Software-Qualitätssicherung (QS-VU)
LV 180.764

Einführung in Software Testen

Block 3 – Theoretische Hintergründe

Dietmar Winkler

Vienna University of Technology
Institute of Software Technology and Interactive Systems

dietmar.winkler@tuwien.ac.at
http://qse.ifs.tuwien.ac.at

Folgen von Fehlern

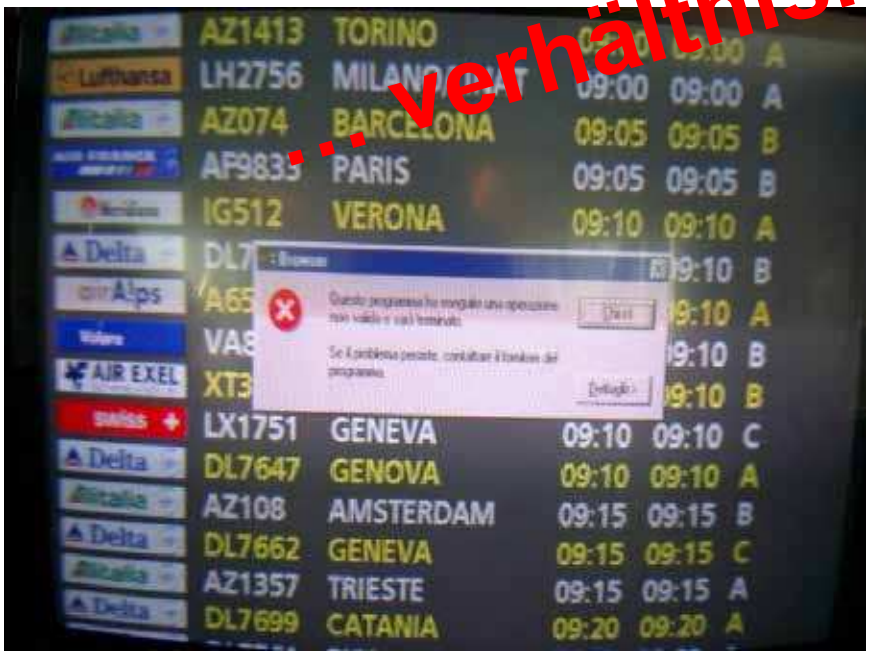
Folgen von Fehlern

Folgen von Fehlern

Folgen von Fehlern

Folgen von Fehlern

. . . verhältnismäßig harmlos . . .



Kleine Fehler => große Auswirkung

- 1991: Patriot Raketenfehler (Ungenaue Zeitberechnung)
  => 28 Menschen wurden getötet.
- 1994: Pentium Divisionsfehler (fehlerhafte FOR-Schleife)
  => >400 Mio USD Verlust für INTEL.
- 1996: Ariane 5 Explosion (Umwandlungsfehler)
  => 500 Mio USD Schaden.
- 1999: eBay: 21 Stunden Stillstand
  => 1.2 Mio Bieter verloren ihre Gebote, Imageverlust.
- 2004: Telekom-Ausfall in Ostösterreich (Telefon, Handy und Bankomaten) von Donnerstag Vormittag bis Freitag Mittag durch einen Softwarefehler.
- 2007: Skype Ausfall: gleichzeitiger Restart „vieler“ Benutzer nach einem Update.
- 2016: Schaltjahr (29.02) wurde nicht erkannt
  => 1200 Koffer am Flughafen Düsseldorf gestrandet.
- …

Quellen:
http://www5.in.tum.de/~huckle/bugs.html
Software Testen VO, Sommersemester 2008

Motivation und Ziele

- Ziel in der Softwareentwicklung ist die Herstellung von qualitativ hochwertigen und testbaren Softwareprodukten:
  - Anwendung von Software Prozessen durch z.B. Life-Cycle Modell, V-Modell, Scrum. *Wann* müssen *welche* Produkte in welchem *Fertigstellungsgrad* und auf welchem *Qualitätsniveau* vorliegen?
  - Konstruktive Methoden zur Herstellung der Produkte, z.B. Model-Driven Development, Test-Driven Development. *Wie* werden die Produkte erstellt?
  - Analytische Methoden zur Verifikation und Validierung der Projektergebnisse, z.B. Reviews und Inspektionen, Testen. Wie erfolgt die Überprüfung der Qualitäts-eigenschaften eines Produktes.
- Aktivitäten der Qualitätssicherung unterstützen die Entwickler in allen Phasen der Software Entwicklung.
- Die Qualitätssicherung ist als integraler Bestandteil des Entwicklungsprozesses zu betrachten!

- Die Verwendung von Prozessen, konstruktiven und analytischen Methoden erhöht die Wahrscheinlichkeit für die Herstellung qualitativ hochwertiger Produkte.
- Software Testen dient – als analytische QS Methode – der Erkennung von Fehlern!

Inhalt und Überblick

- Einführung in Software Testen
  - Definition Testen
  - Verifikation und Validierung
  - Grundlegender Testprozess
- Testen im Softwareentwicklungsprozess
  - Testen im V-Modell
  - Testen in agilen Prozessen am Beispiel Scrum („Test-Driven Development“)
- Testprozess und Testplanung
  - Grundlegender Aufbau von Testfällen
  - Äquivalenzklassen, Testüberdeckung
  - Auswahl von Testfällen (Value-Based Testing)
- Testmethoden
  - Testebenen (Unit-, Modul-, Integrations-, System- & Abnahmetests)
  - Black- und Whitebox Testen

Was ist „Software Testen“?

Was versteht man unter Software Testen?

- „Testen ist das Ausführen eines Programms, mit der Absicht, Fehler zu finden.“ (Myers, 1979)

Was ist ein Fehler?

- „Ein Fehler ist eine Abweichung zwischen einem Programm und dessen Spezifikation“ (Kaner et al, 1999) => Verifikation.
- „Ein Software-Fehler liegt vor, wenn ein Programm nicht das tut, was ein Endbenutzer vernünftigerweise erwartet.“ (Myers, 1979) => Validierung.

- Testen – als Teil der Qualitätssicherung ist ein wichtiger und begleitender Bestandteil des Software-Entwicklungsprozesses, nicht eine Pflichtübung am Ende.

Was ist Testen nicht?

- Testen beschäftigt sich nicht mit der Lokalisierung von Fehlerstellen und dessen Entfernung (vgl. Debugging)
- Durch Testen kann keine Fehlerfreiheit nachgewiesen werden (vgl. Formaler Korrektheitsbeweis).

Verifikation vs. Validierung

Verifikation:

§ „Bauen wir das Produkt richtig?“

§ Umsetzung im Vergleich zur Spezifikation in vorangegangenen Phasen.

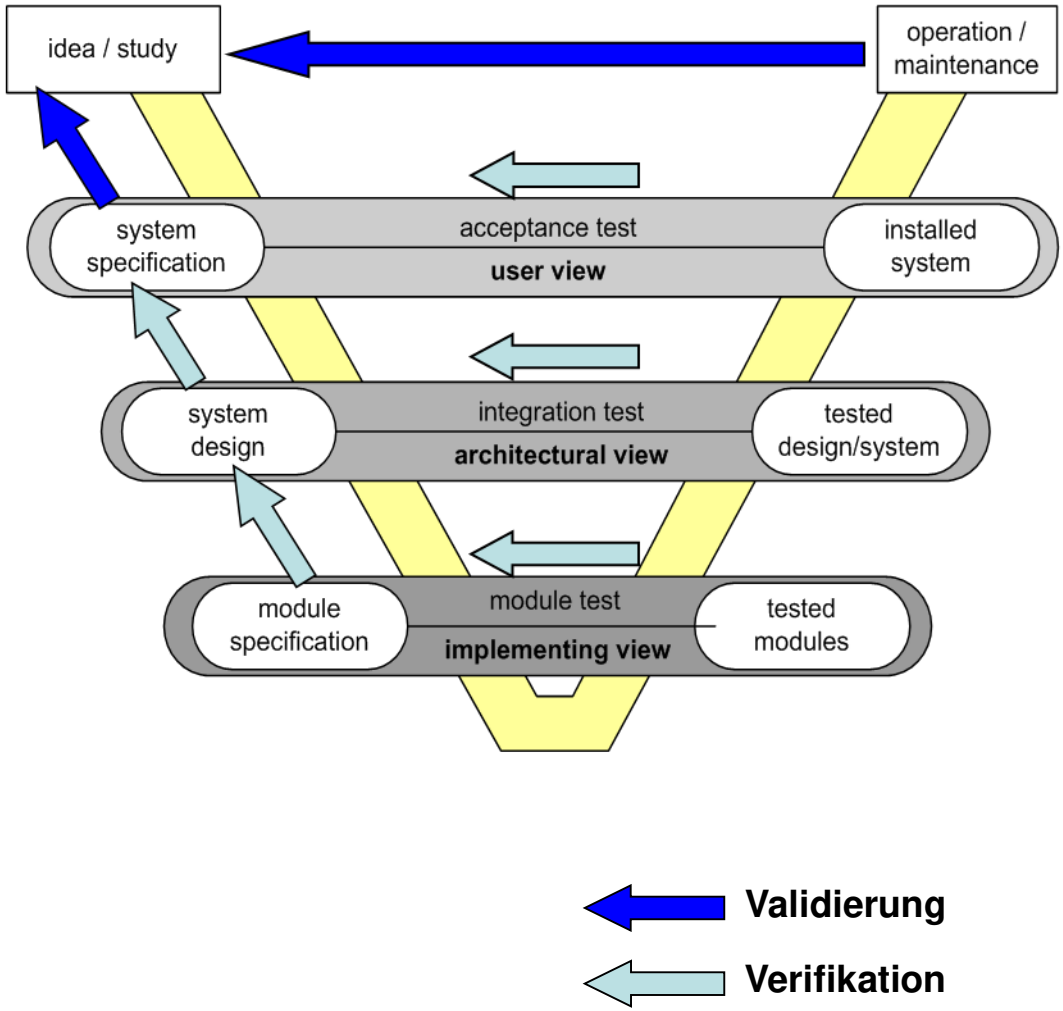
§ Test der Umsetzung gegen die Spezifikation

Validierung:

§ „Bauen wir das richtige Produkt?“

§ Entspricht die Lösung und damit die Spezifikation dem, was der Kunde erwartet?

§ Test der Umsetzung gegen die Nutzeranforderungen.



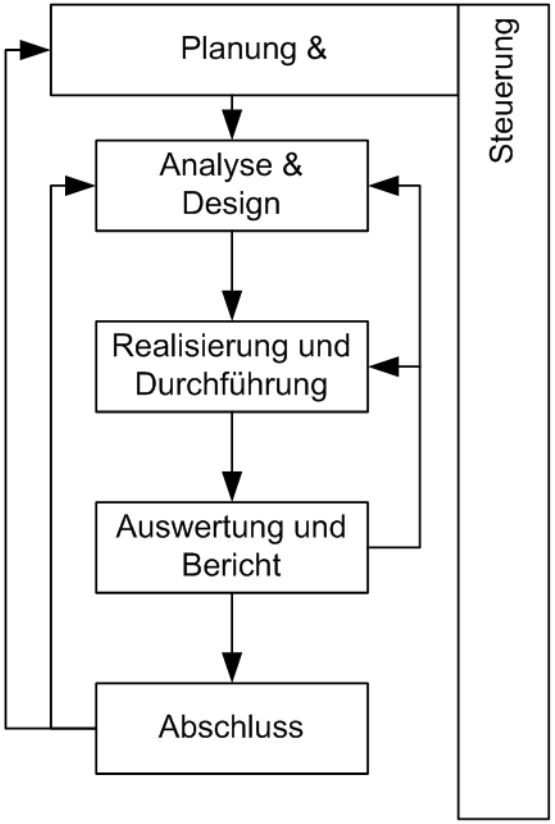
Grundlegender Test Prozess

- Der Testprozess ist in den Entwicklungsprozess eingebettet.
- Was tun Sie, wenn Sie die Aufgabe bekommen, ein Programm zu testen?

  „Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.“ [Pol et al., 2000]

- Phasen im Testprozess
  - Planung / Verwaltung
  - Analyse / Design (Spezifikation)
  - Realisierung und Durchführung
  - Auswertung und Bericht
  - Abschluss

- Planung einzelner Teststufen (z.B. Modultest, Integrationstest, Akzeptanztest)
- Masterplan über alle Teststufen (eingebettet in den Projektplan / Qualitätsplan)



Spillner, 2000

Inhalt und Überblick

- Einführung in Software Testen
  - Definition Testen
  - Verifikation und Validierung
  - Grundlegender Testprozess
- Testen im Softwareentwicklungsprozess
  - Testen im V-Modell
  - Testen in agilen Prozessen am Beispiel Scrum („Test-Driven Development“)
- Testprozess und Testplanung
  - Grundlegender Aufbau von Testfällen
  - Äquivalenzklassen, Testüberdeckung
  - Auswahl von Testfällen (Value-Based Testing)
- Testmethoden
  - Testebenen (Unit-, Modul-, Integrations-, System- & Abnahmetests)
  - Black- und Whitebox Testen

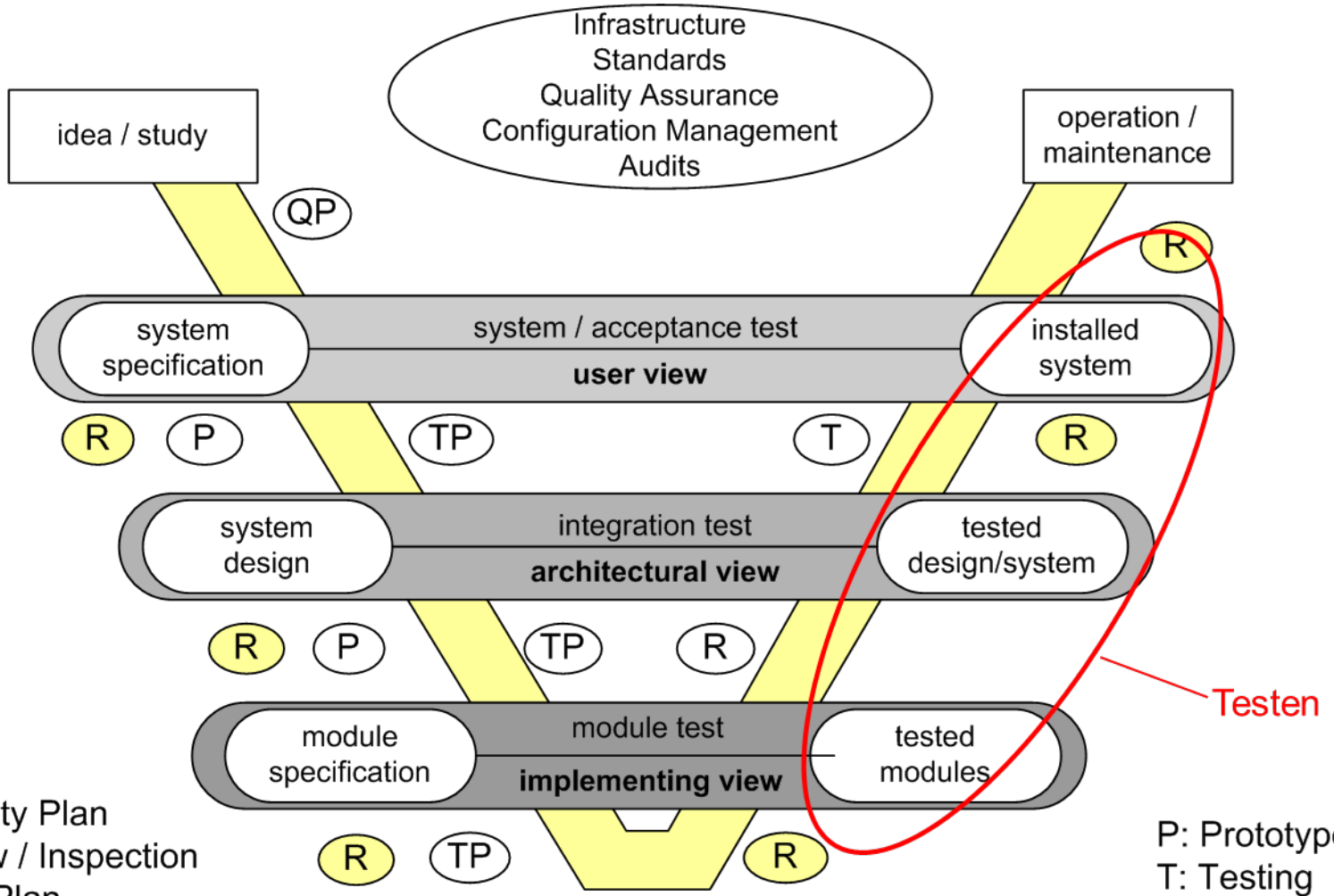
Testen im Entwicklungsprozess

- Qualitätssicherung ist ein integraler Bestandteil eines Entwicklungsprozesses.
- Reviews und Softwareinspektionen (siehe Block 2) dienen dazu, Fehler in frühen Phasen der Entwicklung zu erkennen.
- Testen in traditionellen Prozessen dient dazu, Fehler in implementierten Softwarelösungen zu erkennen (Abweichungen von der Spezifikation (Verifikation) bzw. Abweichungen von den Kundenanforderungen (Validierung)).
- Engere Kopplung von Testen und Implementierung in agilen Ansätzen (Test-Driven Development).

Beispiele:

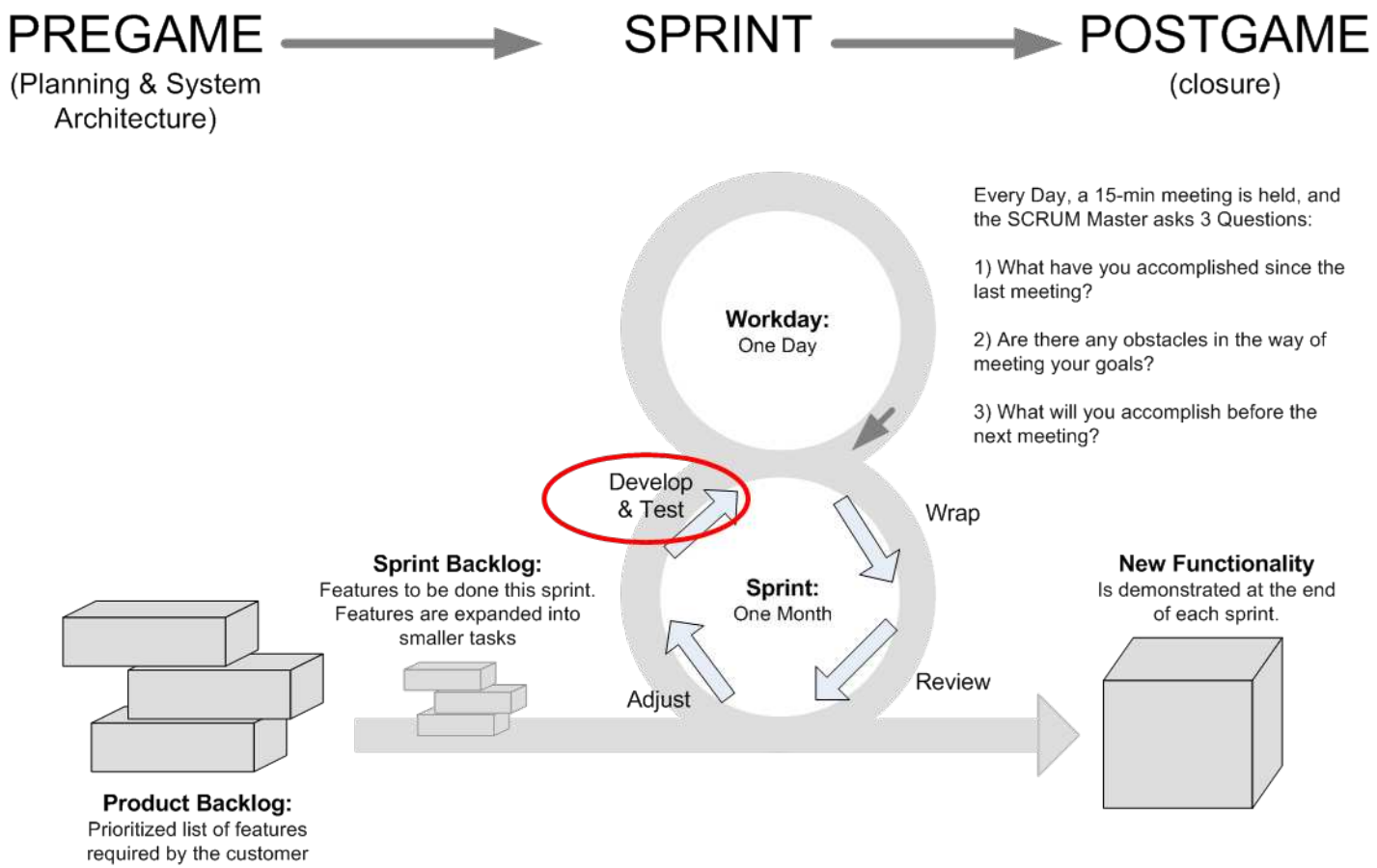
- Testen in traditionellen Prozessen am Beispiel V-Modell
- Testen in agilen Prozessen am Beispiel SCRUM
- Test-Driven Development als agile Praxis.

Testen im V-Modell



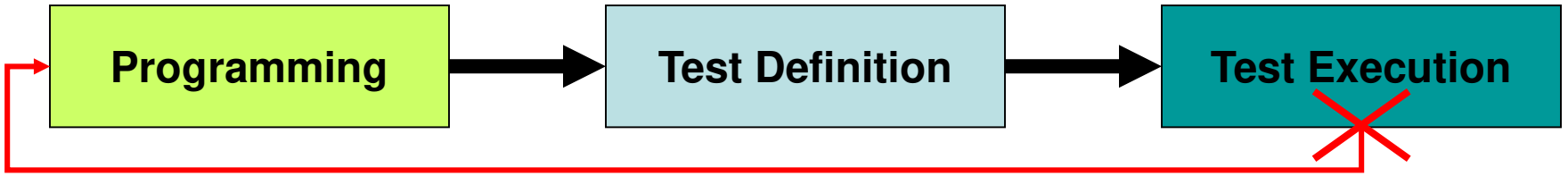
SCRUM Phasen

- Scrum besteht aus einem Set von Prozeduren, Rollen und Methoden für Projektmanagement.
- Selbst-Organisierende Teams.



Test-Driven Development

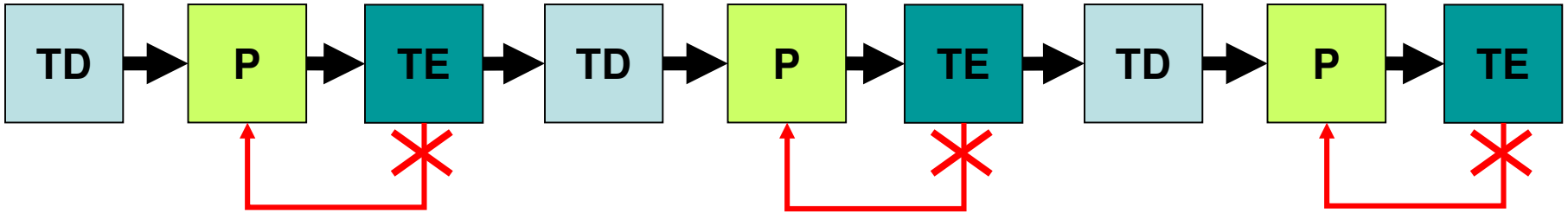
Testen in traditionellen Software Prozessen:



Finding Bugs becomes difficult, particularly
in a team-environment

Testen in agilen Software Prozessen

(1) Write Test before Implementation (2) Short Cycles



Test-Driven Development Process

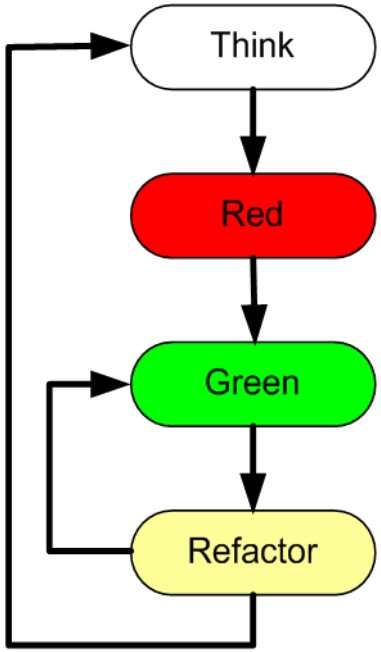
Test Driven Development Schritte:

1. Think
   - Auswahl der zu implementierenden Anforderungen.
   - Spezifikation der Testfälle.

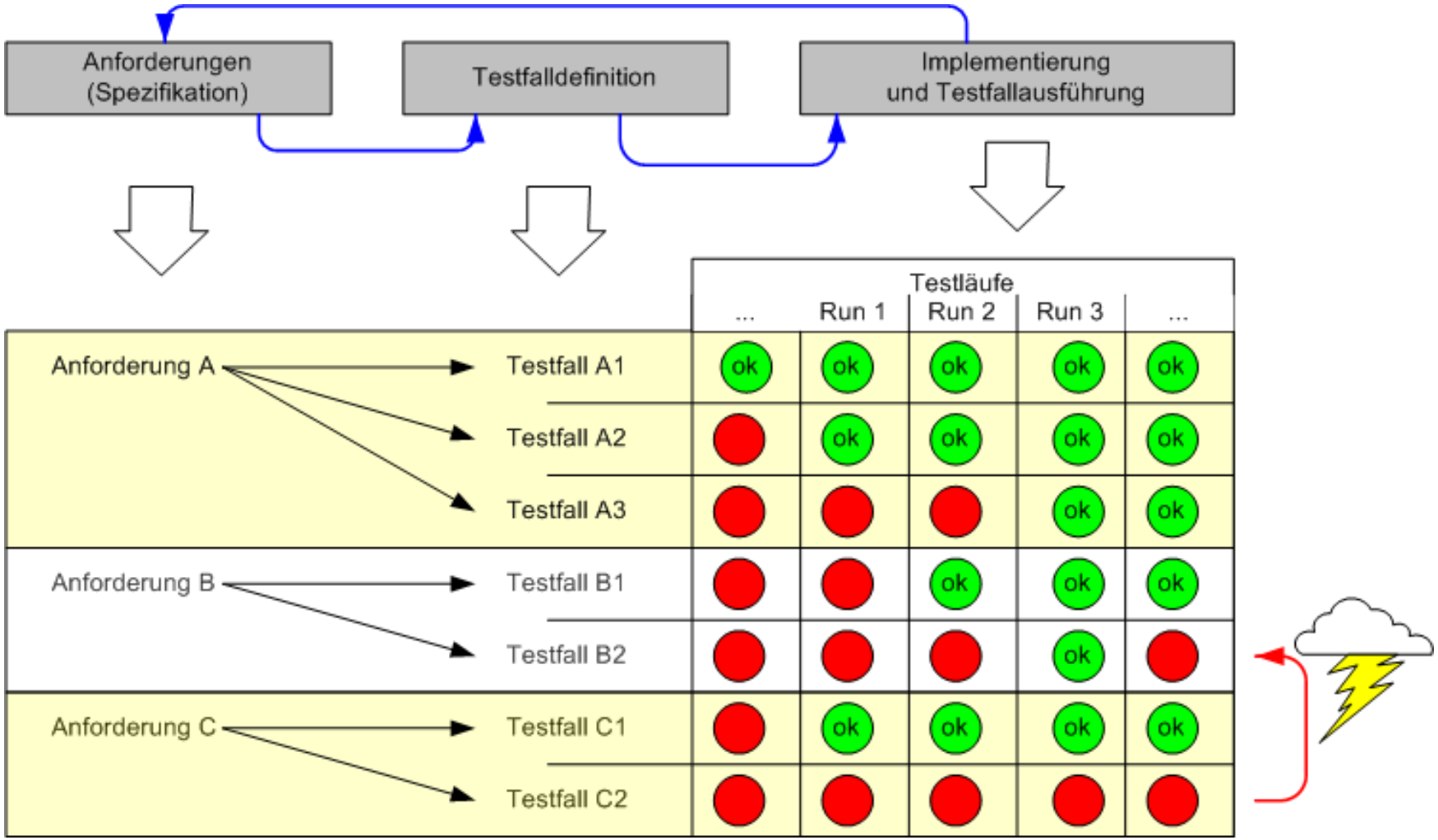
2. Red: Implementierung und Ausführung der Testfälle
   - Alle Tests müssen fehlschlagen.

3. Green: Implementierung der Komponenten und Klassen und Ausführung der Testfälle
   - Testfall erfolgreich weiter bei Schritt 4.
   - Testfall schlägt fehlt weiter bei Schritt 3.

4. Refactor: Änderung und Optimierung der Implementierung ohne Änderung der Funktionalität; Ausführung der Testfälle.
   - Testfälle dürfen nicht fehlschlagen.
   - Auswahl der nächsten Anforderung (Schritt 1)



Frequent Testing in a CI&T Environment



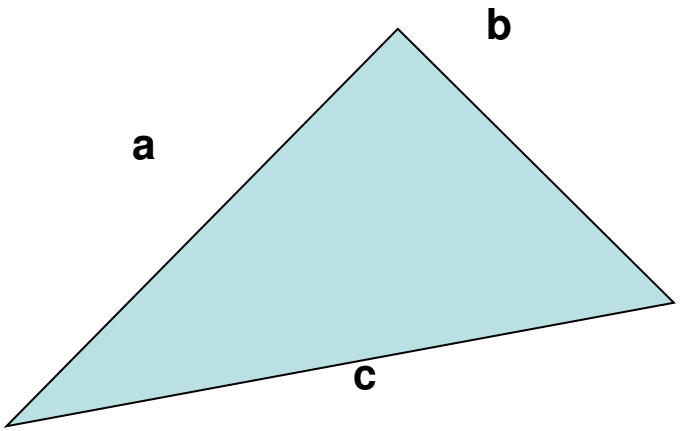
Inhalt und Überblick

- Einführung in Software Testen
  - Definition Testen
  - Verifikation und Validierung
  - Grundlegender Testprozess
- Testen im Softwareentwicklungsprozess
  - Testen im V-Modell
  - Testen in agilen Prozessen am Beispiel Scrum („Test-Driven Development“)
- Testprozess und Testplanung
  - Grundlegender Aufbau von Testfällen
  - Äquivalenzklassen, Testüberdeckung
  - Auswahl von Testfällen (Value-Based Testing)
- Testmethoden
  - Testebenen (Unit-, Modul-, Integrations-, System- & Abnahmetests)
  - Black- und Whitebox Testen

Testen: Ein Beispiel

§ Ein Programm liest 3 ganze Zahlen a, b und c und soll feststellen, ob sie die Seiten eines
  – gleichseitigen Dreiecks
  – gleichschenkligen Dreiecks
  – rechtwinkeligen Dreiecks
  – sonstigen gültigen Dreiecks

bilden.



§ Wie würden Sie diese Methode testen?
§ Welche Teststrategie würden Sie verwenden?
§ Welche Testfälle sind erforderlich?

Beispiel: Testfälle (1)

Gültige Dreiecke

- Gleichseitig: 3, 3, 3
- Gleichschenkelig: 5, 5, 3
- Rechtwinkelig: 3, 4, 5
- Sonstiges: 3, 5, 7
- Permutationen davon
  (3,5,5), (5,3,5),
  (3,5,4), (4,5,3), (4,3,5), (5,3,4), (5,4,3),
  (3,7,5), (5,7,3), (5,3,7), (7,3,5), (7,5,3)

Ungültige Dreiecke

- a + b < c: 3, 3, 7
- a + b = c: 3, 4, 7
- Negative Seitenlänge: 3, 4, -5
- 0, 0, 0
- Nur 2 Seitenlängen (?): 3, 4
- 4 Seitenlängen (?): 3, 4, 4, 5
- Permutationen davon
  (3,7,3), (7,3,3),
  (3,7,4), (7,4,3),...
  (3,-5,4), (-5,3,4)...

Aufbau von Testfällen

- Tests bestehen aus einer Menge von Testfällen (Testsuite)

- Ein Testfall besteht aus einer Menge (V, E, A, R)
  - V: Vorbedingungen (z.B. System- oder DB-Zustände).
  - E: Eingabewerte.
  - A: Aktionen zur Eingabe der Werte.
  - R: erwartete Ergebnisse.

- Testerfolg
  - Ein Test ist erfolgreich, wenn er einen Fehler gefunden hat.
  - Tests, die keinen Fehler aufdecken, sollen bessere Information über die Produktqualität liefern (systematische Testüberdeckung!).

Beispiel: Testfälle (2)

Test Nr	Typ	Vorbedingung	Eingabe wert	Beschreibung / Aktion	Erwartes Ergebnis	Tatsächliches Ergebnis
..	..	..	..	..	..	..
30	NF	System läuft.	(3,3,3)	Dreieckü berprüfung wird aufgerufen;	Ausgabe „gleichseitig“	gleichseitig
31	NF	System läuft.	(5,3,3)	Dreieckü berprüfung wird aufgerufen;	Ausgabe: „gleichschenklig“	gleichschenklig
32	FF	System läuft.	(3,3,7)	Dreieckü berprüfung wird aufgerufen;	Ausgabe: „ungültig“	gleichseitig
33	FF	System läuft.	(3,4,4,5)	Dreieckü berprüfung wird aufgerufen;	Ausgabe: „ungültig“	ungültig
..	..	..	..	..	..	..

- Einschätzung der Qualität von Testfällen.
- Notwendigkeit gute und wichtige Testfälle auszuwählen.

Test-Abdeckung

Abdeckung aller Funktionen

§ Jede spezifizierte Funktion wird in mindestens einem Testfall ausgeführt.

```
class C {
  void f1 (int x) {...}
  int f2 (boolean x, int y) {...}
}

-> c.f1(...), c.f2(...), ...
```

Abdeckung aller Eingabeklassen

§ Für jede Funktion wird jeder Eingabeparameter in mindestens einem Testfall verwendet.

```
int f2 (boolean x, int y) {...}
// x: false | true
// y: <0 |0 | >0

-> f2(false, -1), f2(true, -1),
   f2(false,  0), f2(true,  0),
   f2(false,  3), f2(true,  3),..
```

Abdeckung aller Ausgabeklassen

§ Für jede Funktion wird jede Ausgabewertklasse in mindestens einem Testfall erzeugt.

```
int f2 (boolean x, int y) {...}
// return: == 0 | >0

x = f2 (false, 0)
x = f2 (true,  3)
```

Äquivalenzklassenzerlegung

- Zerlegung einer Menge von Daten (Input oder Output) in Untermengen (Klassen), die äquivalente Ergebnisse oder Auswirkungen produzieren.
- Jede Klasse(nkombination) sollte mindestens einmal getestet werden.
- Dadurch sinkt die Anzahl der Testfälle und somit der Testaufwand.

Vorgehensweise:

- Finde zu testende Funktionen.
- Finde Eingabe-Vektor (A, B, C, ...) (= Faktoren) je Funktion.
  - Explizite: Parameter.
  - Implizite: globale Variablen, Systemzustand (Objekte, Datenbank), …
- Finde für jeden Faktor seine un-/gültigen Äquivalenzklassen (Klassen A1, A2, ...; B1, B2, ...; ...) und wähle je einen Wert aus jeder Klasse.
- Wahl der Kombinationen: (A1, B1), (A1, B2), ... bestimmt Intensität.

Äquivalenzklassenzerlegung

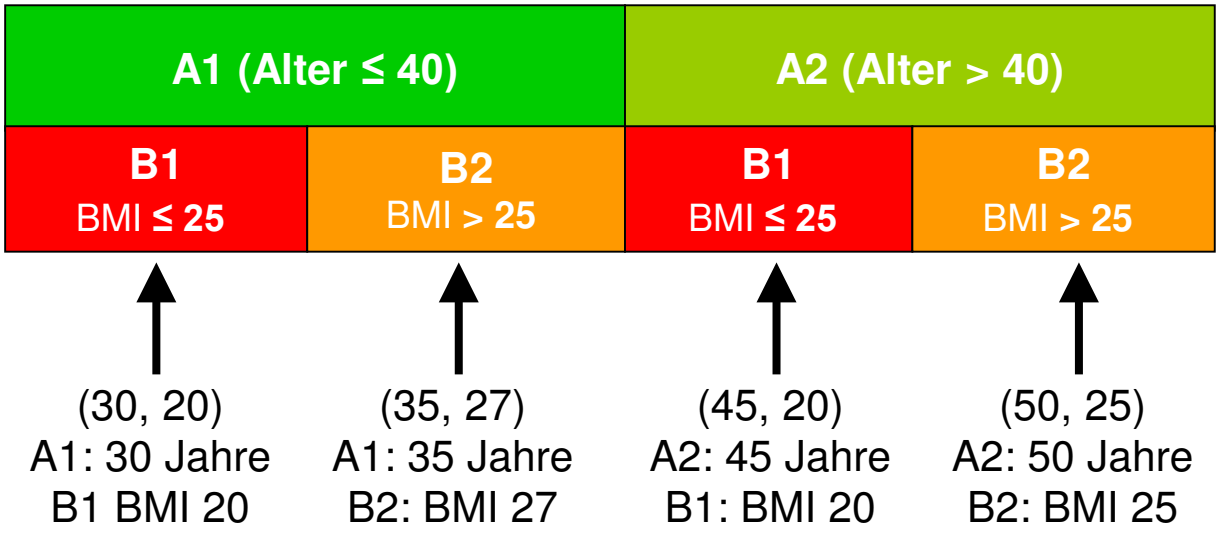
- Anforderung: 18 < Alter ≤ 65 → drei Äquivalenzklassen
  - A1: Alter ≤ 18 (ungültig)
  - A2: 18 < Alter ≤ 65 (gültig)
  - A3: Alter > 65 (ungültig)

  Auswahl eines Repräsentanten einer Äquivalenzklasse,

  Drei Testfälle: x ∈ A1, z.B. 10, x ∈ A2, z.B. 35, x ∈ A3, z.B. 70

Äquivalenzklassenzerlegung

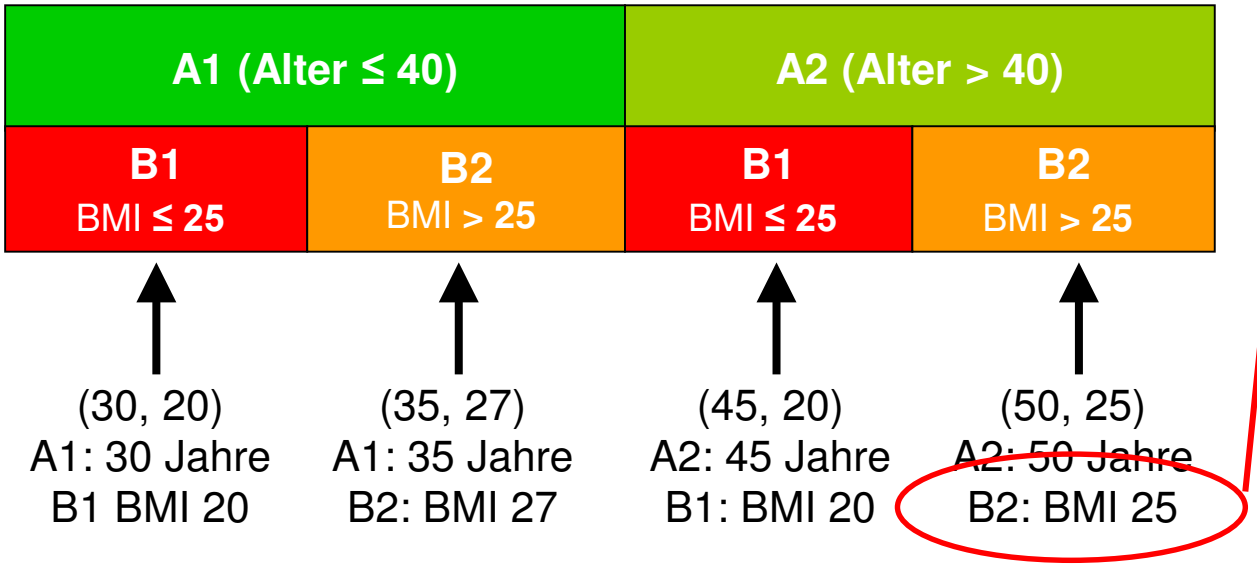
- Alter über 40, Body Mass Index (BMI) über 25 je 2 Klassen, 4 Kombinationen
  - A1: Alter ≤ 40, B1: BMI <= 25, z.B. (30, 20)
  - A2: Alter > 40, B1: BMI <= 25, z.B. (45, 20)
  - A1: Alter ≤ 40, B2: BMI > 25, z.B. (35, 27)
  - A2: Alter > 40, B2: BMI > 25, z.B. (50, 25)



Äquivalenzklassenzerlegung

- Alter über 40, Body Mass Index (BMI) über 25 je 2 Klassen, 4 Kombinationen
  - A1: Alter ≤ 40, B1: BMI <= 25, z.B. (30, 20)
  - A2: Alter > 40, B1: BMI <= 25, z.B. (45, 20)
  - A1: Alter ≤ 40, B2: BMI > 25, z.B. (35, 27)
  - A2: Alter > 40, B2: BMI > 25, z.B. (50, 25)

Häufige Fehlerquelle: Grenzwerte



Grenzwertanalyse

- Erweiterung der Äquivalenzklassenzerlegung für bessere Überdeckung.
- Grenzwerte werden als Klassenrepräsentanten gewählt
  je Klassengrenze ein Testfall.

- Beispiel: Anforderung: 18 < Alter ≤ 65
  - 18 (ungültig), 19 (gültig), 65 (gültig) and 66 (ungültig)
  - Ev. mehrere Testfälle je Klassengrenze: Alter ≤ 18
  - 17 (gültig), 18 (gültig) and 19 (ungültig)
    Bedenke Tippfehler: Alter = 18!

Wertbeitrag von Software Testen

- Zentrale Fragestellung: Ist jeder Testfall, jeder Fehler, jede Anforderung gleich viel „wert“?
- Beispiel: 10 unterschiedliche Bezahlverfahren einer Online-Plattform, wobei 80% des Umsatzes immer über dasselbe Verfahren abgewickelt wird.
- Die Antwort: NEIN => Value-Based Testing.

- Requirements-Based testing

  Welche Anforderungen bringen dem Kunden den meisten nutzen?

- Risk-Based testing

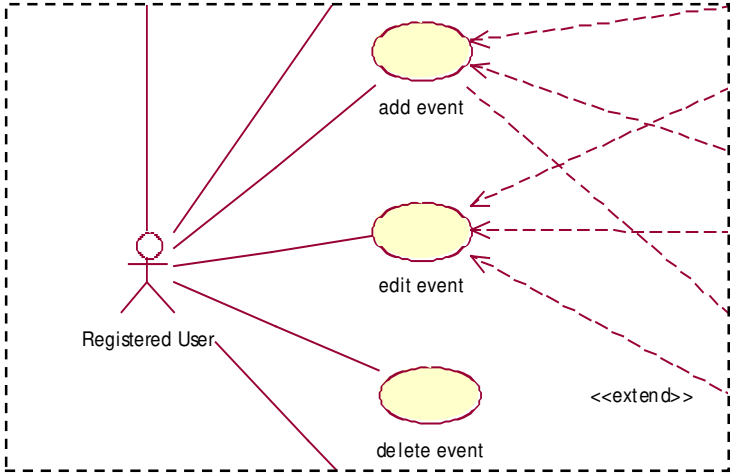
  Welche Risiken gefährden diesen Nutzen?

- Test case selection techniques

  Welche Testfälle addressieren dieses Risiko?

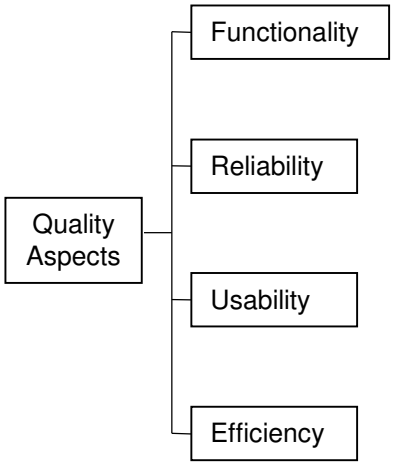
Beispiel: Risk-Based Testing (1)

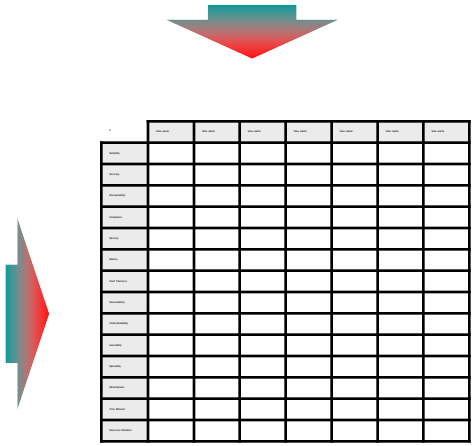
Features / Anwendungsfälle



Risikoeinschätzung:
A … Critical,
B ... Important,
C ... Less important

Quality Criteria





Security aspect \ Feature	View events	View details	Add event	Edit event	Attach document	Set properties	Delete event
General identity	B	B	C	C	C	C	C
Message content authenticity	B	B	C	C	C	C	C
Message content origin	C	C	B	B	B	A	B
Integrity	C	C	A	A	A	A	
Secrecy and privacy	B	B	C	B	C	C	C
Accountability			B	B	B	B	B

Ramler et al, 2006

Inhalt und Überblick

- Einführung in Software Testen
  - Definition Testen
  - Verifikation und Validierung
  - Grundlegender Testprozess
- Testen im Softwareentwicklungsprozess
  - Testen im V-Modell
  - Testen in agilen Prozessen am Beispiel Scrum („Test-Driven Development“)
- Testprozess und Testplanung
  - Grundlegender Aufbau von Testfällen
  - Äquivalenzklassen, Testüberdeckung
  - Auswahl von Testfällen (Value-Based Testing)
- Testmethoden
  - Testebenen (Unit-, Modul-, Integrations-, System- & Abnahmetests)
  - Black- und Whitebox Testen

Test-Stufen und Test-Typen

Test-Stufen (Software Prozess)

- (Privater Test)
- Unit-Test
- Integrations-Test
- System-Test
- Regressions-Test: Test nach Durchführung von Änderungen.
- Akzeptanz-Test

Test-Typen

- Review
- Funktionaler Test
- Massen-Test (große Datenmenge)
- Stress-Test (Systemverhalten bei Überlastung)
- Performance-Test (z.B. Antwortzeiten
- …

- Jede Teststufe besteht aus einem oder mehreren Testtypen.
- Jeder Testtyp testet ein oder mehrere Qualitätskriterien und verlangt eigene Test-Methoden.

„Private Tests“ & Komponententests

Private Tests

- Tests durch den Entwickler vor der „Freigabe“ an andere (Tester).
- Meist undokumentiert.
- Endkriterium oft „Ich bin mir sicher, dass das Programm korrekt funktioniert“.
- Häufige Fehleinschätzung: „Hier kann eh nichts schief gehen“.
- Besser: objektive Kriterien bzw. Methoden,
  z.B. Abdeckung aller Codezeilen (C0-Überdeckung), aller Methoden etc.

Komponententests

- Ziel: Aufspürung von Fehlern in der Implementierung.
- Module werden jeweils einzeln gegen ihre Spezifikation getestet: Übergabe von Daten, Prüfung des Outputs.
- Modultests erlauben eine genaue Lokalisierung und frühe Erkennung von Implementierungsfehlern.
- Modultests können bei größeren Systemen sehr aufwendig werden; daher ist bei größeren Tests besonderer Wert auf strukturiertes Vorgehen und Dokumentation des tatsächlichen Vorgehens zu legen.

Integrations- System- & Abnahmetests

Integrationstests

- Ziel: Test der Interaktion zwischen Modulen.
- Zusammenführung von Modulen zu größeren Strukturen (Häufig durch Entwickler).
- Integrationsstrategien: „Big-Bang“, Top-Down, Bottom-Up, Build-Integration.
- Build-Integration und Integrationstest sind – sofern möglich – zu bevorzugen.

Systemtests

- Ziel: Test des Gesamtsystems gegen die Anforderungsanalyse bzw. den System-Entwurf.
- Validierung bzw. Verifikation.

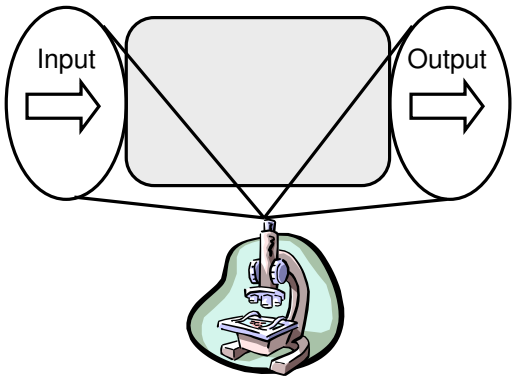
Abnahme- oder Akzeptanztest

- Ziel: Validierung des Systems gegen die Kundenanforderungen.
- Vom Kunden durchgeführter Systemtest,
  meist mit einer Submenge an Testfällen aus dem Systemtest.

Black Box vs. White Box

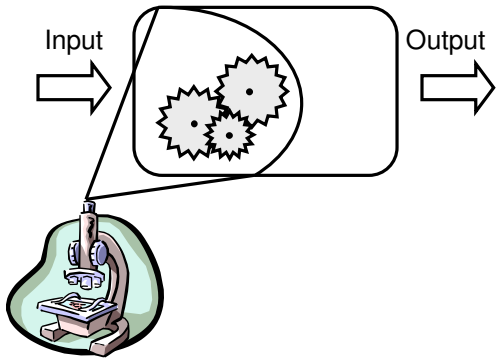
Black Box Tests

- Basiert auf Spezifikation.
- Wissen über innere Struktur wird ignoriert.
- Daten-getriebener Test.
- Input-Output-getriebene Tests.
- Anforderungsüberdeckung.
- Partitionierung (Äquivalenzklassen) der Eingabe-Daten.



White Box (Glass Box)

- Basiert auf der Struktur von Code bzw. SE Modellen.
- Wissen über innere Struktur notwendig.
- Logik-getriebene Tests.
- Überdeckung von Knoten, Kanten, Pfaden.
- Partitionierung (Äquivalenzklassen) von Daten für Bedingungsauswertung.



Zusammenfassung

- Software Testen ist – wie alle QS-Techniken – als integraler Bestandteil eines Entwicklungsprojektes zu behandeln.
- Testen hat zum Ziel, Fehler in Software-Systemen zu finden. Je früher wichtige Fehler gefunden werden, desto kostengünstiger ist die Korrektur.
- Ein Testfall besteht aus einer Beschreibung der nötigen Vorbedingungen, der Eingabewerte, der für die Eingabe notwendigen Aktionen und der erwarteten Ergebnisse (Orakel).
- Ein qualitativ hochwertiger Testfall hat eine hohe Chance, wichtige Fehler zu finden, und hat eine hohe Abdeckung. Die Abdeckung ist ein Maß für die Effektivität und Intensität von Tests.
- Zur Herleitung von Testfällen gibt es verschiedene Ansätze: Black-Box und White-Box.
- In Software-Projekten gibt es verschiedene Testphasen: Private Tests, Modultests, Integrationstests, Systemtests und Abnahmetests

Referenzen

Bücher und Skripten

- Kaner C., Falk J., Nguyen H.Q.: “Testing Computer Software", Wiley, 1999.
- Pol M., Koomen T., Spillner A.: “Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software, mit TPI und TMap”, 2000.
- Sommerville I.: „Software Engineering“; 8. Auflage; Addison Wesley, 2007.
- Spillner A., Linz T., "Basiswissen Software-Test", dPunkt, 2003.
- Thaller G.E.: "Software-Test", dPunkt, 2002
- Ramler R., Biffl S.: „Value-Based Management of Software Testing“, in S. Biffl, A. Aurum, B. Boehm, H. Erdogemus, P. Grünbacher (editors). Value-Based Software Engineering, Springer, 2006.
- Myers G.J., Sandler C.: The Art of Software Testing“, Wiley, 1979.
- Beedle M., Schwaber K.: „Agile Software Development with Scrum“, Prentice Hall, 2002.
- Schatten A., Biffl S., Demolsky M., Gostischa-Franta E., Östreicher T., Winkler W.: Best Practice Software Engineering. Eine praxiserprobte Zusammenstellung von komponenten-orientierten Konzepten, Methoden und Werkzeugen, Spektrum Akademischer Verlag, 2010.

Web Ressourcen

- Biffl S., Winkler D., Frast D.: „Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung“, Skriptum zur Lehrveranstaltung, 2004, http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm
- Software Engineering – Best Practices (Blog): http://best-practice-software-engineering.blogspot.com
- Software Engineering – Best Practices (Web): http://bpse.ifs.tuwien.ac.at/
- IEEE Software Engineering Body of Knowledge: http://www.swebok.org; Kapitel11 „Software Quality“.
- Skriptum zur Lehrveranstaltung: „Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung“, http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm.

Welche Testebenen sind aus Ihrer Sicht die wichtigsten?