Prof. Dr. Uwe Schmidt

ptlwedel
PHYSIKALISCH-TECHNISCHE LEHRANSTALT

18. August 2009

Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 2009 (IA 252)

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.

Verwenden Sie in den zu entwickelnden Java Programmteilen keine impliziten Konversionen zwischen einfachen Datentypen und kein Autoboxing und Autounboxing.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 10 Seiten.

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 von Textzeilen entwickelt werden. Diese Zeilenfolgen werden durch eine Schnittstelle *LineStream* beschrieben.

```
interface LineStream {
  boolean hasMoreLines();
  String nextLine();
}
```

Das Prädikat *hasMoreLines* zeigt an, ob noch Zeilen in dem Zeilenstrom enthalten sind, mit *nextLine* kann man dann die nächste Zeile aus dem Zeilenstrom lesen.

Um aus einem *Reader*–Object einen Datenstrom zu konstruieren, kann die Klasse *ReaderLineStream* verwendet werden.

```
import java.io.Reader;

public class ReaderLineStream
  implements LineStream {

  private Reader r;
  private int lookAhead;

  public ReaderLineStream(Reader r1) {
    r = r1;
    advance();
  }

  private boolean eof() {
    return lookAhead == -1;
  }

  private void advance() {
    try {
      lookAhead = r.read();
    }
    catch (Exception e) {
      lookAhead = -1;
    }
    if ( eof() ) {
      try {
        r.close();
      }
      catch (Exception e) { }
```

```
    }
  }

  public boolean hasMoreLines() {
    return ! eof();
  }

  public String nextLine() {
    StringBuffer l = new StringBuffer(" ");

    while ( ! eof() ) {
      if ( lookAhead == '\n' ) {
        advance();
        return new String(l);
      }

      if ( lookAhead == '\r' ) {
        advance();
        if ( lookAhead == '\n' )
          advance();
        return new String(l);
      }

      l.append((char)lookAhead);
      advance();
    }

    return new String(l);
  }
}
```

In Java wird bei Methodenaufrufen immer das dynamische Binden verwendet. Dieses ist von der Laufzeit her ineffizienter als statisches Binden. In der Klasse *ReaderLineStream* werden sehr häufig die Hilfsmethoden *advance* und *eof* aufgerufen. Ist es sinnvoll, die Anweisungen für diese Methoden an die Aufrufstellen einzukopieren und diese Hilfsroutinen zu löschen, um Laufzeit für dynamisches Binden zu vermeiden?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Analog zu einem *Reader*–Objekt kann man aus einem *String*–Objekt einen Zeilenstrom konstruieren. Dieses wird durch die Klasse *StringLineStream* beschrieben. Vervollständigen Sie die Klasse *StringLineStream* so, dass sie eine analoge Semantik besitzt wie die *Reader*–Klasse.

Hinweis: In der *String*–Klasse sind die Methoden *length* und *charAt* definiert.

```
public class StringLineStream
  implements LineStream {

  private String s;
  private int lookAhead;

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

  public StringLineStream(String s1) {

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

     ..........................................
    advance();
  }

  private boolean eof() {

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

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

  private void advance() {

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

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

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

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

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

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

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

```
  public boolean hasMoreLines() {
    return ! eof();
  }

  public String nextLine() {
    StringBuffer l = new StringBuffer(" ");

    while ( ! eof() ) {
      if ( lookAhead == '\n' ) {
        advance();
        return new String(l);
      }

      if ( lookAhead == '\r' ) {
        advance();
        if ( lookAhead == '\n' )
          advance();
        return new String(l);
      }

      l.append((char)lookAhead);
      advance();
    }

    return new String(l);
  }
}
```

Ist diese Implementierung der beiden Klassen *ReaderLineStream* und *StringLineStream* aus Software-technischer Sicht eine gute Lösung?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Weiter sei folgende Klasse gegeben:

```
public class CatLines
   implements LineStream {
   protected LineStream s;
   protected String lookAhead;

   public CatLines(LineStream s1) { init(s1); }
   protected CatLines() { }

   protected void init(LineStream s1) {
      s = s1;
      advance();
   }
   protected boolean isRedundantLine() {
      return false;
   }
   protected final void advance() {
      if ( s.hasMoreLines() ) {
         lookAhead = s.nextLine();
         if ( isRedundantLine() )
            advance();
      }
      else
         lookAhead = null;
   }
   public boolean hasMoreLines() {
      return lookAhead != null;
   }
   public String nextLine() {
      String res = lookAhead;
      advance();
      return res;
   }
}
```

Ist es aus Software-technischer Sicht sinnvoll, diese Klasse zu realisieren?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Macht es aus Software-technischer Sicht einen Unterschied, ob die Methode *advance* in der Klasse *CatLines* als private oder protected final deklariert wird?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Entwickeln Sie die einfachste Klasse *FastCatLines*, die die gleiche Funktionalität implementiert wie *CatLines*.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie eine Klasse *RemoveEmptyLines*, die aus einem Zeilenstrom alle Leerzeilen (Zeilen die keine Zeichen enthalten) löscht.

public class RemoveEmptyLines

```
   ..........................................
{
   public RemoveEmptyLines(LineStream s1) {

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

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


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

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

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

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

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

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

RemoveEmptyLines ist ein ganz spezieller Filter zum Selektieren von Zeilen. Eine Verallgemeinerung wird dann möglich, wenn man beliebige Prädikate auf die Zeilen eines Zeilenstroms anwenden kann. Hierzu sei folgende Schnittstelle gegeben:

```
interface StringPredicate {
  boolean isValid(String s);
}
```

Vervollständigen Sie die folgende Klasse *GrepLines*, mit der über ein Prädikat die Zeilen ausgewählt werden können, die dieses Prädikat erfüllen. Dieses ist eine Verallgemeinerung der Klasse *RemoveEmptyLines*. Die Klasse wird also eine ähnliche Struktur besitzen.

```
public class GrepLines

  ..............................................
  {
  private StringPredicate p;

  public GrepLines(StringPredicate p1, LineStream s1) {

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

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

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


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

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

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

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

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

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

Zwei (oder mehrere) Zeilenströme können zu einem neuen Strom kombiniert werden. Die Klasse *CombineLineStreams* legt hierfür das gemeinsame Muster fest:

```
abstract public class CombineLineStreams
   implements LineStream {
   protected LineStream s1;
   protected LineStream s2;

   public CombineLineStreams(LineStream l1, LineStream l2) {
      s1 = l1;
      s2 = l2;
   }
}
```

Vervollständigen Sie die Klasse *ConcatLineStreams*. Diese soll ermöglichen, dass zwei Zeilenströme zu einem konkateniert werden, ähnlich dem *cat*–Kommando unter UNIX.

```
public class ConcatLineStreams extends CombineLineStreams {
   public ConcatLineStreams(LineStream l1, LineStream l2) {
      super(l1,l2);
   }
   public boolean hasMoreLines() {

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .
   }
   public String nextLine() {

      . . . . . .

      . . . . . .

      . . . . . .
   }
}
```