System Design Document

Johan Blomberg
johblomb@student.chalmers.se

Gustav Grännsjö
grannsjo@student.chalmers.se

Jonatan Gustafsson
jongust@student.chalmers.se

Robert Palm
palmro@student.chalmers.se

Version 9.
28/5 2016
This version overrides all previous versions.

Contents

1 Introduction

1.1 Design goals

The application should conform to the MVC design pattern where the View should be detachable as well as modular and the model should be static. The Controller will be a bit of both with a static part directly interacting with the model. Testability on the model is a means to secure functionality. For usability see RAD.

1.2 Definitions, acronyms and abbreviations

- GUI – Graphical User Interface, this is what the user sees when they runs the application.
- Java – The object oriented programming language used by the application. Java is platform independent.
- JRE – Java Runtime Environment, needed to run Java applications.
- MVC – Model–View–Controller, a design pattern which separates the aforementioned parts of an application.
- JUnit – A testing framework for Java.
- Gradle – A build automation system.
- Git – A version control system used for collaboration when coding.
- Turn – The period of time in which the computer and player performs their available actions before passing control to the other.
- Unit – A character controlled by either the player or the computer. A unit is capable of moving and attacking. The game ends when the player or computer is out of usable units.
- Stats – Statistics. The health, attack, speed/movement, etc. values of a unit.
- Move, Movement – The action of a unit that change the position of a unit on the game board.
- Weapons – Units can have different types of weapons, which defines how well they perform in combat against other units.
- Game board – The board the game is played upon. Moving outside the boundaries of the board is impossible. Also referred to as a “map”.
- Tiles – The game board consists of a rectangle grid of tiles.
- Terrain – All tiles have a certain kind of terrain, each of which has a different impact on combat and movement.

- Combat – When two units fight. They remove their respective attack from the other unit's health.
- Cursor – The cursor is the visual representation of where the action marker is.
- Marked – The tile where the cursor is.
- Selected – When a unit is marked and the action button is pressed, it becomes the selected unit.
- Move Buttons – Up, down, left and right. Moves the cursor in the cardinal directions and navigates menus.
- Action Button – Selects and confirms actions/units.
- Abort Button – Deselects and cancel actions/units.
- Info Button – Show detailed information about the marked and/or selected unit/tile.
- Map – A specific type of game board.

2 System design

2.1 Overview

The application will operate on an MVC design where the controller tells both model and view when to update. The view will have and use the ability to read data from the model and will in some cases report to the controller when certain time based actions has been taken.

2.1.1 Aggregates

Whenever another part of the program wants to tell the model what to do or needs to know something about the model it will go by the aggregate class GameModel. This class will take care of any action sent into it by itself, an outside force does not need to know what the model looks like. Data should be readable from its current State, but not directly changeable.

2.1.2 States

To prevent the aggregate root class from being way too big and doing too much, a lot of it is split up into several states which handle the input from the controller and contain the game logic. These states are reachable from the outside, but the states themselves know when to switch states, no direct changes should be made from the outside except for sending action commands. This is to make the application easily detachable from any view and controller modules.

2.1.3 Core Class Interactions

The GameModel aggregate root has access to a list of all units as well as the game board. It also holds all the game states, one of which is active at a time.

The game board handles all the tiles in the game. And each tile can in turn have one or no units standing on them. Units do not know on what tile they stand to counteract a double dependency. So if you want to know where a certain unit is standing or move it anywhere, you will need a reference to the game board. This is why most of the game logic is placed in the states, since they are highest in the chain of command and thus have access to everything. This makes the model package completely separate from both the controller and the view.

2.1.4 Deeper model structure

To assist the game's logic there are some utilities, the most important of which are combat rules and path finding. These are used by the different parts of the game model as necessary.

Because the model function radically different depending on the situation it contains a assembly of states, these states are an extension of the models logic and should be viewed as such, some are gateways to other states and other communicate with both the board and the unit on it.

2.1.5 Event Paths

The user input can be collected in many ways but LibGDX is used. The call chain is LibGDX → controller class → GameModel aggregate root → current state. LibGDX will tell the view to render the screen every time it sends an action to the controller (60 times per second by default). The view will then ask the aggregate root for the current state and then proceed to ask that state for the values it needs to know or the objects it wants to ask.

2.1.6 View

The view package is divided into several View classes, each responsible for a certain part of the program's visuals. Examples include classes for drawing the board, main menu and status screens. Each of these classes has a render method, which asks the GameModel for the currently active state. If the current state is the right one, the class draws what it needs to for that frame. The ViewHandler class aggregates an instance of each of these classes and calls their respective render methods inside its own render method. Thus, the rest of the program only needs to call upon ViewHandler's render method to draw every graphical aspect.

2.2 Software decomposition

2.2.1 General

The application should be composed into the following package structure Package diagram, and explanation. For each package an UML class diagram in appendix

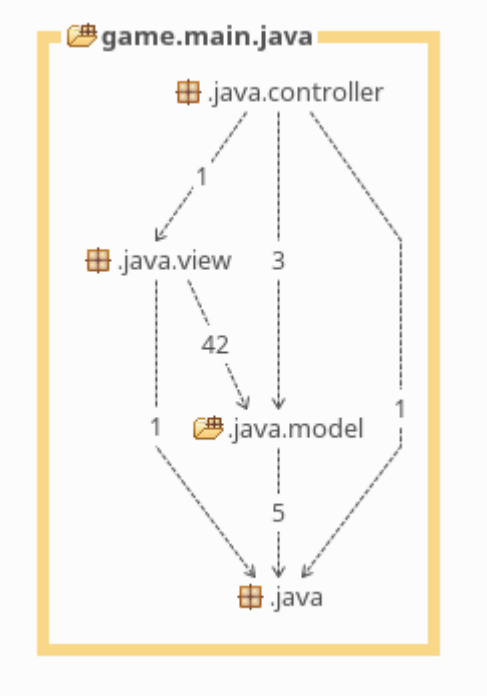
2.2.2 Decomposition into subsystems

N/A

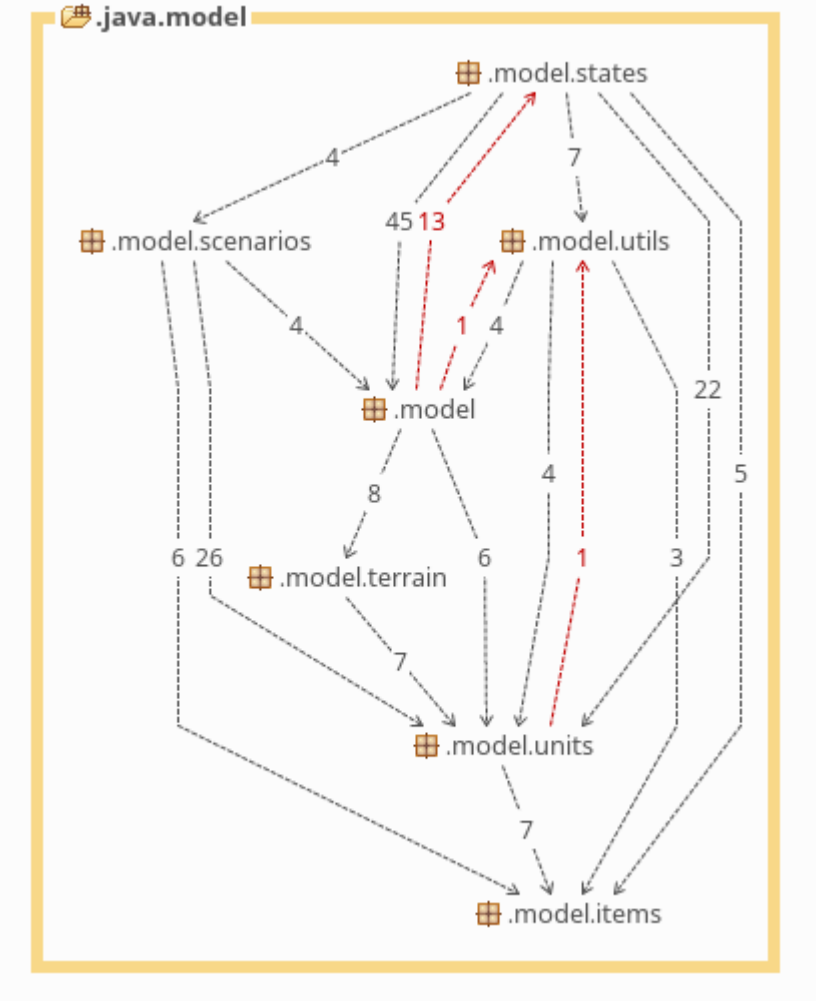
2.2.3 Layering

N/A

2.2.4 Dependency analysis



Dependencies between the top level packages. Everything looks good.



In the model package there exists some double dependencies between internal packages. The circular dependencies between .model and .model.states exist because the states are game logic parts of the game model that have been separated from the main class as to not have an excessive number of lines of code in one single class. The dependencies of .model.utils refers to different and totally separate utility classes and are thus not circular between classes, only packages. This minor problem could be solved by a reorganisation of the inner model packages

2.3 Concurrency issues

N/A

2.4 Persistent data management

As it is currently, no data persists between sessions. This should easily be implemented in the future.

2.5 Access control and security

N/A

2.6 Boundary conditions

N/A. Application will be closeable like a normal desktop application.

References

Appendix

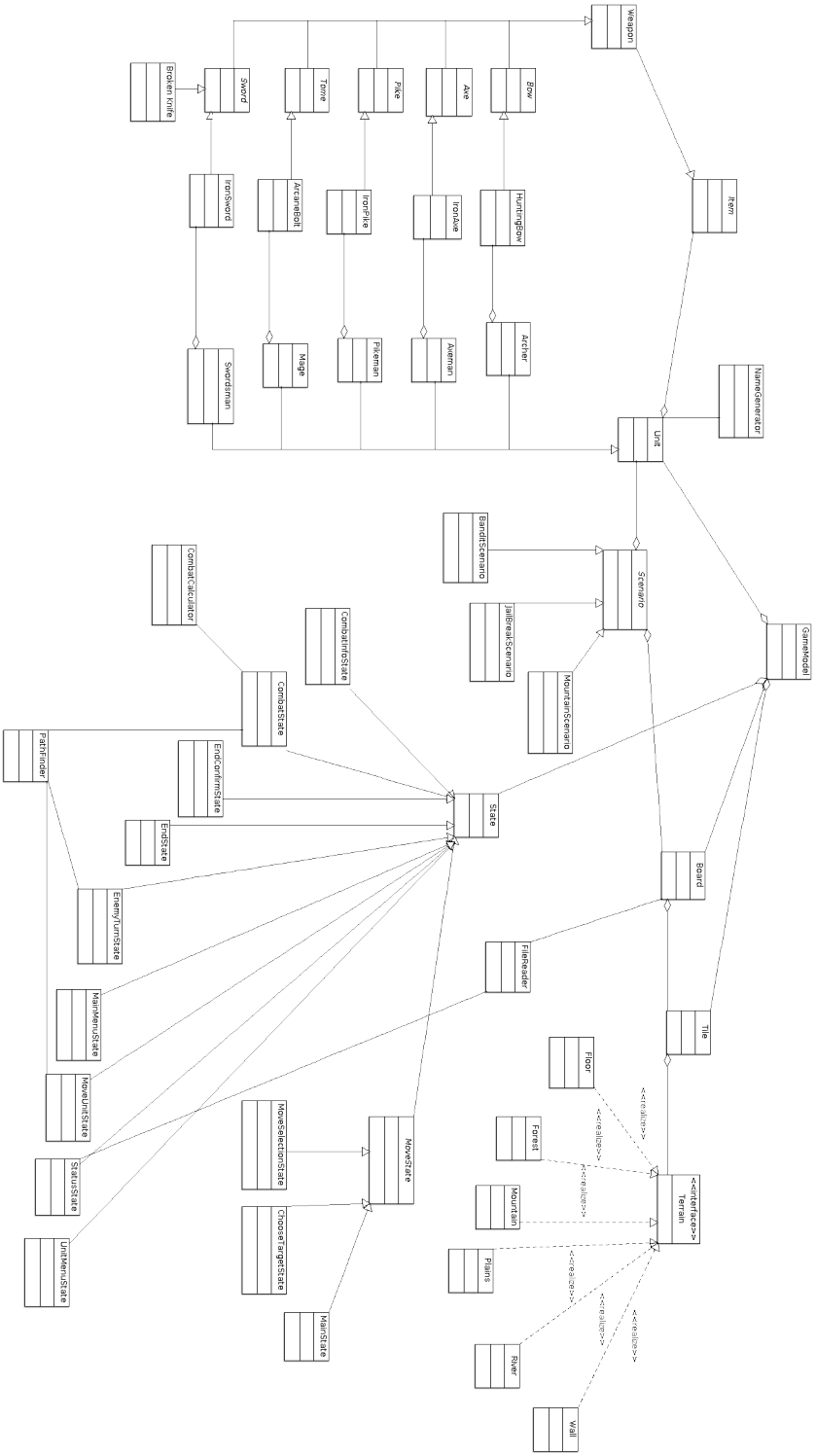


Figure 1: Model UML

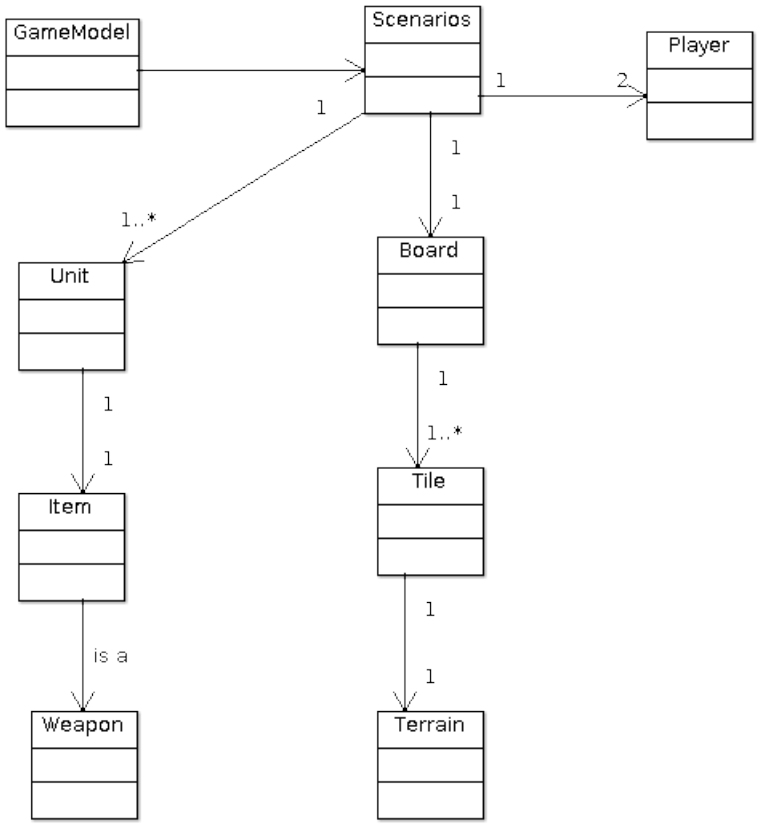


Figure 2: Design Model