Automatisiertes Testen von Software in C++

(mit dem Test Framework Google Test)

06.05.2019

Florian Wolters (wolters.fl@gmail.com)

Übersicht

Fragen an das Publikum

1. Wer testet seine Software?

2. Wer verwendet ein Test Framework?

3. Wer weiß, was testbaren Code von schlecht / nicht testbaren Code unterscheidet?

4. Wer hält die Entwickler-Tests fortlaufend auf Stand?

Fokus des Vortrags

- Software-Entwicklungs-Aktivität: Software-Konstruktion
- Test-Kategorie: Entwickler-Tests
- Programmiersprache: C++11
- Programmierparadigma: Object-Oriented Programming (OOP)
- Test Framework: Google Test
- nur ein Einstieg, da unzählige weitere relevante Aspekte:
  - Zusammenhänge Clean Code, Build System, Continuous Integration (CI), …
  - Advanced Test Framework Features: Custom Assertions/Matcher
  - Advanced Test Doubles: Compile-Time, Link-Time, Run-Time

Inhalt (1/4)

- Warum Testen?
- Wie und Wann Testen?
- Software-Qualität
  - Merkmale
  - Probleme
  - Verbesserungs-Techniken

Inhalt (2/4)

- Entwickler-Tests
  - Terminologie
  - „Anforderungen“
  - positive Merkmale
  - Unit Tests
  - Praktiken
    - Plain Old Unit Testing (POUT)
    - Test-Driven Development (TDD)
    - Refactoring
  - SOLID > STUPID
  - Wozu xUnit Test Framework?
  - Wozu Mock Object Framework?

Inhalt (3/4)

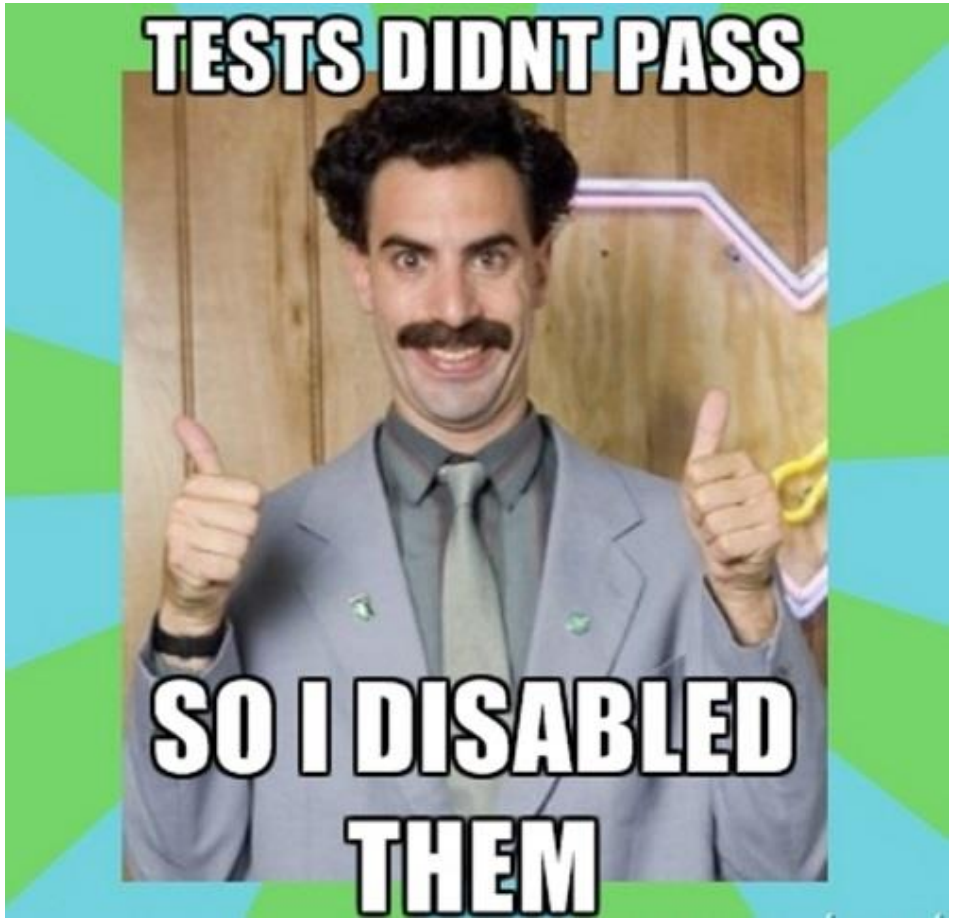
- Testen in C++
  - Werkzeuge
    - Test Frameworks
    - Mock Object Frameworks
    - Code Coverage Tools
  - Google Test
    - Warum?
    - Praxis-Teil
      - Assertion Methods
      - Matchers
      - Testcase Classes
      - Parameterized Tests
      - Test Doubles (Google Mock)

Inhalt (4/4)

- Fazit / way to go
- Literatur
  - Vorträge
  - Bücher
  - Webseiten

Warum Testen?

Warum Testen?



Warum Testen? | Gründe

- **primär**: Verbessern der **Wirtschaftlichkeit** des Unternehmens
  - Lieferung angemessener Qualität in angemessener Zeit
- **katastrophale Folgen** für Leben und Dinge durch Software-Bugs: Software-Qualität ist für bestimmte Systeme nicht verhandelbar
- Kostenreduktion: **teures** Auffinden/Beheben von Bugs in Production Code
- Kunden-Anforderung(en) mit unbestimmtem Rechtsbegriff **Stand der Technik** von Software (Prozess und Quelltext): Plain Old Unit Testing (POUT) als defacto Standard in der Software-Entwicklung
- Tests sollten das Verbessern der Software-Qualität **ermöglichen**
  - Existenz von Tests erlauben erst Refactoring
  - „Angst“ des Entwicklers vor Änderungen wird vermindert

Warum Testen? | Grenzen

- Testen ist „nur“ **ein wichtiger** Teil jedes Software-Qualität-Prozesses
- kollaborative Entwicklungs-Praktiken finden i.d.R. mehr Fehler als Testen
- Testen fällt Entwicklern grundsätzlich schwer:
  - Test-Ziel „Finden von Fehlern“ arbeitet gegen andere SW-Entwicklungs-Praktiken, z.B. „Vermeiden von Fehlern“
  - Testen kann niemals die Nicht-Existenz von Fehlern nachweisen.
    Beispiel: Ergebnis „0 Fehler" kann sowohl als ineffektive/unvollständige Tests, als auch als „perfekte“ Software ausgelegt werden
  - Testen selbst verbessert **nicht** die Software-Qualität, Test-Ergebnisse sind ein **Indikator** für die SW-Qualität
- Testen ist kostspielig. Der Verzicht auf Testen ist:
  - kostspieliger
  - rechtlich schwer vertretbar

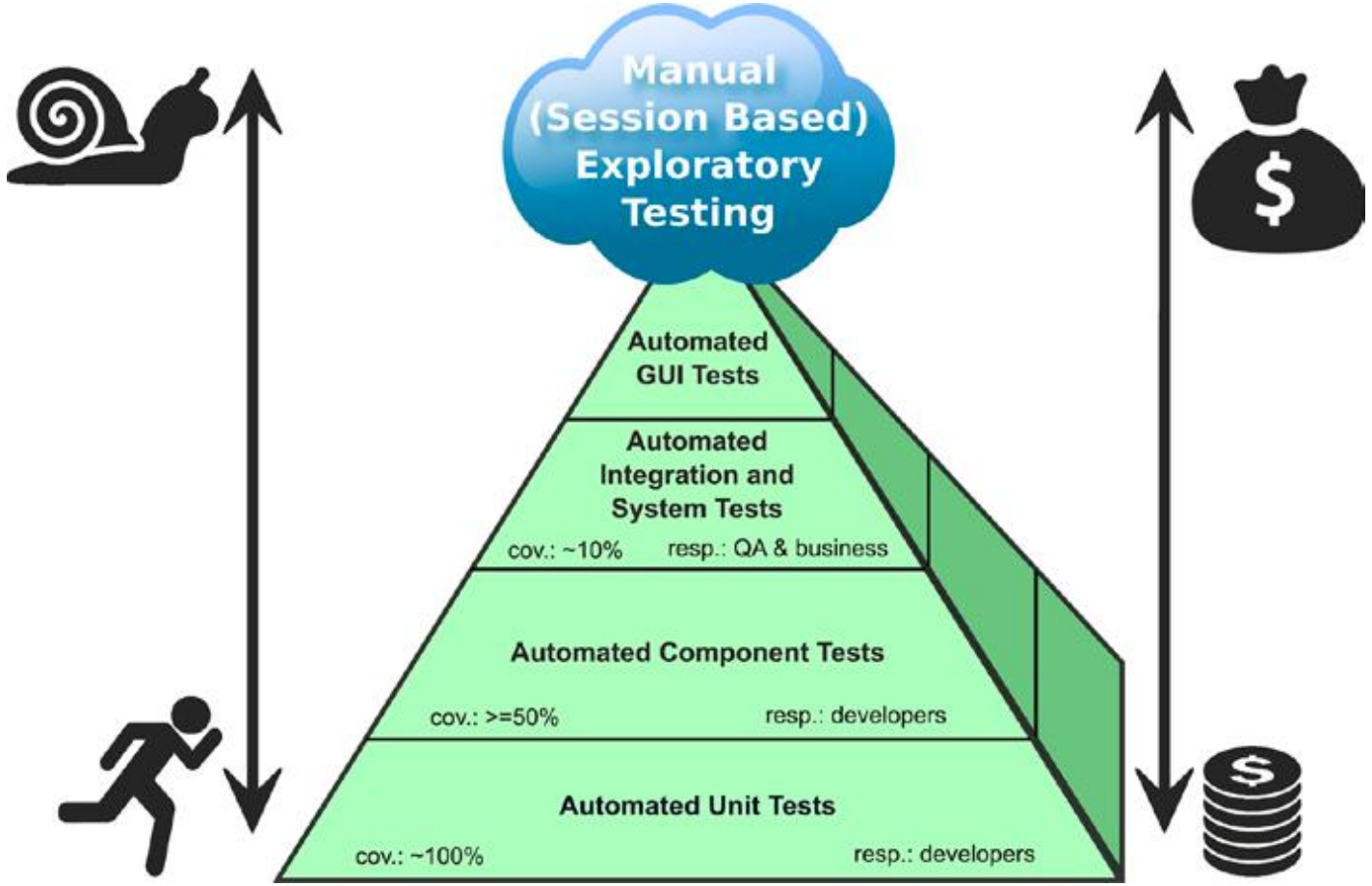
Wie und wann testen?

Wie und wann testen?

Wie und wann testen? | Test-Level (ISTQB)

1. **Unit** Test (Komponententest): Testen einer Software-Komponente, die isoliert getestet werden kann
2. **Integration** Test (Integrationstest): Testen der Schnittstellen und des Zusammenspiels zwischen integrierten Software-Komponenten
3. **System** Test (Systemtest): Testen eines integrierten Systems
4. **Acceptance** Test (Abnahmetest): Testen hinsichtlich der Benutzeranforderungen

Wie und wann testen? | Test Automation Pyramid (Roth 2017, S. 11)



Wie und wann testen? | Test-Level (ISTQB)

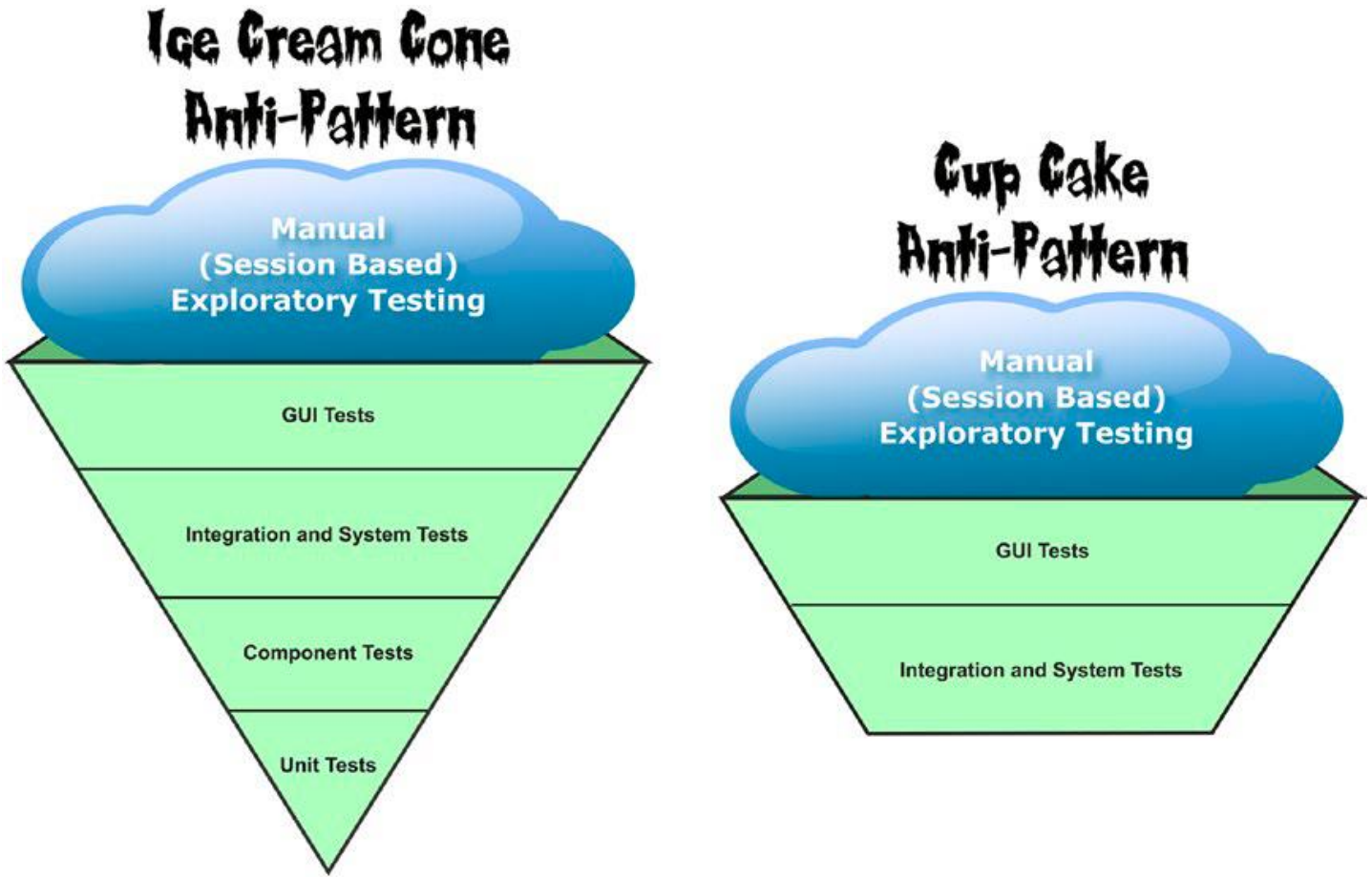
System- & Akzeptanz-Tests sind häufig

- komplex,
- erfordern umfangreiche Organisation,
- können nicht einfach automatisiert werden
  (Beispiel: UI-Tests die hart zu schreiben, fragil und langsam sind.)

und werden daher häufig **manuell** ausgeführt

System- & Akzeptanz-Tests sind zu **zeitintensiv** und zu **teuer** für den täglichen Einsatz während der SW-Entwicklung

Wie und wann testen? | **Degraded** Test Automation Pyramids (Roth 2017, S. 12)



Software-Qualität

Software-Qualität | Merkmale

- externe Qualität
  - Qualität aus der Sicht des Anwenders/Kunden
  - Indikatoren können von außen gemessen werden
- interne Qualität
  - Qualität aus der Sicht des Entwicklers
  - Indikatoren können **nur** von innen gemessen werden
- FURPS <-> ISO/IEC 9126
  - **F**unctionality <-> Funktionalität
  - **U**sability <-> Benutzbarkeit
  - **R**eliability <-> Zuverlässigkeit
  - **P**erformance <-> Effizienz
  - **S**upportability <-> Änderbarkeit

Software-Qualität | Probleme

- in manchen Organisationen gilt:
  - Software-Qualität (vor allem interne) wird als sekundäres Ziel wahrgenommen
  - Quick & Dirty Programmierung ist die Regel und nicht die Ausnahme
  - Entwickler die ihre Aufgabe schnell „abschließen“ werden mehr belohnt als Entwickler die hohen Wert auf (auch interne) Software-Qualität legen

- eine Organisation **muss** Entwicklern aufzeigen, dass Qualität Priorität hat (siehe Folie „Warum Testen?“)
- Entwickler muss selbst ein Bewusstsein für Qualität erwerben

Software-Qualität | Verbesserungs-Techniken

- Zielvorgaben bzgl. Software-Qualität
- explizite Aktivitäten bzgl. Qualitäts-Sicherung
- Test-Strategie
- Richtlinien bzgl. Software-Engineering
- informelle technische Reviews
- formelle technische Reviews
- externe Audits

Entwickler-Tests

Entwickler-Tests | Terminologie | xUnit

Benennung	Bedeutung
System Under Test (SUT)	Das zu testende Software-Artifakt: • Class Under Test (CUT) • Object Under Test (OUT) • Method(s) Under Test (MUT) • Application Under Test (AUT)
Test Method	`TEST(TheTestCase, TheTestMethod) {}`
Assertion Method	`ASSERT_EQ(expected, actual);`
Testcase Class	`class TheTestCaseClass : public ::testing::Test;`
Test Runner	CLI: `the_test_executable.exe` GUI: IDE Add-In
Fixture Setup	`TheTestCaseClass();` `auto SetUp() -> void;`
Fixture Teardown	`auto TearDown() -> void;` `~TheTestCaseClass();`

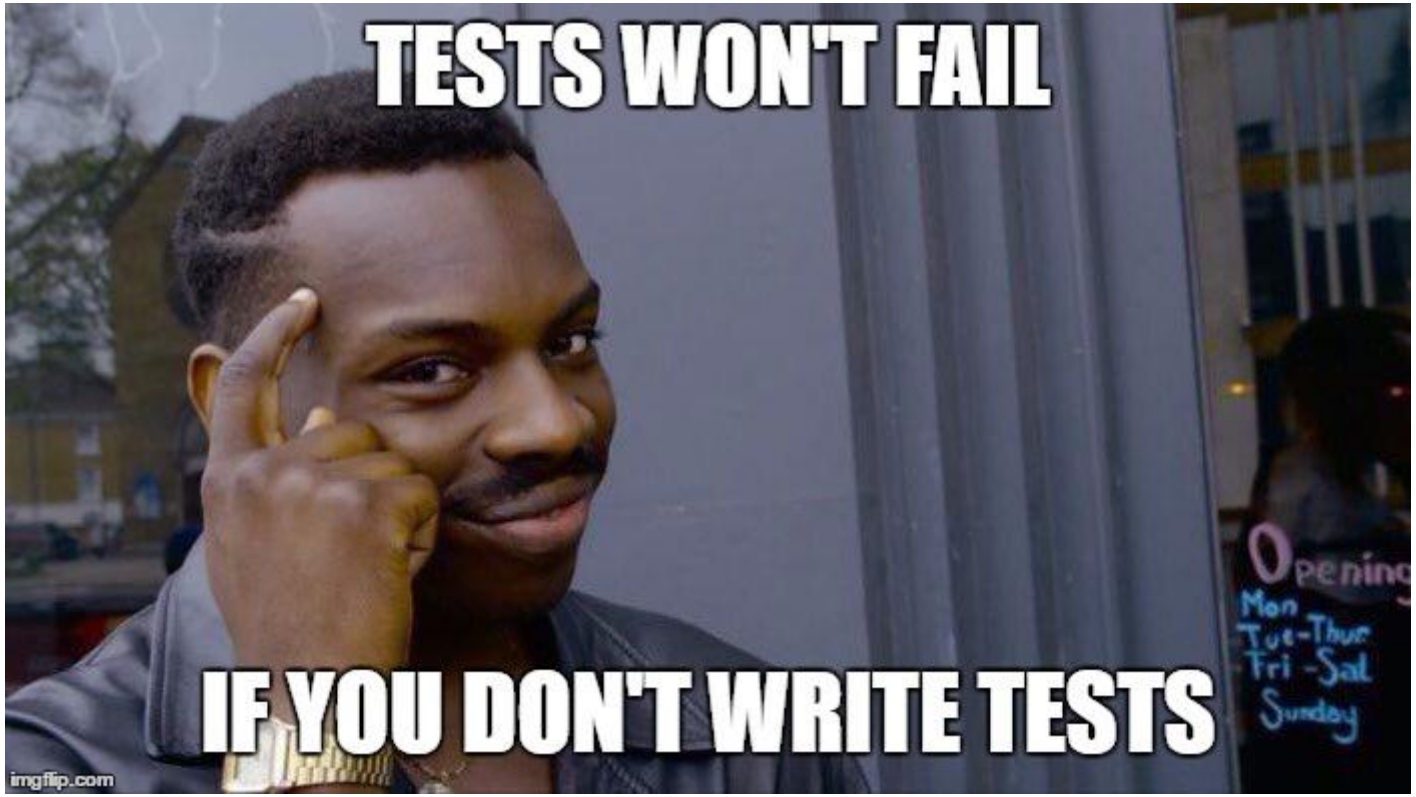
Entwickler-Tests | Terminologie | Test Double

Benennung	Bedeutung
Test Double	Oberbenennung. Ersetzen einer Komponente, von der das SUT abhängt durch ein test-spezifisches Äquivalent
Dummy	Wird im Code weitergereicht, aber nicht verwendet
Fake	Benutzerdefinierte Implementierung, die näher an der realen Implementierung ist, als ein Stub (z.B. Verwendung interner Zustände)
Stub	Liefert gleiche Ausgabe, unabhängig von Eingabe
Mock	Liefert Ausgabe, abhängig von Konfiguration (bestimmte Eingabe ergibt definierte Ausgabe)
Spy	Protokolliert Aufrufe und Werte

Entwickler-Tests | „Anforderungen“

- Ein neuer Test sollte „leicht“ zu erstellen sein.
  - Verwenden eines Test Frameworks
  - Convention-over-Configuration (CoC) Integration des Test Frameworks in die Entwicklungs-Werkzeuge
- Sämtliche Abhängigkeiten des SUT sollten leicht zu ersetzen sein.
  - Verwenden eines Mock Object Frameworks
  - CoC Integration des Mock Object Frameworks in die Entwicklungs-Werkzeuge
- Test-Code sollte wie Production-Code behandelt werden.
  - automatisierte statische Code Analyse
  - manuelles/(teil)automatisiertes Code Review

Entwickler-Tests | positive Merkmale



Entwickler-Tests | positive Merkmale (1/3)

1. **Schreiben von Tests mit einem Testing Framework.**
2. Korrektheit: „Verhält sich die API korrekt?“
3. Lesbarkeit:
   - „Für Tests gibt es keine Tests.“
   - „Anforderungen an Software-Qualität wie für Produktions-Code.“
   - „Die Korrektheit eines Tests sollte durch ein Code Review feststellbar sein.“
   - kein Boilerplate-Code, z.B. dupliziertes oder nutzloses Setup
   - Kontext für den Leser, z.B. durch angemessene Abstraktion
   - keine Verwendung komplexer Test Frameworks Features, falls unnötig

Entwickler-Tests | positive Merkmale (2/3)

4. Vollständigkeit
   - Zielvorstellung: 100% Bedingungsüberdeckung (Condition Coverage) (C_3)
   - Berücksichtige Randfällen in Positiv- **und** Negativ-Tests
   - Schreibe Tests **nur** für eigene API, **nicht** für Third-Party-APIs

5. Demonstrierbarkeit
   - Tests sollen demonstrieren wie die API funktioniert
   - keine Hacks (`friend`, Test-Only-Funktionen in API, usw.)
   - „lebende“ Dokumentation der API

Entwickler-Tests | positive Merkmale (3/3)

6. Elastizität: „Der Test sollte nur fehlschlagen, falls sich das zu testende Verhalten des SUT verändert hat.“
   - keine flaky Tests, d.h. Tests die beim wiederholten Ausführen unterschiedliche Ergebnisse liefern
   - keine brittle Tests, d.h. Tests die aufgrund von Änderungen unzusammenhängend mit dem SUT scheitern
   - Reihenfolge der Ausführung der Tests sollte egal sein
   - Unit Tests sollten hermetisch sein, d.h. kein Input/Output
   - keine zu enge Kopplung durch Test Doubles

Entwickler-Tests | Unit Tests| F.I.R.S.T.

- F.I.R.S.T. beschreibt Qualitäts-Merkmale eines Unit Tests:
  - **F**ast: schnell in der Ausführung (Compile & Runtime)
  - **I**solated: unabängig von anderen Tests, z.B. der Ausführungs-Reihenfolge
  - **R**epeatable: beliebig oft wiederholbar, ohne Eingriff von außen
  - **S**elf-Validating: der Test bestimmt selbst, ob er erfolgreich war, ohne dass eine manuelle Bewertung durch einen Menschen erforderlich ist
  - **T**imely: Test wird kurz vor / zeitgleich / kurz nach dem Schreiben des Production Code erstellt

- Isolated und Repeatable erfordern den Verzicht auf mutable global state zwischen Tests

Entwickler-Tests | Unit Tests| Aufbau (1/2)

- Modelle zur Beschreibung des Aufbaus eines Unit Tests:
  - 3A (Arrange-Act-Assert)
    - **A**rrange: Aufbau der Test Fixture (Erzeugung von SUT und Test Doubles)
    - **A**ct: Interaktion mit dem SUT, um das zu verifizierende Ergebnis zu erzeugen
    - **A**ssert: Überprüfung ob das erwartete Ergebniss durch den Act erzielt wurde
  - Four-Phase Test:
    - Setup
    - Exercise
    - Verify
    - Teardown

Entwickler-Tests | Unit Tests| Aufbau (2/2)

```
TEST(StringTest,
     EmptyOnDefaultConstructedInstanceReturnsTrue) {
  // Arrange / Setup
  auto const sut = std::string{};
  // Act / Exercise
  auto const result = sut.empty();
  // Assert / Verify
  ASSERT_TRUE(result);
  // Teardown
}
```

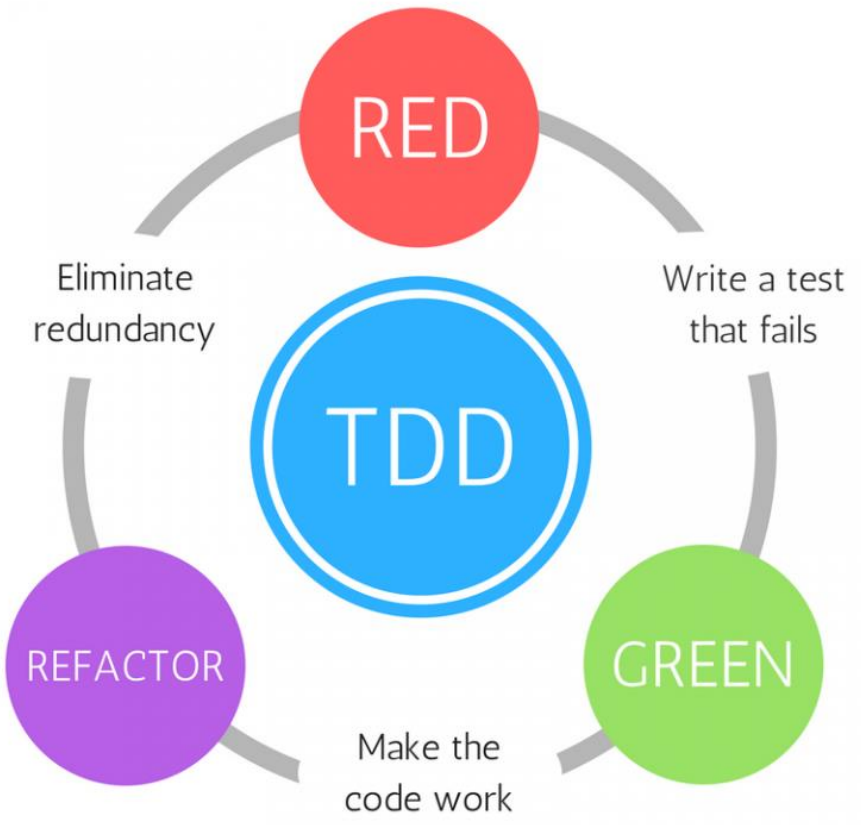
Entwickler-Tests | Praktiken | POUT

- Plain Old Unit Testing (POUT)
  - a.k.a. „test after“
  - Schreiben von Tests nach Schreiben des Production-Codes

- Probleme:
  - Production-Code ist möglicherweise untestbar
  - Test wird evtl. niemals geschrieben
  - Schreiben von unnötigem Production-Code, d.h. Verletzung von You Ain‘t Gonna Need It (YAGNI)
  - Gefahr zu geringer Fokussierung auf der externen API
  - (zu) geringe Code Coverage

Entwickler-Tests | Praktiken | TDD

- Test-Driven Development (TDD)
  - „test first“ software development practice
  - mantra: red-green-refactor
  - laws:
    1. “You may not write production code until you have written a failing unit test.”
    2. “You may not write more of a unit test than is sufficient to fail, and not compiling is failing.”
    3. “You may not write more production code than is sufficient to pass the currently failing test.”



Entwickler-Tests | Praktiken | Refactoring

- Zitat: “refactoring” without tests isn’t refactoring, **it is just moving shit around**
  (Corey Haines (@coreyhaines), 2013-12-12, on Twitter)

- Entwickler meinen oftmals **Reorganisation** des Codes, wenn Refactoring genannt wird

Bild © PremiumVector / Shutterstock

Entwickler-Tests |SOLID statt STUPID

- Befolgen von **SOLID**-Prinzipien **erleichtert** Entwickler-Tests bzw. **ermöglicht** diese erst
- Befolgen von **STUPID**-Prinzipien **erschwert** Entwickler-Tests bzw. macht diese **unmöglich**

* größter Nutzen für Entwickler-Tests

SOLID	STUPID
Single Responsibility Principle (SRP)*	**S**ingleton
Open-Closed Principle (OCP)	**T**ight Coupling
Liskov Substitution Principle (LSP)	**U**ntestability
	Premature Optimization
Interface Segregation Principle (ISP)	**I**ndescriptive Naming
Dependency Inversion Principle (DIP)*	**D**uplication

Entwickler-Tests | SOLID (1/3)

- SRP: „A class should have only one reason to change.“
- Befolgen von SRP führt zu vielen Klassen
- viele Klassen können im Zusammenhang mit DIP zu „Wiring Mess“ führen

Entwickler-Tests | SOLID (2/3)

- OCP: „Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.“
- LSP: „Subtypes must be substitutable for their base types.“
- ISP: „Clients should not be force to depend on methods that they do not use.

Entwickler-Tests | SOLID (3/3)

- DIP:
  - „High-level modules should not depend upon low-level modules. Both should depend on abstractions.“
  - „Abstractions should not depend on details. Details should depend on abstractions.“

- “Dependency Injection is a 25-dollar term for a 5-cent concept.” (James Shore, 2016-03-22)

```
class DependencyInterface {
 public:
  virtual ~DependencyInterface() = default;
};
class NonOwningConstructorInjection {
 public:
  explicit NonOwningConstructorInjection(DependencyInterface&);
};
class OwningConstructorInjection {
 public:
  explicit OwningConstructorInjection(std::unique_ptr<DependencyInterface>);
};
```

Entwickler-Tests | STUPID (1/2)

- Singleton: Creational (Anti-)Pattern führt zu Tight Coupling, falls falsch verwendet
- Tight Coupling: (Zu) eng gekoppelte Software ist schwierig wiederzuverwenden und hart zu testen
- Untestability: Tight Coupling auf nicht durch Test Doubles zu ersetzende Software führt zu untestbarer Software

Entwickler-Tests | STUPID (2/2)

- Premature Optimization: führt (auf Mikroebene) zu erhöhter Komplexität
  - “Premature Optimization is the root of all evil […].” (Knuth 1974, S. 671)
  - Keep it simple and stupid (KISS)
- Indescriptive Naming:
  - “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” (Fowler 1999, S. 15)
  - zweckbeschreibende Namen, aussprechbare Namen, Kodierungen vermeiden, usw.
- Duplication: führt zu erhöhter Komplexität und niedrigerer Wartbarkeit
  - Don’t Repeat Yourself (DRY)
  - Keep it simple and stupid (KISS)

Wozu xUnit Test Framework? (1/2)

- xUnit-Paradigma ist weit verbreitet und verstanden

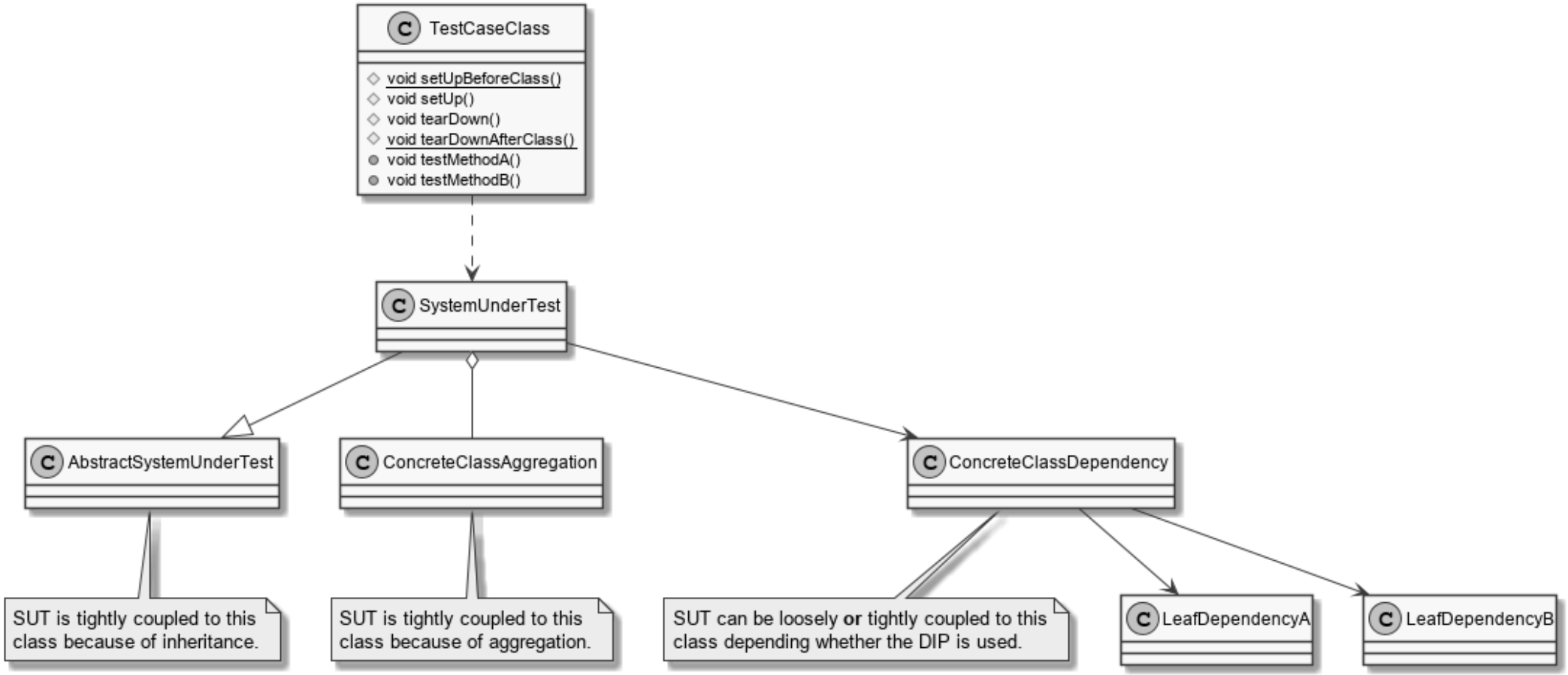
- Automatische Hinzufügen von Tests (Test Discovery/Registration)
  - keine manuelle Registrierung durch den Benutzer
  - C++: `main()`-Funktion wird durch das Framework bereitgestellt

- Test Runner, mindestens CLI
  - zufällige Ausführung von Tests
  - unabhängige Ausführung von Tests, d.h. scheiternde Assertion beendet Test Programm nicht
  - Möglichkeit Untermenge von Tests auszuführen, d.h. Filtern von Tests

Wozu xUnit Test Framework? (2/2)

- Assertions
  - nicht-triviale Assertions, z.B. Approximation, Exceptions
  - nützliche und anpassbare Ausgabe (im Fehlerfall)
  - Erstellen benutzerdefinierter Assertions
- Test Fixtures/Context, d.h. Definition von Vorbedingungen für Tests
- Werkzeug-Integration
  - Integration Development Environment (IDE), z.B. MSVS Add-In
  - Continuous Integration (CI), i.d.R. über JUnit-XML

Wozu Mock Object Framework? (1/2)



Wozu Mock Object Framework? (2/2)

- Erstellen von Mocks mit Framework einfacher als manuell
- Mock-Implementierung ist Teil des Tests, dies macht das Verhalten des Mocks i.d.R. einfacher zu verstehen
- weniger Boilerplate-Code
- nützliche Ausgabe durch Framework bei unerwarteten Aufrufen

Testen in C++ |Werkzeuge | Test Frameworks (Stand: 01.05.2019)

populäre Open-Source Bibliotheken:

- Google Test: BSD-3-Clause, 2019-04-29, 214 Beitragende
- Catch2: BSL-1.0, 2019-04-27, 175 Beitragende
- Doctest: MIT, 2019-03-24, 26 Beitragende
- Boost.Test: BSL-1.0, 2019-03-28, 70 Beitragende
- CppUTest: BSD-3-Clause, 2019-04-28, 81 Beitragende
- CppUnit: LGPL-2.1, 2017-04-13, 2+ Beitragende

und viele Weitere kommerzielle und nicht-kommerzielle: Wikipedia führt 77 Unit Test-Werkzeuge für C++ auf

Testen in C++ | Werkzeuge | Mock Object Frameworks (Stand: 01.05.2019)

- Google Mock: BSD-3-Clause, 2019-04-29, 214 Beitragende
  Hinweis: ist Teil von Google Test
- Trompeloeil: BSL-1.0, 2019-04-02, 9 Beitragende
- HippoMocks: LGPL-2.1, 2019-03-11, 20 Beitragende
- FakeIt: MIT, 24 2018-11-09, 24 Beitragende
- Mockitopp: MIT, 2017-01-02, 3 Beitragende

Testen in C++ | Werkzeuge | Code Coverage

- Linux
  - clang: llvm-cov / gcovr
  - gnu: gcov / gcovr
- Windows
  - clang: llvm-cov / gcovr
  - msvc: OpenCppCoverage
- kommerziell:
  - Verifysoft Testwell CTC++ (gnu-linux, msvc-windows)
  - BullseyeCoverage (clang-linux, clang-windows, gnu-linux, msvc-windows)
  - Microsoft Visual Studio Ultimate (msvc-windows)

Testen in C++ | Google Test | Warum?

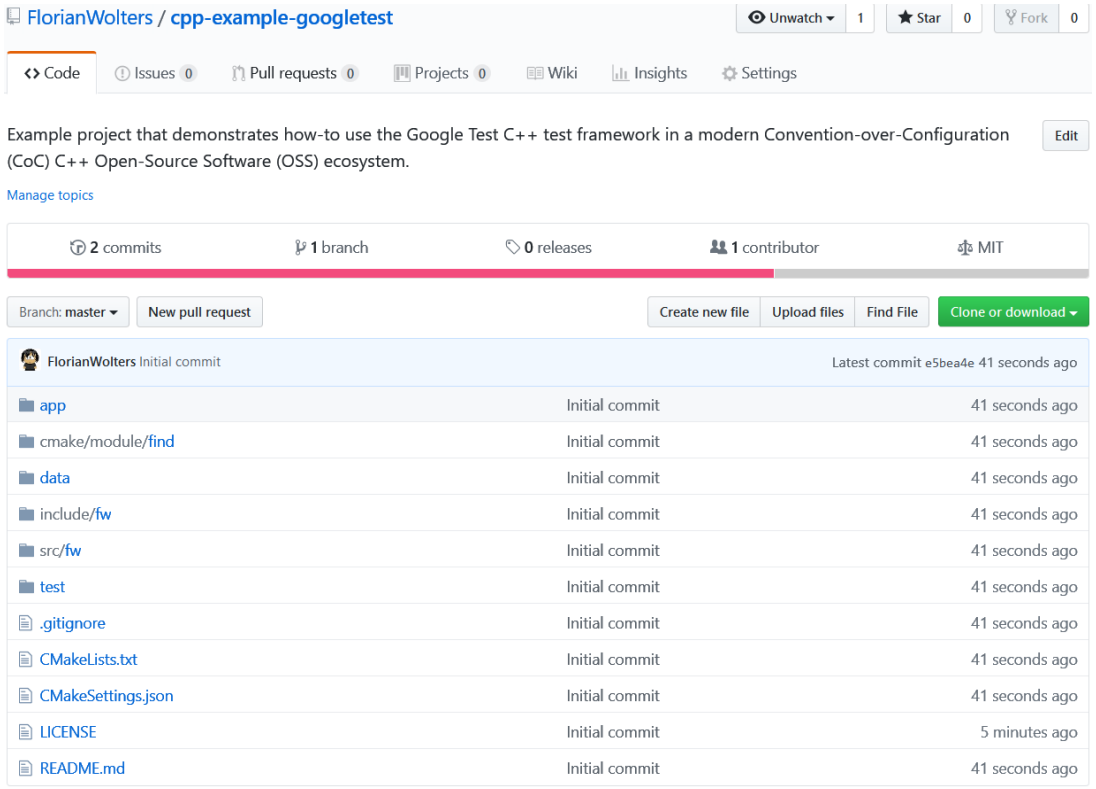
- Vorteile
  - weit verbreitet (populärste C++ Testing Framework 2017)
  - stabile API
  - Verwenden von Test Doubles (Google Mock)
  - mehr Funktionen als andere Testing Frameworks
  - bessere Dokumentation als andere Testing Frameworks
- Nachteile
  - Verwendung von Boilerplate-Macros
  - Software Build Integration, da nicht leichtgewichtige Header-Only-Bibliothek

Testen in C++ | Google Test | Praxis-Teil (1/4)

- IDE: Visual Studio Community 2019 v16.0.3
  - Test Adapter for Google Test v0.10.1.8
  - Clang Power Tools v4.10.3
- Build System Generator: CMake v3.14.3
- Build System: Ninja v1.9.0
- C++ Package Manager: vcpkg 2018.11.23
- Test Framework: Google Test 2019-01-04-2
- Mock Object Framework: Google Mock 2019-01-04-2
- Code Coverage Tool:
  - OpenCppCoverage v0.9.7.0
  - ReportGenerator v4.1.5
- C++ Libraries
  - docopt 2018-11-01
  - fmt 5.3.0-1

Testen in C++ | Google Test | Praxis-Teil (2/4)

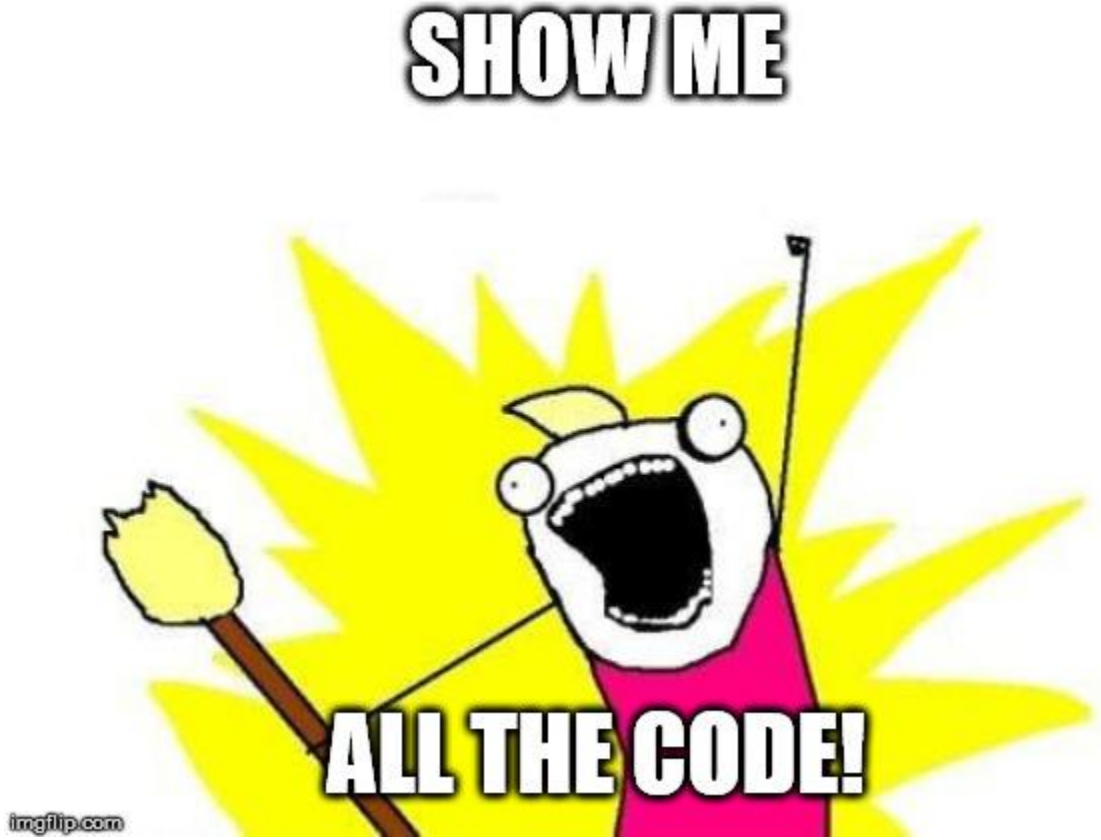
- Quelltext als Open-Source Software (OSS) auf GitHub (MIT License)
- wird evtl. (bei Motivation) zukünftig erweitert, u.a.:
  - Cloud Continuous Integration (CI):
    - Travis CI
    - AppVeyor
  - Open-Source Static Code Analysis Tools
    - Cppcheck
    - clang-tidy
  - Doxygen
  - …



Testen in C++ | Google Test | Praxis-Teil (3/4)

- Batch-Build-Skripte: https://gist.github.com/FlorianWolters/645b8e1b25b5f1e478333ac7de7b5b9b
  - v1.8.0 with MSVS 2013 x86 v120
  - master with MSVS 2017 x86_64 v141
  - master with MSVS 2019 x86_64 v142
- anpassbar bzgl. Compiler, Architektur und Platform Toolset

Testen in C++ | Google Test | Praxis-Teil (4/4)



Fazit / way to go

Fazit / way to go (1/4)

- Herausforderungen
  - Wie überzeugt man Unternehmen davon, dass Entwickler-Tests wichtig sind?
  - Wie bringt man Entwickler dazu
    - Tests zu schreiben?
    - besser zu testende Software zu schreiben?
  - Entwickler-Tests sind nur ein Baustein und sollten kombiniert werden mit Coding Standards, Coding Guidelines, Best Practices, Code Reviews, Continuous Integration (CI), usw.

Vorschreiben von Entwickler-Tests ohne klare Vorgaben und Prozesse kann schädlich sein!

Fazit / way to go (2/4)

1. Anfangen Entwickler-Test zu schreiben:
   1. **alle** mit dem **gleichen** xUnit Test Framework / Mock Object Framework
      - Probleme: fehlende Integration (Software Build, Package Management)
   2. mit einem **Code Coverage** Werkzeug
      - Probleme: s.o.
   3. Test-Code mit **gleicher** Qualität wie Produktions-Code
      - Probleme:
        - fehlende verpflichtende Kodier-Regeln und Best Practices
        - unpassende Werkzeuge zur statischen Code Analyse
        - unzureichende Integration der Werkzeuge zur statischen Code Analyse
   4. **TFD/TDD** > Test während Implementierung > Test nach Implementierung

Bauen/Testen (statisch/dynamisch) von Software sollte zero-conf sein!

Fazit / way to go (3/4)

2. Anfangen Continuous Integration (CI) zu nutzen:
   - nach push in VCS-branch:
     - Software wird für die Build Configuration `Release` gebaut
     - alle statischen Tests werden ausgeführt
     - alle dynamische Tests (mit aktivierter Code Coverage) werden ausgeführt
     - die API-Referenzdokumentation wird generiert
     - Artefakte zur Distribution (ZIP-Dateien, Installer) werden generiert
   - CI-Dashboard ist für **alle** sichtbar
     - Manager (Projekt, Qualitätssicherung) erhalten **direktes unverfälschtes** Feedback

Entwickler-Tests ohne CI werden keinen/wenig Nutzen haben!

Fazit / way to go (4/4)

3. Änderungen in Software-Entwicklungs-Prozess:
   - Entwickler-Tests als Pflicht (Kontrolle über Code Coverage in CI)
   - kein direktes Arbeiten im `master`-Branch, d.h. `master` enthält nur abgeschlossene und genehmigte Arbeit
   - Code Reviews als Pflicht (im Optimalfall nicht zu umgehen):
     - review before commit, d.h. kein Branching
       - +: technisch einfach realisierbar
       - -: kann umgangen werden
     - review after commit, d.h. `push` erstellt neuen Pull Request (PR) Branch
       - +: sauberer `master`-Branch ist wahrscheinlicher
       - -: höhere technische Anforderungen (`push` in PR-Branch -> Review-Freigabe in CI -> `merge` des PR in `master` durch CI)

Literatur

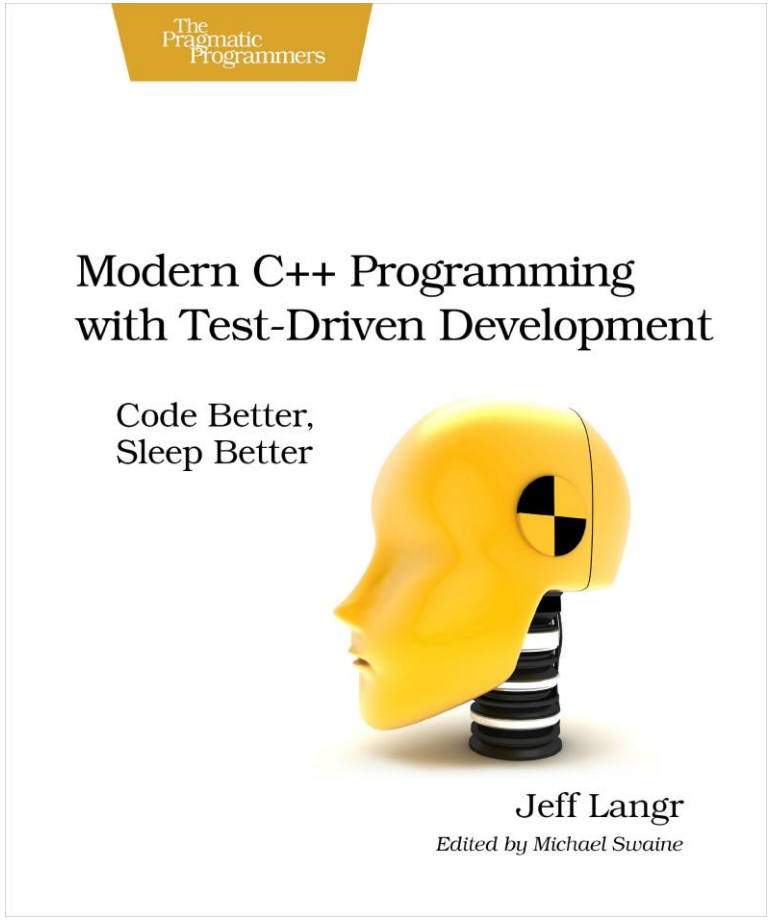
Literatur | Vorträge

YouTube-Playlist: C++ Testing

> “Conference talks related to the topic of software testing in the C++ programming language.”

Empfehlung: Clean Code Talks von Misko Hevery (in Playlist enthalten)

Literatur | Bücher

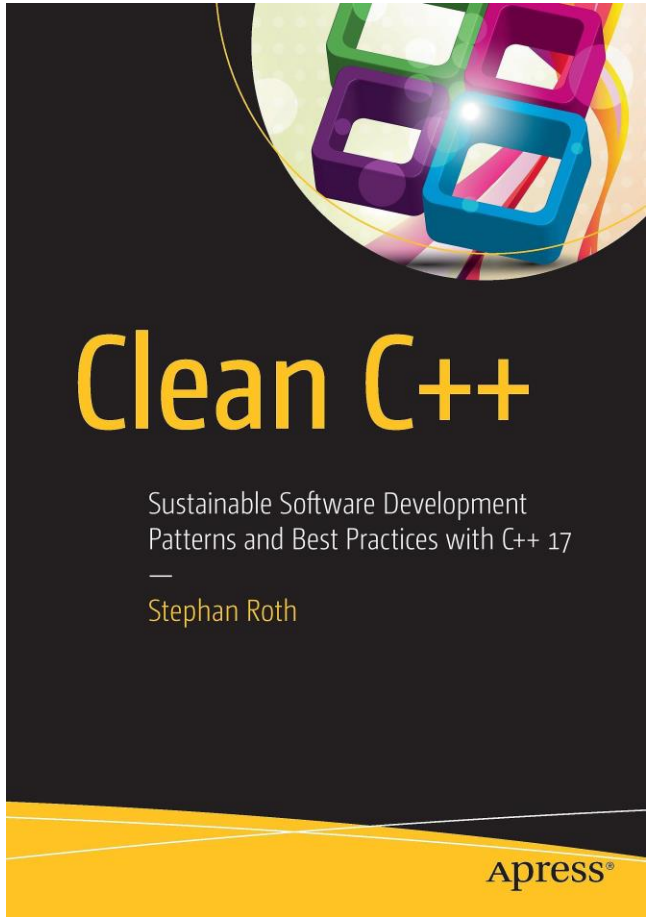


Langr, J. (2013). *Modern C++ Programming with Test Driven Development: Code Better, Sleep Better*. Raleigh: Pragmatic Bookshelf.

Literatur | Bücher

Spillner, A. and Breymann, U. (2016). *Lean Testing für C++-Programmierer*. Heidelberg: dpunkt.

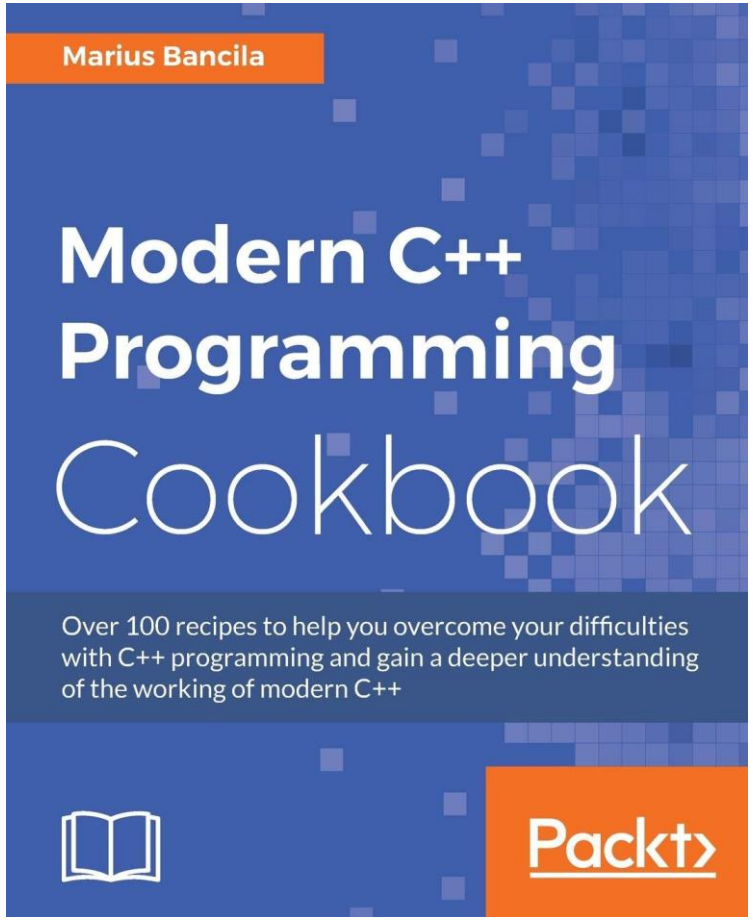
Literatur | Bücher



Roth, S. (2017). *Clean C++: Sustainable Software Development Patterns and Best Practices with C++ 17*. New York City: Apress.

section 2: Building a Safety Net

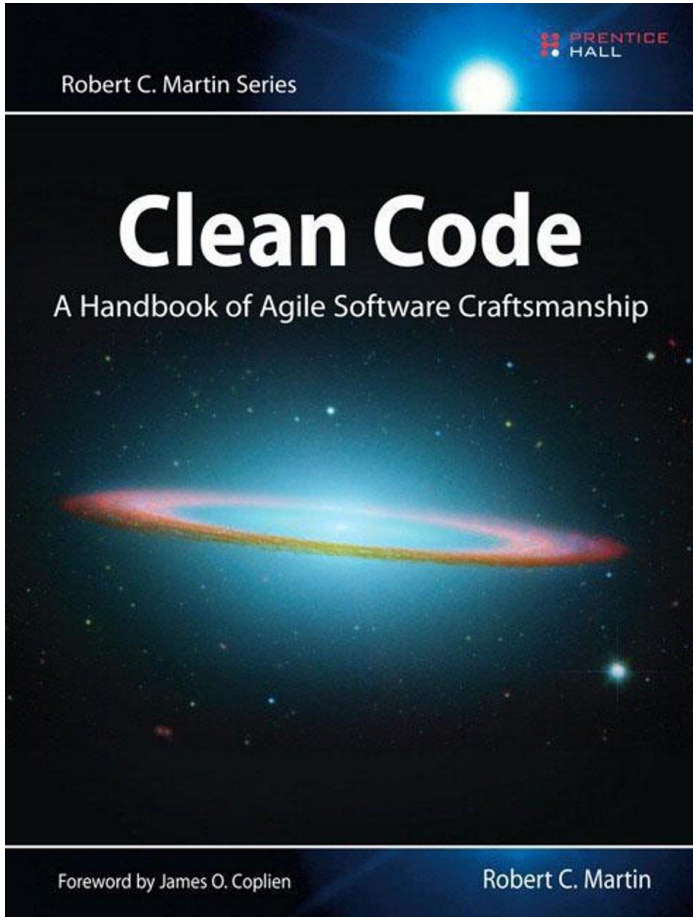
Literatur | Bücher



Bancila, M. (2017) *Modern C++ Programming Cookbook*. Birmingham: Packt Publishing.

section 11: Exploring Testing Frameworks

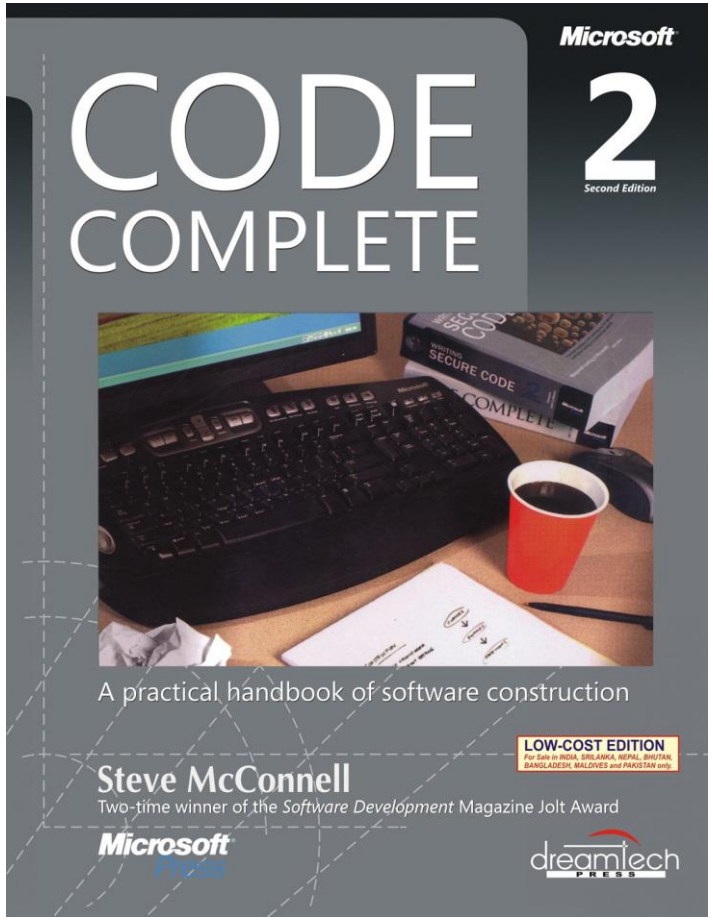
Literatur | Bücher



Martin, R.C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)*. Upper Saddle River: Prentice Hall.

chapter 9: Unit Tests

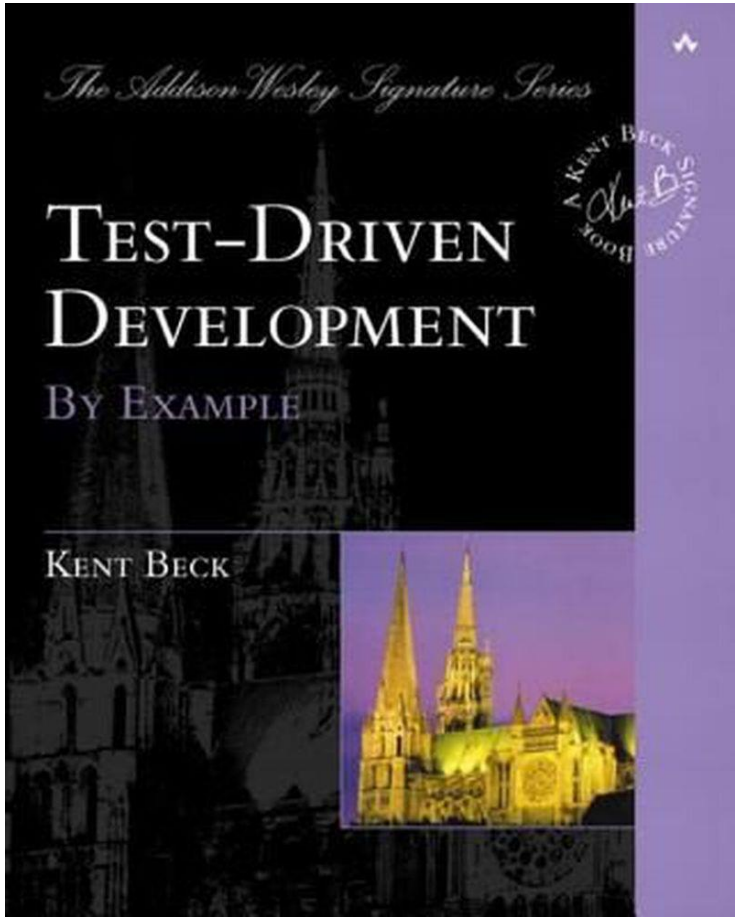
Literatur | Bücher



McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. 2nd ed. Redmond: Microsoft Press.

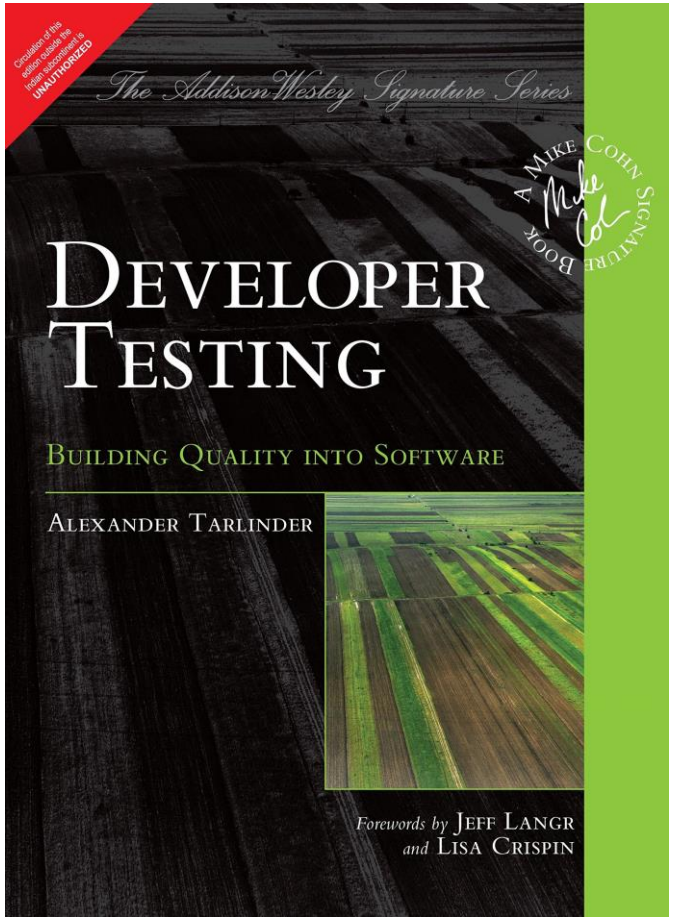
part V, section 22: Developer Testing

Literatur | Bücher



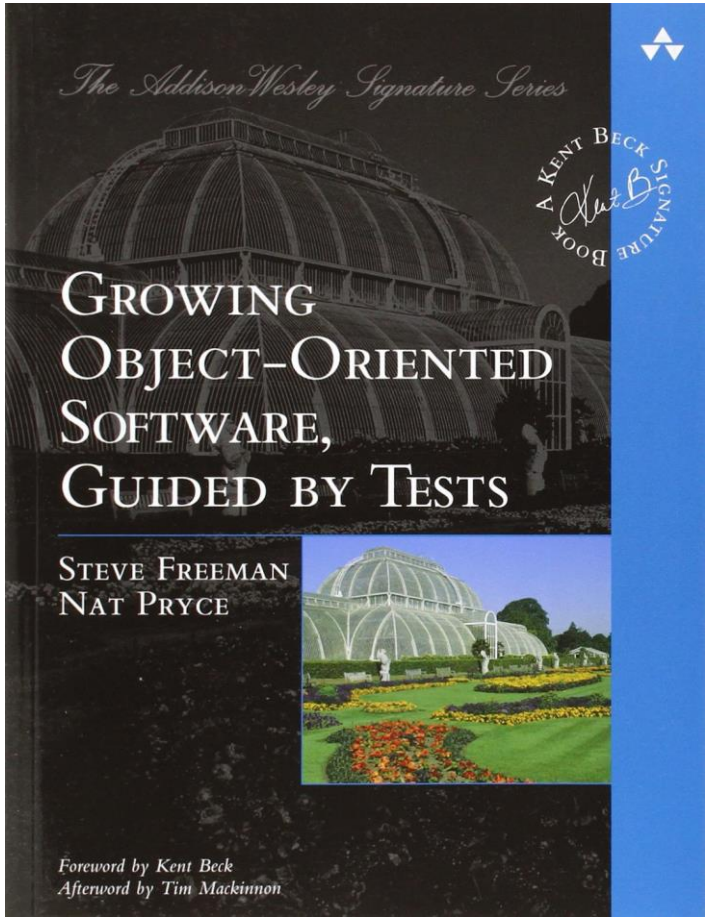
Beck, K. (2002). *Test Driven Development: By Example (Addison-Wesley Signature Series)*. Boston: Addison-Wesley.

Literatur | Bücher



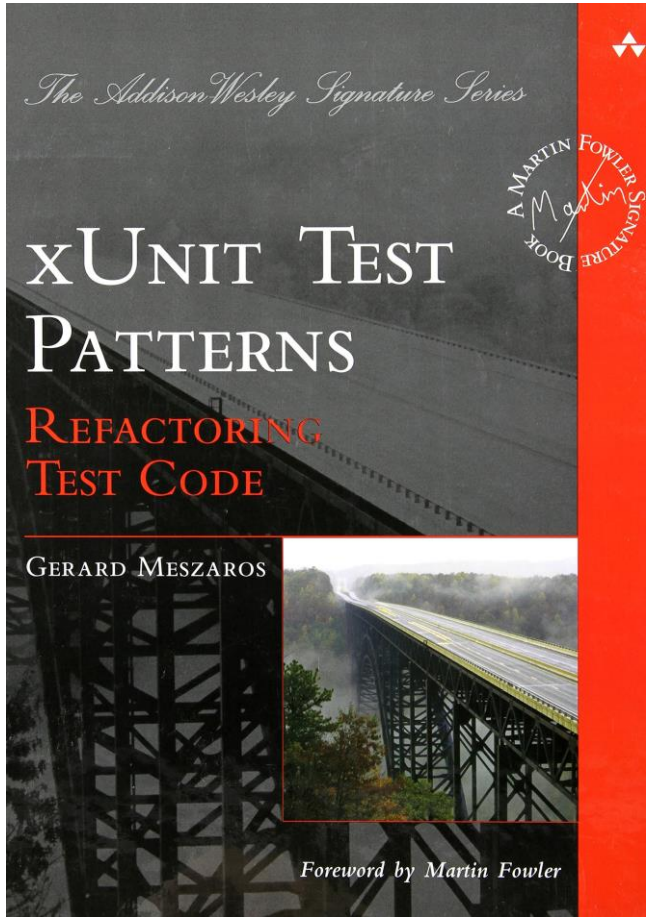
Tarlinder. A. (2016). *Developer Testing: Building Quality Into Software (Mike Cohn Addison-Wesley Signature)*. Boston: Addison-Wesley.

Literatur | Bücher



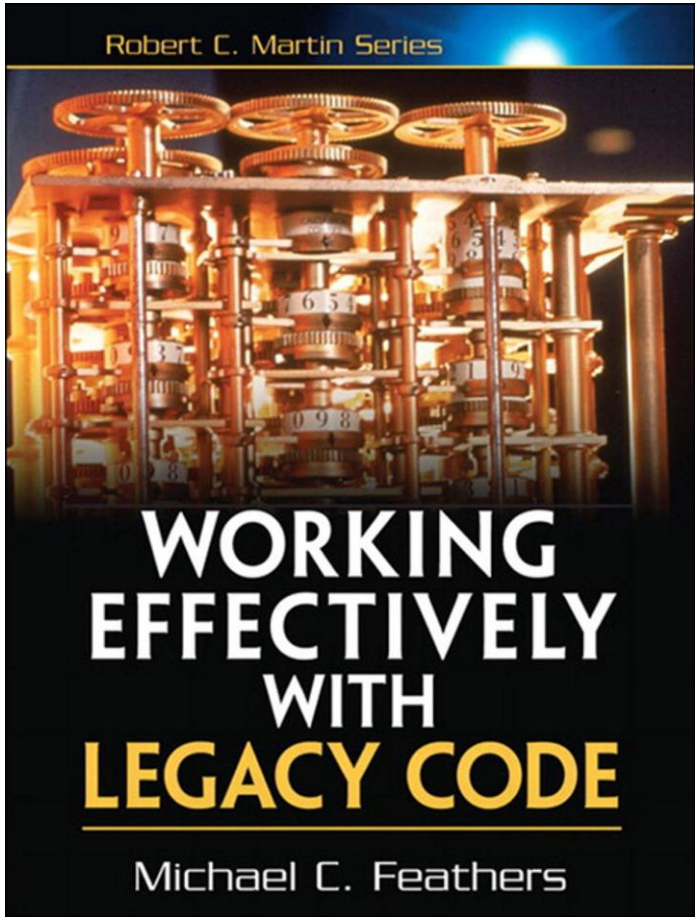
Freeman, S. and Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Boston: Addison-Wesley.

Literatur | Bücher



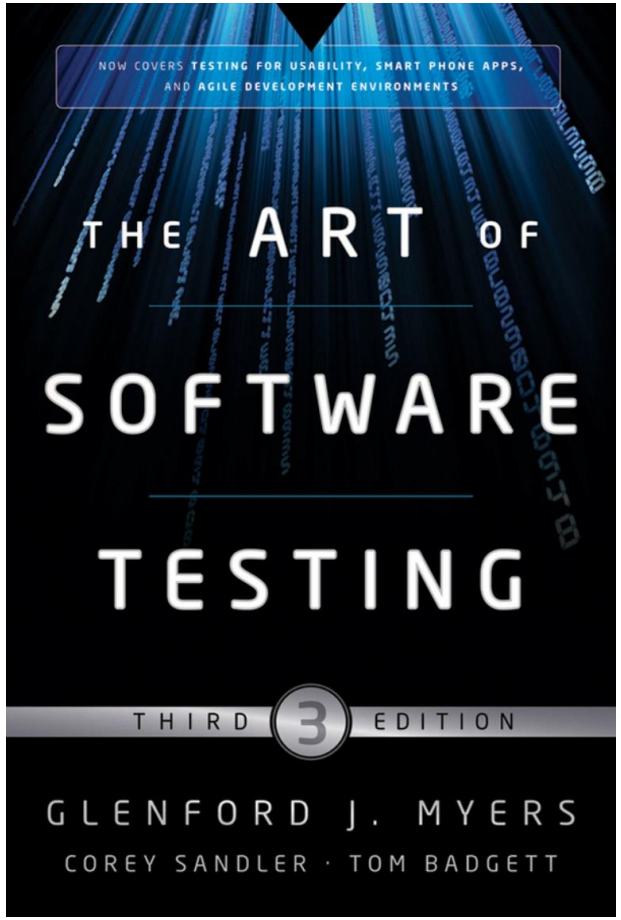
Mezzaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code (Martin Fowler Addison-Wesley Signature)*. Boston: Addison-Wesley.

Literatur | Bücher



Feathers, M.C. (2004). *Working Effectively with Legacy Code (Robert C. Martin Series)*. Upper Saddle River: Prentice Hall.

Literatur | Bücher

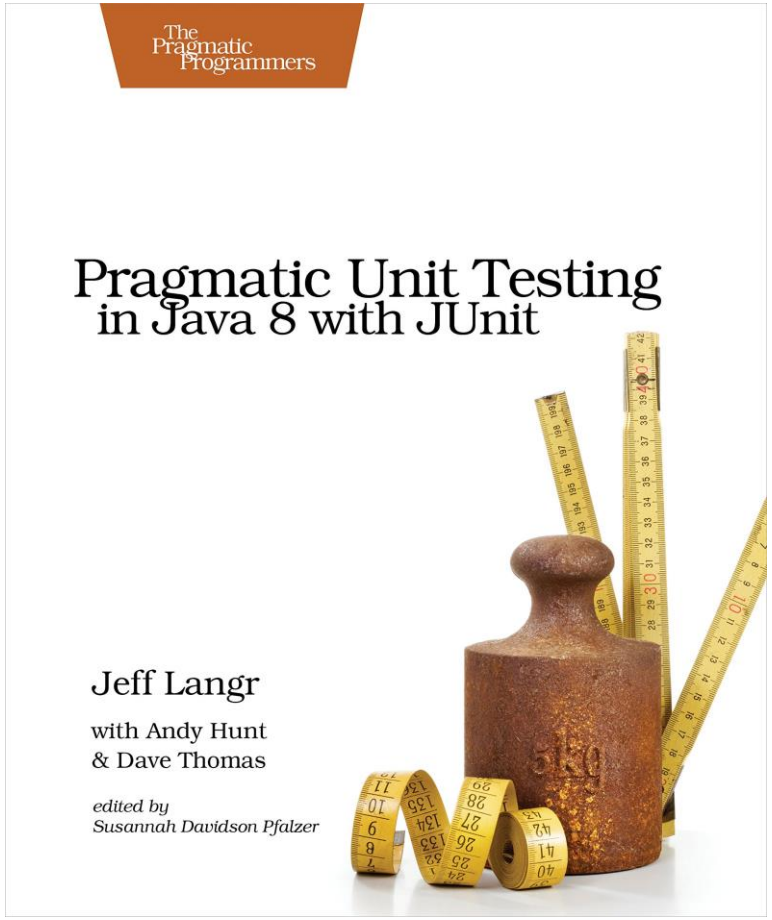


Corey, G.J. et al (2011). *The Art of Software Testing*. 3rd ed. New Jersey: Jon Wiley & Sons.

Literatur | Bücher

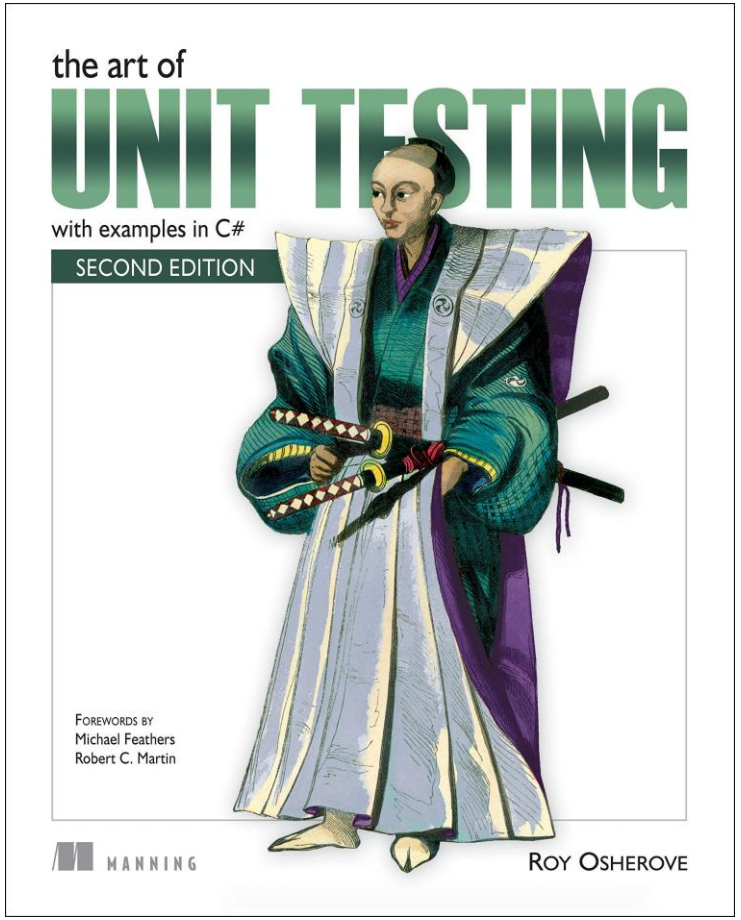
National Institute of Standards & Technology (NIST) (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing*. [online] Available at: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf [Accessed 11 Apr. 2019].

Literatur | Bücher



Langr, J. et al (2015). *Pragmatic Unit Testing in Java 8 with JUnit. Raleigh*: Pragmatic Bookshelf.

Literatur | Bücher



Osherove, R. (2013). *The Art of Unit Testing: with Examples in C#*. 2nd ed. New York: Manning.

Literatur | Webseiten | Developer Testing

- Hevery, M. (2008). *My Unified Theory of Bugs*. [online] Available at: https://testing.googleblog.com/2008/11/my-unified-theory-of-bugs.html [Accessed 11 Apr. 2019].

- Vocke, H. (2018). *The Practical Test Pyramid*. [online] Available at: https://martinfowler.com/articles/practical-test-pyramid.html [Accessed 11 Apr. 2019].

- Wacker, M. (2015). *Just Say No to More End-to-End Tests*. [online] Available at: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html [Accessed 11 Apr. 2019].

Literatur | Webseiten | Developer Testing

- Hevery, M. (2008). *Changing Developer Behaviour, Part I*. [online] Available at: http://www.alphaitjournal.com/2008/08/hevery-changing-developer-behaviour.html [Accessed 11 Apr. 2019].

- Hevery, M. (2008). *Changing Developer Behaviour, Part II*. [online] Available at: http://www.alphaitjournal.com/2009/06/hevery-changing-developer-behaviour.html [Accessed 11 Apr. 2019].

- Scott, A. (2012). *Testing Pyramids & Ice-Cream Cones*. [online] Available at: https://watirmelon.blog/testing-pyramids [Accessed 11 Apr. 2019].

Literatur | Webseiten | Unit Testing

- Ottinger, T. and Langr, J. (2009). *F.I.R.S.T*. [online] Available at: https://agileinaflash.blogspot.com/2009/02/first.html [Accessed 11 Apr. 2019].

- Wake, B. (2011). *3A – Arrange, Act, Assert*. [online] Available at: https://xp123.com/articles/3a-arrange-act-assert [Accessed 11 Apr. 2019].

Literatur | Webseiten | Test Doubles

- Bhatt, N. (2011). *Dummy vs. Stub vs. Spy vs. Fake vs. Mock*. [online] Available at: https://nirajrules.wordpress.com/2011/08/27/dummy-vs-stub-vs-spy-vs-fake-vs-mock [Accessed 11 Apr. 2019].

- Fowler, M. (2007). *Mocks Aren‘t Stubs*. [online] Available at: https://martinfowler.com/articles/mocksArentStubs.html [Accessed 11 Apr. 2019].

- Mezzaros, G. (2009). Test Double. [online] Available at: http://xunitpatterns.com/Test Double.html [Accesses 11 Apr. 2019].

Literatur | Webseiten | TDD

- Billups, T. (2010). *How test first development changed my life*. [online] Available at: https://toranbillups.com/blog/archive/2010/04/22/How-test-first-development-changed-my-life [Accessed 11 Apr. 2019].

- Martin, R.C. (2005). *The Three Laws of TDD*. [online] Available at: http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd [Accessed 11 Apr. 2019].

- Martin, R.C. (2016). Giving Up on TDD. [online] Available at: https://blog.cleancoder.com/uncle-bob/2016/03/19/GivingUpOnTDD.html [Accessed 11 Apr. 2019].

- Beck, K. and Hansson, D.H. and Fowler. M. (2014). Is TDD Dead? Available at: https://martinfowler.com/articles/is-tdd-dead [Accessed 11 Apr. 2019].

Literatur | Webseiten | TDD

- Wikipedia (2019). *Test-driven development*. [online] Available at: https://en.wikipedia.org/wiki/Test-driven_development [Accessed 11 Apr. 2019].

- WikiWikiWeb (2014). *Test Driven Development*. [online] Available at: http://wiki.c2.com/?TestDrivenDevelopment [Accessed 11 Apr. 2019].

- Kirtland, C. (2015). *TDD Problems*. [online] Available at: https://sites.google.com/site/tddproblems [Accessed 11 Apr. 2019].

- Vaccari, M. (200x). *TDD Resources*. [online] Available at: http://matteo.vaccari.name/blog/tdd-resources [Accessed 11 Apr. 2019].

Literatur | Webseiten | Google Test

- Google (2019). *Google Test Primer*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googletest/docs/primer.md [Accessed 11 Apr. 2019].

- Google (2019). *Advanced Google Test Topics*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googletest/docs/advanced.md [Accessed 11 Apr. 2019].

- Google (2019). *Google Test Frequently Asked Questions*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googletest/docs/faq.md [Accessed 11 Apr. 2019].

Literatur | Webseiten | Google Mock

- Google (2019). *Google Mock For Dummies*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googlemock/docs/ForDummies.md [Accessed 11 Apr. 2019].

- Google (2019). *Google Mock Cook Book*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googlemock/docs/CookBook.md [Accessed 11 Apr. 2019].

- Google (2019). *Google Mock Cheat Sheet*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googlemock/docs/CheatSheet.md [Accessed 11 Apr. 2019].

- Google (2019). *Google Mock Frequently Asked Questions*. [online] Available at: https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googlemock/docs/FrequentlyAskedQuestions.md [Accessed 11 Apr. 2019].

Literatur | Webseiten | Modern C++

- Turner, J. (2019). *cppbestpractices*. [online] Available at: https://lefticus.gitbooks.io/cpp-best-practices/content [Accessed 11 Apr. 2019].
- Stroustrup, B. and Sutter, H. (2019). *C++ Core Guidelines*. [online] Available at: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines [Accessed 11 Apr. 2019].
- Rigtorp, E. (2019) *Awesome Modern C++*. [online] Available at: https://github.com/rigtorp/awesome-modern-cpp [Accessed 11 Apr. 2019].
- AUTOSAR (2017). *AUTOSAR Guidelines for the use of the C++14 language in critical and safety-related systems*. [online] Available at: https://autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf [Accessed 01 Apr. 2019].