Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

19.August 2008

Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 2008 (BInf 211, BTInf 211, BMInf 211, BWInf 211)

Zeit: 75 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 10 Seiten.

Vorsicht: Lesen gefährdet die Dummheit!

Aufgabe 1:

Bitte lesen Sie diese Aufgabe vollständig durch, bevor Sie beginnen, die einzelnen Lösungsteile zu entwickeln.

In dieser Aufgabe soll ein einfacher Baukasten für die Verarbeitung von Folgen über den reellen Zahlen entwickelt werden. Folgen besitzen folgende Schnittstelle:

```
interface Sequence {
    double at(int i);
}
```

Die Funtion *at* berechnet den Wert der Zahlenfolge an der Stelle *i*. Legale Indizes sind alle natürlichen Zahlen ab 0. Für Aufrufe mit negativen Indizes soll in dieser Aufgabe keine Fehlererkennung gemacht werden.

Jede Folge kann durch ein Objekt einer Klasse repräsentiert werden, die dieses Interface *Sequence* implementiert. Dieses ist unflexibel, insbesondere können so zur Laufzeit nicht dynamisch neue Folgen erzeugt werden. Geschickter ist es, Klassen für einfache Folgen zu entwickeln und Kombinator–Klassen, dass heisst Klassen mit denen man aus bestehenden Folgen neue zusammensetzen kann.

Für einfache Folgen, gibt es einige einfache Klassen. Mit der Klasse *Const* können konstante Zahlenfolgen generiert werden.

```
public
class Const implements Sequence {
  private
  double value;

  public Const() { this(0.0); }

  public Const(double value) { this.value = value; }

  public double at(int i) { return value; }
}
```

Mit *Ident* erhält man die Zahlenfolge $0.0, 1.0, 2.0, \ldots$.

```
public
class Ident implements Sequence {
  public double at(int i) { return (double)i; }
}
```

Entwickeln Sie eine Klasse *Square* für die Folge der Quadratzahlen: $0.0, 1.0, 4.0, 9.0, 16.0...$.

```
public
class Square implements Sequence {

    ..........................................

    ..........................................

    ..........................................
   public double at(int i) {
      return

       ..........................................
   }
}
```

Aus einer Folge können neue Folgen erzeugt werden. Eine einfache Art ist die, jeden Folgenwert mit einem Faktor zu multiplizieren. Entwickeln Sie hierfür eine Klasse *Scale*:

```
public
class Scale implements Sequence {

    ..........................................

    ..........................................

    ..........................................
   public Scale( ..........................................) {

       ..........................................

       ..........................................
   }

   public double at(int i) {

       ..........................................

       ..........................................
   }
}
```

Folgen können durch Kombination zweier anderer Folgen konstruiert werden, zum Beispiel durch Addition oder Multiplikation der Folgeglieder. Die Addition der Folgen *new Ident()* und *new Const(1.0)* ergibt dabei die Folge $1.0, 2.0, 3.0, ...$ Entwickeln Sie eine Klasse *Mult* zur Multiplikation von Folgen.

```
public class Mult implements Sequence {

   ..........................................

   ..........................................

   public Mult( ..........................................) {

      ..........................................

      ..........................................
   }

   public double at(int i) {
      return

      ..........................................
   }
}
```

Geben Sie einen Ausdruck an, der die Folge für die Quadratzahlen erzeugt, ohne dass die Klasse *Square* genutzt wird. Bitte verwenden Sie auch keine anonymen Klassen.

....

....

Aus Folgen können Reihen gebildet werden. Eine Reihe ist eine Folge, die durch das Aufsummieren der Glieder einer gegebenen Folge entstehen. Dieses soll hier mit Hilfe der Klasse *Sum* realisiert werden. Der Algorithmus soll so arbeiten, dass für die Zahlenfolge $1.0, 1.0, 1.0, ...$ (*new Const(1.0)*) die Zahlenfolge $0.0, 1.0, 2.0, 3.0, ...$ entsteht. Versuchen Sie, eine nicht iterative Lösung für den Algorithmus zu entwickeln.

```
public class Sum implements Sequence {

    .....

    .....

    public Sum( ..... ) {

    .....
    }

    public double at(int i) {

        .....

        .....

        .....

        .....

        .....

        .....
    }
}
```

Aus einer Folge kann eine Differenzenfolge erzeugt werden, indem der i-te Wert der Ausgangsfolge vom $i + 1$-ten subtrahiert wird. Entwickeln Sie hierfür eine Klasse *Diff*.

```
public class Diff implements Sequence {

   ..........................................

   public Diff( ..........................................) {

      ..........................................
   }

   public double at(int i) {

      ..........................................

      ..........................................
   }
}
```

In einer Anwendung kann es vorkommen, dass für einen Index der Folgewert mehrfach berechnet wird. Dieses kann zu Laufzeitineffizenzen führen. Solche Ineffizienzen kann man beheben, indem man die Werte einer Folge spechert und diesen Cache mit der gleichen Schnittstelle *Sequence* versieht. Diese Ineffizenzen können in dieser Aufgabe insbesondere in der *Sum*-Klasse entstehen.

Nutzen Sie für die Implementierung einer solchen Cache-Klasse als Cachespeicher die Klasse *IntMap*. Der Konstruktor dieser Klasse soll eine neue leere Tabelle (*Map*) erzeugen. Mit *insert* kann man ein beliebiges Objekt unter dem Schlüssel *i* speicher. Mit *lookup* kann man testen, ob für einen Schlüssel *i* ein Wert gespeichert ist, und wenn ja welcher. *lookup* gibt *null* im Fall nicht gefunden und sonst die Referenz auf den gespeicherten Wert als Resultat.

```
public class IntMap {
   public IntMap() {
      // ...
   }
   public void insert(int k, Object a) {
      // ...
   }
   public Object lookup(int k) {
      // ...
   }
}
```

Entwickeln Sie eine Klasse *SequenceCache* für die Vermeidung der wiederholten Berechnung der Glieder einer Zahlenfolge.

```
public class SequenceCache implements Sequence {

   ..........................................

   ..........................................

   private

   ..........................................

   public SequenceCache( .......................................... ) {

      ..........................................

      ..........................................
   }
   public double at(int i) {
      double res;

      ..........................................

      ..........................................

      ..........................................

      ..........................................

      ..........................................

      ..........................................

      ..........................................
      return res;
   }
}
```

Würde eine Folge *new SequenceCache(new Sum(f))* den Cache in einer effizienten Art nutzen?

ja ☐ nein ☐

Begründung:

....

....

Aufgabe 2:

Gegeben sei die folgende Java-Klasse.

```
public class Buffer {
    private boolean empty = true;
    private Data value = null;

    public void put(Data d) {
        value = d;
        empty = false;
    }

    public Data get() {
        Data d = value;

        value = null;
        empty = true;

        return d;
    }
}
```

Diese Klasse implementiert einen Puffer für ein Exemplar aus der Klasse *Data*. Es soll dabei sicher gestellt sein, dass der Puffer entweder leer ist, angezeigt durch die Variable *empty*, oder voll, also eine Referenz auf ein *Data*-Objekt enthält. Diese Eigenschaft wird in der Variablen *empty* gespeichert.

Diese Klasse ist nicht *Thread*-sicher. Außerdem wird nicht sichergestellt, dass die *put*- und *get*-Operationen immer genau wechselseitig aufgerufen werden, so dass alle mit *put* geschriebenen Daten auch genau einmal mit *get* gelesen werden.

Erweitern Sie die *get*- und *put*-Methoden so, dass diese *Thread*-sicher sind und dass die zusätzlichen Bedinungen für den Einsatz in einem Erzeuger-Verbraucher-Muster für die *Buffer*-Klasse erfüllt sind.

Hinweis: in Java gibt es die Methoden *wait()* und *notify()*. *wait()* kann eine überprüfte Ausnahme *InterruptedException* auslösen.

Die modifizierte *put()*-Methode:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Die modifizierte *get()*-Methode:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .