System Design Document for the “Psycho Hero” game project (SDD)

Contents

Version: 1.0
Date: 22th of May

Author: Carl Jansson, Erik Tholén, Peter Eliasson & Simon Persson

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The design should be as loosely coupled as possible to easily adapt to the MVC pattern. All logical objects in the game, such as the hero himself, enemies and so on, has to be implemented as separate models with corresponding views. The models itself should have as little logical control over the game as possible, containing only placeholders for position, speed and other relevant values. By distributing the code into models whenever possible, the game should also be more testable.

1.2 Definitions, acronyms and abbreviations

- Hero, the main character in the game that's controlled by the user.
- GUI, graphical user interface.
- MVC, a specific design pattern that separates the logic, the GUI and the data of an application.
- Java, a platform independent programming language.
- Enemy, made up enemy character(s) that tries to kill the hero in different ways.
- Slick2D, an open source Java 2D game library.

2 System design

2.1 Overview

The application will be using a MVC pattern. However, due to the characteristics of a game, the view and the controller might not always be as separated as in a standard MVC application. The application will be written with the Slick2D library as a base.

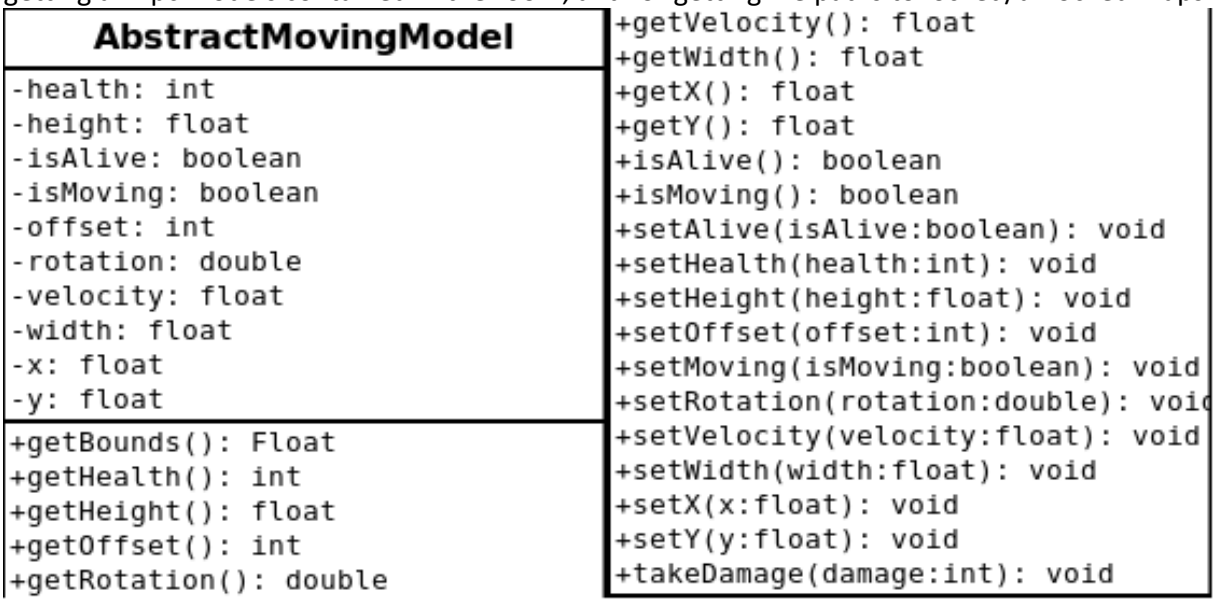
2.1.1 **The models' functionality**

There are going to be three different kinds of models - models for moving objects, weapons and rooms.

The AbstractMovingModel class will be an abstract class containing all the common data of a moving model (such as position, rotation etc). All classes representing a moving object in the game ultimately extends this class via either AbstractCharacterModel or AbstractProjectileModel. This is to avoid duplicating code and to be able to use Java's polymorphism on method calls to each moving model in the game. Thanks to polymorphism it becomes easy to control different kinds of moving models using the same controller.

All weapon models extends the AbstractWeaponModel class via subclasses for each weapon type. Every weapon has parameters for damage to be dealt when fired, a name and a firing speed.

Every room in the game will implement the interface RoomModel, which has methods for getting all npc models contained in the room, and for getting file paths to locked/unlocked maps.



2.1.2 The views' functionality

All moving models will have it's own view. the CharacterView, which is used by the hero and all the npcs, gets its resource file path from the model thus making sure that it uses correct pictures and animations. Projectile and HUD has its own View classes. The purpose of the views is to separate the graphics from the models. For example, the HeroView gets images from the file system, and then creates and renders an animation for the HeroModel.

2.1.3 The controllers' functionality

There will be a main GameController class forwarding update, render and initialize calls from slick to all other controllers in the application. Every moving model and its corresponding view in the game is going to have its own controller. These controllers extends the abstract class AbstractMovingModelController, a class containing some common functionality for such a controller. For example, AbstractMovingModelController has set and get methods for both the view and the model. AbstractNpcController is one of the classes extending AbstractMovingModelController. This class implements logic for pathfinding and movement to make it easy to implement new enemies.

As for other controllers, there will also be a RoomsController that manages all the different rooms in the game. The RoomsController has methods for adding rooms, and for shifting the room that the game currently displays.

For controlling sound, there will be a SoundController class with methods for playing both sound effects and music.

The application (or game) is divided into different kinds of states, like menu states and the playing state itself. For controlling and switching between these, the class StateController is used.

2.1.4 Events

The game features an event handling and logging system that can be used to save events in the game, which later could be used to generate statistics about previous game sessions. However, the implementation of said feature will be left for a later time as the user interface required would be too complex to justify with regards to the limited amount of time.

2.2 Software decomposition

2.2.1 General

The application is decomposed into the following modules (ie packages):

- controllers, the controllers for the moving models, rooms and sound.
- database, the database in which to store all events, scores etc.
- event, Different ways to use events.
- exceptions, Special exceptions thrown by the game.
- main, the Main class that is used to start the game.
- menu, the classes used to create menu items.
- models, the weapon models and the moving models.
- settings, Constants and key bindings.
- states, The StateController and all the different states of the game.
- util, Utilities.
- views, the views for each type of model.

2.2.2 Decomposition into subsystems

There are two subsystems in the packages event and database respectively. The database will be used to store and manage data such as high scores, bullets shot etc. The event system described in 2.1.4 is another subsystem that will later be used for logging statistics.

2.2.3 Layering

The layering of the program is as shown in figure 2.

2.2.4 Dependency analysis

Dependencies are shown in figure 3 below.

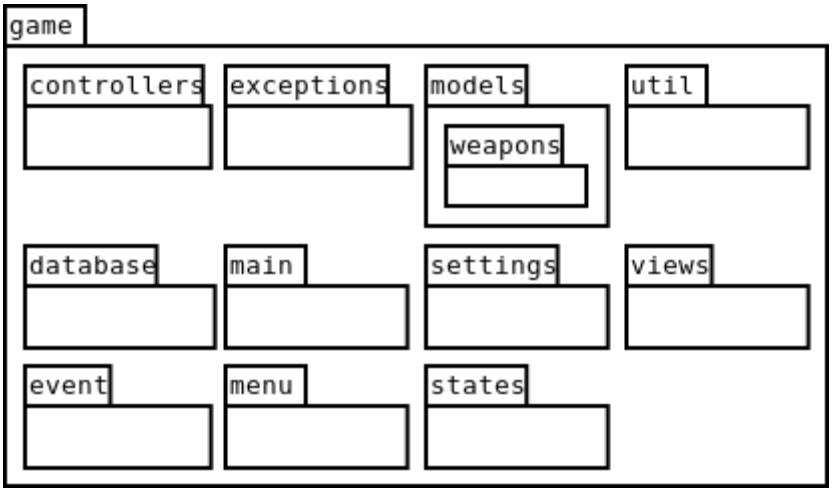


Figure 2: High level design

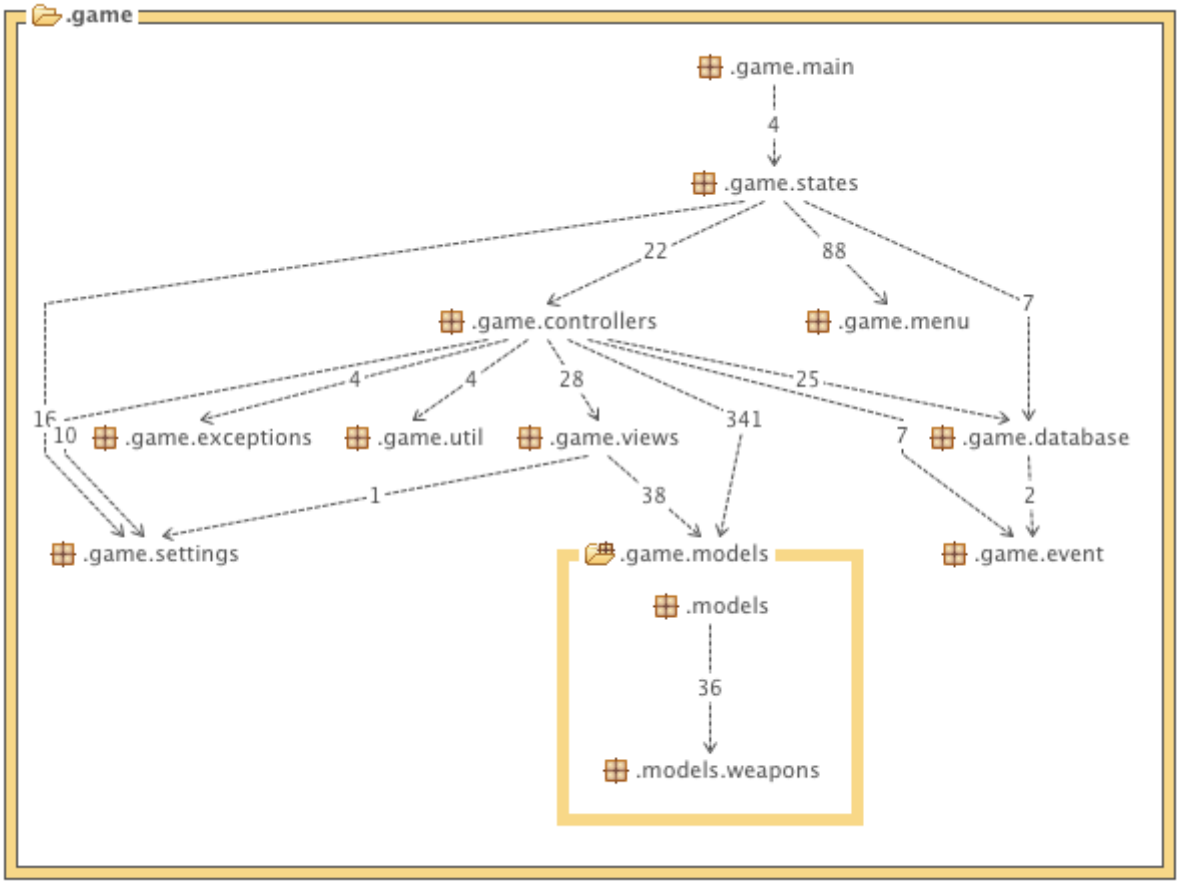


Figure 3: Layering and Dependency analysis.

2.3 Concurrency issues

N/A, Slick2D is single-threaded, or at least does only expose a single thread to the implementation.

2.4 Persistent data management

Settings that can be bound to a user on the local machine, such as key bindings and menu options, will be saved using the Java Preferences library. For saving data that can not be saved as a single value, such as the multiple high scores, we will be using a database.

The database solution used in the project is called SQLite. The reason being that it stores all the data as a single file, making it a lot easier to both manage and set up the database compared to the more complex solutions, such as MySQL. The cons with using SQLite is that it does not scale, nor does it perform, at the same level as MySQL or similar databases. However, in our situation, where the database is not a main component of the application but rather quite the small part, we came to the conclusion that the pros of using SQLite as our database widely outweigh the cons. If we were to change our minds later on, porting from one database to another should be trivial, as they both communicate using the SQL language.

Another great feature that using SQLite provides is the possibility to run the database in RAM. This allowed us to extensively test the database class without modifying the already stored data.

2.5 Access control and security

For database security, PreparedStatements will be used to secure the application from SQL injections when inserting data.

2.6 Boundary conditions

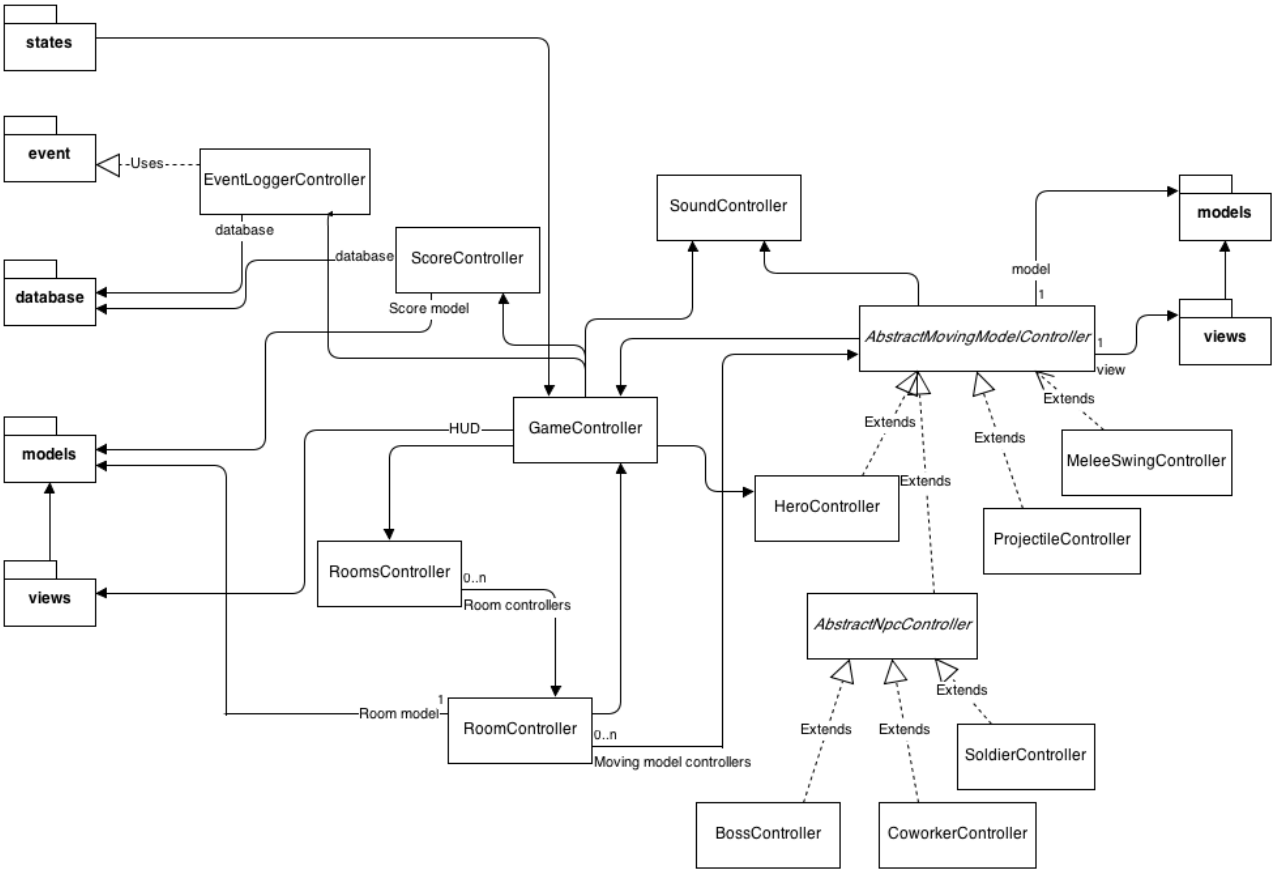
N/A.

3 References

- SQLite, see http://www.sqlite.org/
- PreparedStatement, see http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/PreparedStatement.html
- MVC, see http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

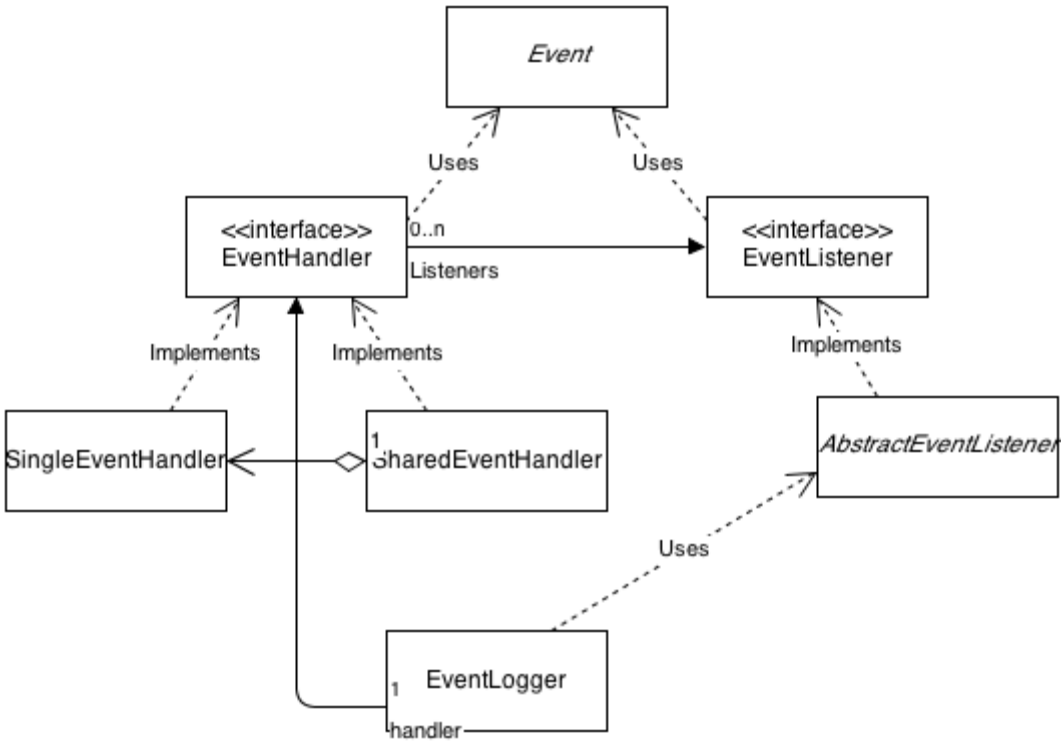
APPENDIX

General game logic



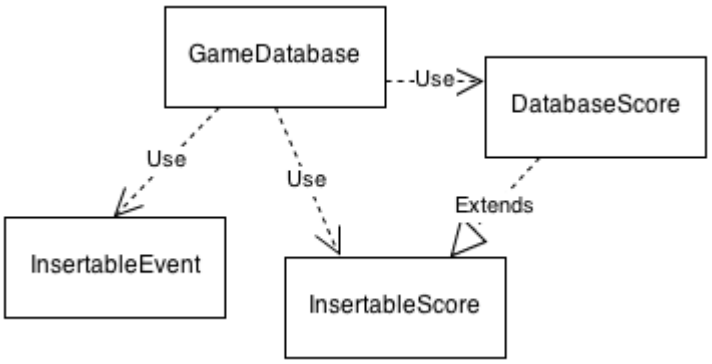
The play state class in the state package forwards Slick2D calls (init, update and render) to the GameController when the game is running.
The GameController in turn forwards these calls to the HeroController, the ScoreController and the RoomsController.
The RoomsController again forwards calls to the currently active RoomController.
The active RoomController handles sending the calls onwards to the last controllers in the chain, the various implementations of AbstractMovingModelControllers registered to the room.
The GameController also directly requests the HUDView in the View package to render, as it the HUD does not have any logic associated with it, and therefor does not have a controller.
Every implementation of the AbstractMovingModelController has a seperate view and model.
The ScoreController and the EventLoggerController both uses the database to save score/events.
Every implementation of the AbstractMovingModelController has access to the GameController and the SoundController so that the controllers can affect the more global state of the application. For example, the hero controller uses the sound controller to play a sound when the hero is hit.
Views in the views package are usually connected to one, or in some cases several, models from the model package. However, they only query the models for a current state, and does not modify the models in any way.

Event



The abstract Event class is to be extended by all events.
The EventHandler is responsible for handling publishing events to all its listeners.
The SingleEventHandler is an instantiated implementation of the EventHandler interface. The SharedEventHandler is a singelton implementation that uses a single SingleEventHandler.
The EventListener interface is a generic type interface that handles any event extending Event.
The AbstractEventListener is a generic class implements the EventListener interface for a specific sub-type of events.
The EventLogger is a utility class that listens to an event handler and logs all event.

Database



The GameDatabase is the class responsible for handling inserting and retrieving score and information about events to and from the database.

The InsertableScore and InsertableEvent are two classes that maps the data to the database schema.
The DatabaseScore is a class returned by the GameDatabase, holding some additional information about the score that is saved. Such as time inserted.

Sequence diagrams