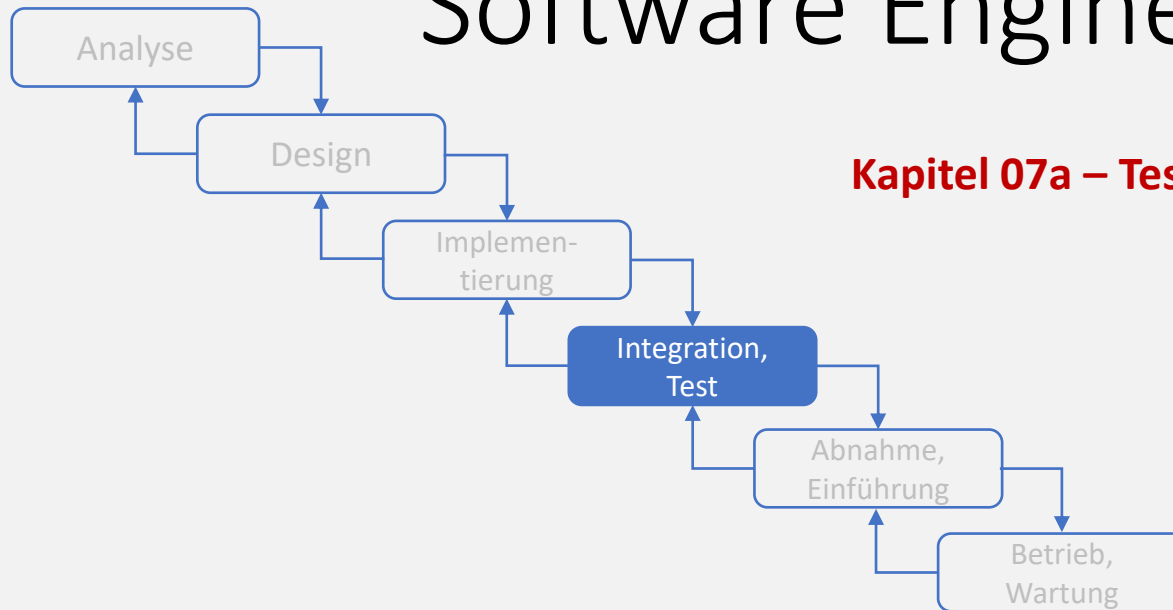




Software Engineering

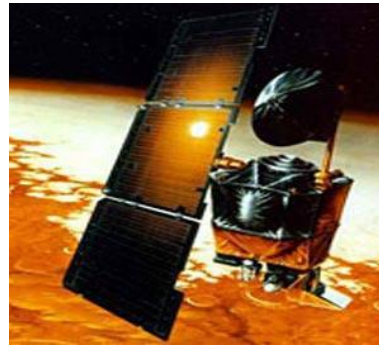
Kapitel 07a – Testen



Gliederung

1. Motivation für Qualitätssicherungsmaßnahmen
2. Arten von Tests
3. Testphasen

Testen: Motivation



- Vgl. dazu Folien Kapitel 1 „Einführung und Motivation“

Testen



Ziel der Testphase:

- Ausführen des Programms mit künstlichen Daten
- Validierungstests:
 - “Nachweisen, dass Programm das tut, was es soll”
 - Entwicklern und Kunden zeigen, dass die Anforderungen erfüllt sind
- Fehlertests:
 - “Fehler finden, bevor das System produktiv benutzt wird”
 - Situationen aufspüren, in denen Software falsch/unerwünscht/nicht spezifikationsgemäß verhält

Aber:

- Abwesenheit von Fehlern kann durch Testen nicht nachgewiesen werden
- Testen ist Teil eines umfassenderen Verifikations- und Validierungsprozesses

Warum?

Testen: Verifikation vs Validierung

- **Verifikation:**

- Klärt die Frage: „Bauen wir das Produkt richtig?“
- Formales Prüfen der Implementierung gegen die Spezifikation

- **Validierung:**

- Klärt die Frage: „Bauen wir das richtige Produkt?“
- Ausführen des Codes und prüfen gegen die wirklichen Bedürfnisse des Nutzers

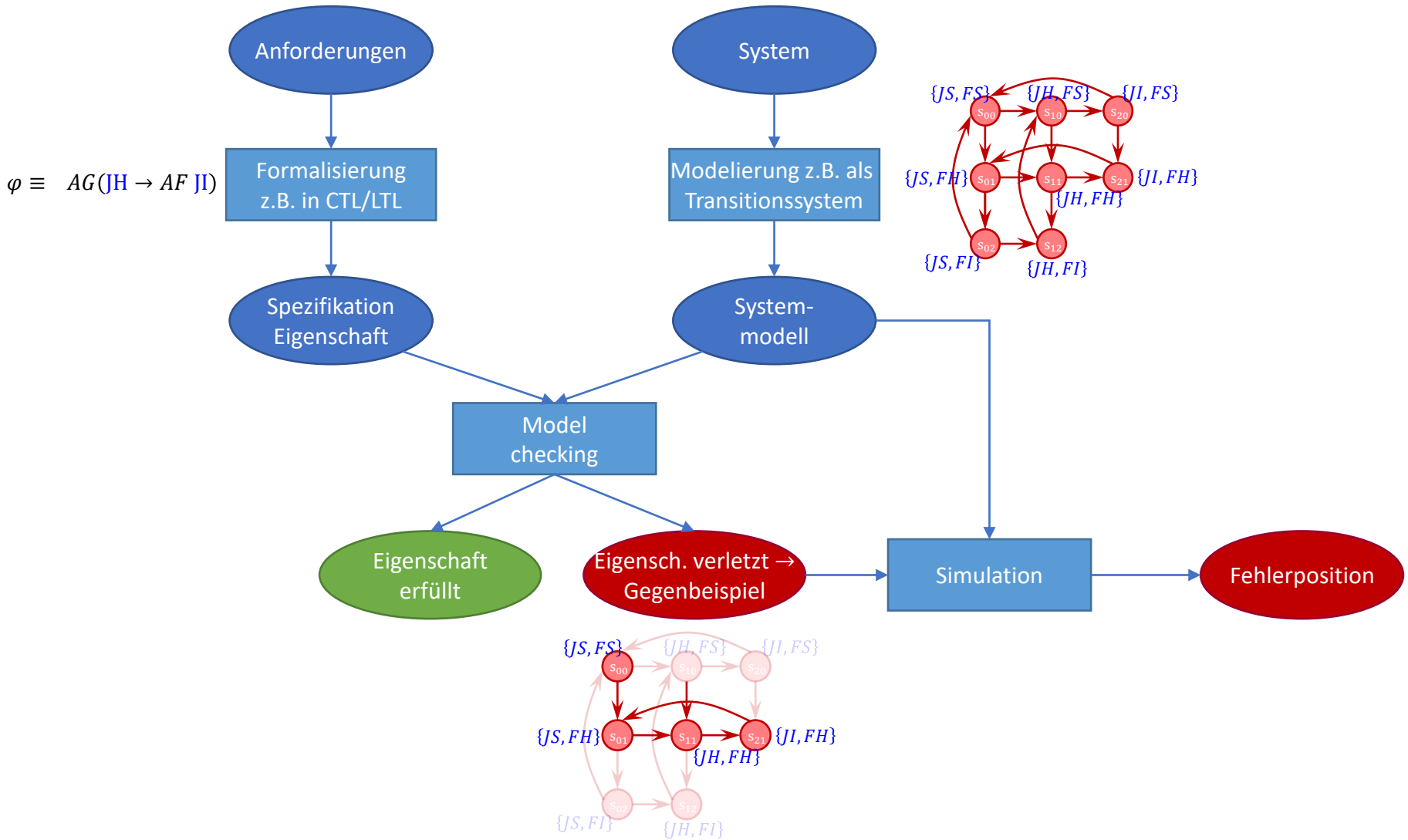
Arten von QS-Maßnahmen (1)

1. Verifikation: Korrektheit formal beweisen

- Sehr aufwendig, daher sehr teuer
- In kritischen Systemen (Luft- und Raumfahrt, AKWs, ...) vorgeschrieben
- Werkzeugunterstützung durch so genannte „Model Checker“, „Hoare Kalkül“

$$\frac{s \vdash_{\Delta} \phi_1 \wedge \phi_2}{s \vdash_{\Delta} \phi_1 \quad s \vdash_{\Delta} \phi_2} \quad \frac{s \vdash_{\Delta} \phi_1 \vee \phi_2}{s \vdash_{\Delta} \phi_1}$$
$$\frac{s \vdash_{\Delta} [a]\phi}{s_1 \vdash_{\Delta} \phi \cdots s_n \vdash_{\Delta} \phi} \text{ if } \{s_1, \dots, s_n\} = \{s' \mid s \xrightarrow{a} s'\}$$

Motivation: Model Checking (s. Master)



Beispiel: Hoare-Kalkül (formaler Korrektheits-Beweis)

Vorbedingung: $a, b \in \mathbb{N}$; sei G größter gemeinsame Teiler von a und b, $E = x + y$

Nachbedingung: $x = G$

$\{G \text{ ist ggT}(a, b) \wedge a \in \mathbb{N}^+ \wedge b \in \mathbb{N}^+\}$

$x := a; y := b;$

$\{INV: G \text{ ist ggT}(x, y) \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y > 0\}$

solange $x \neq y$ wiederhole

$\{INV \wedge x \neq y \wedge x + y = k\}$

falls $x > y$:

$\{G \text{ ist ggT}(x, y) \wedge x > y \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y = k\} \Rightarrow$

$\{G \text{ ist ggT}(x - y, y) \wedge x - y \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x - y + y = k - y < k\}$

$x := x - y$

$\{INV \wedge x + y < k\}$

sonst

$\{G \text{ ist ggT}(x, y) \wedge y > x \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y = k\} \Rightarrow$

$\{G \text{ ist ggT}(x, y - x) \wedge x \in \mathbb{N}^+ \wedge y - x \in \mathbb{N}^+ \wedge x + y - x = k - x < k\}$

$y := y - x$

$\{INV \wedge x + y < k\}$

$\{INV \wedge x + y < k\}$

$\{INV \wedge x = y\} \Rightarrow \{x = G\}$

Arten von QS-Maßnahmen (1)

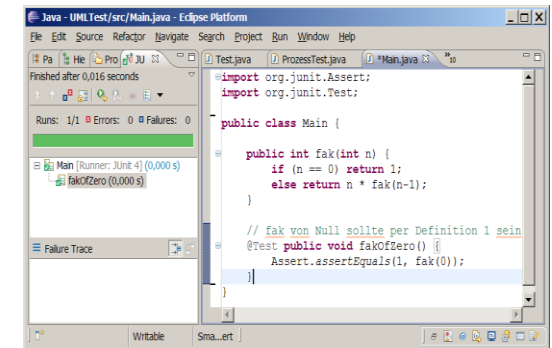
1. Verifikation: Korrektheit formal beweisen

- Sehr aufwendig, daher sehr teuer
- In kritischen Systemen (Luft- und Raumfahrt, AKWs, ...) vorgeschrieben
- Werkzeugunterstützung durch so genannte „Model Checker“, „Hoare Kalkül“

$$\frac{s \vdash_{\Delta} \phi_1 \wedge \phi_2}{s \vdash_{\Delta} \phi_1 \quad s \vdash_{\Delta} \phi_2} \quad \frac{s \vdash_{\Delta} \phi_1 \vee \phi_2}{s \vdash_{\Delta} \phi_1}$$
$$\frac{s \vdash_{\Delta} [a]\phi}{s_1 \vdash_{\Delta} \phi \cdots s_n \vdash_{\Delta} \phi} \text{ if } \{s_1, \dots, s_n\} = \{s' \mid s \xrightarrow{a} s'\}$$

2. Testen: stichprobenartig Experimente durchführen

- **Modultest (Unit Test):** „Korrektheit“ der Implementierung eines Bausteins gegenüber Spezifikation (Schnittstelle)
- **Integrationstest:** „Korrektheit“ der Implementierung eines Pakets/des Gesamtsystems gegenüber Schnittstelle/Anforderungsdefinition
- **Installations-/Abnahmetest:** „Korrektheit“ des Gesamtsystems auf Zielmaschine und hinsichtlich der Vorstellungen des Auftraggebers



```
import org.junit.Assert;
import org.junit.Test;

public class Main {

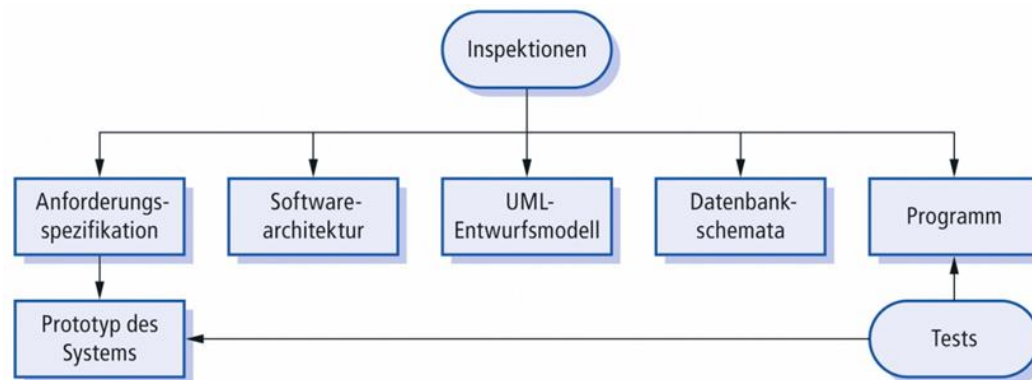
    public int fak(int n) {
        if (n == 0) return 1;
        else return n * fak(n-1);
    }

    // fak von Null sollte per Definition 1 sein
    @Test public void fakOffZero() {
        Assert.assertEquals(1, fak(0));
    }
}
```

Arten von QS-Maßnahmen (2)

3. Prüfen: menschliche (nicht automatisierte) Begutachtung

- **Inspektion** (meistens für ausführbare Dokumente/Quelltext):
 - Ein Test-Team sucht Defekte im Dokument mittels Checklisten, bewertet sie;
 - Autor darf Fragen klären; fester Ablauf Suche, Treffen, Korrektur, Prüfung
- **Walkthrough** (nur für ausführbare Dokumente/Quelltext):
 - Manueller Programmdurchlauf, Test-Team spielt Computer
- **Review** (für alle Dokumente):
 - Beurteilung eines Dokuments durch eine Person,
 - Auch als Teil von Inspektion oder Walkthrough



Vor- und Nachteile von Software-Inspektionen

Vorteile:

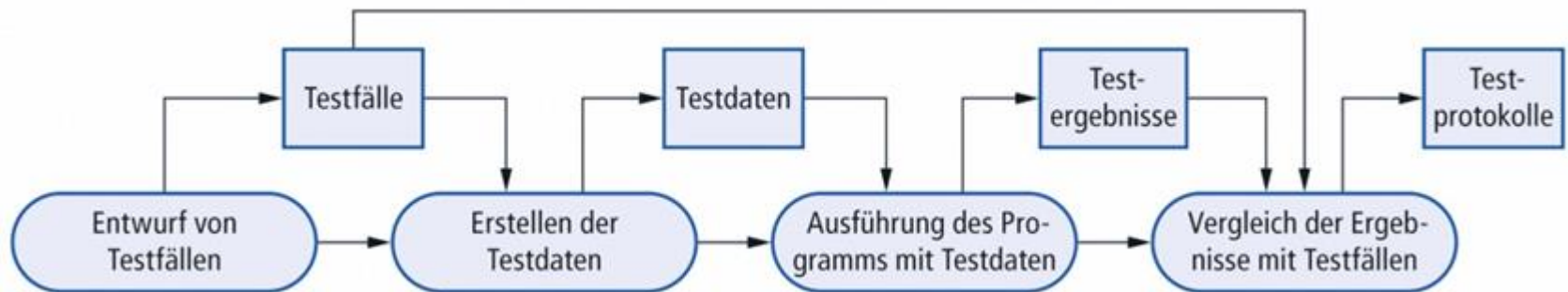
- Beim Testen können bestehende Fehler andere Fehler überdecken
- Unvollständige Software-Versionen können inspiziert aber evtl. nicht getestet werden
- Über die Fehlersuche hinaus können weitere Qualitätsmerkmale geprüft werden

Nachteile:

- Häufig höhere Kosten
- Inspektionen können prüfen, ob Spezifikationen eingehalten werden, aber nicht, ob die wirklichen Benutzeranforderungen erfüllt sind
- Inspektionen können nicht/nur schwer nicht-funktionale Charakteristika überprüfen (Performance, Usability)

→ **Inspektionen und Tests ergänzen sich**

Modell des Softwaretestprozesses



Testphasen

- **Entwicklertests**
- Freigabetests
- Benutzertests

Testphase: Entwicklertests

- Testen des Systems während der Entwicklung durch das Entwicklerteam
- Dabei unterscheiden wir drei Detailstufen:
 - Modultests (Unit Testing)
 - Komponententests
 - Systemtests

Testphase: Entwicklertests

Detailstufe: Modultests

Modultests:

- Testen von einzelnen Einheiten (z.B. Funktionalität von Objekten und Methoden)
- Dabei sollten alle Eigenschaften des Objekts abgedeckt werden:
 - Alle zum Objekt definierten Operationen
 - Alle Attributwerte des Objekts
 - Alle möglichen Zustände des Objekts (Ereignisse simulieren, die zu Zustandswechseln führen)
- Vererbung erschwert das Testen, da z.B. zu testende Funktionalität auch in erbenden Klassen getestet werden muss
- Zwei Arten von Test Cases:
 - Normale Programmausführung (die Komponente tut, was sie soll)
 - Ungewöhnliche/falsche Eingaben (die Komponente kann damit umgehen, ohne abzustürzen)

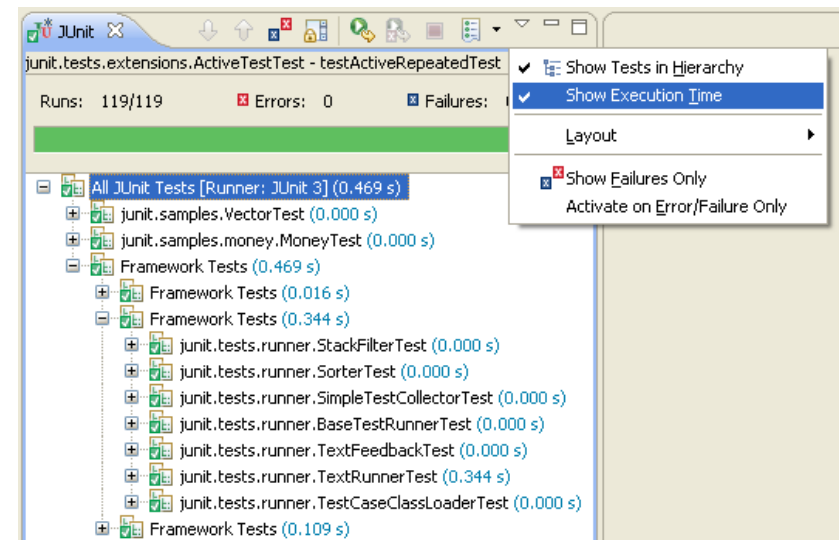
Testphase: Entwicklertests

Detailstufe: Modultests

Automatisierte Modultests:

- **Ziel:** wenn möglich, Modultests automatisieren
- **Durchführung:** Verwendung von Testframeworks (z.B. JUnit, CppUnit), um Tests zu schreiben, durchzuführen und zu protokollieren
- **Vorteil:** Ermöglicht, bei jeder Änderung **alle Tests** laufen zu lassen und das Ergebnis graphisch darzustellen

Was sind weitere Vorteile
Automatisierung von Tests?



Testphase: Entwicklertests

Detailstufe: Modultests

Auswahl der Testfälle: (2 Strategien)

1. Klassenbasierte Tests:

- Identifikation von Gruppen von Eingabedaten (Äquivalenzklassen) mit ähnlichem Verhalten
- Mindestens ein Test pro Gruppe

2. Richtlinienbasierte Tests:

- Verwendung von Richtlinien zur Testfalldefinition
- Richtlinien spiegeln vorausgegangene Erfahrung mit Art von Fehlern wider, die Programmierern häufig unterlaufen

Beispiele?

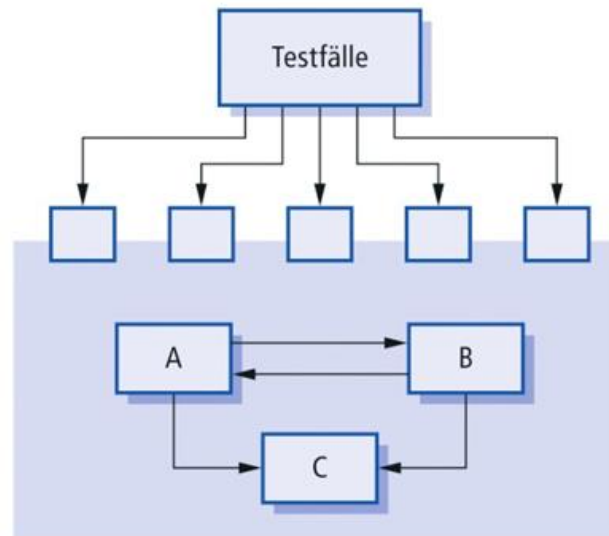
Testphase: Entwicklertests

Detailstufe: Komponententests

Komponententests:

- Softwarekomponenten bestehen aus zusammengesetzten, interagierenden Objekten
- Zugriff über definierte Schnittstellen

→ Hauptfokus: Schnittstellen



Testphase: Entwicklertests

Detailstufe: Systemtests

Systemtests:

- Getestete Komponenten (s. vorherige Detailstufe) werden integriert
- Evtl. zugekaufte Komponenten werden integriert
- Es wird geprüft, ob Zusammenspiel der Komponenten korrekt ist

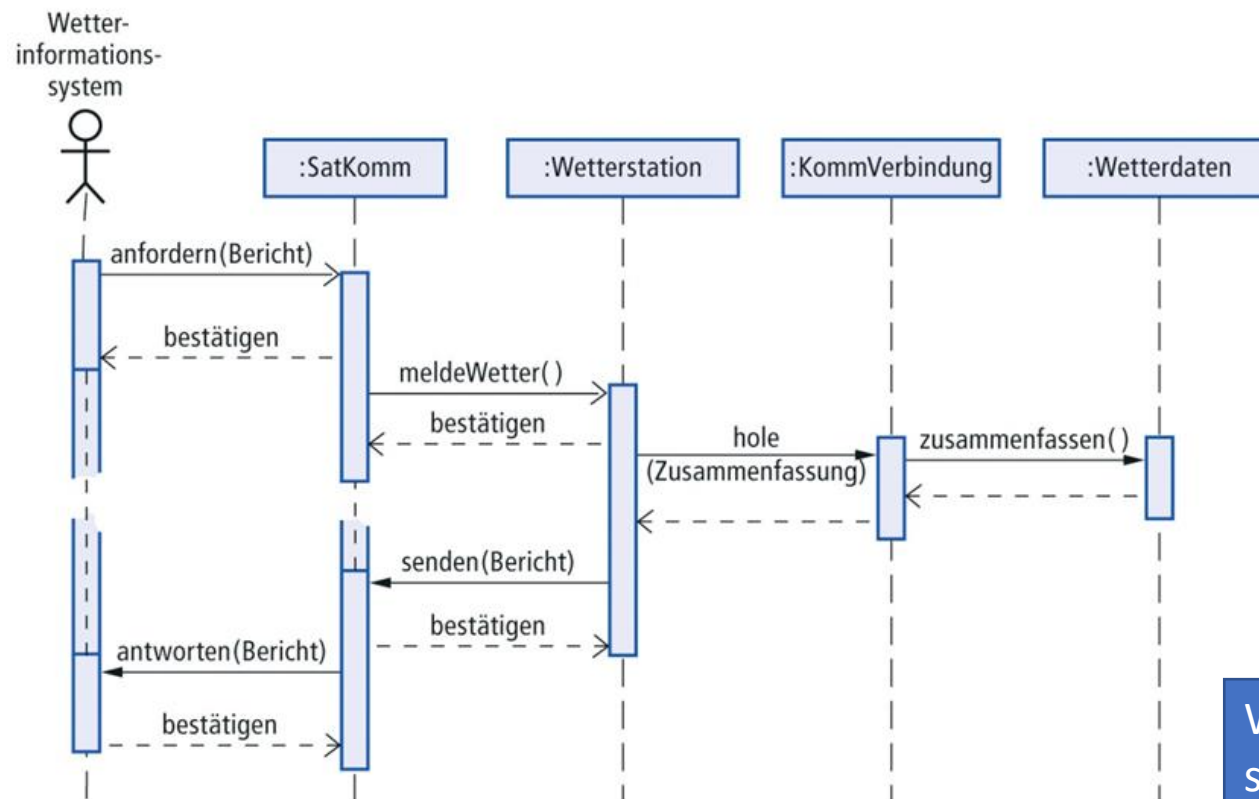
→ Hauptfokus: Interaktion der Komponenten

Welche UML-Diagrammklasse
ist hierfür prädestiniert?

Testphase: Entwicklertests

Detailstufe: Systemtests

- Sequenzdiagramme helfen, Systemtests zu definieren
- Sequenzdiagramme stellen nämlich Interaktion zwischen Komponenten dar

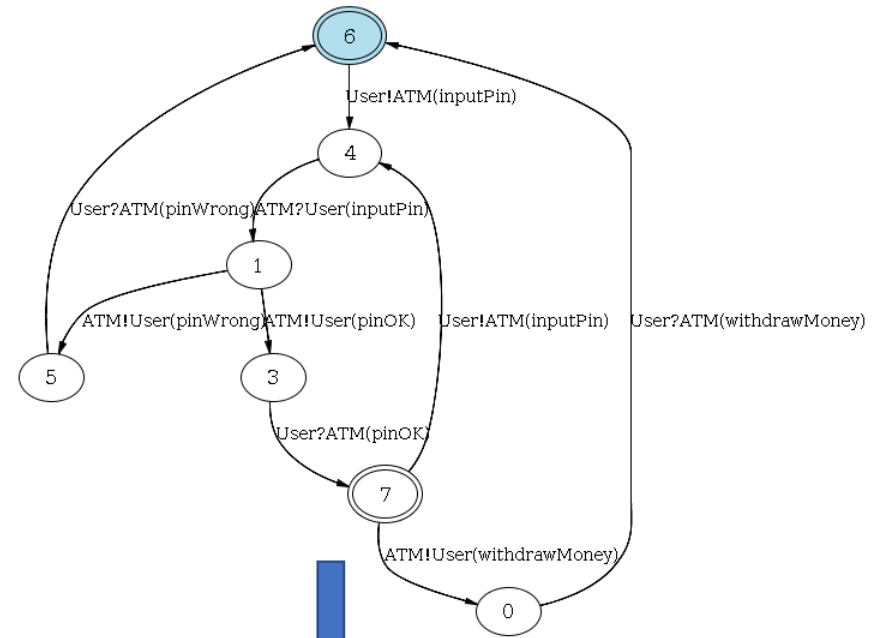
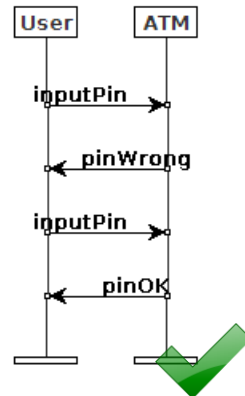
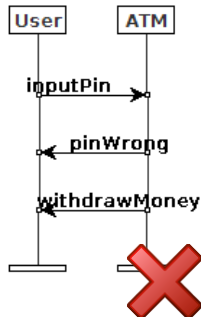
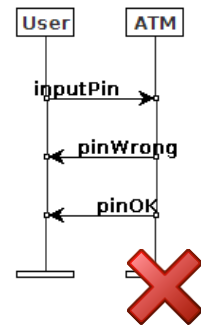


Welche Konventionen sind verletzt?

Testphase: Entwicklertests

Detailstufe: Systemtests

- Toolvorstellung “Smyle”:



Erstellung beliebig vieler Testfälle
aus dem Modell



: positives Szenario



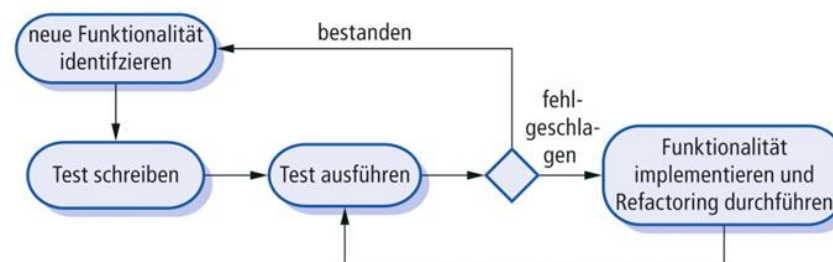
: negatives Szenario

Testgetriebene Entwicklung (Test-Driven Development, TDD)

Testgetriebene Entwicklung:

- Ansatz zur Programmentwicklung, bei dem Codeerstellung und Testen ineinander greifen
- Vorgehen:
 - Identifizierung geforderter Funktionalität (kleinschrittig)
 - Automatisierten Test für Funktionalität schreiben
 - Implementieren der Funktionalität
 - Testen der Funktionalität
- TDD ist oft Teil von agilen Methoden (z.B. Extreme Programming, s. nächstes Kapitel)

Live Codebeispiel TDD



Testphasen

- Entwicklertests
- **Freigabetests**
- Benutzertests

Testphase: Freigabetests

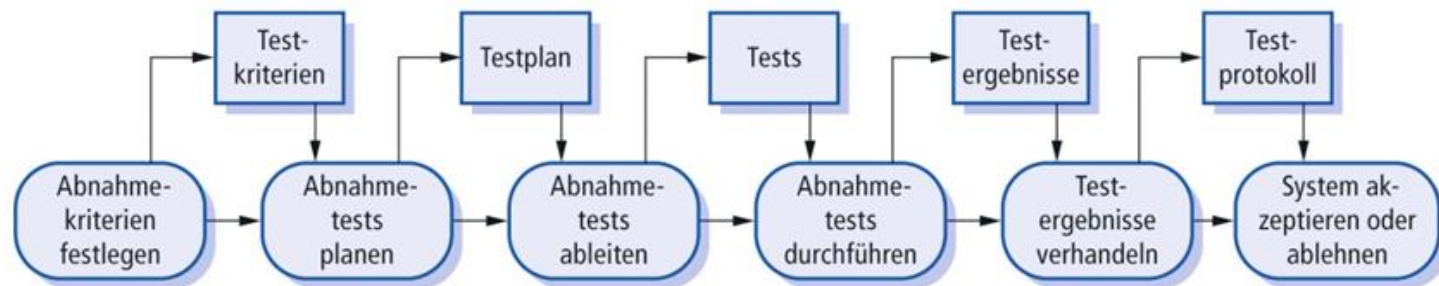
- Separates Testteam testet vollständige Version des Systems, bevor dieses für Benutzer freigegeben wird
- **Ziel:**
 - Nachweis, dass System gebrauchsfähig/ gut genug für externen Gebrauch ist
 - Nachweis der spezifizierten Funktionalität, Performance, Zuverlässigkeit während des normalen Gebrauchs
- **Übliches Mittel:**
 - Blackbox Testing
- **Unterscheidung von Freigabetests in:**
 - Anforderungsbasiertes Testen (dazu sollten Anforderungen überprüfbar/testbar sein)
 - Szenariobasiertes Testen (s. Smyle: Erfinden von Benutzerszenarien, die Nutzung des Systems beschreiben)
 - Leistungstests (z.B.: Performance, Zuverlässigkeit, Lasttests)

Testphasen

- Entwicklertests
- Freigabetests
- **Benutzertests**

Testphase: Benutzertests

- Benutzer eines Systems testen das System in ihrer eigenen Umgebung
- **Beispiele:**
 - Marketinggruppe, die entscheidet, ob Produkt verkauft werden kann
 - **Alphatests:** Benutzer arbeiten mit Entwicklern zusammen
 - **Betatests:** Benutzer erhalten Release der Software und experimentieren damit; Probleme werden mit Entwicklern besprochen
 - **Abnahmetests** mit Kunden; Kunden testen und entscheiden, ob System abgenommen und installiert wird



Grundregeln für Testen (Quiz)



- Tests können Anwesenheit aber _____ von Fehlern zeigen!
- Komponente soll:
 - nicht nur das tun, was sie soll, sondern
 - _____ (s. Tool Smyle)
- Immer mit _____ und _____ Eingaben testen
- Testdaten _____ festlegen und/oder Tests _____ implementieren
- Ein Test darf _____ vom Entwickler durchgeführt werden
- Auf durch Korrekturen _____ Fehler prüfen
- Auf _____ Komponenten konzentrieren
- Tests _____ und Ergebnisse _____ und _____

Literatur

- Software Engineering, I. Sommerville, 9. Auflage, 2012, Pearson Studium