Prof. Dr. Uwe Schmidt

ptlwedel
PHYSIKALISCH-TECHNISCHE LEHRANSTALT

10. August 2005

Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 2005 (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.

Viel Erfolg !

Diese Klausur besteht einschließlich dieses Deckblattes aus 9 Seiten

---

Aufgabe 1:

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:

Aufgabe 2:

Gegeben sei eine Klassenhierarchie zur Verarbeitung von einstelligen reellwertigen Funktionen, reelwertige Funktionen werden also durch Java–Objekte dargetellt.

Die Schnittstelle für die Funktionen wird als Java–Interface realisiert. Sie enthält eine Methode *at* zur Berechnung der Funktion an einer Stelle x. Außerdem ist eine Methode *derive* (ableiten) zur Berechnug der 1. Ableitung zu implementieren. Einige häufig verwendete Funktionen, drei konstante Funktionen (*zero*, *one*, *minus*1) und die Sinus–, Cosinus– und Exponetialfunktion, werden beim Laden der Schnittstelle erzeugt. Diese sind global zugreifbar.

Die Schnittstelle:

```
interface Function {
    double at(double x);
    Function derive();

    static final Function
        zero = new ConstFunction(0.0),
        one = new ConstFunction(1.0),
        minus1 = new ConstFunction(−1.0),
        sin = new Function() {
                public double at(double x) {
                    return Math.sin(x);
                }
                public Function derive() {
                    return cos;
                }
            },
        cos = new Function() {
                public double at(double x) {
                    return Math.cos(x);
                }
                public Function derive() {
                    return
                        new MultFunction(minus1, sin);
                }
            },
        exp = new Function() {
                public double at(double x) {
                    return Math.exp(x);
                }
                public Function derive() {
                    return exp;
                }
            };
}
```

Die Klasse für konstante Funktionen (*ConstFunction*):

```
public final
class ConstFunction implements Function {
  private
  double c;

  public ConstFunction(double c) {
    this.c = c;
  }

  public double at(double x) {
    return
      c;
  }

  public Function derive() {
    return
      zero;
  }
}
```

Erweitern Sie diese Klassenhierarchie um eine Klasse *AddFunction* für Funktionen der Form $f(x) = f_1(x) + f_2(x)$. Hinweis: $f'(x) = f_1'(x) + f_2'(x)$

Erweitern Sie diese Klassenhierarchie um eine Klasse $MultFunction$ für Funktionen der Form $f(x) = f_1(x) * f_2(x)$. Diese wird von dem Interface $Function$ schon verwendet. Hinweis: $f'(x) = f_1'(x) * f_2(x) + f_1(x) * f_2'(x)$

Ist es sinnvoll, die konkreten Klassen in diesem Beispiel als **final** zu deklarieren?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Kann das **final** Attribut vom Compiler im Allgemeinen zur Verbesserung des JVM–Codes genutzt werden?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Die Schnittstelle *Function* ist als Java–Interface deklariert. Ist diese Deklaration von Vorteil gegenüber einer Deklaration als abstrakte Klasse?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .
