System design document for the Dash project

Version: 1.0

Date: 28-05-2017

Authors: Lucas Ruud, lucasr@student.chalmers.se, Erik Lundin, erilundi@student.chalmers.se, Niklas Baerveldt, nikbae@student.chalmers.se

This version overrides all previous versions.

1 Introduction

This document describes the design and implementation of the software application Dash. Dash is a 2D side scrolling platform game which implements a randomly designed level structure. The level is randomly generated according to a preset number of values the first time but the user can affect the generation by picking up powerups in the game which changes these values.

1.1 Design goals

Some design goals:

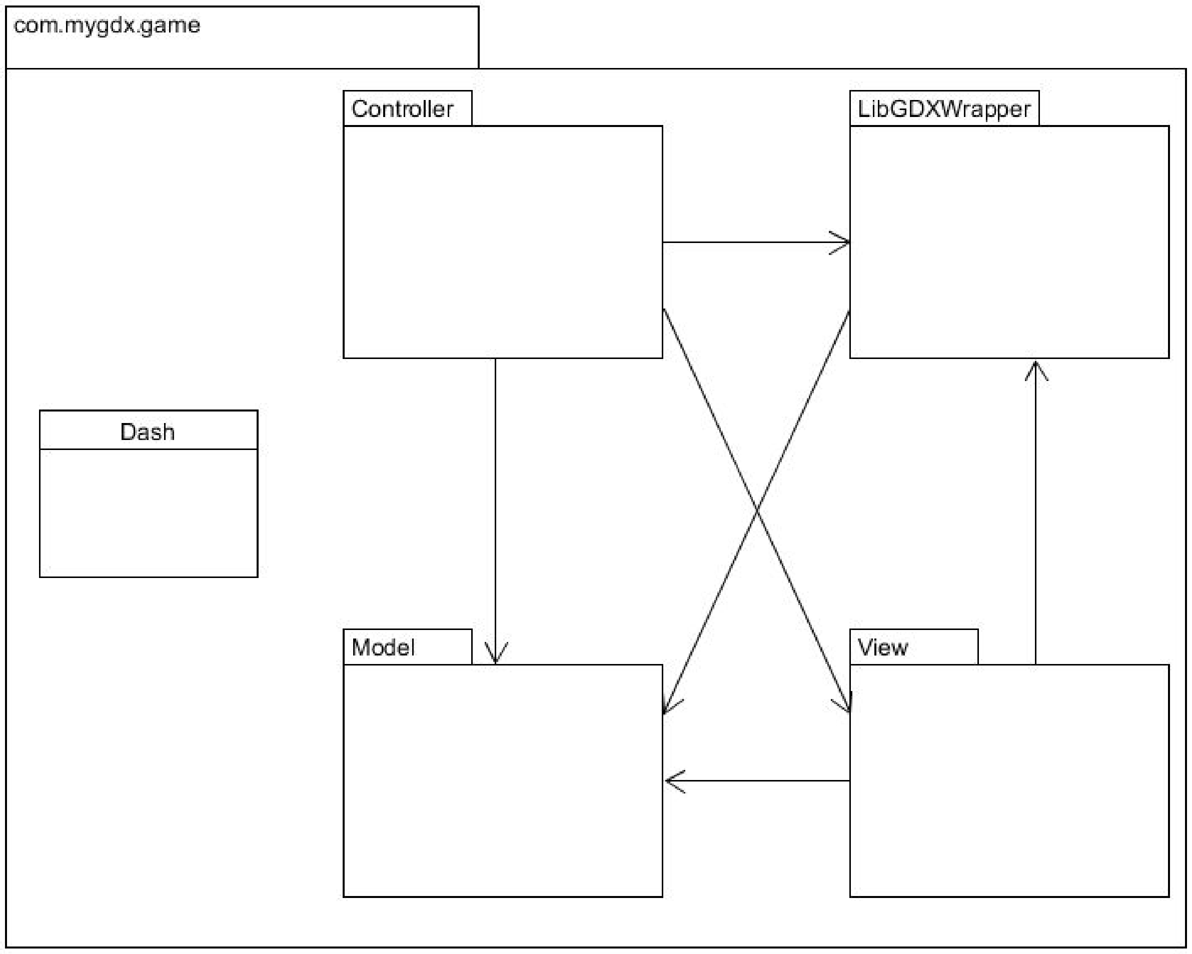
- The design should be testable, i.e. it should be possible to test individual parts on their own.
- The game should be open for extension, such as new types of enemies or map layouts.

1.2 Definitions, acronyms and abbreviations

Player: The entity controlled by the user
User: The person using the application
Health: A measurement of the player's life, if depleted the player dies.
Enemy: An AI-controlled entity within the game
Power-up: A game changing effect.
Obstacle: An object interfering with the players ability to continue
MVC: A way of structuring the program to separate different parts from each other.
Projectile: The player and enemies can both shoot these to try to kill one another.

2 System architecture

The application will use the MVC design pattern to separate the model, view and controller parts from each other.
The application will run on a single desktop.
The application is divided into the following top level packages (arrows indicate dependencies):

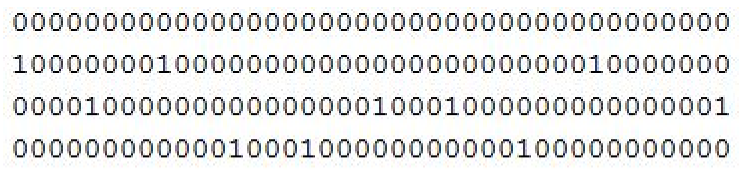


- Dash: This is the class used as a entrypoint for the application.
- Model: This is where the model of the application resides
- View: This is where the different graphical elements of the application is kept.
- Controller: This is the place for the different classes that controls what happens in the application.
- LibgdxWrapper: A package where all of the functionality from the libgdx library is stored.

2.1 General observations

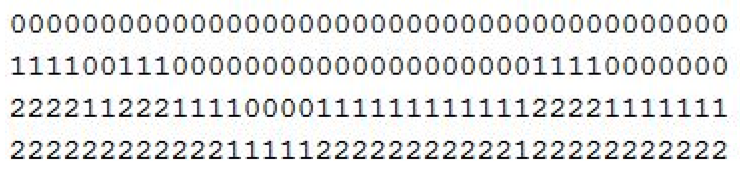
Generation algorithm

The Generator class inside the model is where the logical map is created. When called it creates an int 2D-array with preset width and height.



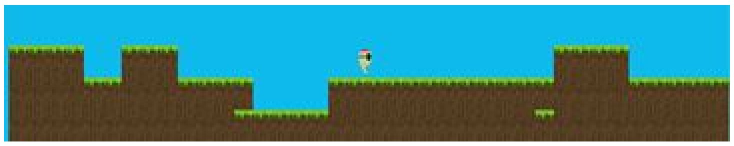
```
0000000000000000000000000000000000000000
1000000010000000000000000000000010000000
0000100000000000000010001000000000000001
0000000000001000100000000000100000000000
```

It manipulates the array, starting of with placing ‘1’s at random in a set of rows, in this case the bottom three, in the 2D-array with a number of columns, here set to two, between each ‘1’.



```
0000000000000000000000000000000000000000
1111001110000000000000000000111100000000
2222112221111000011111111111122221111111
2222222222221111122222222222221222222222
```

 After the base nodes are set a growFromPoints method is called, which starts of from one node, identifies where the next basenode is and starts to place out ‘1’s in between the two nodes in a way that forms a consecutive path of ‘1’s between them. Underneath the now continuous path of ‘1’s, ‘2’s are placed.



The ‘1’s are later used to determine where hitboxes should be placed in the world. The ‘2’s are currently not used, but could be used in the future.

Later platforms are added above the one-row as sequences of ‘3’s, and clouds are added as sequences of ‘4’s, the pictures above only illustrates the part of the matrix which models the ground.

MVC pattern

The program is structured after the MVC (Model, View, Controller) pattern. This is done to reduce the potential complexity of the program by reducing and managing the number of dependencies between classes and packages.
The model includes all of the logic and data of the program. Although the LibgdxWrapper package is also technically part of the model, we separate it from the “normal” model package. This is done to separate the LibGDX specific content from the rest of the model, so that we can test the code.
The view includes all of the screens and other graphical components that will be shown to the player.
The controller contains the various controller classes for different parts of the program.

State pattern

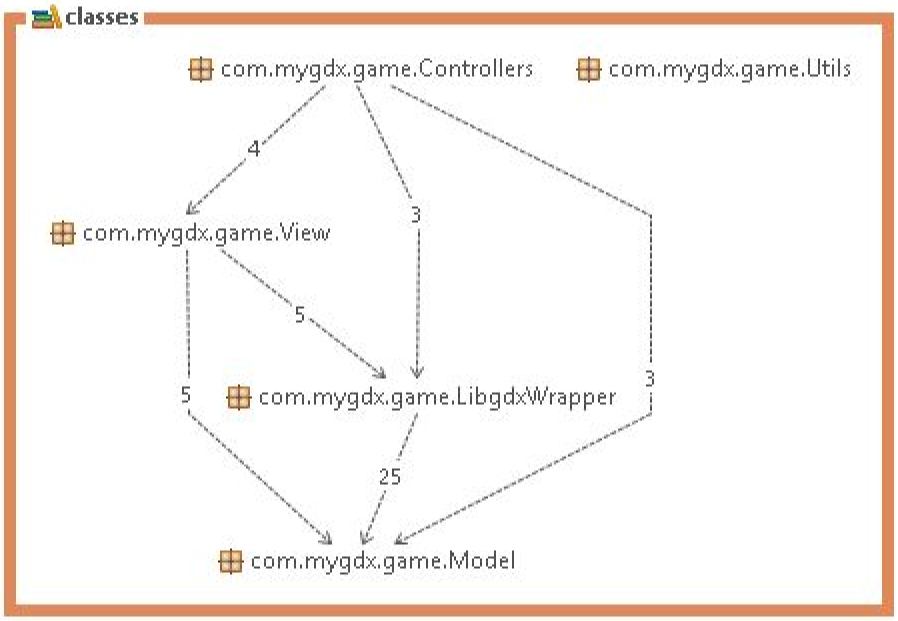
The EnemyBehavior and PowerUpModifier classes both implement the state pattern. This determines the way a powerup or enemy behaves when it is created. This pattern makes it easy to implement more modifiers for powerups and more behaviors for enemies, since you just have to add classes that dictates how these modifiers or behaviors should work.
The state pattern is used to increase the modularity of the program and in turn contribute to the design goal that the game should be open for extension.

Singleton pattern

The singleton pattern is implemented in the Generator class so that only one instance of a Generator exists at a time. This pattern is used because the Generator is only needed to generate one map in the game, so no more than one instance of this class is required at a time.

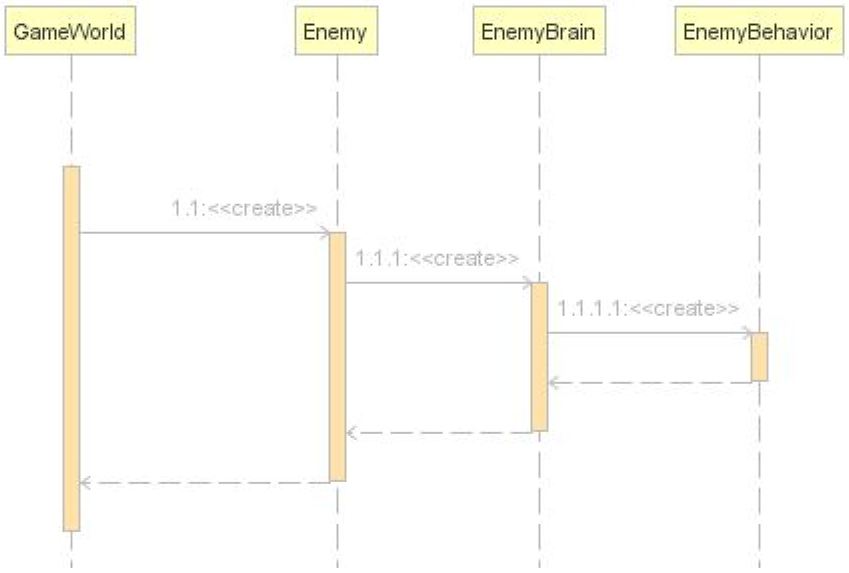
2.2 Diagrams

Dependency analysis



Sequence diagram

Sekvensdiagram för skapandet av enemies i GameWorld



2.3 Quality

Tests

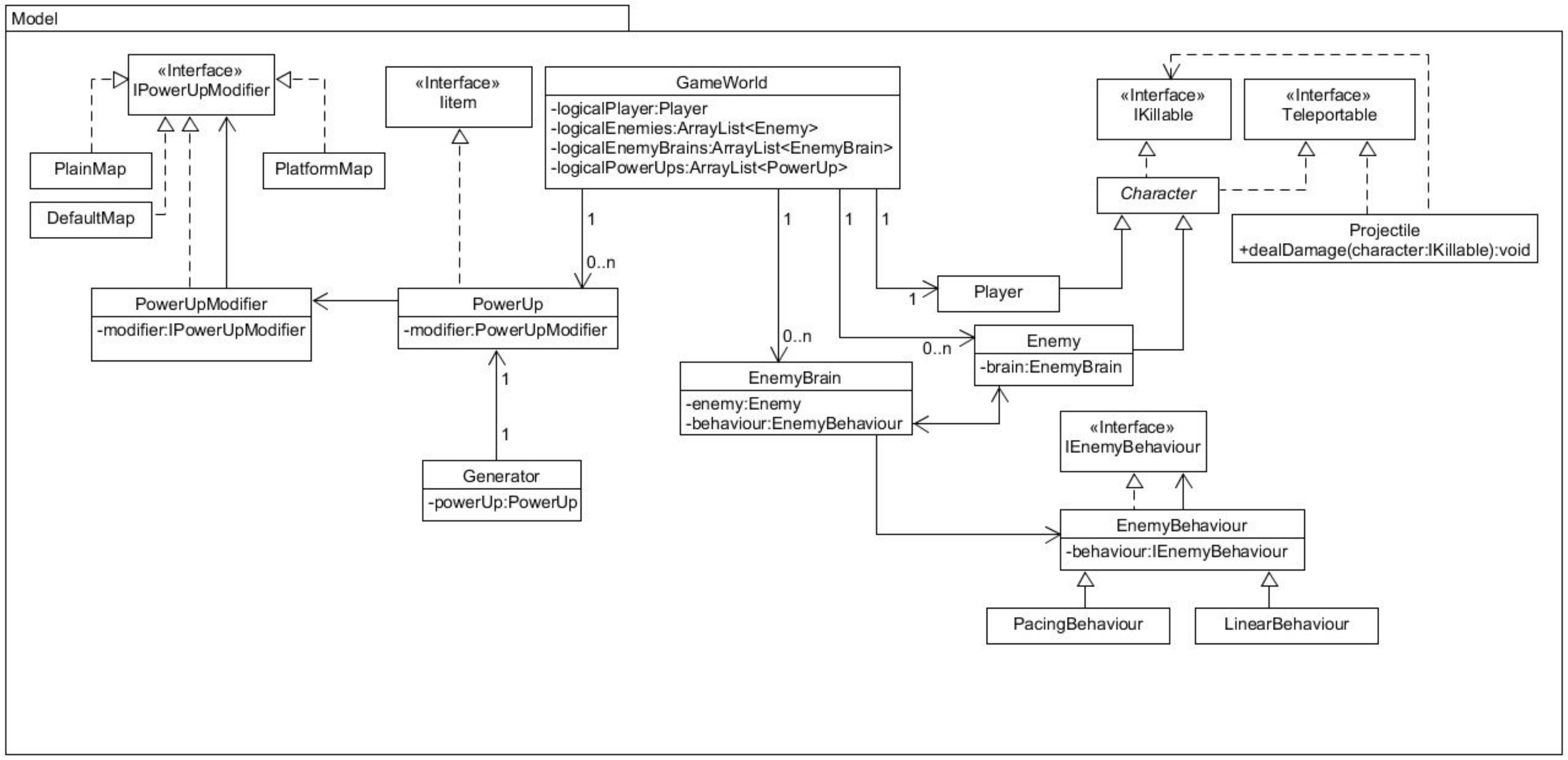
Several tests are implemented in the application. Here is a list over what is tested:

- Player:
    - A test to check if the player's health is reduced correctly
    - A test to check if the player's x velocity is reversed correctly
    - A test to check if the player's y velocity is reversed correctly
    - A test to check if the player is dead
    - A test to check if the player has crossed a certain x position on the map
- Enemy
    - A test to check if the enemies health is reduced correctly
    - A test to check if the enemies x velocity is reversed correctly
    - A test to check if the enemies y velocity is reversed correctly
    - A test to check if the enemies can die
- Generator
    - A test to check if a generated array is cleared correctly, by setting the value of all of the cells in it to 0
    - A test to check if a generated array starts with a 1 in a certain specified position
    - A test to check if a generated arrays next column value is a 1
- Projectile
    - A test to check if a projectile spawns correctly
    - A test to check if a projectile collides correctly
- Game world
    - A test to check if the logical enemies in a logical world are removed correctly
    - A test to check if the logical enemy brains in a logical world are removed correctly
    - A test to check if the logical powerups in a logical world are removed correctly
    - A test to check if the logical enemies in a logical world are created correctly
    - A test to check if the logical enemy brains in a logical world are created correctly
    - A test to check if the logical powerups in a logical world are created correctly
- Use cases
    - A test to check if the use case “the player moves takes damage and dies” functions correctly
    - A test to check if the use case “the player moves takes damage and survives” functions correctly

3 Subsystem decomposition

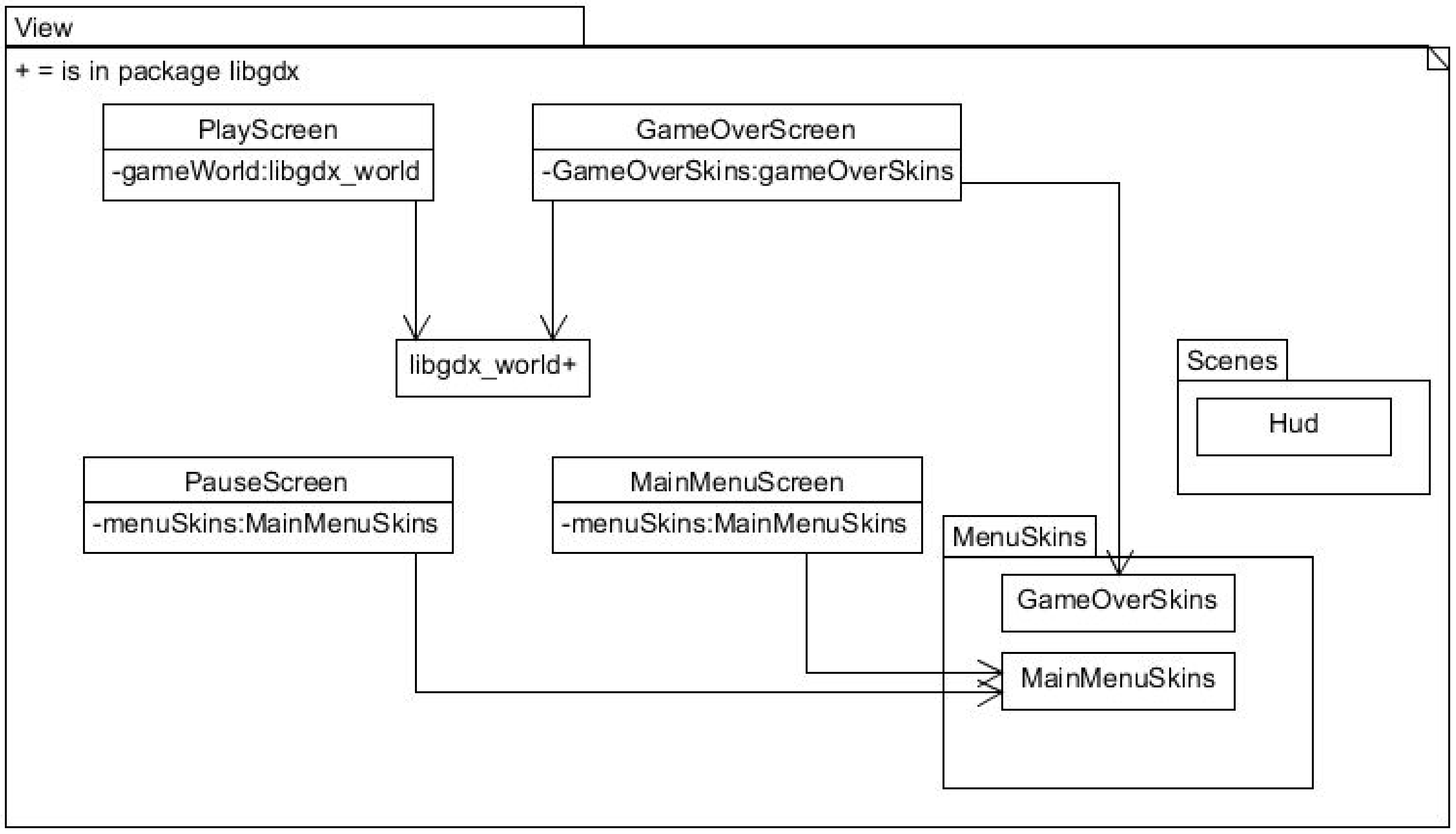
3.1 Model

The package containing the model of the application. Logic and data regarding various entities in the game, such as enemies, powerups and the player, is kept here. This data is later used in other parts of the game to decide how these entities should be modified and in which way. The data is updated whenever it needs to be, for example when a character changes position in the game world the model is updated to reflect this. Another important part of this package is that it contains the Generator class which is responsible for generating the map of the game.



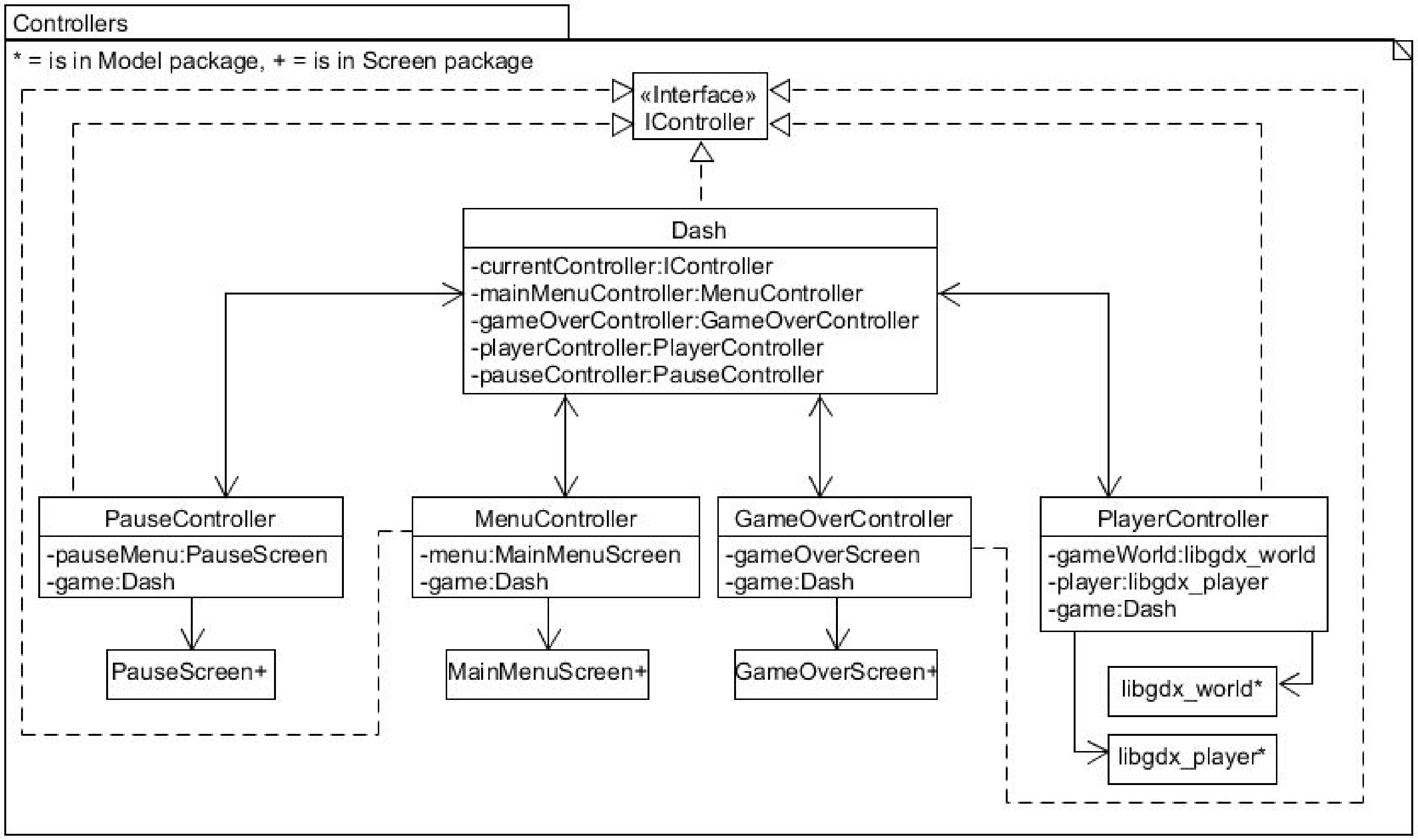
3.2 View

The package containing the graphical components of the application. The graphical interface presented to the user is handled here. It includes showing different screens to the user, such as the main menu screen and game over screen. It also handles the drawing of textures to the game world.



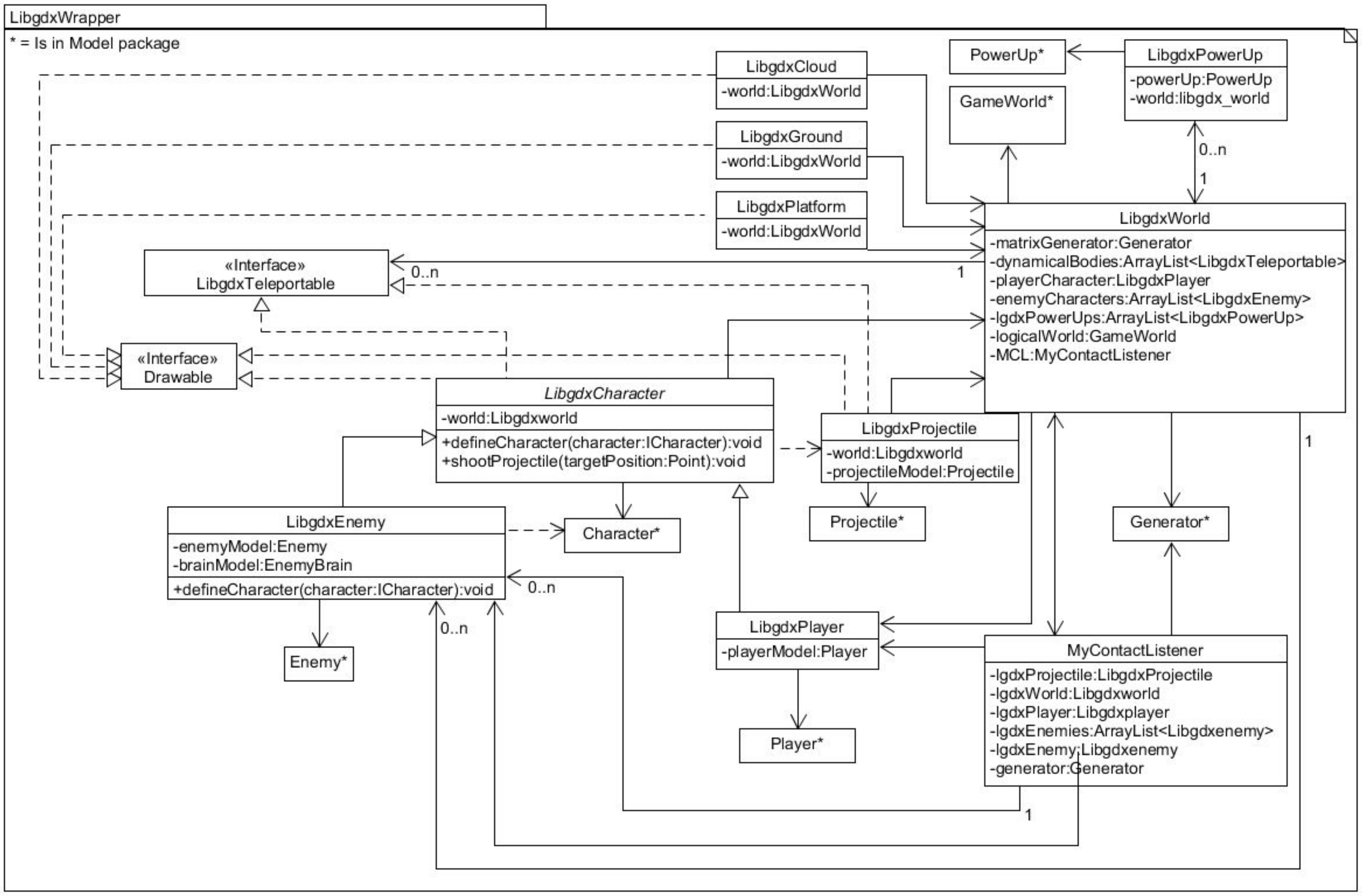
3.3 Controller

The Package containing the controllers of the application. The controllers are responsible with taking the input from the user and modifying the model and view to represent a change in the game. It could be that the user movers the player character or clicks the new game button to start a new game. It is also from here that the entire game is “set up” by creating all of the necessary controllers which in turn creates the rest of the game.



3.4 LibgdxWrapper

The package containing the libgdx functionality of the application. Here are functions related to libgdx handled. For example, all characters in the game are represented by a class in libgdx called body. It is through the use of this body that the user is able to move the player character around in the game world. By applying forces to the body and making it move, functionality for this is handled by the libgdx library. The libgdx functionality is separated from the rest of the model in order to test the “normal” model functions



4 Persistent data management

The application does not use any way of storing persistent data since there isn't anything the game currently saves.

5 Access control and security

No security implementations are used, the game is launched and exited as a normal desktop application. The only role within the application is the user role.