Klassenentwurf II

Lernziele

- Sie festigen Ihre Kenntnisse über das Konzept der statischen Methoden sowie deren Verwendung.
- Sie wissen, wie Sie beliebige Klassenvariablen geeignet initialisieren und Sie können dieses Wissen in konkreten Beispielen auch anwenden.
- Sie können allfällige Probleme in einfachen Klassendesigns erkennen und beheben.
- Sie können Enum Typen schreiben und geeignet einsetzen.

Aufgabe 1 (Auf Papier!)

Welche Aussagen sind in Bezug auf statische Methoden korrekt?

- ☑ Statische Methoden kann man aufrufen, ohne zuerst ein Objekt der jeweiligen Klasse erzeugen zu müssen.
- ☑ Objektmethoden können auf Klassenmethoden und Klassenvariablen der zugehörigen Klasse zugreifen.
- ☑ Im Code einer statischen Methode kann man die Instanzvariablen oder Objektmethoden der zugehörigen Klasse nicht direkt verwenden, da eine statische Methode eine Klassenmethode ist und somit auf der Klasse und nicht auf einer konkreten Objektinstanz operiert. Dies geht nur, wenn man in der statischen Methode eine Referenz auf ein Objekt dieser Klasse hat.
- ☐ Ein Objekt muss existieren, bevor eine statische Methode ausgeführt werden kann.
- ☐ Eine statische Methode sollte immer wie folgt aufgerufen werden:
  `Klassenname.methodenname(<Argumentliste>);`
- ☑ Statische Methoden eignen sich vor allem für Aufgaben, die nicht von einem Zustand abhängen. Sie verarbeiten also primär Informationen, die ihnen als Argumente übergeben werden.
- ☑ In Java gibt es viele Klassen (String, Math etc.), die statische Methoden haben. Die Math Klasse ist hierfür ein gutes Beispiel. Mathematische Funktionen wie Sinus und Cosinus hängen nicht von einem Zustand ab. Sie geben abhängig von einer Eingabe einen Wert zurück.
- ☑ Falls statische Methoden einen Zustand benötigen, können statische Variablen (Klassenvariablen) verwendet werden. Die statischen Methoden einer Klasse können auf die Klassenvariablen dieser Klasse zugreifen.

Aufgabe 2

Forken Sie für diese Aufgabe das Projekt https://github.engineering.zhaw.ch/prog1-kurs/06_Praktikum-2_Befehlssystem. Nutzen Sie **Eclipse** um die eigene Projektkopie auf Ihren Computer zu holen und zu bearbeiten.

Dieses Projekt enthält einen Teil des Codes eines Abenteuerspiels. Der enthaltene Teil ist für die Entgegennahme (Klasse `Parser`), Speicherung (Klasse `Befehl`) und Prüfung (Klasse `Befehlswort`) von Befehlen sowie das Auslösen von passenden Aktionen (Klasse `Kontroller`) zuständig.

Betrachten Sie zuerst die Klasse `Befehlswort` und beantworten Sie die nachfolgenden Fragen:

- Wo und wie wird das Datenfeld *`gueltigeBefehle`* initialisiert und mit Werten abgefüllt?

  Im statischen HashSet<>, dh noch bevor ein Objekt instantiiert wird.

- Zu welchem Zeitpunkt (zur Kompilierzeit, bei Programmstart, jeweils bei der Erzeugung eines Objektes,…?) erfolgt die Befüllung?

  Beim Programmstart

Vervollständigen Sie nun die Klasse `Anwendung` so, dass diese vom Benutzer Befehle von der Standardeingabe entgegennimmt und anschliessend an den Kontroller weiterreicht, bis der Benutzer den Befehl zum Beenden der Anwendung eingibt. Dieses Verhalten soll durch Aufruf der Objektmethode `start()` ausgelöst werden.
Die Anwendungsklasse soll ausführbar gemacht werden (Hinweis: `main`-Methode!). Beim Starten der Anwendung soll das typische Vorgehen für Java Anwendungen befolgt werden, indem zunächst ein Objekt dieser Klasse erstellt wird um anschliessend damit die Anwendung zu starten/zu steuern.

Bevor Sie weitermachen: Testen Sie die Anwendung!

Aufgabe 3

Neben statischen Methoden kennen Sie ja bereits statische Variablen. Diese können z.B. eingesetzt werden, um Text- oder Zahlwerte im Code zu ersetzten (sogenannte Magic Numbers).

Aus Wikipedia: *„The term **magic number** or **magic constant** also refers to the programming practice of using numbers directly in source code. The use of unnamed magic numbers in code obscures the developers' intent in choosing that number, increases opportunities for subtle errors (e.g. is every digit correct in each of the occurrences of PI?) and makes it more difficult for the program to be adapted and extended in the future. Replacing all significant magic numbers with named constants makes programs easier to read, understand and maintain."*

So könnten z.B. bei Verwendung von Richtungswerten "norden", "sueden", "westen" und "osten" im Code folgende Konstanten definiert werden (Sie müssen dies nicht implementieren):

```
private final static String NORDEN = "norden";
private final static String OSTEN  = "osten";
private final static String SUEDEN = "sueden";
private final static String WESTEN = "westen";
```

Anstelle des jeweiligen Textes (z.B. "norden") würde dann jeweils diese Konstante verwendet. Dadurch können sich keine Schreibfehler einschleichen und das Refactoring (z.B. Änderung der Schreibweise) wird einfacher. Für Texte ist dies allerdings immer noch nicht die beste Lösung in Bezug auf Fehlervermeidung. Ein Nachteil liegt darin, dass man diese Variablen nicht benutzen **muss**. Deshalb werden Sie nachfolgend eine bessere Lösung für ein ähnlich gelagertes Problem implementieren.

Studieren Sie nun den Code der Klassen `Kontroller` und `Befehlswort`. Ändern Sie in der Klasse `Befehlswort` das Befehlswort "gehe" auf "laufe". Kompilieren Sie die Anwendung und testen Sie sie.

- Notieren Sie hier Ihre Beobachtungen. Unter welchem Begriff haben Sie das hier zugrundeliegende Problem kennengelernt? Formulieren Sie in Ihren eigenen Worten den Kern des Problems in max. 2 Sätzen.

  Das Spiel weiss nicht, was ich meine, weil es immer noch auf „gehe" hört. Das Problem ist eines der implizierten Kohäsion.

Wir könnten das Problem nun lösen, indem wir für die Textwerte der Befehle Konstanten einführen, und jeweils die passende Konstante verwenden, statt der Textwerte selber. Dies würde allerdings nicht verhindern, dass im Kontroller oder anderorts trotzdem explizite Textwerte verwendet werden könnten. Durch den Ersatz der Befehlsworte in Form von Strings durch Enums kann dieses Problem gelöst werden.

Schreiben Sie nun die Klasse `Befehlswort` so um, dass diese zu einem Enum wird. Der Enum Typ `Befehlswort` soll folgender Spezifikation genügen:

- Enum Werte:
  - UNBEKANNT, GEHE, HILFE und BEENDEN
- Datenfelder
  - `befehl`, mit dem zum jeweiligen Enum Wert gehörenden Befehlswort
- Konstruktoren und Methoden:
  - Privater (!) Konstruktor, welcher das Datenfeld `befehl` setzt. Konstruktoren von Enum Typen müssen privat (oder *package* privat) sein.
    Mehr Infos zu Enums: http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
  - `gibBefehlswort(String wort)`
    Gibt das zum Befehlswort gehörenden Enum Objekt zurück. Für unbekannte Werte von `wort` soll das Enum Objekt UNBEKANNT zurückgegeben werden.

    Hinweis: `Befehlswort.values()` liefert Ihnen die Enum Objekte als Array vom Typ `Befehlswort[]` zurück.
  - `gibBefehlsworteAlsText()`
    Gibt die gültigen Befehlsworte durch Leerzeichen getrennt als String zurück.
  - Getter für das Datenfeld `befehl`

Passen Sie dann die anderen Klassen an, damit das Programm wieder korrekt funktioniert. Achten Sie darauf, dass ausser bei der Definition der gültigen Werte für den Enum Typen `Befehlswort` nirgends die Textwerte der Befehle („gehe“, „hilfe“ etc.) im Code auftauchen! Ansonsten wird Ihre Lösung nicht akzeptiert.

Aufgabe 4

Forken Sie für diese Aufgabe das Projekt https://github.engineering.zhaw.ch/prog1-kurs/06_Praktikum-2_Zuul-besser. Nutzen Sie **Eclipse** um die eigene Projektkopie auf Ihren Computer zu holen und zu bearbeiten.

Fügen Sie die Klassen `Befehlswort` und `Befehl` aus Aufgabe 3 in dieses Projekt ein. Passen Sie das Projekt so an, dass es mit diesen Klassen zusammenarbeitet und die beabsichtigte Funktionalität erfüllt. Ignorieren Sie für diese Aufgabe die Klassen `Person` und `Gegenstand`.

Testen Sie Ihre Anwendung, indem Sie eine geeignete `main`-Methode für das Spiel schreiben.

Aufgabe 5

Analysieren Sie nun die Klassen `Person` und `Gegenstand` sowie deren Verwendung im restlichen Code auf Verbesserungsmöglichkeiten bezüglich der Qualitätskriterien wie lose Kopplung, hohe Kohäsion, Entwurf nach Verantwortlichkeiten und keine Code Duplizierung. Verbessern Sie die beiden Klassen und deren Verwendung durch geeignete Refactorings.

Beachten Sie bei Ihren Verbesserungen das Gesetz von Demeter. Ihre Veränderungen sollten nicht dazu führen, dass dieses verletzt wird.

Das **Gesetz von Demeter** (englisch: *Law of Demeter*, kurz: *LoD*) ist eine Entwurfs-Richtlinie in der objektorientierten Softwareentwicklung. Sie besagt im Wesentlichen, dass Objekte nur mit Objekten in

ihrer unmittelbaren Umgebung kommunizieren sollen. Dadurch soll die Kopplung in einem Softwaresystem verringert und somit die Wartbarkeit erhöht werden.

Die Richtlinie kann umgangssprachlich zu der Aussage „Sprich nur zu deinen nächsten Freunden“ zusammengefasst werden. Formal ausgedrückt soll eine Methode **m** einer Klasse **K** ausschliesslich auf folgende Programm-Elemente zugreifen:

- Methoden von K selbst
- Methoden der Parameter von m
- Methoden der mit K assoziierten Objekte (in Datenfelder gespeicherte Objekte)
- Methoden von Objekten, die m erzeugt

Aufgabe 6 (Optional)

Machen Sie die Anwendung fit für mehrsprachige Befehlseingabe und Textausgabe. Überlegen Sie zuerst, wie Sie dies erreichen können, ohne jeweils die Anwendungsklassen anpassen zu müssen. Implementieren Sie Ihren eigenen Lösungsansatz oder Recherchieren Sie das Konzept der sogenannten Resourcebundles und passen Sie die Anwendung entsprechend an.

Ein Startpunkt für Ihre Recherche zum Thema Resourcebundles:
http://docs.oracle.com/javase/tutorial/i18n/resbundle/propfile.html
Es gibt auch Plugins für Eclipse, welche Helfen, Texte in Ihrem Programm zu “Internationalisieren“:
http://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.wb.doc.user%2Fhtml%2Ffeatures%2Finternationalization.html