Virtual Arcade

Final Report

Version: 1.0
Date: 23/10 - 2020
Authors: *Georges Kayembe, Mhd Jamal Basal, Jonathan Stigson, Oliver Ljung, Robert Sahlqvist*

1 Introduction

1.1 Purpose of application

The project aims to implement a virtual arcade with five or more integrated retro games. The application has great focus on recreating a nostalgic feeling whilst introducing a modern vibe into old retro arcade games.

1.2. General characteristics of application

The program is a platform independent desktop application able to launch five different retro games. The five games available are as follows: Breakout, Frogger, Pong, Snake and Space Invaders.

1.3 Definitions, acronyms, and abbreviations

GUI: “Graphical User Interface” also “User Interface”, the part of the program that the user sees and interacts with.

DoD: “Definition of Data”, a list of requirements that have to met for a User Story to be considered done.

User Story: short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

RAD: “Requirement and Analysis document”.

SDD: “System Design Document”.

UML: “Unified Modeling Language”, A visual language for mapping and representing systems. Can be used for modelling och class diagrams, sequence diagrams and more. In this project used for domain and design models.

MVC: ”Model, View, Controller”. A well known design pattern which is hugely used in applications that have a GUI representation. It aims to separate the different areas of code(logic, viewing,...)

FXML: An XML-based user interface markup language created by Oracle Corporation for defining the user interface of a JavaFX[1] application.

Source root: The directory from which point on is strictly programming code. In this project’s case, only Java-files.

SOLID: A set of design principles, defining broad aspects of good object oriented design.

JavaFX: JavaFX[1] is a software platform for creating and delivering desktop applications, as well as rich Internet applications (RIAs) that can run across a wide variety of devices.

2 Requirements

2.1 User Stories

Story Identifier: GM
Story Name: Game Selection
Description: As a player, I want to be able to select a game from the game selection menu and so that I can then play it.
Confirmation: Story is done the player is able to select a game and then be able to play said game.
Functional: When the player is in the game selection menu he/she should be able to select a game and then be taken to that game,

Story Identifier: P1
Story Name: Pong Movement
Description: As a player, I want to be able to move either the right or the left paddle, depending on if I'm player one or player two, so that I am able to stop the ball from passing my paddle.
Confirmation: Story is done when the paddle or paddles move in the correct direction when the player presses the arrow keys.
Functional: If the player presses the up arrow key, the paddle moves up. If the player presses the down arrow key, the paddle moves down.

Story Identifier: P2
Story Name: Pong Score
Description: As a player, I want to be able to receive points when I score a goal.
Confirmation: Story is done when a player gets a point if said player scores a goal on the opposite player's paddle.
Functional: If the ball goes past a paddle, the player with the opposite paddle gets a point on the scoreboard at the top of the screen. The scoreboard numbers are placed on either side of a dividing line.

Story Identifier: P3
Story Name: Pong Ball Bounce (Players)
Description: As a player, I want the ball to bounce off the players.
Confirmation: Story is done when the ball bounces off the players' paddles if the ball hits them.
Functional: If the ball "touches" any of the paddles, the ball's x direction is flipped so that it goes in the opposite direction.

Story Identifier: P4
Story Name: Pong Ball Bounce (Walls)
Description: As a player, I want the ball to bounce off the walls.

Confirmation: Story is done when the ball bounces off the top and bottom walls if the ball hits any of them.
Functional: If the ball "touches" the top or the bottom wall, the ball's y direction is flipped so that it goes in the opposite direction.

Story Identifier: P5
Story Name: Pong Game Pause
Description: As a player, I want to be able to pause the game.
Confirmation: Story is done when a player is able to pause the game with the push of a button.
Functional: If the player hits a designated key, everything in the game stops moving and becomes unresponsive until the player hits the key again.

Story Identifier: P6
Story Name: Pong AI Player
Description: As a player, I want to be able play against an AI bot if a second player isn't available.
Confirmation: Story is done when a player is able to play against a bot rather than another player.
Functional: If the player hits a designated key, then the right paddle becomes controlled by a bot which reacts to the ball's position.

Story Identifier: P7
Story Name: Pong Ball Respawn
Description: As a player, I want the ball to respawn in the middle after a goal.
Confirmation: Story is done when the ball disappears and then reappears in the middle of the game after it scores a goal.
Functional: If the ball hits the right or left edge of the screen it despawns and respawns in the middle of the game with a random direction.

Story Identifier: SI1
Story Name: SpaceInvaders Player Movement
Description: As a player, I want to be able to move the player using the arrow keys so that I can avoid getting hit by enemies.
Confirmation: Story is done when the player character moves in the correct direction when the player presses the arrow keys.
Functional: If the player presses the right arrow key, player moves right. If player presses the left arrow key, player moves to the left

Story Identifier: SI2
Story Name: SpaceInvaders Shooting
Description: As a player, I want to be able to shoot when I press a button and with that shot I want to be able to destroy enemy aliens so that I can get rid of all the aliens.

Confirmation: Story is done when the player shoots a projectile that can destroy enemy aliens with the press of a specific button.
Functional: If the player presses the “shoot button”(button not determined yet) a projectile that can destroy enemy aliens will be spawned.

Story Identifier: SI3
Story Name: SpaceInvaders Score
Description: As a player, I want to be able to gain score by destroying enemy aliens, so that I can get a better high score.
Confirmation: Story is done when the player gains score by destroying enemy aliens.
Functional: If the player shoots and destroys an enemy alien score should be added.

Story Identifier: SI4
Story Name: SpaceInvaders Death and Respawn
Description: As a player, I need to be able to die and then respawn if I have lives left so that the game will be challenging.
Confirmation: Story is done when the player dies upon collision with an alien bullet, player also has 3 lives.
Functional: If the player collides with an enemy bullet he/she will die and respawn if there are lives left.

Story Identifier: SI5
Story Name: SpaceInvaders Enemy Movement
Description: As a player, I want the enemy aliens to move in the original pattern so that the game is recognizable.
Confirmation: Story is done when the player can visually see that the enemies are moving in the original games pattern.
Functional: When the game updates the enemies shall move according to the original pattern

Story Identifier: SI6
Story Name: SpaceInvaders Pause
Description: As a player, I want to be able to pause the game so I can take a breather if needed.
Confirmation: Story is done when the player can pause the game and with that get a pause menu.
Functional: When the player presses the escape button the game will pause and the player will arrive at a ‘pause menu’.

Story Identifier: F1
Story Name: Hippity hoppity
Description: As a user, I need to be able to move the Frog so that I can complete the goal of the game.
Confirmation: User Story is done when keypresses register and frog moves accordingly.

Functional: Does the frog move when I use the arrow keys? Does the frog stay within the boundaries of the game's playing field?

Story Identifier: F2
Story Name: Lives lost
Description: As a user, I need to know what the rules of interacting with obstacles are so I have the necessary information to complete the game.
Confirmation: User Story is done when clear indicators (visual or other) answer questions of functional criteria and align with functionality of the program.
Functional:
Do I have multiple lives?
Do I lose a life when I get hit by a car?
Do I lose a life if I fall into the river?

Story Identifier: F3
Story Name: End of the road
Description: As a user, I need to know what the rules of winning/losing are so I have the necessary information to complete the game.
Confirmation: User Story is done when clear indicators (visual or other) answer questions of functional criteria and align with functionality of the program.
Functional:
Do I get "Game Over" if I lose all my lives?
Do I get points for getting a frog across the road and the river?
Do I finish the level if I get all the frogs across the road and the river?

Story Identifier: F4
Story Name: Modern Vibe
Description: As a user, I need the game to look new and modern while still maintaining the feeling of the original game so that I can relive what it was like to play the original while still feeling like it is a new and modern game.
Confirmation: User Story is done when functionality and visuals match the original Frogger (good enough).
Non-functional: Is the game recognisable?

Story Identifier: SG1
Story Name: Snake Player Movement
Description: As a player, I want to be able to move the player using the arrow keys so that I can turn towards the food.
Confirmation: Story is done when the player character moves in all four directions when the player presses the arrow keys.
Functional: Does the Snake move when I use the arrow keys?

Story Identifier: SG2
Story Name: Snake Score

Description: As a player, I want to be able to gain score by eating food, so that I can get a better high score.
Confirmation: Story is done when the player gains score by eating food.
Functional: If the player eats a food score should be added.

Story Identifier: SG3
Story Name: Snake Death and Restart
Description: As a player, I need to be able to die and then start over.
Confirmation: Story is done when the player dies upon collision with himself or a wall.
Functional: If the player collides with an himself/wall he will die and start over again .

Story Identifier: SG4
Story Name: Snake Pause
Description: As a player, I want to be able to pause the game so I can take a breather if needed and also go back to the menu.
Confirmation: Story is done when the player can pause the game and with that get a pause menu.
Functional: When the player presses the escape button the game will pause and the player will arrive at a 'pause menu'.

Story Identifier: SG5
Story Name: Snake Speed
Description: As a player, I need to be able to choose the player's speed (Slow, normal, fast and insane).
Confirmation: Story is done when the player chooses one of these speeds.
Functional: The player's speed will be increased according to the speed chosen.

Story Identifier: SG6
Story Name: Wall Collisions
Description: As a player, I need to be able to choose between activating/deactivating a collision with the wall.
Confirmation: Story is done when the player chooses Off/On (Wall Collisions) .
Functional: If the Wall Collisions (OFF): The player appears form the opposite side when the collision occurs. If the Wall Collisions (ON): The player will die when the collision occurs.

Story Identifier: SG7
Story Name: Gameboard Size
Description: As a player, I need to be able to choose the Gameboard size (Small, medium, and big).
Confirmation: Story is done when the player chooses one of these sizes.
Functional: The board size will be changed according to the size chosen.

Story Identifier: BO1

Story Name: Breakout Menu1
Description: As a player, I want to be able to see a description about the game to understand how the game works.
Confirmation: Story is done when the player starts the game and clicks on the button “Help” from the game menu.
Functional: If the player clicks on the ”Help button”, the player will see a short description about the game.

Story Identifier: BO2
Story Name: Breakout Menu2
Description: As a player, I want to be able to see a record of the best score for the purpose of measuring upp myself and increasing my skills .
Confirmation: Story is done when the player starts the game and clicks on the button “Scores” from the game menu.
Functional: If the player clicks on the ”Score button”, the player will see a list which contains the five best players and high scores.

Story Identifier: BO3
Story Name: Breakout Menu3
Description: As a player, I want to be able to identify myself in order to track my score.
Confirmation: Story is done when the player starts the game and clicks on the button “Play” from the game menu. Then the player will be able to write his first & last name and start a new game.
Functional: If the player presses the ”Play button”, the player will be able to identify his/her self and then starts a new game.

Story Identifier: BO4
Story Name: Breakout Stop
Description: As a player, I want to be able to end upp the current breakout game so that I can choose to start a new one.
Confirmation: Story is done when the player clicks on EXIT-button.
Functional: If the player presses the ”EXIT button”, the player will be able to move to the game menu and start a new game.

Story Identifier: BO5
Story Name: Breakout Pause
Description: As a player, I would like to be able to pause the game-play, in order to return to the game at a later time.
Confirmation: Story is done when the player clicks on PAUSE. The game will continue when the player clicks on PAUSE again.
Functional: If the player presses the “PAUSE button” the game will pause successfully. When the player hits the “PAUSE button” again the game will resume as usually.

Story Identifier: BO6

Story Name: Breakout Move
Description: As a player, I would like to be able to move the paddle horizontally (right and left) for the purpose to hit the ball.
Confirmation: Story is done when the player clicks on ‘Q’ or ‘W’.
Functional: If the player presses the key ‘Q’ the paddle would move to the left. If the player presses the key ‘W’ the paddle would move to the right.

Story Identifier: BO7
Story Name: Breakout Score1
Description: As a player, I would like to see my current score while playing the game.
Confirmation: Story is done when the player starts a new game.
Functional: The score will be displayed at the top of the screen while the player is playing the game.

Story Identifier: BO8
Story Name: Breakout Score2
Description: As a player, I would like to save my score when I lose or win a game.
Confirmation: Story is done when the player hits the last brick or when the ball hits the bottom. The score will automatically be saved.
Functional: If the player hits the last brick or if the ball hits the bottom, the score will automatically be saved. The player should be able to see his/her score if and only if the score is in the top 5 high scores.

2.2 Definition of Done

To determine whether a User Story has been completed or not, all criteria from either group of criteria need to be fulfilled. Those groups of criteria are:
Group 1:

- If there can visually or otherwise be determined that it fulfills its intended purpose.
- If it can fully be tested via JUnit.

Group 2:

- If there can visually or otherwise be determined that it fulfills its intended purpose.
- If peer review determines that it fulfills its intended purpose.

Group 2 is only applicable if the aspects of the User Story cannot be tested via JUnit.

2.3 User interface

Screenshots from the application:

Figure 1. Start Menu:

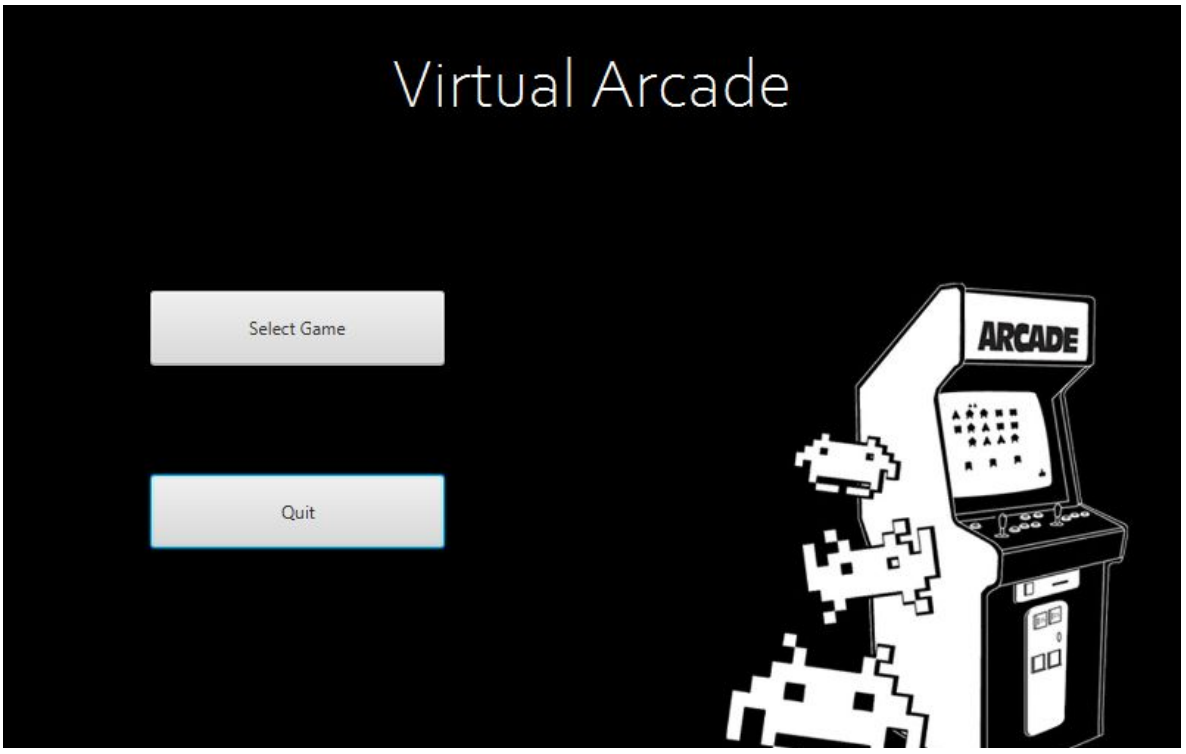


Figure 2. Game Select Menu:

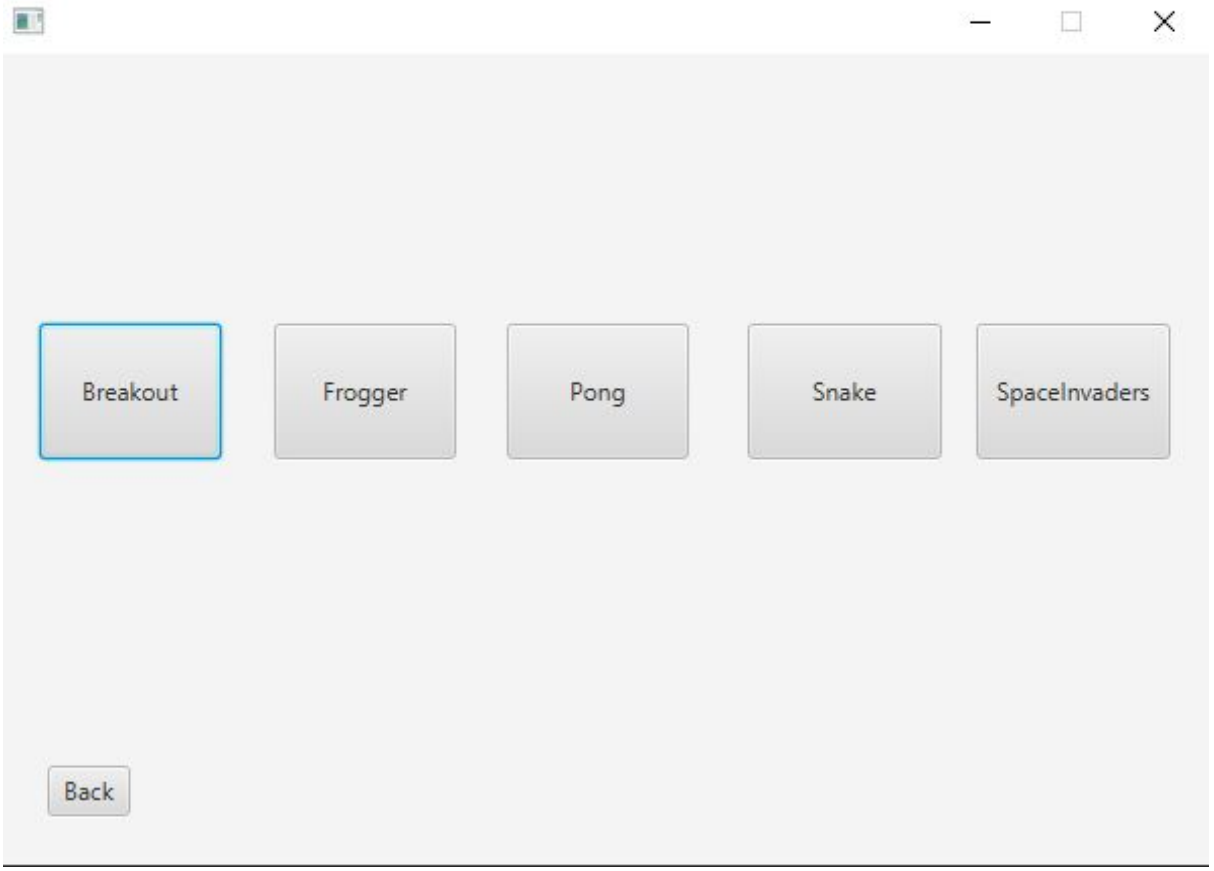


Figure 3. Breakout Menu:

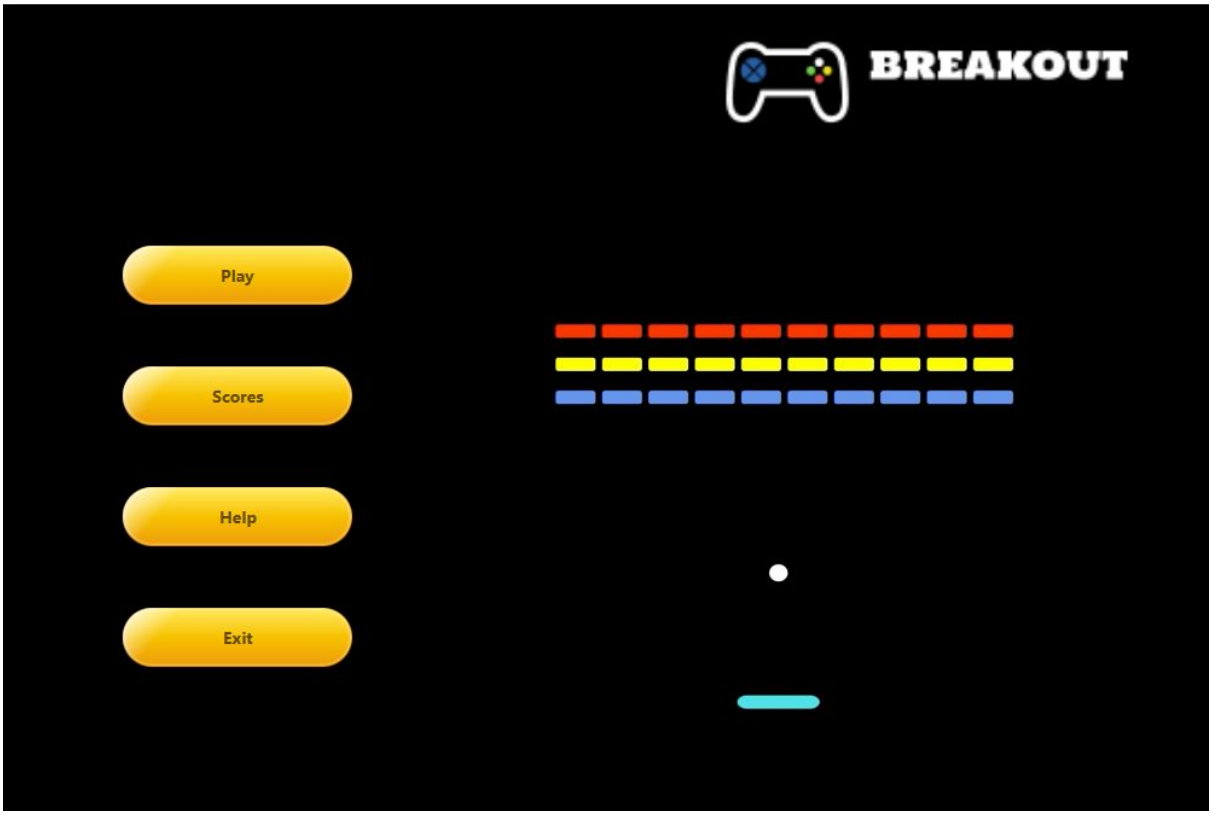


Figure 4. Breakout Game:

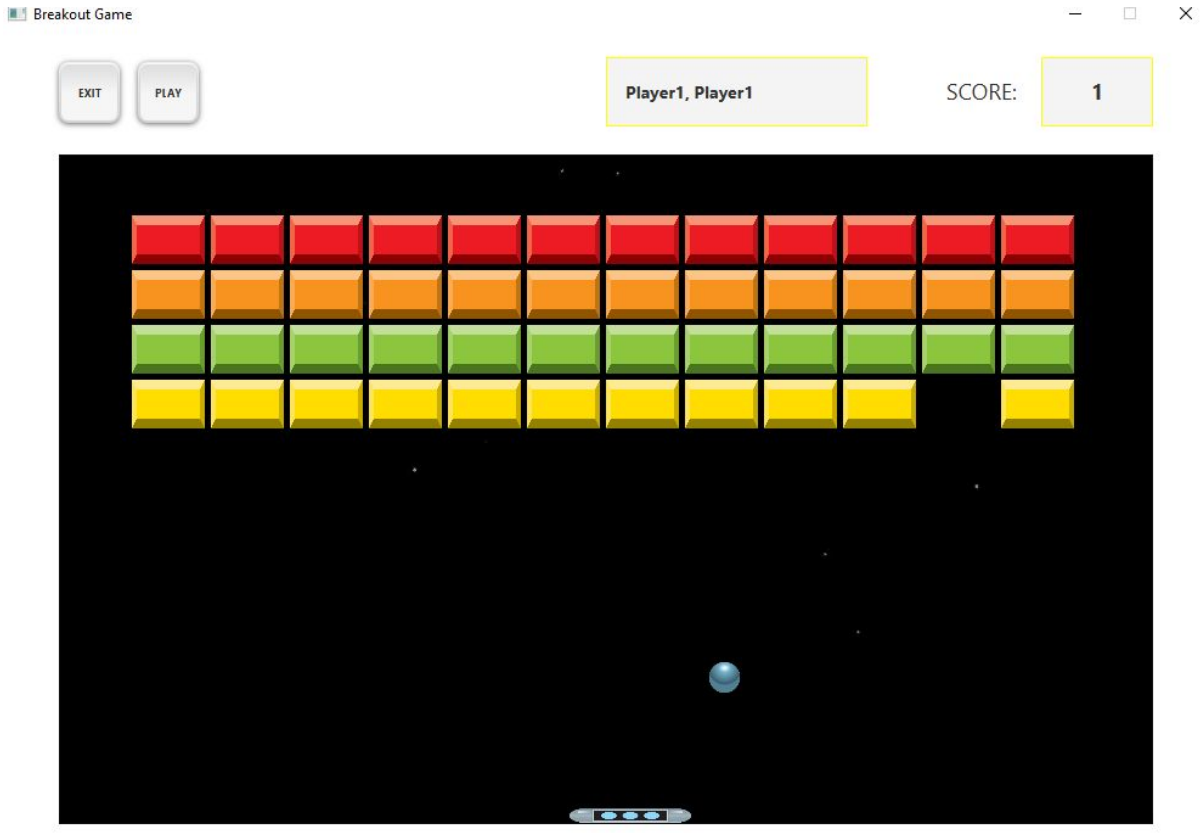


Figure 5. Frogger Game:

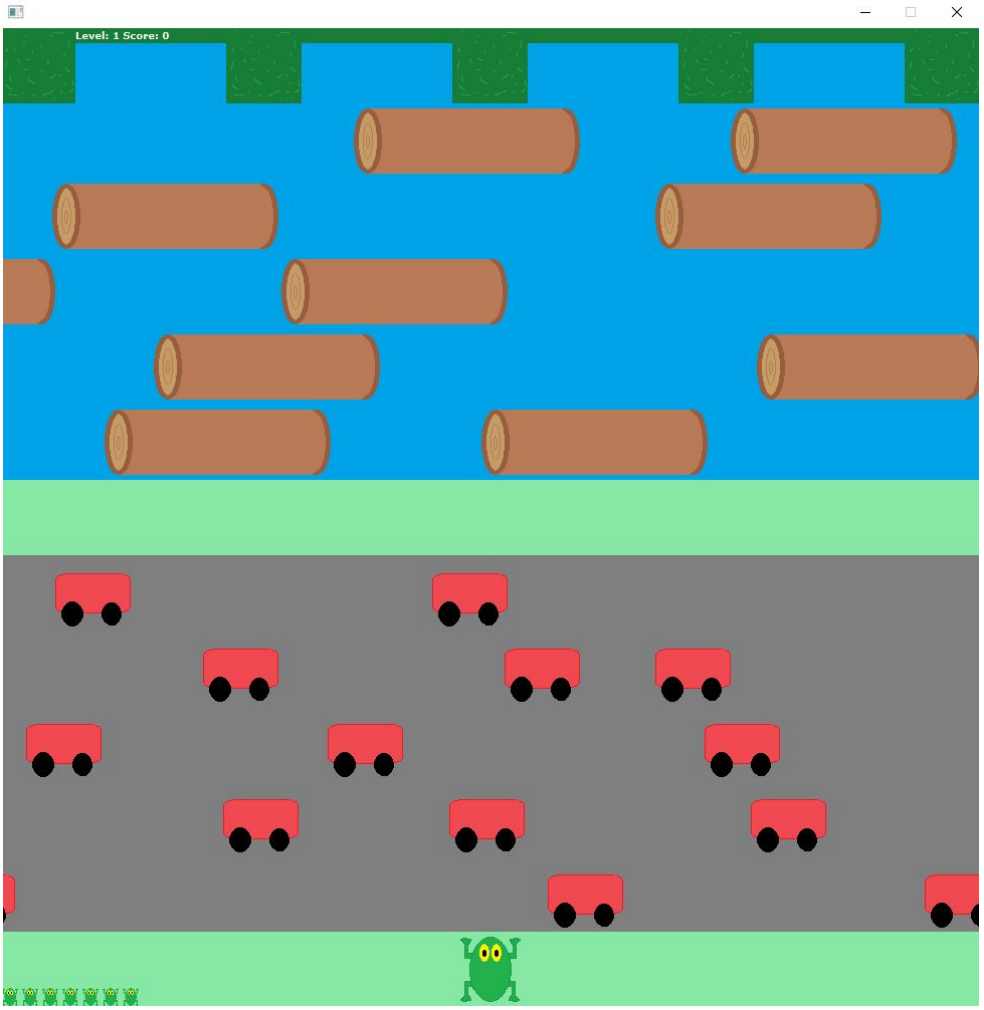


Figure 6. Pong Game:

Figure 7. Snake Menu:

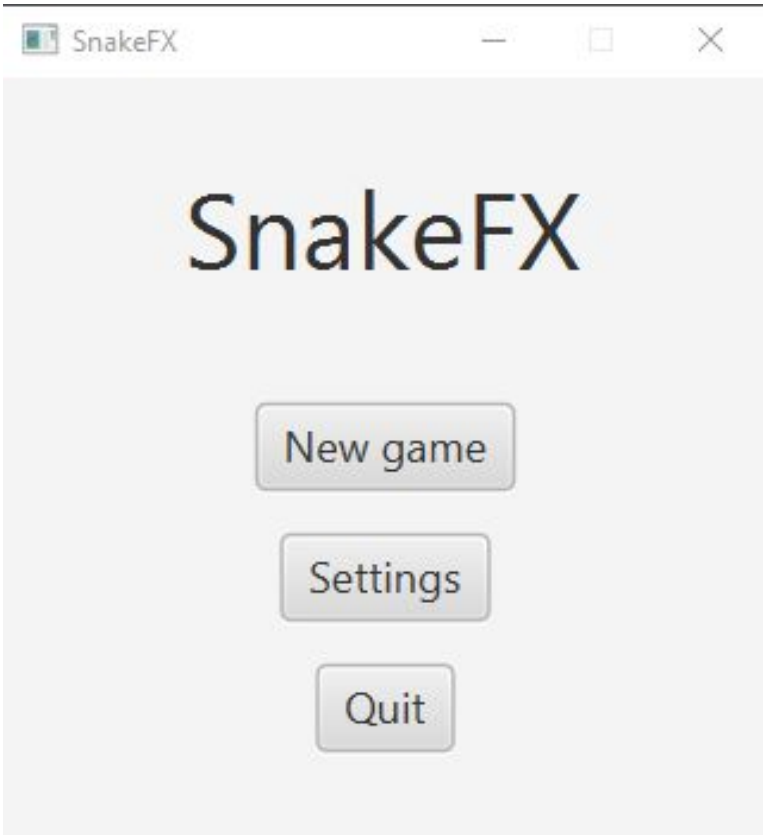


Figure 8. Snake Settings:

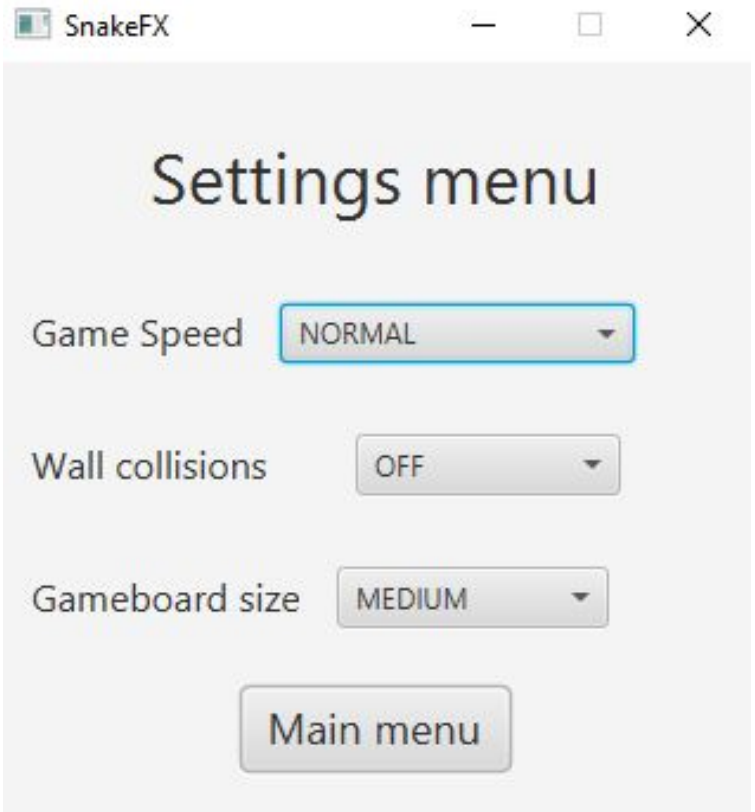


Figure 9. Snake Game:

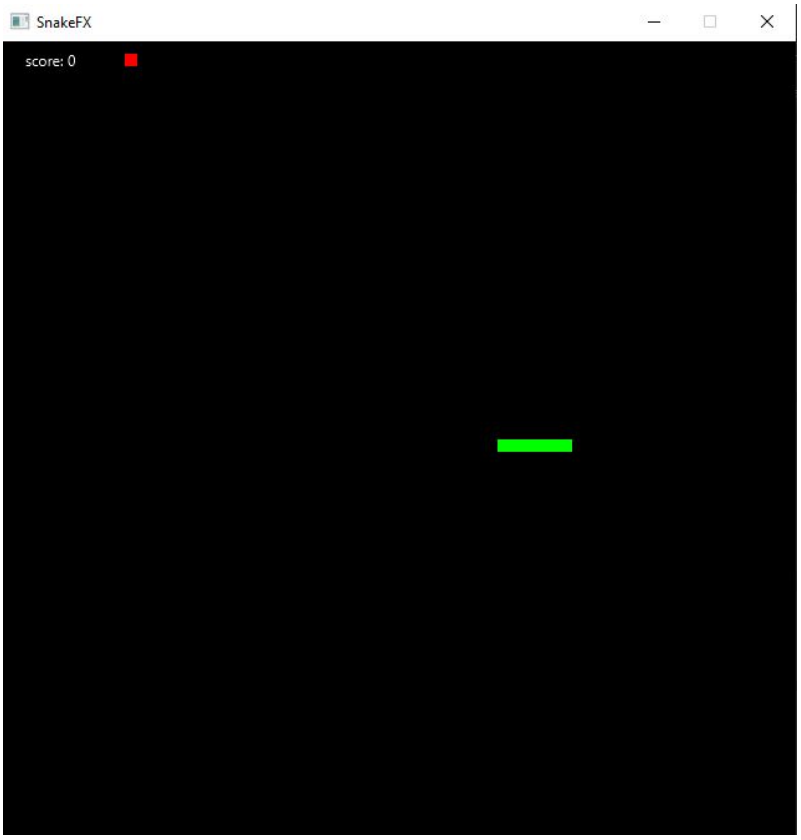


Figure 10. Space Invaders Game:

Figure 11. Space Invaders Pause:

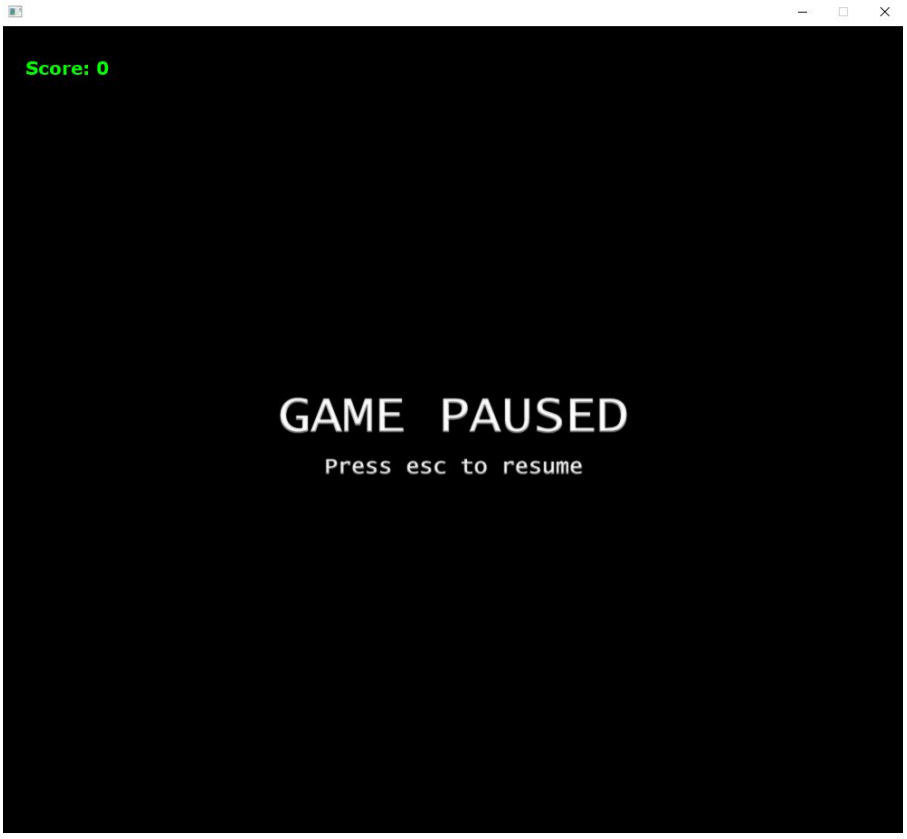
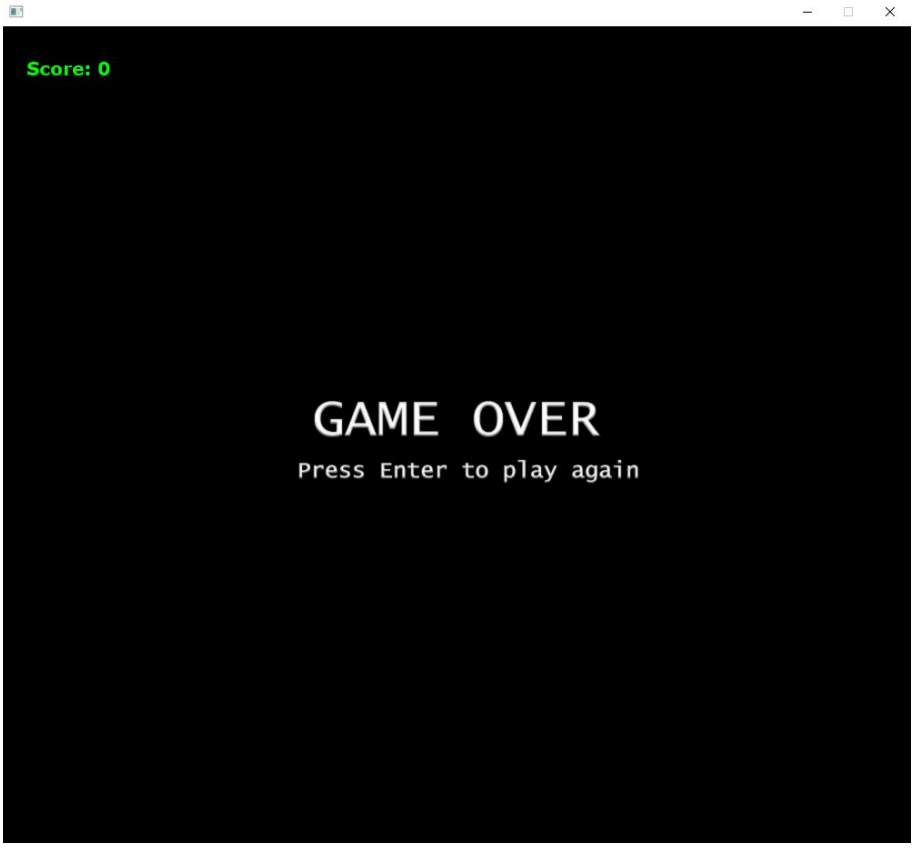


Figure 12. Space Invaders Game Over:



3. Domain model

3.1 Domain model

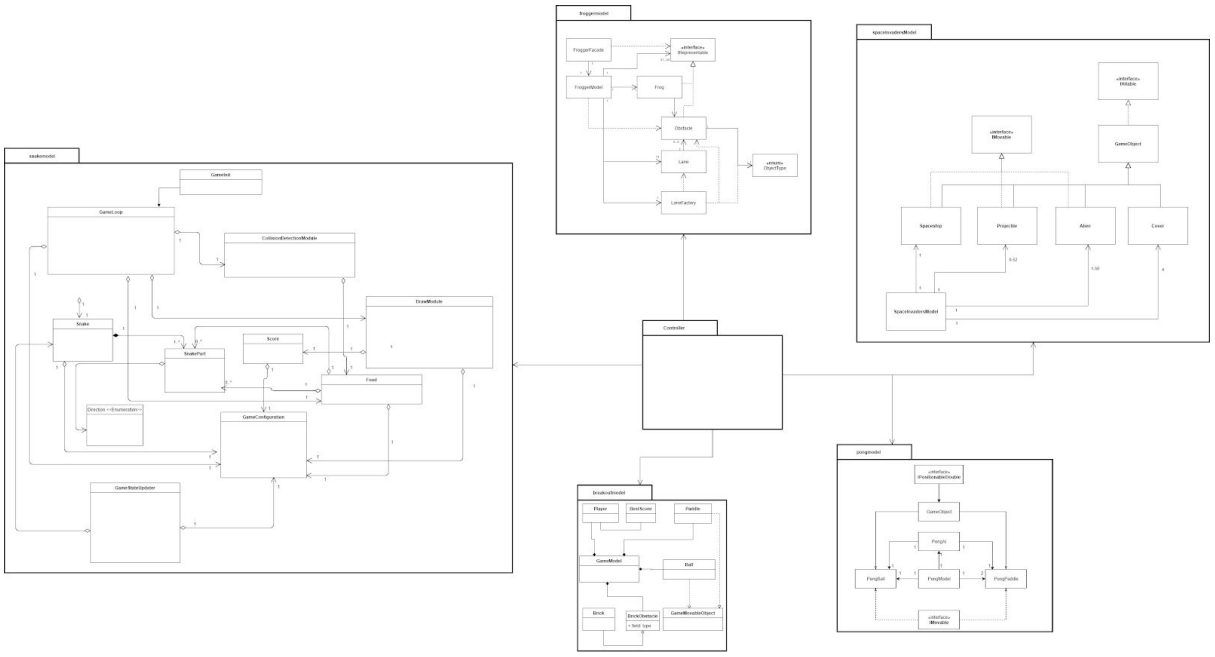


Figure 1. Domain Model[1]
(see attached pdf for more detailed view)

3.2 Class responsibilities

Explanation of responsibilities of classes in diagram.

Frogger:

- FroggerModel: Responsible for running the game.
- FroggerFacade: Responsible for representing the game of Frogger to other packages.
- Frog: Responsible for representing the player.
- Lane: Responsible for handling obstacles.
- Obstacle: Responsible for handling objects Frog needs to either avoid or in some way interact with.
- LaneFactory: Responsible for creating lanes for FroggerModel.
- ObstacleType: Responsible for representing the type of an Obstacle.
- IRepresentable: Responsible for handling objects that need visual representation.

Snake:

- CollisionDetectionModule: The class that detects a collision between the Snake, food and the wall.
- Direction: It represents the direction in which the player is moving.
- Food: Responsible for generating the food at random positions.
- GameStateUpdater: Responsible for the game state update.
- Score: Responsible to save and show the player's score.
- Snake: Responsible for representing the player.
- SnakePart: It represents the player parts.

Space Invaders

- SpaceinvadersModel: Model of the game, responsible for running and updating the game.
- Spaceship: Responsible for representing the player in the game.
- Alien class: Responsible for representing the enemies in the game.
- Projectile: Represents the projectiles fired by the player and the enemies
- GameObjectt: Abstract class representing an object in the game Alien, Projectile, Spaceship and Barrier all extend this class.
- IHitable: Interface which represents an object able to collide with other objects.
-
- IMovable: Interface which represents an object able to move.

Pong

- PongModel: Responsible for game logic and calling the update functions in PongBall and PongPaddle.
- PongBall: Responsible for representing the Pong Ball's position and moving the Ball in question.
- PongPaddle: Responsible for representing the Pong Paddles' position and moving the Paddle in question.
- PongAI: Responsible for making the right PongPaddle move up or down in response to the PongBall's movements.
- GameObject: Abstract class which has the properties for position and dimensions of both PongPaddle and PongBall.
- IMovable: Interface containing the Update function.
- IPositionable: Interface containing the getters of positional and dimensional variables.

Breakout

- GameModel : Class for Breakout controller. This class has some methods to detect collision between the ball and the wall, the ball and the paddle and the ball and bricks.
- Player: The class represents the player.
- bestScore: Responsible to save and show the player's score.
- Ball: The class has a move() method that moves the ball on the Board. If the ball hits the borders, the directions are changed accordingly.

- Paddle: This is the Paddle class. The paddle is controlled with left and right arrow keys. The paddle moves only in the horizontal direction, so we only update the x coordinate. If the ball misses the paddle and hits the bottom, we stop the game.
- Brick: This class creates object bricks for breakout. Yellow bricks earn one point each, green bricks earn two points, orange bricks earn three points and the top-level red bricks score four points each.
- BrickOnstacle: The class randomly creates and returns a brick obstacle

4 System architecture

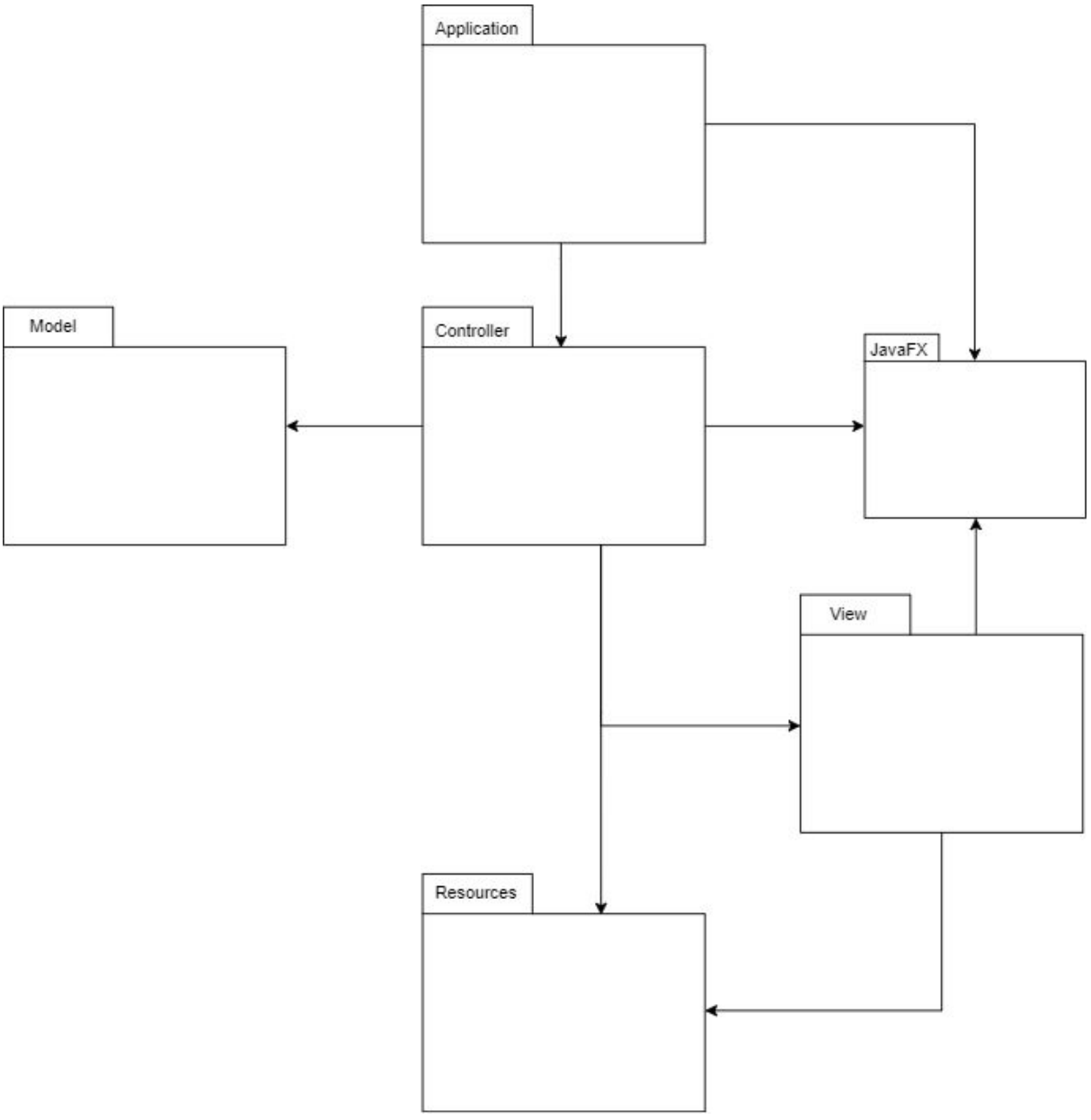


Figure 1. Package diagram[2]

The dependency between Controller and Resources comes from Maven[3] in combination with FXML-files. An FXML-file is most often considered a view but Maven does not allow non .java files to exist inside the source root meaning the FXML files cannot be included in the View package. If they were in the View package the Controller would not have any connection to the Resource package.

At a high level the program architecture was designed with design principles in mind. Most prominently were the “SOLID” principles considered. Coherency and independence were constantly taken into consideration when implementing components, this meant following the “High-Cohesion, Low-Coupling” principle which resulted in a code with less complexity and better readability.

The program was designed with extensibility and maintainability in mind, meaning following the well known “MVC” pattern to make sure that the model was without external dependencies so that it could potentially be reused.

To further increase re-usability of the code, classes were constantly grouped into logically coherent components and the structure was built trying to resemble reality.

4.1 Model

The model package holds classes representing the logic of the application, ergo, each individual game. Seeing as most of the games use continuous motion of some sort each game’s model is equipped with an update function that updates the state of the game.

Each game’s model also holds a list of IRepresentable, which represents an object that should have visual representation.

4.2 View

The view package holds classes that are used for the visual representation of elements from the model package and takes use of the library JavaFX[1] in order to accomplish this.

4.3 Controller

The controller package holds classes and interfaces that are used to manage views in view package and to handle user input such as key presses. In short, supply model with user input and relay information from model to view in order to visually represent the games. The controller package also takes use of JavaFX[1] as a timer for updating and drawing the games and as a way to relay which view MainApplication should be showing.

4.4 Application

The application package holds the class MainApplication which is responsible for starting the game by connecting it’s stage to the ViewNavigator and then telling the viewnavigator to load the start menu scene.

4.5 Resources

The resource package holds resources or data that can be accessed by the code. Images, icons, fonts and FXML files can all be found in the resources package.

5 System design

Design model
The design model can be found as an attached pdf file (see appendix).

Façade pattern:
The Facade pattern is used in package froggermodel within the model package where FroggerFacade acts as the interface that other packages can interact with.

Singleton pattern:
The Singleton pattern is used in the controller package, namely in ViewNavigator to prevent the creation of multiple instances of ViewNavigator which in turn would lead to the creation of multiple instances of the same controllers, models and views.

MVC(A):
MVC(A) is used to structure the entire project. The application starts the program, model holds game logic, controller relays info from model to view and provides user input information for model and view gives a visual representation of the program.
For more detail see chapters 2.1 - 2.4.

Module pattern:
Like MVC(A) the Module Pattern helps give structure to the project, dividing classes that are related in some way and organizing them into packages. This goes further than MVC as there are subpackages within each MVC-package. These have been divided by what game they are related to. Meaning all game logic classes associated with the game Frogger are located in the package froggermodel, SpaceInvaders in package spaceInvadersModel and so on. The one exception is the package common which contains interfaces and classes that multiple games use in order to avoid code duplication.

Factory pattern:
The Factory pattern is used in package froggermodel with the class LaneFactory whose sole purpose is to create different kinds of instances of Lane for FroggerModel.

6 Persistent data management

The project does not store any persistent data.
In regards to storage of resources the project has a maven[3] specific resource package outside of the source root where resources such as fxml-files, images and icons are stored.

7 Quality

In order to test the program the testing framework JUnit[4] was used. This made it easy to test functionality without a graphical interface initially. The tests are located in a seperate folder test where all games have their own testing file with test testing their functionality.

If any irregularities were found in the source code this would be brought up during weekly meetings in order for the responsible person to correct their mistake.

8 Peer review of Group 4 (done by Group 18, JGMOR)

Do the design and implementation follow design principles?

In the class Movement, the function addPoint both manipulates code and returns a value which goes against Command-query principle. Then, there are a lot of redundant dependencies within the Controller package as well as a lot of dependencies going both in and out of the package. This results in cohesion to be lower and coupling to be higher, opposite of what is preferred.
As far as the SOLID-principles go they are followed to a great extent. An abundance of getters and setters might be troublesome when it comes to Open-Closed principle. Could lead to unwanted dependencies and unexpected changes to variables.

Are design patterns used?

MVC, Observer and Module patterns are used.
Singleton and Facade patterns were attempted but weren't implemented properly.
No Singleton structure actually exists in the code.
Facade pattern fails due to dependencies going past the facade, rendering it superfluous.

Does the project use a consistent coding style?

Yes, code chunks are well-organised. The code style is consistent. The comments are found above the code. No redundant methods or spaces that pollute the styling.

Is the code easy to understand? Are proper names used?

It is fairly easy to understand what the code is supposed to do due to usage of java docs comments and clear documentation in RAD and SDD. Although the supposed function of the 5code is easy to understand it can sometimes be hard to grasp what the code is actually doing because of unclear method names. Some method/variable names were hard to understand what exactly they were for. An example would be the variable markedPoint in the class Game, we felt it was unclear whether this was for the spot that the player wanted to move, the selected piece or a piece that could be 'killed' by a move. This confusion was then amplified by the fact that there was another variable called clickedPoint which was used in conjunction with the previously mentioned markedPoint.

Does it have MVC structure, and is the model isolated from the other parts?

The project has a MVC structure which mostly follows the rules of a typical MVC structure. The model is isolated in its own package and from what we can see only reliant on the two interfaces Observer and TimerObserver. The first problem we found with the model and MVC structure as a whole are some unnecessary dependencies between the model and the controller. The controller has dependencies on over half of the classes included in the model which could easily be reduced. The second problem is that the controller includes

functionality which is supposed to be found in the view, namely a draw function. Instead of the controller telling the view to draw it just draws images itself.

Is the code documented?

Kind of, it's well commented, but it doesn't have proper documentation in the form of JavaDoc or similar.

Does the code use proper abstractions?

The command package is well written, as it is easy to remove/add new commands and incorporate them into the design without having to rewrite a lot of code.

Is the code well tested?

Yes. There are plenty of tests (especially for the model package) that cover the most essential parts of the code.

Is the design modular? Are there any unnecessary dependencies? Is the code reusable?Is it easy to maintain? Can we easily add/remove functionality?

The program is designed with such a concept of modularization, but we can also observe unnecessary redundant dependencies in the controller package. The code is largely reusable and it can reasonably be possible to expand the program's functionality.

Are there any security problems, are there any performance issues?

The program could not be run with Maven via command prompt. Upon further inspection the pom.xml appeared to be missing a plugin for javafx, meaning the project could not be run.

Can the design or code be improved? Are there better solutions?

- All classes and interfaces should be organized into packages. Observer and TimeObserver do not belong to a specific package.
- User Story #006 does not fulfill the functional criteria of adding time to the timer when the player finishes a turn even though the User Story is marked as implemented.
- There is not a User Story including anything in regards to losing the game when your timer runs out.
- User Story #014 mentions being able to move all pieces according to the rules of chess, however the moves *castling*, *en passant* and *Queening* cannot be performed. Admittedly the User Story is named "Apply Basic Rules" so it might be clever to add another User Story for these moves.
- The arrows for the design model might need looking over. For instance in Figure 4 of the SDD the usage dependency arrow from MenuController to ChessController should be an association arrow.
- The initial sketch shows chess coordinates by the side of the board however they are not implemented and no User Story for implementation exists.

- The library java.awt is used in Board, Game, Movement, Ply. There is not however any mention of java.awt in the pom.xml-file meaning there are not any dependencies on the library. In addition to this javafx is also used which fills all the same responsibilities. Therefore, removing all dependencies on java.awt and focusing on using just the one library is better and more efficient.
- There is no mention of the libraries used in the reference section of neither the RAD nor the SDD.
- ChessTimer might need some looking over. There are two instances of the class yet, by reading the code and functions of the class, one gets the impression that there should only be one.
- There are a few inconsistencies when it comes to the implementation of the design pattern MVC(A). A lot of the responsibility that View should have is instead put on the Controller for instance the function drawPieces().

9 References

[1] Diagrams.net. [Online]. Available: https://app.diagrams.net/ Accessed: 2020-10-22.
[2] JavaFX. [Online]. Available: https://openjfx.io/, Accessed on: 2020-10-23.
[3] Maven. “Welcome to Apache Maven”. [Online]. Available: https://maven.apache.org/#, Accessed on: 2020-10-23.
[4] JUnit, “JUnit - About”, 2020. [Online]. Available: https://junit.org/junit5/, Accessed on: 2020-10-23.

10 Appendix

- A pdf file named “DomainModel” can be found in the same folder as this file.
- A pdf file named “DesignModel” can be found in the same folder as this file.