Architektur

Inhalt

[Allgemeine Überlegungen 1](#_Toc400891380)

[libGDX 1](#_Toc400891381)

[Pakete 1](#_Toc400891382)

[UI 1](#_Toc400891383)

[libGDX 1](#_Toc400891384)

[Renderer 1](#_Toc400891385)

[Domain 1](#_Toc400891386)

[GameControllers 1](#_Toc400891387)

[ObjectControllers 1](#_Toc400891388)

[Technical Services 1](#_Toc400891389)

# Allgemeine Überlegungen

Im Zentrum unserer Architektur-Überlegungen stand die Frage, ob und wie wir die grafische Repräsentation unserer Spielobjekte von deren Logik und Zustand trennen. Die Aufteilung des Systems nach dem Model-View-Controller Designpattern ist zwar sinnvoll für viele Business-Applikationen, aber nicht unbedingt geeignet oder intuitiv für Games anzuwenden.

Im Gegensatz zu Business-Applikationen sind die abstrakten Repräsentationen der Daten nicht immer unabhängig von deren Darstellung. Kollisionserkennung und Spielphysik sind beispielsweise direkt von der Darstellung der Spielobjekte abhängig. Ausserdem sind die Spielobjekte genau wie deren Darstellung Entitäten, die sich in einem virtuellen Raum befinden, was bei einer strikten Trennung von Objekten und Darstellung zu viel Redundanz führen kann. Deshalb ist die Anwendung von MVC auf Game-Architekturen nicht immer sinnvoll oder intuitiv anzuwenden.

In unserem Fall hat eine Analyse ergeben, dass wir mit unserem relativ simplen Spielaufbau – wir verzichten z.B. auf eine Physiksimulation und Kollisionsabfragen beschränken sich auf rechteckige Objekte – doch mit einer MVC-Struktur arbeiten können. Das ermöglicht uns ein modulareres System, was uns die Vorteile bringt, dass wir Arbeiten leichter aufteilen können und z.B. die grafische Repräsentation am Anfang simpel halten können und später einfach mit detaillierteren Darstellungen austauschen können. Auch Erweiterungen wie verschiedene Skins wären so theoretisch möglich (sind aber keine Anforderung).

Im MVC-Designpattern ist nicht genau geklärt, ob die Business-Logik Teil des Controllers oder des Models ist. De-Facto werden beide Ansätze in der Softwareentwicklung eingesetzt. Wir haben uns dafür entschieden, die Business- bzw. in unserem Fall Spiellogik als Teil des Controllers zu modellieren. Die Model-Objekte sind simple POJOs und enthalten kaum Logik. Sie sollten ausschliesslich Funktionen zur Datenvalidation enthalten, damit sie stets einen konsistenten Zustand wahren können.

libGDX

Für die Spieleentwicklung auf Android suchten wir ein Einsteigerfreundliches, aber effizientes Spieleentwicklungsframework. Nach der Analyse von vielen Optionen fiel unsere Wahl auf das libGDX (<http://libgdx.badlogicgames.com/>) Framework. Die Gründe für diese Entscheidung sind in erster Linie folgende:

* Open Source
  + libGDX steht unter der Apache 2.0 Lizenz
* Viele, modulare Features
  + Bietet alles, was die Entwicklung eines Spies unseres Scopes erfordern (und mehr)
  + APIs zu Bibliotheken wie z.B. Box2D für 2D-Physikberechnung sind optional
* In aktiver Entwicklung
  + Version 1.4.1 ist erst kürzlich (10.10.14) erschienen
* Grosse, aktive Community
  + libGDX hat sein eigenes Forum, einen IRC Channel.
  + Auch z.B. auf StackOverflow und Reddit finden sich viele libGDX-Entwickler
* Sehr gut dokumentiert
  + Vollständige Javadoc
  + Diverse Tutorials und Erläuterungen
  + Beispielcode und Demos
* Cross Platform
  + Ermöglicht neben Android auch die Entwicklung für Windows, Mac, Linux, iOS, BlackBerry und HTML5

# Pakete

Im Folgenden werden die Pakete (vgl. Paketdiagramm) mit ihren Aufgaben und Beziehungen beschrieben.

UI

Das UI-Paket enthält Pakete, die für die Benutzereingaben und die grafische Darstellung verantwortlich sind.

libGDX

libGDX ist ein Java-Spieleentwicklungsframework. Es bietet Funktionen zum Abfangen und Behandeln von Benutzereingaben, worunter sowohl Touch-Inputs, Touch-Gesten und Sensorinputs wie die des Gyroskops und des Beschleunigungssensors eines Android-Smartphones gehören. Es bietet auch extensive High- und Low-Level APIs zu OpenGL ES 2.0 und Möglichkeiten zur Audio-Wiedergabe. Die Aufgabe des libGDX-Frameworks in unserem Projekt ist also das Handling des gesamten Inputs und Outputs.

Die Grafik-API des libGDX-Pakets wird von den Klassen im Renderer-Paket importiert.

Die User-Input- und Sensorik-API des libGDX-Pakets wird von den Controller-Klassen des Domain.ObjectController-Pakets genutzt.

Renderer

Das Renderer-Paket enthält Objekte (Klassen, evtl. weitere Pakete), die für die grafische Darstellung von Objekten der Spielwelt zuständig sind. So wird beispielsweise die Breite des Frachtschiffs im Model-Objekt (Siehe Domain.Model-Paket) Ship gespeichert und die ShipRenderer ist dann dafür verantwortlich, die korrekten Bilddateien zu laden und daraus anhand der Model-Daten das Schiff korrekt zusammenzustellen (wir gehen von einem modularen Aufbau der Daten aus. Der Vorteil dieser Trennung ist, dass die grafische Repräsentation von Objekten getrennt ist und so z.B. erst zur Laufzeit bestimmt werden kann, wie ein Objekt gerendert wird (eine Art Dependency Injection).

Die Klassen des Renderer-Pakets importieren die Grafik-APIs von libGDX.

Die Klassen des Renderer-Pakets werden von den Controller-Klassen des Domain.ObjectController-Pakets genutzt.

Domain

Das Domain-Paket enthält Objekte und Pakete, welche direkt Bestandteile der Spiellogik sind. Darunter fallen alle Controller-Klassen und die Model-Klassen.

MainController

Der MainController ist die Einstiegsklasse mit der main-Methode. Seine Aufgabe ist es nicht nur, die grundlegende Konfiguration (wie etwa die Framerate) der Applikation zu laden, der MainController enthält auch den Main-Loop.

GameControllers

ObjectControllers

Technical Services