System Design Document

for Space Dodger

Authors: Isak Almeros, Tobias Engblom, Viktor Sundberg, Irja Vuorela, Olle Westerlund
2020-10-23
Version 3

1 Introduction

The application described in this document is a game where the user controls a spaceship to avoid different types of damaging projectiles on the playing field. There are also items that the player can pick up to gain different benefits or disadvantages. The user gets points based on how long they can survive.

1.1 Definitions, acronyms, and abbreviations

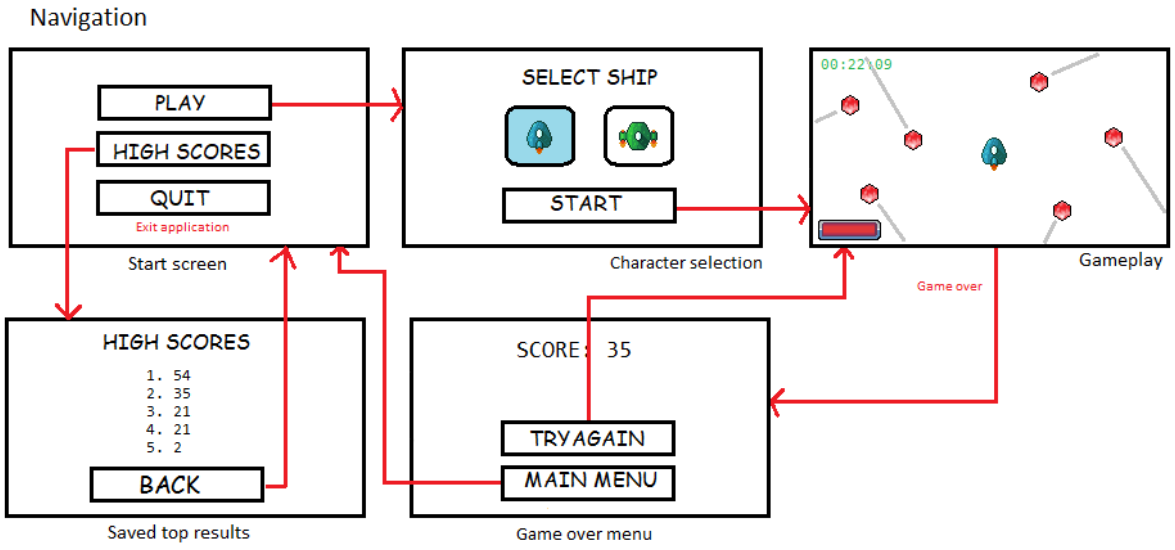
- Unit test: a unit test tests a specific functionality in the code and determines a specific behaviour or state.
- Health points (HP) - A value representing the player's remaining life.
- High score: A list with the top scoring rounds a player has played.
- Power up: A game object the player can pick up that gives a temporary advantage.
- Character: The entity controlled by the player during the game.
- Game Over: The player's hp reaches zero and the game round is therefore ended.
- Wraparound: The playing field has no walls. If the player moves their character off screen the character appears on the opposite side and continues in the same direction.
- Laser beam: A laser beam that fills the playing field from either top to bottom or from side to side, forcing the player to utilize the wraparound to avoid taking damage.
- Debuff: A game object that gives the player a disadvantage.

2 System architecture

2.1 User flow

When the application starts, the user sees a home screen with three possible options. The user can select either to start a new game, look at the current high scores or quit the application. If the user selects to create a new game, he/she will be redirected to another view where the user can select different kinds of spaceships and a play button to start the game.

The game will then start with the player's selected spaceship in the middle of the screen and with asteroids flying in from the sides. When the player's health points reach zero, the game is over and the user is met by a game-over view that shows the player's total points for the game. In this view the player can select either to play again right away with the same spaceship, or return to the main menu. The list over the top scores are saved locally.



2.2 Technical flow

2.2.1 Startup

On startup the project's main class runs the start method which initializes our program window, by creating a “Stage” object. A “scene” is inserted into the stage, which displays the contents inside the window. The start method creates instances of the application's different views such as the main menu, the game over menu and so forth and the main menu is displayed in the window.
A “viewcontroller” is created, which handles the navigation between all menus and the game. Every time we navigate to a different menu, the scenes root in the stage object is switched. The start method also creates a game loop to be used further down in the functionality of the application. Lastly our sound handler is called to play and loop the applications background music directly from startup.

2.2.2 Runtime

Once the application is up and running the user is prompted with the main menu. This menu contains three buttons allowing the user to choose to either play, view highscores or quit the application. When a button is clicked our view handler executes the logic to launch the selected view. When the play button is clicked, the user is prompted with a character selection scene which contains four different sprites for the player to choose between. The player can then select one of the four, marking their choice as selected, and then press either play to launch the actual game or press back to go back to the main menu.

When the play button in the character selection menu is pressed a start game method is called and the game loop starts. The game loop initializes the user's character (spaceship) and calls the wave manager class to execute one of its scenarios containing the spawning pattern and behaviour of hostile and friendly projectiles. The wave manager class in turn calls the projectile factory to create the projectiles in any given scenario through modular building blocks that can be used to build different gaming experiences.

The game loop is also responsible for executing updates on the game objects movement as well as calling the collision handler class to check for and handle collisions in relation to elapsed time. The collision handler class in turn is responsible for handling the logic of what needs to happen when two objects collide, including removal of objects from the list of game objects and notifying the observers. All objects except spaceships and laserbeams are removed upon collision with the aforementioned objects. The soundhandler then plays the correct sound depending on what collision has happened.

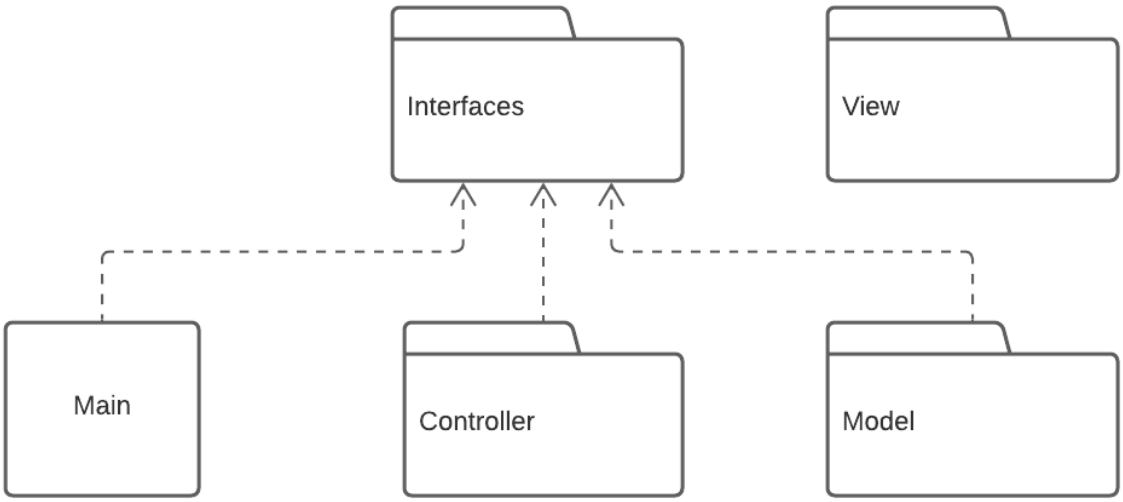
When the user's character takes lethal damage the game class notifies the observers who listen for game over. Our view controller observes the game over state and in turn launches the game over menu and stops the game loop. The game over menu displays the users collected score along with two buttons allowing the user to choose to either play again or go back to the main menu. If the user managed to collect enough score to place in the top ten of previously acquired scores, the score is saved to a local text file and the high scores are then sorted by size.

2.2.3 Closing

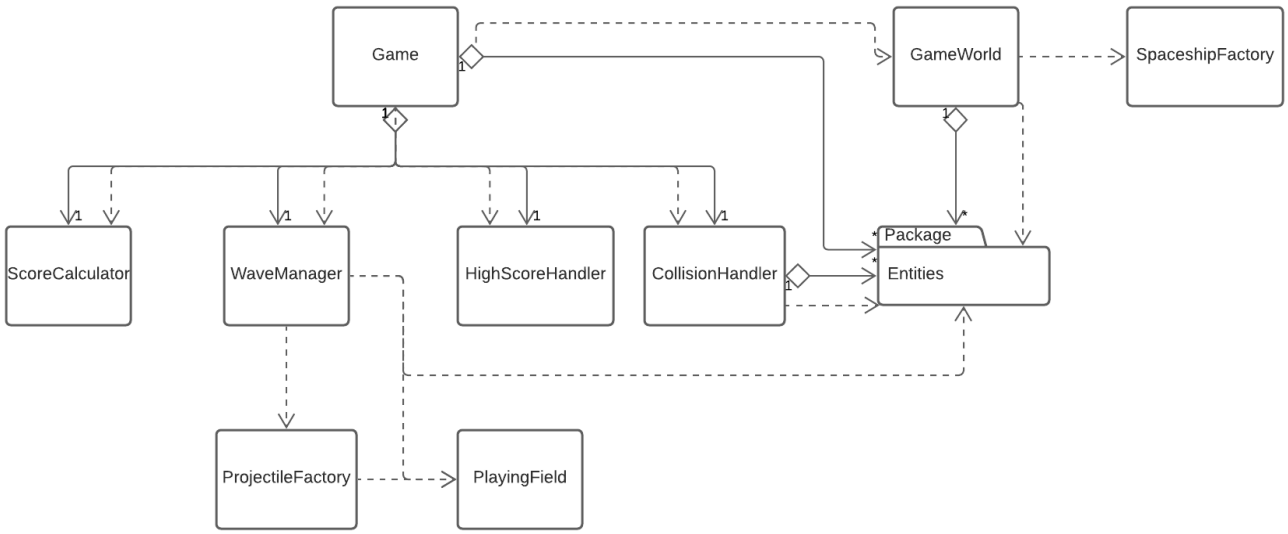
Upon closing of our application the program simply exits the application. If the application is terminated during an active game round the score and progress is simply dismissed as the application is terminated.

3 System design

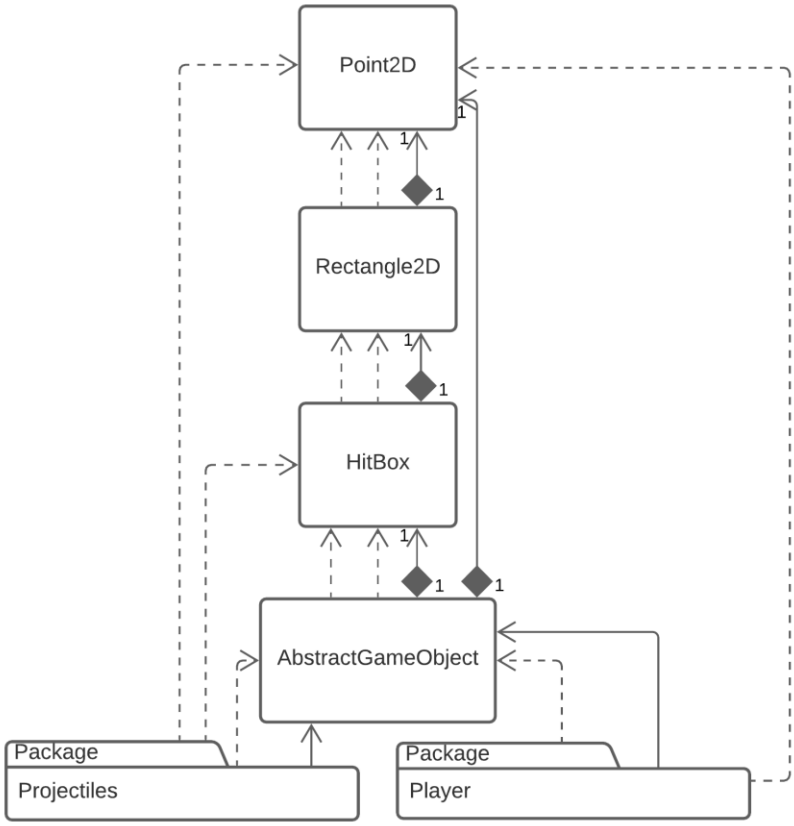
Top level view



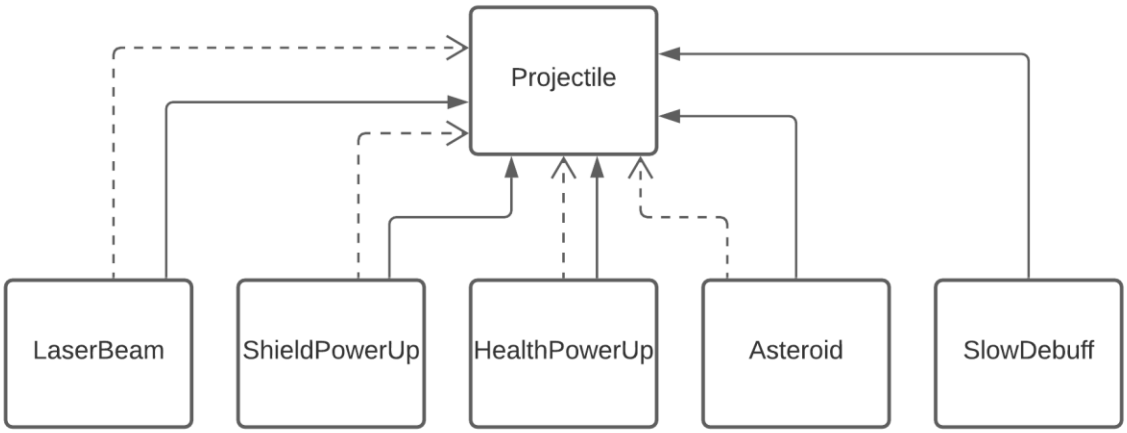
Model package



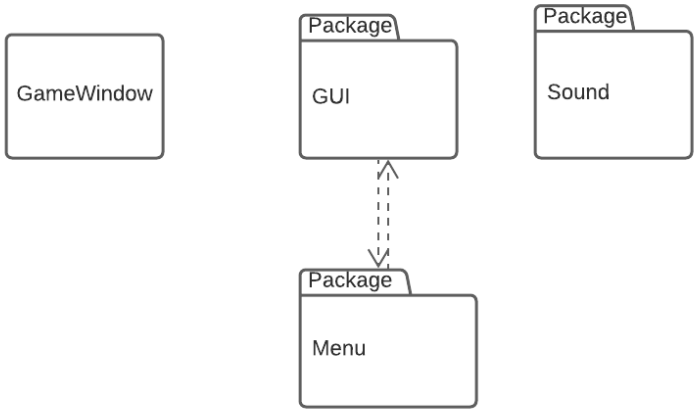
Entities package



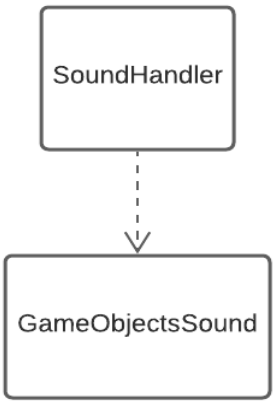
Projectiles package



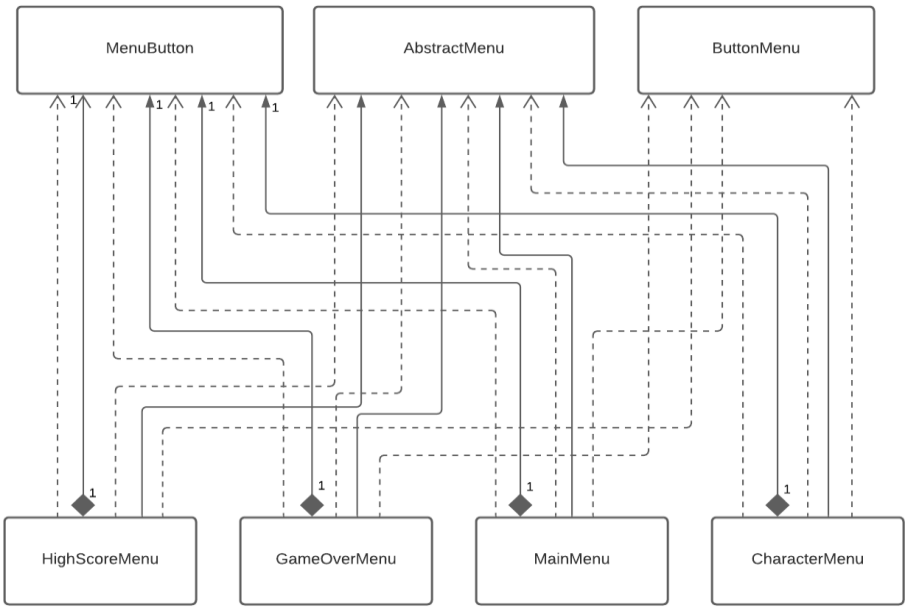
View package



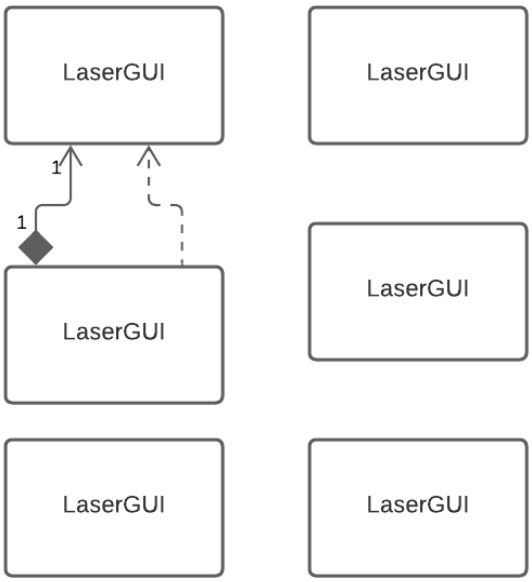
Sound package



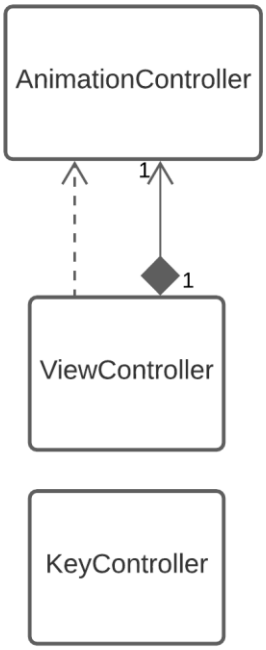
Menu package



GUI package



Controller package



MVC pattern: The code is separated into three different main packages; model, view and controller. The model does not know about any of the other parts. A key controller handles all key inputs in the game and manipulates the model. When the model updates its state it will notify that it has changed. The view then updates depending on the new state of the model.

Singleton pattern: The application uses a singleton pattern when it creates the model, only one model can be created at a time and after that only that instance is returned if some part of the application needs it.

Factory method: When the application creates different types of projectiles it uses a factory method for this.

Observer pattern: Observers are being notified when changes have occured in the model, specifically:

- in each frame in the game loop in the AnimationController class.
- each time a collision with a spaceship or a laser beam occurs.
- when the game has ended.

The Main class observes the elapsed time in the simulation and updates all views in the simulation. The Main class also observes the CollisionHandler class to update sounds associated with collisions.

ViewController observes the game over state, and launches the game over menu when the game is over.

Open/Closed Principle:

The projectile class is an abstract super class for all the different types of projectiles, this makes it easier to add new different kinds of projectiles that have the same base properties and behavior.

4 Persistent data management

All the images that the application uses is in the folder “resources” which is in the same folder as the source root.

The result of the games, the list of high scores, are saved locally in a text document.

5 Quality

5.1 Testing

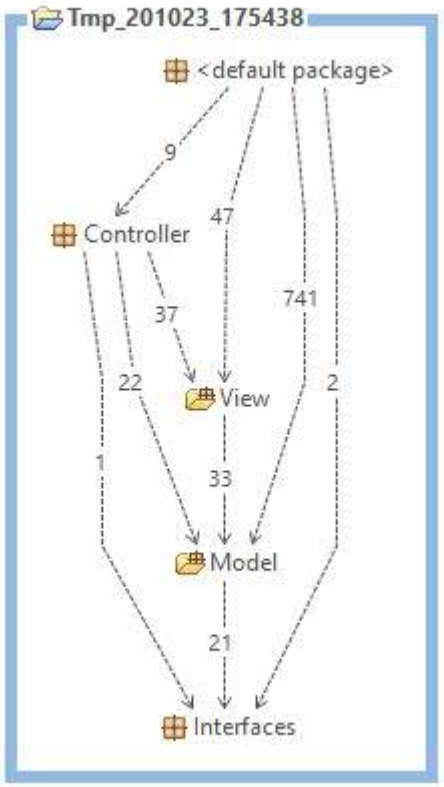
The code is tested with unit tests with help of JUnit.
Testfolder: DodgerGame/src/test/java
Private methods pertaining to the model in some instances lack 100% line coverage in our testing. This is because of unreachable method lines, such as exceptions, and leaving these out of our testing has been an active decision by the group, not an oversight.

5.2 Known Issues

- At some times the images won’t load from the resource folder, what we have found is that the images need to be added in the target folder. We don’t know why this needs to be done manually sometimes.
- The bottom half of vertical laser beams doesn’t detect collisions with other objects. We can see that the hit box for the laser beam is correct as well as it’s position, so we are not sure why it only works for the top half. It has been doubled in length as a workaround (only the top half is on the playing field) until this issue is solved.
- Somehow Isak committed to Github both with his real name Isak Almeros and his github name WayneGretzky. Only the commits by WayneGretzky are seen under contributions, but if pressing “pulse” you can also see the commits made by Isak Almeros.

5.3 Dependency Analysis

The project has been analyzed using STAN to determine if any circular dependencies exist within the project. The project contains zero circle dependencies. The following image displays the project's dependencies between packages.



5.4 Quality Report

To verify the project's code quality we have used the PMD plugin.

5.5 Access control and security

The application uses no form of access control.

6 References

JavaFX - https://openjfx.io/
PMD plugin - https://plugins.jetbrains.com/plugin/1137-pmdplugin
STAN - http://stan4j.com/