelschritte
 - Offensichtliche Implementierung: größere Schritte
 - Bei Problemen: Revert & Wechsel zu Mini-Schritten
- **Durch Sicherheitsnetz (Tests) Fokussierung auf aktuelle Aufgabe**
- **TODO-Liste**
 - Konzentration auf aktuellen Task
 - Liste noch fehlender Tests und Refactorings
 - Weniger Ablenkung: „...was ist denn mit Fall X?“

Regressionstests

➤ Wie geht man mit Bug-Reports um?

- Es gibt offensichtlich ungetestete Funktionalität
→ fehlender Test!
- Für jeden Bug ein neuer, zusätzlicher Test
- Stetige Verbesserung der Test-Abdeckung und Code-Qualität

➤ TDD-Zyklus

- Design: Neuer Bug-Report von Kunde oder Tester
- **Red**: erstelle möglichst einfachen Test, der den Bug reproduziert
- **Green**: fixe den Bug
- Refactor: Aufräumen des Bugfix

Blackbox vs. Whitebox

➤ Testen von private Code?

- **Pro**
 - + Schreiben von „nötigen“ Tests u.U. einfacher
 - + Testen ist besser als nicht-Testen
- **Contra**
 - Privater Code sollte extern nicht relevant sein, also auch nicht für den Tester
 - Notwendigkeit deutet auf Design-Probleme hin
 - Relevanter Zustand sollte extern (public) sichtbar sein

Testen komplexer Systeme

- **Abhängigkeiten: der zu testende Code nutzt externe Ressourcen**
 - Datenbanken, Prozesse / IPC, Dateien, Netzwerk, ...
- **Mock-Objekte**
 - Simulieren reale Objekte so weit wie nötig
 - Beispiele: Netzwerkkommunikation, Datenbankzugriffe
 - Weniger Aufwand für Setup der Testsysteme
 - Beschleunigte Test-Ausführung
 - In einfachen Fällen: manuelle Implementierung (z.B. via Vererbung / Interfaces)
- **Mock-Libraries für viele Sprachen verfügbar**
 - googlemock (C++), EasyMock (Java)
 - Vereinfacht das schreiben von komplexen Mock-Objekten
 - Überprüfung erwarteter Eingaben
 - Einfache Definition der Reaktion des Mock-Objekts

Legacy-Code Brownfield Projekte

➤ Typische Situation

- „Weiß jemand, was dieser Code macht?“
- „Wo ist denn die Entwickler-Doku dazu?“
- „Wer weiß welche Konsequenzen die Änderung haben könnte...“
- „Nach der letzten Änderung hatten wir wochenlang Probleme...“

➤ Iteratives Vorgehen

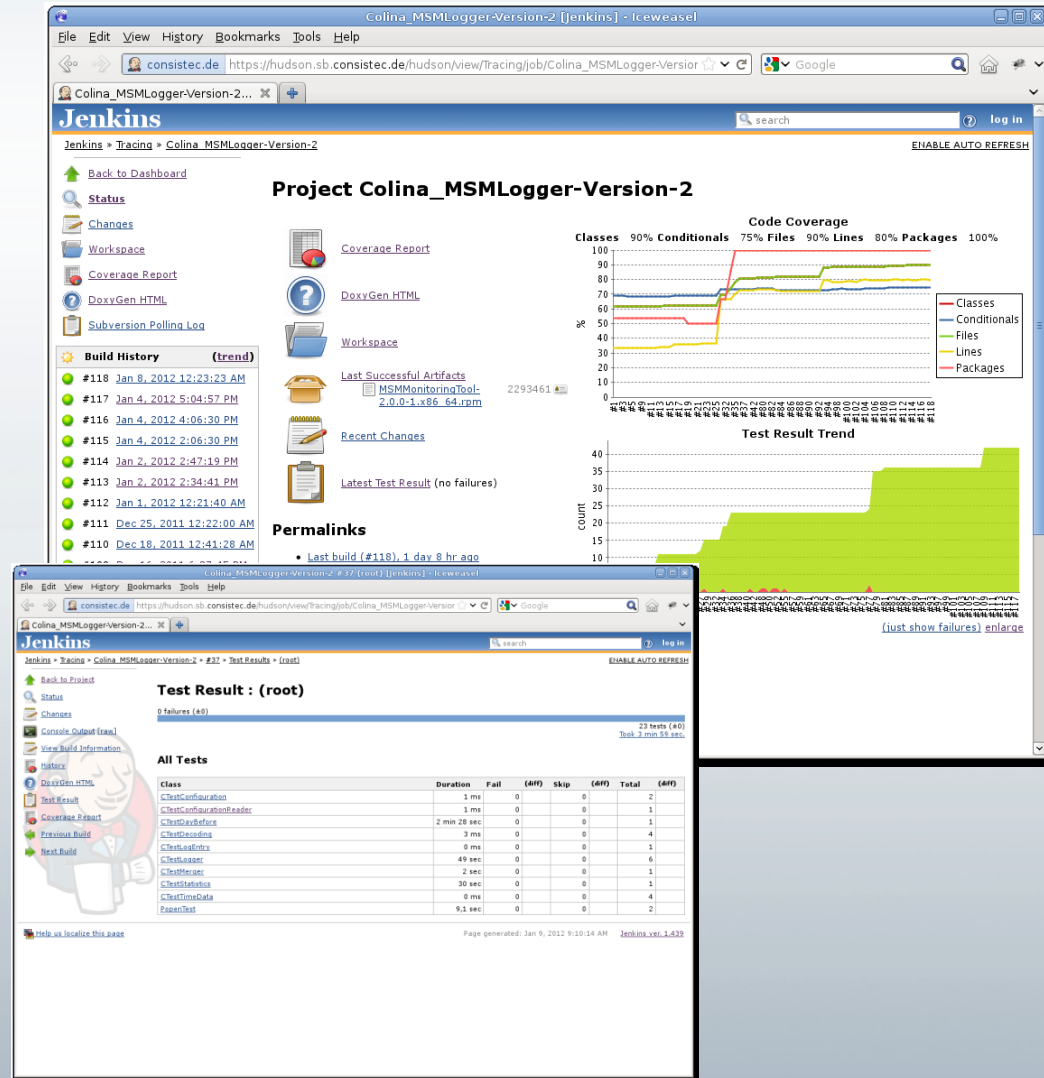
- **Red**: schreibe einen Test um dein Verständnis zu prüfen
- **Green**: verfeinere den Test bis er durchläuft
- Refactor: Wenn „genügend“ Tests / Verständnis für eine Komponente vorhanden sind
- Implementiere neue Funktionalität mit TDD

➤ Stetige Verbesserung der Situation mit jeder Änderung

➤ WICHTIG: vorsichtig vorgehen, Konzentration auf sich ändernde Teile

Team-Aspekte

- Schritt-für-Schritt Einführung
- Keine „roten“ Commits
- Continuous Integration
 - Build-Vorgang und Tests
 - Alle Zielplattformen
 - Unbekannte Abhängigkeiten
 - Unvollständige Commits



Live Session



Fazit

- **Aufwand / Nutzen?**
 - Etwa gleichviel Test- und Produktionscode
 - Nicht zwingend mehr Tests als bei Test-Last Ansätzen
 - Test-Qualität nicht zwingend besser als bei Test-Last Ansätzen
 - Per Definition 100% Testabdeckung
- **Wenig harte Fakten bzgl. Qualitäts- oder Produktivitätssteigerung ...**

Fazit

➤ ... aber dramatisch besseres subjektives Gefühl

- Weniger Stress bei guter Produktivität: „*rapid unhurriedness*“ (K. Beck)
- Weniger Angst bei Änderungen und Refactorings → höhere Code-Qualität
- Kein „Vergessen“ der Tests
- Kein Design auf Vorrat / Verdacht: „*just enough design to have the perfect architecture for the current system*“
- Keine garantierte Korrektheit, aber „Attractor“: „*code is more likely to change for the better over time instead of for the worse;*“ (K. Beck)
- Ausführbare Spezifikation und Entwickler-Doku
- Tendenz zu besser testbarem Code
 - ✓ Kleinere, unabhängigere Klassen
 - ✓ Lose Kopplung der Komponenten

Fragen

- **Kontakt:** `dirk.leinenbach@consistec.de`
- **Folien und Demo-Code auf Github verfügbar:**
 - <https://github.com/dirkl/TDD-Lecture>
 - <https://github.com/dirkl/TDD-Lecture/zipball/master>
- **Bücher**
 - Kent Beck, *Test Driven Development by Example*
 - Johannes Link, *Unit Tests mit Java. Der Test-First-Ansatz*
 - Roy Oshero, *The Art of Unit Testing: Deutsche Ausgabe*
 - Robert C. Martin, *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*