Willkommen zur Vorlesung *Methodische Grundlagen des Software-Engineering* im Sommersemester 2012

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

4.4 Black-Box-Test

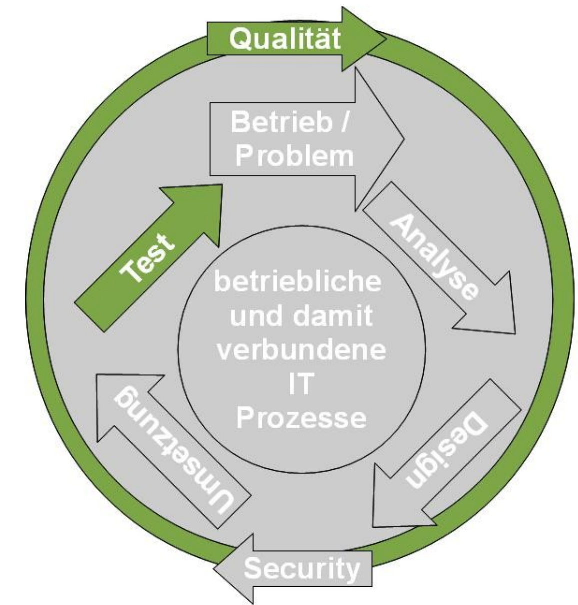
Basierend auf dem Foliensatz
„Basiswissen Softwaretest - Certified Tester“
des „German Testing Board“
(nach Certified Tester Foundation Level Syllabus,
deutschsprachige Ausgabe, Version 2011)
(mit freundlicher Genehmigung)

© Copyright 2007 – 2013 by GTB
V 2.0 / 2011

Der zum Kapitel 4 (Testen) der Vorlesung gehörende Foliensatz ist als Werk urheberrechtlich geschützt durch das German Testing Board; d.h. die Verwertung ist – soweit sie nicht ausdrücklich durch das Urheberrechtsgesetz (UrhG) gestattet ist – nur mit Zustimmung der Berechtigten zulässig. Der Foliensatz darf nicht öffentlich zugänglich gemacht oder im Internet frei zur Verfügung gestellt werden.

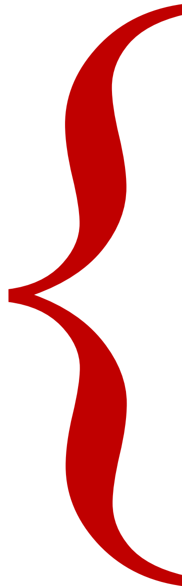
Einordnung
Black-Box-Test

- Geschäfts-Prozesse
- Qualitätsmanagement
- Testen
  - Einführung
  - Grundlagen Softwaretesten
  - Testen im Softwarelebenszyklus
  - Statischer Test
  - Black-Box-Test
  - White-Box-Test
  - Test-Management
  - Testwerkzeuge
  - Fuzzing
- Sicheres Software Design



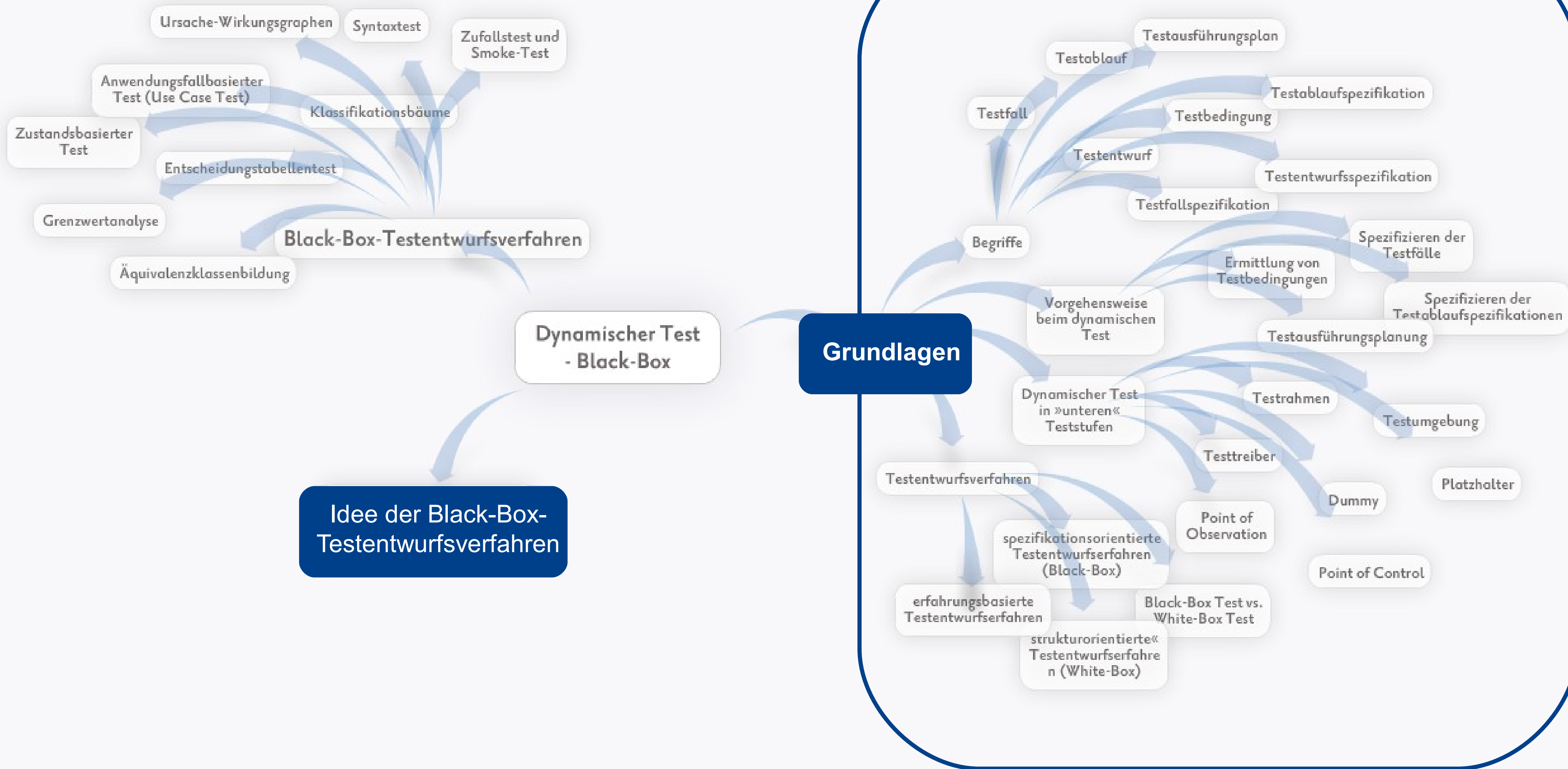
4.4 Black-Box-Test

4.4 Black-Box-Test



- Dynamischer Test – Grundlagen
- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- Grenzwertanalyse
- Zustandsbasierter Test
- Entscheidungstabellentest
- Weitere Black-Box-Testentwurfsverfahren

Dynamischer Test – Black Box



Statischer Test vs. dynamischer Test

Programme sind statische Beschreibungen von dynamischen Prozessen (Berechnungen).

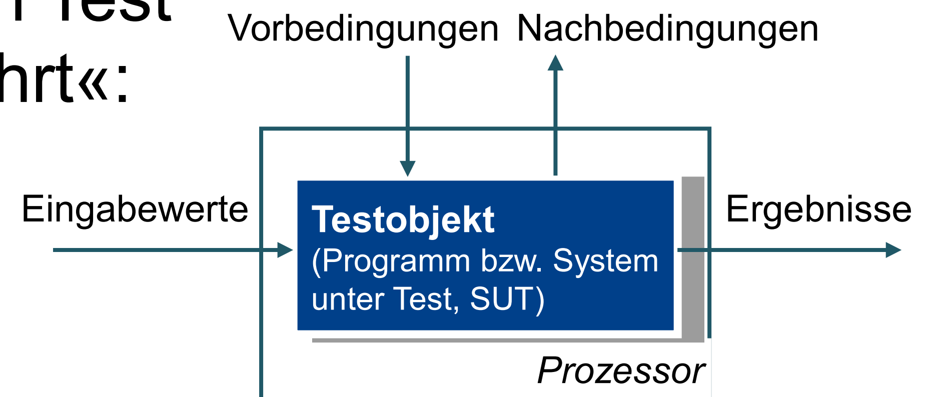
Statische Tests prüfen die Testobjekte »an sich«:

- Artefakte des Entwicklungsprozesses, z.B. informelle Texte, Modelle, formale Texte, Programmcode, ...

Dynamische Tests prüfen die durch »Interpretation« einer Beschreibung (Testobjekt) resultierenden Prozesse.

Das Testobjekt wird im dynamischen Test also auf einem Prozessor »ausgeführt«:

- Bereitstellen von Eingabewerten
- Beobachten der Ergebnisse.



Dynamischer Test – Begriffe (1)

Dynamischer Test: Prüfung des Testobjekts durch Ausführung auf einem Rechner.

Testbasis: Alle Dokumente, aus denen die Anforderungen ersichtlich werden, die an Komponente / System gestellt werden (bzw. Dokumentation für Herleitung / Auswahl der Testfälle).

Testbedingung: Einheit oder Ereignis (z.B. Funktion, Transaktion), Feature, Qualitätsmerkmal oder strukturelles Element einer Komponente oder eines Systems, welche bzw. welches durch einen oder mehrere Testfälle verifiziert werden kann.

Testentwurfsspezifikation: Ergebnisdokument, das Testbedingungen für Testobjekt, detaillierte Testvorgehensweise und zugeordnete logische Testfälle identifiziert [nach IEEE 829].

Testfallspezifikation: Ein Dokument, das eine Menge von Testfällen für ein Testobjekt spezifiziert (inkl. Testdaten und Vor-/Nachbedingung), bei dem die Testfälle jeweils Ziele, Eingaben, Testaktionen, vorausgesagte Ergebnisse und Vorbedingungen für die Ausführung enthalten [nach IEEE 829].

Testsuite: Zusammenstellung (Aggregation) mehrerer Testfälle für Test einer Komponente / Systems, bei der Nachbedingungen eines Tests als Vorbedingungen des folgenden Tests genutzt werden.

Dynamischer Test – Begriffe (2)

Testfall: Umfasst folgende Angaben:
- die für die Ausführung notwendigen **Vorbedingungen**,
- die Menge der **Eingabewerte** (ein Eingabewert je Parameter des Testobjekts),
- die Menge der **vorausgesagten Ergebnisse**, sowie
- die **erwarteten Nachbedingungen**.

Testfälle werden entwickelt im Hinblick auf bestimmtes Ziel bzw. auf Testbedingung, wie z.B. bestimm-ten Programmpfad auszuführen oder Übereinstimmung mit spezifischen Anforderungen zu prüfen (wie Eingaben an das Testobjekt zu übergeben und Sollwerte abzulesen sind) [nach IEEE 610].

Vorbedingung: Bedingungen an den Zustand des Testobjekts und seiner Umgebung, die vor der Durchführung eines Testfalls oder Testablaufs erfüllt sein müssen.

Nachbedingung: Zustand des Testobjekts (und/oder der Umgebung), in dem sich Test-objekt (oder Umgebung) nach Ausführung eines Testfalls / Testlaufs befinden muss.

Testablaufspezifikation: Ein Dokument, das eine Folge von Schritten zur Testausführung festlegt. Auch bekannt als Testskript oder Testdrehbuch [nach IEEE 829].

Testskript: Bezeichnet üblicherweise eine Testablaufspezifikation, insbesondere eine automatisierte.

Dynamischer Test – Begriffe (3)

Testausführungsplan: Ein Plan für die Ausführung von Testskripten. Testskripte sind im Testausführungsplan mit ihrem Kontext und in der auszuführenden Reihenfolge festgelegt.

Testlauf: Ausführung eines / mehrerer Testfälle mit bestimmter Version des Testobjekts.

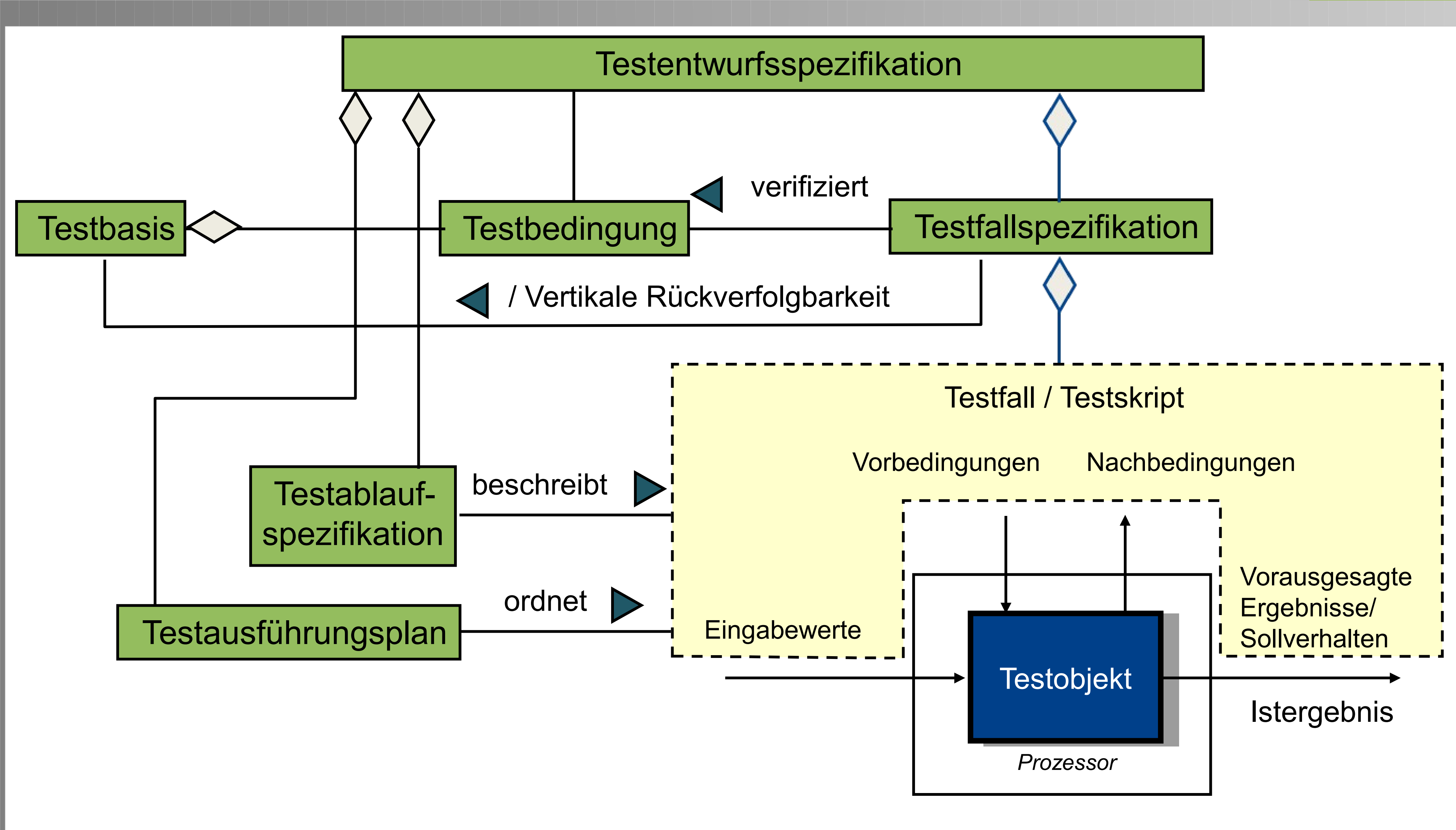
Rückverfolgbarkeit: Fähigkeit, zusammengehörige Teile von Dokumentation und Software zu identifizieren, insbesondere die Anforderungen mit den dazu gehörigen Testfällen.

Vertikale Rückverfolgbarkeit: Die Rückverfolgung von Anforderungen durch die Ebenen der Entwicklungsdokumentation bis zu den Komponenten.

Horizontale Rückverfolgbarkeit: Das Verfolgen von Anforderungen einer Teststufe über die Ebenen der Testdokumentation (z.B. Testkonzept, Testentwurfsspezifikation, Testfallspezifikation, Testablaufspezifikation oder Testskripte).

Rückverfolgbarkeit erlaubt sowohl eine Analyse der Auswirkungen (impact analysis) aufgrund von geänderten Anforderungen, als auch die Bestimmung einer Anforderungsüberdeckung, bezogen auf eine bestimmte Menge von Tests.

Dynamischer Test - Zusammenhänge

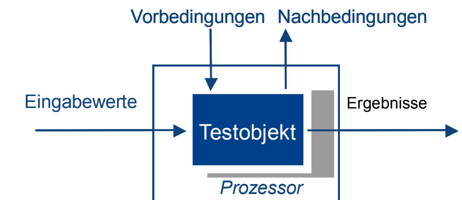


Dynamischer Test – Vorgehensweise

1. **Entwerfen von Tests durch Ermittlung von Testbedingungen**
   - Analyse der dem Test zugrunde liegenden Dokumente, um festzustellen, was getestet werden muss, um also die Testbedingungen festzulegen.
   - Testbedingungen müssen auf Spezifikationen / Anforderungen zurückverfolgbar sein, um Auswirkungen von Änderungen an Spezifikationen / Anforderungen auf Tests (impact analysis) und die Abdeckung von Spezifikationen bzw. Anforderungen durch Tests (Überdeckung) zu ermitteln.
2. **Spezifizieren der Testfälle**
   - Entwicklung und detaillierte Beschreibung der Testfälle und Testdaten unter Verwendung von Testentwurfsverfahren.
   - Vorausgesagte (erwartete) Ergebnisse beinhalten Ausgaben, Änderungen von Daten und Zuständen sowie alle anderen Folgen des Tests – falls diese nicht definiert werden, könnte plausibles, aber fehlerhaftes Ergebnis als richtig angesehen werden (=> vor der Testdurchführung festlegen !).
3. **Spezifizieren der Testablaufspezifikationen**
   - Testfälle in eine ausführbare Reihenfolge bringen. Vor- und Nachbedingungen der Testfälle beachten.
4. **Testausführungsplanung**
   - Reihenfolge angeben, in der die verschiedenen Testablaufspezifikationen bzw. Testskripte (automatisch) ausgeführt werden, und wann und von wem sie auszuführen sind.
   - Faktoren wie Regressionstests (wiederholte Testausführung), Priorisierung und logische Abhängigkeiten zwischen Testfällen bzw. Testskripten berücksichtigen.

Dynamischer Test in »unteren« Teststufen

Ziele des dynamischen Tests:



- Durch stichprobenhafte Programmläufe Nachweis der Erfüllung der festgelegten Anforderungen durch das Testobjekt.
- Aufdeckung von eventuellen Abweichungen und Fehlerwirkungen.
- Dabei mit möglichst wenig Aufwand möglichst viele Anforderungen überprüfen bzw. Fehler nachweisen.

Die Ausführung dynamischer Tests erfordert die Ausführbarkeit des Testobjekts.

- Zur Testdurchführung muss ein ablauffähiges Programm vorliegen !

Einzelne Klassen, Module/Komponenten und Teilsysteme sind jedoch in der Regel für sich alleine nicht lauffähig:

- Bieten oft Operationen/Dienste für andere Softwareeinheiten an.
- Haben kein »Hauptprogramm« zur Koordination des Ablaufes.
- Stützen sich oft auf Operationen/Dienste anderer Softwareeinheiten ab.

In den »unteren« Teststufen Klassen-/Modultest, Komponententest und Integrationstest muss das Testobjekt also in in einen entsprechenden »Testrahmen« eingebettet werden.

Dynamischer Test – Begriffe (4)

Testrahmen (test harness, test bed): Eine Testumgebung, die aus den für die Testausführung benötigten Treibern und Platzhaltern besteht.

Testumgebung: Umgebung, die benötigt wird, um Tests auszuführen. Umfasst Hardware, Instrumentierung, Simulatoren, Softwarewerkzeuge u.a. unterstützende Hilfsmittel [nach IEEE 610].

Platzhalter (stub): Rudimentäre oder spezielle Implementierung einer Softwarekomponente, um eine noch nicht implementierte Komponente zu ersetzen bzw. zu simulieren [nach IEEE 610].

Dummy: Platzhalter mit rudimentärer Funktionalität

Mock: Platzhalter mit erweiterter Funktionalität für Testzwecke (Logging, Plausibilitätsprüfung)

Simulator: Werkzeug, mit dem die Einsatz- bzw. Produktivumgebung nachgebildet wird.

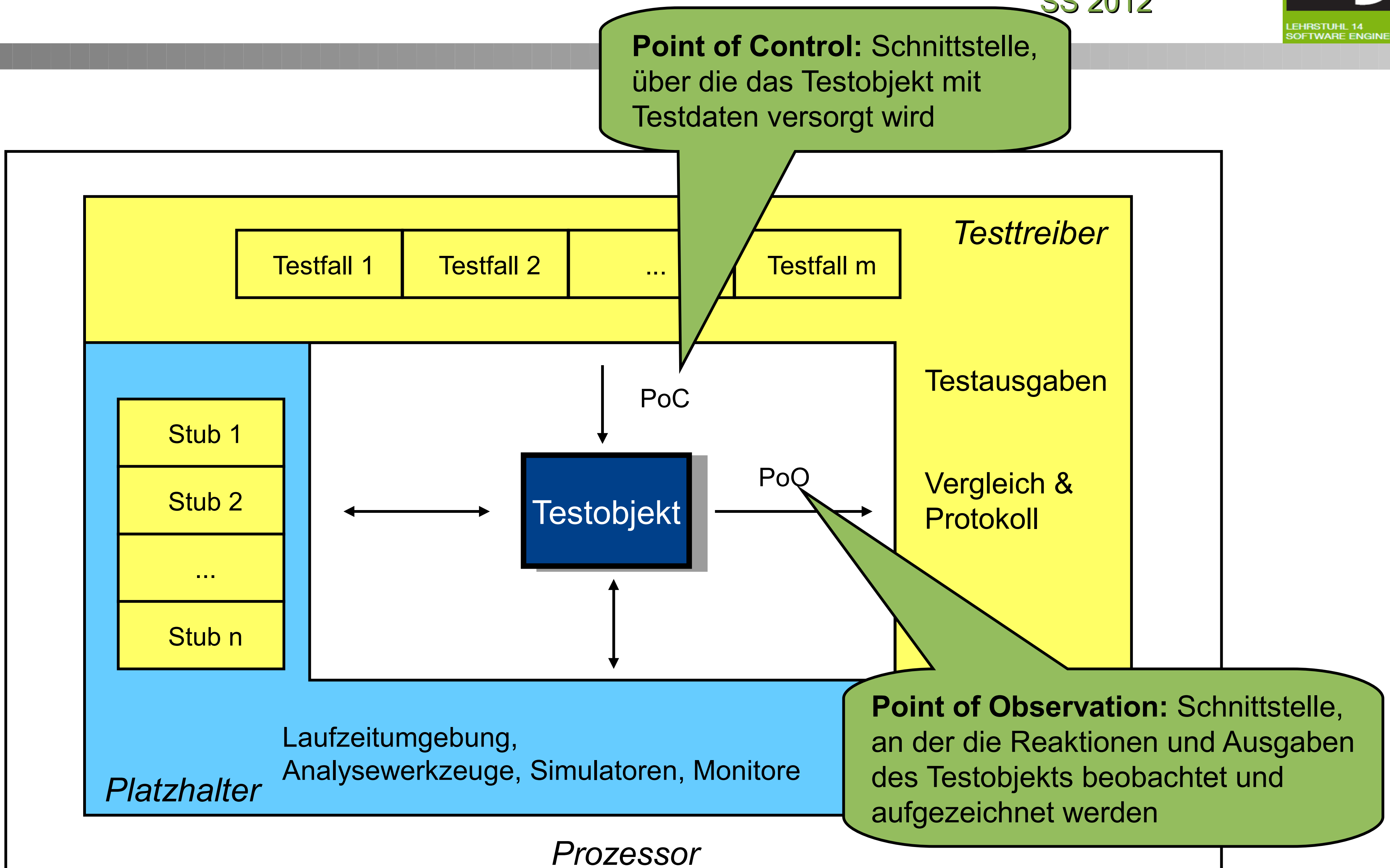
Testentwurfsverfahren: Vorgehensweise, nach der Testfälle abgeleitet oder ausgewählt werden.

Eingangskriterien: Menge der generischen und spezifischen Bedingungen, die es in Prozess ermöglichen, mit bestimmter Aktivität (z.B. Testphase) fortzuschreiten. Zweck ist es, die Durchführung der Aktivität zu verhindern, wenn dafür ein höherer Mehraufwand benötigt (verschwendet) wird als für die Schaffung der Eingangskriterien [Gilb und Graham].

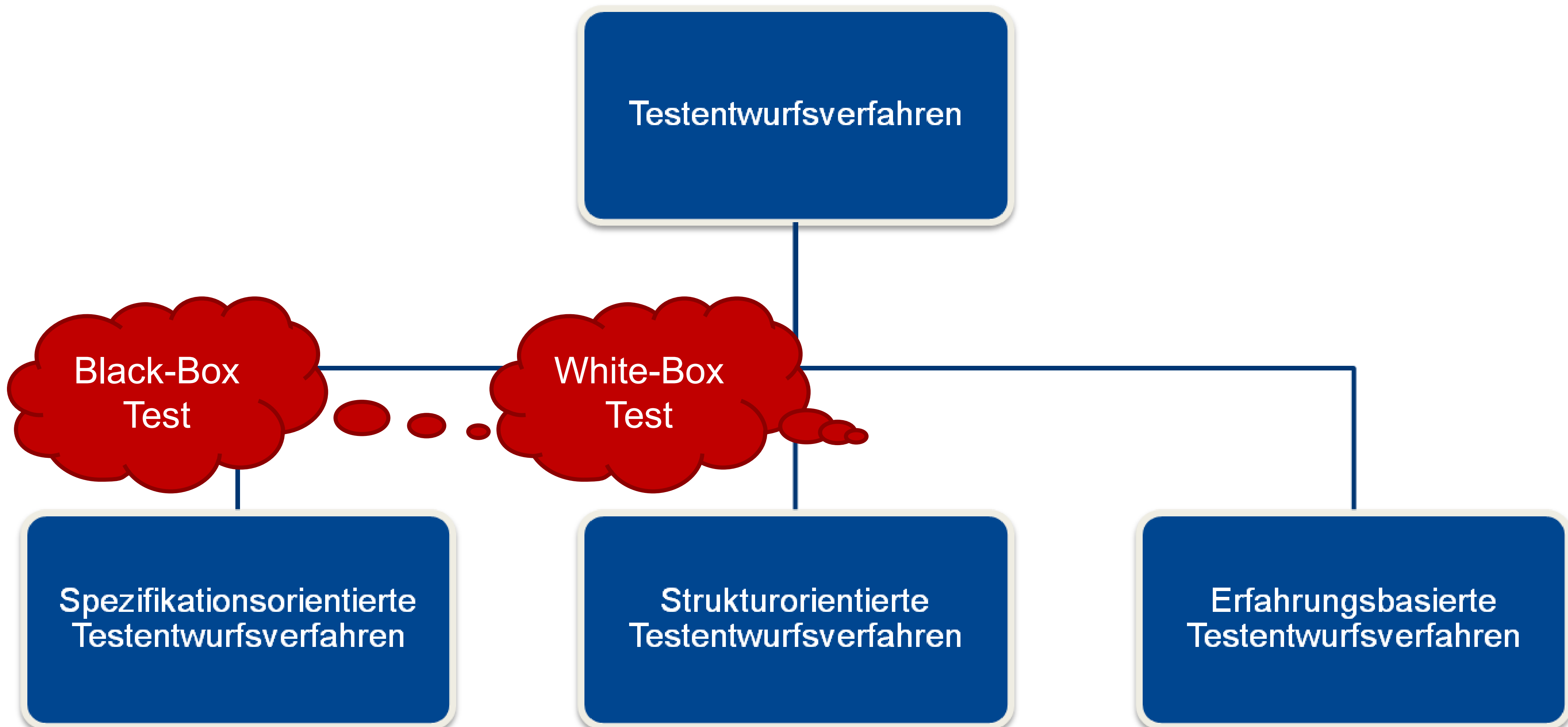
Ausgangskriterien: Die Menge der abgestimmten generischen und spezifischen Bedingungen, die von allen Beteiligten für den Abschluss eines Prozesses akzeptiert wurden. Ausgangsbedingungen für eine Aktivität verhindern es, dass die Aktivität als abgeschlossen betrachtet wird, obwohl Teile noch nicht fertig sind [nach Gilb und Graham].

Ausgangskriterien werden in Berichten referenziert und zur Planung der Beendigung des Testens verwendet.

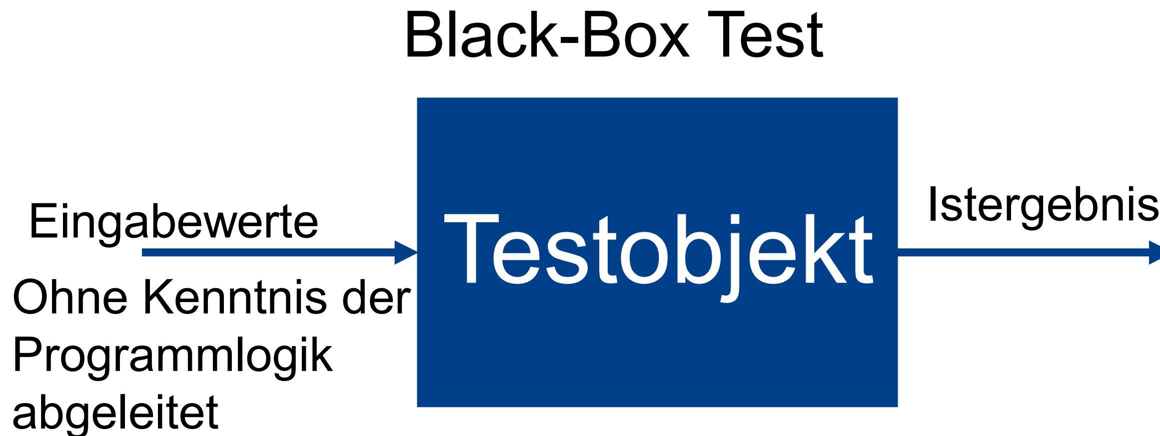
Aufbau eines Testrahmens

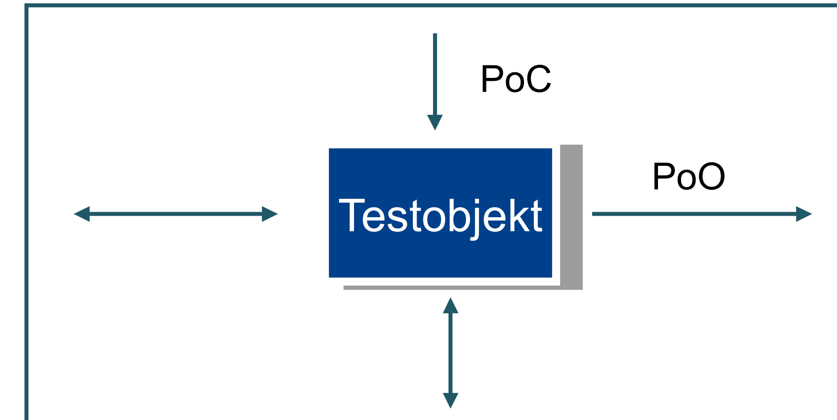


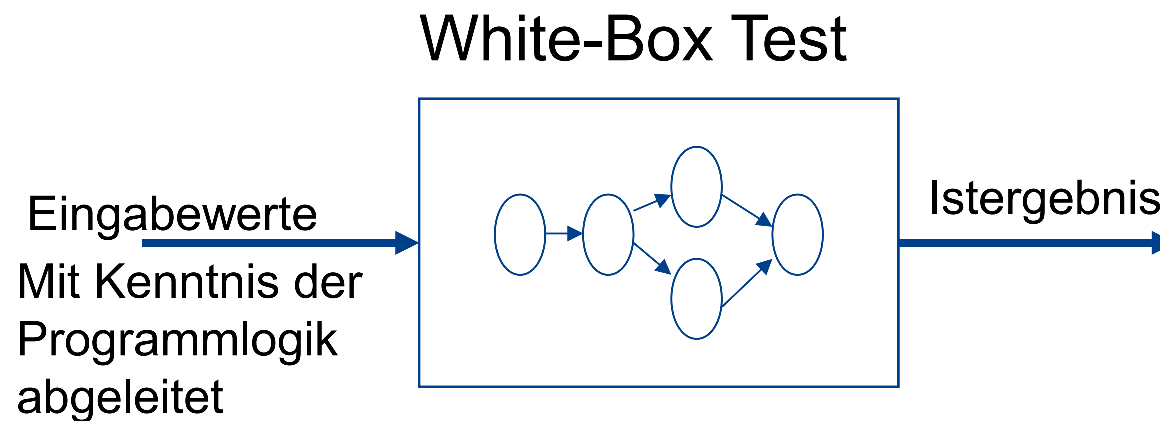
Testentwurfsverfahren

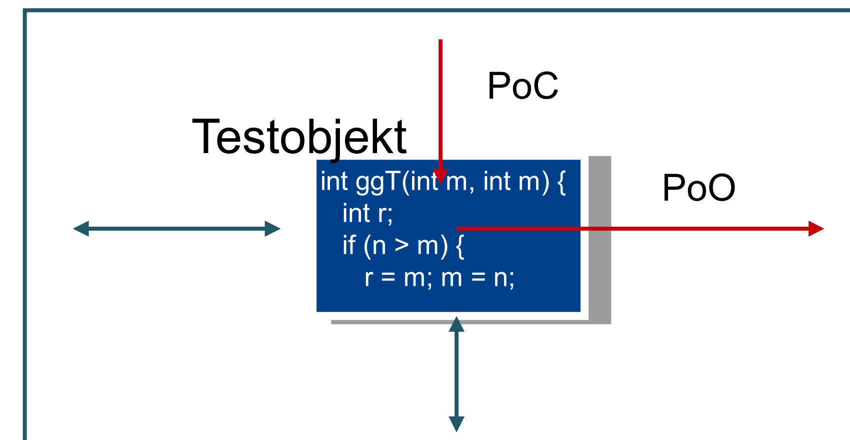


Black-Box Test vs. White-Box Test









Black-Box-Testentwurfsverfahren (Testobjekt »undurchsichtig«)

Synonym: **spezifikationsorientierte Testentwurfsverfahren.**

Keine Informationen über den Programmtext und den inneren Aufbau.

- Beobachtet wird das Verhalten des Testobjekts von außen (PoO - Point of Observation liegt außerhalb des Testobjekts).
- Steuerung des Ablaufs des Testobjektes nur durch die Wahl der Eingabetestdaten (PoC - Point of Control liegt außerhalb des Testobjektes).
- Modelle bzw. Anforderungen an das Testobjekt, ob formal oder nicht formal, werden zur Spezifikation des zu lösenden Problems, der Software oder ihrer Komponente herangezogen.
- Von diesen Modellen können systematisch Testfälle abgeleitet werden.

White-Box-Testentwurfsverfahren (Testobjekt »durchsichtig«)

Synonym: **strukturorientierte Testentwurfsverfahren.**

Testfälle können auf Grund der Programmstruktur des Testobjektes gewonnen werden.

- Informationen über den Aufbau der Software werden für die Ableitung von Testfällen verwendet, beispielsweise der Code und der Algorithmus.
- Überdeckungsgrad des Codes kann für vorhandene Testfälle gemessen werden.
- Weitere Testfälle können zur Erhöhung des Überdeckungsgrades systematisch abgeleitet werden.
- Während der Ausführung der Testfälle wird der innere Ablauf im Testobjekt analysiert (Point of Observation liegt innerhalb des Testobjekts).
- Eingriff in den Ablauf im Testobjekt möglich, z.B. wenn für Negativtests die zu provozierende Fehlbedienung über die Komponentenschnittstelle nicht auslösbar ist (Point of Control kann innerhalb des Testobjekts liegen).

Erfahrungsbasierte Testentwurfserfahren

- Nutzen Wissen und die Erfahrung von Menschen zur Ableitung der Testfälle.
- Wissen von Testern, Entwicklern, Anwendern und Betroffenen über die Software, ihre Verwendung und ihre Umgebung.
- Wissen über wahrscheinliche Fehler und ihre Verteilung.

Spezifikationsorientierter / Black-Box-Test: Begriffe

Funktionaler Test

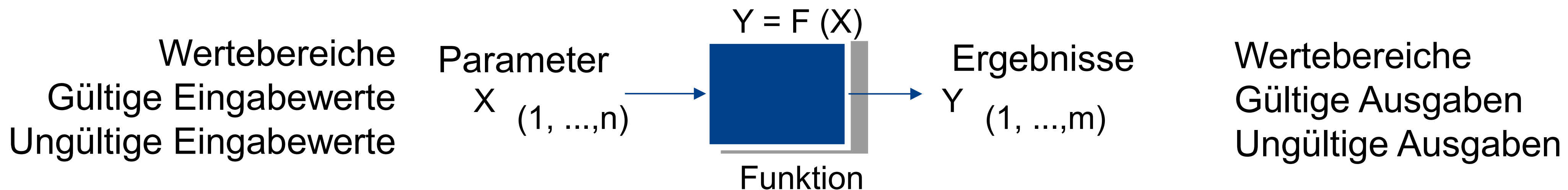
- Dynamischer Test, bei dem die Testfälle unter Verwendung der funktionalen Spezifikation des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand der funktionalen Spezifikation bewertet wird.

Funktionalität

- Die Fähigkeit eines Softwareprodukts beim Einsatz unter spezifizierten Bedingungen Funktionen zu liefern, die festgelegte und vorausgesetzte Erfordernisse erfüllen [ISO 9126].

  Untermerkmale der Funktionalität nach ISO 9126 sind: Angemessenheit, Richtigkeit, Interoperabilität, Sicherheit und Konformität

ISO/IEC 9126:200x: Bewerten von Softwareprodukten, Qualitätsmerkmale und Leitfaden zu ihrer Verwendung



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

4.4 Black-Box-Test

- Dynamischer Test – Grundlagen
- Idee der Black-Box-Testentwurfsverfahren
- **Äquivalenzklassenbildung**
- Grenzwertanalyse
- Zustandsbasierter Test
- Entscheidungstabellentest
- Weitere Black-Box-Testentwurfsverfahren

Dynamischer Test - Black-Box

Dynamischer Test - Black-Box

Black-Box-Testentwurfsverfahren

Äquivalenz-klassenbildung

Grenzwertanalyse

Zustandsbasierter Test

Anwendungsfallbasierter Test (Use Case Test)

Ursache-Wirkungsgraphen

Syntaxtest

Zufallstest und Smoke-Test

Klassifikationsbäume

Entscheidungstabellentest

Idee der Black-Box-Testentwurfsverfahren

Grundlagen

Begriffe

Testfall

Testablauf

Testausführungsplan

Testablaufspezifikation

Testbedingung

Testentwurf

Testentwurfsspezifikation

Testfallspezifikation

Vorgehensweise beim dynamischen Test

Ermittlung von Testbedingungen

Spezifizieren der Testfälle

Spezifizieren der Testablaufspezifikationen

Testausführungsplanung

Dynamischer Test in »unteren« Teststufen

Testrahmen

Testumgebung

Testtreiber

Dummy

Platzhalter

Point of Observation

Point of Control

Testentwurfsverfahren

spezifikationsorientierte Testentwurfserfahren (Black-Box)

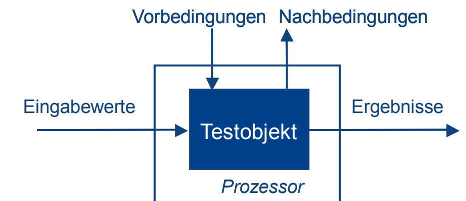
erfahrungsbasierte Testentwurfserfahren

strukturorientierte« Testentwurfserfahren (White-Box)

Black-Box Test vs. White-Box Test

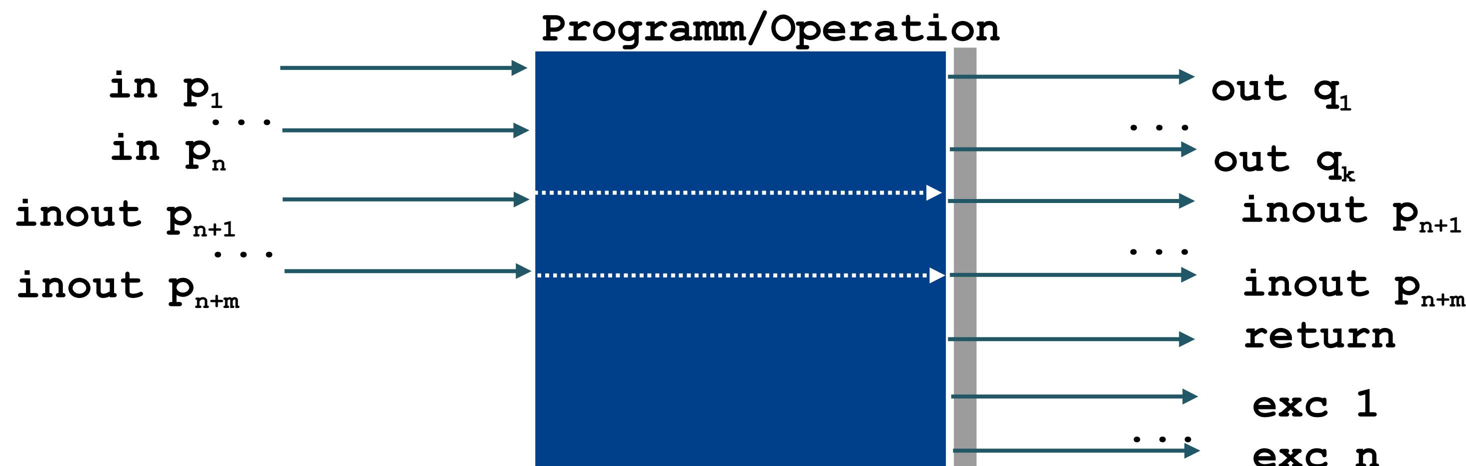
Programme und Operationen

Vereinfachende Annahme: Programme / Operationen berechnen Ausgaben aus Eingaben (zustandslos – frühere Ausführungen spielen keine Rolle).

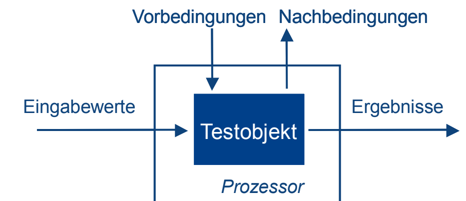


Signatur einer Operation:

- 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 (*greatest common divisor, gcd*):

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

Logische Spezifikation:

ggT: Z x Z → IN
ggT(0,0) = 0 ∧
[m<>0∨n<>0 => ggT(m,n)|m ∧ ggT(m,n)|n ∧ ∀o∈IN: o>ggT(m,n) ⇒ (¬(o|m) ∨ ¬(o|n))]

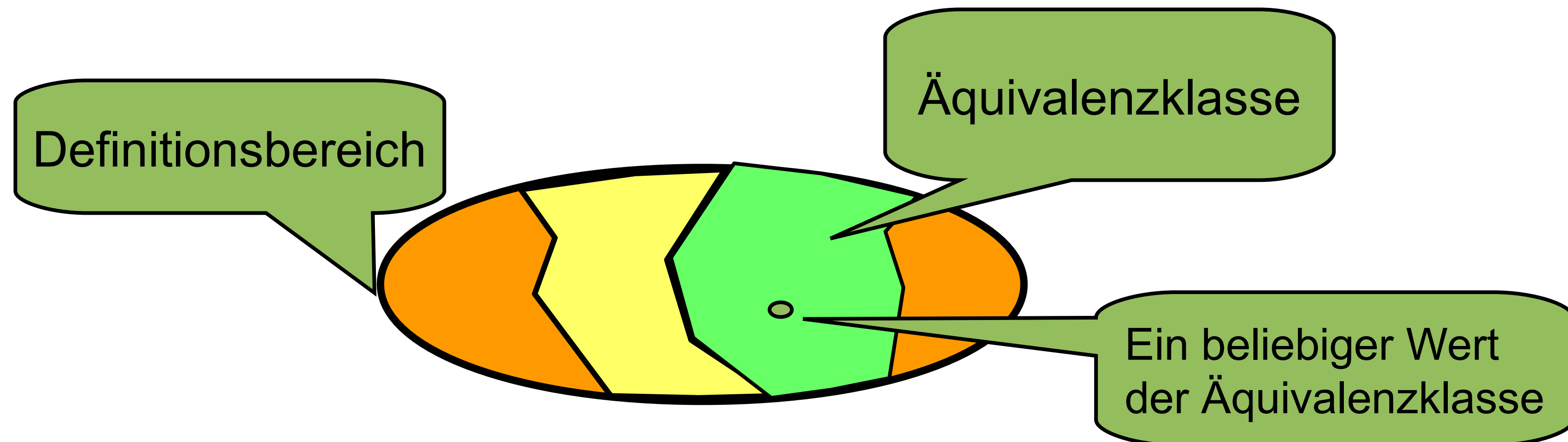
Spezifikation in UML / Java:

```
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)
… )
```

Grundidee der Äquivalenzbildung

Die Definitionsbereiche der Ein- und Ausgaben werden so in **Äquivalenzklassen (ÄK)** zerlegt (partitioniert), dass alle Werte einer Klasse äquivalentes Verhalten des Prüflings ergeben sollten. Die Wahl nur eines Testwertes pro ÄK (Repräsentant) ergibt dann eine sinnvolle Stichprobe.

- Wenn ein Wert der ÄK einen Fehler aufdeckt, wird erhofft dass auch jeder andere Wert der ÄK diesen Fehler aufdeckt.
- Wenn ein Wert der ÄK keinen Fehler aufdeckt, wird erhofft, dass auch kein anderer Wert der ÄK einen Fehler aufdeckt.



Äquivalenzklassen für ggT: Erster Schritt

Definitionsbereiche der Ein- und Ausgaben

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

Gültige von ungültigen Teilbereichen der Java-Implementierung unterscheiden:

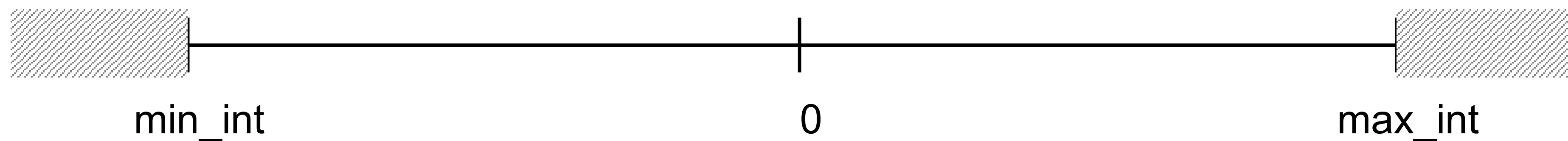
?

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 der Spezifikation (Ein- und Ausgaben !).

Äquivalenzklassenbildung für jede weitere Beschränkung:

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

Äquivalenzklassenbildung: Beispiel Wertebereich

Falls eine Beschränkung **einen Wertebereich** (Intervall) spezifiziert: 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 ?
Gültige Äquivalenzklasse ?
Ungültige Äquivalenzklasse ?

Äquivalenzklassenbildung: Beispiel Wertebereich

Falls eine Beschränkung **einen Wertebereich** (Intervall) spezifiziert: 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

Falls eine Beschränkung eine **minimale und maximale** Anzahl von Werten spezifiziert: 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 ?
Ungültige Äquivalenzklasse ?

Äquivalenzklassenbildung: Beispiel minimal / maximal

Falls eine Beschränkung eine **minimale und maximale** Anzahl von Werten spezifiziert: 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

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

Beispiel

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

Gültige Äquivalenzklasse: ?

Ungültige Äquivalenzklasse: ?

Äquivalenzklassenbildung: Beispiel Menge von Werten

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

Äquivalenzklassenbildung: Beispiel Situation

Falls eine Beschränkung **eine Situation** spezifiziert, die zwingend 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: ?

Ungültige Äquivalenzklasse: ?

Äquivalenzklassenbildung: Beispiel Situation

Falls eine Beschränkung **eine Situation** spezifiziert, die zwingend 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).

Frage

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 ?

Frage

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

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 m_i Äquivalenzklassen (i=1..n) gibt es:

$\prod_{i=1 \ldots n} m_i$ unterschiedliche Kombinationen (Testfälle)

Anzahl der Testfälle minimieren: Heuristiken

Testfälle aus allen Repräsentanten kombinieren und anschließend nach »Häufigkeit« sortieren (»**Benutzungsprofile**«).

- Testfälle dann in dieser Reihenfolge priorisieren.
- Nur mit »benutzungsrelevanten« Testfällen testen.
- Testfälle bevorzugen, die Grenzwerte oder Grenzwert-Kombinationen enthalten.

Sicherstellen, dass jeder Repräsentant einer Äquivalenzklasse mit jedem Repräsentanten jeder anderen Äquivalenzklasse in einem Testfall zur Ausführung kommt.

- D.h. paarweise Kombination statt vollständiger Kombination.

Minimalkriterium: Mindestens ein Repräsentant jeder Äquivalenzklasse in mindestens einem Testfall.

Repräsentanten ungültiger Äquivalenzklassen nicht mit Repräsentanten anderer ungültiger Äquivalenzklassen kombinieren.

Äquivalenzklassenbildung: Fehlermaskierung

Bei gleichzeitiger Behandlung verschiedener ungültiger ÄK bleiben bestimmte Fehler evtl. unentdeckt !

Beispiel:

Eingabebereich

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

Äquivalenzklassen

```
wert_gÄK1:  ?
wert_uÄK1:  ?
wert_uÄK2:  ?
farbe_gÄK1: ?
farbe_uÄK1: ?
```

Testdaten:

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

⇒ Welche Fehler werden evt. übersehen ?

Äquivalenzklassenbildung: Fehlermaskierung

Bei gleichzeitiger Behandlung verschiedener ungültiger ÄK bleiben 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)

Ein spezifisches Ausgangskriterium für den Test nach der Äquivalenzklassenbildung lässt sich anhand der durchgeführten Tests der Repräsentanten der jeweiligen Äquivalenzklassen im Verhältnis zur Gesamtzahl aller definierten Äquivalenzklassen festlegen:

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

Beispiel: Sind 18 Äquivalenzklassen aus den Anforderungen bzw. der Spezifikation für ein Eingabedatum ermittelt worden und sind von diesen 18 nur 15 in den Testfällen getestet worden, so ist eine **Äquivalenzklassen-Überdeckung** von ca. 83 % erreicht:

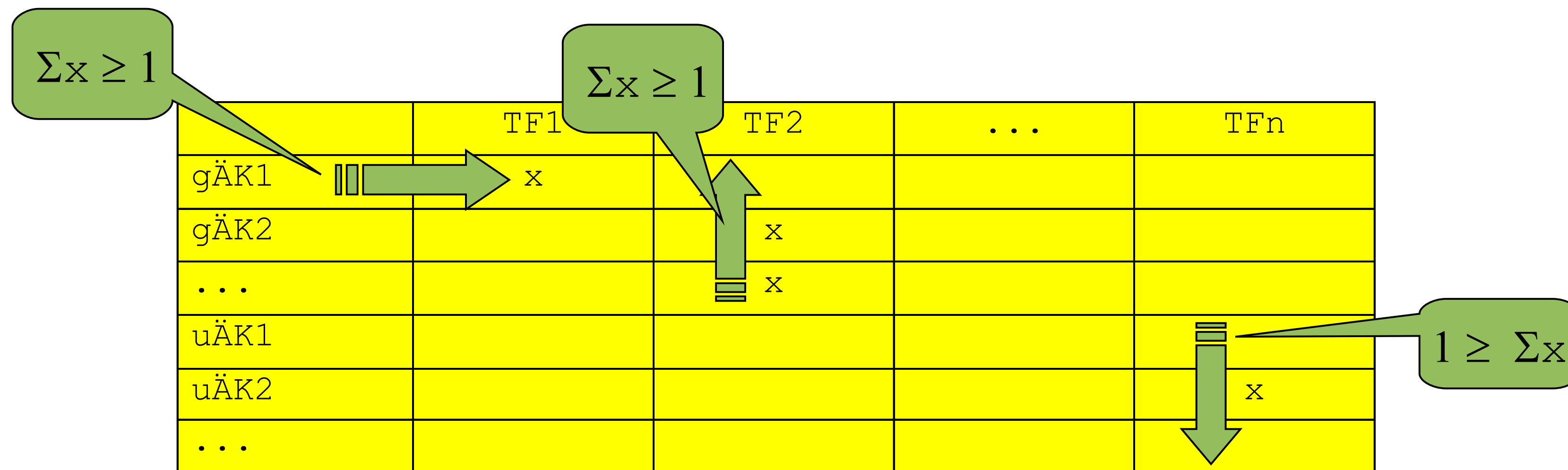
- ÄK-Überdeckung = 15 / 18 ≈ 83 %

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 !).



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

Äquivalenzklassen, Testfälle und Testdaten für ggT


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

Äquivalenzklassen für Eingabeparameter

n, m *(analog)*: `int`

- `gÄKx_1 : ?`
- `gÄKx_2 : ?`
- `gÄKx_3 : ?`
- `uÄKx_1 : ?`
- `uÄKx_2 : ?`

Testfälle:

TF1 : {n = ?, m = ?; ggT = ?}
TF2 : {n = ?, m = ?; ggT = ?}
TF3 : {n = ?, m = ?; ggT = ?}
TF4 : {n = ?, m = ?; ggT = ?}
TF5 : {n = ?, m = ?; ggT = ?}
TF6 : {n = ?, m = ?; ggT = ?}
TF7 : {n = ?, m = ?; ggT = ?}

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

Nachteile:

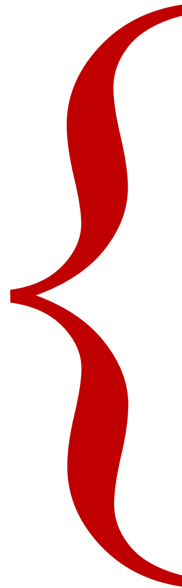
- Betrachtet Bedingungen für einzelne Ein- oder Ausgabeparameter.
- Beachtung von Wechselwirkungen und Abhängigkeiten von Bedingungen sehr aufwändig.

Empfehlung:

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

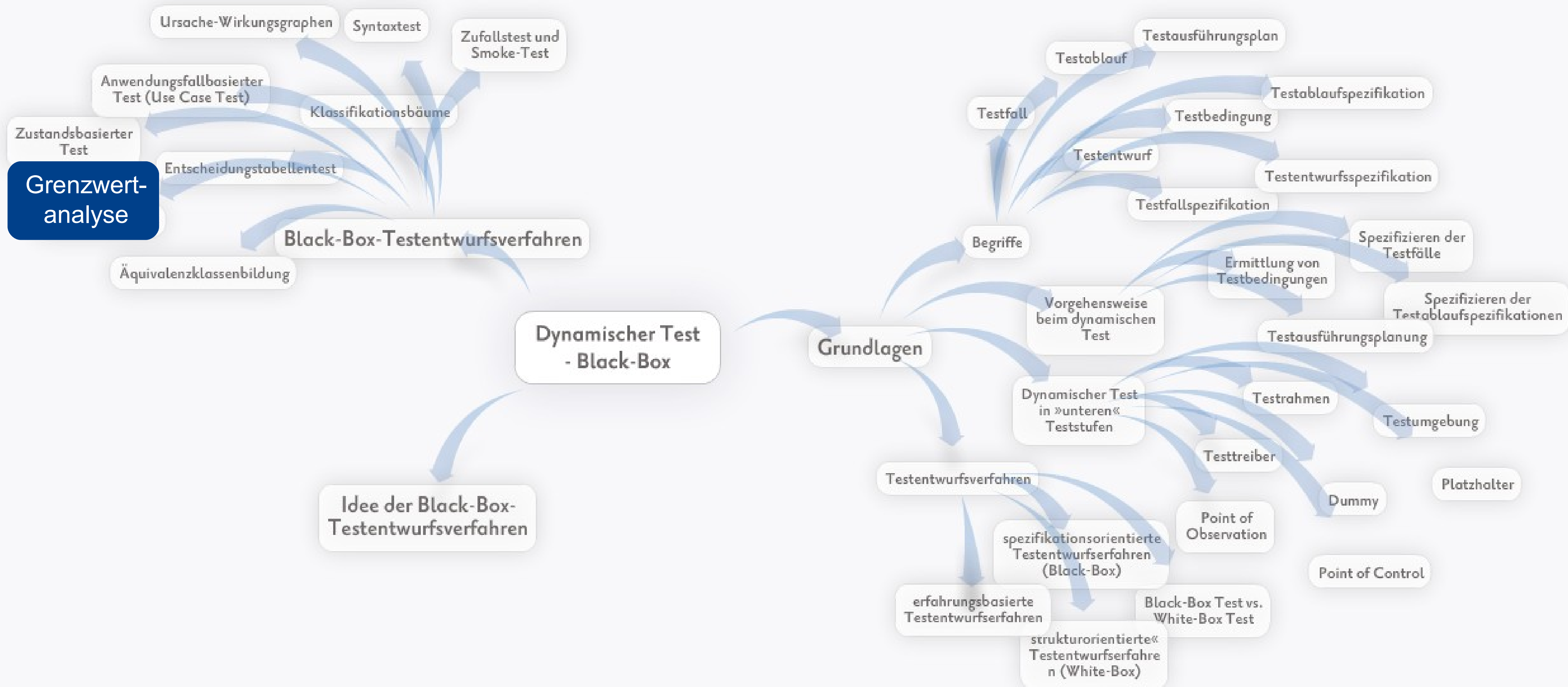
4.4 Black-Box-Test

4.4 Black-Box-Test



- Dynamischer Test – Grundlagen
- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- Grenzwertanalyse
- Zustandsbasierter Test
- Entscheidungstabellentest
- Weitere Black-Box-Testentwurfsverfahren

Dynamischer Test - Black-Box



Grenzwertanalyse: Idee

Idee: In **Verzweigungs- und Schleifenbedingungen** gibt es oft **Grenzbereiche**, für welche die Bedingung gerade noch zutrifft (oder gerade nicht mehr).

- Solche Fallunterscheidungen sind **fehlerträchtig** (*off by one*).
- Testdaten, die solche Grenzwerte prüfen, decken Fehlerwirkungen mit höherer Wahrscheinlichkeit auf als Testdaten, die dies nicht tun.

Beste Erfolge bei Kombination mit anderen Verfahren.

Bei Kombination mit der **Äquivalenzklassenbildung**:

- **Grenzen der ÄK** (größte und kleinste Werte) testen.
- Jeder »Rand« einer ÄK muss in einer Testdatenkombination vorkommen.

Grenzwertanalyse: Anwendung

In der Regel werden der Grenzwert selbst sowie die Werte unmittelbar über bzw. unter dem Grenzwert getestet.

Atomare (geordnete) Bereiche (integer, real, char):

- Werte auf den Grenzen,
- Werte »rechts bzw. links neben« den Grenzen (ungültige Werte, kleiner bzw. größer als Grenze).

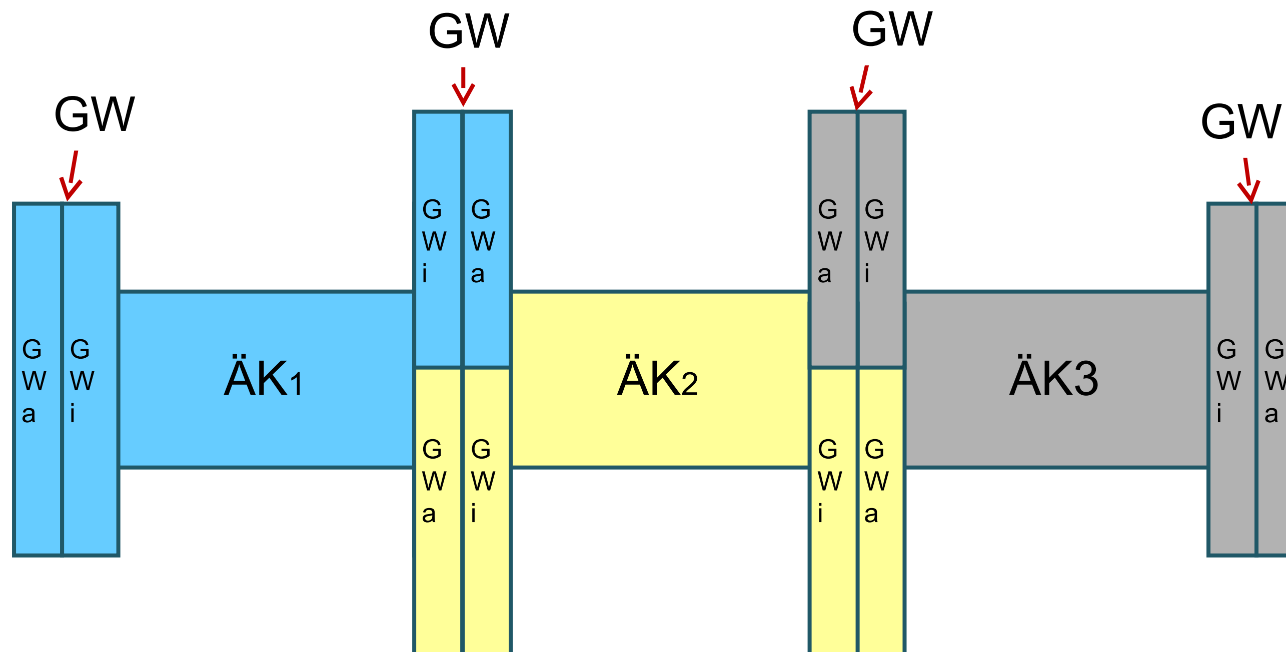
Mengenwertige Bereiche (z.B. bei Datenstrukturen, Beziehungen):

- 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: Innere / äußere Grenzwerte

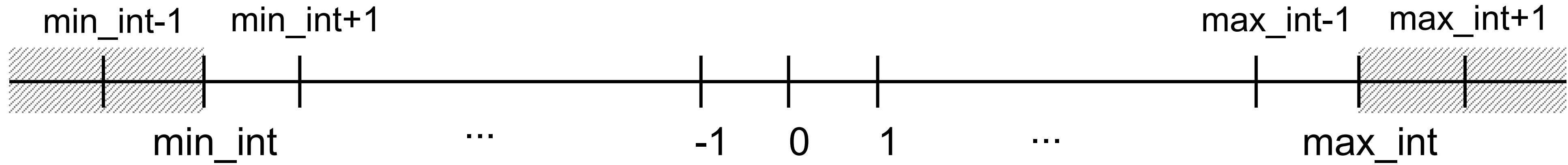
Zusammenfallen der entsprechenden Grenzwerte **benachbarter Äquivalenzklassen**:



An den Grenzen immer zwei Tests, egal ob auf bzw. nur vor oder nach der Grenze getestet werden kann.

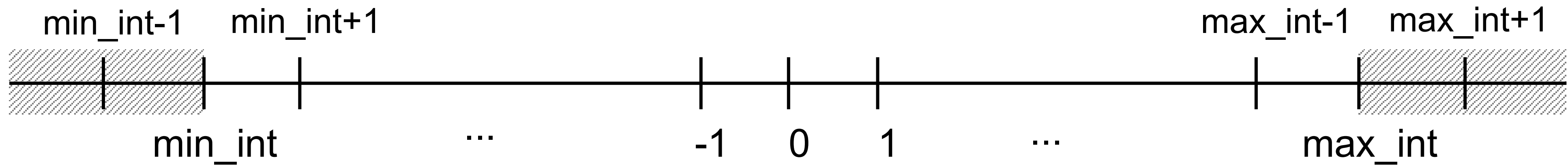
ÄK - Äquivalenzklasse
GW – Grenzwert:
i - innerhalb der ÄK
a - außerhalb der ÄK

Beispiele zur Grenzwertanalyse



Datentyp	Grenzen	Größer	Kleiner
integer	?	?	?
char[5]	?	?	?
double	?	?	?

Beispiele zur Grenzwertanalyse



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

Grenzwertanalyse: Ausgangskriterium (Testendekriterium)

In Analogie zum Ausgangskriterium der Äquivalenzklassenbildung lässt sich auch eine anzustrebende **Überdeckung der Grenzwerte (GW)** vorab festlegen und nach der Durchführung der Tests berechnen:

GW-Überdeckungsgrad = Anzahl getestete GW / Gesamtzahl GW

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** (z.B. sequentielle Datei, lineare Liste, Tabelle).
- **Komplexe Datenstrukturen: leere Mengen testen** (z.B. leere Liste, Null-Matrix).
- **Bei numerischen Berechnungen**: eng zusammen und weit auseinander liegende Werte wählen.

Grenzwertanalyse: Vor- und Nachteile

Vorteile:

- An den Grenzen von Äquivalenzklassen sind häufiger Fehler zu finden als innerhalb dieser Klassen.
- »Die Grenzwertanalyse ist bei richtiger Anwendung eine der nützlichsten Methoden für den Testfallentwurf.« Myers, Glenford J.: Methodisches Testen von Programmen, Oldenbourg, 2001 (7. Auflage)
- Effiziente Kombination mit anderen Verfahren, die Freiheitsgrade in der Wahl der Testdaten lassen.

Nachteile:

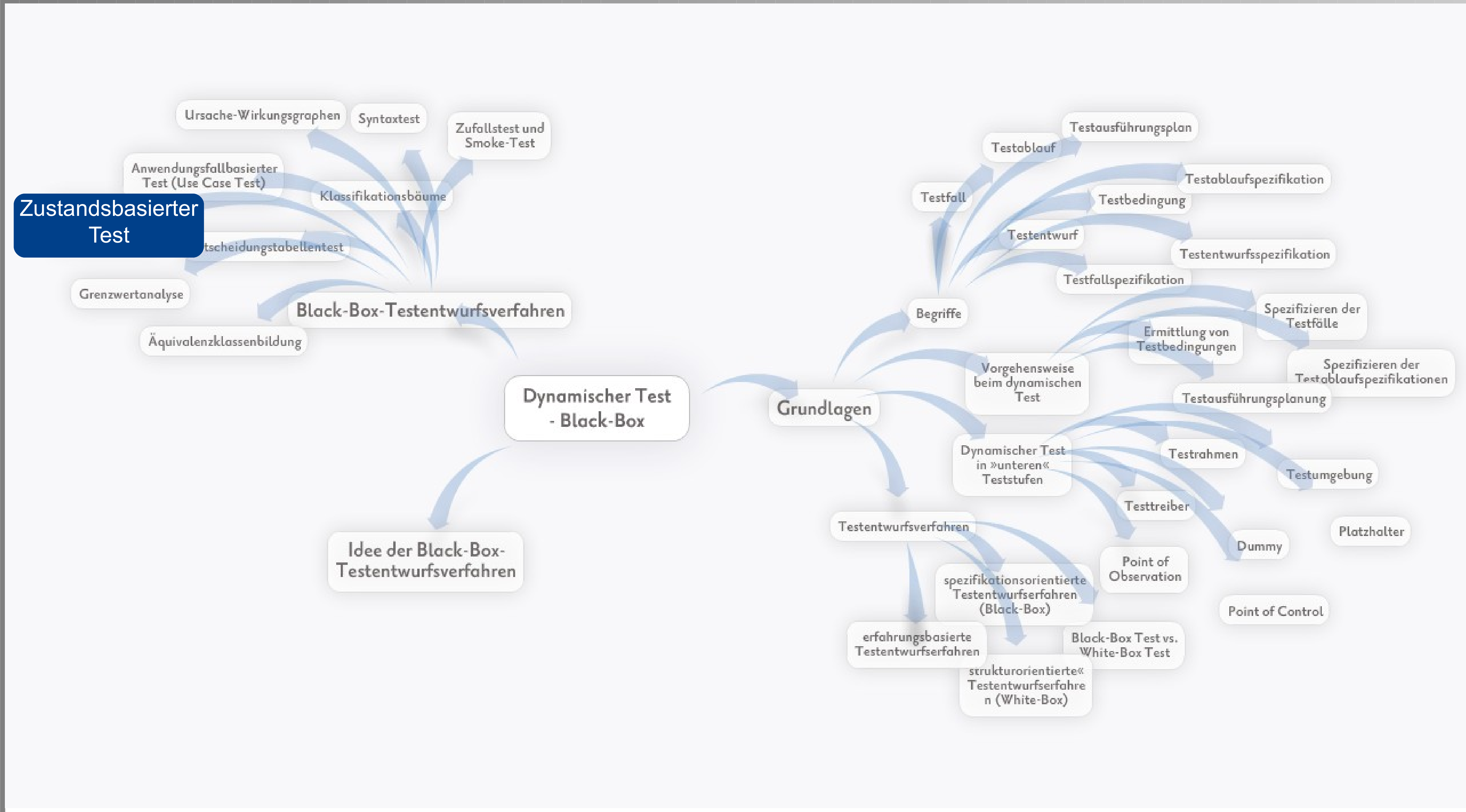
- Rezepte für die Auswahl von Testdaten schwierig anzugeben.
- Bestimmung aller relevanten Grenzwerte schwierig.
- Kreativität zur Findung erfolgreicher Testdaten gefordert.
- Oft nicht effizient genug angewendet, da sie zu einfach erscheint.

4.4 Black-Box-Test

4.4 Black-Box-Test

- Dynamischer Test – Grundlagen
- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- Grenzwertanalyse
- **Zustandsbasierter Test**
- Entscheidungstabellentest
- Weitere Black-Box-Testentwurfsverfahren

Dynamischer Test – Black-Box



Zustandsbasierter Test

Bei vielen Systemen hat neben Eingabe-Werten auch der bisherige Ablauf des Systems Einfluss auf die Berechnung der Ausgaben bzw. auf das Systemverhalten.

- Ein »endlicher Automat« oder ein Zustandsautomat (*finite automaton, finite state machine, sequential machine*) besteht aus einer endlichen Anzahl von internen Konfigurationen – **Zustände** genannt.
- Der Zustand eines Systems beinhaltet implizit die Informationen, die sich aus den bisherigen Eingaben ergeben haben und die benötigt werden, um die Reaktion des Systems auf noch folgende Eingaben zu bestimmen.

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

Das System oder Testobjekt kann beginnend von einem Startzustand unterschiedliche Zustände annehmen.

- Zustandsänderungen oder –übergänge werden durch Ereignisse, zum Beispiel Funktionsaufrufe, ausgelöst.
- Bei den Zustandsänderungen können Aktionen durchzuführen sein.
- Neben dem Startzustand gibt es einen weiteren speziellen Zustand, den Endzustand (oder mehrere, verschiedene Endzustände).

Zustandsbasierter Test: Begriffe

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

Zustandsübergang: Ein Übergang zwischen zwei Zuständen einer Komponente oder eines Systems.

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

Zustandsübergangstabelle: Eine Tabelle, die für jeden Zustand in Verbindung mit jedem möglichen Ereignis die resultierenden Übergänge darstellt. Das können sowohl gültige als auch ungültige Übergänge sein.

Zustandsbasierter Test: Ein Black-Box-Testentwurfsverfahren, mit dem Testfälle entworfen werden, um gültige und ungültige Zustandsübergänge zu prüfen.

Beispiel zur Zustandsmodellierung: Stapel (Stack)

Klasse `Stapel`

Zustandserhaltende Operationen

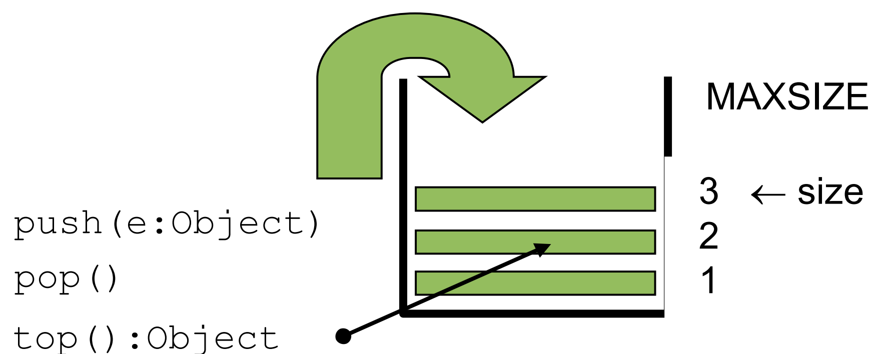
```
size():integer; // Anzahl gestapelter Elemente
MAX():integer;  // Maximale Anzahl
top():Object;   // Zeiger auf oberstes Element
```

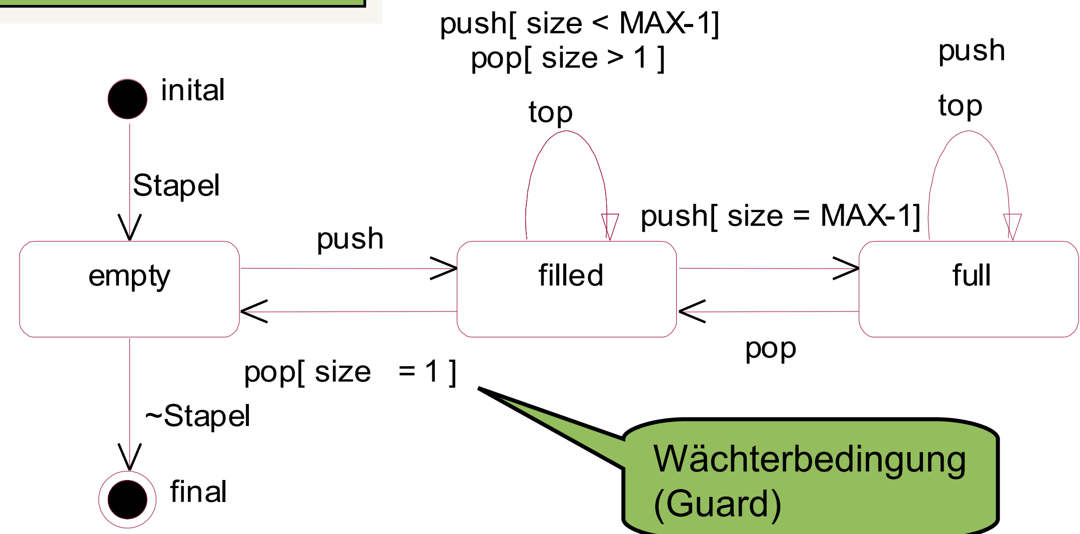
Zustandsverändernde Operationen

```
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, dass sich das Testobjekt konform zum Zustandsdiagramm verhält (**Zustands-Konformanztest**).
- Zusätzlich Test unter nicht-konformanten Benutzungen (**Zustands-Robustheitstest**).

Arbeitsschritte des zustandsbasierten Tests

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:

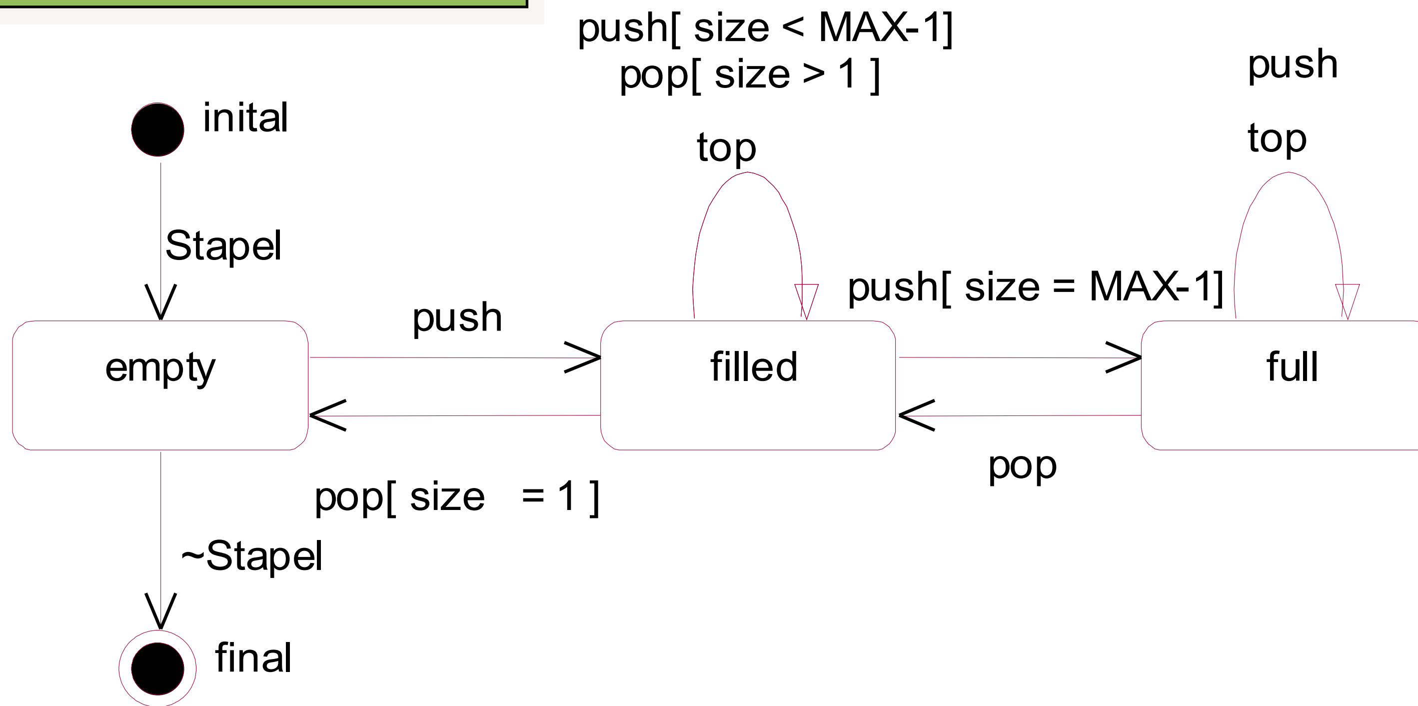
```
empty:  size() = 0;
filled: 0 < size() < MAX();
full:   size() = MAX();
```

Zwei »Pseudo-Zustände«:

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

Acht Zustandsübergänge:

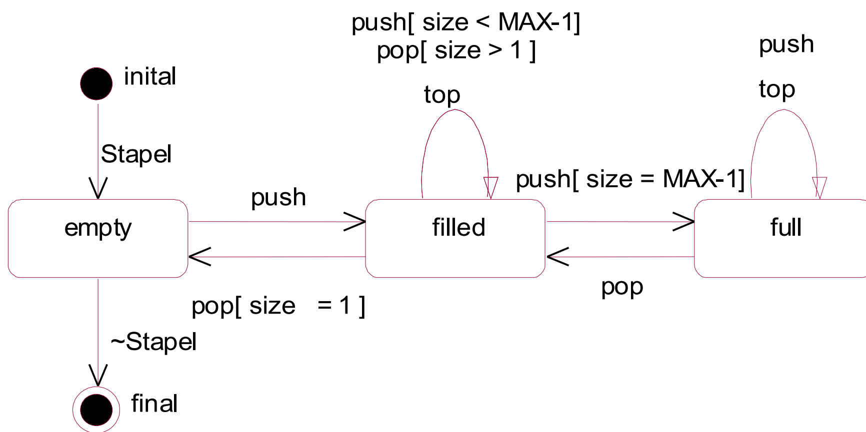
```
initial → empty; empty → final
empty → filled; filled → empty (Zyklus!)
filled → full; full → filled   (Zyklus!)
filled → filled; full → full   (Zyklen!)
```



2. Prüfung auf Vollständigkeit

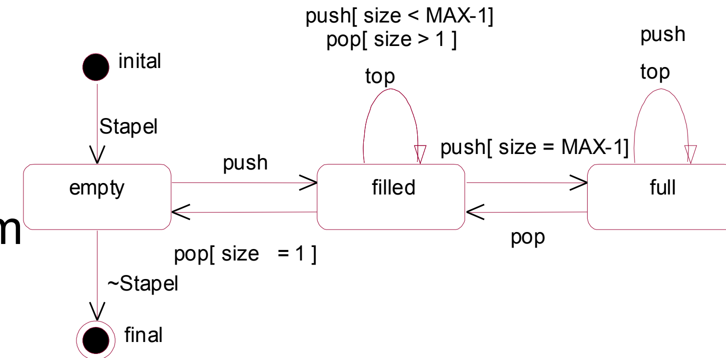
Zustandsdiagramm hinsichtlich der »**Vollständigkeit**« untersuchen:

- Ggf. Zustandsübergangstabelle anlegen.
- Ggf. auch die 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

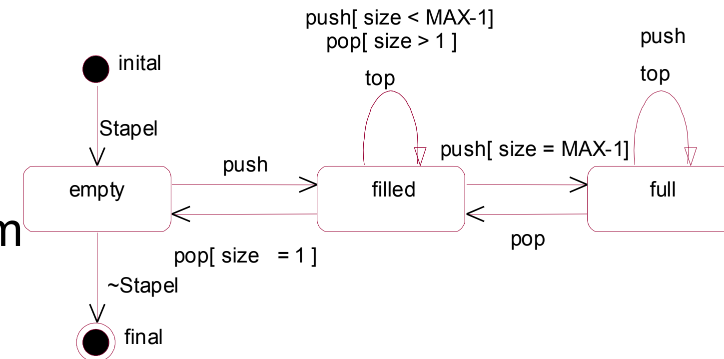
3. Aufbau des Übergangsbaumes: Zustands-Konformanztest



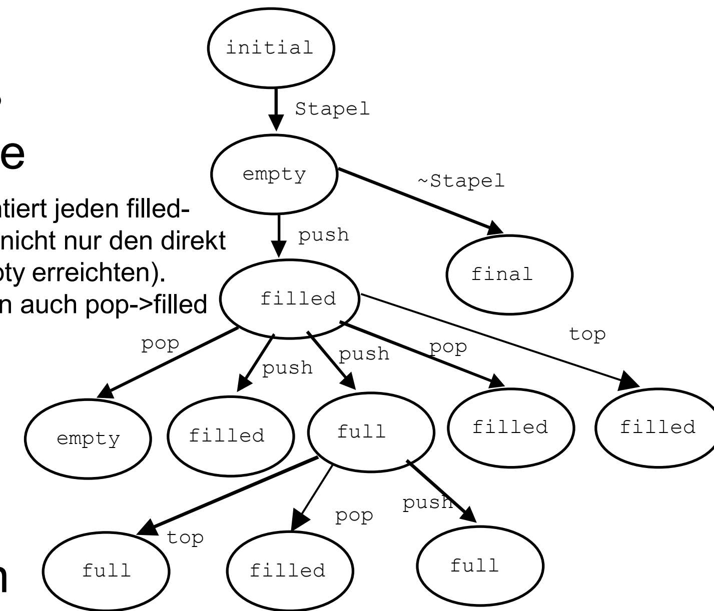
1. Der Anfangszustand wird die **Wurzel** des Baumes.
2. Für jeden möglichen **Übergang** vom Anfangszustand zu einem Folgezustand im Zustandsdiagramm erhält der Übergangsbaum von der Wurzel aus einen **Zweig** zu einem **Knoten**, der den **Nachfolgezustand** repräsentiert. Am Zweig wird das Ereignis (Operation) und ggf. die Wächterbedingung notiert.
3. Der letzte Schritt wird für jedes Blatt des Übergangs-baums (anstelle des Anfangszustands) so lange wiederholt, bis eine der beiden **Endbedingungen** eintritt:
   - Der dem Blatt entsprechende Zustand ist auf einer »höheren Ebene« bereits einmal im Baum enthalten.
   - Der dem Blatt entsprechende Zustand ist ein Endzustand und hat somit keine weiteren Übergänge, die zu berücksichtigen wären.
   [Dabei wird jedes Blatt unabhängig von der davor liegenden Historie betrachtet.]

?

1. Der Anfangszustand wird die **Wurzel** des Baumes.
2. Für jeden möglichen **Übergang** vom Anfangszustand zu einem Folgezustand im Zustandsdiagramm erhält der Übergangsbaum von der Wurzel aus einen **Zweig** zu einem **Knoten**, der den **Nachfolgezustand** repräsentiert. Am Zweig wird das Ereignis (Operation) und ggf. die Wächterbedingung notiert.
3. Der letzte Schritt wird für jedes Blatt des Übergangs-baums (anstelle des Anfangszustands) so lange wiederholt, bis eine der beiden **Endbedingungen** eintritt:
   - Der dem Blatt entsprechende Zustand ist auf einer »höheren Ebene« bereits einmal im Baum enthalten.
   - Der dem Blatt entsprechende Zustand ist ein Endzustand und hat somit keine weiteren Übergänge, die zu berücksichtigen wären.
   [Dabei wird jedes Blatt unabhängig von der davor liegenden Historie betrachtet.]



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

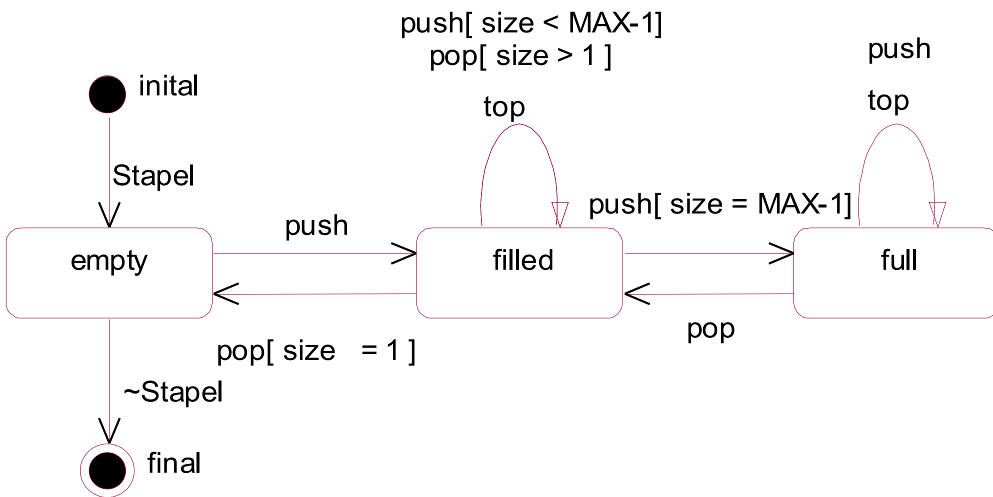


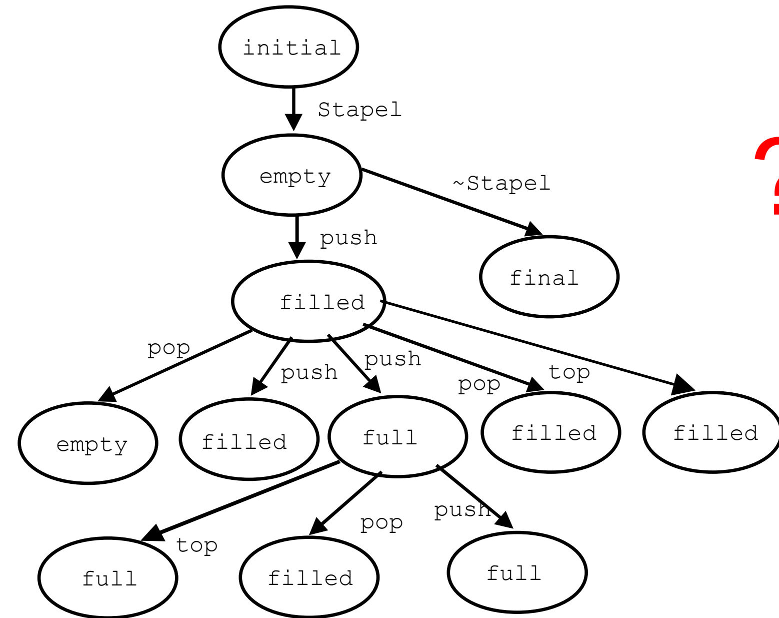
(Wächterbedingungen hier nicht dargestellt)

4. Erweitern des Übergangsbaumes: Zustands-Robustheitstest

Robustheit unter spezifikationsverletzenden Benutzungen prüfen.

Für Botschaften, für die aus dem betrachteten Knoten kein Übergang spezifiziert ist, den Übergangsbaum um einen neuen »Fehler«-Zustand erweitern.





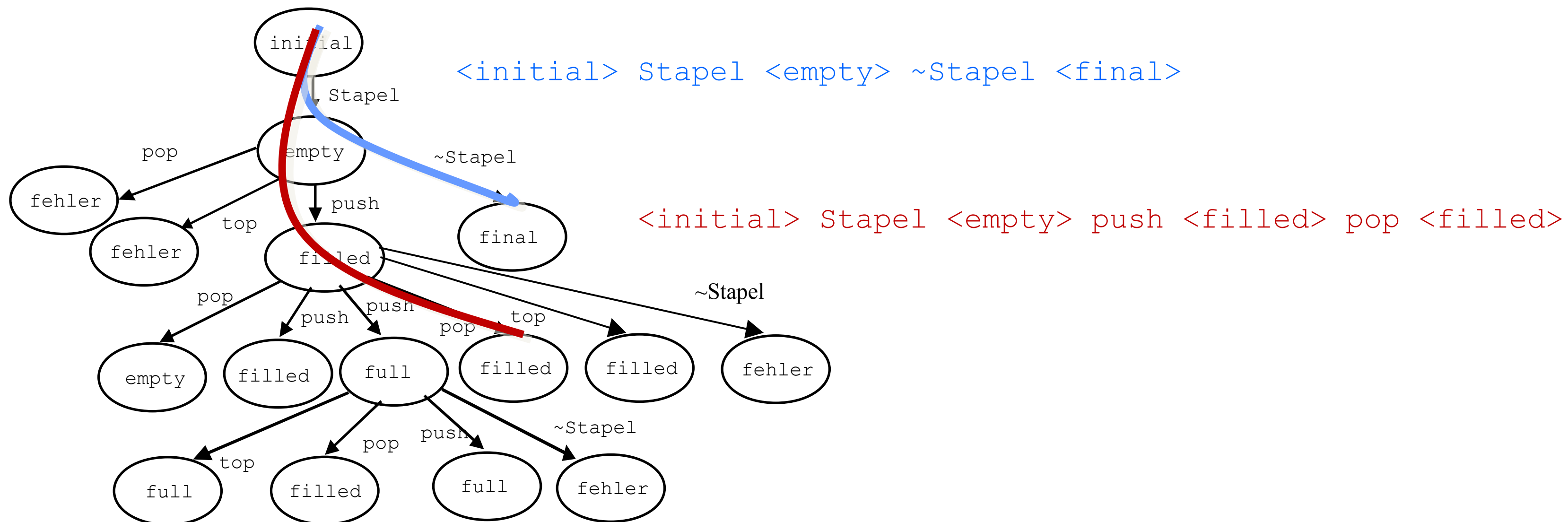
4. Erweitern des Übergangsbaumes: Zustands-Robustheitstest

Robustheit unter spezifikationsverletzenden Benutzungen prüfen.

Für Botschaften, für die aus dem betrachteten Knoten kein Übergang spezifiziert ist, den Übergangsbaum um einen neuen »Fehler«-Zustand erweitern.

5. Generieren der Testfälle (1)

- Pfade von der Wurzel zu Blättern im erweiterten Übergangsbaum als Funktions-Sequenzen auffassen.
- Stimulierung des Testobjekts mit den entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab (aber nicht unbedingt alle möglichen Variablenbelegungen => nicht der komplette theoretisch mögliche Zustandsraum; wäre i.A. ja auch unpraktikabel).
- Parameter ergänzen.



5. Generieren der Testfälle (2)

- Stimulierung des Testobjekts mit den entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab (aber nicht unbedingt alle möglichen Variablenbelegungen => nicht der komplette theoretisch mögliche Zustandsraum; wäre i.A. ja auch unpraktikabel).
- 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 den entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab (aber nicht unbedingt alle möglichen Variablenbelegungen => nicht der komplette theoretisch mögliche Zustandsraum; wäre i.A. ja auch unpraktikabel).
- Für Konformanztests: Wächterbedingungen beachten !
- Konformanztests, die Wächterbedingungen verletzen, können aber als zusätzliche Robustheitstests sinnvoll sein.

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

Ein vollständiger zustandsbasierter Testfall umfasst folgende Informationen:

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

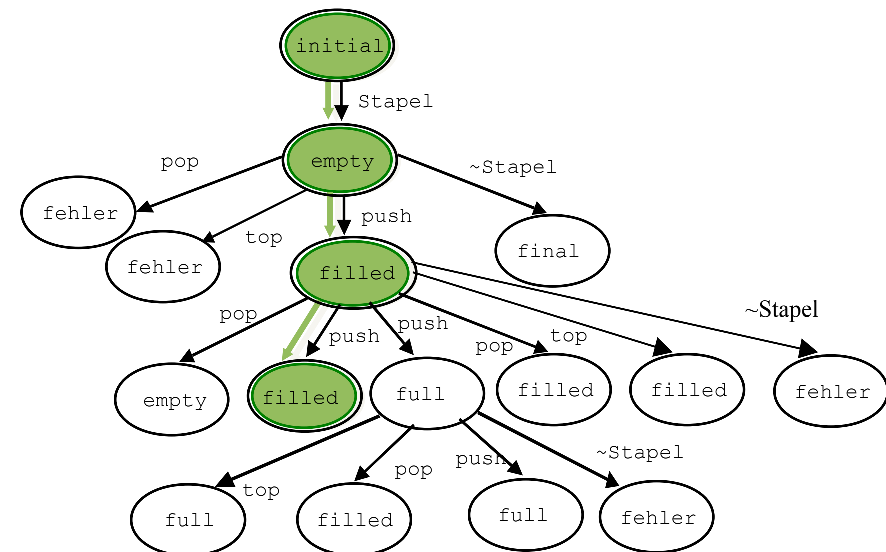
Ferner sind für jeden im Testfall erwarteten Zustandsübergang folgende Aspekte festzulegen:

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

Ausführen der Tests

- Testfälle bzw. Botschaftsfolgen in ein Testskript verkapseln.
- Unter Benutzung eines 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 wurde mindestens einmal eingenommen. Darüberhinaus:

Z-Überdeckungsgrad = Anzahl getestete Z / Gesamtzahl Z

Weitere Kriterien:

- Jeder Zustandsübergang wurde mindestens einmal ausgeführt. Darüberhinaus:

  ZÜ-Überdeckungsgrad = Anzahl getestete ZÜ / Gesamtzahl ZÜ
- Alle spezifikationsverletzenden Zustandsübergänge wurden angeregt.
- Jede Aktion (Funktion) wurde mindestens einmal ausgeführt.

Bei hoch kritischen Anwendungen:

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

Tipps für den zustandsbasierten Test

- Das Zustandsdiagramm bereits bei der Spezifikation unter Testgesichtspunkten bewerten und bei einer hohen Anzahl von Zuständen und Übergängen auf den erhöhten Testaufwand hinweisen und auf eine Vereinfachung dringen, soweit dieses möglich ist.
- Ebenfalls bei der Spezifikation darauf achten, dass die unterschiedlichen Zustände leicht zu ermitteln sind und sich nicht aus einer vielfältigen Kombination von Werten von unterschiedlichen Variablen ergeben.
- Ggf. **Werkzeuge** zum **Model-Checking** verwenden:
  - **Erreichbarkeit** von Zuständen
  - Bei interagierenden Testobjekten: **Deadlock, Livelock** (System „dreht sich im Kreis“), ...

Bewertung des zustandsbasierten Tests

Oft manifestiert sich der Zustand nicht in einer einzigen Variablen, sondern er ergibt sich aus Konstellation der unterschiedlichen Werte der Variablen:

- Der so aufgespannte **Zustandsraum** ist oft sehr **komplex**.
  [Das ist allerdings eine Folge der Systemkomplexität und nicht des Testansatzes.]
- **Überprüfung** / Bewertung der einzelnen Testfälle kann **aufwändig** werden.

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

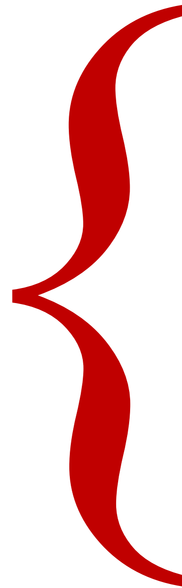
- Andere vorgestellte Testentwurfsverfahren berücksichtigen dies nicht, da sie nicht auf Abhängigkeit des Verhaltens der Funktionen vom Zustand eingehen.

Besonders geeignet zum Test **objektorientierter Systeme**:

- Objekte können unterschiedliche Zustände annehmen.
- Die jeweiligen **Methoden** zur Manipulation der Objekte müssen dann entsprechend auf die unterschiedlichen **Zustände** reagieren.
- Beim objektorientierten Testen hat der zustandsbasierte Test deshalb eine herausgehobene Bedeutung.

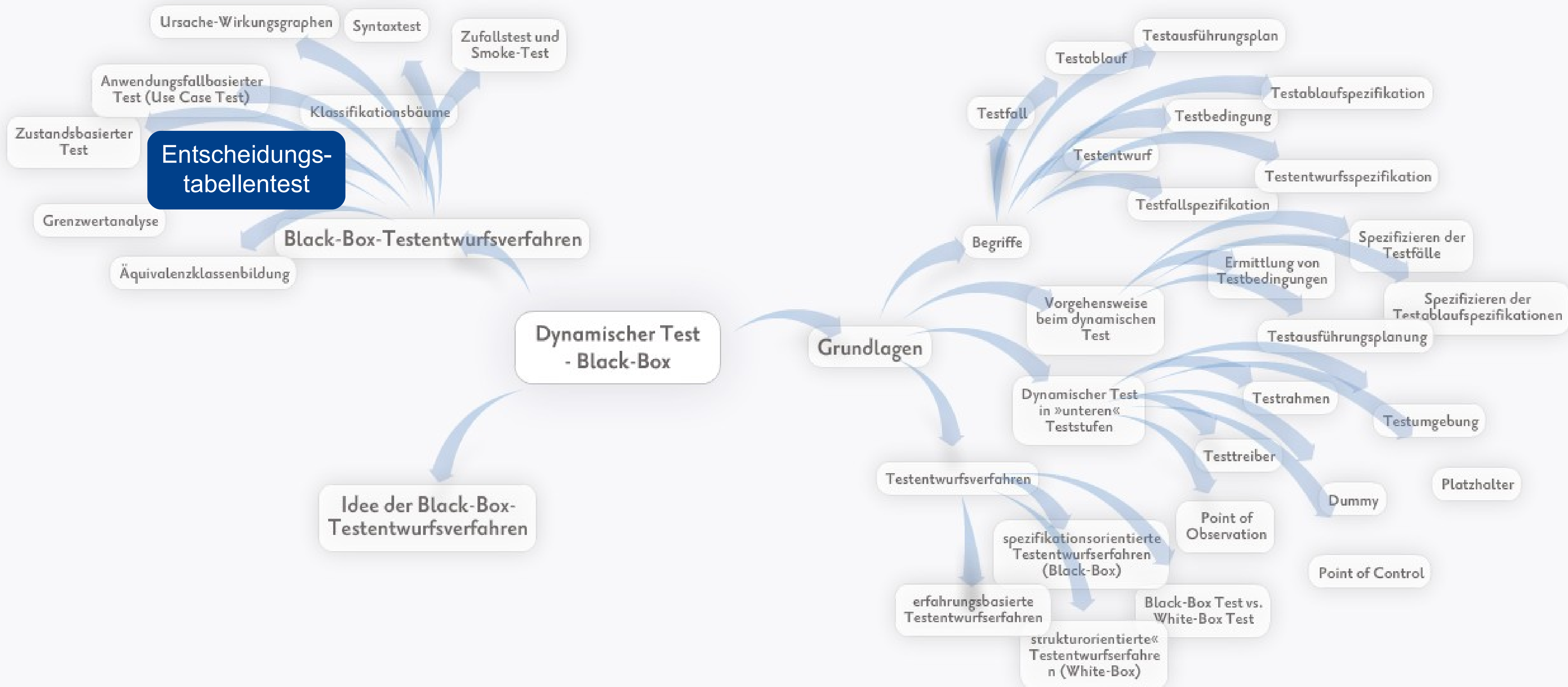
4.4 Black-Box-Test

4.4 Black-Box-Test



- Dynamischer Test – Grundlagen
- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- Grenzwertanalyse
- Zustandsbasierter Test
- **Entscheidungstabellentest**
- Weitere Black-Box-Testentwurfsverfahren

Dynamischer Test – Black-Box



Entscheidungstabellentest

- Anwendbar bei Systemanforderungen bzw. Spezifikationen mit **logischen Bedingungen** und komplexen, vom System umzusetzende **Regeln** in Geschäftsprozessen.
- Spezifikation untersuchen und Eingabebedingungen und Aktionen des Systems ermitteln und so festsetzen, dass sie entweder »wahr« oder »falsch« sein können (boolesche Werte, ja/nein).
- Entscheidungstabelle enthält Kombinationen von »wahr« und »falsch« für alle Eingabebedingungen und die daraus resultierenden Aktionen.
- Jede **Spalte** der Tabelle entspricht einer **Regel im Geschäftsprozess**, die eine eindeutige Kombination der Bedingungen definiert, welche wiederum die Ausführung der mit dieser Regel verbundenen Aktionen nach sich zieht.
- Bei Entscheidungstabellentest üblicherweise verwendeter **Standardüberdeckungsgrad**:
  Wenigstens **ein Testfall pro Spalte** (beinhaltet in der Regel die Abdeckung aller Kombinationen der auslösenden Bedingungen).

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

In einem Warenwirtschaftssystem gelten folgende Geschäftsregeln:

- Die Bestellmenge muss größer als Null sein.
- Teil-Lieferungen sind nicht erlaubt.
- Bei Annahme einer Bestellung muss Lagermenge entsprechend reduziert werden.
- Wird Mindestmenge eines Lagerartikels unterschritten, muss Nachbestellung erfolgen.

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 „–“)

Analyse von Entscheidungstabellen

ET ***vollständig***, wenn bei n Bedingungen alle 2^n Kombinationen enthalten sind (Spalten im oberen Teil betrachten).

ET ***redundanzfrei***, wenn die verschiedenen Bedingungen zu jeweils anderen Aktionen führen (d.h. es gibt keine Bedingung, die man entfernen könnte, ohne an Präzision zu verlieren).

ET ***widerspruchsfrei***, wenn logische Beziehungen zwischen Bedingungen zu damit 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 sind (Spalten im oberen Teil betrachten).

ET ***redundanzfrei***, wenn die verschiedenen Bedingungen zu jeweils anderen Aktionen führen (d.h. es gibt keine Bedingung, die man entfernen könnte, ohne an Präzision zu verlieren).

ET ***widerspruchsfrei***, wenn logische Beziehungen zwischen Bedingungen zu damit 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

Jede Spalte (Regel) entspricht einem Testfall.

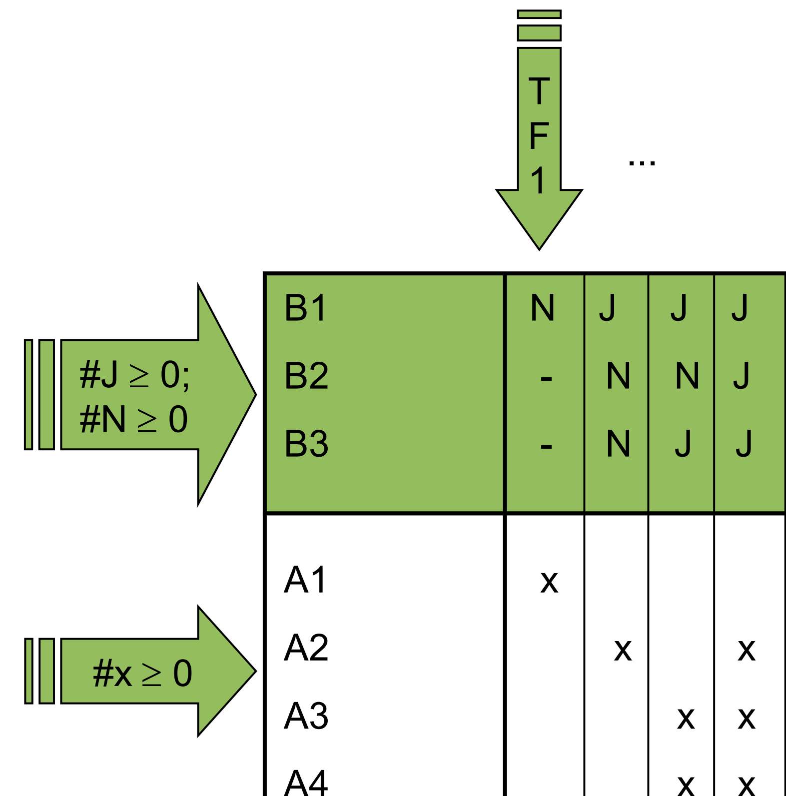
- Voraussetzungen pro Tabelle gleich.
- Bedingungen beziehen sich auf Eingaben.
- Aktionen spiegeln vorausgesagtes (erwartetes) Ergebnis wider.

Überdeckungskriterien z.B.:

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

Vorteile:

- Stärke des Entscheidungstabellentests ist, dass er Kombinationen von Bedingungen ableitet, die andernfalls möglicherweise nicht getestet werden.
- Entscheidungstabellen-Technik kann allgemein zur Problemlösung angewandt werden, wenn Abläufe von mehreren logischen Entscheidungen abhängen.
- Logische Zusammenhänge systematisch formulierbar.
- Entscheidungstabellen können einfach auf Redundanz, Widerspruchsfreiheit und Vollständigkeit geprüft werden.
- Zwingen nicht zur Strukturierung eines Ablaufs.
- Anwendbar auch bei einfacheren zustandsabhängigen Problemen.

Nachteile:

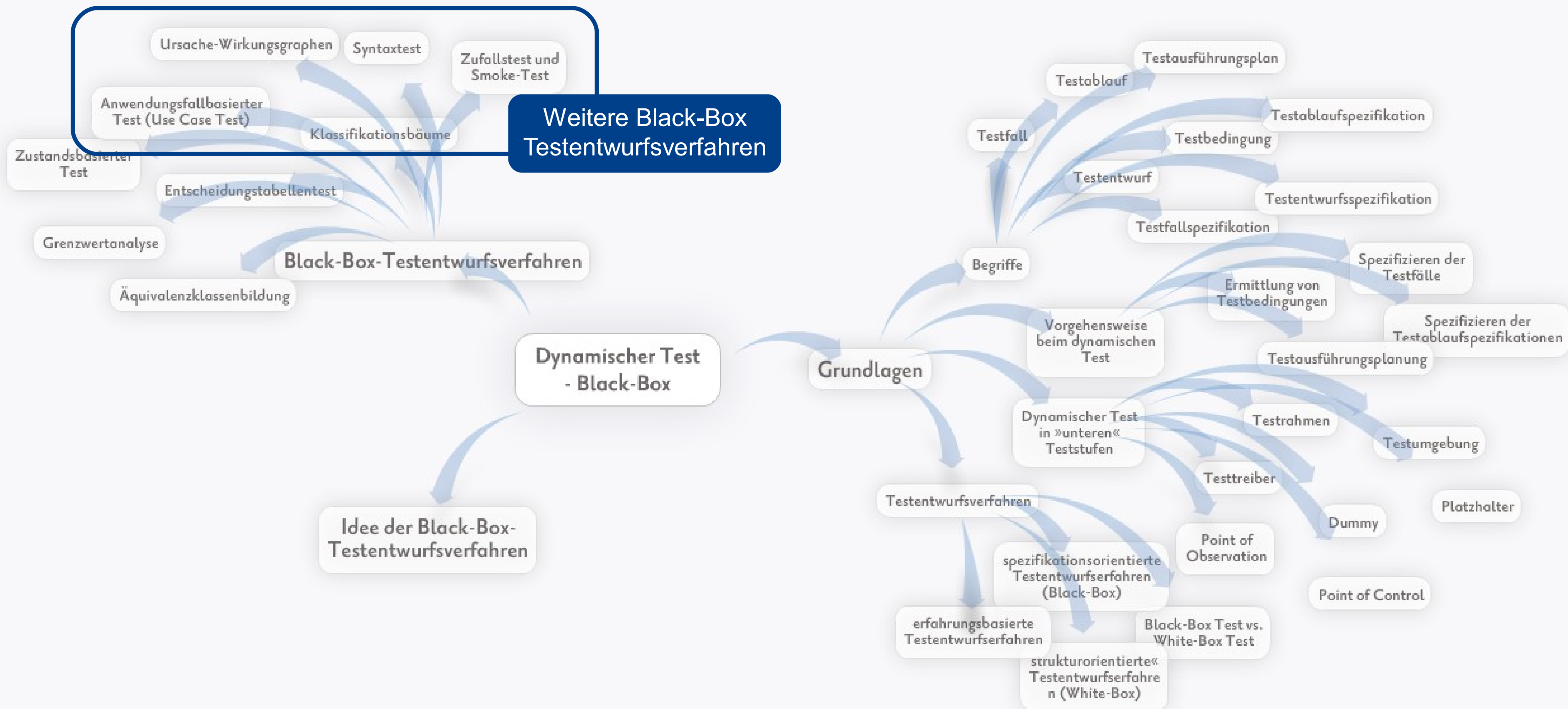
- Unübersichtlich bei zu vielen Bedingungen.
- Zusammenhänge zwischen einzelnen Bedingungen nur implizit ausdrückbar (vgl. Ursache-Wirkungsgraphen).

4.4 Black-Box-Test

4.4 Black-Box-Test

- Dynamischer Test – Grundlagen
- Idee der Black-Box-Testentwurfsverfahren
- Äquivalenzklassenbildung
- Grenzwertanalyse
- Zustandsbasierter Test
- Entscheidungstabellentest
- **Weitere Black-Box-Testentwurfsverfahren**

Dynamischer Test – Black-Box



Weitere Black-Box-Testentwurfsverfahren

- Anwendungsfallbasierter Test (Use Case Test)
- Tests unter Verwendung weiterer Modell-Arten:
  - Ursache-Wirkungsgraphen
  - Klassifikationsbäume
- Syntaxtest
- Zufallstest und Smoke-Test

Anwendungsfallbasierter Test

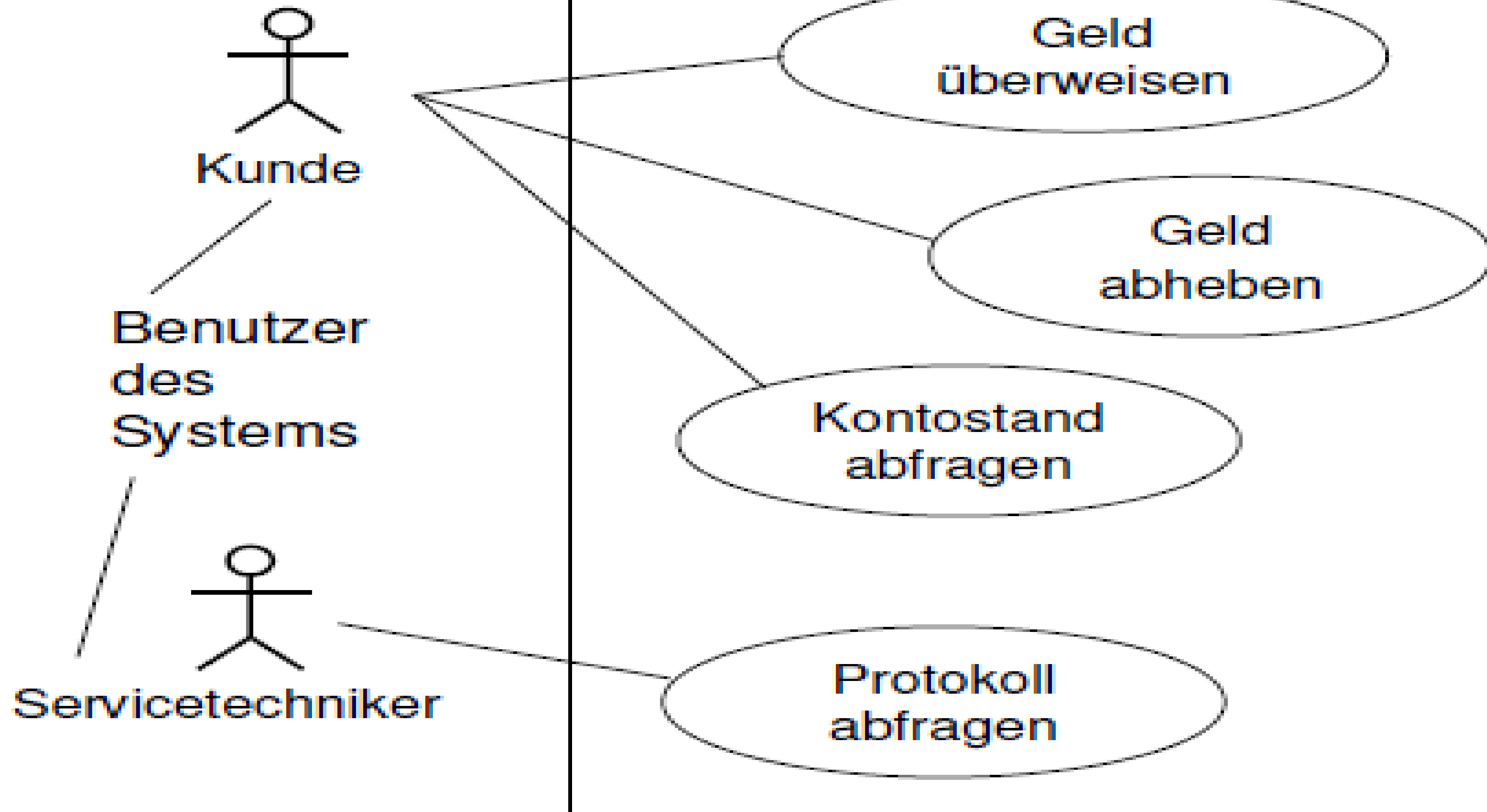
Anwendungsfall (Use Case, Geschäftsszenario): Interaktionen zwischen Aktoren, die zu konkretem Ergebnis führen. Aktor: Benutzer oder etwas, was Informationen mit System austauscht.

- Anwendungsfall hat Vorbedingungen, die erfüllt sein müssen, damit der Anwendungsfall erfolgreich durchgeführt werden kann und endet mit Nachbedingungen, den beobachtbaren Ergebnissen und dem Endzustand des Systems, wenn er vollständig abgewickelt wurde.
- Ein Anwendungsfall hat üblicherweise ein Hauptszenario (das wahrscheinlichste Szenario) und manchmal mehrere alternative Abläufe und Ausnahmefälle (Varianten).
- Anwendungsfälle beschreiben die »Prozessabläufe” durch das System auf Grundlage seiner voraussichtlich tatsächlichen Verwendung.
- Daher sind von Anwendungsfällen abgeleitete Testfälle bestens geeignet, Fehler in den im Praxiseinsatz zu erwartenden Prozessabläufen des Systems aufzudecken.

Anwendungsfallbasierter Test: Ein Black-Box-Testentwurfsverfahren, bei dem Testfälle so entworfen werden, dass damit Szenarien der Anwendungsfälle durchgeführt werden.

- Konkrete Abläufe von Anwendungsfällen (»Szenarien«) sind für den Entwurf von Abnahmetests mit Kunden-/Anwenderbeteiligung sehr hilfreich. Indem gegenseitige Beeinflussung unterschiedlicher Komponenten betrachtet wird, können sie Fehler im Umfeld der Integration aufdecken, die durch den Test der einzelnen Komponenten nicht gefunden werden könnten.

Anwendungsfallbezogener Test: Beispiel Anwendungsfalldiagramm



Beispiel Anwendungsfall-Beschreibung

Anwendungsfall Geld abheben **In Modell** Bankautomat
Aktoren Kunde
Vorbedingung Kartenleser betriebsbereit **UND** Bedienpult gesperrt
Normaler Ablauf

1. Der Kunde meldet sich am Bankautomaten an (include: Anmelden)
2. Der Kunde wählt als Transaktion »Auszahlung«
3. Der Kunde gibt den abzuhebenden Betrag ein
4. Der Bankautomat prüft den Betrag und meldet ihn an den Zentralrechner
5. Der Bankautomat aktualisiert die Karte und gibt sie aus
6. Der Kunde entnimmt die Karte
7. Der Bankautomat gibt das Geld aus
8. Der Kunde entnimmt das Geld

Alternativer Ablauf

4.a Der Betrag ist zu hoch und muss neu eingegeben werden
4.b Der Betrag ist nicht in Scheinen auszahlbar und muss neu eingegeben werden

Nachbedingung Saldo des Kontos um Auszahlungsbetrag reduziert
UND Geldvorrat um den Auszahlungsbetrag reduziert
UND Kartenleser betriebsbereit **UND** Bedienpult gesperrt
Ausnahmeablauf

6.a Die Karte wird nicht innerhalb von 60 Sekunden entnommen

Nachbedingung Karte eingezogen **UND** Kartenleser betriebsbereit **UND** Bedienpult gesperrt
Ausnahmeablauf

1.-4. Der Kunde bricht den Vorgang ab
5.a Der Bankautomat aktualisiert die Karte und gibt sie aus
6.a Der Kunde entnimmt die Karte

Nachbedingung Kartenleser betriebsbereit **UND** Bedienpult gesperrt
END Auszahlung.

Ablauforientierte Testfälle

Testfälle so auswählen, dass die geforderte Überdeckung des Anwendungsfalls erzielt wird:

- Normaler Ablauf
- Alternative Abläufe
- Ausnahmeabläufe
- Mögliche Wiederholungen innerhalb der Szenarien

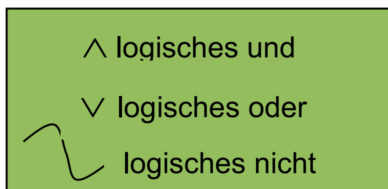
Weitere Black-Box-Testentwurfsverfahren

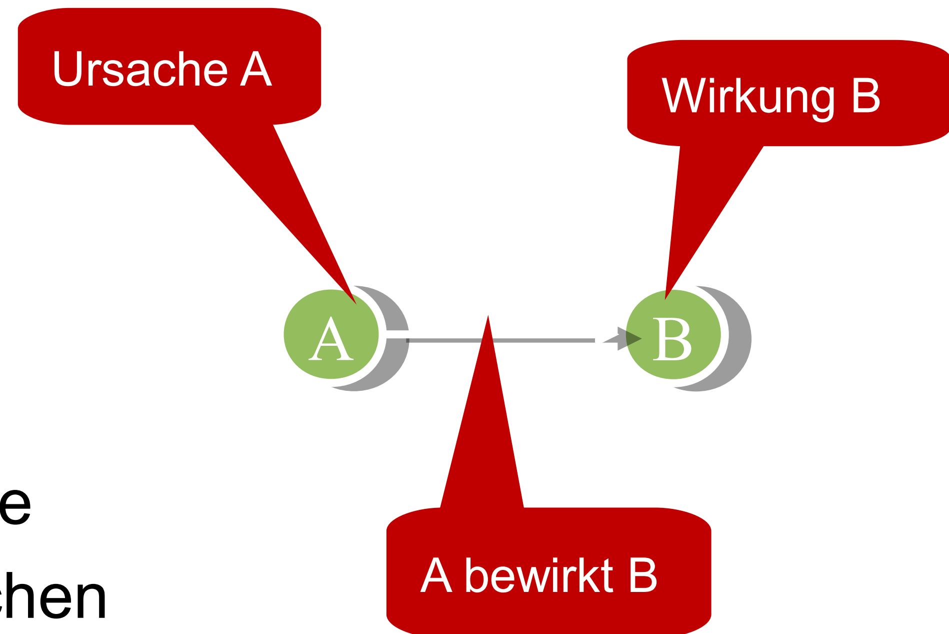
- Anwendungsfallbasierter Test (Use Case Test)
- Tests unter Verwendung weiterer Modell-Arten:
  - Ursache-Wirkungsgraphen
  - Klassifikationsbäume
- Syntaxtest
- Zufallstest und Smoke-Test

Ursache-Wirkungsgraphen

Graphische Beschreibung von (logischen) Wirkzusammenhängen.

- Ursachen:
  - Eingaben
  - Dateiinhalte / Datenbanken
  - Initiale Systemzustände
- Wirkungen:
  - Ausgaben
  - Resultierende Systemzustände
- Logische Verknüpfungen zwischen Wirkungen:





Ursache-Wirkungsgraphen: Beispiel

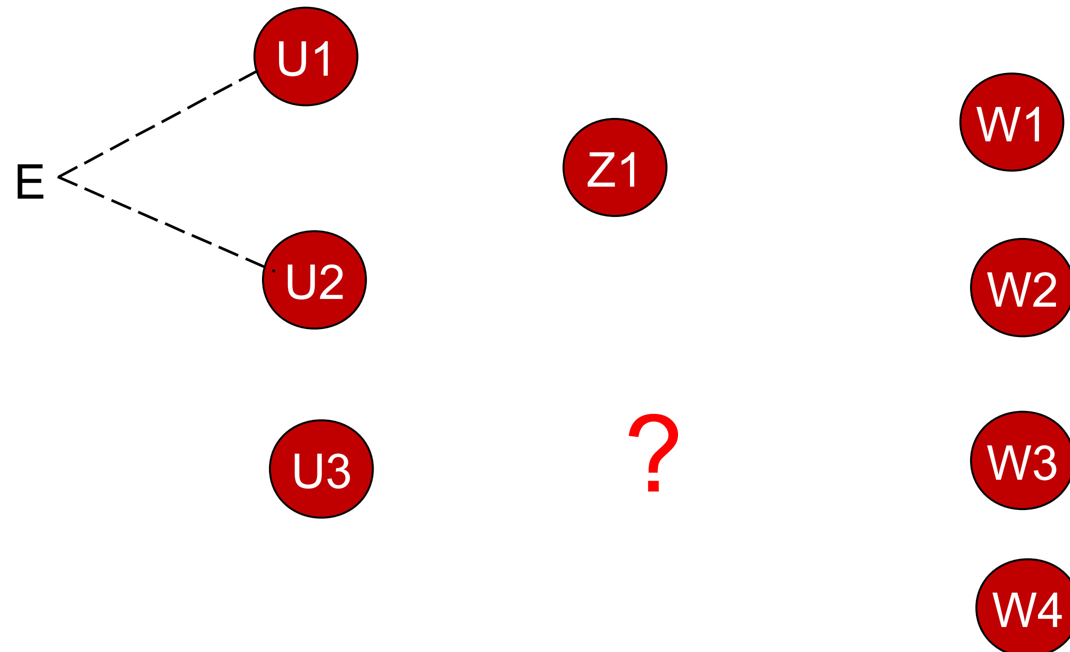
`zaehle(gesamtzahl, vokalAnzahl: int)`

- Liest solange Zeichen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist, oder `gesamtzahl` den größten `int`-Wert erreicht hat.
- Ist gelesenes Zeichen großer Konsonant oder Vokal, wird `gesamtzahl` um 1 erhöht.
- Ist der Großbuchstabe ein Vokal, wird `vokalAnzahl` um 1 erhöht.
- Bei Beendigung werden `gesamtzahl` und `vokalAnzahl` zurückgegeben.

U1: Großer Konsonant
U2: Großer Vokal
U3: gesamtzahl < MaxCardinal
Z1: Zeichen gelesen

W1: Gesamtzahl erhöhen
W2: Vokalanzahl erhöhen
W3: Zeichen lesen
W4: Programmende

∧ logisches und
∨ logisches oder
~ logisches nicht



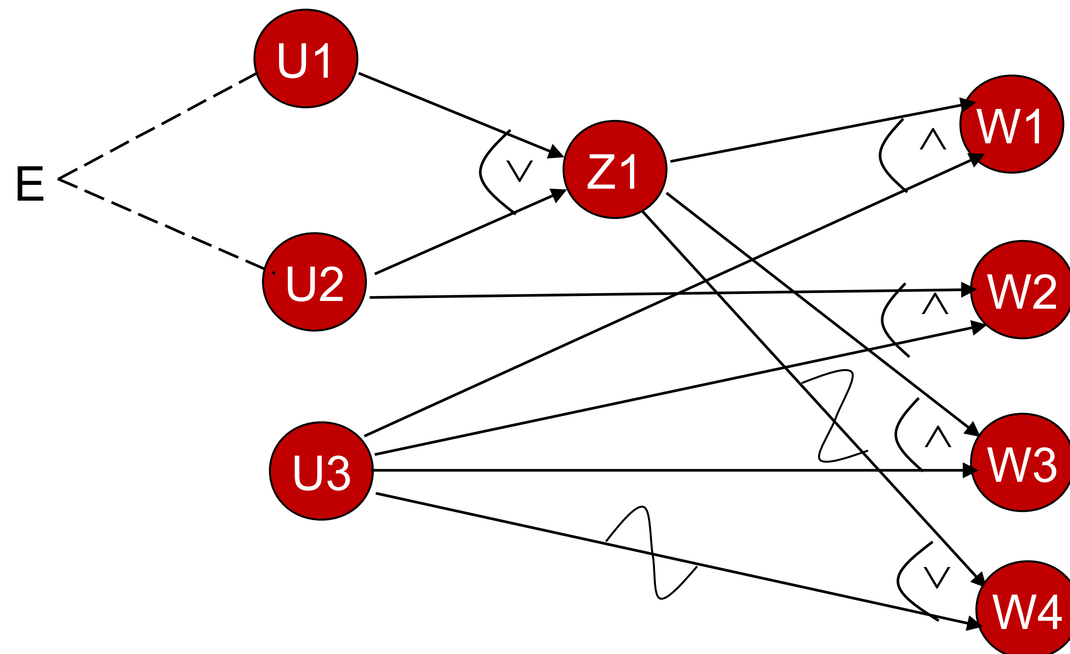
Ursache-Wirkungsgraphen: Beispiel

`zaehle(gesamtzahl, vokalAnzahl: int)`

- Liest solange Zeichen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist, oder `gesamtzahl` den größten `int`-Wert erreicht hat.
- Ist gelesenes Zeichen großer Konsonant oder Vokal, wird `gesamtzahl` um 1 erhöht.
- Ist der Großbuchstabe ein Vokal, wird `vokalAnzahl` um 1 erhöht.
- Bei Beendigung werden `gesamtzahl` und `vokalAnzahl` zurückgegeben.

U1: Großer Konsonant
U2: Großer Vokal
U3: gesamtzahl < MaxCardinal
Z1: Zeichen gelesen

W1: Gesamtzahl erhöhen
W2: Vokalanzahl erhöhen
W3: Zeichen lesen
W4: Programmende



∧ logisches und
∨ logisches oder
logisches nicht

Klassifikationsbäume

large

Aspects

Input Domain

Size

Colour

Shape

size

colour

shape

small large

red green blue

circle

triangle

square

shape of triangle

equilateral isosceles scalene

1

2

3

Eingaben eines logischen Testfalls

M. Grochtmann, J. Wegener, K. Grimm: Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. Proc. Quality Week 1995

Weitere Black-Box-Testentwurfsverfahren

- Anwendungsfallbasierter Test (Use Case Test)
- Tests unter Verwendung weiterer Modell-Arten:
  - Ursache-Wirkungsgraphen
  - Klassifikationsbäume
- Syntaxtest
- Zufallstest und Smoke-Test

Syntaxtest

- Verfahren zur Ermittlung der Testfälle, das bei Vorliegen einer formalen Spezifikation der Syntax der Eingaben angewendet werden kann.
- Die Regeln der syntaktischen Beschreibung werden genutzt, um Testfälle zu spezifizieren, welche sowohl die Einhaltung als auch die Verletzung der syntaktischen Regeln für die Eingaben berücksichtigen.

Beispiel: Syntaxtest (1)

Float grammar in BNF:

```
float  =  int "e" int.
int    =  ["+"|"-"] nat.
nat    =  {dig}.
dig    =  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9“.
```

Tests auf gültige Syntax basierend auf der Bestimmung der Optionen

- float hat keine Option
- int hat drei Optionen: nat [opt_1], "+" nat [opt_2] und "-" nat [opt_3]
- nat hat zwei Optionen: eine einzelne Ziffer [opt_4] und mehrere Ziffern [opt_5]
- dig hat zehn Optionen: eine für jede Ziffer [opt_6 to opt_15]

→ Es sind 15 Optionen zu testen

Beispiel: Syntaxtest (2)

Testfall	Eingabe	Option	Erwartetes Ergebnis
1	3e2	opt_1	Gültig
2	+2e+5	opt_2	Gültig
3	-6e-7	opt_3	Gültig
4	6e-2	opt_4	Gültig
5	1234567890e3	opt_5	Gültig
6	0e0	opt_6	Gültig
7	1e1	opt_7	Gültig
8	2e2	opt_8	Gültig
9	3e3	opt_9	Gültig
10	4e4	opt_10	Gültig
11	5e5	opt_11	Gültig
12	6e6	opt_12	Gültig
13	7e7	opt_13	Gültig
14	8e8	opt_14	Gültig
15	9e9	opt_15	Gültig

Keine minimale Testfallmenge. Es können Tests gewählt werden, die mehrere Optionen abdecken.

Beispiel: Syntaxtest (3)

Test auf ungültige Syntax bspw. durch Mutation

- m1: Ungültiger Wert für ein Element
- m2: Ersetze ein Element durch ein anderes
- m3: Weglassen von Elementen
- m4: Hinzufügen von Elementen

Numerierung der Elemente zur Anwendung der Mutationen:

float	=	int "e" int.	*el_1 = el_2 el_3 el_4.*
int	=	["+"|"-"] nat.	*el_5 = el_6 el_7.*
nat	=	{dig}.	*el_8 = el_9.*
dig	=	"0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".	*el_10 = el_11.*

Beispiel: Syntaxtest (4)

Testfall	Eingabe	Mutation	Element	Erwartetes Ergebnis
1	xe0	m1	x für el_2	Ungültig
2	0x0	m1	x für el_3	Ungültig
3	0ex	m1	x für el_4	Ungültig
4	x0e0	m1	x für el_6	Ungültig
5	+xe0	m1	x für el_7	Ungültig
6	ee0	m2	el_3 für el_2	Ungültig
7	+e0	m2	el_6 für el_2	Ungültig
8	000	m2	el_2 für el_3	Ungültig
9	0+0	m2	el_6 für el_3	Ungültig
10	0ee	m2	el_3 für el_4	Ungültig
11	0e+	m2	el_6 für el_4	Ungültig
12	e0e0	m2	el_3 für el_6	Ungültig
13	+ee0	m2	el_3 für el_7	Ungültig
14	++e0	m2	el_6 für el_7	Ungültig
15	e0	m3	el_2	Ungültig
16	00	m3	el_3	Ungültig
17	0e	m3	el_4	Ungültig
18	y0e0	m4	y in el_1	Ungültig
19	0ye0	m4	y in el_1	Ungültig
20	0ey0	m4	y in el_1	Ungültig
21	0e0y	m4	y in el_1	Ungültig
22	y+0e0	m4	y in el_5	Ungültig
23	+y0e0	m4	y in el_5	Ungültig
24	+0yeo	m4	y in el_5	Ungültig

Zufallstest und Smoke-Test

Zufallstest.

- Wählt aus der Menge der möglichen Werte eines Eingabedatums zufällig Repräsentanten für die Testfälle aus.
- Ist eine statistische Verteilung der Werte zu vermuten oder gegeben (z.B. eine Normal-Verteilung), so soll diese auch für die Wahl der Repräsentanten herangezogen werden, um möglichst realitätsnahe Testfälle zu erhalten und um Aussagen zur Zuverlässigkeit des Systems zu erhalten.

Smoke-Test:

- »Ausprobieren« des Testobjektes, das vorwiegend die prinzipielle Lauf- und Testfähigkeit des Testobjekts prüft.
- Es wird kein Testorakel verwendet und folglich werden auch keine Sollergebnisse erstellt.
- Beim Smoke-Test wird versucht, einen offensichtlichen Absturz des Testobjektes zu erzeugen.
- Oft als »Installationstest«.

Bewertung der Black-Box-Testentwurfsverfahren

Grundlage der Black-Box-Testentwurfsverfahren sind **Anforderungen** und **Spezifikation** des Systems bzw. der einzelnen Komponenten und ihres Zusammenwirkens.

- **Fehlerhafte Anforderungen** oder **Spezifikationen** werden **nicht erkannt**.
- Testobjekt verhält sich so, wie die Spezifikation (bzw. die Anforderungen) es fordert, auch wenn diese fehlerhaft ist.

Nicht geforderte Funktionalität wird (zunächst) nicht (oder nur zufällig) erkannt:

- **Zusätzliche Funktionen** sind weder **spezifiziert** noch vom Kunden gefordert.
- Testfälle, die diese zusätzlichen Funktionen zur Ausführung bringen, werden - wenn überhaupt - nur zufällig durchgeführt.
- Überdeckungskriterien auf der Grundlage der Anforderungen bzw. der Spezifikation (nicht der nicht beschriebenen und nur vermuteten Funktionen).
- **Mutationstesten** kann hier allerdings Abhilfe schaffen.

Im Mittelpunkt der Black-Box-Testentwurfsverfahren steht die **Prüfung der Funktionalität** des Testobjektes. Es ist sicherlich unumstritten, dass das korrekte Funktionieren eines Softwaresystems höchste Priorität hat und somit die Black-Box-Testentwurfsverfahren stets einzusetzen sind.

Zusammenfassung

- Dynamische Tests führen das Testobjekt aus.
- Spezieller Testrahmen oft notwendig.
- Auswahl der Testfälle als (gute) Stichprobe.
- Black-Box-Testentwurfsverfahren benötigen zur Auswahl der Testfälle keine Kenntnis der Programmlogik.
- Funktionale Tests leiten die Testfälle anhand der Spezifikation des Testobjekts ab.
- Äquivalenzklassenbildung in Kombination mit der Grenzwertanalyse zur Erstellung der Testfälle einsetzen.
- Zustandsbasierte Tests mit Übergangsbaum und Zustands-Konformanztest sowie erweitertem Übergangsbaum und Zustands-Robustheitstest.
- Entscheidungstabellentest bei regelbasierten Anforderungen.
- Anwendungsfallbasierter Test für Szenarien der Systembenutzung.

Nach diesem Abschnitt sollten Sie...

- Den dynamischen Test charakterisieren und von anderen Testarten abgrenzen können.
- Die Funktionen eines Testrahmens für den dynamischen Test erklären können.
- White-Box- und Black-Box-Testentwurfsverfahren zur Ermittlung von Testfällen charakterisieren und voneinander abgrenzen können.
- Die Grundidee der Black-Box-Testentwurfsverfahren erläutern können.
- Die Äquivalenzklassenbildung und die Grenzwertanalyse kennen und auf einfache Beispiele anwenden können.
- Den zustandsbasierter Test und den Entscheidungstabellentest kennen und auf einfache Beispiele anwenden können.
- Den anwendungsfallbasierten Test kennen und weitere Black-Box-Testentwurfsverfahren nennen können.

Folgende Fragen sollten Sie jetzt beantworten können

- Welches ist der wesentliche Unterschied zwischen dynamischen und statischen Tests ?
- Welche grundlegenden Funktionen und Eigenschaften muss ein Testrahmen haben ?
- Welches ist der wesentliche Unterschied zwischen Black-Box-Testentwurfsverfahren und White-Box Verfahren zur Testfallermittlung ?
- Worauf basieren funktionale Tests ?
- Was versteht man unter der Äquivalenzklassenbildung ?
- Wann ist die Grenzwertanalyse einsetzbar ?
- Wann ist der zustandsbasierte Test einsetzbar ?
- Wie ist eine Entscheidungstabelle aufgebaut ?
- Wann ist der anwendungsfallbasierte Test einsetzbar ?
- Welche weiteren Black-Box-Testentwurfsverfahren gibt es noch ?

Diese Begriffe sollten Sie kennen ...

LEHRSTUHL 14
SOFTWARE ENGINEERING

spezifikationsorientierte Testentwurfsverfahren
Anwendungsfallbasierter Test
strukturorientierte Testentwurfsverfahren
White-Box-Testentwurfsverfahren
Testausführungsplan
Rückverfolgbarkeit
Zustandsübergangsdiagramm
Testentwurfsspezifikation
Testbedingung
Äquivalenzklassenbildung
Überdeckungsgrad
Grenzwertanalyse
Erfahrungsbasiertes Testentwurfsverfahren
Testfallspezifikation
Testfall
Testentwurf
Testablauf
Testablaufspezifikation
Black-Box-Testentwurfsverfahren
Testentwurfsverfahren
Entscheidungstabellentest
Testskript
Use Case
Eingabe- und Ausgabewerte
Zustandsübergangstabelle
zustandsbasierter Test