Objektorientierte Systeme

Objektorientierte Systeme

Unterschiede zum Test funktionaler Module:

- Operationen sind funktional unabhängig, stehen aber über den Zustand gemeinsamer Objektattribute miteinander in Verbindung
  * Operation hinterlässt Objektzustand, der das Verhalten der nachfolgenden Operation beeinflusst
  * Argumentbereich einer Operation umfasst Parameter und Zustand des Objekts
  * Operationen einer Klasse sind über gemeinsame Daten gekoppelt
- Kontrollstrukturen einzelner Operationen sind in der Regel einfacher und weniger verschachtelt

Objektorientierte Systeme (2)

- Vererben von Attributen und Operationen schafft neue Abhängigkeiten → Redundanz wird eliminiert auf Kosten der gegenseitigen Abhängigkeiten
- neue Testverfahren aufgrund von Polymorphismus und dynamischer Bindung → testen jeder möglichen Bindung

Objektorientierte Systeme (3)

Klassentest:

- prüfe zuerst Operationen, die nicht zustandsverändernd sind, dann zustandsändernde Operationen prüfen
- durch Äquivalenzklassenbildung und Grenzwertanalyse aus den Parametern Testfälle ableiten (ggf. muss Objekt vorher in den zulässigen Zustand versetzt werden)
- Test jeder Folge abhängiger Operationen (Zustands- oder Zustandsübergang-Überdeckungstest)

Objektorientierte Systeme (4)

moderne objektorientierte Systeme sind

- Client/Server-Systeme mit
- graphischen Oberflächen und
- Zugriff auf Objekt-/relationale Datenbanken

Testanforderungen aufgrund der Objektorientierung sind gering im Vergleich zu den Anforderungen, die sich aus der Systemverteilung und der Komplexität der Oberflächen ergeben!

Client/Server-Testanforderungen

Geschäftstransaktion: wird von mehreren Teilprogrammen auf verschiedenen Rechnern ausgeführt und verarbeitet Daten, die in verschiedenen Datenbanken gespeichert sind

- Datenübertragung muss funktionieren
- Objekte müssen richtig erzeugt werden
- Schnittstellen müssen übereinstimmen
- Daten müssen zusammenpassen
- Zugriffe auf Datenbanken müssen synchronisiert sein
- parallel laufende Transaktionen dürfen sich nicht stören
- Speicher muss wieder freigegeben werden
- Fehler müssen abgefangen werden

Client/Server-Testanforderungen (2)

- Transaktionen ordnungsgemäß abschließen
- in Fehlerfällen alte Zustände wiederherstellen

$\Rightarrow$ Handshaking, Two Phase Commit und Rollback testen!

Probleme, die bei verteilten Systemen auftreten können:

- Performance-Engpässe im Netz (Zugriff auf gemeinsame Server-Objekte, Bandbreite, Geschwindigkeit)
- Netzwerkausfälle/Time-Outs führen zum Abbruch laufender Transaktionen
- Fehler in der Synchronisation parallel arbeitender Komponenten
- Deadlocks beim Zugriff auf gemeinsame Ressourcen

GUI-Testanforderungen

Früher: 3270-Terminals

- festformatierte Masken
- ein Großteil der Felder war geschützt, der andere Teil ließ nur bestimmte Eingaben zu
- einziges Eingabemedium war die Tastatur

Heute: WIMP (windows, icons, menus, pointing device)

- oft kann ein Endbenutzer seine Oberfläche individuell anpassen (Farben, Schriftgrößen, Layout, Sprache)
- Benutzer kann wahlweise mit Maus oder Tastatur arbeiten bzw. positionieren

GUI-Testanforderungen (2)

- es können mehrere Masken aktiv sein, so dass Benutzer gleichzeitig mehrere Transaktionen bearbeiten kann
- Reihenfolge der Eingabesignale ist beliebig kombinierbar (kein fester Kontrollfluss)

$\Rightarrow$ viel höherer Aufwand als früher

Hilfsmittel: eine automatische Aufzeichnung von Eingabesignalen und deren Rückspulung sowie eine flächendeckende Bombardierung der Oberfläche mit Mausklicks

alle Bedienungsmöglichkeiten müssen getestet werden:

- die erlaubten $\rightarrow$ zeige, dass sie funktionieren
- die nicht erlaubten $\rightarrow$ zeige, dass sie abgefangen werden

Datenbank-Testanforderungen

- heute noch selten: oo-Anwendung nutzt oo-Datenbank
- oft: oo-Anwendungen sind mit relationalen Datenbanksystemen kombiniert
- Objekte werden erst zur Laufzeit aus den Relationen gebildet und nachher wieder in Relationen aufgelöst (fehleranfällig)
- relationale Datenbanken sind komplexe Strukturen mit verschleierten Abhängigkeiten (z.B. referenzielle Integrität: wird die Basisrelation gelöscht, werden abhängige Relationen automatisch mit gelöscht)
- Ergebnisse von DB-Abfragen sind davon abhängig, welche Datenkonstellation zur Zeit der Abfrage vorliegt $\rightarrow$ DB-Zustände archivieren

Datenbank-Testanforderungen (2)

zu testen sind:

- **stored procedures:** Funktionen in prozeduraler Sprache (3GL im Gegensatz zu SQL als 4GL-Sprache, SQL ist nicht Turing-vollständig)
- **trigger:** stored procedures, die bei bestimmten Aktionen ausgeführt werden
- **rules** sollen verhindern, dass die DB in einen logisch inkonsistenten Zustand gerät. Eingabedaten werden geprüft, bevor sie in den Tabellen gespeichert werden.
- **constraints:** Regeln, die unerwünschte Datenkombinationen verhindern (z.B. Fremdschlüssel-Constraints)

Wann Stored Procedures einsetzen?

Algorithmen Änderungen der Datensätze abhängig von Bedingungen $\rightarrow$ Kontrollanweisungen hilfreich

Beispiel Mitarbeitergehälter in Abhängigkeit der Position (Projekt-, Gruppenleiter, Sachbearbeiter) ändern $\rightarrow$ SQL bietet nur eingeschränkte Zugriffsmöglichkeiten

```
update Mitarbeiter
set gehalt = gehalt * 1.08
where exists (
   select * from Arbeiten
   where Nr = MNr and Aufgabe = 'Projektleiter'
);
```

Wann Stored Procedures einsetzen? (2)

```
update Mitarbeiter
set gehalt = gehalt * 1.06
where exists (
   select * from Arbeiten T1
   where Nr = MNr and Aufgabe = 'Gruppenleiter'
   and not exists (
      select * from Arbeiten T2
      where T1.MNr = T2.MNr
      and Aufgabe = 'Projektleiter'
   )
);
```


Wann Stored Procedures einsetzen? (3)

```
update Mitarbeiter
set gehalt = gehalt * 1.04
where exists (
   select * from Arbeiten T1
   where Nr = MNr and Aufgabe = 'Sachbearbeiter'
   and not exists (
      select * from Arbeiten T2
      where T1.MNr = T2.MNr
      and Aufgabe in ('Projektleiter', 'Gruppenleiter')
   )
);
...
```


Wann Stored Procedures einsetzen? (4)

einfacher und verständlicher wäre

```
define function bool isProjLeiter(nr) as ...
define function bool isGrupLeiter(nr) as ...
define function bool isSachbearb(nr) as ...
for each employee in table Mitarbeiter
loop
    if isProjLeiter(employee.MNr) then mult = 1.08
    else if isGrupLeiter(employee.MNr) then mult = 1.06
    else if isSachbearb(employee.MNr) then mult = 1.04
    else mult = 1.02;

    update Mitarbeiter set Gehalt = Gehalt * mult
    where MNr = employee.MNr;
end loop;
```


Wann Stored Procedures einsetzen? (5)

Client/Server-Architekturen: steht nur ein Netzwerk mit geringer Bandbreite zur Verfügung → mehrere SQL-Statements hintereinander auf Server ablaufen lassen und nur endgültiges Ergebnis zum Client übertragen

→ Effizienz

SQL ist nicht Turing-vollständig: es gibt Probleme, die mit SQL nicht lösbar sind (aufgrund fehlender Schleifen) → prozedurale Erweiterungen müssen genutzt werden

Wann Stored Procedures einsetzen? (6)

Funktionen: werden SQL-Statements von verschiedenen Anwendern benutzt → Anweisungen parametrisieren, als Funktion auf Server hinterlegen
→ Wiederverwendbarkeit, Portabilität

Vorsicht:

- oft müssen stored procedures umgeschrieben werden, wenn auf eine andere Datenbank/Version portiert wird
- oft ist dynamic SQL die bessere Alternative

Gibt es weitere Gründe für Stored Procedures?

Klassenhierarchien auf RDBS abbilden*

- benutze eine einzige Tabelle für eine Klassenhierarchie
- verwende eine Tabelle für jede konkrete Klasse
- oder verwende eine Tabelle pro Klasse

Beispiel:

- abstrakte Klasse `Person`: Attribute `Name` und `Telnr`
- `Mitarbeiter` Unterklasse von `Person`: `EinstDatum`
- `LeitenderMitarb` ist ein `Mitarbeiter`, der zusätzlich einen `Bonus` erhält
- `Kunde` ist Unterklasse von `Person`, besitzt zusätzliche Felder `KundeID` und `Präferenzen`

* www.ambysoft.com/mappingObjects.html

Klassenhierarchien auf RDBS abbilden (2)

1. Eine Tabelle für die gesamte Klassenhierarchie
→ jedes Attribut aller Klassen der Hierarchie wird in einer einzigen Tabelle gespeichert

```
Person (Name, Telnr, EinstDatum, KundeID, Präferenzen,
        Bonus)
```

Vorteile:

- Einfachheit
- gute Unterstützung von Polymorphismus (Person kann sowohl Mitarbeiter als auch Kunde sein)
- online-Anfragen werden sehr gut unterstützt (jede Information ist in einer einzigen Tabelle gespeichert)

Klassenhierarchien auf RDBS abbilden (3)

Nachteile:

- großer Speicherbedarf
- enge Kopplung (wenn in einer Klasse ein Attribut hinzugefügt wird, muss das Attribut in die Tabelle aufgenommen werden und kann somit alle anderen Klassen beeinflussen)

Klassenhierarchien auf RDBS abbilden (4)

2. Eine Tabelle pro konkreter Klasse
→ jede Tabelle enthält die Attribute der jeweiligen Klasse, die es repräsentiert, sowie alle von übergeordneten Klassen geerbten Attribute

```
Mitarbeiter (Name, Telnr, EinstDatum)
LeitenderMitarb (Name, Telnr, EinstDatum, Bonus)
Kunde (Name, Telnr, KundeID, Präferenzen)
```

Vorteile:

- Einfachheit
- online-Abfragen werden gut unterstützt

Klassenhierarchien auf RDBS abbilden (5)

Nachteile:

- ändern einer Klasse → nicht nur repräsentierende Tabelle, sondern auch alle Tabellen ändern, die Unterklassen beschreiben
- ändert ein Objekt seine Rolle (z.B. aus Kunde wird Mitarbeiter) müssen die Daten aus der einen Tabelle in die andere kopiert werden
- mehrere Rollen werden nur durch redundante Daten und allen damit verbundenen Problemen unterstützt
- ggf. muss ein eindeutiger Schlüssel OID in die Tabellen eingefügt werden, um Polymorphie abzubilden (wäre hier nötig, falls `<Name, Telnr>` kein Schlüssel ist)

Klassenhierarchien auf RDBS abbilden (6)

3. Eine Tabelle pro Klasse
→ jede Tabelle enthält nur die Attribute, die für diese Klasse spezifisch sind, keine geerbten Attribute

```
Person (OID, Name, Telnr)
Mitarbeiter (OID, EinstDatum)
LeitenderMitarb (OID, Bonus)
Kunde (OID, KundeID, Präferenzen)
```

Vorteile:

- Modell repräsentiert Objektorientierung am besten
- Polymorphismus wird sehr gut unterstützt: für jede Rolle ein Eintrag in entsprechender Tabelle vorhanden

Klassenhierarchien auf RDBS abbilden (7)

Vorteile: (Fortsetzung)

- Oberklassen können einfach modifiziert, neue Unterklassen eingefügt werden: nur eine Tabelle ändern

Nachteile:

- kompliziertestes Modell
- künstliches Attribut `OID` einfügen: entscheide, ob Person ein Mitarbeiter oder Kunde ist
- für jede Klasse muss eine Tabelle angelegt werden
- Zugriff auf Daten ist komplizierter und online-Abfragen dauern länger: Daten müssen aus verschiedenen Tabellen zusammengesucht werden

Test objektorientierter Software

- Klassentest: White-Box-Test (implementierungsbasiert)
- Integrationstest: Grey-Box-Test (schnittstellenbasiert)
- Systemtest: Black-Box-Test (spezifikationsbasiert)

zu lösende Fragen:

- in welcher Reihenfolge sollen Tests in OOPs erfolgen?
- wie sind OOPs zu strukturieren, damit erfolgreich getestet werden kann?

zwei strukturelle Richtungen für White-Box-Test:

- top-down entlang der Vererbungshierarchie
- entlang der Assoziationen bzw. Assoziationsketten

Test objektorientierter Software (2)

Klassentest: (Basis Klassendiagramme)
entlang der Klassenhierarchie von der generalisierten Klasse in Richtung spezialisierter Klasse testen (Oberklassen zuerst testen)

Ansätze für Klassentest

- Test über Testtreiber
- Built-In Test
- hierarchisch inkrementeller Test

Integrationstest (Basis Sequenzdiagramme)
entlang der Assoziationsketten das Zusammenspiel der Objekte testen (ist das gegenseitige Aufrufen der Methoden korrekt?)

Test objektorientierter Software (3)

Testendekriterien:

- Überdeckung der angebotenen Operationen einer Klasse

$$\frac{\#\text{ aufgerufene Operationen der Klasse}}{\#\text{ angebotener Operationen der Klasse}} \cdot 100\%$$

- Überdeckung der aufgerufenen Operationen

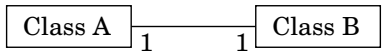
$$\frac{\#\text{ aufgerufene Operationen anderer Klassen}}{\#\text{ aufrufbarer Operationen anderer Klassen}} \cdot 100\%$$

- Überdeckung der ausgelösten/behandelten Ausnahmen einer Klasse

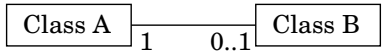
$$\frac{\#\text{ ausgelöster Ausnahmen}}{\#\text{ auslösbarer Ausnahmen}} \cdot 100\%$$

- usw.

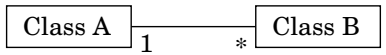
Assoziationen



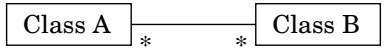
Ein ClassA-Objekt hat genau eine Assoziation auf ein ClassB-Objekt und umgekehrt.



Ein ClassA-Objekt hat entweder keine oder genau eine Assoziation auf ein ClassB-Objekt. Ein ClassB-Objekt hat eine Assoziation zu genau einem ClassA-Objekt.



Ein ClassA-Objekt hat beliebig viele Assoziation auf ClassB-Objekte. Ein ClassB-Objekt hat eine Assoziation zu genau einem ClassA-Objekt.



Ein ClassA-Objekt hat beliebig viele Assoziation auf ClassB-Objekte und umgekehrt.

Assoziationen (2)

Multiplizitätsangabe größer 1 (z.B. 1-zu-*-Beziehung) → erstelle Testfälle für vier klassische Anomalien bei Verwendung von Collections (z.B. Listen):

- **Einfügen** neuer Assoziationen an der richtigen Stelle in Collection?
- **Änderungen** wie Umsortieren auf Collection korrekt?
- **Löschen:** wird das richtige Objekt gelöscht und ist der Zustand der Collection nach dem Löschen korrekt?
- Problem der **inkonsistenten Verbindungen** bei bidirektionalen Assoziationen

Vererbung

- **strikt:** Unterklasse erweitert Oberklasse
  - * neue Testfälle für die in der Unterklasse neu hinzugekommenen Methoden entwickeln
  - * ererbte Methoden mit bestehenden Tests testen
- **nicht-strikt:** Basisfunktionalität wird überschrieben
  - * neue Testfälle für neue Methoden
  - * neue Testfälle für überschriebene Methoden
  - * neue Testfälle für alte Methoden, die überschriebene Methoden aufrufen

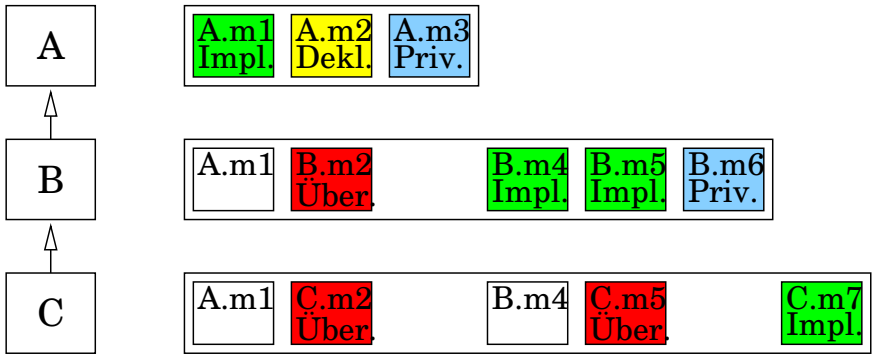
⇒ top-down Testreihenfolge: zuerst Oberklasse testen

Problem: Oberklassen, die nicht im Quelltext vorliegen

Vererbung (2)

Flattening:

- finden der Problembereiche durch Vererbung
- stelle für eine Klasse alle eigenen und ererbten Eigenschaften und Methoden explizit dar



Vererbung (3)

Substitutionsprinzip: eine abgeleitete Unterklasse muss sich in jedem Fall wie ihre Oberklasse verhalten → Exemplare einer Unterklasse müssen an Stelle von Exemplaren jeder Oberklasse einsetzbar sein

- gilt immer bei strikter Vererbung
- die Vorbedingungen jeder Methode müssen gleich oder schwächer sein als in der Oberklasse
- Nachbedingungen jeder Methode müssen gleich oder stärker sein als in der Oberklasse → es muss mindestens so viel definiert sein wie in der Oberklasse
- Klasseninvarianten müssen gleich oder stärker sein als in der Oberklasse → es dürfen mehr Zusicherungen gemacht werden

Vererbung (4)

das Zusicherungs-Verantwortlichkeitsprinzip:

- eine Klasse kann keine Zusicherungen auf die Eigenschaften ihrer Oberklasse erzwingen
- eine Klasse gibt nur Zusicherungen auf eigene Attribute

$\Rightarrow$ Verantwortlichkeiten für Attribute sind sauber getrennt

Abhängigkeits-Umkehrprinzip: eine Abhängigkeit findet nur zu Interfaces (vollständig abstrakten Klassen) statt $\rightarrow$ Entkopplung von Abhängigkeiten

Compiler testet obige Prinzipien nicht!

Vererbung (5)

typische Fehler in Vererbungshierarchien:

- Operatoren werden unvollständig überschrieben (C++)
- direkter Zugriff auf Attribute: Attribute sind `private`! Sonst: Seiteneffekte in Unterklassen führen zu Fehlern in Oberklasse
- Verstöße gegen das Zusicherungs-Verantwortlichkeitsprinzip (siehe Rechteck-Quadrat-Problem)
- Verstoß gegen das Substitutionsprinzip (Unterklasse akzeptiert nicht alle Nachrichten der Oberklasse oder die Unterklasse hinterlässt Status, der für Oberklasse nicht erlaubt ist)
- Unterklasse berechnet Ergebnisse, die nicht konform zu Zusicherungen und Invarianten der Oberklasse sind

Teststrategie bei Vererbung

- verhält sich die Funktionalität aus getesteten Basisklassen auch in abgeleiteten Klassen korrekt?
- können wir Basisklassentests für Tests der abgeleiteten Unterklassen wiederverwenden?

Zur Beantwortung der Fragen drei Axiome [Binder: Testing object-oriented systems - models, patterns, and tools. Addison-Wesley]

Anti-Komposition:

- Testfälle für Teilmodule sind nicht notwendig auch ausreichend für Test des Gesamtmoduls (Interaktionen zwischen den Teilmodulen beachten)
- oder: adäquater Test für Methode einer Klasse ist nicht notwendig auch für aufrufende Server-Klasse geeignet

Teststrategie bei Vererbung (2)

Anti-Extensionalität:

- die Testfälle für eine Implementierung decken nicht die Tests einer anderen Implementierung für die gleiche Anforderung ab
- oder: adäquater Test für bestimmte Implementierung einer Spezifikation ist für eine andere Implementierung nicht notwendig auch angemessen
- Beispiel: Tests für `Oberklasse::insert()` decken nicht die Tests für `Unterklasse::insert()` ab

Teststrategie bei Vererbung (3)

Anti-Dekomposition:

- Testabdeckung für einen Klassentest deckt nicht die Tests für benutzte Klassen ab
- oder: adäquater Test für Server-Klasse ist nicht notwendig für Methode der aufgerufenen Klasse sinnvoll
- Beispiel: Testfälle für `Konto` decken nicht die Tests für `Datum` und `Geld` ab

Modalität einer Klasse

non-modale Klassen

- **keine** Abhängigkeiten zwischen den Methoden
- Methoden können in beliebiger Reihenfolge aufgerufen werden, ohne sich gegenseitig zu beeinflussen
- der Zustand der Objekte hat keinen Einfluss auf das Verhalten der Methoden

uni-modale Klassen

- **sequenzabhängige** Einschränkungen in der Reihenfolge der Methodenaufrufe (z.B. `Ampel`-Klasse)
- Test muss Reihenfolge beachten
- Beispiel: Konto eröffnen, bevor Einzahlung möglich

Modalität einer Klasse (2)

quasi-modale Klassen

- **inhaltsabhängige** Einschränkungen
- Zustand nach der letzten Methodenausführung ist Voraussetzung für Ausführung der nächsten Methode
- Ergebnis einer Methode wird anders ausfallen, je nachdem in welchem Zustand sich das Zielobjekt befindet
- Beispiel: Konto gesperrt $\rightarrow$ jede Reihenfolge von Bewegungen wird abgelehnt
- Beispiel: `push` nur möglich, solange `Stack` nicht voll ist

Modalität einer Klasse (3)

modale Klassen

- **fachliche** Einschränkungen
- sowohl von der Reihenfolge der Methodenaufrufe als auch vom Zustand der Objekte abhängig
- um einen Fall zu testen, müssen andere Testfälle vorangegangen sein, um das Objekt in einen gewissen Zustand zu bringen

Testmuster:

- beschreibt typische Fehlersituationen
- listet notwendige Tests auf

Testmuster: Modale Klasse

nutzt zustandsraumbasiertes Testen → prüfe alle gültigen Zustände und Zustandsübergänge

- gültige Methodenaufrufe, die akzeptiert werden sollen
- illegale Methodenaufrufe, die abzulehnen sind
- resultierender Zustand für obige Methodenaufrufe
- Ergebnisse jedes Methodenaufrufs

Vorgehen beim Test: Basis Zustandsmodell

- in Zustandsbaum überführen → erstelle Testfall für jeden Weg von der Wurzel zu einem Blatt
- leite Wahrheitstabelle für jede bedingte Transition ab → teste ungültige Transitionen der Reihe nach durch

Testmuster: Modale Klasse (2)

typische Fehler:

- fehlende Transition: Methode wird abgelehnt, obwohl fachlich erlaubt
- falsche Aktion: Ergebnis einer Methode ist in einem bestimmten Zustand falsch
- ungültiger resultierender Zustand
- korrupter oder inkonsistenter Zustand
- Nebenweg erlaubt verbotenen Methodenaufruf

Testmuster: Modale Hierarchie

Vorgehen bei Vererbungshierarchie mit modalen Klassen?

- Klasse im Test ist konkrete Unterklasse und
  * erbt von einer modalen Oberklasse,
  * implementiert eigenes Verhalten sowie
  * erweitert das modale Verhalten der Oberklasse
- Klasse muss konform zu ihrem eigenen Zustandsmodell und dem seiner Oberklasse sein

⇒ Zustandsmodelle der Oberklassen müssen bekannt sein

⇒ Änderungen des Zustandsmodells einer Oberklasse gegen das eigene Zustandsmodell der Unterklasse prüfen

Testmuster: Modale Hierarchie (2)

typische Fehler:

- überschreibende Methode in Unterklasse erzeugt Zustandsmenge M_U mit $M_U \subset M_O$ oder $M_U \supset M_O$, ohne aber Zustände der Oberklasse ändern zu können
- Oberklasse hat Fehler, die nur bei Nutzung in der Unterklasse auftreten
- Anforderungen an Oberklasse nicht beachtet:
  * gültige Nachricht der Oberklasse wird abgelehnt
  * ungültige Nachricht der Oberklasse wird akzeptiert
  * falsche Aktion auf Nachricht der Oberklasse
- statusbasierte Fehler durch Erweiterungen in der Unterklasse

Testmuster: Modale Hierarchie (3)

Vorgehen beim Test:

- Voraussetzungen:
  * alle Oberklassen müssen ihre Tests bestanden haben
  * alle Methoden der beteiligten Klassen sind getestet mit Zweigabdeckung gleich 100%
- entwickle Zustandsmodell für zu testende Klasse, das auch ererbte Zustände berücksichtigt (Flattening)
- dann Test analog Testmuster Modale Klasse

Testmuster: polymorpher Server

polymorphe Server-Klasse:

- Oberklasse, die die Basis für eine große Anzahl konkreter Klassen liefert
- zum Zeitpunkt der Implementierung sind nur wenige konkrete Klassen bekannt
- wird verwendet, um für die Zukunft einfache Erweiterbarkeit zu schaffen

$\Rightarrow$ stelle sicher, dass das Substitutionsprinzip erfüllt ist

$\Rightarrow$ jede überschriebene Methode des Servers in der konkreten Unterklasse testen

Testmuster: polymorpher Server (2)

Vorgehen beim Test:

- jede polymorphe Methode im eigenen Klassenumfeld und in allen Client-Unterklassen, die sie erben, testen
- Client-Klassen dürfen durch ihre Überschreibungen keine Fehler in der Server-Klasse erzeugen $\rightarrow$ Substitutionsprinzip
- Testende: jede Server-Methode im Kontext jeder konkreten Unterklasse ausgeführt und getestet
- Test ist für jede neu hinzukommende Client-Klasse zu wiederholen, alte Klassen müssen nicht neu getestet werden, sofern die Server-Klasse unverändert bleibt
- Korrektheit der bestehenden Klassen lässt keine Rückschlüsse auf neue Client-Klassen zu