System Design Document for Killer Kidz Project (SDD)

Table of Contents:

Version: 1.0
Date: 31st of May, 2015
Author: Kim Berger, Oscar Beronius, Matilda Horppu, Marie Klevedal

This version overrides all previous versions.

1 Introduction

1.1 Design Goals

The design of the application must follow the MVC design pattern. It is also important that the coupling between the view and the rest of the code is loose so that it will be easy to change the GUI in order for it to be able to fit other platforms in the future. The design must also make the code easy to test, i.e. each part of the code should be able to be tested separately. Another important aspect of the design is that it should be easy to add more functionality to the application, e.g. more levels, more cany types and more kids.

1.2 Definitions, Acronyms and Abbreviations

- GUI - Graphical User Interface
- JRE - Java Runtime Environment
- MVC - Model, View, Controller design pattern, which is an pattern that will split the program into three parts, the View which handles all the graphics, Controller which handles the interaction from player to the game and also interactions between the model and view and lastly an Model which is the engine of the application and handles all data and calculations on this data.
- Wave - A group of kids that runs out on the game board and that have to be gone before more kids can emerge.
- FPS - Frames per second, how many times the software is updating its components per second.

2 System Design

2.1 Overview

As previously stated the application is built using the MVC design pattern and it is using the the passive version of MVC. Thus, the project is divided into three main packages, one for the model classes, one for the controller classes and one for the view classes. In these main packages there may be subpackages in order to get a nice structure. And the game is thereafter updating itself using passive type, which means we update the game in a set interval.

2.1.1 Model Functionality

In the model package there is a class called Model which is the coordinator for all the model classes. This coordinator class will handle all communication with the controller and is also responsible for updating the game.

For all the moving objects in the game there is a class, Entity, which they all extend in order to create another hierarchical level instead of all these objects just extending Object.

For the objects Kid, Candy, Player and Level there are abstract classes. These will be further explained in separate paragraphs.

2.1.2 Candy Functionality

In the Entity package, a package named candyModels can be found. CandyModels contains the models for all candies, including an abstract class Candy that all candymodels extend, as well as an abstract class CandyFactory. There are three candy types that have different properties, such as modifiers(one candy deaccelerates kids that it hits) or upgrade options. Candy factory is used to create all objects of the different candies. By using candy factory, we can avoid circular dependencies when more complicated types of candies are created, for example jelly bean that, when upgraded in a certain way, duplicates into more jelly bean objects.

All candies have an "upgrade tree" of four paths, that correspond to four different types of upgrades. The upgrade status of a candy is described by an array of four ints, where each int represents a level of upgrade for a corresponding path in the upgrade tree. When a candy is created, an array that contain the upgrade status of that certain candy is retrieved from the player that throws the candy.

2.1.3 Kid Functionality

In the Entity package, which is in the Model package, there is a Kids package, which contains all the kid classes. The abstract class Kid extends Entity and contains all features that the kids have in common. There are four kids extending this class, all with their own moving pattern and weakness for different kinds of candy. There is an enum class with the four different kid types. The kids are generated using the factory design pattern, hence there is a class KidFactory with a createKid() method that is called each time a new kid should be created. Along with start coordinates of the kid, the method takes an instance of the enum, to know what kid to create. In this way, the rest of the application doesn’t have access to the four non-abstract kid classes.

2.1.4 Level Functionality

In the Model package there is a package that contains all the levels. A level is started from the method startLevel() in model. What level that should be started is stored in the variable currentLevel which is being switched upon in startLevel().

Every level extends the abstract class Level and their purpose is to create Kid objects at different intervals. In order to simulate different waves of kids there is switch block in the update method and depending on what the variable currentWave containes different cases will be executed. In each case, or wave, differend kids may be created at randomized intervals and positions. Also the type of kid that is being

randomized with different probability in certain levels. When the kids are being created they are added in an ArrayList in Level. The class model can then get this list and add the elements to a list in model which then is used when updating the game.

2.1.5 Player Functionality

The Player class can be found in a package named Players in Entity. The player class contains data regarding his/her position, an arrayList candyData that contains arrays of ints that each represent a candy that the player has unlocked, and it's upgrades. The updateDir function takes in an array of four booleans, that represent the four movement keys that can be pressed. When updateDir is called, it updates what direction the player is going to move based on the keys that are pressed.

The update method is called each time the time controller ticks. It updates the player's position depending on the direction that is set. If both the up and down keys are pressed, the player will stand still, same goes for left and right key. This is handled in the update method.

The player class contains an arrayList<Candy> activeCandies that contains all candies that the player has thrown and is active on the game board. ThrowCandy calls candyFactory that creates a new candy object based on what candy the player has currently selected, also adds the candy in the list of active candies.

2.1.6 Candy Shop Functionality

Model for handling all logic in candy shop. changePllayer changes browsing player, changeCandy changes candy that is to be upgraded, move takes in an int 0-3, and moves the cursor left, up, right or down depending on the value of the int. getStatus returns the status of all rows and columns in the candy shop. If row and col represents an upgrade, getStatus will return "not" if the player cannot buy the upgrade, "have" if the player already possesses the upgrade and "buy" if the player is eligble to buy the upgrade.

2.1.7 Updating Mechanism

The Model (engine) part of the game, is updating itself using our own class TimeController which holds an timer. But the view-part is then updating with an 60FPS by calling the model’s current state at each view-call.

2.1.8 GUI Structure

We use libGDX with the Lightweight Java Game Library Application as a base, and after that we have an class GameManager as an applicationListener, which manage which Screen we should render (using StatePattern with the views as states), which can shift between these Screens:

- MainMenu - the first screen the user will see, which he can navigate through to get to an How-to-play Screen, the Play screen or the option to quit the game.

- HowToPlay - a screen with basic information how to play the game.
- Playfield - The actual game, where you move the player (an triangle) with arrow keys, throw candy with space and change candy with numbers 1-n (if you have unlocked more the one) to be able to throw that specific candy.
- CandyShop - Here you will be able to upgrade your current candy and also unlock more types of candy.

And the Screen PlayfieldView class(screen/state) is also using multiple classes which each has their own responsibility to what to render, and then add all these to its render-method.

2.1.9 Event handling

The only events are when the user presses a key. To register this, two of our controller classes (MainMenuController and GameController) extend libGDX's class InputAdapter. MainMenuController uses a state pattern to know if user is viewing HowToPlay or is in the MainMenu, in this way it is easy to add more choises in the MainMenu. The controller class MainMenuController is not an InputAdapter, but gets its calls from GameManager who keeps track of whether the player is playing or in the CandyShop.

2.2 Software Decomposition

2.2.1 General

The application is decomposed into the following modules,

- **controller** - The Control parts for the MVC Pattern.
- **model** - The Model parts for the MVC Pattern.
    - **entity** - all the classes for things that move around at the playfield
        - **candymodels** - the classes representing the candies
        - **kids** - the classes representing the kids
        - **players** - one class representing the player
    - **levelmodels** - the classes for the levels
    - **Candyshop** - The class that handles all logic in candy shop
- **View** - main GUI and the View parts for the MVC Pattern.
    - **gameStates** - all the GUI classes that each render a specific screen(state)
        - **playfieldGUI** - the specific GUI classes which have the responsibility to render each part in the playfield.
    - **inGameEntities** - the graphical classes with the responsibility to render all moving objects in game.
- **core** - The GameManager, who handles which screen(state) to render.
- **default package** - holding the Main class which is the entry for our application.

2.2.2 Decomposition into subsystems
NA.

2.2.3 Layering
NA.

2.2.4 Dependency analysis
Dependencies are as shown in the enclosed folder STAN, and there exist no circular dependencies.

2.3 Concurrency issues
None.

2.4 Persistent data management
NA.

2.5 Access control and security
NA.

2.6 Boundary conditions
NA.

3 References

Appendix
- **Player** - The unit the user will control (an triangle) to move around and stop the progressing children by throwing candy at them.
- **Level** - A set of waves
- **Waves** - A set of kids
- **Money** - The reward the player gets when killing kids, can be used to buy candy upgrades.
- **Candy** - A projectile the player can throw at kids to make them content and thus leave the toy store alone.
- **CandyType** - The player may choose from unique types of candies, where each has their different strengths and weaknesses.
- **CandyUpgrades** - After you finished a level, you will progress into the CandyShop, where you can spend gained money from
- **Kids** - The goal of the game is for the player to stop these from progressing from the right into the shop.
- **KidTypes** - There exists different type of kids, where each have their own strengths and weaknesses.
- **Lives** - an indicator which tells how many kids the player may fail to stop, before losing the game.

- **FPS** - The number of times that the screen is updated per second.