2511 Softwarearchitektur, Software-Qualitätsmanagement

02 Übung Software-Fehler

02 Übung Software-Fehler
von Simeon Liniger - Montag, 5. September 2016, 13:26

Eure Kreativität ist gefragt! Schreibt als Gruppe ein kleines Java-Konsolen-Programm, das ein Fehler enthält. Ziel ist es dabei, den Fehler so zu verstecken, dass er von der Rest der Klasse nicht sofort erkannt wird.

Aufgabe

- Schreibt ein kleines Konsolen-Programm welches bei der Kompilierung oder Ausführung auf einen Fehler oder in ein unerwartetes Verhalten läuft.
- Schreibt ein Forumsbeitrag mit folgenden Inhalt:
  - Source-Code (als Text oder Bild) (wenn möglich nicht mehr als 10-15 Code-Zeilen)
  - Kurzer Beschrieb, was passieren sollte und was das Resultat beim Ausführen ist. (Z.B. sollte "Bla" auf der Konsole ausgeben, aber es erscheint eine Null-Pointer-Exception.)
- Im Plenum könnt ihr anschliessend das Rätsel auflösen. Macht euch dazu Gedanken, um welche Kategorie von Code-Fehler es sich dabei handelt. (Syntax, Semantisch, Typen, Parallelität, Portabilität, Optimierung, …) (Wir werden diese Fehlertypen gleich im Unterricht noch kurz behandeln.)
- Überlegt euch, wie der Fehler hätte verhindert werden können.

Beispiel

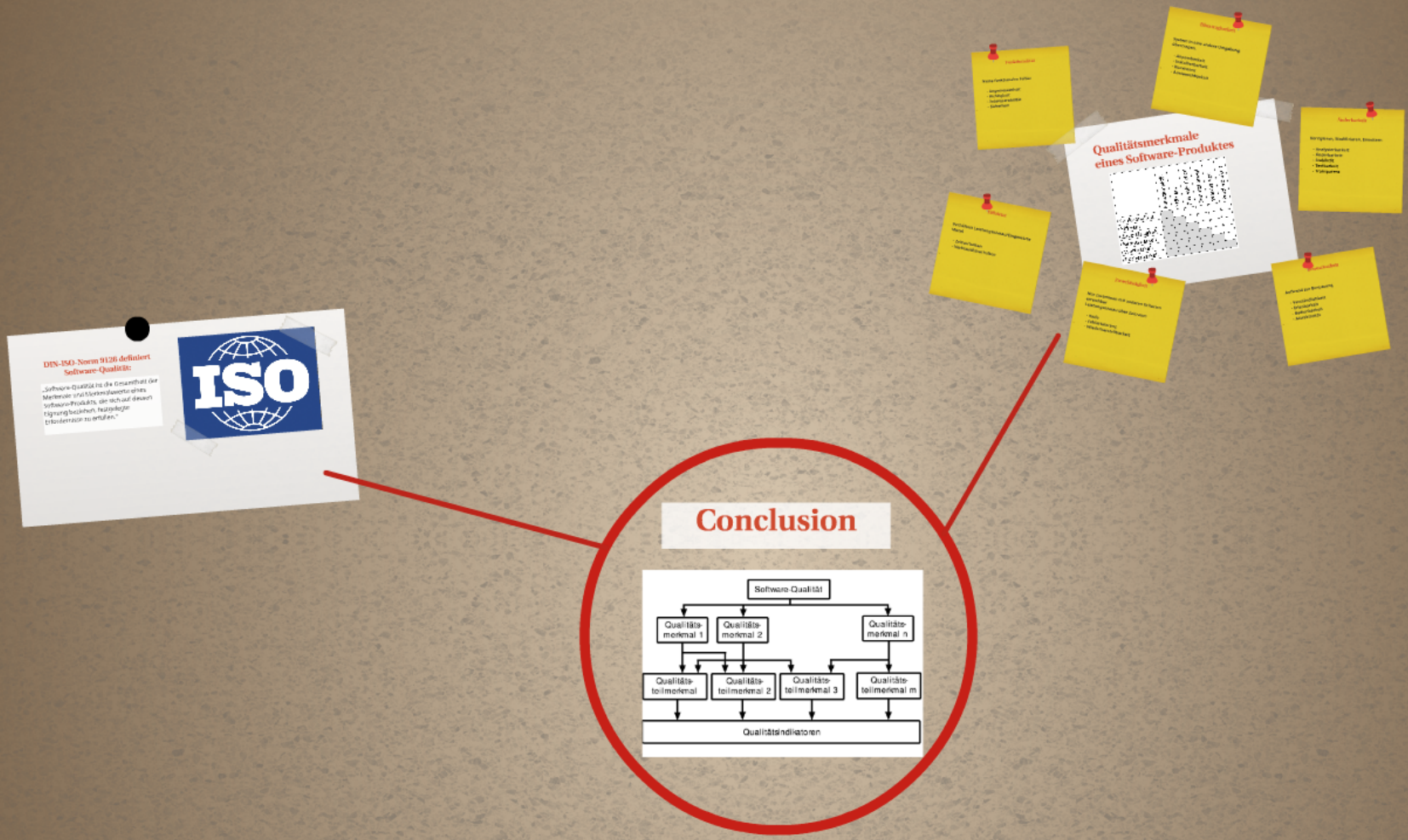
Beim Ausführen der main-Methode erscheint "0" in der Ausgabe anstelle der erwarteten "15".

```java
public class ErrorFeature {

    public static void main(String[] args) {
            int a = 5;
            int b = 3;
            int result = 0;

            multiply(a,b);

            System.out.println(result);
    }

    public static int multiply(int a, int b)
    {
        int result = a * b;
        return result;
    }
}
```

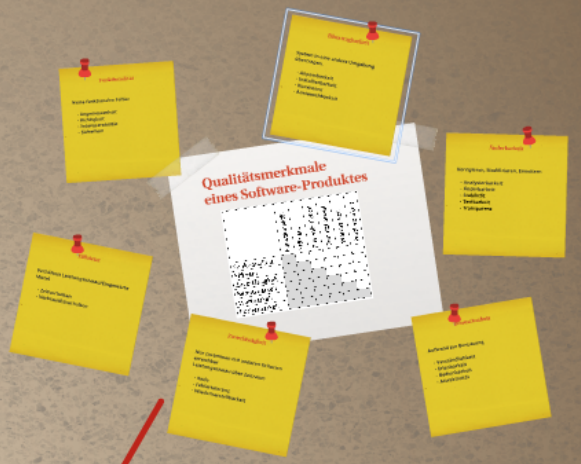
Dauerlink | Antworten

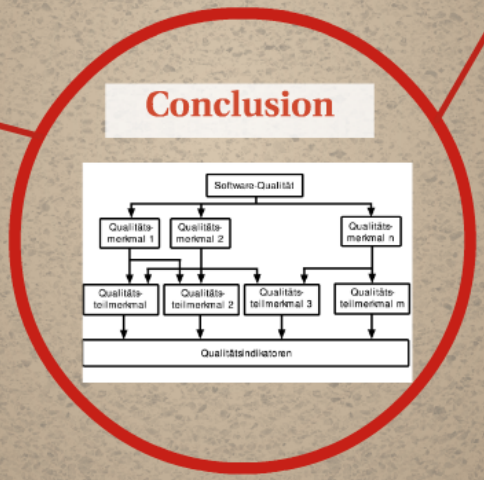
Re: Software-Fehler
von Ljubisa Markovic - Montag, 5. September 2016, 13:23

```java
import java.io.*;


public class BeispielQualität{


public static void main(String[] args) {

    int aWholeNumber; //Deklaration

    double notAWholeNumber; //Deklaration

    String aString;//Deklaration


    aWholeNumber = 0.0; //Initialisierung

    notAWholeNumber = 0; //Initialisierung

    aString = ""; //Initialisierung

    System.out.println(aWholeNumber);

    System.out.println(notAWholeNumber);

    System.out.println(gString);


    int anotherWholeNumber = 0; //Deklaration und Initialisierung
```



Was bedeutet Software-Qualiät?

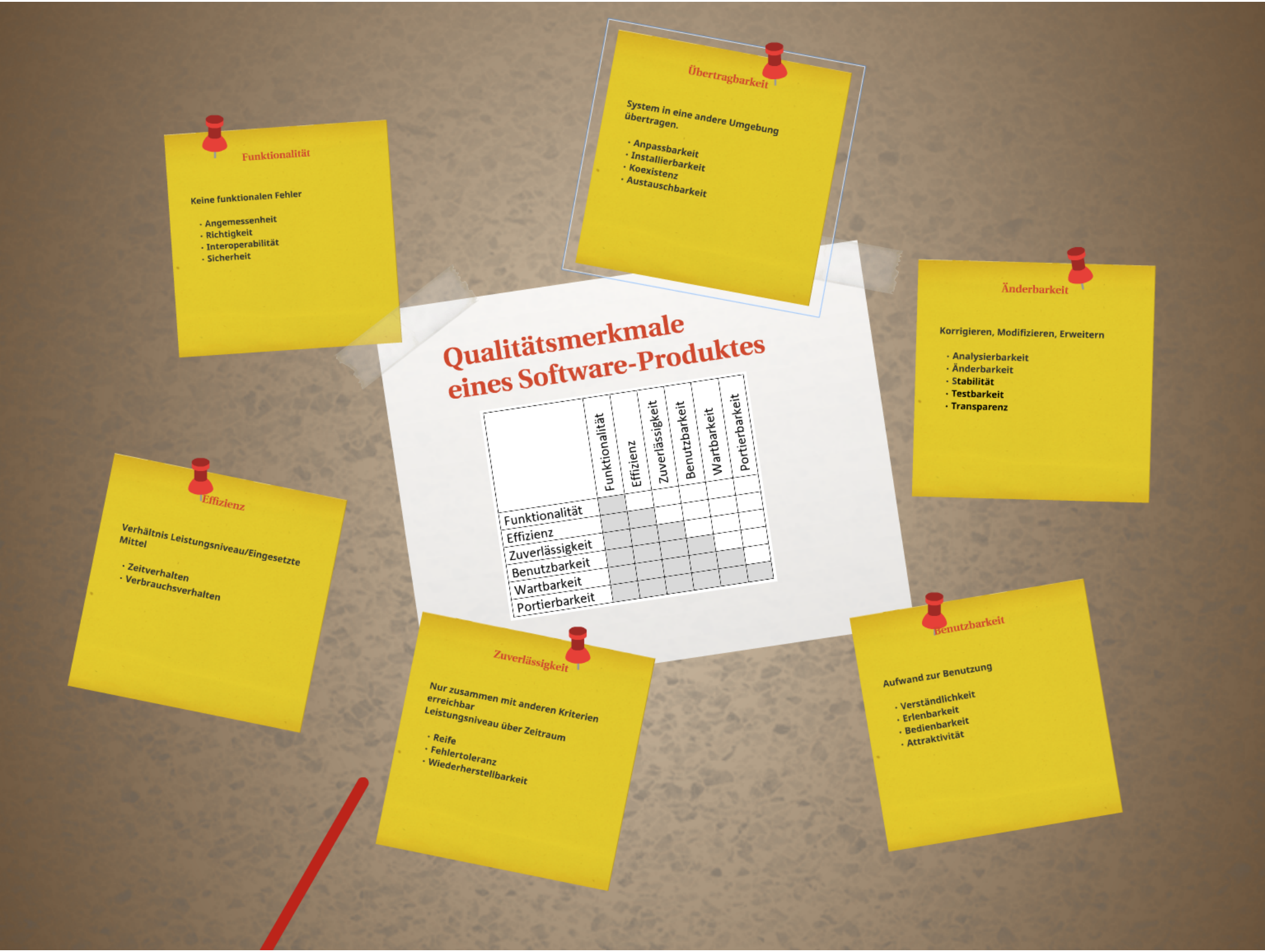




Was bedeutet Software-Qualiät?

DIN-ISO-Norm 9126 definiert Software-Qualität:

„Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen.“


Funktionalität
Keine funktionalen Fehler
· Angemessenheit
· Richtigkeit
· Interoperabilität
· Sicherheit
Übertragbarkeit
System in eine andere Umgebung übertragen.
· Anpassbarkeit
· Installierbarkeit
· Koexistenz
· Austauschbarkeit
Änderbarkeit
Korrigieren, Modifizieren, Erweitern
· Analysierbarkeit
· Änderbarkeit
· Stabilität
· Testbarkeit
· Transparenz
Qualitätsmerkmale eines Software-Produktes
Funktionalität
Effizienz
Zuverlässigkeit
Benutzbarkeit
Wartbarkeit
Portierbarkeit
Effizienz
Verhältnis Leistungsniveau/Eingesetzte Mittel
· Zeitverhalten
· Verbrauchsverhalten
Zuverlässigkeit
Nur zusammen mit anderen Kriterien erreichbar
Leistungsniveau über Zeitraum
· Reife
· Fehlertoleranz
· Wiederherstellbarkeit
Benutzbarkeit
Aufwand zur Benutzung
· Verständlichkeit
· Erlenbarkeit
· Bedienbarkeit
· Attraktivität

Funktionalität

Keine funktionalen Fehler

- **Angemessenheit**
- **Richtigkeit**
- **Interoperabilität**
- **Sicherheit**

Effizienz

Verhältnis Leistungsniveau/Eingesetzte Mittel

- **Zeitverhalten**
- **Verbrauchsverhalten**

Zuverlässigkeit

Nur zusammen mit anderen Kriterien erreichbar
Leistungsniveau über Zeitraum

- **Reife**
- **Fehlertoleranz**
- **Wiederherstellbarkeit**

Benutzbarkeit

Aufwand zur Benutzung

- **Verständlichkeit**
- **Erlenbarkeit**
- **Bedienbarkeit**
- **Attraktivität**

Änderbarkeit

Korrigieren, Modifizieren, Erweitern

- **Analysierbarkeit**
- **Änderbarkeit**
- **Stabilität**
- **Testbarkeit**
- **Transparenz**

Übertragbarkeit

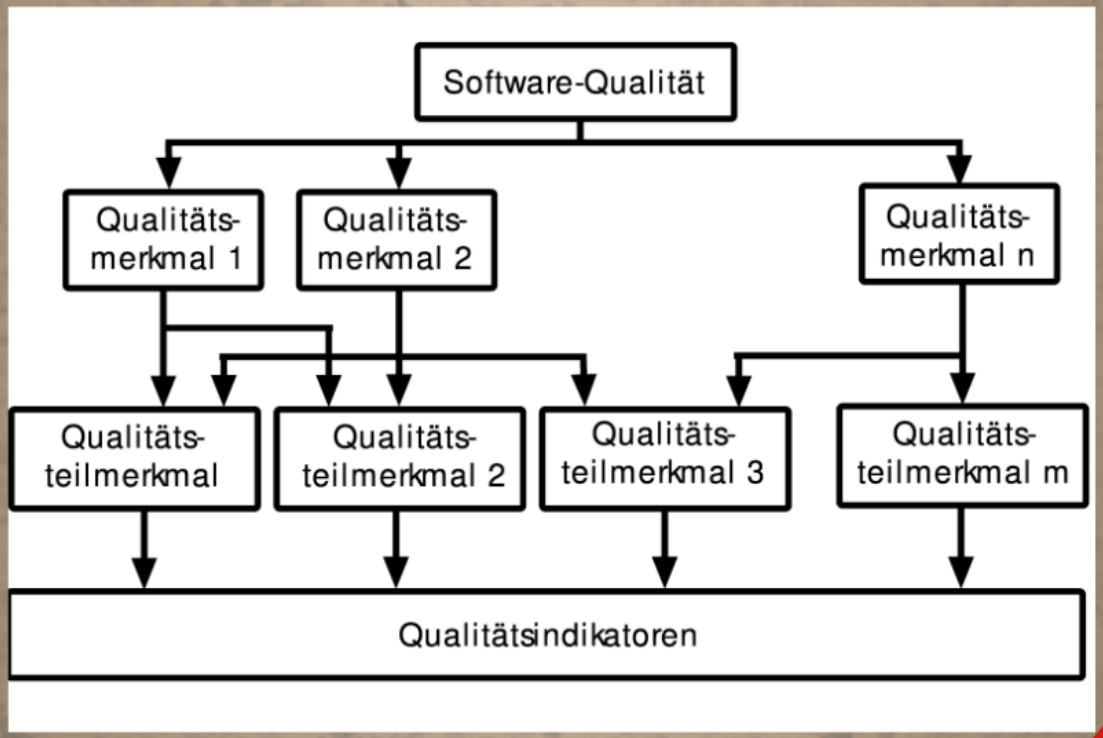
System in eine andere Umgebung übertragen.

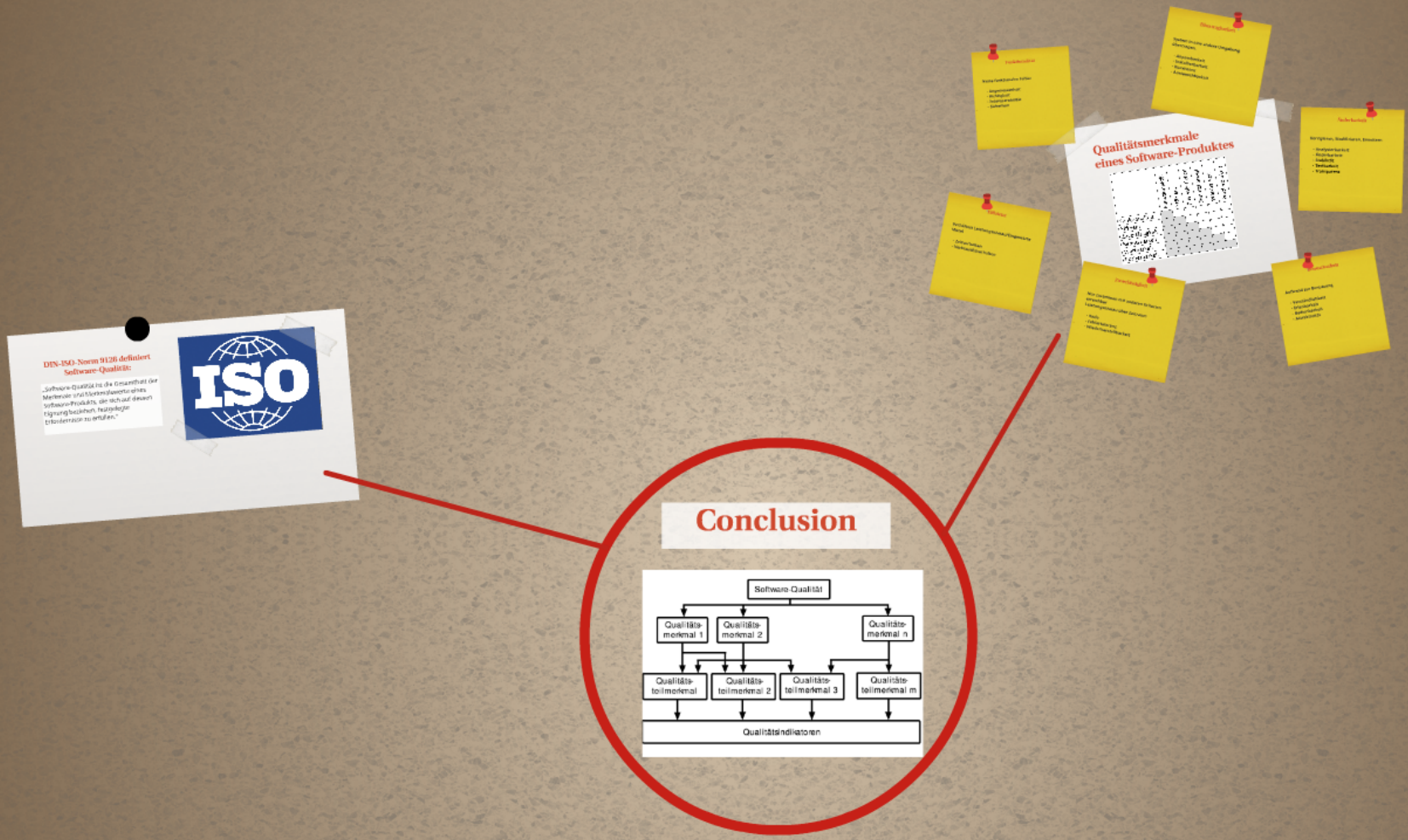
- Anpassbarkeit
- Installierbarkeit
- Koexistenz
- Austauschbarkeit

Qualitätsmerkmale eines Software-Produktes

	Funktionalität	Effizienz	Zuverlässigkeit	Benutzbarkeit	Wartbarkeit	Portierbarkeit
Funktionalität						
Effizienz						
Zuverlässigkeit						
Benutzbarkeit						
Wartbarkeit						
Portierbarkeit						

Conclusion


Software-Qualität
Qualitäts-merkmal 1
Qualitäts-merkmal 2
Qualitäts-merkmal n
Qualitäts-teilmerkmal
Qualitäts-teilmerkmal 2
Qualitäts-teilmerkmal 3
Qualitäts-teilmerkmal m
Qualitätsindikatoren



Was bedeutet Software-Qualiät?

Arbeitsblatt Test Driven Development

Einleitung

Durch den Einsatz von passenden Software-Hilfsmittel, lässt sich der Entwickler-Alltag massgeblich vereinfachen. Zusätzlich können Sie uns dabei auch eine grosse Hilfe sein, die Qualität der ausgelieferten Software hochzuhalten. Zu solchen Hilfsmittel gehören Produkte wie ein Versions-Verwaltungssystem, ein Build-Server und Tools zur Test-Automatisierung.

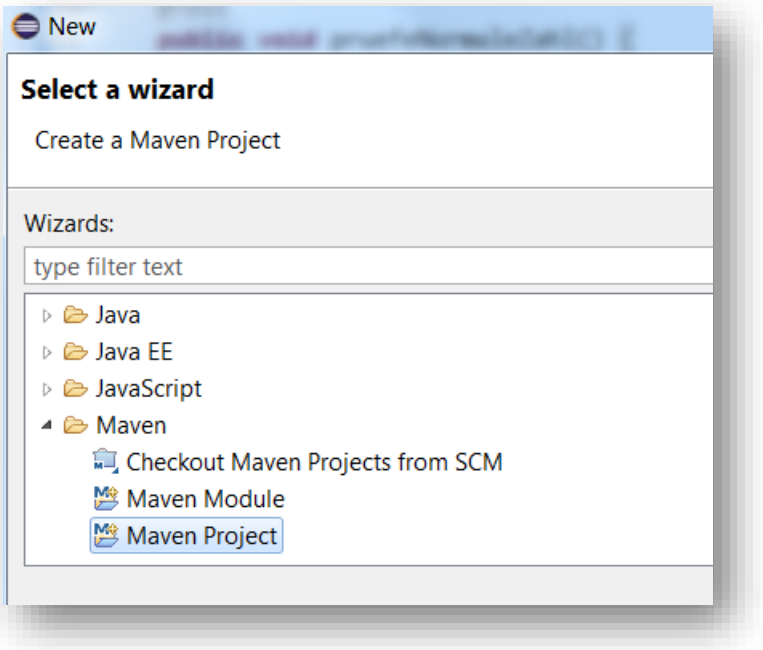
In diesem Arbeitsblatt wollen wir uns mit der Test-Driven Vorgehensweise (Testgetriebene Entwicklung) beschäftigen. Dazu werden wir das Build-Management-Tool Maven und die JUnit Bibliothek einsetzen. Dies erlaubt uns ein Build-Prozess aufzusetzen, bei dem unsere Unit-Test automatisiert ausgeführt werden.

Grundregeln beim Test Driven Developement (TDD)

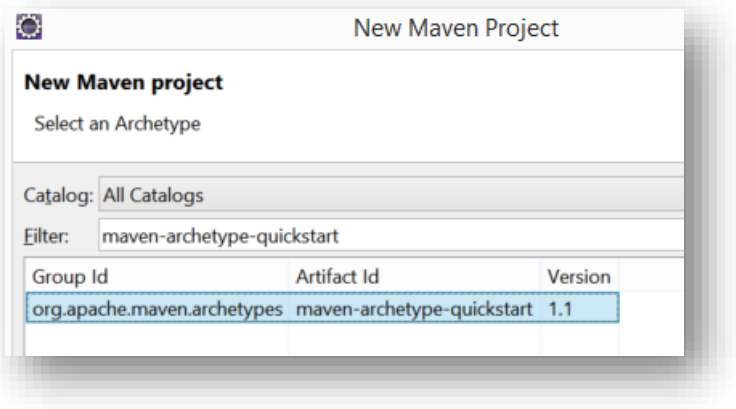
1. Produktiver Code darf ohne fehlschlagende Tests nicht geschrieben werden.
2. Nicht mehr Testcode als unbedingt notwendig ist, um einen Fehler anzuzeigen, darf geschrieben werden.
3. Nicht mehr produktiver Code als unbedingt notwendig darf für den Erfolg des fehlgeschlagenen Tests geschrieben werden.
4. Jeder Testfall (Testmethode) sollte nur genau ein Verhalten überprüfen.

Maven-Projekt erstellen

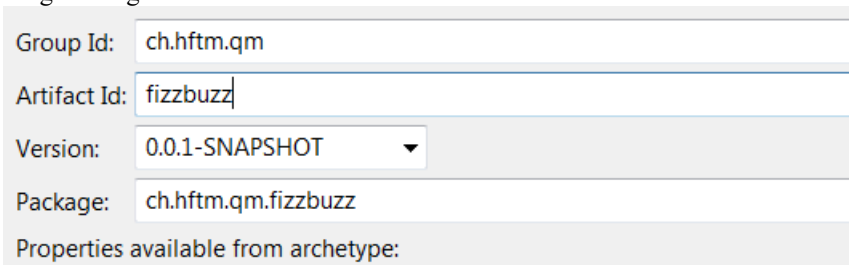
- Prüfe ob deine Eclipse-Umgebung bereits das Plugin m2e (Maven Integration for Eclipse) enthält. Falls nicht, installiere es gemäss http://www.eclipse.org/m2e/.
- Erstelle nun ein neues Maven Projekt aus Eclipse:



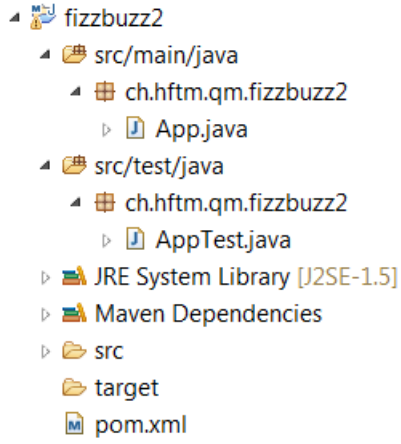
- Wähle das Quick-Start-Template:



- Vergebe folgenden Namen:



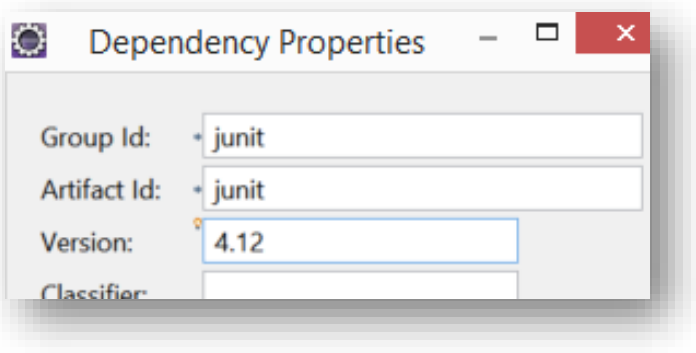
- Wenn du „Finish“ wählst, wird ein folgendes Java-Projekt für dich erstellt:



Maven und JUnit

Bevor wir mit einem ersten Unit-Test beginnen können, müssen wir die JUnit-Library in unser Maven-Projekt aufnehmen.

- Dazu öffnen wir das pom.xml und wählen im Reiter „Dependencies“ unsere junit-Referenz aus und aktualisieren die Version:



- Nach dem Speichern des pom.xml steht dir JUnit bereits in der aktuellen Version in deinem Projekt zur Verfügung. (Nachvollziehbar wenn du in Eclipse das Verzeichnis «Maven Dependencies» öffnest.)

So haben wir bereits ein erster Vorteil von der Arbeit mit Maven kennengelernt. Es kümmert sich für uns um die Verwaltung und das Herunterladen der nötigen Abhängigkeiten/Bibliotheken.

Maven tut aber mehr als das. Es zeichnet sich im Projekt für den ganzen Build-Vorgang verantwortlich. In der Standardkonfiguration führt Maven unsere Unit-Tests dank dem Surefire-Plugin im Build-Verlauf bereits aus. Somit sind wir bereit für unsere Test-Driven Implementation.

Beachte: Damit Maven unsere Unit-Tests automatisch erkennt und startet, müssen wir folgendes beachten:

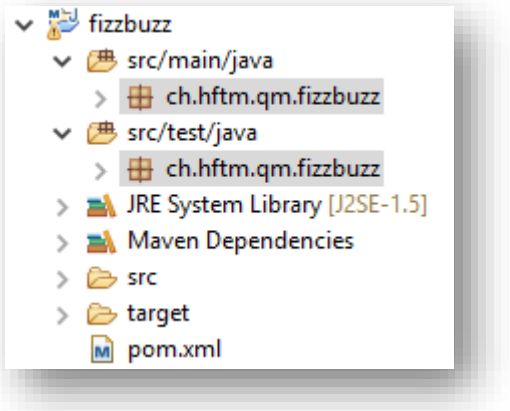
- Die Test-Klassen müssen anstelle von src/main/java im Verzeichnis src/test/java liegen.
- Die Test-Klasse selbst sollte das Wort „Test“ am Anfang oder am Ende der Klasse enthalten. (z.B. TestKlasse oder KlasseTest)

Erster Unit-Test für FizzBuzz

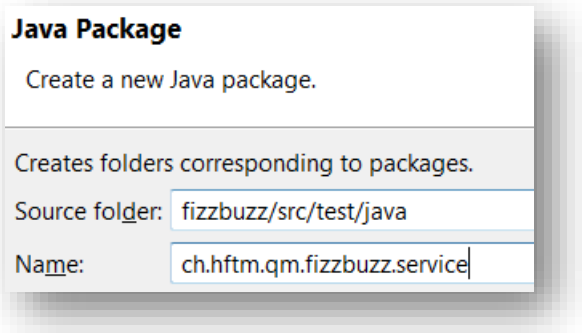
„Fizz-Buzz“ dient als Mittel zum Zweck, Kindern spielerisch die Regeln der Division zu vermitteln. Inhalt dieses Spiels ist es, einen Zähler, der bei 1 anfängt, immer weiter hoch zu zählen und die Zahlen zu nennen. Ist eine Nummer durch 3 teilbar, wird sie durch Fizz ersetzt, wenn sie durch 5 teilbar ist, durch Buzz. Folglich wird die Zahl, wenn sie durch beides teilbar ist, zu FizzBuzz. Also, wie beschrieben: [1, 2, 3, 4, 5] wird dann zu [1, 2, Fizz, 4, Buzz]. Das Spiel könnte endlos so fortgeführt werden.

Im Rahmen dieses Arbeitsblattes wollen wir die Methode `FizzBuzzService.getResultOfNumber(int number)` implementieren, welcher wir eine Int-Zahl übergeben und wir anschliessend Fizz, Buzz, FizzBuzz oder die Zahl als String zurück als Resultat erhalten. Wir werden diese Methode in der im Paket `ch.hftm.qm.fizzbuzz.service` ablegen. Dabei gehen wir nach Test-Driven-Development vor und erstellen einen Unit-Test, bevor wir eine andere Code-Zeile für unseren Service schreiben.

- Lösche zuerst einmal die bestehenden Packages:



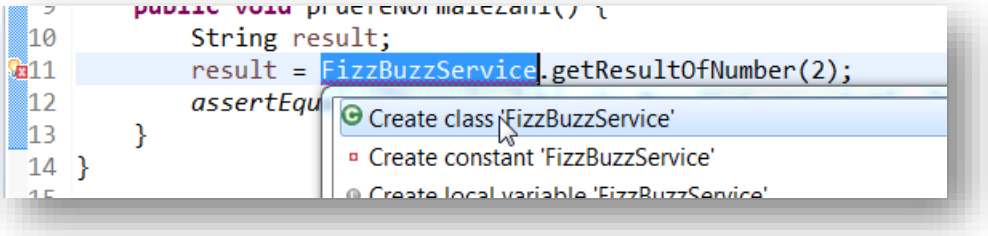
- Ok, da wir mit einem Test loslegen möchten, brauchen wir zuerst ein Package im test-Verzeichnis. Erzeuge also im src/test/java Source-Folder ein Package mit dem Namen `ch.hftm.qm.fizzbuzz.service`:



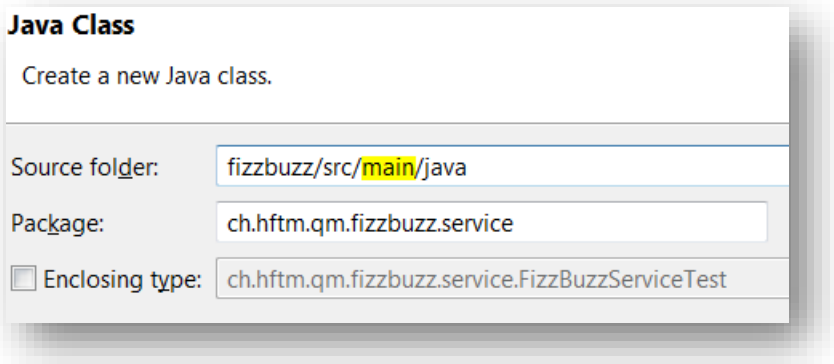
- Jetzt erstelle darin die Klasse „FizzBuzzServiceTest“ mit einem ersten Test. Damit der Test als jUnit-Test erkannt wird und als Unit-Test ausgeführt werden kann, müssen wir die Methode mit `@Test` annotieren.
  Das Ziel dieses ersten Testes ist es eine normale Zahl zu prüfen, welche nicht durch drei oder fünf teilbar ist. Somit sollte beim Übergeben der Zahl 2 ein String mit dem Wert «2» als Resultat zurückkommen. Dies wird mit der Methode assertEquals aus der jUnit-Bibliothek zum Abschluss des Testes überprüft.

```
package ch.hftm.qm.fizzbuzz.service;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class FizzBuzzServiceTest {
    @Test
    public void pruefeNormaleZahl() {
        String result;
        result = FizzBuzzService.getResultOfNumber(2);
        assertEquals("Normale Zahl als Resultat erwartet", "2", result);
    }
}
```

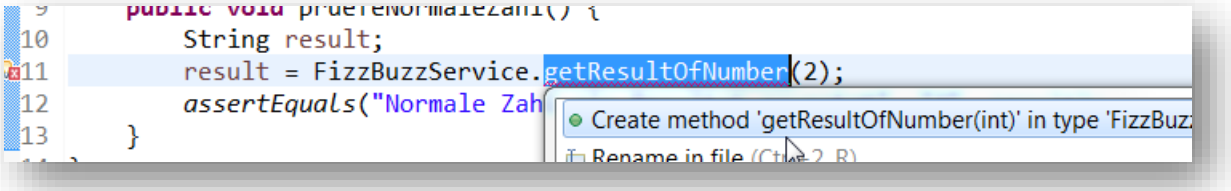
- Mit einem Klick mit der rechten Maustaste auf die neue Test-Klasse, dann unter «Run As / jUnit Test» können wir den Test ein erstes Mal ausführen. Wie Eclipse bereits für uns anzeigt, kann der Source-Code noch nicht kompiliert werden und somit verläuft auch der Test natürlich nicht erfolgreich.
- Da wir nun einen fehlerhaften Test-Fall haben, dürfen wir mit dem Implementieren der Service-Schicht beginnen. Am einfachsten geht dies, wenn wir die uns die fehlende Klasse über Eclipse generieren lassen.



- Wichtig ist nun, den Source-Folder «main» für die neue Klasse zu wählen, damit diese nicht im test-Verzeichnis landet:



- Ebenfalls über Eclipse können wir nun die fehlende Methode generieren:



- Mit einer kleinen Anpassung an der Service-Klasse sollte unser Test bereits erfolgreich durchlaufen:

```
package ch.hftm.qm.fizzbuzz.service;

public class FizzBuzzService {

    public static String getResultOfNumber(int number) {
        return Integer.toString(number);
    }

}
```

- Prüfe dies kurz gemäss den Anweisungen auf der letzten Seite.

Ein nächster Test

Mit dem bisherigen Source-Code, haben wir bereits eine erste Anforderung abgedeckt, nämlich, dass Zahlen die nicht durch drei und fünf teilbar sind, einfach als String-Zahlen zurückkommen. Nun fehlt uns also noch folgendes:

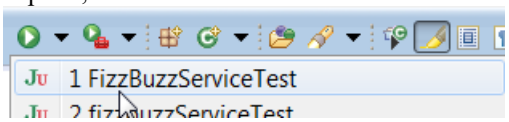
- ☐ Zahlen die durch drei teilbar sind
- ☐ Zahlen die durch fünf teilbar sind
- ☐ Zahlen die durch fünf und drei teilbar sind

Also folgend wir dieser Liste und erstellen den nächsten Test:

- Erstelle ein neuer Test in deiner Test-Klasse im folgendem Rahmen:

```
@Test
public void pruefeDurchDreiTeilbareZahl() {
    String result;
    result = FizzBuzzService.getResultOfNumber(3);
    assertEquals("Fizz bei durch drei teilbare Zahl als Resultat erwartet", "Fizz", result);
}
```

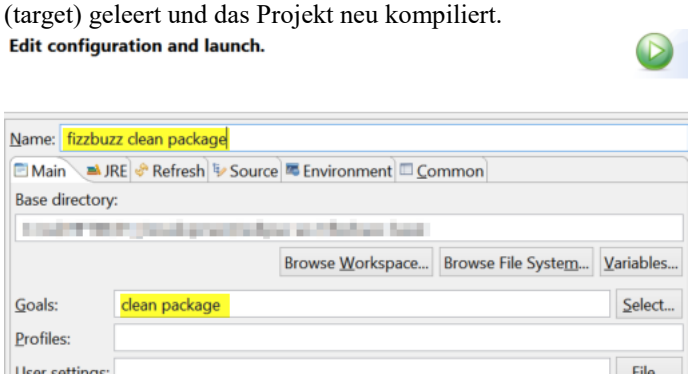
- Beim Speichern stellst du fest, das sich dein Code erfolgreich kompilieren lässt. Bevor du nun den Service anpasst, starte die Test-Klasse als JUnit-Test z.B. wie folgt:



- Hier kannst du nun deine Test-Resultate einsehen.

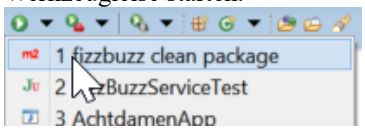
Dank Maven können wir das ganze auch in unserem Build-Prozess nachvollziehen. Klicke dazu mit der rechten Maustaste auf dein Projekt und wähle «Run As / Maven build..»

Setze den Namen und die Goals entsprechend der nächsten Abbildung. Mit „clean package“ wird das Ausgabeverzeichnis (target) geleert und das Projekt neu kompiliert.



Mit dem Klick auf „Run“ wird der Kompilierungsvorgang ausgeführt und anschliessend auch die Unit-Tests gestartet. In unserem Beispiel schlägt der Maven-Prozess fehl, weil noch nicht alle Unit-Tests erfolgreich verlaufen. Versuche den Umstand zu korrigieren und stelle sicher, dass der „clean package“-Prozess erfolgreich durchläuft.

Übrigens: In Zukunft kannst du die erstellte Konfiguration für „clean package“ direkt über folgendes Icon in der Werkzeugleiste starten.



FizzBuzz-Service fertigstellen

Setze nun die Arbeit in dieser Manier fort. Denke daran, nur produktiven Code schreiben zu dürfen, sofern ein Unit-Test dies benötigt. Sobald du eine funktionierende FizzBuzz-Methode hast, erstellst du eine weitere, die einen gewünschten Bereich von Zahlen-Werten aneinander hängen kann. Wenn der Benutzer zum Beispiel den Zahlenbereich 6-11 wünscht, erscheint der Rückgabe-Wert „Fizz, 7, 8, Fizz, Buzz, 11“.

Möglichkeiten zur Vertiefung für Fortgeschrittene

Eine Vertiefung in die beiden Themen ist vorallem für Software-Entwickler empfohlen und vielleicht auch gerade für die Diplomarbeit sehr spannend. Gerne stellt der Dozent die Themen auf Wunsch auch genauer vor.

- GitLab CI
  Wir haben im Kurs bereits über Continuous Integration und Delivery gesprochen. Dank GitLab kannst du Continuous Integration für dieses Projekt ganz leicht integrieren:
  - Stelle das Projekt auf GitLab
  - Erzeuge im Root-Verzeichnis des Projektes eine Text-Datei mit dem Namen «.gitlab-ci.yml» mit folgendem Inhalt:
    ```
    maven-test:
      stage: test
      script:
        - mvn test
    ```
    Damit wird ein Job mit der Bezeichnung «maven-test» definiert im dem die Unit-Tests bei jedem Push auf GitLab durch GitLab CI ausgeführt werden. (Mehr zu GitLab CI erfährst du unter https://dev.hftm.ch:4430/doc/gitlab/blob/master/pages/31_GitLabCI.md)
- Beispiel mit Integrationstests
  Ein ausführliches FizzBuzz Lösungsbeispiel inkl. Integrationstest, Mocking und GitLab CI findest du unter https://dev.hftm.ch:4430/swq/fizzbuzz-maven-ci

2511 Softwarearchitektur, Software-Qualitätsmanagement

06 Code-Analyse Hilfsmittel

06 Code-Analyse Hilfsmittel
von Simeon Liniger - Montag, 19. September 2016, 11:00

Einleitung

Moderne Entwicklungsumgebungen nehmen uns bei der täglichen Arbeit beim Schreiben von Source-Code bereits viel Arbeit ab, in dem Sie uns durch eine Syntax- und Semmantik-Analyse den Source-Code auf Unstimmigkeiten prüfen. So ermöglicht auch die im Studium eingesetzte Entwicklungsumgebung Eclipse bereits im Standard eine Konformitätsanalyse von Java-Source-Code.

Wir wollen in dieser Übung nun einige Hilfsmittel kennenlernen, die über die Standard-Funktionalität von Eclipse hinausgehen. Als Gruppe dürft ihr dabei eines der folgenden Tools genauer betrachten und der Klasse vorstellen:

- Eclipse Plugins
  - EclipseMetrics
  - FindBugs
  - eclipse pmd
  - eclipse-cs Checkstyle
  - EclEmma (nur für Gruppen, die schon Erfahrung mit Unit-Testing haben)
- Visual Studio Plugins
  - JetBrains Resharper
  - StyleCop
- Weitere Hilfsmittel
  - Manuelle Code Reviews mit GitLab Merge Requests

heise Developer

Architektur/Methoden 08.05.2014 - 12:11

Alexander Birk, Christoph Lukas

Eine Einführung in Continuous Delivery, Teil 1: Grundlagen

Der Reihe nach

Continuous Delivery verspricht Produktivsetzungen der entwickelten Software auf Knopfdruck bei besserer Qualität. Was sind die zentralen Ideen dahinter und welche Voraussetzungen muss man schaffen, um damit erfolgreich zu sein? In einer mehrteiligen Artikelserie sollen zusätzlich zu Hintergrundinformationen ganz konkrete Schritte auf dem Weg zu einer Continuous Delivery Pipeline beschrieben werden.

Softwareentwicklung nach einem klassischen Phasenmodell sieht die drei Schritte Entwicklung, Qualitätssicherung und Auslieferung üblicherweise nur einmal für jedes Release vor: Nach Abschluss einer monatelangen Entwicklungsphase werden die für das nächste Release vorgesehenen Änderungen der Software an die Qualitätssicherung übergeben. In der anschließenden Testphase untersucht man diese dort auf die Erfüllung der Anforderungen hin und bessert bei Abweichungen nach. Danach lässt sich der Softwarestand auf dem Produktionssystem installieren.

Von der Idee bis zum Kunden - Continuous Delivery ist ein Teil dieser Kette (Abb. 1)

In den letzten Jahren sind viele Unternehmen zu einem inkrementell-iterativen Vorgehen übergegangen, bei dem auch Zwischenstände der entwickelten Software schon qualitätsgesichert und an den Kunden ausgeliefert werden. Die Kette aus Entwicklung, Qualitätssicherung und Lieferung wird also bereits mehrfach im Entwicklungsprozess der Software durchlaufen.

Was aber, wenn die Software mit agilen Methoden entwickelt wird, bei denen eine Lieferung lauffähiger Software im Rhythmus von nur wenigen Wochen zum Fundament der Methode gehört? Ein manueller Test des gesamten Funktionsumfangs alle zwei Wochen und eine anschließende Lieferung werden hier schnell zum Showstopper: Testumgebungen stehen nicht in genügender Anzahl zur Verfügung, die Installation der Software ist vom Betriebsteam durchzuführen, die Konfiguration der Anwendung auf der Testumgebung passt nicht und erfordert mühsame Abstimmungen zwischen Entwicklung und Betriebsteam. Erst danach kann das eigentliche Testen beginnen. Wie passt das in einen Zeitrahmen von nur zwei Wochen?

Continuous Delivery versucht, genau diese Probleme durch radikale Automatisierung zu beheben. Die Schritte Entwicklung, Qualitätssicherung und Auslieferung der Software werden in viel kleineren Einheiten abgebildet: Bei Änderungen oder Erweiterungen an der Software wird dieser Softwarestand direkt im Anschluss automatisiert getestet. Die Tests liefern schnelles Feedback über Seiteneffekte und Regressionen. Sind sie erfolgreich, lässt sich der Softwarestand auf Knopfdruck und innerhalb von Minuten auf dem Produktionssystem installieren. Die Kette aus Entwicklung, Qualitätssicherung und Produktivsetzung erfolgt kontinuierlich mit jeder Änderung an der Software und führt zu einer automatisierten Pipeline: Auf der einen Seite kommt jeweils ein neuer Softwarestand hinein, auf der anderen ein qualitätsgesichertes, produktionsreifes und installierbares Softwarepaket heraus.

Erste Ideen, den Software-Entwicklungsprozess in diese Richtung zu verändern, haben Jez Humble, Chris Read und Dan North bereits auf der Agile 2006 in einem **Vortrag [1]** vorgestellt, die bisher ausführlichste Auseinandersetzung mit dem Thema findet sich im 2010 erschienen Buch "**Continuous Delivery [2]**" von Humble und David Farley.

Jede Sourcecode-Änderung triggert die Continuous Delivery Pipeline (Abb. 2)

In der Continuous Delivery Pipeline wird die Qualitätssicherung durch eine Kombination von automatischen und manuellen Tests oder Freigabeschritten realisiert. Man versucht, sowohl funktionale als auch nichtfunktionale Anforderungen durch unterschiedliche Testtypen mit hohem Automatisierungsgrad zu validieren:

- Unit-Tests prüfen einzelne Komponenten isoliert in ihren Funktionen.
- Akzeptanztests sorgen für das Einhalten der mit den Anforderungen formulierten Akzeptanzkriterien.
- Performancetests oder statische Codeanalyse überprüfen nichtfunktionale Anforderungen.

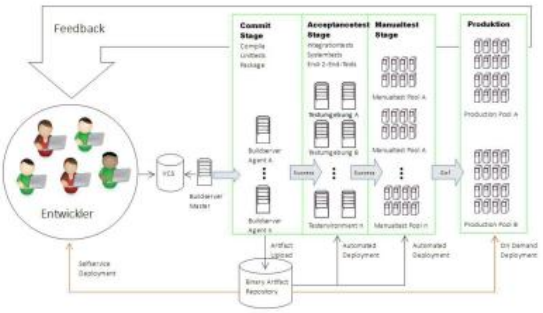
Die Tests werden in mehreren Stufen organisiert und diese nacheinander für jeden Softwarestand ausgeführt. Nur wenn die eine Teststufe erfolgreich war, starten die Tests der nächsten Stufe überhaupt. Der Ablauf folgt hier dem als "Stop the Line" bekannten Prinzip des Lean Manufacturing. Je weiter es ein Softwarestand erfolgreich durch die Pipeline geschafft hat, desto mehr kann man darauf vertrauen, dass die Änderung keine Regressionen erzeugt hat und die Software nach wie vor die Anforderungen erfüllt.

Stufenweise heranführen

Die erste Teststufe in der Continuous Delivery Pipeline ist die sogenannte Commit Stage, die das Versionskontrollsystem direkt durch die Änderung der Software ausgelöst hat. Hier werden die Komponenten der Software gebaut und die jeweiligen Unit-Tests ausgeführt. War diese Teststufe erfolgreich, kommt es in der zweiten Teststufe, der sogenannten Acceptance Test Stage, zur Ausführung der Integrations-, Akzeptanz- und Systemtests. Im Anschluss an diese Phase können dann zusätzliche manuelle Tests oder Freigabeschritte angebunden sein.

Auf dem Weg durch die Commit und Acceptance Test Stage werden die Ergebnisse der automatisierten Tests kontinuierlich als Feedback an die beteiligten Entwickler zurückgemeldet. Das kann in Form von E-Mails, Instant Messages oder "**Extreme Feedback Devices [3]**" geschehen. Das erste

Feedback aus der Pipeline signalisiert den Entwicklern innerhalb weniger Minuten, ob größere Regressionen gefunden wurden. Die schnelle Rückmeldung hilft dabei, Fehler frühzeitig zu entdecken und zu beheben. Da die Pipeline für jede Änderung an der Software gestartet wird, bezieht sich das Feedback aus der Pipeline immer genau auf eine einzelne Änderung der Software und ermöglicht somit einen direkten Rückschluss von der gefundenen Regression auf die auslösende Änderung.



The Big Picture: Die Continuous Delivery Pipeline aus der Vogelperspektive (Abb. 3)

Wie sieht die technische Umsetzung von Commit und Acceptance Test Stage bei der Realisierung einer Continuous Delivery Pipeline aus? Die einzelnen Stufen der Pipeline lassen sich mit einem Build-Server abbilden, auf dem die gewünschten Aufgaben in mehreren Jobs modelliert werden. In den Jobs der Commit Stage werden auf ihm die Komponenten der Software zuerst kompiliert und im Anschluss die jeweiligen Unit-Tests ausgeführt. Bei erfolgreichem Durchlauf fasst man jetzt die erzeugten Binärartefakte der Softwarekomponenten zum Paket oder Bundle zusammen. Dieses Bundle wird in einem Repository abgelegt und ab diesem Moment ausschließlich für den weiteren Durchlauf des Softwarestandes durch die Pipeline genutzt. Will man diesen Stand später auf dem Produktionssystem installieren, kommt dafür ebenfalls genau dieses Bundle zum Einsatz.

Hier werden zwei wichtige Prinzipien von Continuous Delivery deutlich:

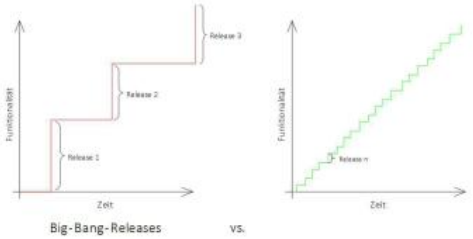
- Das Testen jeder Änderung an der Software.
- Der Softwarestand wird nur ein einziges Mal an zentraler Stelle gebaut.

Sie gewährleisten, dass man die Testergebnisse konkret einzelnen Änderungen an der Software zuordnen kann und vor allem eine komplette Rückverfolgbarkeit von den Binärartefakten in der Produktion bis zur konkreten Sourcecode-Version gewährleistet ist.

Nach erfolgreichem Abschluss der Commit Stage wird das Bundle auf dem Build-Server in die Acceptance Test Stage übergeben. Hier führt dieser verschiedenartige Akzeptanztests aus: Integrationstests, die das Zusammenspiel mehrerer Komponenten testen, und Systemtests, die die Software als Ganzes aus Sicht des Benutzers prüfen. Zusätzlich kann man hier auch weitere Tests integrieren, die nichtfunktionale Anforderungen wie die Performance der Software untersuchen. Für die Ausführung dieser Akzeptanztests holt der Build-Server jeweils das in der Commit Stage erzeugte Bundle aus dem zentralen Repository, installiert die Komponenten der Software auf einer geeigneten Testumgebung und startet dann die Tests gegen oder auf dieser Umgebung. Im Anschluss spielt er die Auswertung der Testergebnisse als Feedback an die Entwickler zurück.

Hat der Softwarestand die Acceptance Test Stage erfolgreich passiert, können weitere Pipeline-Schritte mit manuellen Tests erfolgen. Auch hierfür wird der Softwarestand aus einem Bundle auf eine Testumgebung installiert. Hat er alle nötigen Teststufen passiert, lässt sich das entsprechende Bundle "auf Knopfdruck" auf dem Produktionssystem installieren.

Der Aufwand, eine solche Continuous Delivery Pipeline erfolgreich für ein Projekt umzusetzen, ist nicht unerheblich. Was gewinnt man dadurch?



Kleine Deltas - Release wird zum "Non-Event" (Abb. 4)

Wo gewinnen Projekte mit Continuous Delivery?

Die meisten können sicherlich spontan eine ganze Reihe von Softwareprojekten anführen, die mit Pauken und Trompeten gescheitert sind: LKW-Maut, ObamaCare et cetera, um nur einige Beispiele zu nennen. Ohne bei den genannten Projekten die wirklichen Gründe zu kennen, lässt sich sagen, dass je größer der Umfang eines einzelnes Software-Releases ist, desto höher leider auch das Risiko ist, dass hierbei etwas schiefläuft. Es kommt zum Beispiel häufig vor, dass der Funktionsumfang sich nur schlecht manuell testen lässt, die Auswirkung von Änderungen auf die Performance nicht früh genug beachtet wurden oder das Zusammenspiel verschiedener Komponenten und deren Lastverhalten zu wenig Berücksichtigung beim Testen gefunden haben.

Continuous Delivery entschärft diese Situation erheblich, indem der gesamte Release-Prozess automatisiert und damit drastisch verschlankt wird. Durch den Wegfall der vielen manuell auszuführenden Aufgaben lassen sich bereits kleine Gruppen von Änderungen produktiv einsetzen. So sinkt das Risiko, dass etwas schiefgeht. Und sollte das doch einmal passieren, stellt auch ein Rollback kein großes Problem mehr dar. Firmen mit Continuous-Delivery-Erfahrung tendieren in einem solchen Fall sogar dazu, eher den in Produktion aufgetretenen Bug zu fixen und sofort eine korrigierte Version der Software auszurollen, statt auf eine ältere Version zurückzugehen (Rollforward statt Rollback).

Die Entwicklung einer Software stellt außerdem immer auch eine Vorleistung dar: Zuerst wird ein Konzept entwickelt, dann die eigentliche Software erstellt, getestet und vielleicht ein halbes Jahr nach dem Projektstart produktiv gesetzt. Erst in dem Moment zeigt sich in den meisten Fällen, ob sich mit der Software wirklich das gewünschte Ziel erreichen lässt. Wird die Software so vom Kunden angenommen oder hat man ein halbes Jahr lang an den Wünschen des Kunden vorbei entwickelt?

Da man kleine Features innerhalb von Tagen bis zum Kunden bekommt, lässt sich auch experimentieren: Zeigen die Benutzer Interesse am Feature? Lohnt es sich, hier weiter zu investieren? Wird das Feature besser in dieser oder besser in jener Variante angenommen? Die Durchlaufzeit von der Idee bis zur Produktivsetzung neuer Features lässt sich mit Continuous Delivery von mehreren Monaten zu Tagen oder gar Stunden reduzieren. Die Software bringt deutlich eher Geld als früher, und man kann endlich schnell auf Änderungen am Markt reagieren.

Für die Entwickler entsteht durch den Einsatz von Continuous Delivery eine neue Kategorie von Feedback. Innerhalb von Minuten gibt es eine Rückmeldung darüber, ob durch die letzte Änderung womöglich an ganz anderer Stelle im Code etwas kaputt gegangen ist. Das gibt ein Gefühl von Sicherheit beim Entwickeln und erlaubt, auch ohne Angst kontinuierlich zu refaktorisieren und so die Codequalität auf einem konstanten Niveau zu halten. Das Feedback über die Softwarequalität steht dabei nicht nur den Entwicklern zur Verfügung. Auch die Product Owner können so schnell im Blick haben, ob alle Akzeptanztests noch erfolgreich durchlaufen.

Wenn in der Entwicklung konsequent verschiedenartige Tests zu jedem neuen Feature dazugehören und diese im Sinn von Continuous Delivery mit jeder Änderung an der Software ausgeführt werden, fallen viele Fehler und Regressionen früh auf. Hier kann Continuous Delivery also Sparpotenzial bescheren: Denn in Produktion entdeckte Fehler kosten häufig ein Vielfaches. Test und Testbarkeit rücken mehr in den Fokus. Das heißt, beim Schreiben von Tests fällt auf, ob der Code testbar ist oder nicht. Ist er es nicht, findet gleich eine Qualitätsverbesserung statt, zum Beispiel durch höhere Entkopplung oder Verringerung von Abhängigkeiten.

Herausforderungen Technik und Entwicklungskultur

Möchte ein Unternehmen Continuous Delivery in einem Projekt etablieren, gilt es auf der technischen Seite zuerst einmal, die Sicherung von Softwarefunktionalität durch Tests im Entwicklungsprozess zu verankern. Das Erstellen von Tests auf den verschiedenen Teststufen muss integraler Bestandteil der Softwareentwicklung sein. Das ist leichter gesagt, als getan: Das Know-how zum Schreiben von Tests ist in den Teams zu verteilen, Frameworks für die Testerstellung sind zu evaluieren, die erstellten Tests müssen unabhängig voneinander sein, die Laufzeiten müssen im Blick behalten werden. Für alle Projektbeteiligten hat die Zeit für das Schreiben von Tests zur Entwicklungszeit eines Features dazuzugehören. Selbst dann, wenn die Zeit bis zu einem fixen Termin knapp wird.

Ist diese Hürde genommen, dreht sich auf der technischen Seite viel um Automatisierung: Die Ausführung der Tests ist zu automatisieren, genauso wie die Bereitstellung der Testdaten. Die Tests müssen zu einer Continuous Delivery Pipeline zusammengebaut werden. Spätestens bei der Umsetzung der Acceptance Test Stage wird es dann richtig spannend. Hier geht kein Weg daran vorbei, auch das komplette Deployment der gesamten Software zu automatisieren und gleichzeitig eine sinnvolle Konfiguration der Software auf der Testumgebung bereitzustellen.

Gerade die beiden Aufgaben "Automatisierung des Deployments" und "Konfigurations-Management" zeigen ein neues Problem auf: Bisher waren die Installation der Software und die Bereitstellung und Konfiguration von Test- (und Produktiv-)Umgebungen kein Thema für das Entwicklungsteam. Wenn die Entwicklung abgeschlossen war, wurde die Software an das Betriebsteam übergeben, das dann die Installation auf einem Testsystem durchgeführt hat. Funktionierte die Software dort nicht wie erwartet, begann oft das bekannte Ping-Pong-Spiel zwischen den Abteilungen.

Die Entwickler vermuten den Fehler in der Konfiguration auf der Testumgebung, das Betriebsteam sieht nur wieder einen der unzähligen Entwickler-Bugs. Logfiles werden hin- und hergeschickt, bis nach einer viel zu langen Zeit der tatsächliche Fehler gefunden ist. So kann Continuous Delivery nicht funktionieren. Jeder Softwarestand ist beim Lauf durch die Pipeline zigmal auf einer Testumgebung zu installieren, und die dort ausgeführten Tests müssen zuverlässige Ergebnisse liefern. Das kann nur funktionieren, wenn Entwicklung und Betrieb das Deployment und die Konfiguration der Anwendung gemeinsam planen, anpassen und warten.

Hier steckt die größte, aber gerne übersehene Herausforderung bei der Einführung von Continuous Delivery. Schafft man es, die "Silogrenzen" zwischen Development und Operations aufzubrechen und die beiden Teams miteinander statt gegeneinander arbeiten zu lassen? Nur wenn es gelingt, eine teamübergreifende Zusammenarbeit zu erreichen und so eine **DevOps-Kultur [4]** zu etablieren, kann man überhaupt mit Continuous Delivery erfolgreich sein. Das "Tool" Continuous Delivery liefert einem allerdings gute Voraussetzungen, um dieses Problem anzugehen. Man könnte sogar sagen: Es zwingt dazu, die Brücke vom Entwickler zu Operations zu schlagen und die beiden Teams zusammenzubringen.

Continuous Delivery macht die Qualität des aktuellen Entwicklungsstandes transparenter als früher. Es wird nicht nur offensichtlich, wer wie oft Änderungen an der Software vornimmt, sondern auch, mit welcher Änderung Regressionen eingebaut wurden. Diese Transparenz können alle Projektbeteiligten als Hilfsmittel zur Verbesserung des Entwicklungsprozesses verstehen. Das muss allerdings nicht so sein. Fehlt es an einer offenen Firmenkultur und gibt es keine "Fail-Safe-Environment", könnten Entwickler dieses Plus an Transparenz auch nur als ein weiteres Kontrollinstrument des Managements empfinden. Auch daran kann die Einführung von Continuous Delivery scheitern.

Sie erfordert außerdem in bereits laufenden Projekten einen gewissen Leidensdruck. Die Implementierung der Automatisierungen erfordert Aufwand, der in dieser Zeit nicht für die Entwicklung neuer Features zur Verfügung steht. Droht ein Projekt jedoch daran zu scheitern, dass das Ausrollen eines neuen Releases viel zu hohe Kosten produziert, oder daran, dass zu viele Fehler erst in Produktion entdeckt werden, kann sich der Aufwand schnell rentieren.

Fazit

Die Entscheidung, in einem Projekt Continuous Delivery einzusetzen, bedeutet in vielerlei Hinsicht einen Bruch mit bisher gewohnten Entwicklungstechniken und Arbeitsabläufen. Statt die für ein Software Release notwendigen Schritte nur selten und mit hohem Zeitaufwand durchzuführen, werden sie weitgehend automatisiert und kontinuierlich durchgeführt. Das ermöglicht eine risikoärmere Entwicklung und eine deutlich kürzere Time-to-Market. Die eingesparte Zeit kommt direkt dem entwickelten Produkt zugute. Allerdings steht dem ein initialer Aufwand für die Einrichtung einer Continuous Delivery Pipeline sowie der Aufwand für Weiterentwicklung und Wartung der Systeme im laufenden Betrieb gegenüber.

Zentrale Voraussetzung für den erfolgreichen Einsatz von Continuous Delivery ist eine gute Zusammenarbeit von Entwicklern und Betriebsteam. Beide Teams müssen lernen, sich für die Belange des jeweils anderen zu interessieren und zu engagieren. Man könnte auch sagen: Ohne aktiv gelebte DevOps-Kultur bleibt Continuous Delivery ein zahnloser Tiger im Entwicklungsprozess.

Begleitet von einer offenen und vertrauensvollen Unternehmenskultur schafft Continuous Delivery aber die Voraussetzungen für eine höhere Motivation bei den Entwicklern und eröffnet neue Möglichkeiten, auf Änderungen am Markt zu reagieren. Continuous Delivery kann so helfen, Potenzial in der Entwicklung und im Geschäftsmodell freizusetzen. (ane)

Alexander Birk und Christoph Lukas
*arbeiten seit 15 Jahren als freiberufliche Entwickler, Coaches und Berater unter dem Namen **pingworks [5]**. Sie unterstützen Unternehmen und Teams bei der Anwendung von agilen Entwicklungsmethoden und der Umsetzung von Continuous Delivery.*

URL dieses Artikels:

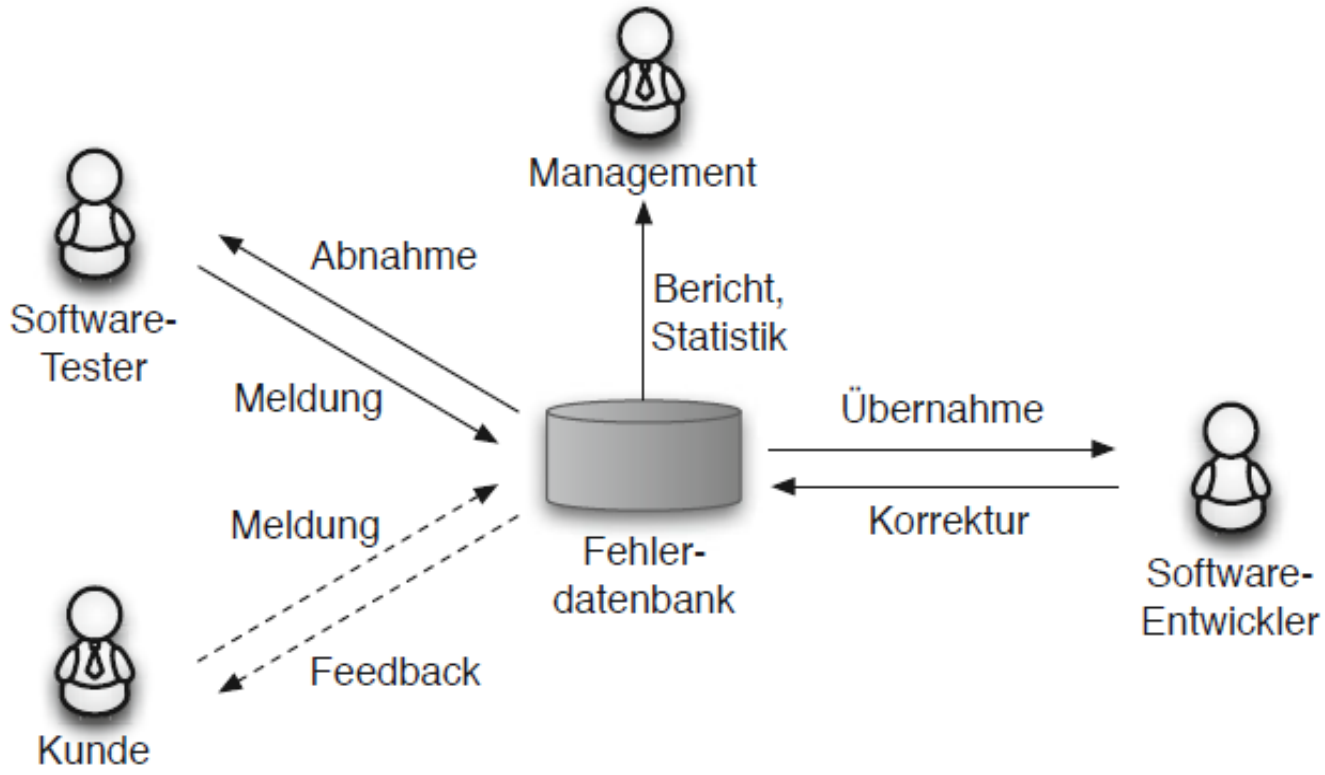
http://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-1-Grundlagen-2176380.html

Links in diesem Artikel:

[1] http://dl.acm.org/citation.cfm?id=1155519
[2] http://my.safaribooksonline.com/9780321670250
[3] http://jenkins-ci.org/node/433
[4] https://en.wikipedia.org/wiki/DevOps
[5] http://www.pingworks.de/?pk_campaign=Heise-CD1&pk_kwd=Author

Bugtracking

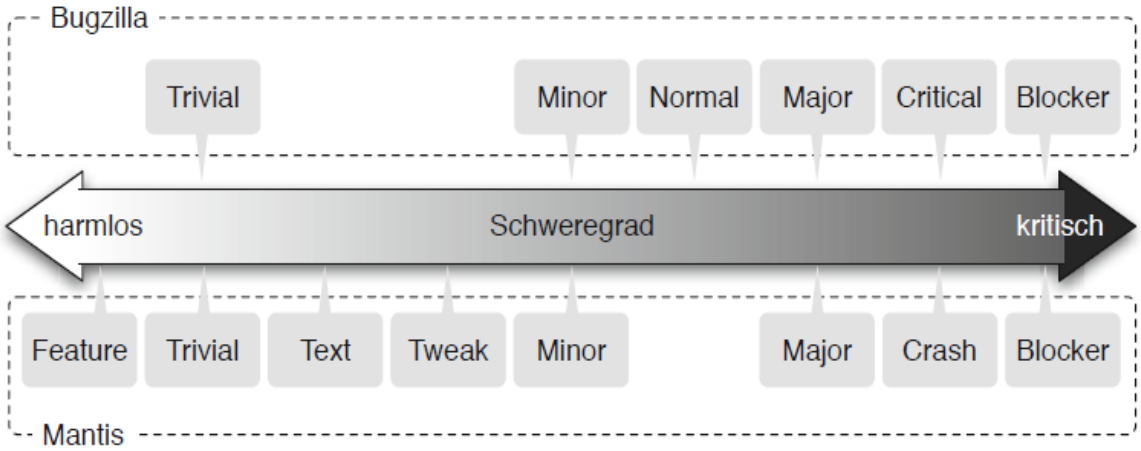
Rollenverteilung Bugtracking-System

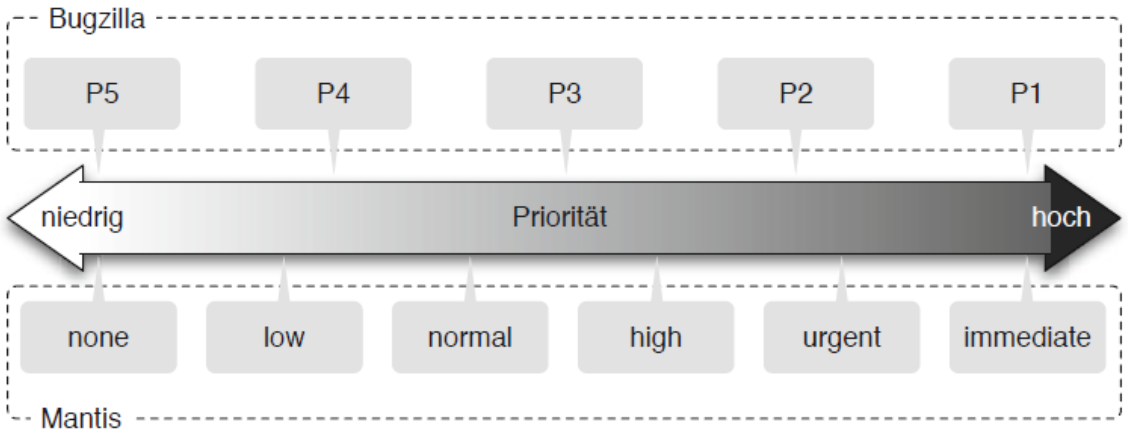


Template-Struktur Bug-Eintrag

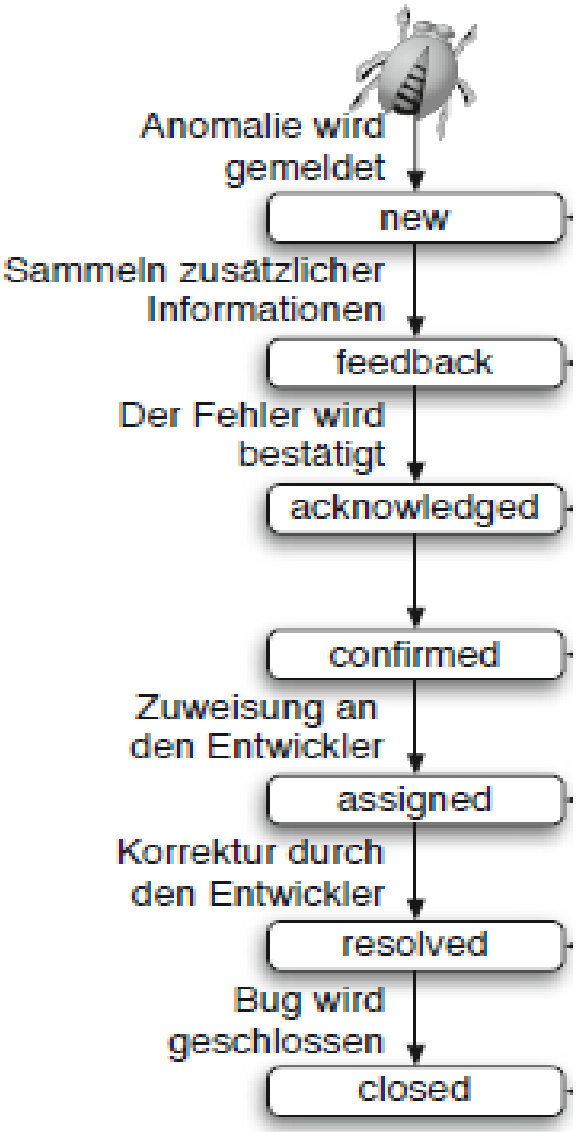
Identifikationsmerkmale	
ID	Eindeutige Identifikationsnummer (automatisch vergeben)
Category	Auf welches Produkt oder Projekt bezieht sich der Fehler?
Reporter	Von wem wurde der Eintrag erstellt?
Date submitted	Wann wurde der Eintrag erstellt?
Last update	Wann wurde der Eintrag zuletzt geändert?
Assigned To	Welchem Mitarbeiter wurde der Fehler zugeteilt?
Klassifikationsmerkmale	
Severity	Wie schwerwiegend ist der Fehler?
Priority	Wie dringlich ist die Korrektur?
View status	Für wen ist der Fehlereintrag sichtbar?
Resolution	Auf welche Weise wurde der Fehler korrigiert?
Status	In welchem Bearbeitungszustand befindet sich der Fehler?
Relationship	Steht der Fehlereintrag in Bezug zu anderen Einträgen?
Beschreibungsmerkmale	
Summary	Kurzbeschreibung / Titel des Fehlers
Description	Ausführliche Beschreibung des Fehlverhaltens
Attachments	Zusätzlich abgelegte Dateien (z. B. Testfälle)

Klassifikation





Zustände Mantis



Rollenmodell Mantis

	Viewer	Reporter	Updater	Developer	Manager	Administrator
View	✔	✔	✔	✔	✔	✔
Report		✔	✔	✔	✔	✔
Monitor		✔	✔	✔	✔	✔
Update			✔	✔	✔	✔
Handle				✔	✔	✔
Assign				✔	✔	✔
Move				✔	✔	✔
Delete				✔	✔	✔
Reopen				✔	✔	✔
Update readonly					✔	✔

Prüfebene

Abnahmeebene

Systemebene

Integrationsebene

Unit-Ebene

Test-Fall

Funktional

Operational

Temporal

Prüfkriterium

Black-box

White-box

Grey-box

Prüfmethodik

Software-Verifikation

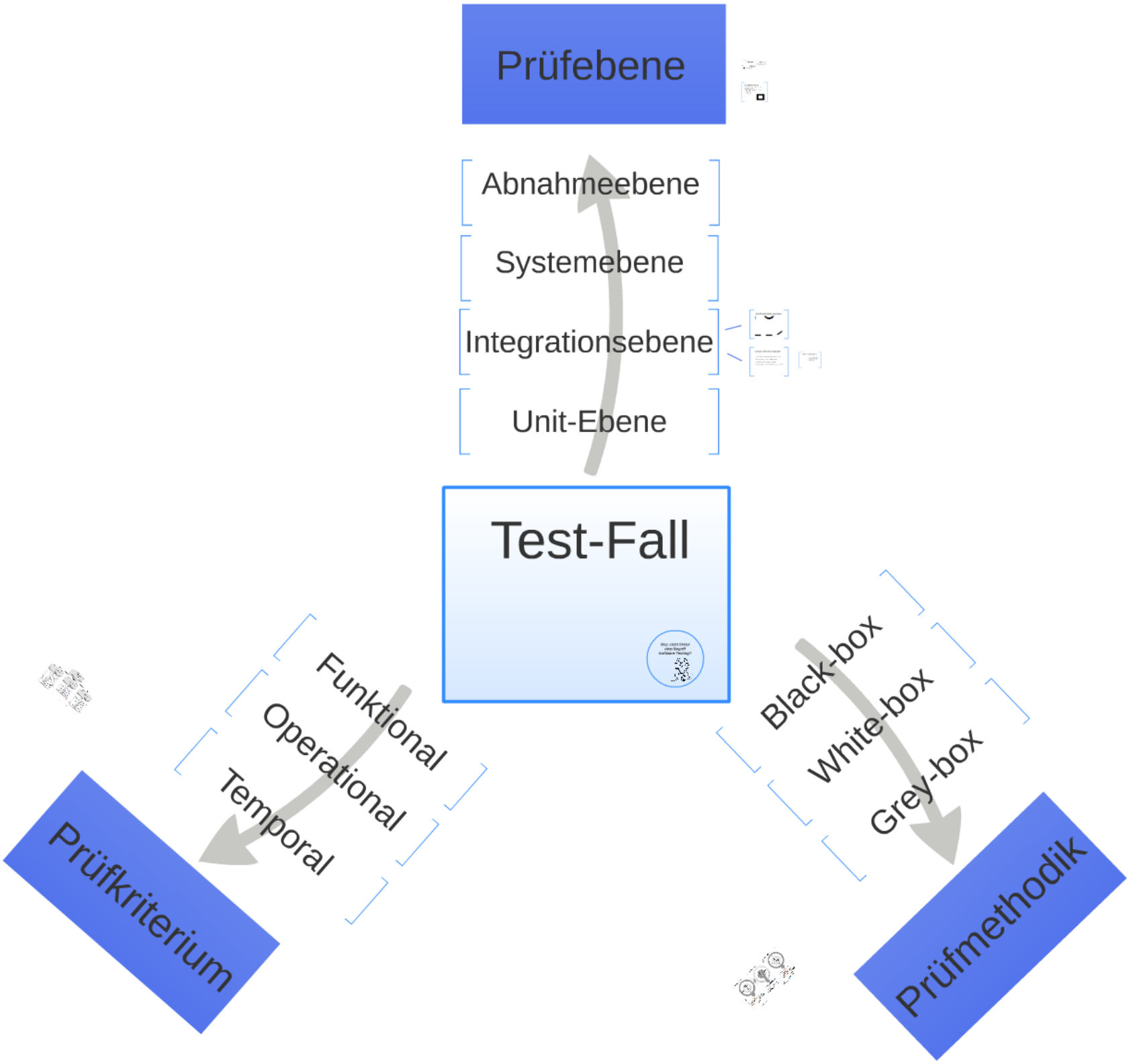
Ebenfalls Analytische Qualitätssicherungsmethode

Ziel: Nachweis zur Erfüllung der Spezifikation mit Hilfe mathematischer Methoden

Was steht hinter dem Begriff Software-Testing?

Test-Fall

Was steht hinter dem Begriff Software-Testing?


Prüfebene
Abnahmeebene
Systemebene
Integrationsebene
Unit-Ebene
Test-Fall
Funktional
Operational
Temporal
Prüfkriterium
Black-box
White-box
Grey-box
Prüfmethodik

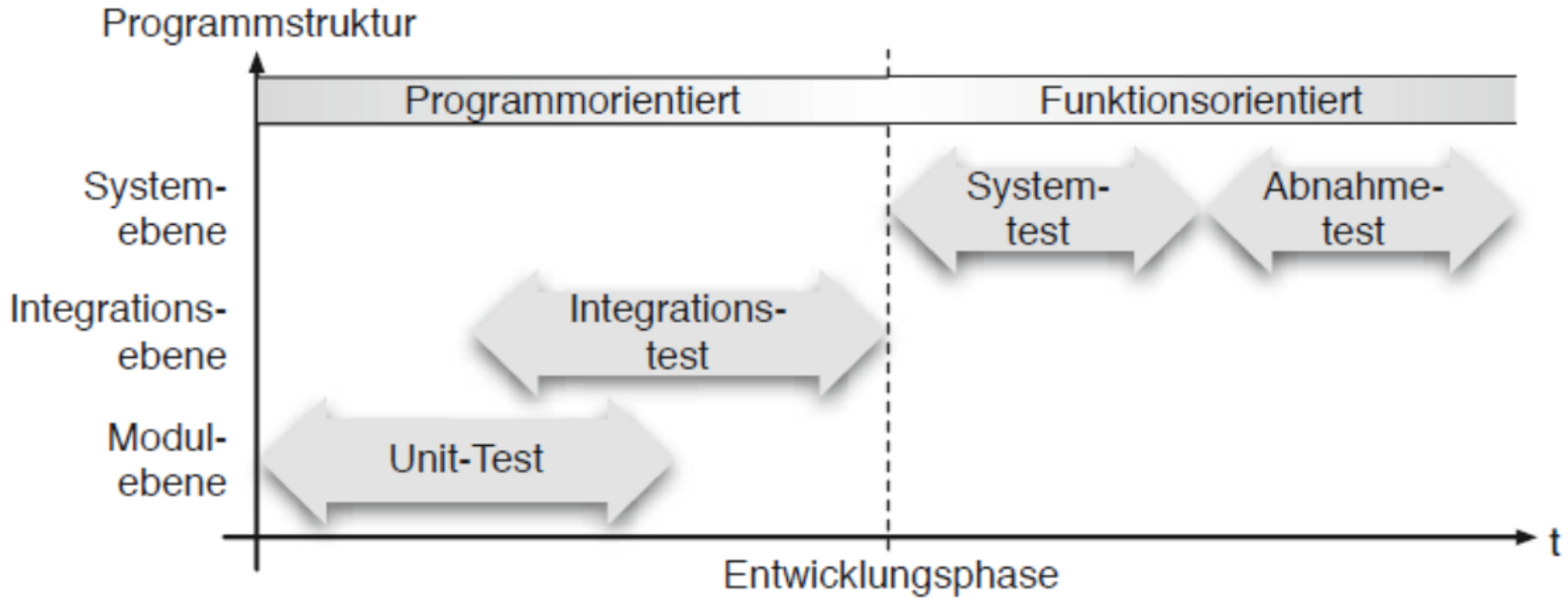
Prüfebene

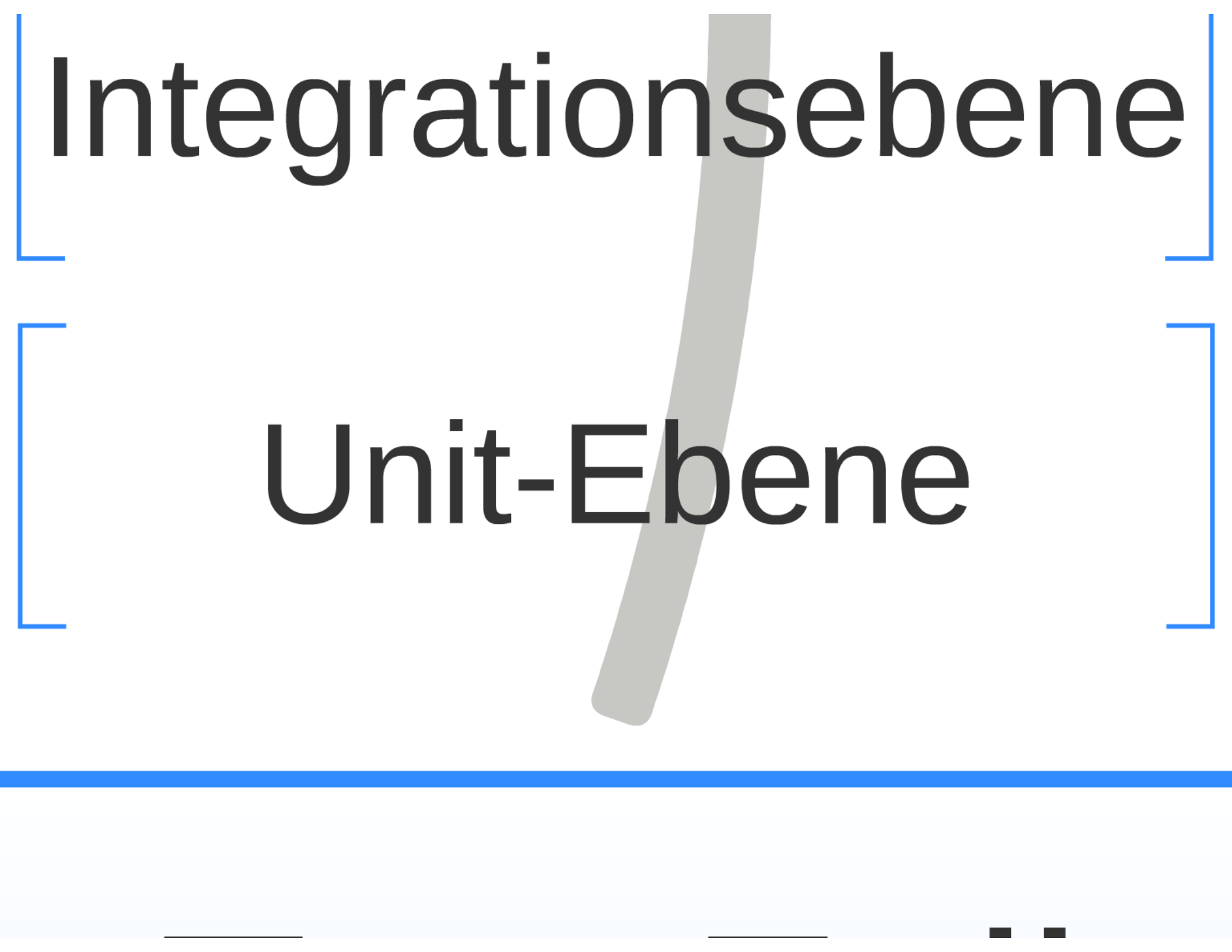
Abnahmeebene

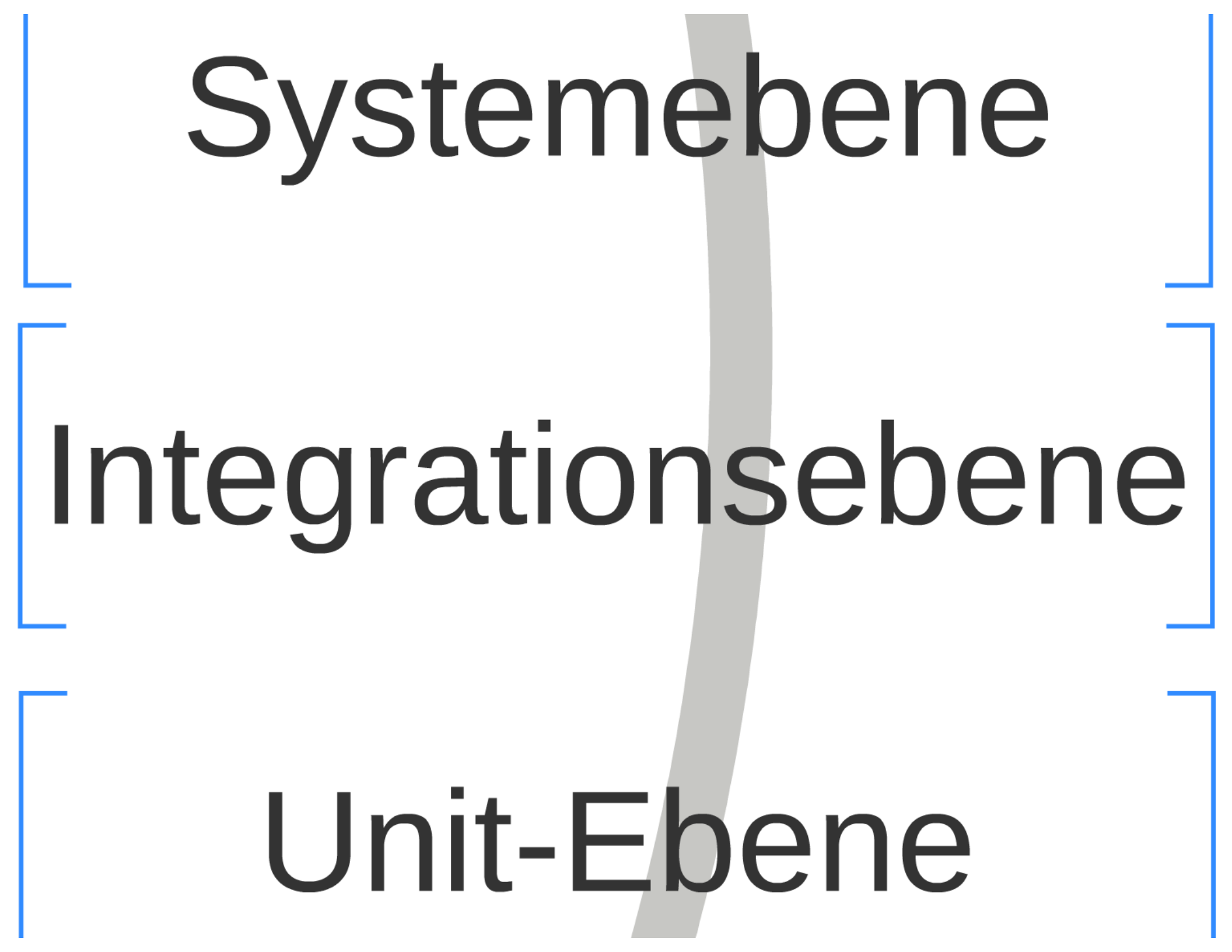
Systemebene

Integrationsebene

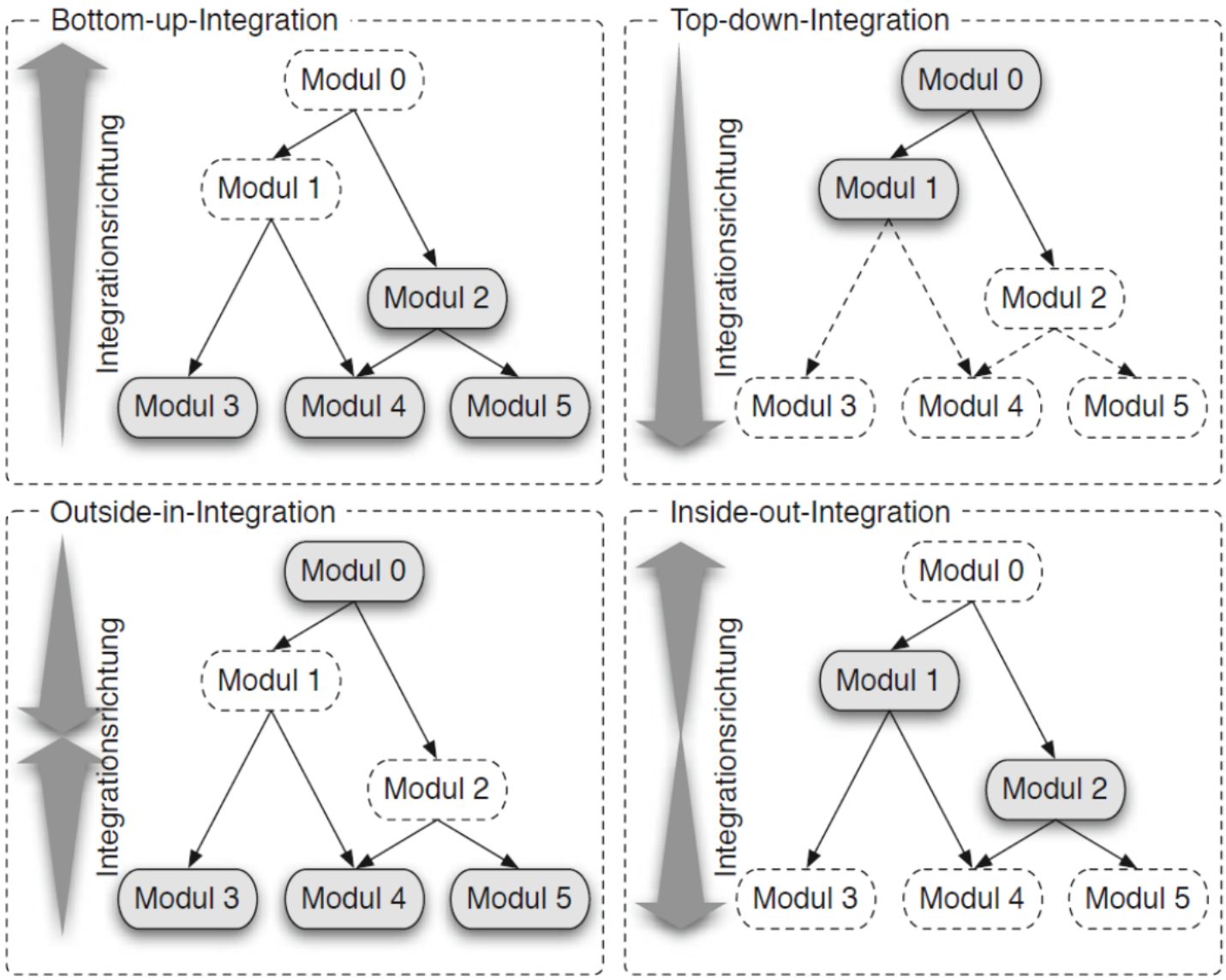
Unit-Ebene


Programmstruktur
Programmorientiert
Funktionsorientiert
System-
ebene
Integrations-
ebene
Modul-
ebene
System-
test
Abnahme-
test
Integrations-
test
Unit-Test
t
Entwicklungsphase


Integrationsebene
Unit-Ebene


Systemebene
Integrationsebene
Unit-Ebene

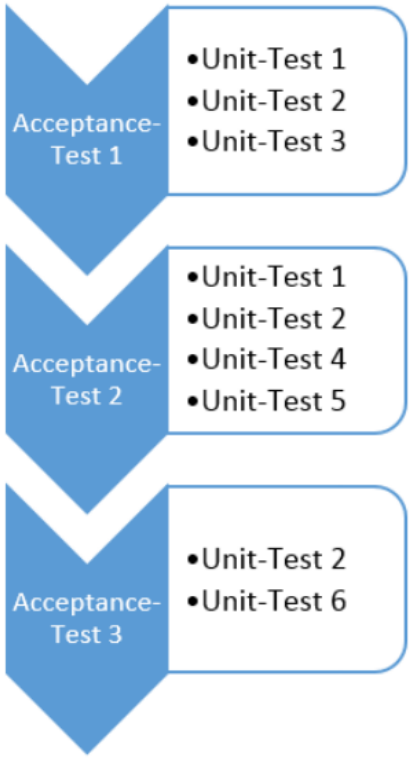
Strukturorientierte Integration



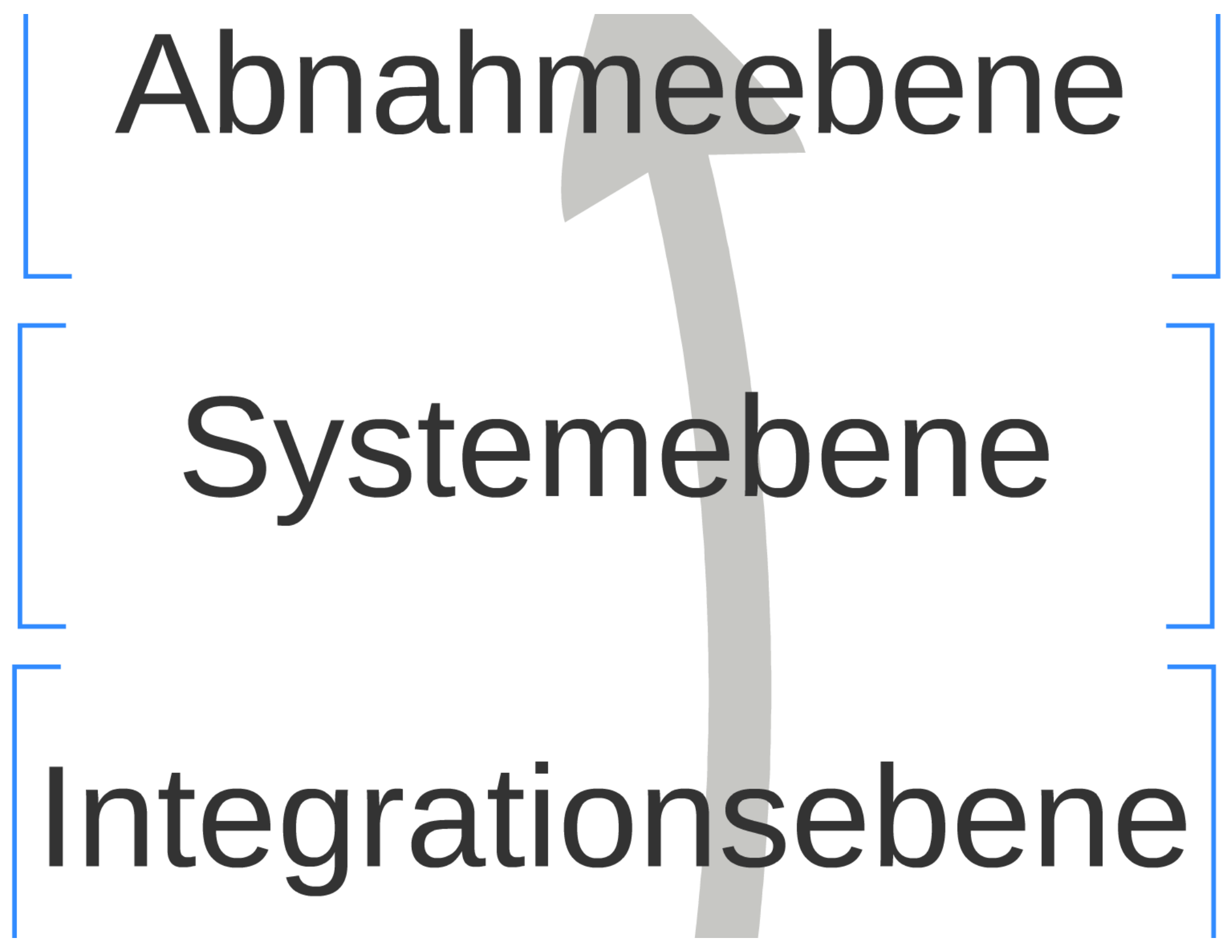
Funktionsorientierte Integration

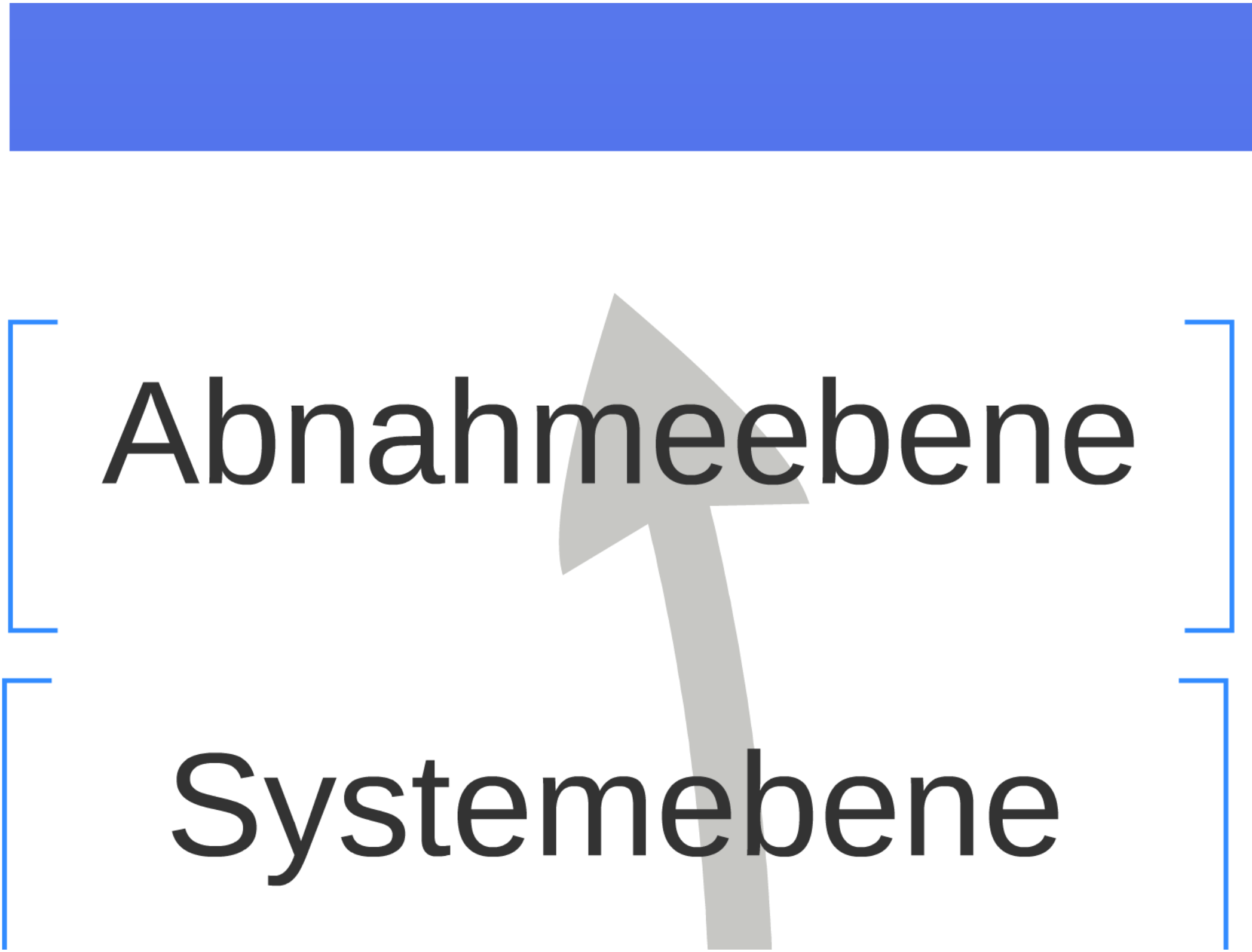
- *Termingetriebene Integration (Schedule driven)*
- *Risikogetriebene Integration (Risc driven)*
- *Testgetriebene Integration (Test driven)*
- *Anwendungsgetriebene Integration (Use case driven)*

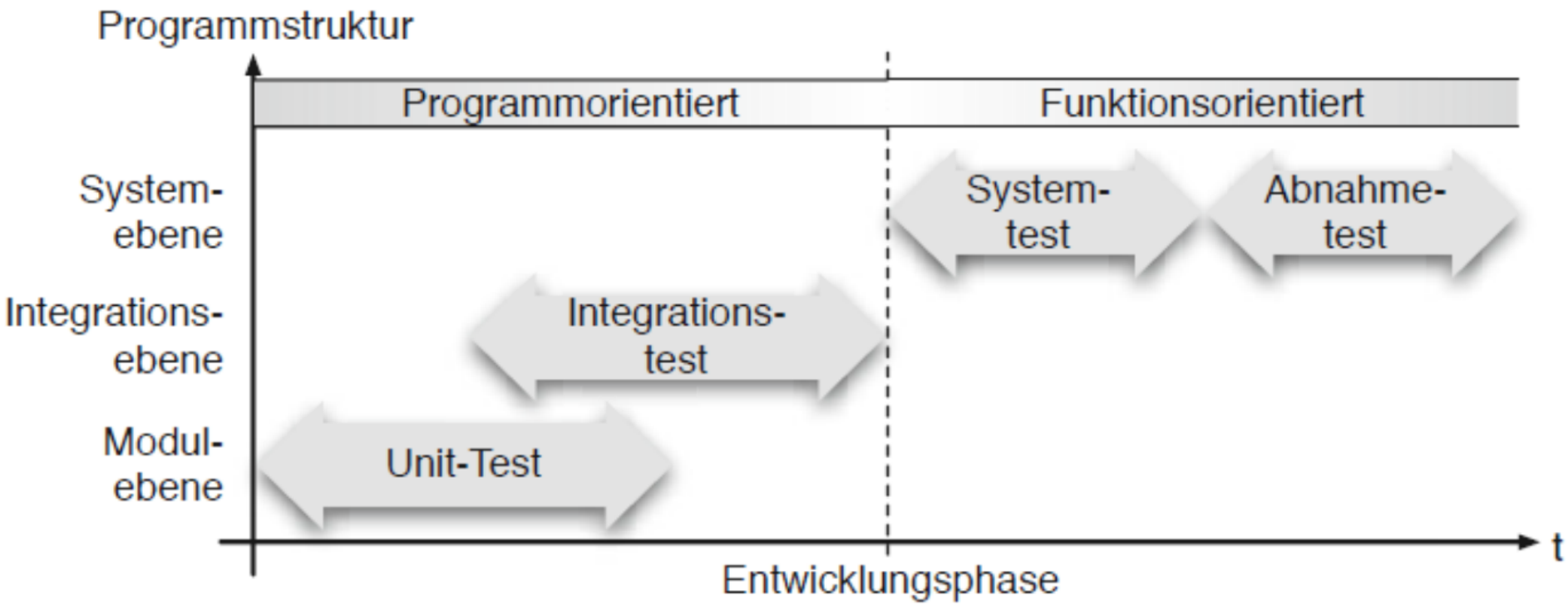
Test Driven Development



- **Entwicklung wird mit dem Schreiben von Automatisierten Tests begonnen**
- **Code wird nur geschrieben, wenn ein Test fehlschlägt**


Abnahmeebene
Systemebene
Integrationsebene


Abnahmeebene
Systemebene


Programmstruktur
Programmorientiert
Funktionsorientiert
System-
ebene
Integrations-
ebene
Modul-
ebene
System-
test
Abnahme-
test
Integrations-
test
Unit-Test
t
Entwicklungsphase

User-Expirience-Testing

Eingesetzt in Ebenen Integration, System und Abnahme

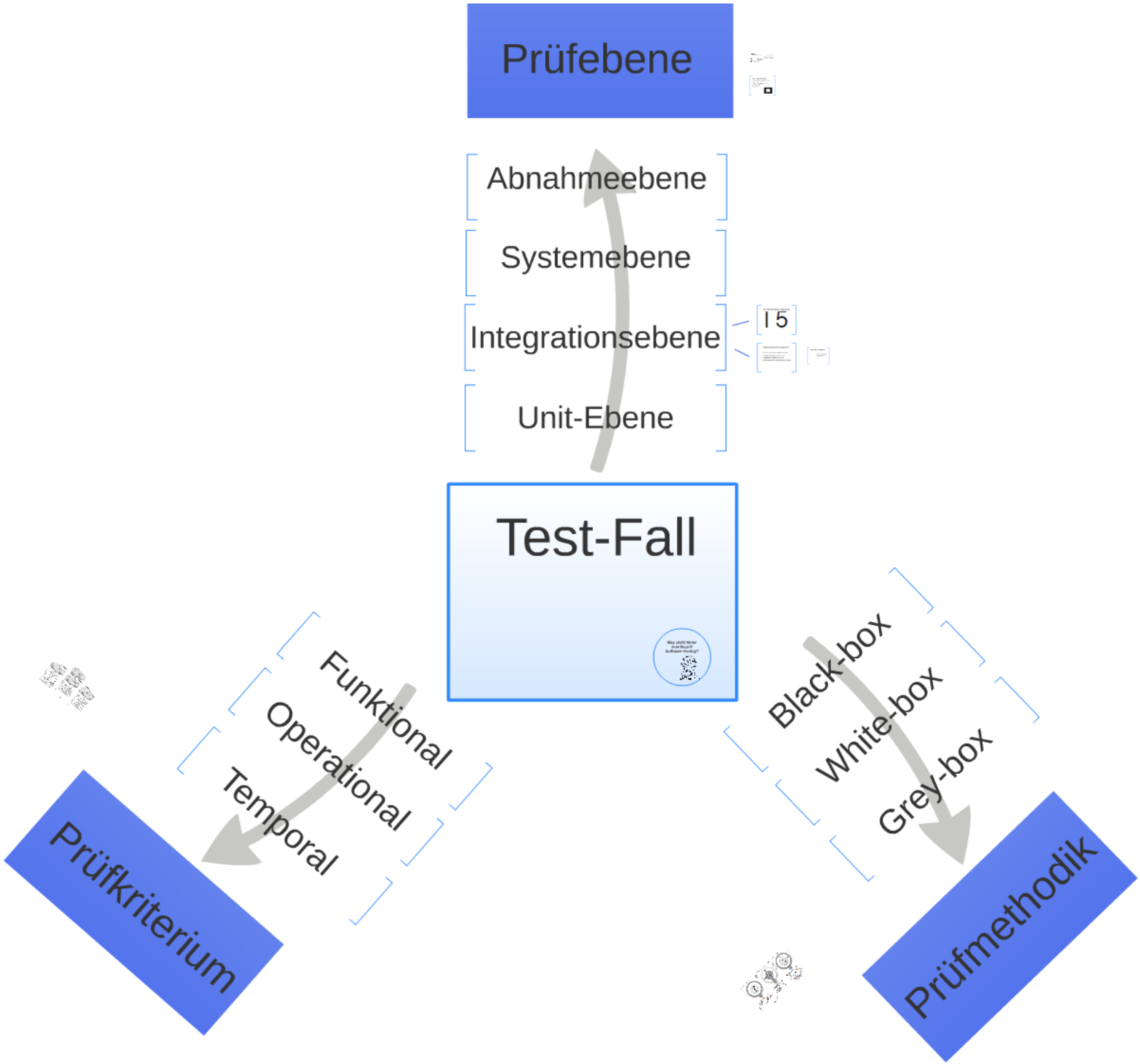
Bei Produkten am Massenmarkt:

- Alpha-Test in Anwendungsumgebung des Herstellers
- Beta-Test in Kunden-Umgebung

Outsourcing-Möglichkeiten:

- Low-Cost-Testing
- Crowd-Testing


YouTube

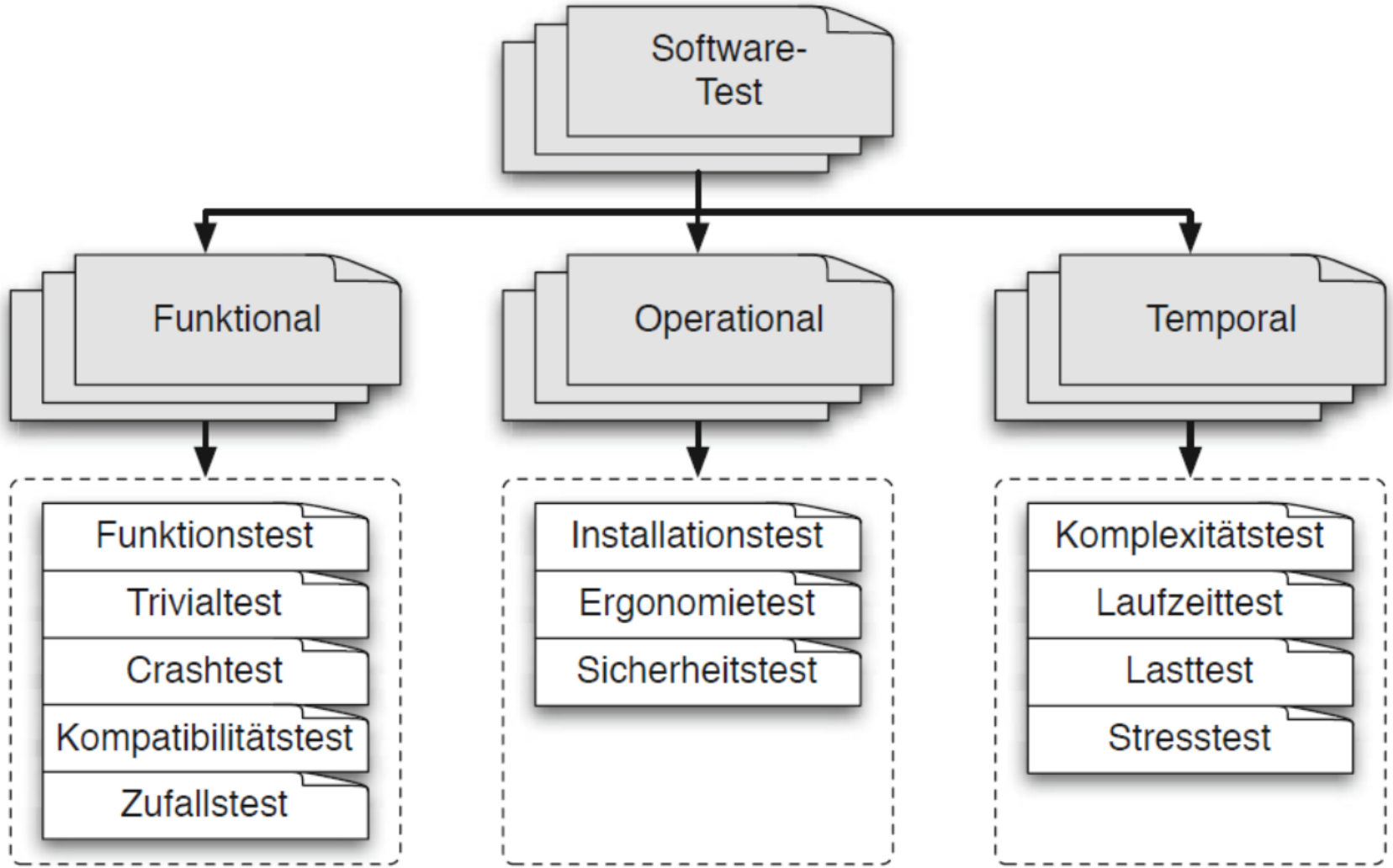

Prüfebene
Abnahmeebene
Systemebene
Integrationsebene
I 5
Unit-Ebene
Test-Fall
Funktional
Operational
Temporal
Prüfkriterium
Black-box
White-box
Grey-box
Prüfmethodik

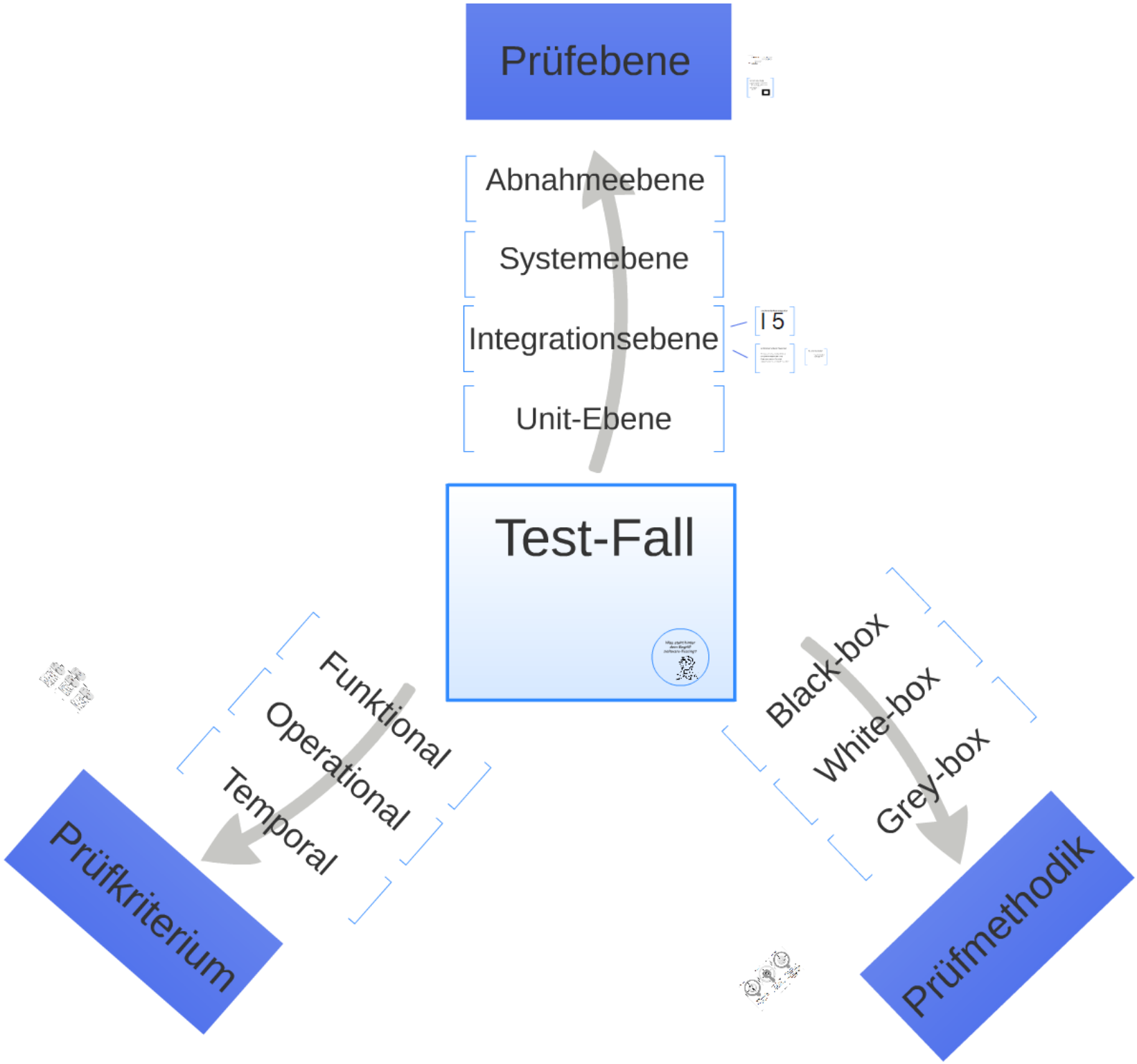
Funktional

Operational

Temporal

Prüfkriterium

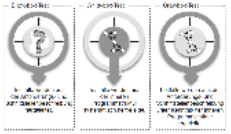

Software-
Test
Funktional
Operational
Temporal
Funktionstest
Trivialtest
Crashtest
Kompatibilitätstest
Zufallstest
Installationstest
Ergonomietest
Sicherheitstest
Komplexitätstest
Laufzeittest
Lasttest
Stresstest


Prüfebene
Abnahmeebene
Systemebene
Integrationsebene
Unit-Ebene
Test-Fall
Funktional
Operational
Temporal
Prüfkriterium
Black-box
White-box
Grey-box
Prüfmethodik

Black-box

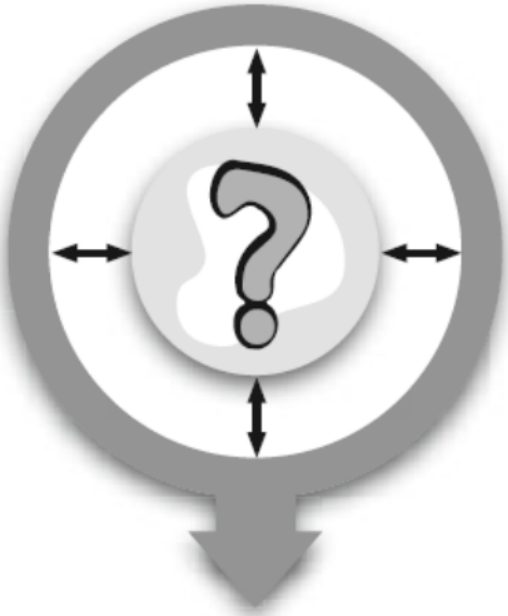
White-box

Grey-box



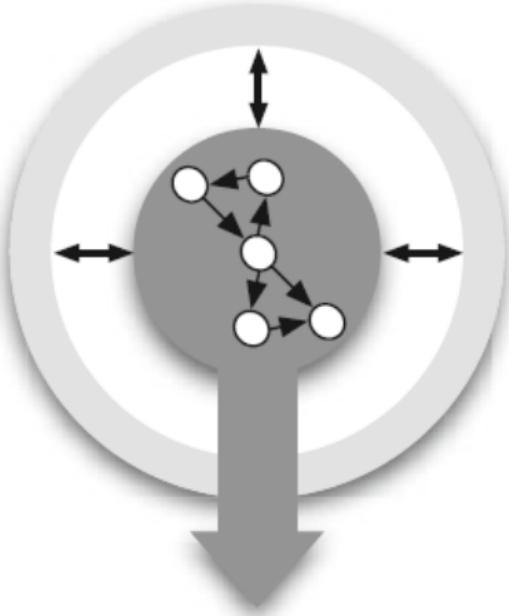
Prüfmethodik

Black-box-Test



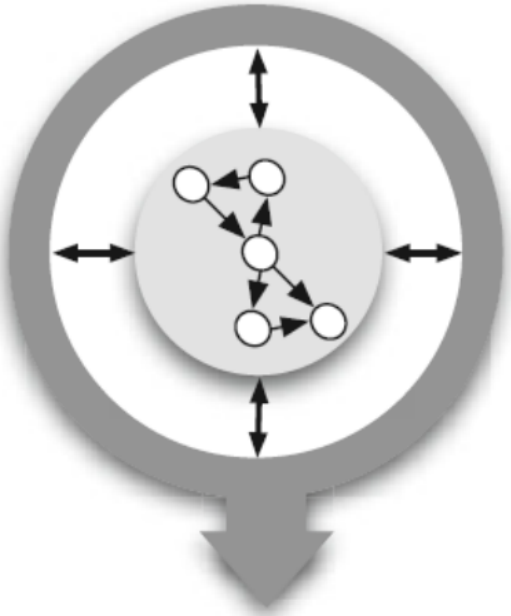
Testfälle werden aus der Anforderungs- und Schnittstellenbeschreibung hergeleitet.

White-box-Test

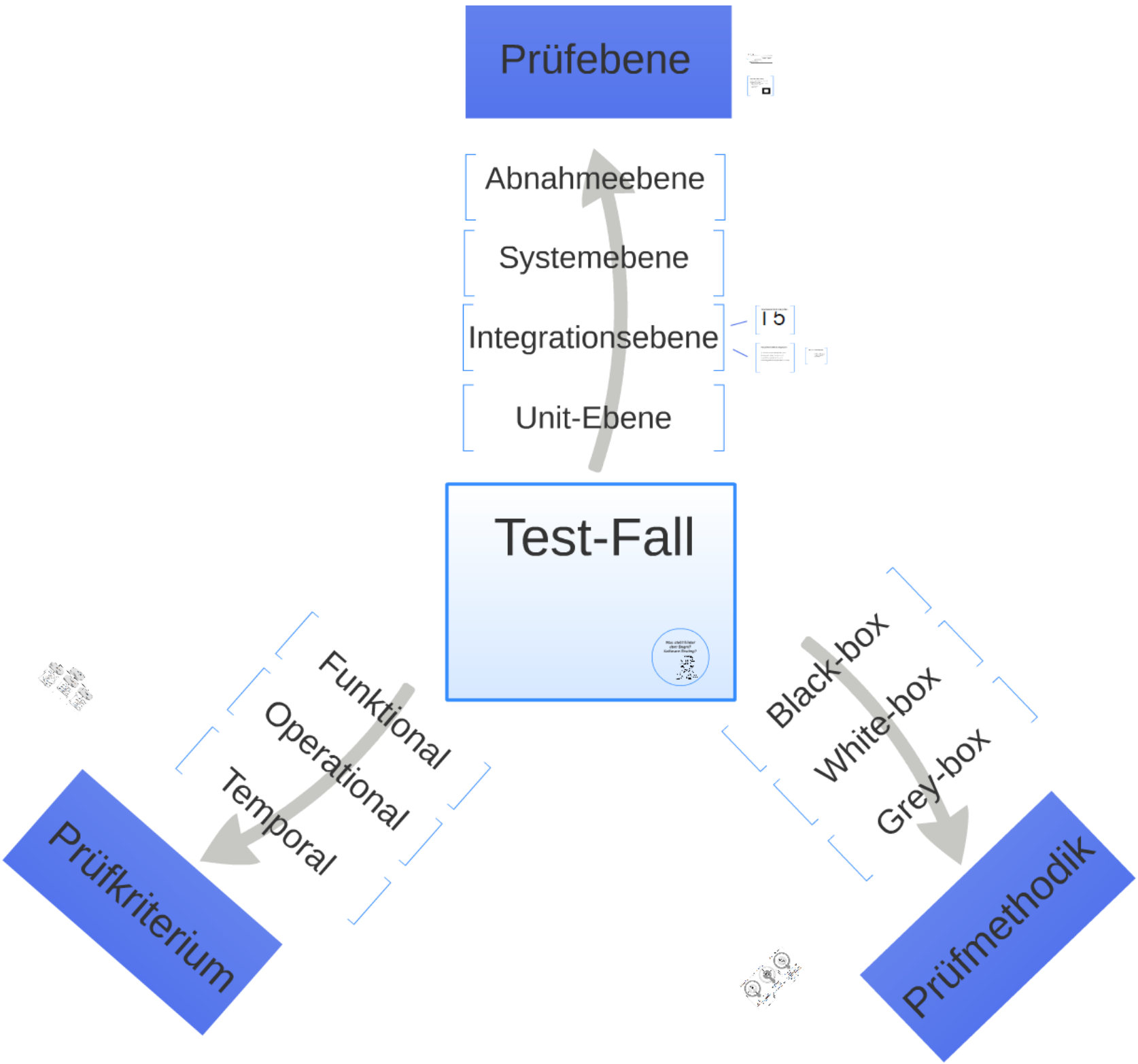


Testfälle werden aus der inneren Programmstruktur systematisch hergeleitet.

Gray-box-Test

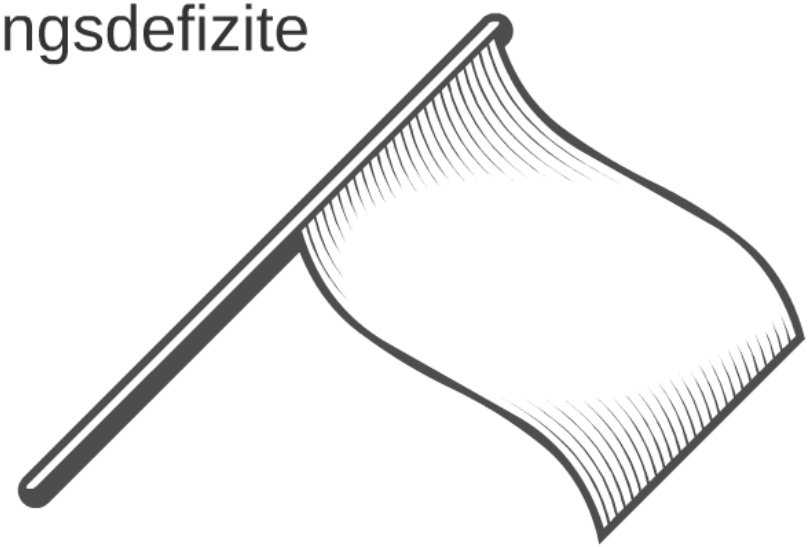


Testfälle werden aus der Anforderungs- und Schnittstellenbeschreibung unter Kenntnis der inneren Programmstruktur hergeleitet.


Prüfebene
Abnahmeebene
Systemebene
Integrationsebene
Unit-Ebene
Test-Fall
Funktional
Operational
Temporal
Prüfkriterium
Black-box
White-box
Grey-box
Prüfmethodik

Die Grenzen des SW-Testings

- Unklare oder fehlende Anforderungen
- Programmkomplexität
- Mangelnde Werkzeugunterstützung
- Fehlende Management-Unterstützung
- Ausbildungs- und Fortbildungsdefizite
- Zeitprobleme

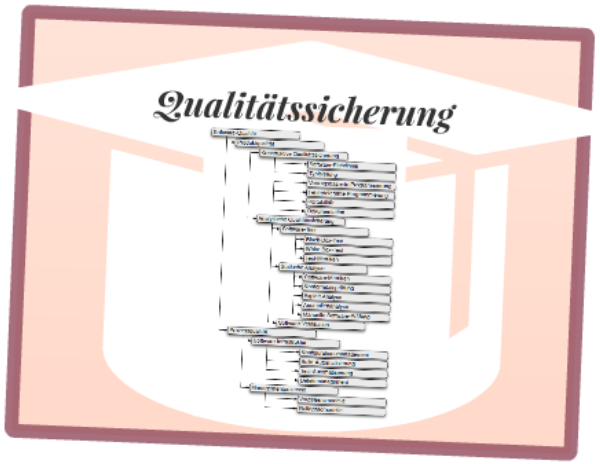


Software-Verifikation

Ebenfalls Analytische Qualitätssicherungsmethode

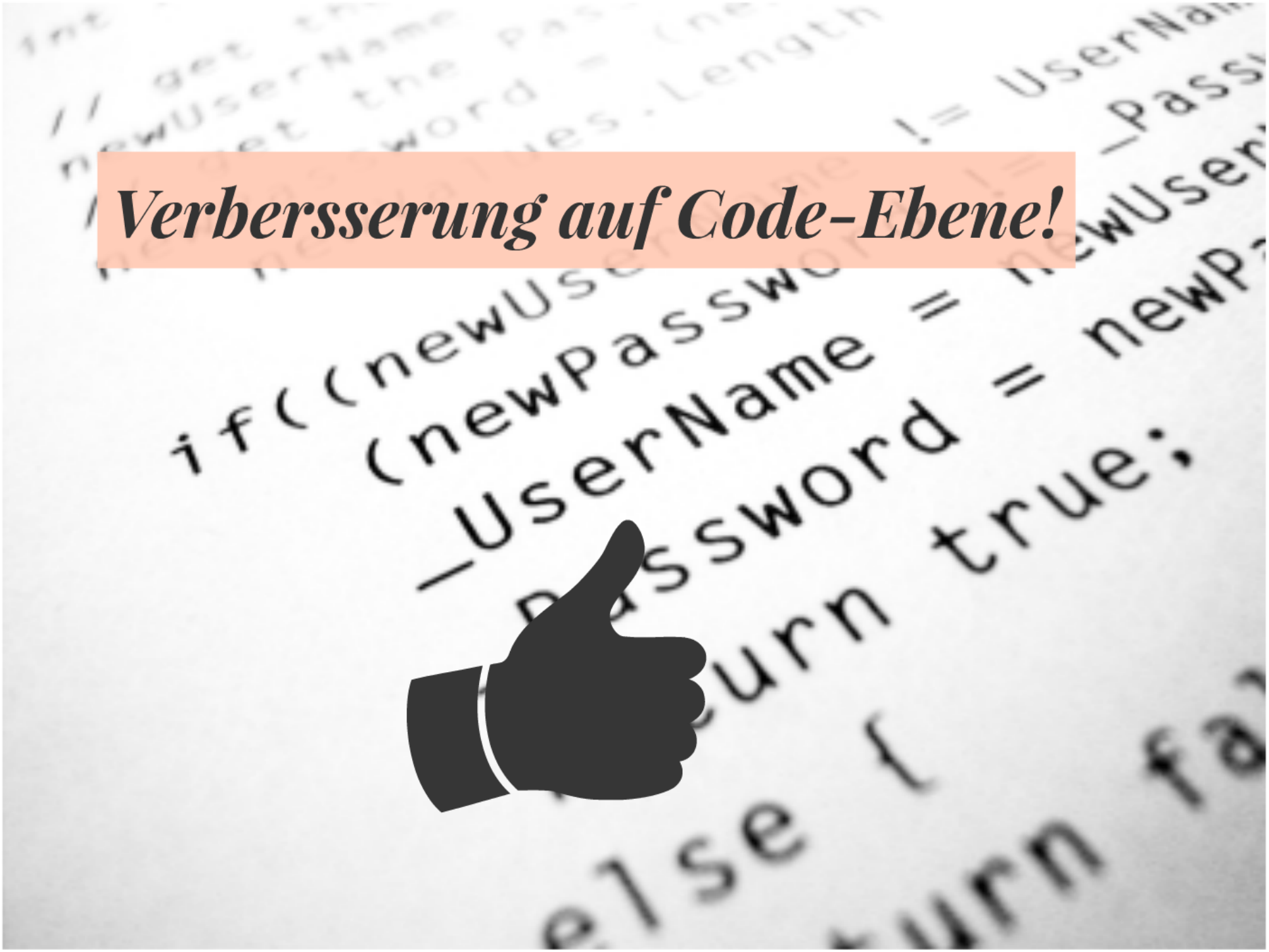
Ziel: Nachweis zur Erfüllung der Spezifikation mit Hilfe mathematischer Methoden


Qualitätssicherung
Komplexität!


Qualitätssicherung

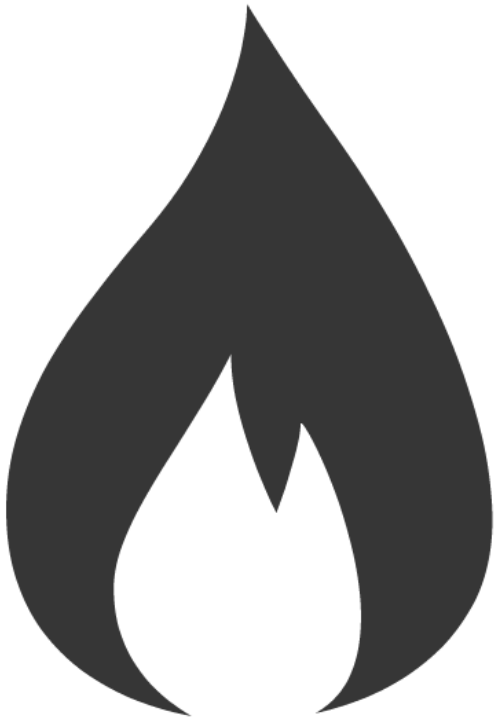
Weshalb braucht es Qualitätssicherung?

Ist Software wirklich so schlecht?


Verbersserung auf Code-Ebene!

Fehler auf Code-Ebene:

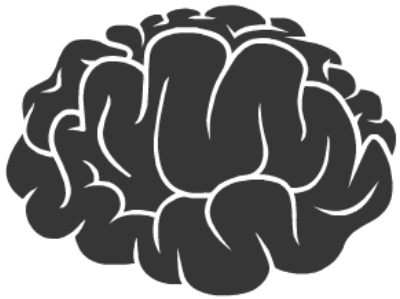
- Syntax-Fehler
- Semantische Fehler
- Typen-Fehler
- Fehler in der Parallelität
- Portabilitätsfehler
- Optimierungsfehler



Andere Formen von Fehler

Fehler ausserhalb des Codes

- Spezifikationsfehler
- Hardware
- Fehler aus Fremd-Abhängigkeiten



Ist Software wirklich so schlecht?

Andere Formen von Fehler

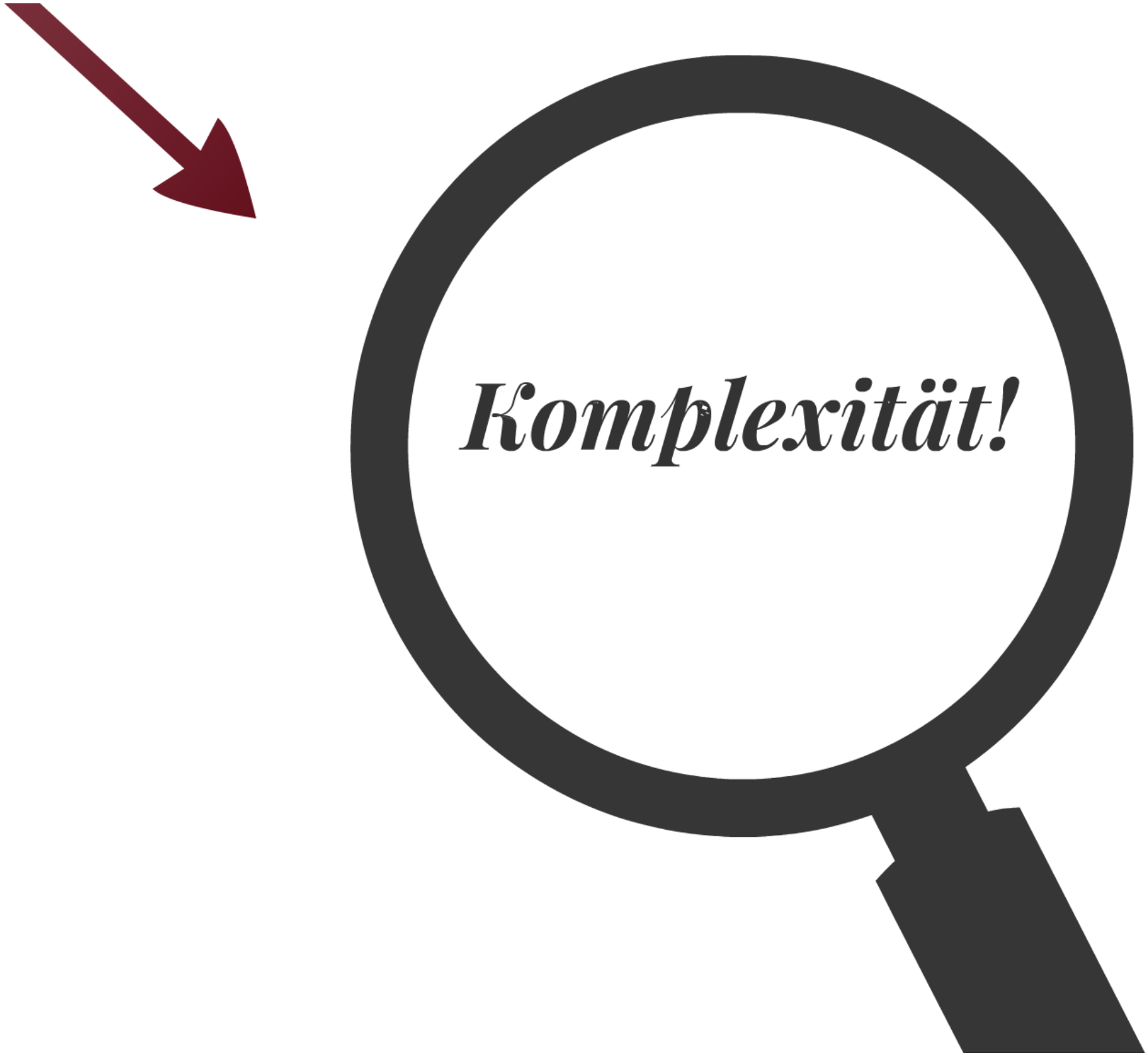
Fehler auf Code-Ebene:

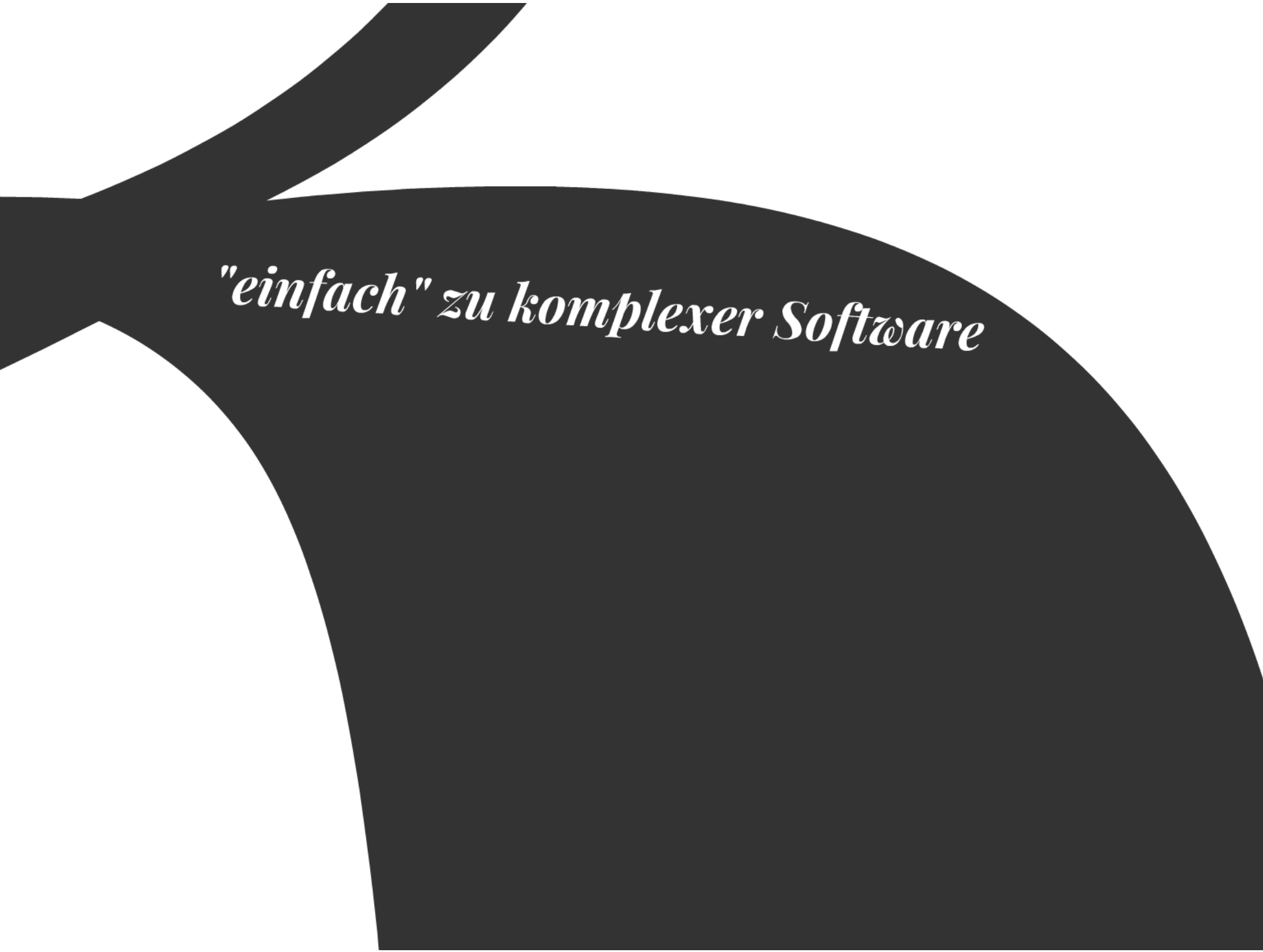
- Syntax-Fehler
- Semantische Fehler
- Typen-Fehler
- Fehler in der Parallelität
- Portabilitätsfehler
- Optimierungsfehler

Fehler ausserhalb des Codes

- Spezifikationsfehler
- Hardware
- Fehler aus Fremd-Abhängigkeiten

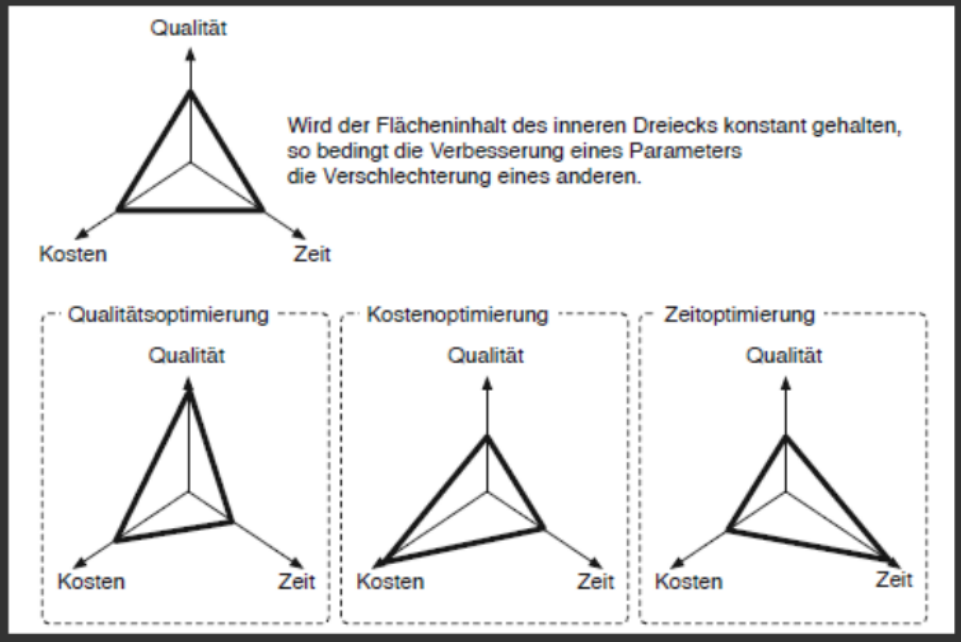
t es
ung?


Komplexität!


"einfach" zu komplexer Software

Verteiltes Wissen!

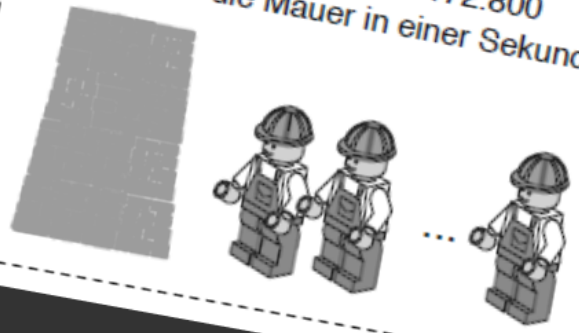
Challenge: Projekt-Management in Softwareentwicklungs-Projekten!



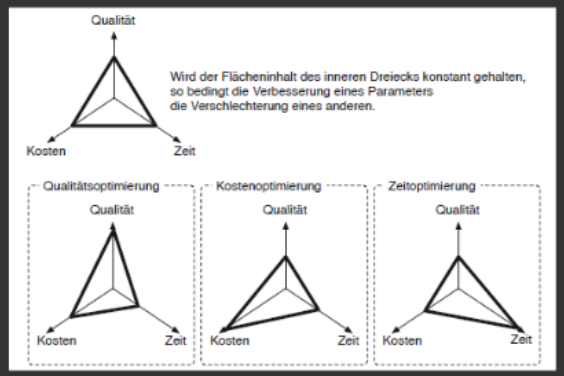
Problematik Zeitschätzung

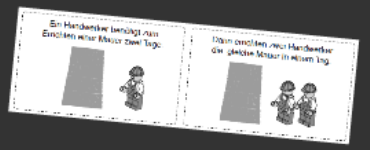
$$\text{Entwicklungsdauer} = \frac{\text{Zeit in Mannjahren}}{\text{Anzahl Mitarbeiter}}$$

Folgerichtig errichten 172.800 Handwerker die Mauer in einer Sekunde.



Challenge: Projekt-Management in Softwareentwicklungs-Projekten!

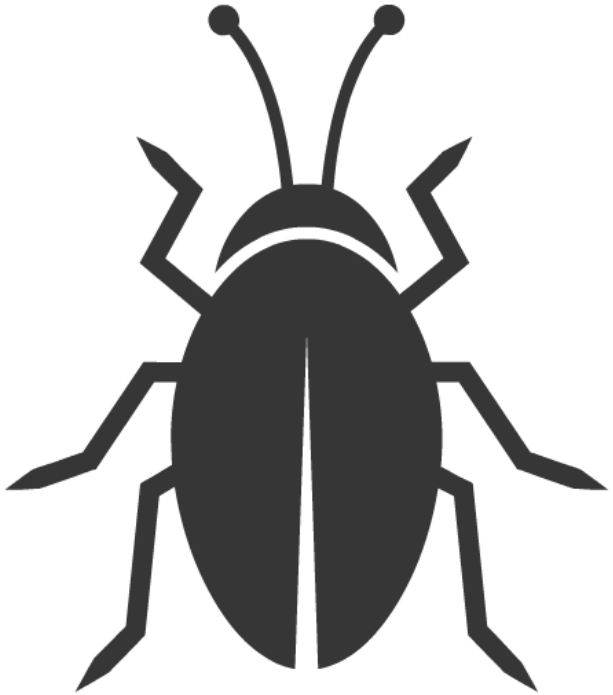




Problematik Zeitschätzung

Software-Alterung & Lebenszyklus

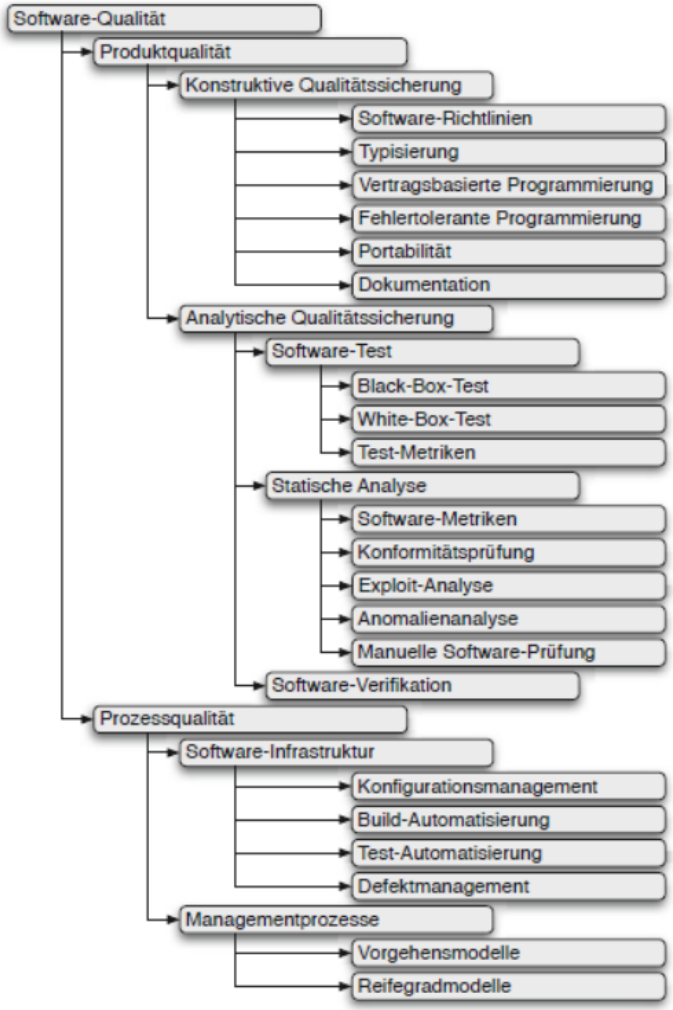
nd Software-Projekte zum Scheitern verurteil




Qualitätssicherung


Qualitätssicherung
Weshalb braucht es Qualitätssicherung?
Komplexität!

Qualitätssicherung

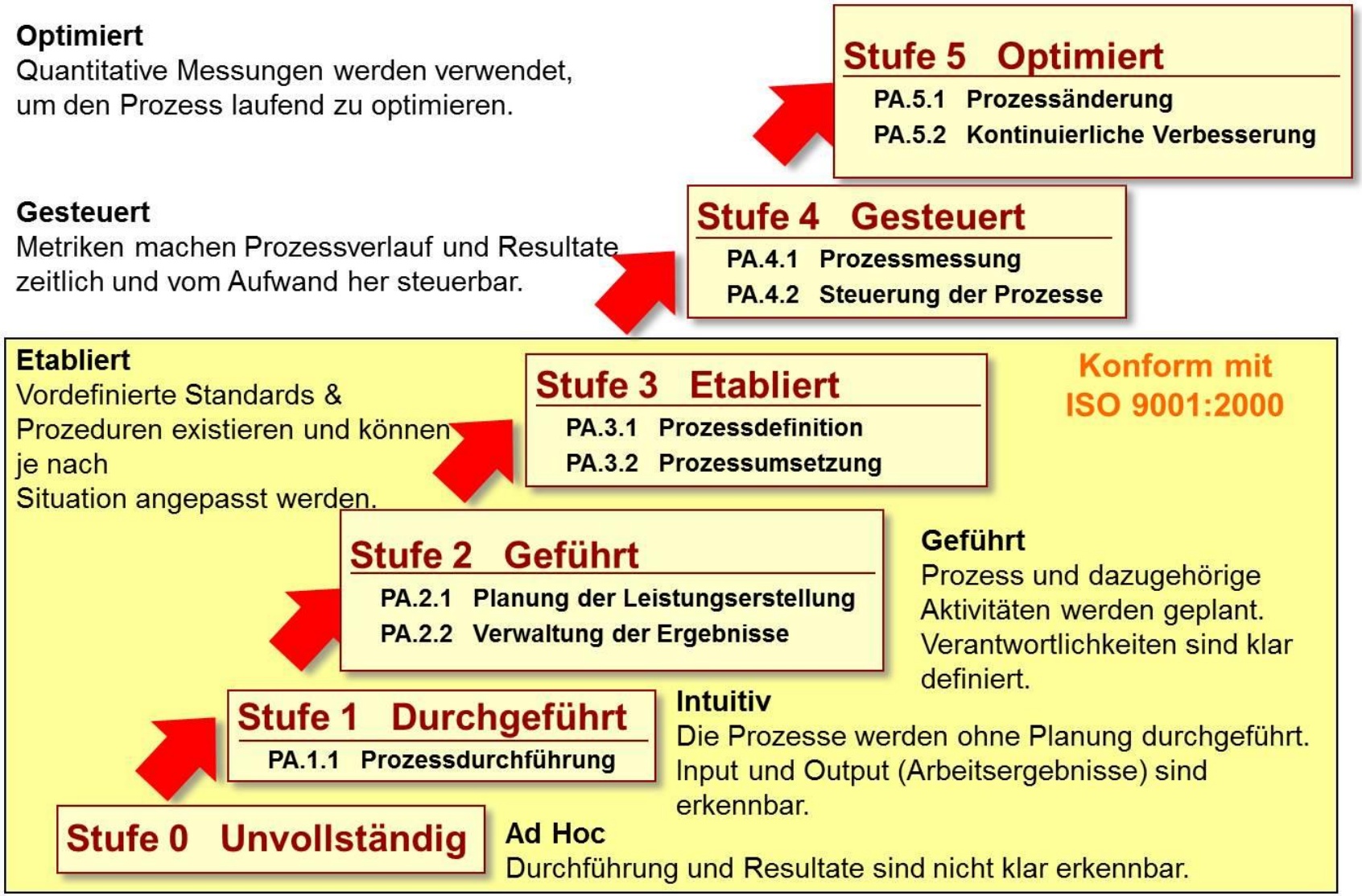


Managementprozesse

Managementprozesse & QS

- Vorgehensmodelle
- Reifegradmodelle

Reifegradmodelle



Software-Produktion?

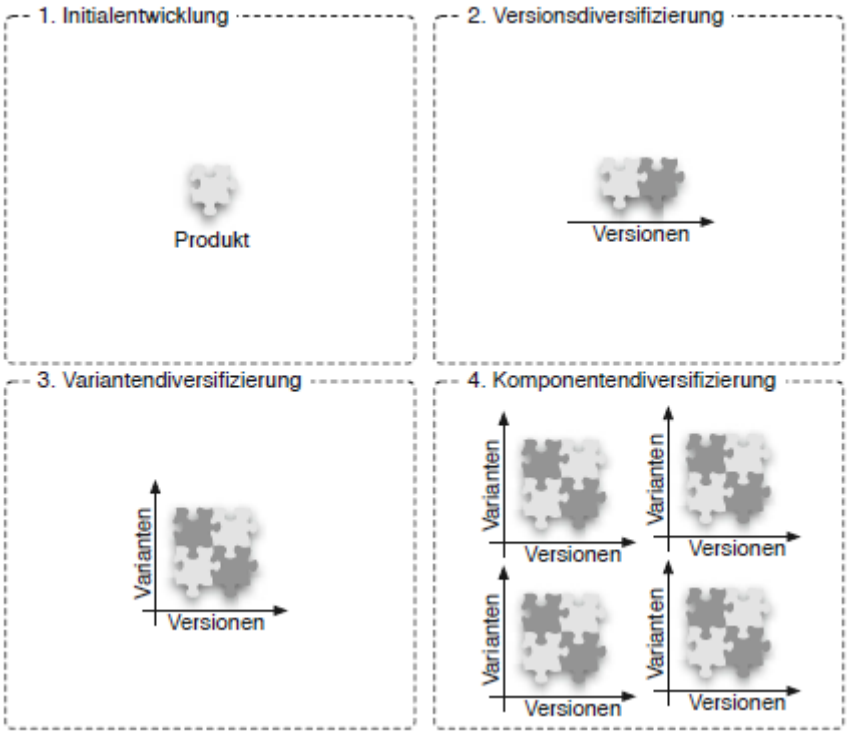
“Fundamentally, you get good software by thinking about it, designing it well, implementing it carefully, and testing it intelligently, not by mindlessly using an expensive mechanical process.”
Stephen C. Johnson

Qualitätssicherung mit Software-Infrastruktur

Inhaltsübersicht

- Problematiken im Software-Entwicklungsprozess
- Software-Infrastruktur im Entwicklungsprozess
- Test-Automatisierung
- Continuous Delivery

Problematiken im Software-Entwicklungsprozess



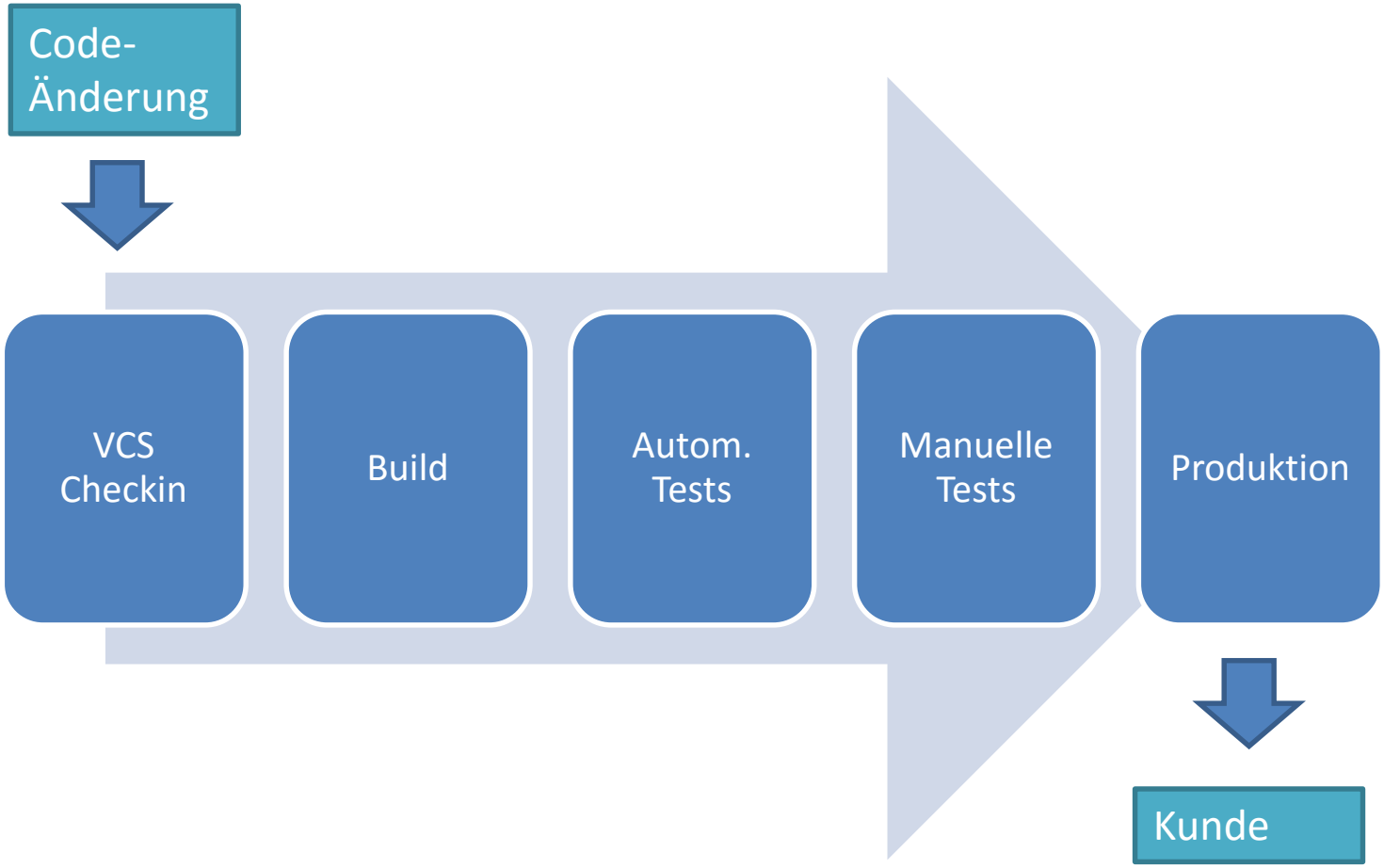
Nötige Software-Infrastruktur

- Versionsverwaltung
- Defektmanagement
- Build-Server mit Test- und Delivery-Automatisierung

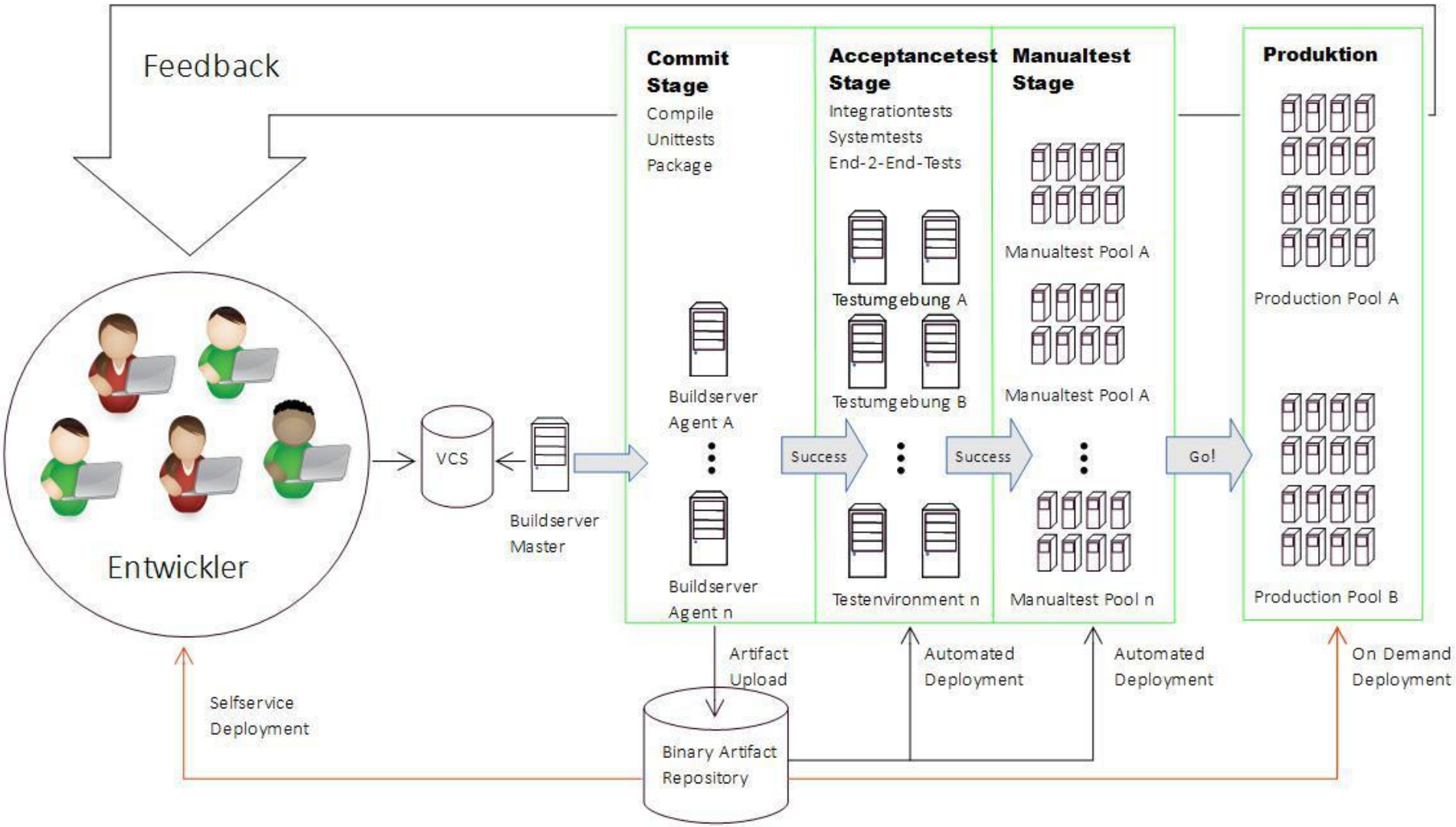
Test-Automatisierung

Continuous Delivery

Continuous Delivery Pipeline



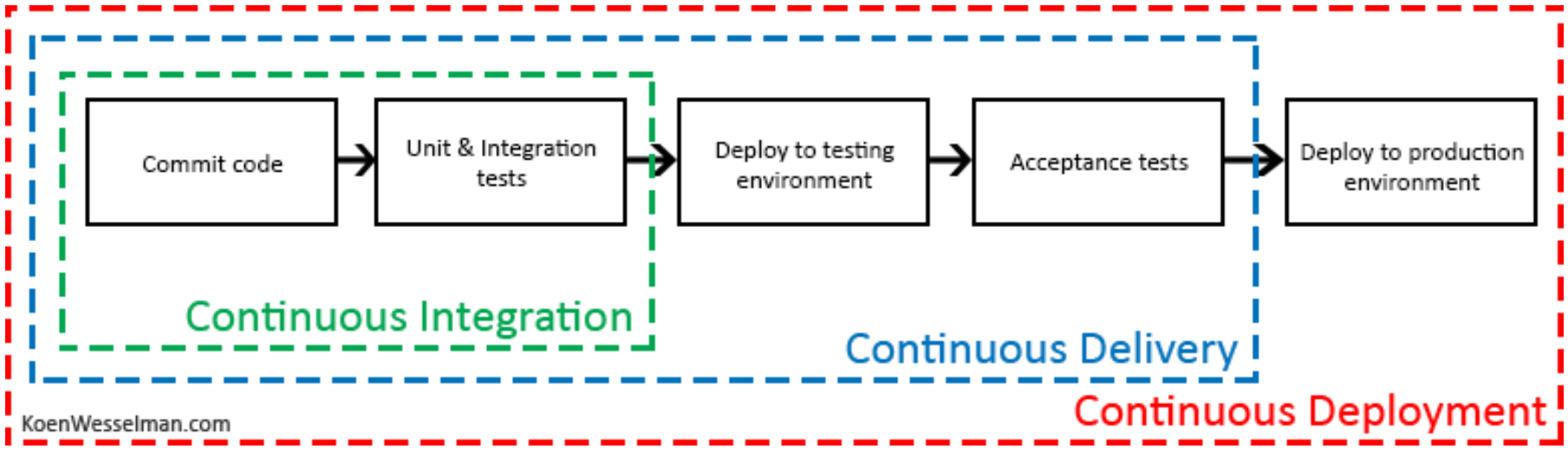
Continuous Delivery Pipeline II



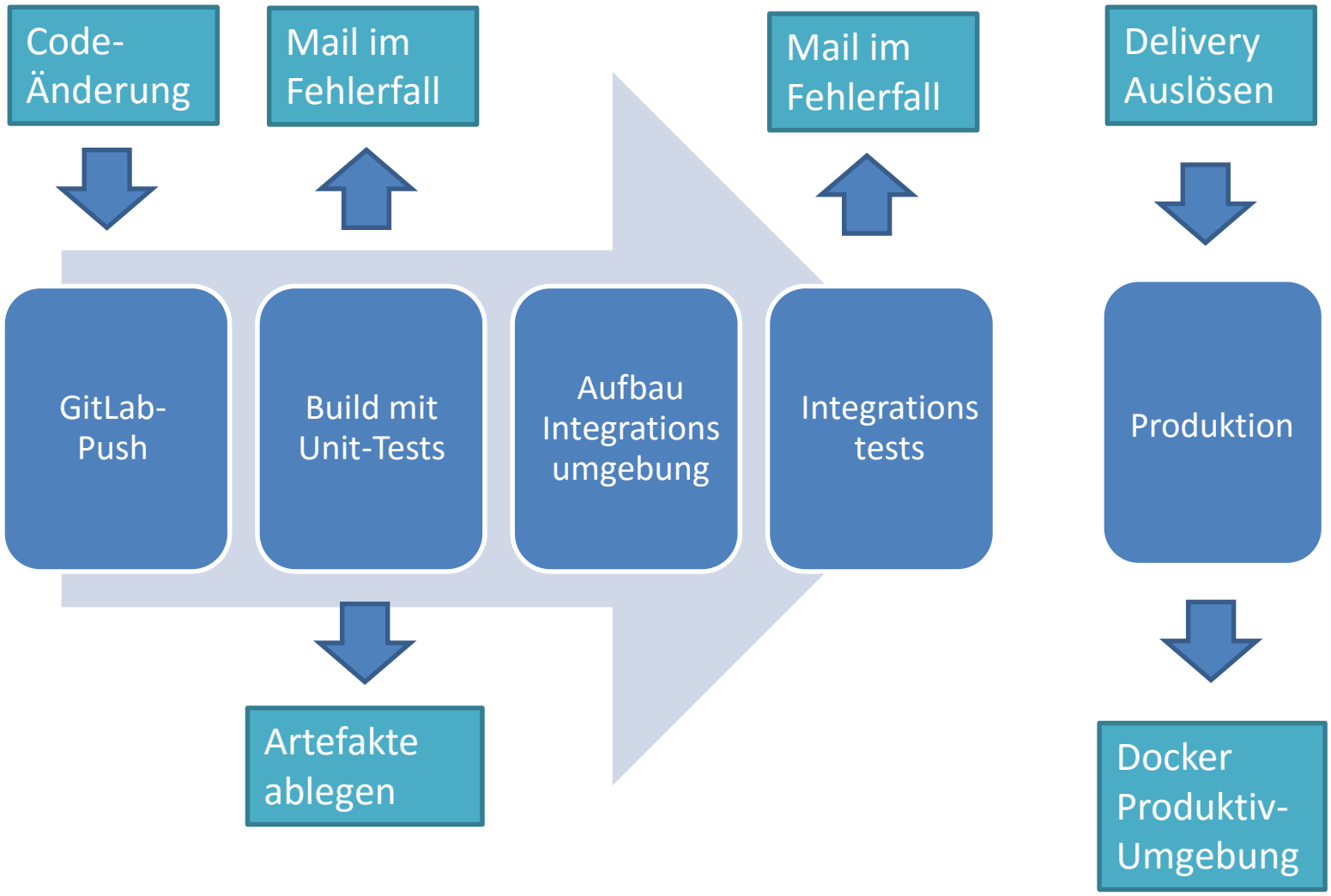
Prinzipien von Continous Delivery

- Jede Änderung an der Software wird getestet.
- Der Software-Stand wird nur ein einziges Mal an zentraler Stelle gebaut.

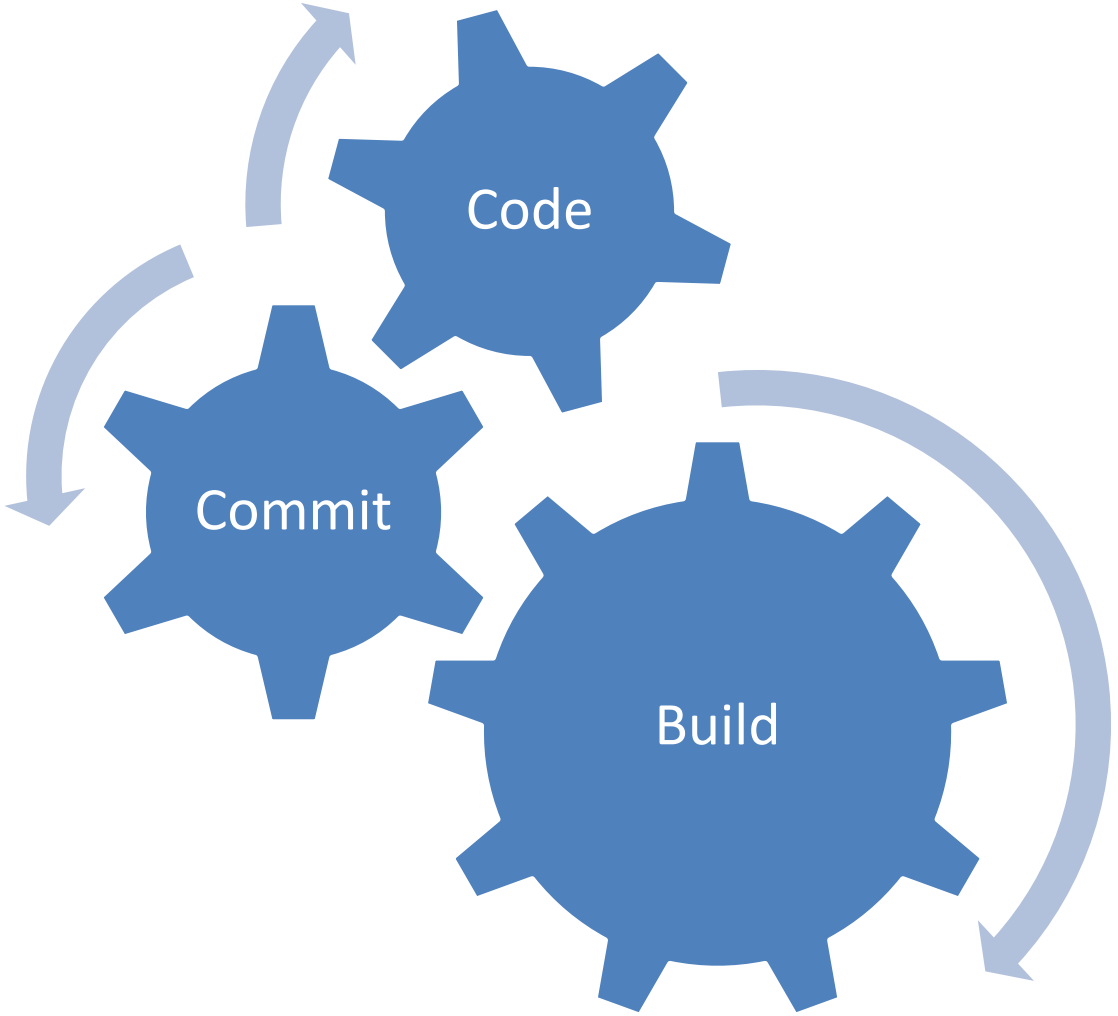
Unterschiede CI & CD



Continious Delivery in TaaS



Vorteile & Werkzeuge



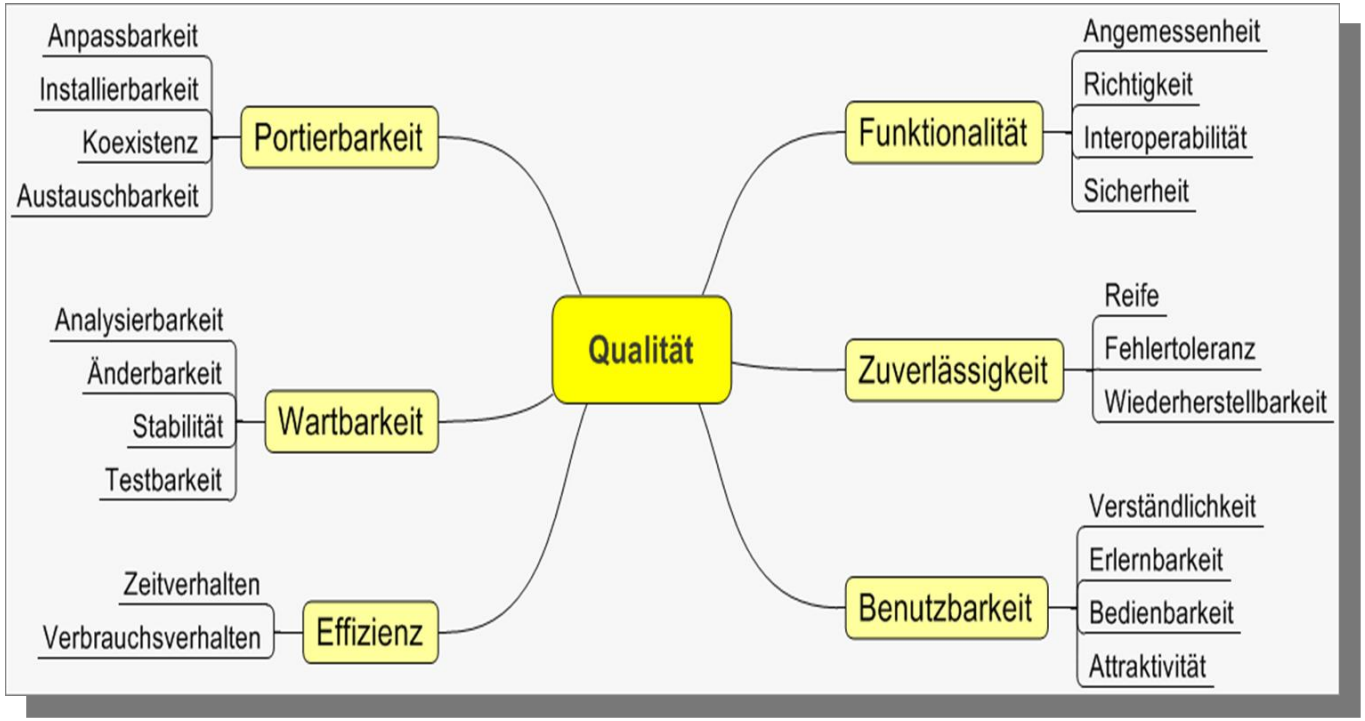
Software-Alterung

Inhaltsübersicht

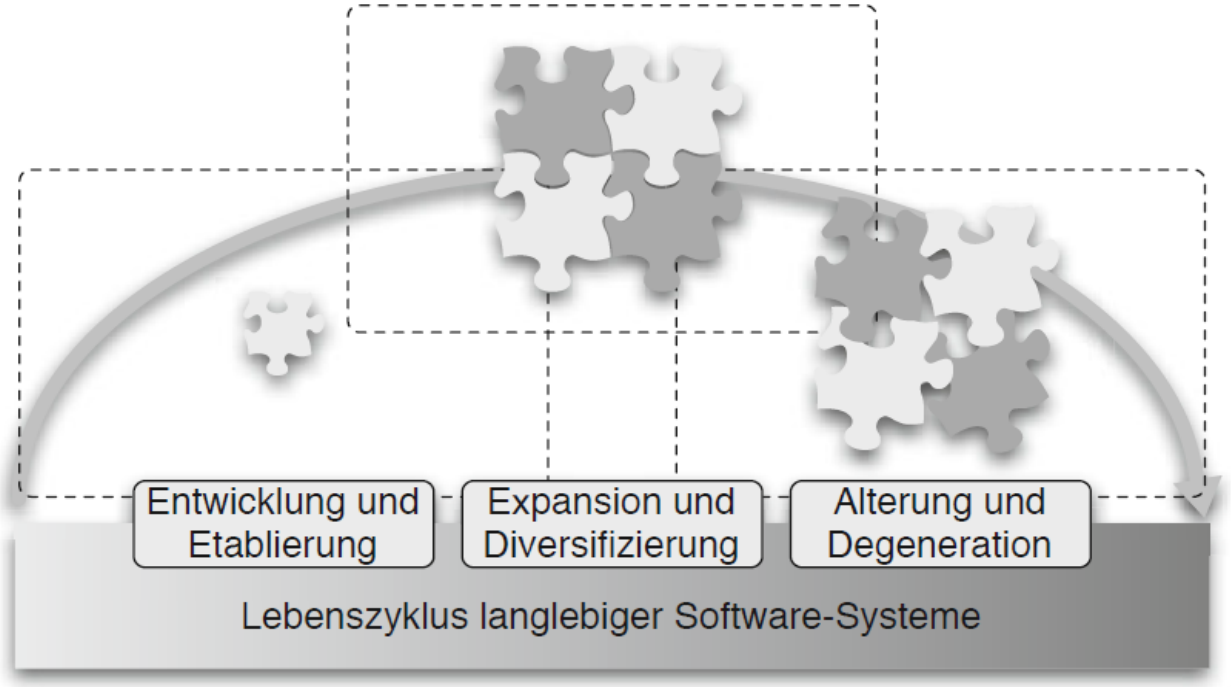
- Was bedeutet Software-Alterung
- Lebenszyklus langlebiger Software-Systeme
- Problematik: Software-Änderbarkeit
- Was tun gegen Software-Alterung

Software-Alterung

- Degradierung verschiedener Qualitätsmerkmale abhängig von der Verlaufszeit



Häufiger Lebenszyklus langlebiger Software-Systeme



Problematik: Software-Änderbarkeit

- Problematik Kaschieren statt Reparieren
- Kompatibilitätsanforderungen
- Sekundärwissen geht verloren

Was tun gegen Software-Alterung?

- Sensibilität in Entwicklung und Management
- Investitionsbereitschaft
- Mut zum Refactoring & Redesign

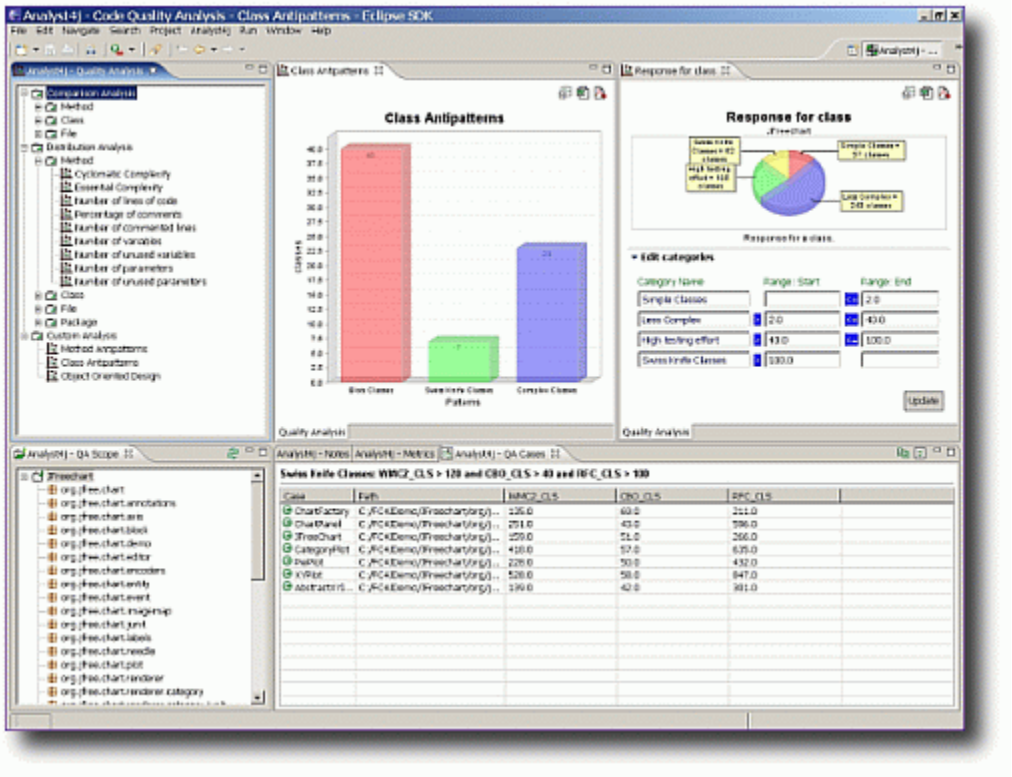
Statische Code-Analyse

Übersicht

- Software-Metriken
- Konformitätsanalyse
- Exploit-Analyse
- Anomalien-Analyse
- Manuelle Software-Prüfimg

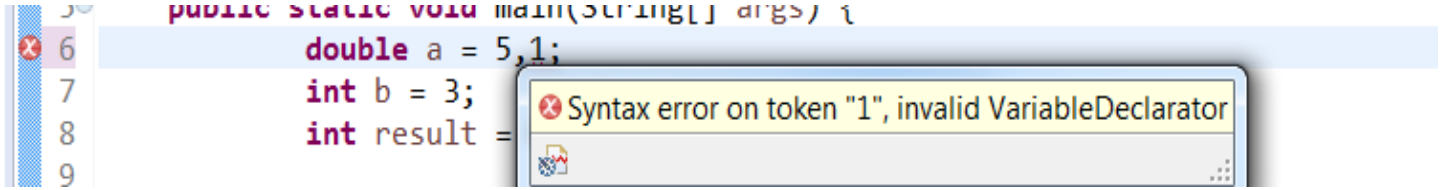
Software-Metriken

- LOC und NCSS
- Halstead-Metriken & McCabe-Metrik
- Objektorientiere Metriken

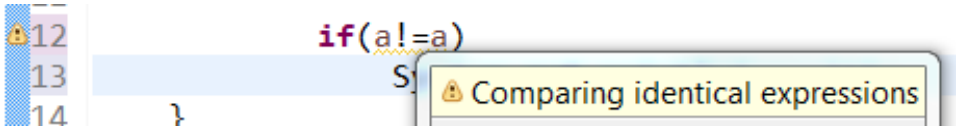


Konformitätsanalyse

- Syntax-Analyse



- Semantik-Analyse



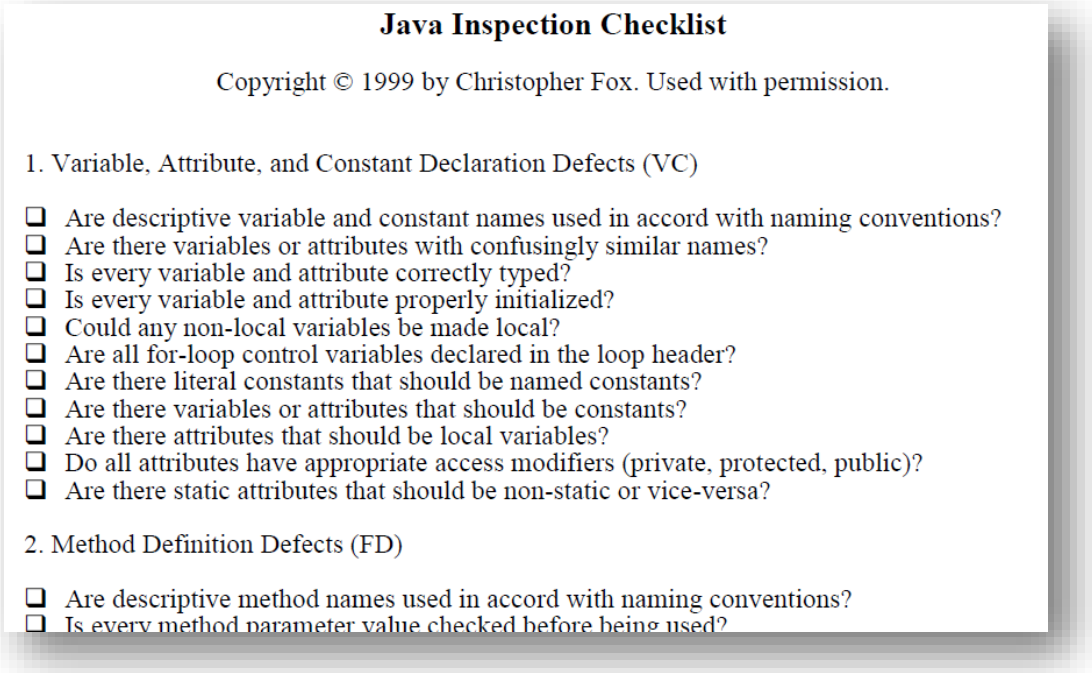
Exploit-Analyse

Anomalien-Analyse

- Kontrollflussanomalien
- Datenflussanomalien

Manuelle Software-Prüfung

- Walkthroughs
- Reviews
- Inspektion



Java Inspection Checklist

Copyright © 1999 by Christopher Fox. Used with permission.

1. Variable, Attribute, and Constant Declaration Defects (VC)

- ❑ Are descriptive variable and constant names used in accord with naming conventions?
- ❑ Are there variables or attributes with confusingly similar names?
- ❑ Is every variable and attribute correctly typed?
- ❑ Is every variable and attribute properly initialized?
- ❑ Could any non-local variables be made local?
- ❑ Are all for-loop control variables declared in the loop header?
- ❑ Are there literal constants that should be named constants?
- ❑ Are there variables or attributes that should be constants?
- ❑ Are there attributes that should be local variables?
- ❑ Do all attributes have appropriate access modifiers (private, protected, public)?
- ❑ Are there static attributes that should be non-static or vice-versa?

2. Method Definition Defects (FD)

- ❑ Are descriptive method names used in accord with naming conventions?
- ❑ Is every method parameter value checked before being used?

Hilfsmittel: Checklisten, Protokolle, CI-Prozess, spefifische Tools…

Übung: Bugtracking mit Mantis

Einleitung

Bug-Tracking-Systeme wie Bugzilla und Mantis erlauben das Sammeln, die Koordination und die Auswertung von Software-Defekten. In dieser Übung wollen wir uns das Mantis-System genauer betrachten.

Aufgabe

- Starte eine „Free Trial“ auf www.mantishub.com.
- Erstellt ein Projekt für euere Beispiel-Architektur und versucht gemäss der Baustein-Struktur 2-4 Teil-Projekte zu definieren und in Manits zu pflegen.
- Erstelle folgende drei Benutzer mit den entsprechenden Rollen:
    - Entwickler
    - Tester
    - Manager
- Verfasse als Benutzer „Tester“ mindestens zwei saubere Fehlerberichte pro Teilprojekt mit verschiedenen Prioritäten.
- Hole als Benutzer „Entwickler“ bei einem Fehlerbericht ein zusätzliches Feedback beim Benutzer „Tester“ ab.
- Stelle ein Fehlerbericht dem Benutzer „Entwickler“ im Status „assigned“ zu und löse den Fall als Benutzer „Entwickler“.
- Spiele einige Situationen durch und schliesse zwei Fehlerberichte ab.
- Schaffe dir als Benutzer „Manager“ einen Überblick über die aktuellen und bearbeiteten Fälle.

Zeit

Ca. 30 Minuten

Sozialform/Lernform

Gruppenarbeit

Übung: Test-Fälle

Einleitung

Im Kurs Software-Architektur, habt ihr die Architektur eines Software-Systems beschrieben. Nun gilt es für dieses Projekt beispielhaft für jede Prüf-Ebene je zwei Test-Fälle zu definieren.

Aufgabe

1. Versucht kurz in ein paar Sätzen ein Integrationsvorgehen für die Umsetzung der Architektur zu definieren. Welche Elemente werden zuerst umgesetzt? Nach welcher Vorgehensweise richtet ihr euch?
2. Definiert für jede Prüf-Ebene (Unit, Integration, System, Abnahme) mindestens zwei Test-Fälle. Versucht pro Testfall, folgende Punkte festzuhalten:
    - Titel/Bezeichnung
    - Definition der Prüfebene, des Prüfkriteriums und der Prüfmethodik
    - Beschreibung
    - Wenn nötig: Test-Vorraussetzungen
    - Testschritte
    - Erwartetes Ergebnis

Postet euer Ergebnis ins passende Forum auf Moodle.

Zeit

Ca. 30 Minuten

Sozialform/Lernform

Gruppenarbeit

Übung: Test-Konzept für ein Systemtest

Einleitung

Im Kurs Software-Architektur, habt ihr die Architektur eines Software-Systems beschrieben. Nun gilt es für dieses Projekt ein Test-Konzept zu schreiben, aufgrund dessen anschliessend ein Systemtest geplant wird. Verwendet dabei die Dokumentvorlage des Hermes 5-Testkonzeptes. (Auf Moodle verfügbar)

Am Schluss dürft ihr euere Testfälle aus der letzten Übung, sowie das Test-Konzept kurz der Klasse vorstellen.

Aufgabe

Füllt die Dokumentvorlage unter folgenden Voraussetzungen:

- Im Abschnitt „Testziele“ solltet ihr im Minimum folgende Themen festhalten:
  - Beschrieb des Zieles des Systemtestes
  - Beschrieb des Vorgehens, um die Gesamtheit des Systems umfänglich zu testen.
  - Lasst dabei die Priorisierung der Qualitätsmerkmale aus der Architekturdokumentation einfliessen.
- Beschreibt im Abschnitt „Test-Objekte“ die verschiedenen Hauptkomponenten gemäss der von euch entworfener Architektur.
- Beschreibt unter „Test-Arten“, wie ihr die Qualität des Software-Produktes durchgehend prüfen könnt. Denkt dabei auch über die Prüftechniken (Black-Box, White-Box, Grey-Box) nach.
- „Testabdeckung“
  - Definiert im Minimum 6 Testfälle für den Systemtest.
  - Teilt jeden Test zusätzlich einem Prüfkriterium zu. (Buch Kapitel 4.2.2 Prüfkriterien)
  - Beschreibt welche Teile des Systemtests ihr bewusst ausgeklammert habt.
- Ergänzt die Abschnitte „Testrahmen“ und „Testinfrastruktur“
- Haltet mindestens 6 Testfallbeschreibungen im Abschnitt „Testfallbeschreibung“ fest.
- Der Testplan dürft ihr weglassen.

Sendet euer Konzept am Schluss an simeon.liniger@hftm.ch.

Zeit

Vorbereitung: Ca. 45 Minuten
Präsentation: Maximal 5 Minuten

Sozialform/Lernform

Gruppenarbeit