Universität Augsburg

Fakultät für Angewandte Informatik
Lehrprofessur für Informatik
Prof. Dr. Robert Lorenz

SS 2012

16.05.2012

Betreutes Programmieren Vorlesung Informatik II, Blatt 4

Musterlösung: ca. 86 Codezeilen*

ToDo-Manager

Der ToDoManager hilft Ihnen, Aufgaben ihrer Wichtigkeit nach zu sortieren. Sie geben Aufgaben mit ihrer Beschreibung und einer Priorität in das System ein, und der Manager wählt eine geeignete Sortierung für Sie aus, die er Ihnen dann präsentiert.

Aufgabenstellung

Setzen Sie das UML-Klassendiagramm aus Abbildung 1 in Java-Code um und **berücksichtigen Sie dabei die weiteren Beschreibungen und Hinweise nach dem Diagramm.**

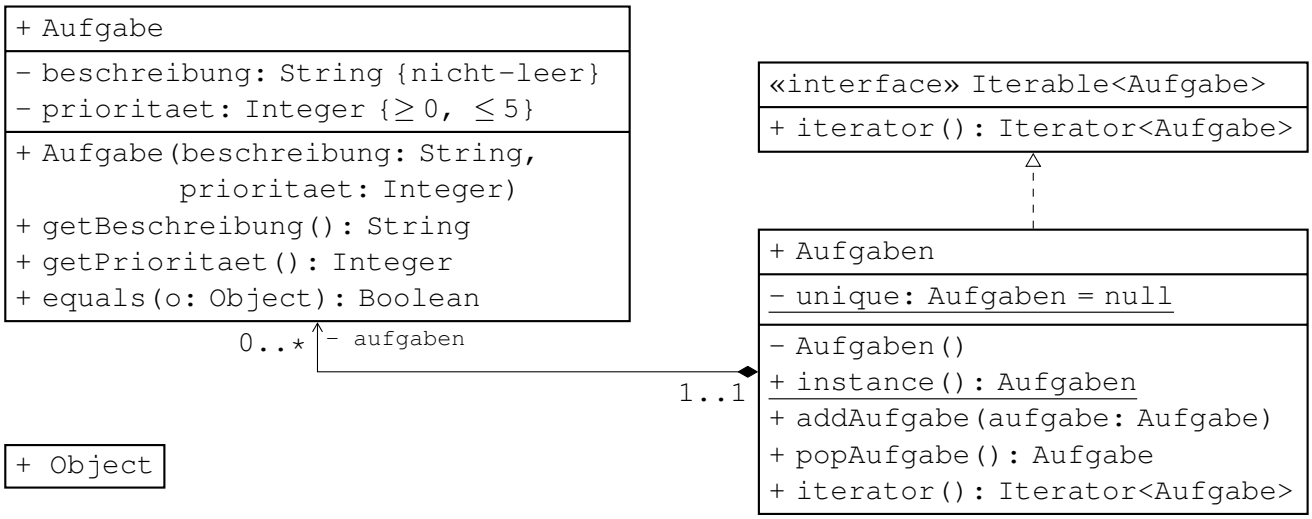

Abbildung 1: UML-Klassendiagramm der Datenkomponenten

Überlegen Sie, wie Sie die Einschränkungen der Attribute von **`Aufgabe`** umsetzen können.

Die Methode **`equals()`** soll genau dann **`true`** zurückliefern, wenn sowohl die Beschreibung als auch die Priorität von zwei **`Aufgabe`**n gleich sind.

Die Methode **`addAufgabe()`** soll eine neue Aufgabe nur dann zum System hinzufügen, wenn nicht bereits eine identische Aufgabe im System vorhanden ist.

Die Methode **`popAufgabe()`** gibt die dringlichste Aufgabe, die im System enthalten ist, zurück und löscht sie aus dem Container.

*__Nicht gezählt__: Leerzeilen, Kommentare, `import`s. __In eigenen Zeilen__: `{` und `}` (ca. 30x), jede Anweisung, außer bei Deklarationen gleichen Typs (z. B. `int a, b, c = 0`, ...) oder bei einer einzelnen Anweisung nach einem `if` oder einem `else`. __Nicht benutzt__: `?:`-Operator.

Wie Sie sehen, soll die Klasse **Aufgaben** die Schnittstelle **Iterable** implementieren. Weiter erkennen Sie das Singleton-Muster, in dem **Aufgaben** zu erstellen ist.

Erstellen Sie eine **main()**-Methode, um Ihre Implementierung zu testen.

Hinweise

Sollten Sie noch grundlegendere Hinweise als die in diesem Abschnitt gegebenen benötigen, dann werfen Sie einen Blick in den Anhang ab Seite 3. Dort sind weitergehende Erklärungen zusammengestellt.

Beginnen Sie mit der Klasse **Aufgabe**. Eine Möglichkeit die Einschränkungen zu berücksichtigen ist, im Konstruktor die übergebenen Parameter abzufragen und ggf. die Attribute passend zu initialiseren. [1] Wenn Sie in der Methode **equals()** die beiden Beschreibungen miteinander vergleichen, benutzen Sie die Methode **equalsIgnoreCase()** der Klasse **String**. Auf diese Weise wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Bei der Klasse **Aufgaben** schreiben Sie zunächst das Singleton-Muster, das Sie bereits aus der Vorlesung kennen.

Überlegen Sie nun, welchen Container von Java Sie benutzen könnten, um die einzelnen **Aufgabe**n zu verwalten. Recherchieren Sie in der Dokumentation nach Klassen, die eine Liste oder ein Warteschlange repräsentieren, und benutzen Sie weitestgehend deren Methoden.

Sollte Ihnen die Implementierung der eigentlichen Verwaltungsoperationen nicht in der angegebenen Zeit gelingen, ist das kein Beinbruch und nicht das Kriterium für die Vergabe des Punktes! Das wesentliche Augenmerk sollten Sie auf die Umsetzung des UML-Diagramms in Code legen, wozu allerdings das Singleton-Muster, die Einschränkungen sowie die Schnittstelle zählen. Lassen Sie in diesem Fall die Methoden **addAufgabe()** *und* **popAufgabe()** *leer.*

Schreiben Sie abschließend eine eigene Klasse, die die **main()**-Methode enthält und in der Sie verschiedene **Aufgabe**n erzeugen und in den Container einfügen. Testen Sie, ob die Reihenfolge richtig ist, ob **Aufgabe**n nicht doppelt eingefügt werden und ähnliches.

[1] Wird bspw. eine Priorität größer als 5 übergeben, wird das Attribut **prioritaet** trotzdem auf den Wert **5** gesetzt.

Weitergehende Erklärungen

B.1 Konstruktoren

Eine Erklärung zu Konstruktoren findet sich bereits auf dem letzten Blatt zum Betreuten Programmieren. Das Blatt finden Sie auf der Veranstaltungsseite im DigiCampus.

B.2 Singleton-Muster

Grundsätzlich handelt es sich bei Mustern um allgemeine Lösungen zu wiederkehrenden Problemen. Diese Lösungen sind meistens in Form von Objekten, deren Methoden und Beziehungen zwischen den Objekten gegeben, falls es sich um mehrere handelt.

Das Singleton-Muster löst das Problem, wenn man sicherstellen möchte, dass von einer Klasse höchstens ein Objekt erzeugt werden kann. Aus der Vorlesung wissen Sie, dass dies bei Containerklassen sinnvoll ist. Ein Container kann z. B. ein Lager repräsentieren und zur Verwaltung von Geräten eingesetzt werden. Möchte man nun sichergehen, dass nur ein Lager im System existiert, erstellt man **Lager** im Singleton-Muster.

```
public class Lager
{
  private static Lager unique = null;

  public static Lager instance()
  {
    if (unique == null) unique = new Lager();

    return unique;
  }

  private Lager()
  {
    ⟨Rumpf des privaten Konstruktors⟩
  }

  ⟨sonstige Attribute und Methoden für Lager⟩
}
```

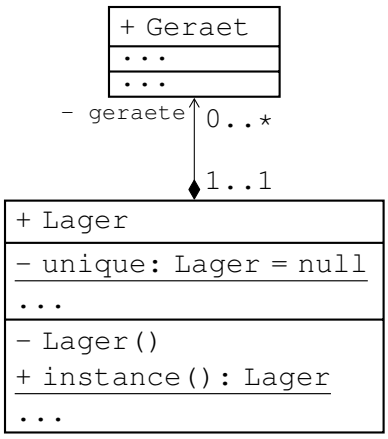

Abbildung 2: Singleton in UML

Das statische Attribut **unique** nimmt das einzige Objekt der Klasse auf. Von außerhalb der Klasse kann kein Objekt erzeugt werden, da der Konstruktor **private** ist. Da die Methode **instance()** ebenfalls statisch ist, kann Sie ohne ein Objekt, einfach über **Lager.instance()**, aufgerufen werden. Auf diese Weise bekommt man Zugriff auf das einzige Objekt der Klasse. Alle weiteren Attribute und Methoden werden wie gehabt in die Klasse eingefügt.

In UML erkennt man das Singleton-Muster, neben den beiden statischen Komponenten und dem privaten Konstruktor, vor allem an der Multiplizität `1..1`, die angibt, dass ein Objekt genau einmal existiert. Dies kann man in Abbildung 2 sehen. Dort ist zusätzlich zu sehen, dass `Lager` ein Container ist, der beliebig viele Objekte von `Geraet` verwaltet. Die genauen Verwaltungsoperationen wurden hier der Einfachheit halber weggelassen.

B.3 Vererbung

Dabei handelt es sich um einen integralen Bestandteil der *objektorientierten Programmierung* (OOP). Die Idee dahinter ist, dass man bereits bestehende Klassen erweitern kann. Aus diesem Grund wird Vererbung in Java mit dem Schlüsselwort **`extends`** eingeleitet. Eine abgeleitete Klasse – oder *Kindklasse* – übernimmt alle Attribute und Methoden der ursprünglichen Klasse – der *Elternklasse* – kann sie aber erweitern und ergänzen.

Eine Vererbung drückt eine *ist-ein*-Beziehung aus. Wenn es mehrere gleichartige Klassen gibt, die alle eine gemeinsame Basis besitzen, so ist dies ein guter Kandidat für eine Vererbungshierarchie.

Bspw. sind Batman und Superman beides Superhelden. Ein Superheld hat einen Tarnnamen, unter dem er ein normales Leben führt und den er nicht einfach preisgibt, und einen Heldennamen, unter dem er in der Öffentlichkeit auftritt. In Abbildung 3 ist die gemeinsame Basisklasse in UML dargestellt.

```java
public class Superheld
{
  private String heldenname;
  private String tarnname;

  protected String specialGimmick;

  public Superheld(String tarnname, String heldenname)
  {
    this.tarnname = tarnname;
    this.heldenname = heldenname;
  }

  public String getHeldenname()
  {
    return heldenname;
  }

  protected String getTarnname()
  {
    return tarnname;
  }
}
```

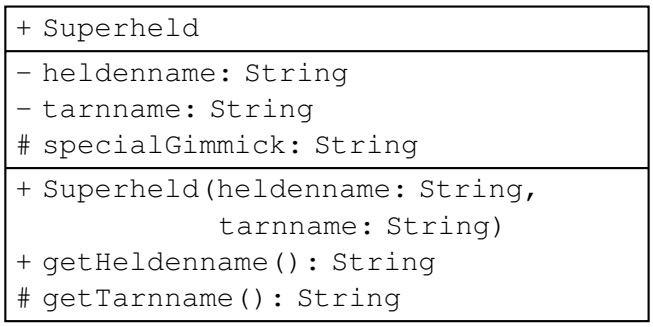

Abbildung 3: UML Superheld

Der Code ist die direkte Entsprechung des Diagramms in Java. Wie nach dem Geheimnisprinzip üblich, sind die Namen als private Attribute angelegt. Den Heldennamen darf jeder erfahren, weswegen der entsprechende Getter öffentlich zugänglich ist. Der Getter für den Tarnnamen ist mit dem Schlüsselwort **`protected`** versehen (in UML mit #), was bedeutet, dass er nur von Kindklassen aus aufrufbar ist. Wenn Batman also ein Superheld ist, dann hat er über den Getter Zugriff auf seinen Tarnnamen, aber sonst niemand. Er sollte sich dann eine Methode zulegen, die über eine Art Zugriffskontrolle seinen Tarnnamen an Eingeweihte herausrückt, um seine Identität nicht auffliegen zu lassen.

Da die Namen der Parameter im Konstruktor den Namen der Attribute gleichen, sind die Attribute im Konstruktor verdeckt. Sie müssen über das Schlüsselwort **`this`** angesprochen werden, das eine Referenz auf das aktuelle Objekt darstellt.

Jeder Superheld hat auch ein besonderes Gimmick. In diesem Fall wird das Geheimnisprinzip umgangen, da ein Superheld nicht erst jedesmal seine Eltern bitten kann, wenn er sein Gimmick einsetzen will. Das würde einfach zu lange dauern. Das Attribut wird deswegen mit **protected** versehen.

```
public class Batman extends Superheld
{
  ⟨zusätzlicher Batkram⟩

  public Batman()
  {
    super("Bruce Wayne", "Batman");

    ⟨aktiviere batmäßige Coolness und anderen Batkram⟩
  }

  ⟨coole Bataktionen⟩
}
```

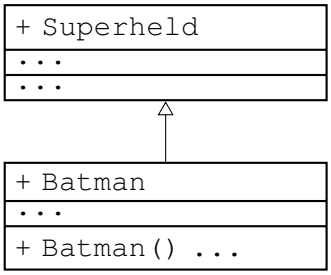

Abbildung 4: UML Batman

Abbildung 4 kann man entnehmen, dass Vererbung in UML durch eine durchgezogene Linie mit einer leeren Pfeilspitze dargestellt wird. In UML spricht man von *Generalisierung* und nicht von Vererbung, da man die gegenteilige Richtung betrachtet. Vererbung ist gewissermaßen eine *Spezialisierung*.

Wenn eine Kindklasse eine Methode der Elternklasse überschreibt, indem sie eine Methode gleichen Namens und gleicher Signatur – bei Konstruktoren ist hier nur die Parameterliste von Bedeutung – anlegt, dann kann man in Java von innerhalb dieser Methode die Methode der Elternklasse über das Schlüsselwort **super** aufrufen. Innerhalb eines Konstruktors kann jeder Konstruktor der Elternklasse mit der passenden Parameterliste aufgerufen werden. Auf diese Weise kann bestehende Funktionalität erweitert werden.

Batman hat mehrere Gimmicks und überschreibt deshalb das Attribut **specialGimmick** aus der **Superheld**-Klasse und macht es zu einem Feld.

```
private String[] specialGimmick;
```

Das Attribut der Elternklasse ist nun verdeckt kann aber über **super** angesprochen werden, sodass die *batmäßige Coolness* im Konstruktor auf diese Weise gesetzt werden kann.

```
super.specialGimmick = "batmäßige Coolness";
```

Die anderen Gimmicks werden direkt über das Feld angesprochen. Mehr als zehn Gimmicks schafft der Batgürtel allerdings auch nicht.

```
specialGimmick = new String[10];
```

Eine Kindklasse kann also auch Attribute einer Elternklasse überschreiben und deren Typ verändern. Beide Attribute stehen danach zur Verfügung.

In Java gibt es keine Mehrfachvererbung. Jede Klasse hat also genau eine Elternklasse. Für diese Regel gibt es eine Ausnahme: **Object** ist szsg. die *Adamklasse* und wurde direkt in der Sonne geboren. Alle Klassen sind direkt oder indirekt über eine Vererbungshierarchie Kinder von **Object**.

Schreibt man sich also in Java eine neue Klasse und lässt die **extends**-Klausel weg, dann wird sie automatisch durch **extends Object** aufgefüllt.

B.3.1 Die Methode `equals()`

Diese Methode wird in Java eingesetzt, um herauszufinden, ob zwei Objekte eigentlich das gleiche Objekt sind. Die Methode besitzen alle Objekte in Java, da jede Klasse automatisch von der Klasse **`Object`** erbt und **`equals()`** dort definiert ist.

Zunächst verwundert der Parameter vom Typ **`Object`**. Auf diese Weise wird die Universalität der Methode sichergestellt. Da jedes Objekt in Java auch einen Anteil vom Typ **`Object`** besitzt, können so Äpfel mit Birnen verglichen werden.

Um auf Attribute und Methoden zuzugreifen, die eine bestimmte Klasse besitzt, muss man den Parameter geeignet casten. Dies ist allerdings nur zulässig, falls das Objekt tatsächlich auch vom Zieltyp ist. Um herauszufinden, welcher Typ wirklich hinter einem Parameter steckt, gibt es den **`instanceof`**-Operator. Er wird zusammen mit einem Objekt und einem Klassennamen aufgerufen und gibt ein boolesches Ergebnis [2] zurück.

So wird eine **`equals()`**-Methode fast immer nach dem gleichen Schema ablaufen. Zuerst wird auf Klassenzugehörigkeit getestet, dann wird ggf. ein Typecast auf die entsprechende Klasse vorgenommen und **`this`** mit dem gecasteten Objekt verglichen. Der Vergleich wird üblicherweise auf den Attributen der beiden Objekte durchgeführt.

```
public boolean equals(Object o)
{
  if (o instanceof Klasse)
  {
    Klasse tmp = (Klasse) o;

    ⟨Vergleiche Attribute von this und tmp⟩
  }

  return false;
}
```

B.3.2 Polymorphismus

Hierbei handelt es sich um einen weiteren integralen Bestandteil von OOP, der mit der Vererbung zusammenhängt und eine große Stärke dieses Programmierparadigmas darstellt.

Polymorphismus beschreibt die Vielgestaltigkeit von Objektreferenzen. Da ein Objekt einer Kinderklasse immer auch ein Objekt einer seiner Elternklassen ist, kann es auch einer Variablen solchen Typs zugewiesen werden.

```
Klasse klasse = new Klasse();

Object o = klasse;
```

Ein weiteres Objekt wird im Folgenden benötigt.

```
Klasse anderes = new Klasse();
```

Nun kommt die Magie. Der folgende Aufrauf ruft ganz klar die Methode **`equals()`** der Klasse **`Klasse`** auf.

```
klasse.equals(anderes);
```

Aber welche Methode wird in dieser Zeile aufgerufen?

```
o.equals(anderes);
```

[2] Also **`true`** oder **`false`**.

Dieselbe Methode wie im vorherigen Aufruf, denn das Objekt, das hinter der Referenz steckt, weiß, dass es ein Objekt der Klasse **`Klasse`** ist.

Solche Methoden werden *virtuelle Methoden* genannt, da es nur vom Objekt selbst und nicht der Trägervariablen abhängt, welche Methode – also wo in der Vererbungshierarchie – aufgerufen wird. In Java sind Methoden immer virtuell. Das bedeutet, dass in Java grundsätzlich immer die „jüngste“ Methode ausgeführt wird, die existiert.

B.4 Schnittstellen

Eine Schnittstelle wird in UML durch «`interface`», wie in Abbildung 5 gezeigt, gekennzeichnet. Eine Schnittstelle entspricht dem Konzept der Vorgabe von Methoden. Wenn eine Klasse eine bestimmte Schnittstelle implementiert, dann muss sie ganz bestimmte Methoden zur Verfügung stellen; eben diejenigen, die die Schnittstelle vorgibt.

In Java ähnelt eine Schnittstelle einer Klasse. Dabei wird nur das Schlüsselwort **`class`** gegen das Schlüsselwort **`interface`** ausgetauscht. Weiter dürfen keine Attribute mehr enthalten sein und die Methoden werden nur noch deklariert, enthalten also keine Rümpfe.

```
public interface Geraet
{
  public void einschalten();
  public void ausschalten();
  public boolean istBereit();
}
```

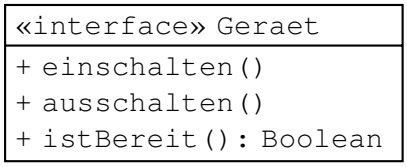

Abbildung 5: Schnittstelle in UML

Nun können verschiedene Klassen diese Schnittstelle implementieren. Auf diese Weise entsteht eine Sammlung von Klassen, die alle eine bestimmte Funktionalität zur Verfügung stellen: Alle Geräte kann man ein- und ausschalten und darauf überprüfen, ob sie benutzt werden können.

```
public class Kuehlschrank implements Geraet
{
  ⟨eigene Attribute und Methoden⟩

  public void einschalten()
  {
    ⟨eigene Implementierung dieser Methode⟩
  }

  public void ausschalten()
  {
    ⟨eigene Implementierung dieser Methode⟩
  }

  public boolean istBereit()
  {
    ⟨eigene Implementierung dieser Methode⟩
  }
}
```

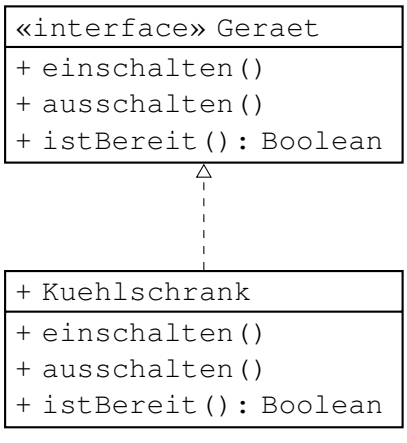

Abbildung 6: Implementierte Schnittstelle in UML

Bspw. verhält sich ein Kühlschrank wie ein Gerät oder kann wie ein Gerät benutzt werden. Er implementiert also die Schnittstelle `Geraet` , was in Java durch das Schlüsselwort **`implements`** und in UML durch eine gestrichelte Linie mit einer leeren Pfeilspitze, wie in Abbildung 6 zu sehen, ausgedrückt wird.