Testen

- Motivation
- Verschiedene Test-Strategien, Blackbox-Tests
- Äquivalenzklassen
- Beispiel
- Randwerte
- Grenzen von BB-Tests
- Ausblick

Fehler vermeiden vs. Fehler finden

- Konstruktive Maßnahmen: Fehler vermeiden
  - Z.B.
    - → Schulung in eingesetzten Methoden
    - → Auswahl geeigneter Tools
    - → Anpassung der Prozesse
- Analytische Maßnahmen: Fehler finden
  - Z.B.
    - → Testen
    - → Reviews
    - → Codeanalysen

Definition Testen

- Testen ist die Vorführung eines Programms oder Systems mit dem Ziel zu zeigen, dass es tut, was es tun sollte

  Hetzel: The complete guide to software testing, 1984

- Testen ist der Prozess ein Programm auszuführen mit der Absicht, Fehler zu finden

  Myers: The art of software testing, 1979

- Testing. The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

  IEEE 610-1990 Standard Glossary of Software Engineering Terminology

Erschöpfendes Testen

- Wir testen eine Methode *m*, deren Ausführung von einem Parameter des Typs *boolean* abhängt

- Wie viele Ausführungsmöglichkeiten gibt es für diese Methode?
  - Überlegen Sie, wie die Methode intern aufgebaut sein könnte

- Wie viele *Testfälle* benötigt man also, um diese Methode erschöpfend („gründlich“) zu testen?
- Welche?

Erschöpfendes Testen

- Wir testen eine Methode *m3*, deren Ausführung von *drei* Parametern des Typs *boolean* abhängt

- Wie viele Ausführungsmöglichkeiten gibt es für diese Methode?
  - Überlegen Sie, wie die Methode intern aufgebaut sein könnte

- Wie viele *Testfälle* benötigt man also, um diese Methode erschöpfend („gründlich“) zu testen?
- Welche?

Erschöpfendes Testen

- Wir testen eine Methode *mi*, deren Ausführung von einem Parameter des Typs *int* abhängt

- Wie viele Ausführungsmöglichkeiten gibt es für diese Methode?

- Wie viele *Testfälle* benötigt man also, um diese Methode erschöpfend zu testen?
- Welche?

Erschöpfendes Testen

- Wir testen eine Methode *mi3*, deren Ausführung von *drei* Parametern des Typs *int* abhängt
- Wie viele Ausführungsmöglichkeiten gibt es für diese Methode?
- Wie viele *Testfälle* benötigt man also, um diese Methode erschöpfend zu testen?
- Welche?

Konsequenzen

- *Erschöpfendes* Testen ist nicht möglich!
- Wir wissen das (spätestens) jetzt
- Unsere Motivation, unsere Programme nach „Fertigstellung“ noch zu testen (wenn das sowieso nicht möglich ist), ist nicht besonders groß
- Entsprechend dürftig fallen oft die Tests aus (nur Alibi!)

Der Ausweg 1: Wir begrenzen die Komplexität

→ Wir wählen beim Testen immer eine *Stichprobe*
→ Wir testen *systematisch* (und ein bisschen *intuitiv*)

Der Ausweg 2a: Wir testen die Programme von *Anderen*

Der Ausweg 2b: Wir schreiben die Tests *vor dem Programmieren*

Konsequenzen

- *Erschöpfendes* Testen ist nicht möglich!
- Wir wissen das (spätestens) jetzt
- Unsere Motivation, unsere Programme nach „Fertigstellung“ noch zu testen (wenn das sowieso nicht möglich ist), ist nicht besonders groß
- Entsprechend dürftig fallen oft die Tests aus (nur Alibi!)

Der Ausweg 1: Wir begrenzen die Komplexität

→ Wir wählen beim Testen immer eine *Stichprobe*
→ Wir testen *systematisch* (und ein bisschen *intuitiv*)

Rest des Kapitels

Der Ausweg 2a: Wir testen die Programme von *Anderen*

Der Ausweg 2b: Wir schreiben die Tests *vor dem Programmieren*

Zum Ausweg 1: systematisch oder intuitiv testen?

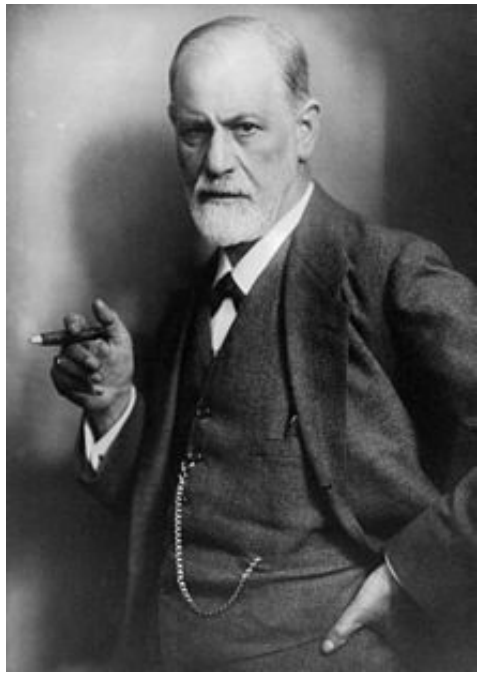
- Systematisch
  - Die Randbedingungen wurden definiert oder präzise erfasst
    - → Das sind sämtliche Gegebenheiten, die auf die Resultate Einfluss haben
      - Betriebssystem, Compiler, Middleware, ... incl. Versionsnummer
  - Die Eingaben (Stichproben!) wurden wiederholbar anhand einer einheitlichen Methode ausgewählt
  - Die Testergebnisse werden dokumentiert und nach Kriterien beurteilt, die vor dem Test festgelegt wurden
- Intuitiv
  - …was könnte nicht funktionieren?
  - Erfahrung hilft...!

Beides ist wichtig!

Zum Ausweg 2a: die psychologische Seite

Jede Suche geht von einer Annahme nach dem Gesuchten aus:

- Wenn wir ein Programm untersuchen in der Annahme, dass es funktioniert, werden wir vermutlich auch keinen Fehler finden.
- Wenn sich ein Fehler nicht „enttarnen“ lässt, so liegt es häufig daran, dass wir uns den Fehler nicht vorstellen konnten.



- Wenn wir aber zeigen wollen, dass ein Programm fehlerhaft ist, haben unsere Testdaten eine höhere Wahrscheinlichkeit, Fehler aufzudecken.

Testen von Software

Definition:
Testen ist der Prozess ein Programm auszuführen mit der *Absicht, Fehler zu finden*.

- Testen kann immer nur die *Anwesenheit* von Fehlern zeigen, *niemals* deren *Abwesenheit* beweisen!
- Beim Test wird die Qualität (die Verlässlichkeit, der Wert) eines Programms erhöht
- Die Qualität erhöhen bedeutet Fehler zu finden und zu beseitigen
- Daher sollte man nicht zeigen wollen, dass das Programm funktioniert sondern mit der Annahme beginnen, dass es Fehler enthält

  Diese Annahme gilt ohnehin für die meisten Programme...

Daher ist es effektiver, Programme anderer Entwickler zu testen!

Klarstellung

- Die **Fehlerursache** (error) liegt im Kopf des Programmierers; sie bildet die Grundlage für den Fehler.

- Der eigentliche **Fehler** (defect) liegt in den Daten oder im Programm und kann dort lokalisiert und behoben werden.

- Das **Fehlverhalten** (failure) wird beim Programmlauf sichtbar:
  - Zeigt ein Programm ein fehlerhaftes Verhalten (am Bildschirm oder in einer Datei,...) so kann auf einen *Fehler* geschlossen werden.
  - Daher ist es ein Unterschied, ob man ein *Fehlverhalten* beseitigt oder sich über die *Ursache* Gedanken macht und den Fehler beseitigt.

Beispiel 1/2

```
public static void main(String[] args) {

    double x = 2.5;
    if ( x * 2.24 == 5.6 )
        System.out.println("Hurra!");

}
```

Fehlerursache:
Der Entwickler hat nicht an die Rechen-Ungenauigkeiten von Gleitkommazahlen in Java gedacht

Beispiel 2/2

Fehlverhalten beseitigen

```
if ( x * 2.24 == 5.600000000000000005 )
    System.out.println("Hurra!");
```

Fehler beseitigen

```
if ( Math.abs(x * 2.24 - 5.6) < 0.000001 )
    System.out.println("Hurra!");
```

Verschiedene Test-Strategien

Ziel all dieser Strategien: die Wahl guter ***Stichproben***

- Blackbox-Tests
  - Äquivalenzklassenbildung (*Equivalence Partitioning*)
  - Randwertanalyse (*Boundary-Value Analysis*)
- Whitebox-Tests
  Man unterscheidet verschiedene ***Überdeckungsgrade***:
  - Statement Coverage
  - Decision Coverage, Branch Coverage
  - Condition Coverage
  - Decision / Condition Coverage
  - Multiple Condition Coverage
- Graybox-Tests
  - Mischung aus Black- und Whitebox-Tests
- Mutation-Tests
- ...

Blackbox-Tests: Allgemeines

- Andere Bezeichnung: funktionales Testen
- Der Tester sieht das Programm als **Black Box**, d. h.

$y1 = f(x1, x2)$

$y2 = g(x2, x3)$

  - die innere Struktur des Programms interessiert nicht,
  - *Testfälle* werden *ausschließlich aus der Spezifikation* abgeleitet
- Der Tester ist interessiert am Finden von Testfällen, für die das Programm nicht mit seiner Spezifikation übereinstimmt

-> Was gehört zu einem *Testfall*?

Definition Testfall

- A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

  IEEE 610-1990 Standard Glossary of Software Engineering Terminology

Zu einem **Testfall** gehören also

1. Der Anfangszustand, sofern vorhanden / relevant (z.B. Daten in einer Datei / Datenbank)
2. Die Eingabedaten (z.B. Parameter für eine Methode)
3. Die auszuführende Methode / Feature
4. Das *erwartete* Ergebnis (z.B. Funktionsergebnis)
5. Der *erwartete* Endzustand, sofern vorhanden / relevant (z.B. Ausgaben auf der Konsole, Daten in einer Datei / Datenbank)

Erschöpfendes Testen

- Wenn man versucht, mittels Blackbox-Tests alle Fehler zu finden, muss man **erschöpfend testen**
- Erschöpfendes Testen bedeutet,
  - *jede mögliche Eingabe* wird als Testfall vorgesehen und das Programm damit getestet;
  - auch muss jede dieser Eingaben mit *jedem möglichen Anfangszustand* kombiniert werden.
- Beispiel „Einzahlung auf ein Bankkonto“:
  - Es müsste *jeder mögliche Betrag* eingezahlt werden, um das Ergebnis zu überprüfen!
  - Das müsste man für *jeden möglichen Kontostand vor dem Einzahlen* (= Anfangszustand) tun!

→ Erschöpfendes Testen ist unmöglich
(außer für triviale Programme)

→ Man kann eigentlich nie aufhören, zu testen...

F R A G E N

Es gibt Möglichkeiten, die Effizienz von Blackbox-Tests deutlich zu verbessern

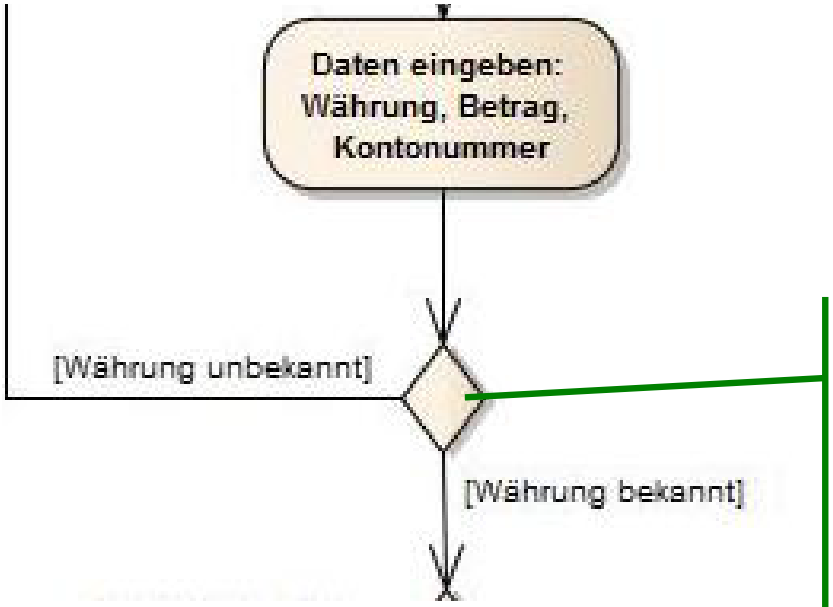
- Überlegung am Beispiel „Geld abheben“:
  Für welche Eingaben / Anfangszustände wird sich das Programm *vermutlich* gleich verhalten?

Wir können nicht ins Programm hineinschauen: ***Black Box***!

Eigentlich müssten ***jede*** bekannte und jede unbekannte Währung getestet werden!

Wir nehmen an:
Wird ***eine*** unbekannte Währung abgelehnt, dann klappt das auch mit ***allen anderen***

Es gibt Möglichkeiten, die Effizienz von Blackbox-Tests deutlich zu verbessern



Eigentlich müssten ***jede*** bekannte und jede unbekannte Währung getestet werden!

Wir nehmen an:
Wird ***eine*** unbekannte Währung abgelehnt, dann klappt das auch mit ***allen anderen***

- Konsequenz:
  Wenn diese Annahme stimmt, dann reicht es, das Programm mit (wenigen) repräsentativen Eingaben und Anfangszuständen zu testen (statt mit *allen* möglichen!)

→ Wir überlegen uns **Äquivalenzklassen** für die Eingaben / Anfangszustände und wählen *einen Repräsentanten / Stellvertreter pro Klasse*

(Rechtfertigung von) BB-Tests mit Äquivalenzklassen

- Wir *nehmen an*, dass sich das Programm für mehrere „ähnliche“ Eingaben gleich (*äquivalent*) verhält:

  Funktioniert es für *eine* der Eingaben korrekt, wird es *auch für die anderen* „ähnlichen“ korrekt funktionieren
  Beispiel: ungültige Währungen sollen erkannt werden
  Konsequenz: es reicht, mit *einer* ungültigen Währung zu testen

- Wir fassen Eingaben, für die das Programm sich gleich verhält, zu einer sogenannten **Äquivalenzklasse** zusammen

→ Jetzt benötigen wir für jede Äquivalenzklasse nur noch einen repräsentativen Testfall!

→ Wir müssen viel weniger testen, erreichen aber die gleiche Qualität wie vorher (*wenn unsere Annahme stimmt*)!

HOCHSCHULE DER WIRTSCHAFT FÜR MANAGEMENT
UNIVERSITY OF APPLIED MANAGEMENT STUDIES

Formale Zusammenfassung

- Es ist günstig, den Eingaberaum (Eingaben, Anfangszustände) beim Testen in **Äquivalenzklassen** aufzuteilen

- Aus diesen Überlegungen folgt eine Technik namens **Äquivalenzklassen-Bildung** oder **Testklassen-Bildung** (engl.: **Equivalence Partitioning**)

Äquivalenzklassen (1)

Vorgehen:

1. Wir überlegen uns, für welche Anfangszustände und Eingabedaten sich die zu testende Methode (vermutlich) einheitlich verhalten wird
2. Solche *gleichartigen Eingabedaten*
   - fassen wir zu je einer Klasse zusammen und
   - testen aus jeder Klasse nur einen Repräsentant

- Die Klasseneinteilung ist eine *Äquivalenzrelation*

Beispiel: Radarkontrolle für Tempo 50, Toleranzbereich 10%

- Mögliche Äquivalenzklassen
  - Erlaubter Bereich: 0 – 50 km/h
  - Toleranzbereich: 51 – 55 km/h
  - Geschwindigkeitsüberschreitung: ≥ 56 km/h

Äquivalenzklassen (2)

- Es gibt 2 Arten von Äquivalenzklassen
  - Gültige Äquivalenzklassen
    → alle gültigen Anfangszustände / Eingaben für eine Methode
  - Ungültige Äquivalenzklassen
    → alle anderen Eingaben (nicht: Anfangszustände)

Beispiel: Radarkontrolle für Tempo 50, Toleranzbereich 10%

- Es gibt 3 gültige Äquivalenzklassen
  - Erlaubtes Tempo: 1 bis 50 km/h
  - Toleranzbereich: 51 bis 55 km/h
  - Geschwindigkeitsüberschreitung: ab 56 km/h
- Es gibt 1 ungültige Äquivalenzklasse (könnte in 2 geteilt werden)
  - Keine / negative Geschwindigkeit: weniger als 0 km/h

Äquivalenzklassen (3)

- Wir haben Äquivalenzklassen für *Bereiche* betrachtet (z.B. Geschwindigkeit)
- Es gibt zwei weitere Möglichkeiten, wie Anfangs- oder Eingabedaten beschaffen sein können, erstens:

  Menge:
  Bei einer (begrenzten) Menge von Werten stellen jeder gültige Wert und jeder ungültige Wert eine eigene Äquivalenzklasse dar

  Es müssen *alle enthaltenen* Werte (d.h. gültigen Äquivalenzklassen) getestet werden sowie *ein nicht enthaltener* Wert (d.h. eine ungültige Äquivalenzklasse Beispiel Ampelfarben

  → Alle gültigen Elemente: rot, gelb, grün

  → Ein (beliebiges) ungültiges Element: blau

  In Java repräsentiert durch

  → Aufzählungstypen (enum; immer mindestens ein ungültiger Wert: null)

  → int-Konstanten (die in if-Anweisungen unterteilt werden)

  → String-Konstanten (die in if-Anweisungen unterteilt werden)

Äquivalenzklassen: Mengen in Java

- In Java können Mengen unterschiedlich repräsentiert werden
  - Aufzählungstypen (enum; immer mindestens ein ungültiger Wert: null)
    Beispiel: enum Ampelfarbe { ROT, ROT_GELB, GRÜN, GELB; }
    Ampelfarbe aktuelleAmpelfarbe = ROT;
  - int-Konstanten
    Beispiel: static final int AMPEL_ROT = 1;
    static final int AMPEL_GELB = 2;
    ...
    int aktuelleAmpelfarbe = AMPEL_ROT;
    leider: aktuelleAmpelfarbe = 42;
  - String-Konstanten
    Beispiel: static final String AMPEL_ROT = “ROT“;
    static final String AMPEL_GELB = “GELB“;
    ...
    String aktuelleAmpelfarbe = AMPEL_ROT;
    leider: aktuelleAmpelfarbe = “hello world!“;
- Entsprechend muss getestet werden!

Äquivalenzklassen (4)

- Wir haben Äquivalenzklassen für *Bereiche* (z.B. Geschwindigkeit) und für *Mengen* (z.B. Ampelfarben) betrachtet
- Es gibt noch eine weitere Möglichkeit, wie Anfangs- oder Eingabedaten beschaffen sein können:

  Muss-Situation:
  Eine Muss-Situation besitzt *eine gültige* und *eine ungültige* Äquivalenzklasse, d.h. sie ist mit einem gültigen sowie einem ungültigen Wert zu testen

  Beispiel aus der Java-Sprachbeschreibung:
  „Ein Bezeichner beginnt mit einem Buchstaben“
  (anders formuliert: „... muss mit einem Buchstaben beginnen“):

  → Ein gültiger Wert: „abc“

  → Ein ungültiger Wert: „3bc“

Äquivalenzklassen (5)

Achtung:

Wenn es einen Grund gibt anzunehmen, dass Elemente einer Äquivalenzklasse *nicht einheitlich* behandelt werden, *splitte* diese Äquivalenzklasse in zwei oder mehr Klassen auf!

Testfälle bilden

Nachdem alle Äquivalenzklassen identifiziert und aufgeschrieben sind, werden daraus wie folgt *Testfälle* abgeleitet:

1. Vergib eine eindeutige Nummer für jede Äquivalenzklasse
2. Bis alle *gültigen* Äquivalenzklassen abgedeckt sind, entwirf einen neuen *Testfall*, der *so viele wie möglich* von bisher nicht abgedeckten gültigen Klassen abdeckt
   (damit möglichst wenige Testfälle gebraucht werden)
3. Bis alle *ungültigen* Äquivalenzklassen abgedeckt sind, entwirf einen neuen Testfall, der *genau eine* der bisher nicht abgedeckten ungültigen Klassen abdeckt
   Beispiel, warum „genau eine“ ungültige Klasse pro Testfall:
   angenommen `geldAbheben(„unbekannte Währung“, 100, „unbekannte Kontonummer“)` liefert einen Fehler:
   - Welche Falscheingabe wurde entdeckt?
   - Würde die andere Falscheingabe ebenfalls entdeckt werden?

HOCHSCHULE DER WIRTSCHAFT FÜR MANAGEMENT
UNIVERSITY OF APPLIED MANAGEMENT STUDIES

Anwendungsbeispiel

Ein Kunde geht zur Bank und weist sich mit seinem Namen aus, um ein Konto zu eröffnen.

Er bekommt dann eine (neue) Kontonummer zugewiesen.

Für die Kontoeröffnung ist eine Mindesteinzahlung von € 50 verpflichtend.

Die Einzahlung kann in allen bekannten Währungen erfolgen.

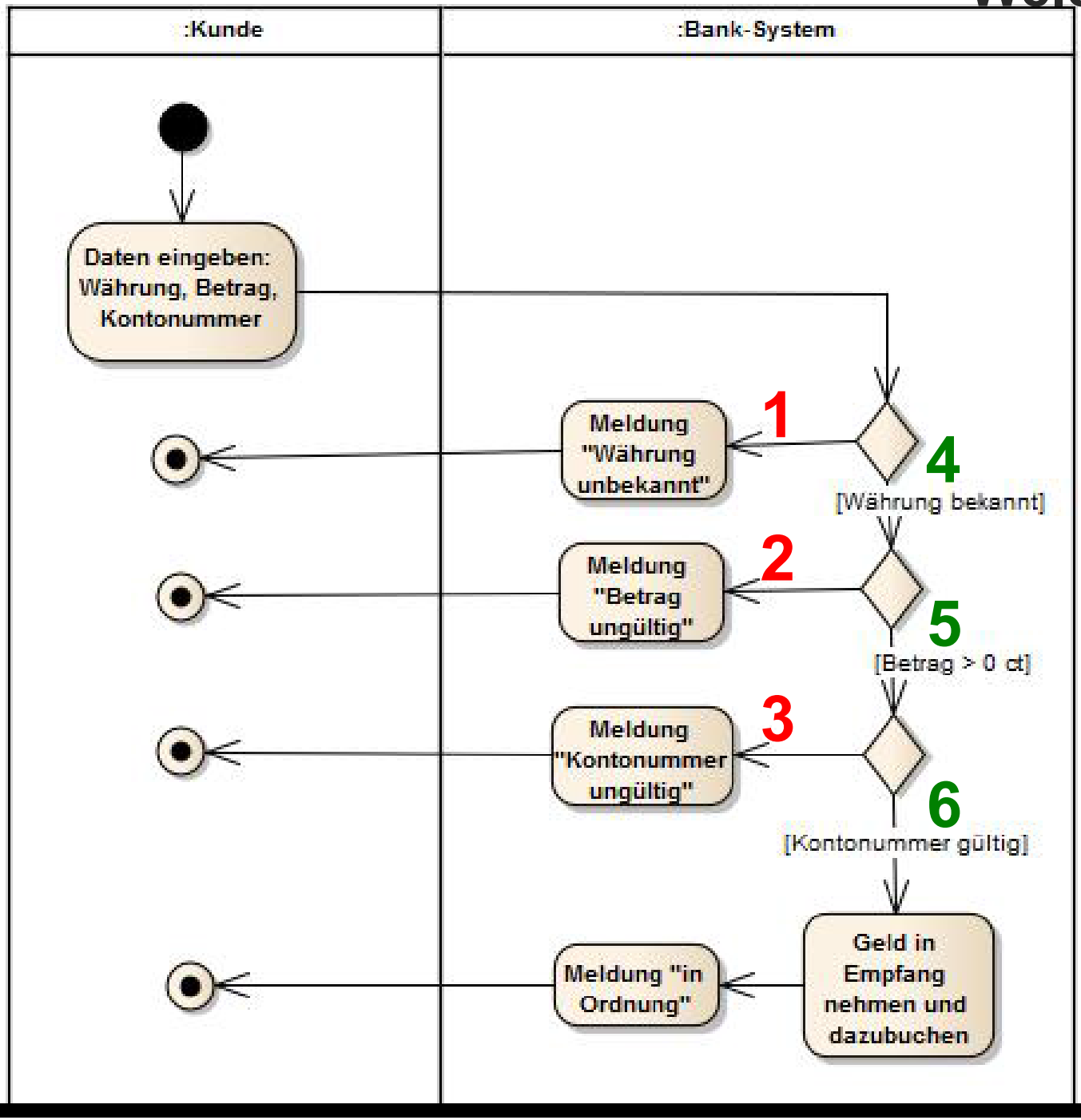
Partnerarbeit: Geld einzahlen

```
void zahleEin(String währung, double betrag, long kontonummer) {
```

Fragen

- Welche Äquivalenzklassen gibt es?
- Welche ***Werte*** kann man wählen, um möglichst viele Äquivalenzklassen zusammenzufassen?
- Erstellen Sie Testfälle: Wie viele benötigen Sie?

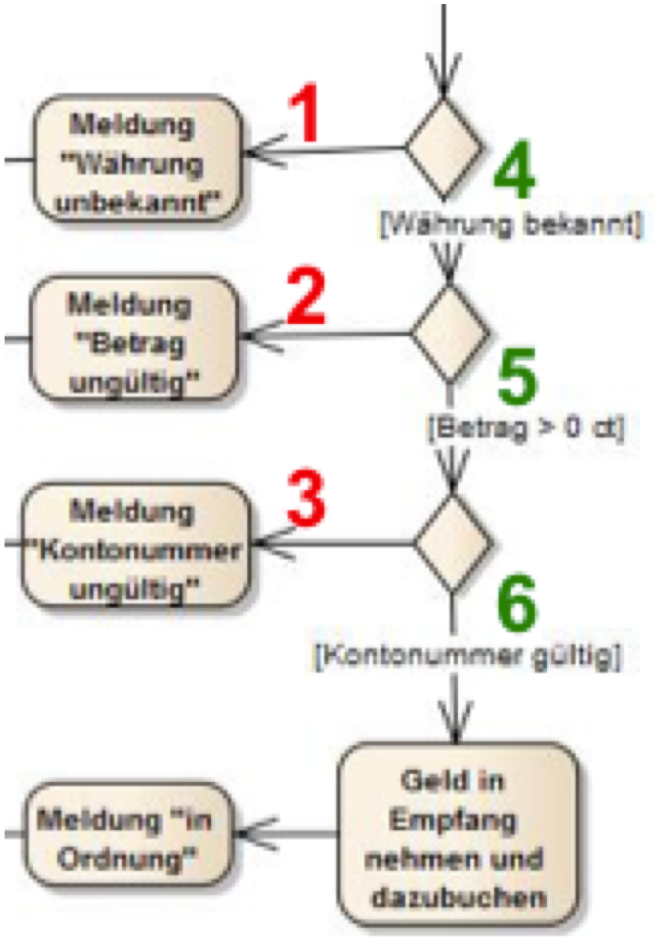
Beispiel „Geld einzahlen“: Welche Fälle müssen wir testen?



Mindestens(!) vier Fälle sind zu testen

1. Währung *unbekannt*, Betrag positiv und Kontonummer bekannt
2. Währung bekannt, Betrag *negativ* und Kontonummer bekannt
3. Währung bekannt, Betrag positiv und Kontonummer *unbekannt*

4.-6. Währung bekannt, Betrag positiv und Kontonummer bekannt



Mindestens(!) vier Fälle sind zu testen

1. Währung *unbekannt*, Betrag positiv und Kontonummer bekannt
2. Währung bekannt, Betrag *negativ* und Kontonummer bekannt
3. Währung bekannt, Betrag positiv und Kontonummer *unbekannt*

4.-6. Währung bekannt, Betrag positiv und Kontonummer bekannt

1. zahleEin("CAD", 0.01, 123456)
2. zahleEin("EUR", 0, 123456)
3. zahleEin("EUR", 0.01, 111111)
4. zahleEin("EUR", 0.01, 123456)
5. zahleEin(null, 0.01, 123456)
6. zahleEin("", 0.01, 123456)

Voraussetzung:
Konto 123456 existiert

Betrag vorher + 1ct == Betrag nachher

implementierungs- / sprachabhängige
ungültige Sonderfälle

F R A G E N

Randwertanalyse

Die **Boundary Value Analysis** (**Randwertanalyse**) unterscheidet sich in zwei Punkten von der Äquivalenzklassen-Bildung:

- Anstatt aus einer Äquivalenzklasse einen *beliebigen* Wert als Repräsentanten für einen Testfall zu wählen, werden ein oder mehrere Werte so gewählt, dass die Enden der Bereiche jeder Äquivalenzklasse überprüft werden.
- Anstatt sich nur auf den Eingaberaum zu konzentrieren, wird auch der *Ergebnisraum* bei der Bildung der Testfälle beachtet

Randwertanalyse

Die **Boundary Value Analysis** (**Randwertanalyse**) unterscheidet sich in zwei Punkten von der Äquivalenzklassen-Bildung:

- Anstatt aus einer Äquivalenzklasse einen *beliebigen* Wert als Repräsentanten für einen Testfall zu wählen, werden ein oder mehrere Werte so gewählt, dass die Enden der Bereiche jeder Äquivalenzklasse überprüft werden.

Beispiel: Radarkontrolle für Tempo 50, Toleranzbereich 10%

- Es gibt 3 gültige Äquivalenzklassen
  - Erlaubtes Tempo: 1 bis 50 km/h -> 1 und 50
  - Toleranzbereich: 51 bis 55 km/h -> 51 und 55
  - Geschwindigkeitsüberschreitung: ab 56 km/h -> 56
- Es gibt 1 ungültige Äquivalenzklasse (könnte geteilt werden)
  - Keine / negative Geschwindigkeit: weniger als 0 km/h -> -1 (und evtl. 0)

Randwertanalyse

Die **Boundary Value Analysis** (**Randwertanalyse**) unterscheidet sich in zwei Punkten von der Äquivalenzklassen-Bildung:

- Anstatt aus einer Äquivalenzklasse einen *beliebigen* Wert als Repräsentanten für einen Testfall zu wählen, werden ein oder mehrere Werte so gewählt, dass die Enden der Bereiche jeder Äquivalenzklasse überprüft werden.
- Anstatt sich nur auf den Eingaberaum zu konzentrieren, wird auch der *Ergebnisraum* bei der Bildung der Testfälle beachtet

Randwertanalyse

Die **Boundary Value Analysis** (**Randwertanalyse**) unterscheidet sich in zwei Punkten von der Äquivalenzklassen-Bildung:

- Anstatt sich nur auf den Eingaberaum zu konzentrieren, wird auch der *Ergebnisraum* bei der Bildung der Testfälle beachtet

Beispiel Sinusfunktion

- Das Ergebnis einer Sinusfunktion sollte +1.0 und -1.0 erreichen

-> zusätzliche Testwerte: $^1/_2\ \pi$ und $^3/_2\ \pi$
bzw. 90° und 270°

Anleitung für das Bilden von Testfällen (1)

1. Für jede Eingabe und den Ausgangszustand Äquivalenzklassen (ÄK) bilden: gültige, ungültige
   a) Integer-Bereiche (gültige oder ungültige)
   b) Mengen (Aufzählungen, Boolean, ...):
      → jeder gültige Wert ist eine ÄK
      → alle ungültigen Werte bilden eine ÄK
   c) Muss-Situation: eine gültige, eine ungültige ÄK
   d) An Besonderheiten der Sprache denken (z.B. null für Strings)
2. Repräsentanten für jede ÄK bestimmen
   a) Integer-Bereiche: mit Randwertanalyse bestimmen
      Hinweis: kann mehr als ein Repräsentant sein
   b) Mengen:
      → der einzige Wert einer gültigen ÄK ist ihr Repräsentant
      → ein beliebiger Wert einer ungültigen ÄK ist ihr Repräsentant
   c) Muss-Situation: ein Repräsentant jeder ÄK
   d) Besondere Werte

Anleitung für das Bilden von Testfällen (2)

3. Testfälle bilden
   - „gültige“ Testfälle
     - → Für jede Eingabe einen gültigen Repräsentanten wählen
     - → Jeden gültigen Repräsentanten mindestens einmal verwenden
     - → So viele gültige Repräsentanten wie möglich miteinander kombinieren, um weniger Testfälle zu brauchen
   - „ungültige“ Testfälle
     - → Für genau eine Eingabe einen ungültigen Repräsentanten wählen, für alle anderen Eingaben einen gültigen
     - → Jeden ungültigen Repräsentanten mindestens einmal verwenden

Methode „Geld einzahlen“

lle Geldbeträge werden für den Kontostand nach EUR-ct umgerechnet.	
ie Methode liefert kein Ergebnis (Typ void).	
m Fehlerfall wird eine entsprechende Meldung ausgegeben.	
	Ausgangszustand
1	Parameter
2	
3	
4	
5	
6	
7	
8	
	Meldung
	Endzustand

Was müssen wir für unsere Tests berücksichtigen?

Was gehört zu einem Testfall?

Methode „Geld einzahlen“

lle Geldbeträge werden für den Kontostand nach EUR-ct umgerechnet.
ie Methode liefert kein Ergebnis (Typ void).
m Fehlerfall wird eine entsprechende Meldung ausgegeben.

			Äquivalenzklassen
	Ausgangszustand	Kontonummer	
		Guthaben	
1	Parameter	Währung	
2			
3			
4			
5		Betrag	
6			
7		Kontonummer	
8			
	Meldung		
	Endzustand	Kontonummer	
		Guthaben	

Welche Äquivalenzklassen gibt es?

Methode „Geld einzahlen“

lle Geldbeträge werden für den Kontostand nach EUR-ct umgerechnet.
ie Methode liefert kein Ergebnis (Typ void).
m Fehlerfall wird eine entsprechende Meldung ausgegeben.

			Äquivalenzklassen		Randwerte, krit. Werte
	Ausgangszustand	Kontonummer	int	g	
		Guthaben	0	g	
1	Parameter	Währung	existiert	g	
2				g	
3			existiert nicht	u	
				u	
4				u	
5		Betrag	> 0	g	
6			<= 0	u	
7		Kontonummer	existiert	g	
8			existiert nicht	u	
	Meldung				
	Endzustand	Kontonummer			
		Guthaben			

Welche Rand***werte*** haben unsere Äquivalenzklassen?

Methode „Geld einzahlen“

lle Geldbeträge werden für den Kontostand nach EUR-ct umgerechnet.
ie Methode liefert kein Ergebnis (Typ void).
m Fehlerfall wird eine entsprechende Meldung ausgegeben.

			Äquivalenzklassen		Randwerte, krit. Werte
	Ausgangszustand	Kontonummer	int	g	123456
		Guthaben	0	g	0
1	Parameter	Währung	existiert	g	"EUR"
2				g	"USD"
3			existiert nicht	u	"CAD"
				u	""
4				u	null
5		Betrag	> 0	g	0.01
6			<= 0	u	0
7		Kontonummer	existiert	g	123456
8			existiert nicht	u	111111
	Meldung				
	Endzustand	Kontonummer			
		Guthaben			

Welche dieser Werte kombinieren wir jeweils zu Testfällen?

Wie viele Testfälle benötigen wir (mindestens)?

Methode „Geld einzahlen“

lle Geldbeträge werden für den Kontostand nach EUR-ct umgerechnet.
ie Methode liefert kein Ergebnis (Typ void).
m Fehlerfall wird eine entsprechende Meldung ausgegeben.

			Äquivalenzklassen		Randwerte, krit. Werte	TF 1, gültig	TF 2, gültig	TF 3, ungültig	TF 4, ungültig	TF 5, ungültig	TF 6, ungültig	TF 7, ungültig
	Ausgangszustand	Kontonummer	int	g	123456							
		Guthaben	0	g	0							
1	Parameter	Währung	existiert	g	"EUR"	"EUR"					"EUR"	"EUR"
2				g	"USD"		"USD"					
3			existiert nicht	u	"CAD"			"CAD"				
				u	""				""			
4				u	null					null		
5		Betrag	> 0	g	0.01	0.01	0.01	0.01	0.01	0.01		0.01
6			<= 0	u	0						0	
7		Kontonummer	existiert	g	123456	123456	123456	123456	123456	123456	123456	
8			existiert nicht	u	111111							111111
	Meldung					in Ordnung	in Ordnung	Währung unbekannt	Währung unbekannt	Währung unbekannt	Betrag ungültig	Kontonumme ungültig
	Endzustand	Kontonummer				123456	123456	123456	123456	123456	123456	123456
		Guthaben				0.01	0,00809	0	0	0	0	0

Mit 7 Testfällen ist diese Methode *umfassend* getestet!

F R A G E N

Grenzen von BB-Tests 1/2

- Ein Sortierprogramm verwendet ShakerSort für Zahlenfolgen mit bis zu 15 Elementen, bei mehr als 15 Elementen wird QuickSort verwendet
  Die “interne Grenze” 15/16 hat sich der Entwickler ausgedacht (vielleicht sogar gemessen). Sie tritt aber in der Spezifikation nicht auf, ist also für den Blackbox-Tester völlig unvorhersehbar

```
void sort ( Object [ ] sortArray ) {
      if ( sortArray.length <=15 )
            shakerSort( sortArray );
      else
            quickSort( sortArray );
}
```

- Die Chance, diesen (nicht böswilligen!) "Fehler" mittels Blackbox-Tests zu entdecken, ist gleich Null
- Folge:
  Vermutlich wird die sort-Methode nie mit mehr als 15 Objekten getestet und der QuickSort-Algorithmus bleibt völlig ungetestet

Grenzen von BB-Tests 2/2

Der Code in einem Programm für eine Bank könnte wie folgt aussehen:

```
// Einlesen einer Kontonummer für eine beliebige Transaktion
int nummer = ...;

// Bestimmen des Kontos
Konto konto = table.getKonto(nummer);

// korrekte Bearbeitung
...

// zusätzlicher Code
if ( konto.getBesitzer().equals("Knauber") )
        konto.betrag += 1.0;          // Kontostand um 1 Euro erhöhen
...
```

- Diesen (böswilligen) Fehler würde man mittels Blackbox-Test *nicht* entdecken
- Folge:
  Das Bankprogramm wird als "korrekt" freigegeben
  (und ich werde reich...)