System Design Document (SDD)

Group 13

Ruins of Corrosa City(RoCC)

Version: Final

Date: 2015/5/31

Jacob Duvander, Joel Hultin, Jennifer Linder, Jenny Orell

Content

1 Introduction

1.1 Design goals

The design must allow for addition or changing of smaller content in the likes of characters, enemies and maps without the need to in any way alter the code, and with bigger content to be added with as little altering of the code as possible.

As much of the design as possible must be testable. The classes or modules that should be testable must be isolatable and have no couplings or dependencies to external or unisolatable classes, modules or libraries. The design must therefore have a solution for separating necessary external or otherwise untestable parts from any relevant class or module without changing what these classes or modules represent or implements.

The design must allow for solutions for easily changing between views representing different states of the game and corresponding handling of user interaction for each of the states.

The design must have a clear and well defined structure which is easy to overview. This structure and the design must be constructed to help making well defined and representing classes as well as modules with clear responsibilities.

1.2 Definitions, acronyms and abbreviations

MVC(Model-View- Controller)
A design pattern that separates classes in three categories model, view and controller that handles different parts of the project.

Tiled Map Editor
Tiled Map Editor is a separate program that is made for creating game maps.

GUI
Graphical User Interface

JBox2D
It's a 2D physic engine that is written in Java. It's made to simulate realistic movements in a word with forces.

LibGDX
LibGDX is a open-source library that is written for Java. It's made for game development and helps with handling textures, graphics and handeling of input etc.

HTML
HyperText Markup Language

main
The main-method is the method that runs when starting a Java application.

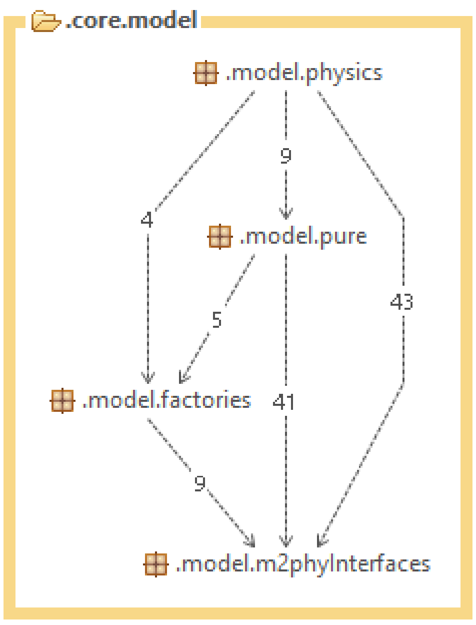
2 System design

2.1 Overview

This application will use a modified active MVC model, with additions of modules for file handling and utility.

2.1.1 The model structure

In the purpose of extracting the parts of the model which depends on from the remaining model a structure based on wrapper classes and the factory pattern. The model is composed by four modules representing interfaces, pure model classes, physics classes and factories.



The logical objects are then structured in a hierarchy for responsibility of other objects. Lowest in the hierarchy are objects representing actual things in the game, such as objects represented visually on the map. The next level in hierarchy represents more abstract objects with the purpose of keeping track of and handling the lowest objects, such as the player. Highest in the hierarchy is the main model object, the RoCCModel. This object in responsible for constructing all objects in the middle tier and most objects in the lower tier, being the main object the rest of the application interacts with and performing as much of the calculations as possible. As the main object for interaction in the application it naturally becomes responsible for distribution of all calls from other modules.

2.1.1.1 Interfaces
Each logical object is represented by one interface defining representing classes and their methods. Independent of how a class is implemented it shall be an accurate representation of the object described in its responding interface.

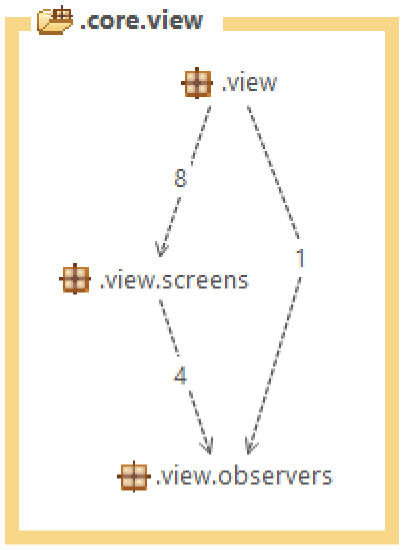
2.1.1.2 Model
Each interface is implemented by a pure model which class represents all logic and actions of the object not involving physics. Whenever an instance of another model object is created it must be done by a factory. These factories shall be set by a parameter in the class constructor. This makes sure physics classes can be used when wanted, but ignored when testing. Everything in these classes must be testable.

2.1.1.3 Physics Classes
The purpose of the physics classes is to extract and handle all instances of physics from the model classes as this is handled with the help of external libraries. Each class shall work as an extension of its corresponding model class while delegating as much as possible of its implementation to that class. Every physics class and its methods must follow the same description as its corresponding model class.

2.1.1.4 Factories
The factories exist to make it possible to create any of the logical objects without knowing how they are implemented. Each factory should be able to create instances of each object described in a common factory interface. For a single factory all instances created should be implemented in similar fashion and the factory should be a representation for that style of implementation.

2.1.2 View
All classes describing each view are located in the screens package. Each view implements LibGDX Screen interface. The purpose of this is for easy interaction between the views and the main class. Observers are used to handle the interaction between view

and controller and are located in the observers package inside view.



2.1.2.1 View manager

GameViewManager is the class used to handle the switching of views. GameViewManager creates all the different screens by using the class ViewFactory and then puts the views in a HashMap. The method setActiveView is called with the key to the requested screen when it needs to be changed. Then to actually change the view the setScreen method in ViewController, located in controller, is called with the active view from GameViewManager as input.

2.1.2.2 Screens

The screens describing the main menu uses Stage which functions as a container where all the graphical items are put. The play screens uses SpriteBatch instead of Stage which functions similar to Stage. The screen components are drawn through the render method included in the Screen interface.

2.1.3 Filehandling

The application handles three types of information, text-files, textures and maps.

2.1.3.1 Text-files

The properties of all characters and enemies as well as options are handled as text-files. The abstract class AbstractTextFileHandler deals with all writing and reading of these, and different types of properties uses their own loader-class which extends the abstract class. If a file is missing or can’t be read default values will be set for corresponding properties.

There are also text-files describing the different textures necessary for rendering a character, enemy or weapon. These are handled by the texture-handler.

2.1.3.2 Textures
Textures for a new character, enemy or weapon must follow the same pattern as textures for existing characters or weapons. Each character has a corresponding texture-folder containing textures covering all move states. These are loaded by the ViewTextureLoader and then given to the views sorted as HashMaps.

2.1.3.3 Maps
The maps used are created by Tiled Map Editor. These maps describe the environment of the game, which enemies, pickupable characters and other objects will exist and where. These maps are handled by the MapLoader class.

2.1.4 Controller
The controller handles all input and events in the application as well as constructing the application. The input handling is done using LibGDX' InputProssecor created as internal classes in MainController.

The application itself is started by the ViewController which extends the LibGDX abstract class Game. This class also function as the controller for the views in the application. Meanwhile all updates in the model are done in the MainController through the method SendUpdate.

2.1.5 Eventhandeling
The application uses four types events, collisions, deaths, game over and viewchanges.

2.1.5.1 Collision
The collisions in the game are handled by the class CollisionListener. CollisionListener listens to a world created by JBox2D for collisions between bodies. These represent picking up objects, taking damage, landing on the ground etc.

Handling the event is done by identifying the objects colliding and the performing the related task. This is done by using a set userData. As the collision events are created during an update period for the representation of the world no changes of the world can be made directly after a collision has happened. Therefore all objects due for addition or removal must be marked and handled later.

2.1.5.2 Deaths
Character and enemy deaths are handled by the classes DeathListener and DeathEvent. Whenever a death occurs a DeathEvent describing what died and how is created and sent to observing DeathListeners for handling.

2.1.5.3 Game Over

Whenever a game ends a game over call is made to a GameLossListener to tell the controller to correctly end the game.

2.1.5.4 View changes

Whenever the view changes or the state of the view changes a listener in the MainController is informed so that the rest of the game can be updated correspondingly. This mainly concerns the changing of input handler.

2.2 Software decomposition

2.2.1 General

This application dissolves into the following modules.

- rocc contains the whole project. It is dependent on the library LibGDX.
- desktop and html contains the main classes and other data needed to run on each platform.
- core, contains all of the code that is not dependant on a specific platform.
- m2phyInterface, contains interfaces describing all the model classes.
- model contains classes implementing the interfaces while containing no physics
- physics contains wrapper classes for each model class adding the physics-implementation for the representation.
- view, is the GUI for the application. view is part of the MVC model.
    - observers, contains the observer interfaces needed for notifying controller of changes in view.
    - screens, contains all the different views.
- controller, is for all the control classes. controller is part of the MVC model.
- utility, is for Enums and global variables.
- filehandlers, is for reading and writing files.
- observers, is for the observer interfaces used to inform of changes in observable classes.
- factories, is for creating model objects.

2.2.2 Layering

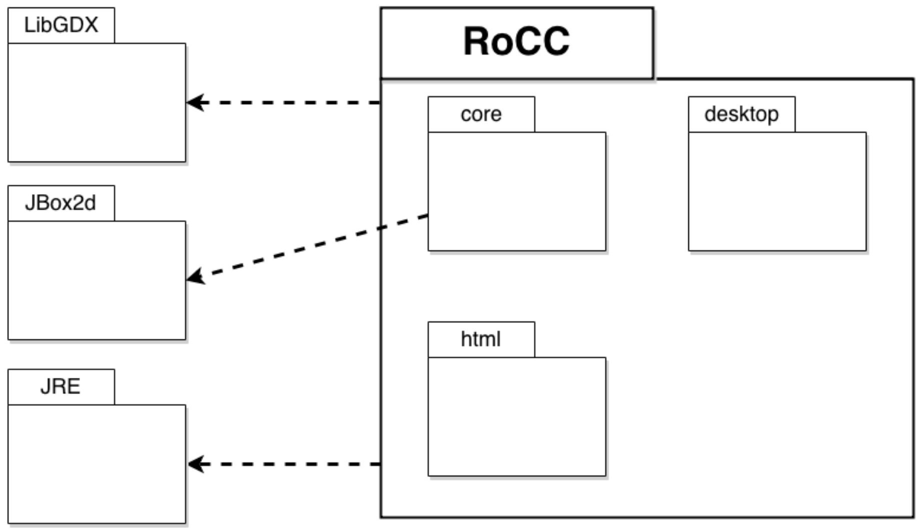


Figure: Project structure for the application with dependencies.
RoCC contains the whole project and is dependent on LibGDX. Core contains all the source code and is dependent on JBox2D. Desktop and HTML is where the main classes are located for each platform.

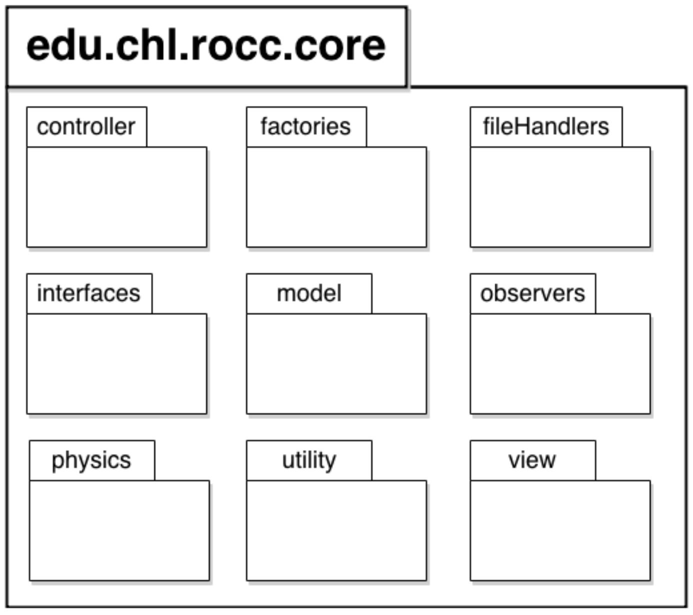


Figure: Package structure for core project.

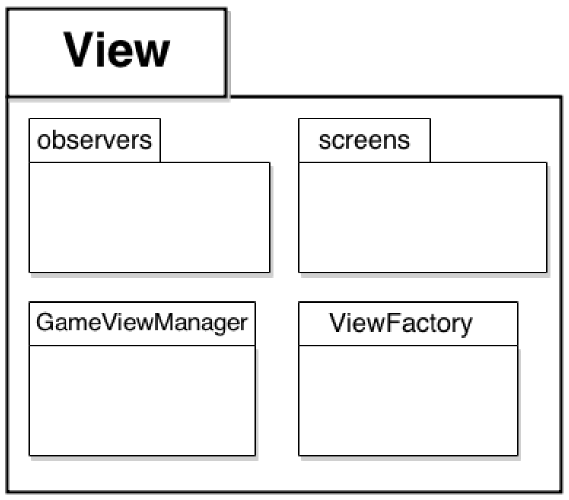


Figure: Package structure for view.

2.2.4 Dependency analysis

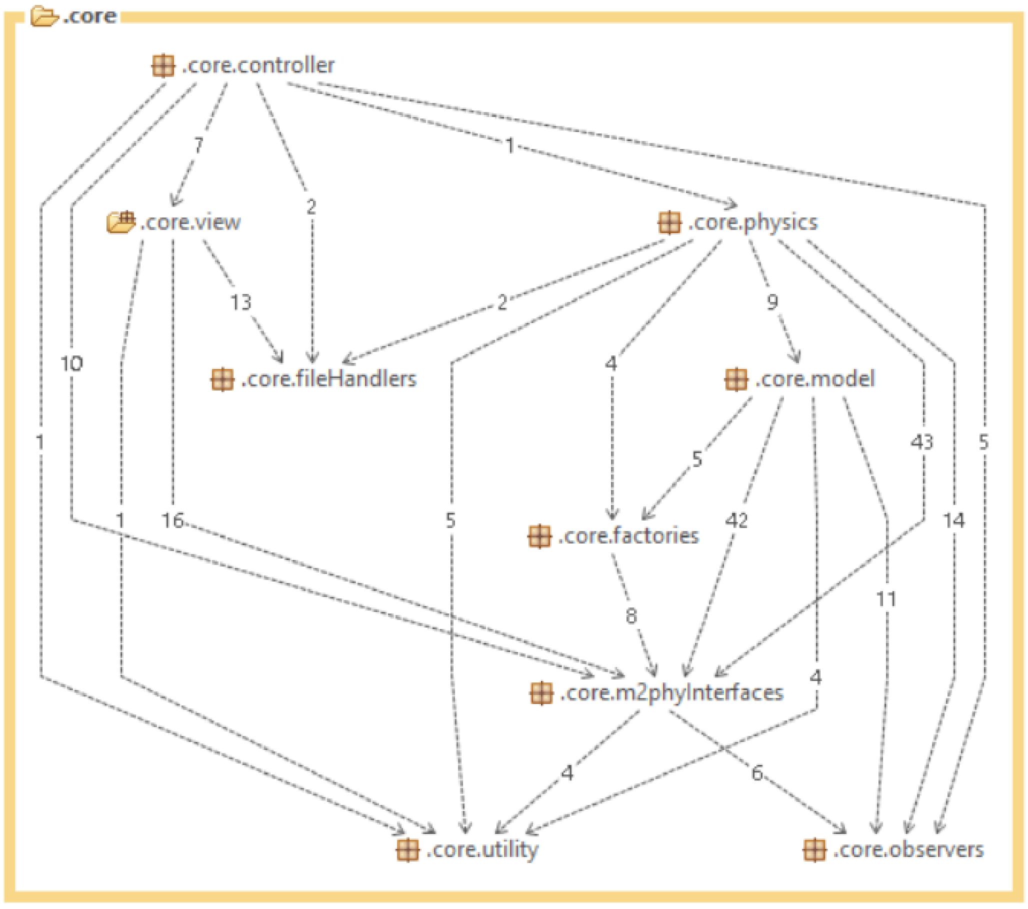


Figure: Dependencies in the structure.

Nothing in the structure is dependent on the controller, and the observer and utility package don't rely on anything.
In the model, every dependency goes downwards in the abstraction layers. As little as possible depend on the physics classes.

2.3 Concurrency issues

The application uses two threads, one for rendering and one for updating the model. Because of this, errors may occur if the threads are not handled correctly. By using the lists synchronized this issue is solved hence only one thread alone may access that list at a time.

2.4 Persistent data management

Text files are saved in the program's assets folder, and may be deleted from the folder without any impact on the program. If one or more lines of the text files are corrupted, the compilation will stop working since it is dependent on a specific structure of the text files.

Every character that is added must have a texture for all properties that is specified by the texture definitions.

2.5 Access control and security

NA

2.6 Boundary conditions

The application is only developed to be launched as a desktop application. However, the project has support for launchers for Android, HTML and iOS aswell, although they are not implemented.

The game is launched with a .jar-file and is shut down by closing the window or exiting from a menu.

3 References

APPENDIX