www.uni-frankfurt.de

Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

V10 Software-Tests zur Qualitätssicherung Teil 1
Prinzipien / Blackboxtests

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Prolog

- Dieser Foliensatz ist u.a.inspiriert durch die Seminarunterlagen von

  Rainer Schmidberger: Grundlagen des Testens, siehe
  http://www.iste.uni-stuttgart.de/fileadmin/user_upload/iste/se/people/schmidberger/downloads/Grundlagen_Testen-110_k.pdf
  (Zuletzt angeschaut am 17.11.2017).

- **Rainer Schmidberger** gilt mein besonderer Dank.

Inhalt

- **Einführung ins Software-Testen**
  - Übersicht zu Prüfverfahren
  - Motivation und Begriffe
  - Fundamentaler Testprozess
  - Testziele
- **Testfallentwurfsverfahren**
  - Spezifikationsbasierter Test
  - Glassbox-Test
  - Erfahrungsbasierte Verfahren
- Teil 2 (Freitag): Umsetzungen in Python – Weitere Testverfahren

Das Objekt der Begierde

Ein Bug!

[Grace Hopper, Logbuch-Seite des Mark II Aiken Relay Calculator mit dem ersten „Bug“ von1947]

Übersicht zu Software-Qualitätssicherung

- Professionelles Testen ist ein wichtiger Teil

  des **Software Engineerings.**

Übersicht zur Software-Qualitätssicherung
Organisatorische und konstruktive Maßnahmen

Die **Entstehung von Fehlern** wird durch geeignete Maßnahmen während der Entwicklung verhindert:

Organisatorisch:
- Regelung von Zuständigkeit
- Regelung durch Richtlinien, Standards, Checklisten
- Schulung der Mitarbeiter

Konstruktiv:
- Einsatz geeigneter Methoden, Sprachen und Werkzeuge.
- Verwendung bestimmter
- Prozessmodelle

Übersicht zu Software-Qualitätssicherung
Analytische maßnahmen ➔ Software-Prüfung

Der Prüfgegenstand (Teil-, Zwischen- oder Endprodukt) wird auf Fehler hin untersucht.

Die Qualität des Prüfgegenstands wird bewertet.

Es wird geprüft, ob ein Prüfgegenstand bestimmte vorgegebene Qualitätskriterien erfüllt.

Übersicht zu Software-Qualitätssicherung
Inspektion, Review, Walkthrough

- Durchsicht (Schreibtisch-Test): Informell, wird vom Urheber selbst durchgeführt.
- Stellungnahme (Einholen einer zweiten Meinung).
- **Technischer Review:** Wohldefiniertes, dokumentiertes Verfahren zur Bewertung eines Prüfgegenstands
- **Walkthrough:** Mischform zwischen der Stellungnahme und dem technischen Review

Übersicht zu Software-Qualitätssicherung
statische Prüfungen (automatisiert)

- Stilanalysen, wie z. B. Konformität zu Programmierrichtlinien
- Architekturanalysen
- Anomalien (z. B. Datenflussanomalien)
- Code-Metriken

Übersicht zu Software-Qualitätssicherung
Dynamische Prüfung = Software Test

„Testen ist die Ausführung eines Programms **mit der Absicht, Fehler zu finden“**
[Myers, *Art of Software Testing*, 1979]

und Dokumentation der genauen Ausführungsbedingungen wie z.B. Version, **Testfälle** und Resultate.

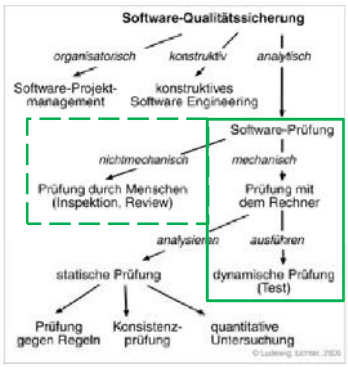
(Automatisches) Testing

Software-Qualitätssicherung

„Nein, gewiß nicht; jedenfalls wollen wir darüber nicht streiten;
es *ist ein weites Feld*.
Und dann sind auch die Menschen so verschieden.

Theodor Fontanes Roman „Effie Briest“ (1894/95).

Was wollen wir uns in PS2 anschauen?



Erste Begriffe

Begriff	Begriffs-Bestimmung	Vermeidbar oder auffindbar durch
Error/Mistake (**Fehlhandlung**, Versagen, Irrtum)	Fehlerhafte Aktion einer Person (Irrtum), die zu einer fehlerhaften Programmstelle führt. **Eine Person macht einen Fehler**	Schulung, Konvention, Prozessverbesserung
Fault/Defect/Bug (**Defekt,** Fehlerzustand, Fehlerursache)	Fehlerhafte Stelle (Zeile) eines Programms, die ein Fehlverhalten auslösen kann. **Das Programm enthält einen Fehler.**	Inspektion, Review, **Wakthrough,** Programmanalyse
Failure (**Fehlverhalten**, Fehlerwirkung, Mangel)	Fehlverhalten eines Programms gegenüber der Spezifikation, das während seiner Ausführung (tatsächlich) auftritt. **Die Anwendung zeigt einen Fehler.**	**Test**

Begriffe nach ISTQB

Prüftechniken:
Statische vs. Dynamische Software Testverfahren

- **Statische Testverfahren:**

  die Software wird bei diesen Tests nicht ausgeführt, sondern der Source Code angeschaut (***non-execution based methods***)

  Hauptmethoden: **Reviews** in verschiedenen Ausprägungen –
  Erlebe "Code Reviews" in den EPR-Übungen

- **Dynamische Testverfahren:**

  Test mit Programmausführung: **Das meinte [Myers 1979] :** „Testen ist die Ausführung eines Programms mit der Absicht, Fehler zu finden“. Testen im engeren Sinn.

  Hauptmethoden: **White-Box Tests + Black-Box Tests (heute!)**

Nutzen von Reviews (1)

- Reviews führen zu einer deutlichen Reduktion von Fehlern.
- Reduktion der zu erwartenden Fehler: 30 bis 75 % aller Fehler (Effektivität von Systemtests: 25 % bis 65 % aller Fehler) [1]
- Fehler, die im Review auffallen, können häufig bedeutend kostengünstiger behoben werden, als wenn diese erst während der Testdurchführung gefunden werden.

[1] Capers Jones, Chief Scientist Emeritus: *Software Quality in 2002: A Survey of the State of the Art.* 23. Juli 2002, abgerufen am 03.05.2017

Nutzen von Reviews (2)

Zu den typischen Schwächen, die mit Reviews entdeckt werden können, gehören:

- Abweichungen von Standards und Richtlinien (Style Guides), z. B. Verletzung von Namenskonventionen, Kommentare, etc.
- Fehler gegenüber (oder auch in) den Anforderungen!
- Fehler im Design
- Unzureichende Wartbarkeit
- Falsche Schnittstellenspezifikation

- Resultate von Code-Reviews sind neben den damit gefundenen Fehlern eine verbesserte Codequalität.
- Reviews können die Softwareentstehungskosten um bis zu 30 % reduzieren.
- … **und das Beste: Sie kennen das schon: Feedback Ihres Tutors!**

Reviews als Philosophie

- Kontinuierliches Review des Quelltextes wie bei der **Paarprogrammierung** (Methoden des Extreme Programming (XP)).
- Also: „Codereview während der Entwicklung“
- **Das sollten Sie kennen! -- Machen Sie das auch so?**
- Ein "öffentliches" Review ist eine Motivation der Open-Source-Software.

Weitere Begriffe aus der Welt des Testens

- **Testfall** (*Test case*): besteht aus Ausführungsbedingungen, Eingabe-daten und Sollresultat.
- ***Testsuite***: Sammlung an Testfällen.
- **Sollresultat** (*Expected result*): das für eine Eingabe spezifizierte Resultat.
- **Überdeckung** (*Coverage*): Testvollständigkeitsmaß.
- ***Testware***: die gesamte Testumgebung (das Testgeschirr – *test harness*).
- **Test-Orakel**: ein Modul, das das Sollresultat für einen Testfall berechnet.
- **Testobjekt:** der Prüfling.
- **SUT**: System (hier im Sinne der Software!) under Test, das Testobjekt.

Mögliche Test-Ausgänge

True-Positive: Korrekt gefundener Fehler.	**False-Positive** Auf Fehler bewertet, obwohl **kein** Fehler vorlag. **„Pseudo-Fehler“**
True-Negative Korrekt auf „kein Fehler“ bewertet	**False-Negative** Auf „**kein** Fehler“ bewertet, obwohl ein Fehler vorlag.

Problemfälle!

Häufig findet man unsystematische Tests (aus der Sicht professioneller Tester)

Laufversuch: Der Entwickler „testet“

- Entwickler übersetzt, bindet und startet sein Programm.
- Läuft das Programm nicht oder sind Ergebnisse offensichtlich falsch, werden die Defekte gesucht und behoben **(“Debugging”).**
- Der „Test “ ist beendet, wenn das Programm läuft und die Ergebnisse "vernünftig" aussehen.

Wegwerf-Test: Testen ohne Systematik

- Jemand „probiert“ das Programm mit verschiedenen Eingabedaten aus.
- Fallen „Ungereimtheiten“ auf, wird eine Notiz gemacht.
- Der Test endet, wenn der Tester findet, es sei genug getestet.

Nach Martin Glinz, Universität Zürich

Ziel: Systematischer Test

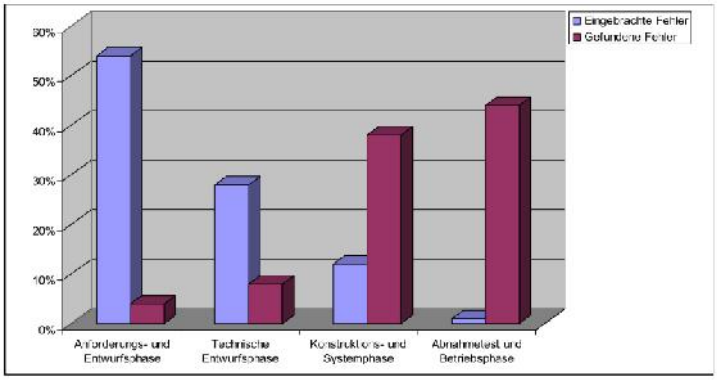
- Spezialisten testen
- **Test ist geplant, eine Testvorschrift liegt vor.**
- Das Programm wird gemäß Testvorschrift - der Testspezifikation – ausgeführt.
- Ist-Resultate werden mit Soll-Resultaten verglichen.
- **Testergebnisse werden dokumentiert.**
- Fehlersuche und -behebung erfolgen separat
- Nicht bestandene Tests werden wiederholt.
- **Test endet, wenn vorher definierte Testziele erreicht sind.**
- Die Testspezifikation wird laufend aktualisiert

Nach Martin Glinz, Universität Zürich

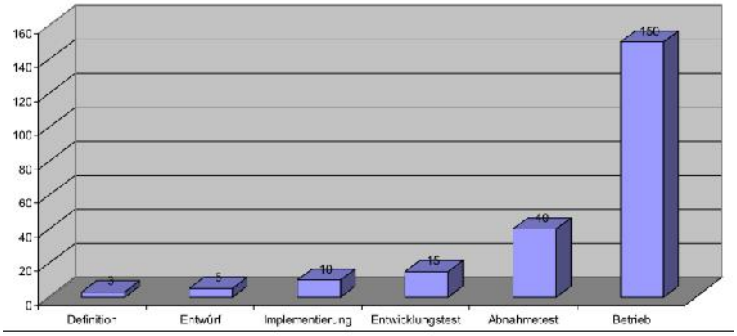
Was gehört zu einem Testauftrag?

- Prüfling (Version, zu testender Teil des Prüflings, genaue Testrahmenbedingungen)
- Termine
- Verfügbarer Testaufwand und ggf. Ressourcen
- Testgüte
  - „Rerun-all“
  - Modifikationstest (nur Änderungen testen)
  - Fehlernachtest (nur behobene Fehler testen)
  - Nur Testfälle einer bestimmten Priorität
- Testende- und Testabbruchkriterien

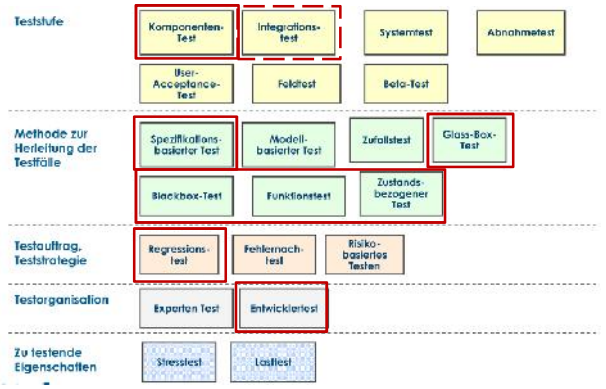
Entstehung und Entdeckung von Fehlern
(aus Balzert, IEEE Software, Jan. 1985, S.83)



Relative Kosten zur Fehlerbeseitigung
(H. Balzert, 1998; Boehm, 1976)



Testarten - Strukturierungsmöglichkeit



Darum wollen wir uns Kümmern!

Wir unterscheiden:

Fokus

- **Unit Test** (Modultest, ***Komponententest***), *unit testing***)**: **Testgegenstand** ist die Funktionalität der Module, Programme, Funktionen, Klassen oder Methoden. **Testziel** ist der Nachweis der technischen Lauffähigkeit und korrekter fachlicher (Teil-) Ergebnisse. **Werden oft noch vom Entwickler durchgeführt.**
- **Integrationstests** *(integration testing)* **t**esten die **Zusammenarbeit** voneinander abhängiger Komponenten. Der Testschwerpunkt liegt auf den Schnittstellen der beteiligten Komponenten.
- **Systemtests** *(integration testing)* testen das "gesamte System" gegen die gesamten Anforderungen (funktionale & nicht-funktionale Anforderungen).
- **Abnahmetest (**Verfahrenstest, Akzeptanztest oder auch *user/operational acceptance test*) ist das Testen der gelieferten Software **durch den Kunden.**

Eine Selbstverständlichkeit: (Nicht überheblich werden!)

Testen kann die Anwesenheit von Fehlern aufzeigen, aber nie einen Nachweis von Fehlerfreiheit liefern!

Edsger W. Dijkstra, Notes on structured programming, Academic Press, 1972

Inhalt

- **Einführung ins Software-Testen**
  - Übersicht zu Prüfverfahren
  - Motivation und Begriffe
  - Fundamentaler Testprozess
  - Testziele
- **Testfallentwurfsverfahren**
  - Spezifikationsbasierter Test
  - Glassbox-Test
  - Erfahrungsbasierte Verfahren

Spezifikationsbasierter Test - Übersicht

- Das SUT wird von außen, an den Systemschnittstellen, betrachtet (der sogenannte Blackbox-Test).
- Grundlage zur Herleitung der Testfälle bildet die Spezifikation.
- Methoden zur Herleitung der Testfälle :
  - Anforderungsbasiertes Testen
  - Äquivalenzklassen-Methode
  - Grenzwertanalyse
  - Klassifikationsbaummethode
  - Entscheidungstabellentest
  - Zustandsbasierter Test
  - Anwendungsfallbasierter Test
  - Zufallstest
  - Modellbasierter Test

Gute Tests – Schlechte Tests

- Ein Testfall ist gut, wenn er mit hoher Wahrscheinlichkeit einen noch nicht entdeckten Fehler aufzeigt.
- Ein idealer Testfall ist
  - Repräsentativ: Er steht stellvertretend für viele andere Testfälle.
  - Fehlersensitiv: Er hat nach der Fehlertheorie eine hohe Wahrscheinlichkeit, einen Fehler aufzuzeigen.
  - Redundanzarm: Er prüft nicht, was auch andere Testfälle schon prüfen.

Anforderungsbasiertes Testen

1. Die Grundlage ist die **Anforderungsspezifikation** (die Aufgabenstellung)
2. Für jede einzelne Anforderung werden Testfälle erstellt **(Anforderungsüberdeckung)**
3. Eine **tabellarische Anforderungsspezifikation** ist hierfür vorteilhaft (Es gibt Requirement-Management-Werkzeuge, die das anforderungsbasierte Testen unterstützen können)

   Vorteil: Wenn systematisch Testfälle zu den Anforderungen erfasst werden, findet damit auch eine Qualitätssicherung der Anforderungen (der Spezifikation) statt.

Dynamische Software Testverfahren

- **Funktionsorientierter Test** (Test gegen eine Spezifikation, **Black Box Test**)
  - **Funktionale Äquivalenzklassenbildung,**
    - als Spezialfall: **Grenzwertanalyse**n
  - Klassifikationsbaummethode, Zustandsbasierter Test, Ursache-Wirkung-Analyse (z. B. mittels Ursache-Wirkungs-Diagramm), Transaktionsflussbasierter Test, Test auf Basis von Entscheidungstabellen
- Strukturorientierter Test (**White /Glass Box Test**)
  - **Kontrollflussorientiert**
    (Maß für die Überdeckung des Kontrollflusses: Anweisungs-, Zweig-, Bedingungs- und Pfadüberdeckungstests
  - Datenflussorientiert

Beispiel-Aufgabe: Einfache römische Zahlen (typische Formulierung)

Schreiben Sie ein Programm, welches eine arabische Zahl zwischen 1 und 49 als Eingabe von der Konsole einliest, diese in eine **einfache** römische Zahl umrechnet und diese dann als String ausgibt. Dabei gibt es drei verschiedenen Zahl-Zeichen mit folgenden Entsprechungen.

arabisch	1	5	10
römisch	I	V	X

Die römische Ziffer mit dem größten Wert steht dabei immer ganz links und die Werte werden kleiner, je weiter rechts das Zeichen steht.

Beispiele: (1) Die Zahl 16 wird zu XVI umgerechnet.
(2) Die Zahl 9 wird zu VIIII umgerechnet.

Achtung: Die Subtraktionsregel wird (noch) nicht angewendet.

Weitere Unterteilung für funktionsorientierte Tests (= Black-Box Tests)

- **Positivtest** (versucht die Anforderungen zu verifizieren)
  Der Testfall prüft also **die korrekte Verarbeitung bei korrekter Handhabung**
- **Negativtest** (prüft die Robustheit einer Anwendung)
  Der Testfall prüft, ob die Anwendung auf eine (falsche) Eingabe oder Bedienung (die also nicht den Anforderungen entspricht) erwartungsgemäß (also ohne Programmabbruch) reagiert, z. B. durch eine Fehlermeldung.

  Beim Negativtest werden absichtlich ungültige Werte eingegeben, Schnittstellen werden mit falschen Werten beliefert, etc..

  Der Negativ-Testfall prüft also auf **"korrekte" Verarbeitung bei fehlerhafter Handhabung ab**.

Methoden zur Testfall (test case) Findung oder Testfallentwurfsverfahren

Äquivalenzklassenbildung

- Es werden die möglichen Werte der Eingaben **und auch** der Ausgaben in Klassen eingeteilt, von denen **vermutet werden kann**, dass Fehler, die bei der Verarbeitung eines Wertes aus dieser Klasse auftreten, auch bei allen anderen Vertretern der Klasse auftreten.
- Wenn andererseits ein Vertreter der Klasse korrekt verarbeitet wird, angenommen, dass auch die Eingabe aller anderen Elemente der Klasse nicht zu Fehlern führt.
- Die Äquivalenzklassenbildung hat zum Ziel **mit möglichst wenig Testfällen** möglichst wirkungsvoll zu testen.

Äquivalenzklassenmethode

- Dazu wird die Spezifikation nach Eingabe und Ausgabegrößen und deren Gültigkeitsbereichen untersucht
- Je Gültigkeitsbereich wird eine Äquivalenzklasse definiert

  Tipp: Wann immer man vermutet, dass Eingabewerte innerhalb einer Äquivalenzklasse nicht gleichartig behandelt werden, sollte in mehrere Äquivalenzklassen aufgeteilt werden.
- Je „Mitte“ einer Äquivalenzklasse wird ein beliebiges Element - ein Repräsentant - als Testfall ausgewählt.
- Vollständigkeit: wenn alle Repräsentanten getestet sind: **(Äquivalenzklassenüberdeckung)**

Beispiel-Aufgabe: Einfache römische Zahlen (typische Formulierung)

Schreiben Sie ein Programm, welches eine arabische Zahl zwischen 1 und 49 als Eingabe von der Konsole einliest, diese in eine **einfache** römische Zahl umrechnet und diese dann als String ausgibt. Dabei gibt es drei verschiedenen Zahl-Zeichen mit folgenden Entsprechungen.

arabisch	1	5	10
römisch	I	V	X

Die römische Ziffer mit dem größten Wert steht dabei immer ganz links und die Werte werden kleiner, je weiter rechts das Zeichen steht.

Beispiele: (1) Die Zahl 16 wird zu XVI umgerechnet.
(2) Die Zahl 9 wird zu VIIII umgerechnet.

Achtung: Die Subtraktionsregel wird (noch) nicht angewendet.

Beispiel: Die ersten Testfälle (tabellarisch)

Äquivalenzklasse (gemäß der Ein-/Ausgabe: in)		+ Test - Test	Repräsentant	Soll-Resultat
1	in <= 0	-	- 9	ERROR: …
2	in >= 50	-	62	ERROR: …
3	in == [1 .. 4]	+	2	II
4	in == [5 .. 9]	+	8	VIII
5	in == [10 .. 49]	+	27	XXVII

1 und 2: Unvollständige Spezifikation. Wir ergänzen die Aufgabenstellung:
In dem Fall in <= 0 soll die Ausgabe 'ERROR: Eingabe: muss größer gleich 1 sein.'
In dem Fall in > 49 soll die Ausgabe 'ERROR: Eingabe: muss kleiner als 50 sein.'

Das systematische Testen ist **auch** eine Qualitätssicherung der Anforderungen!

Die ersten Testfälle: Äquivalenzklassen (vervollständigt)

Äquivalenzklasse (gemäß der Eingabe: in)		+ -	Repräsentant	Soll-Resultat
6	in ist **einstelliges Integer-**Literal zwischen [1..9]	+	3	III
7	" **zweistellig** 2. Stelle [0..9]	+	20	XX
8	in ist ein Alphazeichen aus dem ASCII-Zeichensatz	-	K	'ERROR: Nur Ziffern erlaubt'
9	in ist Sonderzeichen	-	ß	'ERROR: Nur Ziffern erlaubt'
10	in ist Float-Literal	-	15.7	'ERROR: Nur Integer erlaubt'

Diskussion der Testfälle 6 bis 10

1. Benutzereingaben müssen immer besonders gut abgesichert sein.
2. Die Python-Funktion `input` liest einen String ein und liefert diesen als String zurück.
3. Es muss getestet werden, ob dieses ein gültiges Integer Literal ist.

   6 einstellig und erste Stelle im Intervall [1 .. 9]
   7 zweistellig, dann zweite Stelle im Intervall [0 .. 9]
   8 usw.

2. Schritt: Grenzwertanalyse

- Aufdeckung von Fehlern im Zusammenhang mit der Behandlung der Grenzen von Wertebereichen.
- **Als Ergänzung zur Äquivalenzklassenbildung wird kein beliebiger Repräsentant der Klasse als Testfall ausgewählt, sondern Repräsentanten an den „Rändern“ der Klassen.**
- Die Methode ist dann anwendbar, wenn die Menge der Elemente, die in eine Äquivalenzklasse fallen, auf natürliche Weise geordnet sind.
- Hintergrund der Grenzwertanalyse: In Verzweigungen und Schleifen gibt es oft Grenzwerte, für welche die Bedingung gerade noch zutrifft (oder gerade nicht mehr). Diese Grenzwerte sollen getestet werden.

Grenzwertanalyse: Testfälle

- Bereiche
  - (noch gültige) Werte auf den Grenzen
  - Werte „rechts“ bzw. „links neben“ den Grenzen (ungültige Werte, kleiner bzw. größer als Grenze)
- Mengen (z.B. bei Eingabedaten, Datenstrukturen, Beziehungen)
  - Kleinste und größte gültige Anzahl
  - Zweitkleinste und zweitgrößte gültige Anzahl
  - Kleinste und größte ungültige Anzahl (oft: leere Menge)
- Bei mehreren Daten kartesisches Produkt der Grenzwerte
  - **Achtung, kombinatorische Explosion!**

Beispiel: Grenzwertanalyse (1)

Grenzwerte für die Eingabe (zugehörige Äquivalenzklasse)		+ -	Repräsentant in =	Soll-Resultat
1a 3a	in >= 1 *(gültig!)* in == [1 .. 4] *(gültig!) – Grenze*	+	1	I
1b 3b	in <= 0 *(ungültig!)* in == [1 .. 4] (ungültig, kleiner)	-	0	'Falsche Eingabe: ..
2a 5c	in < 50 *(gültig!)* in == [10 .. 49] *(gültig – Grenze)*	+	49	XXXXVIIII
2b 5d	in < 50 *(ungültig!)* in == [10 .. 49] *(ungültig – größer)*	- -	50	'Falsche Eingabe: ..
3c 4a	in == [1 .. 4] *(gültig, Grenze)* in == [5 .. 9] *(ungültig, kleiner)*	+ -	4	IIII

Beispiel: Grenzwertanalyse (2)

Grenzwerte für die Eingabe (zugehörige Äquivalenzklasse)		+ -	Repräsentant in =	Soll-Resultat
3d	in == [1 .. 4] *(ungültig, größer)*	-	5	V
4b	in == [5 .. 9] *(gültig, Grenze)*	+		
4c	in == [5 .. 9] *(gültig - Grenze)*	+	9	VIIII
5a	in == [10 .. 49] *(ungültig – kleiner)*	-		
4d	in == [5 .. 9] *(ungültig – größer)*	-	10	X
5b	in == [10 .. 49] *(gültig - Grenze)*	+		

Beispiel: Grenzwertanalyse (3)

Grenzwerte für die Eingabe (zugehörige Äquivalenzklasse)		+ -	Repräsentant in =	Soll-Resultat
6a	in ist **nullstellig**	-	nullstellig <Return>	'Falsche Eingabe …'
6b	in ist **einstellig**	+	Zu vertiefen	
7a	in ist **zweistellig**	+	Zu Vertiefen	
7b	In ist **dreistellig**	-	100	'Falsche Eingabe …'

Beispiel: Grenzwertanalyse (2)

Grenzwerte **einstellig (6b)**		Repräsentant in =	Soll-Resultat
Integer Literal in [1..9]	+	0 1 9	'Falsche Eingabe: .. I VIIII
Integer Literal in [1..9]	-	A ß	'Falsche Eingabe: .. 'Falsche Eingabe: ..

Grenzwerte **zweistellig (6c)**		Repräsentant in =	Soll-Resultat
Integer Literal in [10..49]	+	10 49	X XXXXVIIII
Integer Literal in [10..49]	-	09 50	'Falsche Eingabe: .. 'Falsche Eingabe: ..
Integer Literal in [10..49]	-	1. 9. AB yz	'Falsche Eingabe: .. 'Falsche Eingabe: .. 'Falsche Eingabe: .. 'Falsche Eingabe: ..

Zusammenfassung erste Suche

- Ganz schön viele Testfälle.
- Dieses einfache Beispiel erzeugt durch Betrachtung der Äquivalenz-klassen und der Grenzwerte schon sehr viele Testfälle.
- Nur systematisches Arbeiten und auch Gespür für mögliche Probleme ergibt eine angemessene Menge von Testfällen.

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)
- Effiziente Kombination mit anderen Verfahren, die Freiheitsgrade in der Wahl der Testdaten lassen

Nachteile

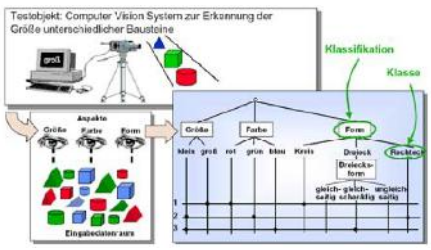
- Rezepte für die Auswahl von Testdaten schwierig anzugeben.
- Bestimmung aller relevanten Grenzwerte (sehr) schwierig.
- Kreativität zur Findung erfolgreicher Testdaten gefordert.
- Methode oft nicht effizient genug angewendet, da sie zu einfach erscheint.

Spezifikationsbasierter Test - Übersicht

- Das SUT wird von außen, an den Systemschnittstellen, betrachtet (der sogenannte Blackbox-Test).
- Grundlage zur Herleitung der Testfälle bildet die Spezifikation.
- Methoden zur Herleitung der Testfälle :
  - Anforderungsbasiertes Testen
  - Äquivalenzklassen-Methode
  - Grenzwertanalyse
  - **Klassifikationsbaummethode**
  - Entscheidungstabellentest
  - Zustandsbasierter Test
  - Anwendungsfallbasierter Test
  - Zufallstest
  - Modellbasierter Test

Klassifikationsbaummethode

Behandelt das Problem der kombinatorischen Vielfalt – der Explosion von Testfällen:



Aus: Pitschinetz, R.; Grochtmann, M.; Wegener, J.: Test eingebetteter Systeme
http://www.systematic-testing.com/

Klassifikationsbaum Übersicht

Einfache grafische Methode zur Unterstützung der Testfall-Erzeugung

Werkzeugunterstützung möglich

Begriffe:

- Klassifikation: Eingabeparameter oder erwartetes Ergebnis
- Klasse: Wertebereich einer Klassifikation. Der Wertebereich kann z.B. eine Äquivalenzklasse sein.
- Die Klassen müssen disjunkt sein und den Wertebereich der Klassifikation vollständig wiedergeben.

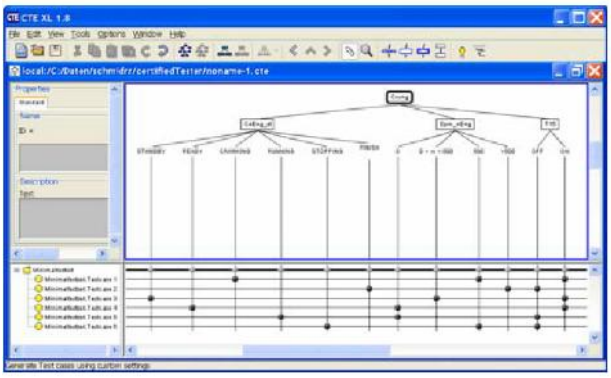
Klassifikationsbaum Methode

1. Klassifikationen wählen
Der gesamten Eingabe- oder Ausgabedatenraum des Prüflings wird in **testrelevante Gesichtspunkte** untergliedert ➔ Klassifikationen

2. Klassen wählen
Jede Klassifikation wird in Wertebereiche unterteilt. Wertebereiche können z.B. Äquivalenzklassen sein. Die Wertebereiche (Klassen) sind disjunkt.

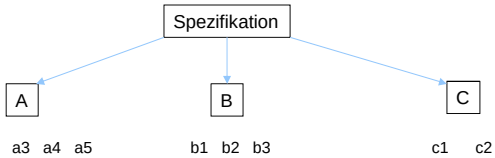
3. Testfälle wählen
Testfälle entstehen durch die Kombination von Klassen unterschiedlicher Klassifikationen.

Klassifikationsbaum – ein Beispiel



Klassifikationsbaum: Testfälle erzeugen

- 3 Klassifikationen (A, B, C) mit folgenden Klassen:



- Vollständige Kombination aller Klassen ergibt 5 x 3 x 2 = 30 Testfälle
  ➔ kombinatorische Explosion
- Um jede Klasse mindestens einmal in einem Testfall zu berücksichtigen sind 5 Testfälle erforderlich

Testfälle generieren

- **Vollständig (oft nicht möglich)**
- **Zufällig:**
  Jede Klasse wird einmal markiert. Die Auswahl geschieht zufällig.
- **Twowise**
  Kombiniert jeweils zwei der angegebenen Klassifikation vollständig miteinander (die weiteren Klassifikationen z.B. zufällig).

  A und B (C zufällig) ➔ 5 x 3 Fälle = 15 Fälle
  A und C (B zufällig) ➔ 5 x 2 Fälle = 10 Fälle
  B und C (A zufällig) ➔ 3 x 2 Fälle = 6 Fälle

  ➔ Bei großen Klassenzahlen sinnvoll

Das ist schon Alles für heut.
Aber, da ist noch Einiges zu tun:

Ausblick für Freitag:

- Umsetzungen in Python: doctest und ("unittest")
- Glassbox Tests und Testabdeckung

Danke für Ihre Aufmerksamkeit!