Softwarequalitätssicherung

Sommersemester 2003

Dr. Thomas Santen
Softwaretechnologie
TU Dresden

Testenftwurf auf Klassenebene

Methodentest: Kontrollfluss / Algorithmus innerhalb einer Methode

Klassentest: Interaktionen zwischen Methoden einer Klasse

Fehlermöglichkeiten:

- falsche Nutzung der (globalen) Instanzvariablen
  (öffentliche / private Attribute)
- Fehler in der Abfolge von Methodenaufrufen

Welche Unterschiede zwischen Klassen gibt es bezüglich der zulässigen Abfolge von Methodenaufrufen?

Modalität einer Klasse

Kategorisierung nach den Einschränkungen bezüglich Werten der Instanzvariablen und Zustand:

nicht-modal: keine Einschränkungen auf Abfolge von Methodenaufrufen

unimodal: zulässige Methodenaufrufe abhängig von Zustand aber nicht Datenwerten

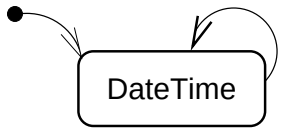
quasi-modal: zulässige Methodenaufrufe abhängig von Datenwerten

modal: zulässige Methodenaufrufe abhängig von Datenwerten und Zustand

Nicht-modale Klasse

Implementierungen einfacher Datentypen sind oft nicht-modal, z.B.:

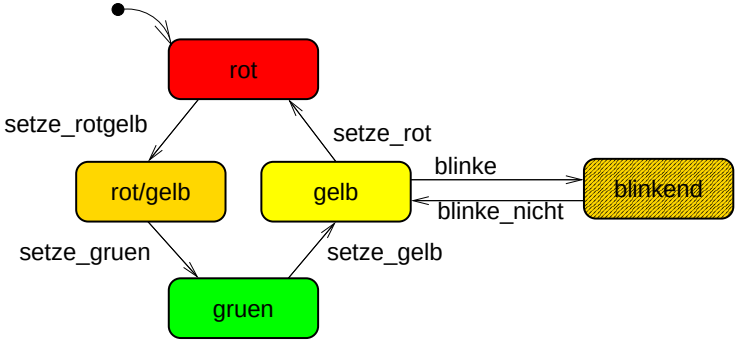
```
class DateTime {
   public DateTime()
   public void setDate(Date d)
   public void setTime(Time t)

   public Date getDate()
   public Time getTime()
};
```



Die `set`- und `get`-Methoden können immer aufgerufen werden.

UNIMODALE KLASSE

Klassen mit Kontrollfunktionen sind oft unimodal, z.B. Verkehrsampel:

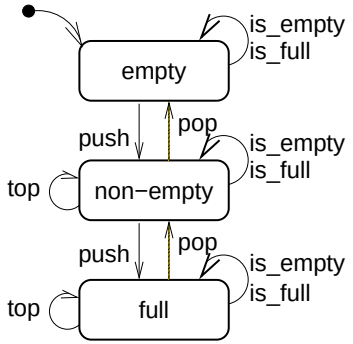


Methodenaufrufe sind nur in best. Zuständen zulässig, z.B. `blinke` nur in `gelb`.

QUASI-MODALE KLASSSE

Viele *Container*- und *Collection*-Klassen, d.h. abstrakte Datentypen, sind quasi-modal, z.B. Stack:

```
class Stack {
   private Item[] content

   public Stack()
   public void push(Item i)
   public Item pop()

   public boolean is_empty()
   public boolean is_full()
};
```



Zustände (`empty`, `...`) definieren sich nur über den Wert von `content`.

MODALE KLASSE

Repräsentationen von Anwendungsbereichen sind oft modal, z.B. Konto:

- Geld von Konto abheben nur, falls Guthaben positiv
  und Konto weder gesperrt noch geschlossen
- Sperren eines Kontos nur, falls es noch nicht gesperrt
  und nicht geschlossen ist (unabhängig vom Kontostand)

- Modalität einer Klasse beeinflusst mögliches Fehlverhalten
- Testfälle auf Klasseneben an Modalität orientieren

MUSTER: INVARIANT BOUNDARIES (1)

Absicht: Auswahl test-effizienter Wertkombinationen für Klassen, Schnittstellen und Komponenten, die aus komplexen und einfachen Datentypen zusammengesetzt sind.

Kontext: Wie findet man Wertebelegungen für die Variablen im Bereich der IUT?

Ansatz: Zerlegung einer *Invariante* und Analyse der sich ergebenden Grenzen zulässiger und unzulässiger Varialbenwerte der Invarianten.

Anwendbarkeit: Auf alle IUT, für die sich Invarianten aufstellen lassen, z.B. Klassen, Subsysteme, Systeme.

MUSTER: INVARIANT BOUNDARIES (2)

Fehlermodell: Fehler hängen oft mit den Grenzbereichen von Variablenwerten zusammen. Unübliche, aber zulässige Wertekombinationen von Instanzvariablen werden auch oft falsch behandelt.

Annahme: Unübliche, grenzwertige Kombinationen von Variablenwerten decken Fehler auf.

Strategie:

1. Klasseninvariante aufstellen
2. Grenzpunkte (*On/Off*-Punkte) bestimmen
3. Werte für die in der Invariante nicht genannten Variablen bestimmen

1. KLASSENINVARIANTE BESTIMMEN

```
class CustomerProfile {
  Account account1 = new Account();
  Account account2 = new Account();
  Money creditLimit = new Money();
  short txCounter;
  ...
```

`Account` ist modale Klasse mit abstrakten Zuständen *open*, *overdrawn*, *frozen*, *inactive* und *closed*.

`Money` ist skalarer Datentyp mit Wertebereich $\pm 10^{12}$

Klasseninvariante:

$$\begin{aligned} &(\mathit{txCounter} \geqslant 0 \wedge \mathit{txCounter} \leqslant 5000) && \wedge \\ &(\mathit{creditLimit} > 99.99 \wedge \mathit{creditLimit} \leqslant 1000000.00) && \wedge \\ &\neg\,(\mathit{account}1.\mathit{isClosed}() \vee \mathit{account}2.\mathit{isClosed}()) \end{aligned}$$

2. GRENZPUNKTE BESTIMMEN – *On/OFF/In/Out*

Klassifikation der Punkte im Wertebereich einer Variablen:

***On*-Punkt:** liegt auf einer Grenze des Bereichs

***Off*-Punkt:** liegt auf keiner Grenze des Bereichs

***In*-Punkt:** liegt innerhalb der Bereichsgrenzen

***Out*-Punkt:** liegt außerhalb der Bereichsgrenzen

2. GRENZPUNKTE BESTIMMEN – 1×1-AUSWAHLKRITERIUM

Ein *On*-Punkt und ein *Off*-Punkt für jede Bereichsgrenze!

Auswahlregeln:

- ein *On*-Punkt und ein *Off*-Punkt für jede relationale Bedingung
- ein *On*-Punkt und zwei *Off*-Punkte für jede Gleichheit auf geordneten Typen
- ein *On*-Punkt und ein *Off*-Punkt für jeden ungeordneten Typ
- ein *On*-Punkt und mindestens ein *Off*-Punkt für jede abstrakte Zustandsinvariante
- ein *On*-Punkt und ein *Off*-Punkt für jede nicht-lineare Grenze
- gleiche Tests für aneinandergrenzende Unterbereiche *nicht* wiederholen!

2. Grenzpunkte bestimmen – Beispiel (1)

Bedingung	*On*-Punkt	*Off*-Punkt
$txCounter \geqslant 0$	0	−1
$txCounter \leqslant 5000$	5000	5001
$creditLimit > 99.99$	99.99	100.00
$creditLimit \leqslant 1000000.00$	1000000.00	1000000.01
$\neg\ account1.isClosed()$	*True*	*False*
$\neg\ account1.isClosed()$	*True*	*False*

2. Grenzpunkte bestimmen – Beispiel (2)

Abhängigkeiten zwischen Variablen, z.B.:

$$creditLimit \geqslant (txCounter \times 100) + 100$$

On-Punkte: Grenzpunkte für *txCounter* einsetzen

$$creditLimit_{lowon} = (0 \times 100) + 100 = 100.00$$
$$creditLimit_{highon} = (5000 \times 100) + 100 = 500100.00$$

Off-Punkt: Schwerpunkt der unabhängigen Variablen einsetzen und so variieren, dass Bedingung „gerade“ verletzt wird.

$$creditLimit_{midpoint} = (2500 \times 100) + 100 = 250100.00$$
$$creditLimit_{off} = creditLimit_{midpoint} + 0.01 = 250100.01$$

Muster: Nonmodal Class Test (1)

Absicht: Entwicklung einer Test-Suite für eine nicht-modale Klasse

Kontext: Nicht-modale Klassen lassen jede Folge von Methodenaufrufen zu, haben aber oft einen komplexen Datenzustand und eine komplexe Schnittstelle. Wie kann man Methodenaufrufsequenzen finden, die Fehler aufdecken?

Fehlermodell: Keine Einschränkung der Aufrufsequenzen, daher zustandsbasiertes Testen nicht sinnvoll. Häufige Fehler:

- zulässige Sequenz wird abgelehnt
- zulässige Sequenz ergibt falschen Wert
- abstrakte Zustandsabfragen sind inkonsistent
- zulässige Veränderung wird abgelehnt
- etc

Muster: Nonmodal Class Test (2)

Strategie:

1. Testfallentwicklung mit dem Muster *Invariant Boundaries*
2. Wähle eine Aufrufabfolge-Strategie: *Define-Use*, zufällig, verdächtig
3. Setze OUT auf einen Testfall aus (1)
4. Rufe alle Zugriffsmethoden auf und vergleiche Ausgaben mit dem Testfall
5. Wiederhole (3) und (4) bis alle Sequenzen nach (2) ausgeführt sind

---

Aufrufabfolge-Strategie:

***Define-Use*:** ein definierender Methodenaufruf wird von Aufrufen aller Zugriffsmethoden gefolgt

zufällig: pseudozufällige Erzeugung von Aufrufsequenzen

verdächtig: inhaltlich besondere Sequenzen,
z.B. Schaltjahr und 29.2. in `DateTime`

MUSTER: NONMODAL CLASS TEST (3)

Eingangskriterium: minimale Ausführbarkeit durch α/ω-Zyklus auf CUT nachgewiesen

Ausgangskriterium:

- Überdeckung einer nicht-modalen Klasse
  1. alle *Define-Use*-Paare sind ausgeführt
  2. alle Testfälle aus *Invariant Boundaries* mindestens einmal als Zustand angenommen
- mindestens Zweigüberdeckung für jede Methode der CUT

Konsequenzen: Größe der Testsuite exponentiell in Länge der *Define-Use*-Sequenzen! Zusicherungsprüfung zur Laufzeit (Nachbedingung, Klasseninvariante) kann Aufwand reduzieren.

MUSTER: QUASI-MODAL CLASS TEST (1)

Absicht: Entwicklung einer Test-Suite für eine Klasse, deren Einschränkungen auf Aufrufsequenzen sich mit dem Datenzustand ändern.

Kontext: *Container*- und *Collection*-Klassen sind oft quasi-modal. Hierarchien solcher Klassen sind häufig Teil von Standard-Bibliotheken. Oft interessiert nicht der genaue Datenzustand, sondern eine Abstraktion davon (z.B.: leer,nicht-leer,voll).

Fehlermodell: Quasi-modale Fehler treten auf, wenn Klasseninvarianten, die sich auf alle Elemente eines Containers beziehen, verletzt werden.

Beispiel: zweimaliges `add(123)` auf Menge ganzer Zahlen

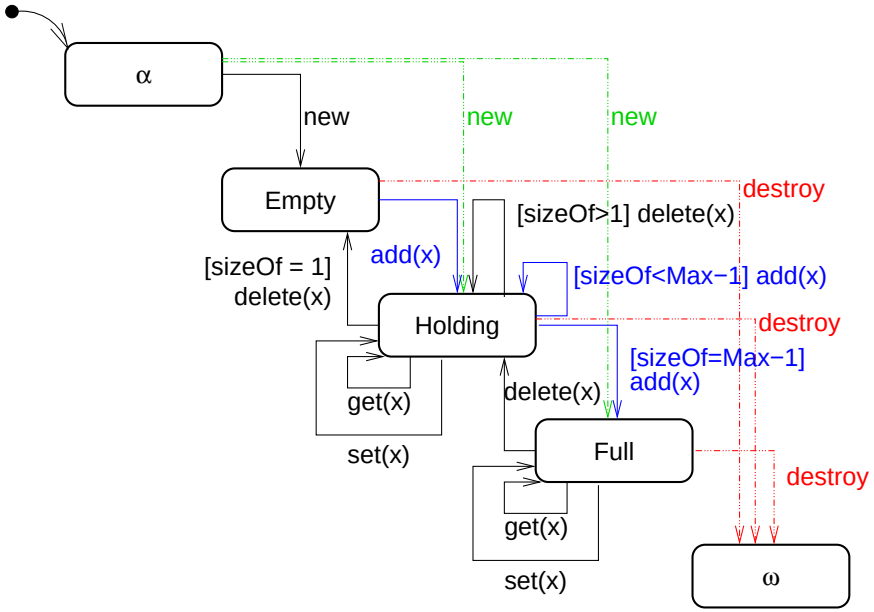
- zweiter Aufruf löst Ausnahme *duplicate* aus, trägt aber trotzdem 123 nochmals in Menge ein
- erst zweimaliges `remove(123)` führt deshalb dazu, dass `is_member(123)` falsch wird

MUSTER: QUASI-MODAL CLASS TEST (2)

Strategie – Testmodell:

- generisches Zustandsmodell für *Collections*
- *Invariant Boundaries* für Parameter, die Verhalten bestimmen (z.B. `Stack`: maximale Größe, tatsächliche Größe)
- Operations-Paar-Modell, das für die betrachtete Klasse spezifisch ist (z.B. zweimaliges `add(123)`) ok für `Stack` aber nicht für `Set`.

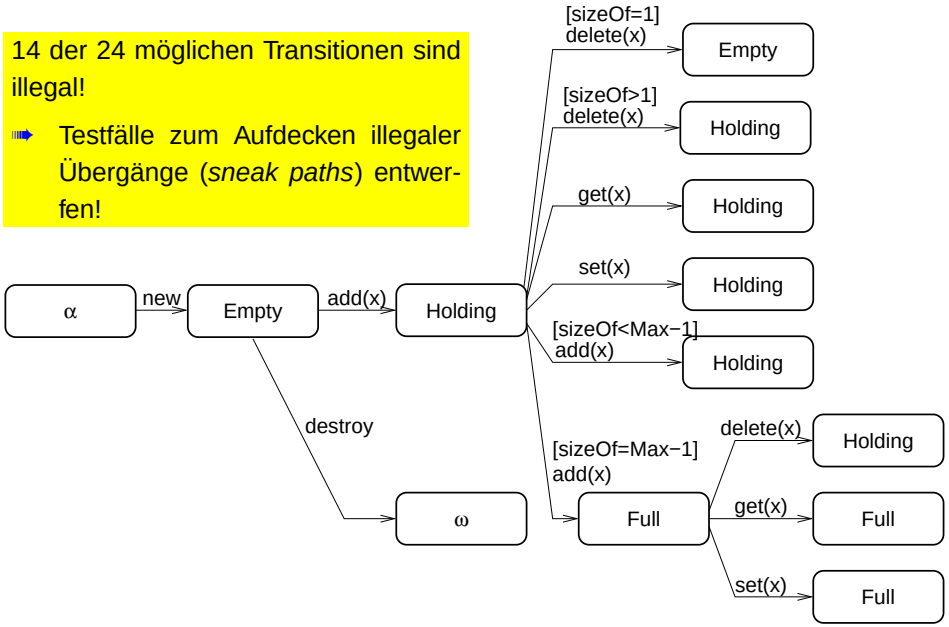
GENERISCHES ZUSTANDSMODELL FÜR QUASI-MODALE KLASSEN



Transitionsbaum für generisches Zustandsmodell

14 der 24 möglichen Transitionen sind illegal!

➠ Testfälle zum Aufdecken illegaler Übergänge (*sneak paths*) entwerfen!



Kollektions-Typen für Entwurf von Aufrufsequenzen

sequentielle Kollektion: Elemente können nur in fester Reihenfolge angesprochen werden (Liste, Puffer)

geordnete Kollektion: Elemente werden nach best. Ordnung hinzugefügt und gelöscht (Stack, Queue, Baum)

Kollektion mit Schlüssel: Zugriff wird über Primärschlüssel realisiert (Suchbaum, Datenbankschnittstelle)

Kollektion mit Schlüssel – Testfälle für Operations-Paare

Operations-Paar	Schlüssel x	Schlüssel y	erwartetes Ergebnis
`add(x), add(y)`	max	max	zweiter zurückgewiesen
`get(x), get(y)`	max	max	akzeptiert
`set(x), set(y)`	max	max	akzeptiert
`delete(x), delete(y)`	max	max	zweiter zurückgewiesen
`add(x), add(y)`	min	max	akzeptiert
`get(x), get(y)`	min	max	akzeptiert
`set(x), set(y)`	min	max	akzeptiert
`delete(x), delete(y)`	min	max	akzeptiert
⋮	max	min	akzeptiert
`add(x), add(y)`	min	min	zweiter zurückgewiesen
`get(x), get(y)`	min	min	akzeptiert
`set(x), set(y)`	min	min	akzeptiert
`delete(x), delete(y)`	min	min	zweiter zurückgewiesen

Muster: Quasi-modal Class Test (3)

Eingangskriterium: Minimale Ausführbarkeit durch α/ω-Zyklus nachgewiesen

Ausgangskriterium:

- mindestens Zweigüberdeckung auf jeder Methode der CUT
- erzeugte Testfälle überdecken alle Äste im Transitionsbaum und produzieren vollständige Menge von Operationspaaren für illegale Übergänge
- aufwändiger: Klassen-Flussgraphen erstellen und Überdeckung aller α/ω-Pfade sicherstellen

Konsequenzen: Operations-Paare finden nicht alle Fehler (aber viele).

Anforderungen für Einsetzbarkeit des Musters:

- testbares Verhaltensmodell
- interner Zustand der CUT ist beobachtbar

MUSTER: MODAL CLASS TEST (1)

Absicht: Entwicklung einer Test-Suite für eine Klasse, die aufruf- und datenbezogenen Einschränkungen auf Aufrufsequenzen hat.

Kontext: Klassen, die Problembereiche repräsentieren, sind oft modal (z.B. `Account`). Klassen, die zusammen das *State*-Entwurfsmuster implementieren, sind wahrscheinlich modal.

Fehlermodell: Fünf Arten, das Zustandsmodell einer Klasse fehlerhaft zu implementieren:

1. fehlende Transition – ein Aufruf wird in einem zulässigen Zustand abgelehnt
2. falsche Aktion – inkorrekte Änderung der Instanzvariablen
3. falscher Folgezustand – Transition in einen falschen Zustand
4. fehlerhafter Zustand – es wird kein zulässiger Zustand erreicht
5. illegaler Übergang – Aufruf wird akzeptiert, der eigentlich abgelehnt werden sollte

ZUSTANDSFEHLER UND MÖGLICHE URSACHEN

		mögliche Ursache		
Fehler		lexi.	dyn.	allgemein
Transformation	fehlt	✗	✗	...
	falsch	✗	✗	...
	extra	✗		kreatives Programmieren
Übergang	fehlt	✗	✗	SUT kann Sequenz nicht erzeugen
	falsch	✗	✗	wie „fehlt“ oder „extra“
	extra	✗	✗	*sneak path* erlaubt unspez. Sequenz
Ausgabeaktion	fehlt	✗	✗	...
	falsch	✗	✗	...
	extra	✗		kreatives Programmieren
Zustand	fehlt	✗	✗	...
	falsch	✗	✗	falsche Anweisungen; falsche Nutzung d. Klient
	extra	✗		kreatives Programmieren

MUSTER: MODAL CLASS TEST (2)

Strategie: ähnlich wie bei *Quasi-Modal Class Test*, aber zusätzlich Beachtung der Transitionsbedingungen

Ein-/Ausgangskriterium: wie *Quasi-Modal Class Test*

---

es gibt noch eine ganze Reihe anderer Testentwurfsmuster für flache Klassen, Subsysteme, usw. ...

ZUSAMMENFASSUNG – TESTENTWURFSMUSTER

- Strukturierung des Testentwurfsprozesses
- Muster (u.a.) für spezielle Probleme / Fehlerklassen der Objektorientierung
- jedes Muster für (eingegrenzten) Test-Zweck anwendbar (vgl. Fehlermodell)
- Muster ergänzen sich

  Bottom-Up Teststrategie: Muster für kleinere Einheiten garantieren minimale Ausführbarkeit größerer Einheiten für deren Tests
- konzise, leicht zugängliche Dokumentation von Teststrategien