DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Zielgruppe: **Wirtschaftsinformatik**

Modul: **Software Engineering**

Foliensatz: # Implementierung und Test

Prof. Dr. Malte Prieß

Lernziele

- Best Practices: Implementierung und Test
- Konfigurationsmanagement
- DevOps
- Continuous Integration, Continuous Delivery und Continuous Deployment
- Testing
  - Komponententests, Systemtests, Abnahmetests
  - Test-Driven Development (TDD)
  - Komponententests mit Hilfe des JUnit-Frameworks

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Best Practices: Implementierung und Test

Prof. Dr. Malte Prieß

Lernziele

- Best Practices: Implementierung und Test
- DevOps
- Konfigurationsmanagement
- Continuous Integration, Continuous Delivery und Continuous Deployment
- Testing
  - Komponententests, Systemtests, Abnahmetests
  - Test-Driven Development (TDD)
  - Komponententests mit Hilfes des JUnit-Frameworks

Implementation embodies

(1) the process of **translating a design into software components**, hardware components, or both

and (2) the result of the process in (1).

Definition IEEE-Standard 610.12 (1990)

Ziele der Implementierungsphase eines Softwareprojekts

- Umsetzung des Entwurfs in lauffähige Software, i.d.R. Erzeugung von Quelltext in ausgewählter Programmiersprache
- Einhaltung der nicht-funktionalen Anforderungen, z.B. Usability, Performance

Vorbedingungen

- Ein konzeptioneller Entwurf der Software muss vorhanden sein
  - Je detaillierter der Entwurf ist, desto einfacher gelingt die Umsetzung
- Ein Team zur Umsetzung muss definiert und arbeitsteilig organisiert sein
  - Modularisierung und abgegrenzte Arbeitspakete sind für eine effiziente Teamarbeit erforderlich
- Entscheidungen bzgl. Programmiersprache, Frameworks und Toolunterstützung müssen getroffen sein
  - Wechseln eines Tools (z.B. IDE, Versionskontrolle, Projekt-management) während des Projekts ist aufwändig

Architekturentwurf ist der Bauplan für ein komplexes System

- Ein guter Bauplan sorgt für eine **geordnete Umsetzung** durch hierarchische **Gliederung** des neuen Systems in **Komponenten**
- Ein Bauplan ist insb. in Projekten notwendig, an welchen **mehrere Personen** an unterschiedlichen Komponenten arbeiten
- Der Bauplan gibt den Entwicklern **Orientierung** über ein komplexes System und hilft **Verantwortlichkeiten** für Komponenten/Arbeitspakete **zuzuweisen**

Quelle: Gamma et al. (2010)

Faktoren zur Auswahl der Programmiersprache

- Kenntnisse im Team
  - Programmiersprache ist Werkzeug der Entwickler
- Anforderungen des Auftraggebers
  - Integration des Projekts in Systemlandschaft des Auftraggebers
- Verfügbarkeit und Lizenzen von Entwicklungswerkzeugen
  - „Tooling“ bestimmt Arbeitseffizienz mit Programmiersprache
- Verfügbarkeit und Lizenzen von Bibliotheken und Frameworks
  - Bibliotheken und Frameworks sind maßgeblich für Wiederverwendung

Typische Einsatzgebiete von Programmiersprachen

Programmiersprache	Einsatzgebiet
C++	Mobile Anwendungen, Spiele, Embedded Systems, Scientific Computing, Smartcards
C	Embedded Systems
Java	Enterprise-Applikationen, Webapplikationen, Smartcards, Mobile Applikationen
C# (bzw. weitere .NET Sprachen)	Ähnlich wie Java
FORTRAN	Finanzapplikationen, Scientific Computing
BASIC (Visual Basic, RealBasic, ...)	Client-Anwendungen
Skriptsprachen (Ruby, Perl, Python, PHP, ...)	Webapplikationen, Systemadministration, Prototyping
COBOL	Hauptsächlich in Legacy-Systemen zwecks Wartung, kaum Neuentwicklungen

Quelle: Gamma et al. (2010)

IDE als Cockpit des Entwicklers

A software development tool for **creating applications**, such as desktop and web applications. IDEs blend **user interface design** and layout tools with **coding and debugging tools**, which allows a developer to easily link functionality to user interface components.

Definition IEEE-Standard 610.12 (1990)

Features einer IDE?

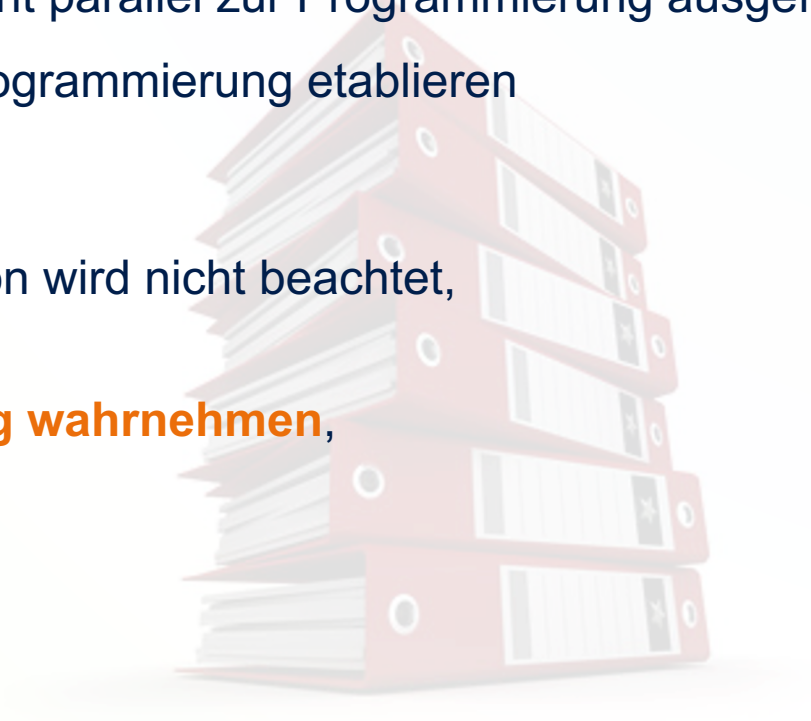
- Software-Projektverwaltung
- Quelltext-Editor für ausgewählte Programmiersprachen
- Syntax-Highlighting, Formatierung und Code Completion
- Editor für User Interface Design
- Compiler/Interpreter
- Debugger und Profiler
- Build Tools und Dependency Management (z.B. Maven, Gradle, Ant)
- Versionsmanagement (z.B. Subversion, Git)
- Refactoring
- Testing und Continuous Integration
- Umfangreiche Suchfunktionen
- Customizing und Erweiterung durch zusätzliche Plug-ins

Bibliotheken und Frameworks

Bibliothek	Framework
Wiederverwendbare Funktionalität	Wiederverwendbares Verhaltensmuster (Anwendungsskelett)
Aufruf der Bibliothek aus eigenem Code	Framework ruft eigenen Code auf (*Inversion of Control*)
Beispiele: - Apache Commons Math - Hibernate, EclipseLink - Log4J - Google Charts, JFreeCharts - XStream, Google GSON - Facebook SDK	Beispiele: - Ruby on Rails - Play Framework, Grails, JSF - Zend, Laravel, CakePHP - Django (Python) - Apache Hadoop - JUnit

Maßvolle Dokumentation

- Zu wenig Dokumentation
  - Ziel: Sicherstellung der **Wiederverwendung und Wartbarkeit**
  - Ohne Dokumentation werden Dritte angebotene Schnittstellen nicht nutzen
  - Dokumentation **wird nicht nachgeholt**, wenn nicht parallel zur Programmierung ausgeführt
  - Dokumentation **als begleitende Tätigkeit** der Programmierung etablieren
- Zu viel oder veraltete Dokumentation
  - Ausgedehnte und widersprüchliche Dokumentation wird nicht beachtet, keine Selbstverständlichkeiten erläutern
  - Entwickler sollen Dokumentation **nicht als Zwang wahrnehmen**, sondern **Zweck der Tätigkeit** erkennen



Intern- und extern-gerichtete Quelltext-Dokumentation

» **Intern:**

› Kommentare, die sich an das Projektteam
zwecks Nachvollziehbarkeit bei zukünftigen Änderungen richten

```
// Einzeiliger Kommentar

/*
 * Mehrzeiliger
 * Kommentar
 */
```

Intern- und extern-gerichtete Quelltext-Dokumentation

- **Extern:**
  - Kommentare an Außenstehende gerichtet, insb. Nutzer von Schnittstellen, zwecks richtiger Verwendung der entwickelten Software
  - Verarbeitung durch automatische Dokumentationsgeneratoren, z.B. JavaDoc

```java
/**
 * Dokumentationskommentar fuer JavaDoc
 * @author maltepriess
 * @version 1.0
 */
public class UMLClassDiagramm01 {
```

java.lang.Object
com.priess.UMLClassDiagramm.UMLClassDiagramm01

```
public class UMLClassDiagramm01
extends java.lang.Object
```

Dokumentationskommentar fuer JavaDoc

Version:

1.0

Author:

maltepriess

Constructor Summary

Constructors

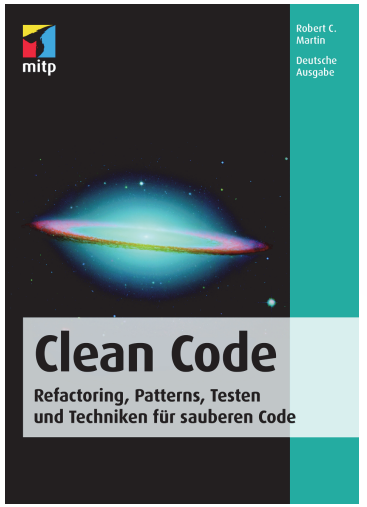
Konsens über gemeinsame Software-Entwicklung

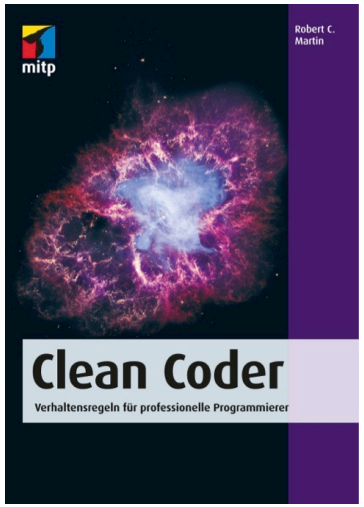
- Lokales Tooling (Arbeitsplatz) mit möglichst wenig Varianten
  - Versionen von Betriebssystem, IDE, …
- Zentrales Tooling (Server) einheitlich
  - Versionsmanagement, Build Tools, Cont. Integration, Cont. Delivery, Cont. Deployment, …
- Zuweisung und Nachverfolgung von Arbeitspaketen (Ticket-System)
- Ablauf von Besprechungen
- Coding Conventions
  - Formatierungsrichtlinien
  - Benennung der Bezeichner
  - Richtlinien für Dokumentation und Logging
  - Standards für Internationalisierung, Exception-Handling u.ä.
- Code Ownership

Clean Architecture, Clean Code, Clean Coder

- Robert C. Martin („Uncle Bob“)
  - Clean Architecture: Das Praxis-Handbuch für professionelles Softwaredesign, Regeln und Paradigmen für effiziente Softwarestrukturen / Clean Architecture: A Craftsman's Guide to Software Structure and Design
  - Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code / Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin)
  - Clean Coder: Verhaltensregeln für professionelle Programmierer / The Clean Coder: A Code of Conduct for Professional Programmers

- https://cleancoders.com





- Clean Architecture
  - Praktische Lösungen für den Aufbau von Softwarearchitekturen.
  - Allgemeingültige Regeln für die Verbesserung der Produktivität in der Softwareentwicklung über den gesamten Lebenszyklus.
  - Wie Softwareentwickler wesentliche Prinzipien des Softwaredesigns meistern, warum Softwarearchitekturen häufig scheitern und wie man solche Fehlschläge verhindern kann.

- Clean Code
  - Guten Code schreiben und schlechten Code überarbeiten.
  - Welche Prinzipien, Patterns und Praktiken sind anzuwenden, um sauberen Code zu schreiben?
  - Wie kann schlechter Code in guten Code umgewandelt werden?
  - Saubere Fehlerbehandlung sowie die Anwendung sauberen Codes während der Testphase.

- Clean Coder
  - Es geht um Disziplinen, Techniken, Tools und Methoden.
  - Mit Konflikten, knappen Zeitplänen und unvernünftigen Managern umgehen.
  - Beim Programmieren im Fluss bleiben und Schreibblockaden überwinden.
  - Mit unerbittlichem Druck umgehen und Burnout vermeiden.
  - Zeitmanagements optimieren.
  - Für Umgebungen sorgen, in denen Programmierer und Teams wachsen und sich wohlfühlen.
  - Wann Sie Nein sagen sollten und wie Sie das anstellen.
  - Wann Sie Ja sagen sollten und was ein Ja wirklich bedeutet.
  - ...

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Implementierung und Test

DevOps

Prof. Dr. Malte Prieß

Lernziele

- Best Practices: Implementierung und Test
- DevOps
- Konfigurationsmanagement
- Continuous Integration, Continuous Delivery und Continuous Deployment
- Testing
  - Komponententests, Systemtests, Abnahmetests
  - Test-Driven Development (TDD)
  - Komponententests mit Hilfe des JUnit-Frameworks

- Kunstwort aus den Begriffen **Development** (Entwicklung, *Dev*) und **IT Operations** (IT-Betrieb, *Ops*).
- Ziele: Durch gemeinsame Anreize, Prozesse und Software-Werkzeuge eine **effektivere und effizientere Zusammenarbeit der Bereiche *Dev*, *Ops* und Qualitätssicherung** ermöglichen, um
  - eine Verbesserung der Qualität der Software,
  - eine Erhöhung der Geschwindigkeit der Entwicklung und der Auslieferung sowie
  - eine Verbesserung des Miteinanders der beteiligten Teams
- zu erreichen.
- Der *DevOps*-Gedanke kann auch als eine **bereichsübergreifende, unternehmensweite Kollaboration** der Manager, Entwickler, Tester und Administratoren unter Einbeziehung der Kunden verstanden werden.
- Alle Beteiligten arbeiten zusammen an der Erreichung des gemeinsamen Ziels: der schnellen Bereitstellung einer qualitativ hochwertigen Software für den Kunden.

- Erfordert Umgang mit ggf. neuen, **bereichsübergreifende Key Performance Indicators** (KPIs) und damit **gemeinsame Anreiz-Metriken**.
- Erfordert Umgang mit **Infrastructure as code**, **Versionsverwaltung** und **automatisierten Tests**.
- **Viele stabile Releases** sollen ermöglicht werden. Dazu ist die Entwicklung einer verbesserten (agilen) **Zusammenarbeit von Entwicklern und IT-Betrieb** notwendig.
- **Verstärkte Automatisierung** von *Dev*- und *Ops*-Aufgaben.
- Automatisiert ablaufen sollen zum Beispiel der **Build** aus dem Repository, **statische und dynamische Code-Analysen** sowie Unit-, Integrations-, System- und Performance-**Tests**.
- Ein kontinuierliches, möglichst **automatisiertes Monitoring** überwacht die sogenannte **Deployment Pipeline**.

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Implementierung und Test

Konfigurationsmanagement

Prof. Dr. Malte Prieß

Lernziele

- Best Practices: Implementierung und Test
- DevOps
- Konfigurationsmanagement
- Continuous Integration, Continuous Delivery und Continuous Deployment
- Testing
  - Komponententests, Systemtests, Abnahmetests
  - Test-Driven Development (TDD)
  - Komponententests mit Hilfe des JUnit-Frameworks

Motivation

“Software engineering is multi-person construction of multi-version software.“

Parnas, 1974

- Gestern lief das System noch. Was wurde geändert?
- Handelt es sich bei diesem System um die fehlerhafte Version des Systems?
- Wie ist der Zustand dieses Elements? Wurde es bereits getestet?
- Welche Änderungen wurden im Vergleich zur Vorgängerversion durchgeführt?
- Diesen Fehler habe ich schon vor Wochen korrigiert. Warum taucht er jetzt wieder auf?
- Wer hat das geändert?

Ziele des Konfigurationsmanagement

- Koordination verschiedener Versionen von Software-Komponenten
- Reduzierung des Aufwands bei Änderungen eines komplexen Systems
- Keine Einschränkung der Agilität, sondern Vermeidung von Chaos

“Software configuration management is the discipline of managing the evolution of large and complex software systems.“

Tichy, 1988

Tichy: Tools for software configuration management, Intl. Workshop on Software Version and Configuration Control (SCM), Grassau, 1988.

Source Code Management  

- Jeder Entwickler hat Zugriff auf aktuellen Entwicklungsstand
- Jeder Entwickler kann eigene Änderungen als unfertige Version mit anderen teilen

Build Management   

- Jeder Entwickler kann mit wenig Aufwand das System oder Teilkomponenten davon bauen und ausführen, inkl. automatisierter Auflösung von Abhängigkeiten

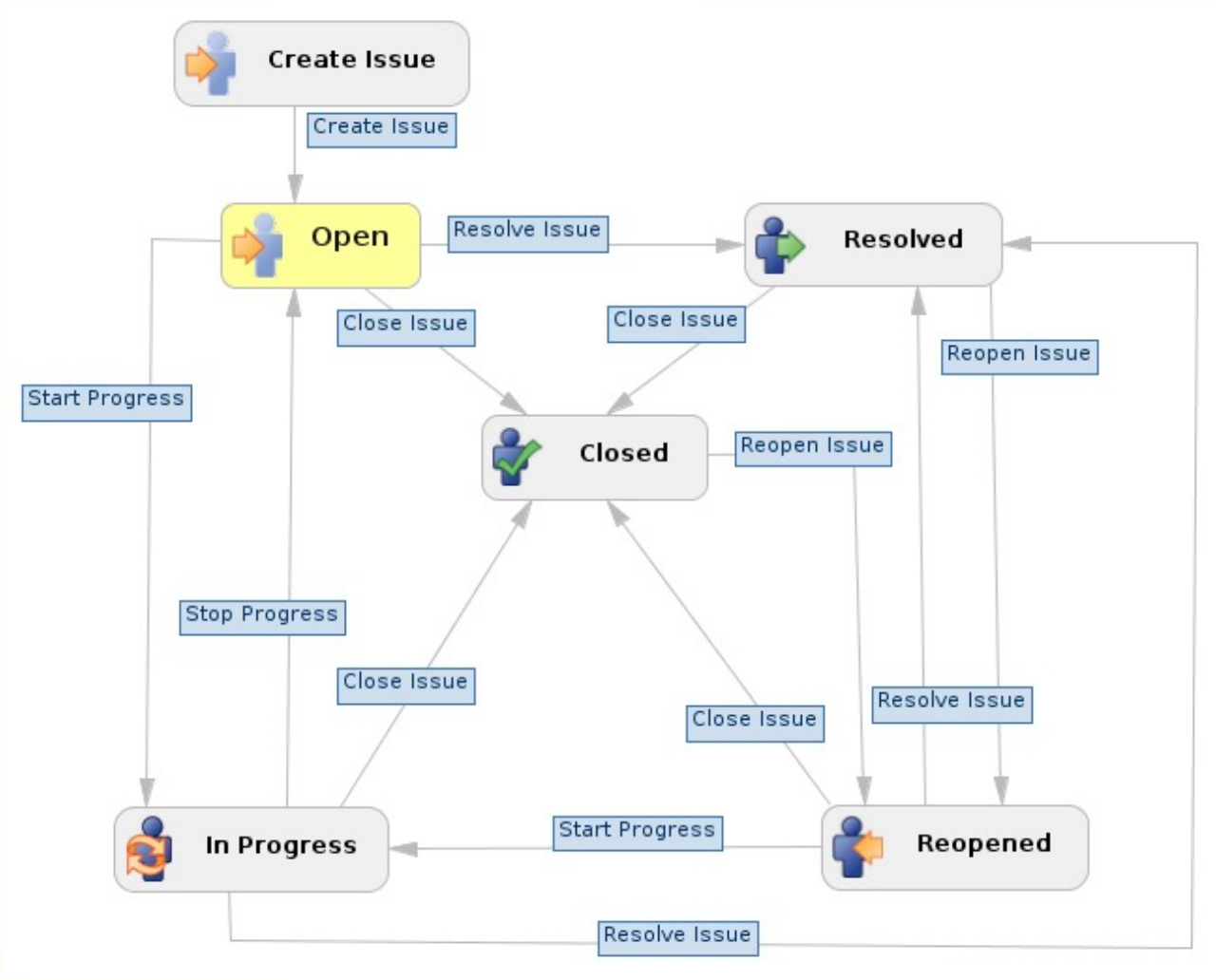
Continuous Integration   

- Damit die Teilkomponenten des Systems miteinander funktionieren, werden kontinuierlich unfertige Versionen gebaut und automatisiert getestet
- Änderungen einer Komponente, die zu Fehlern im System führen, werden transparent dargestellt

Software Project Management

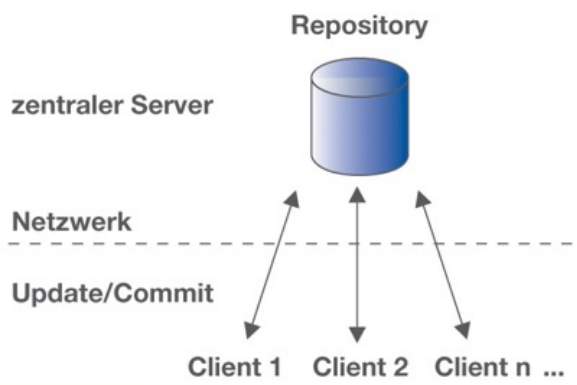
- Abbildung von Projektstrukturen (Komponenten, Meilensteinen, Versionen, etc.)
- Issue-Tracking
  - Erfassung, Klassifikation und Zuweisung von Arbeitspaketen
  - Dokumentation der Bearbeitung und Erledigung von Arbeitspaketen
- Integration mit anderen Werkzeugen, z.B. Source Code Management (bspw. Atlassian: BitBucket + JIRA)
- Auswertungen für Projekt-Controlling

JIRA Default Issue-Lifecycle

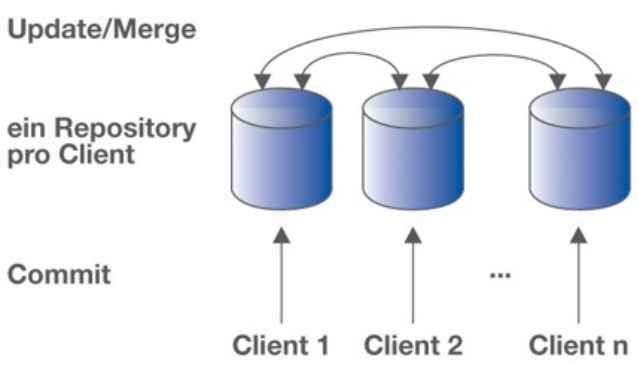


Versionsverwaltung

» **Zentral** (z.B. Subversion) oder **dezentral** (z.B. Git)



Zentrale Versionsverwaltung

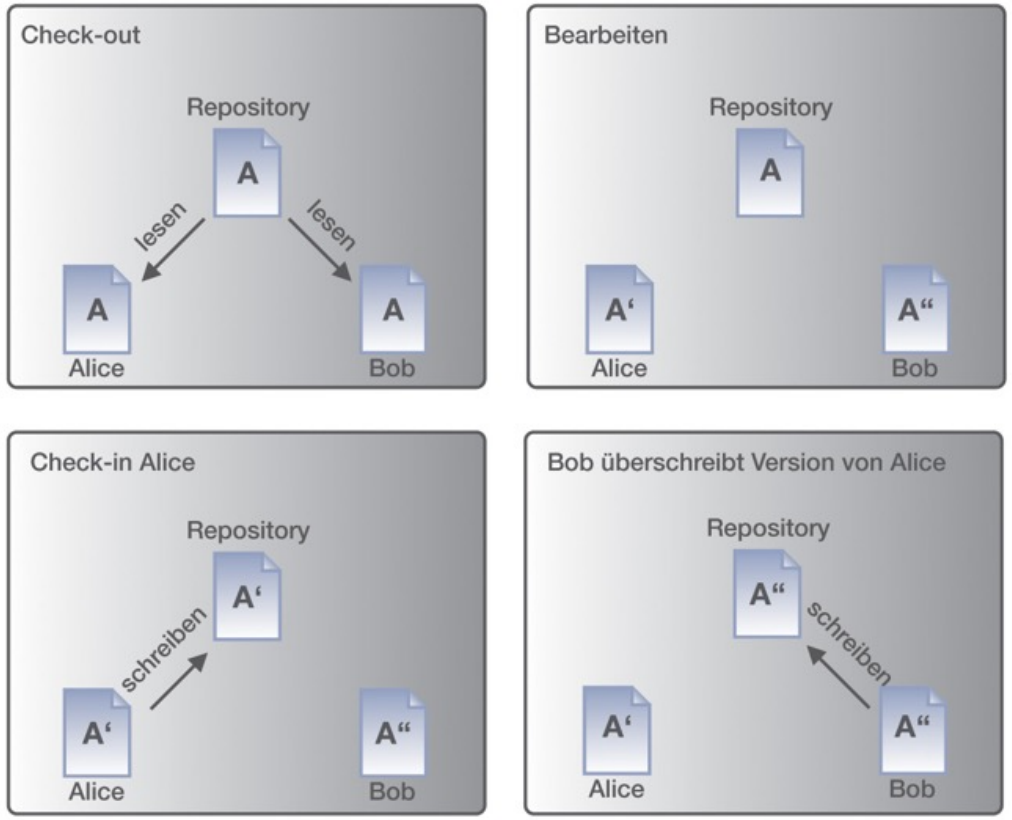


Dezentrale Versionsverwaltung

Quelle: Gamma et al. (2010)

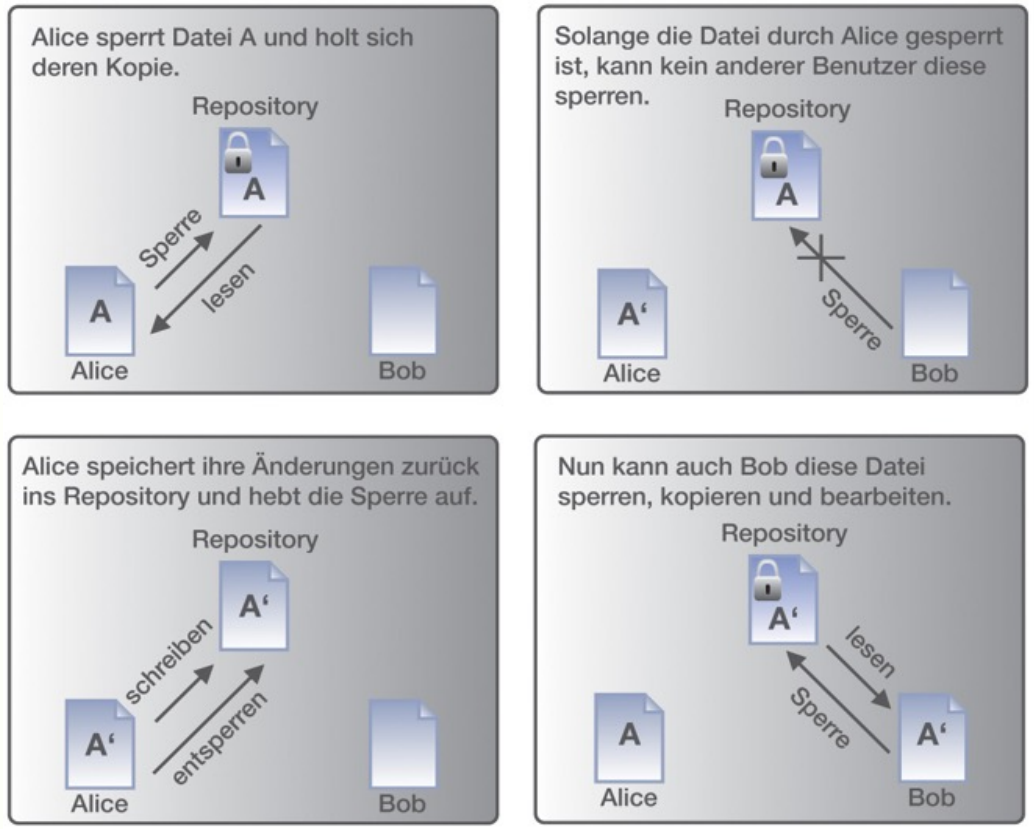
Versionsverwaltung

» Anforderung an Protokoll: Vermeiden von Lost Updates



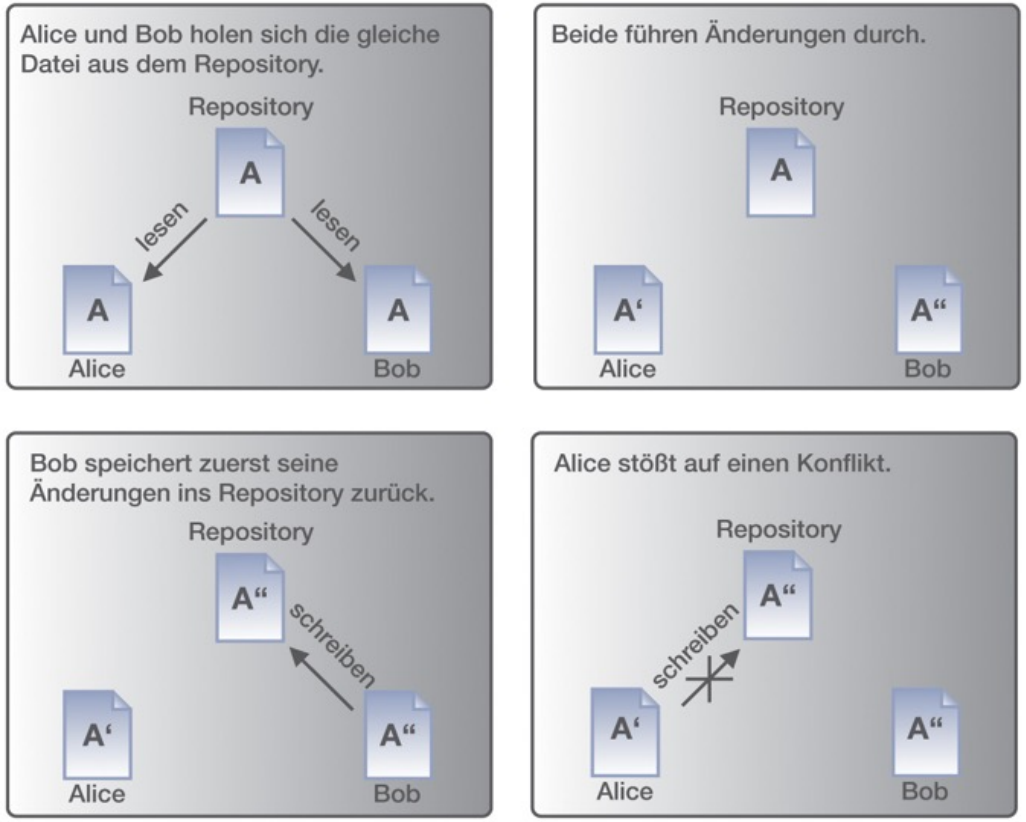
Versionsverwaltung

» Strategie ***Lock-Modify-Unlock***: Sperren verhindern Lost Updates



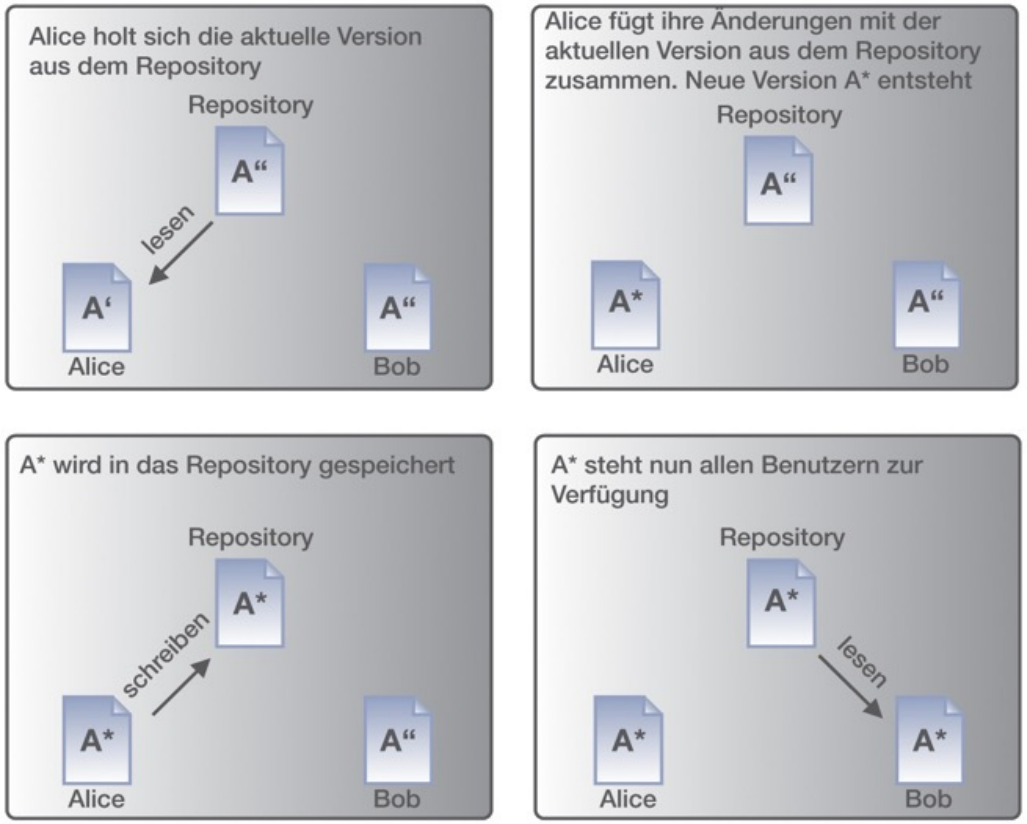
Versionsverwaltung

» Strategie ***Copy-Modify-Merge***: Keine Sperren, sondern Konflikterkennung



Versionsverwaltung

- Strategie ***Copy-Modify-Merge***: Keine Sperren, sondern Konflikterkennung
- Nachrangiger Commit muss Konflikt (ggf. manuell) auflösen



Protokoll

- `svn`: Bearbeiten von Repository-Inhalten

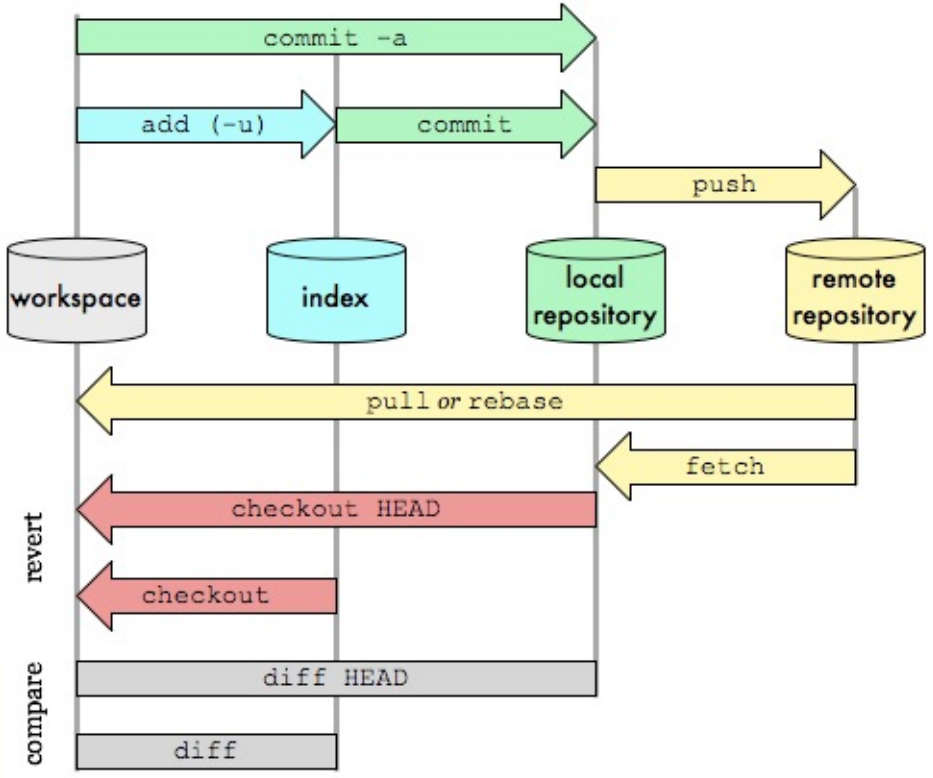
  - `add` - Schedule a new file or directory (including contained files) for inclusion in the repository
  - `checkout`, `co` - Create a local working copy of a remote repository
  - `commit`, `ci` - Commit (check in) local changes to the repository
  - `copy`, `cp` - Copy one or more files in a working copy or in the repository
  - `delete`, `del`, `remove`, `rm` - Items specified are scheduled for deletion upon the next commit. Working copy files not yet committed are deleted immediately.
  - `diff`, `di` - Displays differences in files from the directory
  - `help`, `?`, `h` - Subversion help and help on sub-commands
  - `move`, `mv`, `rename`, `ren` - Moves files or directories in your working copy or repository
  - `resolve` - Resolve conflicts on working copy files or directories
  - `revert` - Undo all local edits or optionally a file or directory
  - `status` - Print the status of working copy files and directories
  - `update` - Bring changes from the repository into your working copy

- `svnadmin`: Erstellen, Verändern, Reparieren von Repositories
- `svnsync`: Inkrementelles Spiegeln von Repositories

Tutorial für den Einstieg: http://openoffice.apache.org/svn-basics.html

Dezentrale Versionsverwaltung mit Git

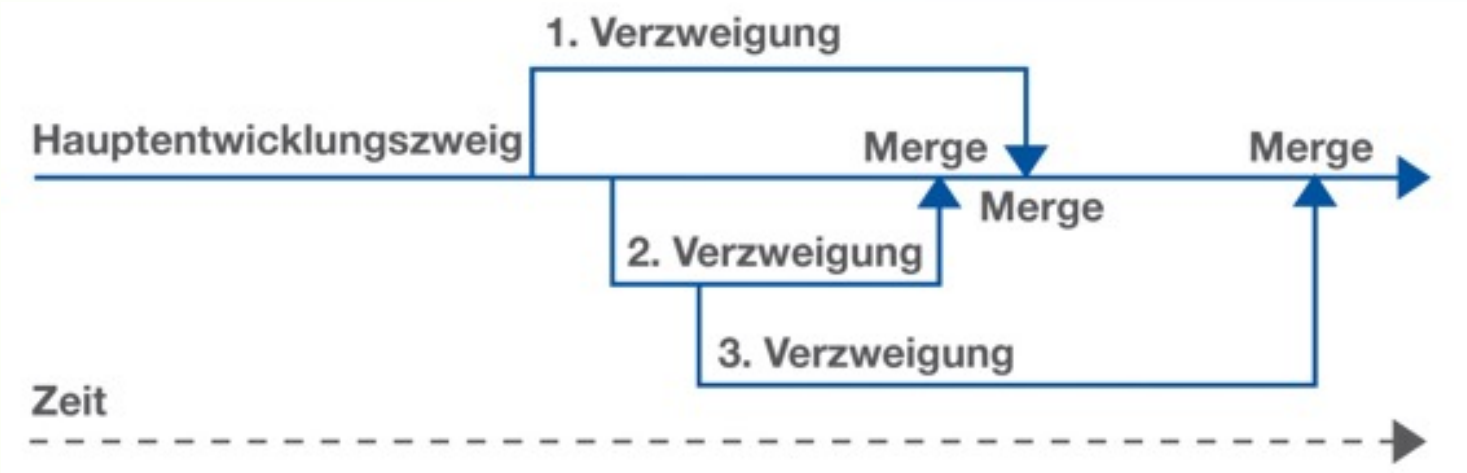
- Entwicklung seit 2005, zunächst zur Versionierung des Linux-Kernels
- GitHub: Populärer Hosting-Service, kostenfrei für Open Source-Projekte



Linus Torvalds: *"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'."*

Branching and Merging

- **Branches** werden für Projekte oder Releases eröffnet
- Branches müssen abschließend in den **Hauptzweig** (**Main, Trunk**) integriert werden (**Merge**)



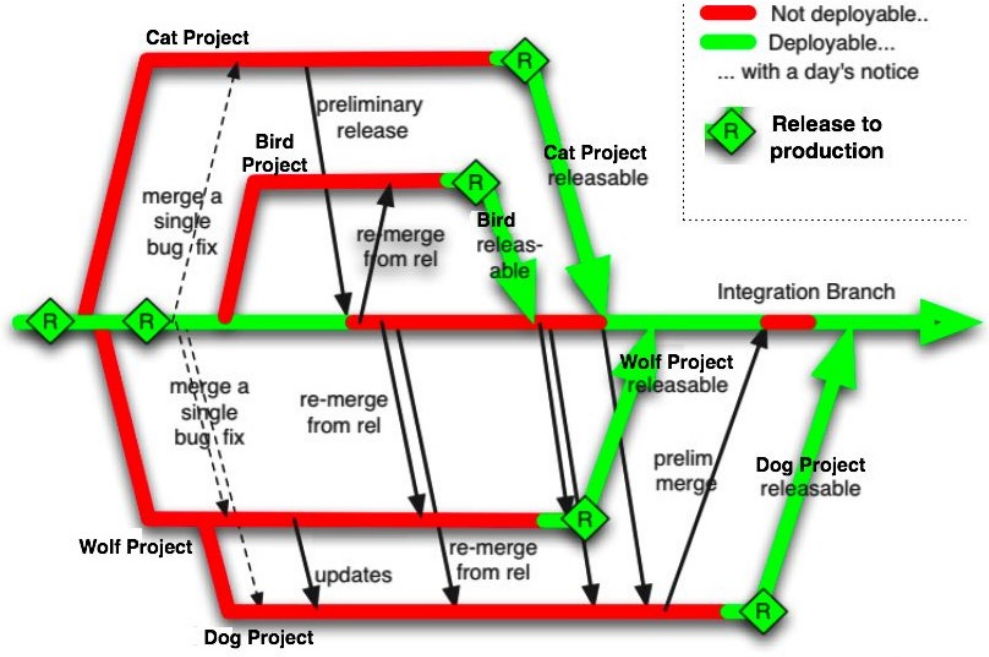
- Je länger ein Branch nicht synchronisiert wird, desto aufwändiger ist das Merging
- Viele offene Branches erhöhen die Komplexität

Work on Main (Strategie 1)

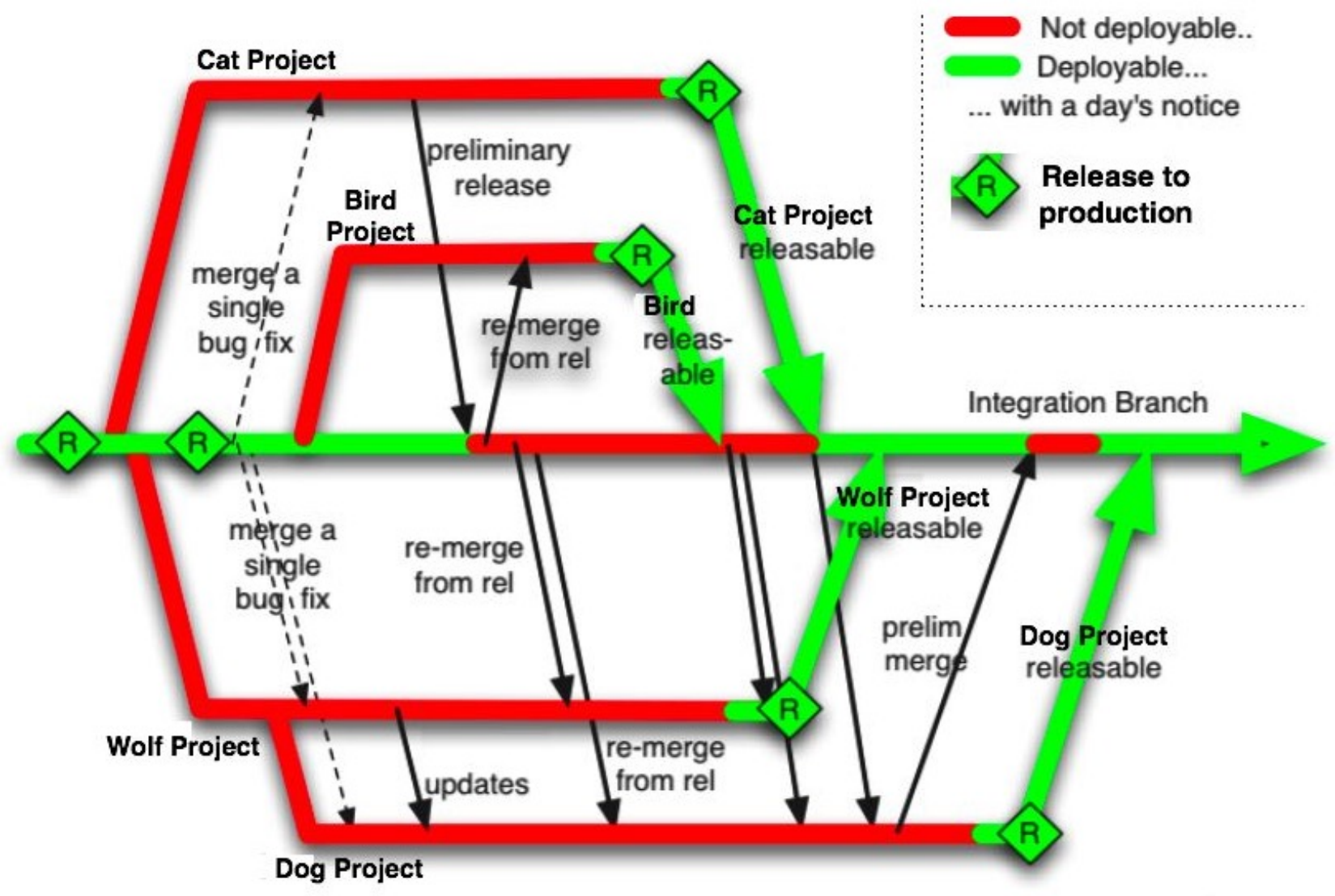
+ Code ist umgehend integriert, Tests beziehen sich auf integrierten Code
+ Alle Entwickler tauschen ihre Änderungen laufend untereinander aus
+ Merge-Aufwand zum Ende eines Projekts wird vermieden
– *Breaks* (bedingt durch Commits, die zu Build-Fehlern führen) wirken sich auf alle Entwickler aus
– *Commit early, commit often*: Freigabe unfertiger Code-Fragmente in Main Branch ist nicht geeignet, exploratives Vorgehen ist eingeschränkt
– Führt zu vielen lokalen Tests (um keine Breaks zu verursachen)

Branch for Project/Feature (Strategie 2)

+ Feature wird in Branch entwickelt ohne Entwickler anderer Projekte zu stören
+ Geeignet für Open Source-Projekte ohne betrieblichen Zeitdruck und mit wenigen, sehr kompetenten Entwicklern
– Hoher Merge-Aufwand am Projektende
– oder *Instable Main*, wenn während des Projekts unfertiger Code integriert wird

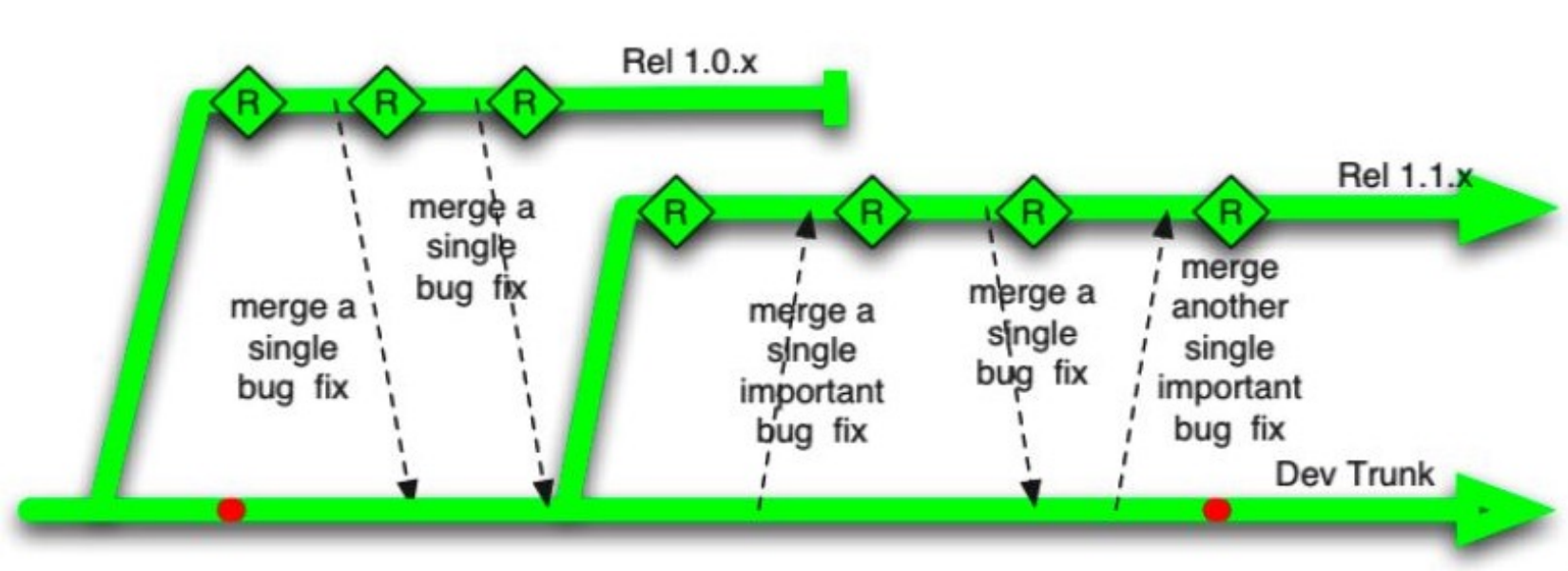


Branching and Merging



Branch for Release (Strategie 3)

+ Branch wird kurz vor Release erzeugt, finale Tests beziehen sich auf Branch
+ Weiterentwicklung in Main möglich → Kein *Code Freeze* mehr erforderlich
– Bugfixes im Release-Branch müssen möglichst schnell in Main integriert werden



DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Implementierung und Test

Continuous Integration, Continuous Delivery und Continuous Deployment

Prof. Dr. Malte Prieß

Lernziele

- Best Practices: Implementierung und Test
- DevOps
- Konfigurationsmanagement
- Continuous Integration, Continuous Delivery und Continuous Deployment
- Testing
  - Komponententests, Systemtests, Abnahmetests
  - Test-Driven Development (TDD)
  - Komponententests mit Hilfe des JUnit-Frameworks

*Continuous Integration is a software development practice where members of a **team integrate their work frequently**, usually each person integrates at least daily – leading to multiple integrations per day.*

*Each integration is **verified by an automated build (including test)** to detect integration errors as quickly as possible.*

Martin Fowler

Quelle: http://martinfowler.com/articles/continuousIntegration.html

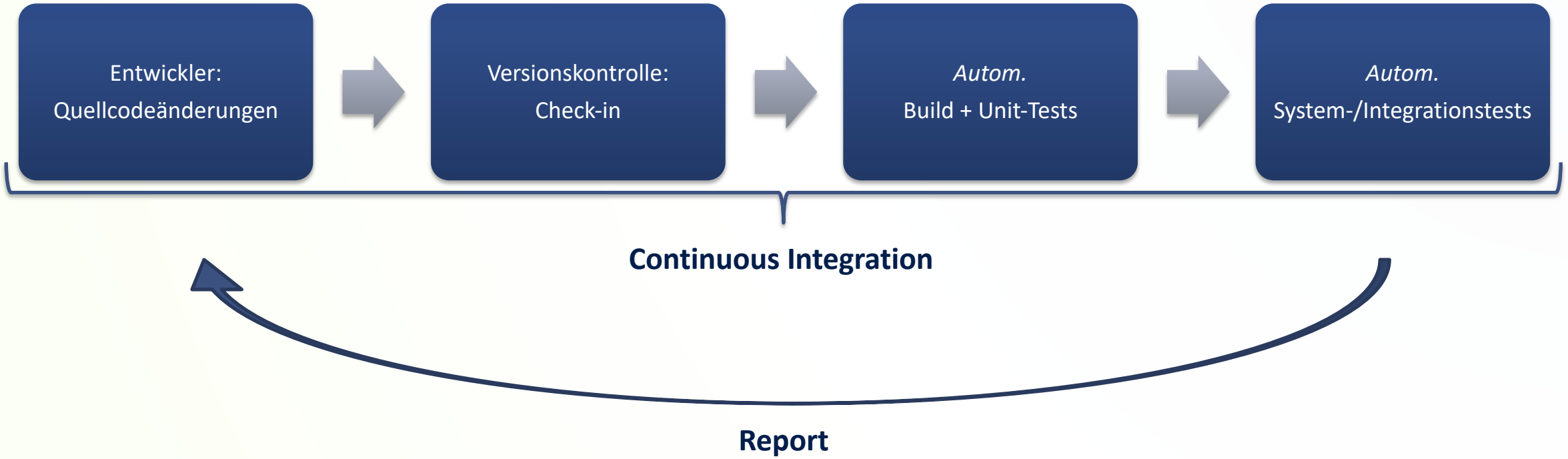
Voraussetzungen für Continuous Integration

- Versionsverwaltung
- Automatischer Build
- Zustimmung des Teams, Teilergebnisse regelmäßig einzuchecken

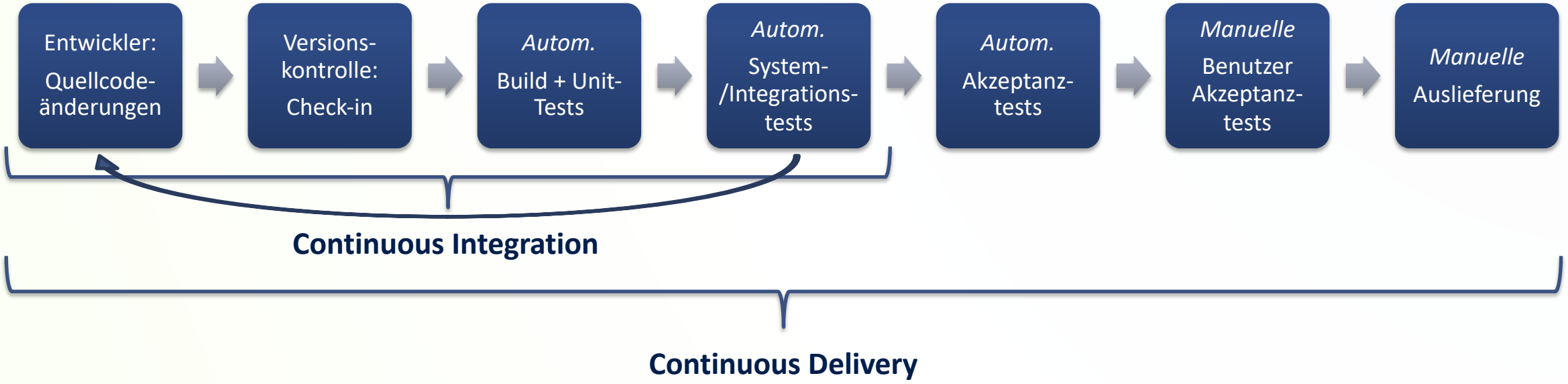
Continuous Integration Best Practices

- *Check In Regularly*
- *Create a Comprehensive Automated Test Suite*
- *Keep the Build and Test Process Short*
- *Manage Your Development Workspace*

Continuous Integration



Typische Deployment Pipeline – von der Entwicklungs- zur Produktionsumgebung

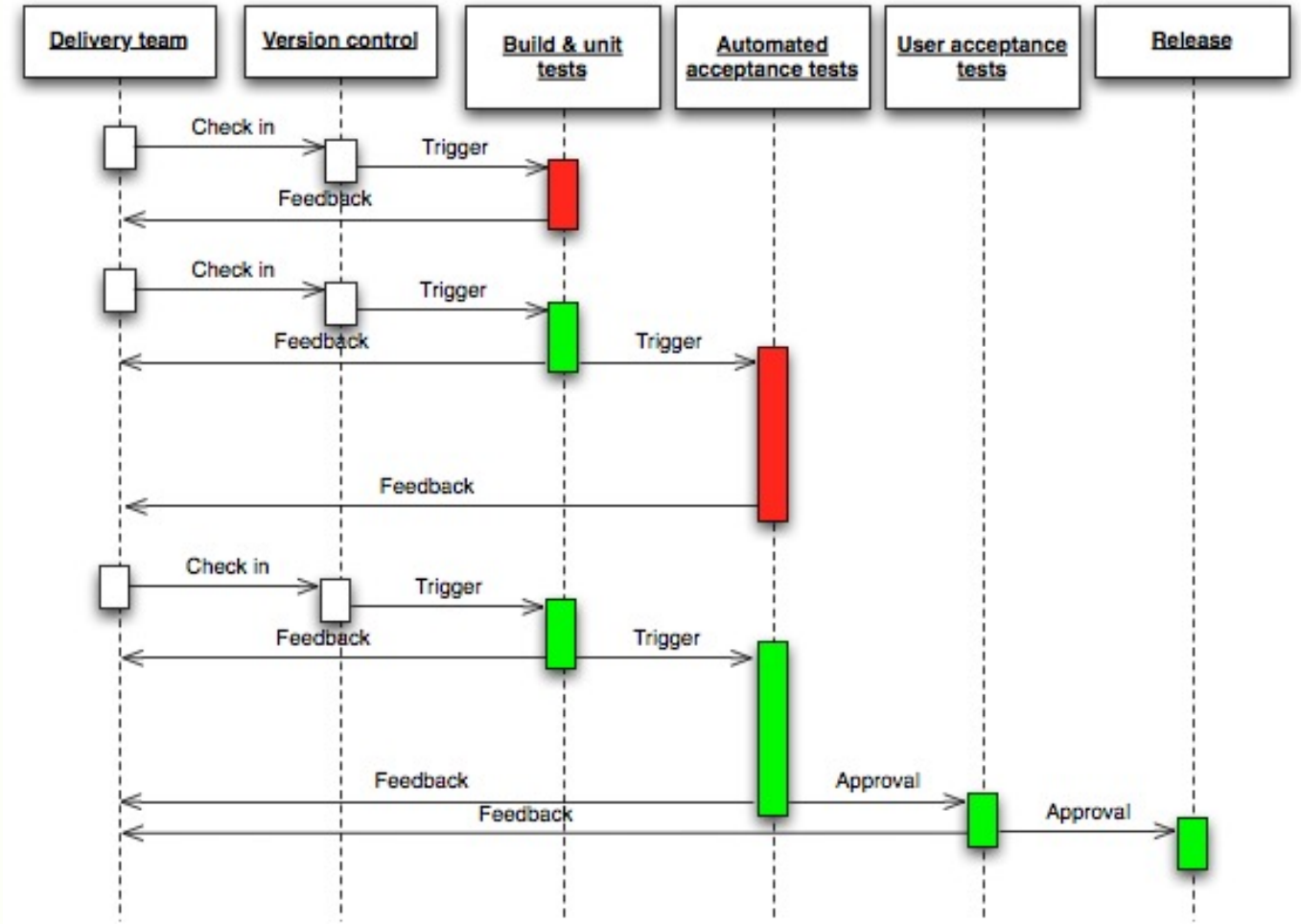


Continuous Delivery
is a software development discipline where you build software in such a way that the ***software can be released to production at any time****.*

Martin Fowler

Humble, Read, North: The Deployment Production Line, Proceedings of the conference on AGILE 2006, 2006. Quelle: http://martinfowler.com/bliki/ContinuousDelivery.html

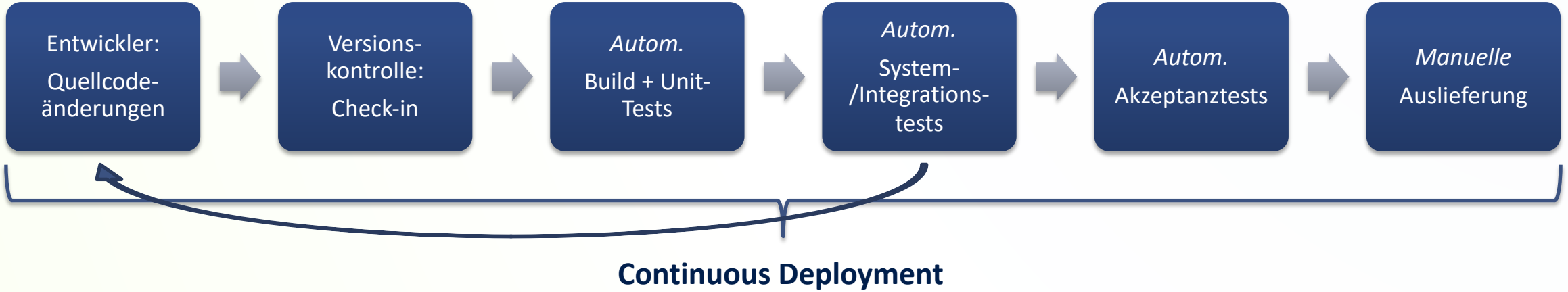
Continuous Delivery



Humble, Farley: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 2010.

Continuous Deployment:

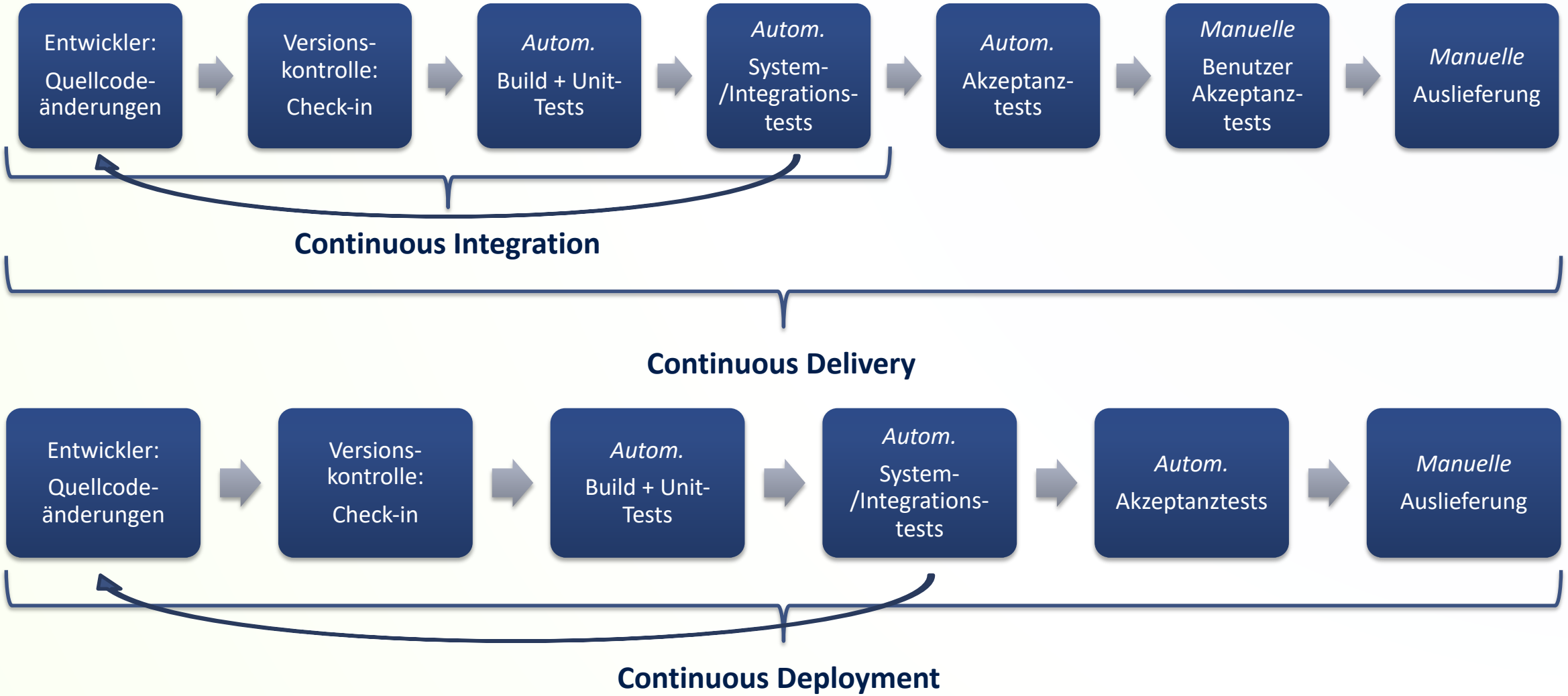
Zielt auf die **vollständige Automatisierung** der Deployment Pipeline ab.



Kritik

- Automatisiertes Deployment in die Produktionsumgebung häufig nicht erwünscht (Unternehmensleitung und Anwender)
- Manuelle User-Akzeptanztests zur Qualitätssicherung unersetzlich (z.B. durch interne fachliche Tester)

Cont. Integration, Cont. Delivery, Cont. Deployment



How to rate the quality of a software team?

- Do you use source control?
- Can you make a build in one step?
- Do you make daily builds?
- Do you have a bug database?
- Do you fix bugs before writing new code?
- Do you have an up-to-date schedule?
- Do you have a spec?
- Do programmers have quiet working conditions?
- Do you use the best tools money can buy?
- Do you have testers?
- Do new candidates write code during their interview?
- Do you do hallway usability testing?

Quelle: http://www.joelonsoftware.com/articles/fog0000000043.html

12 Fragen an Unternehmen, die zeitgemäß Software entwickeln?

- Verwenden Sie ein Werkzeug zur Quellcodeverwaltung?
- Können Sie einen Build in einem Schritt erstellen?
- Erstellen Sie tägliche Builds?
- Haben Sie eine Fehlerdatenbank?
- Beheben Sie Fehler, bevor Sie neuen Code schreiben?
- Führen Sie einen aktuellen Terminplan?
- Haben Sie Spezifikationen?
- Haben die Programmierer eine ruhige Arbeitsumgebung?
- Verwenden Sie die besten auf dem Markt erhältlichen Tools?
- Haben Sie Tester?
- Müssen neue Kandidaten beim Vorstellungsgespräch Code schreiben?
- Verwenden Sie Usability-Tests?

Quelle: http://www.joelonsoftware.com/articles/fog0000000043.html

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Implementierung und Test

Testing

Prof. Dr. Malte Prieß

Lernziele

- Best Practices: Implementierung und Test
- DevOps
- Konfigurationsmanagement
- Continuous Integration, Continuous Delivery und Continuous Deployment
- Testing
  - Komponententests, Systemtests, Abnahmetests
  - Test-Driven Development (TDD)
  - Komponententests mit Hilfe des JUnit-Frameworks

Error, Fault, Failure

Begriff	Definition	Vermeidung/Erkennung
***Error**, Mistake* (Fehlhandlung, Versagen, Irrtum)	Fehlerhafte Aktion einer Person, die zu einer fehlerhaften Programmstelle führt. **Eine Person macht einen Fehler.**	Schulung, Konvention, Prozessverbesserung
***Fault**, Defect, Bug* (Defekt, Fehlerursache)	Fehlerhafte Anweisung o.ä. eines Programms, die ein Fehlverhalten auslösen kann. **Die Anwendung enthält einen Fehler.**	Inspektion, Review, Programmanalyse
Failure (Fehlverhalten, Fehlerwirkung)	Fehlverhalten eines Programms gegenüber der Spezifikation, das zur Laufzeit tatsächlich auftritt. **Die Anwendung zeigt einen Fehler.**	(Automatisierte) Tests (Komponententest, Systemtest/Integrationstest, Akzeptanztest)

Komponententests (Unit-Tests)

- Einzelne Komponenten/Module werden unabhängig voneinander getestet
- Komponenten sind zusammengehörige Datenstrukturen und Funktionen

Systemtest/Integrationstest

- Test des gesamten zusammengefügten Systems,
  wobei neue Funktionen besonders intensiv getestet werden sollten

Abnahmetest (Akzeptanztest)

- Test mit tatsächlichen Anwenderdaten um zu prüfen,
  dass die Anforderungen der Anwender erfüllt werden

Komponententests (Unit-Tests)

Fokus in dieser Vorlesung

- Einzelne Komponenten/Module werden unabhängig voneinander getestet
- Komponenten sind zusammengehörige Datenstrukturen und Funktionen

Systemtest/Integrationstest

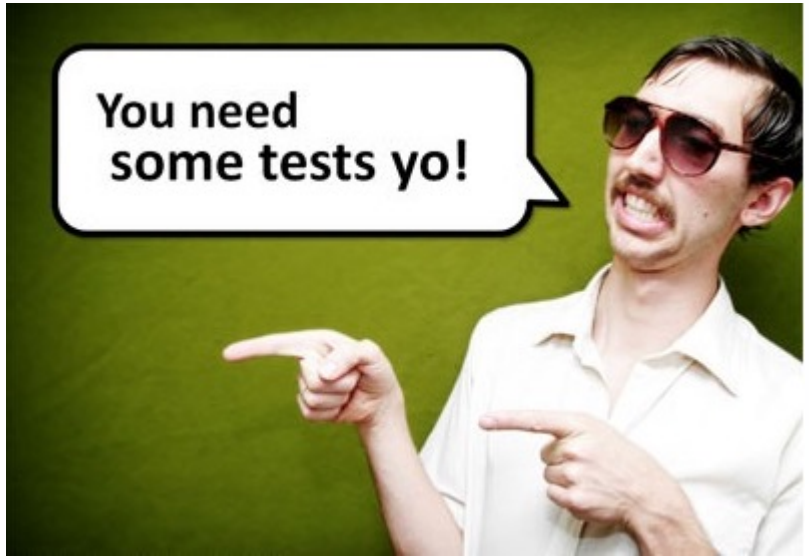
- Test des gesamten zusammengefügten Systems,
  wobei neue Funktionen besonders intensiv getestet werden sollten

Abnahmetest (Akzeptanztest)

- Test mit tatsächlichen Anwenderdaten um zu prüfen,
  dass die Anforderungen der Anwender erfüllt werden

Komponententest (Unit-Test)

- Überprüft die Funktionalität einer Anwendungseinheit (*Unit*)
- Wird kurz vor oder kurz nach Implementierung der Unit erstellt
- Automatisiert durchführbar!
- Erlaubt sofortige Rückmeldung im Fehlerfall
- Integriert in Continuous Integration-Prozess (*Nightly Build*)



Black-Box- und White-Box-Testing

Black-Box-Testing

- Funktionsorientiert, Implementierung/Funktionsweise ist unbekannt.
- Ziel ist es, die Übereinstimmung eines Softwaresystems mit seiner Spezifikation zu überprüfen.
- Tests werden anhand der Spezifikation/Anforderung entwickelt.
- Nur nach außen sichtbares Verhalten fließt in den Test ein.
- Meist von fachlich orientierten Testern oder speziellen Test-Abteilungen/Test-Teams entwickelt, nicht von Software-Entwicklern.

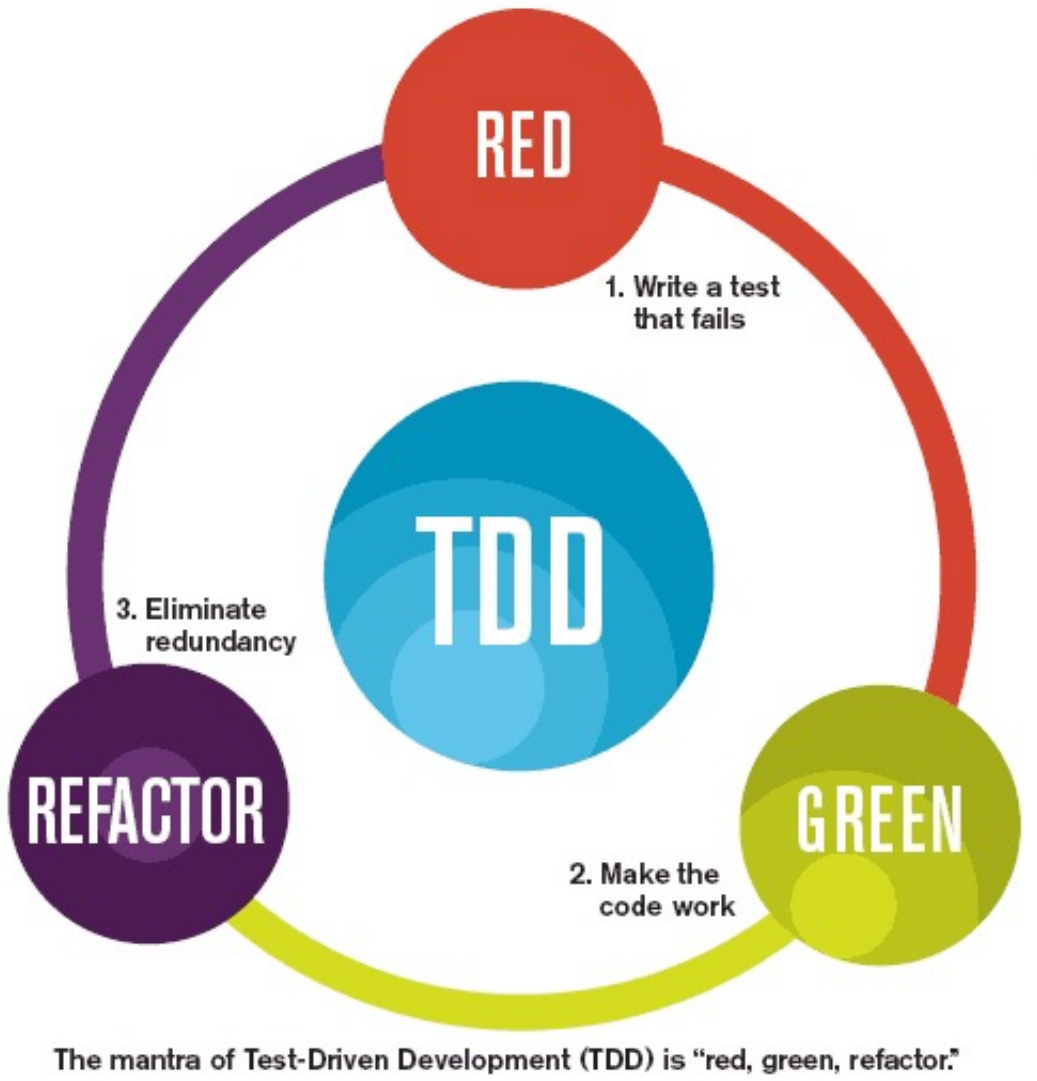
White-Box-Testing

- Strukturorientiert, Implementierung ist bekannt.
- Tests werden mit Kenntnissen über die innere Funktionsweise entwickelt.
- Es wird am Code geprüft.
- Entwicklertests sind i. d. R. White-Box-Tests.
- Gefahr, dass Missverständnisse beim Entwickler zu falschen Testfällen führt.

Eine Kombination aus beiden Ansätzen ist erforderlich, um eine möglichst hohe Überdeckung der Testfälle zu erreichen.

Testgetriebene Entwicklung

- Schnittstelle wird spezifiziert
- Unit-Testfälle werden erstellt
- Unit wird implementiert
- Refactoring



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Testgetriebene Entwicklung („Test-First“)

- Bevor man eine Zeile Produktionscode schreibt, entsteht ein entsprechender Test, der diesen Code motiviert
- Es wird nur so viel Produktionscode geschrieben, wie es der Test verlangt. Mit anderen Worten: Läuft der Test, steht der Code.
- Die Entwicklung findet in kleinen Schritten statt, in denen sich Testen und Kodieren abwechseln. Eine solche „Mikro-Iteration“ dauert nicht länger als 10 Minuten.
- Zum Zeitpunkt der Integration von Produktionscode ins Gesamtsystem müssen alle Unit Tests erfolgreich laufen.

Quelle: Link, Fröhlich (2002)

JUnit

***JUnit** is a simple framework to write repeatable tests.*
*It is an instance of the **xUnit** architecture for unit testing frameworks.*

- **JUnit Usage and Idioms**
- http://junit.org/

- Link, Fröhlich: Unit Tests mit Java, 1. Aufl., dpunkt-Verl., 2002.

- **Programmiersprache gleich Testsprache:** JUnit ist ein reines Java-Framework

- **Trennung von Anwendungs- und Testcode:** Testfälle werden in einer eigenen Klassenhierarchie erstellt

- Ab JUnit 4:
  - Neuer Namensraum `org.junit.*`
  - Verwendung von Annotationen (bspw. `@Test`), um Testfälle zu kennzeichnen
- Vor JUnit 4:
  - Ableitung der Klasse `junit.framework.TestCase`
  - Verwendung von Namenskonventionen `public void test...()`

- Ausführung und Verifikation einzelner **Testfälle ist voneinander unabhängig** (JUnit 4: bspw. über `@Before` Annotation realisiert)
- Erkennen der **Testergebnisse auf einen Blick**
- Beliebige **Zusammenfassung von Tests in Testsuiten**:
  - Vor JUnit 4: `junit.framework.TestSuite`
  - Nach JUnit 4: `@RunWith` und `@Suite` Annotationen

*** Übung JUnit ***

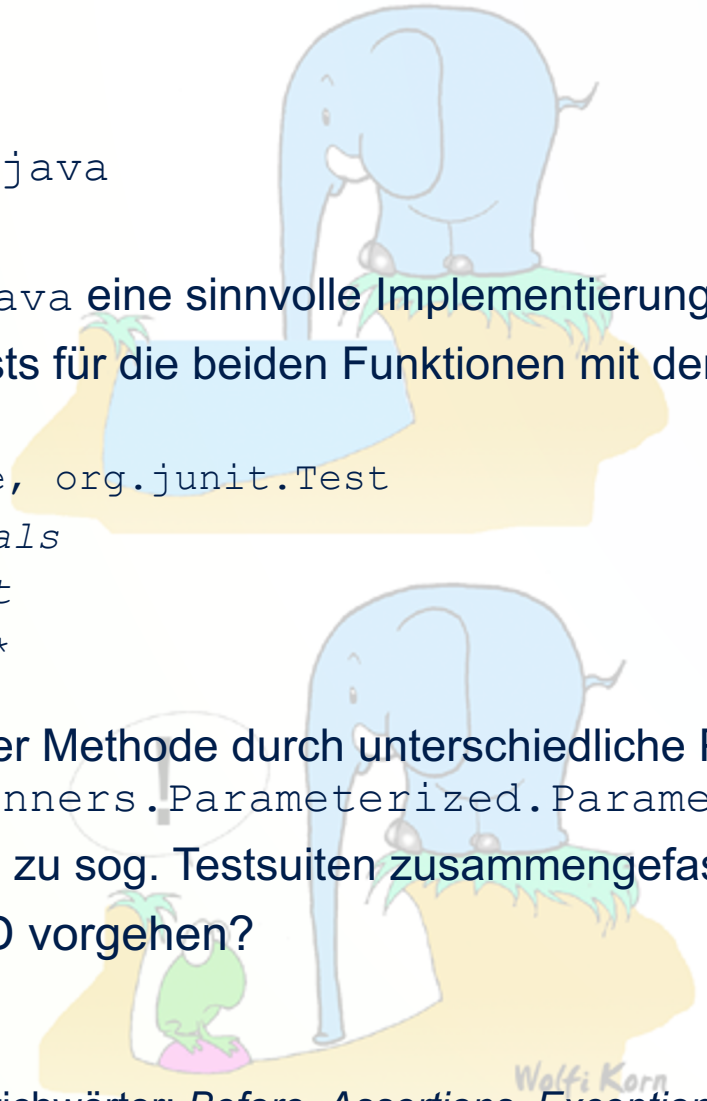
Zu testendes Interface + Klasse:

`/MyMath_JUnit/src/MyMath.java`

`/MyMath_JUnit/src/MyMathImpl.java`

Aufgaben:

1. Ergänzen Sie in `MyMathImpl.java` eine sinnvolle Implementierung des Interfaces `MyMath.java`
2. Schreiben Sie geeignete Unit-Tests für die beiden Funktionen mit dem JUnit-Framework unter Verwendung von
   - Annotationen: `org.junit.Before, org.junit.Test`
   - `org.junit.Assert.assertEquals`
   - `org.junit.Assert.assertThat`
   - `org.hamcrest.CoreMatchers.*`
3. Wie können mehrfache Tests einer Methode durch unterschiedliche Parameter realisiert werden (unter Verwendung der `org.junit.runners.Parameterized.Parameter` Annotation)?
4. Wie können mehrere Testklassen zu sog. Testsuiten zusammengefasst werden?
5. Wie würde man in (2) beim TDD vorgehen?



Hilfestellungen:

- `http://junit.org/junit4/` (Stichwörter: *Before, Assertions*, *Exception Testing*, *Matchers and assertThat*)
- Oder auch direkt über die Junit API: `http://junit.org/junit4/javadoc/latest/index.html`

*** Übung Build Management ***

1. Erstellung eines einfachen HelloWorld-Gradle-Projekts
2. Demonstration einiger Gradle-Jobs (build, test, clean, usw.)
3. Artefakte
   - jar: /build/libs/*.jar
   - html: /build/reports/…
   - xml: /build/test-results/…
4. Aufsetzen des vorherigen MyMath-Beispiels als Gradle-Projekt
5. Aufsetzen einer CI/CD Pipeline für das Gradle-Projekts (4) unter GitLab

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

★ Vielen Dank für die Aufmerksamkeit ★