

Softwarequalität

Vorlesung 4 –Testen: Einführung , Black-box-Verfahren

Prof. Dr. Joel Greenyer

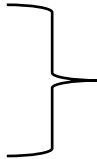


25. April 2016

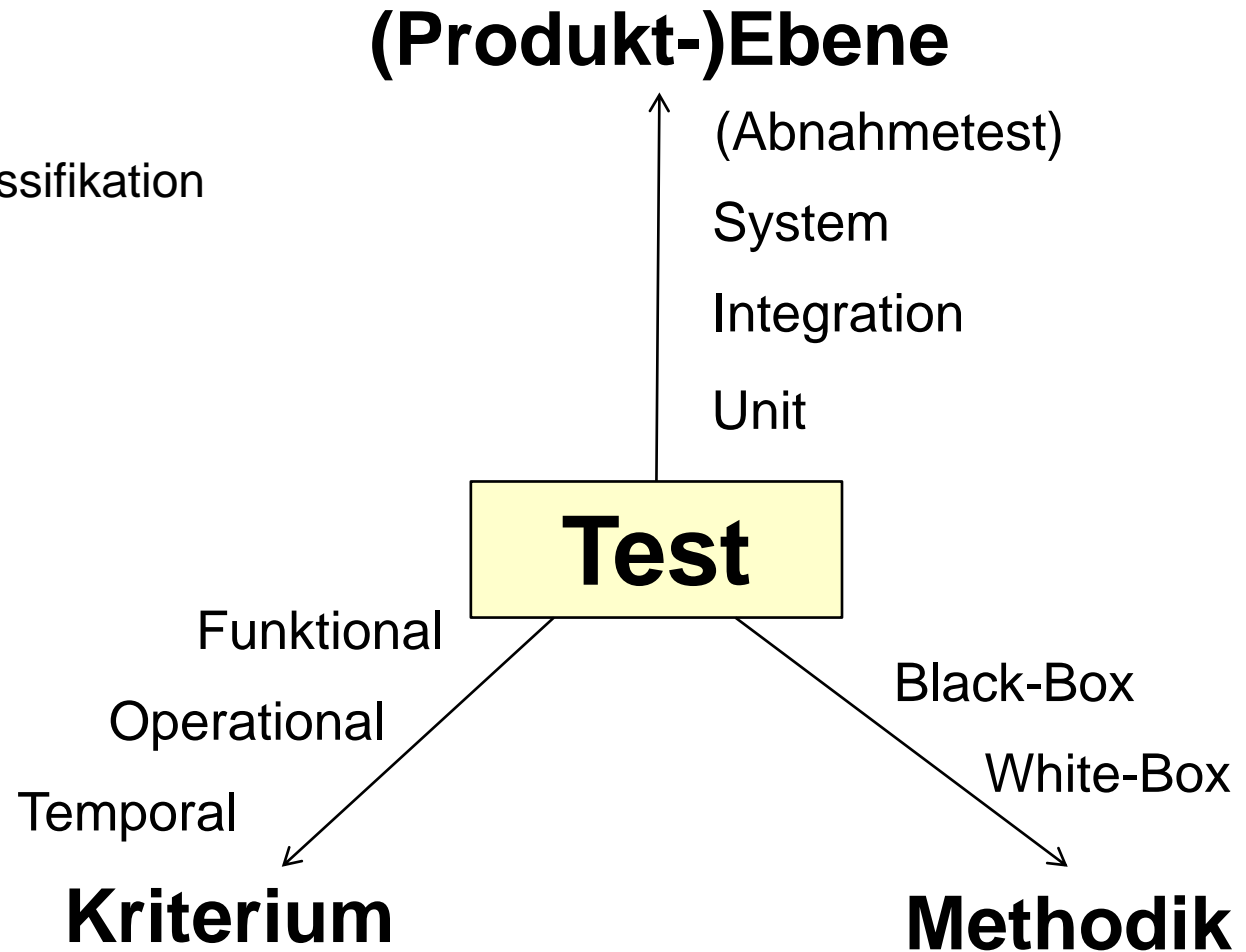


Analytische Qualitätssicherung auf Produktebene

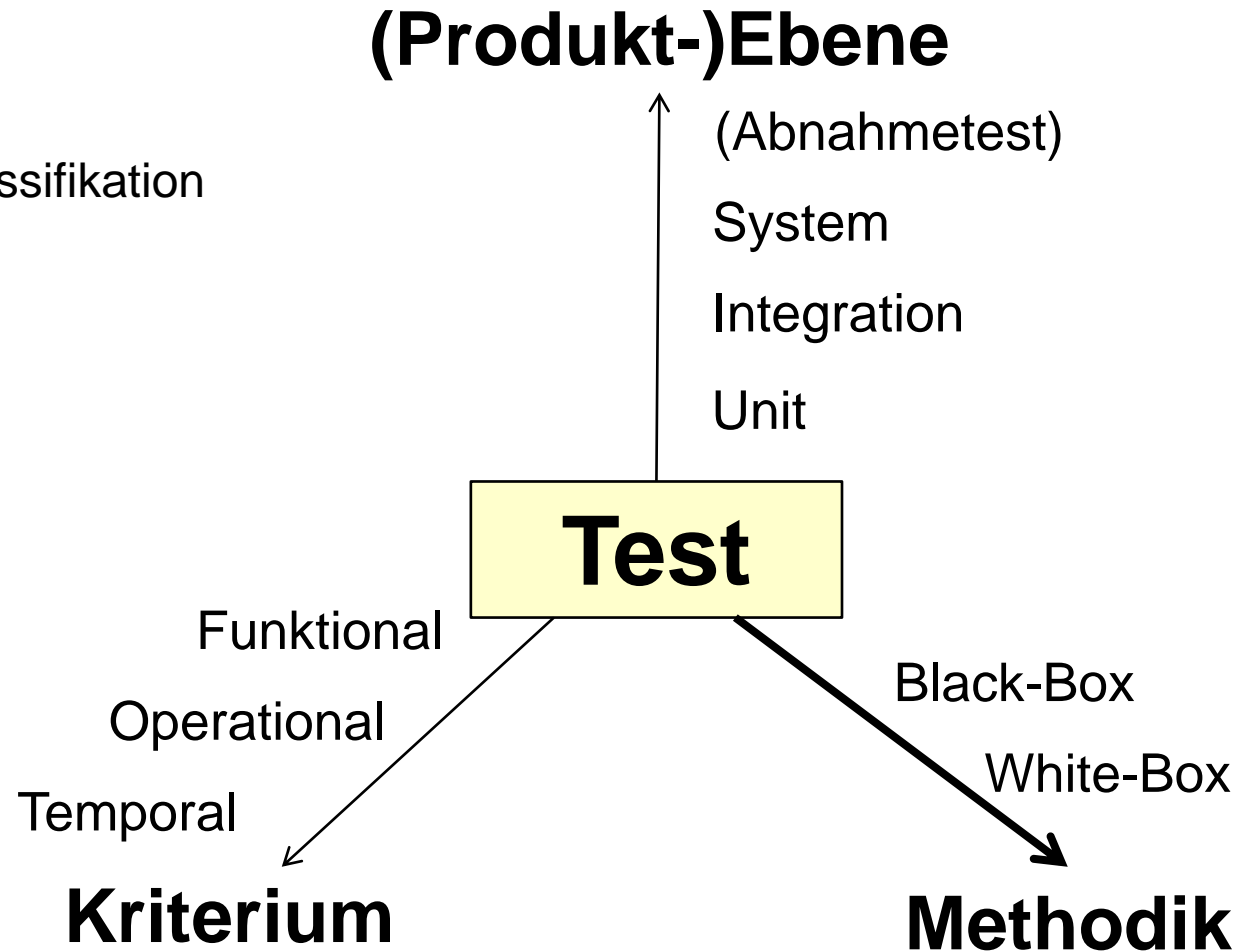
- (Wdh.) **Ein Test ist die Ausführung eines Programms mit dem Ziel, Fehler zu finden**
- Testen ist eine Qualitätssicherungsmethode
 - **analytisch**
 - auf **Produktebene**
- Weitere analytische Qualitätssicherungsmethode auf Produktebene
 - Statische Analyse
 - Inspektionen/Reviews
 - Formale Verifikation


 - ... von Code, Modellen oder Dokumenten
 - ... von Code oder Modellen

Eine mögliche Klassifikation



Eine mögliche Klassifikation



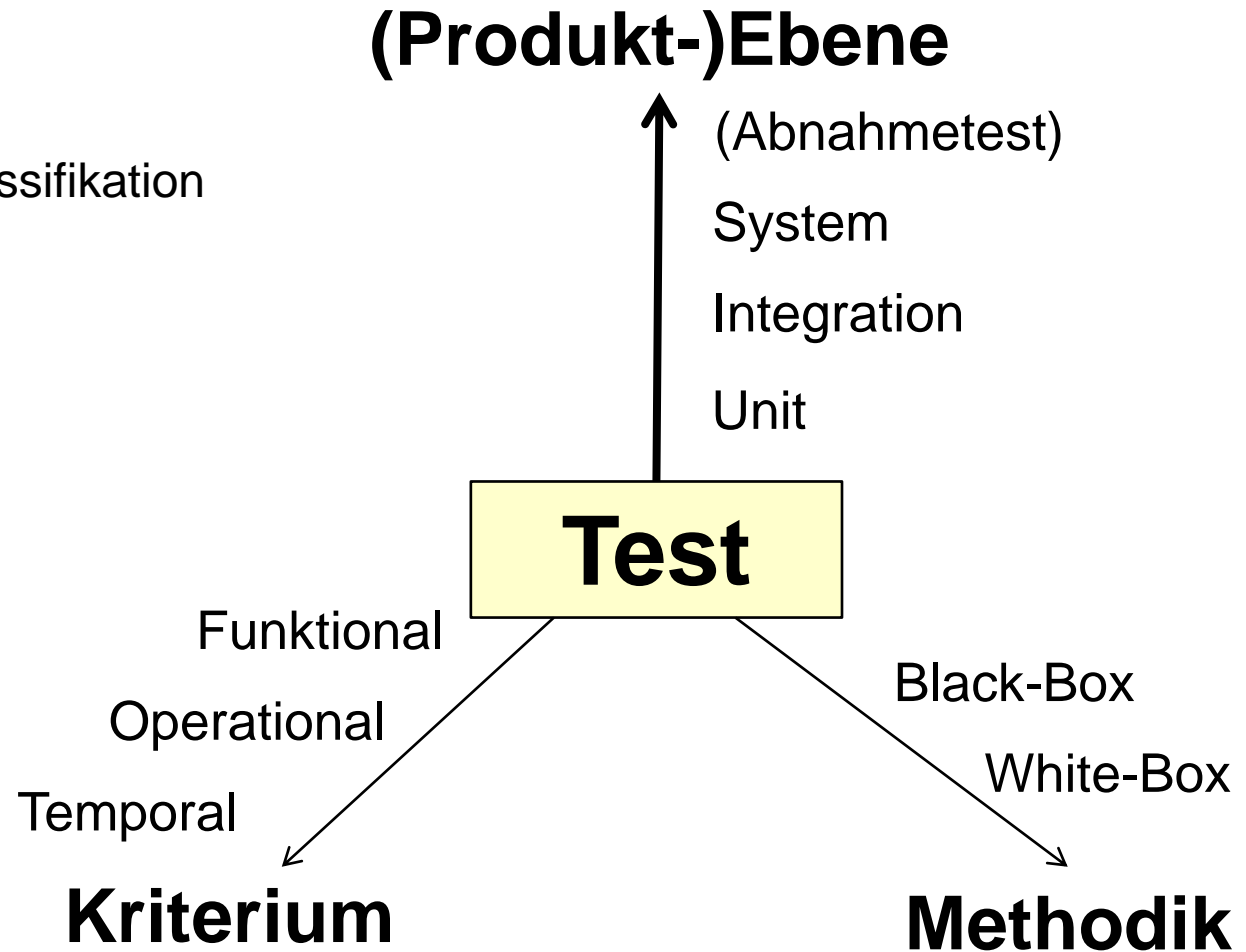
- **Black-Box**

- Testen eines Subjekts **ohne** Kenntnis über dessen inneren Aufbau
- Überprüfen der Ausgabewerte (Ist-Werte) gegen Sollwerte
- Sollwerte werden aus einer Spezifikation abgeleitet

- **White-Box (Glass-Box)**

- Testen eines Subjekts **mit** Kenntnis über dessen inneren Aufbau
- Idee: Sicherstellen, dass möglichst viele Programmteile durch Tests ausgeführt werden (**Coverage**) – da in Code, der nicht bei Tests ausgeführt wird, auch kein Fehler gefunden werden können
- Sollwerte werden weiterhin von einer Spezifikation abgeleitet
 - *Klar, wovon sonst!*

Eine mögliche Klassifikation





Testklassifikation: Testebene

- **Unit-Test**

- Eine **Unit** ist ein kleinster Teil eines Programms, welcher groß genug ist, um ihn zu testen
 - z.B. eine einzelne Funktion
 - oder auch ein oder mehrere Klassen
(die zum Teil als *Komponenten* oder *Module* bezeichnet werden)

- **Integrationstest**

- Testet Softwarekomponenten, die sich wiederum aus mehreren Komponenten zusammensetzen

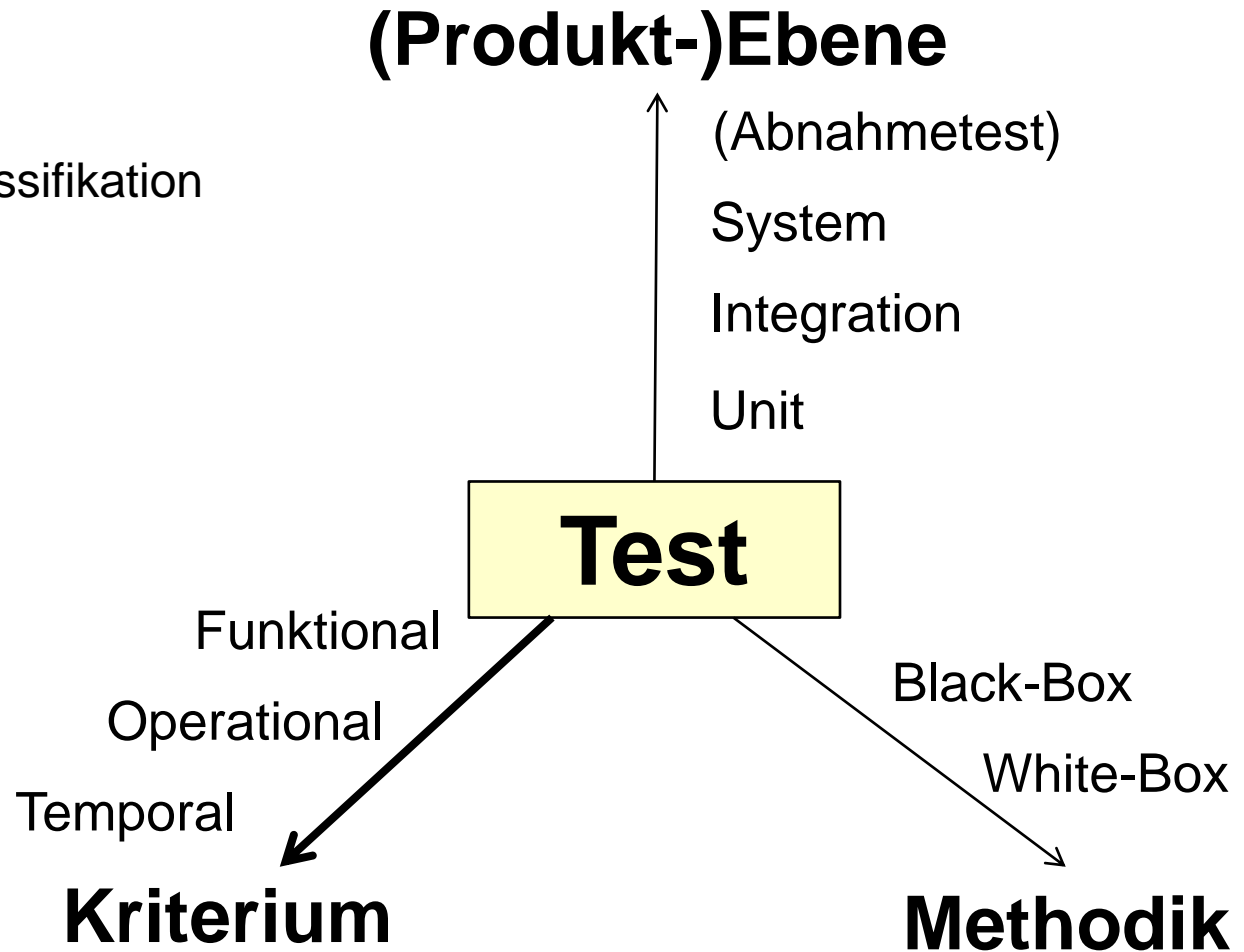
- **Systemtest**

- Testet das System als Ganzes

- **Abnahmetests**

- Auch auf Systemebene, werden meist vom Kunden und nicht Hersteller durchgeführt, um Vertragserfüllung zu überprüfen

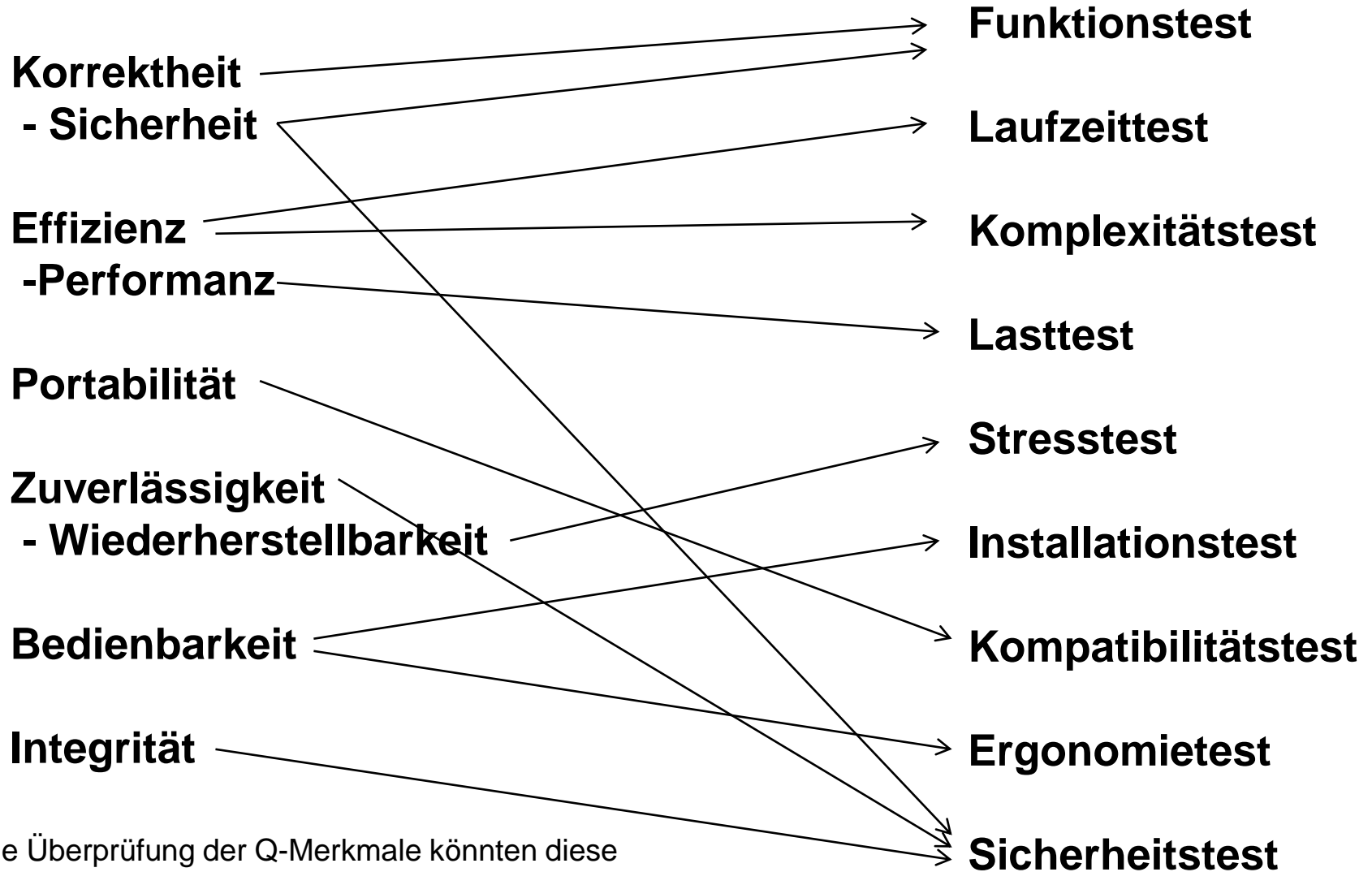
Eine mögliche Klassifikation



- **Funktionale Tests** (Funktionstests)
 - Überprüfen, ob ein Subjekt bei bestimmten Eingaben auch wie spezifiziert Ausgaben erzeugt (die meisten Tests sind solche)
- **Temporale Tests**
 - Laufzeittests: Testen von Berechnungsdauern mit Stoppuhr, (auch Testen von Speicherverbrauch z.B. durch Profiler)
 - Komplexitätstests: Ist Speicher-/Zeitverbrauch relativ zur Größe der Eingabe wie erwartet?
 - Lasttests: Testen Verhalten eines Systems bei vielen Anfragen
 - Stresstests: Testen mit zu vielen Anfragen; erholt sich das System?
- **Operationale Tests**
 - Sonstige Tests zum reibungslosen Betrieb des Systems
 - Z.B. Installations-, Kompatibilitäts-, Ergonomie-, Sicherheitstests



Q-Merkmale und verschiedene Arten von Tests



Für die Überprüfung der Q-Merkmale könnten diese Tests genutzt werden (wenn Q-Merkmale in testbare Eigenschaften konkretisiert wurden)



Hintergrund:

Fault, Error, Failure – Defekt, Error, Fehler

- 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 inkorrekt **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

- **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

- Gegeben ist eine Methode mit drei Integer-Parametern, welche die Kantenlängen eines Dreiecks beschreiben
 - Die Methode soll ausgeben, ob das Dreieck **gleichseitig**, **gleichschenkelig** 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){
    ...
}
```



Beispiel: Methode getTriangleKind

- Haben Sie...
 1. Einen Test mit einem korrekten gleichseitigem Dreieck?
 2. Einen Test mit einem korrekten gleichschenkligen Dreieck?
 3. Einen Test mit einem korrekten ungleichseitigem Dreieck?
 4. Drei gleichschenklige Dreiecke getestet, sodass alle Permutationen der gleichen Seiten abgedeckt sind (also z.B. 3-3-4, 3-4-3, 4-3-3)
 5. Einen Test mit einer Seite der Länge 0?
 6. Einen Test wo alle Seiten Länge 0 haben?
 7. Einen Test mit einer Seite mit negativer Länge?
 8. Einen Test mit einer Seite/allen Seiten Integer.MAX_VALUE/MIN_VALUE
 9. Einen Test wo eine Seite gleich der Summe der beiden anderen ist?
 10. Davon drei Permutationen?
 11. Einen Test wo eine Seite länger als die Summer der beiden anderen ist?
 12. Auch bei allen Tests die erwarteten SOLL-Werte angegeben?
(Was ist bei Fall 11?)

- Was macht die Methode `getTriangleKind`, wenn Eingabe kein Dreieck ergibt (s. Fall 11 auf vorheriger Folie)?
 - Der Fall wurde anscheinend bei der Spezifikation vergessen
 - Es sollte definiert werden, dass eine solche Eingabe ungültig ist
 - oder was in diesem Fall die Ausgabe ist.
- Oft wird erst bei Tests klar, dass die Spezifikation ungenau ist, unvollständig, oder sogar widersprüchlich ist
- Auch solche Fälle sollte ein Tester aufdecken!
- Ist die Spezifikation unvollständig oder widersprüchlich, muss mit dem Kunden eine Nachbesserung besprochen werden

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

- **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

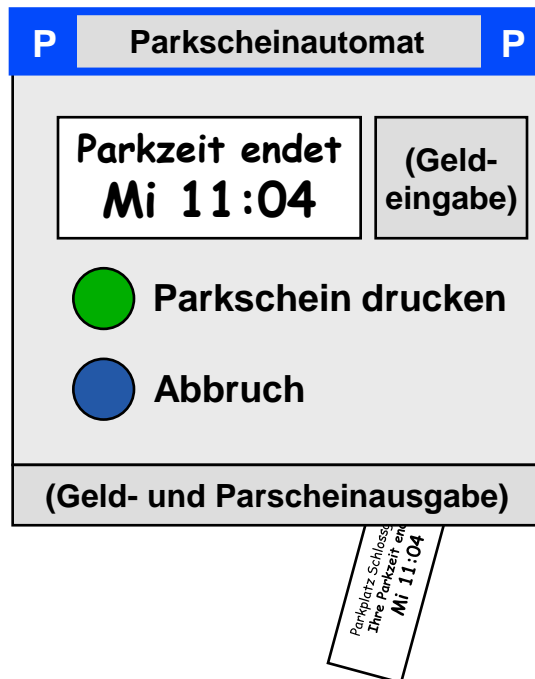
- **Wir können meist nicht jeden Fall testen!**
- Daher ist das Ziel möglichst viele verschiedene Fälle abzudecken, in denen der Fehler liegen könnte
- Dazu gibt es mehrere Methoden
 - ➔ **Alle Anforderungen abdecken**
 - Alle Fälle und Sonderfälle in den Anforderungen abdecken
 - **Äquivalenzklassenbildung**
 - 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

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



Parkplatz Schlossgasse
Ihre Parkzeit endet
Mi 11:04



Anforderungen genauer

- Welches Münzen nimmt der Automat?
 - 1€, 50c, 20c, 10c
- Wieviel kostet das Parken?
 - 1,20€ pro Stunde
- Was passiert bei Einwurf über 2,40€?
 - Geld verfällt, keine Rückgabe
- Wann ist der Automat in Betrieb?
 - 24h
- Wann muss man für das Parken zahlen?
 - Zwischen 9 und 19 Uhr, jeden Tag



Zu testenden Methode der Parkscheinautomatensoftware

- Folgende Methode soll getestet werden:
 - `java.util.Date` repräsentiert Zeit, bis auf die Millisekunde
 - `java.lang.IllegalArgumentException` ist eine typische Exception, die bei ungültigen Eingaben geworfen wird

```
/**
 * @param startDate start date and time based on which end time is calculated
 * @param centsPaid amount paid in Euro cents
 * (must be at least 50 and divisible by 10)
 * @return date and time parking time ends
 * @throws IllegalArgumentException if centsPaid < 50 and not divisible by 10
 */
public static Date getParkingTimeEnd(Date startDate, int centsPaid)
    throws IllegalArgumentException {
    ...
}
```

Anforderungen:

- R01** Höchstparkzeit: 2 Stunden,
Beträge über 240c verfallen
- R02** Exception wenn Zahlungsbetrag kleiner 50c
- R03** Exception wenn Zahlungsbetrag 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

```
/**
 * ...
 */
public static Date getParkingTimeEnd(Date startDate, int centsPaid)
    throws IllegalArgumentException {
    ...
}
```



Tests der Methode `getParkingTimeEnd`

- Ein Test (Testfall T01):
 - Eingabe:
 - Einwurf von 2,40€ (240 cents, volle Parkzeit)
 - Startzeit 12.4.2014, 10:00 Uhr
 - Sollresultat:
 - 12.4.2014, 12:00 Uhr

```
/**
 * @param startDate start date and time based on which end time is calculated
 * @param centsPaid amount paid in Euro cents
 * (must be at least 50 and divisible by 10)
 * @return date and time parking time ends
 * @throws IllegalArgumentException if centsPaid < 50 and not divisible by 10
 */
public static Date getParkingTimeEnd(Date startDate, int centsPaid)
    throws IllegalArgumentException {
    ...
}
```

- Mehrere Testfälle in einer Tabelle

ID	startDate	centsPaid	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! Mind. 50c
T04	2.1.-14:12	50	2.1.-14:37
T05	29.2.-17:46	67	Exception! Nicht Vielfaches von 10

Sind das gute Testfälle?

```
/**
 * @param startDate start date and time based on which end time is calculated
 * @param centsPaid amount paid in Euro cents
 * (must be at least 50 and divisible by 10)
 * @return date and time parking time ends
 * @throws IllegalArgumentException if centsPaid < 50 and not divisible by 10
 */
public static Date getParkingTimeEnd(Date startDate, int centsPaid)
    throws IllegalArgumentException {
    ...
}
```




Abdecken von Anforderungen

- 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

- 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	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			

- Beim Versuch einen Test für Anforderung R06 zu definieren, wird deutlich, dass Spezifikation wieder lückenhaft ist!
 - Anforderung R06 : Parken kostet nichts von 19:00 bis 9:00
 - Bedeutet das, dass das Geld verfällt oder für den nächsten Tag zählt?
 - Auch hier: *Kunde entscheidet u. Nachbessern der Spezifikation!*

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	X	
T02	3.4.-9:00	100	3.4.-9:50				X	X	
T03	28.11.-11:30	20	Exception!						
T04	2.1.-14:12	50	2.1.-14:37				X	X	
T05	29.2.-17:46	67	Exception!						
						X			



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...