ShroomTowerDefence

System design document

Contents

Version 1.0

Date May 16, 2012

Authors Emil Edholm, Emil Johansson, Johan Andersson, Johan Gustafsson

This version overrides all previous versions.

1 Introduction

This section gives an overview of the design goals and defines some acronyms etc.

1.1 Design goals

The design must be loosely coupled to make it possible to switch GUI and/or partition the application into a client-server architecture. The design must be testable i.e. it should be possible to isolate parts (modules, classes) for test. For usability see RAD.

1.2 Definitions, acronyms and abbreviations

GUI Stands for Graphical User Interface

Java Platform independent programming language.

JRE The Java Run time Environment. Additional software needed to run a Java application

Host A computer where the game will run

Wave Represents a session of the game where enemies will try to reach the player base.

Loot value Represents an amount of money the player is given when an enemy is killed.

Sprite An image that represents game objects, such as towers and enemies.

Guice A dependency injection framework developed by Google.

Guava A set of libraries containing several different classes. Only the event bus is used at the moment

(de)Serialize To save/load an object to/from file

2 System design

See figure 5 in appendix A. Note that some classes have been omitted for brevity.

2.1 Overview

The application will use a modified MVC model. The game uses a single game controller that is split up into a build controller (controls building of towers) and a wave controller (controls enemy releases and movement).

2.1.1 Rules

When the player starts a new game he is given a set amount of lives. If a monster manages to run through the players defenses and into the player base the player will lose a life. The game will end when all lives has been lost.

When an enemy is killed by the players defense the player will be awarded with that enemy's loot value. If the player wants to he/she can save the accumulated score will be when the game ends in a highscore system.

2.1.2 The model functionality

The models functionality will be exposed by the interface IGame. At runtime a implementation is created på Guice (using dependency injection). The implementation of the game controller is split up into smaller pieces as stated in section 2.1.

2.1.3 Value objects

The model IGame expose functionality. This is quite different from when the application just need the core data of the object (for example to display in GUI). The design separates the functionality aspect and the data aspect. Implementations of IBoardTile are value objects.

2.1.4 Unique identifiers, global look-ups

We will not use any globally unique identifiers for any entity except the classes that implements Serializable. There will be no look ups from anywhere in the application (objects will be directly connected or accessible without an identifier).

2.1.5 Tiles

The game board itself is represented by a matrix of board tiles that consist of, walkable, buildable, terrain or player base tile. Walkable tiles are essentially path tiles for enemies to walk upon. Buildable tiles represents tiles which towers can be built upon and terrain tiles represents tiles that the player character can walk on. The player base tile represent the end goal for enemies to reach.

2.1.6 Event handling

Many events, state changing or not, can happen during the play (Build tower, sell, upgrade, enemy enter base etc). A need for a flexible event handling is evident. If this is done at an individual level i.e. each receiver and sender connects, we possibly end up with a hard

to understand web of connections (also possible many receivers for one event/sender). How and when should connections be set?

To solve the above we decided to utilize the Google Guava eventbus. In the receiving class, the receiving method is annotated to indicate that it is a event handler. It is therefore not Interface based. Makes it possible to name the event handler to better show what it does.

All connections of senders/receivers and transmitting of events is handled by the event-bus. The connections could be setup at application start for static parts. Dynamic parts must have means to connect to the bus at any time

2.1.7 Internal representation of text

None of the text in the game should be localizable and all text (where applicable) is written in English.

2.2 Software decomposition

See figure 6 in appendix B

2.2.1 General

std top level package that contains the main method.

core contains the game logic.

core.anno this holds the annotations used to describe towers, tiles and enemies.

core.effects effect interface and all concrete effects are stored in this package.

core.enemies package containing interface and abstract implementation of enemies.

core.events all custom events used in the game are stored in this package.

core.exported concrete implementations of towers and enemies are stored here to be loaded dynamically.

core.tiles tiles used in the games are stored in this package.

core.tiles.towers subpackage including interfaces and abstract implementation of towers.

gui all graphical user interface classes are stored in the gui package.

mapeditor containing all logic for the mapeditor.

mapeditor.events subpackage containing all events used in the mapeditor. tests, self explanatory.

utilities package including global utility classes.

2.2.2 Decomposition into subsystems

A system of reading the Enemies and Towers from disk and reading their annotation exists in the Core and Utilities package. Utilities also contains an IO class used to serialize and deserialize objects, as well as a FileScanner class used to list files in directories. A system of creating Sprites dynamically using Guice dependency injection also exists in the utilities package.

A system of reading and creating Maps exists in the MapEditor package.

2.2.3 Layering

No layer visualisation exists at the moment.

2.2.4 Dependency analysis

Dependencies are as shown in figures 1, 2, 3 and 4. There are no circular dependencies except between core and some of its sub packages. (no problem since this is just a administrative separation of classes for easier management).

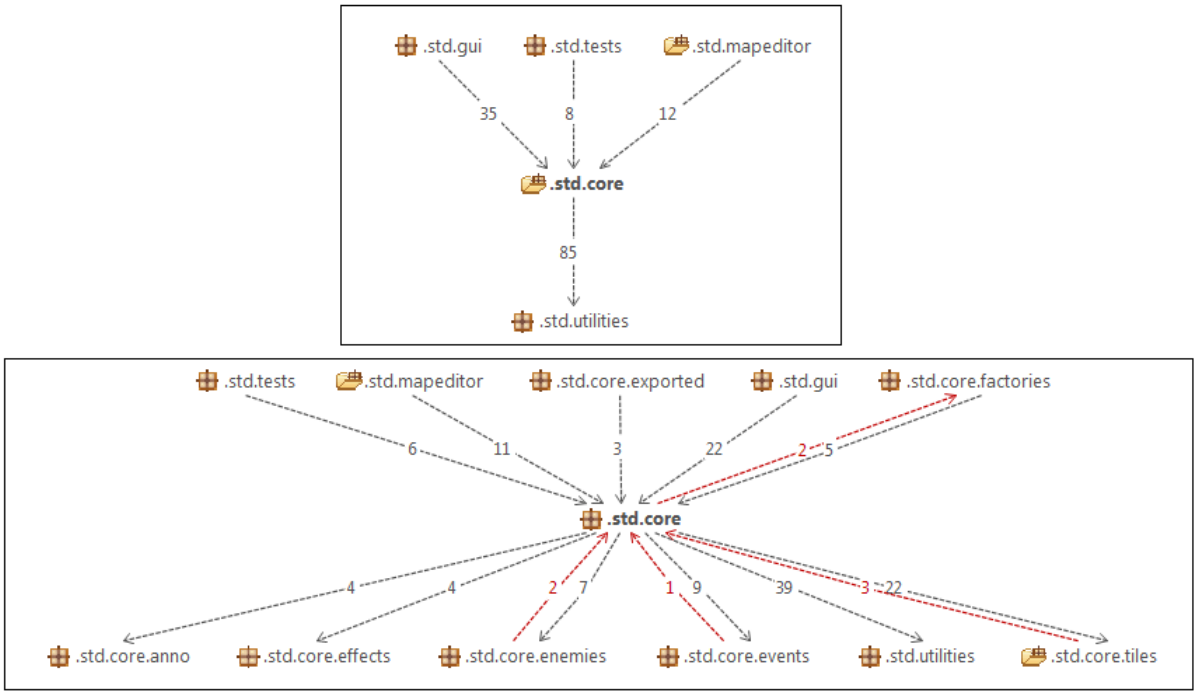

Figure 1: Dependencies on the core package

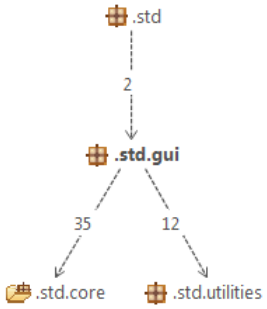

Figure 2: Dependencies on the GUI package

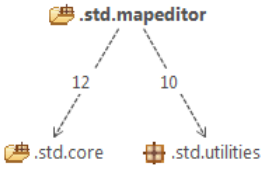

Figure 3: Dependencies on the MapEditor package

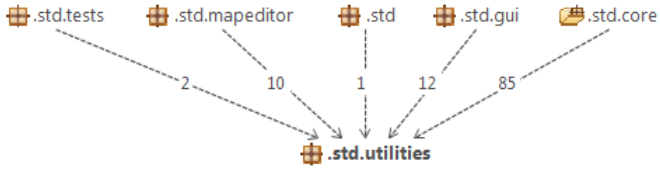

Figure 4: Dependencies on the Utilities package

2.3 Concurrency issues

N/A. This is a single threaded application. Everything will be handled by the Slick event bus. A fast computer may throw concurrency issues because of the sharing of often accessed data, such as the list of enemies.

2.4 Persistent data management

All persistent data will be stored in binary files. The files that should be stored:

- Map files, a serialized version of each concrete map. Should be saved in the map folder under the name levelx.map where x is the level that the map represents.
- The highscore is saved and loaded between each run.

2.5 Access control and security

N/A

2.6 Boundary conditions

N/A. Application launched and exited as normal desktop application (scripts).

3 Refrences

Slick - `http://slick.cokeandcode.com/`

Nifty-GUI - `http://nifty-gui.lessvoid.com/`

Guice - `http://code.google.com/p/google-guice/`

Guava - `http://code.google.com/p/guava-libraries/`

A System design

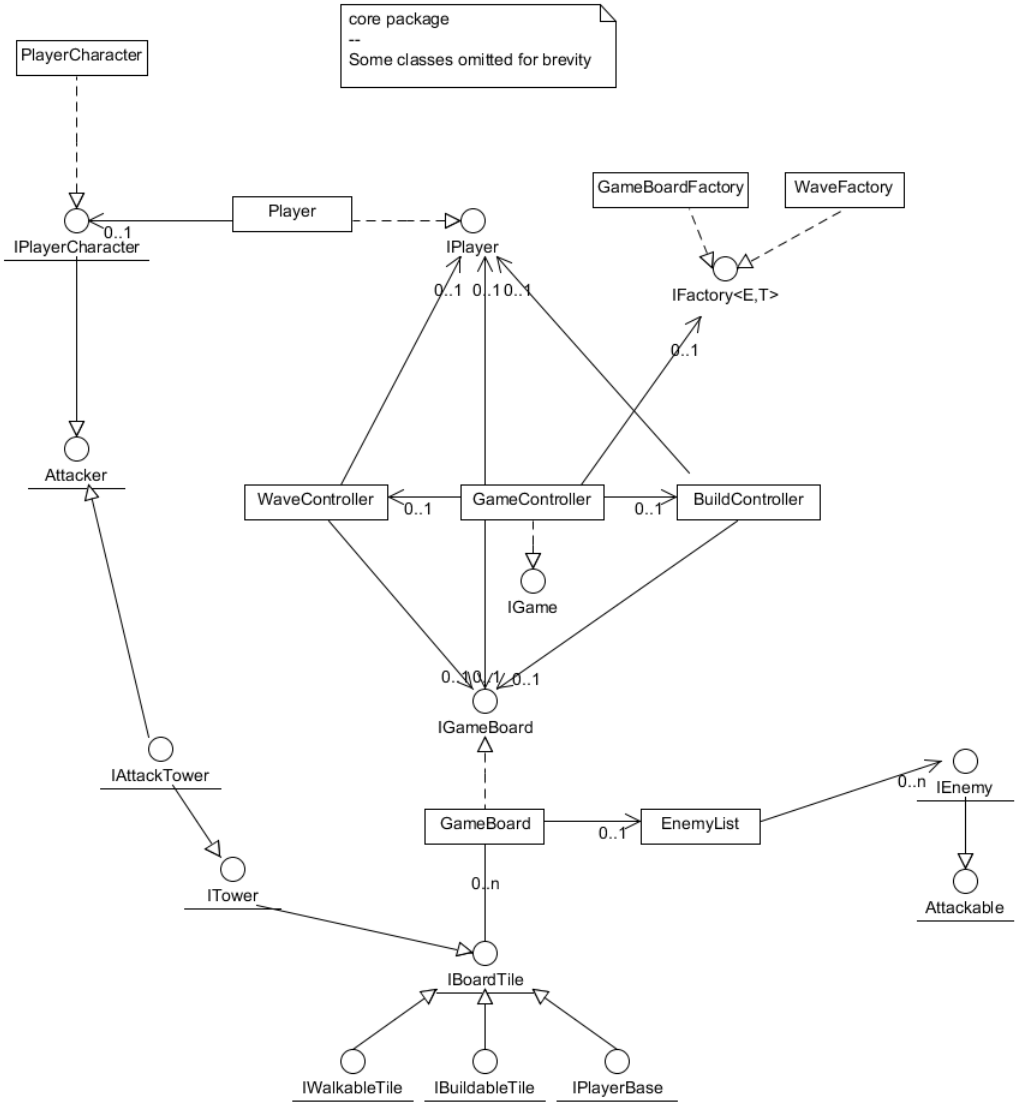

Figure 5: The system design of the core package

B Software decomposition

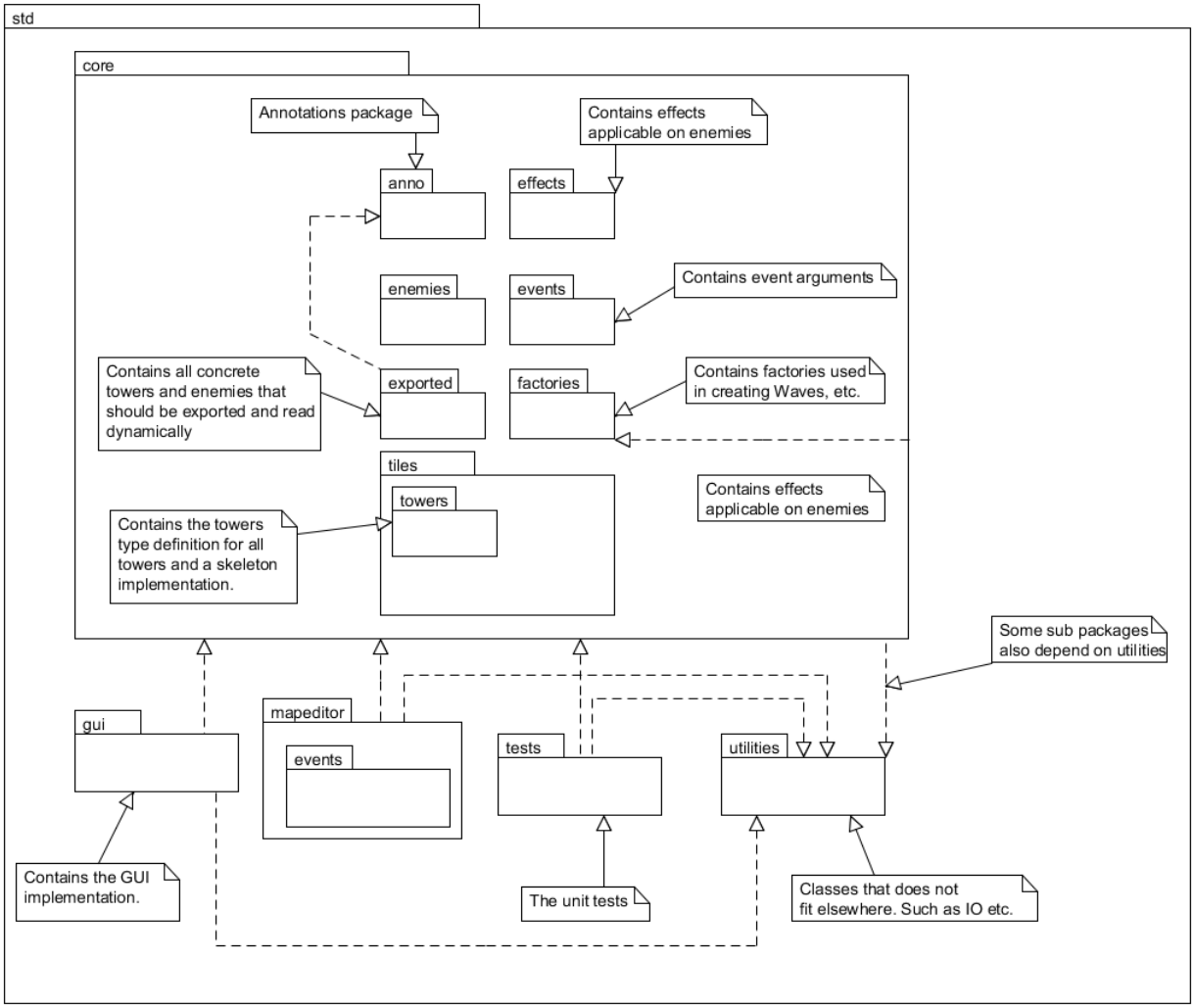

Figure 6: A package overview of the core package