System design document for RunningMan

Contents

Version: 1.1
Date 2015-05-31

Author Armand Ghaffarpour, Simon Lindkvist, Jesper Olsson, Johan Tobin

This version overrides all previous versions.

1. Introduction

1.1 Design goals

The design goals are mainly to use the MVC design pattern. The idea is that the model shouldn't know anything about the framework, so that it will be easy to swap. As long as the visual framework provides the tilemap functionality, it can be used. The design should be as modular as possible, so that certain features can be deleted and the application remains runnable. The design should also make it easy for testing the code without creating unnecessary dependencies.

1.2 Definitions, acronyms and abbreviations

- GUI, graphical user interface
- Java, platform independent programming language
- Libgdx, external library for a game engine
- MVC, Model View Controller is a design that tells you how to partition an application.
- Game loop, all the code that makes the game interactive and dynamic goes in the game loop, but is separated into different parts. The game loop itself is a infinite loop that makes your game keep running. It's the place where all parts will be updated and drawn on the screen.
- Mock objects, simulated objects that mimic the behavior of real objects in controlled ways.

2. System design

2.1 Overview

The application will mainly use an active MVC pattern, but it will also have elements of passive MVC, such as Observer pattern to handle audio output etc. The model will handle the internal representation of data for a player, enemies and level objects. The visual representations will be handled by the view classes, using the libgdx framework. Libgdx is also used for handling keyboard input in the PlayerController and for reading tile maps of the tmx format in the map handler class. Interfaces are used when needed in the testing of the code to make mock objects to prevent dependencies.

2.1.1 The model functionality

The game loop is situated in the libgdx render method, which is called upon all the time. This render method updates the GameController, which updates the other controllers, and tells the LevelView and HudView to draw themselves. The PlayerController handles the input from the keyboard and updates the Player model accordingly.

2.1.2 Rules

The Level model handles the level timer, which keeps track of how much time is left of the level, and updates the time. The GameController checks if the time us up, and if it is, quits the level and takes the user to the main menu. The Player model keeps track of if the player is dead or not, and whether the player has finished the level or not, and the GameController checks those states in every update.

2.1.3 Event handling

The application uses propertychangesupport and the propertychangelistener libraries, which are used to communicate between models and audio. This type of communication is also used to tell the game when to switch screen.

The application handles input from the keyboard using the classes Input and InputProcessor. InputProcessor extends the class InputAdapter from the Libgdx framework, and the class Input is used to store the buttons that are pressed. In every update, the player controller checks with the Input class if any buttons have been pressed.

2.1.4 Map generation and loading

In order to make it easy to create new maps and loading maps, we created a system wherein we read from a .tmx file to get the positions for every object in a level. With this system the MVC pattern is followed and new maps can easily be made by external programs e.g. Tiled, which is a map editor that generates a .tmx file.

2.1.5 Handling collisions with Visitor pattern

In order to tell certain objects how to behave when colliding with other objects, we have used a Visitor pattern. This pattern contains of two two interfaces, Visitable and Visitor. The Visitable interface is implemented by all objects that can be visited by other objects, meaning all visual objects in the game. Visitor is implemented by all objects that are able move, and therefore visit other objects. In our case this is Enemy and Player. Visitable contains of one method, which is acceptVisitor. Visitor contains an arbitrary amount of methods, depending on the amount of objects that implements Visitable. All those methods are overloaded methods named visit, with a parameter for each visitable object.

2.2 Software decomposition

2.2.1 General

The application is divided into the following six main packages:

- **model -** contains interfaces and classes of things that are not game objects themselves, but are used by game objects, such as Gravity, Position, Size etc.
    - **model.gameobjects -** Contains interfaces and classes of actual objects that are in the game, such as Level, Player, Enemy, etc.
- **view -** contains the graphical representations of all objects that are visible.
- **controller -** contains all controllers, that update the model. PlayerController and WeaponController handle input from the user.
- **screen -** contains the different screen representations of the application
    - **screen.mainmenu -** contains screen, view and controller for the main menu.
    - **screen.loadlevelmenu -** contains screen, view and controller for the load level menu, the menu that you choose level from.
    - **screen.highscoremenu -** contains screen and controller for the high score menu, which is showed when the player finishes a game.
- **util -** contains miscellaneous functionality that are used by all other parts of the application
    - **util.highscore -** contains functionality to save and load highscores from file, and to display highscores.
    - **util.input -** contains functionality to handle and save keyboard input from the user, which are used by the controllers.
    - **util.map -** contains functionality to handle tile map files (tmx-files), so that we are able to use tile maps to create levels.
- **audio -** contains functionality to handle the audio output

2.2.2 Decomposition into subsystems

The MapHandler class in the util.map package uses libgdx TmxMapLoader to read the tmx-files containing the graphical representation of a level.

2.2.3 Layering

The layering are shown in the Figures from the STAN program below.

2.2.4 Dependency analysis

The internal dependencies are shown below. There are no circular dependencies.

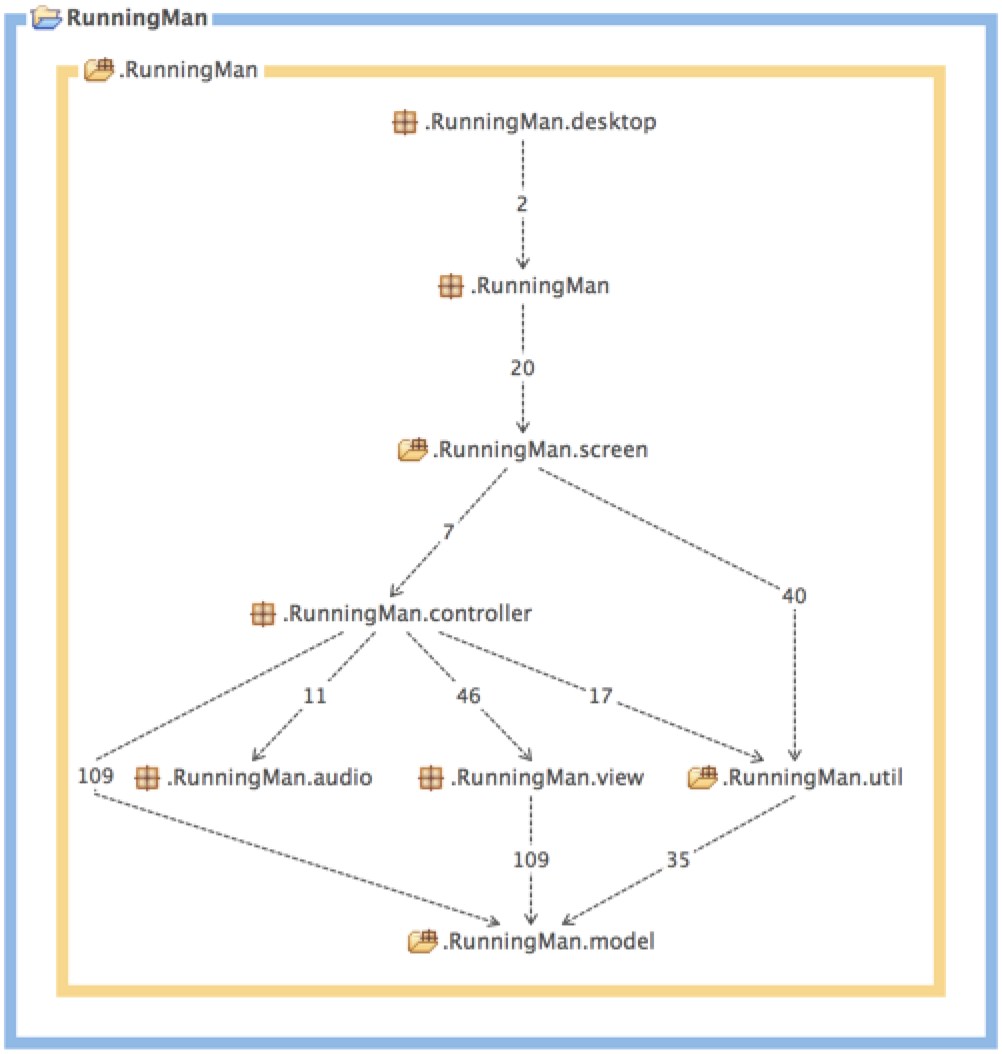


Figure 1: external packages

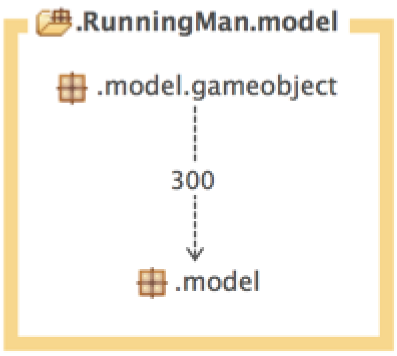


Figure 2: external model package

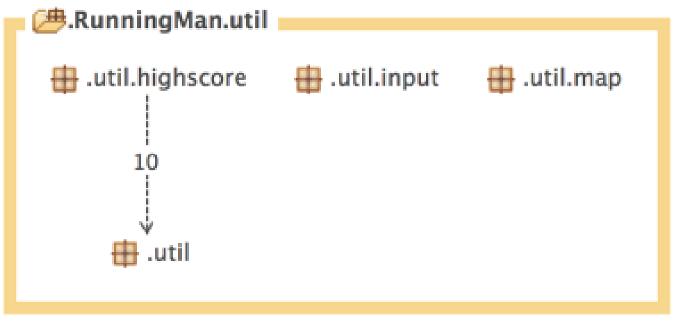


Figure 3: util package

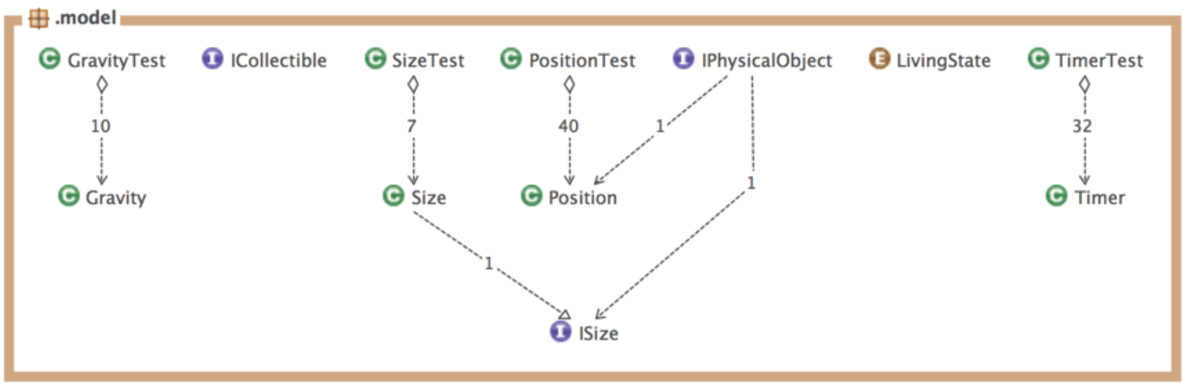


Figure 4: model package

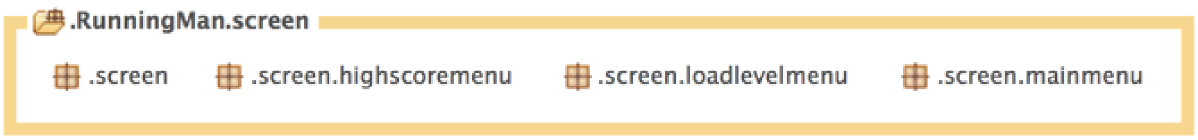


Figure 5: screen package

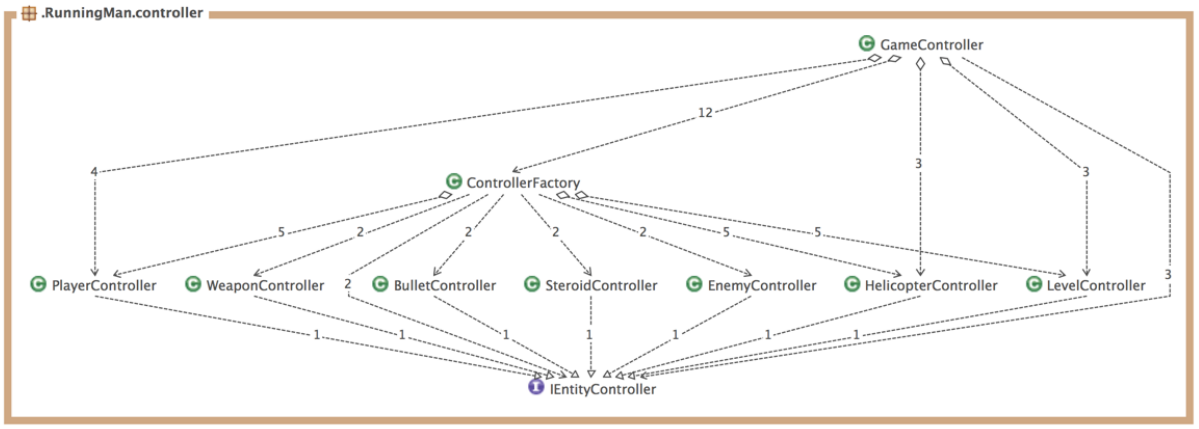
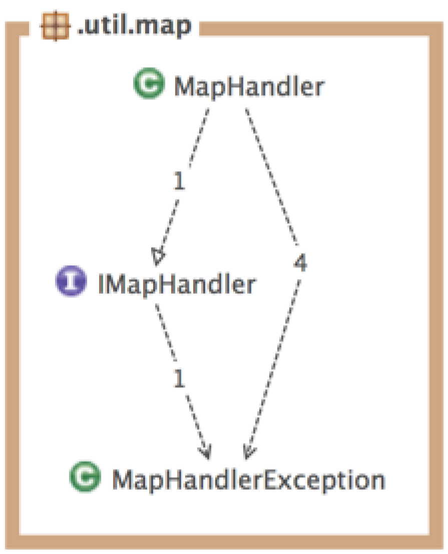


Figure 6: controller package



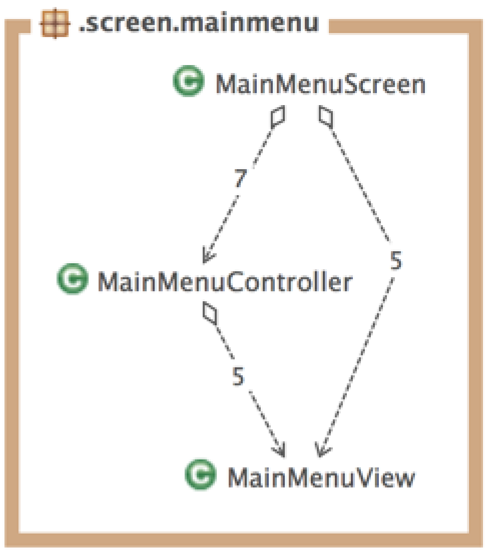


Figure 7: map package

Figure 8: main menu package

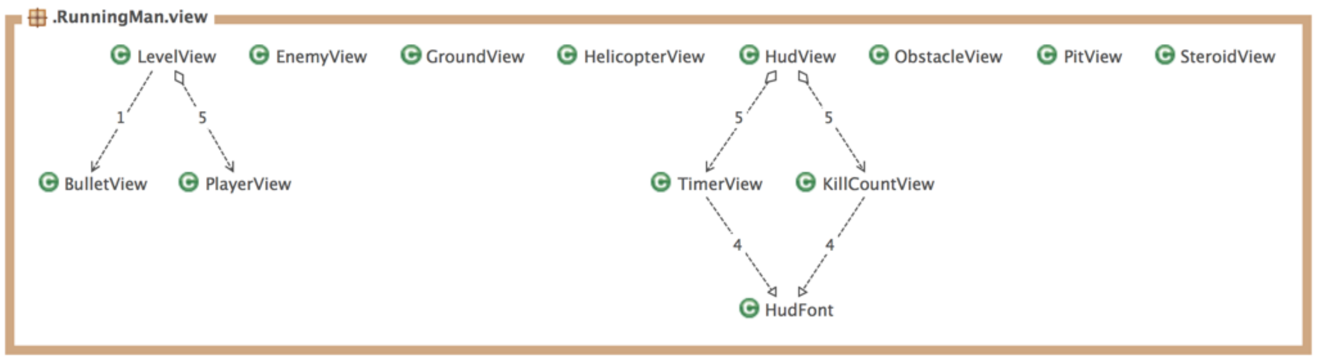


Figure 9: view package

2.3 Concurrency issues

NA. This is a single thread application. Therefore there are no concurrency issues. The game is however based on events in the controller classes that are determined from the user input.

2.4 Persistent data management

No data is saved between playing sessions. However all objects in the game are inserted into arraylists for management.

2.5 Access control and security

NA

2.6 Boundary conditions

NA - The application will be launched as a normal desktop application.

3. References

1. MVC, see https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html
2. Libgdx, see http://libgdx.badlogicgames.com
3. Visitor pattern, see http://en.wikipedia.org/wiki/Visitor_pattern