System Design Document

Mattias Torstensson, Karl Wikström, Agnes Mårdh, Luka Mrkonjic

May 2017

Version: 2

Date: 27-05-2017
Revised by: Mattias Torstensson
This version overrides all previous versions.

1 Introduction

This document describes the structure of the source code for the game application described the in the Requirements and Analysis Document.

1.1 Design goals

- Only the application's input handling, graphics, and file handling should be coupled to the LibGDX framework. This will allow the application to be ported to other devices such as mobile phones with ease, but also makes it possible to replace the framework without having to rebuild the application's model.
- Most of the application should be testable with automated tests. Parts of the application that aren't should be manually testable.

1.2 Definitions, acronyms and abbreviations

1.2.1 MVC

MVC, Model-View-Controller, is a design pattern that separates an application into three parts. It separates the logic(Model), the output(View) and the input(Controller). A passive MVC pattern is a version of the pattern where the Model does not update the View when it changes, it remains passive. Instead it is the Controller that notifies the View when the Model changes [1].

1.2.2 Strategy Pattern

The Strategy design pattern is a design pattern where parts of the code are made interchangeable with each other, usually with the help of a common interface[2].

1.2.3 PMD

PMD is a program that looks for common programming flaws in an applications source code[3]. PMD can be used in common integrated development environments for Java, such as Eclipse.

1.2.4 Java

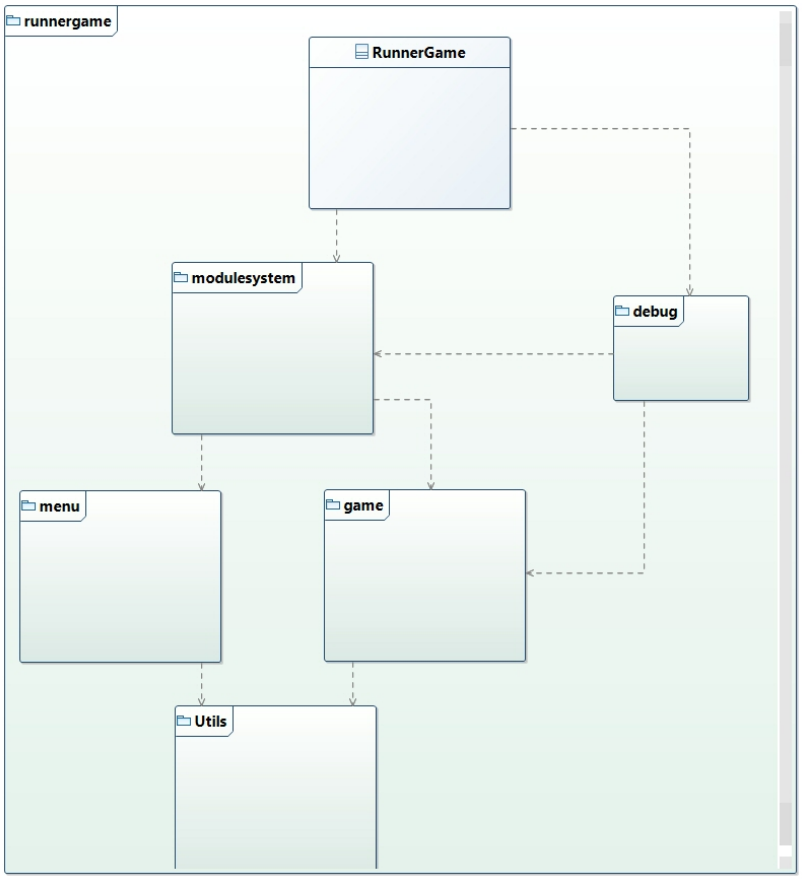
Java is an object-oriented, platform-independent programming language.

2 System architecture

The application will run on a single desktop computer. The application will be written in Java and will be made out of modules that use their own independent passive MVC pattern.

2.1 Top level

At the top level the application is made out of the following class and five packages:



2.1.1 Modulesystem

In order to make it easy to swap between different states of the application, such as the main menu and the game, these states will be split up into their own individual modules that have their own passive MVC pattern. The application will be able to swap which of these modules it is currently running at will. This will make the modularity of the different states high and keep their coupling low.

The modulesystem package contains the interface used to turn a controller into a module, with the help of the Strategy design pattern. It also contains the classes that use the interface to turn a controller to a module.

2.1.2 RunnerGame

The class RunnerGame serves as the entry point for the application as well as the class that handles which module is currently in use. Since this is the class

that controls the modules it is also the class which has the entire applications update and render loop.

2.1.3 Menu

The menu package contains the entire MVC model that is the application's menu.

2.1.4 Game

The game package contains the entire MVC model that is the game part of the application.

2.1.5 Utils

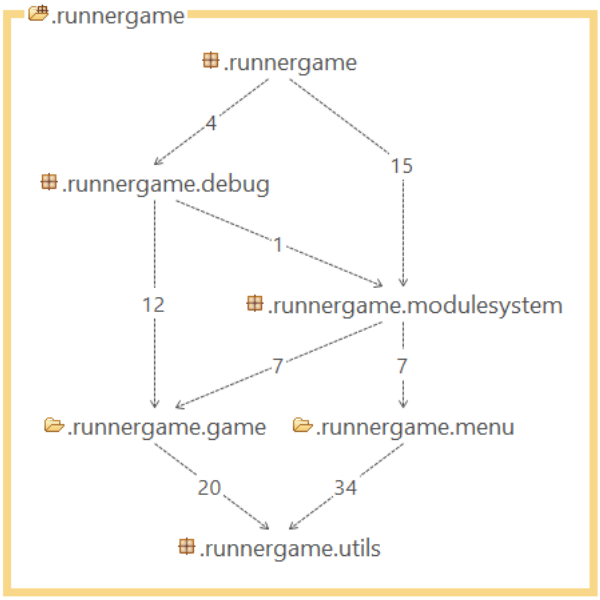
The Utils package contains classes that are used by multiple modules, such as the highscore class.

2.1.6 Debug

The debug package contains all classes used to help with debugging and manually testing certain parts of the application, such as a class that makes it possible to view the world generation step by step. Due to not actually being part of the game the utils package has strange dependencies relative to the rest of the application, but it is set up so that it can easily be removed from the application before release.

2.2 Dependency Analysis

Dependencies for the top level are as shown:



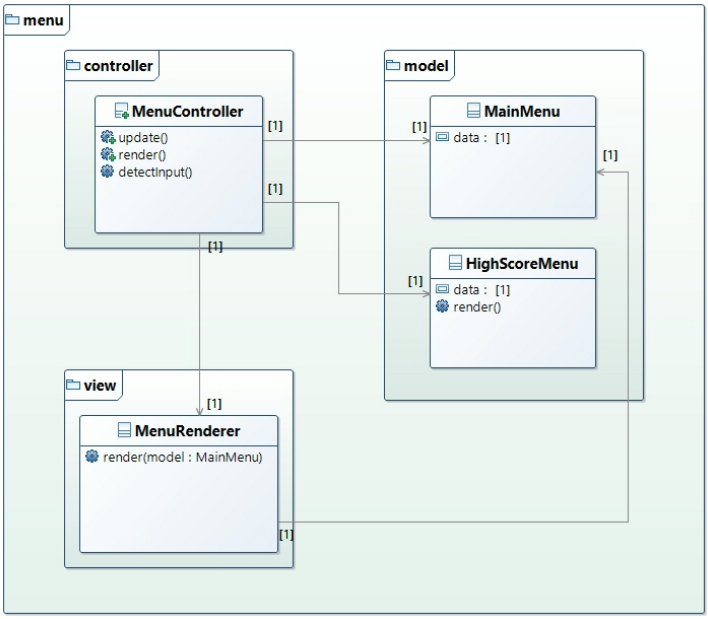
3 Subsystem decomposition

Beyond the Module system at the top level the application is split into two subsystems, the menu and the game. Both of these systems are stand alone MVC models that have no knowledge of any other system at the top level. The subsystems are split into three packages:

- The Model package has all the data and logic of the module. For example in the Game module it handles where the game's character is positioned and how it moves. While the model can be used by the controller to make it respond to input it has no knowledge of the controller.
- The View package contains all the code that takes care of rendering the module's data. The view has knowledge of the model so that it can read the appropriate data from it, but it has no knowledge of the controller that forwards the data from the model to the view.
- The Controller package serves as the entry point to the module, since the controller is the part of the module that tells the model and view to either update the logic or update the graphics respectively when using a passive MVC pattern. It is inside the controller package that all the input from the user is taken care of and forwarded to the model in the form of method calls. The controller has knowledge of both the view and the model.

3.1 Menu

The Menu package contains the entire MVC model of the application's menu component, and has the following design model:

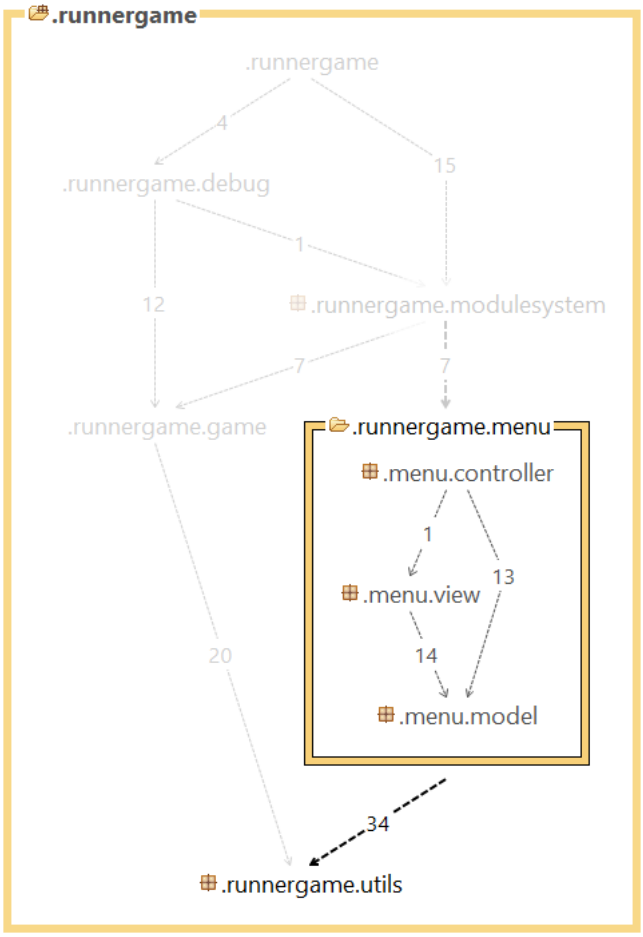


The entry point of the menu is the MenuController class in the .controller package. Creating an instance of it and calling its public methods will run the menu. The controller has knowledge of both the package's view and model. The controllers public methods are used to update the other components.

The controller class takes care of input from the user and with the help of the data inside the model classes decides how to respond.

3.1.1 Dependency Analysis

Dependencies for the menu module are as shown:



3.1.2 Quality

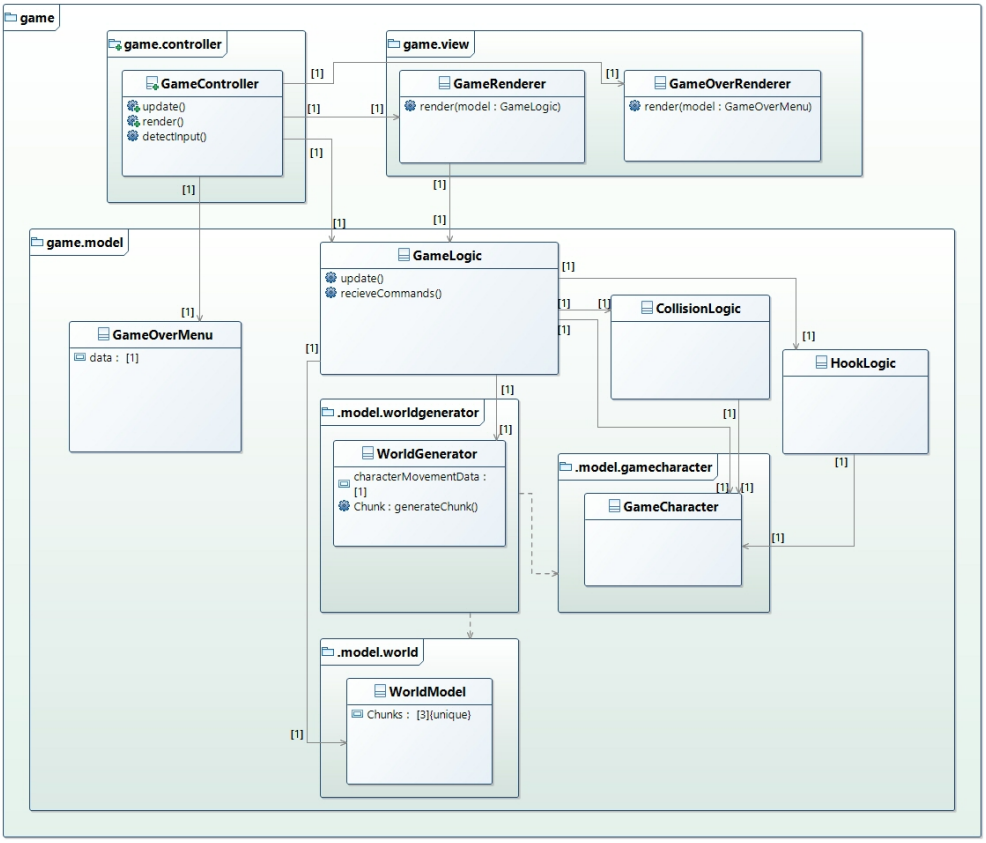
The current implementation of the HighScoreMenu has both its view and model inside of the same class instead of being split into their own seperate classes.

The menu module has only been tested manually. The tests consisted of trying all the use cases that affected the menu and making sure they worked, aswell as manually testing if the collision of the menu's buttons worked appropriately by manually clicking on them repeatedly. All of the use cases for the menu appear to be working correctly.

The module has been analyzed by PMD and currently has no violations of "critical" or higher level, using the default configuration for the Eclipse plugin.

3.2 Game

The Game package contains the entire MVC model of the application's game component, and has the following design model:



The entry point of the game is the GameController class in the controller package, simply creating an instance of the controller and then calling both of its public methods at the desired framerate will run the game. The controller has instances of the view, GameRenderer, and the model, GameLogic. The controller's update and render methods tell the model and the view respectively to update their part of the system.

The entirety of the model is contained inside the GameLogic class, as GameLogic has instances of all the other components of the model.

GameLogic's update method runs one frame of the game's simulation when called. During the update, GameLogic updates the GameCharacter's state depending on the input it has previously recieved via commands sent from the

controller. It also uses its two helper classes, CollisionLogic and HookLogic to further manipulate the GameCharacter. Finally it manipulates the world, WorldModel, by replacing the Chunks of the world that the character has ran past by creating new Chunks with the help of the WorldGenerator class.

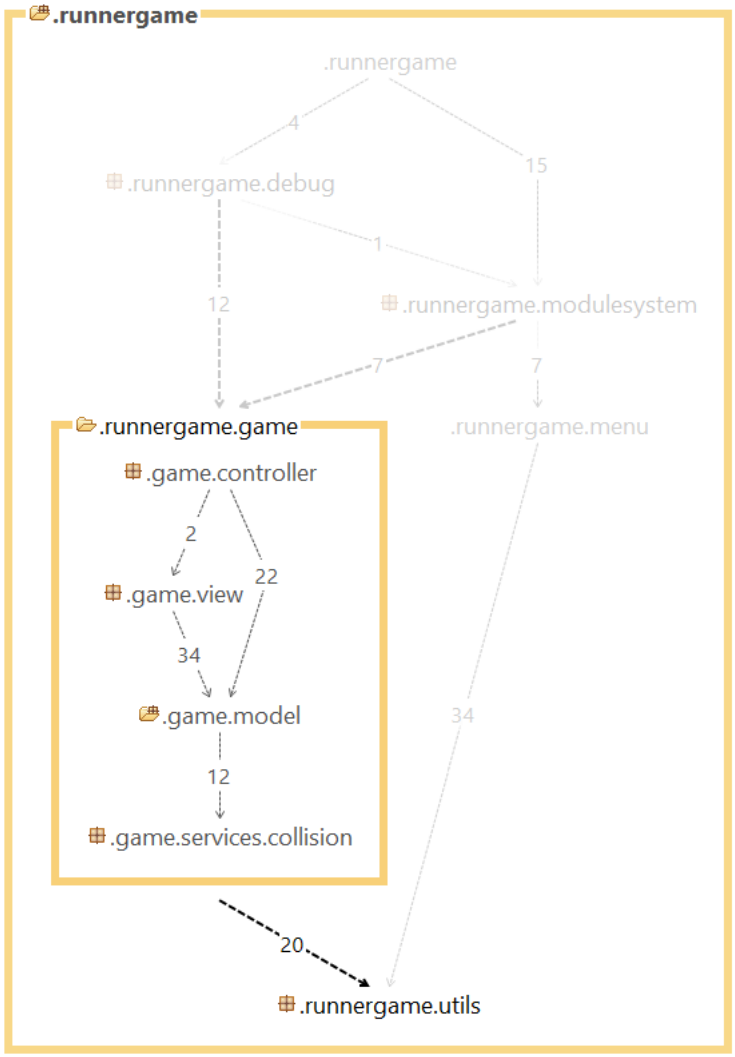
The world package contains all the data that its primary class, WorldModel, uses to represent the world that the character is in.

The gamecharacter package contains all the data and logic that handles the position and movement of the players character, GameCharacter.

The worldgenerator package contains all the logic that helps its primary class, WorldGenerator, to generate parts, or Chunks, of the world. It also contains data detailing which points of the world the character can move to when standing on a given point. This data guarantees that the generated world can successfully be navigated by the character. Because the generator creates parts of the world, aswell as extracts data from the character, it depends on both the world- and the gamecharacter package.

3.2.1 Dependency Analysis

Dependencies for the game module are as shown:



3.2.2 Quality

Tests for the game module can be found in the core/test folder in the source code. Parts of the game that lack tests have been tested manually by trying to fulfill the different use cases for the application and have all passed.

The module has been analyzed by PMD and currently has no violations of "critical" or higher level, using the default configuration for the Eclipse plugin.

4 Persistent data management

All handling of assets and save data is being done with the help of LibGDX's file handling systems. Assets are stored in core/assets. Saved high score data is stored at different places depending on OS:

- Windows: `%UserProfile%/.prefs/MyPreferences`
- Linux and OS X: `~/.prefs/MyPreferences`

5 Access control and security

NA.

References

[1] Khademul Basher. *MVC Patterns (Active and Passive Model) and its implementation using ASP.NET Web forms*. From 26 April 2017. URL: `https://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-its`.

[2] *Strategy Design Pattern*. From 27 April 2017. SourceMaking.com. URL: `https://sourcemaking.com/design_patterns/strategy`.

[3] *PMD*. From 27 April 2017. PMD. URL: `https://pmd.github.io/`.