Proximity

System design document (SDD)

Version 6.1
Date 31/5 2015
Author: Linda Evaldsson, Simon Gislén, Johan Swanberg and Hanna Römer
This version overrides all previous versions

Contents

1. Introduction

1.1. Design goals

The design should be loosely coupled with external libraries and the main focus is to have an advanced modular functionality. It must be easy to expand the program and add new components and functions. In addition, the design needs to be testable, that is the classes needs to be isolated and encapsulated.

1.2. Definitions, acronyms and abbreviations

- **Map**, the game board
- **Experience**, Points gained from killing enemies. When a certain amount of experience is collected the next level is reached.
- **Wave**, The enemies come in clusters, the current wave is the cluster the player is currently facing.
- **GUI,** graphical user interface.
- **MVC**, Model-View-Controller, a way to organize the structure of an application to avoid mixing application code (controller), data (model) and GUI (view) (Model-View-Controller, 2015).
- **Passive MVC,** an alternative way of structuring MVC where the model don't tell the view to update, instead the controller tells the view to update (Basher, 2013).
- **Java**, a platform independent programming language.
- **Encapsulation**, hiding properties in objects from the "outside" to protect from manipulation and to bundle related methods.
- **Java math library,** The math library included with java.
- **Stan,** a program which analysis code structure (www.stan4j.com)
- **libGDX**, a platform independent library for creating games (www.libgdx.com)

2. System design

2.1. Overview

The application will be written in java and use passive MVC.

2.1.1. Game states

To differentiate between different views the application will have several game states that it can switch between. Playing a map is one game state and viewing the menu is another. This will make it easy to keep coupling low and connections between classes loose, since these states need a minimum of communication with one another.

2.1.2. Maps and play state

The application will feature different maps. The interface of the start menu will display a list of available maps. Each map will have a previous map that has to be won before the map is available. Selecting a certain map and pressing start will switch screen to a game state and call a method in the game state that will take the selected Map as an argument. This way only one play-state is needed, since it can adapt to the chosen map.

2.1.3. Encapsulation

The application will avoid globally accessible variables and methods to make the interface for programming easier to understand, expand and change. Using encapsulation will make it easier if the project were to expand with new people working on it, it is easier to get a grip of a program that is encapsulated.

Encapsulation also prevents modification of data that should not be manipulated from outside the program.

2.1.4. Position related calculations

Since the game requires a smooth frame rate both speed and reliability is important. In the application the methods and implementations should be adapted to prioritize speed.

For example for calculating the angle between two points tan should be used instead of cos or sine since both cos and sine requires the use of the hypotenuse for a complete answer. The reason the hypotenuse calculation is slow is because it uses a square root calculation. Generally all implementations will avoid using square root calculations because no relatively efficient square root calculating method has been found.

The program should also aim to implement methods that ask for distance to the power of two, since that also avoids a square root calculation.

The java math library will be avoided since it prioritizes precision over speed. For example the method for calculating angles uses a mathematical formula to calculate and approximate an answer with 8 decimals precision. It is a lot faster to find an approximation in a lookup table, at the cost of precision.

2.1.5. Image cache

A speed improving technique that should be used for images and fonts is cache. The Image class for representing images should contains a static hashmap that is used to save and lookup textures once they've been loaded into memory. This means that if the program tries to create a texture that already exists, it will instead be given the reference to the already existing resource. The cache for fonts works the same way.

Since all creeps, towers and projectiles use the same image resource, and this image is never manipulated, the image data is stored as a static variable in the object class, which also means that the image never has to be disposed.

2.1.6 The library libGDX

The library libGDX will be used for the project as it is a library for game developing that simplifies the process and also uses OpenGL ES 2.0 for all rendering of graphics, so the graphics card on the unit the application is run on is used efficiently (libGDX, 2015). This library will handle the updating logic of the application, the logic that makes the application update 60 times per second.

2.1.7 Services to lower dependencies

To lower dependency on external sources service classes should be written for most or all of external classes used. The following services are needed:

- Image, a service for the class Texture from libGDX
- ProximityAudio, a services for playing sound
- ProximitySound, a services for creating a sound
- ProximityBatch, a services for rendering images, fonts, etc.
- ProximityShapeRenderer, a service for rendering shapes
- ProximityFont, a service for creating Fonts

2.1.8 Garbage collecting

In the library libGDX that will be used for this project, several supplied classes directly communicate with hardware such as the graphics card (libGDX, 2015). These classes implement logic that requires manual handling of object disposal from RAM. Because of this the services that have been written for these classes need to implement an interface ProximityDisposable. All classes that implement ProximityDisposable should be added to a DisposableCollector on creation. When screens change in the game (for example from the game screen to the main menu) the DisposableCollector will dispose of all its objects and get cleared.

The rest of the objects should be handled by normal java GC (Garbage collection). However it was detected during developing that the objects did not get disposed correctly, because of this manual calls to the garbage collector will be made in appropriate places in the application instead, where CPU performance is not vital (as the garbage collector temporarily takes up a lot of CPU) (Goetz, 2004).

2.2. Software decomposition

2.2.1. General

The application is mainly divided into the following packages, see figure 1. See appendix 1 for a more thorough display of the packages and how they depend on each other.

- Models, where the data is stored, for example Creeps and Spells
- Viewers, where the data is rendered to the screen
- Utilities, helper classes that are very general and can be used in a lot of different scenarios
- Controllers, where input is handled and commands are given to the model classes

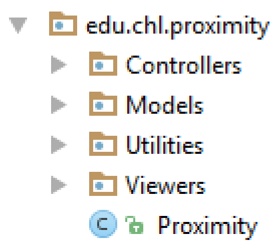
```
edu.chl.proximity
    Controllers
    Models
    Utilities
    Viewers
    Proximity
```

Figure 1: Image of how the packages are divided in Proximity

2.2.2. Decomposition into subsystems

No subsystems.

2.2.3. Layering

Se appendix 1, figure 1-5 for an analysis of the layering in the most important packages. The application has a package for service classes for dependencies, the rest of the model can be seen in the appendix. The project application is not build to communicate with other applications.

2.2.4. Dependency analysis

See appendix 1, figure 1-5 for an analysis of the dependencies in the most important packages. There are no circle dependencies in the application.

2.3. Concurrency issues

No thread concurrency issues, the application is single threaded. The application can run into errors if two instances are open at the same time, and both try to accesses the same save-file at once. This issue will be considered very rare and unimportant.

2.4. Persistent data management

The application data is stored as a data file, containing a Hashmap object typed as <String, Double>. The object can read and write to different numbered files, which enables handling multiple save slots for different people in the future (the application does not currently support multiple save files). The save file is used to store:

- Experience, the amount of experience the player has
- Level, the level the player has
- Waves, how many waves the player has defeated on each map

2.5. Access control and security

NA

2.6. Boundary conditions

NA. Application launched and exited as normal desktop applications.

3. References

Model-View-Controller. (2013). In iOS Developer Library.
Gotten 2015-05-06 from
https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

Basher, K. (2013). MVC Patterns (Active and Passive Model) and its implementation using ASP.NET Web forms.
Gotten from http://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-its

The role of Performance. (2011). Uppsala universitet.
Gotten from
http://www.it.uu.se/edu/course/homepage/dark/ht11/dark6-performance.pdf

libGDX. (2015). Goals and Features.
Hämtad från http://libGDX.badlogicgames.com/features.html

Goetz, B. (2004). Java theory and practice: Garbage collection and performance.
Hämtad från http://www.ibm.com/developerworks/library/j-jtp01274/index.html

APPENDIX

1. Dependency and layering analysis

Below you find images from STAN representing the structure of the program for the biggest and most important packages.

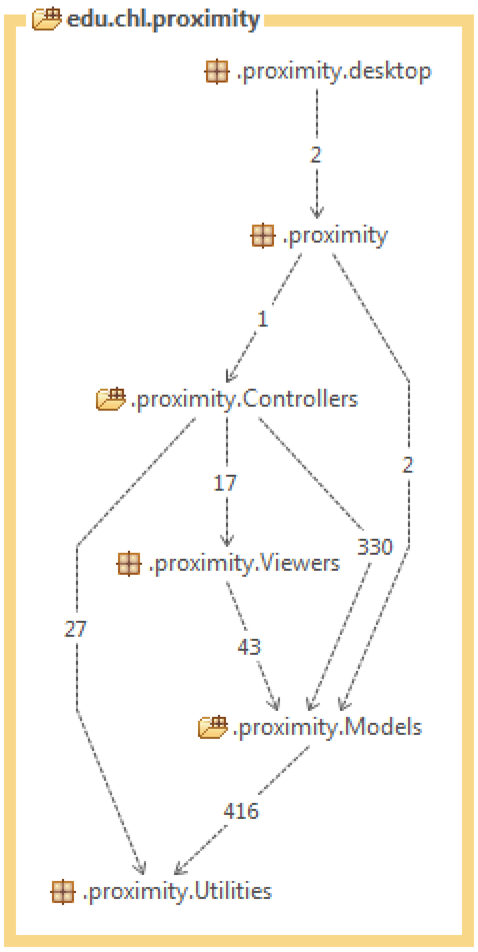

Figure 1: The overall structure

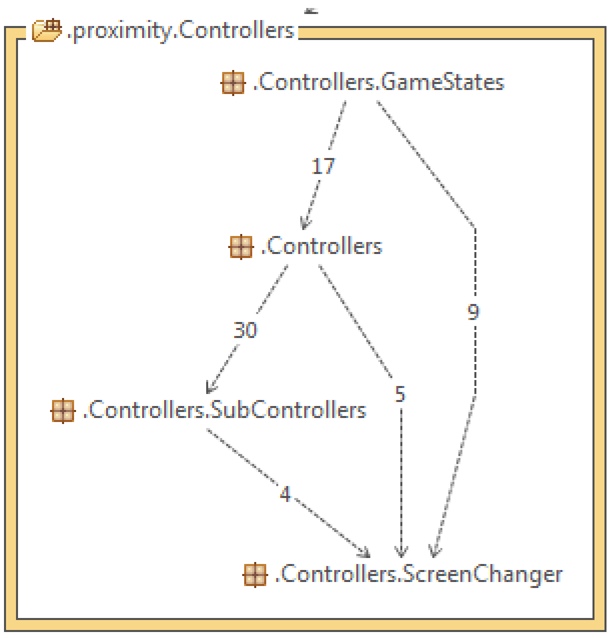

Figure 2: The structure in the Controller package

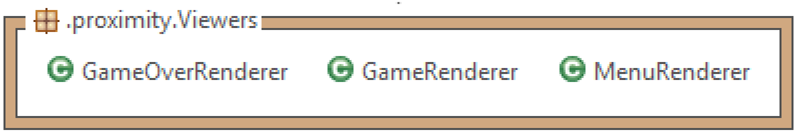

Figure 3: The structure in the Viewers package

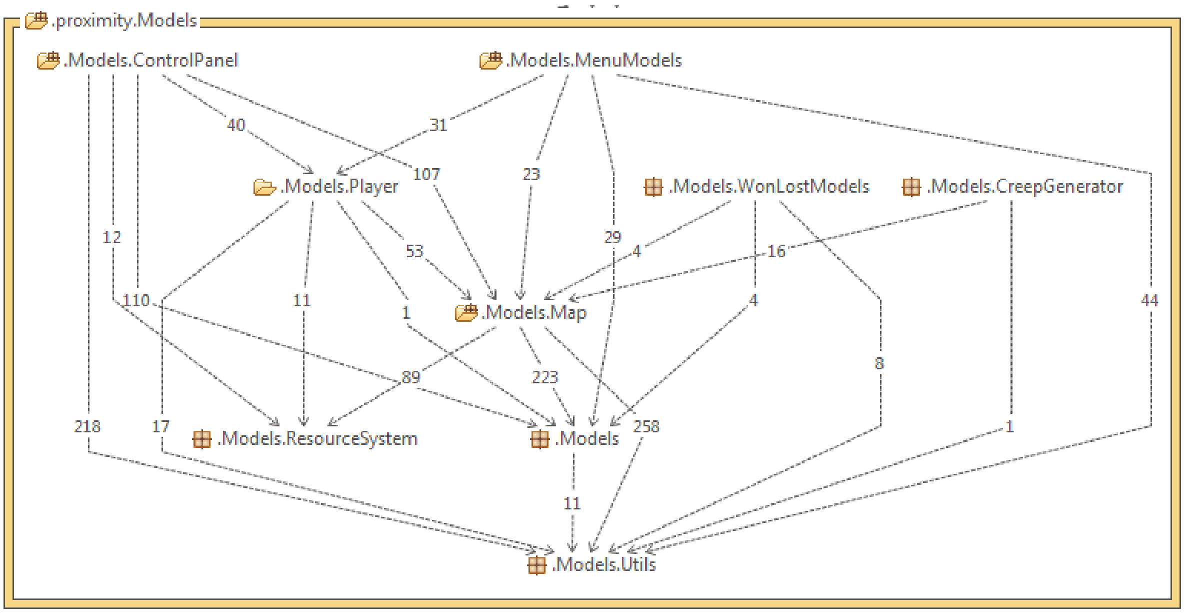

Figure 4: The structure in the Models package

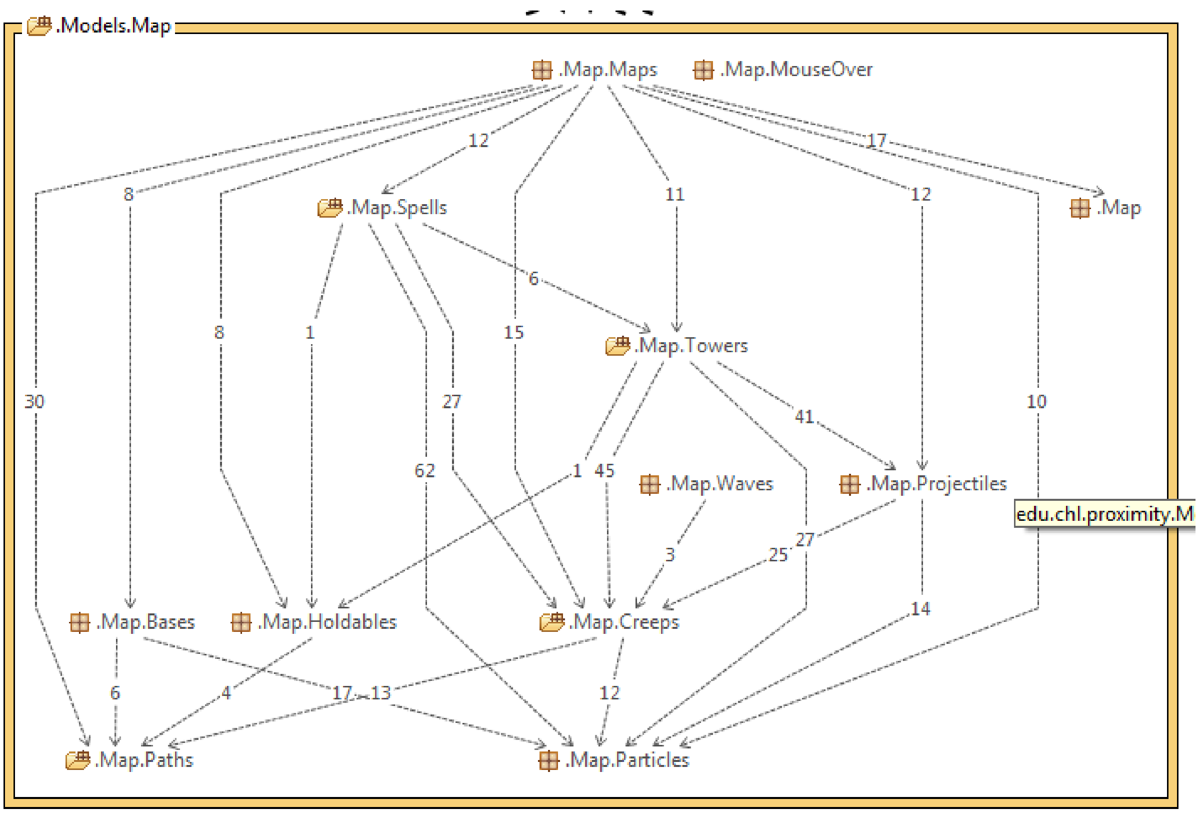

Figure 5: The structure of the package Map in the Models package