Memory - Dokumentation

A - Team
Technische Hochschule Mittelhessen

26. Juli 2012

Inhaltsverzeichnis

1 Prolog

Die Gruppe A besteht aus Markus Kretsch, Frank Kevin Zey, Florian Thomas und Hagen Lauer.

1.1 Anforderungen

Muss

- Memory-Spielfeld mit sinnvoller Größe (z.B. 8x8 Karten/Felder) für 2 bis 6 Spieler.
- Jeder Spieler bekommt einen Namen, der über Spielsitzungen hinweg gespeichert wird.
- Über jeden Spieler wird eine Statistik angezeigt, wie z.B. Anzahl gewonnener verlorener Spiele, oder Anzahl richtiger Treffer pro 100 Züge.

Kann

- Mehrspielermodus über mehrere Smartphones innerhalb eines LANs.
- Weitere Spielkarten können z.B. von der SD-Karte nachinstalliert werden.

1.2 Lösung / Idee

- Bilder sollen mit Grid und Imageview dargestellt werden. Dabei bieten diese gute Möglichkeiten Klicks zu erkennen und entsprechend zu behandeln.
- Für die Spieler wird eine SQLite Datenbank verwendet.
- Wir haben gute Bibliotheken gefunden um die Daten der Spieler wie gewünscht statistisch auszuwerten und darzustellen.
- Netzwerkspiele werden über WiFi und JavaSockets realisiert, dabei soll es einen Host und mehrere Clients pro Sitzung geben. Das Spielsystem muss also die entsprechende Flexibilität für lokale und Netzwerkspiele mitbringen.
- Spielkarten sollen per .zip File von der SD Karte des Geräts nachladbar sein. Bilder werden in einer Datenbank gespeichert. Das Spiel lädt die Bilder für das Spielfeld aus der Datenbank.

2 Architektur

Wir haben uns selbst als Ziel gesetzt, dass wir in 2 Richtungen entwickeln: Das Spiel Memory als sehr spezifische Implementierung und ein "Framework", das alle typischen Funktionen für ein rundenbasiertes Spiel mitbringt. So konnten wir mit entsprechenden Oberklassen (Game.java) und abgeleiteten Klassen (z.B. Memory.java) garantieren, dass am Ende beide Zweige zusammen führen. Zum Framework gehört Game.java als Oberklasse aller implementierbaren rundenbasierten Spiele, eine Engine die im Wesentlichen Datenbankzugriffe kapselt, ein einfaches Menüsystem, das leicht konfigurier- und erweiterbar ist und einen statistischen Teil der mit der Datenbank zusammen arbeitet. Wir haben dann als Konkrete Implementierung für Memory die Klasse Memory.java von Game erben lassen und entsprechend für Memory angepasst. Ebenso wird die Netzwerkimplementierung eine von Game abgeleitete Klasse sein, natürlich noch mit einigen spezifischen Anpassungen zur Kommunikation der Geräte.

2.1 Design

Wie in Abbildung 1 zu sehen ist entsteht das Framework völlig unabhängig vom zu implementierenden Spiel.
Auf Grund einiger zeitlicher Probleme ist das Projekt dann doch etwas mehr ineinander Verwachsen als gedacht, jedoch hat der Grundgedanke der Unabhängigkeit dabei geholfen unser Projekt zu strukturieren und damit schnell zu implementieren. Gerade was die Datenbanken und das Spielinterface (die Menüführung) Angeht ist das Spiel durch die gezielte Verwendung von ListViews und die verschachtelten Menüs sehr leicht Anpassbar geworden und bietet in seiner grundsätzlichen Ausprägung auch genügend Funktionalität mit um einfache Spiele zu unterstützen. Die Memory-Erweiterungen beginnen im UI hauptsächlich bei den Settings: Abbildung 2

In der Implementierung der Spieler-Typen macht die Frameworkausrichtung und die Benutzung von Oberklassen und Ableitungen etwas anschaulich: Abbildung 3

Player ist hier eine Oberklasse, sie ist Teil des Frameworks. NetworkPlayer ist eine Ableitung, wobei hier noch nicht ganz klar ist ob sie Teil des Frameworks wird. Die Attribute könnten, sofern man

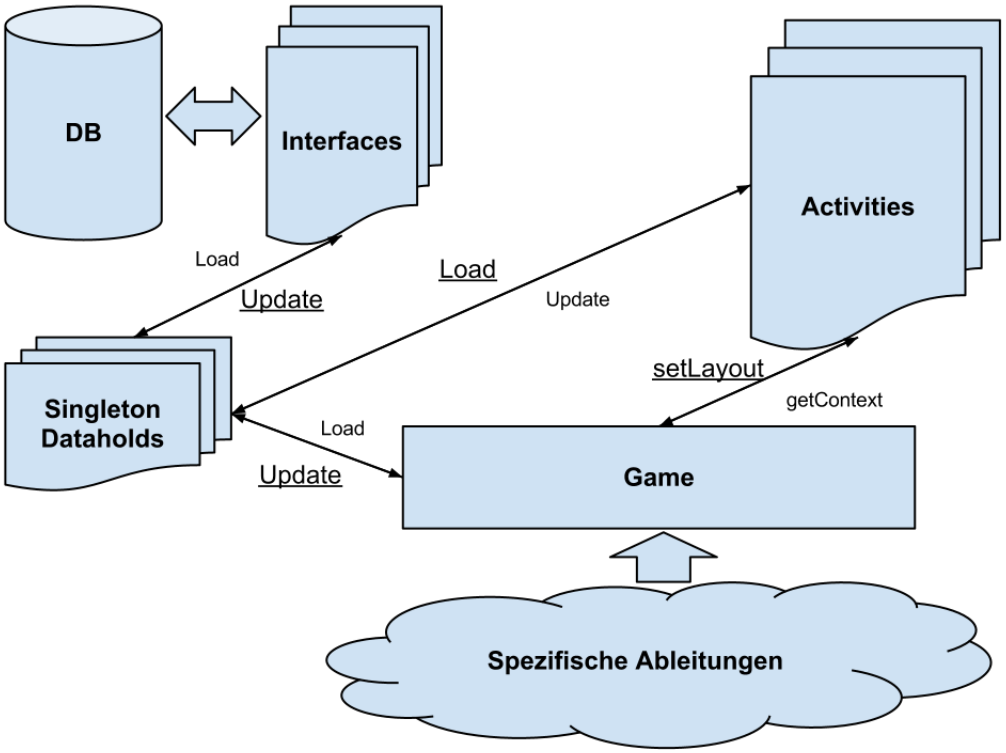

Abbildung 1: Konzept

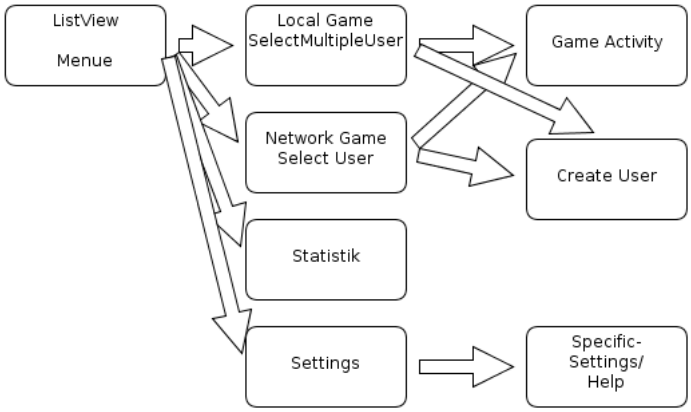

Abbildung 2: Activity Diagramm

denn einheitliche Netzwerkoperationen vorgibt, gut wiederverwendet werden. Die PlayerList ist ein reiner Datenlieferant der in der Regel beim Start der Applikation mit Daten befüllt wird. Das Objekt ist in der App einzigartig und wurde leider in der aktuellen Version etwas missbraucht für eher spezifische Eigenschaften der Memory-Implementierung. Dies wird allerdings im Code klarer und spielt hier keine große Rolle. Wichtig zu merken ist: Player werden in einem PlayerList Objekt gehalten. Dabei unterscheidet man noch in 2 Listen: Players, das sind alle aus der Datenbank gelesenen Spieler, und Session, das sind die zu einer Sitzung (lokales Spiel) gehörenden Spieler. Hintergrund war hier die Auftrennung für den Round Robin Algorithmus den wir beiderseits in Game.java als auch in Player.java realisiert haben. Dabei benötigt Game.java auch nur die Spielerliste der gerade

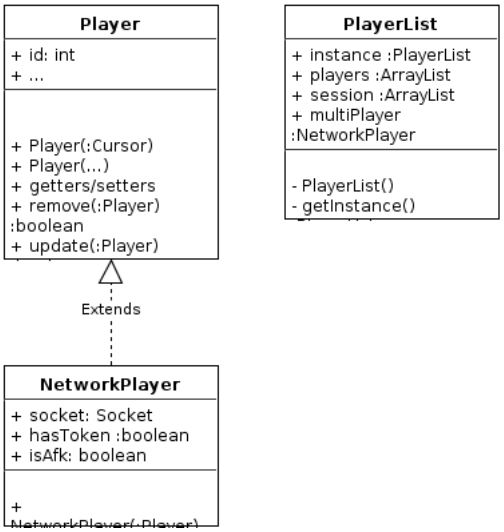

Abbildung 3: UML Player Klassen

teilnehmenden Spieler also nur Session. Die Spieler die in eine Session kommen werden in der SelectMultipleUserActivity ausgewählt.
Das Spiel und dessen Rundenprinzip ist in Game.java bzw. Memory.java realisiert. Essenziell ist dabei natürlich wie mit den Usereingaben umgegangen wird. In unserem Fall muss ein *OnItemClickListener* implementiert werden. Wie, wann und wo dieser definiert und implementiert ist interessiert das Framework nicht. Es etwartet lediglich in der GameActivity, dass man dort an Stelle des traditionellen *setContentView(int)* Aufrufs mit der Methode *assemleLayout()* von Game.java ein View-Objekt erhält. Diese View ist dann Basis der Spielanzeige und ist frei konfigurierbar.

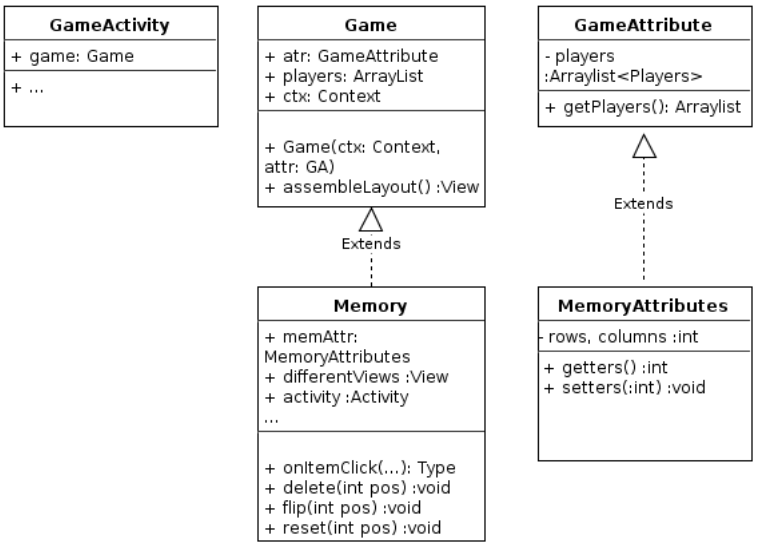

Abbildung 4: UML Game

Memory.java ereitert Game.java und überschreibt dabei z.B. die Methode *assembleLayout()*. Die GameAttributes sind lediglich dazu gedacht mehr Flexibilität in die Architektur zu bringen, da unter

Umständen dort für Initialisierungen etc. wesentlich mehr Attribute gebraucht werden könnten als wir sie bei Memory brauchen. Diese Attribute sollten, wenn man sie denn als Übergabe-Container verwendet in der gesamten Applikation zugreifbar gemacht werden. Im Falle von Memory werden die GameAttributes meistens an den Stellen erzeugt wo man sie auch tatsächlich benötigt.
Netzwerk und Kommunikation sollten in der GameActivity Klasse abgehandelt werden und bei bedarf über Methoden in die Game-Logik hereingereicht werden (z.B. Spielerwechsel mit *turn()*). Die Netzwerk Implementierung ist auf Grund der knappen Zeit sehr spezifisch geworden, hier muss auf jeden Fall noch daran gearbeitet werden Protokolle zu erstellen auf deren Basis eine einheitliche Netzwerk-Implementierung mehrerer Spiele möglich ist. Das Thema Netzwerk wird noch genauer behandelt.

2.2 Engine und Datenbank

Die Engine umfasst die Typisierung der Spieler und Datenbankschnittstellen.

2.2.1 Nachladen eines Kartendecks

Beim Nachladen eines Kartendecks wird eine Zip File mit JPG Dateien erwartet. Diese Zip File muss eine JPG Datei mir Namen *0.jpg* enthalten. Die Zip File wird aus der SD Karte geladen, dazu ist es zwingend notwendig, dass diese sich im root-Verzeichnis befindet. Die Zip File wird von der Applikation geladen und entpackt. Während dem Entpacken werden die Informationen und einzelnen Bilder in einem deck Objekt gespeichert. Dabei kam es zu verschiedenen Problemen, einfachste Reader haben die jeweiligen Daten nicht aus dem Stream lesen können, somit wurde sich auf einen einfach InputStream geeinigt. Das ausgelesene Byte Array wurde anschliesend über eine BitmapFactory zu einem Bitmap Objekt dekodiert.
Wenn das Objekt vollständig erzeugt wurde, speichert es automatisch alle Daten in der Datenbank. Dabei kam es zu mehreren Problemen. Die Tabelle für ein Deck musste um eine weitere Tabelle in einer 1-¿n Beziehung ergänzt werden, nur so können alle Bilder mit Verweis auf das jeweilige Deck gespeichert werden.
Beim Laden der einzelnen Bilddaten kam es wegen den Streams wiedermal zu dem Problem, das ImageReader und ähnliche nicht wirkten, ein einfacher InputStream löst erneut das Problem.

2.2.2 Allgemeines Datenbankdesign

Die Datenbank umfasst 3 Tabellen. Diese Tabellen beinhalten die Informationen jedes einzelnen Spielers. Es wird der Nickname, die Siege, Niederlagen, sowie Unentschieden, getroffene Kartenpaare und die allgemeine durchgeführte Züge. Die jeweilige ID des Spielers entspricht dem jeweiligen Datensatz in der Datenbank.
Neben dem Spieler werden auch zusätzliche Kartendecks gespeichert. Da zu jedem Kartendeck es eine Rückseite vorhanden sein muss und 32 Vorderseiten, handelt es sich um eine 1 -> n Beziehung. In der Datenbank werden zu jedem Kartendeck der Name gespeichert, dieser wird aus dem Dateinamen entnommen. Zusätzlich wird direkt die Rückseite in der Kartendeck Tabelle gespeichert, da die Rückseite nur einmal vorkommt und nicht noch aus der Kartentabelle gefiltert werden muss. Die Kartentabelle enthält die ID des jeweiligen Kartendecks als Referenz und einen Blob in dem die JPG Vorderseitenbilder als Byte Feld gespeichert wird.

Grösstes Problem während der Entwicklung des Datenbank Layouts waren immer wieder kleine Anpassungen durch ständige Weiterentwicklung der Software.

2.3 Statistik

crunch

2.4 Memory

Aufrgund des Aufbaues eines Memoryspieles bietet es sich an ein GridView zu verwenden. Zur Visualisierung der verschiedenen Karten in dem GridView haben wir uns für die ImageViews entschieden, weil sie einfach zu verstehen sind.

Die Bilder werden in der Klasse "Theme" aufbewart, welche diese aus der Datenbank liest, wenn es nicht das Default Theme ist. Dieses wird anhand von drawable Ressourcen geladen. Jedes Theme besteht aus einer Liste der Vorderseiten und einer Rückseite.

Das Memory lädt die ImageViews erstmal mit den Rückseiten in das GridView, weil man ja nicht sehen soll welche Karte sich darunter versteckt. Welches Bild sich wirklich unter dem Bild versteckt wird als id gespeichert. Wenn der User nun das erste Mal auf ein Bild klickt wird ein Zähler erhöht, um zu sehen ob das zweite Bilder gerade geklickt wird. Beim zweiten klick auf ein Bild muss überprüft werden, ob es sich bei dem zweiten Bild

- Um die selbe Position handelt
- Ob sie die selbe id haben

Für den Fall das die die selbe Position haben, wird der Zähler wieder runtergezählt, für den Fall, dass die ids identisch sind, werden die ImageViews disabled und das Bild auf null gesetzt, sodass der Anwender sieht, dass das Bild weg ist. Ansonsten werden die Bilder wieder zurückgesetzt.

Hier entstand das Problem, dass die Bilder sofort verschwanden und der User nicht sehen konnte was den unter der 2. Karte liegt. Aus diesem Grund haben wir Threads erstellt, die eine gewisse Zeit warten und dann erst gegebene Aktionen durchführen, wie zum Beispiel das Reseten oder das Deleten zweier Karten.

Dies führte dann aber zu dem Problem, dass der User klicken konnte während die Karten, die vorher angeklickt wurden, noch zu sehen waren und somit einige Eigenarten hervorhohlen konnten. Desswegen haben wir für diesen Zeitraum die ImageView disabled damit dies nicht passiert.

Unser Memoryspiel ist in der Lage auf viele zeilen/reihen Kombinationen zu reagieren. Aufgrund der Feldgröße 8x8 sollte ein Theme 32 Karten + Rückseite haben haben.

Um nun standartwerte benutzen/setzen zu können benutzen wir die Properties Klasse, welche einiges an Arbeit abnimmt. Die Konfigurationsdatei wird in dem lokalen Speicherbereich der Applikation gesichert. Um diesen herauszufinden benutzen wir die Funktion "getFilesDir()".

Diese wird beim Start ausgelesen, wenn nicht da oder unvollständig werden die Defaultwerte eingetragen, und bei den Settings gesetzt und auch geladen.

Beim laden des Memorys sind wir über den Fehler gestoßen, dass wir out of memory gegangen sind. Dies konnten wir beheben, indem wir in der Activity in der onDestroy methode die geladenen

Bitmaps wieder recyclen.

2.5 Netzwerk

crunch

Benutzerdokumentation

Die Benutzerdokumentation wird per *Help* in der App bereitgestellt.

API Dokumentation

Die Code Dokumentation ist per JavaDoc im Quelltext abgewickelt.