Architektur

Inhalt

[Allgemeine Überlegungen 1](#_Toc401217124)

[libGDX 1](#_Toc401217125)

[Pakete 2](#_Toc401217126)

[UI 2](#_Toc401217127)

[UI::Renderer 2](#_Toc401217128)

[UI::Menus 2](#_Toc401217129)

[Domain 3](#_Toc401217130)

[Domain::Game 3](#_Toc401217131)

[Domain::GameObjects 3](#_Toc401217132)

[Domain::User 3](#_Toc401217133)

[Domain::Level 3](#_Toc401217134)

[TechnicalServices 3](#_Toc401217135)

[TechnicalServices::Persistence 3](#_Toc401217136)

[TechnicalServices::Android 3](#_Toc401217137)

[TechnicalServices::libGDX 4](#_Toc401217138)

# Allgemeine Überlegungen

Im Zentrum unserer Architektur-Überlegungen stand zuerst die Frage, ob und wie wir die grafische Repräsentation unserer Spielobjekte von deren Logik und Zustand trennen. Das führte uns zum klassischen Model-View-Controller-Pattern. Nach Absprache mit unserem Dozenten kamen wir aber zum Schluss, dass dies nicht dem angestrebten Domain-Driven Design entsprach, sondern zu einer unnötig künstlichen Trennung der Domänenobjekte führte.

Die Architektur richtet sich nun stark nach dem Domänenmodell und führt Spielobjekte ein, die sowohl einen inneren Zustand haben als auch Logik bereitstellen. Bewusst getrennt werden allerdings die Renderer, mit dem Ziel genug Abstraktion zu erreichen, damit die visuelle Darstellung der Spielobjekte leicht ausgetauscht werden kann. Ausserdem werden die Renderer selber ein gewisses Mass an Logik beinhalten müssen (z.B. Handling von Animationen, Zusammensetzen von Tilesets), die aber nicht direkt mit der Domäne zusammenhängt.

## 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.

## Paketdiagramm



## UI

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

### UI::Renderer

Das Renderer-Paket enthält Objekte (Klassen, evtl. weitere Pakete), die für die grafische Darstellung von Objekten der Spielwelt zuständig sind. So muss beispielsweise der ShipRenderer die Breite des Ship-Objekts auslesen und ist dann dafür verantwortlich, die korrekten Bilddateien zu laden und daraus das Schiff korrekt zusammenzustellen. Der technische 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). Fachlich macht diese Trennung ebenfalls Sinn, da die Darstellungslogik nichts mit der Domänenlogik zu tun hat. Die Klassen des UI::Renderer-Pakets nutzen die Grafik-APIs von libGDX und greifen auf Objekte des Pakets Domain::GameObjects zu.

### UI::Menus

Das Menus-Paket enthält die Logik für die Menüs. Darunter fällt zum Beispiel das Hauptmenü oder das Einstellungs-Menü. Die Menüs haben die Aufgabe, auf Benutzerinputs (z.B. Touch-Input auf ein Button-Objekt) mit den korrekten Aktion (z.B. Schnelles Spiel starten) zu reagieren. Die Klassen des UI::Menus-Pakets nutzen die Klasse(n) aus dem UI::Renderer Paket für die visuelle Darstellung und starten Objekte aus dem Domain::Game Paket.

## Domain

Das UI-Paket enthält die Domänenlogik des Projekts. Sein Aufbau richtet sich stark nach dem Domänenmodell.

### Domain::Game

Die Objekte im Game-Paket sind für die übergreifende Steuerung der Spiellogik zuständig. Jeder Spielmodus wird von einer eigenen Klasse (oder von einem eigenen Paket, falls mehrere Klassen pro Spielmodus anfallen) definiert. Gemeinsame Operationen werden durch Vererbung generalisiert. Die Objekte im Domain::Game-Paket laden Einstellungen aus den Config- und Stats Klassen aus dem Domain::User-Pakets, im Falle des CareerGames werden zusätzlich auch Level geladen. Die Objekte des Domain::Game-Pakets enthalten Klassen aus dem Domain::GameObjects-Paket und nutzen das Input-Handling von libGDX.

### Domain::GameObjects

Die Objekte im GameObjects-Paket repräsentieren die Spielobjekte wie sie im Domänenmodell definiert wurden. Ein GameObject, wie etwa eine Instanz der Ship-Klasse, enthält sowohl Zustandsinformationen als auch Logik. Sie werden im Spiel von den Renderern dargestellt. Für die Benutzerinteraktion wird das Inputhandling von libGDX genutzt.

### Domain::User

Das User-Paket enthält Klassen und Funktionalität für die Benutzerspezifischen Einstellungen (etwa das Handicap und spielübergreifende Einstellungen wie Ton an/aus) und Statistiken (wie der Highscore, der Spielfortschritt im Karrieremodus, etc.). Die Daten werden mithilfe des TechnicalServices::Persistence-Pakets gelesen und anschliessend von den Objekten im Domain::Game-Paket verwendet.

### Domain::Level

Im Level-Paket befinden sich eine oder mehrere Klassen für das Level-Handling. Ein Level wird über das TechnicalServices::Persistence-Paket geladen und enthält vordefinierte Parameter für ein Spiel, inklusive einer Liste von Containern mit der Reihenfolge wie sie im Spiel erscheinen werden. Die Levels werden ausschliesslich vom CareerGame genutzt.

## TechnicalServices

Das TechnicalServices Paket enthält unterstützende, von der Domänenlogik losgelöste und externe Pakete an, die den technischen „Unterboden“ des Projekts bilden.

### TechnicalServices::Persistence

Das Persistence-Paket soll als wiederverwendbare Zwischenschicht zwischen den zu persistierenden Objekten (Domain::User::Config, Domain::User::Stats und Domain::Level) und den Android-Bibliotheken dienen. Grund dafür ist, dass Android mehrere Möglichkeiten zur Persistenz anbietet (z.B. Shared Preferences, Internal/External Storage, etc.) und wir noch nicht sicher sind, welche für uns am geeignetsten ist. Ausserdem könnte sich das möglicherweise in Zukunft mit steigenden Speicheranforderungen auch ändern.

### TechnicalServices::Android

Die Android-Bibliotheken stellen die API zum Android-Betriebssystem dar und sind essentiell für eine Android-Applikation. In unserem Fall werden wir insbesondere die Storage-Funktionalitäten nutzen, welche von unserem TechnicalServices::Persistence-Paket abstrahiert wird.

### TechnicalServices::libGDX

LibGDX ist ein externes Framework (siehe libGDX unter Allgemeine Überlegungen), welches hauptsächlich für das Inputhandling als auch für die Grafikschnittstelle verwendet wird. Es findet in vielen Bereichen unserer Applikation Anwendung, insbesondere im UI::Renderer-Paket, dem Domain::Game-Paket und dem Domain::GameObjects-Paket