System Design Document for NeedForGhetto (SDD)

Contents

Version: 1.3
Date: May 30, 2015

Authors: Anton, Marcus, Amar, Jonathan

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The design must be modular to provide the possibility of switching the GUI, controller and/or logic. The design must be written in such a way that it is easy to maintain and change parts of the code.

1.2 Definitions, acronyms and abbreviations

- GUI, Graphical User Interface
- Java, platform independent programming language.
- MVC, a way to partition an application with a GUI into distinct parts avoiding mixing GUI-code and application code.
- JSON, file format used for transmitting data in a structured way.
- TrueType, a font format with high degree of control over how to the font is displayed.
- Android, mobile operating system.
- libGDX, framework for developing games in Java.
- asset, binary file that contains e.g. sound, texture or font data.
- preferences, interface used to write persistent data.

2 System Design

2.1 Overview

The application uses a modified version of the MVC pattern for Android.

2.1.1 Model Functionality

The entry point to the model is the World class. We split the model into different classes such as Player, Enemy etc, to keep the design modular and prevent the classes from getting unnecessarily long and complex.

2.1.2 Global lookups

The state of the game is tracked with a global variable. The game state can take on the values *PAUSED*, *RUNNING* or *VICTORY*, and is used to decide if we want to render or not in the *GameScreen* class, the VICTORY state is a special case to see if the user have won. Winning is defined as killing the boss of the level.

There is a special state called *godmode* which is mainly used for testing purposes, it disables the collision control and makes the player invincible.

2.1.3 Event handling

Generally when the application gets an input from the user it is handled by the controller package which then updates the model accordingly. The view is continuously rendered so a callback from the model is not needed.

Event not handled this way is buttons in menus and the back-key, they are handled directly in the corresponding Screen class. The motivation for this design choice is that these input have nothing to do with the model.

2.2 Software decomposition

2.2.1 General

The application is decomposed into two main packages, android and core, this is required by libGDX, see figure 1. The android package is for android specific code and assets. The application code resides in the core package, the core package is dependant on libGDX as it uses some of its functions and classes.

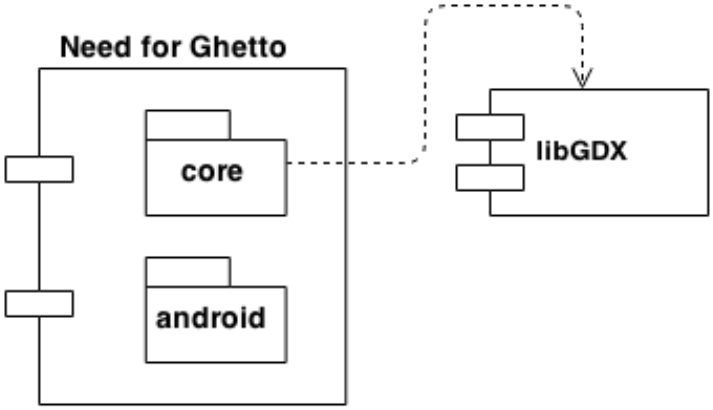

Figure 1: top level packages and libGDX dependency

The core package is further decomposed into: (see figure 2)

- view, the GUI part of the game screen, more specific the rendering.
- screen, the GUI part of the application.
- controller, the controller classes for MVC.
- parallax, for parallax background.
- highscore, highscore module for reading / writing highscore from file.
- gamestate, keeps track of the game state.
- model, game logic for the application, model part of MVC.

The rendering of the game is decoupled from the GameScreen class to make it easier to debug, modify and add code.

The model is further decomposed of 3 package and 4 classes, the reason for the packages is to reduce clutter and make the package easier to navigate. (see figure 4)

There is also a package for testing.

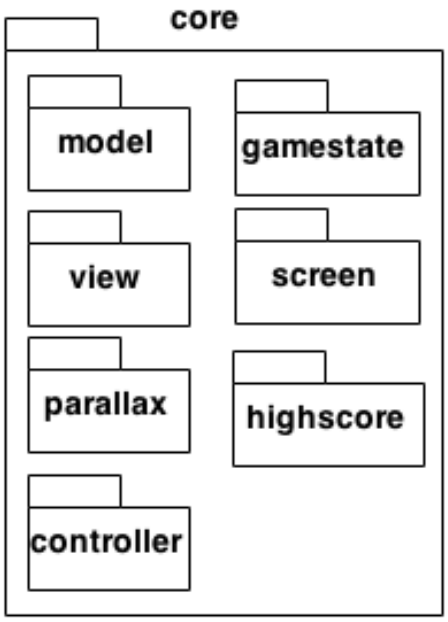

Figure 2: Packages with the *core* package

2.2.2 Decomposition into subsystem

The only subsystem present are parallax and highscore.

2.2.3 Layering

see figure 3

2.2.4 Dependency analysis

see figure 3

2.3 Concurrency issues

All concurrency is handled by libGDX.

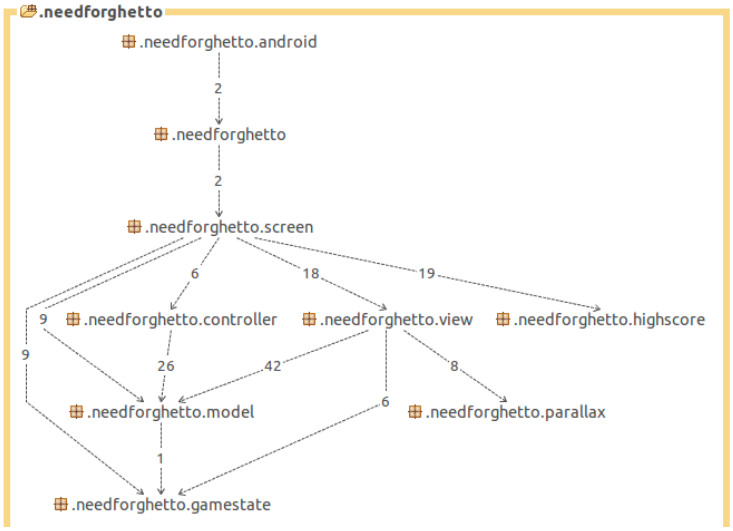

Figure 3: Layering and Dependency analysis

2.4 Persistent data management

Assets, such as textures, fonts and level information, are persistent and operated on with functions provided by libGDX. The different formats used for storing data are the following:

- High score data is handled through the preferences interface.
- Level design data is stored as JSON files.
- Fonts are stored as TrueType files.

2.5 Access control and security

NA

2.6 Boundary conditions

The application is launched and exited as a normal android application.

3 References

1. MVC, `http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`
2. game type, `http://en.wikipedia.org/wiki/Shoot_%27em_up`

Appendix A Class diagrams

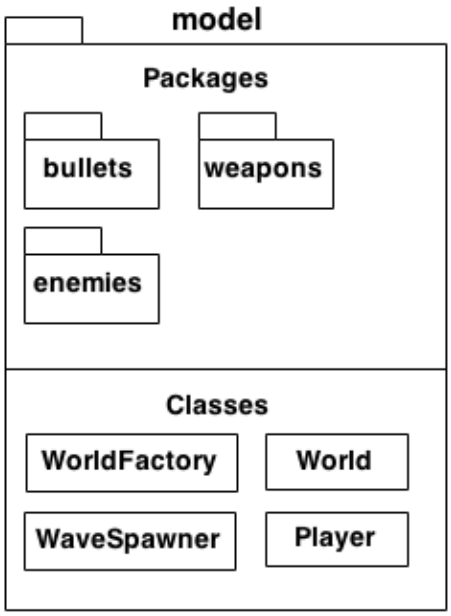

Figure 4: Packages and classes within the *model* package

Explanation of packages and classes:

- packages
  - bullets is the package containing different types of bullets and the abstract Bullet class.
  - weapons contains the abstract Weapon class and different types of weapons.
  - enemies package contains the different types of enemies and the abstract Enemy class, also contains a EnemyFactory used by the WaveSpawner.
- classes
  - WorldFactory builds a the complete model.
  - World is the main class for the model.
  - Player is what the user controls.
  - WaveSpawner is the class responsible for spawning waves of enemies.