Softwarequalität

Vorlesung 8 – Testen: Black-box-Verfahren

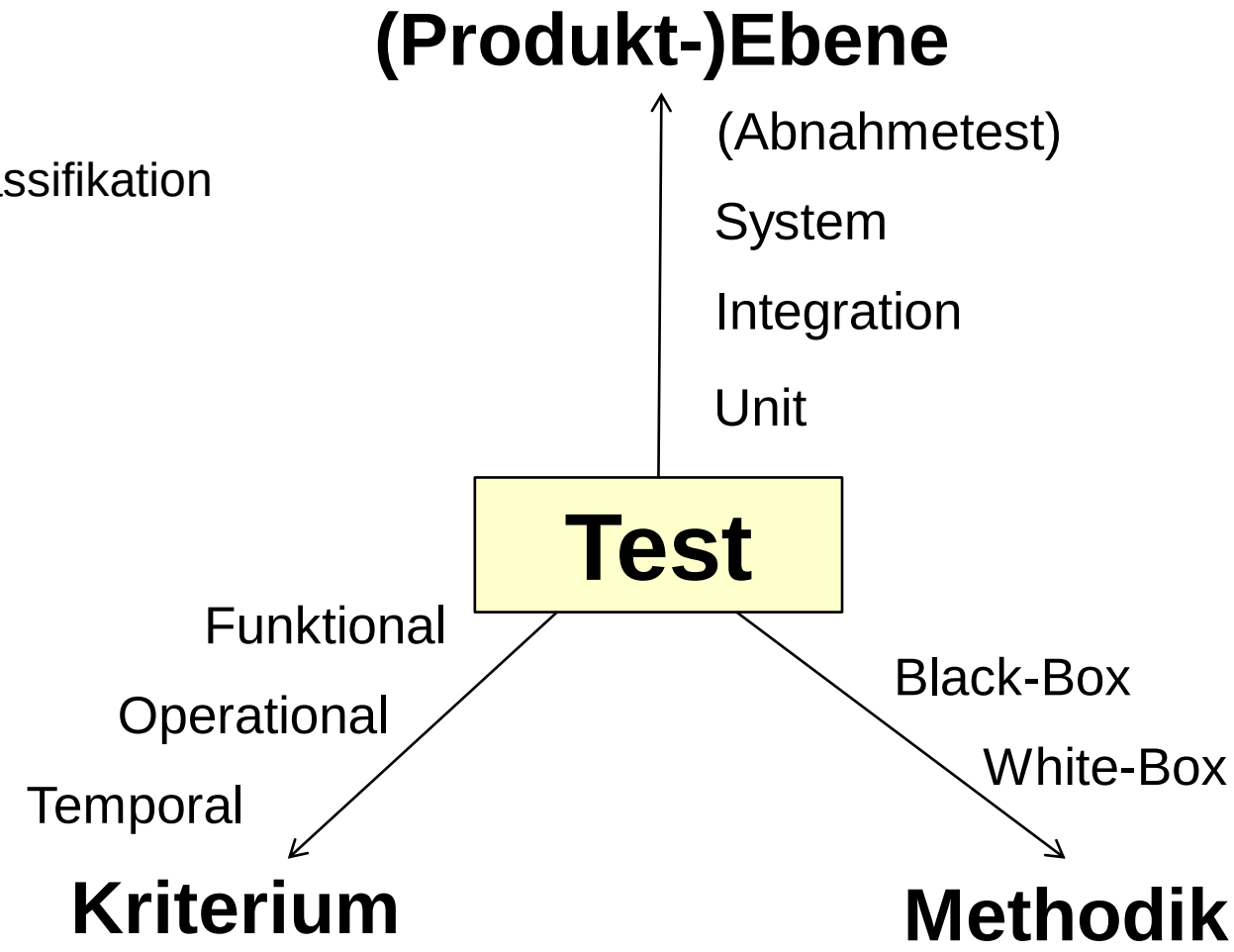
Prof. Dr. Joel Greenyer

30. Mai 2016

Testklassifikation

In der 4. Vorlesung… (24.4.)

Eine mögliche Klassifikation



basierend auf Hoffmann, Dirk W. *Software-Qualität*, eXamen.press, 2. Auflage, Springer Berlin Heidelberg, 2013

Hintergrund: Fault, Error, Failure – Defekt, Error, Fehler

In der 4. Vorlesung… (24.4.)

- Ein **Defekt** (engl. **Fault**) ist eine **fehlerhafte Stelle im Programmcode (statisch)**
  - Hier hat ein Entwickler etwas falsch gemacht
  - Z.B. falsche Anweisungen, Anweisungen zu viel, zu wenig, …
- Ein **Error** ist ein inkorrekter **interner Zustand zur Laufzeit** eines Programms
  - Intern: Nicht direkt von außen zu beobachten
- Ein **Fehler** (**Fehlverhalten**, engl. **Failure**) ist ein von außen beobachtbares Verhalten eines Programms, welches vom spezifizierte Verhalten abweicht
  - Vgl. V2: **Ein Fehler ist die Nichterfüllung einer Anforderung**
  - Manchmal werden auch Defekte als **Fehler** bezeichnet

Hintergrund: Vom Defekt zum Error zum Fehler

In der 4. Vorlesung… (24.4.)

- **Damit ein Defekt zu einem Fehler führt** müssen drei Bedingungen erfüllt sein
  - **Erreichbarkeit**: Die Stelle des Defekts im Programms muss *erreicht* werden
  - **Infektion**: Der *Zustand* des Programms muss daraufhin *inkorrekt* sein
  - **Propagation**: Der inkorrekte Zustand muss zu einer *inkorrekten Ausgabe führen* (also beobachtbar sein)

- Das **Ziel eines Tests** (auch des Testers) ist, **Fehler zu finden!**
  - Das **Finden und Beheben des *Defekts*** (z.B. durch Debugging) ist wiederum **Aufgabe des Entwicklers**

Beispiel: Methode getTriangleKind

In der 4. Vorlesung… (24.4.)

- Gegeben ist eine Methode mit drei Integer-Parametern, welche die Kantenlängen eines Dreiecks beschreiben
  - Die Methode soll ausgeben, ob das Dreieck **gleichseitig**, **gleichschenklig** oder **ungleichseitig** ist.
- **Was sind sinnvolle Tests für diese Methode (u. warum)?**

```
/**
 * @param sideA length of first side of triangle
 * @param sideB length of second side of triangle
 * @param sideC length of third side of triangle
 * @return
 * 1 if triangle is scalene (all sides have different lengths)
 * 2 if triangle is isosceles (exactly two sides have equal lengths)
 * 3 if triangle is equilateral (all sides have equal lengths)
 */
public static int getTriangleKind(int sideA, int sideB, int sideC){
        ...
}
```

Black-Box-Tests

In der 4. Vorlesung… (24.4.)

- Wie viele Tests gibt es theoretisch für diese Methode?
  - Eingabe: drei Integer, mit jeweils (in Java) 32 Bit
- Mögliche Eingaben sind also 2^{96} = ~$7.92*10^{28}$
  - Nehmen wir an, wir könnten eine Milliarde Tests pro Sekunde durchführen, dann würde das Ausführen aller Tests ~$7.92*10^{19}$ Sekunden = **~2.512.308.552.583 Jahre** dauern

```
/**
 * @param sideA length of first side of triangle
 * @param sideB length of second side of triangle
 * @param sideC length of third side of triangle
 * @return
 * 1 if triangle is scalene (all sides have different lengths)
 * 2 if triangle is isosceles (exactly two sides have equal lengths)
 * 3 if triangle is equilateral (all sides have equal lengths)
 * 0 if sides form no triangle (one value greater the sum of the two others)
 */
public static int getTriangleKind(int sideA, int sideB, int sideC){
        ...
}
```

Wir können meist nicht jeden Fall testen

In der 4. Vorlesung… (24.4.)

- **Wir können meist nicht jeden Fall testen!**
- Daher ist das Ziel möglichst viele verschiedene Fälle abzudecken, in denen der Fehler liegen könnte

- Dazu gibt es mehrere Methoden
  - **Alle Anforderungen abdecken**
    - Alle Fälle und Sonderfälle in den Anforderungen abdecken
  - **Äquivalenzklassenbildung**
    - Die Eingaben werden in Bereiche eingeteilt, für die angenommen wird, dass das Programm für alle Eingaben aus einem Bereich dasselbe Verhalten hat
    - (Zum Teil ist auch Ziel hier, alle Anforderungen abzudecken)
  - **Grenzwerttests**
    - Programmierer machen oft Fehler an oder in der Nähe von Grenzen von Äquivalenzklassen

Und noch einige mehr…

Wir können meist nicht jeden Fall testen

In der 4. Vorlesung… (24.4.)

- **Wir können meist nicht jeden Fall testen!**
- Daher ist das Ziel möglichst viele verschiedene Fälle abzudecken, in denen der Fehler liegen könnte

- Dazu gibt es mehrere Methoden

Alle Anforderungen abdecken

- Alle Fälle und Sonderfälle in den Anforderungen abdecken

– **Äquivalenzklassenbildung**

- Die Eingaben werden in Bereiche eingeteilt, für die angenommen wird, dass das Programm für alle Eingaben aus einem Bereich dasselbe Verhalten hat
- (Zum Teil ist auch Ziel hier, alle Anforderungen abzudecken)

– **Grenzwerttests**

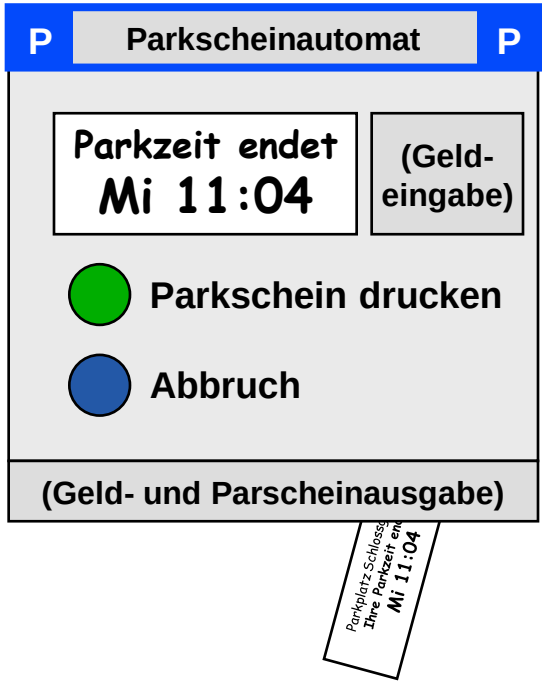
- Programmierer machen oft Fehler an oder in der Nähe von Grenzen von Äquivalenzklassen

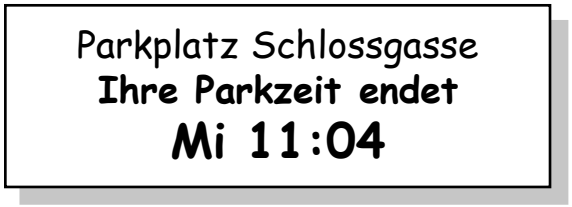
Und noch einige mehr…

Alle Anforderungen abdecken – Beispiel: Parkscheinautomat

In der 4. Vorlesung… (24.4.)

- Sie sollen die Software eines Parkscheinautomats testen
- Parkscheine sollen für **maximal zwei Stunden** gekauft werden können
- Parkscheine sollen Ende der Parkzeit anzeigen





Abdecken von Anforderungen

In der 4. Vorlesung… (24.4.)

- Wie viele Anforderungen decken die Tests ab?

Anforderungen an `getParkingTimeEnd`

R01 Höchstparkzeit: 2 Stunden, Beträge über 240c verfallen
R02 Exception wenn Zahlbetrag kleiner 50c
R03 Exception wenn Zahlbetrag nicht teilbar durch 10
R04 Stunde kostet 120c, andere Parkzeiten entsprechen
R05 Parken kostet von 9:00 bis 19:00 Uhr
R06 Parken kostet nichts von 19:00 bis 9:00

Testfalltabelle (unsystematisch erstellt)

ID	jetztZeit	centsPayed	Sollwert parkzeitEnde	R01	R02	R03	R04	R05	R06
T01	12.4.-10:00	240	12.4.-12:00				X	X	
T02	3.4.-9:00	100	3.4.-9:50				X	X	
T03	28.11.-11:30	20	Exception!		X				
T04	2.1.-14:12	50	2.1.-14:37				X	X	
T05	29.2.-17:46	67	Exception!			X			

Abdecken von Anforderungen

In der 4. Vorlesung… (24.4.)

- Wie viele Anforderungen decken die Tests ab?

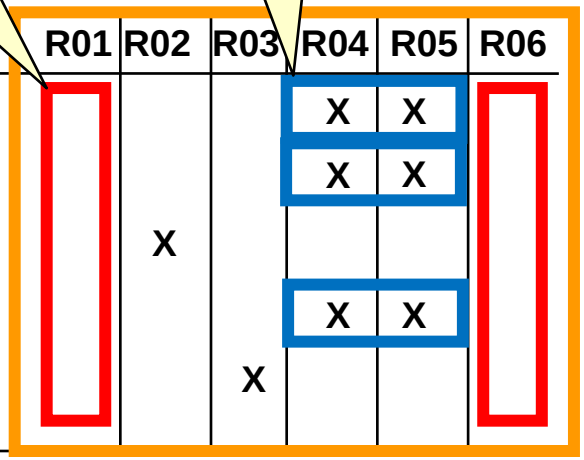
Disclaimer: Es könnte trotzdem Gründe geben, diese Tests durchzuführen. Ggf. ist 9:00 Uhr ein Grenzwert, wo ein Programmierer etwas falsch machen könnte…

Problem 1:
Einige Anforderungen (R01 und R06) sind *gar nicht* getestet

Problem 2:
Einige Anforderungen sind mehrfach getestet. Für die *Effizienz* reicht ein Kreuz pro Spalte.

Testfalltabelle (unsystematisch erstellt)

ID	jetztZeit	centsPayed	Sollwert parkzeitEnde
T01	12.4.-10:00	240	12.4.-12:00
T02	3.4.-9:00	100	3.4.-9:50
T03	28.11.-11:30	20	Exception!
T04	2.1.-14:12	50	2.1.-14:37
T05	29.2.-17:46	67	Exception!



Wir können meist nicht jeden Fall testen

- Wir können meist nicht jeden Fall testen
- Daher ist das Ziel möglichst viele verschiedene Fälle abzudecken, in denen der Fehler liegen könnte

- Dazu gibt es mehrere Methoden
  - **Alle Anforderungen abdecken**
    - Alle Fälle und Sonderfälle in den Anforderungen abdecken
  - **Äquivalenzklassenbildung**
    - Die Eingaben werden in Bereiche eingeteilt, für die angenommen wird, dass das Programm für alle Eingaben aus einem Bereich dasselbe Verhalten hat
    - (Zum Teil ist auch Ziel hier, alle Anforderungen abzudecken)
  - **Grenzwerttests**
    - Programmierer machen oft Fehler an oder in der Nähe von Grenzen von Äquivalenzklassen

Und noch einige mehr…

Äquivalenzklassenbildung

- Beispiel: Die Methode `getSeason` berechnet aus der Monatszahl die Jahreszeit
- **Annahme**: die Methode ***verhält sich gleichartig*** z.B. für die Eingaben 3,4,5
- Wenn ich also nicht alles testen kann (Testen kostet), dann macht es Sinn, dass ich mich darauf beschränke, einen Wert aus jeder Partition zu testen

```
/**
 * @param month number of month (must be number between 1 and 12)
 * @return “Spring” if month 3,4,5; “Summer” if month 6,7,8;
 * “Fall” if month 9,10,11; “Winter” if month 12,1,2;
 */
public static String getSeason(int month) {
        ...
}
```

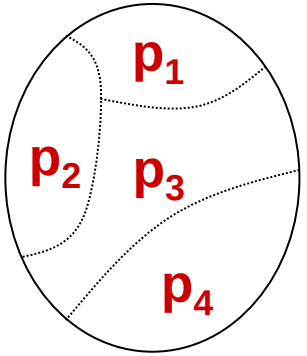
Äquivalenzklassenbildung

- Wertebereich der Eingabe
  - month ∈ {n | n ∈ ℕ ∧ 1 ≤ n ≤ 12}, andere int-Werte sind ungültig
- Partitionierung – hier als Menge von Teilmengen des Wertebereichs modelliert
  - {{3, 4, 5}, {6, 7, 8}, {9, 10, 11}, {12, 1, 2}}
- Tests z.B. (Eingabe → Sollwert)
  - 3 → „Spring“; 7 → „Summer“; 9 → „Fall“; 2 → „Winter“

```
/**
 * @param month number of month (must be number between 1 and 12)
 * @return “Spring” if month 3,4,5; “Summer” if month 6,7,8;
 * “Fall” if month 9,10,11; “Winter” if month 12,1,2;
 */
public static String getSeason(int month) {
        ...
}
```

Äquivalenzklassenbildung

- Bei der **Äquivalenzklassenbildung** muss eine Partitionierung $P = \{p_1, p_2, \ldots, p_n\}$ des Wertebereichs der Eingabe W gefunden werden.
- Die Idee: Wertebereich der Eingabe so aufteilen,
  - dass **zu vermuten ist**, dass sich Programm für Eingaben aus einer Teilmenge p_i **gleichartig verhält**
  - oder **vermutlich den gleichen Fehler auslöst**



- **Partitionierung** bedeutet immer
  - Die sich ergebenden Teilmengen müssen **disjunkt** sein
    - Also für alle $p_i, p_j \in P$ gilt $p_i \cap p_j = \emptyset$
  - Die sich ergebenden Teilmengen müssen **vollständig** sein, also zusammen den gesamten Wertebereich W ergeben
    - Also $W = \bigcup_{i=1-n} p_i$

Mehrdimensionale Äquivalenzklassenbildung

- Besteht die Eingabe **aus mehreren Parametern**, oder kann ein Parameter **verschieden partitioniert** werden, ergibt sich eine **mehrdimensionale Äquivalenzklassenbildung**

- Beispiel Parkscheinautomat:
  - startDate
    1. nach 9:00 Uhr und vor 19:00 Uhr (zahlpflichtig)
    2. sonst
  - centsPayed
    1. < 50 c oder nicht teilbar durch 10 (ungültig)
    2. >= 50 und <= 240 und teilbar durch 10 (gültig, „verfällt nicht“)
    3. > 240 und teilbar durch 10 (gültig, „verfällt“)

centsPayed \ startDate	1.	2.
1.	T11	T21
2.	T12	T22
3.	T13	T23

- Teilmengen T11, …, T23 bilden wieder eine Partitionierung