System design document for the SSHA project (SDD)

Contents

Version 0.9
2013-05-26
Sebastian Bellevik, Adam Jilsén, Tomas Hasselquist, Rasmus Lorentzon

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The design must have loose coupling on some parts but may have tight coupling on others to make it as efficient as possible. The application would be better with only loose coupling but considering the time frame for an application of this scale some compromises had to be made such as making some classes have tight coupling. It should however be mostly possible to isolate classes to test them. For usability see RAD.

1.2 Definitions, acronyms and abbreviations

- GUI, graphical user interface.
- Java, platform independent programming language.
- JRE, the java Run time Environment. Additional software which is necessary to run a Java application.
- Host, the computer the game is run on.
- Round, a breaking point where all players but one is dead. All players are sent back to get ready for another round when this happens.
- Gold (for players), the resource which the player uses to buy and upgrade skills and items.
- Arena, the map where the players can move around and use skills on.
- State, the application is built up with states where a new state is a new GUI and has new functionality.
- Render, generating images and or motion pictures
- MVC, a way to disctinct parts to separate application code, data and GUI.
- LWJGL, light weight java game library uses OpenGL to render images.

2 System design

2.1 Overview

The application uses a modified MVC model. The library Slick2D does however apply both View and Control in the same class and the application has tried to work with this.

2.1.1 The model functionality

The game is based around the model PlayerModel which has a data class Player connected to it. The class Player does in turn keep Skill classes that has StatusEffectShell classes. The StatusEffectShell classes implements an interface, called IStatusEffectShell, to commit the special various effects different effects can have on players. The class Skill is abstract to make the different skills easier to modify when they need to do specific purposes such as an upgrade of a skill.

2.1.2 Rules

The rules in the game always stay the same. The players are to in some way survive the longest in a round. This can be achieved in various different ways such as killing other players and/or luring them into traps that will hurt them. After a round the players get gold to upgrade or buy new skills and items. This will make a user able to be very specific in what gamestyle he or she uses.

2.1.3 Unique identifiers, global look-ups

The class MainHub will act as a hub for different classes to communicate. The class is a singleton and will hold whatever information that will have to be sent between different states since they can not communicate directly.

2.1.4 Render

The rendering of the arena will only use the Player class with all the data to draw the images at the correct positions. The calculations will only take place in the PlayerModel class to minimize the work for the rendering process. The application uses LWJGL to render images.

2.1.5 Event handling

Many different events may happen throughout a round. Therefor a class called PlayerModel has been implemented to take care of whatever the player may do on the arena. The PlayerModel calculates positions and checks if the player has been hit by either skills or by walking into obstacles, after which various effects may take place.
To solve the issues with web connections we decided to use a system where the different hosts send their own players information to the server and in turn always recieve all the information from all players connected to that session. The own host will then sort this information and make actions depending on what information was recieved.

2.2 Software decomposition

2.2.1 General

Package diagram. For each package an UML class diagram in
appendix

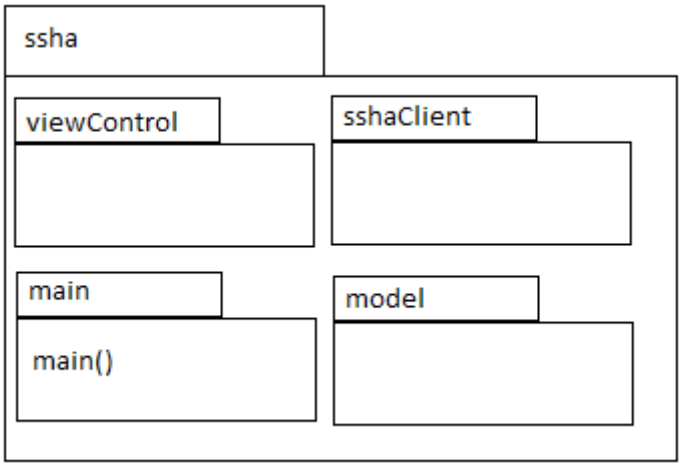
- control, the gameEngine and PlayerControl
- model, core classes that calculates and keeps data. Model Part for MVC
- sshaclient, the necessary client classes to handle the connection to the server
- view, the different states. View Part and Control Part for MVC due to Slick Engine building it that way.

2.2.2 Decomposition into subsystems

The subsystems used are Slick2D GameEngine to handle the IO and update frequence.

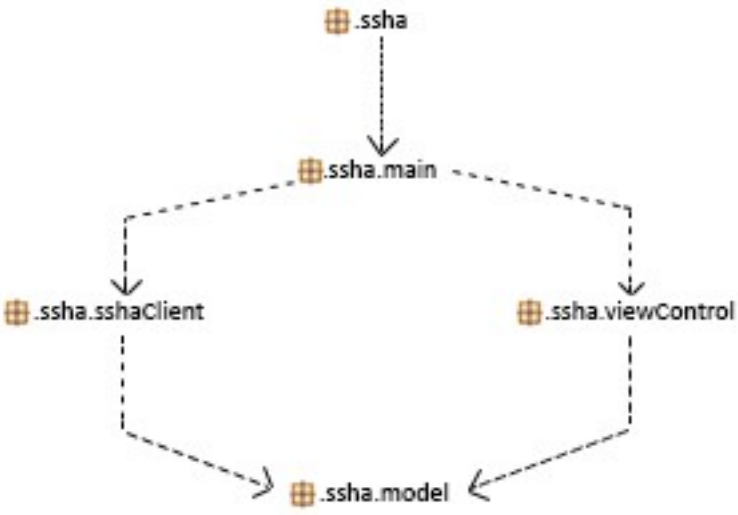
2.2.3 Layering

The layering is as indicated in the figure below.



2.2.4 Dependency analysis

Dependencies are shown in the figure below. No circular dependencies.



2.3 Concurrency issues

The application consists of several threads to make it run smoother. All the calculations for different players take effect in different threads so that each player is part of its own thread. This is handled by making the methods different threads can use and modify synchronized. This way concurrency issues will not appear as they will take turn on which thread uses the methods.

2.4 Persistent data management

There is no persistent data as everything is supposed to reset after every gameplay.

2.5 Access control and security

NA

2.6 Boundary conditions

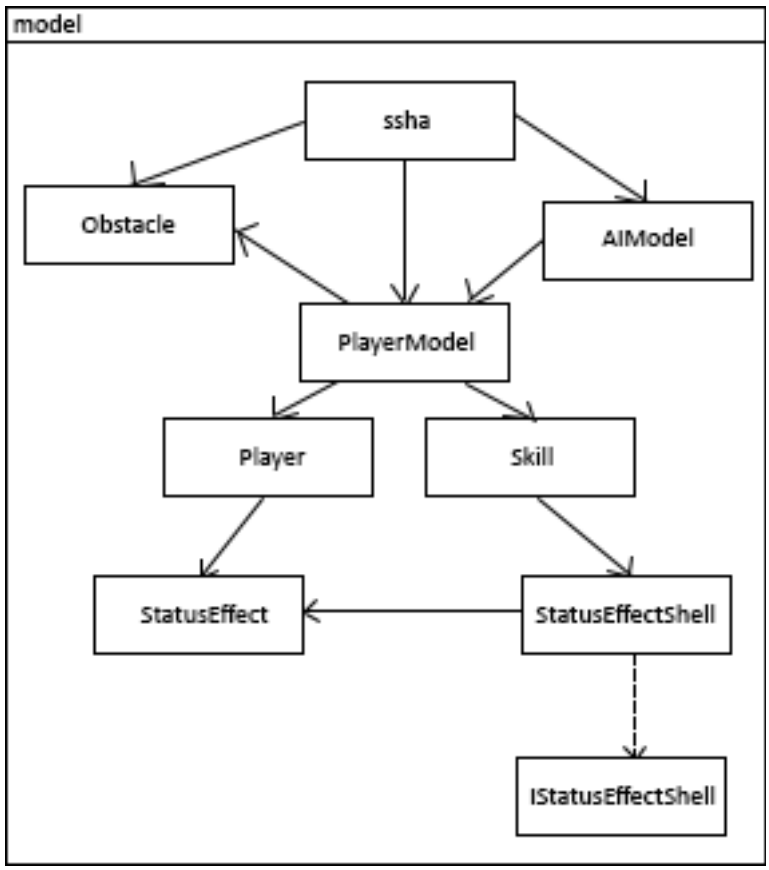
The application can only be run on the platform that Jar was packaged for. The library Slick2D can however not run on Mac OS when running with Java 7 and therefor the application will only be runnable on windows and possibly unix. (Not tested on unix but should be possible if the correct Jar file containing to unix natives is run)

3 References

1. MVC, see http://en.wikipedia.org/wiki/Model-view-controller
2. Render, see http://en.wikipedia.org/wiki/Render
3. LWJGL, see http://www.lwjgl.org/

APPENDIX

Class diagrams for packages (Model base package)



- ssha is the game engines control class contacting model.
- PlayerModel calculates everything necessary for a player and the class also checks if player is hit by skills and activates various effects depending on what happened. Playermodel can also affect obstacles depending on what happened.
- Player is the data class that PlayerModel modifies
- Obstacle is an extendable class to create obstacles
- AIModel is a runnable class using its own thread to calculate what the AI should do with its model
- Skill is a base class for various skills
- StatusEffectShell is a shell with the necessary data to create a statusEffect and add it to a player through the PlayerModel
- StatusEffect controls the effect when it is added to a player through the PlayerModel. The effects trigger through PlayerModel though.

Sequence diagram for method move()