Matecki / Nemirovski / Stumfol *Stand: 16.04.2020*

Praktikum Programmierung 2

Aufgabenblock III: Objektorientierte Programmierung mit Java

Aufgabe 1 – Fragen zur Theorie (max. 15 Punkte)

1.1 Begriffe zu Methoden (7P) ★☆☆

Was bedeutet:
- erweitern
- überschreiben
- überladen

von Methoden? Geben Sie jeweils ein Beispiel an. Was bedeutet die Annotation `@Override` in diesem Kontext?

1.2 Begriffe zu Vererbung (8P) ★☆☆

Erläutern Sie die Begriffe:
- Klassifikation
- Abstraktion
- Spezialisierung
- Vererbung
- Abstrakte Klasse
- Schnittstelle

Was ist der Unterschied zwischen einer abstrakten Klasse und einer Schnittstelle?

Aufgabe 2 – Entwurf von Klassenhierarchien mittels Spezialisierung und Abstraktion (max. 30 Punkte)

2.1 Spezialisierung (6P) ★☆☆

Ausgehend von einem allgemeinen Typ **`Person`** sollen mittels Spezialisierung die Typen **`Professor`** und **`Student`** gefunden und in Java implementiert werden.

- Eigenschaften einer **`Person`** sind:
  - Name vom Typ Zeichenkette
  - Geburtsdatum vom Typ **`Date`** aus einer vorherigen Aufgabe

- Zusätzliche Eigenschaften des Typs **`Student`** sind:
  - Eine ganzzahlige Matrikelnummer
  - Aktuelle Semestereinstufung ≥ 0

- Zusätzliche Eigenschaften eines Typs **`Professor`** sind:
  - Gehalt ≥ 100 und Gehalt ≤ 500
  - Lehrgebiet und Fakultät vom Typ Zeichenkette
  - Eintrittsdatum vom Typ **`Date`** aus einer vorherigen Aufgabe
  - Mindestalter beim Eintritt 20 Jahre

Details und Hinweise:

- Stellen Sie die gefundene Hierarchie in einem UML-Klassendiagramm dar.
- Legen Sie für dieses Aufgabenblatt ein neues Projekt **`A3`** an.
- Kopieren Sie Ihre **`Date`**-Implementierung aus dem vorherigen Aufgabenblatt. Diese verwenden Sie bitte von nun ab.
- Implementieren Sie alle Klassen in Java.
- Achten Sie bei der Implementierung der Konstruktoren und Methoden auf die korrekte Werte der Parameter.
- Implementieren Sie zu jeder Eigenschaft die dazugehörige getter- und setter-Methode.
- Die Klasse **`UniApp`** testet alle Klassen vollständig.

2.2 Abstraktion (12P) ★★☆

Die vorherige Teilaufgabe soll nun mit Hilfe der Abstraktion erweitert werden.

- Zusätzlich zu den oben genannten Typen soll nun ein weiterer Typ **`Assistant`** mit folgenden Eigenschaften eingeführt werden:
  - Gehalt ≥ 100 und Gehalt ≤ 500
  - Ganzzahlige Personalnummer ≥ 0

- Eintrittsdatum
- Mindestalter beim Eintritt 20 Jahre

- Finden Sie mittels Abstraktion einen allgemeineren Typ **`Employee`** der beiden Typen **`Professor`** und **`Assistant`** und weisen Sie diesem Typ die gemeinsamen Eigenschaften und Verhalten zu. Fügen Sie diesen Typ in die bereits vorhandene Klassifikation ein.

Details und Hinweise:
- Stellen Sie die neue Hierarchie in einem UML-Klassendiagramm dar.
- Passen Sie die bereits vorhandenen Klassen an.
- Erweitern Sie die Tests innerhalb der Klasse **`UniApp`**.
- Bei der Abnahme zeigen Sie zu den beiden Teilaufgaben 2.1 und 2.2 nur die neuesten Versionen Ihrer Sourcecodes.

2.3 Klassenhierarchie `Vehicle` (12P) ★★☆

Gegeben sei ein allgemeiner Typ **`Vehicle`** mit den Attributen:
- **`private owner : String`**
- **`private actV : integer { actV ≥ 0 }`**

und den Methoden:
- **`public accelerate(dV : integer) : void { actV += dV; dV ≥ 0 }`**
- **`public decelerate(dV : integer) : void { actV -= dV; dV ≥ 0 }`**
- **`public setVelocity(v : integer) : void { actV = v; v ≥ 0 }`**
- **`public getVelocity() : integer`**
- **`public setOwner(o : String) : void`**
- **`public getOwner() : String`**
- **`public print() : void`**

Entwickeln Sie die Klasse **`PKW`** als Erweiterung von **`Vehicle`**. Fügen Sie der Klasse folgendes Attribut hinzu:

- **`private maxV : integer { 0 ≤ actV ≤ maxV }`**

und folgende Methoden:

- **`public getMaxVelocity() : int`**
- **`public print(n : int) : void`**

hinzu. Die neue **`print()`**-Methode ruft intern die Methode **`print()`** der Basis- oder Superklasse auf und fügt der Ausgabe am Ende **`n`**-Leerzeilen hinzu.

Details und Hinweise:

- Welche Methoden werden überschrieben, welche erweitert und welche sind überladen?
- Stellen Sie die gefundene Hierarchie in einem UML-Klassendiagramm dar.
- Implementieren Sie beiden Klassen **`Vehicle`** und **`PWK`** in Java.
- Achten Sie bei der Implementierung der Konstruktoren und Methoden auf die korrekte Werte der Parameter und vor allem auf die Invariante **`actV ≥ 0`** - die ja stets eingehalten werden muss.
- Die Klasse **`VehicleApp`** testet alle Klassen vollständig.

Aufgabe 3 - Abstrakte Klassen, Vererbung und Schnittstellen (max. 55 Punkte)

Gegeben sei folgender Code der Klasse **`Airplane`** (Flugzeug):

```
class Airplane {
   public final static int DEFAULT_MAX_SPEED   = 100;
   public final static int DEFAULT_WINGS_COUNT =   1;

   private String manufacturer;                  // Herstellername
   private int maxSpeed = DEFAULT_MAX_SPEED;     // Max. Geschwindigkeit > 0
   private int wingsCount = DEFAULT_WINGS_COUNT; // Anzahl Flügelpaare > 0

   public Airplane(String manufacturer, int maxSpeed, int wingsCount) {
      this.manufacturer = manufacturer;
      setWingsCount(wingsCount);
      setMaxSpeed(maxSpeed);
   }

   public String getManufacturer() { return this.manufacturer; }
   public void setManufacturer(String manufacturer) {
       this.manufacturer = manufacturer;
   }

   public int getMaxSpeed() { return this.maxSpeed; }
   public void setMaxSpeed(int maxSpeed) {
      if (maxSpeed > 0)
         this.maxSpeed = maxSpeed;
   }

   public int getWingsCount() { return this.wingsCount; }
   public void setWingsCount(int wingsCount) {
      if (wingsCount > 0)
         this.wingsCount = wingsCount;
   }

   // per default soll ein Flugzeug keinen looping können
   public boolean getLooping() { return false; }

   @Override
   public String toString() {
     return "Airplane [manufacturer=" + manufacturer + ", maxSpeed=" +
            maxSpeed + ", wingsCount=" + wingsCount + "]";
   }
}
```

In den nachfolgenden Teilaufgaben soll nun inkrementell eine Klassenbibliothek mehrerer Flugzeugtypen entwickelt werden.

3.1 Erweiterung I "Klasse TransportAircraft" (6P) ★☆☆

Es ist eine Klasse zu entwickeln, die ein Verkehrsflugzeug mit folgender Spezifikation abstrahiert:

- Ein Verkehrsflugzeug ist ein Flugzeug, das genau ein Flügelpaar und die folgenden Eigenschaften hat:
  - Maximale Anzahl der Passagiere, > 0
  - Reisegeschwindigkeit (in der Regel etwas geringer als die maximale Geschwindigkeit), > 0
  - Name der Fluggesellschaft
- Ein Verkehrsflugzeug soll keinen Looping können.

Die Klasse stellt folgende Konstruktoren und Methoden bereit:

- Konstruktor, der die Attribute der Basisklasse und die spezifischen übergeben bekommt
- getter-Methoden aller Attribute
- alle „sinnvollen" setter-Methoden
- **`toString()`**-Methode, die alle Eigenschaften eines Verkehrsflugzeuges ausgibt

Details und Hinweise:

- Implementieren Sie die Klasse **`TransportAircraft`** in Java.
- Implementieren Sie eine Testklasse **`TestAirplane`**. Diese testet in einer statischen Methode **`testTransportAircraft()`** alle Konstruktoren und Methoden. Auch die Methoden, die von der Basisklasse vererbt wurden, sind hierbei zu testen.
- Achten Sie bei der Implementierung der setter-Methoden und Konstruktoren auf korrekte Werte der Parameter.
- Verwenden Sie für die Standardwerte Konstanten.

3.2 Erweiterung II "Klasse DoubleDecker" (12P) ★★☆

Es ist eine Klasse zu entwickeln, die einen Doppeldecker mit folgender Spezifikation abstrahiert:

- Ein Doppeldecker ist ein Flugzeug, das genau zwei Flügelpaare und die folgenden Eigenschaften hat:
  - es kann Loopings mit einer Mindestgeschwindigkeit von 320 km/h fliegen
  - das Cockpit kann offen oder geschlossen sein
  - im Falle eines offenen Cockpit kann kein Looping geflogen werden

Die Klasse stellt folgende Konstruktoren und Methoden bereit:

- Der erste Konstruktor hat folgende Parameter: Hersteller, maximale Geschwindigkeit und die Information, ob ein Cockpit offen oder geschlossen ist.
- Der zweite Konstruktor hat die Parameter Hersteller und maximale Geschwindigkeit. Dabei ist das Cockpit geöffnet.
- getter-Methoden aller Attribute
- alle „sinnvollen" setter-Methoden
- Die Methode **`getLooping()`** ergibt **`true`**, falls ein Looping möglich ist.
- **`toString()`**-Methode, die alle Eigenschaften eines Doppeldeckers ausgibt.

Details und Hinweise:

- Implementieren Sie die Klasse **`DoubleDecker`** in Java.
- Die Klasse soll nicht erweiterbar sein.
- Erweitern Sie die Testklasse **`TestAirplane`** um die statische Methode **`testDoubleDecker()`**, die alle Konstruktoren und Methoden testet. Auch die Methoden, die von der Basisklasse vererbt wurden, sind hierbei zu testen.
- Erweitern Sie die Testklasse **`TestAirplane`** um die statische Methode **`testAirplane()`**, die in einem Array mehrere Verkehrsflugzeuge und Doppeldecker speichert. Geben Sie in einer Schleife alle Flugzeuge aus.
- Achten Sie bei der Implementierung der setter-Methoden und Konstruktoren auf korrekte Werte der Parameter.
- Die Mindestgeschwindigkeit für einen Looping ist mit einer Konstanten **`LOOPING_SPEED`** zu realisieren.
- Die Methode **`getLooping()`** ergibt true, falls die maximale Geschwindigkeit grösser oder gleich 320 km/h ist und das Cockpit geschlossen ist.
- Achten Sie bei der Implementierung der setter-Methoden und Konstruktoren auf korrekte Werte der Parameter.

3.3 Erweiterung III "Abstrakte Klasse FlyingBody" (12P) ★★☆

Es ist eine abstrakte Klasse zu entwickeln, die ein Flugobjekt mit folgenden Daten

- Hersteller
- Maximale Geschwindigkeit, > 0

speichert und folgende Methoden / Konstruktoren bereitstellt:

- „Voll qualifizierter" Konstruktor
- getter- und setter-Methoden
- Abstrakte Methode **`calcArrivalTime()`**, die abhängig von Abflugszeit, Entfernung und Reisegeschwindigkeit die geschätzte Ankunftszeit ermittelt
- **`toString()`**-Methode, die alle Eigenschaften eines Fluggerätes ausgibt

Details und Hinweise:

- Implementieren Sie die Klasse **`FlyingBody`** in Java.
- Verwenden für die abstrakte Methode folgende Signatur:

  ```
  public LocalTime calcArrivalTime(LocalTime departure, int distance);
  ```

  Die Parameter haben jeweils die folgende Bedeutung:
  - **`departure`**: Abflugzeit
  - **`distance`**: Distanz zum Flugziel in km
- Verändern Sie die Klasse **`Airplane`** so, so dass diese die beiden Eigenschaften von der Klasse **`FlyingBody`** erbt, ohne diese selbst zu deklarieren. Die Methode **`calcArrivalTime()`** wird in dieser Klasse nicht implementiert.
- Beide Klassen **`TransportAircraft`** und **`DoubleDecker`** implementieren die Methode **`calcArrivalTime()`**.
- In der Klasse **`TransportAircraft`** ist die Methode **`calcArrivalTime()`** zu überladen: Ein dritter Parameter **`flyingWithMaxSpeed`** vom Type **`boolean`** legt fest, ob die maximale Geschwindigkeit (= **`true`**) oder die Reisegeschwindigkeit (= **`false`**) bei der Berechnung verwendet werden soll. Achten Sie bei der Implementierung beider Methoden auf eine redundanzfreie Lösung.
- In der Klasse **`DoubleDecker`** wird immer die maximale Geschwindigkeit für die Berechnung der Ankunftszeit verwendet.
- Ändern Sie den Namen der vorhandenen Testklasse **`TestAirplane`** in **`TestFlyingBody`** um. Erweitern Sie die bereits vorhanden Testmethoden um den Aufruf der **`calcArrivalTime()`**-Methoden.
- Achten Sie bei der Implementierung der setter-Methoden und Konstruktoren auf korrekte Werte der Parameter.
- Verwenden Sie für die Standardwerte Konstanten.

3.4 Erweiterung IV "Klasse Runway" (6P) ★☆☆

Es ist eine Klasse zu entwickeln, die eine Start- bzw. Landebahn mit folgenden Daten

- Länge der Bahn, > 0 (Standardlänge 100)
- Breite der Bahn, > 0 (Standardbreite 50)

abstrahiert und folgende Methoden / Konstruktoren bereitstellt:

- Defaultkonstruktor
- „Voll qualifizierter" Konstruktor
- Kopierkonstruktor
- getter- und setter-Methoden aller Methoden
- **`toString()`**-Methode, die alle Eigenschaften eines Fluggerätes ausgibt.

Details und Hinweise:

- Implementieren Sie die Klasse **`Runway`** in Java.
- Erweitern Sie die Testklasse **`TestFlyingBody`** um die statische Methode **`testRunway()`**, die alle Konstruktoren und Methoden testet.
- Achten Sie bei der Implementierung der setter-Methoden und Konstruktoren auf korrekte Werte der Parameter.
- Verwenden Sie für die Standardwerte Konstanten.

3.5 Erweiterung V "Schnittstelle Landable" (12P) ★★☆

Implementieren Sie ein Interface **`Landable`**, mit folgender Methodedeklaration:

```
public boolean landingCheck(Runway r);
```

Die Methode **`landingCheck()`** überprüft, ob die Möglichkeit der Landung auf einem als Parameter gegebenen Landebahn möglich ist oder nicht.

Details und Hinweise:

- Implementieren Sie die Schnittstelle **`Landable`** in Java.
- Ändern Sie die Klasse **`Airplane`**, so dass diese das Interface **`Landable`** implementiert. Dazu sind zwei weitere Eigenschaften innerhalb der Klasse **`Airplane`** erforderlich:
  - Erforderliche Mindestlänge der Landebahn, > 0
  - Erforderliche Mindestbreite der Landebahn, > 0
- Erweitern Sie die vorhandene Testklasse, die den Einsatz der Methode **`landingCheck()`** demonstriert.
- Verwenden Sie für die Standardwerte dieselben Konstanten.

3.6 UML-Diagramme und Hinweise zur Abgabe (7P) ★☆☆

3.6.1 UML-Diagramm

Erstellen Sie ein UML-Klassendiagramm für die als Lösung der vorherigen Teilaufgabe entstandenen Klassenhierarchie. Nutzen Sie dafür ein beliebiges Tool oder zeichnen Sie das Diagramm auf einem Papierblatt von Hand.

Die darzustellenden Klassen / Schnittstellen beinhalten Klassennamen, Attribute und fachlichen Methoden (= Methoden, die Logik implementieren)!

3.6.2 Hinweis zur Abnahme

Bei der Abnahme werden nur die Endversionen der entwickelten Klassen und Interfaces berücksichtigt. Zwischenversionen brauchen nicht vorgezeigt werden.