Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Dokumentation Programmierprojekt:

Spielesammlung

Oliver Gawron, Philipp Jung, Leonard Nickels, Frank Rügamer

Eingereicht am: 3. Juli 2017

Betreuer: Vitaliy Schreibmann
Zweitprüfer: Prof. Dr. Steffen Heinzl

Inhaltsverzeichnis

1 Einleitung

1.1 Spiele auf dem Smartphone

... sind heutzutage gang und gäbe. Ob kleine Mini-Spiele oder umfangreichere mit Tiefe, sie alle sind begehrt. Auch Sammlungen kleinerer Spiele erfreuen sich großer Beliebtheit. Doch muss man für diese meist schon viel Arbeit in die Menus und Navigation stecken. Das kann sehr viel Zeit in Anspruch nehmen, Zeit die eigentlich in die Implementierung der Spiele selbst investiert werden sollte.

1.2 Ein Gerüst für Spielesammlungen

Deshalb haben wir ein Gerüst geschaffen welches das Hinzufügen von Spielen erleichtert. Zudem sind in der App auch exemplarisch einige Karten- und Brettspiele implementiert, um die Handhabung zu veranschaulichen.

Sowas wie UnterUnterPunkt

2 Grundlagen

2.1 JDroid Library

Philipp Jung

Die wichtigste Komponente der Kartenspiele ist die JDroid library. Die von Äegidius Plüss entwickelte Java Bibliothek, implementiert ebenfalls die JCardGame Bibliothek für Android, welche grundlegend für unsere Kartenspiele ist. Sie verfügt über eine ausführliche Dokumentation. Im folgenden erkläre ich die wichtigsten von uns genutzten Funktionen der Bibliothek.

2.1.1 Rank und Suit enum

Zu Anfang muss man die Farben und Wertigkeiten der Karten in Form eines Enums festlegen.

Beispiel anhand von Schafkopf:

```
1  public enum Suit
2  {
3      EICHEL, GRUEN, HERZ, SCHELLEN
4  }

6  public enum Rank
7  {
8      ASS, OBER, UNTER, ZEHN, KOENIG, NEUN, ACHT, SIEBEN
9  }
```

Karten werden durch sprites dargestellt, die man in drawables ablegt. Durch die Namensgebung der Sprites, ordnet die Bibliothek die Sprites der richtigen Karte zu. Der Name des Eichel Ass Sprites muss also folgendermaßen lauten:
eichel0.gif

2.1.2 Locations

Um Hände und Kartenstapel anzeigen zu können, muss man zunächst im Konstruktor ein Board erstellen, und Ausrichtung, Farbe, sowie windowZoom(int) angeben. Der windowZoom unterteilt den Bildschirm in Zellen relativ zur Bildschirmgröße, um

darauf sogenannte Locations anzulegen, auf welchen man Karten ablegen kann oder Textfelder.

Am beispiel Schafkopf haben wir 3 Location Arrays genutzt:

- HandLocations (Für alle 2er Kartenstapel)
- StackLocations (Ablage Stapel für gestochene Karten)
- BidLocations (Stapel um einen Stich zu berechnen)

2.1.3 Einstiegspunkt und initPlayers

Anders als normal ist der Einstiegspunkt nicht in OnCreate(), sondern wurde durch die Bibliothek überschrieben und startet in main(). Dort werden das Deck auf Basis der enums, sämtliche Locations, die Hände und die Spieler initialisiert . In initPlayers() läuft das Spielgeschehen ab, es wird dem Spieler der soeben an der Reihe ist der CardListener hinzugefügt und longPressed(Card card) wartet darauf, dass eine Karte gedrückt gehalten wird. Wurde eine Karte ausgewählt wird sie auf den jeweiligen bid transferiert und atTarget() wertet den Stich dann aus und/oder gibt einen marker für den aktiven Spieler mittels der Methode setPlayerMove(int playerWon) weiter, welche jeweils .setTouchEnable() der einzelnen Kartenstapel auf true oder false setzt.

2.1.4 Deck und Hands

Das Deck wird initialisiert aus allen Suits und Ranks, die man Anfangs innerhalb der enums festlegt. Zudem legt man ebenfalls fest wie die Rückseite der Karten aussehen soll. Anschließend wird mit einer vorgegebenen Methode gemischt und ausgeteilt.

dealingOut(int AnzahlHände, int AnzahlKarten, Boolean Mischen)

```
10  deck = new Deck(Suit.values(), Rank.values(), "cover");
11  hands = deck.dealingOut(16, 2, true);
```

Eine Hand beinhaltet Kartenobjekte und verwaltet diese. Sprich Stack, bids und die Spieler Hände bestehen alle aus Hand Objekten, die auf bestimmten Locations angezeigt werden. Mittels .setView() legt man diese Locations für jede Hand einzeln fest und zeigt sie mit .draw() an.

```
12  stacks[i].setView(board, new StackLayout(stackLocations[i]));
13  stacks[i].draw();
```

2.1.5 Textactor

Textactor sind Textfelder welche man auf bestimmten Locations anzeigen lassen kann. Da sie sowohl im Schafkopf als anzeige für die verbleibenden Karten, als auch als Punkteanzeige verwendet werden, ist es wichtig sie an dieser Stelle zu erläutern. Zuerst muss ein Textactor mit einem String initialisiert werden, im unteren Fall mit der Anzahl der Karten des ersten 2er Stapels. Desweiteren braucht jeder Actor eine Location. Mit addActor(TextActor, Location) zeigt man den Actor auf dem Board an und mit removeActor(TextActor) wird er wieder entfernt.

```
14 TextActor t1 = new TextActor(hands[0].getNumberOfCards() + "");
15 Location l1 = new Location(100, 750);
16 addActor(t1, l1.toReal());
17 removeActor(t1);
```

3 Architektur

3.1 Die Spieleverwaltung

Frank Rügamer

Ziel eines derartigen Frameworks ist, dieses möglichst modular zu entwickeln. Dazu gehört auch, es zu ermöglichen, Spiele einfach und dynamisch hinzufügen zu können. Sicher könnte man dessen benötigten Klassen erstellen und anschließend an irgendeiner Stelle in die App hardcoden. Doch wir wollten diesen ganzen Prozess vereinfachen, und damit flüssiger, angenehmer und vor allem modularer gestalten. Dadurch wird auch das Ändern oder Löschen von Spielen vereinfacht. Außerdem wird so der Weg für zukünftige Automatisierung – wie beispielsweise Spielständen – geebnet.

Ist das das Ziel?

3.1.1 Deklaration von Spielen

Die Deklaration von Spielen erfolgt strukturiert in einer XML-Datei, welche durch eine **Document Type Definition** semantisch abgesichert ist. So wird gewährleistet, dass die Spiele im passenden Format vorliegen und so entsprechend weiterverarbeitet werden können. Die Datei erlaubt außerdem die Zuordnung in selbst gewählte Spiele-Kategorien, wie zum Beispiel *Kartenspiele* oder *Brettspiele*.

Bessere Graphik und Caption

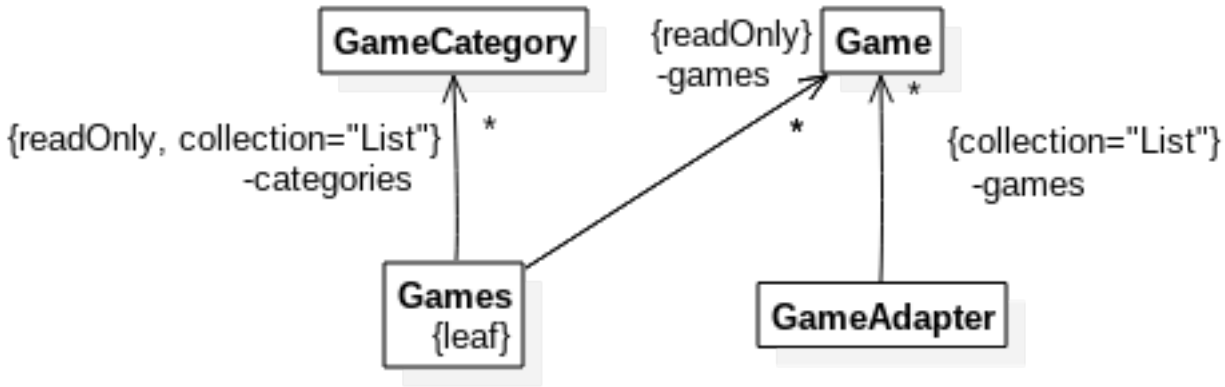

Abbildung 3.1: Test

3.2 Die Spielstände

Leonard Nickels

Da man Spiele nicht immer in einem Zug durchspielt und man zwischendurch Pause macht, ist es sinvoll das Speichern und Laden der Spiele zu ermöglichen. Für diese Funktion sind die Klassen `SavegameStorage` , `Savegame` und `SavegameAdapter` zuständig.

3.2.1 Klassen

`Savegame` ist das Spielstand-Objekt, welches alle Daten speichert, die nötig sind um ein Spiel fortsetzen zu können. Jedes `Savegame` ist dabei einzigartig.

`SavegameStorage` kümmert sich um das Speichern und Laden der `Savegame` Objekte. Diese werden als Liste serialisiert und auf dem Gerät gespeichert.

`SavegameAdapter` erstellt aus den gepeicherten `Savegame` Objekten, für jeden Spielstand eine Karte, welche im Startbildschirm in einer Liste angezeigt wird.

3.2.2 Aufbau

Wenn ein Spiel gespeichert oder geladen werden soll, so geschieht dies durch einen Aufruf der Instanz `SavegameStorage` . Um hier nicht ständig neuen Code schreiben zu müssen, haben wir diese Aufgabe in die Klasse `GameActivity` gelegt. Da jedes Spiel von `GameActivity` erben sollte, müssen dadurch zwei abstrakte Methoden implementiert werden, welche genau hierfür zuständig sind. Die Methode zum Laden heißt `onLoadGame` und die Methode zum Speichern `onSaveGame` . Diese Methoden arbeiten lediglich mit einem **Bundle**, welches die nötigen Informationen beinhaltet. Beim Speichern sollten also alle wichtigen Informationen hinzugefügt werden, damit man dann beim Laden Zugriff auf jene hat.

4 Implementierung

4.1 Chess

Oliver Gawron

Schach ist eins der bekanntesten und gleichzeitig eins der anspruchsvollsten Spiele der Welt. Aufgrund der Komplexität und benötigten Weitsichtigkeit schaffte es erst 1996 der Schachcomputer "Deep Blue"vom IBM den damalig amtierenden Schachweltmeister Garro Kasparow zu besiegen. Heutzutage existieren viele Implementiereungen fähiger Schachprogramme und KI's. In diesem Projekt wurde wurde "Carballo"genommen.

Kommt das hier überhaut hin? Wenn ja, mehr?

4.1.1 Das Spielfeld

Für das klassische karierte Spielfeld wurde die Klasse `CheckeredGameboardView` erstell, welche wie der Name schon sag von der Androidklasse `View` erbt. Haputbestandteil ist ein Zweidimensionales Array aus Androids `Rect` , welche die einzelnen Felder des Spielfelds darstellen. Diese werden nach Aufruf von `onSizeChanged` der Größe des Displays angepasst und je nach Einstellung um die Stärke des gewünschten Randes verschoben, sodass auf jedem Gerät ein identisches Spielerlebnis erzegut werden kann. Um bei einem Touch auf die View zu ermitteln, auf welches der Felder getippt wurde setzt die Methode `getSquareFromTouch(int x, int y)` die in `Rect` mitgelieferte Funktion `contains(int x, int y)` ein und gibt die Array-Koordinaten des gesuchten Kastens zurrück. Bei der Colorierung und Markierung der Felder bezieht sich die Klasse auf die in den Einstellungen gespeicherten Werte.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings