Prof. Dr. Uwe Schmidt

ptlwedel
PRIVATE BERUFSFACHSCHULE

14. August 2014

Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 2014 (IA 252)

Zeit: 90 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.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

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

Tipp: Bei der Entwicklung der Lösung können kleine Skizzen manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 9 Seiten.

Aufgabe 1:

Bitte lesen Sie diese Aufgabe vollständig durch, bevor Sie die Lösungen für die Teilaufgaben entwickeln. Es gibt Vorwärtsreferenzen.

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

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.

Für einige häufig verwendete Funktionen, die Identität ($ident$), drei konstante Funktionen ($zero$, one, $minus1$) und die Sinus–, Cosinus– und Exponentialfunktion (sin, cos, exp), werden statische finale Variablen, also Konstanten, beim Laden der Schnittstelle initializiert. Die Objekte werden hier mit Hilfe anonymer Klassen erzeugt, die von $Function$ erben. Die Konstanten für diese Funktionsobjekte sind global nutzbar.

Die Schnittstelle:

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

    static final Function
        ident = new Function() {
                public double at(double x) {
                    return x;
                }
                public Function derive() {
                    return one;
                }
            };

    static final Function
        zero = new ConstFunction(0.0),
        one = new ConstFunction(1.0),
        minus1 = new ConstFunction(−1.0);

    static final Function
        sin = new Function() {
                public double at(double x) {
                    return Math.sin(x);
                }
                public Function derive() {
                    return cos;
                }
            };
```

```
    static final Function
        cos = new Function() {
                public double at(double x) {
                    return Math.cos(x);
                }
                public Function derive() {
                    return
                        new MultFunction(minus1, sin);
                }
            };

    static final Function
        exp = new Function() {
                public double at(double x) {
                    return Math.exp(x);
                }
                public Function derive() {
                    return exp;
                }
            };
}
```

Die Schnittstelle wird von der folgenden abstrakten Klasse (nächste Seite) beerbt:

```
abstract
public
class AbstractFunction implements Function {
    static public Function constFunction(double c) {
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
    }
    static public Function multFunction(Function f1, Function f2) {
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
    }
    static public Function addFunction(Function f1, Function f2) {
        if (f1 == zero) return f2;
        if (f2 == zero) return f1;
        return new AddFunction(f1, f2);
    }
}
```

Entwickeln Sie analog zur Funktion *addFunction* den Funktionsrumpf für *constFunction*, und zwar so, dass die Funktionsobjekte für die konstanten Funktionen $f(x) = 0$, $f(x) = 1$ und $f(x) = -1$ wiederverwendet werden.

Entwickeln Sie in gleicher Weise den Funktionsrumpf für *multFunction*, und zwar so das für Funktionen der Form $f(x) = 0 * f_2(x)$, $f(x) = f_1(x) * 0$, $f(x) = 1 * f_2(x)$ und $f(x) = f_1(x) * 1$ schon vorhandene Funktionsobjekte genutzt werden.

Die Klasse für konstante Funktionen ($ConstFunction$):

final
class ConstFunction

.....
{

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}

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

final
class AddFunction

.....
{

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}

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: $f'(x) = f_1'(x) * f_2(x) + f_1(x) * f_2'(x)$

final
class MultFunction

.....
{

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}

Ist es sinnvoll, die Klasse *AbstractFunction* in diesem Beispiel als **public** zu deklarieren?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Kann man die Funktionen aus *AbstractFunction* in den hier zu entwickelnden Funktionen nutzen?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

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

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

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 Effizienzsteigerung 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 abstrakter Klasse?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Veranschaulichen Sie durch ein Objektdiagramm, was für eine Objektstruktur in der Variablen f aufgebaut wird, wenn der folgende Ausdruck, der die Funktion $(x^2)'$ repräsentiert, ausgewertet wird:

Function f = multFunction(ident,ident).derive();