Inhaltsverzeichnis

1 Einleitung

1.1 Team

Kevin-Dominick Berg

Niels Peter Oldenburg

Jan Olschewski

1.2 Lastenheft

1.2.1 Zielbestimmung

Ziel ist die Gestaltung und Umsetzung eines Spiels. Als Vorlage dient dazu der Spieleklassiker „Frogger“ in dem der Spieler einen Frosch über eine belebte Straße und Baumstämme im Fluss bewegen muss. Im Gegensatz zur Vorlage bewegt der Spieler in diesem Spiel einen Fußgänger durch die Fußgängerzone, wobei Hindernisse in Form von Passanten, welche sich horizontal bewegen, auftreten. Zusätzlich gehen die Fußgänger in verschiedenen Geschwindigkeiten über die Spielfläche. Wenn der Spieler einen Abschnitt erfolgreich überquert, wird ein neuer Abschnitt erstellt, in dem sich mehr und schnellere Passanten bewegen. Im Ablauf des Spiels ist nur eine begrenzte Anzahl an Zusammenstößen erlaubt, bevor das Spiel terminiert.

1.2.2 Produkteinsatz

Das Spiel soll auf Java-fähigen Heimcomputern, mit der Tastatur als Eingabegerät, lauffähig sein und Arkade-interessierte Nutzer aller Altersgruppen ansprechen.

1.2.3 Produktfunktionen

/LF010/	Spiel starten
/LF020/	Spiel pausieren
/LF030/	Spiel beenden
/LF040/	Spielernamen eingeben
/LF050/	Spieler bewegen
/LF060/	stehende Hindernisse
/LF070/	sich bewegende Passanten
/LF080/	Kollisionsabfrage
/LF090/	verschiedene Level aufrufen
/LF100/	Spielende-Bildschirm
/LF110/	Punktzahl anzeigen
/LF120/	Höchstpunktzahlliste anzeigen
/LF130/	Hintergrundmusik
/LF140/	Autopause wenn Spielfenster in den Hintergrund gerät

1.2.4 Produktdaten

/LD10/	Spielername

/LD20/ Standort des Spielers
/LD30/ Map und Hindernisse
/LD40/ Passanten und deren Standort
/LD50/ aktuelles Level
/LD60/ Highscoredaten

1.2.5 Produktleistungen

/LL10/ Unzulässige Tastatureingaben werden ignoriert

1.2.6 Qualitätsanforderungen

Das Spiel soll unmittelbar auf Eingaben reagieren. Spielerbewegungen sollen möglichst gleichmäßig und nicht als abrupte Schritte wahrgenommen werden.

1.2.7 Ergänzungen

keine

2 Allgemeines Vorgehen

2.1 Zeitplan

Wir haben uns zum Ziel gesetzt einen Prototypen zum Jahreswechsel fertig zu haben, um über ein ausreichend großes Zeitfenster für Erweiterungen und Refactoring zu verfügen.

Dieses Ziel haben wir erreicht.

2.2 Verwendete Technologien und Libraries

2.2.1 Technologien

- Java 1.8.025
- Netbeans 8.0.1.
- Bitbucket Repository → Smartgit

2.2.2 Externe Libraries

Für die Umsetzung aller geplanten Funktionen des Spiels, war die Einbindung folgender externer Libraries notwendig:

- JDOM (XML Parser)
- JMail

3 Aufbau des Programms

FroggerCloneHD.java beinhaltet die main-Methode, erzeugt die Spielinstanz (Instanz der Klasse Spiel.java) und stößt damit das eigentliche Spiel an.

3.1 Spiel.java

Der Konstruktor der Spiel.java Klasse führt folgende Operationen durch:

- Das Spielfeld wird mit Hilfe des Singleton-Entwurfsmuster initialisiert.
  (Das Singleton-Entwurfsmuster wird in der Spielfeld.java angewendet)
- Die Gamesize wird in Abhängigkeit von der eingestellten Auflösung errechnet. Die Auflösung erhält die Spiel-Klasse dabei über den XML-Reader.
- Die Stringvariable *modus* wird auf MODUS_LOAD gesetzt.
  (Diese Variable gibt das gesamte Spiel über den Status des Spiels an)
- Die Methode *controlModus()* wird aufgerufen, wodurch die eigentliche Hauptschleife beginnt.

3.1.1 controlModus()

Diese Methode ist die wichtigste Methode, sie steuert das Spiel über ein switch-case-*Konstrukt.*

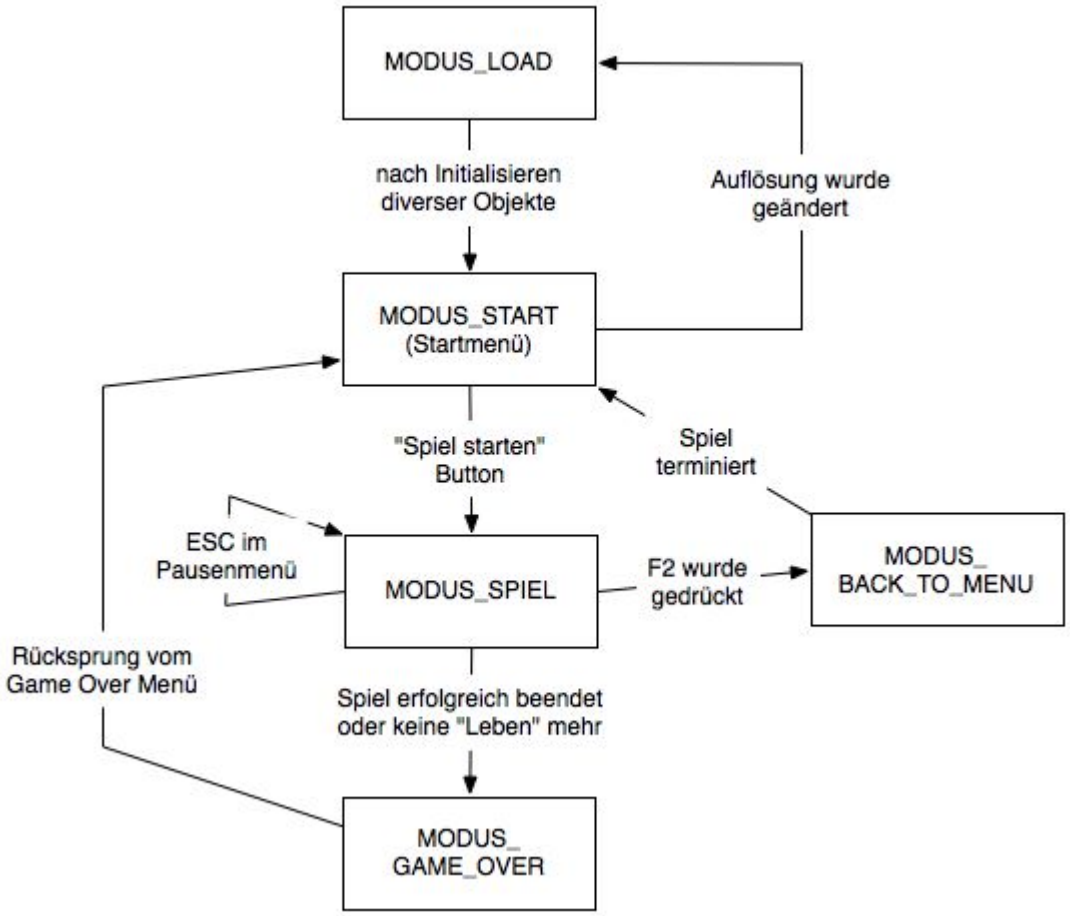

Abbildung 1: Schematischer Ablauf anhand mögl. Spielzustände

3.1.1.1 MODUS_LOAD

Im *case* Modus_Load wird möglichst viel vorweg initialisiert, um ein flüssigeres Spielerlebnis zu ermöglichen. Nach Abschluss des MODUS_LOAD wird der Modus auf MODUS_START gesetzt.

3.1.1.2 MODUS_START

Der MODUS_START beginnt mit der Überprüfung der Bildschirmauflösung. Wenn die Auflösung verändert wurde, wird der Modus zurück auf MODUS_LOAD gesetzt, damit sich die Größe des Spielfeldes u.s.w. neu errechnet.

Außerdem wird in diesem Modus der Konstruktor des Startmenüs aufgerufen, falls noch keine Instanz des Menüs erzeugt wurde. Dieses Menü wird in diesem Modus sichtbar gemacht und der Spieler hat die Möglichkeit nun in andere Untermenüs zu gelangen oder das Spiel zu starten. Sobald der Spieler auf den Button mit der Aufschrift „Spiel Starten“ klickt, wird der Modus auf MODUS_SPIEL gesetzt.

3.1.1.3 MODUS_SPIEL

In diesem Modus findet die komplette Steuerung des laufenden Spiels statt.

Am Anfang dieses Modus wird das Level initialisiert. Diese Initialisierung beinhaltet auch das Instanziieren der Gegner des jeweiligen Levels. Die Leveldaten werden dabei aus dem XML-Reader geladen. Neben der Levelinitialisierung wird auch der Timer für die Punkteberechnung resettet und der Spieler auf seine Startposition bewegt.

Sollte das Spiel bereits beendet worden sein, z. B. durch das Verlieren des letzten Lebens, wird an dieser Stelle die *submitHighscoreMenu* Methode aufgerufen und der Spieler hat die Möglichkeit seinen erreichten Punktestand einer dritten Person per Mail mitzuteilen.

Das eigentliche Spiel (genauer: das aktuelle Level) wird in einer weiteren *while*-Schleife ausgeführt. Hier findet z. B. die Verarbeitung von Bewegungen des Spielers und der Gegner inklusive Kollisionsabfragen statt, es werden die Spieler-Leben überprüft und die Targetfelder überwacht. Weiterhin werden innerhalb dieser *while*-Schleife *repaint()* zur Zeichnung des Spiels und *Ticker.tick()* aufgerufen. Zum Ticker finden sich nähere Erläuterungen im Kapitel 5.1.2.

An dieser Stelle gibt es verschiedene Wege, wie das Programm seinen weiteren Verlauf nimmt:

- Wenn der Spieler keine Leben mehr hat, wird der Modus auf MODUS_GAME_OVER gesetzt, die *while*-Schleife des aktuellen Levels terminiert und der Modus wird auf MODUS_START zurückgesetzt.
- Hat der Player alle Targetfelder eines Levels erreicht, so terminiert die zuletzt gestartete *while*-Schleife und es wird das nächste Level geladen. Dafür wird der Modus Spiel erneut von vorne aufgerufen und erzeugt das Level analog in einer *while*-Schleife.
- Menü das Spiel verbleibt im MODUS_SPIEL.
- Das Drücken von F2 ermöglicht die Rückkehr in den MODUS_START.
- Wenn alle Level durchgespielt wurden, wird der Modus ebenfalls auf MODUS_GAME_OVER gesetzt.

Am Ende eines erfolgreich absolvierten Levels werden dem Spieler in Abhängigkeit der Spielzeit Punkte gutgeschrieben.

3.1.1.4 MODUS_GAME_OVER

In diesem Modus wird der gezeichnete Spielframe verworfen und es wird das Game-Over-Menü angezeigt. Sobald das Game-Over-Menü nicht mehr angezeigt wird, es also z. B. durch den User verlassen wurde, wird der Modus wieder auf MODUS_START gesetzt.

4 User Interface

4.1 HUD

Das HUD zeigt dem Spieler während des Spiels seine aktuelle Statistik (Health und Punkte) an.

Die Punkte werden während des Spiels unten links gezeichnet, während die Healthbar das Spielers unten rechts gezeichnet wird.

Die Healthbar setzt sich aus vier Herzen zusammen. Für jedes vorhandene Leben wird ein rotes Herz gezeichnet und für jedes abgezogene Leben wird ein graues Herz gezeichnet.

4.2 Keyboard-Listener

Über den Keyboard-Listener hat der Spieler die Möglichkeit mit seiner Figur zu interagieren. Außerdem besteht die Möglichkeit das Spiel zu pausieren, zu beenden oder in das Hauptmenü zurück zu kehren.

Es gibt bei den Keyboard-Events jedoch zwei verschiedene Gruppen. Die erste Gruppe ist permanent Verfügbar. Dazu zählt das Event zum Beenden des Spiels, zum pausieren des Spiels und zum zurückkehren in das Hauptmenü.

Die andere Gruppe ist nur dann Verfügbar, wenn das Spiel nicht pausiert ist. Dazu zählt die Steuerung des Spielers.

Insgesamt beinhaltet der Keyboard-Listener folgende Tastenbelegungen:

- W oder Pfeil hoch = Spieler bewegt sich um 1 Feld nach oben
- S oder Pfeil runter = Spieler bewegt sich um 1 Feld nach unten
- A oder Pfeil links = Spieler bewegt sich um 1 Feld nach links
- D oder Pfeil rechts = Spieler bewegt sich um 1 Feld nach rechts
- Escape oder P = Pausiert das Spiel
- F2 = Zurück ins Hauptmenü
- F4 = Spiel beenden

4.3 Menüs

Dieses Menü der Klasse Menu.java unterteilt sich in vier verschiedene Darstellungen des Menüs.

4.3.1 Startmenü

Das Startmenü beinhaltet vier Buttons. Aus der Beschriftung dieser Button ergibt sich auch der Funktionszweck: „Spiel Starten“, „Options“, „Highscore“, „Spiel beenden“.

4.3.2 Game-Over-Menü

- Punktzahl anzeigen
- Einem Freund senden
- zurück zum Hauptmenü

4.3.3 Options-Menü

In diesem Menü besteht die Möglichkeit die Auflösung, also die Darstellungsgröße des Spiels, zu verändern.
Wurde die Größe verändert so wird diese durch einen Klick auf den Button „Übernehmen“ übernommen und per XML-Reader gespeichert.
Außerdem enthält dieses Menü einen Zurück-Button, um in das Hauptmenü zurück zu kehren.

4.3.4 Highscore-Menü

In diesem Menü werden die drei höchsten Punktzahlen die der User erreicht hat per HighscoreXMLReader geladen und angezeigt.
Über den hier vorhandenen Zurück-Button besteht auch hier wieder die Möglichkeit in das Hauptmenü zurück zu kehren.

4.3.5 Pausenmenü

Das Pausenmenü ist keine Instanz einer Klasse. Dieses Menü soll der Vollständigkeit halber hier aber trotzdem erwähnt werden. Das Pausenmenü wird in der *paint()* Methode der Klasse Spiel gezeichnet, sobald das Observable-Objekt (Ticker.java) pausiert wird. Dazu findet ein einfacher Zugriff auf eine Variable des Tickers statt, die aussagt, ob der Ticker gerade pausiert ist oder nicht.

Das Pausenmenü zeichnet über das aktuelle Spielfeld ein Transparentes Quadrat, welches genau so groß ist wie das Spielfeld. Es enthält die Aufschrift „P A U S E“ und zeigt dem User außerdem die Möglichkeiten das Spiel per Escapetaste fortzusetzen, per F4 zu beenden oder mit F2 in das Hauptmenü zurück zu kehren an.

5 Patterns

Im Rahmen unsere Projektes haben wir das Observer-Pattern und das Singleton-Pattern implementiert.

5.1 Observer-Pattern

5.1.1 Observer-Pattern allgemein

Das Observer-Pattern beschreibt eine Situation in der ein Objekt auf Datenänderungen eines anderen Objekts reagieren soll. Das reagierende Objekt nennt sich Observer, während das zu beobachtende Objekt Observable genannt wird. Ein Observable-Objekt kann von mehreren Observern betrachtet werden. Es teilt seinen Beobachtern Änderungen seiner Daten mit, indem die Methode „*notifyObservers()*“ aufgerufen wird. Um bei Datenänderungen informiert zu werden, müssen sich die Observer beim Observableobjekt über die Methode *addObserver()* anmelden.

5.1.2 Observer-Pattern bei uns

Im Rahmen unseres Spiels verwenden wir das Observer-Pattern, um die Spielgeschwindigkeit zu regulieren. Dazu informiert das Observableobjekt alle Objekte der Observerklassen nach Ablauf einer festen Zeit.

5.1.2.1 Observable

Das Observable Objekt ist ein Objekt der Klasse *Ticker*, welches bei Aufruf der Methode tick() die aktuelle Systemzeit berechnet. Außerdem erhöht die Methode *tick()* die Systemzeit um 15 ms, damit das Spiel flüssig läuft. Die Systemzeit wird den Observer-Objekten mitgeteilt. Die *tick()* Methode wird in der Klasse Spiel aufgerufen und läuft permanent in einer *while*-Schleife im MODUS_SPIEL.

Das Observable Objekt erbt die Methode *addObserver()* von der Superklasse (Observable) und ermöglicht damit das anmelden von Observern.

5.1.2.2 Observer

Bei Benachrichtigung durch das Observable-Objekt wird in den Objekten der Observerklassen die jeweilige update Methode ausgeführt.

Player

Die Player-Instanz prüft in der update-Methode, ob der Player sich aktuell auf einem Targetfeld befindet und berechnet seine für eine Kollisionsabfrage benötigte Bounding-Box.

Walker

Die update Methode der Walker-Objekte berechnet analog zum Player die Bounding-Box, berechnet die Koordinatenänderungen für Bewegungen, ändert die Bewegungsrichtung sowie dreht das Bild bei Kollision mit Feldern vom Typ „fix“ und sorgt dafür das die Walker-Objekte beim Verlassen des Spielfelds auf der gegenüberliegenden Seite wieder hinein laufen.

Vehicle

Das Prinzip ähnelt dem des Walkers.

5.2 Singleton

5.2.1 Singleton-Pattern allgemein

Das Singleton-Pattern sorgt im wesentlichen dafür, dass es nur eine Instanz von einer bestimmten Klasse erzeugt werden kann. Der Konstruktor dieser Klasse wird dabei mit dem Modifizierer „private“ versehen. Die Instanz wird in der Klasse selbst erzeugt und mit dem Modifizierer „static“ versehen.

Außerdem enthält die Klasse die Methode *getInstance()*, welche die statische Instanz zurückgibt. Sollte diese noch nicht erzeugt worden sein, so erzeugt diese Methode eine Instanz dieser Klasse. Durch den Modifizierer „static“ wird die Methode auch ohne bestehende Instanz der Klasse verfügbar gemacht.

Gegenüber einer Globalen Variable haben wir durch die Verwendung des Singleton Musters den Vorteil, dass sich alle notwendigen Methoden innerhalb der Spielfeld-Klasse selbst befinden. Die Kapselung des Spielfelds in einer eigenen Klasse ermöglicht die Beschränkung des Zugriff von außen auf von uns definierte Methoden. Dies erhöht die Sicherheit und erleichtert sowohl die

Wartung, als auch Erweiterung.

5.2.2 Singleton-Pattern bei uns

Zur Umsetzung des Spielfelds wurde das Singleton-Pattern verwendet. Dies ermöglicht den Zugriff auf die Attribute des Spielfelds aus jeder anderen Klasse. Dies ist unter Anderem notwendig, um zu überprüfen ob Bewegungen des Players zulässig sind.

Weitere Anwendungsgebiete des Singleton-Patterns in diesem Spiel sind der XML-Reader und der Highscore-XML-Reader. Die aktuelle Bildschirmauflösung wird in der config.xml gespeichert und wird für die variable grafische Darstellung sämtlicher Objekte benötigt. Dieser und noch weitere Verwendungszwecke machen es notwendig den Zugriff auf die Daten innerhalb der config.xml aus jeder Klasse heraus nehmen zu können.

6 Parallele Programmierung

Moderne Computersysteme vermitteln den Nutzern, dass sie mehrere Programme gleichzeitig ausführen können. Tatsächlich ist die Anzahl der ausführbaren Programme auf die Anzahl der Prozessorkerne beschränkt. Der Eindruck von Simultanität wird durch hochfrequente Prozess- bzw. Threadwechsel erzeugt. Das bedeutet, dass sie für kurze Zeit ausgeführt werden und dann wieder pausieren, solange ein anderer Prozess bzw. Thread ausgeführt wird. Diese Quasiparallelität wird auch als **Nebenläufigkeit** bezeichnet.

In diesem Projekt haben wir vier verschiedene nebenläufige Prozesse eingesetzt. Der Erste ist die Hintergrundmusik des Spiels. Sie wird über einen eigenen Thread realisiert, der die Audiofile „bg.wav“ als Loop abspielt. Würden wir die Musik nicht parallel laufen lassen, so könnte man nicht spielen während die Musik läuft. Des Weiteren starten wir einen neuen Audio-Thread bei einer Kollision mit einem Gegner und bei einem Levelübergang.
Eine weitere Nebenläufigkeit starten wir in der Klasse Spiel. In diesem Thread startet das Objekt „Timer“ seine Zeitberechnung. Diese Zeit spielt eine Rolle bei der Berechnung der erreichten Punktzahl des Spielers.

7 XML

Im Rahmen unseres Projektes verwenden wir zwei XML Reader. Der Erste ist dafür verantwortlich Daten aus der config.xml zu beschaffen und der Zweite verwaltet die highscore.xml. Diese sind beide unter Verwendung des Singleton-Patterns erstellt worden.

7.1 Unsere XML

Wir nutzen zwei XML Files und spezifische Reader, um die Informationen aus ihnen in unserem Spiel zu verwenden.

Für die Umsetzung unserer XML-Reader haben wir den SAXBuilder aus der JDOM Library verwendet.

7.1.1.1 config.xml

In der config.xml Datei speichern wir drei Blöcke von Informationen.

Der erste Block beinhaltet generelle Daten wie die Speicherorte von Hintergrundmusik, Kollisionssound und Levelübergangssound sowie die aktuelle gewählte Auflösung.

Der zweite Datenblock beinhaltet die unterschiedlichen Level.

Dabei setzt sich jedes Level aus der eigentlichen Karte und den Fußgängern und Fahrzeugen zusammen die sich in dieser Karte bewegen und mit denen der Spieler nicht zusammenstoßen darf.

Die Karte selbst setzt sich aus 20*20 Feldern zusammen. Dafür werden innerhalb der <map> Tags eines jeden Levels 20 Lines von 0-19 angelegt in denen Strings mit 20 Zeichen als Value angelegt werden. In diesen Strings ist der generelle Aufbau eines Levels codiert. Zusätzlich wird die Startposition des Spielers festgelegt.

Innerhalb der <walkers> Tags werden die „feindlichen“ Fußgänger angelegt. Sie erhalten x und y Koordinaten für ihre Positionierung, ihre Geschwindigkeit und Fortbewegungsrichtung werden festgelegt. Zusätzlich wird ihnen ein Typ zugeordnet.

In den <vehicles> Tags werden anlog zu den Walkern die „feindlichen“ Fahrzeuge angelegt.

7.1.1.2 highscore-XML

In der highscore.xml werden drei Highscores gespeichert.

7.2 XML-Reader

Im Rahmen unseres Projektes verwenden wir zwei XML Reader. Der Erste ist dafür verantwortlich Daten aus der config.xml zu beschaffen und der zweite verwaltet die highscore.xml. Sie sind beide unter Verwendung des Singleton Patterns erstellt worden.

7.2.1 XML-Reader (config.xml)

Der XML Reader für die config.xml beinhaltet Getter-Methoden für die Pfade sämtlicher Texturen und Audiodateien sowie eine Getter- und Setter-Methode für die Auflösung und diverse nachfolgend erläuterte Get-Methoden für die Leveldaten:

getLevels()

Zählt die Level und gibt die Anzahl zurück.

getMap(int levelnumber)

Holt das aktuelle Level aus der XML, indem ein Array vom Typ String erstellt und zurückgegeben wird. Mehrfaches aufrufen der Methode *getLine()* liefert die Elemente des Arrays.

getLine()

Liefert den als String gespeicherten Aufbau einer Zeile des Levels.

CountLevels()

Zählt die Anzahl der Levels und ermöglicht uns damit, das Spiel über die config.xml zu erweitern.

getStartposition()

Liefert die Startposition des Spielers im aktuellen Level.

getWalkers(String lvl)

Erstellt ein Array mit allen in der config.xml für ein Level angelegten Walkern und gibt diesen an das aufrufende Objekt zurück.

getVehicles(String lvl)

Selbes Prinzip wie bei *getWalkers()*.

saveDoc()

Speichert das XML-Dokument, wird für den Setter der Auflösung benötigt.

7.2.2 HighscoreXMLReader (highscore.xml)

getHighscore()

Holt die aktuellen Highscores aus der highscore.xml und liefert sie in Form eines Arrays vom Typ Integer an die aufrufende Klasse.

submitHighscore(int score)

Gleicht den Score eines Spielers mit den bestehenden drei Highscoreeinträgen ab und aktualisiert die Einträge, wenn notwendig, unter Verwendung der *replaceScore(index, score)* Methode.

replaceScore(int index, int score)

Wird von *submitHighscore(int score)* verwendet, um Scores aus dem Highscore-Array in die highscore.xml einzutragen.

SaveDoc()

Speichert das XML-Dokument.

8 Netzwerkprogrammierung

8.1 Netzwerkprogrammierung allgemein

Unter Netzwerkprogrammierung versteht man im weitesten Sinne die Kommunikation verschiedener Systeme miteinander. Dieses Prinzip kann zum Beispiel dazu verwendet werden, um sich mit anderen Spielern zu messen.

8.2 Netzwerkprogrammierung bei uns

Für die Realisierung der Netzwerkprogrammierung in unserem Spiel haben wurde durch die Einbindung der Jmail Library die Grundlagen dafür geschaffen, dass der Mailversand über das Spiel möglich ist. Über die Klasse Mail, welche diverse Eigenschaften aus der Jmail Library importiert, werden die notwendigen Konfigurationsdaten zur Verbindung mit dem Mail-Server bereitgestellt. Nachdem die Konfigurationsdaten geladen wurden, findet der Versand über den Mail-Server statt. Der Konstruktor der Mail-Klasse nimmt drei Parameter entgegen: Der Spielername, die Empfänger-Mailadresse und die erreichte Punktzahl. Mithilfe dieser wird der Text der Email befüllt und an den entsprechenden Adressaten versendet. Als Transportprotokoll für den Mailversand wurde das Simple Mail Transfer Protocol (SMTP) verwendet.

Der Konstruktor der Mail-Klasse wird über das Game-Over-Menü über den Button submitHighscore aufgerufen. In dem Game-Over-Menü selbst besteht die Möglichkeit den Spielernamen und die Empfänger-Mailadresse einzutragen.

Abbildungsverzeichnis