Softwarequalitätssicherung

Sommersemester 2003

Dr. Thomas Santen
Softwaretechnologie
TU Dresden

Nochmals Myers' Dreiecks-Problem

Die folgende Spezifikation formuliert Myers' klassisches Problem im Kontext einer objektorientierten Entwicklung.

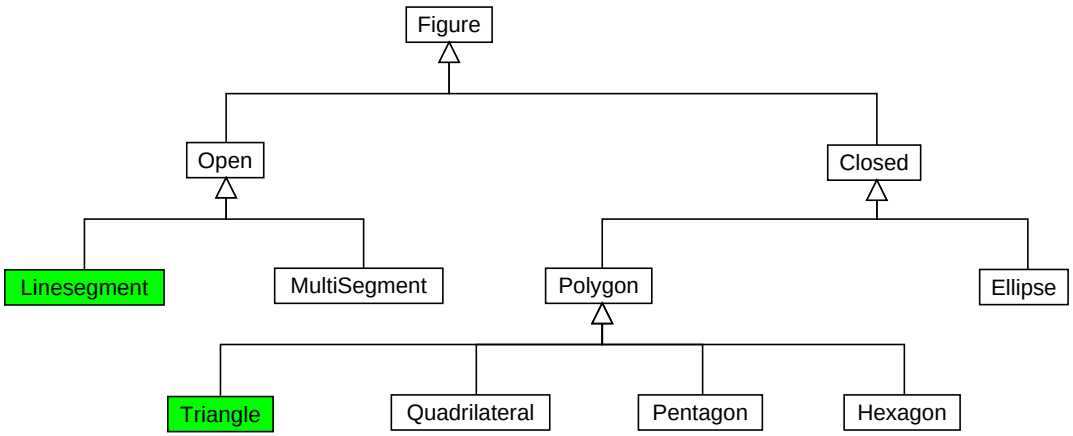
Spezifikation:

Die Länge eines *Linienabschnitts* wird durch eine ganze Zahl beschrieben. Ein Dreieck ist ein geschlossenes *Polygon*, das aus drei Linienabschnitten besteht. Die Art des so beschriebenen *Dreiecks* soll durch Abfragen bestimmt werden, die beschreiben, ob das Dreieck *gleichseitig*, *gleichschenklig* oder *rechtwinklig* ist.

Aufgabe:

Beschreiben Sie Testfälle für eine Klasse, die innerhalb einer Hierarchie von geometrischen Figuren nach der oben angegebenen Spezifikation Dreiecke implementiert.

Klassenhierarchie für geometrische Figuren



Die Klasse Triangle

```
class Triangle extends Polygon {
   public Triangle(LineSegment a, LineSegment b, LineSegment c)
   public void setA(LineSegment a)
   public void setB(LineSegment b)
   public void setC(LineSegment c)

   public LineSegment getA()
   public LineSegment getB()
   public LineSegment getC()

   public boolean is_isoceles()
   public boolean is_scalene()
   public boolean is_equilateral()

   public void draw(int r, int g, int b)
   public void erase()
   abstract float area()
   abstract Point center()                                    };
```

Klassische Testfälle – funktionale Äquivalenzklassen

	Testfall	Längen a	b	c	Ausgabe
1	gültiges rechtwinkliges Dreieck	5	3	4	scalene
2	gültiges gleichschenkliges Dreieck	3	3	4	isosceles
3	gültiges gleichseitiges Dreieck	3	3	3	equilateral
4	1. Permutation zweier gleicher Seiten	50	50	25	isoceles
5	2. Permutation zweier gleicher Seiten	50	25	50	isoceles
6	3. Permutation zweier gleicher Seiten	25	50	50	isoceles
7	eine Seite Null	1000	1000	0	ungültig
⋮	⋮		⋮		⋮
33	eine Seite mit Maximallänge	1	1	32767	ungültig

Wo steckt das Problem?

Warum sind diese Testfälle nicht ausreichend,
um die Klasse **`Triangle`** sorgfältig zu testen?

Besonderheiten objektorientierter Programmierung

Modularisierung: Methoden stehen immer im Kontext von Klassen.

Kapselung: Der Zustand von Objekten ist nur über die Schnittstelle (d.h. Methoden, öffentliche Attribute) lesbar.

Vererbung: Re-Definition ändert auch den operationalen Kontext nicht veränderter Methoden.

Objektidentität: Jedes Objekt hat eine Identität unter der es von vielen anderen Objekten angesprochen werden kann

Polymorphie: Der dynamische Typ einer Objektvariablen ist oft unklar und damit auch der tatsächlich für einen Methodenaufruf ausgeführte Code.

Abstrakte Klassen / Generizität: Parametrisierte Algorithmen (Aufrufe abstrakter Methoden) lassen sich nur für konkrete Instantiierungen ausführen.

Modularisierung

In imperativer Programmierung monolithisch als eine Prozedur realisierte Funktionen werden objektorientiert über mehrere Methoden verteilt.

---

im Beispiel:

- Eingabe der Seiten(-länge) und Ausgabe der Dreieckseigenschaft
- Konstruktion eines Dreiecks aus – vorher konstruierten – Linienabschnitten
- Abfrage `is_...` statt Ausgabeparameter

neuer Testfall:

➠ höchstens eine der Methoden `is_isoceles`,`is_scalene` und `is_equilateral` liefert `true`.

KAPSELUNG

- alle Objekte haben einen *internen* Zustand
- der interne Zustand ist nur über Methoden manipulierbar
- Seiteneffekte von Methoden auf den Zustand sind nicht direkt sichtbar
- fehlender Zugriff auf den internen Zustand erschwert die Auswertung von Tests

---

im Beispiel:

- Konstruktor und `set`-Methoden bestimmen Dreieck
- `get`-Methoden *sollen* intern gespeicherte Linienabschnitte zurückgeben

neue Testfälle:

- erzeugt der Konstruktor wirklich das Dreieck aus den Parametern?
- bleibt das Ergebnis von Tests über mehrere Aufrufe gleich?
- liefern `get`-Methoden vor und nach `draw` dasselbe Ergebnis?

OBJEKTIDENTITÄT

- jedes Objekt hat eindeutige Identität
- mehrere Referenzen auf dasselbe Objekt möglich *Aliasing*
- konkurrierende Manipulationen über Alias-Referenzen

---

im Beispiel:

- Linienabschnitte eines Dreiecks können von „anderswo“ verändert werden

neue Testfälle:

- sind die *Koordinaten* der Linienabschnitte vor und nach Ausführung von `draw`, `erase`, usw. gleich?

VERERBUNG

- Re-Definition ändert den Kontext auch unveränderter Methoden
- Aufbrechen der Kapselung
- über Super-Klassen verteilte Initialisierung
- kurze Code-Stücke mit sehr großen operationalen Effekten
- Vererbung kann verschiedene Zwecke haben: Code-Wiederverwendung, Problemzerlegung, Schnittstellendefinition (vgl. Polymorphie)

---

im Beispiel:

- ergibt Initialisierung einen konsistenten Gesamtzustand?
- „passen“ die von `Figure` und `Closed` geerbten Methoden zu den in `Triangle` (re-)definierten Methoden?

POLYMORPHIE

- starke Entkopplung von Klienten und Server-Objekten
- Änderungen im Server werden in Klienten erst zur Laufzeit sichtbar
- ausgeführter Code wird zur Laufzeit dynamisch bestimmt
- alle Möglichkeiten dynamischen Bindens sind intellektuell schwer zu fassen

---

im Beispiel:

- Instanzen von `Triangle` über die Schnittstellen von `Figure`, `Closed` und `Polygon` manipulierbar
- mit *Aliasing* kann das sogar konkurrierend geschehen

neue Testfälle:

- Sind die Methoden-Implementierungen von `Triangle` konsistent mit den Anforderungen an die entsprechenden Methoden aus `Figure`, aus `Closed`, aus `Polygon`? Auch bei *Aliasing*?

Abstrakte Klassen / Generizität

- Parametrisierung mit Methoden (abstrakte Klassen) oder ganzen Objekt-Typen
- nur nach Konkretisierung und Instantiierung testbar
- Parameterraum ist extrem groß – alle möglichen Instantiierungen

---

- Aussagen aus einem Test gelten im allgemeinen nur für die gewählte Instanz
- Generalisierungen von Testergebnissen sind problematisch

Zusammenfassung

- klassische Testtechniken müssen für objektorientierte Implementierungen erweitert werden
  - funktionsorientierte Verfahren decken Objekt-Interaktionen nicht ab
  - Überdeckungsanforderungen strukturorientierter Verfahren sind fragwürdig
- Ergänzende Tests müssen sich auf die spezifischen Fehlermöglichkeiten in objektorientierten Programmen beziehen (*fault model*).
- Vererbung und Polymorphie brechen Lokalität auf – deshalb:
  - Testen von Klassenhierarchien statt einzelnen Klassen
  - Regressionstesten unveränderten Codes