LEHRSTUHL 14
SOFTWARE ENGINEERING

Vorlesung (WS 2014/15)
Softwarekonstruktion

Prof. Dr. Jan Jürjens

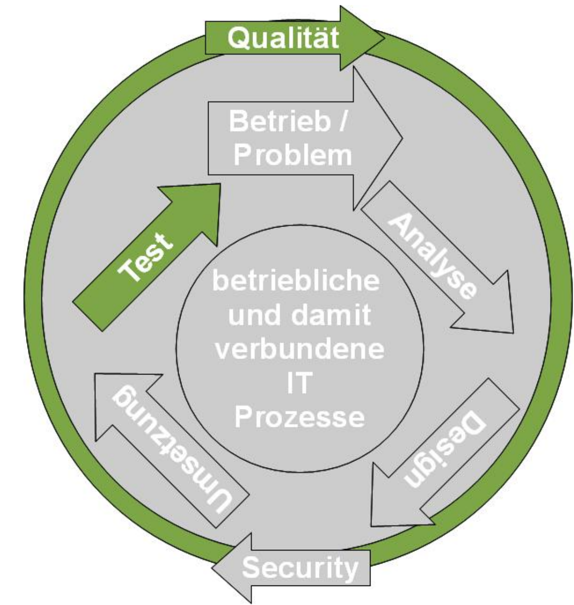
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

Teil 2.3: Black-Box-Test

v. 26.12.2014

Einordnung
Black-Box-Test

- Modellgetriebene SW-Entwicklung
- Qualitätsmanagement
- Testen
  - Grundlagen Softwareverifikation
  - Softwaremetriken
  - Black-Box-Test
  - White-Box-Test
  - Testen im Softwarelebenszyklus



[Basierend auf „Basiswissen Softwaretest - Certified Tester“ des „German Testing Board“ (nach Certified Tester Foundation Level Syllabus, deutschsprachige Ausgabe, Version 2011)]

Literatur (s. Vorlesungswebseite):

- Andreas Spillner, Tilo Linz: Basiswissen Softwaretest.
  - **Kapitel 5.**
- Eike Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme.

Einleitung Black Box Test

- **Vorheriger Abschnitt:** Berechnung der Komplexität des Codes → Softwaremetriken
- **Dieser Abschnitt:** Black Box Testen
  - Äquivalenzklassenbildung
  - Zustandsbasiertes Testen
  - Entscheidungstabellenbasiertes Testen

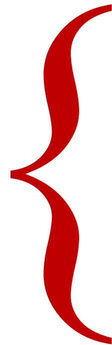
Was Sie wiederholen sollten!

Was Sie sich für diesen Abschnitt aus SWT noch einmal anschauen sollten:

- Unterschiede von **Black-Box-Tests** und **White-Box-Tests**
- Bildung von **Äquivalenzklassen**

2.3 Black-Box-Test

2.3 Black-Box-Test

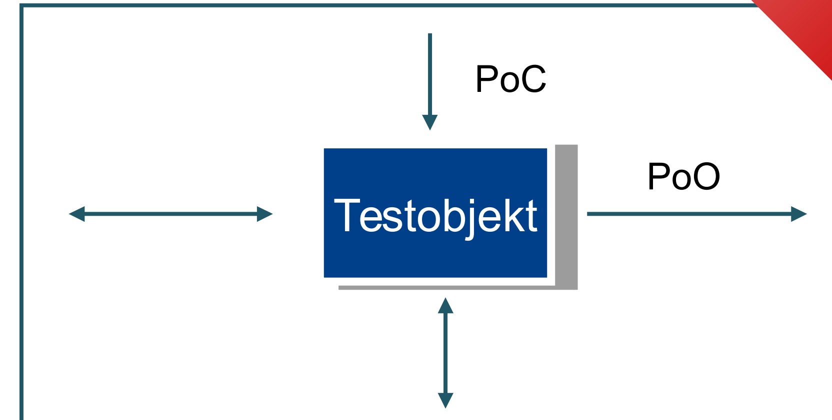


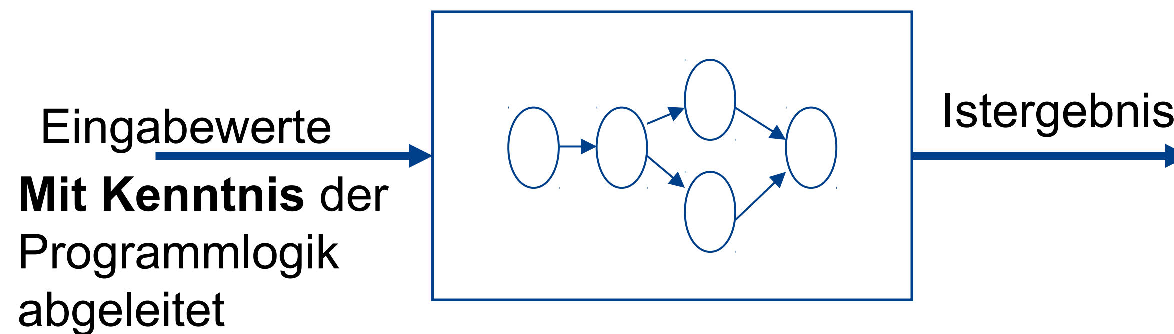
- **Idee der Black-Box-Testentwurfsverfahren**
- Äquivalenzklassenbildung
- Zustandsbasierter Test
- Entscheidungstabellentest

Black-Box Test vs. White-Box Test

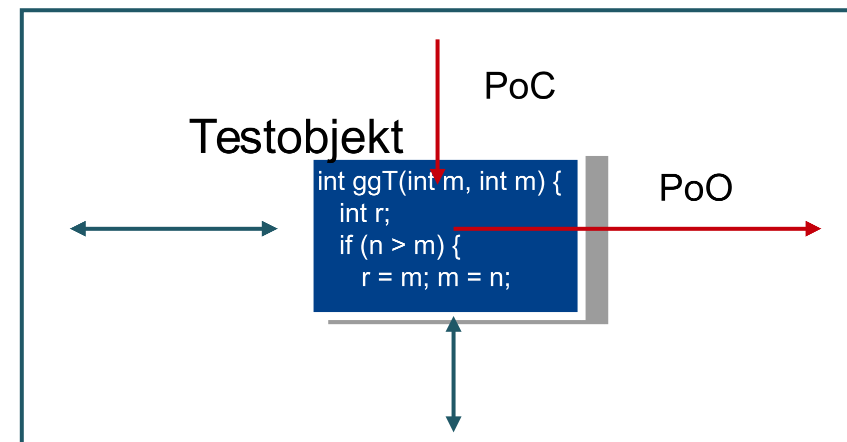
Wiederholung SWT

Ohne Kenntnis der Programmlogik abgeleitet

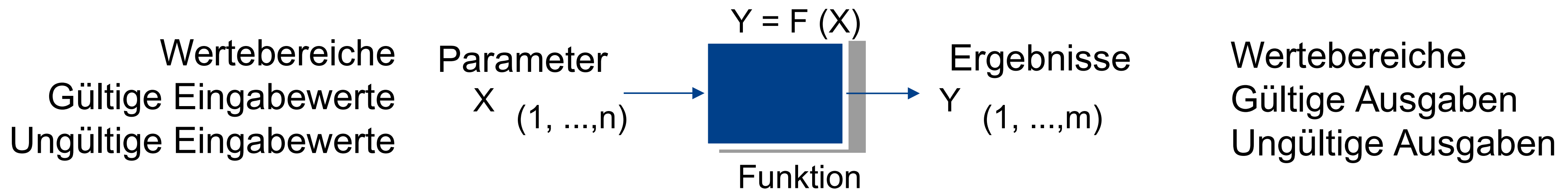




Mit Kenntnis der Programmlogik abgeleitet



Spezifikationsorientierte Testfall- und Testdatenermittlung



Äquivalenzklassenbildung

- Repräsentative Eingaben
- Gültige Dateneingaben
- Ungültige Dateneingaben
- Erreichen der gültigen Ausgaben

Grenzwertanalyse

- Wertebereiche
- Wertebereichsgrenzen

Zustandsbasierter Test

- Komplexe (innere) Zustände und Zustandsübergänge

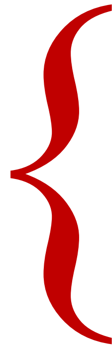
Entscheidungstabellentest

- Bedingungen und Aktionen

Anwendungsfallbasierter Test

- Szenarien der Systemnutzung

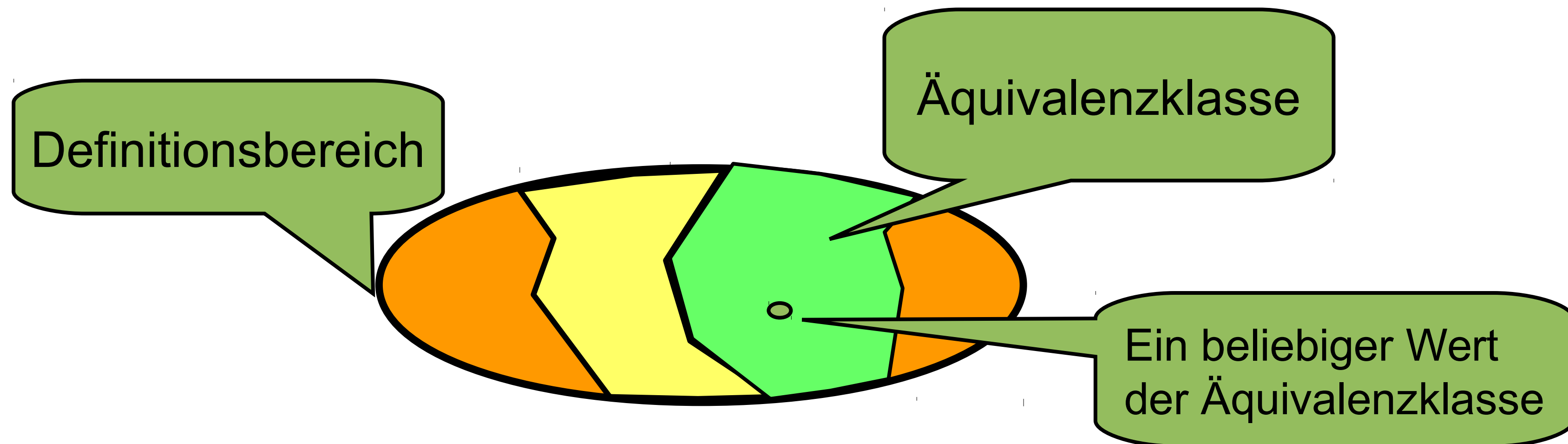
2.3 Black-Box-Test



- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- Zustandsbasierter Test
- Entscheidungstabellentest

Grundidee der Äquivalenzbildung

- Zerlegung der Definitionsbereiche der Ein- und Ausgaben in **Äquivalenzklassen (ÄK):** Werte einer Klasse = Äquivalentes Verhalten des Prüflings.
- Wahl Testwertes pro ÄK: **Sinnvolle Stichprobe**.
- Wenn Wert der ÄK **Fehler**
  - **aufdeckt.** → Alle Werte der ÄK sollen diesen Fehler aufdecken.
  - **nicht aufdeckt.** → Kein Wert der ÄK soll einen Fehler aufdecken.



Testfälle für jeden Parameter tabellarisch notieren

Eindeutige Kennzeichnung jeder Äquivalenzklasse (gÄKn, uÄKn):

	TF1	TF2	...	TFn
gÄK1	x			
gÄK2		x		
...			x	
uÄK1				
uÄK2				x
...				

Pro **Parameter** mindestens **zwei Äquivalenzklassen**

- Eine mit gültigen Werten
- Eine mit ungültigen Werten

Bei ***n*** **Parametern** mit $\boldsymbol{m_i}$ **Äquivalenzklassen** (i=1..n) gibt es:

$\Pi_{i=1...n}\, m_i$ unterschiedliche Kombinationen (Testfälle)

Äquivalenzklassenbildung: Fehlermaskierung

Gleichzeitige Behandlung verschiedener ungültiger ÄK:

Bestimmte Fehler evtl. unentdeckt !

Beispiel:

Eingabebereich

```
1 <= wert <= 99; farbe IN (rot, gruen, gelb)
```

Äquivalenzklassen

```
wert_gÄK1:  1 <= wert <= 99
wert_uÄK1:  wert < 1
wert_uÄK2:  wert > 99
farbe_gÄK1: farbe IN (rot, gruen, gelb)
farbe_uÄK1: NOT farbe IN (rot, gruen, gelb)
```

Testdaten

`wert_uÄK1` und `farbe_uÄK1`: z.B. `wert=0, farbe=schwarz`

→ Fehlerhafte Behandlung von `farbe=schwarz` bei `wert_gÄK1` ggf. unentdeckt (und umgekehrt).

Äquivalenzklassenbildung
Ausgangskriterium (Testendekriterium)

- **Spezifisches Ausgangskriterium** festlegen:
  Nach Äquivalenzklassenbildung anhand durchgeführte Tests der Repräsentanten der jeweiligen Äquivalenzklassen im Verhältnis zur Gesamtzahl aller definierten Äquivalenzklassen:

ÄK-Überdeckungsgrad = (Anzahl getestete ÄK / Gesamtzahl ÄK)

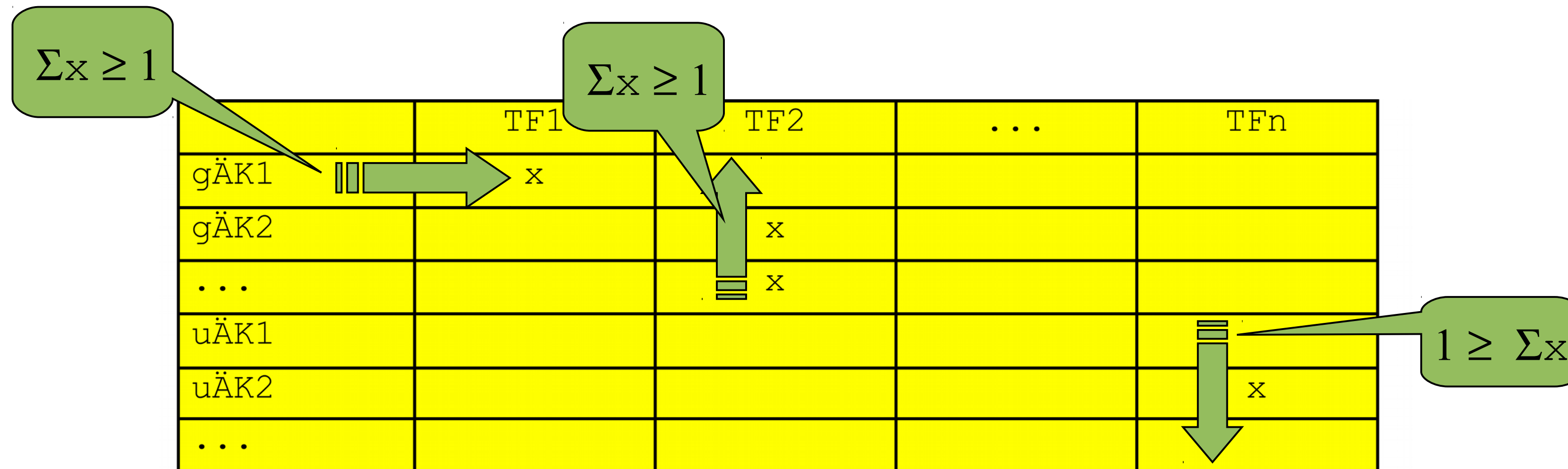
- **Beispiel:** Ermittlung von 18 Äquivalenzklassen aus Anforderungen für ein Eingabedatum und Testen von 15 von 18 Testfällen.
  → Erreichen von ca. 83 % **Äquivalenzklassen-Überdeckung**:
- ÄK-Überdeckung = 15 / 18 ≈ 83 %

Auswahl der Testfälle
Minimalkriterium

Alle ÄK‘s durch mindestens einen Testfall abdecken

Dabei pro Testfall:

- **Mehrere gültige Äquivalenzklassen** - für verschiedene Beschränkungen - abdecken, **oder**
- **Genau eine ungültige Äquivalenzklasse**
  → Einzelne Prüfung notwendig wegen Fehlermaskierung !



Äquivalenzklassen Beispiel: Testfälle und Testdaten für ggT


```
public int ggT(int m, int n)
```

Äquivalenzklassen für Eingabeparameter n, m *(analog)*: `int`

- gÄKx_1 : min_int ≤ n < 0
- gÄKx_2 : n = 0
- gÄKx_3 : 0 < n ≤ max_int
- uÄKx_1 : n < min_int
- uÄKx_2 : n > max_int

Testfälle:

TF1 : {n = -1, m = -1; ggT = 1}
TF2 : {n = 0, m = 0; ggT = 0}
TF3 : {n = 1, m = 1; ggT = 1}
TF4 : {n = min_int-1, m = -1; error}
TF5 : {n = max_int+1, m = -1; error}
TF6 : {n = -1, m = min_int-1; error}
TF7 : {n = -1, m = max_int+1; error}

	TF1	TF2	TF3	TF4	TF5	TF6	TF7
gÄK1_1	x					x	x
gÄK1_2		x					
gÄK1_3			x				
uÄK1_1				x			
UÄK1_2					x		
gÄK2_1	x			x	x		
gÄK2_2		x					
gÄK2_3			x				
uÄK2_1						x	
UÄK2_2							x

Äquivalenzklassenbildung
Vor- und Nachteile

Vorteile:

- Anzahl Testfälle kleiner als bei unsystematischer Fehlersuche.
- Geeignet für Programme mit vielen verschiedenen Ein- und Ausgabebedingungen.

Nachteile:

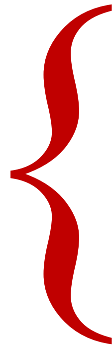
- Betrachtet Bedingungen für einzelne Ein- oder Ausgabeparameter.
- Beachtung von Wechselwirkungen und Abhängigkeiten von Bedingungen sehr aufwändig.

Empfehlung:

- Zur Auswahl wirkungsvoller Testdaten: Kombination der ÄK-Bildung mit fehlerorientierten Verfahren, z.B. Grenzwertanalyse.

2.3 Black-Box-Test

2.3 Black-Box-Test



- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- **Zustandsbasierter Test**
- Entscheidungstabellentest

Zustandsbasierter Test

Bei vielen Systemen: **Einfluss** des bisherigen Ablaufs des Systems auf Berechnung der Ausgaben.

- Endlicher Automat besteht aus endlicher Anzahl von internen Konfigurationen – **Zustände**.
- Zustand eines Systems beinhaltet implizit **Informationen**.
  - → Ergibt sich aus bisherigen Eingaben.
  - → Nötig um Reaktion des Systems auf folgende Eingaben zu bestimmen.

System: Annahme von unterschiedlichen Zuständen beginnend vom Startzustand.

H. Balzert: Lehrbuch der Softwaretechnik, Bd. I, Spektrum, 2002

- **Auslösung von Zustandsänderungen** oder –übergänge durch Ereignisse, z.B. Funktionsaufrufe.
- Bei Zustandsänderungen Aktionen durchführbar.
- **Spezieller Zustand:** Startzustand und Endzustand.

Beispiel zur Zustandsmodellierung: Stapel (Stack)

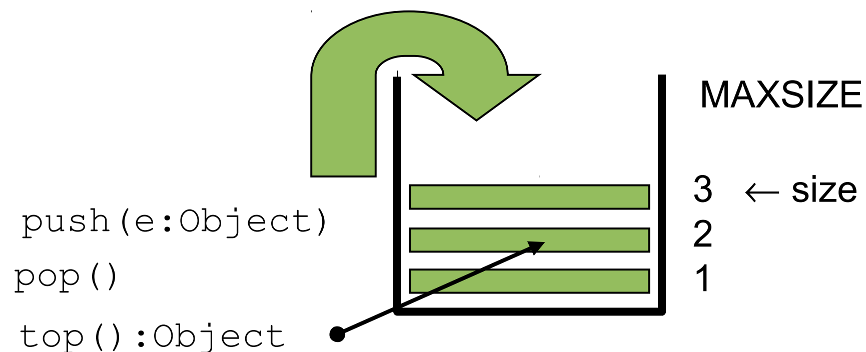
Klasse `Stapel`

Zustandserhaltende Operationen

```
size():integer; // Anzahl gestapelter Elemente
MAX():integer;  // Maximale Anzahl
top():Object;   // Zeiger auf oberstes Element
```

Zustandsverändernde Operationen

```
Stapel(Max:integer);          // Konstruktor
~Stapel();                    // Destruktor
push(element:Object);         // Stapelt Element
pop();              // Entfernt oberstes Element
```



Drei Zustände:

```
empty:   size() = 0;
filled: 0 < size() < MAX();
full:    size() = MAX();
```

Ziele des zustandsbasierten Tests

- Nachweis der Konformität des Testobjekts zum Zustandsdiagramm (**Zustands-Konformanztest**).
- Zusätzlich Test unter nicht-konformanten Benutzungen (**Zustands-Robustheitstest**).

Arbeitsschritte des zustandsbasierten Tests

LEHRSTUHL 14
SOFTWARE ENGINEERING

1. Erstellung des **Zustandsdiagrammes**
2. Prüfung auf **Vollständigkeit**
3. Ableiten des **Übergangsbaumes** für den **Zustands-Konformanztest**
4. Erweitern des Übergangsbaumes für den **Zustands-Robustheitstest**
5. Generieren der **Botschaftssequenzen** und Ergänzen der Botschaftsparameter
6. **Ausführen der Tests** und **Überdeckungsmessung**

1. Erstellung des Zustandsdiagrammes

Drei Zustände:

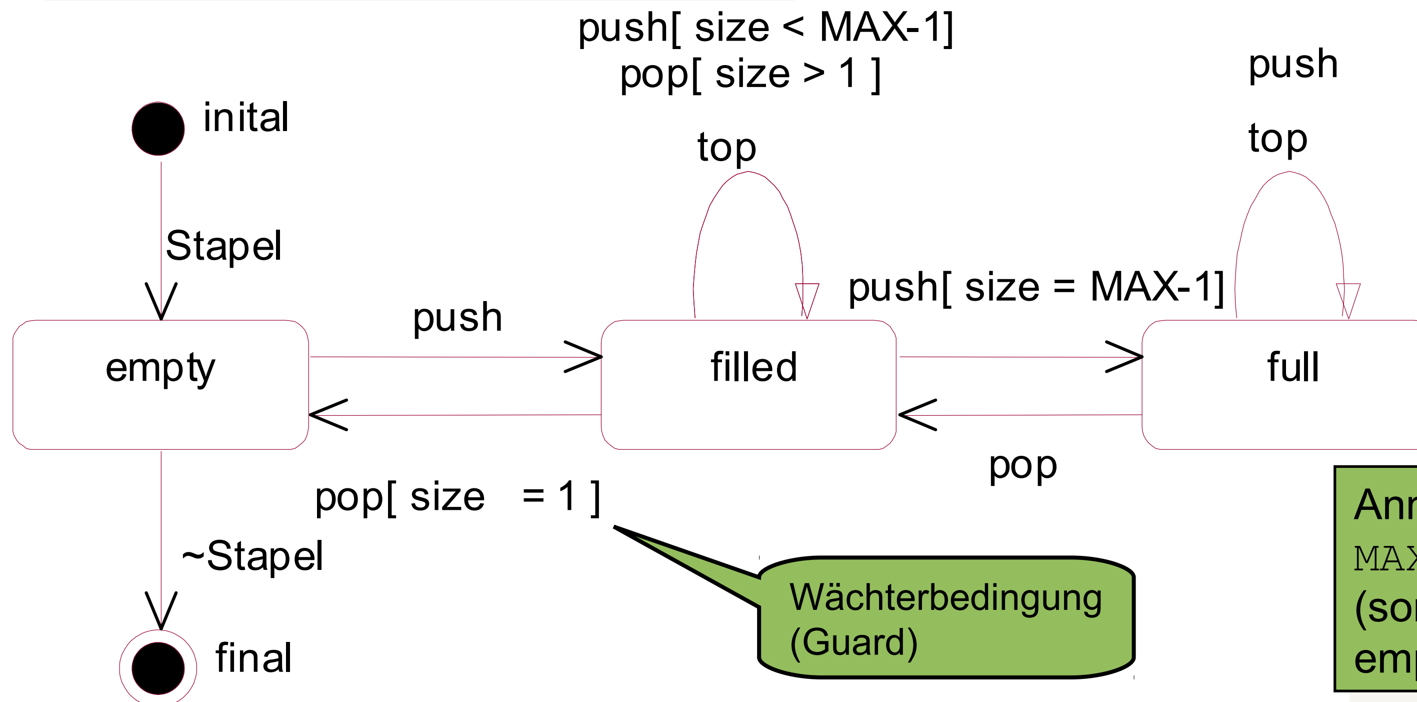
```
empty:  size() = 0;
filled: 0 < size() < MAX();
full:   size() = MAX();
```

Zwei »Pseudo-Zustände«:

```
Initial: Vor Erzeugung;
final:   Nach Zerstörung
```

Acht Zustandsübergänge:

```
initial → empty; empty → final
empty → filled; filled → empty (Zyklus!)
filled → full; full → filled   (Zyklus!)
filled → filled; full → full   (Zyklen!)
```



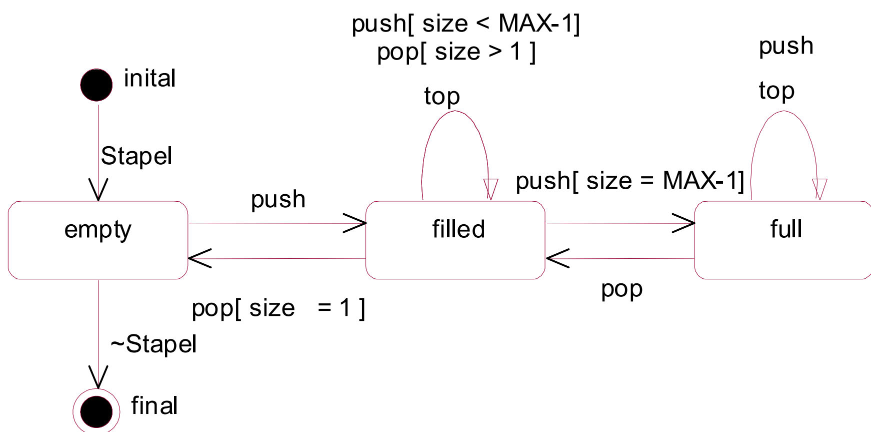
Annahme zur Vereinfachung:
`MAX()>1`
(sonst noch Transition „push“ von empty nach full benötigt)

2. Prüfung auf Vollständigkeit

LEHRSTUHL 14
SOFTWARE ENGINEERING

Zustandsdiagramm hinsichtlich der »**Vollständigkeit**« untersuchen:

- **Zustandsübergangstabelle** anlegen.
- **Wächterbedingungen** bez. eines Ereignisses auf »Vollständigkeit« und Konsistenz prüfen.
- Nicht spezifizierte Zustands/Ereignis-Paare hinterfragen.



Zustand / Ereignis	initial	empty	filled	full
Stapel()	empty	N/A	N/A	N/A
~Stapel()	N/A	final	?	?
push()	N/A	filled	filled, full	full
pop()	N/A	?	empty, filled	filled
top()	N/A	?	filled	full

3. Aufbau des Übergangsbaumes: Zustands-Konformanztest

1. Anfangszustand: **Wurzel** des Baumes.
2. Für jeden möglichen **Übergang** vom Anfangszustand zum Folgezustand im Zustandsdiagramm:

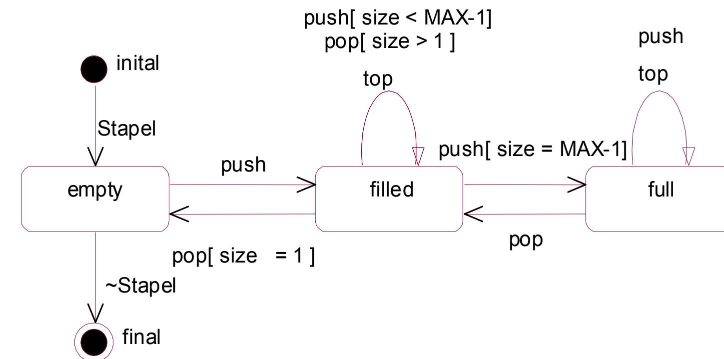
   - Übergangsbaum erhält von Wurzel aus **Zweig** zum **Knoten: Nachfolgezustand**.

   - Notieren: Ereignis und Wächterbedingung am Zweig.

3. Schritt 2 für jedes Blatt des Übergangsbaums wiederholen, bis eine der **Endbedingungen** eintritt:
   - Dem Blatt entsprechender Zustand: Auf »höherer Ebene« einmal im Baum enthalten.
   - Dem Blatt entsprechender Zustand: Endzustand und hat keine weitere Übergänge zu berücksichtigen.

→ Jedes Blatt unabhängig von davor liegender Historie betrachten.

Anm.: **Zyklen** werden dadurch höchstens einmal durchlaufen, garantiert endlichen Übergangsbaum, endliche Länge von Testfolgen und somit endliche Menge von Testfolgen.



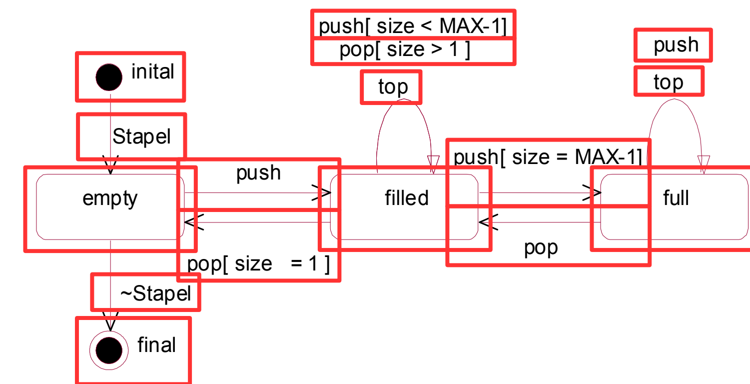
?

3. Aufbau des Übergangsbaumes: Zustands-Konformanztest

1. Anfangszustand: **Wurzel** des Baumes.
2. Für jeden möglichen **Übergang** vom Anfangszustand zum Folgezustand im Zustandsdiagramm:

   - Übergangsbaum erhält von Wurzel aus **Zweig** zum **Knoten: Nachfolgezustand**.

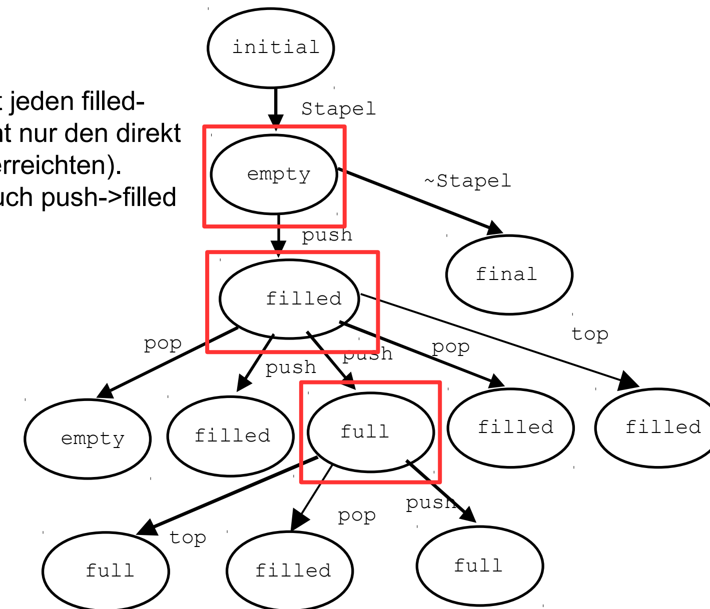
   - Notieren: Ereignis und Wächterbedingung am Zweig.



3. Schritt 2 für jedes Blatt des Übergangsbaums wiederholen, bis eine der **Endbedingungen** eintritt:
   - Dem Blatt entsprechender Zustand: Auf »höherer Ebene« einmal im Baum enthalten.
   - Dem Blatt entsprechender Zustand: Endzustand und hat keine weitere Übergänge zu berücksichtigen.

→ Jedes Blatt unabhängig von davor liegender Historie betrachten.

Repräsentiert jeden filled-Zustand (nicht nur den direkt nach empty erreichten). Deswegen auch push->filled

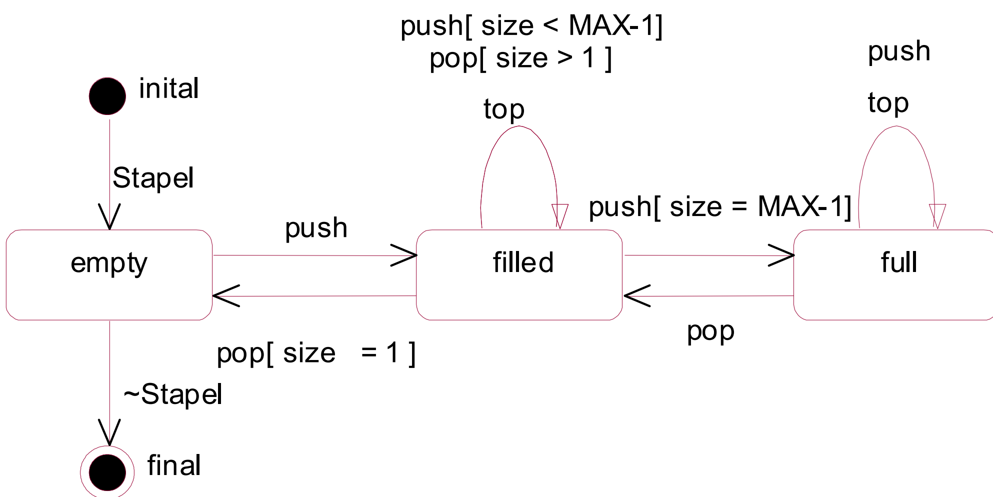


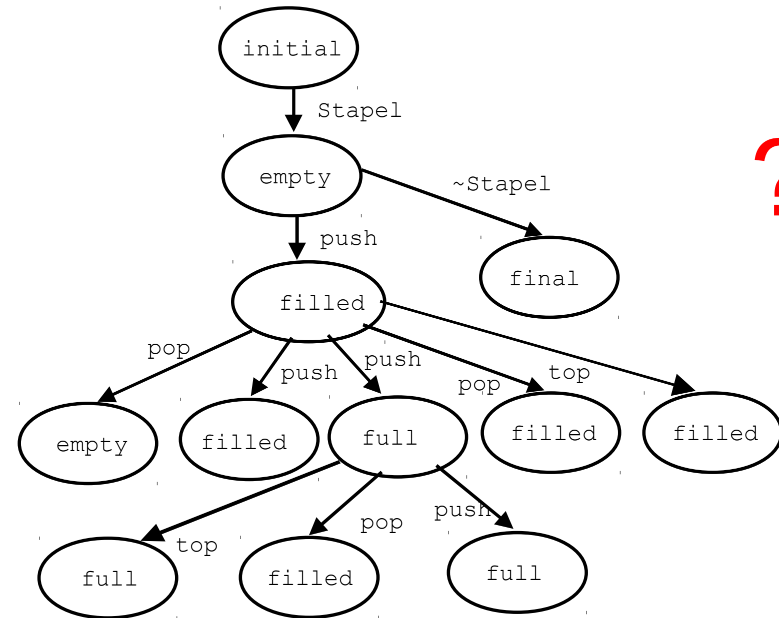
(Wächterbedingungen hier nicht dargestellt)

Anm.: **Zyklen** werden dadurch höchstens einmal durchlaufen, garantiert endlichen Übergangsbaum, endliche Länge von Testfolgen und somit endliche Menge von Testfolgen.

4. Erweitern des Übergangsbaumes: Zustands-Robustheitstest

- **Robustheit** unter spezifikationsverletzenden Benutzungen prüfen.
- Für Botschaften, für die aus betrachtetem Knoten kein Übergang spezifiziert ist, **Übergangsbaum** um neuen »Fehler«-Zustand erweitern.

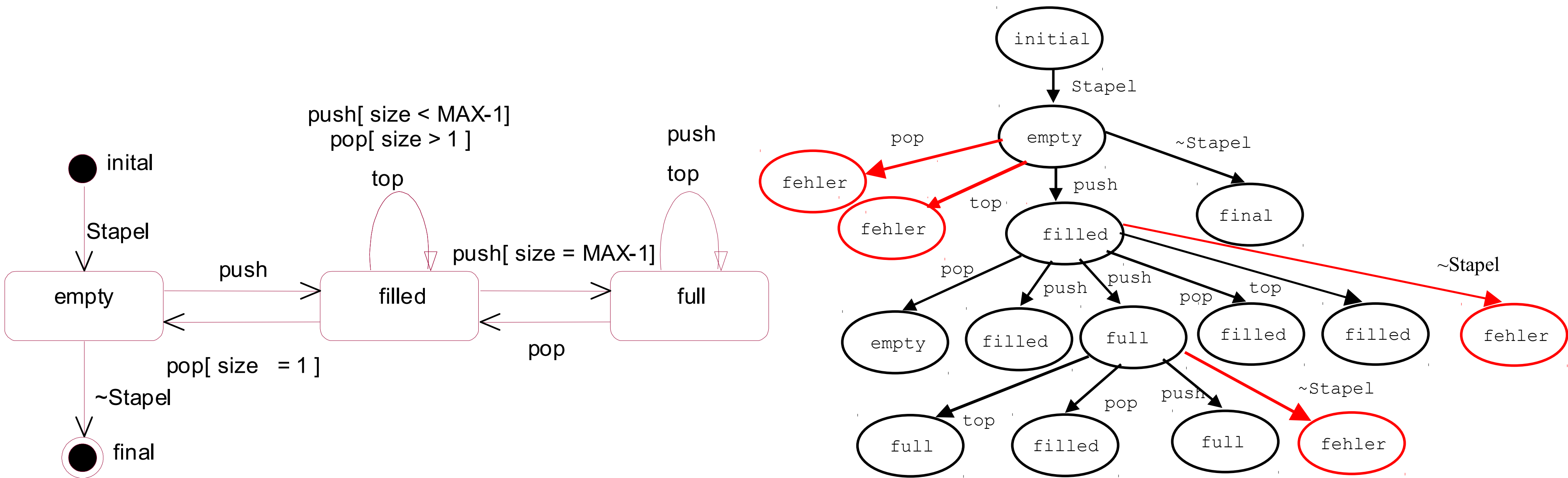




4. Erweitern des Übergangsbaumes: Zustands-Robustheitstest

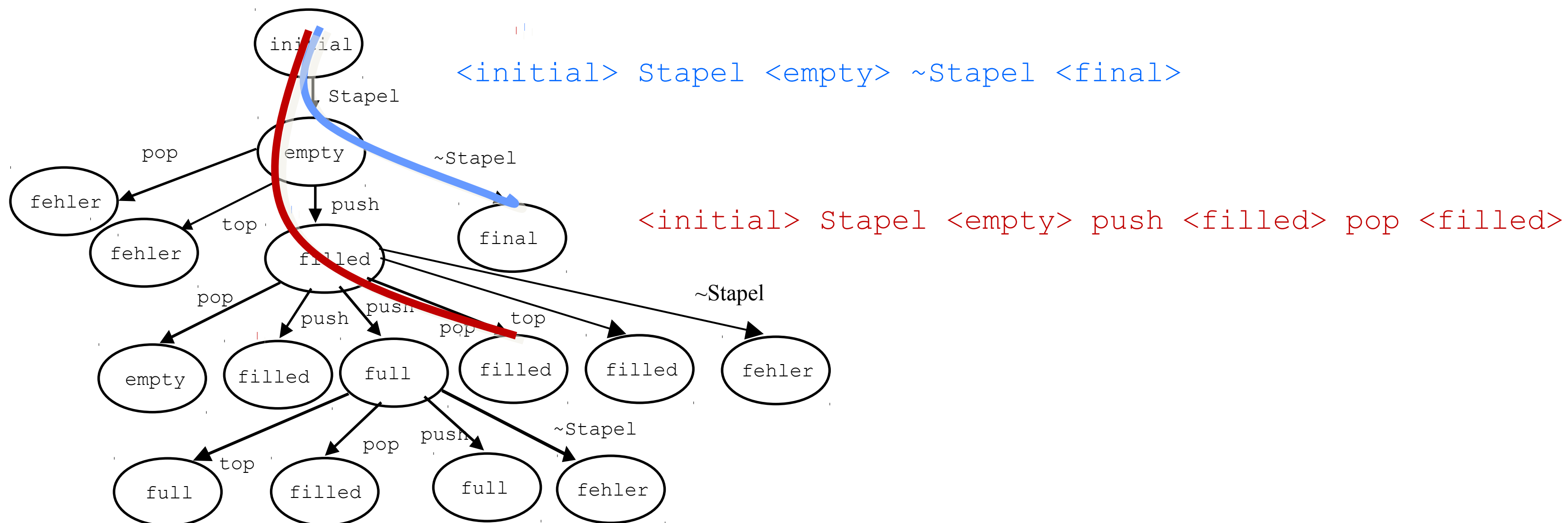
LEHRSTUHL 14
SOFTWARE ENGINEERING

- **Robustheit** unter spezifikationsverletzenden Benutzungen prüfen.
- Für Botschaften, für die aus betrachtetem Knoten kein Übergang spezifiziert ist, **Übergangsbaum** um neuen »Fehler«-Zustand erweitern.



5. Generieren der Testfälle (1)

- Pfade von Wurzel zu Blättern im erweiterten Übergangsbaum: **Funktions-Sequenzen**.
- **Stimulierung** des Testobjekts mit entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab.
- Nicht unbedingt alle möglichen Variablenbelegungen → **Nicht kompletter** theoretisch möglicher **Zustandsraum**.
- Parameter ergänzen.



5. Generieren der Testfälle (2)

LEHRSTUHL 14
SOFTWARE ENGINEERING

- **Stimulierung** des Testobjekts mit entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab.
- Nicht unbedingt alle möglichen Variablenbelegungen → **Nicht kompletter** theoretisch möglicher **Zustandsraum**.
- Für Konformanztests: Wächterbedingungen beachten !

Zustands-Konformanztest:

```
K1 = <initial> new Stapel() <empty> ~Stapel() <final>
K2 = <initial> new Stapel() <empty> push() <filled> pop() <empty>
K3 = <initial> new Stapel() <empty> push() <filled> push() <filled>
K4 = <initial> new Stapel() <empty> push() <filled> pop() <filled>
...
K8 = <initial> new Stapel() <empty> push() <filled> push() <full> push() <full>
```

Welche Folge verletzt Wächter-bedingung ?

Zustands-Robustheitstest:

```
R1 = <initial> new Stapel() <empty> pop() <fehler>
R2 = <initial> new Stapel() <empty> top() <fehler>
R3 = <initial> new Stapel() <empty> push() <filled> ~Stapel() <fehler>
R4 = <initial> new Stapel() <empty> push() <filled> push() <full> ~Stapel() <fehler>
```

5. Generieren der Testfälle (2)

- **Stimulierung** des Testobjekts mit entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab
- Nicht unbedingt alle möglichen Variablenbelegungen → **Nicht kompletter** theoretisch möglicher **Zustandsraum**.
- Für Konformanztests: Wächterbedingungen beachten !
- Konformanztests, die Wächterbedingungen verletzen, **sinnvoll für Robustheitstests**.

Zustands-Konformanztest:

```
K1 = <initial> new Stapel() <empty> ~Stapel() <final>
K2 = <initial> new Stapel() <empty> push() <filled> pop
K3 = <initial> new Stapel() <empty> push() <filled> push() <filled>
K4 = <initial> new Stapel() <empty> push() <filled> pop() <filled>
...
K8 = <initial> new Stapel() <empty> push() <filled> push() <full> push() <full>
```

Wächterbedingung: size() > 1 !!

push() <filled> pop() <filled>

Zustands-Robustheitstest:

```
R1 = <initial> new Stapel() <empty> pop() <fehler>
R2 = <initial> new Stapel() <empty> top() <fehler>
R3 = <initial> new Stapel() <empty> push() <filled> ~Stapel() <fehler>
R4 = <initial> new Stapel() <empty> push() <filled> push() <full> ~Stapel() <fehler>
```

Zustandsbasierte Testfälle

Vollständiger **zustandsbasierter Testfall** umfasst:

- Anfangszustand des Testobjektes (Komponente oder System)
- Eingaben für das Testobjekt
- Erwartete Ausgaben bzw. das erwartete Verhalten
- Erwarteter Endzustand

Für jeden **im Testfall erwarteten Zustandsübergang**

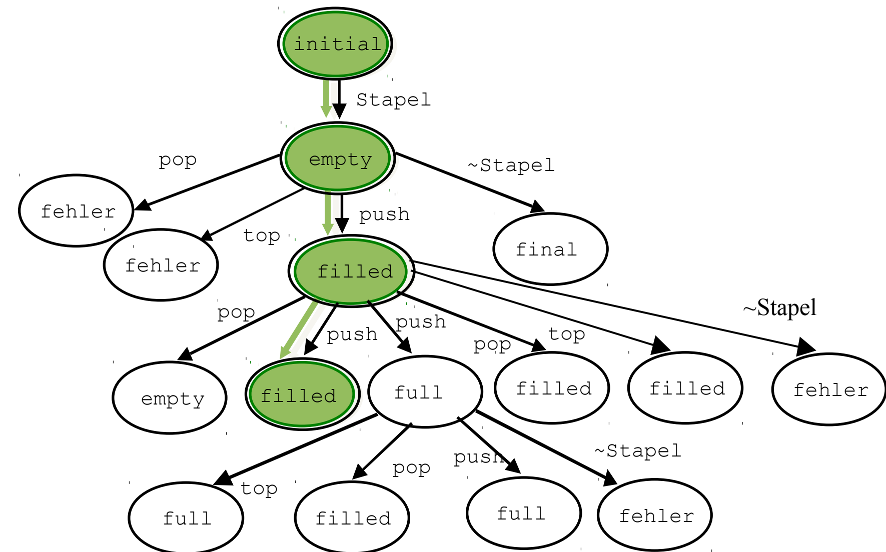
- Zustand vor dem Übergang
- Auslösendes Ereignis, das den Übergang bewirkt
- Erwartete Reaktion, ausgelöst durch den Übergang
- Nächster erwarteter Zustand

festlegen.

Ausführen der Tests

- Testfälle in **Testskript** verkapseln.
- Unter Benutzung **Testtreibers** ausführen.
- Zustände über **zustandserhaltende Operationen** ermitteln und protokollieren.

```
K3' = //<initial>
          Stapel OUT = new Stapel(5)
      //<empty>
          OUT.push(new Object())
      //<filled>
          OUT.push(new Object())
      //<filled>
          if (OUT.size() != 2) then
         throw WrongStateException;
```



Minimalkriterium: Jeder Zustand mindestens einmal eingenommen.

Z-Überdeckungsgrad = Anzahl getestete Z / Gesamtzahl Z

Weitere Kriterien:

- Jeder Zustandsübergang mindestens einmal ausgeführt.

  ZÜ-Überdeckungsgrad = Anzahl getestete ZÜ / Gesamtzahl ZÜ
- Alle spezifikationsverletzenden Zustandsübergänge angeregt.
- Jede Funktion mindestens einmal ausgeführt.

Bei **hoch kritischen Anwendungen:**

- Alle Zustandsübergänge und »Zyklen« im Zustandsdiagramm.
- Alle Zustandsübergänge in jeder beliebigen Reihenfolge mit allen möglichen Zuständen, auch mehrfach hintereinander.

Bewertung des zustandsbasierten Tests

Zustand: Konstellation unterschiedlicher Werte der Variablen:

- Aufgespannter **Zustandsraum:** Sehr **komplex**.
  [Folge der Systemkomplexität und nicht des Testansatzes.]
- **Überprüfung** einzelner Testfälle: **Aufwändig**.

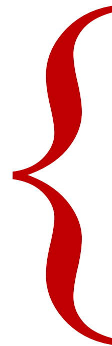
Zustandsbasierte Tests dort, wo **Funktionalität** durch jeweiligen **Zustand** des Testobjektes unterschiedlich **beeinflusst** wird.

- Keine Berücksichtigung anderer vorgestellter Testentwurfsverfahren.
  → Gehen nicht auf Abhängigkeit des Verhaltens der Funktionen vom Zustand ein.

Geeignet zum Test **objektorientierter Systeme**:

- Objekte können unterschiedliche Zustände annehmen.
- Jeweilige **Methoden** zur Manipulation der Objekte: Entsprechend auf unterschiedliche **Zustände** reagieren.
- **Beim objektorientierten Testen:** Zustandsbasierter Test hat herausgehobene Bedeutung.

2.3 Black-Box-Test



Idee der Black-Box-Testentwurfsverfahren
Äquivalenzklassenbildung
Zustandsbasierter Test
Entscheidungstabellentest

Entscheidungstabellentest

- Anwendbar bei Systemanforderungen mit
  - **logischen Bedingungen** und
  - komplexen, vom System umzusetzende **Regeln** in Geschäftsprozessen.
- Spezifikation untersuchen und Eingabebedingungen und Aktionen des Systems ermitteln und festsetzen (»wahr« oder »falsch«).
- **Entscheidungstabelle** enthält Kombinationen von »wahr« und »falsch« für alle Eingabebedingungen und daraus resultierende Aktionen.
- Jede **Spalte** der Tabelle: **Regel im Geschäftsprozess.**

  → Definiert eindeutige Kombination der Bedingungen.

  → Zieht Ausführung mit dieser Regel verbundene Aktionen mit sich.
- Bei Entscheidungstabellentest verwendeter **Standardüberdeckungsgrad**: Wenigstens **ein Testfall pro Spalte**.

Entscheidungstabellentest – Die vier Quadranten einer Entscheidungstabelle

Bedingungen	Regeln
Aktionen	Aktionszeiger

- Bedingungen:
  - Mögliche Zustände von Objekten
- Regeln:
  - Kombinationen von Bedingungswerten
- Aktionen:
  - Aktivitäten, die abhängig von den Regeln auszuführen sind
- Aktionszeiger:
  - Belegungen der Bedingungen mit Aktionen

Entscheidungstabellentest: Beispiel

Geschäftsregeln im Warenwirtschaftssystem:

- **Bestellmenge** muss größer als Null sein.
- **Teil-Lieferungen** nicht erlaubt.
- Bei Annahme einer Bestellung muss **Lagermenge** entsprechend reduzieren.
- Beim Unterstritten von Mindestmenge eines Lagerartikels: **Nachbestellung.**

Textteil	Regelteil			
Bestellmenge > 0	N	J	J	J
Bestellmenge > Art-Lagermenge	-	J	N	N
Art-Lagermenge - Bestellmenge >= Art-Mindestmenge	-	-	N	J
Melde "Bestellmenge ungültig"	X			
Melde "Menge nicht ausreichend"		X		
Reduziere Lagermenge			X	X
Schreibe Nachbestellung			X	

Bedingungsanzeiger:
- N = nicht erfüllt
- J = erfüllt
- \- = ohne Bedeutung
- # = nicht definiert

Aktionsanzeiger:
- X = ausführen
- = nicht ausführen (auch „–“)

- ET ***vollständig***, wenn bei n Bedingungen alle 2^n Kombinationen enthalten (Spalten im oberen Teil).
- ET ***redundanzfrei***, wenn verschiedene Bedingungen zu anderen Aktionen führen.
- ET ***widerspruchsfrei***, wenn logische Beziehungen zwischen Bedingungen zu konsistenten Aktionen führen.

Vollständig, redundanzfrei, widerspruchsfrei ?

Bestellmenge > 0	N	N	N	N	J	J	J	J
Bestellmenge > Art-Lagermenge	N	N	J	J	N	N	J	J
Art-Lagermenge - Bestellmenge >= Art-Mindestmenge	N	J	N	J	N	J	N	J
Melde "Bestellmenge ungültig"	X	X	X	X				
Melde "Menge nicht ausreichend"							X	X
Reduziere Lagermenge					X	X		
Schreibe Nachbestellung					X			

Bestellmenge > 0	N	J	J	J
Bestellmenge > Art-Lagermenge	-	J	N	N
Art-Lagermenge - Bestellmenge >= Art-Mindestmenge	-	-	N	J
Melde "Bestellmenge ungültig"	X			
Melde "Menge nicht ausreichend"		X		
Reduziere Lagermenge			X	X
Schreibe Nachbestellung			X	

Analyse von Entscheidungstabellen

- ET ***vollständig***, wenn bei n Bedingungen alle 2^n Kombinationen enthalten (Spalten im oberen Teil).
- ET ***redundanzfrei***, wenn verschiedene Bedingungen zu anderen Aktionen führen.
- ET ***widerspruchsfrei***, wenn logische Beziehungen zwischen Bedingungen zu konsistenten Aktionen führen.

Vollständig, redundanzfrei, widerspruchsfrei.

Bestellmenge > 0	N	N	N	N	J	J	J	J
Bestellmenge > Art-Lagermenge	N	N	J	J	N	N	J	J
Art-Lagermenge - Bestellmenge >= Art-Mindestmenge	N	J	N	J	N	J	N	J
Melde "Bestellmenge ungültig"	X	X	X	X				
Melde "Menge nicht ausreichend"							X	X
Reduziere Lagermenge					X	X		
Schreibe Nachbestellung					X			

Redundanzfrei, widerspruchsfrei.

Bestellmenge > 0	N	J	J	J
Bestellmenge > Art-Lagermenge	-	J	N	N
Art-Lagermenge - Bestellmenge >= Art-Mindestmenge	-	-	N	J
Melde "Bestellmenge ungültig"	X			
Melde "Menge nicht ausreichend"		X		
Reduziere Lagermenge			X	X
Schreibe Nachbestellung			X	

Entscheidungstabellentest: Testfälle und -daten

LEHRSTUHL 14
SOFTWARE ENGINEERING

Jede Spalte (Regel): Ein Testfall.

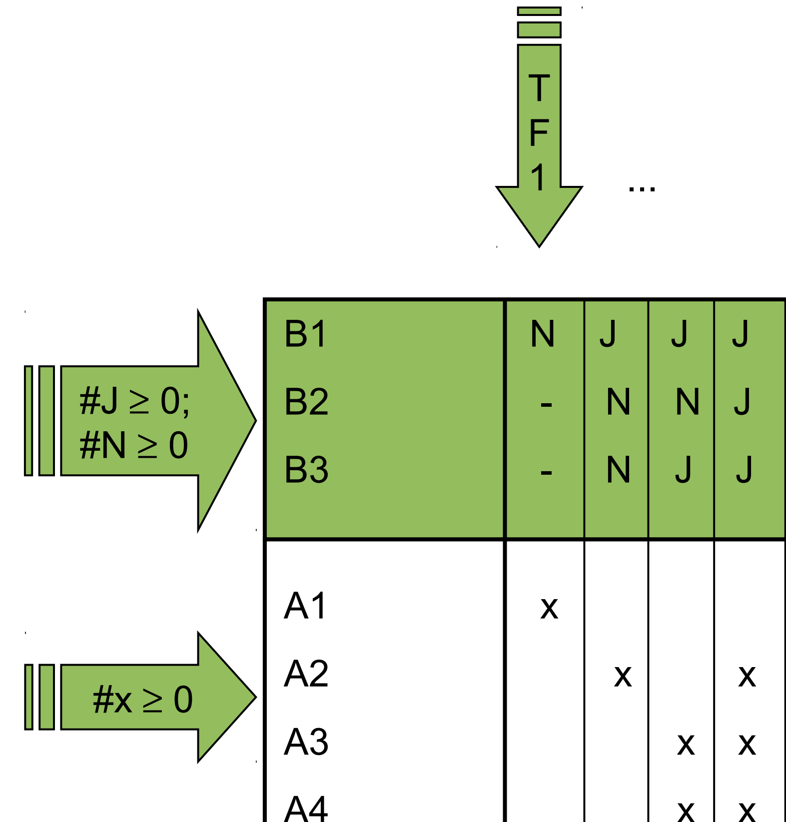
- Voraussetzungen pro Tabelle gleich.
- Bedingungen beziehen sich auf Eingaben.
- Aktionen spiegeln vorausgesagtes Ergebnis wider.

Überdeckungskriterien:

- Alle Bedingungen mindestens einmal „N“ bzw. „J“.
- Alle Aktionen mindestens einmal „x“.
- Alle Bedingungskombinationen.

Konkrete Testdaten aus Wertebereichen ableiten:

- Äquivalenzklassenbildung
- Grenzwertanalyse
- ...



B1	N	J	J	J
B2	-	N	N	J
B3	-	N	J	J
A1	x			
A2		x		x
A3			x	x
A4			x	x

Bewertung des Entscheidungstabellentests

LEHRSTUHL 14
SOFTWARE ENGINEERING

Vorteile:

- Stärke des **Entscheidungstabellentests:** Ableitung Kombinationen von Bedingungen, die andernfalls nicht getestet werden.
- Entscheidungstabellen-Technik zur **Problemlösung** anwendbar, wenn Abläufe von mehreren logischen Entscheidungen abhängen.
- **Logische Zusammenhänge** systematisch formulierbar.
- Entscheidungstabellen auf Redundanz, Widerspruchsfreiheit und Vollständigkeit prüfbar.
- Zwingen nicht zur Strukturierung eines Ablaufs.
- Anwendbar auch bei einfacheren zustandsabhängigen Problemen.

Nachteile:

- **Unübersichtlich** bei zu vielen Bedingungen.
- **Zusammenhänge** zwischen einzelnen Bedingungen nur **implizit ausdrückbar.**

Aufgedeckte Fehler pro Testkriterium:

- **Angabe konkreter Zahlen** fragwürdig: Studien unterschiedlicher Art mit unterschiedlichen Ansätzen.
- Im Vergleich gilt:
  - **Äquivalenzklassenmethode besser** als Zufallstest.
    - (Nur) geringe Überlegenheit resultiert aus Problem, dass keine algorithmisch-konstruktiven Verfahren zum Erzeugen von vollständig homogenen Äquivalenzklassen gibt.
      - **Homogen:** Für jedes Eingabedatum einer Klasse tritt ein/kein Fehler auf.
  - **Zustandsbasiertes / Entscheidungstabellen-Testen besser** als Äquivalenzklassenmethode.

→ Konzentration auf **fehlerträchtige Eingaben** (Zustandsübergange, fehlersensitive Ursachenkombinationen).

Bewertung der Black-Box-Testentwurfsverfahren

Grundlagen: **Anforderungen** und **Spezifikation** des Systems und ihr Zusammenwirken.

- **Fehlerhafte Anforderungen** oder **Spezifikationen nicht erkannt**.
- Testobjekt verhält sich nach Forderung der Spezifikation, auch falls fehlerhaft.

Nicht geforderte Funktionalität nicht erkannt:

- **Zusätzliche Funktionen nicht spezifiziert**.
- Testfälle, die diese zusätzlichen Funktionen zur Ausführung bringen, werden nur zufällig durchgeführt.
- Überdeckungskriterien auf Grundlage der Anforderungen.
- **Mutationstesten** kann hier Abhilfe schaffen.

Im Mittelpunkt: **Prüfung der Funktionalität** des Testobjektes.

Korrektes Funktionieren eines Softwaresystems hat höchste Priorität.
→ Black-Box-Testentwurfsverfahren einsetzen.

Zusammenfassung

In **diesem** Abschnitt:

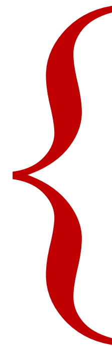
- Dynamische Tests führen Testobjekt aus.
- Auswahl der Testfälle als (gute) Stichprobe.
- Black-Box-Testentwurfsverfahren benötigen zur Auswahl der Testfälle keine Kenntnis der Programmlogik.
- Funktionale Tests leiten Testfälle anhand Spezifikation des Testobjekts ab.
- Äquivalenzklassenbildung zur Erstellung der Testfälle.
- Zustandsbasierte Tests mit Übergangsbaum und Zustands-Konformanztest sowie erweitertem Übergangsbaum und Zustands-Robustheitstest.
- Entscheidungstabellentest bei regelbasierten Anforderungen.

Im **nächsten** Abschnitt:

- **White-Box-Test:** Analyse interner Struktur der Testobjekte.

2.3 Black-Box-Test

2.3 Black-Box-Test



- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung, Grenzwerttesten
- Zustandsbasierter Test
- Entscheidungstabellentest

Grenzwertanalyse
Idee

- **Idee**: **Grenzbereiche** in **Verzweigungs- und Schleifenbedingungen**, für die die Bedingung gerade noch zutrifft.
  - Solche Fallunterscheidungen: **fehlerträchtig**.
  - Testdaten, die solche Grenzwerte prüfen, decken Fehlerwirkungen mit höherer Wahrscheinlichkeit.
- Beste Erfolge bei Kombination mit anderen Verfahren.
- Bei Kombination mit **Äquivalenzklassenbildung**:
  - **Grenzen der ÄK** testen.
  - Jeder »Rand« einer ÄK in einer Testdatenkombination.

Grenzwertanalyse
Ziel

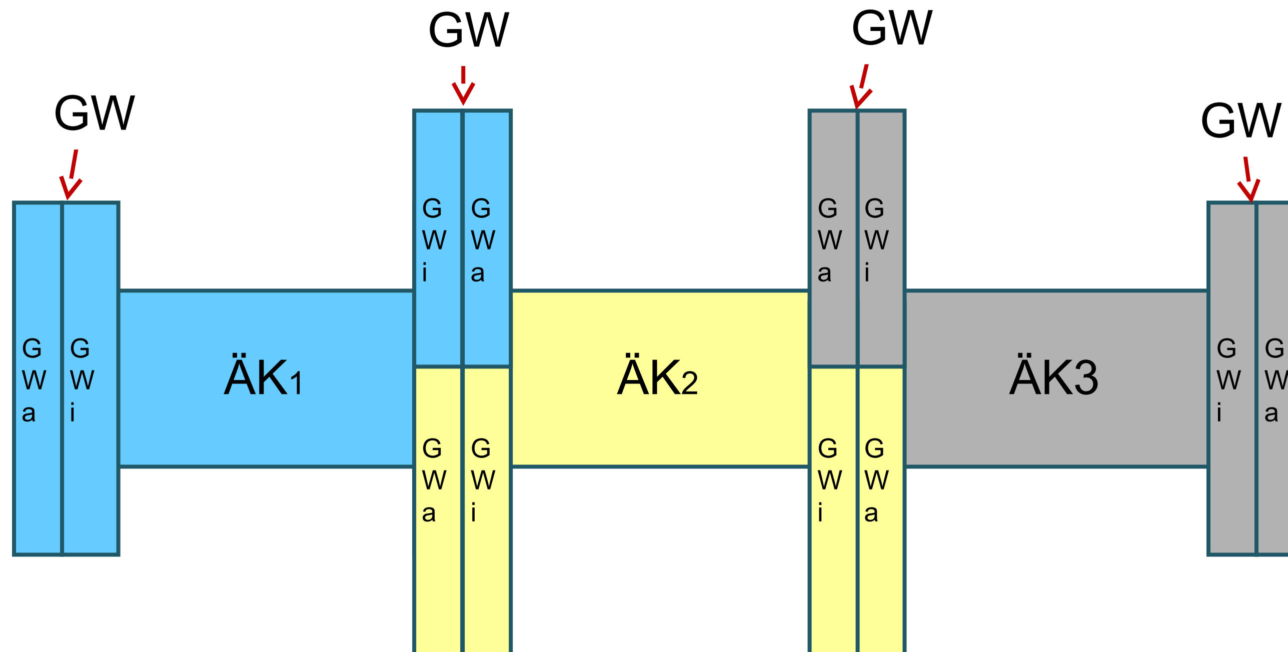
Ziel: Auswahl von Werten aus Äquivalenzklasse bestehend aus geordneter Menge.

Vorgehen:

- Schritt 1: **Auswahl von Testwerten:** Auf oder neben beiden Grenzen einer **Eingabe**äquivalenzklasse.
  - Äquivalenzklasse ist **Wertebereich:** Nimm größten und kleinsten Wert.
  - Äquivalenzklasse ist **Anzahl von Werten:** Nimm größte und kleinste gültige Anzahl.
- Schritt 2: **Auswahl von Testwerten**: Auf oder neben beiden Grenzen einer **Ausgabe**äquivalenzklasse.

Grenzwertanalyse
Innere / äußere Grenzwerte

Zusammenfallen der Grenzwerte **benachbarter Äquivalenzklassen**:

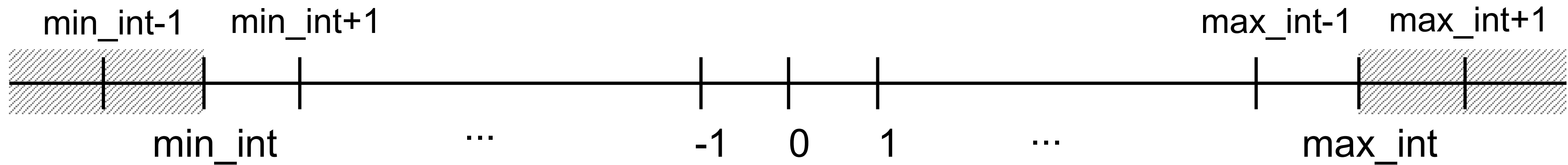


An Grenzen immer zwei Tests, egal ob auf bzw. nur vor oder nach Grenze testen.

ÄK - Äquivalenzklasse
GW – Grenzwert:
i - innerhalb der ÄK
a - außerhalb der ÄK

Beispiele zur Grenzwertanalyse

LEHRSTUHL 14
SOFTWARE ENGINEERING



Datentyp	Grenzen	Größer	Kleiner
integer	0	1	-1
	min_int	min_int + 1	min_int - 1
	max_int	max_int + 1	max_int - 1
char[5]	"xxxxx"	"xxxxxx"	"xxxx"
double	0.0e0,	+ δ	- δ
	min_double (-∞)	min_double + δ	min_double - δ
	max_double (+∞)	max_double + δ	max_double - δ
	NaN (not a number)	??	??

Grenzwertanalyse
Ausgangskriterium (Testendekriterium)

Analog zum Ausgangskriterium der Äquivalenzklassenbildung: Festlegung einer anzustrebenden **Überdeckung der Grenzwerte (GW)** vorab und Berechnung nach Durchführung der Tests:

GW-Überdeckungsgrad = Anzahl getestete GW / Gesamtzahl GW

Grenzwertanalyse
Vor- und Nachteile

Vorteile:

- An Grenzen von Äquivalenzklassen: **Häufiger Fehler zu finden** als innerhalb dieser Klassen.
- »Grenzwertanalyse: Eine der **nützlichsten Methoden** für Testfallentwurf bei richtiger Anwendung.«

  Myers, Glenford J.: Methodisches Testen von Programmen, Oldenbourg, 2001 (7. Auflage)
- Effiziente Kombination mit anderen Verfahren: **Freiheitsgrade bei Wahl** der Testdaten.

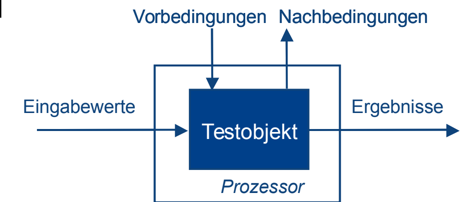
Nachteile:

- Rezepte für Auswahl von Testdaten schwierig anzugeben.
- Bestimmung aller relevanten Grenzwerte schwierig.
- Kreativität zur Findung erfolgreicher Testdaten gefordert.
- **Anwendung nicht effizient** genug, da einfache Erscheinung.

Anhang: weitere Informationen und Beispiele zum Nacharbeiten

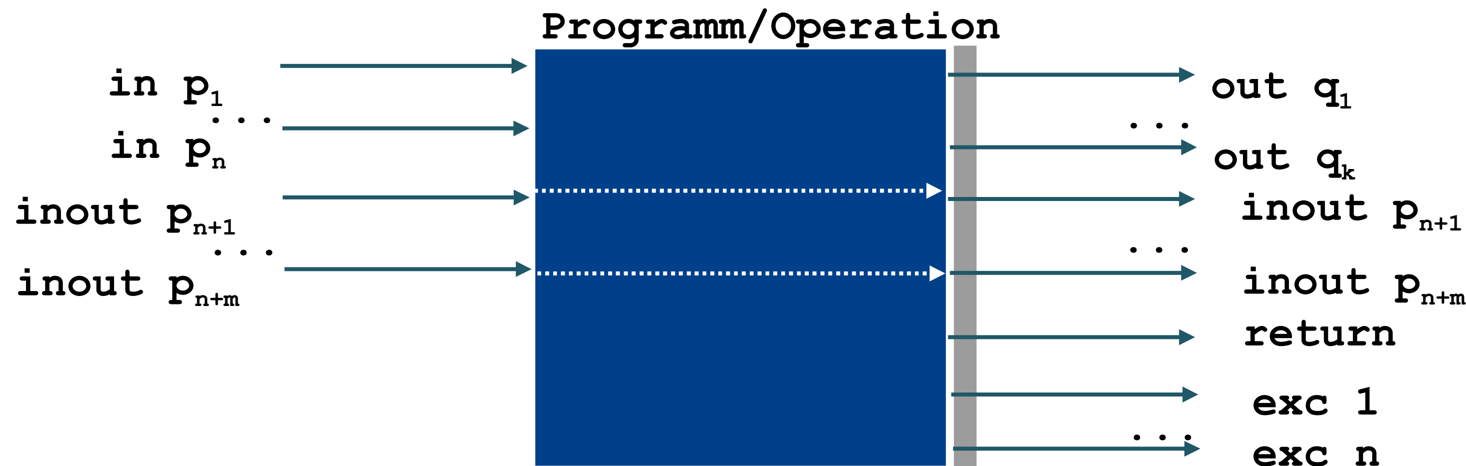
Programme und Operationen

Vereinfachende **Annahme:** Programme / Operationen berechnen Ausgaben aus Eingaben (zustandslos).

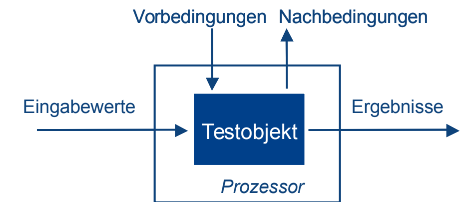


Signatur einer Operation:

- Operationsname, Parametertypen, Rückgabetyp
- Parameter als **`in`**, **`inout`**, **`out`** gekennzeichnet
- Ggf. ein **`out`**-Parameter als Rückgabewert (Funktion)
- Ggf. return oder Exceptions



Operationen in Java (»Methoden«)



- Mehrere **Eingabeparameter:**
  - Atomare Typen: nur call-by-value (in)
  - Klassen bzw. Objekte: call-by-reference (inout)
- Ein **Rückgabewert:**
  - Atomarer Typ (out)
  - Klasse bzw. Objekt (out, inout)
- Ggfs. mehrere **Exceptions.**
- **Typen** spezifizieren Definitionsbereiche.

Beispiel: Funktion ggT

Bestimmung des größten **gemeinsamen Teilers (ggT)** zweier ganzer Zahlen m und n:

ggT(4,8)=4; ggT(5,8)=1; ggT(15,35)=5; ggT(0,0)=0 [per Konvention]

Logische Spezifikation:

$ggT: Z \times Z \rightarrow IN$

$ggT(0,0) = 0 \wedge$

$[m<>0 \vee n<>0 \Rightarrow ggT(m,n)|m \wedge ggT(m,n)|n \wedge \forall o \in IN: o>ggT(m,n) \Rightarrow (\neg(o|m) \vee \neg(o|n))]$

Spezifikation in UML / Java:

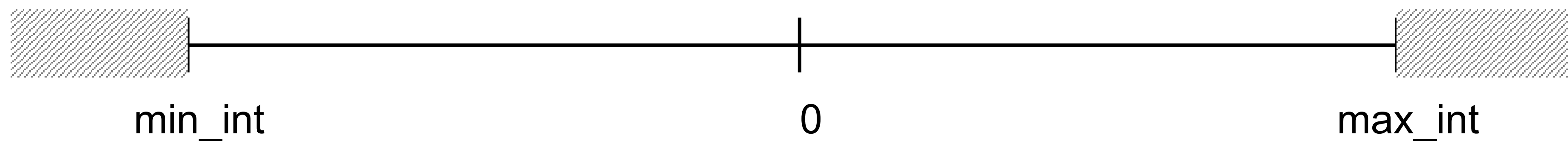
```
public int ggt (int m, int n) {
// pre:  m<>0 or n<>0
// post: m@pre.mod(return) = 0 and
//    n@pre.mod(return) = 0 and
//    forall(i : int | i > return implies
//       (m@pre.mod(i) > 0 or n@pre.mod(i) > 0)
... )
```

Äquivalenzklassen für ggT: Erster Schritt

Definitionsbereiche der Ein- und Ausgaben

- Eingabeparameter: `int`
- Rückgabewert: `int`

Gültige von ungültigen Teilbereichen der Java-Implementierung unterscheiden:



Äquivalenzklassenbildung
Vorgehensweise

Aufstellen der Definitionsbereiche aus Spezifikation.

Äquivalenzklassenbildung für jede Beschränkung:

- Beschränkung spezifiziert **Wertebereich**: Eine gültige und zwei ungültige ÄK.
- Beschränkung spezifiziert **minimale und maximale** Anzahl von Werten: Eine gültige und zwei ungültige ÄK.
- Beschränkung spezifiziert **Menge von Werten**, die unterschiedlich behandelt werden: Für jeden Wert dieser Menge eigene gültige ÄK und zusätzlich eine ungültige ÄK.
- Beschränkung spezifiziert **Situation**, die erfüllt sein muss: Eine gültige und eine ungültige ÄK.
- Werte einer ÄK **nicht gleichwertig** behandelt: Aufspaltung der ÄK in kleinere ÄK.

Beschränkung spezifiziert **Situation**, die erfüllt sein muss: Eine gültige und eine ungültige ÄK.

Beispiel

Laut **Spezifikation** erhält jedes Mitglied im Sportverein eine eindeutige Mitgliedsnummer. Diese beginnt mit dem ersten Buchstaben des Familiennamens des Mitglieds.

Gültige Äquivalenzklasse: erstes Zeichen ein Buchstabe.
Ungültige Äquivalenzklasse: erstes Zeichen kein Buchstabe (z.B. eine Ziffer oder ein Sonderzeichen).

Diskussionsfrage

Gegeben ist eine Funktion zur Bestimmung der Anzahl der Tage eines Monats mit den Übergaben Monat und Jahr.

- ZahlTageMonat(int Monat, int Jahr)

Wie sehen die **Äquivalenzklassen** dazu aus ?

Klassen für **Monat:**

- Gültig:
  - Monate mit 30 Tagen
  - Monate mit 31 Tagen
  - Februar
- Ungültig:
  - > 12
  - < 1

Klassen für **Jahr:**

- Gültig:
  - Schaltjahre
  - Normaljahre

Anzahl der Testfälle minimieren: Heuristiken

- **Testfälle** aus Repräsentanten kombinieren und nach »Häufigkeit« sortieren (»**Benutzungsprofile**«).
  - Testfälle in dieser Reihenfolge priorisieren.
  - Mit »benutzungsrelevanten« Testfällen testen.
  - Testfälle, die Grenzwerte oder Grenzwert-Kombinationen enthalten, bevorzugen.
- **Ausführung**: Jeder Repräsentant einer Äquivalenzklasse mit jedem Repräsentanten anderer Äquivalenzklassen in einem Testfall.

  → Paarweise statt vollständiger Kombination.
- **Minimalkriterium:** Min. ein Repräsentant jeder Äquivalenzklasse in min. einem Testfall.
- Repräsentanten **ungültiger Äquivalenzklassen** nicht miteinander kombinieren.

Äquivalenzklassenbildung

Beispiel Wertebereich

Beschränkung spezifiziert **einen Wertebereich**: Eine gültige und zwei ungültige ÄK.

Beispiel

In der **Spezifikation** des Testobjekts ist festgelegt, dass ganzzahlige Eingabewerte zwischen 1 und 100 möglich sind.

Wertebereich der Eingabe	$1 \leq x \leq 100$
Gültige Äquivalenzklasse	$1 \leq x \leq 100$
Ungültige Äquivalenzklasse	$x < 1$, $x > 100$ und NaN (not a Number)

Äquivalenzklassenbildung
Beispiel minimal / maximal

Beschränkung spezifiziert **minimale und maximale** Anzahl von Werten: Eine gültige und zwei ungültige ÄK.

Beispiel

Laut **Spezifikation** muss sich ein Mitglied eines Sportvereins mindestens einer Sportart zuordnen. Jedes Mitglied kann an maximal drei Sportarten aktiv teilnehmen.

Gültige Äquivalenzklasse $1 \leq x \leq 3$ (1 bis 3 Sportarten)

Ungültige Äquivalenzklasse $x=0$ und $x > 3$ (keine bzw. mehr als 3 Sportarten zugeordnet)

Äquivalenzklassenbildung
Beispiel Menge von Werten

Beschränkung spezifiziert **Menge von Werten**, die unterschiedlich behandelt werden:
Für jeden Wert dieser Menge eigene gültige ÄK und zusätzlich eine ungültige ÄK.

Beispiel

Laut **Spezifikation** gibt es im Sportverein folgende Sportarten: Fußball, Hockey, Handball, Basketball und Volleyball

- **Gültige** Äquivalenzklasse: Fußball, Hockey, Handball, Basketball und Volleyball
- **Ungültige** Äquivalenzklasse: alles andere z.B Badminton

Äquivalenzklassen für ggT

```
public int ggT(int m, int n)
```

Eingabeparameter m: `int`

```
gÄK1_1 : min_int ≤ m < 0
gÄK1_2 : m = 0
gÄK1_3 : 0 < m ≤ max_int
uÄK1_1 : m < min_int
uÄK1_2 : m > max_int
```

Eingabeparameter n: `int`

```
gÄK2_1 : min_int ≤ n < 0
gÄK2_2 : n = 0
gÄK2_3 : 0 < n ≤ max_int
uÄK2_1 : n < min_int
uÄK2_2 : n > max_int
```

Rückgabewert ggT: `int`

```
gÄK3_1 : 1 < ggT ≤ max_int
gÄK3_2 : ggT = 1
uÄK3_1 : ggT = 0
uÄK3_2 : ggT < 0
uÄK3_3 : ggT > max_int
```

Grenzwertanalyse
Anwendung

In Regel: Grenzwert und Werte über bzw. unter dem Grenzwert testen.

Atomare (geordnete) Bereiche:

- Werte auf den Grenzen,
- Werte »rechts bzw. links neben« den Grenzen.

Mengenwertige Bereiche:

- Kleinste und größte gültige Anzahl,
- Zweitkleinste und zweitgrößte gültige Anzahl,
- Kleinste und größte ungültige Anzahl.

Fallen bei Äquivalenzklassen für geordnete Bereiche obere und untere Grenze zweier ÄK zusammen, dann auch die entsprechenden Testfälle.

Grenzwertanalyse
Vorgehen

- **Grenzen des Eingabebereichs**, z.B.:
  - Bereich: [-1.0;+1.0]; Testdaten: -1.001; -1.0; +1.0; +1.001 (-0.999; +0.999)
  - Bereich:]-1.0;+1.0[; Testdaten: -1.0; -0.999; +0.999; +1.0 (-1.001; +1.001)
- **Grenzen der erlaubten Anzahl von Eingabewerten**, z.B.:
  - Eingabedatei mit 1 bis 365 Sätzen; Testfälle 0, 1, 365, 366 (2, 364) Sätze
- **Grenzen des Ausgabebereichs**, z.B.:
  - Programm errechnet Beitrag, der zwischen 0,00 EUR und 600 EUR liegt; Testfälle: 0; 600 EUR; Beiträge < 0; (knapp >0); und für > 600; (knapp < 600)
- **Grenzen der erlaubten Anzahl von Ausgabewerten**, z.B.:
  - Ausgabe von 1 bis 4 Daten; Testfälle: Für 0, 1, 4 und 5 (2, 3) Ausgabewerte
- **Erstes und letztes Element bei geordneten Mengen beachten**.
- **Komplexe Datenstrukturen: leere Mengen testen**.
- **Bei numerischen Berechnungen**: Wahl von eng zusammen und weit auseinander liegender Werte.

Grenzwertanalyse: Beispiel (s. Äquivalenzklassentest)

- Grenzwerte für Eingaben:

Äquivalenzklasse	Gültige Grenzwerte	Ungültige Grenzwerte
(1) Anzahl Parameter	2	1, 3
(2) Dateiname (Länge)	1, 6	0, 7
(6) Zeilenanzahl (Ziffern)	1, 3	0, 4
(7) Zeilenanzahl	1, 999	0, 1000

Keine Untersuchung der Äquivalenzklassen 3-5, da keine geordnete Mengen!

Grenzwertanalyse
Beispiel

Testdaten für **gültige Eingaben**:

Äquivalenzklasse	Testdaten	Ausgewählt bei Äquivalenzklassentest
(1)	PRINT abc1 22	ja
(2)	PRINT a 100	eventuell
(2)	PRINT abcdef 200	eventuell
(6)	PRINT abc 8	eventuell
(6)	PRINT abc 345	eventuell
(7)	PRINT abc 1	eventuell
(7)	PRINT abc 999	eventuell

Testdaten für **ungültige Eingaben:**

Äquivalenzklasse	Testdaten	Ausgewählt bei Äquivalenzklassentest
(1b)	PRINT abc	ja
(1c)	PRINT abc 20 300	eventuell
(2a)	PRINT 20	ja
(2b)	PRINT abcdefg 20	eventuell
(6a)	PRINT abc 4568	eventuell
(7a)	PRINT abc 0	eventuell
(7b)	PRINT abc 1000	eventuell

Grenzwerte für Ausgaben:

- Es können X Seiten gedruckt werden, $1 \leq X \leq 20$
- Letzte Seite enthält Y Zeilen, $1 \leq Y \leq 45$
- Ersten X-1 Seiten enthalten jeweils 45 Zeilen

Äquivalenzklasse	Gültige Grenzwerte	Ungültige Grenzwerte
(1) Anzahl Seiten	1, 20	0, 21
(2) Anzahl Zeilen pro Seite	1, 45	0, 46

Grenzwertanalyse
Beispiel

- Testdaten für **gültige Ausgaben**:
  - PRINT abc 45 (X=1, Y=45)
  - PRINT abc 900 (X=20, Y=45)
  - PRINT abc 46 (X=2, Y=1)
- Testdaten für **ungültige Ausgaben:**
  - PRINT abc 0 (X=0, Y=0)
  - PRINT abc 901 (X=21, Y=1)
  - PRINT abc 46 (X=1, Y=46)

Zustandsbasierter Test: Begriffe

LEHRSTUHL 14
SOFTWARE ENGINEERING

Zustandsautomat: Berechnungsmodell, bestehend aus einer endlichen Anzahl von Zuständen und Zustandsübergängen, ggf. mit begleitenden Aktionen. [IEEE 610].

Zustandsübergang: Übergang zwi. zwei Zuständen einer Komponente oder eines Systems.

Zustandsdiagramm: Diagramm, das Zustände beschreibt, die System oder Komponente annehmen kann, und Ereignisse zeigt, die Zustandswechsel verursachen und/oder ergeben [IEEE 610].

Zustandsübergangstabelle: Tabelle für Darstellung resultierender Übergänge für jeden Zustand in Verbindung mit jedem möglichen Ereignis. → Gültige oder ungültige Übergänge.

Zustandsbasierter Test: Black-Box-Testentwurfsverfahren, zur Entwurf von Testfällen, um gültige und ungültige Zustandsübergänge zu prüfen.

Tipps für den zustandsbasierten Test

- **Zustandsdiagramm**:
  - bei Spezifikation unter **Testgesichtspunkten** bewerten,
  - bei hoher Anzahl von Zuständen und Übergängen auf erhöhten Testaufwand hinweisen,
  - Soweit möglich **auf Vereinfachung dringen**.
- Bei **Spezifikation** achten:
  - **Unterschiedliche Zustände** leicht ermittelbar.
  - **Keine vielfältige Kombination** von Werten von unterschiedlichen Variablen.

  Ggf. **Werkzeuge** zum **Model-Checking** verwenden:
  - **Erreichbarkeit** von Zuständen.
  - Bei interagierenden Testobjekten: **Deadlock, Livelock**, ...