Objektorientierte Programmierung & Modellierung

Universität Koblenz-Landau
Institut für Informatik

Prof. Dr. Ralf Lämmel
Dr. Manfred Jackel

WS 2007/8

3. Teilklausur

15.02.2008

Gruppe A

Bitte in Druckschrift leserlich ausfüllen!

Name																									
Vorname																									
E-Mail-Adresse																			@uni-koblenz.de						
Matrikelnummer																									

Studiengang:
☐ Computervisualistik (Diplom)
☐ Informatik (Diplom)
☐Computervisualistik (BSc)
☐Informatik (BSc)
☐ Anglistik & Medienmanagement (BSc)
☐ B Ed _____
☐ Informationsmanagement (BSc)
☐ Diese Prüfungsleistung melde ich verbindlich als Freiversuch im Sinne der Prüfungsordnung an (nur alte PO).
☐ Diese Prüfungsleistung ist mein 2. Versuch (Nachklausur). Erstversuch im _____ (Semester).

Auswertung:

	1	2	3	4	GESAMT
Punkte					

Aufgabe 01 (12 Punkte)

Die folgende Klasse implementiert das Einzahlen und Abheben eines Kontos. Um nützlich zu sein, müssten weitere Eigenschaften implementiert werden, die wir aber weglassen, weil sie für die Lösung der Aufgabe nicht gebraucht werden.

```
interface Konto {
   void einzahlen(double betrag);
   void abheben(double Betrag);
   double kontostand();
}

public class UngesichertesKonto implements Konto {
   private double stand=0;

   public void einzahlen(double betrag) {
        stand += betrag;
    }

   public void abheben(double betrag) {
        stand -= betrag;
   }

   public double kontostand() { return stand;}
}
```

Nach dem Proxy-Entwurfsmuster soll eine Klasse ProxyKonto erstellt werden, die die Methoden einzahlen und auszahlen so absichert:

- es dürfen nur positive Beträge abgehoben und eingezahlt werden.
- Der Kontostand muss immer nichtnegativ sein.

Aufgabenstellung

a) Implementieren Sie ProxyKonto.
b) Geben Sie eine Klasseninvariante an, indem Sie die vorgegebene Methode invariant() ausprogrammieren.
c) Geben Sie die Vor- und Nachbedingungen der Methoden einzahlen und auszahlen an, indem Sie die assert-Anweisungen programmieren! Sichern Sie am Methodenanfang den Zustand in Hilfsvariablen, um die Nachbedingung formulieren zu können.

Einen Rahmen finden Sie auf dem nächsten Blatt.

```
public class ProxyKonto implements Konto {
   private Konto kto;

   public ProxyKonto(Konto kto) {
     this.kto=new UngesichertesKonto();
   }

   public double kontostand() { return kto.kontostand(); }


   protected void invariant() { // hier ergänzen

     assert kontostand() >= 0;

   }

   public void einzahlen(double betrag) { // hier ergänzen
      invariant();
      assert betrag > 0;
      double alterstand = kontostand();
      kto.einzahlen(betrag)
      assert kontostand() == alterstand + betrag;
      invariant();
   }

   public void abheben(double betrag) { // hier ergänzen
     invariant();
     assert betrag > 0 && kontostand() >= betrag;
     double alterstand = kontostand();
     kto.abheben(betrag);
     assert kontostand() == alterstand - betrag;
     invariant();
   }
}
```

Aufgabe 2 (12 Punkte)

Die Zeichenfolgen

0xabcdef.012345
0x.123
0x2.

stellen gebrochene Hexadezimalzahlen dar. Eine gebrochene Hexadezimalzahl

- beginnt immer mit „0x“,
- muss genau einen Dezimalpunkt enthalten,
- hat einen Numerus oder eine Mantisse oder beides.
- Numerus und Mantisse bestehen aus einem oder mehreren der Ziffern „0“ bis „9“ und der Kleinbuchstaben „a“ bis „f“.

/Aufgabenstellung

Schreiben Sie eine EBNF-Grammatik mit Startsymbol Hexdot, die diese Zahlen erzeugt.

➽

Hexdot = “0x“ (Case1 | Case2)

Case1 = Hexseq “.“ Hexseq?

Case2 = “.“ Hexseq

Hexseq = Hexdig Hexdig*

Hexdig = “0“|“1“|“2“|“3“|“4“|“5“|“6“|“7“|“8“|“9“|“a“|“b“|“c“|“d“|“e“|“f“

Aufgabe 03 (12 Punkte)

Diese Aufgabe umfasst 3 Multiple-Choice Cluster mit je 4 Ankreuzfragen. Für jedes Cluster gilt: wenn alle 4 Kreuze an der richtigen Stelle stehen, gibt es 4 Punkte für das Cluster. Ein falsches Kreuz gibt einen Punkt Abzug. Wer 2 richtige und 2 falsche Kreuzchen in einem Cluster macht, erhält 1+1-1-1=0 Punkte. Zum Trost: es gibt keine negativen Gesamtpunktzahlen, jedes Cluster bringt 0 bis 4 Punkte.

		Ja	Nein	Frage
a)				**Zusicherungen (assertions)**
	1.	☐	☒	Jede Vorbedingung gilt auch als Nachbedingung.
	2.	☐	☒	Die Klasseninvariante gilt vor und nach jeder Anweisung innerhalb der Methoden ihrer Klasse.
	3.	☐	☒	Die Variablen, die in der Klasseninvariante vorkommen, sollten nach Möglichkeit public sein.
	4.	☒	☐	In Java müssen Invarianten explizit geprüft werden.
b)				**ToPLS**
	1.	☒	☐	Die Semantik beschreibt die Bedeutung der Sprachkonstrukte einer Programmiersprache bezüglich eines bestimmten Bedeutungsraumes.
	2.	☐	☒	EBNF kann mehr Sprachen beschreiben als BNF.
	3.	☒	☐	Die abstrakte Syntax enthält nur die Strukturelemente, die für die Semantikdefinition wichtig sind.
	4.	☐	☒	Es gibt mindestens eine EBNF-Grammatik, die nicht kontextfrei ist.
c)				**Testing&Refactoring**
	1.	☒	☐	In Java kann man jede Methode so umschreiben, dass sie nur einen Parameter hat.
	2.	☐	☒	Es ist gut, viele Hierarchiestufen von Ableitungen zu haben.
	3.	☐	☒	Refactoring hat das Ziel, das Verhalten eines Programms zu ändern.
	4.	☐	☒	Beim Pull-up-Refactoring werden Methoden in die abgeleiteten Klassen verschoben.

Aufgabe 04 (12 Punkte)

Die nachfolgende Klassen ContainerComponent, Part und Container sind nach dem Entwurfsmuster Composite entwickelt worden. Jedes Kompositum ContainerComponent kann ein Einzelteil (Part) oder ein Container sein. Ein Containerkann beliebig viele ContainerComponent-Objekt aufnehmen. Jedes Einzelteil hat einen Preis (Container nicht). Die accept-Methode erlaubt die Anwendung des Visitor-Entwurfsmusters.

Aufgabenstellung

Schreiben Sie einen PriceVisitor, der den Gesamtpreis eines ContainerComponent-Elementes ermittelt und implementieren Sie die accept-Methode in den Klassen Part und Container. Das Interface Visitor geben wir vor.

Damit das Beispiel nützlich ist, müssten weitere Attribute und Eigenschaften implementiert werden, die wir aber hier weglassen, um die Sache übersichtlich zu halten. Sie brauchen diese zur Lösung auch nicht.

```
interface ContainerComponent {
      void add(ContainerComponent c);
      void remove(int i);
      ContainerComponent getChild(int i);
      void accept(Visitor v);
}
public class Part implements ContainerComponent {
      private double preis;

      public void add(ContainerComponent c) { }
      public ContainerComponent getChild(int i) {    return null; }
      public double getPreis() { return preis; }
      public void remove(int i) { }
      public void accept(Visitor v) {
      // ➽ hier müssen Sie ergänzen


        v.visit(this);



      }
}

import java.util.Vector;

public class Container implements ContainerComponent {
      private Vector<ContainerComponent> cc = new Vector<ContainerComponent>();
      private double preis = 0;

      public void add(ContainerComponent c) { cc.add(c); }
      public ContainerComponent getChild(int i) {
            return cc.elementAt(i);
      }
      public int getSize() {  return cc.size(); }
      public void remove(int i) { cc.remove(i); }
      public void accept(Visitor v) {
      // ➽ hier müssen Sie ergänzen


         v.visit(this);


      }
}
```

Die PriceVisitor-Klasse auf der nächsten Seite nicht vergessen! ➽

```
 interface Visitor {

    void visit(Part p);

    void visit(Container c);

}

public class PriceVisitor implements Visitor {

   private double gesamtpreis=0.0;     // hier die Preise aller enthal-
                                       // tenen Part-Objekte aufaddieren

   public double getGesamtpreis() { return gesamtpreis(); }

  // ➽ hier ergänzen

   public void visit(Part p) {
      gesamtPreis += p.getPreis();
   }

   public void visit(Container c) {
      int i = 0;
      ContainerComponent child = c.getChild(i);
      while (child != null) {
         child.accept(this);
         child = c.getChild(++i);
      }
   }

}
```