Praktikum Programmiertechnik (Technische Informatik)

SS 2015, Hochschule für Angewandte Wissenschaften (HAW), Hamburg
Prof. Dr. Philipp Jenke, Prof. Dr. Axel Schmolitzky, Prof. Dr. Michael Schäfers, Norbert Kasperczyk-Borgmann

Für dieses Aufgabenblatt gelten die folgenden Regeln:

- Alle Programme für dieses Aufgabenblatt müssen in dem Package `aufgabenblatt4` liegen.
- Der Quellcode für dieses Aufgabenblatt muss alle Quellcode-Konventionen dieser Veranstaltung (EMIL: Java Coding Style Guide) einhalten.

Aufgabenblatt 4: Klassen

Aufgabe 4.1: Tamagochi

In dieser Aufgabe entwickeln Sie ein Tamagotchi-Spiel. Sie interagieren mit dem Tamagotchi, indem Sie Kommandos schicken (*iss, schlaf, spiel, nichts*). Der Zustand des Tamagotchis wird durch drei Eigenschaften beschrieben: Hunger (Objektvariable `hunger`), Langeweile (Objektvariable `langeweile`) und Müdigkeit (Objektvariable `muedigkeit`). Die Variablen nehmen minimal den Wert 0 an - dann ist alles in Ordnung. Die Werte können allerdings zunehmen. Ab dem Wert 4 beschwert sich das Tamagotchi. In jeder Runde werden die Werte etwas schlechter.

Schreiben Sie dazu eine Klasse `Tamagotchi`. Jedes `Tamagotchi` muss über die oben genannten Objektvariablen verfügen. In der `main()`-Methode befindet sich der Game-Loop: eine while-Schleife. In der Schleife werden die folgenden Operationen nacheinander durchgeführt:

- Erhöhen der Werte der Eigenschaften
- Ausgeben des aktuellen Zustands des Tamagotchis
- Abfragen des nächsten Kommandos
- Verarbeiten des Kommandos durch das Tamagotchi

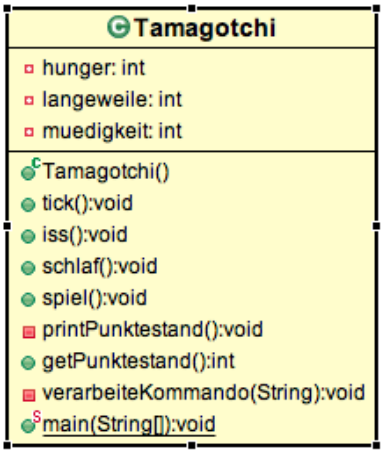

Abbildung 1: Tamagotchi Klassendiagramm

Evolvieren des Tamagotchis
In jeder Runde verändern sich die Werte automatisch. Alle drei Objektvariablen `hunger`, `langeweile` und `muedigkeit` erhöhen sich um 1. Schreiben Sie für dieses Verhalten eine Methode `void tick()`. Das Tamagotchi beschwert sich, wenn mindestens einer der Statuswerte 3 überschreitet. In dem Fall gibt die Methode eine entsprechende Meldung auf der Konsole aus. Alle Objektvariablen sollen als `private` deklariert sein.

Ausgeben des aktuellen Zustands des Tamagotchis
Der Zustand des Tamagotchis wird durch einen Punktestand beschrieben, der zu jedem Zeitpunkt neu berechnet wird: Für jeden Statuswert < 4 gibt es einen Punkt, für jeden Wert >= 4 gibt es zwei Minuspunkte. Schreiben Sie eine Methode `int getPunktestand()`, die den aktuellen Punktestand berechnet und zurückliefert.

Abfragen des nächsten Kommandos
Fragen Sie das Kommando für die aktuelle Runde vom Anwender ab. Die Kommandos werden als Strings repräsentiert. Gültige Kommandos sind: "iss", "schlaf", "spiel", "nichts", "ende". Bei dem Kommando "ende" wird der Game-Loop abgebrochen und damit das Programm beendet. Alle anderen Kommandos werden an das `Tamagotchi` übergeben. Schreiben Sie dazu eine Methode `void verarbeiteKommando(String kommando)`, die als Parameter das Kommando bekommt und dieses verarbeitet. Andere Kommandos soll das `Tamagotchi` ignorieren.
Hinweis: zwei `Strings string1` und `string2` vergleichen Sie mit `string1.equals(string2)`.

Verarbeiten des Kommandos durch das Tamagotchi
Schreiben Sie für jedes gültige Kommando je eine Methode (mit dem gleichen Namen wie das Kommando). Diese Methoden verändern den Zustand des `Tamagotchis`:

"*iss*" → `iss()`: `hunger` wird auf 0 zurückgesetzt, aber `langeweile` erhöht sich um 1
"*schlaf*" → `schlaf()`: `muedigkeit`, `langeweile` werden auf 0 zurückgesetzt aber `hunger` erhöht sich um 1
"*spiel*" → `spiel()`: `langeweile` wir auf 0 zurückgesetzt aber `hunger`, `muedigkeit` erhöhen sich um 1
"*nichts*": keine Aktionen

Aufgabe 4.2: Eisenbahnzüge

Schreiben Sie Klassen, die Eisenbahnzüge repräsentieren. Ein Eisenbahnzug besteht aus einer Lokomotive und einer beliebigen Anzahl Wagen, möglicherweise auch überhaupt keinen.

Lokomotiven und Wagen haben die folgenden Eigenschaften (alle ganzzahlig):
Lokomotive:
- Länge (Meter)
- Typ (irgendeine Zahl)

Wagen:
- Länge (Meter)
- Passagierkapazität (Anzahl Personen)

Definieren Sie die Klassen `Lokomotive` und `Wagen`, jeweils mit den angegebenen Eigenschaften und sinnvollen Methoden. Die oben genannten Eigenschaften sind unveränderlich.

Das interessante Problem ist das Zusammenstellen eines Zuges aus den Einzelteilen. Der erste Wagen hängt direkt an der Lokomotive. Geben Sie der Klasse `Lokomotive` deshalb eine Objektvariable `ersterWagen` vom Typ Wagen, dazu eine Getter- und eine Setter-Methode. An jedem Wagen hängt der jeweils nächste Wagen oder gar nichts beim letzten Wagen. Definieren Sie in der Klasse `Wagen` eine Objektvariable `naechsterWagen` des gleichen Typs Wagen, wieder mit Gettern und Settern. Diese Objektvariable speichert ein anderes Objekt derselben Klasse oder `null` beim letzten Wagen.

Definieren Sie schließlich eine Klasse `Zug`, die den ganzen Zug repräsentiert. Ein `Zug`-Objekt kennt "seine" `Lokomotive` (Objektvariable `lok`), aber nicht die Wagen. Diese können aber, einer nach dem anderen, auf dem Weg über die Lokomotive erreicht werden.

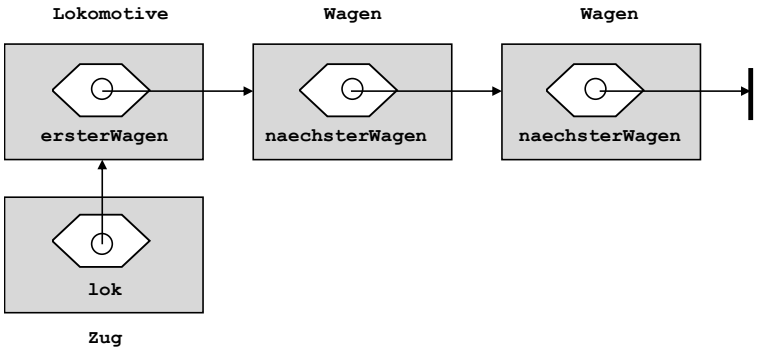

Abbildung 2: Aufbau eines Zuges mit Lokomotive und Wagen.

Die Klasse `Zug` bietet die folgenden Methoden (ergänzen Sie sinnvolle Parameter und Rückgabewerte):
- Konstruktor: Der Zug-Konstruktor erwartet eine Lokomotive und baut einen ziemlich kurzen Zug, der nur aus der Lokomotive, noch ohne Wagen besteht.
- `wagenHinzufuegen`: Hängt für diesen Zug einen gegebenen Wagen an das Ende an.

- `erstenWagenEntfernen`: Hängt den ersten Wagen aus diesem Zug aus und liefert den ausgehängten Wagen als Ergebnis zurück. Die restlichen Wagen rücken nach vorne. Falls es keinen Wagen gibt, ist das Ergebnis `null`.
- `zugAnhaengen`: Akzeptiert als Parameter einen anderen Zug und hängt alle Wagen des anderen Zuges in der gleichen Reihenfolge an diesen Zug an. Im anderen Zug bleibt nur die Lokomotive zurück. Nutzen Sie für diese Methode geschickt die vorher definierten Methoden.
- `getWagenAnzahl`: Liefert die Anzahl der Wagen in diesem Zug (ohne Lokomotive).
- `getKapazitaet`: Liefert die gesamte Passagierkapazität dieses Zuges, das heißt die Summe der Passagierkapazitäten aller Wagen.
- `getLaenge`: Liefert die Gesamtlänge dieses Zuges, d. h. die Summe der Länge der Lokomotive und aller Wagen.
- `info`: Gibt eine Beschreibung dieses Zuges mit allen Bestandteilen (Typ der Lok, Anzahl Wagen, Gesamtlänge, gesamte Passagierkapazität sowie für jeden Wagen Seriennummer, Wagenlänge und Passagierkapazität) auf der Konsole aus.

Schreiben Sie eine Anwendungsklasse, die mehrere Züge zusammenstellt und die Methoden verwendet/testet. Zeichnen Sie außerdem ein Klassendiagramm.