SE2-Projekt: Dokumentation

Autoren

Jolanda Jerg (jj033)
Rajan Serafi (rs113)
Freddy Newton Akdogan (fa019)
Georg Habermann (gh014)

Projektname

Raumschiffe versenken (Raumschiffe_versenken)

Pfad zum Repository

https://gitlab.mi.hdm-stuttgart.de/gh014/Raumschiffe_versenken.git

!Achtung!

nach dem Projekt-Import in IntelliJ bitte einstellen:

- File → Settings → Build, Execution, Deployment → Compiler → Java Compiler → Target Bytecode Version **9**
- File → Project Settings → Project → Project SDK **9.0**
- File → Project Settings → Project → Project language level **9**
- File → Project Settings → Modules → Language level **9**

Abstract

Unser Projekt nennt sich Raumschiffe versenken und ist im Prinzip gleich aufgebaut wie das bekannte Spiel Schiffe versenken. Es handelt sich um ein Spiel, in welchem zwei Spieler teilnehmen können. Beim Start werden die Raumschiffe automatisch vom Computer gesetzt. Diese werden für jeden Spieler unten links im Feld angezeigt, sodass die eigenen Raumschiffe sichtbar sind. Unser Feld hat eine Größe von 10 mal 10 Felder, also insgesamt 100 Felder.
Die Player spielen nacheinander, das heißt immer nur eine Person ist an der Reihe und sieht den Bildschirm, der zweite Spieler wartet bis der Zug beendet ist.
Spieler Eins fängt an mit dem Beschuss und wählt ein Feld aus, um möglichst ein Raumschiff zu treffen. Es gibt vier verschiedene Raumschifftypen: fünf Jäger der Länge eins, vier Bomber der Länge zwei, zwei Fregatten der Länge vier und einen Zerstörer der Länge fünf.
Ist ein Schiff versenkt, so verfärbt sich das Feld rot. Zielt der Spieler daneben, so erscheint ein Kreuz auf dem Feld. Bei jedem Treffer darf nochmal geschossen werden, bis kein Raumschiff mehr versenkt wird.
Das Spiel ist beendet, sobald einer der beiden Spieler alle Raumschiffe versenkt hat (26 Treffer erzielt hat).

Startklasse

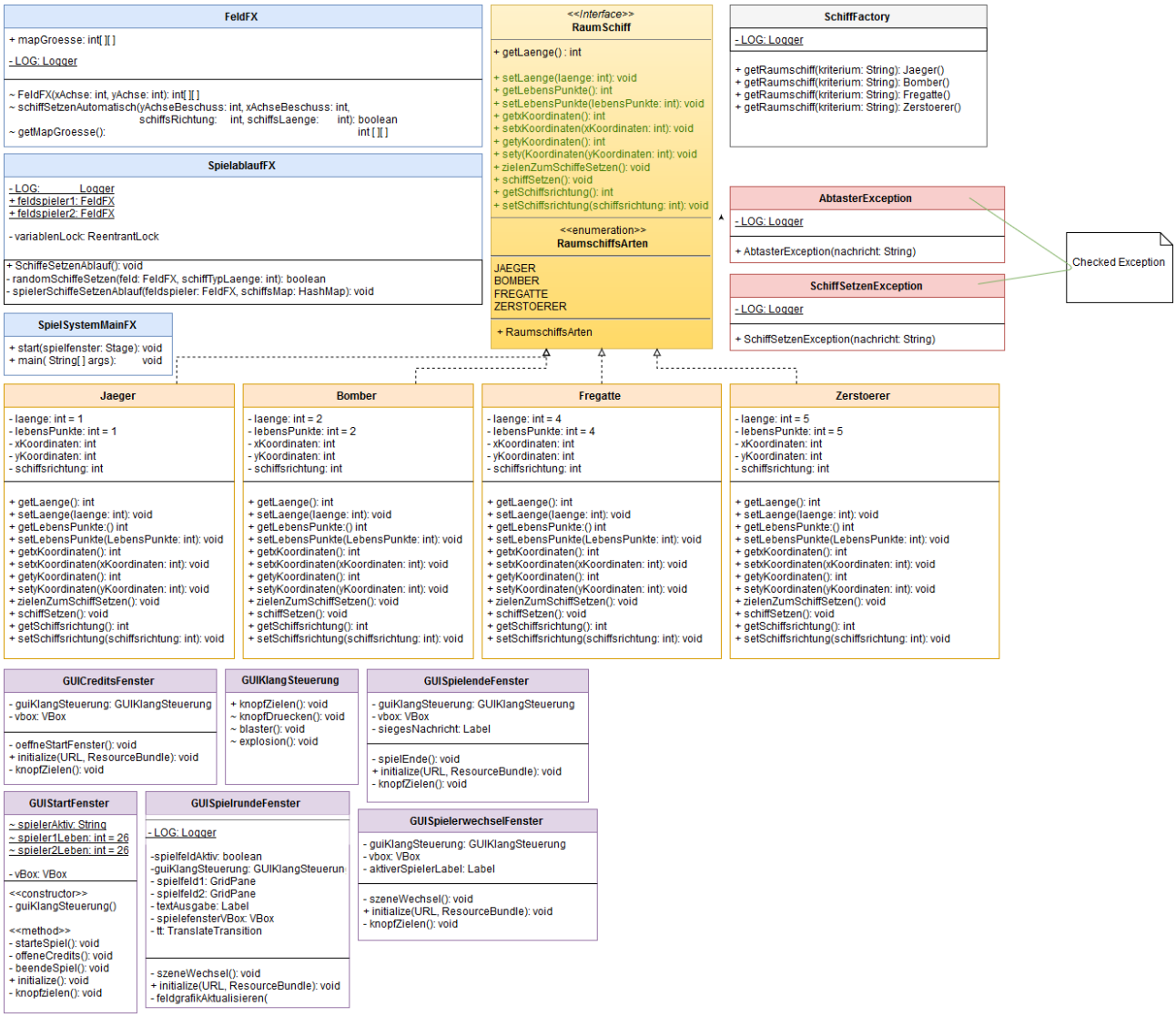
SpielSystemMainFX

Besonderheiten

Für unser Projekt haben wir einen eigenen Soundtrack komponiert. Alle Grafiken und Geräusche wurden ebenfalls selbst erstellt. Die Implementierung folgender Funktionen hatten wir geplant, sie wurden letztendlich jedoch nicht in das Projekt übernommen:

- manuelles Platzieren der Raumschiffe über Drag & Drop
- Platzieren der Raumschiffe mit einem Mindestabstand zueinander
- Benachrichtigung wenn ein Schiff komplett versenkt wurde
- distinkte Raumschiffgrafiken, welche deren Richtung erkennen lassen

UML-Klassendiagramm



Stellungnahmen

Interfaces/Vererbung (Paket Interface_Factory)

In unserem Projekt befindet sich ein Interface im Paket Interface_Factory und die Interface-Klasse heißt RaumSchiff. Wir verwenden die Funktionalität des Interfaces, um die verschiedenen Raumschiffsarten aufzubauen. Alle Schiffe haben gleiche Eigenschaften (gleiche Methoden), nur ihre Implementierung ist unterschiedlich. Deswegen besitzt das Interface leere Methoden, welche in den einzelnen Schiffsklassen angepasst werden.
Zum Beispiel hat jedes Raumschiff eine eigene Länge, die mit der Methode getLänge( ) und setLänge( ) ausgegeben und geändert werden kann. Alle Klassen, die das Interface RaumSchiff implementieren, müssen auch alle Methoden übernehmen und implementieren.
In unserem Projekt ist das Interface sinnvoll, weil wir damit nicht gleiche Methoden bei jedem Raumschiff einzeln erstellen müssen und all unsere Schiffe einige Gemeinsamkeiten haben.

Package-Struktur

All unsere Pakte befinden sich in dem Hauptpaket RaumSchiffeVersenken.
Aufgeteilt sind diese in Core, Exception, GUISteuerung, Interface_Factory und SchiffsArten.
Diese Unterteilung bezieht sich auf die verschiedenen Inhalte des Projekts.

Im Paket Core, also im „Kern“, befinden sich die wichtigsten Klassen, unter anderem die Main-Klasse, die Ablauf- und Funktionsklasse für das Feld. Ohne dem Core-Paket ist unser Projekt nicht ausführbar.
Unsere eigene Exception ist in dem Paket Exception angelegt.
Die verschiedenen GUI-Fenster sind in dem Paket GUISteuerung enthalten. Hier befinden sich die einzelnen Klassen, welche die FXML-Dateien mit dem Programmcode verbinden.
Wie der Name Interface_Factory bereits verrät, befinden sich in diesem Paket unser Interface und die Factory für die Erstellung der einzelnen Raumschiffsarten.
Im letzten Paket ist eine Klasse für jede Schiffsart (Bomber, Fregatte, Jaeger, Zertsoerer).
Für uns erscheint diese Struktur als geordnet und logisch, da jedes Paket einem bestimmten Thema zugeordnet ist.

In dem Paket resources befinden sich alle Dateien, die für unsere GUI notwendig sind. Darunter befinden sich Bilder, FXML-Dateien, Geräusche und Musik.
Alle Bilder sind als PNG-Dateien in dem Ordner Bilder gespeichert.
Die einzelnen grafischen Fenster der GUI wurden als FXML-Dateien erstellt und sind in dem Paket fxml gesammelt.
Außerdem sind Geräusche in unser Projekt implementiert, welche sich als wav-Dateien, in dem Paket Geräusche befinden.
Während dem Spiel läuft eine Hintergrundmusik, die im Paket Musik als mp3 hinterlegt ist.

Unsere Testklasse (FeldTest) befindet sich in einer separaten Klasse (test) und in dem Paket RaumSchiffeVersenken.Core.

Exceptions (Paket Exception)

In dem Paket Exception haben wir unsere eigenen Exceptions erstellt. Die AbtasterException erbt von java.lang.Exception und wird in der Methode schifffSetzenAutomatisch in der Klasse FeldFX geworfen und angewendet, wenn ein Schiff nicht richtig gesetzt wird.

Die zweite heißt SchiffSetzenException und wird in der Methode SchiffSetzenAblauf geworfen, sobald die gesamte Schiffslänge aller Raumschiffe nicht 26 beträgt. Dann kommt die Nachricht, dass das setzen der Schiffe nicht geklappt hat, da nicht jedes Schiff in der HashMap enthalten ist.

Ansonsten verwenden wir die IOException (Input-Output-Exception) bei Methoden, wo Ein- oder Ausgaben zu Fehlern führen können.

Grafische Oberfläche (JavaFX)

Die grafische Oberfläche (GUI) wurde mit dem Scenebuilder erstellt, der mit fxml-Dateien arbeitet. Diese Dateien befinden sich in dem Ordner resources unter dem Paket fxml. Mit dem Scenebuilder ist es relativ einfach grafische Objekte durch passende Layouts richtig anzuordnen.
In unserem Projekt verwenden wir fünf Fenster für das GUI, die jeweils nach ihrer Funktionalität bezeichnet sind.
Zunächst gibt es die Datei startFenster.fxml, die unser Startfenster des Spiels bildet. Angeordnet in einer VBox befinden sich ein Bild und drei Buttons. Die Buttons sind verknüpft mit Events (ActionEvents), sodass wenn ein Button durch Mausklick gedrückt wird, ein neues Fenster geöffnet wird. Diese Verlinkung wird in der GUISteuerung geregelt. Die Klasse GUIStartFenster übernimmt die FXML Elemente (wie zum Beispiel VBox, HBox, GridPane,...) und die ActionEvents, die als Methoden angepasst werden. Jedes neue ActionEvent steht in einer eigenen Methode und sagt dem Event, was passieren soll, wenn der Button gedrückt wird. Dadurch werden die grafischen fxml-Dateien mit dem funktionierenden Code verbunden.

Wird der Start-Button vom User gedrückt, so öffnet sich das spielRundeFenster.fxml. In der VBox sind ein GridPane (unser Spielfeld mit 100 Feldern) und darunter eine HBox angeordnet, welche unterteilt ist in ein weiteres GridPane (kleines Spielfeld, wo die eigenen Schiffe sichtbar sind) und eine neue VBox. Hier sind einige Layouts ineinander verschachtelt, um das gewünschte Layoutkonzept zu erreichen. Die beiden Spielfelder (spielFeld1 und spielFeld2) erhalten jeweils eine ID, genauso wie das Textfeld (textAusgabe) und die äußere VBox (spieleFensterVBox), um so in der GUISpielrunde-Fenster-Klasse aufgerufen werden zu können. In dieser Klasse sind die Methoden enthalten, die den Spielablauf regeln und das ActionEvent szeneWechsel, wenn der User auf den Weiter-Button klickt.

Wird dieser Button gedrückt, so erscheint die fxml-Datei: spielerwechselFenster, wo der nächste Spieler im Textfeld steht und ein neuer Button „Weiter“ erscheint. Diese fxml-Datei ist verknüpft mit der Klasse GUISpielerwechselFenster. Darin verknüpft das ActionEvent szeneWechsel erneut das Fenster spielrundeFenster, allerdings für Spieler 2.

Nun ist Spieler 2 an der Reihe und hat im Prinzip genau das gleiche Spielfeld vor Augen wie Spieler eins, nur mit Spielfeld von Spieler eins. Jedes Mal, wenn ein Spieler fertig ist mit schießen, wird der Button Weiter geklickt und es öffnet sich zuerst das spielerwechseFenster bevor das spielRunde-Fenster wieder geöffnet wird.
In unserem Projekt findet also hauptsächlich ein Wechsel zwischen diesen zwei Szenen statt.

Sobald ein Spieler alle Schiffe versenkt hat, öffnet sich das Fenster spielendeFenster.fxml. Gegliedert in einer VBox befinden sich ein Textfeld (mit der Siegesnachricht) und ein Button zum Beenden des Spiels. Beim Drücken des Buttons tritt das ActionEvent spielEnde( ) ein, welches das Spielfenster schließt.

Das letzte Fenster nennt sich creditsFenster.fxml und zeigt vier Label (Namen der Projektmitglieder) und einen Button (Zurück) an. Dieser Button besitzt das ActionEvent oeffneStartFenster( ) in der Klasse GUICreditsFenster. Wird dieser Button aktiviert, so öffnet sich das Startfenster.

Des Weiteren ist unser Spiel mit Hintergrundmusik und Geräuschen hinterlegt. Für jeden Buttonklick ertönt ein Geräusch aus unserem Resourcen-Ordner (gespeichert als wav-Dateien).
Für die Klangsteuerung haben wir die Klasse GUIKlangSteuerung angelegt, in der alle Geräusche in Methoden implementiert sind. Beim Aufruf eines Geräusches muss deswegen zuerst in jeder Klasse ein Objekt der Klasse GUIKlangSteuerung gebildet werden, um auf die einzelnen Geräusche zugreifen zu können.
Die Titelmusik wird direkt in der Main-Methode (SpielSystemMainFX-Klasse) aufgerufen, da sie das ganze Spiel über durchläuft. Diese ist in der start-Methode implementiert und wird dadurch auch gestartet.

Außerdem gehören zur grafischen Oberfläche alle Bilder, die in dem Resourcen-Ordner unter Bilder als png-Dateien gespeichert sind. Diese werden hauptsächlich in der Klasse GUISpielrundeFenster benötigt, um das Spielfeld je nachdem ob Treffer oder Misstreffer richtig zu markieren. Das Bild für den Hintergrund wird in der CSS-Datei mit den einzelnen fxml-Dateien verknüpft.

Logging

Class	Loglevel
SpielablaufFX	Log.trace bei der Objekt-Erstellung.
Paket Exception in den Class und Catch-Blöcke	Log.error bei der Exceptions behandlung.
FeldFX, SchiffFactory, GUISpielrundeFenster	Log.info für den Spielablauf nötigen Dinge.

Am Beginn einer Klasse wird der Logger erstellt, der mit der log4j2.xml verknüpft ist. Diese xml-Datei befindet sich im Paket resources unter dem Namen log4j2.xml. In dieser Datei ist der Filename A1.log festgelegt, welcher für die Ausgabe der Loggs verantwortlich ist. Außerdem legt die xml-Datei fest, ab welchem Level geloggt wird in der AppenderRef und ab welchem Level die Loggs auf die Konsole geschrieben werden. Bei uns im Projekt darf ab dem Level „debug“ geloggt werden und auf der Konsole erscheinen alle Loggs ab „info“.
Im Code verwenden wir den Logger überall dort, wo stattdessen „System.out.println“ stehen würde. Er dient dafür Infos auf der Konsole auszugeben, wenn etwas im Code passiert oder wenn der User eine Aktion auslöst. Bei jedem Logger übergeben wir eine Nachricht, sodass wir wissen was zu diesem Zeitpunkt im Code passiert.
Bei jeder Exception verwenden wir das Level „error“ für den Logger, da beim Auslösen einer Exception ein Fehler auftritt und dieser behoben werden muss. Bei allgemeinen Aktionen, die der User oder der Computer auslösen verwenden wir meist das Level „info“, sodass wir wissen was passiert ist.

UML (Paket Diagramme)

In unserem UML sind alle Klassen aus unserem Code enthalten. Diese sind nach den Paketen in Farben eingeteilt. Das Paket Core hat die Farbe blau: dazu gehören FeldFX, SpielablaufFX und SpielSystemMainFX. Die Exceptions sind rot markiert. Alle Schiffsarten sind orange gefärbt und beziehen sich auf das Paket SchiffsArten. Ganz unten befinden sich alle GUI-Klassen des Paket GUISteuerung

und diese sind in der Farbe lila. Das Interface RaumSchiff liegt in dem gelben Kasten und die Schiff-Factory ist in dem grauen Feld enthalten.

Der Name der jeweiligen Klasse ist in dem oberen Balken eines Kastens eingetragen.
Das Interface hat dazu noch die Bezeichnung als <<Interface>> über dem Klassennamen.
Unterhalb des Balkens befinden sich die Attribute und Variablen, die in der Klasse enthalten sind. Die Methoden sind darunter festgelegt und sind mit einem durchgezogenen Strich von den Attributen getrennt.
Vor jedem Attribut und jeder Methode steht ein Symbol, das den Modifizierer symbolisiert (public, privat, default, protected). Außerdem sind statische Methoden und Attribute unterstrichen.

In dem Interface befinden sich sämtliche Methoden, die grün hinterlegt sind. Diese sind nur Beispielmethoden, die wir hier allerdings nicht benötigen und deswegen auch nicht implementiert haben.
Der Bezug zwischen den Schiffsklassen, die das Interface implementieren, wird durch den gestrichelten Pfeil dargestellt. Der Pfeil beschreibt die Implementierung des Interfaces in den Klassen Jäger, Bomber, Fregatte und Zerstörer.

Threads (Paket Core → Class SpielablaufFX)

In unserem Projekt verwenden wir zwei Threads, die die gleiche Methode bearbeiten. Diese befinden sich in der Klasse SpielAblaufFX, in der Methode SchiffeSetzenAblauf( ).
Die beiden Threads rufen beide die Methode spielerSchiffeSetzenAblauf( ) auf, beinhalten jedoch unterschiedliche Argumente. In dieser Methode geht es darum die Schiffe automatisch vom Computer auf das jeweilige Feld setzen zu lassen.
Der Sinn von Threads ist es, dass sie gleichzeitig ablaufen. Allerdings sollte beachtet werden, dass verwendete Variablen nicht übernommen oder überschrieben werden, sodass ein Thread möglicherweise mit falschen Daten arbeitet. Aus diesem Grund haben wir ein ReentrantLock eingebaut, der dafür sorgt, dass die Threads synchronisiert und richtig ausgeführt werden. Dieser Lock steht in der Methode spielerSchiffeSetzenAblauf, welche von den beiden Threads verwendet wird.

Streams und Lambda-Funktionen

(Paket Core →Class SpielablaufFX, TestClass FeldTest, Paket GUISteuerung →Class GUISpielrundenFenter)
Die Verwendung eines Streams und einer Lambda-Funktion erfolgt in unserem Projekt über die Klasse SpielAblaufFX, in der Methode SchiffeSetzenAblauf. Der Stream bewirkt die Addition aller Schiffslängen und wird festgelegt in der int-Variablen schiffSetzenControlleur. Dies geschieht über einen parallelStream, der jedes einzelne Element der SchiffsMap durchgeht und die Länge jedes Elements summiert. In dem Stream befindet sich auch die Lambda-Funktion, die mit dem Parameter 'e' jedes Element durchgeht. Wir verwenden einen parallelStream, um alle Schiffsobjekte parallel zu 'streamen', dadurch wird die Performance gesteigert.

Dokumentation und Test-Fälle

Dokumentation:
In unserem Code verwenden wir die Java-Doc vor der Main-Klasse, um die Funktion der Klasse zu beschreiben. Dafür verwenden wir HTML-Tags, sodass auch nur die JavaDoc als Dokument geöffnet werden kann.

Außerdem wird vor jeder Methode die Java-Doc verwendet. Falls die Methode Argumente besitzt, werden diese in der Doc mit @param (=Parameter) aufgelistet und beschrieben, sowie allgemeiner Beschreibungstext in einem <p>-Element (=Text) und Ausgabewerte mit @return Werten.

Test-Fälle
Unsere Testklasse befindet sich in dem Paket RaumSchiffeVersenken.Core und heißt FeldTest.
Insgesamt testen wir fünf verschiedene Elemente aus unserem Code:

1. Test: Es wird die Methode zur Erstellung des Spielfelds getestet, und geprüft ob die Größe des Feldes stimmt.
2. Test: Testet, ob die Methode schiffeSetzenAutomatisch( ) die Schiffe richtig im Spielfeld-Array setzt.
3. Test: Führt einen Test durch, um die Funktionalität der Textausgabe zu prüfen.
4. Test: NullPointerException wird getestet
5. Test: Führt einen Test durch, um die Funktion der HashMap (schiffsMap) zu testen und berechnet dazu die Summe aller Schiffslängen innerhalb der Map

Factories

Unsere Factory ist in dem Paket Interface_Factory in der Klasse SchiffFactory zu finden. Die Factory ist ein Hilfsmittel zum Erzeugen von Objekten der vier verschiedenen Schiffsarten.
Anhand eines Kriteriums (Integer-Werten) wird aus einem Switch-Statement ausgewählt, welches Raumschiff gebildet werden soll. Dadurch ist es möglich, Objekte auf eine einfache und schnelle Art und Weise für die Weiterverwendung zu erzeugen.
Die Methode nennt sich getRaumschiff und gibt ein Objekt eines Schiffes zurück. Durch den Methodenaufruf wird nach gewünschter Schiffsart eine Zahl als Argument übergeben: bei einer eins wird ein Jäger erstellt, bei einer zwei ein Bomber, bei einer vier eine Fregatte und bei einer fünf ein Zerstörer. Das Kriterium ist das Entscheidungsmerkmal für jedes Schiff.