
Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 98 (IA 252)

Zeit: 60 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:

Wann ist es sinnvoll, eine Datenkomponente in einer Klasse mit dem Zugriffsrecht **protected** zu deklarieren?

1. wenn die darauf operierenden Funktionen auch **protected** sind. ja nein weiß nicht
 2. wenn der Zugriff auf Datenkomponenten aus Effizienzgründen aus abgeleiteten Klassen notwendig ist. ja nein weiß nicht
 3. wenn Basisklassen Zugriffsrechte haben sollen, der Benutzer aber nicht. ja nein weiß nicht
 4. wenn abgeleitete Klassen Zugriffsrechte haben sollen, der Benutzer aber nicht. ja nein weiß nicht
 5. wenn abgeleitete Klassen Zugriffsrechte haben sollen, aber nicht die Basisklassen. ja nein weiß nicht
 6. wenn der Benutzer Zugriff haben soll, aber abgeleitete Klassen nicht. ja nein weiß nicht
 7. wenn der Wartungsaufwand für die Basisklassen gering gehalten werden soll. ja nein weiß nicht
 8. wenn der Wartungsaufwand für abgeleitete Klassen gering gehalten werden soll. ja nein weiß nicht
 9. wenn nur die direkt aus der Klasse abgeleitete Klasse Zugriffsrechte haben soll, weitere abgeleitete Klassen aber nicht. ja nein weiß nicht
 10. immer. ja nein weiß nicht
 11. nie. ja nein weiß nicht
-

Aufgabe 2:

In der Objektorientierten Programmierung spielt die Wiederverwendung eine zentrale Rolle. Dabei können schon entwickelte, fertige Klassen von anderen Klassen beerbt oder benutzt werden.

Vererbung (hier mit Java Syntax)

```
class Y extends X { ... };
```

Benutzung

```
class Y {  
    X d;  
    ...  
};
```

Welches sind Vorteile der Benutzung gegenüber der Vererbung der Klasse X in der Klasse Y :

- | | | | |
|---|-----------------------------|-------------------------------|-------------------------------------|
| 1. Die modulare Verständlichkeit nimmt zu. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 2. Die Speicherverwaltung wird einfacher. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 3. Der Quellcode wird kürzer. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 4. Der Objektcode wird kürzer. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 5. Die Laufzeit des Programms verringert sich. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 6. Die Compilierzeit verringert sich. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 7. Die Konstruktoren brauchen nicht neu entwickelt werden. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 8. Der Quelltext wird selbstdokumentierend. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 9. Die Erweiterbarkeit ist verbessert. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 10. Die Flexibilität wird erhöht. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 11. Die Veränderung der Klasse X hat keinen Einfluß auf Programmcode außerhalb von Y . | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |
| 12. Die Namensgebung für Methoden in Y kann besser an die Aufgabenstellung angepaßt werden. | ja <input type="checkbox"/> | nein <input type="checkbox"/> | weiß nicht <input type="checkbox"/> |

Aufgabe 3:

Gegeben sei eine Klassenhierarchie zur Verarbeitung von einstelligen reellwertigen Funktionen, d.h. reelwertige Funktionen werden durch Java-Objekte dargestellt.

Die Schnittstelle für die Funktionen wird als interface realisiert. Sie enthält eine Methode *at* zur Berechnung der Funktion an einer Stelle x . Außerdem ist eine Methode *derive* (= ableiten) zur Berechnung der 1. Ableitung zu implementieren. Drei häufig verwendete Funktionen werden beim Laden der Schnittstelle erzeugt. Diese sind global zugreifbar.

Die Schnittstelle:

```
public
interface Function {

    public
    double at(double x);

    public
    Function derive();

    public static Function sin = new Sine();
    public static Function cos = new Cosine();
    public static Function zero = new ConstFunction(0.0);

}
```

Die Klasse für die Sinus-Funktion (*Sine*), die Klasse für die Kosinus-Funktion (*Cosine*) sei analog definiert.

```
public
final
class Sine implements Function {

    public
    double at(double x) {
        return
            Math.sin(x);
    }

    public
    Function derive() {
        return
            Function.cos;
    }
}
```

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

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Entwerfen Sie eine Schnittstelle *FindZero* für Algorithmen, die zu einer Funktion f in einem Intervall x_1 bis x_2 eine Nullstelle suchen.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

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:

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