Funktionale Testverfahren
Black Box-Tests

Funktionale Testverfahren

Überblick:

- Programmstruktur hat keinen Einfluss auf Testfälle
- Testfälle werden aus der Spezifikation abgeleitet → hat das Programm die vorgeschriebene Funktionalität?
- Testfall besteht aus den Eingabedaten und den erwarteten Ausgabedaten
- verschiedene Arten:
  * Zufallstest
  * funktionale Äquivalenzklassenbildung
  * Grenzwertanalyse
  * Ursache/Wirkungsgraphen
  * Test von Zustandsautomaten

Unsystematisches Testen

Zufallstest:

- füttern des Programms mit zufälligen Werten
- einfachstes funktionales Testverfahren
- erreicht Gleichbehandlung aller Eingabedaten ohne Beachtung menschlicher Präferenzen
- Behandlung von Ausnahmesituationen wird nur mit geringer Wahrscheinlichkeit getestet → Kombination mit Test von extremen/speziellen Werten

Unsystematisches Testen (2)

Fehlererwartungsmethode: Testperson

- legt Liste alle möglichen Fehler an, die beim Programmieren auftreten können (basiert auf Erfahrung)
- konzentriert sich auf Punkte, die Programmierer nicht berücksichtigt oder falsch interpretiert haben können

Beispiel: Liste ganzzahliger Werte sortieren

- leere Eingabeliste
- sortierte Eingabeliste
- Liste mit gleichen Werten

sehr von der persönlichen Erfahrung des Testers abhängig

Funktionale Äquivalenzklassenbildung

- Wertebereiche von Ein- und Ausgaben werden in Äquivalenzklassen eingeteilt → alle Eingabedaten, bei denen sich das Testobjekt gleich verhalten sollte
- Bildung der Äquivalenzklassen ist nur an der Spezifikation orientiert
- Äquivalenzklassen für gültige und ungültige Werte
- teste nur einen Repräsentanten jeder Klasse

Regeln zur Äquivalenzklassenbildung

- pro Eingabebereich eine gültige/ungültige Klasse

  Beispiel: i muss kleiner oder gleich 10 sein
  gültige Klasse: $i \leq 10$
  ungültige Klasse: $i > 10$

  Beispiel: $10 \leq i \leq 20$
  gültige Klasse: $10 \leq i \leq 20$
  ungültige Klassen: $i < 10$ und $i > 20$
- pro Eingabebedingung eine gültige/ungültige Klasse
  Beispiel: erstes Zeichen muss Buchstabe sein
  gültige Klasse: erstes Zeichen ist Buchstabe
  ungültige Klasse: erstes Zeichen ist kein Buchstabe

Regeln zur Äquivalenzklassenbildung (2)

- bei Datenstrukturen die Klassen anhand der Elementanzahl bilden

  Beispiel: Liste enthält 1 bis 256 Elemente
  gültige Klasse: $1, \ldots, 256$ Elemente
  ungültige Klassen: 0 Elemente/mehr als 256 Elemente
- analog für Ausgaben Äquivalenzklassen bilden
- zum Testen ein zufälliges Element der Klasse wählen
- Zweifel an Gleichbehandlung von Werten innerhalb einer Äquivalenzklasse → Äquivalenzklasse weiter unterteilen

Äquivalenzklassen bei `wordcount`?

Herleiten der Äquivalenzklassen

- für jede zu testende Eingabevariable deren Definitionsbereich ermitteln → Äquivalenzklasse aller zulässigen Eingabewerte
  * Komponententest: Funktions-/Methodenparameter
  * Systemtest: Maskenfelder
- Äquivalenzklassen verfeinern: Elemente, die laut Spezifikation unterschiedlich verarbeitet werden, neuer Unteräquivalenzklasse zuordnen
- Äquivalenzklasse aufteilen, bis alle unterschiedlichen Anforderungen durch Äquivalenzklassen abgedeckt werden

Äquivalenzklassen am Beispiel

Reisende Person:

- Kind (K)
- Jugendlicher (J)
- Erwachsener (E)
- Student (Stu)
- Sozialhilfeempfänger (Soz)
- Rentner (R)

gültige Äquivalenzklassen für

- Platzreservierung: [K, J, E, Stu, Soz, R]
- Preisberechnung: [K], [J], [E], [Stu], [Soz], [R]

Kombination der Repräsentanten

- pro Testobjekt existieren für jeden Parameter mind. zwei Äquivalenzklassen
- Problem: Welche Repräsentanten sind zu einem Testfall zu kombinieren?
  * Repräsentanten aller gültigen ÄKlassen kombinieren
  * Repräsentant einer ungültigen ÄKlasse wird nur mit Repräsentanten anderer gültiger ÄKlassen kombiniert $\rightarrow$ keine gegenseitige Fehlermaskierung, die Fehlerwirkung kann eindeutig dem ungültigen Wert zugeordnet werden

$\Rightarrow$ Anzahl gültiger Testfälle ergibt sich aus Produkt der Zahl gültiger Äquivalenzklassen je Parameter (ist evtl. sehr groß)

Kombination der Repräsentanten (2)

Regeln zur Testfalleinschränkung:

- Testfälle aus allen Repräsentanten kombinieren und anhand typischer Benutzungsprofile sortieren/priorisieren
- Testfälle bevorzugen, die Grenzwerte oder Grenzwertkombinationen enthalten
- paarweise Kombination statt vollständiger Kombination
- Minimalkriterium: jeder Repräsentant einer Äquivalenzklasse kommt in mindestens einem Testfall vor

Testfallerstellung am Beispiel

Prämienermittlung für Mitarbeiter anhand von

- Firmenzugehörigkeit (gemessen in Jahren)
  * gültig: [0..1], [2..5], [6..15], [16..50]
  * ungültig: $u < 0$, $u > 50$
- besuchte Schulungen im Jahr (gemessen in Tagen)
  * gültig: [0..2], [3..5], [6..10]
  * ungültig: $v < 0$, $v > 10$
- Anzahl beteiligter Projekte
  * gültig: [1], [2..4], [5..8]
  * ungültig: $w < 1$, $w > 8$

Testfallerstellung am Beispiel (2)

- Anzahl geleiteter Projekte
  * gültig: [0..1], [2..5]
  * ungültig: $x < 0$, $x > 5$
- Kundenbesuche im Jahr (gemessen in Wochen)
  * gültig: [0..5], [6..15], [16..25], [26..40]
  * ungültig: $y < 0$, $y > 40$

Anzahl Testfälle:

- gültig: $4 \cdot 3 \cdot 3 \cdot 2 \cdot 4 = 288$
- ungültig: $2 + 2 + 2 + 2 + 2 = 10$

Testfallerstellung am Beispiel (3)

Testfallreduzierung:

- jeder Repräsentant kommt in mindestens einem Testfall vor ($\rightarrow$ minimale Anforderung)

FZ	Schul.	bet.Proj.	gel.Proj.	Kundenb.
1	2	1	1	5
5	5	4	5	15
15	10	8	1	25
50	2	1	5	40

- paarweise Kombination (als Übung)

Äquivalenzklassenbildung: Anmerkungen

Testende bei Erreichen einer vorgegebenen Überdeckung

$$\text{Überdeckung} = \frac{\text{Anzahl getestete ÄK}}{\text{Gesamtanzahl ÄK}} \cdot 100\%$$

Vorsicht: gutes Ergebnis spiegelt nicht tatsächliche Testintensität wider, wenn Gesamtanzahl ÄK falsch ist!

Bewertung:

- die Systematik hilft, dass spezifizierte Bedingungen und Einschränkungen nicht übersehen und keine unnützen Testfälle durchgeführt werden
- sehr wirkungsvolles Verfahren

Grenzwertanalyse

- basiert auf Äquivalenzklassenbildung
- nutzt Werte, die am Rand einer Äquivalenzklasse liegen denn: Grenzbereiche werden oft fehlerhaft verarbeitet
- sinnvolle Erweiterung und Verbesserung der Äquivalenzklassenbildung

Beispiel: gegeben sind drei Äquivalenzklassen

- $1 \leq \text{month} \leq 12$ (gültig)
- $\text{month} < 1$ (ungültig)
- $\text{month} > 12$ (ungültig)

0, 1, 12 und 13 sind geeignete Repräsentanten

Ursache/Wirkungsgraphen

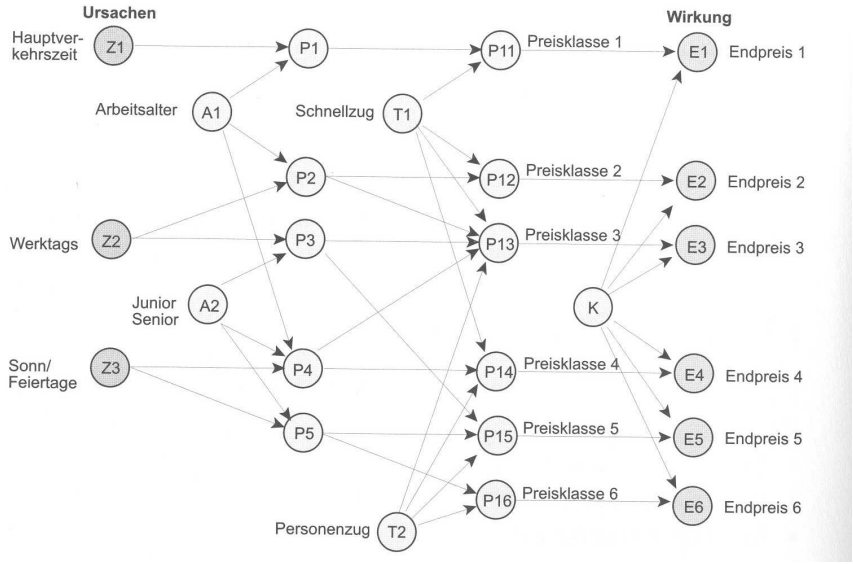
- schon Anfang der 1970er Jahre bei IBM eingesetzt
- beschreiben logische Beziehungen zwischen Eingaben und Ergebnisse mit booleschen Operatoren AND, OR, NOT
- annotierte Pfeile verbinden die Eingabeknoten ggf. über Zwischenergebnisknoten mit den Endergebnisknoten
- übersichtliches und einprägsames Mittel zur Testfallerstellung

Beispiel*: Fahrpreisberechnung abhängig von Entfernung, Fahrklasse, Zugart, Fahrgastalter, Wochentag und Uhrzeit

---

* aus Sneed, Winter: Testen objektorientierter Software. Hanser Verlag

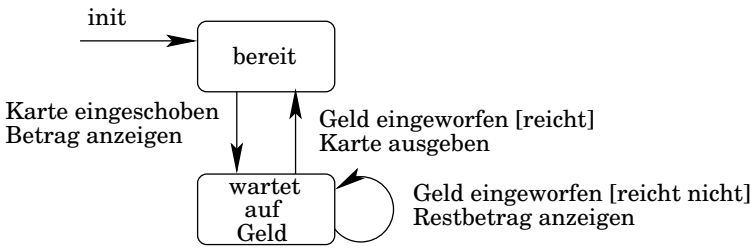
Ursache/Wirkungsgraphen (2)



Test von Zustandsautomaten

- Voraussetzung: Verhalten ist durch Zustandsautomaten spezifiziert
- Ziel: Abdeckung aller Zustandsübergänge

Beispiel:



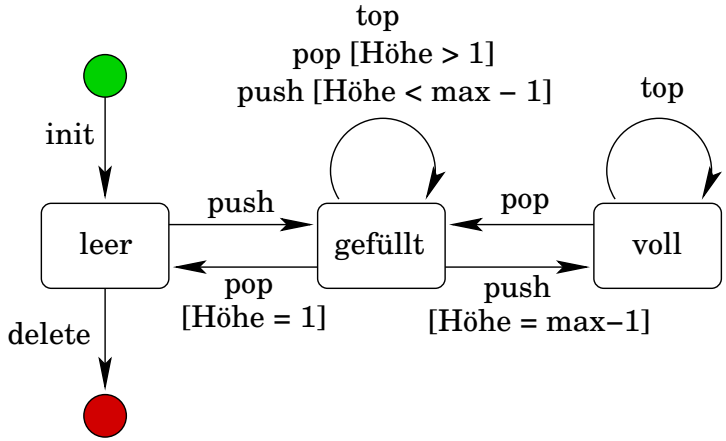
folgender Testfall deckt alle Zustände und Übergänge ab: Karte eingeschoben, Geld eingeworfen [reicht nicht], Geld eingeworfen [reicht]

Test von Zustandsautomaten (2)

- anzuwenden bei Systemen, bei denen nicht nur die Eingabewerte, sondern auch der bisherige Ablauf Einfluss auf das Verhalten/die Berechnung der Ausgabe hat
- Veranschaulichung der Historie durch Zustandsmodelle
- Testobjekt kann beginnend vom Startzustand verschiedene Zustände annehmen
- sehr gut geeignet zum Test objektorientierter Systeme:
  * Objekte können unterschiedliche Zustände annehmen
  * Methoden zur Manipulation der Objekte müssen auf unterschiedliche Zustände reagieren
- Testobjekt beim zustandsbezogenem Testen von OOP:
  * komplettes System mit verschied. Systemzuständen
  * oder eine Klasse mit unterschiedlichen Zuständen

Test von Zustandsautomaten (3)

Beispiel Stapel:



Test von Zustandsautomaten (4)

Testfälle:

- minimale Anforderung: jeder mögliche Zustand muss mindestens einmal erreicht werden:

  init [leer], push [gefüllt], push, push, push [voll]
  → es werden nicht alle Funktionen aufgerufen

- jede Funktion mindestens einmal aufrufen

  init [leer], push [gefüllt], top, pop [leer], delete
  → es werden nicht alle Zustände erreicht

⇒ pro Zustand jede spezifizierte Funktion mindestens einmal ausführen (evtl. auch spezifikationsverletzende Zustandsübergänge prüfen)

Test von Zustandsautomaten (5)

Übergangsbaum erstellen:

- Zustandsdiagramm hat potenziell unendlich viele Zustandsfolgen
- Übergangsbaum enthält eine repräsentative Menge von Zuständen ohne Zyklen
- Anfangszustand wird Wurzel des Baums
- für jeden möglichen Übergang vom Anfangszustand zu einem Folgezustand erhält der Baum von der Wurzel aus eine Verzweigung zu einem Knoten, der den Nachfolgezustand repräsentiert
- wiederhole für jedes Blatt den letzten Schritt

Test von Zustandsautomaten (6)

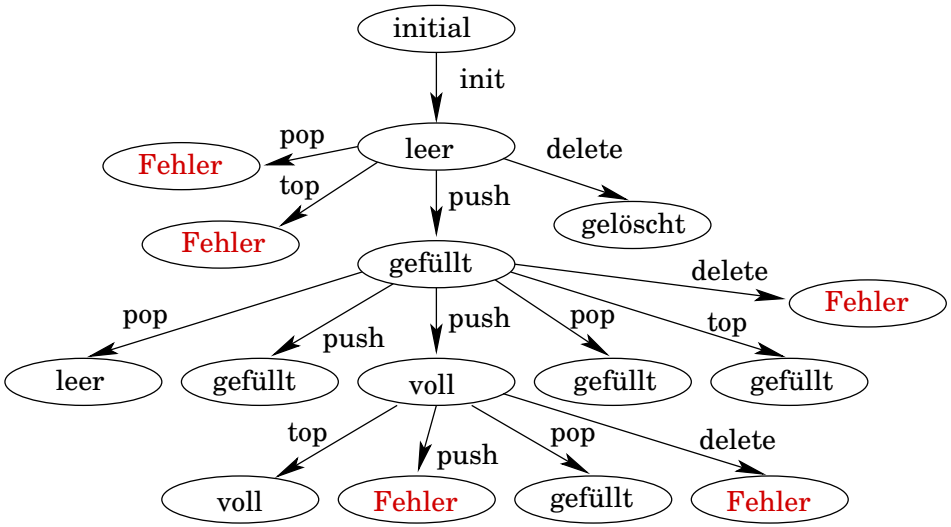
- **Robustheitstest:** fehlerhafte Verwendung der Funktionen testen
- zustandsbezogenes Testen eignet sich bei Systemtest zum Test der grafischen Bedienoberfläche
- Überdeckungsmaße:

$$\frac{\text{Anzahl getesteter Zustände}}{\text{Gesamtanzahl Zustände}} \cdot 100\%$$

  oder

$$\frac{\text{Anzahl getesteter Zustandsübergänge}}{\text{Gesamtanzahl Zustandsübergänge}} \cdot 100\%$$

Test von Zustandsautomaten (7)



Bewertung Black-Box-Tests

- fehlerhafte Spezifikation wird nicht erkannt
  → kann durch Review vermieden werden
- nicht-geforderte Funktionalität wird nicht erkannt
  → bspw. Sicherheitslücken
- Prüfung der Funktionalität hat höchste Priorität
  ⇒ **Black-Box-Tests sind immer durchzuführen**

Testprozess: Bewertung

nur ein Testverfahren alleine reicht nicht aus:

- Strukturtests entdecken keine fehlende Funktionalität
- Funktionstests berücksichtigen die vorliegende Implementierung nur unzureichend (Zweigabdeckung $< 70\%$)

⇒ kombiniertes Vorgehen

1. Funktionstest mit Grenzwertanalyse und Zufallstest
2. Strukturtest der Abschnitte, die noch nicht unter 1. abgedeckt wurden
3. Regressionstest nach Fehlerkorrektur

disversifizierende Testverfahren

Mutationstest

- Ziel: bestimme die Güte der Testdatensätze
- Annahme: Programme sind fast richtig
- kleine Änderungen am Programm (Mutationen) führen bei hinreichend umfassenden Testdatensätzen zu beobachtbaren Verhaltensänderungen
- Mutationen automatisch erzeugen und testen
- Anteil aufgedeckter Mutanten als Gütemaß
- perfekter Testdatensatz erkennt alle Mutanten

Mutationstest (2)

Mutationsoperatoren:

- beschreiben die Erzeugung semantisch veränderter, aber syntaktisch korrekter Programmversionen
- jeder Mutant enthält nur eine Änderung
- für jede Fehlerklasse eigene Operatoren

Fehlerklassen und deren Mutationsoperatoren:

- **Berechnungsfehler:** ändern der arithmetischen Operatoren, löschen arithmetischer Teilausdrücke, ändern von Konstanten
- **Schnittstellenfehler:** vertauschen/ändern von Parametern, aufrufen anderer Prozeduren des Moduls

Mutationstest (3)

Fehlerklassen und deren Mutationsoperatoren:

- **Kontrollflussfehler:** ersetzen logischer Teilausdrücke, ändern logischer und relationaler Operatoren, aufrufen anderer Prozeduren, löschen von Anweisungen
- **Initialisierungsfehler:** ändern von Konstanten, löschen von Zuweisungen
- **Datenflussfehler:** ändern der Indexberechnung, durchtauschen von Variablen in einem Sichtbarkeitsbereich

Bewertung: schon bei kleiner Anzahl Mutanten muss ein hoher Prozentsatz (größer 90%) entdeckt werden, sonst ist die Testsuite zu klein