System design document for Robots Stole My Girlfriend (RSMG)

Table of Contents

Version: 1.0
Date: 2012-05-09
Group: 8
Author: Daniel Jonsson, Johan Grönvall, Victor Rådmark, Johan Rignäs

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The design is aimed to be testable and well structured. To detect errors we are going to use a lot of junit tests, especially for the model part. The Model is to be completely ignorant of the

controller and GUI, so that it can be independently tested.

1.2 Definitions, acronyms and abbreviations

- RSMG: Project name, Robot Stole My Girlfriend
- Platformer: a game where the goal is to get from point A to point B by moving from platform to platform.
- GUI: Graphical user interface
- Java, platform independent programming language.
- Upgrade Point: a form of in-game currency used to purchase upgrades for a character
- MVC, a way to partition an application with a GUI into distinct parts avoiding a mixture of GUI-code, application code and data spread all over.

2 System design

2.1 Overview

The design will follow the MVC model.

2.1.1 Model functionality

The model is split into two major parts,Tiles and Interactive objects. Tiles are static and cannot be moved, they are used to describe the environment. Interactive objects are objects that can interact with the player in some way, these objects can be moved around and are mutable, they are used to describe enemies, items and the main character.

2.1.2 Graphics

We use the graphics library Slick, available at http://slick.cokeandcode.com/. Slick provides us with classes to easily create a game window, play audio, draw images and animations within the window and it also also has a easy way of creating multiply states/views within the application.

Due to Slick's structure there is a really heavy connection between the controller and the view, and in our project those parts are actually in the same package.

2.1.3 Subsystem

See section 2.4

2.1.4 Event handling / Updating

The application will not use events, instead it gets updated by a constantly going loop (this is also true for menus and the level-selection-screen). The application looks for user input by checking if a specific button is pressed down every loop.

The loop will first upgrade values stored in the model, and then redraw all objects.

2.2 Software decomposition

2.2.1 General

The project is decomposed into the following modules:

- Main, the projects entry class
- Controller, handle user actions from GUI
- GUI, receives events from the user
- Model, store variables
- I/O, store saved data

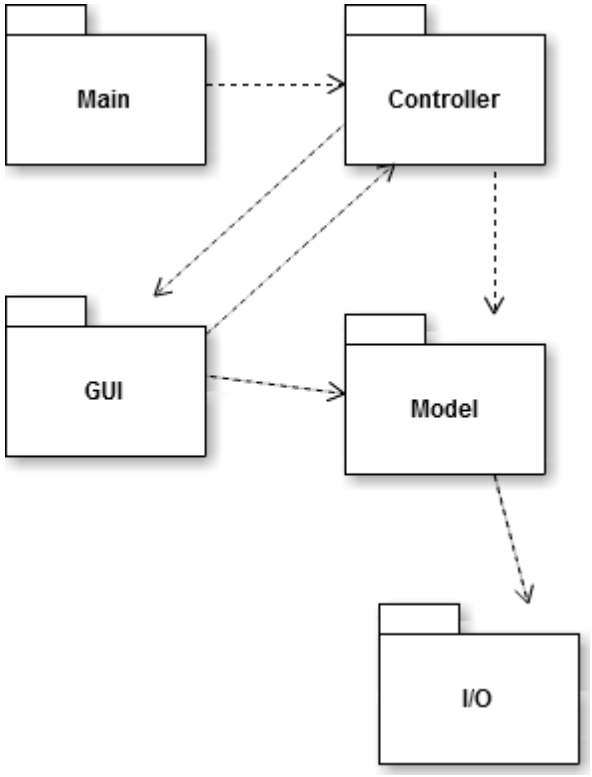

Figure 1: Package diagram

Seeing as how our application will be updating at a constant frequency, our controller and view are going to share a strong connection.
See appendix for UML class diagram.

2.2.2 Layering

2.2.3 Dependency analysis

We have avoided cyclic dependencies by putting both all interactive objects and weapons in the *model.object.unit* packet. This because each IWepaon will have a wielder and each of the LivingObjects will have a weapon.

2.3 Concurrency issues

NA. There is only one thread in the application. This is the AppGameContainer from slicks library that starts the application and keeps it alive. Thereby we have no concurrency issues.

2.4 Persistent data management

User data and levels are stored and read from XML files. To make this easier, we are using the library JDom. The levels can easily be created with a map editor which allows you to add all in-game objects. The map editor can be found at github, https://github.com/MaTachi/2D-Map-Editor.

2.5 Access control and security

NA

2.6 Boundary conditions

NA

3 References

APPENDIX

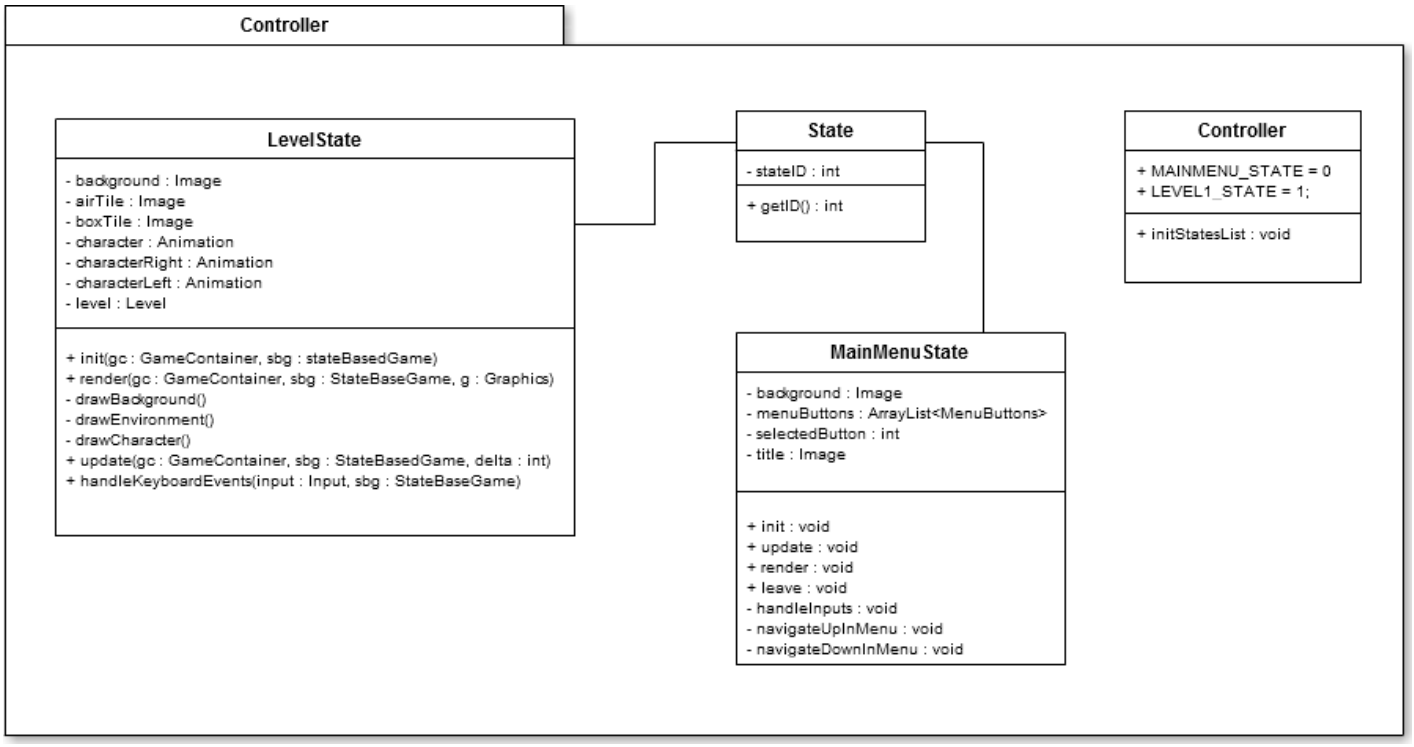

Figure X: Controller UML class diagram

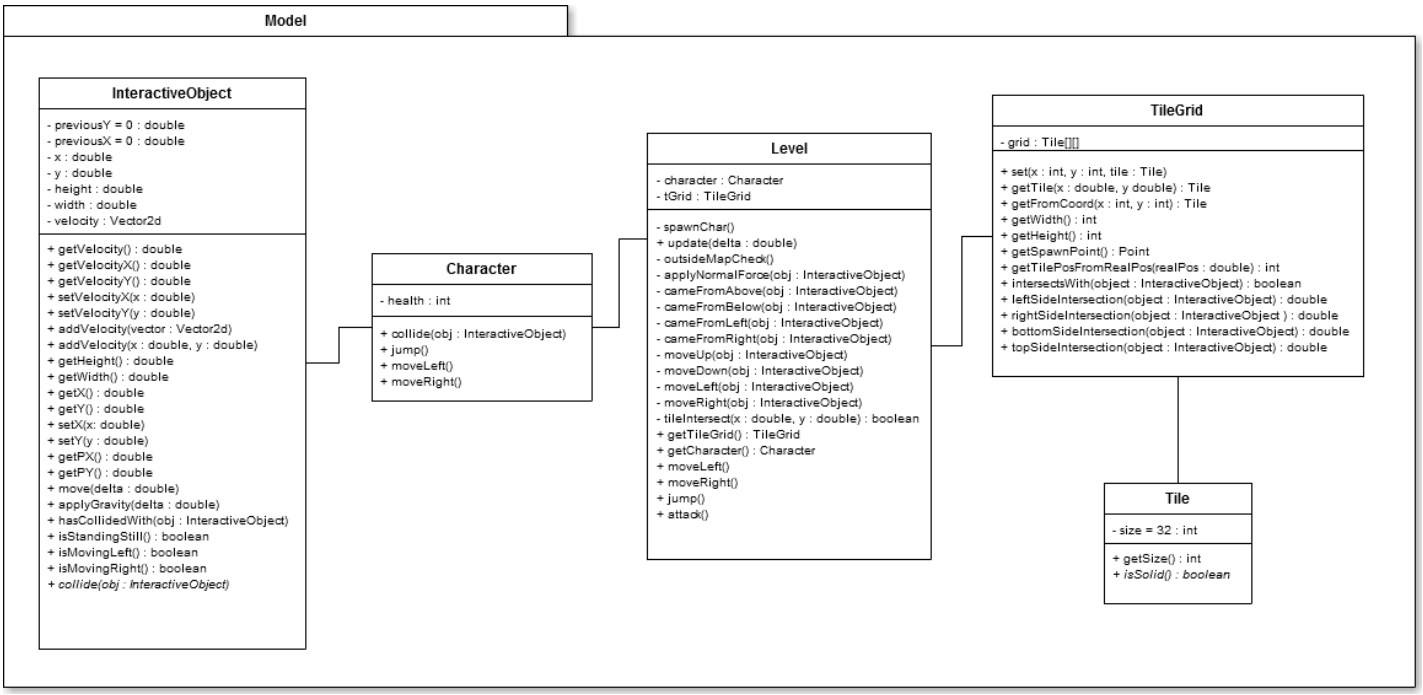

Figure X: Model UML class diagram

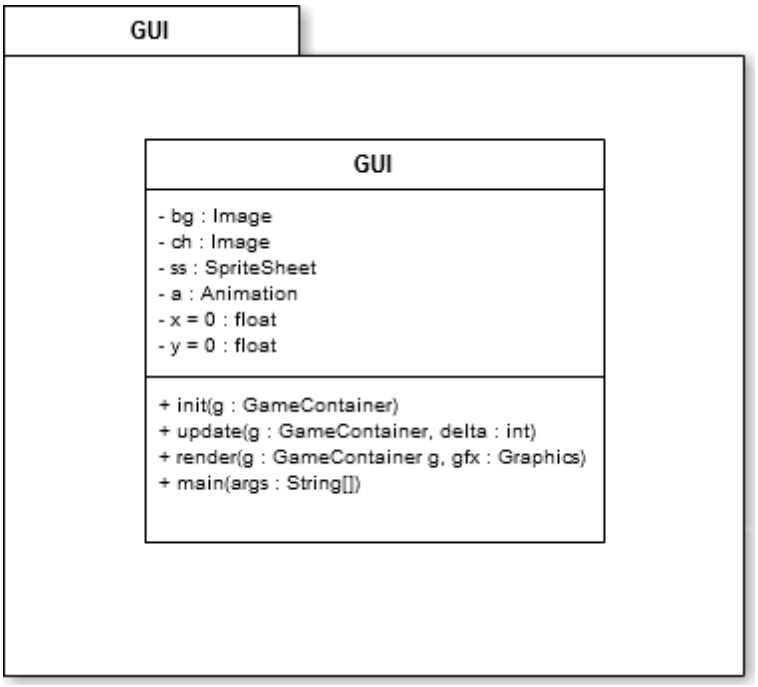

Figure X: GUI UML class diagram

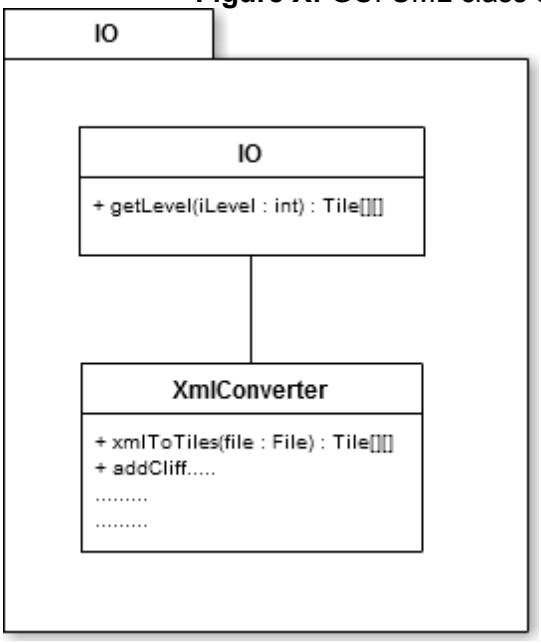

Figure X: IO UML class diagram