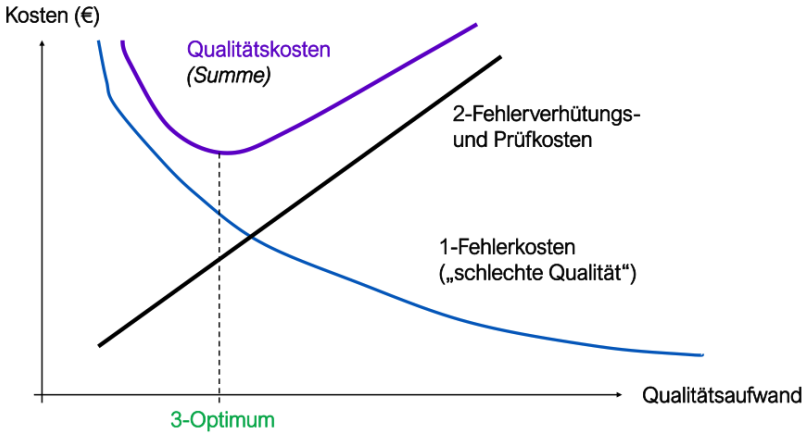
ÜB 01

Wie zeichnet sich schlechte Software-Qualität aus?

- Schlechte Bedienbarkeit
- Unzureichende Sicherheit
- Schlechte Dokumentation
- Hohe Kosten bei Änderung
- Viele Fehler im Programm (Abstürze, Bugs, …)
- Schlechte Performanz

Wann lohnt es sich (noch) Aufwand in die Qualität zu stecken?

- Fehlerkosten (monoton fallend)
- Fehlerverhütungskosten (linear steigend)
- Qualitätskosten (Parabel)
- Das Optimum befindet sich bei dem Minimum der Qualitätskosten



ÜB02

ISO 25010 (Vorgänger 9126)

- Funktionale Eignung
- Effizienz
- Kompatibilität
- Interaktionsfähigkeit
- Verlässlichkeit
- Security (Sicherheit)
- Wartbarkeit
- Flexibilität
- Safety (Betriebssicherheit)

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS FUNCTIONAL CORRECTNESS FUNCTIONAL APPROPRIATENESS iso25000.com	TIME BEHAVIOUR RESOURCE UTILIZATION CAPACITY	CO-EXISTENCE INTEROPERABILITY	APPROPRIATENESS RECOGNIZABILITY LEARNABILITY OPERABILITY USER ERROR PROTECTION USER ENGAGEMENT INCLUSIVITY USER ASSISTANCE SELF-DESCRIPTIVENESS	FAULTLESSNESS AVAILABILITY FAULT TOLERANCE RECOVERABILITY	CONFIDENTIALITY INTEGRITY NON-REPUDIATION ACCOUNTABILITY AUTHENTICITY RESISTANCE	MODULARITY REUSABILITY ANALYSABILITY MODIFIABILITY TESTABILITY	ADAPTABILITY SCALABILITY INSTALLABILITY REPLACEABILITY	OPERATIONAL CONSTRAINT RISK IDENTIFICATION FAIL SAFE HAZARD WARNING SAFE INTEGRATION

Dieses allgemeine Qualitätsmodell ist die Referenz

- Die neun Qualitätsaspekte oben muss man kennen
- Je zwei Unteraspekte sollte man nennen können
- Denn diese Norm ist weithin bekannt
- Bezugspunkt für Projekte
- Referenz für die gesamte Veranstaltung „Softwarequalität“ (inkl. Übungen und Klausur)

ISO 9000

- Qualität ist die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen

Aufbauorganisation

- Softwareentwicklung ist organisiert in der Primärorganisation
- Qualitätsfachleute bilden Sekundär-/Schattenorganisation
- So sind die Qualitätsbeauftragten nicht direkt abhängig vom Projektleiter und können Qualitätssicherungsmaßnahmen besser durchsetzen

Qualitätsmanagement (EN ISO 8402):

- „Alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortlichkeiten festlegen, sowie diese durch Mittel wie Qualitätsplanung, -lenkung, -sicherung und -verbesserung im Rahmen des Qualitätsmanagement-systems verwirklichen.“

Definitionen

- Qualitätsaspekt bzw. -merkmal
  - Eigenschaft, anhand derer die Qualität einer Einheit beschrieben und beurteilt wird (z.B. Bedienbarkeit)
- Qualitätsziel
  - Angestrebtes Qualitätsmerkmal
- Qualitätsanforderung
  - Ein oder mehrere Qualitätsziele

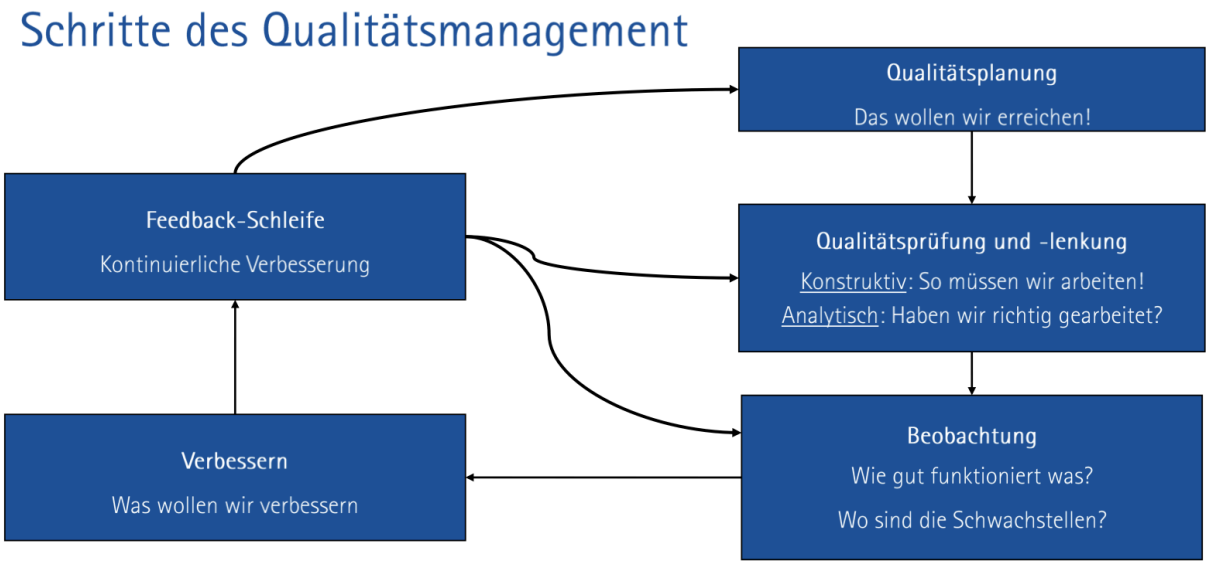
- Qualitätsmetrik
    - Maß für Ausprägung eines Q-Aspekts

Grundsätze der Qualitätsverantwortung

1. Wer für ein (SW-) Produkt verantwortlich ist, ist damit auch untrennbar für dessen Qualität verantwortlich d.h. der Verantwortung für Sachziele, Kosten, Termine
2. Jede/r Mitarbeiter/in ist im eigenen Arbeitsbereich persönlich verantwortlich für die Qualität der eigenen Arbeit
3. Qualitätsfachleute (QB) sind verantwortlich für richtig erbrachte Dienstleistungen um SW-Qualität (Prüfungen, QS-Maßnahmen etc.). Sie sind nicht verantwortlich für die Qualität der erstellten Software

Schritte des Qualitätsmanagements

1. Qualitätsplanung
   - Das wollen wir erreichen
2. Qualitätsprüfung und -lenkung
   - so müssen wir arbeiten (konstruktiv)
   - haben wir richtig gearbeitet (analytisch)
3. Beobachtung
   - Wie gut funktioniert was?
   - Wo sind die Schwachstellen?
4. Verbessern
   - Was wollen wir verbessern
5. Feedback-Schleife
   - Kontinuierliche Verbesserung
   - zurück zu Schritt 1, 2 oder 3

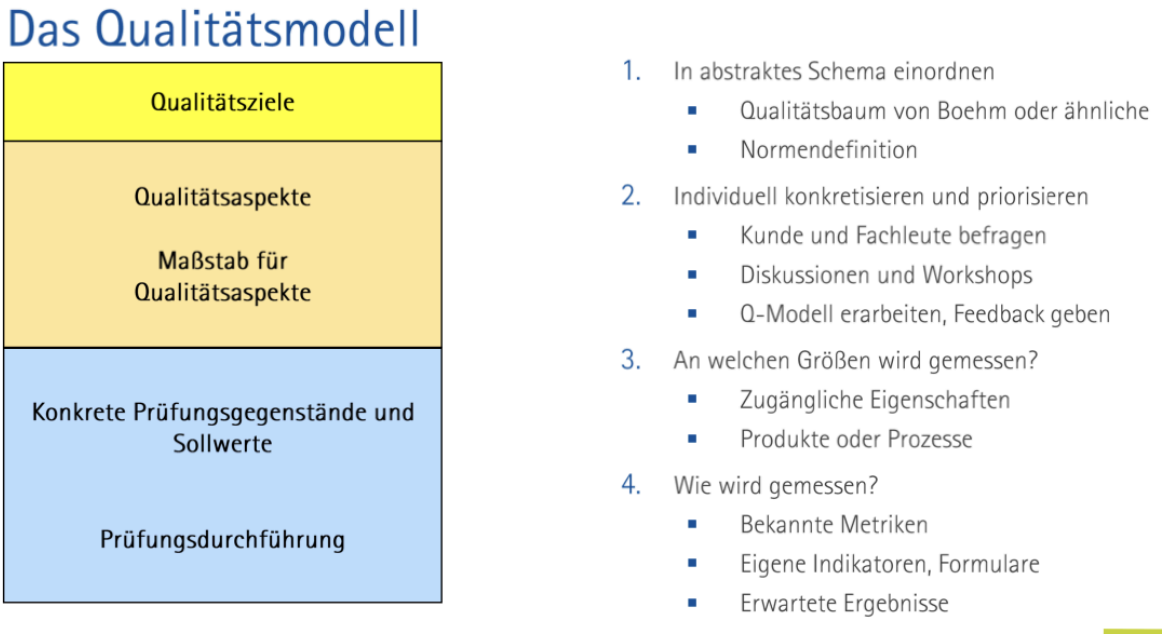


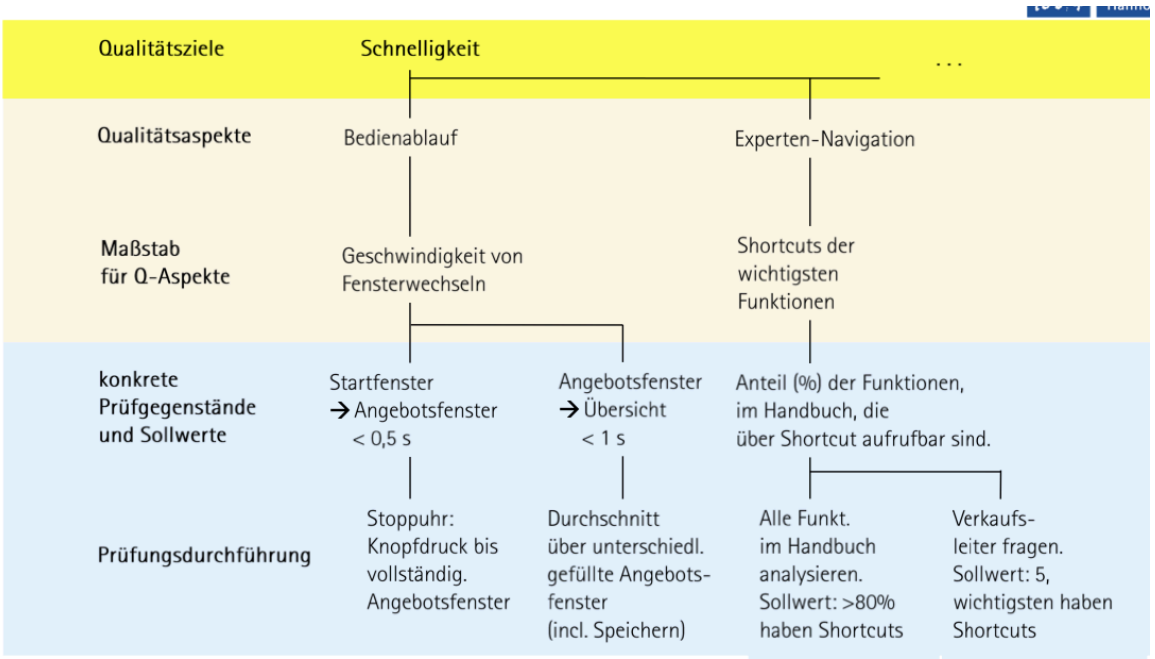
Qualitätssicherungsmaßnahmen

- Konstruktive Maßnahmen
    - Test- und Review-Management
    - Prozess- und Projektmanagement
    - Risikomanagement
    - Schulungen
    - Checklisten und Richtlinien
- Analytische Maßnahmen
    - Manuelle Verfahren: Reviews, Inspektionen
    - Automatische Verfahren: Modell-prüfung, Quelltextanalyse
    - Unit-Test
    - Integrationstest
    - Abnahmetest

Das Qualitätsmodell

1. Qualitätsziele (Abstrakt - Allgemeine Q-Ziele)
2. Qualitätsaspekte und Maßstab für Qualitätsaspekte (Konkret)
3. Konkrete Prüfungsgegenstände und Sollwerte; Prüfungsdurchführung (Messbar, Indikatoren und Metriken)





Aufgabenstellungen

1. Ihr Unternehmen soll eine Software für einen Ticketautomaten für den ÖPNV entwickeln. Dabei wird besonderer Wert auf Lesbarkeit gelegt. Entwickeln Sie zu diesem Qualitätsziel ein Qualitätsmodell
2. Ihr Unternehmen soll eine Anmeldefunktion (Login) für eine Versicherung erstellen. Dabei soll besonderer Wert auf Sicherheit gelegt werden. Entwickeln Sie zu diesem Qualitätsziel ein Qualitätsmodell.
3. Ihr Unternehmen soll eine Webseite mit Hilfe des Corperate Design neu designen. Der wichtigste Aspekt hierbei spielt die Benutzbarkeit (Usability). Entwickeln Sie zu diesem Qualitätsziel ein Qualitätsmodell

ÜB 3

Warum kann es sinnoll in der Softwareentwicklung sein, Metriken zu verwenden?

Erklärt anhand der Metrik LoC, warum es wichtig ist, Metriken konkret zu definieren.

Was misst der McCabe-Wert? -> Zyklomatische Komplexität

Messen

- Eigenschaften der realen Welt Zahlen oder Zeichen zuordnen

Maß

- Zuordnung einer Zahl oder Zeichens und einer Einheit (z.B.. 1m)

McCabe

- Abgeleitet aus dem Programmablaufgraph V(G) = e - n + 2, mit e = #Kanten, n = #Knoten
- Aus Code V(G) = 1 + #if-Bedingungen + #Schleifen + #Switch(+1 je case)
- Fehler Risiko
    - V(G) > 10 : mittel
    - V(G) > 20 : hoch
    - V(G) > 50 : unbeherrschbar
- Bei der Berechnung werden Anweisungen als ganzes betrachtet und boolesche Ausdrücke werden über die Bedingung aufgeteilt

Programmablaufgraph

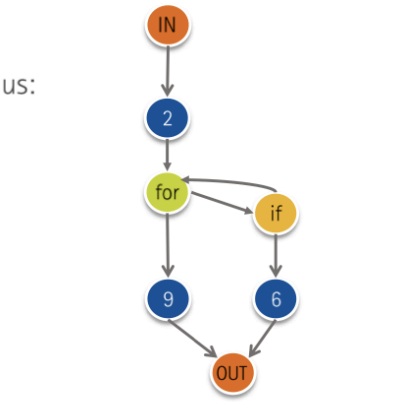
- Jede Anweisung ein Knoten
- Immer IN und OUT Knoten

McCabe-Wert in der Anwendung

- Berechnen Sie den McCabe-Wert für den folgenden Algorithmus:

```
1  public boolean isPrime(int number){
2          int root = (int) Math.sqrt(number) + 1;
3
4          for (int i = 2; i < root; i++){
5                  if (number % i == 0){
6                          return false;
7                  }
8          }
9          return true;
10 }
```

V(G) = 1 + 1 If–Bed. + 1 Schleife + 0 CASE = 3



V(G) = 8 Kanten - 7 Knoten + 2 = 3

ÜB 4

Metrik

- **Funktion, die eine Software-Einheit in einen Zahlenwert abbildet.** Dieser Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit. (IEEE-Standard 1061)
- Wertebereich und Interpretation
- Dieser Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.

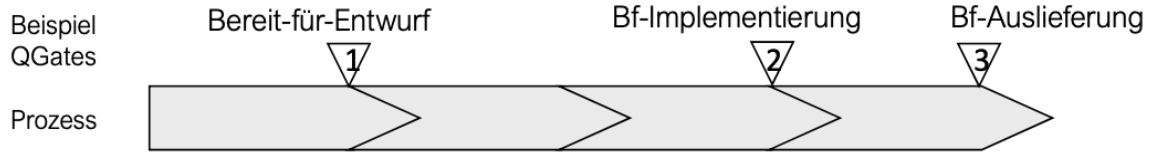
Skalen

- **Nominalskala**

- Gibt es unterschiedliche Werte?
    - Es gibt nur eine Äquivalenzrelation ≅
    - "Ist Programm A in gleicher Sprache geschrieben wie B?"
- **Ordinalskala**
    - Sind die Werte geordnet?
    - Es gibt eine Äquivalenzrelation ≅
    - Und es ist eine Ordnung definiert (z.B. "besser", <)
    - "Programm A wird besser benotet als Programm B"
- **Intervallskala**
- Sind die Abstände zwischen Folgewerten jeweils gleich?
    - Abstände sind von Bedeutung
    - Rechnen mit Intervallen ist erlaubt
    - "Projekt A ist halb so stark verspätet wie B"
- **Rationalskala**
    - Gibt es ein "nichts" (natürliche Null)?
    - Hat einen Nullpunkt, man darf Verhältnisse bilden (mult, div, durchschn, mittelw.)
    - Programm A hat doppelt so viele Getter wie B"

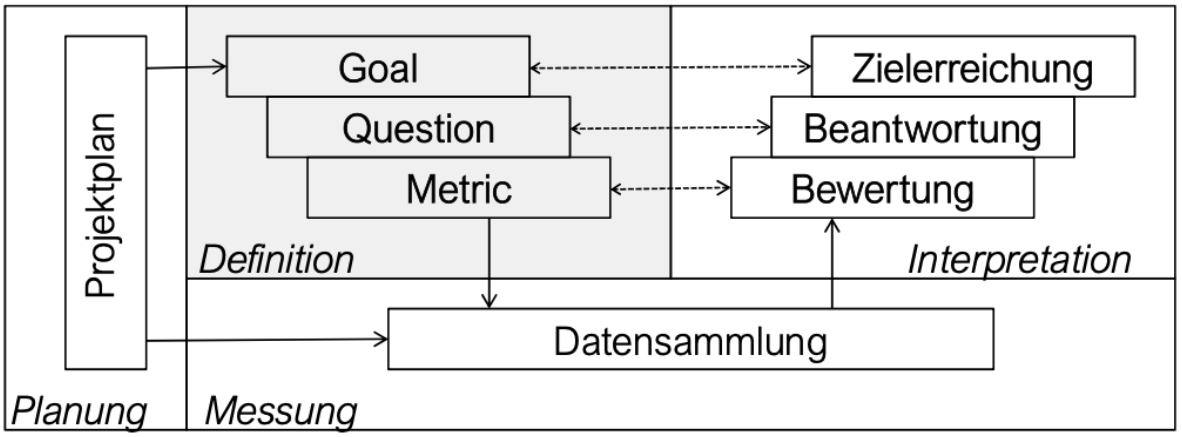
Quality Gates

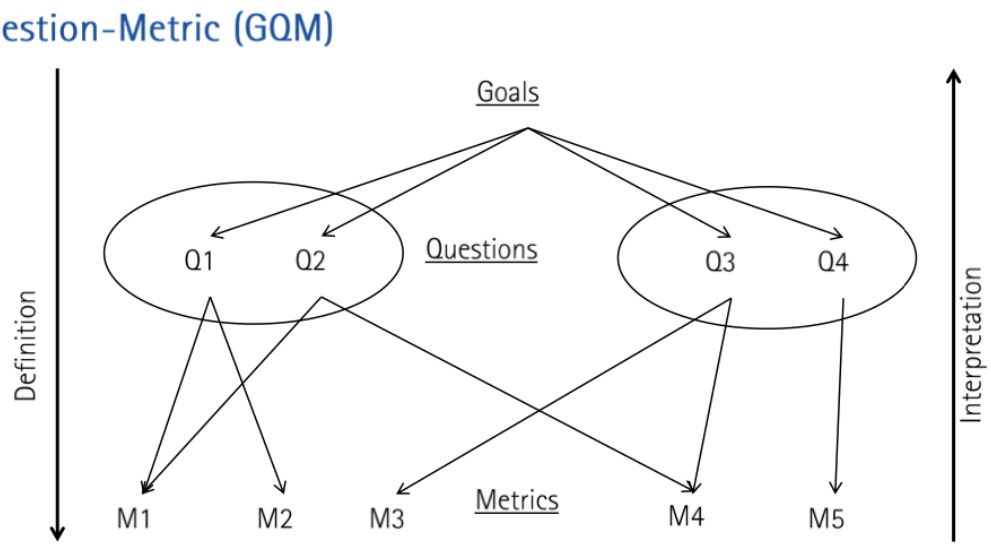
- Kurze, scharfe Prüfung an defineirten Prozessstellen
- Prüfkriterien: essentzielle Fortschrittsindikatoren
- Fortschrittsmessung: Passierte Q-Gates



Goal-Question-Metric (GQM)

- Messen, was man wissen will, nicht das, was leicht zu messen ist

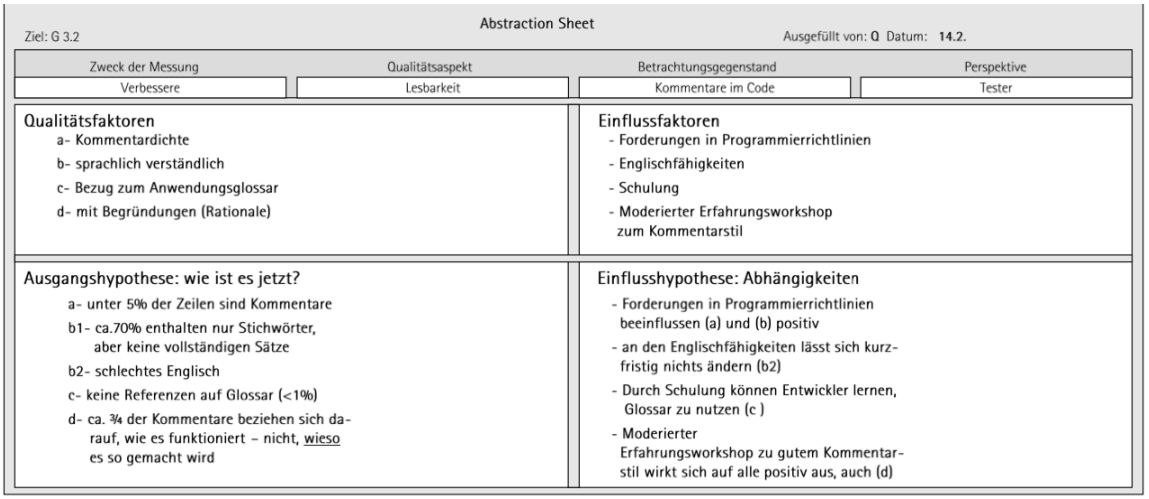




Zielbäume

- Ganz oben: Zweck
- Darunter Qualitätsaspekte
    - Für diese wird der Zweck unntersucht (iso 25010 oder andere)
- Zweck und Qualitätsaspekte werden mit einer Linie verbunden

Abstraction Sheet



Abstraction Sheet

Ziel: G 3.2 — Ausgefüllt von: Q Datum: 14.2.

Zweck der Messung	Qualitätsaspekt	Betrachtungsgegenstand	Perspektive
Verbessere	Lesbarkeit	Kommentare im Code	Tester

Qualitätsfaktoren
- a- Kommentardichte
- b- sprachlich verständlich
- c- Bezug zum Anwendungsglossar
- d- mit Begründungen (Rationale)

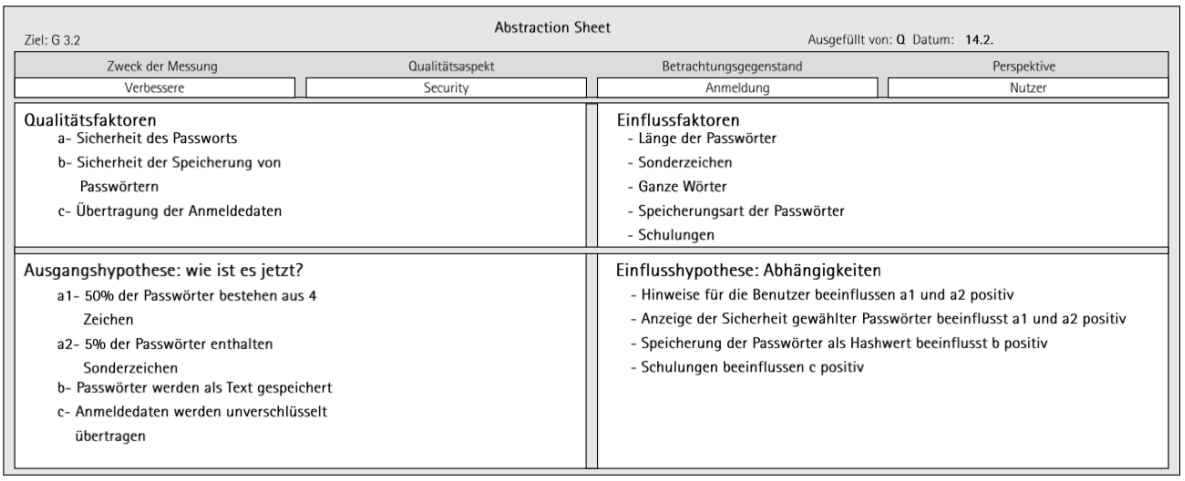
Einflussfaktoren
- Forderungen in Programmierrichtlinien
- Englischfähigkeiten
- Schulung
- Moderierter Erfahrungsworkshop zum Kommentarstil

Ausgangshypothese: wie ist es jetzt?
- a- unter 5% der Zeilen sind Kommentare
- b1- ca.70% enthalten nur Stichwörter, aber keine vollständigen Sätze
- b2- schlechtes Englisch
- c- keine Referenzen auf Glossar (<1%)
- d- ca. ¾ der Kommentare beziehen sich darauf, wie es funktioniert – nicht, wieso es so gemacht wird

Einflusshypothese: Abhängigkeiten
- Forderungen in Programmierrichtlinien beeinflussen (a) und (b) positiv
- an den Englischfähigkeiten lässt sich kurzfristig nichts ändern (b2)
- Durch Schulung können Entwickler lernen, Glossar zu nutzen (c)
- Moderierter Erfahrungsworkshop zu gutem Kommentarstil wirkt sich auf alle positiv aus, auch (d)

Aufgabenstellungen
Berechnen Sie den McCabe-Wert für den folgenden Algorithmus (Algorithmus gegeben)

GQM: Versetzen Sie sich in die Lage einer Firma, die eine Social Network Plattform betreibt. Es kommt immer wieder vor, dass Nutzerdaten in die Hände Dritter gelangen. Dies ist auf Sicherheitslücken in der Software zurückzuführen. Denken Sie sich ein Ziel für die SW aus, welches Sie betrachten wollen. Wenden Sie hierauf GQM an.
Beispiellösung



Abstraction Sheet

Ziel: G 3.2

Ausgefüllt von: Q Datum: 14.2.

Zweck der Messung	Qualitätsaspekt	Betrachtungsgegenstand	Perspektive
Verbessere	Security	Anmeldung	Nutzer

Qualitätsfaktoren

a- Sicherheit des Passworts

b- Sicherheit der Speicherung von Passwörtern

c- Übertragung der Anmeldedaten

Einflussfaktoren

- Länge der Passwörter
- Sonderzeichen
- Ganze Wörter
- Speicherungsart der Passwörter
- Schulungen

Ausgangshypothese: wie ist es jetzt?

a1- 50% der Passwörter bestehen aus 4 Zeichen

a2- 5% der Passwörter enthalten Sonderzeichen

b- Passwörter werden als Text gespeichert

c- Anmeldedaten werden unverschlüsselt übertragen

Einflusshypothese: Abhängigkeiten

- Hinweise für die Benutzer beeinflussen a1 und a2 positiv
- Anzeige der Sicherheit gewählter Passwörter beeinflusst a1 und a2 positiv
- Speicherung der Passwörter als Hashwert beeinflusst b positiv
- Schulungen beeinflussen c positiv

ÜB 5

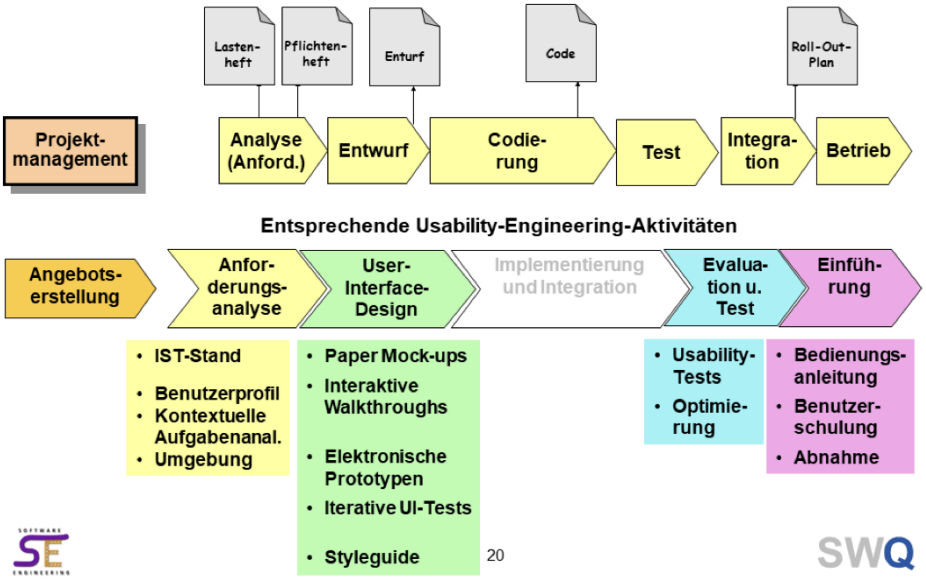
Ergonomie

- **Wissenschaft von der Anpassung der Arbeit an den Menschen**
- Software-Ergonomie
  - Verwendbarkeit der SQ, Anpassung an den Menschen
- Hardware-Ergonomie
  - Verwendbarkeit der HW, Anpassung an den Menschen
- Organisationsergonomie
  - Integration von HW und SW in den Arbeitsablauf der Organisation, unter Anpassung an den Menschen

Usability

- Der Grad, zu dem ein Produkt oder System durch **definierte Benutzer** verwendet werden kann, um **spezielle** Ziele (Effektivität, Effizienz, Zufriedenheit) in einem **bestimmten Nutzungskontext** zu erreichen.
- Ziele
  - **Effektivität**: Genauigkeit u. Vollständigkeit, mit der Benutzer best. Ziele erreichen
  - **Effizienz**: Die von den Benutzern aufgewendeten Mittel im Verhältnis zur Genauigkeit und Vollständigkeit, mit der die Benutzer ihre Ziele erreichen
  - **Zufriedenheit**: Die Freiheit von Unbehagen und Beschwerden sowie die positive Einstellung zur Nutzung des Produkts oder Systems

Usability im Entwicklungsprozess



SWQ

Verschiedene Usability-Ziele

Alle quantitativen Usability-Ziele
(incl. ease-of-use, ease-of-learning)

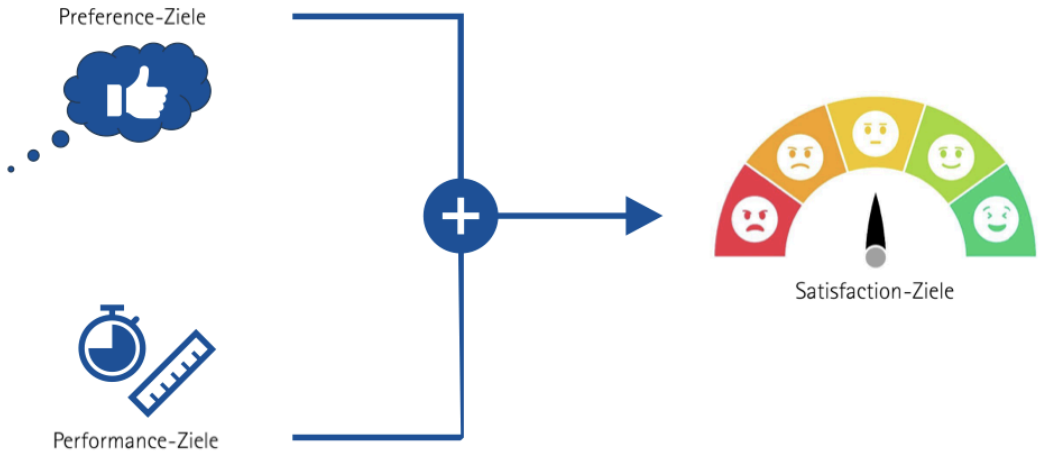
Absolutes Maß *objektiv* — *Relatives Maß* *subjektiv*

Performance-Ziele
quantifizieren die ***Leistung des Benutzers*** bei Erledigung einer Aufgabe.
Dabei werden Zeit und Fehler gemessen.

Preference-Ziele
Auf Basis der ***Vorliebe des Benutzers*** für eine von mehreren Oberflächenalternativen (Vorraussetzung: Benutzer hat damit Erfahrung).

absolut *subjektiv*

Satisfaction-Ziele
Vom Benutzer geäußerte (absolute) Zufriedenheit mit einer bestimmten Oberfläche.



Tools / Features

- Click and Mouse Tools
- Card Sorting/ AI Tools
- "Do it yourserl"
- A/B-Testing
- Website Recordings
- Expectancy Testing

Messbarkeit

- Performance-Ziele
    - Zeit, um eine Aufgabe abzuschließen
    - Anzahl und Arten von Fehlern pro Aufgabe
    - Anzahl von Fehler in einer Zeitspanne
    - Anzahl Klicks für eine Aufgabe
- Preference-Ziele
    - Interviews / Umfragen
    - Eye Tracking Studie
- Satisfaction-Ziele
    - Interviews
    - Umfragen
    - Eye Tracking-Studie

Kontextuelle Aufgabenanalyse

- Betonung auf Aufgabe aus Sicht des Akteurs und auf den Kontext, Ziele, …
- Verstehen, wobei Benutzer unterstütz werden soll; Das geht tiefer als ein Use Case

1. **Wann** wird die Aufgabe durchgeführt (Auslöser, Vorbedingung)?
2. **Von wem** wird die Aufgabe durchgeführt?

3. **Warum** wird die Aufgabe durchgeführt (Handlungsziel, Nachbedingung)?
4. **Wie oft** wird die Aufgabe durchgeführt?
5. Was ist bei der Durchführung im einzelnen zu tun?
6. Welche zusätzlichen Mittel werden benötigt?
7. Wünsche/Anregungen der Nutzer für Dialoggestaltung?
8. Wie sehen erste Gestaltungsideen aus?
9. Welche Abweichungen von Normalvorgehensweise sind nötig?

Kontextuelle Aufgabenanalyse vs. Use Case

Use Case ● Keine UI Details ● Nur funktionale Anforderungen ● Beschreibt Systeminteraktion als einzelne Funktionen ● Abstaktes Konzept	Kont. Aufg. Analyse ● Betonung auf Aufgabe aus Sicht des Akteurs ● Betonung des Kontexts (Erwartungen, Fähigkeiten) ● Ziele, Zeitdruck, Fehlertoleranz je nach Kontext

Experten-Evaluation

1. Orientierung im System
2. Grundlegende Regeln (z.B. Acht Goldene)
3. Einfache Aufgabe (vorher gestellt)
4. Beobachtung
5. Präsentation

Acht Goldene Regeln für Dialogdesign

1. Strebe nach Konsistenz
2. Biete geübten Benutzern Abkürzungen an
3. Biete informatives, angemessenes Feedback
4. Vermittle das Gefühl, den Dialog abzuschließen
5. Erlaube einfache Fehlerbehandlung
6. Erlaube, Aktionen einfach zurückzunehmen
7. Vermittle den Benutzern das Gefühl, die Kontrolle zu haben
8. Beanspruche das Kurzzeitgedächtnis so wenig wie möglich

ÜB 6

Security als Q-Aspekte (Maßnahmen um sichere System zu entwickeln)

- Security-Anforderungen kennen und achtsam umsetzen
- Angriffsszenarien kennen und dagegen schützen (STRIDE)
- Entwickler schulen und unterstützen
- Verdächtige Muster im Systemverhalten erkennen
- Use-Cases auf Codeebene modellieren und tatsächlichen Ablauf damit abgleichen
- Richtig über Security reden (bspw. Einheitliches Vokabular verwenden)

Security (iso 25010)

> Degree to which a product or system **protects** information and data so that persons or other products or systems have the **degree of data access appropriate to their types and levels of authorization**.

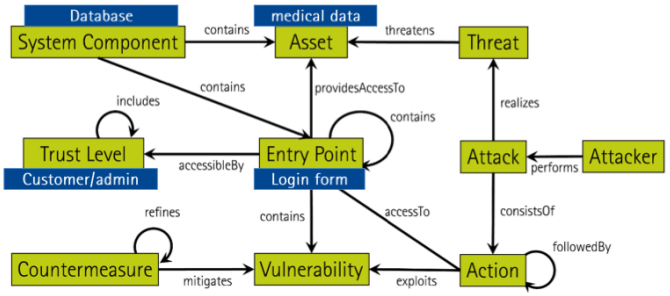
STRIDE

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of Service
- Elevation of privilege

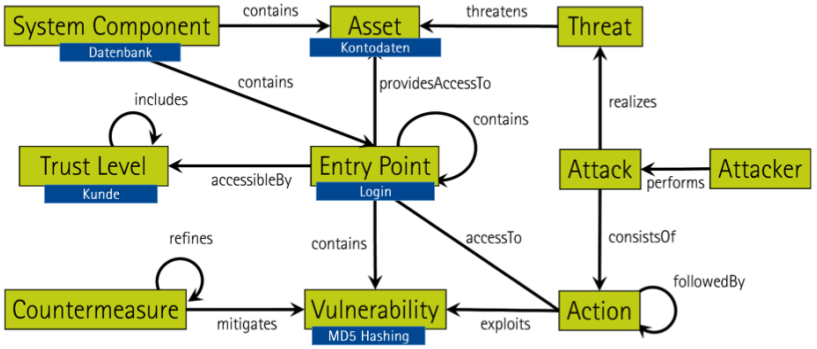
Ontologie Security

- Terminologie verwenden:
  - R17: Der Zugriff auf das Asset medical data in der System Component Database über den Entry Point Login form soll für den Trust Level Customer nicht möglich sein.
  - R18: Für Trust Level admin soll der Zugriff (R17) möglich sein.

„Unbefugte dürfen nicht eindringen können" schwammig



Formulieren Sie nun eine Qualitätsanforderung, basierend auf diesem Beispiel



R1: Der Zugriff auf das Asset Kontodaten in der System Component Datenbank über den Entry Point Login soll für das Trust Level Kunde nicht möglich sein.

R2: Die Vulnerability MD5-Hashing durch welche auf das Asset Kontodaten über den Entry Point Login zugegriffen werden kann soll behoben werden

Q-Aspekt Erklärbarkeit

- Komplexe Software soll erklärt werden können (dabei ist meist gefordert, dass die SW diese Erklärung selbst liefert)
- U.a. besonders interessant im Bereich AI (Blackboxes)
- Erklärbarkeit wird nicht um ihrer selbst Willen gefordert, sondern um unabhängige Ziele zu erreichen
  - Security -> SW soll sicher gegen Angriffe sein
  - Usability -> SW soll besonders gut nutzbar sein
  - Erklärbarkeit -> SW soll gut verständlich sein, oder effizient nutzbar, oder transparent, oder vertrauenswürdig (kein Selbstzweck)
  - Correctness, Suitability, Trusability

Testen

- **Ausführen eines Programms mit dem Ziel Fehler zu finden**
    - Nicht der Wunsch zu zeigen, das ein Programm funktioniert
- "Rumprobieren" ist kein Test!
- Ausführen und "schauen ob es läuft" kein Test, ein Laufzeitversuch

Abnahmetest

- Bei Übergabe der Software an den Kunden wird geprüft, ob die vorher festgelegten Funktionen/Aspekte fehlerfrei funktionieren - entscheidend für den Projektabschluss
- Hier will man keine Fehler finden
- Abnahmetests werden meist in der Spezifikation festgehalten und sind dem Entwicklungsteam bekannt

Fehler

- Unterschied zwischen erwartetem (Soll) und tatsächlichem Ergebnis (Ist) eines Tests

Prüfling

- Das Objekt, welches auf Fehler untersucht wird (z.B. die Software)

Testvorschrift

- Anweisung zur Abwicklung eines Tests (inkl. Setup)

Testprotokoll

- Standardisiertes Formular zur Dokumentation von gefundenen Fehlern

White-box Test

- Code ist bekannt -> Wird gesamter Code ausreichend abgedeckt?

Black-box Test

- Code ist unbekannt -> Werden alle Anforderungen ausreichend getestet?

ÜB 7

Abnahmetest

Szenario II

Finden Sie die vier Fehler im folgenden Abnahmetestfall.

Setup: Das System befindet sich im Hauptmenü und wartet auf eine Eingabe. Der Nutzer „Kurt Schneider" möchte sich im Kalender anmelden und einen neuen Termin mit dem Namen „Vorlesung: SWQ" für Montag, den 20.06.2022 für 13:00 Uhr eintragen.

Nr.	Eingabe	Ausgabe
1	Der Nutzer gibt das Passwort (welches?) ein	Das System maskiert die Eingabe mit „*******" und zeigt diese im Textfeld an.
2	Der Nutzer gibt den Benutzernamen „Kurt Schneider" ein.	Das System zeigt den Namen „Max Mustermann" im Textfeld an.
3	Der Nutzer klickt auf den Button anmelden.	Das System gibt die Nachricht „Willkommen, Kurt Schneider" aus
4	Der Nutzer klickt auf ein beliebiges Feld in der Kalenderübersicht	Das System öffnet ein neues Fenster mit einer Vorschau des ausgewählten Datums
5	Der Nutzer gibt in dem Feld „Titel" „Vorlesung: SWQ" ein	Das System schreibt auf dem Feld „Titel" „Vorlesung: SWQ"
6	Der Nutzer klickt auf „beenden"	Das Fenster schließt sich und im Kalender ist der eingetragene Termin mit den Einträgen Titel: „Vorlesung: SWQ" Datum: 20.06.2022 (woher?) Uhrzeit: 13:00 Uhr (woher?) zu sehen

Testplan

- Testfälle
- Testmethoden (Black -/White-Box-test, Mutationstests, …)
- Konfiguration der Testplattform
- Bereitstellung des Testgeschirrs ("Hilfsgerüst")
- Kriterien für den Start der Testdurchführung
- Kriterien für Testunterbrechung
- Kommunikationswege

Testklassifikation

- Unit-Test
  - Eine Unit ist ein kleinster Teil eines Programms, welcher groß genug ist, um ihn zu testen (z.B. einzelne Funktion oder Klasse)
- Integrationstest
  - Testet Softwarekomponenten, die sich wiederum aus mehreren Komponenten zusammensetzen
- Systemtest
  - Alle SW- und HW-Komponenten zusammengebaut, testet das System als Ganzes
- Abnahmetest
  - Auch auf Systemebene, werden meist vom Kunden und nicht Hersteller durchgeführt, um Vertragserfüllung zu überprüfen

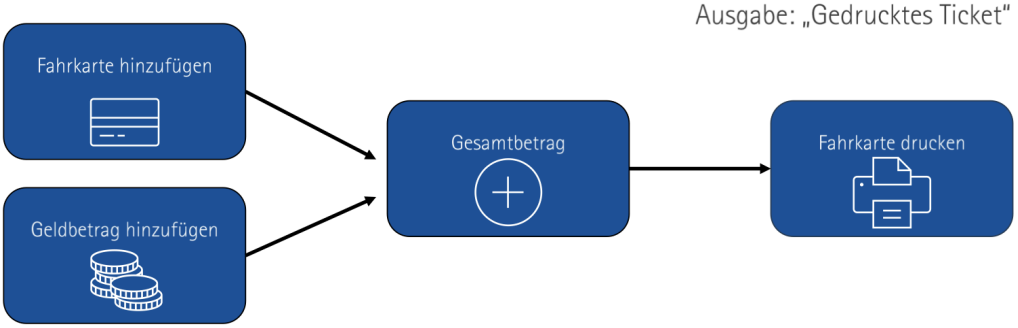
Testklassifikation – visuelles Beispiel

Integrationstest

Schreiben von Testfällen, die mehrere Komponenten und deren Abhängigkeit testen:

Z.B. Eingabe: 10€, 5€, Tagesticket

Ausgabe: „Gedrucktes Ticket"

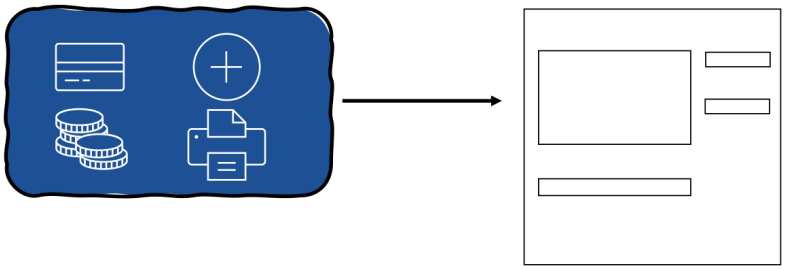


Testklassifikation – visuelles Beispiel

Systemtest

Software und zugehörige Hardware direkt testen.

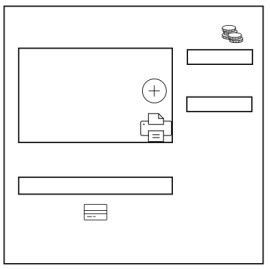
Z.B.: Kunde drückt auf dem Display auf Karte hinzu fügen -> Karte wird hinzugefügt



Testklassifikation – visuelles Beispiel

Abnahmetest

Komplettes System wird getestet. Hierbei soll die Gesamtfunktionalität bewiesen und abgenommen werden.



Z.B.

Eingabe	Ausgabe
Der Kunde drückt auf den Button „Tageskarte hinzufügen"	Das System fügt die Tageskarte dem Warenkorb hinzu und ergänzt den Preis von 15€ in der Gesamtbetragsübersicht.

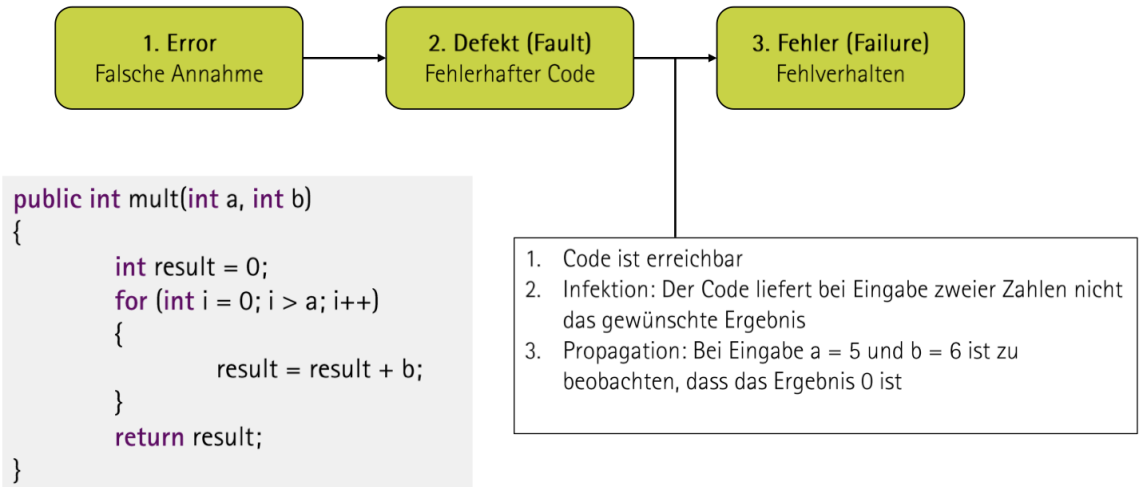
(Nur ein Ausschnitt des gesamten Abnahmetestfalls)

Definitionen

- **Error**
  - Denkfehler, falsche Annahme
- **Defekt** (Fault)
  - Eine fehlerhafte Stelle im Programmcode (statisch)
- **Fehler** (Failure)
  - Ein von außen beobachtbares Verhalten eines Programms, welches vom spezifizierten Verhalten abweicht

Was muss erfüllt sein, damit ein Defekt zu einem Fehler führt?

- **Erreichbarkeit**: Die Stelle des Defektes im Programm muss erreicht werden
- **Infektion**: Der Zustand des Programms muss daraufhin inkorrekt sein
- **Propagation**: Der inkorrekte Zustand muss zu einer inkorrekten Ausgabe führen (also beobachtbar sein)



Junit

- Ist ein Unit-Testing-Framework für Java
- Werkzeug zum Test-Driven-Development
- GWT: Given-When-Then
- Immer Setup, Input, Soll-Output

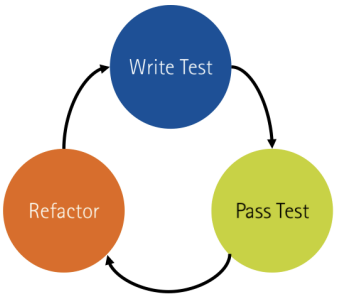
Test First Programming (Test-Driven Development)

- Bevor Veränderungen im Code vorgenommen werden, werden Testfälle geschrieben
- Vorteile
  - Nur benötigtes wird programmiert
  - Funktionalität kann sofort geprüft werden
  - Man hält sich an einen strikten Ablauf
- Nachteile

- Es werden nur Teilfunktionen betrachtet und nicht das "Ganze"

Was ist Test-Driven Developement?

1. Schreibe einen Testfall
2. Lass den Test laufen
3. Schreibe genug Code, damit der Testfall erfüllt ist
4. Lass alle Tests laufen
5. Refactor
6. Wiederhole



ÜB 8

Anforderungsüberdeckung

- Ziel: viele mögliche Fehler durch wenige Testfälle abdecken

1. Alle Anforderungen überdecken also je ein Testfall
2. Annahme: Ähnliche Eingaben finden ähnliche Fehler
3. Eingaben in Äquivalenzklassen einteilen
4. Dann aus jeder Äquivalenzklasse nur einen Repräsentanten nehmen

Äquivalenzklassentests

- Ein Programm reagiert bei der Verarbeitung eines Wertes aus einem bestimmten Bereich genau so wie bei der Verarbeitung jedes anderen Wertes aus diesem Bereich.
- Annahme: Ähnliche eingaben finden ähnliche Fehler
- Es ist also ausreichend aus jedem Bereich einen Wert abzuprüfen
- **Zerlege Wertebereich** der Eingabeparameter oder Definitionsbereiche der Ausgabeparameter in Äquivalenzklassen
- Dann aus jeder Äquivalenzklasse **einen Repräsentanten** nehmen

Vorgehen Äkt

1. Äquivalenzklassen pro Parameter ermitteln
    1. Wertebereich der Eingabeparameter zerlegen
    2. ODER Definitionsbereich der Ausgabeparameter zerlegen (selten)
2. Davon ist immer nur ein Repräsentant notwendig
3. Repräsentanten aller Parameter geschickt kombinieren mit Testdaten

Aufstellen der ÄK

Stellen Sie Äquivalenzklassen für die Eingabeparameter auf! Dabei brauchen Sie nur gültige Werte zu berücksichtigen, also nicht-negative Zahlen vom richtigen Typ.

- Die persönliche Einkommenssteuer von Steuerpflichtigen soll ermittelt werden. Dazu dient die folgende Methode:

```
public Euro steuernZuZahlen (Euro einkommen, int kinderzahl) {...}
```

- An die Methode werden die Anforderungen gestellt:

R01: Unter 20.000 Euro sind keine Steuern zu bezahlen.
R02: Zwischen 20.000 und 50.000 Euro fallen 20% an. Das ist der *Sockelbetrag*.
R03: Für alle Einkommen über 50.000 Euro sind pauschal 40% zu zahlen.
R04: Je Kind werden 5 Prozentpunkte vom *Sockelbetrag* abgezogen (nicht von den 40%).
Beispiel: Bei zwei Kindern ist der Sockelbetrag noch 10% bei einem Einkommen zwischen 20.000 und 50.000 Euro.
R05: Es darf kein negativer Steuersatz entstehen.

Eingabe	Äquivalenzklasse		Anforderung
	Benennung	Beschreibung	
Einkommen	Ä1	Einkommen < 20.000 EUR	R01
	Ä2	20.000 EUR <= Einkommen <= 50.000 EUR	R02
	Ä3	Einkommen > 50.000 EUR	R03
Kinderzahl	Ä4	0 <= Kinderzahl < 5	R04
	Ä5	Kinderzahl >= 5	R05

Stellen Sie die zweidimensionale Äquivalenzklassenmatrix auf. Welche dieser Äquivalenzklassen kann man kombinieren bzw. überschneiden sich?

ÄK-Kombination

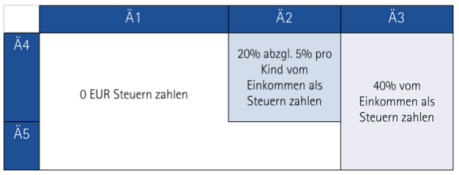
- Kombination der Eingabe-Äquivalenzklassen:

<table>
<tr><th></th><th>Ä1</th><th>Ä2</th><th>Ä3</th></tr>
<tr><td>Ä4</td><td rowspan="2">0 EUR Steuern zahlen</td><td>20% abzgl. 5% pro Kind vom Einkommen als Steuern zahlen</td><td rowspan="2">40% vom Einkommen als Steuern zahlen</td></tr>
<tr><td>Ä5</td><td></td></tr>
</table>

Äquivalenzklassentests

- Kombination der Eingabe-Äquivalenzklassen ->

Geben Sie drei vollständige Testfälle an, die zusammen möglichst viele Ihrer Äquivalenzklassen abdecken. Nennen Sie zu jedem Testfall die davon abgedeckten Äquivalenzklassen.



ID	Einkommen	Kinderzahl	Sollwert	Geprüfte ÄK
1	15.000 EUR	1	0 EUR	Ä1, Ä4
2	30.000 EUR	2	3.000 EUR	Ä2, Ä4
3	60.000 EUR	5	24.000 EUR	Ä3, Ä5

Mehrdimensionale Äquivalenzklassenbildung

Neue Anforderung, neue Variante

Daher jetzt drei Parameter:

`endTime (startTime, startDay, centsPaid)`

Parameter1: `startTime`

1. 0-9h: kostenlos
2. 9-19h: kostenpflichtig
3. 19-0h: kostenlos

Parameter3: `centsPaid`

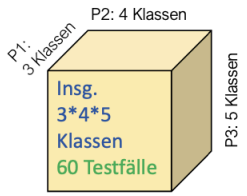
1. < 50 c
2. >= 50 und <= 120, teilbar durch 10
3. >120 und <= 240, teilbar durch 10
4. > 240, teilbar durch 10
5. Nicht teilbar durch 10

Anforderungen:
R07: Übertrag von Restgeld auf Folgetag
R08: Sonntags kostet Parken nichts

Parameter2: `startDay`

1. Montag-Freitag
2. Samstag, ohne Überlauf
3. Samstag, mit Überlauf
4. Sonntag, alles ist Überlauf

Anzahl der Kombinationen



Minimalforderung und Effizienzprinzip

- Testfälle sollen dem Minimal- und dem Effizienzprinzip genügen
  - Minimalforderung: Jede Anforderung durch min. einen Testfall abgedeckt
  - Effizienzprinzip: Möglichst wenig Testfälle erstellen und je Testfall möglichst mehrere unterschiedliche Anforderungen abdecken
- Abdeckung der Anforderungen im Beispiel

Testfall	R01	R02	R03	R04	R05
1	x			x	
2		x		x	
3			x		x

Wie viele Testfälle sollten mindestens für einen Äquivalenzklassentest aufgestellt werden, um dem Minimal- und dem Effizienzprinzip zu genügen?

- Maximum der Anzahl der ÄK pro Eingabeparameter (max(#äkp1, #äkp2, …, #äkpn))

Wie viele Testfälle sind minimal aufzustellen, um alle Kombinationen der Äquivalenzklassen abzudecken?

- Multiplikation der Anzahl der ÄK pro Eingabeparameter (PROD_i_n(#äkpi Testfälle))

Aufgabe zur Äquivalenzklassenmethode

```
public Euro FinSum (Postenliste liste) {...}
```

An die Methode werden die Anforderungen gestellt:

R01: Wenn der Cent-Wert der gesamten Rechnung 0 oder 5 ist, so ist der Rechnungsbetrag die Summe.
R02: Wenn der Cent-Wert < 5 ist, so muss abgerundet werden.
R03: Wenn der Cent-Wert > 5 ist, so muss aufgerundet werden.
R04: Es darf nur die Rechnungssumme gerundet werden, nicht die einzelnen Posten.

Stellen Sie Äquivalenzklassen für die Eingabe und Ausgabe auf

Äquivalenzklassenbildung

- Eingabe-Äquivalenzklassen
  - CentBetrag=0 || CentBetrag=5 nicht runden
  - 1 ≤ CentBetrag ≤ 4 abrunden
  - 6 ≤ CentBetrag ≤ 9 aufrunden
- Repräsentanten
  - 10,05€ Nur zwei möglich Repräsentanten
  - 0,87€ liegt nicht direkt an der Grenze
  - 12,33€ liegt nicht direkt an der Grenze
- Ausgabe-Äquivalenzklassen
  - x,y0€
  - x,y5€
- Repräsentanten
  - 12,55€, 13,00€

Was sind funktionale Tests?

- Überprüfung, ob ein Subjekt bei bestimmten Eingaben auch wie spezifiziert Ausgaben erzeugt (die meisten Tests sind solche)

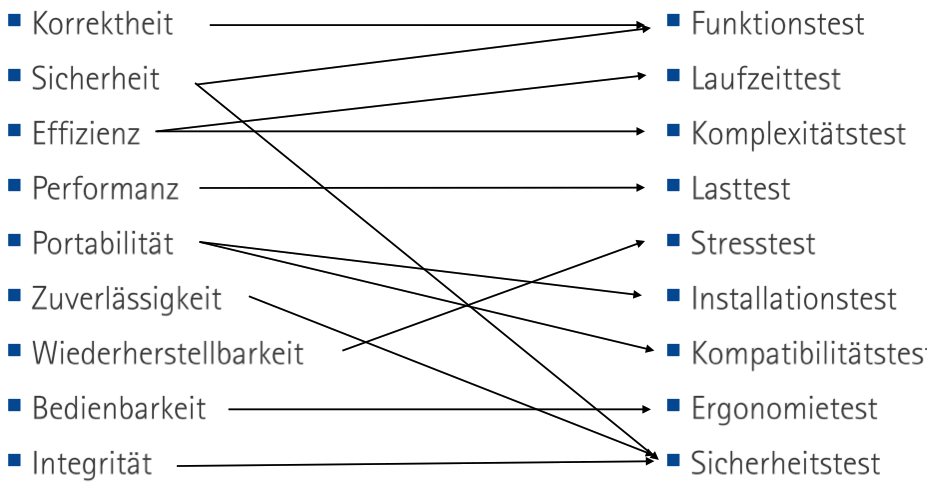
Temporale Tests

- Laufzeittests: Testen von Berechnungsdauern mit Stoppuhr
- Komplexitätstests: Ist Speicher-/Zeitverbrauch relativ zur Größe der Eingabe wie erwartet?
- Lasttests: testen Verhalten eines Systems bei vielen Anfragen
- Stresstests: Testen mit zu vielen Anfragen; erholt sich das System?

Operationale Tests

- Sonstige Tests zum reibungslosen Betrieb des Systems
  - Installationstests
  - Kompatibilitätstests
  - Ergonomietests
  - Sicherheitstests

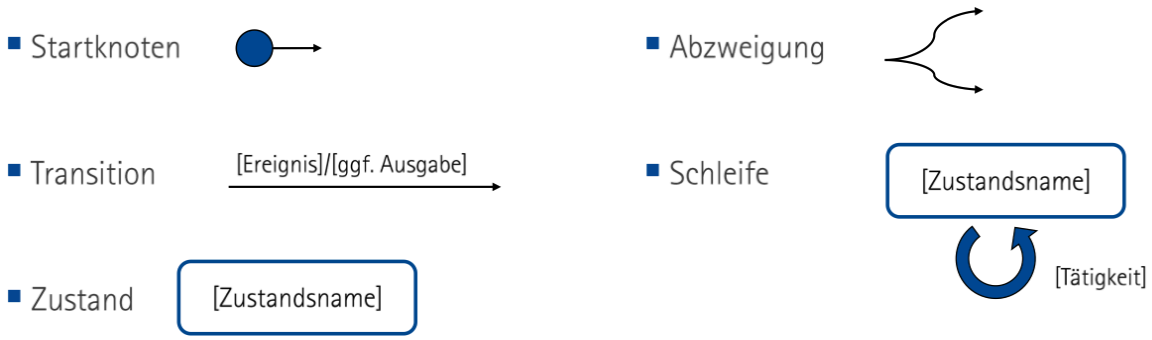
Q-Merkmale und verschiedene Arten von Tests



ÜB 9

Zustandsbasierte Tests

Zustandsübergangsdiagramme

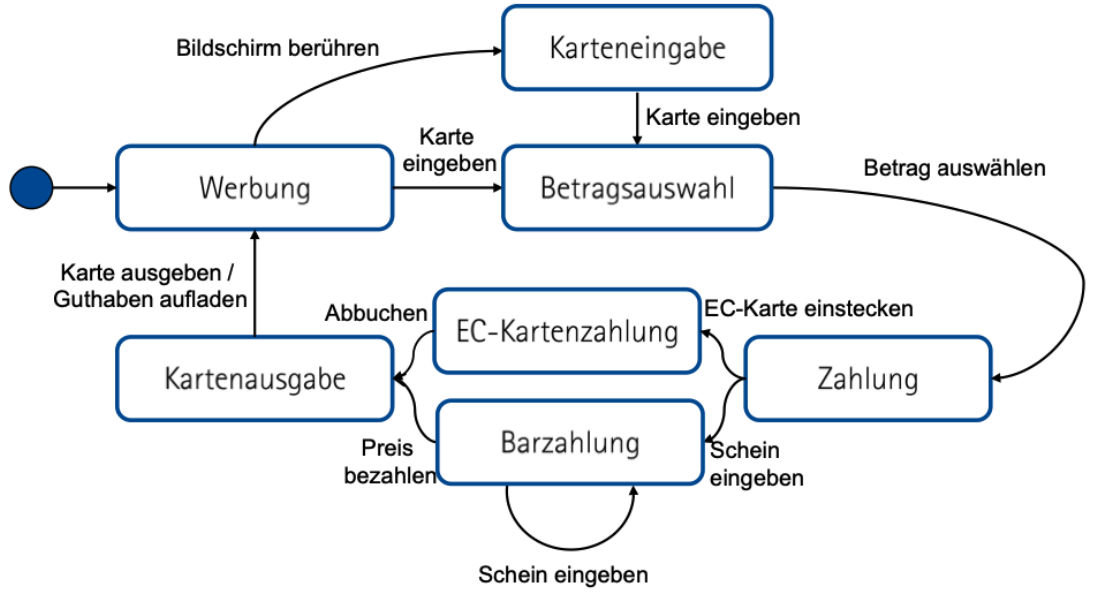


Sie sollen ein System zur Mensakartenaufwertung testen.

Im Ruhezustand soll das System Werbung des Studentenwerkes anzeigen. Der Student kann entweder direkt seine Karte eingeben und kommt dann zur Auswahl des Betrags zum Aufladen (10€ oder 20€) oder er berührt den Bildschirm, wird zum Eingeben seiner Mensakarte aufgefordert und kommt anschließend erst zur Betragsauswahl.

Die Bezahlung des Betrags kann bar per Geldschein oder mit EC-Karte erfolgen. Nach Bezahlung des Betrags wird die Karte automatisch ausgegeben und das System zeigt wieder Werbung.

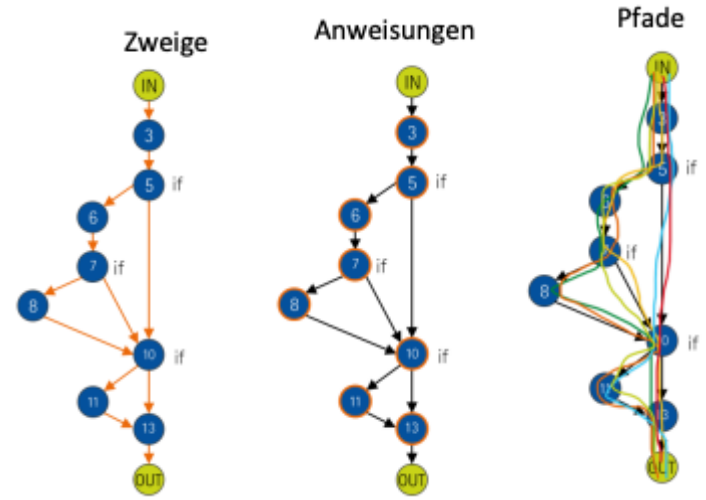
Zeichnen Sie zu dem System ein Zustandsübergangsdiagramm.

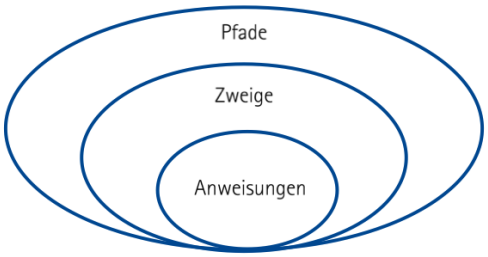


	Berühren	Karte eingeben	Betrag auswählen	EC-Karte einstecken	Schein eingeben	Preis bezahlen	Abbuchen	Karte ausgeben
Werbung	Karteneingabe	Betragsauswahl						
Karteneingabe		Betragsauswahl						
Betragsauswahl			Zahlung					
Zahlung				Kartezahlung	Barzahlung			
EC-Kartenzahlung							Kartenausgabe	
Barzahlung					Barzahlung	Kartenausgabe		
Kartenausgabe								Werbung / Guthaben aufladen

Kontrollflussgraph

- Expandiert: pro Anweisung ein Knoten
- Kollabiert: Verzweigungsfreie Blöcke werden zu einem Knoten





Code Coverage

- Anteil der mit Tests ausgeführten Programmteile

Überdeckung

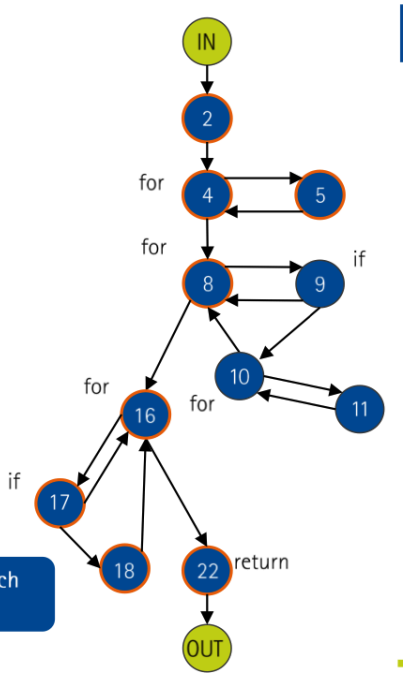
- Zweige
  - Jede Kante (edge) überdecken
- Anweisungen
  - Jeden Knoten (node) überdecken
- Pfade: Jeden möglichen Pfad überdecken
  - Pfad vs: Zweig -> Zweig ist lokal: ein Pfeil; Pfad ist eine Folge von in bis out

Kontrollflussgraphen

```
1  public void printPrimeNumbersUpTo(int n) {
2          boolean[] prime = new boolean[n + 1];
3
4          for (int i = 0; i <= n; i++) {
5                  prime[i] = true;
6          }
7
8          for (int p = 2; p * p <= n; p++) {
9                  if (prime[p] == true) {
10                         for (int i = p * 2; i <= n; i += p) {
11                                 prime[i] = false;
12                         }
13                 }
14         }
15
16         for (int i = 2; i <= n; i++) {
17                 if (prime[i] == true) {
18                         System.out.print(i + " ");
19                 }
20         }
21
22         return;
23 }
```

Für einen Testfall mit der Eingabe 3, wie hoch ist die Anweisungsüberdeckung?

8 / 11 = 72,72%

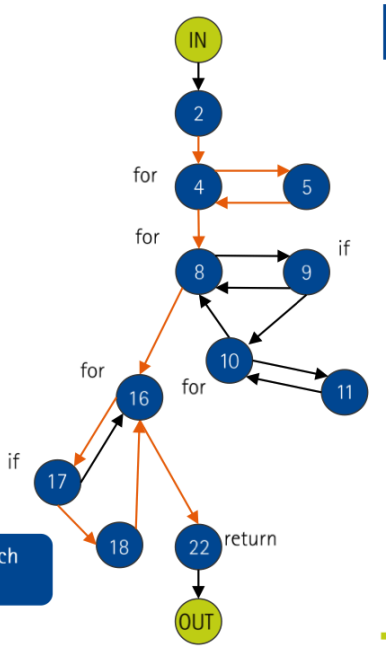


Kontrollflussgraphen

```
1  public void printPrimeNumbersUpTo(int n) {
2          boolean[] prime = new boolean[n + 1];
3
4          for (int i = 0; i <= n; i++) {
5                  prime[i] = true;
6          }
7
8          for (int p = 2; p * p <= n; p++) {
9                  if (prime[p] == true) {
10                         for (int i = p * 2; i <= n; i += p) {
11                                 prime[i] = false;
12                         }
13                 }
14         }
15
16         for (int i = 2; i <= n; i++) {
17                 if (prime[i] == true) {
18                         System.out.print(i + " ");
19                 }
20         }
21
22         return;
23 }
```

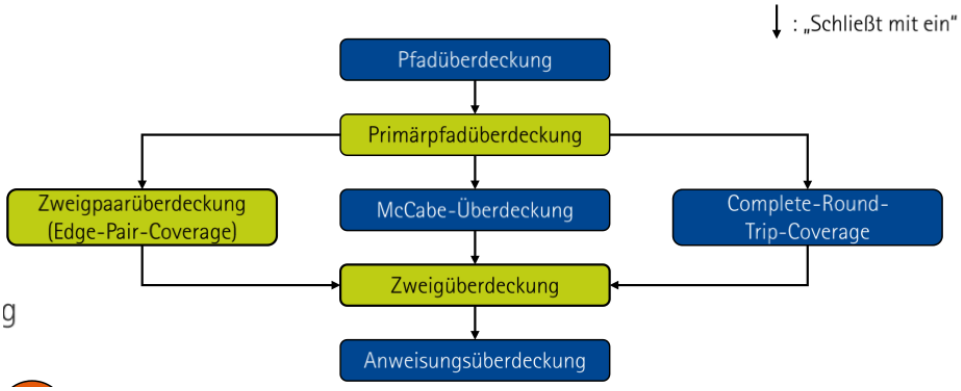
Für einen Testfall mit der Eingabe 3, wie hoch ist die Zweigüberdeckung?

9 / 16 = 56,25%



Fehlender Code wird auch bei 100% Abdeckung nicht gefunden

ÜB 10



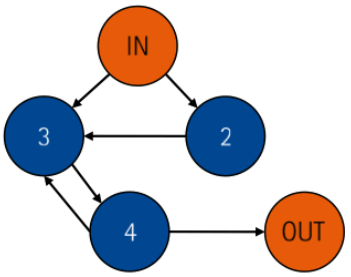
Edge-Pair-Coverage

- Variante von Pfadüberdeckung, bei der alle Kantenpaare überdeckt werden sollen
- Jedes Paar aufeinanderfolgender Kanten überdecken

Primärpfadüberdeckung

- Variante von Pfadüberdeckung, bei der alle Primärpfade überdeckt werden
- Ein **einfacher Pfad** in einem Kontrollflussgraphen ist ein Pfad zwischen zwei Knoten des Graphen
    - In dem kein Knoten zweimal vorkommt
    - Ausnahne: Start- und Endknoten können identisch sein
- Ein **Primärpfad** ist ein maximaler einfacher Pfad
    - Kann nicht zu einem anderen einfachen Pfad erweitert werden

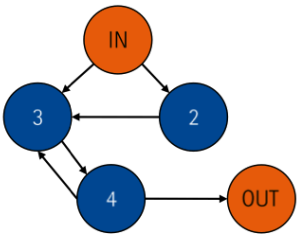
Aufgabe:
Stellen Sie eine minimale Testpfadmenge auf, die alle Primärpfade überdeckt



Beispielaufgabe – Lösung

- Maximale Primärpfade suchen:

(welcher Pfad ist nicht Teilpfad eines anderen?)



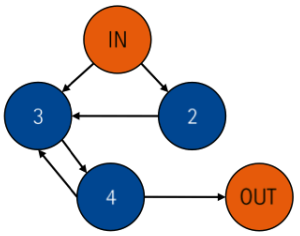
Länge 0	Länge 1	Länge 2	Länge 3	Länge 4
P1=(IN)	P6=(IN,2)	P12=(IN,2,3)	P18=(IN,2,3,4)	P21=(IN,2,3,4,OUT)!
P2=(2)	P7=(IN,3)	P13=(IN,3,4)	P19=(IN,3,4,OUT)!	
P3=(3)	P8=(2,3)	P14=(2,3,4)	P20=(2,3,4,OUT)!	
P4=(4)	P9=(3,4)	P15=(3,4,3)*		
P5=(OUT)!	P10=(4,3)	P16=(3,4,OUT)!		
	P11=(4,OUT)!	P17=(4,3,4)*		

P15=(3,4,3)*
P17=(4,3,4)*
P19=(IN,3,4,OUT)!
P21=(IN,2,3,4,OUT)!

*: Kann nicht erweitert werden, da Anfangs- gleich Endknoten
!: ...da Endknoten keine ausgehende Kante hat

Beispielaufgabe – Lösung

- Die Primärpfade zu Testpfaden erweitern
  - Erweitern zu Start- und Endknoten des Kontrollflussgraphen
  - Mit dem längsten Primärpfad anfangen



~~P15=(3,4,3)*~~
~~P17=(4,3,4)*~~
~~P19=(IN,3,4,OUT)!~~
~~P21=(IN,2,3,4,OUT)!~~

Pt1=(IN,2,3,4,OUT)
Pt2=(IN,3,4,OUT)
Pt3=(IN,3,4,3,4,OUT)

Was ist ein Vorteil von Zustandsautomaten?

- Man kann besser mit dem Kunden die Anforderungen präzisieren
- Zustandsübergänge dienen als gutes Überdeckungskriterium

Wann nutzen Sie Äquivalenzklassen?

- Wo sich das Programm bei unterschiedlichen Eingaben gleichartig verhält

Was für ein Nachteil haben Glas- / Whiteboxtests?

- Fehler können gefunden werden, aber nicht fehlende Funktionen

McCabe-Überdeckung

- Für jeden Kontrollflussgraphen gibt es eine Menge von vollständigen Pfaden, aus denen sich alle anderen vollständigen Pfade zusammensetzen lassen
- Basic Path Coverage: Überdeckung dieser "Basispfade"
- |E| - |N| + 2 viele Basispfade

ÜB 11

Atomare Prädikate

- Einzelne Wahrheitsvariable (Boolean)
- Vergleiche, z.B. "<, !=, =="
- Aufrufe Boolescher Funktionen, z.B. "isAvailable()"

Zusammengesetzte Prädikate

- Durch logishce Junktoren zusammengesetzt, z.B. "(a && b), "!a"

Bedingungsüberdeckung

- **Einfache** Bedingungsüberdeckung: Alle atomaren Prädikate nehmen mindestens einmal beide Wahrheitswerte an
- **Minimale** Bedingungsüberdeckung: Alle atomaren und zusammengesetzten Prädikate nehmen mindestens einmal beide Wahrheitswerte an
- **Mehrfache** Bedingungsüberdeckung: Alle Kombinationen von Auswertungen der atomaren Prädikate getestet

Beispielrechnung

Aufgabe:
100% Überdeckung mit möglichst wenigen Testfällen

Anweisungsüberdeckung (statement coverage):

- Eingabe (4,2,0) führt zu Pfad [1,2,3,4,5]

Zweigüberdeckung (branch coverage):

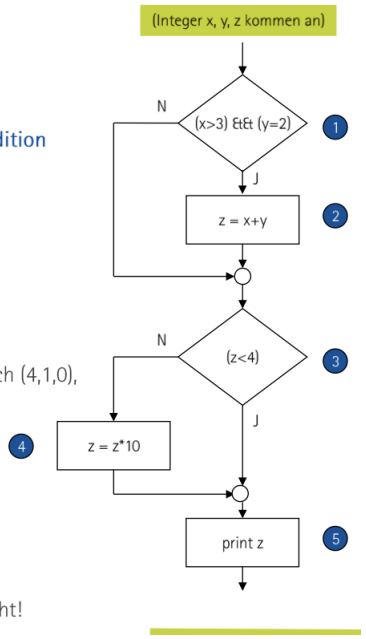
- Eingabe (4,1,0) stimuliert oben Nein-Zweig, unten Ja-Zweig
- Eingabe (4,2,0) stimuliert oben Ja-Zweig, unten Nein-Zweig

Mehrfache Bedingungsüberdeckung (condition coverage):

- (x>3) wahr, (y=2) wahr, durch (4,2,0)
- (x>3) wahr, (y=2) falsch, durch (4,1,0)
- (x>3) falsch, (y=2) wahr, durch (1,2,0)
- (x>3) falsch, (y=2) falsch, durch (1,1,0)
- gleichzeitig wird (z<4) mit überdeckt durch (4,1,0), (4,2,0)

Pfadüberdeckung (path coverage):

- erfordert Testfälle für folgende Pfade
- [1,2,3,5], [1,3,5], [1,2,3,4,5], [1,3,4,5]
- einen Testfall für [1,2,3,5] gibt es aber nicht!



Einfache Bedingungsüberdeckung

	1	2	3a	3b
T1	1	1		
T2	1	0		
T3	0		1	0
T4	0		0	1

Haben hier zwar einfache Bedingungsüberdeckung – aber keine Anweisungsüberdeckung!

```
public static int getTriangleKind(int sideA, int sideB, int sideC){
        if(sideA == sideB){
                if(sideB == sideC){
                        return 3;
                } else {
                        return 2;
                }
        } else if(sideA == sideC || sideB == sideC){
                return 2;
        } else {
                return 1;
        }
}
```


Minimale Bedingungsüberdeckung

	1	2	3a	3b	(3a || 3b)
T1	1	1			
T2	1	0			
T3	0		1	0	1
T4	0		0	1	1
T5	0		0	0	0

Checken hier, dass sowohl die einzelnen Teil-Prädikate als auch das zusammengesetzte Prädikat je beide Werte annehmen.

```
public static int getTriangleKind(int sideA, int sideB, int sideC){
        if(sideA == sideB){
                if(sideB == sideC){
                        return 3;
                } else {
                        return 2;
                }
        } else if(sideA == sideC || sideB == sideC){
                return 2;
        } else {
                return 1;
        }
}
```

Mutationstests

- Erzeuge **Mutanten**: Programmversion mit (zufällig oder systematisch) eingefügten Defekten
- Mutationsoperationen, z.B.:
  - Logische, arithmetische und Vergleichsoperationen ändern (==, >=, <, &&, !, …)
  - Variablen vertauschen oder durch Konstanten ersetzen

    - Konstanten ersetzen
    - …
- Identifiziert die Testmenge alle Mutanten, dann scheinen die Tests effektiv zu sein

Testen in der Entwicklung mit Doubles

- Wenn eine Funktion noch nicht vollständig implementiert ist
- Wenn ein Objekt nicht vollständig mit Daten gefüllt verfügbar
- Wenn die Umgebung eine komplexe Umgebung dafür nicht kostentechnisch (Zeit / Aufwand) sinnvoll aufsetzbar
- Wenn eine Datenbank noch nicht erreichbar oder nur "Production"-Datenbank verfügbar

(Test-)Double

- Ein Platzhalter, der für ein reales Objekt während des Testens eingesetzt wird.
- Solche Platzhalter simulieren das Verhalten der realen Objekte und unterbinden so das Auftreten von möglichen Seiteneffekten
- Bieten mehr Kontrolle über das aktuelle Geschehen bzw. die aktuelle Umgebung
- Müssen mit dem Original Objekt übereinstimmen und nach der Testphase einfach wieder neutralisiert werden können

Einsatz von Doubles

- Um Abhängigkeit von eigener oder externer Software bzw. Systemen zu reduzieren
- Unabhängigkeiten Sinnvoll für
    - Echte Objekte könnten durch das Testen beschädigt werden
    - Es werden nur Interfaces bereitgestellt, aber noch keine Implementierungen
    - Objektzugriff ist zu langsam (z.B. Datenbank) oder Objektrelationen zu komplex
    - Schwer auslösbares Verhalten (z.B. Netzwerkfehler)

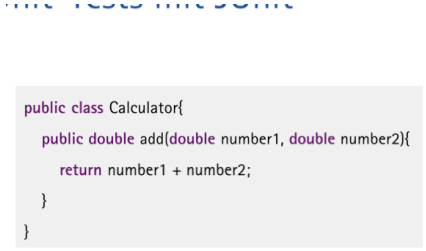
Arten von Doubles

- State verification (zum Prüfen von Systemzuständen)
    - Dummy : Objekt, das im Code weitergegeben, aber nicht verwendet wird
    - Fake: Objekt, das teilweise implementiert (bzw. beschränkt einsatzfähig) ist
    - Stub: Objekt, das echtes Verhalten durch vordefinierte Antworten ersetzt
    - Spy: Objekt, das Aufrufe und Werte protokolliert (und ggf. ausgibt)
- Behavioral verfication (zum Prüfen von Systemverhalten)
    - Mock: Objekt, das die Abhängigkeit unter gewissen Erwartungen ersetzt

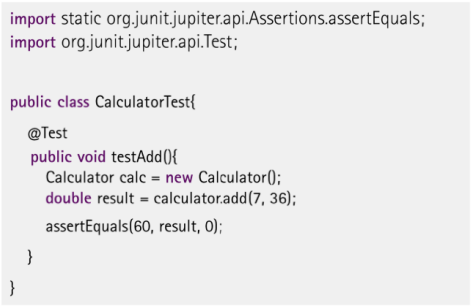
Unit-Tests

- Unit Test: Kleinste testbare Einheit (Methode, Klasse)
- Ein guter Unit-Test ist
    - Einfach zu implementieren
    - Einfach auszuführen
    - Konsistent im Ergebnis

- Vollständig isoliert von anderen Tests



```java
public class Calculator{
    public double add(double number1, double number2){
        return number1 + number2;
    }
}
```



```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest{
    @Test
    public void testAdd(){
        Calculator calc = new Calculator();
        double result = calculator.add(7, 36);
        assertEquals(60, result, 0);
    }
}
```

```java
@BeforAll
public static void setUpCalculator(){
    ...
}
```

```java
@BeforeEach
public void setUp(){
    ...
}
```

Asserterions

- Prüfung eines gewissen Wertes, es gibt viele unterschiedliche Arten und Überladungen:
    - assertEquals, assertTrue, assertFalse, assertNull, assertSame, assertThrows

Mockito Framework

- Erlaubt Verifizierung von Systemverhalten und Systemzuständen
- Beinhaltet viele unterschiedliche Arten von Doubles

ÜB 12

Unterschiedliche Beweisarten

- Automatische Modellprüfung (Model Checking)
    - Basierend auf Zustandsmodell der Software
    - Temporallogik spezifiziert gültige Abläufe
- Abstrakte Interpretation (Symbolische Ausführung)
    - Eingabeparameter als Variablen statt konkreter Werte
    - Bedingungen für Eingabevariablen werden abgeleitet
- Deduktive Beweistechnik z.B. Hoare-Kalkül
    - Bedingungen vor und nach Anweisungen mit Regeln beweisen

Anwendung formaler Methoden

- In kritischen Anwendungen
    - Medizin, Luft- und Raumfahr, Verkehrssysteme, Verschlüsselungsverfahren

Hoare-Kalkül Beweisregeln

- Komposition: Bedingungen vor und nach Anweisungen zusammenführen
- Zuweisung: Nachbedingung gegeben, was muss vor der Zuweisung mindestens gelten
- Fallunterscheidung (if): Nachbedingung gegeben, was ist die schwächste Vorbedingung, die man annehmen kann
- Iteration (while): Vorbedingung angegeben, was gilt nach der Schleife
- Verstärken der Vorbedingung
- Abschwächen der Nachbedingung

Komposition

- Beweisregel für hintereinander folgende Anweisungen
- P, Q und R sind Prädikate (Bedingungen)
- S_1 und S_2 sind aufeinanderfolgende Anweisungen,
  z.B. x := x + 5

(P)
S1;
(Q)
S2;
(R)

Prämisse

$$\frac{\{P\}\ S_1\ \{Q\}, \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Schlussfolgerung

Regel sagt aus: *Wenn* vor S_1 P gilt und nach S_1 Q gilt, sowie vor S_2 Q gilt und nach S_2 R gilt, *dann* gilt P vor der Hintereinanderausführung von S_1 und S_2 und R danach.

Zuweisung

- Zuweisung der Form x := E
- x ist eine Programmvariable, E ein Ausdruck (z.B. y / 2)
- P[E/x] bedeutet: in Bedingung P jedes x durch E ersetzen
- Anwendung der Regel "rückwärts": Nachbedingung P gegeben, was muss dann vorher gelten?

Prämisse

Schlussfolgerung

$$\frac{\{true\}}{\{P[E/x]\}\ x := E\ \{P\}}$$

Fallunterscheidung

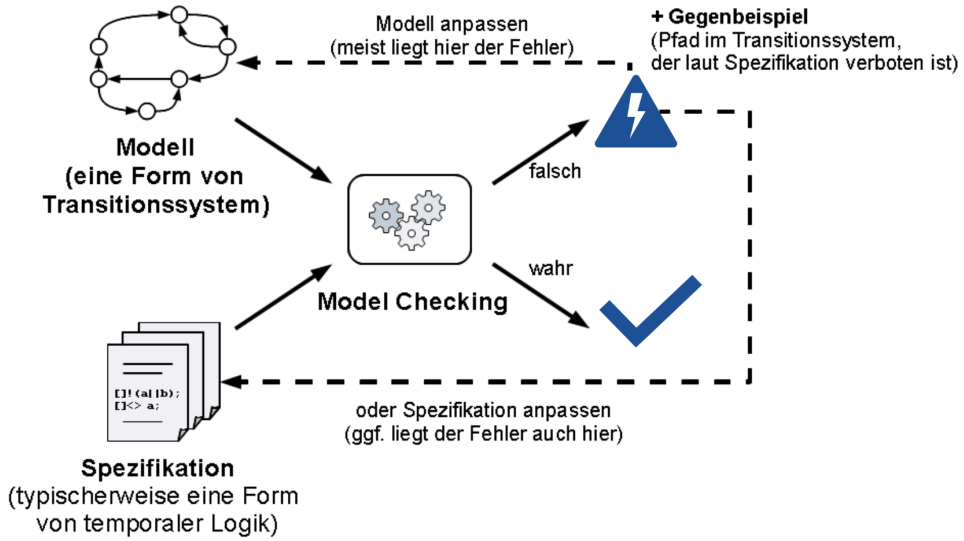
{P}
if B
  then
    {P ∧ B}
    S1
    {Q}
  else
    {P ∧ ¬ B}
    S2
    {Q}
{Q}

$$\frac{\{P \wedge B\}\ S_1\ \{Q\}, \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

Model-Checking

- Voraussetzung: Zustandsraum muss endlich oder endlich darstellbar sein

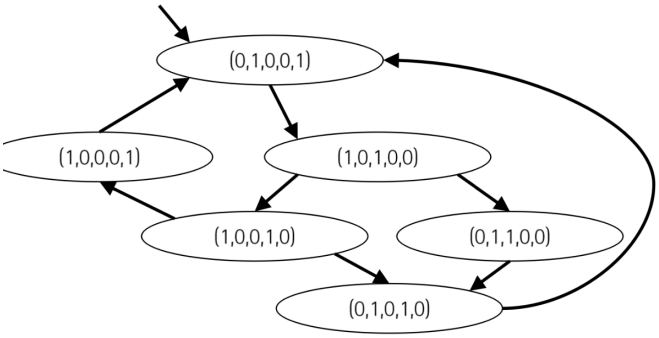
Das Prinzip

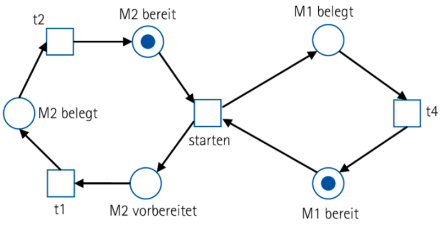


Model Checking

(M1 belegt, M1 bereit, M2 vorbereitet, M2 belegt, M2 bereit)

- 1. Schritt: Überdeckungsgraph aufstellen (Kripke-Struktur)



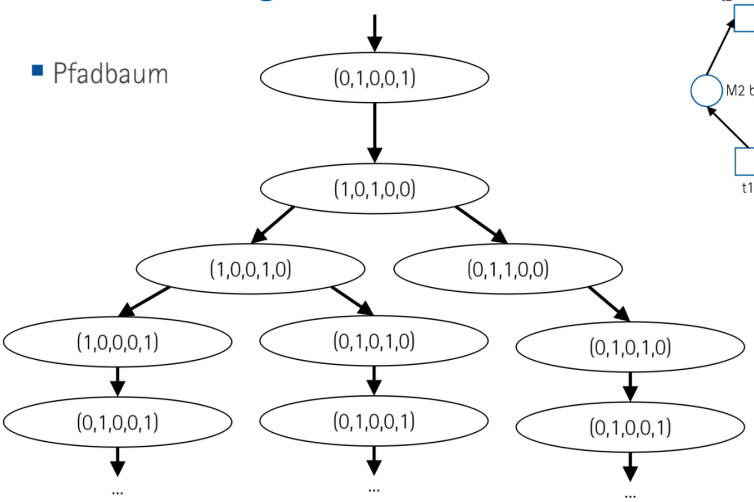


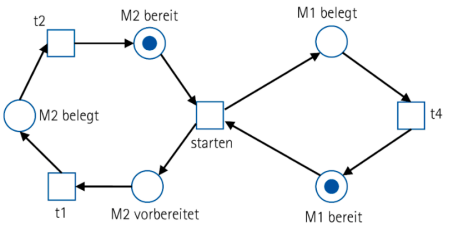
Computational Tree Logic (CTL)

- Einfache CTL-Formeln bestehen aus: Pfadquantor + temporaler Operator + Aussage über Aps
- Pfadquantor
  - A: Etwas gilt für alle Abläufe (all)
  - E: Etwas gilt für einen Ablauf (exists)
- temporaler Operator + Aussage über APs (hier p und q)
  - X p: p gilt im nächsten Zustand next
  - F p: p gilt in irgendeinem zukünftigen Zustand (irgendwann) future
  - G p: p gilt immer (jetzt und in allen zukünftigen Zuständen) general
  - p U q: es gilt jetzt q oder es gilt irgendwann q und bis dahin gilt p p bis q

Model Checking

- Pfadbaum





Testen vs. Reviews

Vorteile von Tests

- Reproduzierbar
- Mehrfach nutzbar
    - Rechenzeit ist billig
- Umgebung wird mitgeprüft
    - Bibliothek, VM
- Systemverhalten veranschaulicht
- Falls automatisiert schnell und billig

Nachteile von Tests

- Bedeutung "fehlerloser" Tests wird überschätzt
- Nicht alle Eigenschaften von Software sind testbar ("Lesbarkeit"?)
- Nicht alle Situationen reproduzierbar
- Test zeigt Fehlerursache nicht
- Falls nicht automatisiert: Aufwändig, insbesondere Wiederholung