System Design Documents for MinMiner

TDA 367

Gabriel Wallin, Omar Oueidat, Erik Strid, Olof Enström

May 28, 2017

Version

Version: Final
Date: May 28, 2017
Author: Revised by Omar Oueidat, Olof Enström, Erik Strid, Gabriel Wallin

This version overrides all previous versions.

Contents

1 Introduction

1.1 Design goals

The main design is to create a program that is loosely coupled. The program should be easily expandable and adding features and complexity should be easy. Also, the design must be testable. The program will follow the MVC design pattern.

1.2 Definition, acronyms and abbreviations

- Miner - The player
- Hull - The miner's health, which decreases upon impact with another object
- Fuel - The gas which the miner uses to be able to move around
- LibGDX - The library used to create the game
- MVC - Model-View-Controller, a design pattern that organizes the program in a way that avoids high coupling and dependencies
- Drilling - The act of the miner moving into the ground and removing a ground object
- Gear - A collective term for the miner's different attributes. Currently Fuel and Hull.

2 System Architecture

2.1 Overview

The design will be structured around an MVC design model[1], with low coupling and loose connection between classes and packages.

2.2 Listeners

The view will heavily depend on listeners to update the playscreen. The model will send a signal to the view that tells it to update accordingly and send necessary data that the view needs. Having listeners was chosen to decrease the dependencies and lower the coupling between the view and model only letting the view access the information that the model wants it to access.

2.3 Game Screens

The switching between the different views of the game will be handled exclusively with the help of screens. The screen interface will assist the application so that it efficiently can switch between different views. Having the Start screen, Play screen and Game-over screen separated from each other and incorporated into Screen objects will provide multiple benefits. Coupling will be kept at a low level since the screens will rarely have to communicate with each other. As a result, the Single responsibility principle will also be supported considering the different screens will have separate objectives.

2.4 LibGDX

The program will be constructed around the LibGDX library, which is a cross-platform game and visualization development framework.

LibGDX will provide the tools to easily construct the game.

LibGDX lets you go as low-level as you want, giving you direct access to file systems, input devices, audio devices and OpenGL via a unified OpenGL ES 2.0 and 3.0 interface

2.5 UML diagrams

As seen in Figure 2 (See: appendix), the model-view-controller is relatively achieved, with the only issue being the little connection between the model and controller.

Looking into the model package, the head logic where everything is stored is in GameModel(See: Figure 3). The GameModel encapsulates the GameWorld and the MinerModel which in turn work together to create the world for the

view to paint out. The package is loosely coupled with little to no dependencies from the view and controller.

The controller package is lacking. It is too dependent on view due to the view creating the button. The classes in the controller package severely lack(See: Figure 4) connection to each other and are only used for simple tasks. It could be argued that the classes are following the single responsibility principle. However a better connection and cohesion with the other classes could lead into better flow through the program. The two classes with dependencies are utilizing the singleton design pattern[2].

3 Software decomposition

3.1 General

The game application will consist of five different packages, Model, View, Controller, EventHandlers and Tools. Packages, Model, View, Controller will adapt the MVC design pattern. The EventHandler package is all of the listeners which was created in order to have a more simplified yet functional eventbus type, without adding an extra external library while meeting the program's needs.

The Tools package is the package where the necessary tools are for creating the world and making it a functional playground. Tools also has the world listener which listens if there are any object collisions in the world.

The model handles all the game logic and contains the game's data. The model tries to not be dependant on any libraries, in an attempt to follow OCP (Open Closed Principle). However, the MinerModel class has dependency upon the LibGDX library in a sense that it takes in a world to create the miner. This is a major obstacle when trying to instantiate a MinerModel for testing its methods.

The controller package is the main input-handler for the program, including all the controllers that the program requires to be able to interact with the user. The package is hurt by dependency from the view package as every controller is required to be built by the view before being assigned as a clickable button or something to interact with.

The view package is where all framework is processed (apart from model initiation), everything renders here and dictates how the application is painted at a certain moment. The biggest class that the package holds is the PlayScreen class which is the main screen the application will be running. The other are complementary view classes or have a different usage area.

3.2 Abstraction layers

For abstraction layers, see appendix for all the UML and their packages UML

3.3 Dependency analysis

As seen in Figure 1, there is a clear dependency issue between the controller and the view package. This is due to the reason that every controller contains its respective view for easy access the shared button. This is a result of the fact that the view and controller of a certain button both must be connected to the same button object. The issue lays in having an unclear view package where there is an abundance of variables and methods which could be abstracted into smaller classes and placed in different packages that would complement the

MVC-oriented design.

The main reason the dependency issue lays, is the external library, LibGDX, that is used to render our main framework.A solution to this problem could be using the idea of the library to create an own type of rendering module using LibGDXs functions as a complement rather than relying on them.

However, the main goal of the program was to relieve the model of anything unnecessary so it is solely for handling data and logic and the final version of the program proves that there is little to no dependency issues between the model and other modules. This allows the model to be reliably tested which is one of the staples of MVC. In the current version of the program, the model is clean. For example, the class GameModel does not import any code from LibGDX, and only handles game logic.

As in the latest version of the program, the issue is not resolved, this is could be fixed in a later version, but due to the deadline and time remaining the decision was made to not ruin the application trying to fix the issue last minute.

3.4 Concurrency issues

The application will be single-threaded and thus will not run into concurrency issues. Also, the application does not use any form of database which will prevent error from occurring when two instances try to load the same files.

3.5 Persistent data management

The game will not save any progress made during the game, nor will there be a save option to keep playing with the same progress when the game has been terminated. All in-game data will be removed.

3.6 Access control and security

NA, application will we launched and exited as a normal application.

References

[1] G. Krasner and S. Pope., *A description of the model-view-controller user interface paradigm in the smalltalk-80 system.* ParcPlace Systems, 1988.

[2] J. Tidwell, *Designing Interfaces.* O'Reilly Media, 2011.

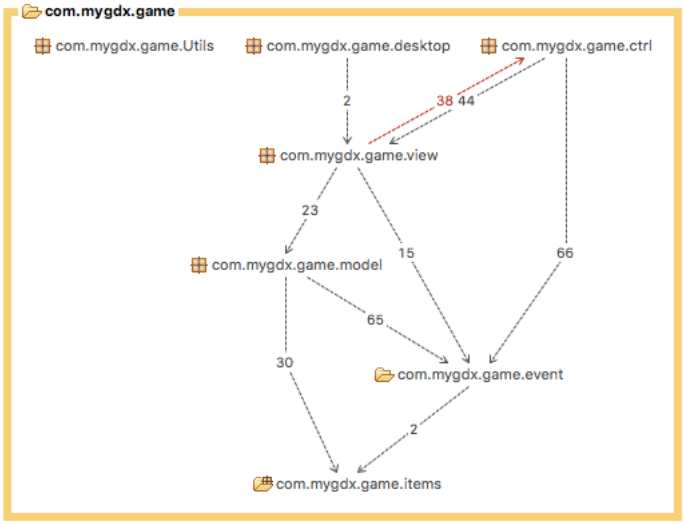

Figure 1: A package analysis generated in STAN

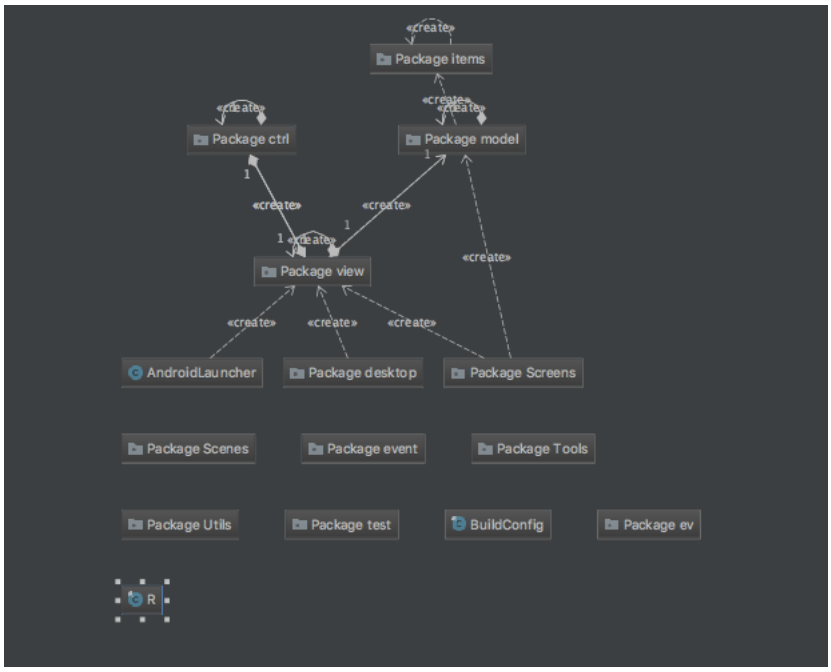

Figure 2: The UML diagram for the whole application

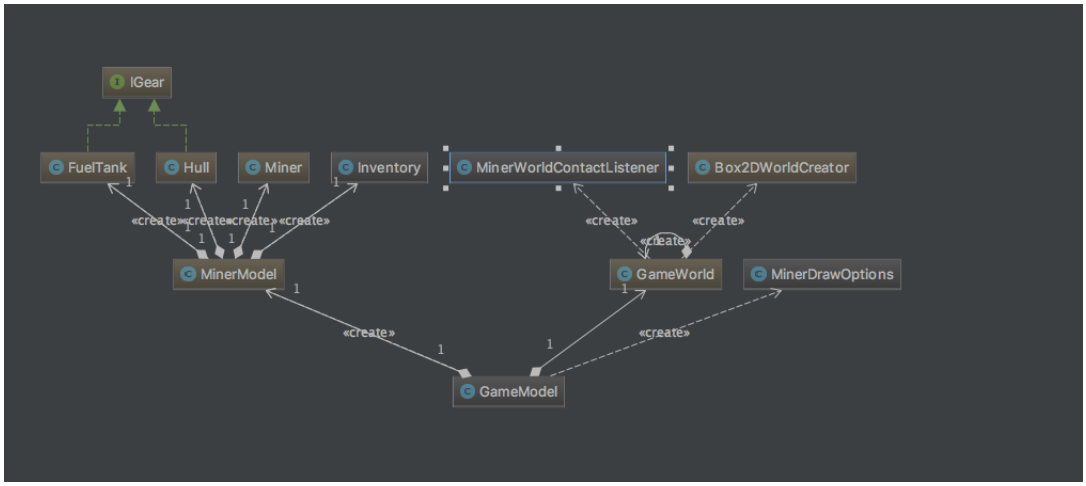

Figure 3: The UML diagram for the Model package

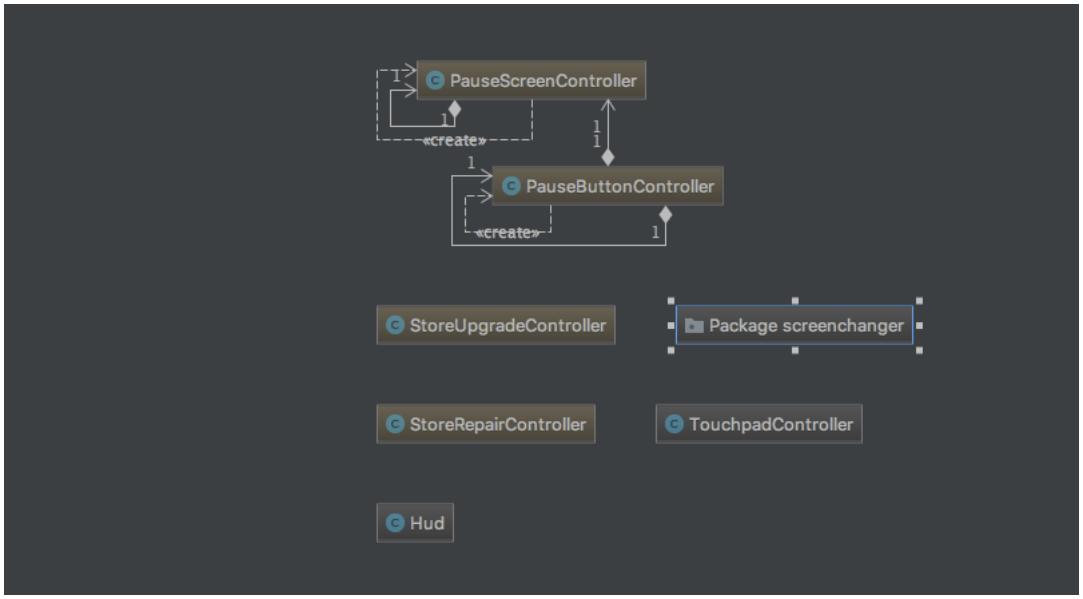

Figure 4: The UML diagram for the controller package