

Vorlesung "Softwaretechnik"

Buchkapitel 11

Analytische Qualitätssicherung 1

Lutz Prechelt

Freie Universität Berlin, Institut für Informatik
<http://www.inf.fu-berlin.de/inst/ag-se/>

- QS: Was, wann, warum?
- Analytische Qualitätssicherung
 - statisch, dynamisch
- Testen
 - Ziele
- Testfallauswahl
 - Funktionstest (black box)
 - Strukturtest (white box)
- Testgegenstandsauswahl
- Ermitteln des erwarteten Verhaltens
- Wiederholung von Tests
 - insbes. Testautomatisierung

Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
 - Anforderungen (Problemraum)
 - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
 - Kognitive Beschränkungen
 - Mängel der Urteilskraft
 - Kommunikation, Koordination
 - Gruppendynamik
 - Verborgene Ziele
 - **Fehler**

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
 - Abstraktion
 - Wiederverwendung
 - Automatisierung
- Methodische Ansätze ("weich")
 - Anforderungsermittlung
 - Entwurf
 - **Qualitätssicherung**
 - Projektmanagement

- Einsicht: Man macht beim Bau von SW zahlreiche Fehler
 - die häufig zu schwerwiegenden Mängeln führen
- Prinzipien:
 - **Konstruktive Qualitätssicherung**: Ergreife vorbeugende Maßnahmen, um zu *vermeiden*, dass etwas falsch gemacht wird (Qualitätsmanagement, Prozessmanagement)
 - **Analytische Qualitätssicherung**: Verwende prüfende Maßnahmen, die entstandene Mängel aufdecken
 - **Softwaretest**: dynamische Prüfung
 - **Durchsichten**: manuelle statische Prüfung



- Qualitätssicherung (QS, engl. quality assurance, QA)
 - Gesamtheit aller Maßnahmen, die nicht darauf zielen, ein Produkt *überhaupt* fertig zu stellen, sondern darauf, es *in guter Qualität* fertig zu stellen

2 grundlegende Herangehensweisen:

- **Analytische QS:** Prüfend
 - Untersuche (Teil)Produkte nach ihrer Fertigstellung auf Qualität
 - Bessere nach, wo Mängel auftreten
- **Konstruktive QS:** Vorbeugend
 - Gestalte den Konstruktionsprozess und sein Umfeld so, dass Qualitätsmängel seltener werden
 - Beseitige bei entdeckten Mängeln nicht nur den Mangel selbst, sondern auch seine Ursache(n) und ggf. deren Ursache(n)

Externe Qualitätseigenschaften (aus Benutzersicht)

- Benutzbarkeit
 - Bedienbarkeit, Erlernbarkeit, Robustheit, ...
- Verlässlichkeit
 - Zuverlässigkeit, Verfügbarkeit, Sicherheit, Schutz
- Brauchbarkeit
 - Angemessenheit, Geschwindigkeit, Skalierbarkeit, Pflege, ...
- ...

(Man kann diese Listen auch deutlich anders machen, siehe z.B. ISO 9126)

Interne Qualitätseigenschaften (aus Entwicklersicht)

- Zuverlässigkeit
 - Korrektheit, Robustheit, Verfügbarkeit, ...
- Wartbarkeit
 - Verstehbarkeit, Änderbarkeit, Testbarkeit, Korrektheit, Robustheit
 - Dafür nötig sind z.B. Strukturiertheit, Dokumentiertheit, Flexibilität, etc.
- Effizienz
 - Speichereffizienz, Laufzeiteff., Skalierbarkeit, ...
- ...

Interne Q. sind Mittel zum Zweck!

QS: Wann macht man das? Wer ist zuständig?

- Faustregel: Dauernd und Jede/r (Grundprinzip von TQM)
- Analytische QS:
 - Untersuche jedes Teilprodukt so früh wie möglich
 - Je früher ein Mangel entdeckt wird, desto weniger Schaden richtet er an [Zugleich Antwort auf die Frage "Warum QS?"]
 - z.B. Anforderungsmängel erst nach Auslieferung zu beseitigen kostet oft über 1.000-mal mehr als bei der Anforderungsbestimmung
 - Analytische QS ist in der SWT Bestandteil aller Aktivitäten
- Konstruktive QS:
 - Beginne damit vor der eigentlichen Entwicklungsarbeit
 - Gestaltung von Organisation und Arbeitsumfeld
 - Auswahl von Prozessen, Technologie, Strategie
 - Lerne aus Projekterfahrungen
 - Post-mortem: Zusammentragen von Erfahrungen nach Projektende und Umsetzen in Prozessverbesserungen
 - Korrigiere den Prozess ggf. auch unterwegs

Analytische QS:

- **Dynamische Verfahren (Test)**
 - **Defekttest**
 - Benutzbarkeitstest
 - Lasttest
 - Akzeptanztest
- **Statische Verfahren**
 - Manuelle Verfahren
 - Durchsichten, Inspektionen
 - Automatische Verfahren
 - Modellprüfung
 - Quelltextanalyse

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Qualitätsmerkmale von Software (ganz grob)

Externe Qualitätseigenschaften (aus Benutzersicht)

- Benutzbarkeit
 - Bedienbarkeit, Erlernbarkeit, Robustheit, ...
- Verlässlichkeit
 - Zuverlässigkeit, Verfügbarkeit, Sicherheit, Schutz
- Brauchbarkeit
 - Angemessenheit, Geschwindigkeit, Skalierbarkeit, Pflege, ...
- ...

Interne Qualitätseigenschaften (aus Entwicklersicht)

- Zuverlässigkeit
 - Korrektheit, Robustheit, Verfügbarkeit, ...
- Wartbarkeit
 - Verstehbarkeit, Änderbarkeit, Testbarkeit, Korrektheit, Robustheit
 - Dafür nötig sind z.B. Strukturiertheit, Dokumentiertheit, Flexibilität, etc.

Ziel von Defekttests

(Man kann diese Listen auch ganz anders machen)

Effizienz
Speichereffizienz,
Laufzeiteffizienz, Skalierbarkeit

- ...
- Interne Q. sind Mittel zum Zweck!

- **Testen:** Anwenden eines *Testfalls* auf einen *Testgegenstand*
- **Testfall:** ein Tupel, das erwartetes Verhalten beschreibt
 - (Systemzustand, Eingaben, Systemverhalten)
 - Zustand+Eingaben heißen auch **Testeingabe**
 - Verhalten heißt auch **Testausgabe**
- **Testgegenstand (Komponente):** System oder Teil davon, das/der getestet werden soll
 - z.B. ein Modul, mehrere Module, ein Subsystem, mehrere Subsysteme
- **Testdurchführung** (Testausführung):
 - Herstellen des Zustands,
 - Aufruf des Testgegenstands,
 - Übermitteln der Eingaben,
 - Vergleich des Verhaltens mit dem erwarteten
 - **Testergebnis:** Übereinstimmung (keine Erkenntnis) oder Abweichung (Defekt erkannt: Erfolg!)

- Ziel des Defektttests ist Herbeiführen von **Versagen (*failure*)**
 - Also einem falschen Verhalten des Programms
 - Falsch im Sinne der Spezifikation (soweit vorhanden), der Anforderungen (wenn klar) oder der Erwartungen (andernfalls)
- Versagen entsteht aufgrund eines **Defekts (*defect, fault*)** im Programm
 - Eventuell führen erst mehrere Defekte gemeinsam dazu
 - Nicht jeder Defekt muss überhaupt zu einem Versagen führen
- Ein Defekt entsteht aufgrund eines **Fehlers (*error*)** der Softwareentwickler
 - Ein Fehler ist entweder ein **Falschtun (*commission*)** oder ein **Versäumnis (*omission*)**
 - Nicht unbedingt beim Kodieren, vielleicht auch bei Anforderungen oder Entwurf
- Fehlern liegt entweder ein **Irrtum (*misconception*)** oder ein **Versehen (*blunder*)** zu Grunde

- **Wie wählt man Zustände und Eingaben aus?**
- Wer wählt Zustände und Eingaben aus?
- Wie wählt man Testgegenstände aus?
- Wie ermittelt man das erwartete Verhalten?
- Wann wiederholt man Tests?
- Wann/wie kann und sollte man Tests automatisieren?
- Wann kann/sollte man mit dem Testen aufhören?

Wie wählt man Zustände und Eingaben aus?

Vorgehensweisen:

- **Funktionstest** (*functional test, black box test*)
 - Äquivalenzklassen
 - Fehlerfälle
 - Extremfälle
- Strukturtest (*structural test, white box test*)
- Bekannte Versagensfälle
- Allgemeine Erfahrung, Intuition

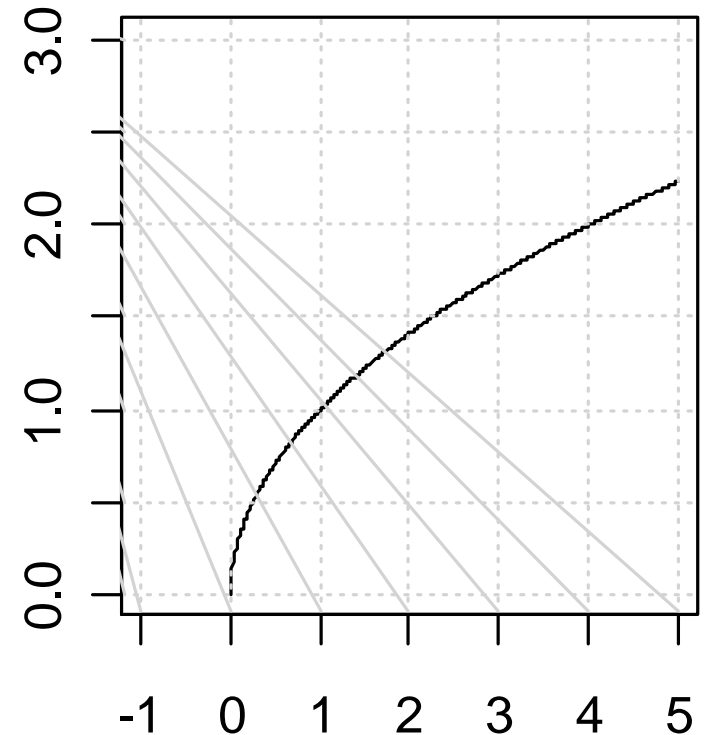
Funktionstest (functional test, black box test)

Wählt Testfälle durch Betrachtung der Spezifikation
(Schnittstelle) der Komponente:

- Für jeden Fall mit andersartigem Verhalten wähle mindestens einen Testfall
- Gruppen solcher Fälle:
 - Äquivalenzklassen gleichwertiger Eingaben
 - Fehlerfälle
 - Heuristik: Randfälle

Funktionstest Beispiel 1: sqrt()

- Spezifikation (java.lang.Math):
`"public static double sqrt(double a)`
 - Returns the correctly rounded positive square root of a double value"
- Testeingaben?
 - 0 1 Fixpunkte
 - -0,164 Fehlerfall
 - 0,5 (wird größer)
 - 1,5 (wird kleiner)
 - Randfälle:
 - Double.NaN
 - Double.POSITIVE_INFINITY
- Beachte: Kein Zustand relevant, sehr klare Semantik
 - So simpel ist es in der Praxis nur ganz selten



- Jede *Component* in einem Java Swing GUI (z.B. ein JButton) kann einen oder mehrere *KeyListener* haben
 - Ein *KeyListener* ist ein Objekt mit Methoden, die bei Tastendrücken aufgerufen werden:
 - `keyPressed(KeyEvent e)`:
Invoked when a key has been pressed
 - `keyReleased(KeyEvent e)`, `keyTyped(KeyEvent e)` entsprechend



- Wir testen das Zufügen von *KeyListeners* zu *Components*
- Spezifikation (`java.awt.Component`)
`public void addKeyListener(KeyListener l)`
 - Adds the specified key listener to receive key events from this component."
- Testfälle: Ersten *KeyListener* zufügen, weiteren K. zufügen, gleichen K. nochmals zufügen, *null* zufügen
 - Zerlegung ist bei weitem nicht eindeutig

- Testfälle: Ersten *KeyListener* zufügen, weiteren K. zufügen, gleichen K. nochmals zufügen, *null* zufügen
 - Zerlegung ist bei weitem nicht eindeutig
- Aber was ist das "erwartete Verhalten"?
 - 1. Listener sind zugefügt laut `getKeyListeners()`? oder
 - 2. Listener werden aufgerufen, wenn ein Tastendruck passiert?
 - Ist die Reihenfolge relevant? Ist sie sichergestellt?
 - Was passiert, wenn ich den selben Listener zweimal zugefügt hatte?
 - Was passiert (soll passieren), wenn ein Listener eine Ausnahme wirft?
 - Ist *null* zufügen erlaubt? Was bedeutet es?

Moral von der Geschichte':

- Oft wird erst beim Testen die Spezifikation richtig geklärt!
 - Das ist meist ein Zeichen für einen schlechten Softwareprozess
 - Denn Klärung beim Entwurf wäre insgesamt billiger gewesen
 - Unbedingt die Dokumentation entsprechend nachbessern

Funktionstest

Beispiel 3: Audioplayer

- Spezifikation: ???
 - (nur ganz vage in Hilfe)
 - Fehlt in der Realität oft!
- Beschränkung auf wichtigste Funktionen:
 - Laufwerkstasten:



Zurück Start Pause Stopp Vor

- Zustände erraten (wären eigentlich der Spez. zu entnehmen):
 - Stopp, Pause, Spielt 1. Titel, Spielt letzten T., Spielt mittleren T.
 - Zustandsmenge wäre größer, wenn wir auch Zufallsfunktion (shuffle) und Endlos (repeat) testen würden
 - Kluge Wahl der Zustandsmenge ist entscheidend für den Test!
- Testfälle:
 - Teste jede Taste in jedem Zustand: $5 \times 5 = 25$ Testfälle

Funktionstest: Quellen von Spezifikationen

Wo sollte die Spezifikation stehen?

- API-Beschreibungen
 - Für Klassen, Module, Komponenten
- Use Cases oder förmliche Spezifikation
 - Für interaktive Systemteile (evtl. auch für interne)
- Benutzerhandbücher
 - Für interaktive Systemteile
 - (Viele Handbücher sind in dieser Hinsicht jämmerlich)
- Diese Dokumente müssen stets aktuell gehalten werden
 - Sonst wird der Test entweder teilweise sinnlos oder unnötig aufwändig

- Wählt Testfälle durch Betrachtung der Spezifikation (Schnittstelle) der Komponente
 - Ein Testfall für jedes verschiedenartige Verhalten
 - z.B. Standardfälle, Fehlerfälle, Randfälle
- Verlangt oft erhebliche Kreativität und Urteilskraft
 - Es ist nicht klar, was alles als "verschiedenartig" gelten sollte
- Gut: Verlangt Nachdenken über das Sollverhalten
 - Sehr oft liegt keine ausreichende Spezifikation vor
- Oft kombinatorische Explosion der Fälle
 - Beschränkung auf wichtigste Fälle nötig

Äh, wo waren wir doch gleich?

Analytische Qualitätssicherung

- Dynamische Verfahren (Test)
 - Defekttest
 - Wie wählt man Zustände und Eingaben aus?
 - Funktionstest (*black box test*)
 - **Strukturtest (*white box test, glass box test*)**

Strukturtest (structural test, white box test)

- Wählt Testfälle durch Betrachtung der Implementation (Struktur) der Komponente
 - Man "guckt also in die Komponente rein", deshalb *white box test*
 - als Gegensatz zu *black box test*
 - Manchmal auch treffender genannt *glass box test*
- Ziel: Abdeckung aller Elemente der Implementation durch die Summe der Testfälle
 - Was heißt "Abdeckung"?
 - Was heißt "Element"?

Arten von Abdeckung (Überdeckung, coverage):

- **Anweisungsüberdeckung** (oft genannt **C₀**)
 - Jede Anweisung wurde mindestens einmal ausgeführt
 - schwächstes Kriterium
 - aber häufigstes in der Praxis eingesetztes Kriterium
 - Für Fehlerbehandlungscode meist schwierig oder sogar unmöglich
- **Bedingungsüberdeckung** (oft genannt **C₁**)
 - Zusätzlich: Jede Steuerbedingung (bei if, while, switch etc.) war mindestens einmal falsch und einmal wahr
- **Schleifenüberdeckung** (manchmal genannt **C₂**)
 - Zusätzlich: Jede Schleife wird einmal 0-fach, einmal 1-fach und einmal mehrfach durchlaufen
- **Datenflusskriterien**
 - Viele verschiedene Kriterien der Art: "Jedes Beschreiben einer Variable wird auch später mal ausgelesen/benutzt"
 - mächtig, aber wenig praktikabel und schwierig anzuwenden

Strukturtest: triviale Beispiele

- 7 Anweisungen (3 if, 3 Zuweisg., 1 assert)
3 Bedingungen
- Bestmögliche Anweisungsüberdeckung ist 86% (6 von 7)
 - denn das *assert* ist nicht erreichbar
- 3 Testfälle nötig, z.B.: $x \in \{2, -5, 0\}$

```

if (x > 0)
    result = 1;
else if (x < 0)
    result = -1;
else if (x == 0)
    result = 0;
else
    assert false;
    
```

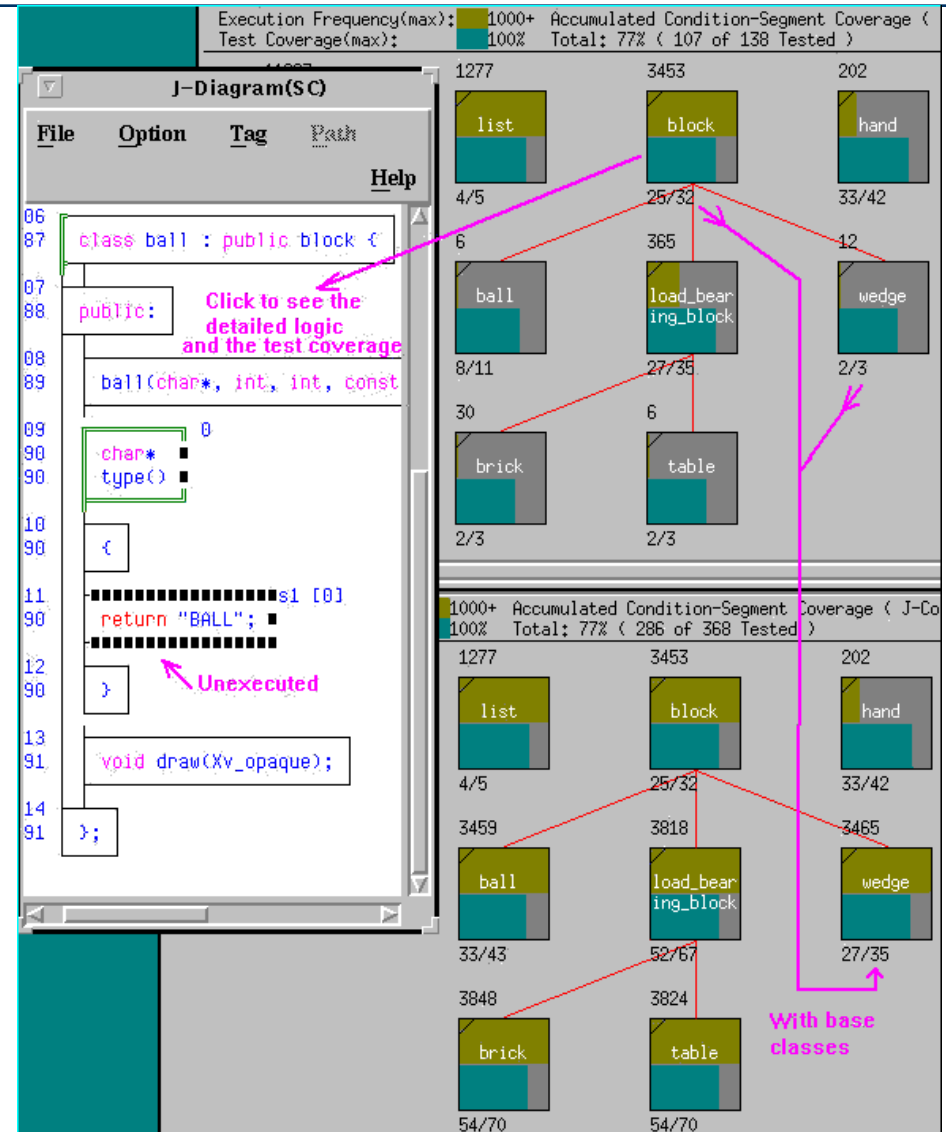
- 5 Anweisungen (2 if, 3 Zuweisungen)
2 Bedingungen
- Bestmögliche Anweisungs- (C_0) und Bedingungsüberdeckung (C_1) ist 100%
- Nur 2 Testfälle nötig, z.B.: $x \in \{-3, 0\}$

```

if (x < 0)
    result = -1;
else
    result = 1;
if (x == 0)
    result = 0;
    
```

Abdeckungsmessung durch Werkzeuge

- Bei komplexer Logik kann man die Abdeckung nicht mehr gezielt konstruieren
- Entsprechende Werkzeuge messen zur Laufzeit die Abdeckung der Tests
 - (meist nur die Anweisungsüberdeckung)
 - und zeigen Übersichten an.
- Testfälle zum Schließen der Lücken muss man sich meist selbst ausdenken
 - Obwohl es auch längst Werkzeuge gibt, die dabei helfen können



- Kontrollfluss ist bei OO nicht mehr direkt aus Quelltext erkennbar:
 - Ein Aufruf `myobj.mymeth()` kann auf viele verschiedene Methodenrumpfe `mymeth()` verweisen
 - Je nachdem, zu welcher Unterklasse `myobj` gerade gehört
- Konsequenz: Nach Einführung einer neuen Unterklasse müssen alle Programmteile neu getestet werden, die Variablen der Oberklasse evtl. mit Exemplaren dieser Unterklasse verwenden
- Kann im Einzelfall furchtbar kompliziert werden
- Nur durch disziplinierten Entwurfsstil zu lösen:
 - Entwurf per Vertrag (Design by contract)
 - Darum ist das Nachdenken über pre-/postconditions so wichtig: Qualitätssicherung!
 - Gründlicher Modultest gegen diesen Vertrag

Wie wählt man Zustände und Eingaben aus?

Vorgehensweisen:

- Funktionstest (*functional test, black box test*)
- Strukturtest (*structural test, white box test*)
- **Bekannte Versagensfälle**
- **Allgemeine Erfahrung, Intuition**

Bekannte Versagensfälle

- Wird ein Versagen im Test nicht aufgedeckt
 - sondern erst später,
wird genau dieser Fall (falls reproduzierbar) nach der Korrektur in jedem Fall getestet
- Evtl. wird dieser Testfall automatisiert und seine Durchführung künftig nach jeder Änderung wiederholt

- Jemand mit genug Testerfahrung "riecht" vielversprechende Testfälle
 - Solcher Intuition sollte man folgen: Oft viel effektiver als schematische Anwendung von Regeln
- Außerdem gibt es ein paar oft anwendbare Faustregeln:
 - Leere Eingaben
 - Riesige Eingaben
 - Völlig unsinnige Eingaben
 - z.B. Binärdaten statt Text; irrwitzige Reihenfolgen von Operationen; etc.
 - Verschachtelte Fehlersituationen
 - Falscheingabe in der Falscheingabe, beim Programmabbruch etc.
 - und andere je nach Domäne

Wer wählt Zustände und Eingaben aus?

1. Jemand, der/die darin geschult ist
 2. Möglichst nicht der/die Entwickler/in des Codes
- Psychologische Gründe sprechen gegen Test nur durch Code-Entwickler:
 1. Man weist sich ungern selbst seine Unzulänglichkeit nach
 2. Bei Fehlverständnis der Spezifikation enthielten ggf. Code und Tests die selben falschen Annahmen
 - Ausnahme: Test-First-Entwicklung
 - Entwickle und implementiere zuerst die Testfälle, *dann* den Code
 - Geht der psychologischen Abneigung aus dem Weg
 - Verwendet Testfallentwicklung zur Vertiefung des Verständnisses der Spezifikation (oder überhaupt zur/als Aufstellung der Spezif.)
 - (Ein paar Entwicklertests sind aber stets empfehlenswert
 - Fürs Größte, denn die Kommunikation von Versagen zurück an Entwickler ist aufwändig und fehlerträchtig.)

- Wie wählt man Zustände und Eingaben aus?
- Wer wählt Zustände und Eingaben aus?
- **Wie wählt man Testgegenstände aus?**
- Wie ermittelt man das erwartete Verhalten?
- Wann wiederholt man Tests?
- Wann/wie kann und sollte man Tests automatisieren?
- Wann kann/sollte man mit dem Testen aufhören?



- Man sollte keinesfalls gleich das ganze System auf einmal testen (*Big-Bang-Test*)
 - z.B. weil erst kleine Teile davon fertig sind
 - (man sollte aber mit dem Testen so früh wie möglich anfangen)
 - z.B. weil das Lokalisieren der Defekte dann viel zu schwierig wäre
- sondern mit wenigen Teilen anfangen und dann schrittweise mehr zusammenfügen.

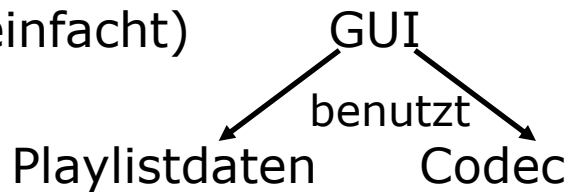
3 Möglichkeiten:

- Bottom-Up-Testen
 - Fange mit den Teilen an, die keine anderen Teile voraussetzen
- Top-Down-Testen
 - Fange beim "Hauptprogramm" an
- Opportunistisches Vorgehen

- Teste stets nur Komponenten, für die alles, was sie aufrufen, schon verfügbar und getestet ist
 - Test beginnt also mit den elementaren Modulen, die nichts anderes aufrufen
- Aufwand: Wir müssen Testtreiber schreiben
- Vorteil: Wir testen nur "reale" und relevante Sachen
- Häufigstes Vorgehen bei gründlichem Test
- Praktikabel

Bottom-up, triviales Beispiel: Audioplayer

(stark vereinfacht)

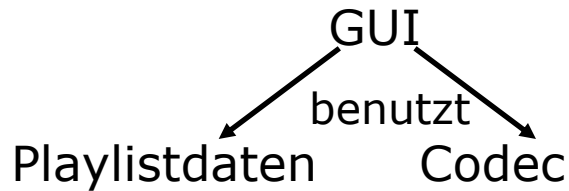


- 1. Teste Codec
 - Treiber für Laden, Start, etc.
- 2. Teste Playlist
 - Treiber für Einfügen, Löschen, Nächster, Listeabrufen etc.
- 3. Teste evtl. GUI ohne Playlist (GUI + Codec)
 - (ohne Treiber): Start, Stopp, Pause, Weiter, Lautstärke, Balance
- 4. Teste Player komplett (GUI + Codec + Playlist)
 - (ohne Treiber): Alle Funktionen



- Teste stets nur Komponenten, für die keine Treiber nötig sind, weil alles, was sie aufruft, schon verfügbar und getestet ist
 - Test beginnt also mit dem "Hauptprogramm", z.B. der Benutzungsschnittstelle
- Aufwand: Wir müssen für alle fehlenden aufgerufenen Teile Simulationen (*Stummel*, *stub*, *dummy*, *mock-up*) schreiben
- Vorteil: Wir testen anschauliche und relevante Sachen
- Nachteil: Stummel sind meist unflexibel
 - Sie enthalten meist Testfälle hartkodiert, sind also aufwändig
- Nachteil: Stummel verbergen oft Defekte
 - weil sie sich anders verhalten als das echte Teil
- Nur sinnvoll, wenn die Entwicklung stark von der Benutzungsschnittstelle getrieben wird
 - z.B. weil die Anforderungen nur dadurch zu klären sind

Top-down, triviales Beispiel: Audioplayer



- 1. Teste GUI
 - Stummel für Codec: macht Bildschirmausgaben wie "Starte Wiedergabe von Song X", "Pause", etc.
 - Stummel für Playlist: fast so kompliziert wie das echte Modul!
- 2. Teste GUI mit echter Playlist
 - Stummel für Codec bleibt im Einsatz
- 3. Teste Player komplett (GUI + Codec + Playlist)



Opportunistisches Vorgehen

- In der Realität geht man selten streng nach Bottom-Up- und schon gar nicht nach Top-Down-Reihenfolge vor
- Statt dessen weicht man ab, wo und wie das praktisch und angemessen erscheint
 - (Siehe das Top-Down-Beispiel eben: In Wirklichkeit würde man nie einen Stummel für das Modul Playlistdaten schreiben
 - denn der wäre kaum einfacher als das Modul selbst
 - also entwickeln wir das zuerst, testen es Bottom-Up und setzen es dann gleich im Top-Down-Test 1 mit ein.)
- Dabei spielen Testprioritäten eine große Rolle:

Überlegungen zur Wahl der Testgegenstände:

- Teile in folgenden Fällen besonders früh testen
 - hat viele Aufrufer (Fehlen behindert weitere Entwicklung stark)
 - hat hohe Risiken für andere Teile
 - Unsicherheiten bei den Anforderungen
 - Auswirkungen auf Architektur (z.B. wegen Effizienz)
- Teile in folgenden Fällen besonders intensiv testen
 - hat viele Aufrufer im System oder Funktionalität ist für Benutzer sehr wichtig (z.B. ständig benutzt)
 - Spezifikation ist sehr komplex oder Versagen ist schwer zu bemerken
 - Sonstwie kritisch
- Ziel ist stets:
 - Möglichst alle Mängel schon im Komponententest aufdecken,
 - weil es später viel schwieriger wird, das Versagen zu lokalisieren

- Wie wählt man Zustände und Eingaben aus?
- Wer wählt Zustände und Eingaben aus?
- Wie wählt man Testgegenstände aus?
- **Wie ermittelt man das erwartete Verhalten?**
- Wann wiederholt man Tests?
- Wann/wie kann und sollte man Tests automatisieren?
- Wann kann/sollte man mit dem Testen aufhören?

Wie ermittelt man das erwartete Verhalten?

- Oft ist das "erwartete Verhalten" für einen Testfall nur mit viel Aufwand zu bestimmen
 - deshalb schreibt man ja schließlich eine Software!
- Mögliche Gründe:
 - Komplizierte Eingaben/Fallunterscheidungen (z.B. Bilanzierung),
 - viele Iterationsschritte (z.B. Optimierungsverfahren)
 - viele mitwirkende Systembestandteile (z.B. verteiltes System)

Lösungsansätze:

- Referenzsystem
- Orakel
- Plausibilitätsprüfung
- Notfalls: Manuelle Bestimmung



- Manchmal ist eine andere Software verfügbar, die als Vergleich dienen kann
- Beispiele:
 - Altsystem
 - z.B. bei Ersatz eines betrieblichen Informationssystems (etwa wegen Technologiewechsel)
 - System der Konkurrenz
 - bei manchen Arten von Standardsoftware (z.B. bei Systemsoftware)
 - Referenzimplementierung eines Standards
 - z.B. in der Java-Welt, W3C-Welt u.a.
 - Frühere Versionen derselben Software
 - falls dies keine Erstentwicklung ist
- In allen Fällen gilt: Nobody's perfect (auch nicht ein Referenzsystem)
 - Also Vorsicht, nicht zu sehr drauf verlassen

- Orakel: Ein Programm, das die Ausgaben eines anderen Programms als *korrekt* oder *falsch* beurteilt
- Manchmal einfacher herzustellen als die Lösung selbst:
 - z.B. bei Gleichungslöser:
 - Ausgabe in die Gleichung einsetzen und gucken ob sie aufgeht
 - Analog bei vielen Arten von Lösern (KI, Robotik etc.)
 - z.B. bei verlustfreier Kompression:
 - Getestete Dekompression anwenden und mit Original vergleichen
 - Dekompression ist oft einfacher!
 - Verlusthafte Kompression ist ein ganz anderes Thema...

- Manchmal lassen sich Orakel finden, die nicht für alle Eingaben oder Verhalten funktionieren
- z.B. für Programm, das Prozessverklebungen erkennt
 - Orakel prüft, ob erkannte Verklebung wirklich eine war
 - Orakel kann nicht urteilen, falls keine Verklebung angezeigt wird

- Richtige Orakel hat man nur selten zur Verfügung
- Annäherungen an Orakel sind aber oft gut machbar:
 - z.B. bei Sortierverfahren:
 - prüfe, ob Ergebnis sortiert ist und richtige Länge hat
 - z.B. bei Optimierungsverfahren:
 - prüfe, ob Ausgabe eine gültige Lösung ist (aber evtl. nicht optimal)
 - z.B. bei Buchhaltung:
 - prüfe Geldmengenerhaltung (aber vielleicht falsch zugeordnet)
 - z.B. bei Verschlüsselungsverfahren:
 - prüfe Entropie der Ausgabedaten
 - z.B. bei GUIs:
 - prüfe, ob gewisse einzelne Elemente den erwarteten Zustand haben
- Allgemein: Prüfe Invarianten, Konsistenzbedingungen, Minimalanforderungen, bekannte Einzelheiten etc.
 - Was halt geht.
 - Kreativität ist gefragt!

- Wie wählt man Zustände und Eingaben aus?
- Wer wählt Zustände und Eingaben aus?
- Wie wählt man Testgegenstände aus?
- Wie ermittelt man das erwartete Verhalten?
- **Wann wiederholt man Tests?**
- Wann/wie kann und sollte man Tests automatisieren?
- Wann kann/sollte man mit dem Testen aufhören?

Wann wiederholt man Tests?

- Faustregel:** Immer, wenn man die Komponente verändert hat
- Das ist bei manuellen Tests, speziell auf Gesamtsystemebene, jedoch sehr aufwändig
 - Aufwand schon bei mäßig großen Systemen z.B. 10 Personen für 2 Monate → über EUR 100.000
 - Das kann man nicht für jede kleine Änderung investieren
 - zumal man ja dann nach jedem gefundenen Fehler von vorn beginnen müsste...
 - Sparsamere Regeln:
 - Wiederhole grundlegende Testfälle ("smoke test")
 - Wiederhole Tests, die direkt auf die geänderten Funktionen zielen
 - Wiederhole Tests, die auf Daten zielen, die von den geänderten Funktionen verändert wurden
 - **Sehr wichtige und schwierige Fragestellung!**
 - Erfahrene Tester haben hier viel spezifisches Wissen über "ihr" System

- Wie wählt man Zustände und Eingaben aus?
- Wer wählt Zustände und Eingaben aus?
- Wie wählt man Testgegenstände aus?
- Wie ermittelt man das erwartete Verhalten?
- Wann wiederholt man Tests?
- **Wann/wie kann und sollte man Tests automatisieren?**
- Wann kann/sollte man mit dem Testen aufhören?

Wann/wie kann und sollte man Tests automatisieren?

Was ist Testautomatisierung?

- Testen umfasst:
 1. Testfall auswählen
 2. Testfall durchführen
 3. Ergebnis(se) prüfen
 4. Erfolg oder Versagen feststellen
- Testautomatisierung bedeutet, einen oder mehrere dieser Schritte programmgesteuert auszuführen
 1. Testeingabe erzeugen
 2. Testeingabe abarbeiten
 3. Ergebnisse überprüfen
 4. Resultat protokollieren und ggf. Versagen anzeigen

Warum Testautomatisierung?

- **Mehr Testen:**

Manuelles Testen kann immer nur einen winzigen Teil des Verhaltensraums eines Systems abdecken

- Vollautomatisiertes Testen kann den Anteil erhöhen
 - wenn auch die Erzeugung der Testfälle mit automatisiert ist

- **Wiederholt Testen:**

Defekte können auch nach Tests in die Software eingefügt werden

- werden also nur gefunden, wenn später erneut getestet wird

- **Zuverlässiger Testen:**

Manuelles Testen ist seinerseits fehleranfällig

- Oft werden Versagen schlichtweg übersehen

- Automatisiertes Rückfalltesten automatisiert alle Schritte außer der Testfallerzeugung.
 - Testfälle werden (1) manuell implementiert
 - Automatisierung (2) führt aus, (3) prüft Ergebnis und (4) protokolliert
- Sichert Integrität des Systems nach Veränderungen ab
 - Paradebeispiel: Refactoring (siehe Agile Prozesse)
- Probleme:
 1. Das Implementieren/Automatisieren der Testfälle ist sehr aufwendig
 - Vor allem das Prüfen der Resultate
 - Grobe Faustregel: 10x so viel Arbeit wie eine manuelle Durchführung
 2. Rückfalltesten ist nur mäßig wirksam
 - Es gäbe viel mehr Defekte mit neuen Testfällen zu finden als durch die Wiederholung von alten

1. Für Portabilitätstests auf verschiedenen Plattformen
 - Betriebssysteme, Prozessoren, Compiler etc.
 - Beispiel: 6 Java EE Application Server mal 4 RDBMS mal 4 Betriebssysteme mal 2 Java VMs = 192 Kombinationen
 - Kann sogar leicht in die Tausende gehen
2. Zum Test verschiedener Systemkonfigurationen
3. Zur Prüfung der Korrektur gefundener Defekte
 - Testfall wird vor dem Vorliegen der Korrektur implementiert
4. Um sicherzustellen, dass sich gleiche Defekte nicht erneut in die Software einschleichen
 - Passiert manchmal durch Unterlaufen oder Versagen der Konfigurationsverwaltung

Sehr wichtige Technik für professionelle SW-Entwicklung!

Für (+) oder gegen (-) automatisiertes Rückfalltesten spricht:

- + Ist die Benutzungsschnittstelle stabil?
- + Wird es Nachfolgeversionen des Systems geben?
- + Wird es vermutlich Defektkorrekturen geben?
- + Ist indeterministisches Systemverhalten zu erwarten?
- + Ist das Testpersonal des Programmierens mächtig?
- + Gehört das Produkt zu einer Familie ähnlicher Produkte?
- ? Wird das Produkt auf Landessprachen lokalisiert werden?
- - Wird eine saubere Konfigurationsverwaltung durchgeführt?
- - Sind starke Ergänzungen zum Testwerkzeug nötig?
- - Wird es Nachfolgeversionen der Testwerkzeuge geben?

- Automatisierungskosten oder Amortisationszeit unterschätzt
- Trainingsbedarf unterschätzt
- Nutzen von Regressionstest überschätzt
- Testen zugunsten von Testautomatisierung vernachlässigt
- Zu viele simple Testfälle automatisiert
- Zu viel mit Aufzeichnen&Abspielen automatisiert
- 100% Automatisierung versucht
- Testtreiber sind nicht entworfen (zu viel quick & dirty); Redundanz statt Wiederverwendung; nicht wartbar
- Testtreiber sind zu spezifisch (Rechnernamen, Bildschirmauflösung, Betriebssystemversion, ...)
- Keine Dokumentation der Teststrategie, Testfälle, Testtreiber oder der einzelnen Testdurchläufe (fehlende Testprotokolle)
- Zu unflexible Testwerkzeuge

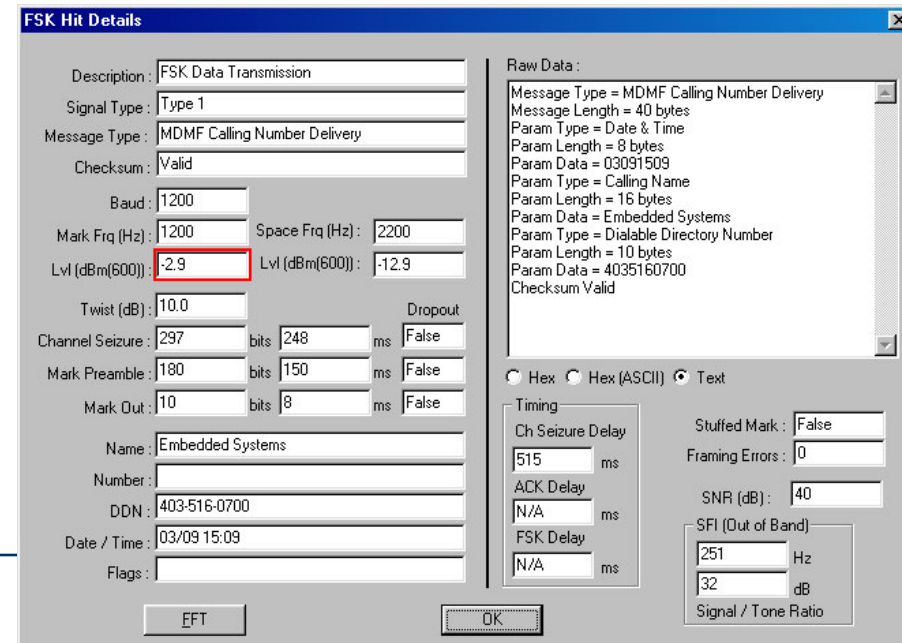
Oft erfolgversprechender/kosten-nützlicher als Rückfalltest:

- Prüfung von Zusicherungen zur Laufzeit
- Dateneingabetreiber
- Testen mit Ergebnisprüfer
- Lasttest, Leistungstest, Stresstest
- Quick and Dirty
- Testframework

siehe nachfolgende Folien

- Zusicherungen sind Bedingungen, die stets erfüllt sind
 - bzw. sein müssten, wenn das Programm korrekt ist
- Solche Bedingungen kann man in den Programmtext integrieren und zur Laufzeit stets überprüfen
 - d.h. jeder Programmlauf enthält unzählige automatisierte Tests
 - Java: `assert`
- Arten von Zusicherungen
 - Voraussetzungen (preconditions) von Methoden
 - Effekte (postconditions) von Methoden
 - Invarianten (invariants) von Schleifen oder Datenstrukturen
- Beispiel: `sqrt(x)` prüft postcondition:
`assert (abs(result*result/x - 1) < 1e-15)`
- Prüfung erfolgt evtl. nur teilweise
 - denn die Prüfung von Quantor-Ausdrücken ist oft zu teuer oder ganz unmöglich

- Automatisiert nur die Testfalldurchführung
 - aber insbesondere nicht die Ergebnisprüfung
- Treiber entnimmt Testfälle (Eingabedaten) einer Tabelle o.ä.
 - und schickt sie ans System
- Prüfung der Ergebnisse erfolgt manuell
- Sehr viel schneller zu bauen als kompletter Rückfalltest
 - und deshalb evtl. lohnend, selbst wenn man die Testfälle gar nicht wiederholen will
- erleichtert auch manuelle oder halbautom. Protokollführung



The screenshot shows the 'FSK Hit Details' window with the following fields:

Description:	FSK Data Transmission		
Signal Type:	Type 1		
Message Type:	MDMF Calling Number Delivery		
Checksum:	Valid		
Baud:	1200		
Mark Frq (Hz):	1200	Space Frq (Hz):	2200
Lvl (dBm(600)):	-2.9	Lvl (dBm(600)):	-12.9
Twist (dB):	10.0	Dropout:	
Channel Seizure:	297 bits	248 ms	False
Mark Preamble:	180 bits	150 ms	False
Mark Out:	10 bits	8 ms	False
Name:	Embedded Systems		
Number:			
DDN:	403-516-0700		
Date / Time:	03/09 15:09		
Flags:			

Raw Data:

```
Message Type = MDMF Calling Number Delivery
Message Length = 40 bytes
Param Type = Date & Time
Param Length = 8 bytes
Param Data = 03091509
Param Type = Calling Name
Param Length = 16 bytes
Param Data = Embedded Systems
Param Type = Dialable Directory Number
Param Length = 10 bytes
Param Data = 4035160700
Checksum Valid
```

Timing:

Ch Seizure Delay	515 ms
ACK Delay	N/A ms
FSK Delay	N/A ms

Stuffed Mark: False

Framing Errors:

SNR (dB):

SFI (Out of Band): Hz

dB

Signal / Tone Ratio

Buttons: EFT, OK

Testen mit Ergebnisprüfer

- Ein Ergebnisprüfer ist ein Programm, das die Korrektheit des Ergebnisses beliebiger Testfälle beurteilen kann
 - automatisiert also die Ergebnisprüfung
 - benötigt für vollen Nutzen auch automatisierte Testfallerzeugung (Zufallstest) und Testfalldurchführung
- Vorteil: Extrem hohe Testfallanzahl möglich

Aber: Woher bekommt man einen Ergebnisprüfer?

- Evtl. Referenzimplementierung
 - z.B. Altsystem, Konkurrenzprodukt
- Alte Version des eigenen Produkts
 - entdeckt nicht Versagen, sondern Verhaltensveränderungen
- Orakel
- Notfalls (besser als nix): Teilorakel, Plausibilitätsprüfung

- Schickt sehr viele (Lasttest), sehr große (Leistungstest, Stresstest) oder sehr falsche (Stresstest) Eingaben
 - (wie üblich ist die Namensgebung aber uneinheitlich)
- Misst die Abarbeitungsgeschwindigkeit (Lasttest, Leistungstest) oder das ordnungsgemäße Überleben (Stresstest) des Systems
- Sind ohne Automatisierung gar nicht durchführbar

Definition:

- Jeder Automatisierungsansatz, der etwas effektiv und nützlich automatisiert, aber
- schlecht oder gar nicht entworfen, hastig implementiert, überhaupt nicht dokumentiert ist.

- Voraussetzung: muss extrem kurze Amortisationszeit versprechen (wenige Tage)
 - wird anschließend weggeworfen

- Anwendung z.B. bei der Evaluation von Fremdsoftware
 - z.B. 5 Produkte sind zu prüfen, aber nur 1 wird später benutzt

Analytische QS:

- **Dynamische Verfahren**
 - **Defektttest**
 - **Wie wählt man Zustände und Eingaben aus?**
 - **Wer wählt Zustände und Eingaben aus?**
 - **Wie wählt man Testgegenstände aus?**
 - **Wie ermittelt man das erwartete Verhalten?**
 - **Wann wiederholt man Tests?**
 - **Wann/wie kann und sollte man Tests automatisieren?**
 - Benutzbarkeitstest
 - Lasttest
 - Akzeptanztest
- **Statische Verfahren**
 - ...

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Danke!