System design document for Ridiculous Rumble Royale

Version: 1.0

Date 2015-05-31

Authors
Tobias Alldén
Oscar Boking
Oskar Jedvert
David Ström

This version overrides all previous versions.

1. Introduction

Ridiculous Rumble Royale is an attempt at making a fast-paced party game that is easy to pick up and play instantly. Combining the intense combat of classic games such as Super Smash Bro's and the accessibility of multiplayer games. The game will be supported on windows 8 machines running the exact same specs as the developers. However most machines that can run java would probably be able to run the game. The advantage to Ridiculus Rumble Royale compared to other games is the unique gameplay. The target demographic for the game are people over the age of seven, It is not meant to be a gory mess but rather a game which can just as well be enjoyed by a family or a group of friends over a remote network.

1.1 Design Goals

The goals of the application is as follows: the client staying in sync with the server, meaning the clients final state will not deviate from the servers final state, having the game follow the MVC pattern, handling dependency code in the model in a service instead.

1.2 Definitions, acronyms and abbreviations

UI - User Interface
The layer exposed to the user which lets him or her change the system.

GUI - Graphical User Interface
A user interface that uses graphics.

Packet
Some data sent over the network.

2. System Design

In this section we explain the overall design choices that have been made when designing the software

2.1 Overview

The application follows the MVC design pattern and uses the external libraries LibGDX with box2D and Kryonet. Libgdx manages the drawing of the program and the window managing, box2D manages all the physics of the game and Kryonet manages the network. The LibGDX parts are used as a service, meaning that it is external to the core. This reduces coupling between the main game model and the external library, meaning that it will be quite easy to switch to another game library instead of LibGDX.
The network type is a client-server type with the server running on one of the clients machines. Input from the client is sent to the server, which updates its model and then sends back its state to all of the clients, the clients then uses the state to render the game.

2.2 Software decomposition

The application is divided into four modules, the game logic module, the constants, the core and the main module

2.2.1 General

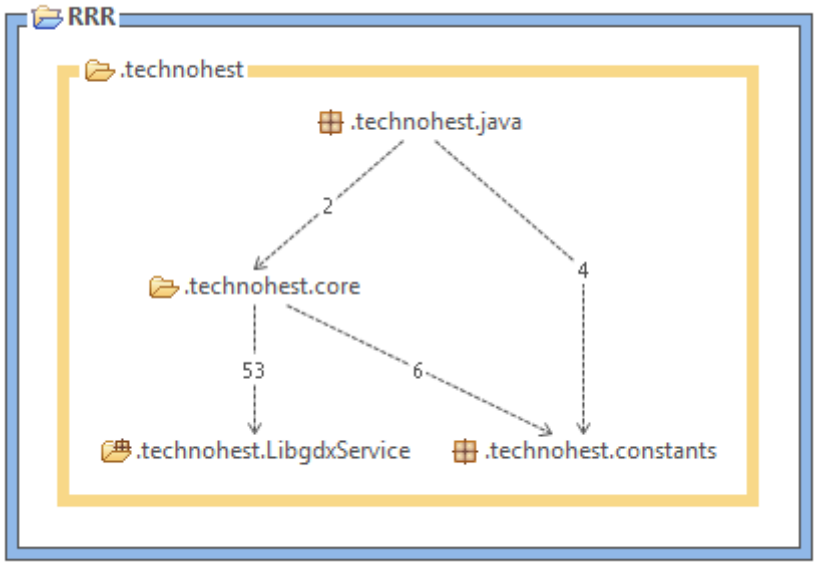


Fig 1. Package Diagram

The technohest.java contains the MainDesktop class, which initializes the application and creates the window with a height and with specified by the settings file.

The technohest.constant package contains constants used in the game, as well as the controls, and settings. These are used throughout the application to specify things as player height and width.

The technohest.core package contains the main MVC structure, thus containing all the models used in the game, as well as the view and the controller. The package also contains handlers such as the input handler and all the classes concerning the network parts of the application.

The technohest.LibgdxService package contains all the LibGDX parts used in the game, this module can be switched to another game logic module if needed, this is the game library service.

2.2.2 Decomposition into subsystems

This application can be decomposed into two subsystems, one of which manages the LibGDX parts of the game, and one part manages the main MVC structure and the network, however both subsystems need one another in one way or another. Without the MVC structure the game logic in quite useless (unless it's replaced by a similar module) and without the game logic the MVC structure only manipulates data, and no actual drawing or movement of players are made.

2.2.3 Layering

The layers in this application are the different modules, all of which could be reused in one way or another. For example, the game logic could be replaced from a LibGDX library to another similar game library, the game logic could be reused in another application with a similar MVC-core structure, the constants could be reused and lastly the main could be reused, since it only creates a LibGDX application it could be replace by any other application using LibGDX.

2.2.4 Dependency analysis

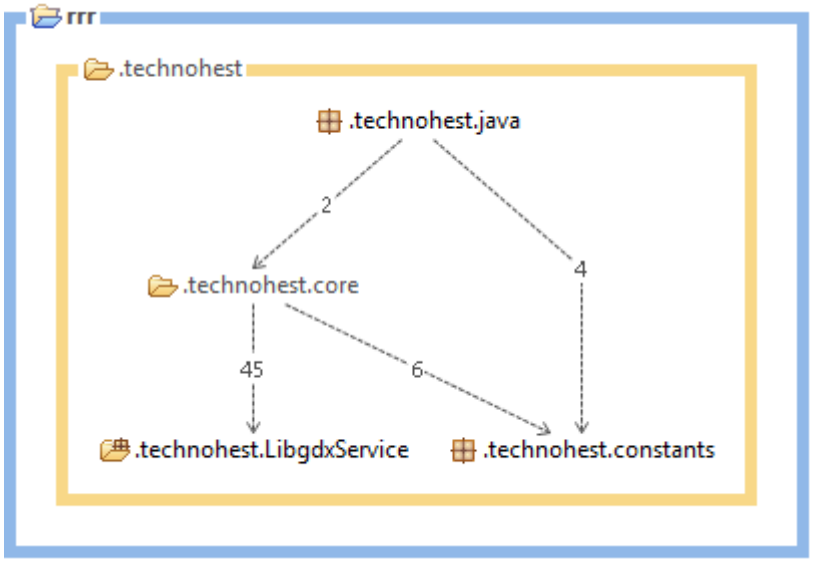


Fig 1 Package Diagram

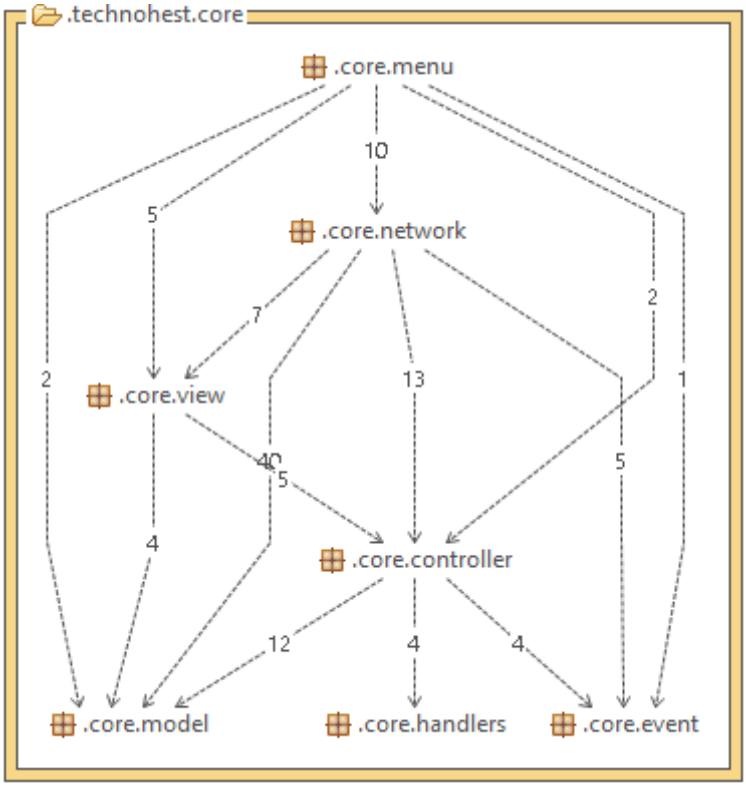


Fig 2 Core Diagram

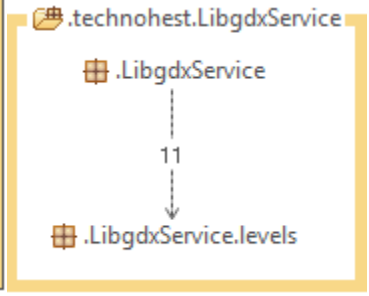


Fig 3 Game Logic Diagram

As seen in the Fig 1-3 the project packages forms a tree-like structure. The reason for this is that each of the modules can be reused separately as long as there is no circular dependence.

2.3 Concurrency issues

The program uses four threads: LWJGL (full name LWJGL Application), Client, Server, and Thread-2. They interact as follows: LWJGL with Client, Client with Server, and Server with Thread-2. Handling concurrency between Server and Client is done by the KryoNet library, while there are in-house solutions to LWJGL communicating with Client and Server communicating with Thread-2.

LWJGL interacts with Client by adding actions to a list which the Client will try to clear of old actions. No issues occur because the list is not iterated through but rather the LWJGL will add actions at the end of the list and the Client will clear actions at the beginning of the list, and the LWJGL and Client will never be dealing with the same element in the list since the action first has to be added and sent over the network before it can be removed by the Client. However, when the LWJGL thread tries to send the actions to the server, the Client thread may be removing actions from the list at the same time. This may result in some actions being cleared before fully serialized and will result in an crash. To prevent this the method sendActionsToServerIfNecessary and the method clearOldActions in the ClientNetworkListener class are synchronized so there is not concurrent editing of the list as it is being serialized and sent.

The way Thread-2 and the Server interacts is similar. The Server thread wants to add actions to a list and Thread-2 wants to perform those actions and clear the list of actions. The adding of actions is done in the method “addActionsToBePerformed” and the actions are performed in the method “performActions”. These are synchronized so that adding and clearing the list is managed safely and also contain as little code as possible so the threads can work as independently as possible.

2.4 Persistent data management

NA

2.5 Access control and security

NA

2.6 Boundary conditions

Since the only input the application takes is controls there is no need for boundary conditions, if a user presses a key which is not used in the game, nothing happens.

3 References

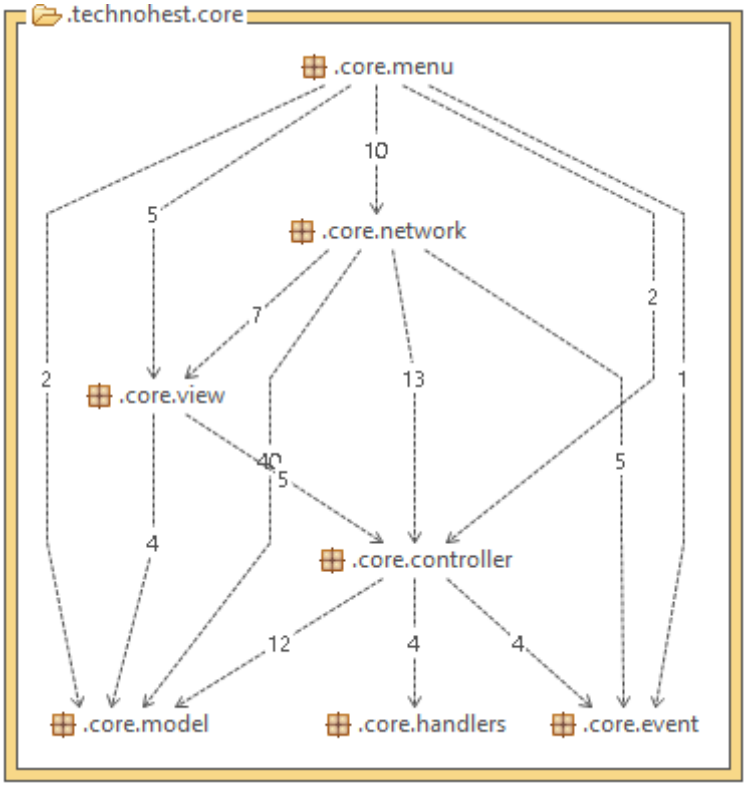
This section contains material which can be useful for understand the design of the program and its continued development.

Box2D & LibGDX
https://www.youtube.com/watch?v=85A1w1iD2oA

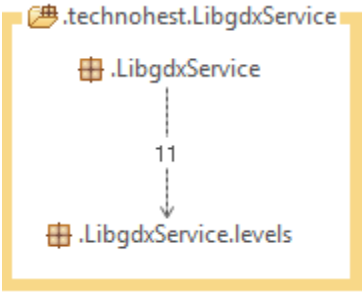
Networking:
http://gafferongames.com/networking-for-game-programmers/
http://www.gabrielgambetta.com/fpm1.html

http://www.diffen.com/difference/TCP_vs_UDP

Appendix



Core Diagram



Game Logic Diagram