Technische Hochschule Augsburg

WI RPOG2 Übungsblatt 5

Prof. Dr. Strohmeier

07.04.2025

Inhaltsverzeichnis

> **Achtung**
>
> Sie können nur lesend auf die Git Repositories zugreifen. Es gibt keine Tests auf Gitlab!

> **Gitlab**
>
> `git clone https://gitlab.informatik.hs-augsburg.de/prog2_wi/exercises/exercise5`

> **Hinweis**
>
> Sie dürfen unvollständige Klassendiagramme bei Bedarf erweitern!

1 Aufgabe: GameDefinition und Game

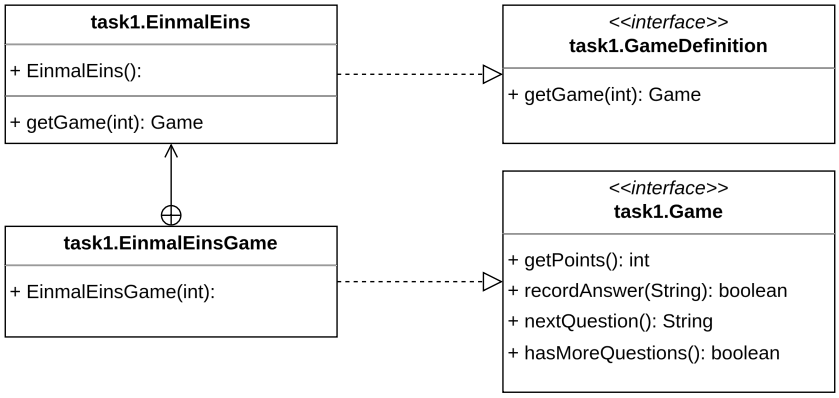

Abb. 1: Unvollständiges Klassendiagramm

Implementieren Sie Ihre Lösung im Paket `de.tha.prog2.blatt5.task1`. Implementieren Sie zunächst die beiden Interfaces.

- `GameDefinition`: definiert eine Methode `getGame(int numQuestions)`, die ein Spielobjekt liefert
- `Game`: definiert ein Spiel, das Fragen stellt, Antworten verarbeitet und Punkte zählt

1.1 `EinmalEins`

Implementieren Sie eine Klasse `EinmalEins`, die das Interface `GameDefinition` implementiert.

Die Methode `getGame(int)` liefert ein Spielobjekt zurück, das genau so viele Fragen stellt, wie als Parameter übergeben wurden.

1.2 `EinmalEinsGame`

Implementieren Sie innerhalb der Klasse `EinmalEins` eine **statische geschachtelte Klasse** `EinmalEinsGame`, die das Interface `Game` implementiert.

- Jede Frage besteht aus einer zufälligen Multiplikationsaufgabe (Zahlen von 1 bis 10)
- Die Methode `nextQuestion()` liefert die Aufgabenstellung im Format:

> Ausgabe für Zufallszahl 3 und 7
>
> ```
> Wie viel ist 3 * 7?
> ```

- Die Methode `recordAnswer(String answer)` vergleicht `answer` mit dem erwarteten Ergebnis
  - Für jede richtige Antwort wird ein Punkt vergeben
- Die Methode `hasMoreQuestions()` gibt an, ob noch Fragen offen sind
- Die Methode `getPoints()` liefert die bisher erreichten Punkte zurück

2 Aufgabe: QuestionGame

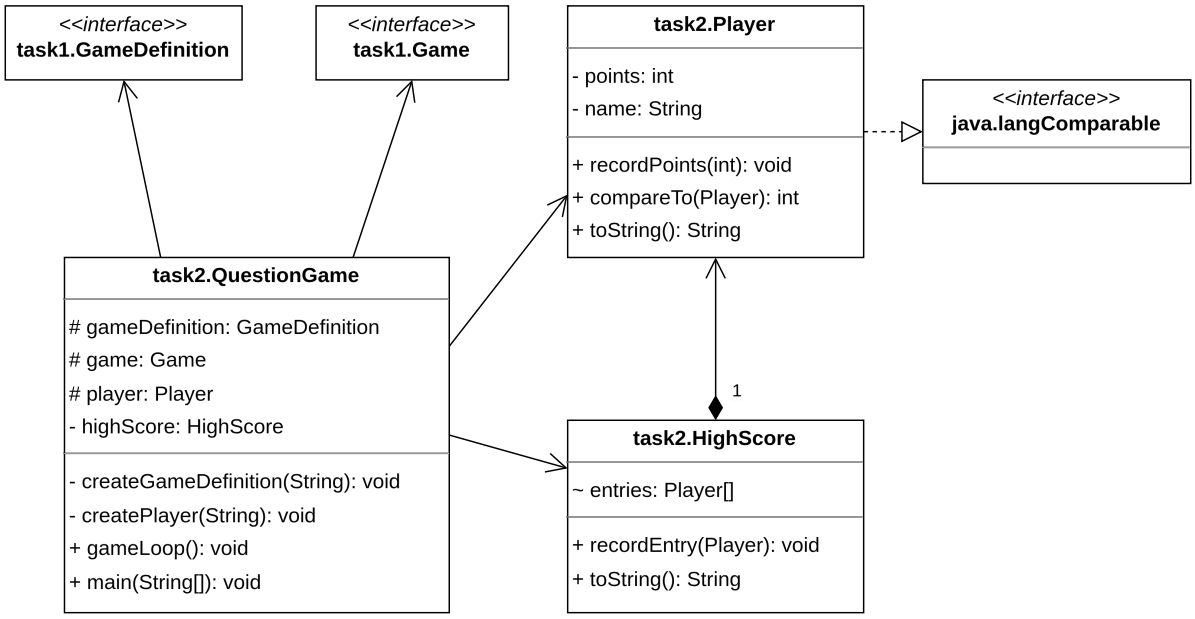

Abb. 2: Unvollständiges Klassendiagramm

Implementieren Sie Ihre Lösung im Paket `de.tha.prog2.blatt5.task2`.

Die Interfaces `Game` und `GameDefinition` wurden bereits implementiert und werden hier wiederverwendet.

2.1 `Player`

Erstellen Sie eine Klasse `Player`, die einen Spielernamen und eine Punktzahl verwaltet.

- Implementieren Sie eine Methode `recordPoints(int points)`, die Punkte zum Spieler addiert
- Überschreiben Sie die Methode `toString()` so, dass die Ausgabe dem Format entspricht

> Ausgabe für `Player("Spieler1")` mit 10 Punkten
>
> ```
> 10 - Spieler1
> ```

- `Player` soll das Interface `Comparable<Player>` implementieren. Zwei Spieler werden anhand ihrer Punktzahl verglichen

2.2 `HighScore`

Erstellen Sie eine Klasse `HighScore`, die ein Ranking der Spieler speichert und verwaltet.

- Die Klasse speichert bis zu 10 `Player`-Objekte in einem Array
- Verwenden Sie ausschließlich **einfache Arrays**, keine JDK-Collections
- Die Methode `recordEntry(Player)` fügt einen Spieler zu HighScore hinzu:
  - Spieler werden beim Einfügen in das Array so einsortiert, dass sie **nach Punkten absteigend** sortiert sind
  - Bereits eingetragene Spieler mit weniger Punkten werden nach hinten verschoben
  - Wird das Array voll, werden weitere Spieler mit schlechterer Punktzahl ignoriert
- Die Methode `toString()` gibt alle gespeicherten Spieler aus, jeweils in einer neuen Zeile

ℹ Ausgabe für 3 Spieler

```
** HIGH SCORE **
 15 - Charlie
 10 - Alice
  5 - Bob
```

2.3 `QuestionGame`

Erstellen Sie eine Klasse `QuestionGame`, die mittels `gameLoop` den Ablauf des Spiels steuert.

1. Beim Start wird der Spielername mittels `Scanner` abgefragt und ein neuer Spieler erstellt.
2. Danach wird gefragt welches Spiel geladen werden soll. Das Spiel wird durch die Methode `createGameDefinition(String)` und mit Hilfe von Reflection dynamisch zur Laufzeit geladen. Es dürfen nur Klassen Klasse geladen werden, die das Interface `GameDefinition` implementieren. Die geladene Klasse erzeugt ein Spiel mit einer bestimmten Anzahl an Fragen.
3. Das Spiel wird solange durchlaufen, wie `hasMoreQuestions()` wahr ist. Es wird mittels `nextQuestion()` die nächste Frage gestellt und die Benutzerantwort über `recordAnswer(String)` ausgewertet.
4. Wenn keine Fragen mehr übrig sind, werden die erreichten Punkte dem `Player` zugewiesen, das Ergebnis im `HighScore` gespeichert und ausgegeben. Der Spieler wird gefragt, ob er erneut spielen möchte. Falls nicht, wird das Programm beendet.

ℹ Ausgabe für eine Spielrunde und 3 Fragen

```
Name des Spielers: Michael
Welches Spiel soll geladen werden: EinmalEins
Wie viel ist 6 * 5? 30
Wie viel ist 1 * 8? 8
Wie viel ist 10 * 8? 90
** HIGH SCORE **
  2 - Michael
Möchten Sie nochmal spielen? [ja/nein]: nein
```

3 Aufgabe: Bugs

In dieser Aufgabe soll eine Annotation entwickelt werden, mit der bekannte Bugs im Code dokumentiert und analysiert werden können. Zusätzlich soll eine Klasse entwickelt werden, die den annotierten Code nach Bugs durchsucht und diese systematisch auflistet.

3.1 Die Annotation `@Bug`

Erstellen Sie eine Annotation `@Bug`, die auf Klassen, Methoden und Konstruktoren angewendet werden kann. Die Annotation enthält:

- ein Feld `description: String` zur Beschreibung des Bugs,
- ein Feld `type: Bug.Type` zur Einstufung der Kritikalität.

Definieren Sie die Enum `Type` als verschachtelten Typ innerhalb der Annotation mit folgende Werten:

- `INVALID`, `ENHANCEMENT`, `MINOR`, `SEVERE`, `CRITICAL`

Die Annotation kann wie folgt verwendet werden:

```
package test;
@Bug(description="A classy bug", type=Bug.Type.INVALID)
public class MyClass {
  @Bug(description = "A heavy bug", type = Bug.Type.SEVERE)
  public void calc() { /* ... */ }
}
```

3.2 Klasse `BugInfo`

Erstellen Sie eine Klasse `BugInfo`, die folgende Informationen über Bugs speichert:

- `clazz`: die betroffene Klasse
- `outer`: umgebende Klasse (falls verschachtelt)
- `method`: betroffene Methode (optional)
- `type`: Bug-Schweregrad
- `description`: Fehlerbeschreibung

Die Klasse enthält:

- einen Konstruktor zur Initialisierung,
- eine `toString()`-Methode, die alle Felder durch Tabs (`'\t'`) getrennt als Textform ausgibt:

> Ausgabe von `toString()` für `test.MyClass`
>
> ```
> test.MyClass    null    null    INVALID A classy bug
> test.MyClass    null    public void test.MyClass.calc() SEVERE  A heavy bug
> ```

3.3 Klasse `BugLister`

Erstellen Sie eine Klasse `BugLister`, die die Annotationen im Code analysiert:

- Die Methode `getBugInfos(Class<?> clazz)` durchsucht die Klasse und deren inneren Klassen nach `@Bug`-Annotationen.
- Die gefundenen Bugs werden als `BugInfo`-Objekte in einer Liste gesammelt und gespeichert.
- Sie dürfen in dieser Aufgabe JDK-Container verwenden.

```
List<BugInfo> bugs = new LinkedList<>();
// ...
bugs.add(new BugInfo( /* ... */ ));
```

- Die Methode `toString()` gibt eine Übersicht aller dokumentierten Bugs zurück.

4 Unit-Test und Testklassen

Im Projekt stehen Unit-Tests und Testklassen zur Verfügung. Die entsprechenden Ordner sind in der Vorlage vom Build ausgeschlossen, um Compilerfehler zu vermeiden. Die entsprechenden Klassen liegen unter:

```
src/test/java/task1
src/test/java/task2
src/test/java/task3
src/main/java/de/tha/prog2/blatt5/task3/annotatedClasses
```

Um diese wieder dem Build hinzuzufügen, gehen Sie wie folgt vor:

1. Rechtsklick auf den entsprechenden Ordner im Projektbaum (links in der Projektübersicht)
2. Wähle **Mark Directory as ⇒ Cancel Exclusion**