# System Design Document for the Romijam project

**Version**: 1.0

**Date:** 2012-05-21

**Author:** Sebastian d’Aubigné, Simon Nilsson, Fanny Malmek, Mike Phoohad

This version overrides all previous versions.

# Table of contents

# 1 Introduction

## 1.1 Design goals

- To create a stable and independent model, with the possibility to exchange the views without complications.
- Adapt the system for the possibility of multiple players simultaneously.
- Strive for as general class definitions as possible.

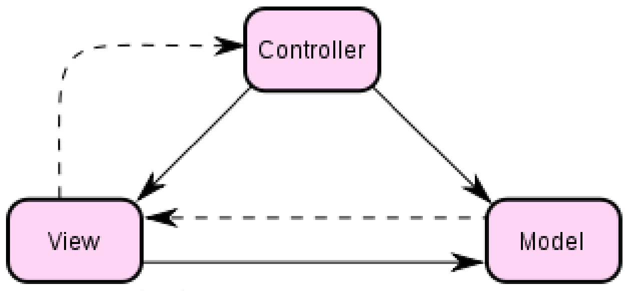
## 1.2 Definitions, acronyms and abbreviations

- GUI, graphical user interface.
- Java, platform independent programming language.
- JRE, the Java Runtime Environment. Additional software needed to run an Java application.
- 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.
- Hitbox, the area of an object where it can collide with other objects, not necessarily the area of the graphics.
- Tiled, a level editor that is used to add new levels in the game
- Action, an action is an order for the model to perform, e.g. Start game or Player 1 jump
- KeyEvent, a keyevent is sent to the controller when a key is pressed or released. This is what is used to control the game

# 2 System design

## 2.1 Overview

The game will be built with a MVC model. The model will be completely independent from the controller and view packages.



$Figure\ A: An\ illustration\ of\ the\ MVC-model$

### 2.1.1 World

The core element in the game model is what we call the world. It contains a list of all entities (see next title) which represents the world where the user of the game will see and explore with his character. It also handles interaction between entities their positions, their movement and the collisions that occur if two objects intersect.

### 2.1.2 Entities

The entities are the design of the game, they are blocks of which the playable levels of the game are made of. A general entity is made out a position, a hitbox(size of the entity) and a 2D-vector which describes the entity movement. All these are represented in the Entity Interface that each entity class depend on.

### 2.1.3 PlayerCharacter

The entity that first and foremost is responsible for the majority of interactions in the game is the PlayerCharacter. It is the entity that represents the user in the game's world, and is able to move around and interact with certain objects in the world. The game has support for having several instances of this entity in the same game.

The game's goal is to get the PlayerCharacter to the end of the level, and if it dies along the way you have to start from the beginning and try again.

### 2.1.4 Game Loop

Application updates will be done with one thread that once every 10 milliseconds will consecutive update model and render graphics. Key actions will be sent to the model as simple method calls, and the model executes the appropriate call inside.

### 2.1.5 KeyEvents

The way the game handles KeyEvents is when a key is pressed, that event is translated into an Action using the KeyBindings-class. It is here you set which key maps to what event. E.g. if you want to change the key "jump" is bound to, you should change the KeyBinding of "jump" to the key-constant you prefer. The upside of this is that you only have to change it in one place.

### 2.1.6 Action

The model uses Actions to translate what it is supposed to do. The main purpose of the controller(s) is(are) sending actions to the model and GameModel's main purpose is to translate the Actions and performing them.

### 2.1.7 States

For each view of the application there will be a separate "state" that will act as a controller for each view, and it will have the ability to call the main controller to switch between the different states. The main controller will only function as a state delegate, and each state will handle all keystrokes and contain model and view, called as update and render methods.

### 2.1.8 Levels

There are different levels in the game, which can be in single and multi-player mode. The levels are created in an external program called Tiled, and saved as tmx-files. The files are then placed in a certain map in the project and read by a class that creates the different objects of the levels depending on the object type, hitbox and position defined in the tmx-file. If the level has more than one player this is defined in the tmx-file as well.

## 2.2 Software decomposition

### 2.2.1 General

The application is composed into the following modules.

- controller
- main
- model
- model.entity
- model.level
- view
- view.entity
- sound

There is also a testing package, containing the following test modules.

- model
- model.entity
- model.entity.blocktest

Package diagram. For each package an UML class diagram in appendix

### 2.2.2 Layering

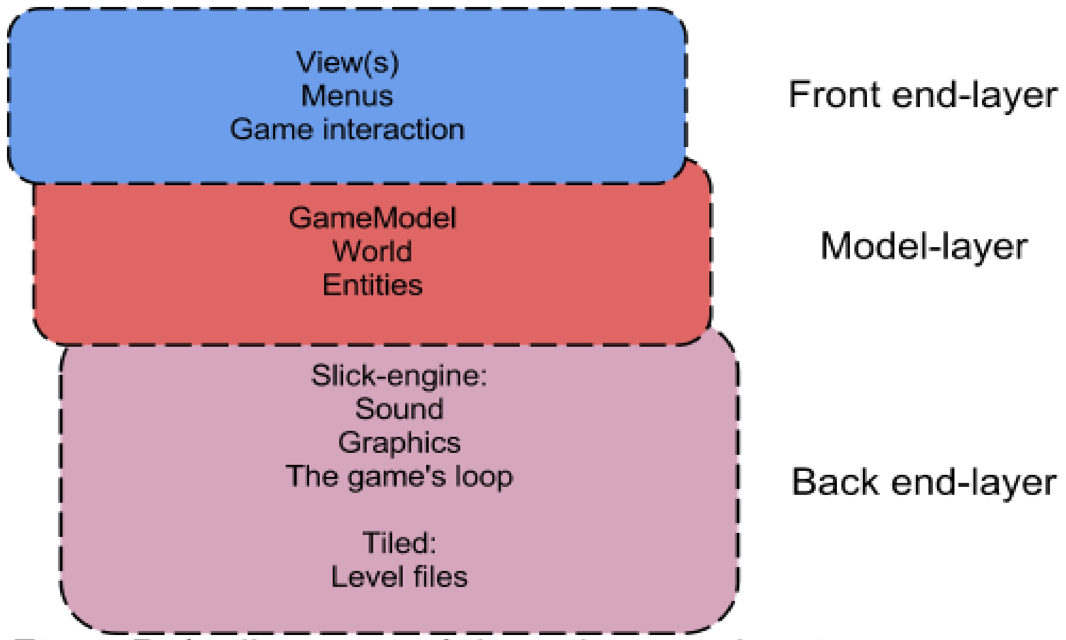


*Figure B: An illustration of the application's layering*

Most of what the user sees is in the front end-layer.
The actual game is contained in the model-layer.
The model-layer uses the back end-layer's parts to bring forward the game to the player in the front end-layer and also to actually run part of the game. For instance the levels are created separately and then deciphered into a format that the game's model understands. And the game's models then creates a world, which the graphics engine draws for the player to see.

### 2.2.3 Dependency analysis

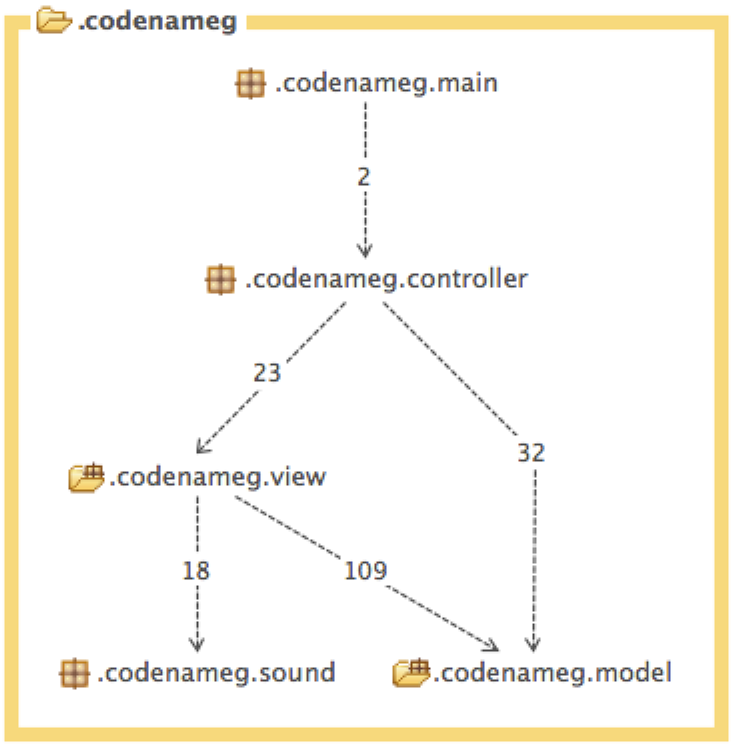


*Figure C: Illustration of the package dependency chain*

The dependencies of the packages of the application is non-circular as shown by Figure C. That the connections between packages is as few as possible and as non-circular as possible is tantamount to the modularity of the application and the cleanliness of the code (so that one thing is only done and that there is a clear dependency hierarchy).

As is also evident by Figure C we have made sure to keep to the MVC-model (see Figure A), as it is a very good method of keeping the code easily interchangeable and easy to understand.

## 2.3 Concurrency issues

The application is affected by two threads. One for executing update() and render() methods and one that sends key events. One concurrency issue that this may cause is if the same method is called from these two threads. This, however, have been insured that it will not be any problem as it looks now. There is a solution to make sure this will not happen, to temporarily save all key events in a list and not execute them until the next update() call. However, we have decided this as low priority.

## 2.4 Persistent data management

N/A. No files (such as progress) are currently saved, but its easily applicable.

## 2.5 Access control and security

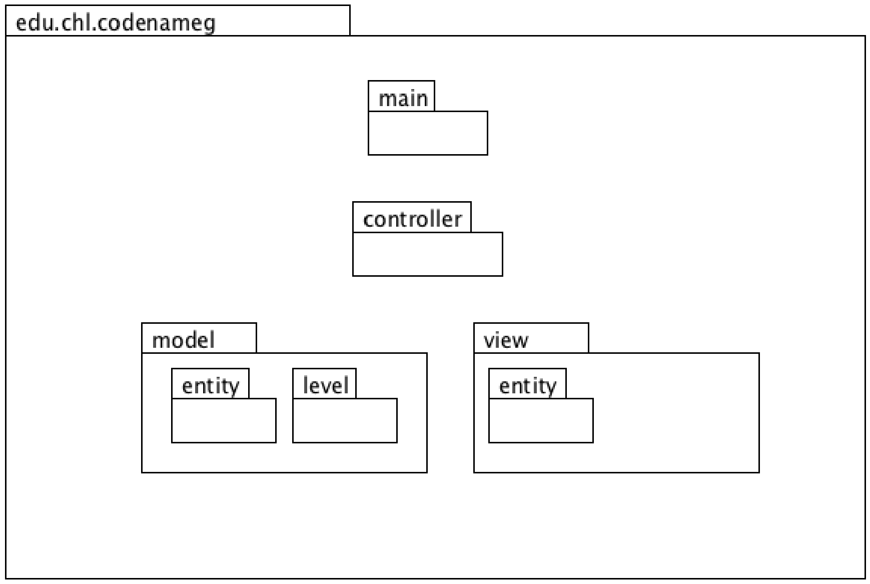
N/A. The application is open source, hence no point in having security or any forms of access controlling.

## 2.6 Boundary conditions

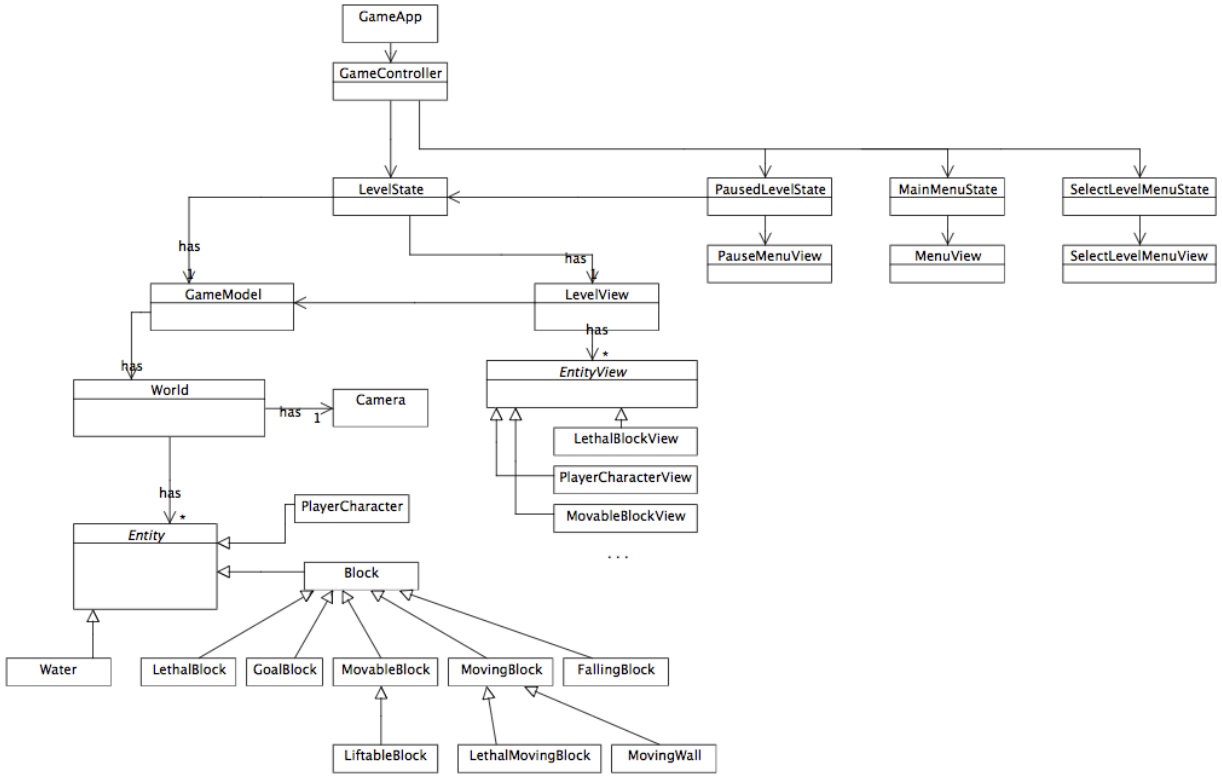
N/A Application launched and exited as normal desktop application (scripts).

# 3 References

# APPENDIX



*Package diagram*



*Design model*