HOCHSCHULE HEILBRONN

Software Engineering komplexer Systeme

Prof. Dr.-Ing. Andreas Heil

HEUTIGER INHALT

- Qualitätssicherung
  - Software Qualität
  - Testebenen
    - UI-Tests, Unit Tests, Integrationstests
  - DevOps Ansatz

Zum Einstieg und Nachlesen: Berühmte Software Fehler

Nicht angemeldet Diskussionsseite Beiträge Benutzerkonto erstellen Anmelden

WIKIPEDIA
Die freie Enzyklopädie

Artikel | Diskussion | Lesen | Bearbeiten | Quelltext bearbeiten | Versionsgeschichte | Wikipedia durchsuchen

Hauptseite
Themenportale
Zufälliger Artikel

Mitmachen
Artikel verbessern
Neuen Artikel anlegen
Autorenportal
Hilfe
Letzte Änderungen
Kontakt
Spenden

Werkzeuge
Links auf diese Seite
Änderungen an verlinkten Seiten
Spezialseiten
Permanenter Link
Seiteninformationen
Wikidata-Datenobjekt
Artikel zitieren

Drucken/exportieren
Buch erstellen
Als PDF herunterladen
Druckversion

In anderen Sprachen
English

Liste von Programmfehlerbeispielen

Die **Liste von Programmfehlerbeispielen** zeigt einige medial betrachtete Beispiele von Programmfehlern, ist nach Branchen (Anwendergruppen) geordnet und beschreibt deren Folgen.

Inhaltsverzeichnis [Verbergen]
1 Luft- und Raumfahrt
2 Medizin
3 Handel
4 Verkehr
5 Finanzwesen
6 Kommunikation
7 Militär
8 Informationstechnik
9 Fehler bei Jahreswechseln
10 Weblinks
11 Einzelnachweise

Luft- und Raumfahrt [Bearbeiten | Quelltext bearbeiten]

- Beim Kampfflugzeug F-16 brachte der Autopilot das Flugzeug in Rückenlage, wenn der Äquator überflogen wurde. Dies kam daher, dass man keine „negativen" Breitengrade als Eingabedaten bedacht hatte. Dieser Fehler wurde sehr spät während der Entwicklung der F-16 anhand eines Simulators entdeckt und beseitigt.[1]
- Am 4. Juni 1996 wurde der Prototyp der Ariane-5-Rakete mit der Startnummer V88 der Europäischen Raumfahrtbehörde eine Minute nach dem Start in vier Kilometern Höhe gesprengt. Der Programmcode für die Steuerung war von der Ariane 4 übernommen worden und funktioniert nur in einem von der Ariane 4 nicht überschreitbaren Bereich (Horizontalgeschwindigkeit). Die Steuersysteme kamen zum Erliegen, als ebendieser Bereich von der Ariane 5 überschritten wurde, da sie höhere Geschwindigkeiten erreicht als die Ariane 4. Bei der Programmierung war es zu einem Fehler bei der Typumwandlung gekommen. Als von Float nach Integer umgewandelt wurde und der Wert 32.768 erreichte, entstand ein Überlauf.[2] Dieser Überlauf hätte durch die verwendete Programmiersprache Ada eigentlich entdeckt und behandelt werden können. Diese Sicherheitsfunktionalität ließen die Verantwortlichen jedoch abschalten. Der Schaden betrug etwa 370

Quelle: https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen

BEISPIEL: THERAC-25

THERAC-25

- Weiterentwicklung von THERAC-23
- Kanadische Firma: *Atomic Energy of Canada Limited* (AECL)
- Merkmale
  - Röntgenstrahlen (*X-Rays)*
  - Elektronenstrahlen
  - Verbesserte Benutzerführung

BEISPIEL: THERAC-25

Unfall am Marietta, Georgia, USA, am 3. Juni 1985

- Patientin erhielt Dosis von 15.000 bis 20.000 rad
- Normal 200 rad
- Folge
  - Arm und Schulter nach Behandlung nicht mehr zu gebrauchen
  - Brüste mussten entfernt werden

Unfall in Hamilton, Ontario, Kanada, am 26. Juli 1985

- Frau erhält Strahlendosis im Nacken zwischen 13.000 und17.000 rad
- Folge
  - Sie starb am 3. November an Krebs

Liste fortsetzbar ....

BEISPIEL: THERAC-25

Was ist passiert?

- Falsche Anzeige der Behandlungsart
- Korrekturen wurden bei Magneteinstellung im Behandlungsraum für acht Sekunden nicht angenommen
- Vorgänger Versionen hatten hierfür eine Hardware-Absicherung
- Unklare Benutzerführung führte zur wiederholten Bestrahlung
- Fehler erst nach mühsamer Kleinarbeit rekonstruiert
- Behandlungen mit THERAC-25 wurde daraufhin eingestellt

HEUTE SIND WIR JA VIEL BESSER…

ABO SHOP AKADEMIE JOBS MEHR

E-PAPER AUDIO APPS ARCHIV ANMELDEN

ZEIT ONLINE

Suche

Politik Gesellschaft Wirtschaft Kultur Wissen Digital Campus Arbeit Entdecken Sport ZEITmagazin Podcasts mehr

Autopilot

Tod durch Software

In den USA hat erstmals der Autopilot eines Teslas einen tödlichen Unfall gebaut. Doch die Entwicklung selbstfahrender Autos dürfte das kaum aufhalten.

Von **Matthias Breitinger**

1. Juli 2016, 17:21 Uhr / 112 Kommentare

MEEDIA

MEDIEN MARKETING DATACENTER JOBS NEWS

Anzeige

08.06.2018 | 03:00 Uhr

Software-Fehler: 14 Millionen Facebook-Nutzer posteten öffentlich, ohne es zu wollen

Fotolia / Montage: MEEDIA

Das Unbehagen gegenüber Facebook wurde 2018 immer größer

Flugsteuerung

Boeing gesteht zweiten Software-Fehler bei 737 Max

Nicht nur das umstrittene MCAS-System hat offenbar Mängel. Boeing entdeckte einen zweiten Software-Fehler bei der Flugsteuerung der 737 Max.

05.04.19 - 11:18 | Felix Stoffels 96 Kommentare

Boeing 737 Max: Neues Softwareproblem. *Boeing*

Am Donnerstagabend Ortszeit (4. April) wandte sich Boeing-Chef Dennis Muilenberg in einer Videobotschaft an die Öffentlichkeit: Zum ersten Mal räumte er dabei ein, dass das umstrittene Steuerungssystem MCAS ein Auslöser für die Abstürze der beiden Boeing 737 Max von Lion Air und Ethiopian Airlines war. Erst wenige Stunden zuvor wurden im ersten Zwischenbericht aus Äthiopien neue Details zum Unglück von Flug ET302 veröffentlicht. Dabei zeigte sich, dass die Piloten der Boeing 737 Max genau nach Vorgaben

FEHLERERWARTUNG

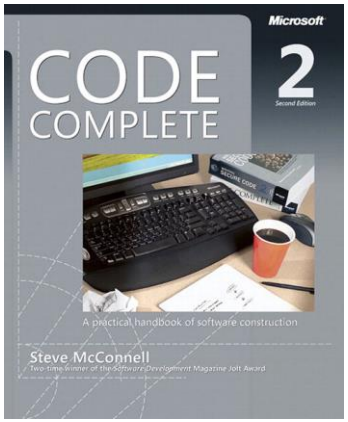
$$\text{Fehlerdichte} = \frac{\text{Fehler}}{\text{1.000 Lines of Code}}$$

Typische Wert:

< 0,5	sehr stabile Programme, erstrebenswertes Maß
0,5 bis 3	reifende Programme
3 bis 6	labile Programme
6 bis 10	fehleranfällige Programme
>10	unbrauchbare Programme

Aus *Code Complete* von Steve McConnell:

Ca. 15 – 50 Fehler pro 1.000 Zeilen ausgeliefertem Code



SOFTWARE QUALITÄT

Was bedeutet Software Qualität

Gesamtheit der Merkmale und Merkmalswerte eines SW-Produktes, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

ISO 9126, ISO 25010

SOFTWARE QUALITÄT

Kompatibilität
- Koexistenz
- Interoperabilität

Effizienz
- Zeitverhalten
- Ressourcenverbrauch

Funktionalität
- Angemessenheit
- Richtigkeit
- Vollständigkeit

Wartbarkeit
- Änderbarkeit
- Modularität
- Prüfbarkeit
- Wiederverwertbarkeit
- Testbarkeit

Software Qualität
nach ISO 25010

Benutzbarkeit
- Verständlichkeit
- Erlernbarkeit
- Bedienbarkeit
- UI Ästhetik

Portabilität
- Installierbarkeit
- Ersetzbarkeit
- Adaptierbarkeit

Zuverlässigkeit
- Fehlertoleranz
- Verfügbarkeit
- Widerherstellbarkeit

Sicherheit (Security)
- Vertraulichkeit
- Integrität
- Authentizität

QUALITÄTSSICHERNDE MASSNAHMEN

- Meilensteine
- Projektplan mit Verantwortlichkeiten
- Entwicklungsprozess
- Anforderungsanalyse
- Architektur
- Iterationen und Reviews (am Ende einer Iteration)
- Dokumentenreviews
- Programmierrichtlinien

Analytische Qualitätssicherung
Ziel: Fehler vermeiden

- Statische Codeanalyse
- Codereviews
- **Testen**

Konstruktive Qualitätssicherung
Ziel: Fehler finden

WIEDERHOLUNG

Verifikation

Überprüfung der Übereinstimmung zwischen einem Software-Produkt und seiner Spezifikation.

Haben wir die Software richtig gebaut?

Validierung

Überprüfung der Eignung eines Software-Produktes bezogen auf seinen Einsatzzweck.

Haben wir die richtige Software gebaut?

TESTEN

Warum wird getestet?

- Testen ist **ein** Mittel zur Software-Qualitätssicherung!
- Software-Qualitätssicherung besteht aus mehr als Testen

Was ist ein Test?

- **Wiederholbare** Überprüfung eines Software-Bausteins in Bezug auf vorher festgelegte Anforderungen
- Besser: **Automatisierbare** Überprüfung!

Was kann Testen nicht?

- Testen kann **nicht die Abwesenheit** von Fehlern (Fehlerfreiheit) nachweisen
- Software **vollständig** zu testen ist in der Praxis **nicht möglich**

TESTUMFANG

Wenn nicht alles getestet werden kann, wieviel muss dann überhaupt testen?

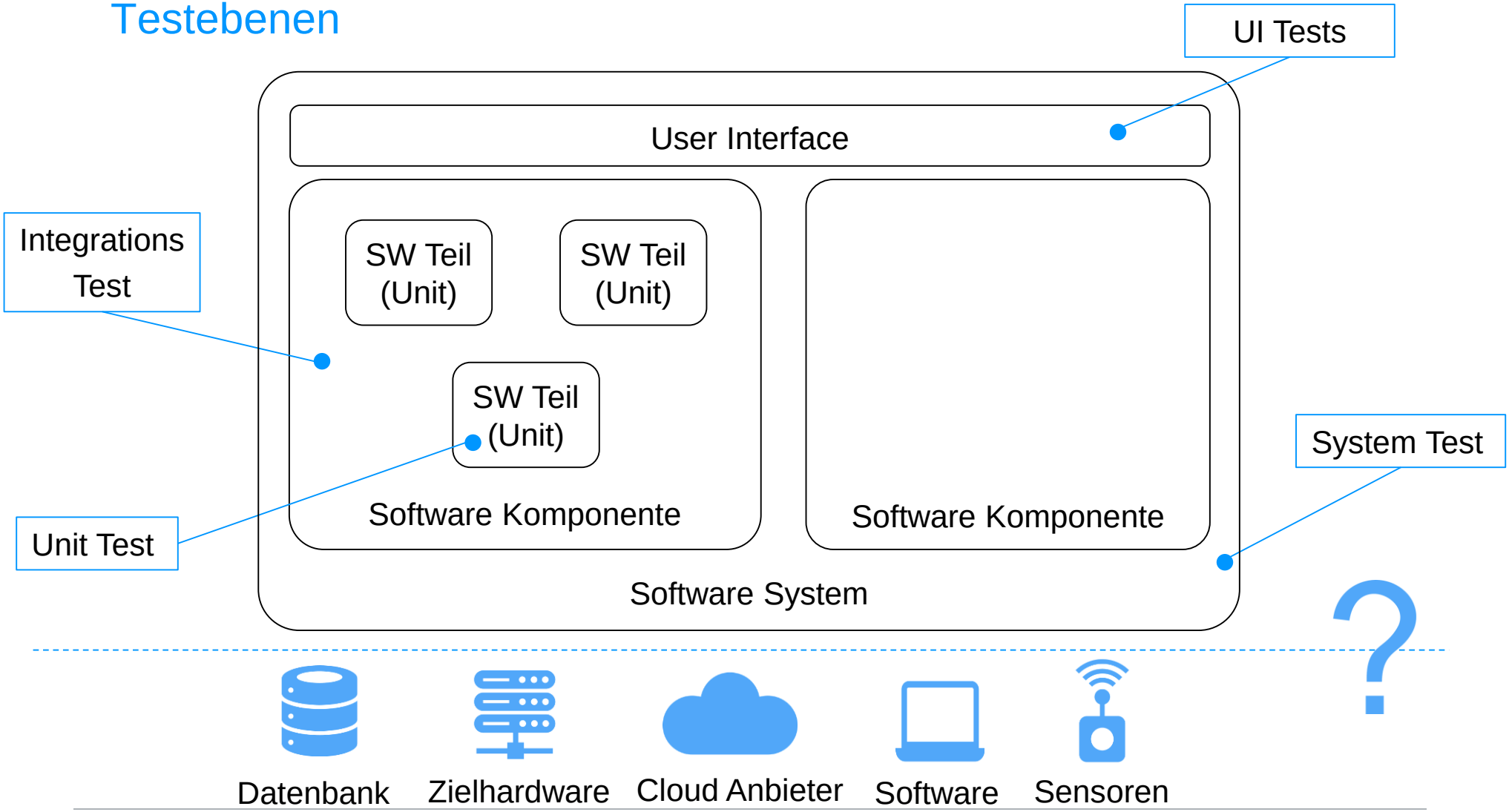
Kontext- und Risiko basierte Entscheidung

- Computerspiel → wirtschaftlicher Schaden beim Hersteller
- Büro-Software → wirtschaftlicher Schaden beim Hersteller und Kunden
- Medizintechnik → Personenschaden (s. THERAC-25)
- Luftfahrt (viele Personenschäden und hohe Sachschäden)

Risiko

- **Ungeplante, negative Abweichung** des erwarteten Verhaltens eines Systems
- Produkt aus der **Wahrscheinlichkeit des Auftretens eines Fehlers** und dem **dadurch erwarteten Verlust**

Testebenen


UI Tests
User Interface
Integrations Test
SW Teil (Unit)
SW Teil (Unit)
SW Teil (Unit)
Software Komponente
Software Komponente
System Test
Unit Test
Software System
?
Datenbank
Zielhardware
Cloud Anbieter
Software
Sensoren

AUTOMATISIERTE UI TESTS

TESTAUTOMATISIERUNG

Zwei grundsätzliche Ansätze für automatisierte UI Tests

Entwickler + Tester

- Programmierung der UI / Coded UI Tests
  - UI Elemente werden via Code/API angesteuert
  - Beispiele: AssertJ Swing
  - Microsoft Coded UI Tests (deprecated) bzw. Appium

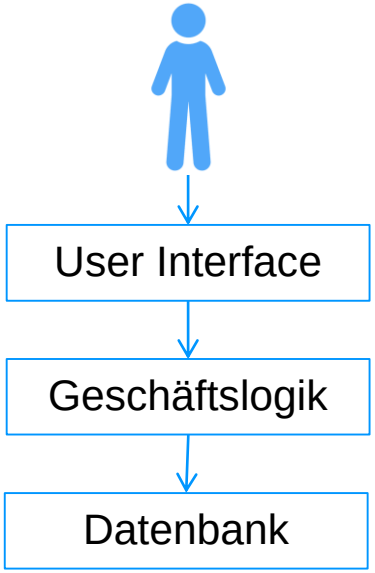
```
session.FindElementByName("Don't Save").Click();
```

- Capture & Replay
  - Marathon
  - Selenium

Tester

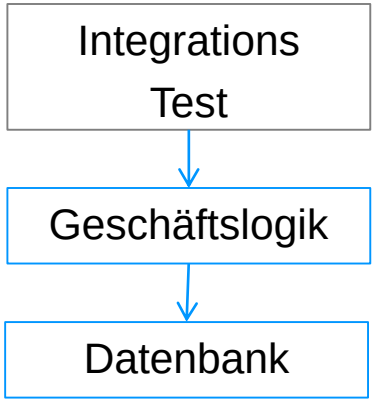
WARUM AUTOMATISIERTE UI TESTS

F5 Experience bzw. Ausprobieren



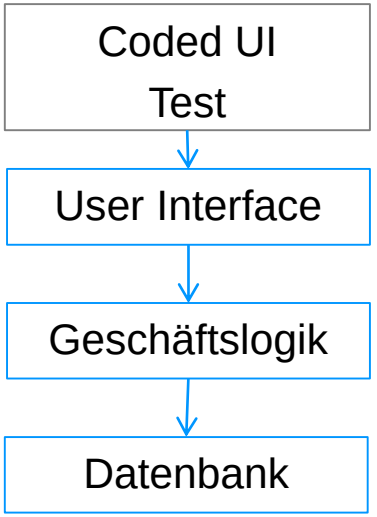
Manuelles testen bzw. probieren so vieler Aspekte der Anwendung wie möglich

Typische Tests ohne UI



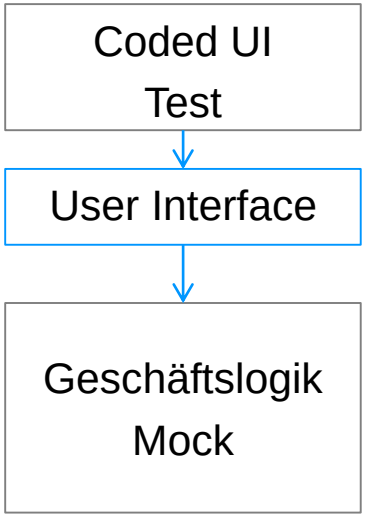
Gesamte Anwendung ohne UI testen

Typische UI Tests



Testen der gesamten Anwendung durch die UI (automatisch)

Tests „nur“ um UI zu verifizieren



UI isoliert Testen

UNIT TESTS

WARUM UNIT TESTS

1. Verifizierung des Codes

   Macht der Code was er soll?

2. Design-Qualität

   Testbarer Code hat meist eine höhere Design Qualität
   (Prinzip: Design for Testability)

3. Dokumentation des Codes / der API

   Unit-Test sind eine ausführbare Dokumentation des Codes / der API

Achtung: Unit Tests können selbst fehlerhaft sein

UNIT TEST

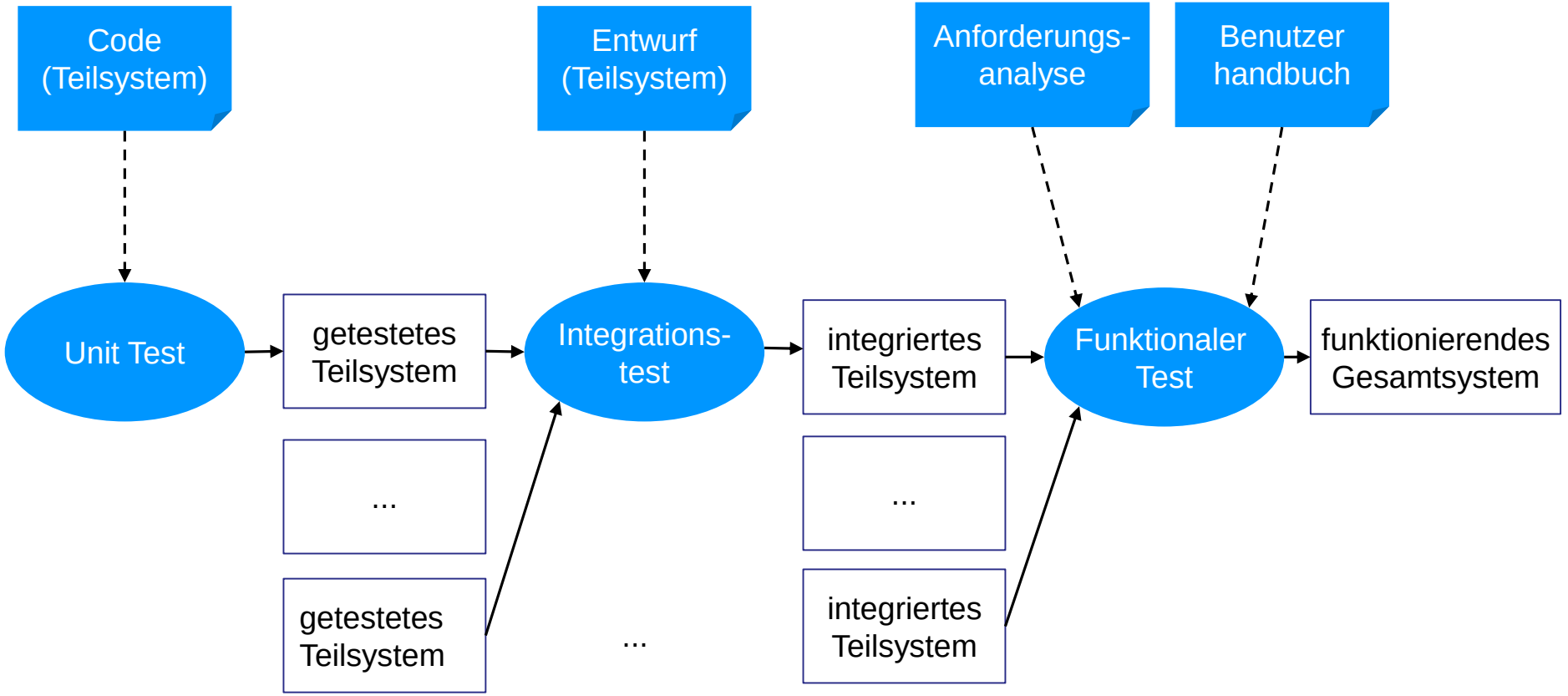
Ein Unit Test ist ein (automatisierbar ausführendes) Stück Code, welches eine Software Einheit (Unit, meist Methode oder Klasse) ausführt und Annahmen über dessen Verhalten überprüft.

EIGENSCHAFTEN GUTER UNIT TESTS

Trustworthy	Rot heißt „Fehler“, grün heißt „alles OK“
Maintainable	Änderung am produktiven Code sollten nur wenig Änderung am Test-Code benötigen (z.B. nur *public* Methoden testen)
Readable	Unit Test ist eine Dokumentation er API, Namensgebungen, nur eine Funktion testen (*AddStockTest* vs. *AddAndSplitStockTest*)
Isoliert	Test hat keine Abhängigkeiten zu anderen Komponenten und insb. Zu anderen Tests
Schnell	Geringe Laufzeit des Tests, da dieser oft ausgeführt wird
Automatisiert	Wird in CI/CD Pipelines verwendet
Konsistent	Liefert immer das gleiche Ergebnis

INTEGRATIONSTESTS

WO EINORDNEN?



INTEGRATIONSTESTS

Was?

- Gruppen von Teilsystemen (z.B. Sammlung von Klassen), aber u.U. auch das ganze System

Ziel?

- Überprüfen der Schnittstellen zwischen den Teilsystemen

Wer?

- Wird von Entwicklern durchgeführt

TESTSTRATEGIEN

Teststrategie = Die Reihenfolge, in der Teilsysteme für Tests für die Integration ausgewählt werden

- Big-Bang Integration
- Bottom-Up Integration
- Top-Down Integration
- Sandwich Testing
- Variationen der o.g. Strategien

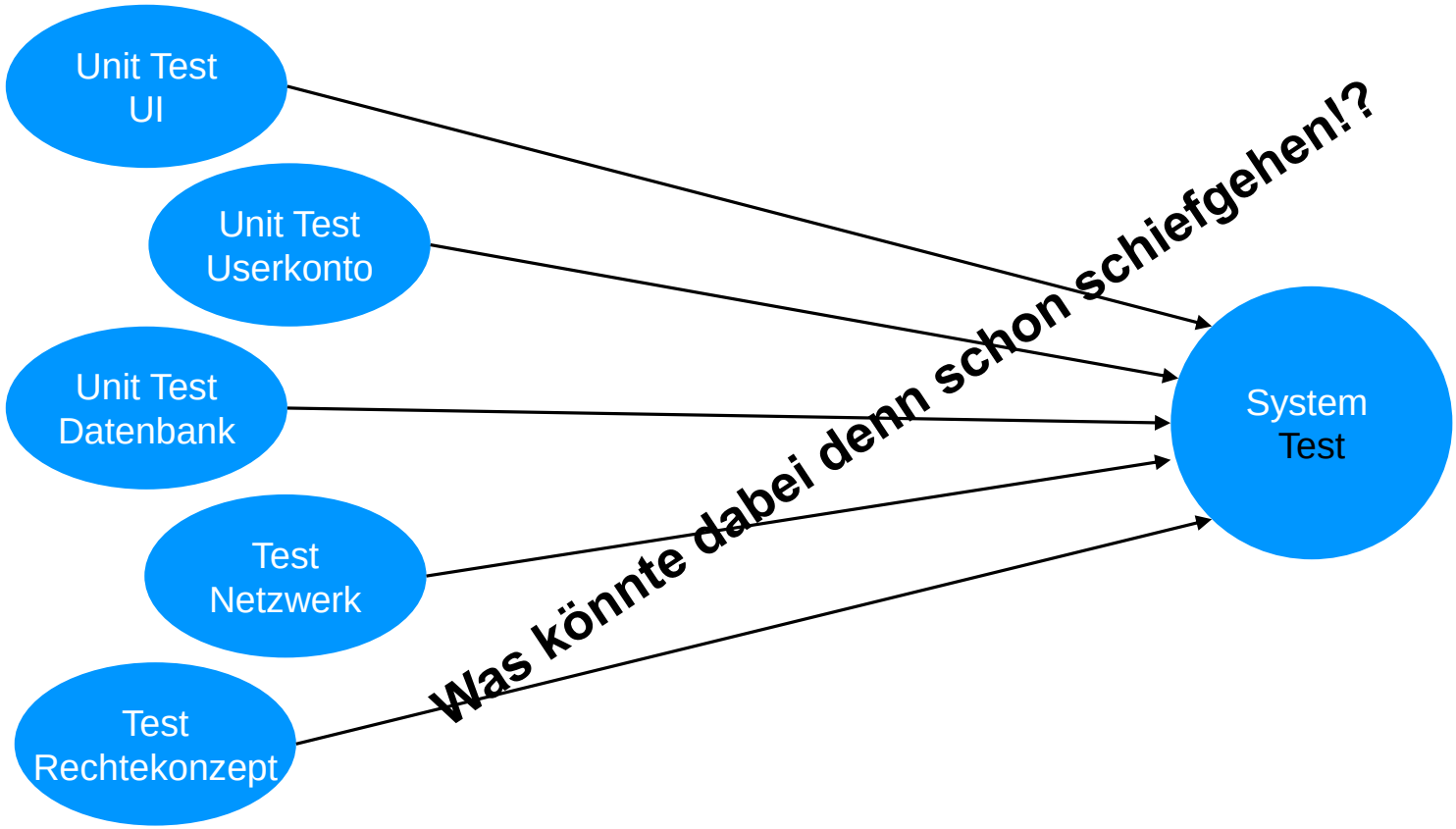
ABLAUF BEI INTEGRATIONSTESTS

Schritte 1 bis 5 solange wiederholen bis da Gesamtsystem getestet ist

1. Wähle eine Komponente zum Testen aus
2. Führe für alle Teilsysteme (Klassen) der Komponenten Modul-Tests (Unit-Tests) durch
3. Führe Anpassungen durch, um Integrationstests lauffähig zu machen (Treiber, Stubs entwickeln)
4. Tests durchführen
5. Audit der Testfälle und Testaktivitäten erstellen

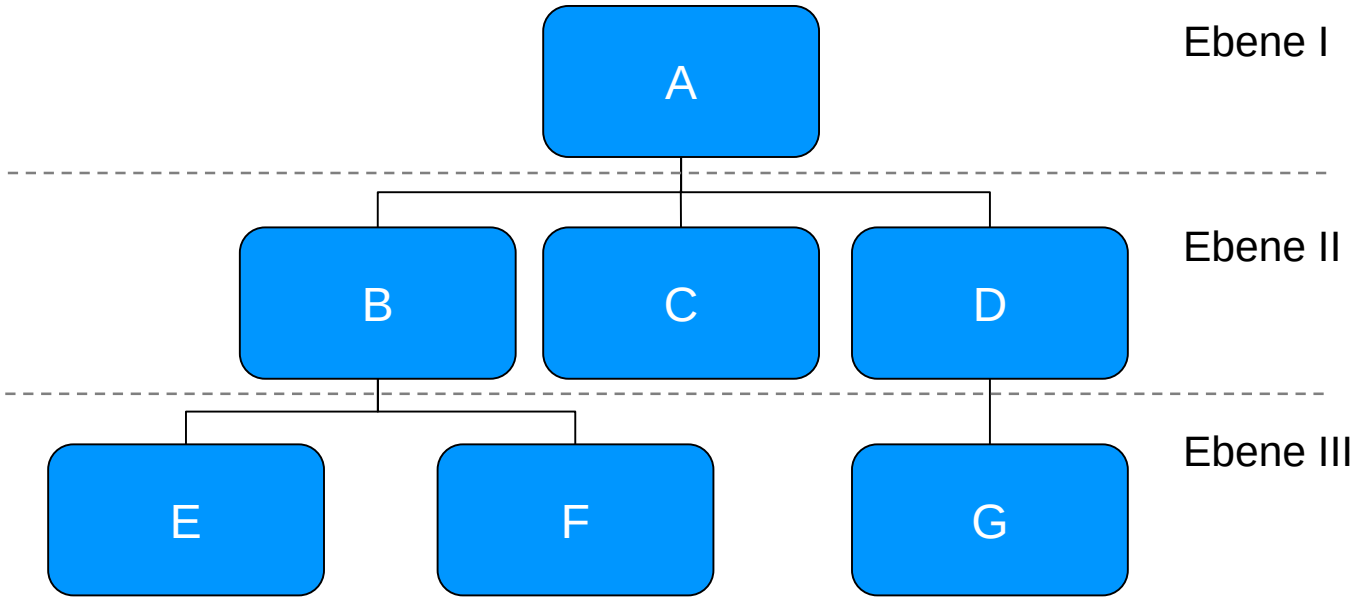
Primäres Ziel von Integrationstests ist es, Fehler in der jeweiligen Konfiguration der Teilsysteme zu identifizieren.

BIG BANG INTEGRATIONSTEST



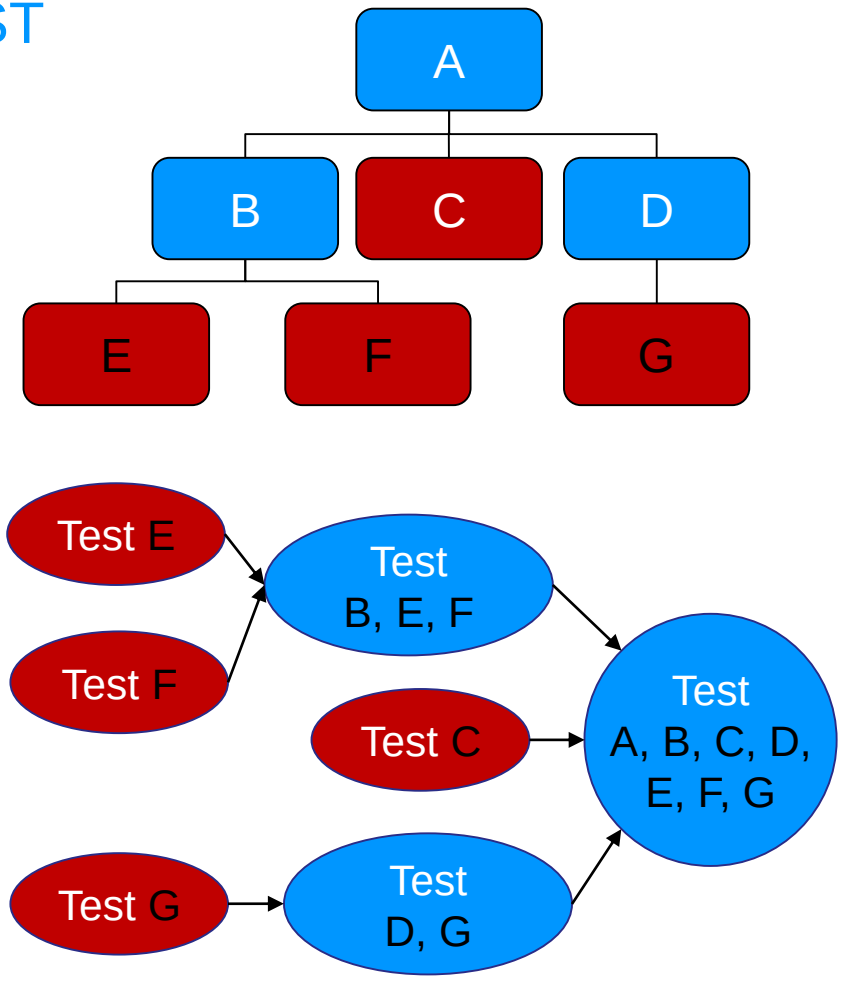
BEISPIEL

Beispiel für Integrationstests:
Aufrufhierarchie mit drei Ebenen



BOTTOM-UP INTEGRATIONSTEST

- Teilsystem in der untersten Ebene der Aufrufhierarchie werden einzeln getestet
- Danach werden die Teilsysteme in der nächsten Ebene getestet, die die zuvor getesteten Teilsysteme aufrufen
- Vorgang wird solange wiederholt bis alle Teilsysteme getestet sind
- Es werden Testtreiber benötigt
  - Führen einen Testfall auf einem Teilsystem aus
  - Protokollieren und die Testergebnisse



BOTTOM-UP INTEGRAITONSTEST

Vor- und Nachteile

Schlecht für funktional aufgeteilte Systeme

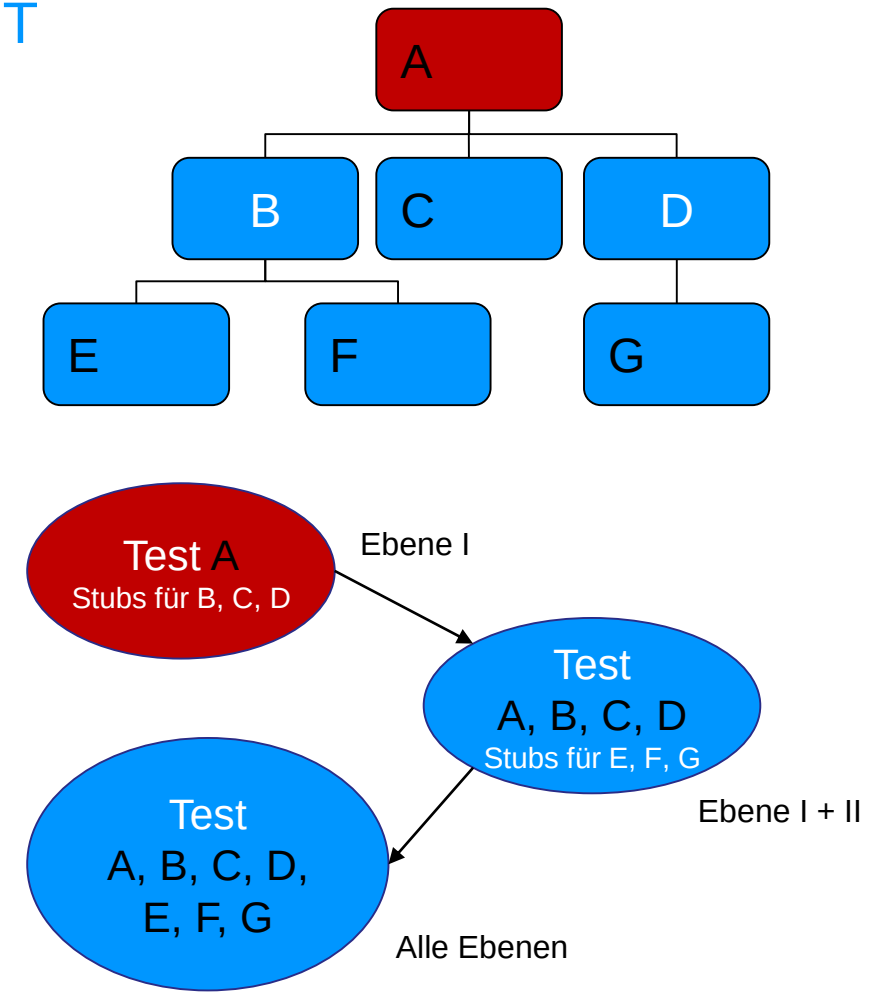
- Die wichtigsten Teilsysteme werden zuletzt getestet
- Vergleichen Sie dies mit Ihrem Projekt

Gut geeignet für

- Objektorientierte Systeme
- Systemen mit strikten Anforderungen an Performance
- Echtzeitsysteme

TOP-DOWN INTEGRATIONSTEST

- Oberste Schicht zuerst Testen
- Danach alle Teilsysteme testen, die vom zuvor getesteten Teilsysteme aufgerufen werden
- Solange wiederholen bis alle Teilsysteme im Test enthalten sind
- Test Stubs
  - Methoden/Programme, die das Verhalten eines fehlenden Teilsystems simulieren
  - Antworten auf Testaufrufe mit definierten Daten, ohne Funktionalität zu implementieren



TOP-DOWN INTEGRATIONSTESTS

Vor- und Nachteile

Pro

- Testfälle können hinsichtlich der Funktionalen Anforderungen eines Systems formuliert werden

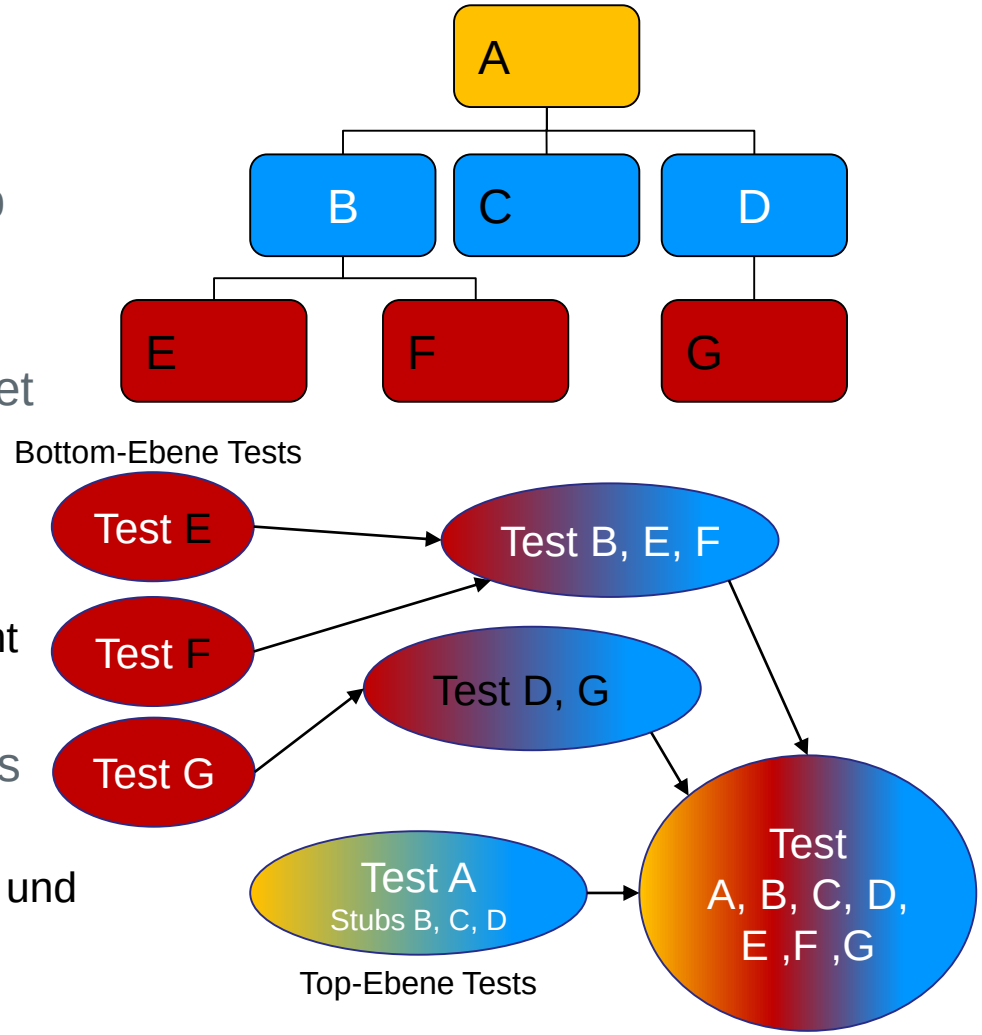
Contra

- Das Schreiben von Stubs ist meist aufwendig
  - Müssen Testen aller Bedingungen erlauben
  - Große Anzahl von Stubs kann benötigt werden

- Vermeidung einer großen Anzahl von Stubs mittels modifizierter Top-Down Teststrategien
  - Vor dem Zusammenfügen wird jede Schicht einzeln getestet
  - Nachteil hierbei: Es werden sowohl Stubs als auch Treiber benötigt

SANDWICH TESTSTRATEGIEN

- Kombinierte Top-Down und Bottom-Up Teststrategie
  - Parallel Ausführung
- System wird in drei Schichten betrachtet
  - Zielschicht (in der Mitte)
  - Eine Schicht über dem Ziel
  - Eine Schicht unter dem Ziel
  - Die Tests konvergiert auf die Zielschicht
- Wie findet man die Zielschicht, wenn es mehr als drei Ebenen gibt?
  - Heuristik: Versuche Anzahl der Treiber und Stubs zu minimieren



SANDWITCH TESTEN

Vor- und Nachteile

Pro

- Frühes Testen von z.B. Benutzeroberfläche möglich
- Testen von oberen und unteren Schichten kann gleichzeitig stattfinden
- Keine Stubs und Treiber für die oberen und unteren Schichten erforderlich
  - Zielschichte ersetzt Treiber für untere Schicht
  - Zielschicht ersetzt Stubs für obere Schicht

Contra

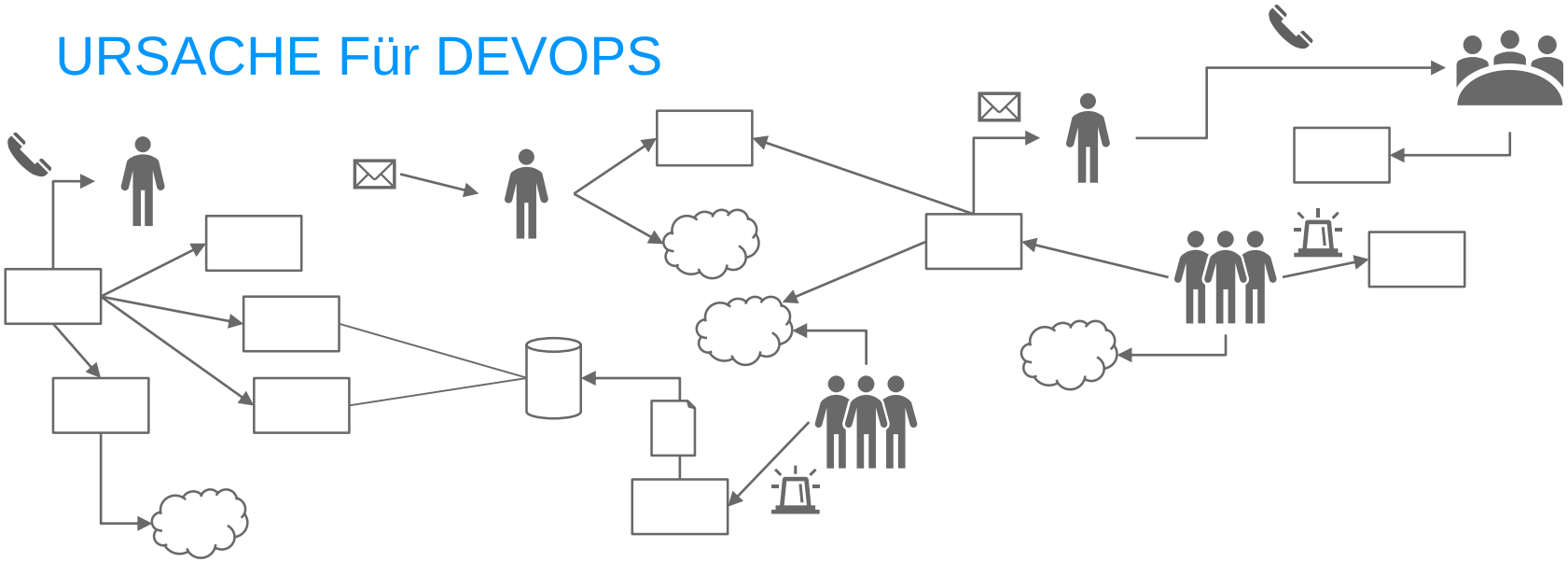
- Keine Modultests der Zielschicht vor der Integration, d.h. kein gründliches Testen der Zielschicht

Lösungsidee

- Modifizierte Teststrategie: Jede Schicht vor der Integration komplett testen: mehr Stubs und Treiber als vorher
- Aber i.d.R. kürzeste Testdauer aufgrund der Parallelität

DER DEVOPS ANSATZ

URSACHE Für DEVOPS



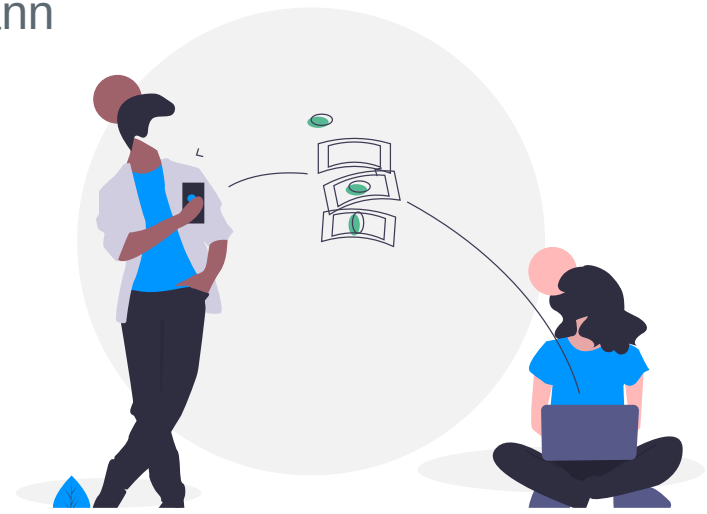
- **Was denken Sie, wie lange es Dauert eine Kundenanforderung in Betrieb zu nehmen?**
  - Laufzeiten von **Wochen und Monaten** sind keine Seltenheit
  - **Probleme** werden dabei erst **am Ende** des Deployment-Durchlaufs **erkannt**
  - In solchen Prozessen sind oftmals **manuelles Testen, manuelles Deployment und persönliche Heldentaten** vorherrschend

AUTOMATISIERUNG ALS WERTSCHÖPFUNG

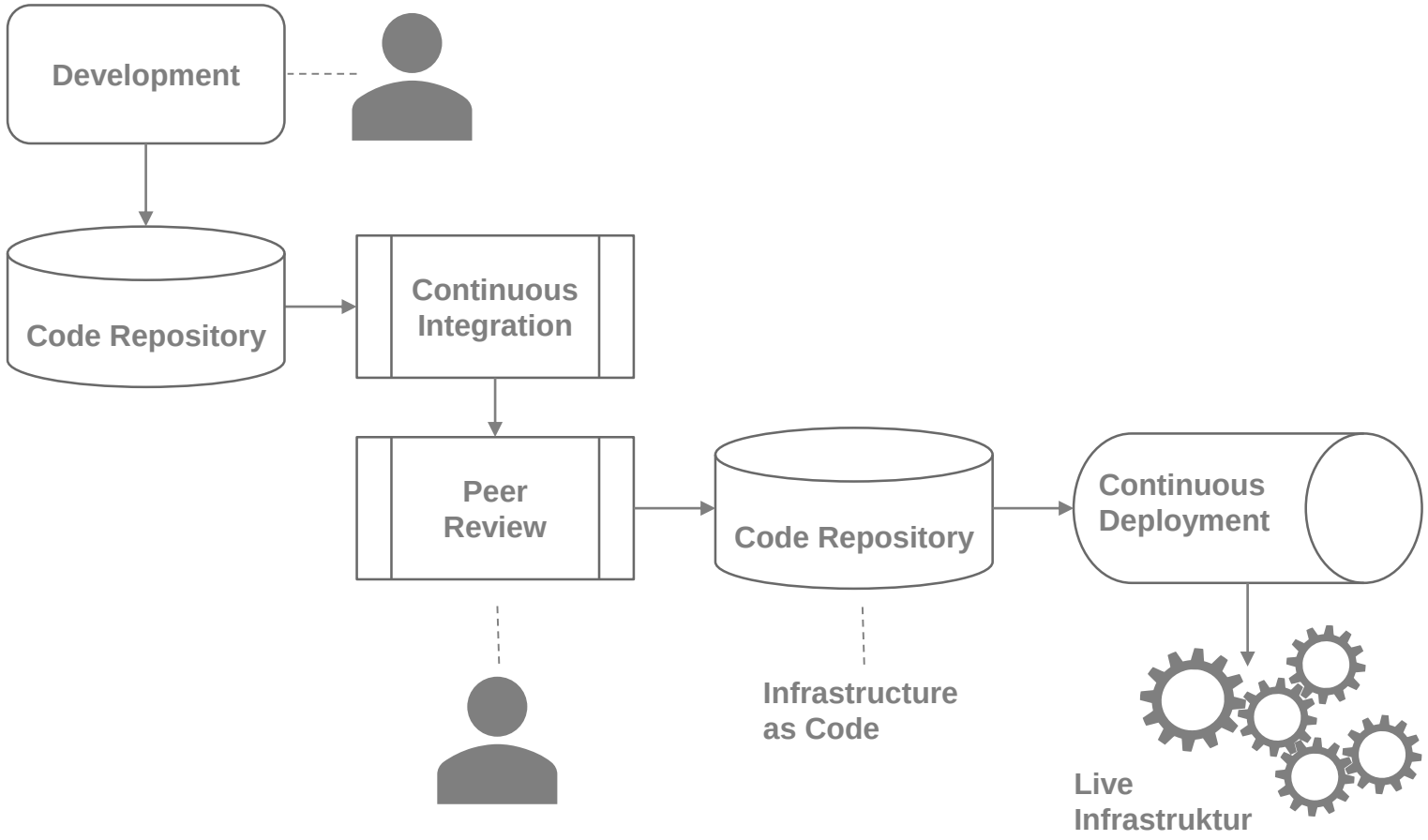
- Von der Entwicklung bis zu dem Moment, in dem eine Kundenanforderung in einer produktiven Umgebung bereitsteht handelt es sich um eine **Technologie Wertekette**

- Eine **Wertschöpfung** beginnt jedoch erst in dem Moment, in dem der Kunde ein Feature tatsächlich produktiv benutzen kann

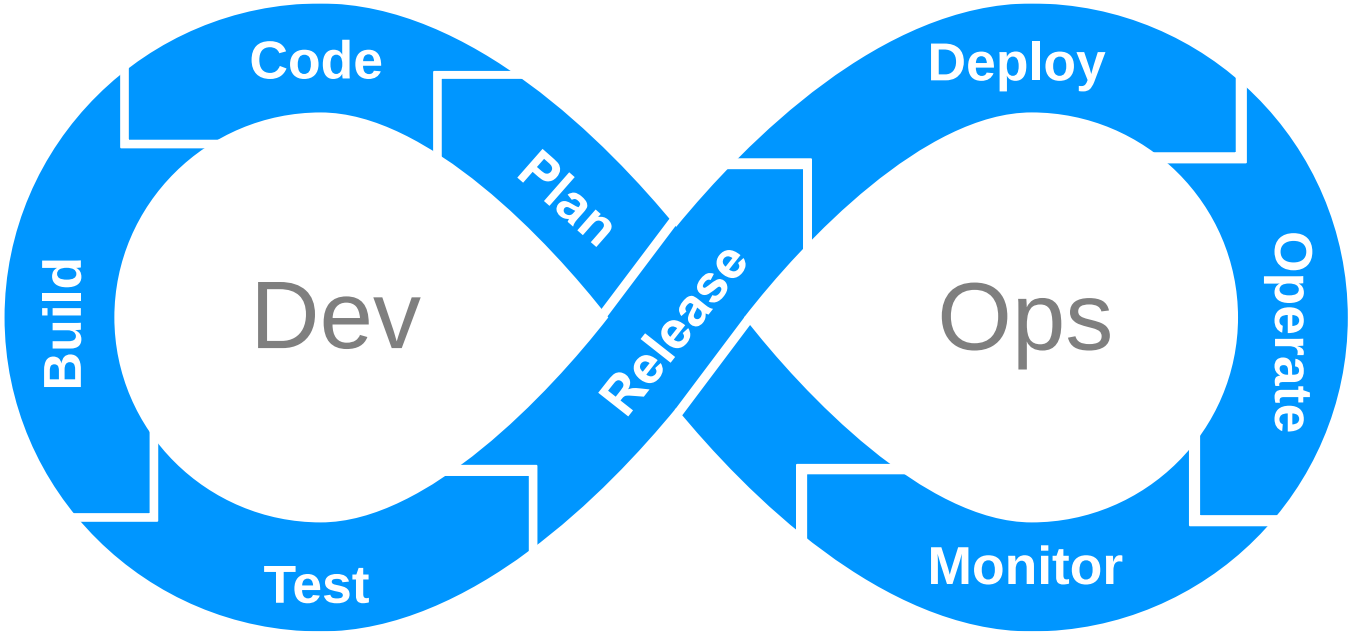
- **Logische Konsequenz**: Die Technologie Wertekette sollte so schnell wie möglich durchlaufen werden. Aber wie?



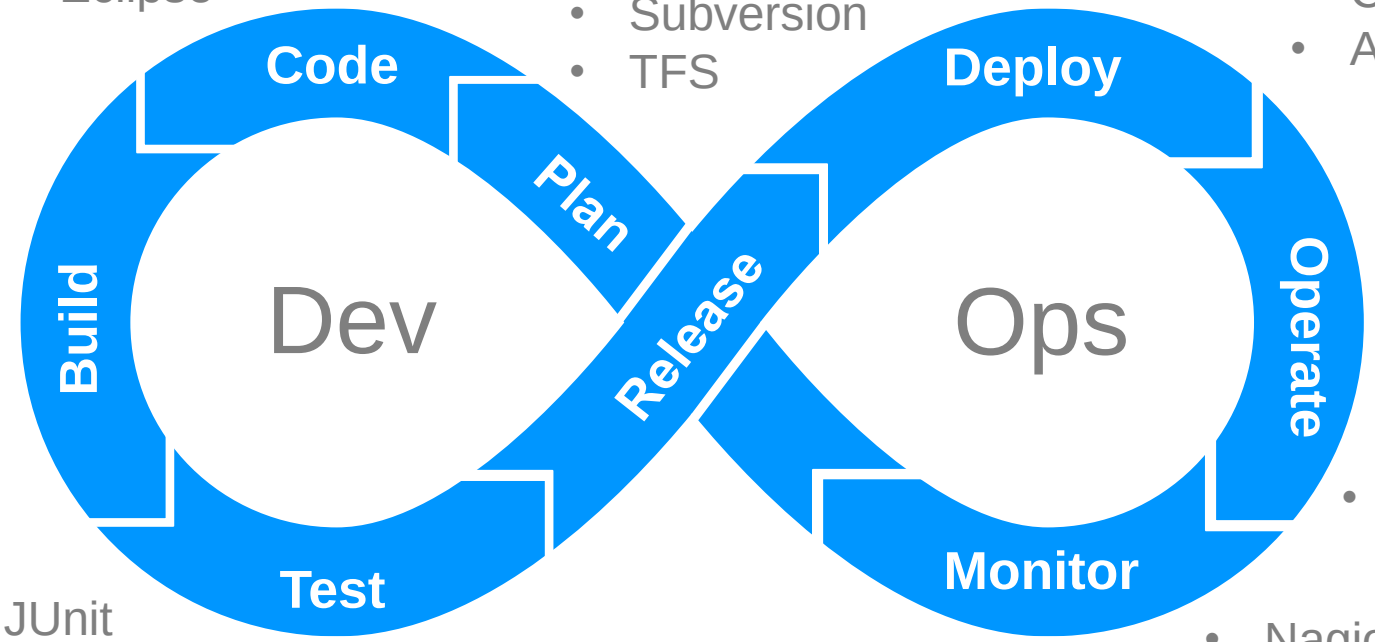
AUTOMATISIERUNGSANSATZ CI/CD



DEVOPS ANSATZ



DEVOPS ANSATZ



- Visual Studio
- Eclipse

- Git
- Subversion
- TFS

- Docker
- AWS
- Azure

- Puppet
- SaltStack
- Chef
- Ansible

- Maven
- Gradle
- Ant
- MSBuild

- JUnit
- XUnit
- Selenium

- Kubernetes

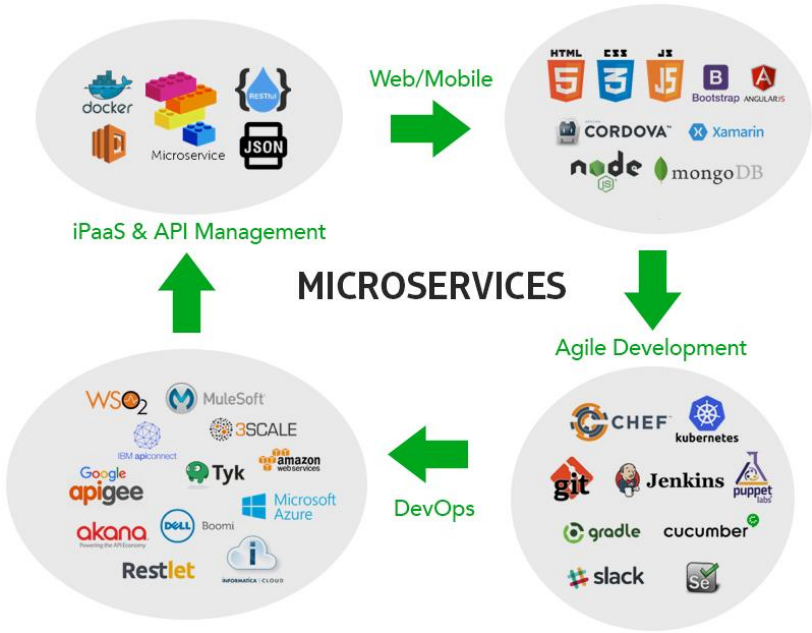
- Nagios
- Prometheus
- Grafana

WIEDERHOLUNG

Hohe Komplexität durch zahlreiche Werkzeuge?

Ja aber Werkzeuge ermöglichen

- Schnellere Entwicklung
- Einfacheres Testen
- Schnelles und häufiges Deployment
- Relativ leichte Fehlerkorrekturen
- Effiziente Verwaltung von Code
- Schnellere Bereitstellung von Funktionalität

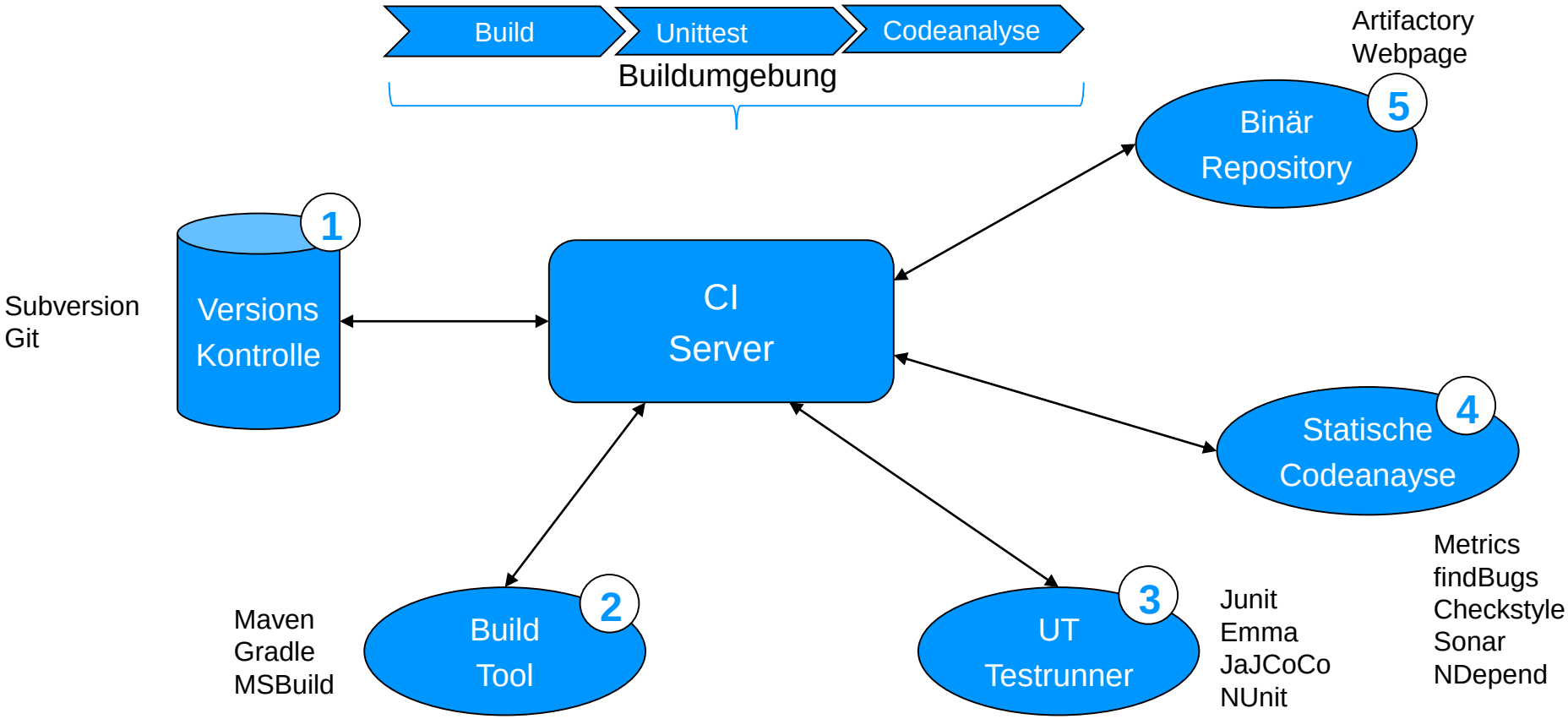


Quelle: https://www.appcentrica.com/the-rise-of-microservices/

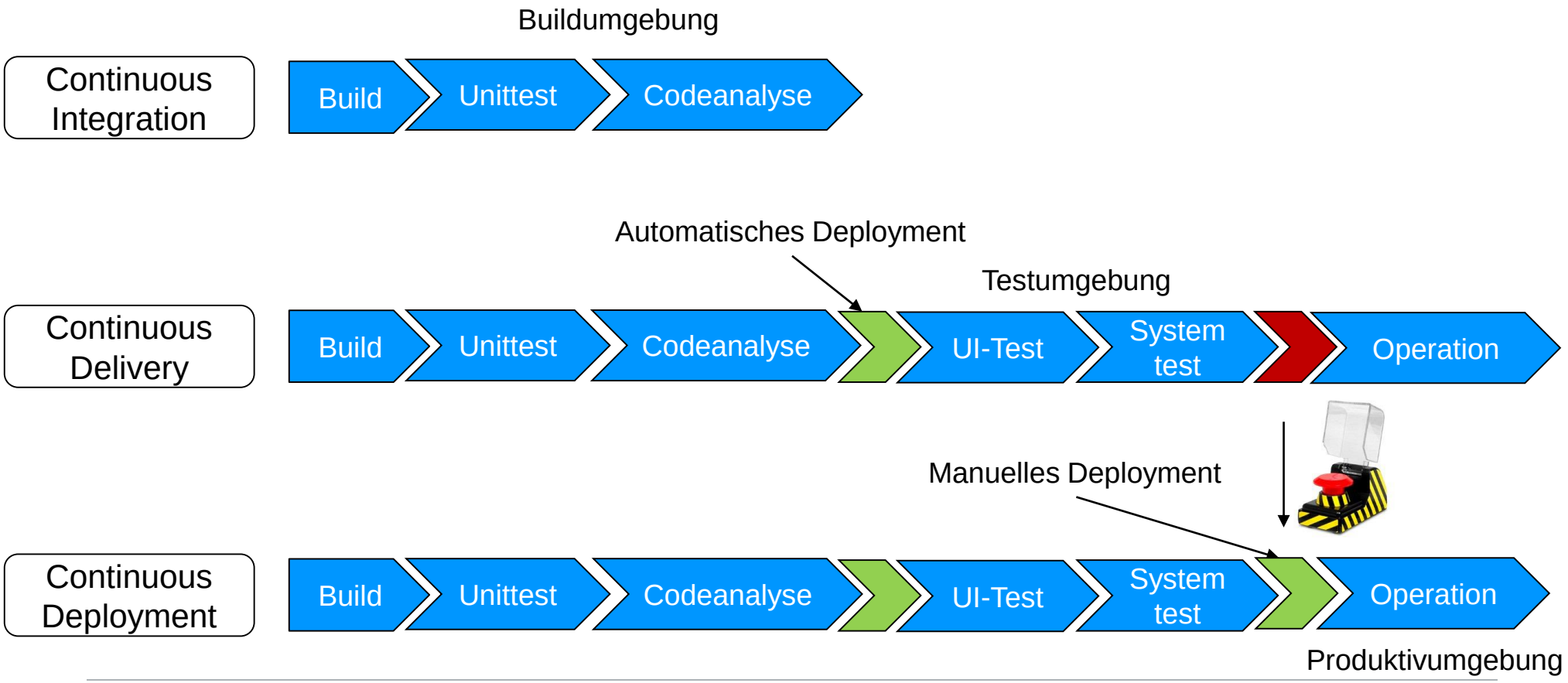
Praxistipp: „Know Your Tools“

Praxiserfahrung: Entwickler scheitern oft an den Tools

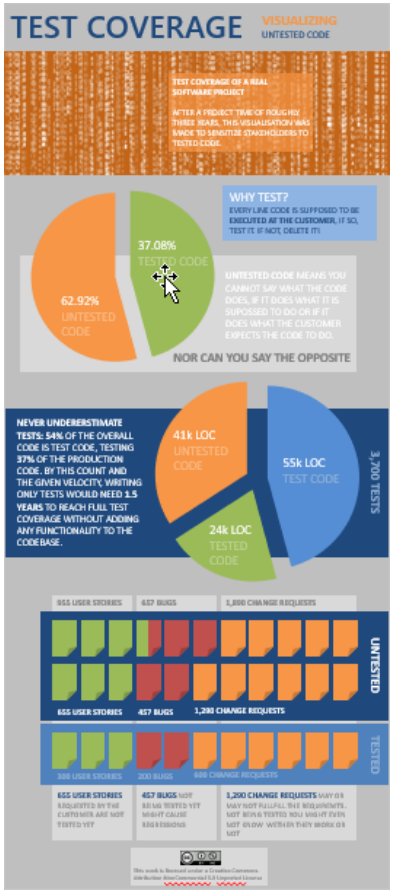
CONTINUOUS INTEGRATION



CONTINUOUS DEPLOYMENT



INFOGRAFIK – REAL WORLD EXAMPLE



Fragen bis hier her?