Software Engineering und Projektmanagement 2.0 VO

Implementierung, Integration & Testen (Teil 1)

Dr. Alexander Zapletal
www.inso.tuwien.ac.at

INSO - Industrial Software
Institut für Rechnergestützte Automation | Fakultät für Informatik | Technische Universität Wien

Ziele und Aufgaben der Implementierung

- Umsetzung der Anforderungen des Kunden
- Umsetzung der Architektur in eine Software
- Zeitgerechte Lieferung der Software
- Entwicklung einer wartbaren Software

Definition (IEEE 610.12, 1990)

(1) „The process of translating a design into hardware components, software components, or both“
(2) „The result of the process in (1)“

Inhalt

Voraussetzungen für die Implementierung

- Architektur muss vorhanden sein, je ausführlicher, desto besser.
- Ein Team muss definiert werden.
- Festlegung von
  - Entwicklungsprozess
  - Programmiersprache
  - Tools
  - Zeitplan

Die Notwendigkeit einer Architektur

- Architektur gibt den Bauplan des Produkts vor.
- Bauplan teilt Produkt in Komponenten mit Verantwortlichkeiten.
  - Typischerweise: mehrere Personen arbeiten an unterschiedlichen Komponenten
  - Bauplan sorgt daher für saubere Umsetzung.
- Entwickler(-Team) hat eine „Anleitung“ zur Orientierung.
- Kreativität trotzdem nötig, WIE die einzelnen Komponenten umgesetzt werden.

Werkzeuge der Implementierung

- Programmiersprache
- Frameworks
- Entwicklungsumgebung (IDE)
- Versionsmanagement (VCS)
- Build Management

Programmiersprachen

- Wahl der Programmiersprachen hat wesentlichen Einfluss auf die Implementierungsphase
- Selten direkt für das Scheitern eines Projektes verantwortlich
- Beherrschung der Sprache notwendig aber nicht hinreichend für Projekterfolg
- Faktoren die diese Sprachwahl beeinflussen:
  - Kenntnisse und Erfahrungen des Teams
  - Eignung der Sprache für die Problem-Domäne (Enterprise System, Mobile Device, Embedded System, …)
  - Verfügbarkeit von Entwicklungstools, Frameworks und Libraries
  - Verfügbarkeit von Entwickler
  - Kundenvorgabe

Framework

- Applikationsskelett, das vom Entwickler angepasst (spezialisiert) wird

- Beispiele: Spring, Hibernate, Angular, JUnit

- Vorteile
  - Wiederverwendung
  - Entwicklungszeit kürzer, Kosten senken
  - Softwarequalität erhöhen
  - Entwicklung kann auf Spezialisierung vorgegebener Strukturen reduziert werden
- Nachteile
  - Komplexität (Black/Magic Box), daher Einarbeitung und Erfahrung notwendig

Kriterien zur Auswahl eines Frameworks

- Deckt das Framework die Anforderungen ab?
  - Siehe Dokumentation
- Wartung und Weiterentwicklung gesichert?
  - Hoher Verbreitungsgrad bedeutet meist gute Wartung, Weiterentwicklung, Stabilität
- Kommerziell: Support, aber Kosten, auch für Änderungen
- Frei: Support bei Bedarf zukaufen,
  aber Risiko durch fehlende Garantien

Library vs. Framework

- Library (Bibliothek)
  - wiederverwendbare Funktionalität
  - wird vom Entwickler-Code aufgerufen

- Framework
  - wiederverwendbares Verhalten
  - Framework ruft Entwickler-Code auf
    (Inversion of Control, Hollywood-Prinzip)

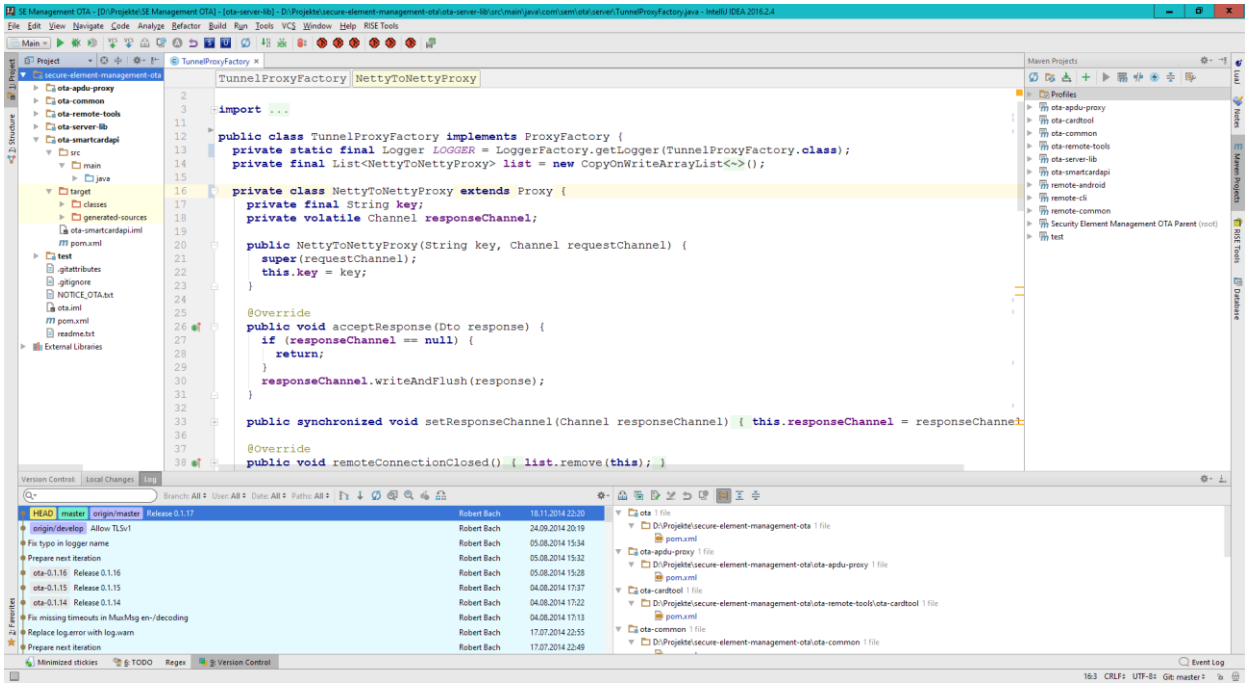
Framework: Beispiel Web Service

```
@Path(”/store”)
public class StoreWebHandler {

 @GET
 @Path(”/books/author/{name}”)
 public
 List<Book> getBooksByAuthor(@QueryParam(”name”) String name) {
    // YOUR CODE HERE
  }

  @POST
  @Path(”/books”)
  public void addBook(Book newBook) {
    // YOUR CODE HERE
  }
}
```

```
GET myhost.com/store/books/author/Miller
```

Die Entwicklungsumgebung (IDE)

- IDE = Integrated Development Environment
- Zentraler Arbeitsbereich für Entwickler
- Zugriff auf alle benötigten Tools
- Flexibles Layouting und individuelle Gestaltung/Anpassung
- Beispiele:
  - JetBrains Familie (IntelliJ IDEA, WebStorm, …)
  - Eclipse
  - NetBeans
  - XCode
  - Visual Studio



Funktionsumfang einer IDE

- Source File Editor
  - Syntax Highlighting
  - Code Completion
  - Snippets / Templates
- Integriert Compiler (inkl. inkrementelles Compilen), Linker, Interpreter
- Debugger, Profiler
- Unterstützung für Refaktorisierung
- Unterstützung für einheitlichen Code-Qualität (statische Analyse) und -Formatierung
- Außerdem: Build Tools, Unit Testing, Volltextsuche, Strukturelle Suche, VCS Integration, …

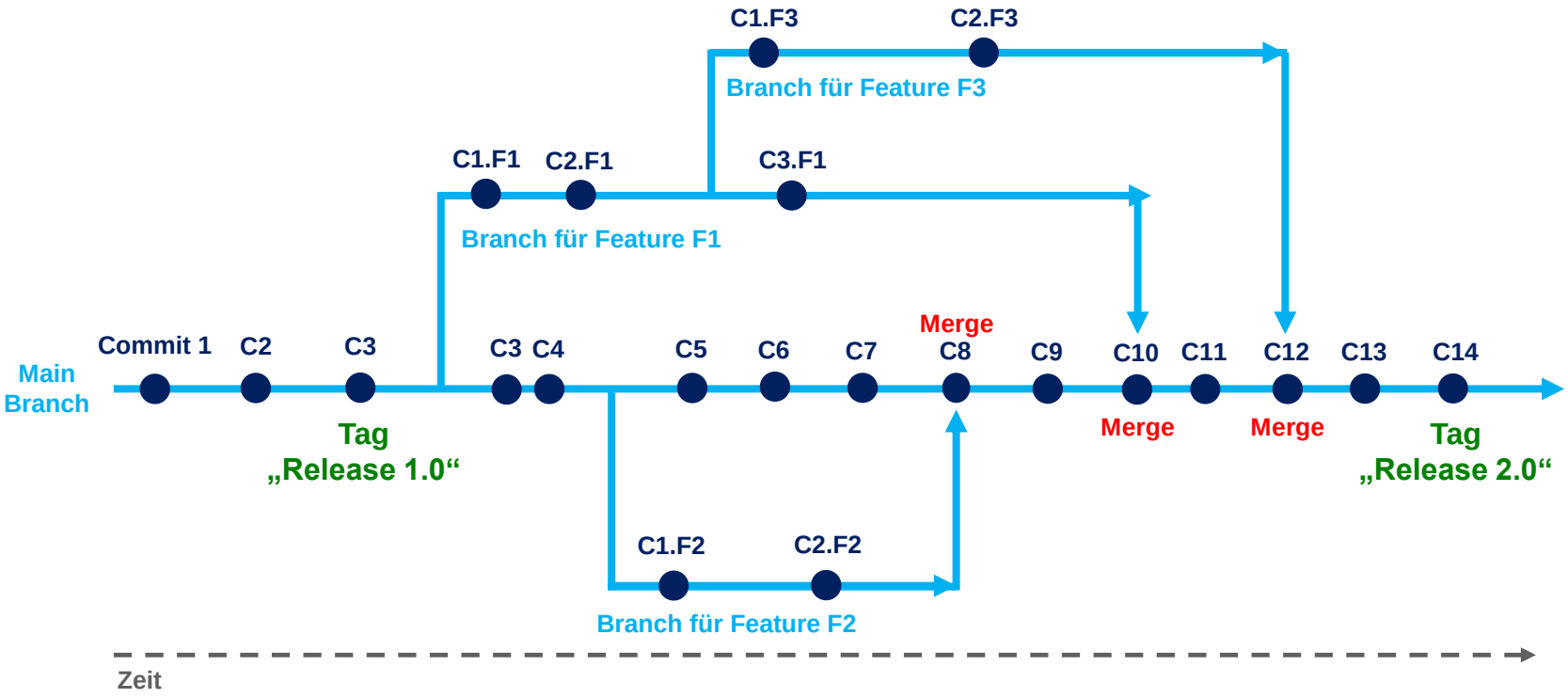
Version Control System

- VCS (Versionsmanagementsystem)
  - ist ein Computersystem, das Änderungen an Dateien über die Zeit hinweg archiviert.
  - erlaubt jederzeit das Laden der aktuellen und jeder archivierten Version einer Datei.
  - Beispiele: Git, Svn, Mercurial

- *Repository*: Datenbestand aus dem die Arbeitskopie generiert wird.
- *Checkout*: Übertragung (Abholen) der Daten aus dem Repository
- *Check-in* oder *Commit*: Übertragung der Daten in das Repository

Version Control System (2)

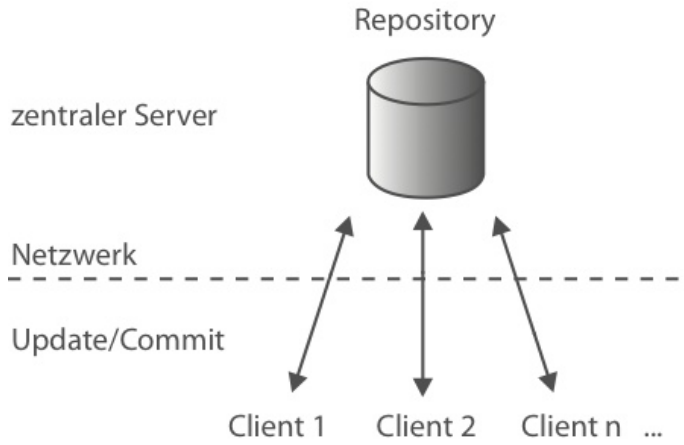
- Verzweigung/Branching: Unter Branching versteht man das Anlegen einer Kopie von Objekten im VCS



Zentrales/Dezentrales Versionsmanagement

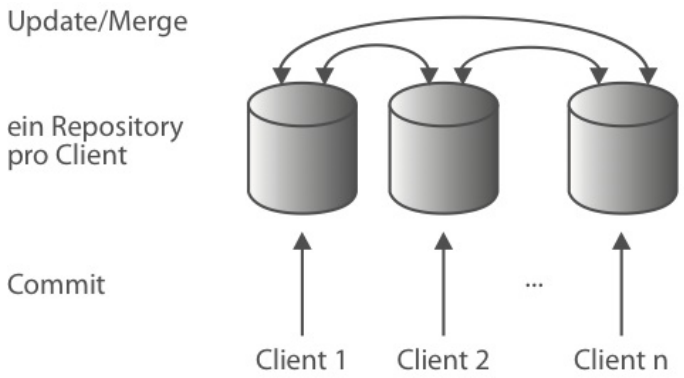
Zentrales VCS:

- Client/Server Ansatz
- Zentraler Server
- Beispiele
  - SVN (Subversion)
  - CVS (Concurrent Versions System)



Dezentrales VCS:

- Kein zentrales Repository
- Jeder Entwickler besitzt sein eigenes Repository
- Änderungen werden zwischen den Repositories ausgetauscht
- Beispiel: Git



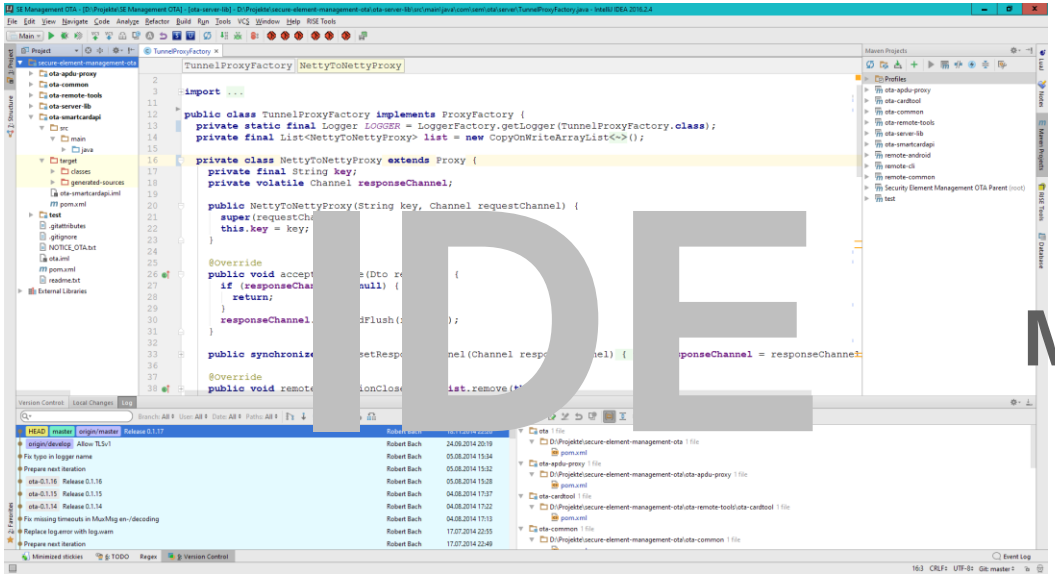
Build Management System (BMS)

- Werkzeug, zum Bauen einer Applikation oder Teilen (Modulen) davon
- Stellt sicher, dass benötigte (Sub-)Module verfügbar sind

- Beispiele
  - Make (Urvater der BMS), für C/C++
  - Maven, für Java
  - Ant, für Java
  - Gradle, für Java, insb. Android

IDE als Allzwecktool

VCS



Build Management

Code-standards

Framework-Support

Wartungsfreundliche Implementierung

Kopplung, Kohäsion
Schnittstellen, Information Hiding
Entwurfsmuster (Design Patterns)

Kopplung und Kohäsion

- Kopplung
  - Maß der Abhängigkeit zwischen Komponenten
  - Hohe Kopplung → schlechter Programmierstil

- Kohäsion
  - Maß der Zusammengehörigkeit innerhalb einer Komponente
  - Hohe Kohäsion durch Methodengruppierungen nach Zweck gewährleistet
  - Single-Responsibility-Prinzip: eine Komponente erfüllt genau eine Aufgabe
  - Noch besser: nur diese Komponente erfüllt diese Aufgabe (Don‘t-Repeat-Yourself-Prinzip)

- **Ziel: Geringe Kopplung, erreicht durch hohe Kohäsion**

Schnittstellen und Information Hiding

- Interfaces zwischen Komponenten (Klasse, Modul, …) sauber definieren

- Information Hiding
  - Komponente soll von anderer Komponente nur so viel wie nötig wissen (z.B. gehaltene Daten, Datenstrukturen, Algorithmen).
  - Kommunikation zwischen Komponenten über Interfaces
  - Siehe auch Datenkapselung als Konzept der objektorientierten Programmierung

- Saubere Schnittstellen auch für Testbarkeit einer Komponente wichtig.

Entwurfsmuster (Design Patterns)

- Toolbox für Programmierer
- Bieten für bestimmtes Programmierprobleme bewährte Lösungen
- Große Anzahl an verfügbaren Entwurfsmuster

- Referenzwerk: *Design Patterns. Elements of Reusable Object-Oriented Software.* (Gamma et al)

- Creational Patterns: Abstract Factory, Builder, Singleton, Prototype, ...
- Structural Patterns: Adapter, Bridge, Decorator, Facade, Proxy,…
- Behavioral Patterns: Command, Iterator, Observer, Strategy, ...

Beispiel: Adapter Pattern

Library 1

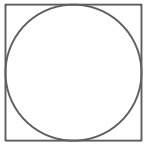
```
Class Primitive
  display()

Class Rectangle : Primitive

Class Circle : Primitive
```

Library 2

```
Interface Shape
  draw()

Class ShapeCombiner
  add(Shape s)
  draw()
```

```
Class ShapeAdapter : Shape
  Primitive primitive;
  ShapeAdapter(Primitive p) { primitive = p }
  draw() { primitive.display() }
```

```
Shape rect = new ShapeAdapter(new Rectangle(-1,-1,1,1))
Shape circle = new ShapeAdapter(new Circle(0,0,1))
new ShapeCombiner().add(rect).add(circle).draw()
```



Entwicklungsrichtlinien

- Diverse Richtlinien für Entwickler eines Projektes
- Ziel: konsistenter Code
- Motivation: kürzere Einarbeitungszeit für Entwickler

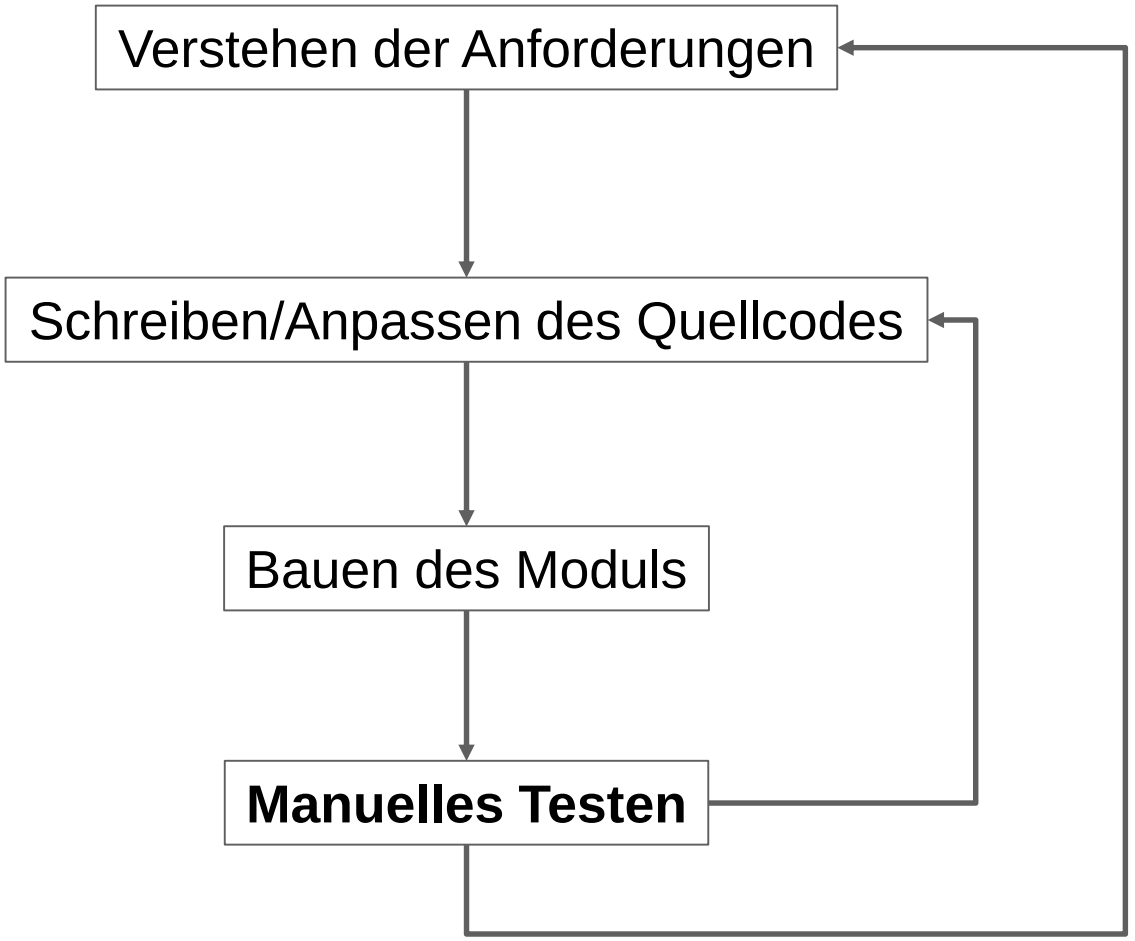
- Richtlinien betreffen:
  - Namenskonventionen für Variablen/Methoden/Klassen/…
  - Programmierstil
  - Formatierung des Quellcodes
  - Kommentarstil
  - Verwendete Design Patterns

- Teilweise automatisiert überprüfbar (Checkstyle,…)

Dokumentation des Codes

- Erhöht die Lesbarkeit und Übersicht
- Code kann auch nach Monaten wieder verstanden werden
- Code kann von anderen Entwicklern besser nachvollzogen werden
- Kommentierung
- Entwicklungsrichtlinien
- Oftmals die einzige Art der Dokumentation, die zum Zeitpunkt der Ablöse des Systems noch erhalten ist

Jeder Entwickler soll auch Tester sein



Besser: Automatisierte Tests

- Tests oft unstrukturiert, oberflächlich und lückenhaft – da lediglich „ausprobieren“
- Lösung: automatisierte, feingranulare Funktionstests (Unit Tests)
- Einfach ausführbar (Unterstützung durch IDE, Integration in Build Prozess)
- Sehr wichtig zur Vermeidung von ungewollten Nebeneffekten

- **Erst eine gute Codeabdeckung durch automatisierte Tests erlaubt das Bauen größerer, robuster Softwaresysteme (und die „unbeschwerter“ Weiterentwicklung) !**

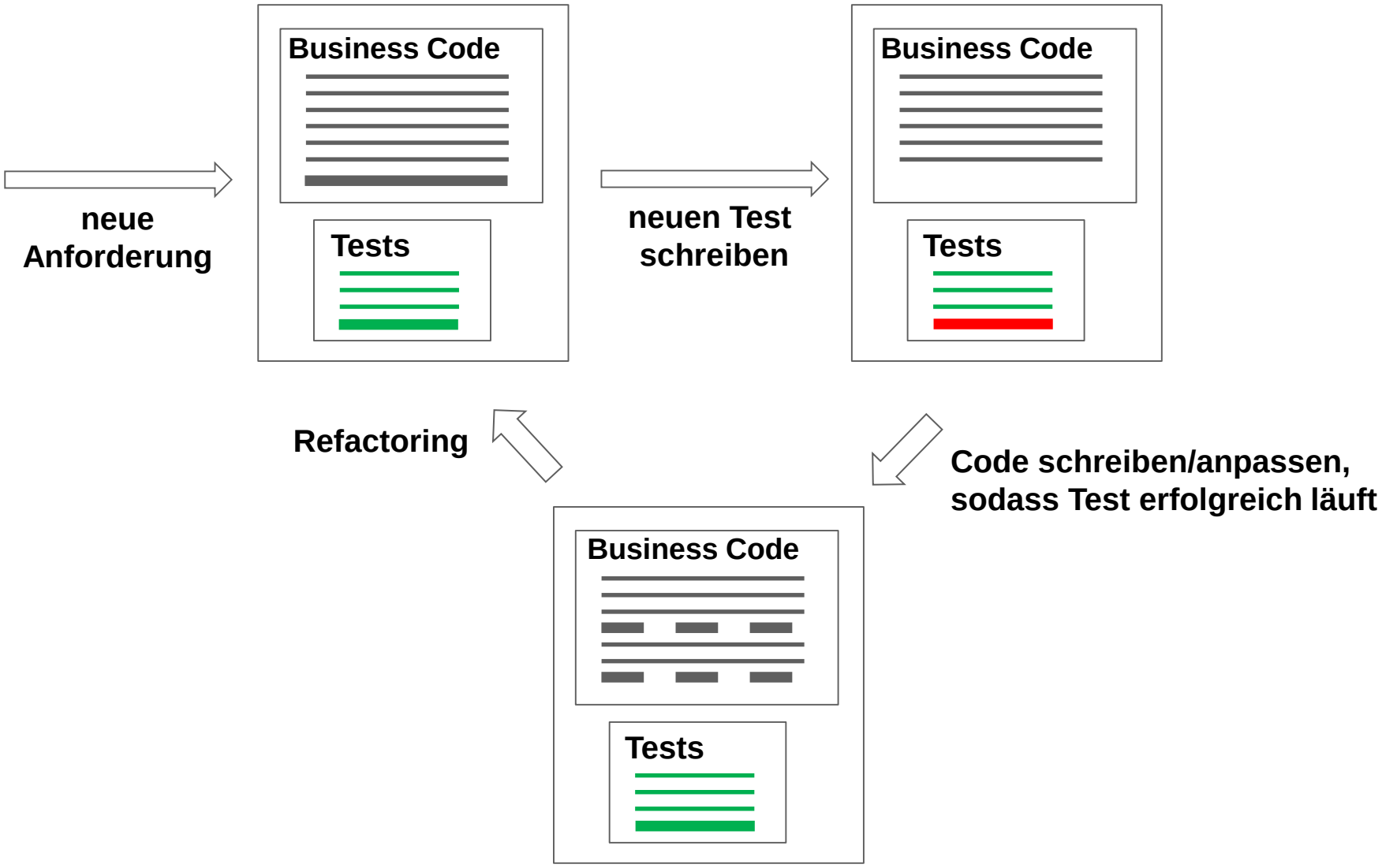
Test Driven Development (TDD)

- TDD ist eine Entwicklungsmethode, bei der zuerst die Tests und erst danach der eigentliche Code geschrieben werden.

- Typischer Ablauf
  1. Schreiben eines fehlschlagenden Tests für eine neue Anforderung
  2. Code schreiben, sodass Test erfolgreich durchläuft
  3. Refactoring

- Zwei goldene Regeln (Beck 2002)
  - „*You should write new business code only when an automated test has failed*“
  - „You should eliminate any duplication that you find“

Test Driven Development (2)



Zusammenfassung Implementierung

- Implementierungen im Team sind ein komplexer Prozess.
- Die Wahl der geeigneten Frameworks können den Entwicklungsprozess positiv beeinflussen.
- Bei der Entwicklung von (größeren) Projekten ist die Unterstützung durch Tools unerlässlich.
- Automatisierte Tests wichtig für robuste, wartbare Softwaresysteme!
- Dokumentation und Entwicklungsrichtlinien erleichtern die Kommunikation und nachfolgende Wartungsphase.
- Die Entwicklung im Team erfordert definierte Entwicklungsprozesse.

Was kommt nach der Implementierung?

Software im Weltall

- Steuercomputer einer Venussonde (1962)
- *Mars Climate Orbiter* (1999)
  - Navigationssystem lieferte Daten im englischen Maßsystem
  - Boardcomputer erwartete Daten im metrischen Maßsystem
- *Mars Polar Lander* (1999)
  - Sensor meldete wegen einer Erschütterung fälschlicherweise Bodenkontakt
  - Bremsraketen wurden zu früh deaktiviert
- Mars-Sonde *Schiaparelli (**2016**)*
  - Unerwartete Bewegungen führten zu falscher Sensormessung
  - Wirkliche Höhe 3,7 km, berechnete Höhe -2 km
  - => zu wenig Bremsung

Inhalt

Stellenwert von Softwaretests

- Keine Software frei von Fehlern
- Testen gehört zu den wichtigsten Aktivitäten des Entwicklungsprozesses.
- Systematisches Testen notwendig
- Eine gut geplante Testphase ist die Grundlage für ein erfolgreiches Projekt.
- Ziel ist es, Fehler so früh wie möglich zu finden
- Softwaretests sind eine dynamische und produktorientierte Qualitätssicherungstechnik.
- Aufgrund wachsender Komplexität und hoher Abhängigkeiten von Softwaresystemen, wird der Softwaretest strategisch immer bedeutender.

Definition Softwaretests

Definition nach IEEE 610.12, 1990 :

„Software testing is a **formal** process carried out by a **specialized testing team** in which a software unit, several integrated software units or an entire software package are examined by **running the programs** on a computer.

All the associated tests are performed according to **approved test procedures on approved test cases**.“

Grundsätze des Softwaretests

International Software Testing Qualifications Board (ISTQB):

1. Testen zeigt die Anwesenheit von Fehlern
2. Vollständiges Testen ist nicht möglich
3. Mit dem Testen frühzeitig beginnen
4. Häufung von Fehlern
5. Tests müssen überarbeitet und erweitert werden
6. Testen ist abhängig vom Umfeld
7. Trugschluss: Keine Fehler bedeutet ein brauchbares System

Ziele und Methodik des Softwaretestens

- Ziele:
  - Verifikation: Prüfen des Systems gegen Spezifikation
    - „Erstellen wir das Produkt richtig“
  - Validierung: Prüfen des Systems gegen (Kunden-)Anforderungen
    - „Erstellen wir das richtige Produkt“

- Methodik:
  - Testfälle identifizieren, mit denen die höchste Wahrscheinlichkeit gegeben ist, festzustellen, ob das Softwaresystem korrekt funktioniert

  - Möglichst hohe funktionale oder nichtfunktionale Abdeckung