CS 319 - Object-Oriented Software Engineering

Final Report

Pong-X

Section 2-Group 2

Mehmet Çağrı Kaymak

Abdullah Alperen

Mehmet Fatih Çağıl

Doğukan Ömer Gür

Table of Contents

1. Introduction

Pong is one of the first popular games of the video game industry. Basically, it is a two dimensional tennis game. Aim of the game is reaching a certain point before the opponent; points are earned when opponent cannot return the ball. There are two paddles in the opposite ends of the screen that controlled by two different users. Different than the original game, we are planning to add brick as obstacles and some features as power-ups; with the power-ups movement speed or size of the paddle will be changed. (Or different features will be added to paddle). Same power-up functionality will be available for the ball. As a last difference, we are going to add different game fields. Classic pong game has a rectangular game field, we are going to add circular and other different game fields. Our game will be compatible with PCs and will be a game for 2 players.

A very basic pong game can be found at http://www.ponggame.org

2. Requirement Analysis

2.1. Overview

When the game opened, a menu will be seen in the screen. This menu is the main menu. There will be 5 different options as “Play Game with AI”, “Play Game with 2 Players”, “Options”, “Credits” and “Quit”. When “Quit” is selected the game will be closed as the name implies. At the “Credits” menu, information about game will be showed. In the “Options” menu, user can be able to change some features in the game. There will be a sound switch which allows user to enable or disable the gameplay music. User can be change the background image from this menu. Any image can be used as background image. Users can change the control buttons if the sticks in the main menu, buttons can be changed with other keyboard buttons if they are not used. When user selects the “Single Player”, screen will show up. In this screen, user can be able to change the difficulty level of AI, type of game field and user will be select one of the 6 brick presets and the score limit of the game. Then the game will start. “Multiplayer” is almost identical to “Single Player” menu,

but there is not difficulty level for AI since AI is not involved to game. Purpose of the game is forcing the opponent to fail the return the ball. 2 different users are going to control different sticks; they will try to hit the ball until one of the fails. When one of them fails, other player will gain score. When one of the players reach the limit, game will be over and that player will be decided as the winner.

During the game there are going to be positive powers and negative powers. When a player sends the ball to a power, that user will gain that power. These powers will change the features of the sticks and the ball. Also there will be brick on the game field that selected by user before starting the game.

When one of the players presses “P” from keyboard, game will be paused. In the pause menu, there are going be 3 different selections; “resume” which continues the game, “restart” which starts over the game and “back to main menu” which goes back to main menu.

Players are going to control the sticks with keyboard, one of the sticks will be controlled with “W” and “S”, other one with “O” and “L”. These buttons can be changed in the options menu. When one of the players gain score, powers will be deleted.

2.1.1 Powers

Powers will show up randomly and when the ball collides with a power the last stick that hit the ball is going to has the power.

2.1.1.1 List of Positive Powers

Long Stick: Length of the stick will be increased when this power is gained, so that player will be able to hit the ball more easily. Having another Long Stick power while one is active will increase the duration of the power. If the user has Short Stick power this will neutralize that power.

Fast Stick: Movement speed of the stick will be increased when this power is gained. Having another Fast Stick power while one is active will increase the duration of the power. If the user has Slow Stick power this will neutralize that power.

Slow Ball: This power will decrease the speed of the ball. Default speed of the ball can be only decrease 1 level between 2 scores, this means that having same power will increase the duration of the power.

Big Ball: This power will increase the size of the ball. Having more than 1 power will increase the duration of the power.

2.1.1.2 List of Negative Powers

Short Stick: Length of the stick will be decreased when this power is gained. Having another Short Stick power while one is active will increase the duration of the power. If the user has Long Stick power this will neutralize that power.

Slow Stick: Movement speed of the stick will be decreased when this power is gained. Having another Slow Stick power while one is active will increase the duration of the power. If the user has Fast Stick power this will neutralize that power.

Fast Ball: This power will increase the speed of the ball. Default speed of the ball can be increased 3 levels. Having more than 3 powers between 2 scores will increase the duration of the power.

Small Ball: This power will decrease the size of the ball. Having more than 1 power will increase the duration of the power.

2.1.2 Bricks

Regular Brick: This type of brick will act like an ordinary obstacle, when the ball hits a regular brick it will bounce as normal. This type of brick will be broken after one hit.

Figure 1: Regular Brick

Fast Brick: When the ball hits a fast brick, speed of the ball will be increase for a small amount of time. This type of brick will be broken after one hit.

Figure 2: Fast Brick

Strong Brick: This type of brick is same as Regular Brick but will be broken after 3 hits instead of 1 hit.

Figure 3: Strong Brick

Negative Brick: When the ball hits a negative brick, it will bounce to where it came. This brick will be broken after one hit.

Figure 4: Negative Brick

2.1.3 Game Field

Game Field is the field that action is happening. When the ball hits the left or right sides of the rectangular game play, the player on the opposite side will gain a score. If the game field is circular, player that hit the ball out of the game field will gain a score.

2.1.4 Ball

Ball is the one of the two main components of the game. Aim of the game depends on the ball, players cannot control the ball directly, they have to use sticks to control the ball. If the ball goes to back of the game field, there is a score in the game.

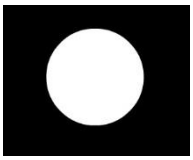


Figure 5: Ball

2.1.5 Stick

Sticks are the only components that players can control. Sticks can be moved vertically (with respect to game field) with keyboard buttons. By having powers features of sticks will be change.

Figure 6: Stick

2.2. Functional Requirements

User(s) is going to control the stick with keyboard buttons, which can be changeable at the options menu.

In the options menu user can enable or disable the game music, change the background image and change the keyboard input buttons if desired.

Game can be paused during gameplay with button “P” and user(s) can continue to game or go back to main menu. When user select to go back to main menu when game is paused, all the progress will be lost.

Powers will change the specific features of the game and powers are going to be reset when there is a score.

Right before starting to game, user can choose game field type, brick set, AI difficulty and score limit. When the score limit is reached game will be over.

2.3. Non-Functional Requirements

The system should response quickly at each step

User can be able to start the game with 2 mouse clicks.

System requirements are going to be kept as low as possible because we do not want it cost too much in terms of performance, so that user can have a smooth game experience.

User interface of the game will be plain and simple for ease of use.

The game will be written carefully in terms of reusability and extendibility so that it can be modified easily on future.

The project should be licensed under Apache 2.0 open-source license, which is one of the most widely used licenses by open-source projects.

Any system that has Java libraries installed will be able to run the game.

2.4. Constraints

The game is going to be written in Java.

The system must be a desktop application.

The game will support English since English is widely known around the world.

2.5. Scenario

Scenario Name	HitBallByUser
Participating Actor Instances	Fatih: User
Flow of Events	Fatih wants to hit the ball and bounce it back to protect his goal.
	Fatih presses the movement key that moves the user's bar to the right
	System responds to the input and moves the bar to the right, as long as Fatih presses on the key.
	After Fatih presses the right key and moves the bar, the ball and the bar collide.
	System detects the collision and changes the ball's direction accordingly.
	Ball bounces back within the rules of the physical collision and drifts away from Fatih's goal, system draws the final states of the objects after collision.

Scenario Name	ChangePreferences
Participating Actor Instances	Fatih: User
Flow of Events	Fatih wants to change the settings of his profile from main menu. He goes to options menu to make the necessary changes
	Fatih mutes the sound to eliminate any in-game sound.
	He also changes the background and chooses one of the options provided on the screen
	To change the move up control of first player, he click and change the current control key.
	Fatih saves the changes before leaving the Options Screen to apply the changes he made on preferences.
	System updates the preferences associated with Fatih's profile and it makes them available in the next game Fatih plays.

Scenario Name	BreakBrick
Participating Actor Instances	Fatih: User
Flow of Events	Fatih catches the ball with his stick by moving it.
	When he hits the ball, it drifts towards the brick with each periodical update of the system and reaches the brick within a certain number of update loop.
	System detects that collision occurs between the puck and the respective breakable brick.
	After collision ball breaks the brick that it get into contact with.
	System handles the collision according to elastic collision principles and ball attains its new vector keeps on moving.

2.6. *Use Case Models*

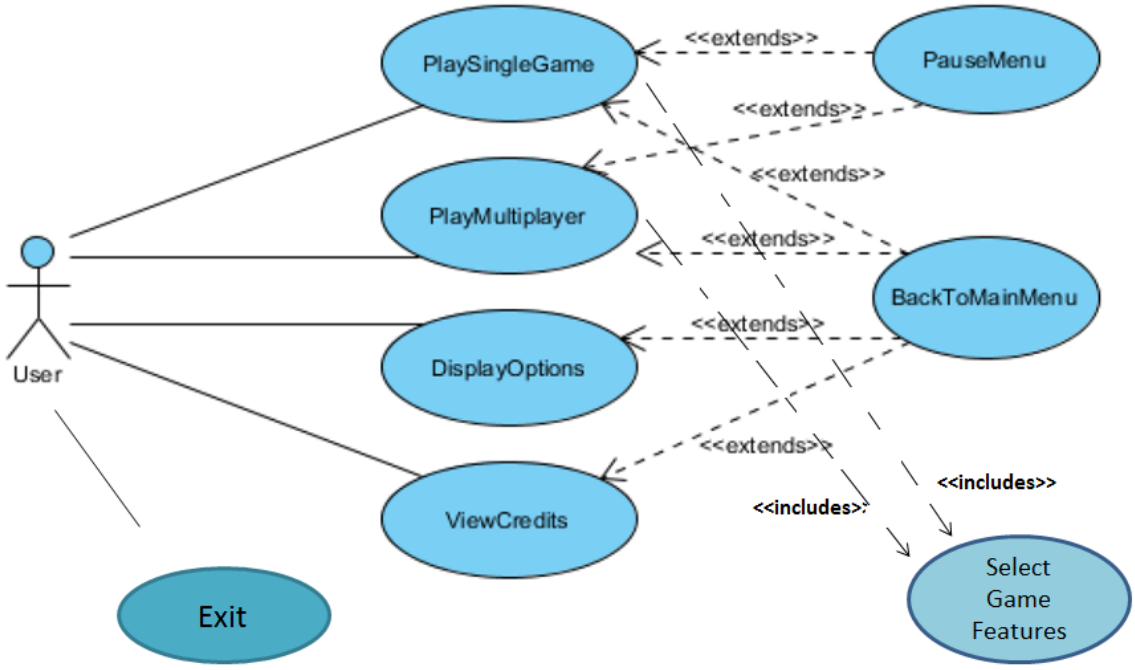


Figure 7: Use Case Model

Scenario Name	PlaySingleGame
Participating Actor Instances	Initiated by User
Flow of Events	1. System shows User the Main Menu
	2. User clicks on Single Player option
	3. System shows the available maps, choose friction rate and difficulty of artificial intelligence, select the brick type by clicking on them with the mouse.
	4. User clicks on play button.
	5. System starts the game. The game is launched with map specifications. The bricks on the map are determined by the preferences. Game starts with 0 to 0 score.
	6. System launches the ball from its starting point with a fixed starting velocity and accelerates using its algorithms.
	7. User presses up and/or down arrow keys to play the game. If the user has assigned another keys (from the Options Menu), these keys are used to play the game.

	8. System responds to the respective arrow key by moving the user stick up or down. When the ball and a stick collides system uses its algorithms to bounce the ball. The stick movement at the instance of collision affects the ball's movement by either increasing or decreasing its movement on the x-axis and y-axis by determining the total speed depending on friction rate.
	11. If the User couldn't block the bar with their bar, it means that the AI scores a goal. Likewise, User scores a goal when the AI can't block the ball with its stick.
	12. System increases user's or AI's score by one and initiates the game from (5). If score of one of them reaches 5, the game terminates and shows the user game score.
Entry Condition	**User launches the game.**
Exit Condition	**User clicks return to main menu button**
Quality Requirements	**System should response in 1/10 seconds**

Scenario Name	DisplayOptions
Participating Actor Instances	Initiated by User
Flow of Events	1. System brings Main Menu screen when user runs the game.
	2. User clicks on Options from the Main Menu.
	3. System shows Options screen.
	4. User selects the background image.
	5. User will be able to change background image by selecting a picture from the computer and uploading it to the system.
	6. User enable or disable the game sound.
	7. When the sound is enabled, the system plays 8-bit default music of the game in background.
	8. User clicks Apply button and system updates the display options according to the preferences.
	9. User clicks Back button

	10. System opens the Main Menu screen.
Entry Condition	**User launches the game.**
Exit Condition	**User clicks return to main menu button**
Quality Requirements	**System should response in 1/10 seconds**

Scenario Name	PauseorExitGame
Participating Actor Instances	Initiated by User
Flow of Events	1. System brings Main Menu screen when user runs the game.
	2. User clicks Single Player or Multiplayer and initiates the preferences of the game.
	3. User starts to play game.
	4. User selects the "p" key on keyboard in the game screen.
	5. System pauses the game. Two options appear on the screen: Resume and Main Menu buttons.
	6. a. User clicks on Resume
	7. System returns back to game screen and stars the game with current situation.
	6. b. User clicks on Main Menu
	7. System loses the game data and returns to Main Menu screen.
	8. User clicks on Exit button and closes the game.
Entry Condition	**This use case extends the PlayGame (SinglePlayerGame or MultiplayerGame) use case. It is initiated whenever the game is stopped due to a user's interference.**
Exit Condition	**User clicks return to main menu button**
Quality Requirements	**System should response in 1/10 seconds**

2.7. *User Interface*

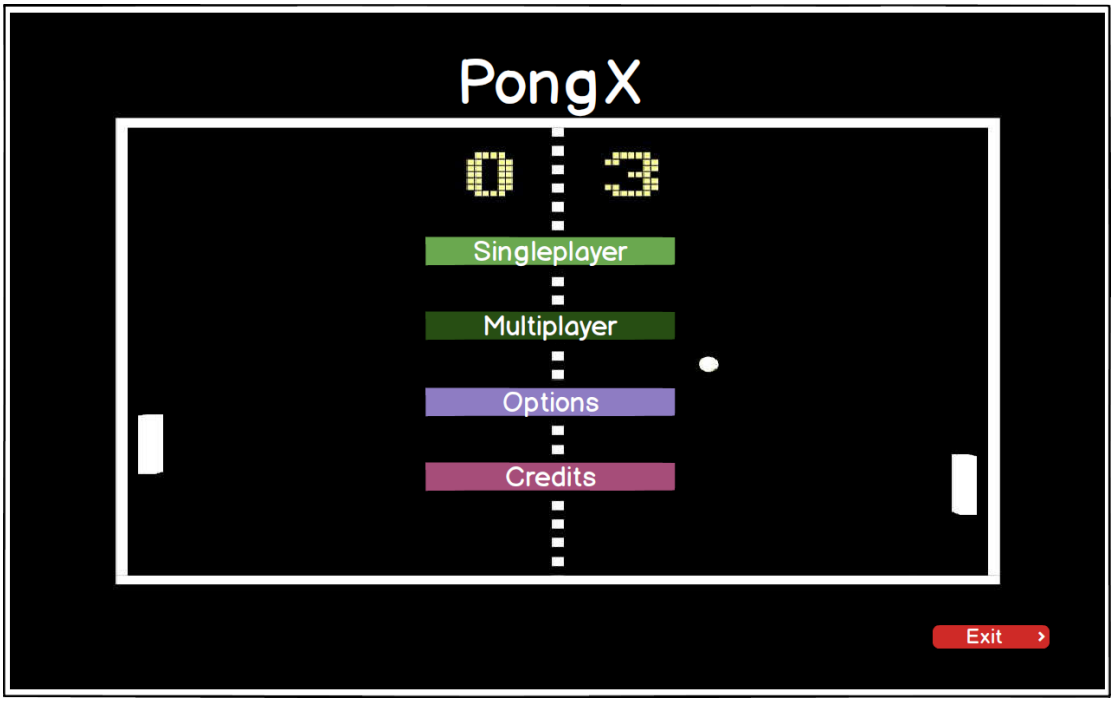


Figure 8: Main Menu

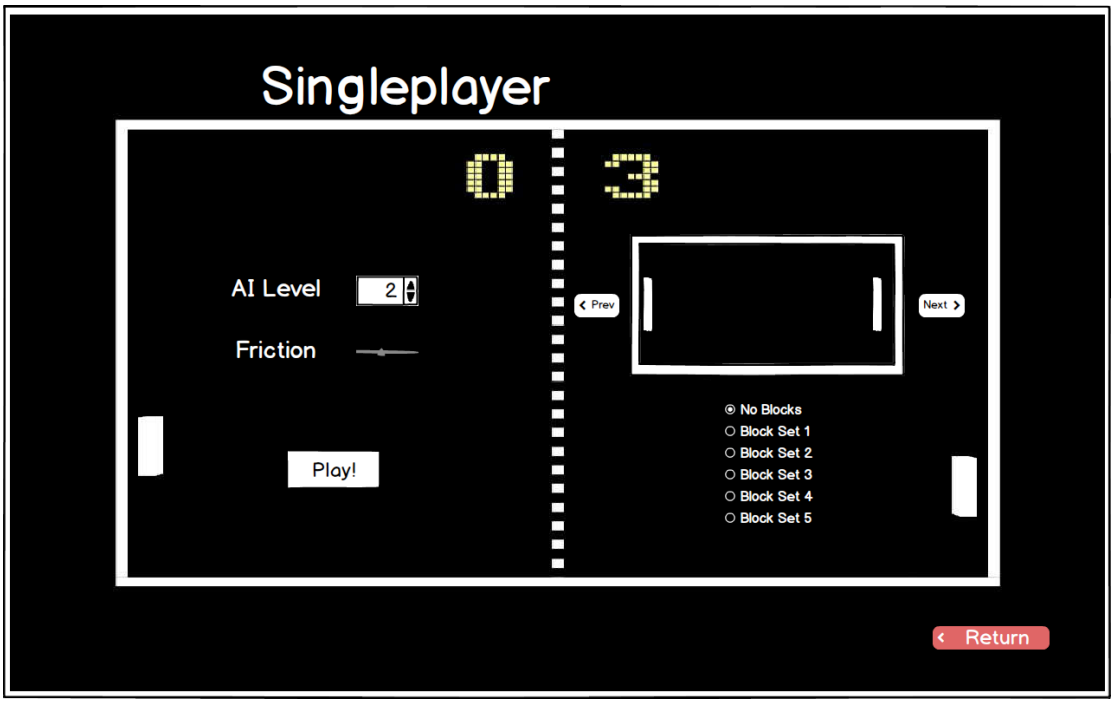


Figure 9: Singleplayer Map Selection Screen

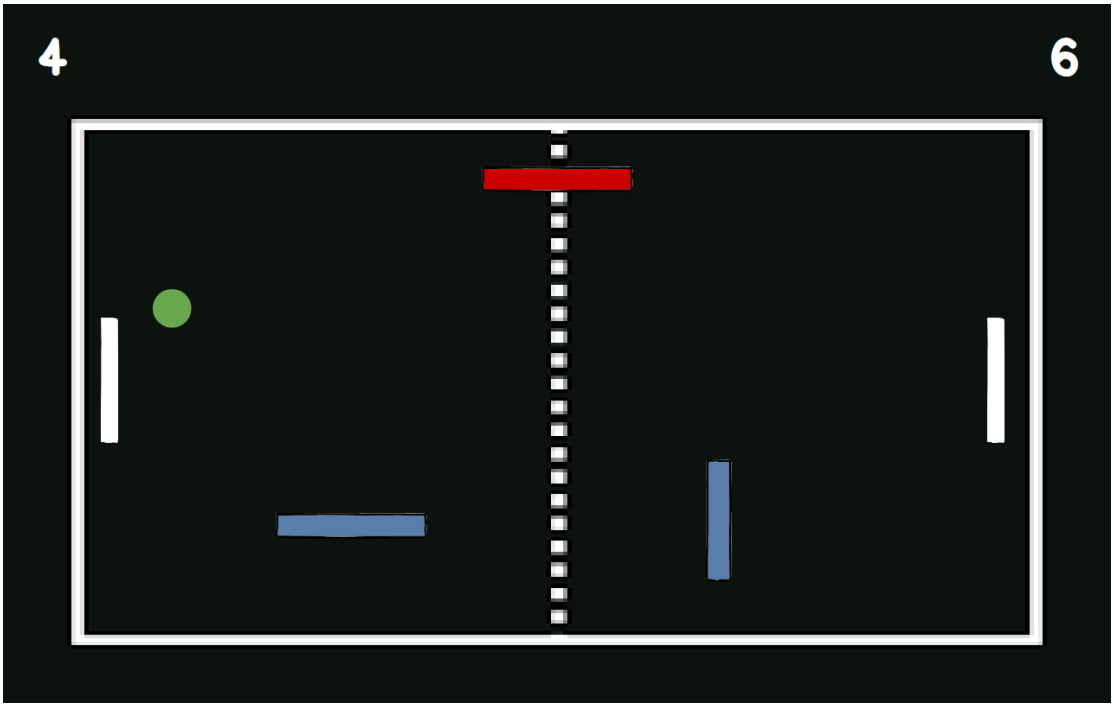


Figure 10: In Game Screen

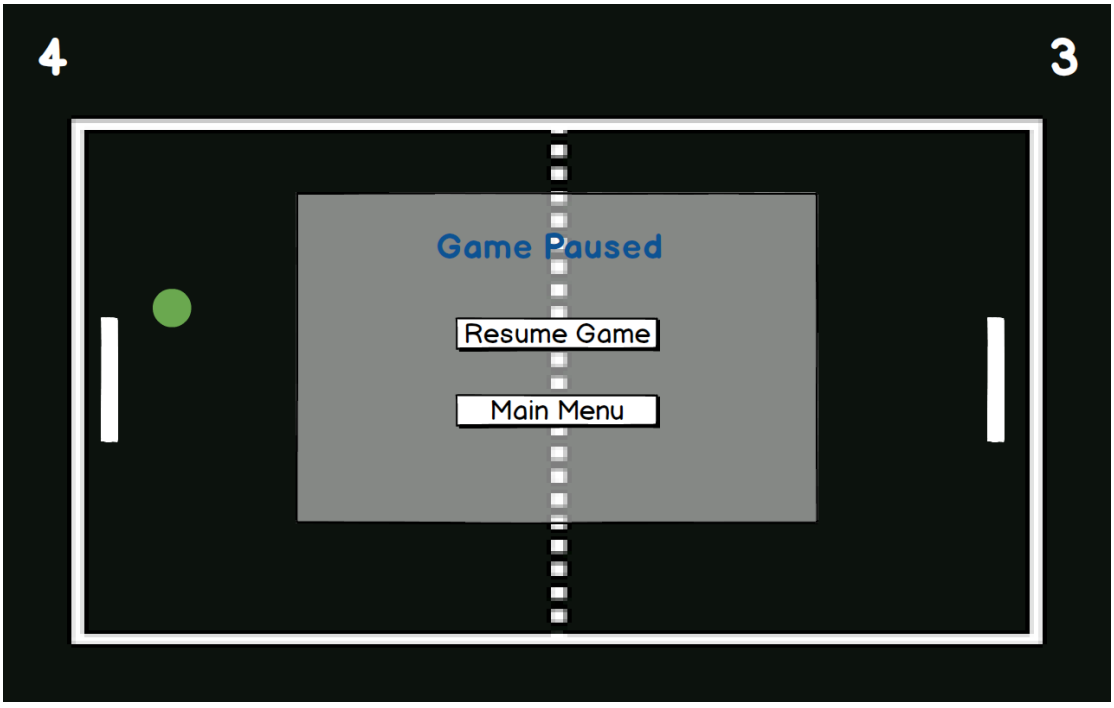


Figure 11: Pause Screen

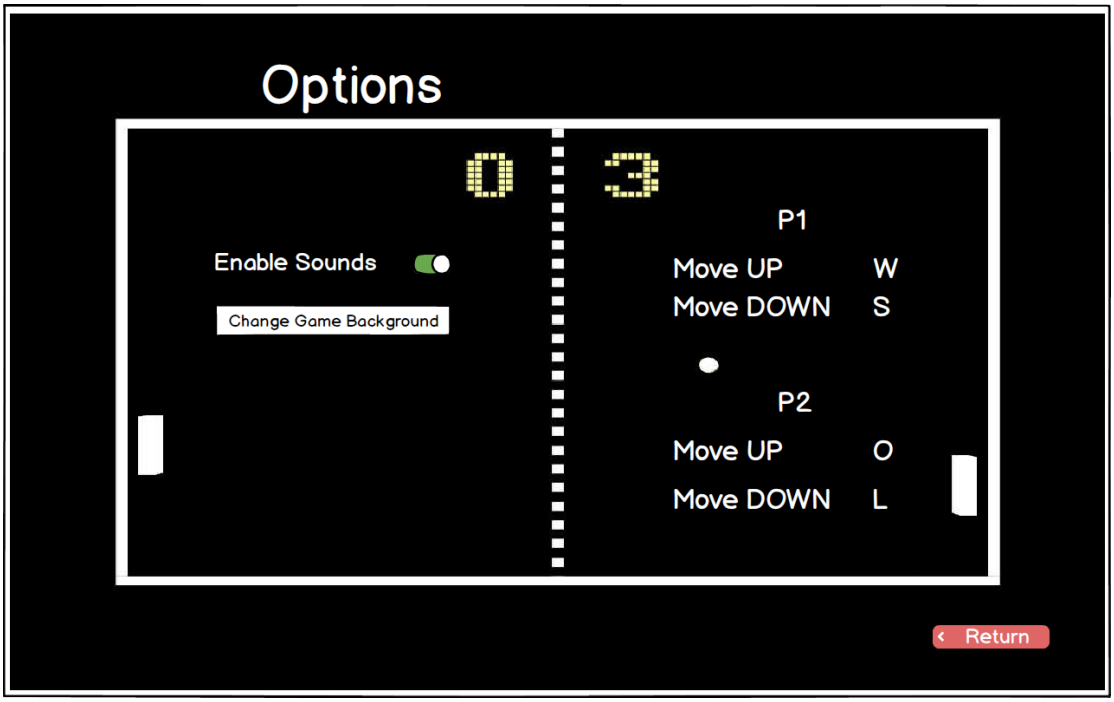


Figure 12: Options Screen

3. Analysis

3.1. Object Model

3.1.1. Domain Lexicon

In this section we will provide some domain information about important terms for designing the project.

Game Field: This entity is a reflection of a field that game is happening. It keeps the sticks, ball, bricks and powers wrapped up. There are different game fields, their only difference are shapes. We have a rectangular game field, circular game field and rounded rectangular game field. This is an environment for other entities to occur.

Powers: There are different types of powers which can affect the process of the game by changing features of the sticks and ball. This is the main difference between our game and classical Pong game.

Stick: This entity is the only object that user can control. User will move it vertically with respect to game field to hit the ball.

Ball: Ball is the other entity object that can be move but not controlled by user. It is going to move according to Newtonian physic laws.

Brick: This is another entity object that exists in the game field. Bricks are not movable objects but can be breakable with ball hits.

3.1.2. Class Diagram

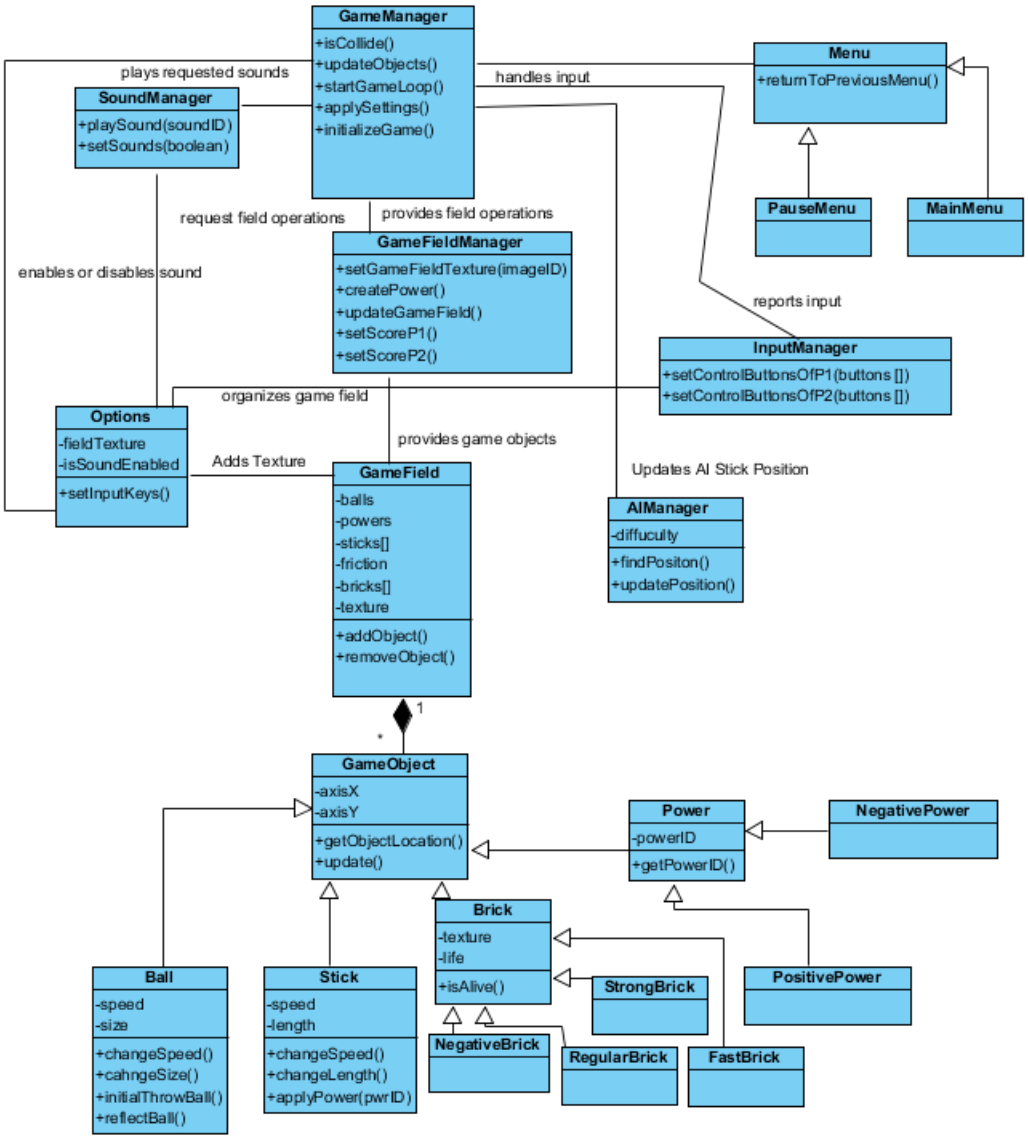


Figure 12: Class Diagram

3.2. Dynamic Models

3.2.1. State Chart & Activity Diagram

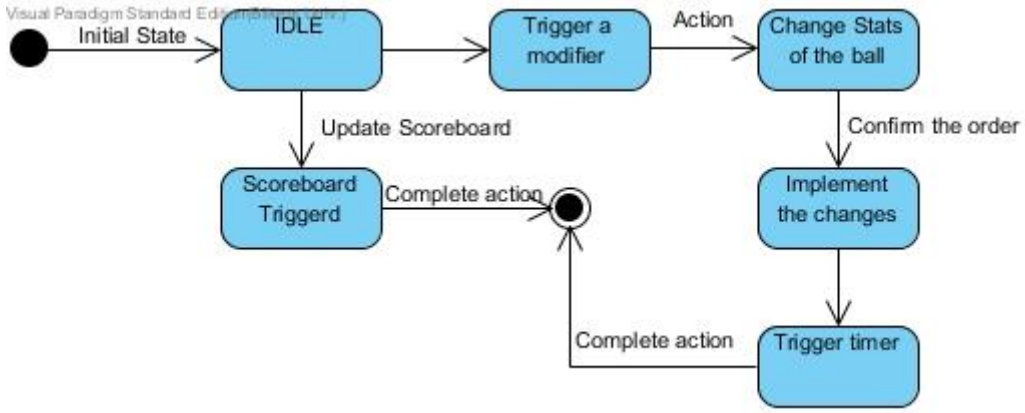


Figure 13: Statechart Diagram for Ball Class

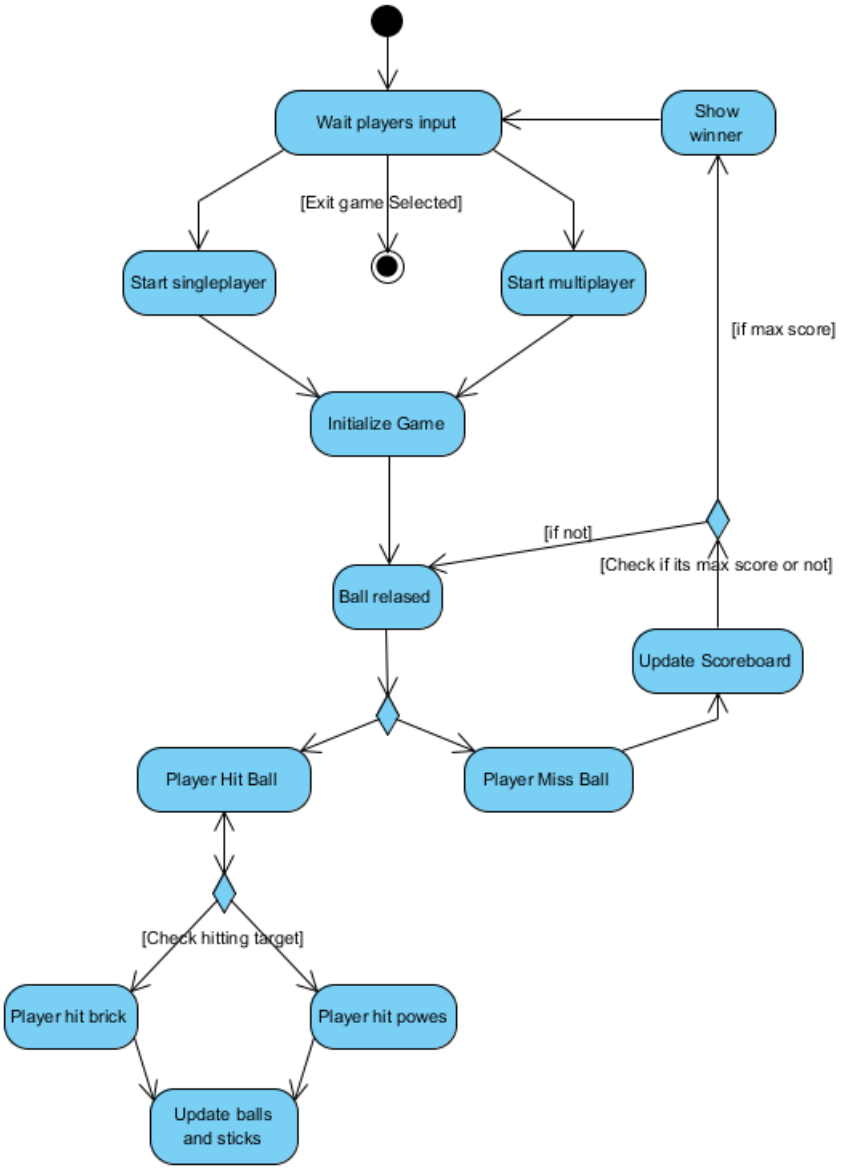


Figure 14: Activity Diagram

3.2.2. Sequence Diagrams

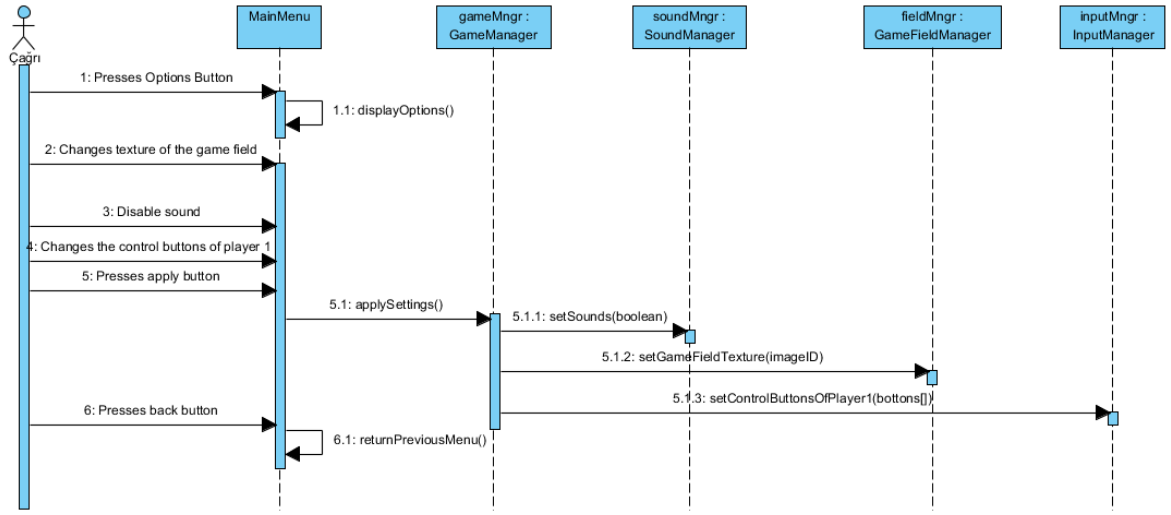


Figure 15: Change settings (Sequence Diagram)

This diagram shows which interactions happen when the options are changed. After user makes specific changes at the options menu, he clicks the appySettings button and it calls applySettings() method of GameManager, then game manager handles applying these settings with using proper methods of other managers.

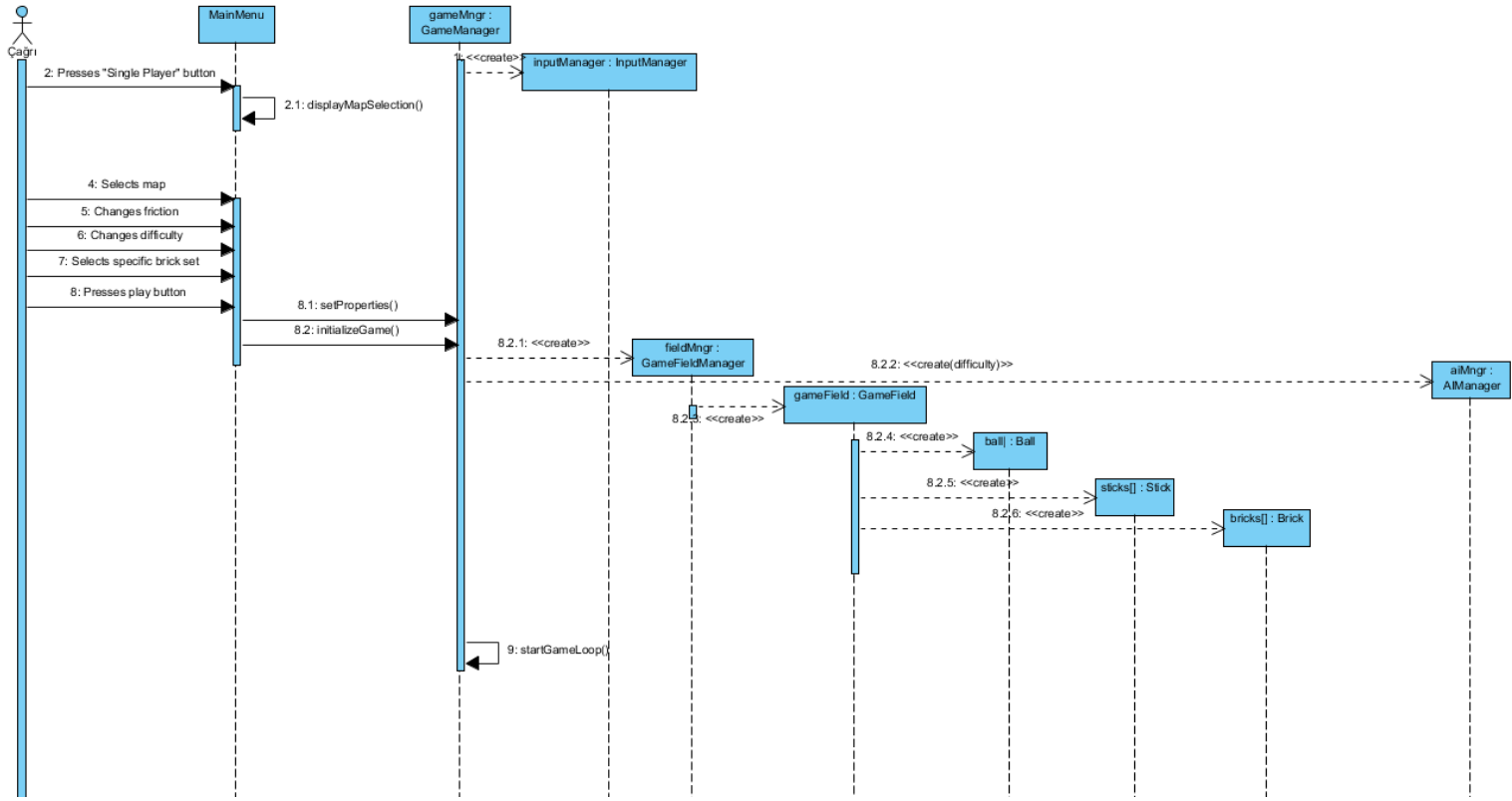


Figure 16: Start Singleplayer game (Sequence Diagram)

Figure 16 shows which interactions happen while starting a single player game. When user presses “Single Player” button, proper menu is shown. After that user makes initial map settings and presses “Play” button. Then GameManagement class takes these settings and creates other managers according to that. GameFieldManager creates GameField object and this object creates rest of the game entities. After all initial creations, game loop starts.

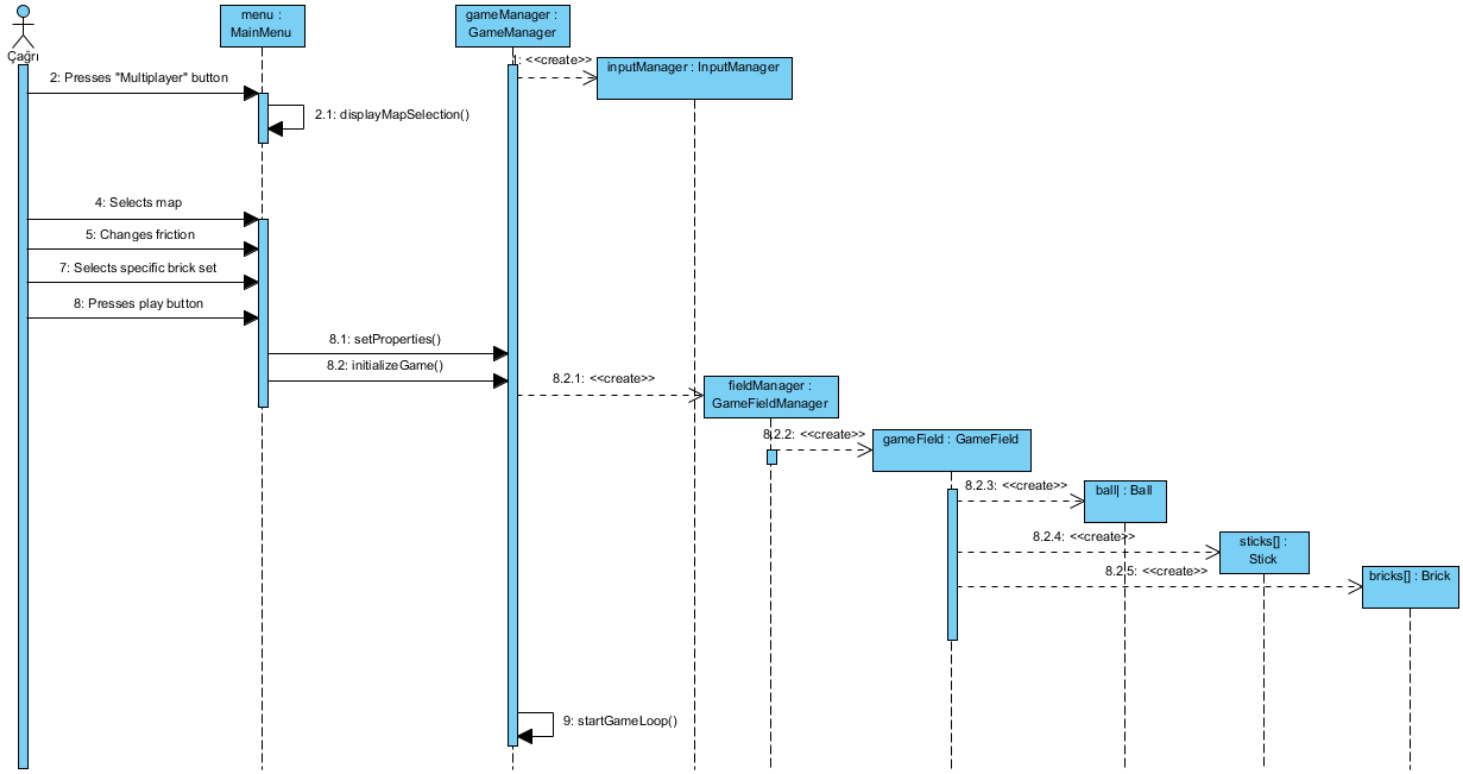


Figure 17: Start Multiplayer game (Sequence Diagram)

Figure 17 shows which interactions happen while starting a multiplayer game. When user presses “MultiPlayer” button, proper menu is shown. After that user makes initial map settings and presses “Play” button. The rest is similar with “Single Player” part. Only difference is that this time there is no AIManager, InputManager handles with both sticks.

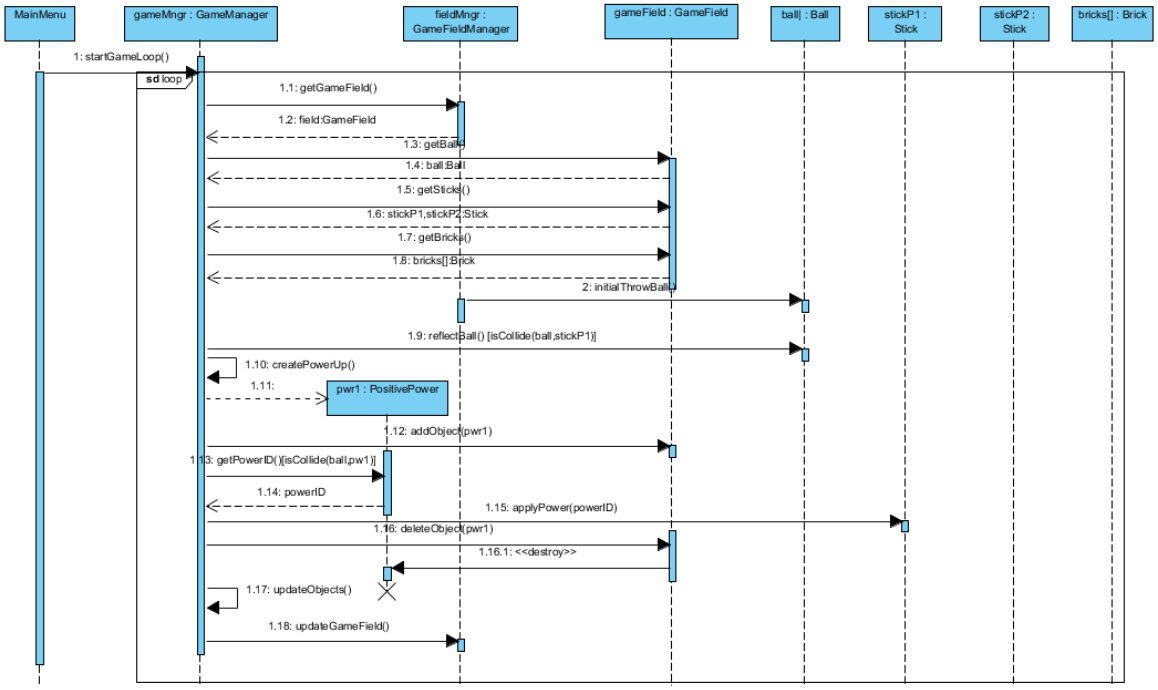


Figure 18: Get Positive Power (Sequence Diagram)

Figure 18 briefly shows what happen when the game enters game loop and the process of getting positive power. When game starts, GameFieldManager gives an initial speed to ball object. When ball hits a stick, it reflects from the stick. Power up objects are created randomly during the loop. When the ball hits a power up, applyPower(ID) method of certain stick is called. Then the power up is destroyed. As a last thing, objects and game field is updated.

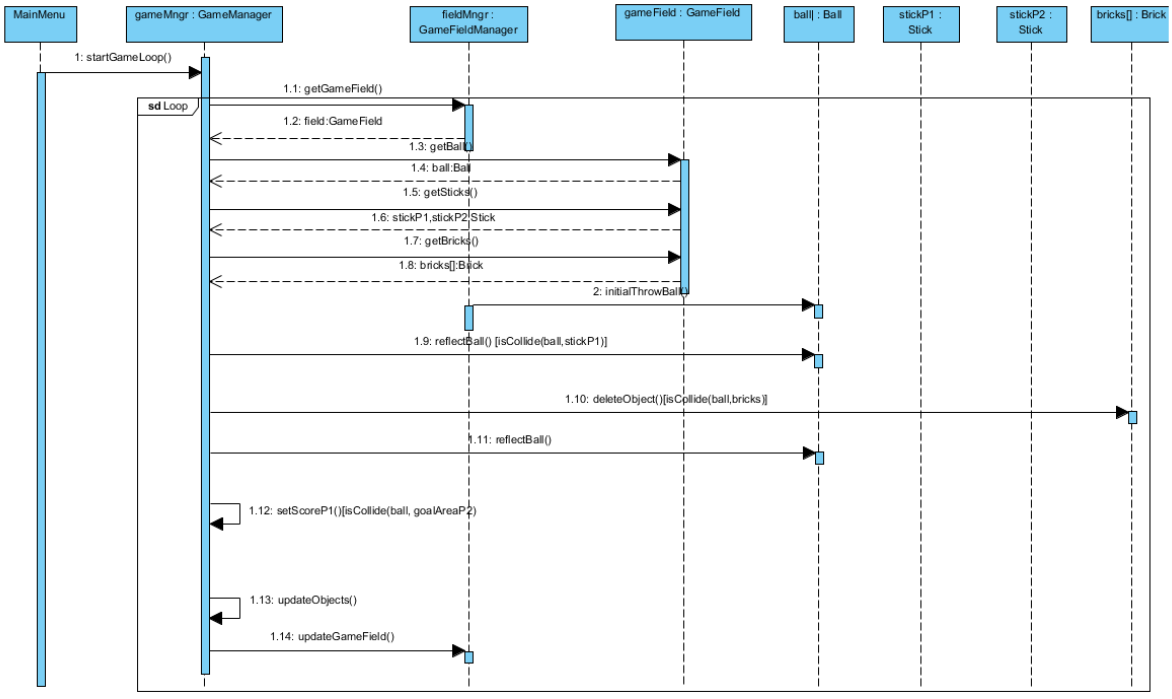


Figure 19: Hitting brick and scoring (Sequence Diagram)

Figure 19 shows which interactions happen when ball hits a brick and a player get score. When the ball hits a brick, the ball reflects from the brick with reflectBall() method. Also, if the ball enters a goal area of the player, other players gets score.

4.Design

Pong X is a two dimensional platform game. High quality graphics and visuality are not the main design goals of the system. Pong X is designed as easy to play, responsive and entertaining game instead of having fancy graphics.

4.1 Design Goals

- Ease of Use: Pong X is a game, so it should be easy to be used. Users should not have a difficulty in using the system. System should provide understandable and clean interfaces.

- Reliability: Our program should perform its intended functions and operations without experiencing system crashes and failures. Program should not crush with unexpected inputs.

- Modifiability: It should be easy to modify the existing functionalities and features of the system in order to develop and reusability when needed.

- Reusability: Program should be implemented considering using the same system again. It should be self documenting and easy to read in order to remain sustainability.

- Adaptability: Java is one of the few languages that provides cross platform compatibility. This feature of Java allows our program to work in all platforms which JRE installed, therefore users will not care about the different operating systems.

- Responsivity: Since our software is a game, its vital that users actions must be responded as soon as possible in order to keep the game flow and provide a quality software. System should respond user's actions immediately.

Trade Offs:

- Ease of Use / Functionality: Users should be able to learn and use the system without having any problems. Usability is more important than the functionality. When functionality increases, complexity of the system is also increases which effects usability in a bad way.
- Responsivity / Visuality: Responsivity is one of the desing goals of the system. Using fancy and high quality models and animations in the software increase the response time. Instead of having fancy visual effects we decided to go with responsive system with sufficient visuality.

4.2. Subsystem Decomposition

In this section, we decomposed the system into different subsystems to clarify how the system is organized. These subsystems are relatively independent from each other to decrease coupling between subsystems. So, to make subsystems less dependent to each other, related parts are combined in one subsystem. The decisions we made during subsystem decomposition will affect other important characteristics of the software we will create. Performance of the software, extendibility and modifiability are closely related the quality of decomposition. Therefore, when we decomposed our system, we tried to achieve low coupling between subsystems and high cohesion within subsystems.

Figure-1 shows three main subsystems of our software. Each of them handles one concern. All classes within a subsystem focus on same concern, so high cohesion principle is succeeded. There are User Interface Subsystem, Game Management Subsystem and Game Entities Subsystem. These subsystems focus separate functions. While User Interface Subsystem handles visual part of the software, Game Management Subsystem handles controlling part of the game and Game Entities Subsystem handles modeling part. Because each of subsystem is almost independent from each other, low coupling principle is succeeded.

When we look inside the subsystems from Figure-2, it can be understood that all classes inside a subsystem are closely related to each other and there is a strong dependency between them. For instance the classes inside Game Management have similar names and similar tasks. They are strongly associated to perform control tasks by sending requests and responds to each other. On the other, hand, subsystems only have direct dependency to the subsystem below it. While User Interface depends on Game Management, Game Management depends on Game Entities.

During the decomposition of the system we tried to increase cohesion and decrease coupling. With the proper decomposition, we could easily assign subsystems to different group members because there is loose coupling between subsystems. Also, loose coupling and high cohesion will provide flexibility and modifiability to the software system.

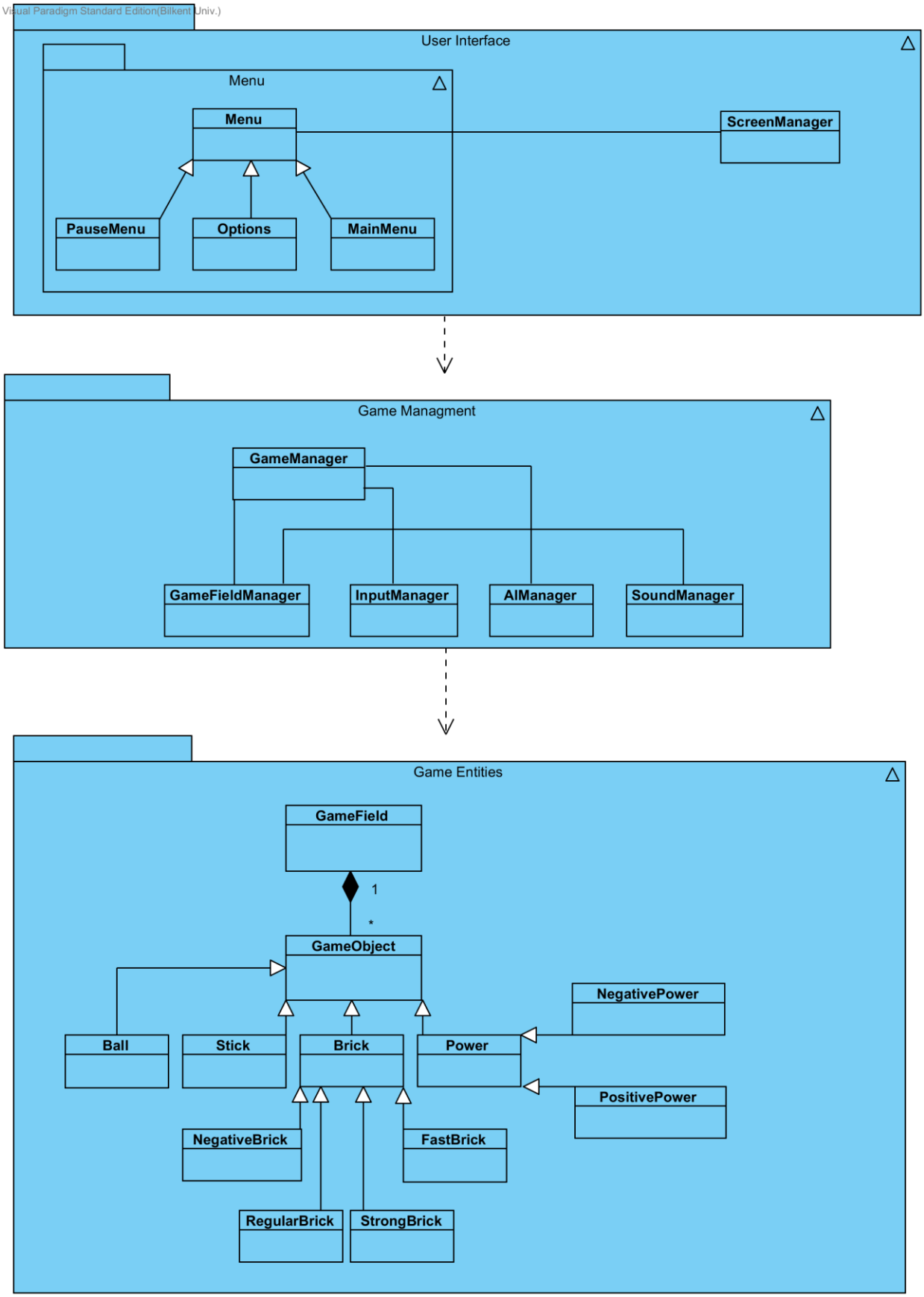


Figure – 20 Basic Subsystem Decomposition

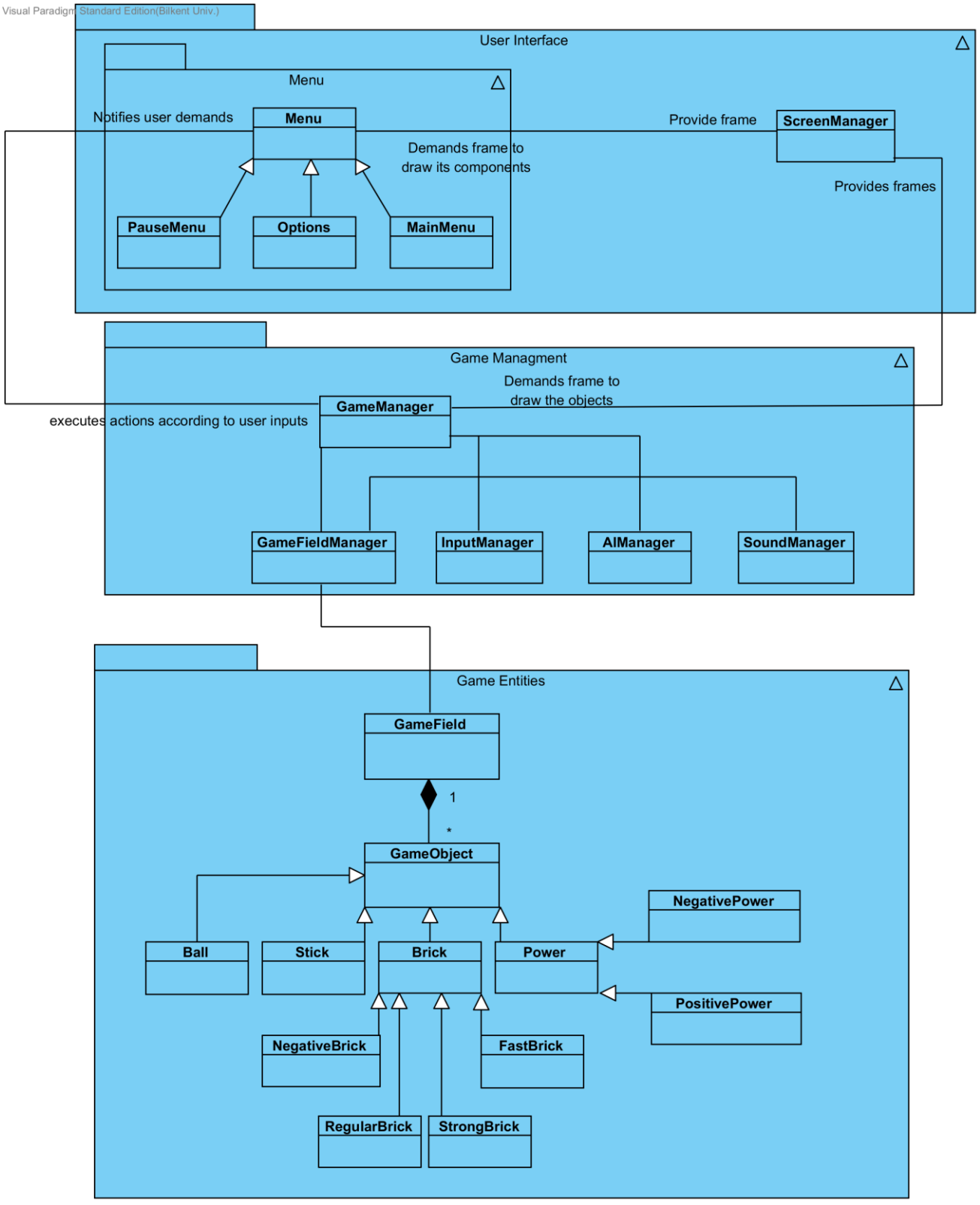


Figure-21 Detailed Subsytem Decomposition

4.3 Architectural Patterns

The architecture we design has three components our layer decomposition is closed that means a layer only access a layer below it. These components are Interface, Game Manager, Game Objects layers. Top component is our Interface. This component basically the layer that users interact. Interface component is not used by any other layers. After that we have Game Manager this part has the logic of our game. Game Objects layer has the all elements of our game. This architectural design is an implementation of the model-view-controller architecture. In the MVC system all layers are separated and have different specialties. Model is the main structure of game all objects of the game field and this correspond to Game Objects layer. View is basically what users see and interact and in our system this correspond to the Interface layer. Game Object and Interface couldn't interact without a controller. So our Game Manager is controller between them. MVC architecture is a very good system for our game we can change the view, controller or model without touching other layers. This helps easy debugging and makes changes easier. Also expansions of the game can be implemented much easier.

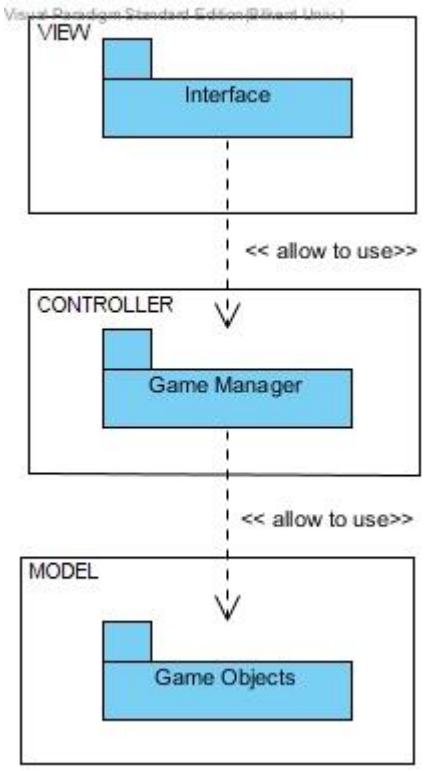


Figure-22 Model View Controller

4.4 Hardware / Software Mapping

The programming language we use to implement PongX will be java. Therefore, any computer that support java can run our game also java is platform independent we believe we can implement our code to other platforms easily. So basically if a computer has an OS and java compiler it will support our game. For hardware our game demand basic keyboard and mouse. PongX played via keyboard and for the menu and other interfaces we use mouse but keyboard could be another option. We don't need high requirements for our game and it is a rare feature in modern games.

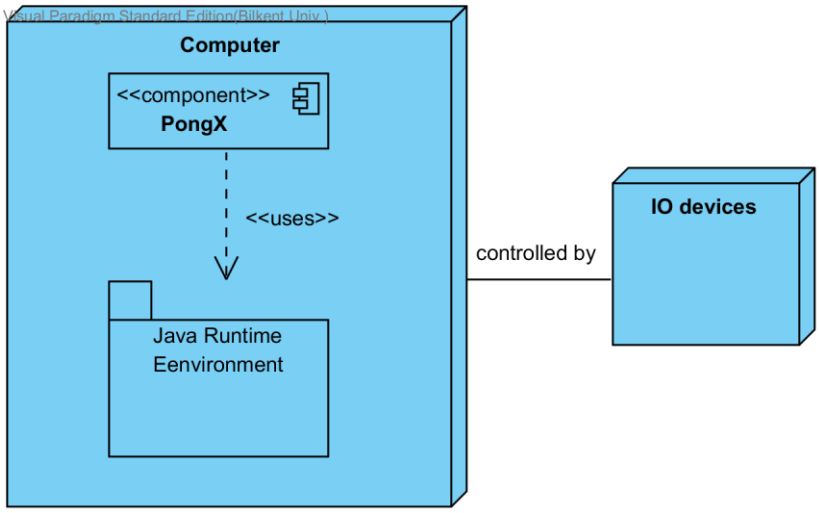


Figure-23 Deployment Diagram

4.5 ADDRESSING KEY CONCERNS

4.5.1 Persistent Data Management

Our game does not need to store persistent data for each game which could be single player or multiplayer. When a game is over, there is no need any data or information about the last game to save because all rounds and all parts of the game have no relation with the plays that are played previously.

Even though there will be no data for each game, it still requires some data to be used during plays such as different background images, map images and the game sound. However, the game does not really need a complicated database system and data management will be used in order to store and manipulate such images. The images of menu's, background and maps will be stored in Portable Network Graphic (PNG) format and the game sound will be stored in the MP3 format. All these data are thought to be stored in

hard disk drive. When any of these data are needed during the game, it just needs to be loaded from the disk.

4.5.2 Access Control and Security

In our game, users will not have any account; each user playing the game will have the same options and permission over the game. Hence user authentication system will not need to be provided. For the same reason, there will be no administrator to control different users.

All these reasons make the problems or questions about control and security of the game less important to worry about. Moreover, our game will not require any network connection as it is an offline game. This is also in the favor of security of the game because it will be less vulnerable against cyber-attacks.

Besides, persistent game data will be used in order to save files such as the game sound and map images. These data will not be protected as the users have access to these data. For example, they can personalize their background image by uploading an image from their computer.

4.5.3 Global Software Control

Event driven control would be the most proper option for software control of the Pong. Event driven control models are driven by externally generated events.The software control system waits until an event occurs. When it happens, the event will be send to the related object where a decision will be made about what to do. In the control system of Pong, it would be the most suitable solution to use Model-View-Controller as an architectural pattern. The system waits for an event to be occurred and in this case, this event will be

clicking on the screen. After that, the system updates itself by sending a notification to view part of the Model-View-Controller.

4.5.4 Boundary Conditions

Initialization

Pong X does not require any install, since it is implemented with Java game will initiated with a .jar file.

Termination

Pong X can be closed(terminated) by clicking “Exit” button at the right bottom of the main menu. If user wants to quit while playing the game, user must enter the pause menu then return to the main menu and exit. Lastly it is possible to terminate the game from the “X” button at the right top which Windows provides automatically.

Errors

Our aim is keeping errors and bugs as low as possible but if an error occurs during the system run ,system will try to keep on. For example if the background image could not loaded, instead of terminating itself; system will work with a blank(black) background.

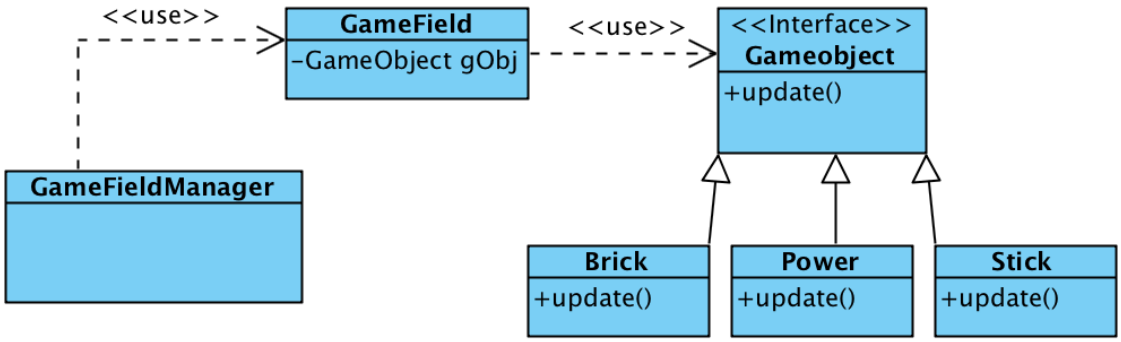
If there is an error with the game engine and it effects how the system works, system should terminate itself.

5. Object Design

5.1. Pattern Applications

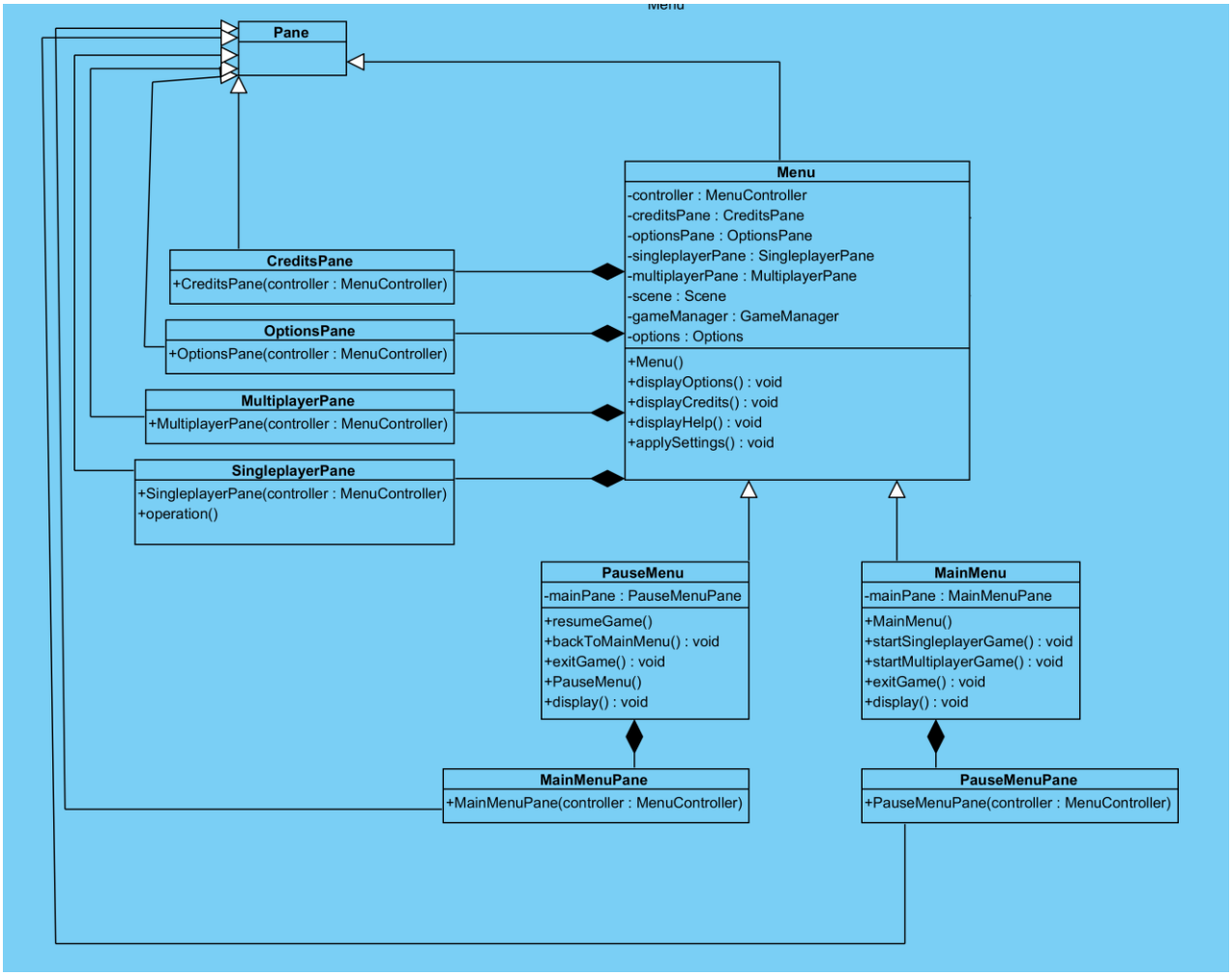
First design pattern that we use is singleton pattern. In a singleton pattern, a class has only one instance and this class is globally accessible. In our design, GameField and GameManager are implemented as singleton pattern. (Drawing are not included here because they already explained in class interfaces)

Other design pattern that we are using is bridge pattern. Bridge pattern is used to decouple the interface of a class from its implementation. In the “GameField” class, where the game happens, there will be some changes on the game objects, but objects will not be updated at the “GameField” class. “GameField” is going to use “GameObject” interface class then objects will be updated.



(Figure-24)

Our last pattern is composite pattern. We are using JavaFX so instead of JPanel we have Pane's. We have panes that contains pane and menu that contains menus, composite pattern is the most suitable design pattern of this situation.



(Figure-25)

5.2. Class Interfaces

5.2.1. Class Interfaces of User Interface

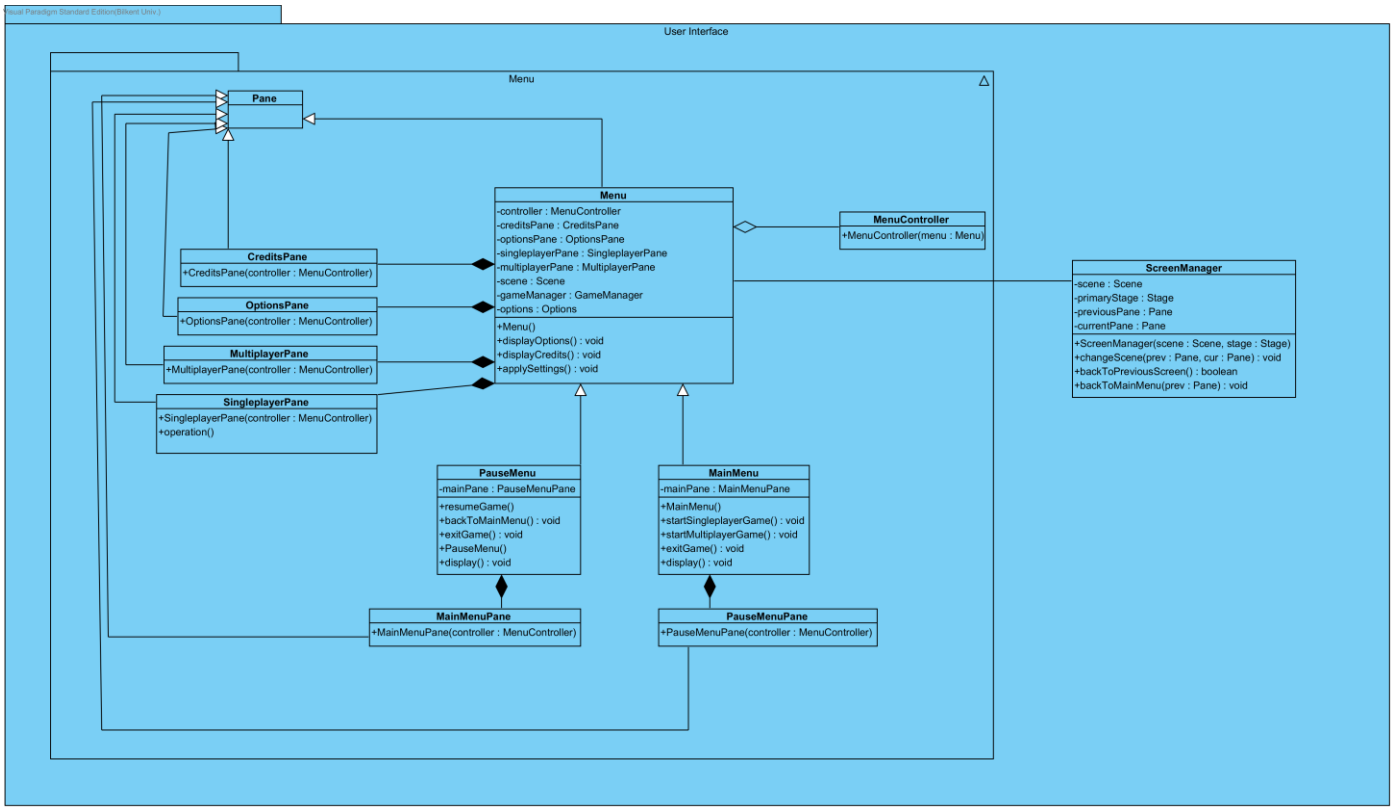
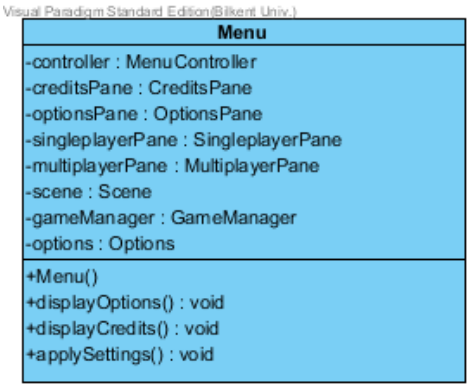


Figure-26: Game Management Subsystem Interface

At this part, classes which are about user interface will be explained in detail. Figure 1, shows general structure and relations.

Menu Class



(Figure-27)

Menu class extends pane. This is parent for other menu parts.

Attributes:

private Scene scene: This is like JFrame, but it is a part of javaFX. It is a container for all visual context including panes.

private MenuController controller: This controller handles user inputs including clicking button. It reaches buttons inside Menu pane, and directly adds action to them.

private CreditsPane creditsPane: This Pane type property of CreditsPane class is used in graphical user interface to show Credits screen when specific button is clicked. While its construction, specific visual components such as buttons, labels are added.

private OptionsPane optionsPane: This Pane type property of OptionsPane class is used in graphical user interface to show Options screen when specific button is clicked. While its construction, specific visual components such as buttons, labels are added. It is related to Options class which will be explained later.

private SingleplayerPane singleplayerPane: This Pane type property of SingleplayerPane class is used in graphical user interface to show SingleplayerPane screen when specific button is clicked. While its construction, specific visual components such as buttons, labels are added. Also, via this buttons in this pane, user could select difficulty of the AI, map backgrounds and specific brick set.

private MultiplayerPane multiplayerPane: This Pane type property of MultiplayerPane class is used in graphical user interface to show MultiplayerPane screen when specific button is clicked. While its construction, specific visual components such as buttons, labels are added. Also, via this buttons in this pane, user could select map backgrounds and specific brick set.

Private GameManager gameManager: This GameManager type property provides reference o Game Management subsystem. Also, changes made in singleplayer or multiplayer pane and options are kept in game manager. So, it could start other objects according to these selections. It will be created when user clicks play button inside multiplayer or singleplayer pane.

Private Options options: This Options type property keeps the options which can be changed via options pane by user. Options object has sound disable selection, background selection and input keys.

Constructors:

public Menu: Initializes *optionsPane, creditsPane, singlePlayerPane, multiplayerPane, gameManager, options and controller* properties.

Methods:

public void displaySingleplayerPane(): This method uses changeScene() method of screenManager and adds *singlePlayerPane* to scene by replacing the current pane on scene.

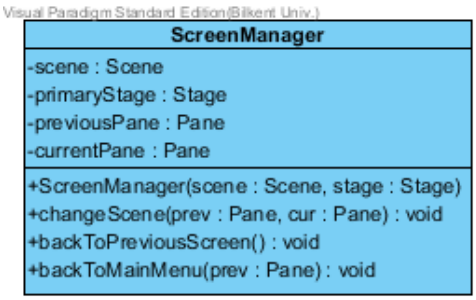
public void displayMultiplayerPane(): This method uses changeScene() method of ScreenManager and adds *multiPlayerPane* to scene by replacing the current pane on scene.

public void displayCredits(): This method uses changeScene() method of ScreenManager and adds *creditsPane* to scene by replacing the current pane on scene.

public void displayOptions(): This method uses changeScene() method of ScreenManager and adds *optionsPane* to scene by replacing the current pane on scene.

public void applySettings(): This method applies the current options by setting values of *options* object which are isSoundEnabled*, paddleKind, fieldTexture* and also input keys.

ScreenManager class



(Figure-28)

This class handles all pane changes. It has reference to both stage and scene. So, with using default methods of these components it changes pane. These references are set with using proper setter methods.

Attributes:

Private static Scene scene: It is used to load proper pane to the scene using this property.

Private static Stage primaryStage: Stage could be thought like windows. It is a container for scenes.

Private static Pane previousPane: It holds reference to previous pane because there are back buttons on most of the panes. So, to go previous pane this property will be used.

private static Pane currentPane: It holds reference of current Pane.

Constructor:

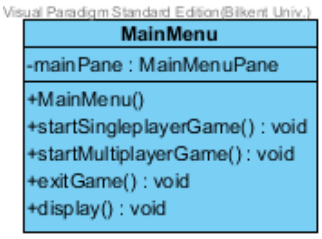
public ScreenManager(): Because all of its methods are static, this constructor is not needed.

Methods:

Public static void changeScene(prev : Pane, cur : Pane): Panes are changed via this method. It changes pane and refresh the scene.

public static void backToPreviousScreen(): Generally, this method is called when back button is pressed. Reload previous pane to the scene.

MainMenu class



(Figure-29)

Attributes:

private MainMenuMainPanel mainPane: This MainMenuMainPane type attribute of MainMenu class is used in graphical user interface to show Main Menu on screen.

Constructor:

public MainMenu(): It initializes MainMenu object.

Methods:

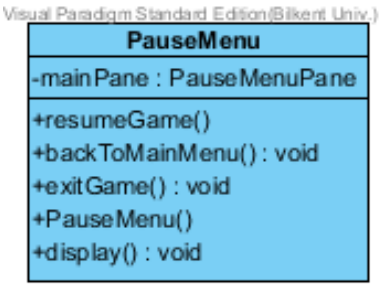
public void display(): This method uses changeScene() method of ScreenManager and adds *mainPane* to scene by replacing the current panel on scene.

public void startSingleplayerGame(): By the reference of *gameManager* attribute of MainMenu class (*gameManager* attribute is inherited from Menu class) this method invokes Game Management subsystem to control gameplay routine. Before this method, game manager is ready for single player game.

public void startMultiplayerGame(): By the reference of *gameManager* attribute of MainMenu class (*gameManager* attribute is inherited from Menu class) this method invokes Game Management subsystem to control gameplay routine. Before this method, game manager is ready for multi player game.

public void exitGame(): This method ends the application.

PauseMenu class



(Figure-30)

Attributes:

private PauseMenuPane mainPane: This PauseMenuPane type attribute of PauseMenu class is used in graphical user interface to show Pause Menu on screen.

Constructor:

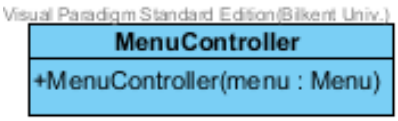
public PauseMenu(): It initializes instances of PauseMenu object.

Methods:

public void resumeGame(): This method removes PauseMenuPane from scene and continuous game loop.

public void backToMainMenu(): This method uses backToMainMenu() method of ScreenManager to stop game and load main menu pane to the scene.

MenuController



(Figure-31)

This controller adds actions to buttons and other components within menus. It does not implement any interface because in javaFX there is no need for it. It simple reaches components via getters and add actioner to them.

Every menu panes has a specific class which extends this. However, these classes are hided for the sake of simplicity. Their button actions are written inside of constructor of these classes.

Constructor:

public MenuController(Menu menu)**:** It initializes instances of MenuController object for a specific menu. Inside of constructor, it adds actions to buttons inside Pane of menu objects.

5.2.2. Class Interfaces of Game Management

In this part, controller classes of the game are used to make sure that the game runs properly and the communication between view and model classes are established well. There are 5 components in this part: 4 controller classes and 1 property class. Controller classes are GameManager, GameFieldManager, InputManager and SoundManager. The property class is Options class. The relationship between these classes are given in the Figure-2. All these classes will be explained in details in this subsystem.

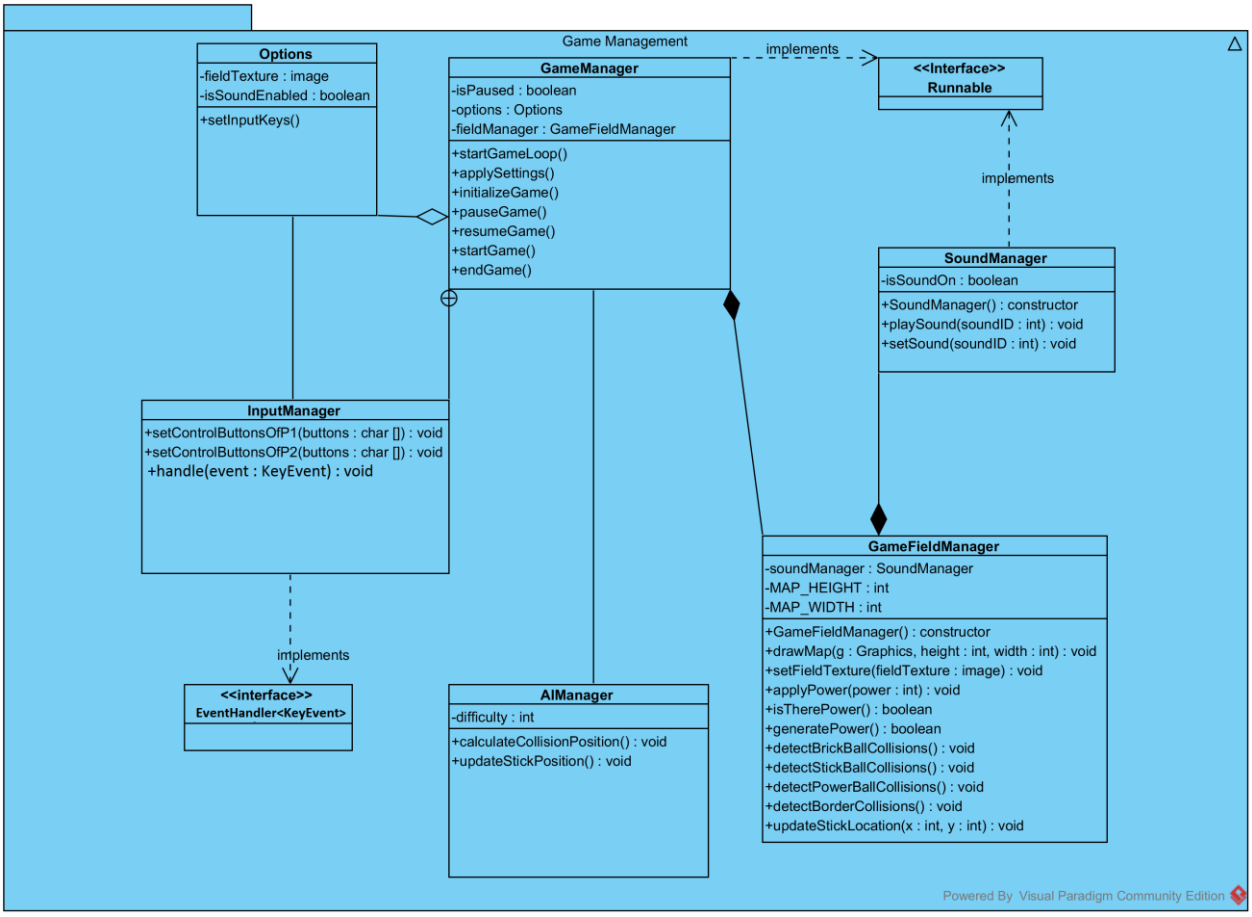
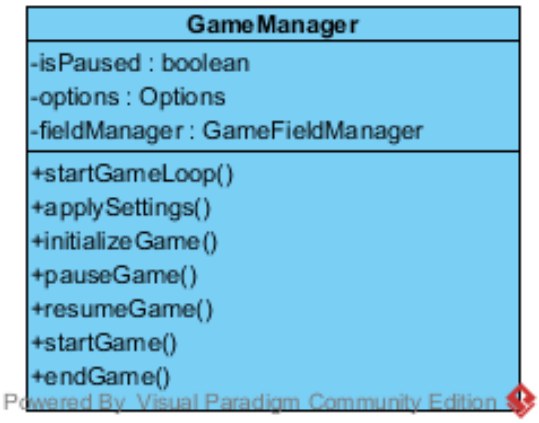


Figure-32: Game Management Subsystem Interface

GameManager Class



(Figure-33)

- GameManager Class is the Façade class of this subsystem. This class realizes the appropriate operations depending on the requests coming from User Interface subsystem. Moreover, this class runs the game in a loop. Note that it needs to implement runnable interface due to the fact that the game loop has to run as a thread.

Attributes:

private boolean isPaused: This attribute is used to understand whether the game is paused or not during the game loop

private Options options: This attribute holds the options of the game such as background image, sound and input keys.

private GameFieldManager fieldManager: This attribute is a GameFieldManager object, by which GameManager class associates with proper methods of GameFieldManager class.

Constructors:

public GameManager(): This constructor initializes the attributes of the GameManager for the first run of the system.

Methods:

public void startGameLoop(): This method runs a loop in which the system is updated continuously.

public void applySettings(): This method changes the option attribute of the class according to the default options that are determined by system or the options that are chosen by the user.

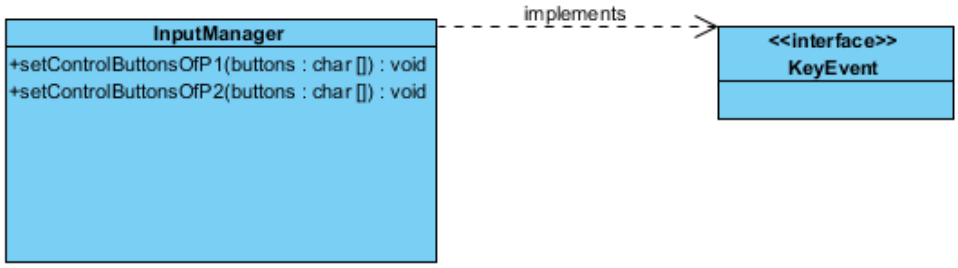
public void pauseGame(): When isPaused attribute is set to true, this method prevents the game loop from being iterated.

public void resumeGame(): When isPaused attribute is set to false, this method enables the game loop to iterate.

public void startGame(): This method starts a new game, by resetting game information stored on GameFieldManager class thanks to the fieldManager attribute.

public void endGame(): This method terminates the game when called and makes sure that all information of the game is deleted.

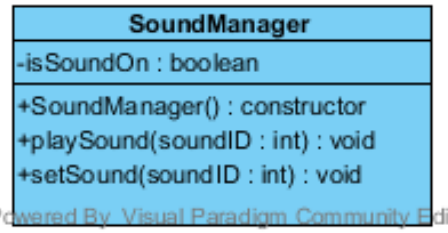
InputManager Class



(Figure-34)

- This class is designed to detect the user actions performed by keys on keyboard to move sticks on the screen. In this context, this class implements the necessary interface of JavaFX.

SoundManager Class



(Figure-35)

- This class enables the play sounds whenever it is referenced by GameFieldManager. However, this class has to implement runnable interface in order to be able to run in a different thread.

Attributes:

private boolean isSoundOn: This attribute is a boolean to determine if the sound is enabled or disabled in the system.

Constructors:

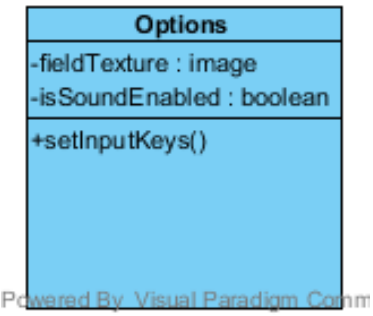
public SoundManager(): This constructor initializes the object of this class. Note that isSoundOn attribute is set false initially.

Methods:

public void playSound(int soundID) : This method is invoked by GameFieldManager class when it is needed. This method plays a sound sample according to given value.

public void setSound(int soundID) : This method is invoked by GameFieldManager class when it is needed. This method sets the ID of the next sound that is about to be played.

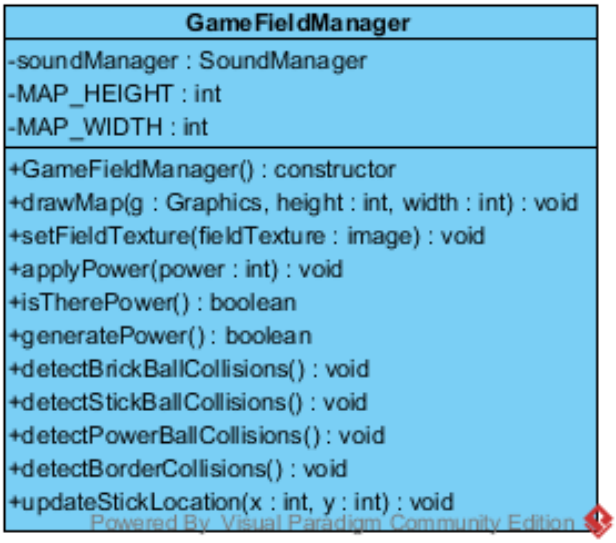
Options Class



(Figure-36)

- This class is a simple property class, which holds the options of the system such as background image. It is used by GameManager class.

GameFieldManagerClass



(Figure-37)

Attributes:

private SoundManager soundManager: This private attribute is the reference to SoundManager class to the play proper sounds during the game.

- MAP_HEIGHT and MAP_WIDTH are the static constant variables. They determines the size of the map.

Constructors:

public GameFieldManager(): It initializes a GameFieldManager object with default attribute values.

Methods:

public drawCurrentMap(Graphics g, int height, int width): This method draws the current map according to given attributes.

public void setFieldTexture(image fieldTexture): It sets the background image by loading the image that is given as parameter.

public void applyPower (int power): This method applies the powerUp to the game depending on its type.

public boolean isTherePower() : This method return true if there is any powerUp in the range of objects in the map.

public void generatePowerUp() : This method decides whether a new powerUp will be created or not by creating a random value and checking its value. If a new powerUp is about to be created, its position on the map is calculated properly.

public void detectBrickBallCollisions() : This method detects the collision between a brick and the ball when they collides. After that, the new state of the map is determined.

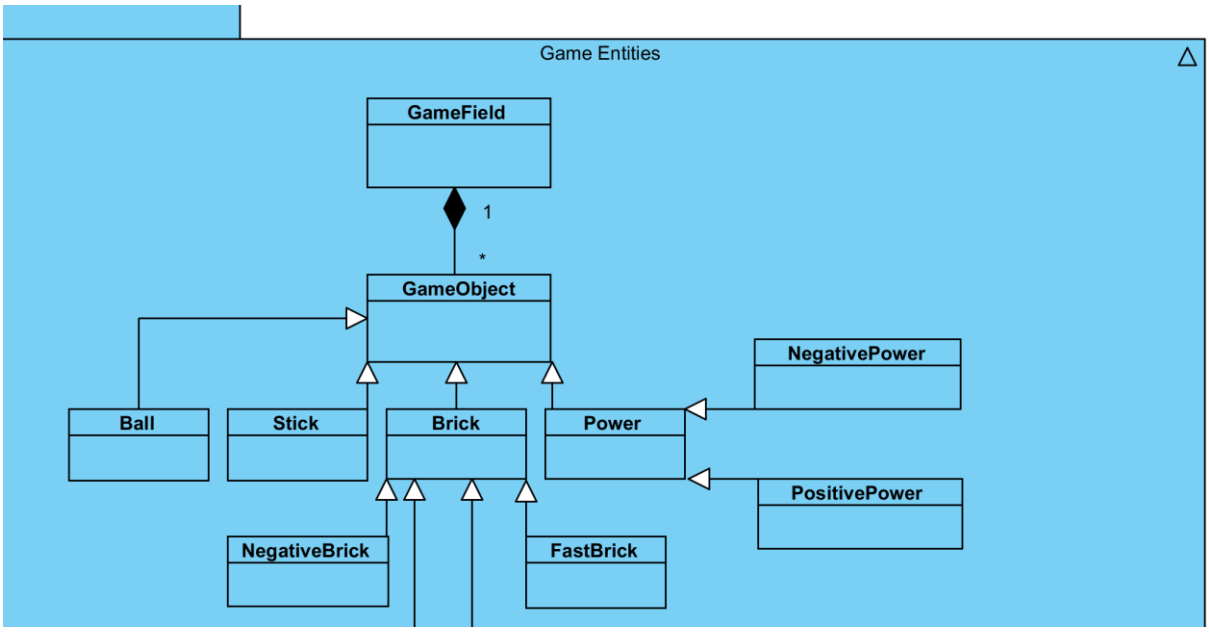
public void detectStickBallCollisions() : This method detects the collision between a stick and the ball when they collides. After that, the new state of the map is determined.

public void detectPowerBallCollisions() : This method detects the collision between a powerUp and the ball when they collides. After that, the new state of the map is determined.

public void detectBorderCollisions() : This method detects the collision between the wall and the ball when they collides. After that, the new state of the map is determined.

public void updateStickLocation(int x, int y): This method updates the position of the sticks depending on the key events on the keyboard.

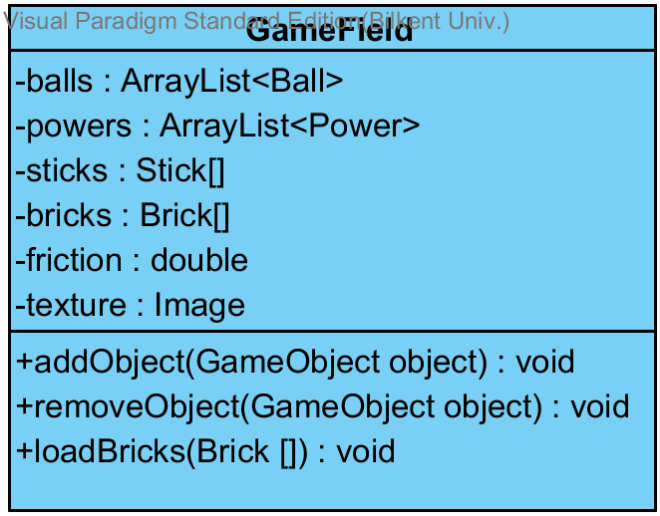
5.2.3. Class Interfaces of Game Entities



(Figure-38)

This figure only shows general relations of the game entities. They are explained in detail at the rest of this part.

GameField



(Figure-39)

This class include all the game objects and their positions on the map because of this it includes methods that create and delete objects on the map of our game.

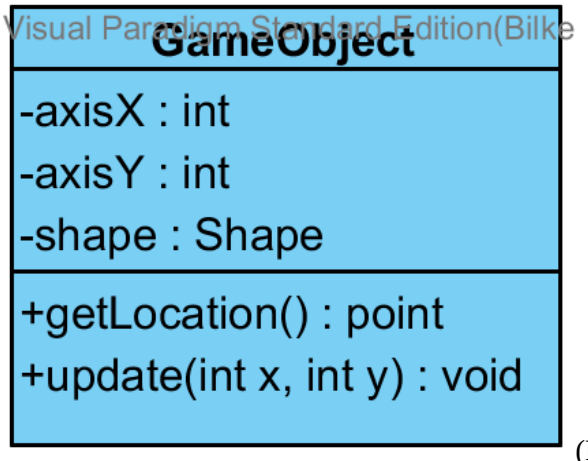
Attributes

- ***private ArrayList<Ball> balls:*** hold the reference to the ball objects on the map.
- ***private Arraylist<Power> powers:*** hold the reference to the power objects on the map.
- ***private Stick sticks []:*** hold the reference to sticks to the player sticks on the map.
- ***private double friction:*** amount of friction that surface has.
- ***private Brick bricks []:*** hold the reference to the brick objects on the map.
- ***private Image texture:*** background of our game field.

Methods

- ***addObject(GameObject object):*** adds a desired game object to the map.
- ***removeObject(Game Object object):*** remove the desired object from the map field.
- ***loadBricks(Bricks []):*** add set of bricks to map field.

GameObject



(Figure-40)

This class is the parent of all of our game object and it include location information and shape information for all of our game objects its main function is changing and getting the location information for all of our objects.

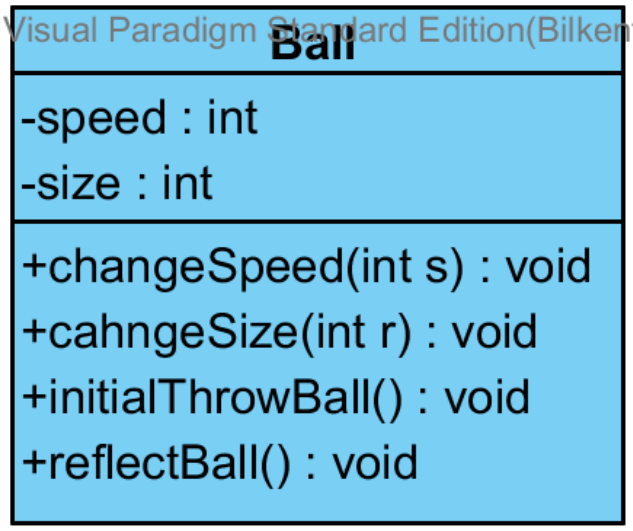
Attributes

- ***private int axisX:*** the location of the object on the x axis.
- ***private int axisY:*** the location of the object on the y axis.
- ***private Shape shape:*** determine the shape of the game object.

Methods

- ***getLocation():*** this method returns the location of the desired object.
- ***update(int x, int y):*** this method add a new location for the object.

Ball



(Figure-41)

One of the basic game objects. It has important methods about the directions of the ball like initial throw and reflect.

Attributes

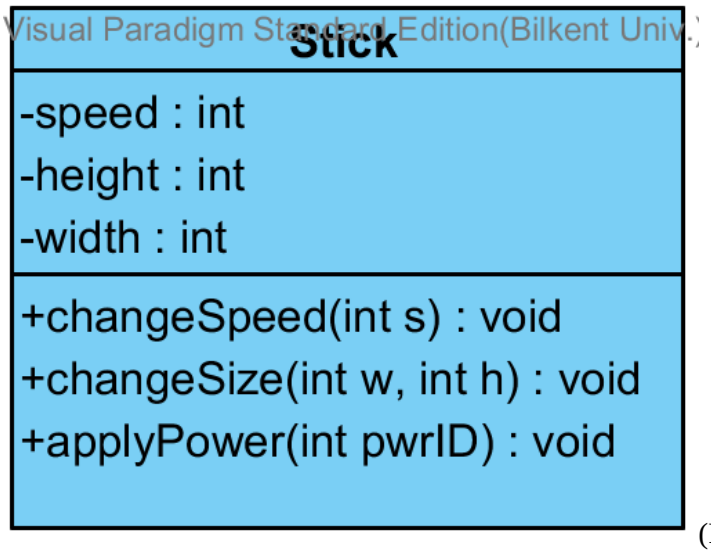
- ***private int speed:*** the value of the speed of the ball.
- ***private int size:*** the value of the radius of ball.

Methods

- ***changeSpeed(int s):*** change the speed for ball generally in order to collisions with fast brick.
- ***ChangeSize(int r):*** change the size of the ball generally in order to collisions with bricks.
- ***InitialThrowBall():*** when the game starts this method release the ball to a random direction.

- ***reflectBall():*** when it collide with an object this method change directions.

Stick



(Figure-42)

The paddles that players control, they have changeable size and speed and they can effect from the power ups.

Variables

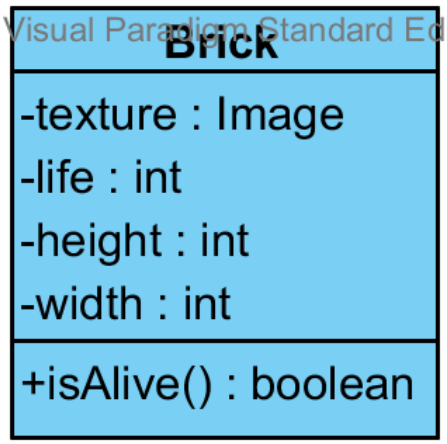
- ***private int speed:*** value of the speed of the stick.
- ***private int height:*** value of the height of the stick.
- ***private int width:*** value of the width of the stick.

Methods

- ***changeSpeed(int s) :*** change the speed for stick generally in order to collisions with powers.
- ***changeSize(int w, int h):*** change the size of the sticks generally in order to collisions with powers.

- ***applyPower(int pwrID):*** when the collider detect a collusion this method implement the changes.

Brick



(Figure-43)

Bricks are our obstacle on the map and the parent of all bricks and there are four kinds of them, bricks consist a life point and texture.

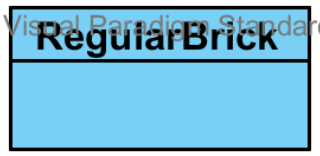
Variables

- ***private Image texture:*** background image of our bricks.
- ***private int life:*** the amount of hits a brick can take.
- ***private int height:*** value of the height of the brick.
- ***private int width:*** value of the width of the brick.

Methods

- ***isAlive():*** this method return a Boolean value after collusions if the life get 0 it became false and with this information game manager delete the brick.

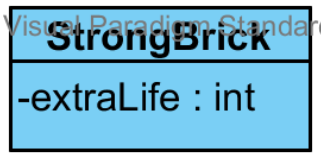
RegularBrick



(Figure-44)

Regular brick is the standard brick that has no extra ability.

StrongBrick



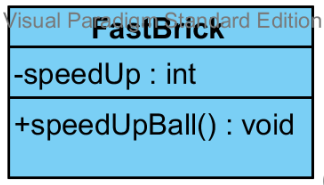
(Figure-45)

Strong brick is like regular brick but it has extra life points.

Attribute

- ***private int extraLife:*** amount of the extra life of the strong bricks.

FastBrick



(Figure-46)

Fast brick is a brick that makes balls faster as they hit them this is a collusion based brick we check that in the game manager, if a ball and a fast brick collides our ball gains velocity.

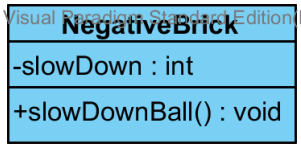
Attribute

- ***private int speedUp:*** the value of the speed gained by ball.

Methods

- ***speedUpBall():*** increase the speed of the ball.

RegularBrick



(Figure-47)

Negative brick is a brick that makes balls reflect differently as they hit them, this is a collusion based brick we check that in the game manager if a ball and a negative brick collide our ball behave a different reflection

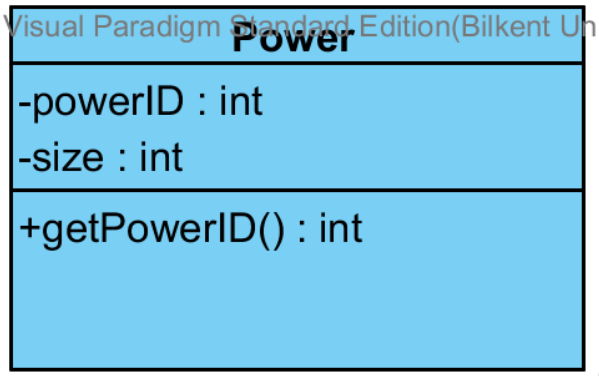
Attribute

- ***private int slowDown:*** the value of the speed loosed by ball.

Methods

- ***speedDownBall():*** decrease the speed of the ball.

Power



(Figure-48)

Powers are increase or decrease the ability of sticks, there are two kind of power one of positive the other negative. The power class is parent for both of them.

Attribute

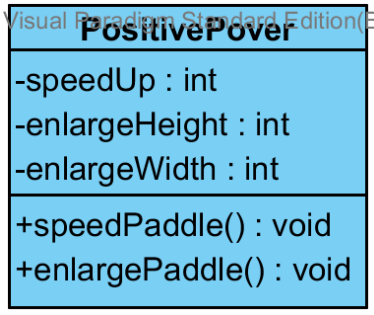
- ***private int size*** the value of the radius of powers.
- ***private int powerID:*** holds the id for the power.

Methods

- ***getPowerID():*** to learn what kind of power the ball encounter.

Type of powers

PositivePower



(Figure-49)

Positive power is enlarging and speed up the players' paddle.

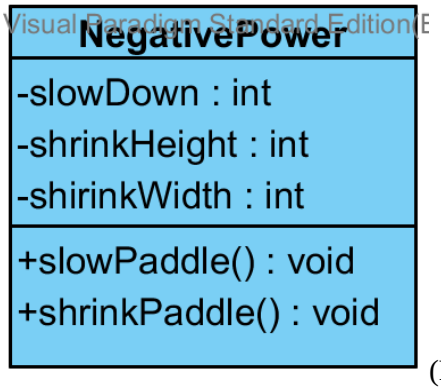
Attribute

- ***private int speedUp:*** the value of the speed gained by paddle.
- ***private int enlargeHeight:*** the value of the height gained by paddle.
- ***private int enlargeWidth:*** the value of the width gained by paddle.

Methods

- ***speedPaddel():*** increase the speed of the paddle.
- ***enlargePaddle():*** increase the size of the paddle.

NegativePower



(Figure-50)

Negative power is shrinking and speed down the players' paddle.

Attribute

- ***private int slowDown:*** the value of the speed loosed by paddle.
- ***private int shrinkHeight:*** the value of the height loosed by paddle.
- ***private int shrinkWidth:*** the value of the width loosed by paddle.

Methods

- ***slowPaddel():*** decrease the speed of the paddle.
- ***shrinkPaddle():*** decrease the size of the paddle.

5.3. Specifying Contracts

1. Context GameManager::isCollide(gameObject o1, gameObject o2): boolean

pre: gameField.updateGameField()

post: setScoreP1()

after the game field updated the game controls for any events and fin that one player has a score and increment their score.

2. Context GameManager::startGameLoop(): void

pre: initilizeGame()

post: ball.initialThrowBall()

after the game initialized game will starts and the ball release for game

3. Context GameManager::isCollide(gameObject o1, gameObject o2): boolean

pre: gameField.updateGameField()

post: fastBrick.speedUpBall()

when ball collide with a fast brick it will gain speed.

4.Context GameManager::isCollide(gameObject o1, gameObject o2): boolean

pre: brick.isAlive() == true

post: brick.setLife(brick.getLife – 1)

5.Context GameManager::pauseGame() : void

Pre: gameManager.isPaused() == false

Post: gameManager.isPaused() == true

Pause game method is only acceptable while game is running.

6. Context GameManager::isCollide(gameObject o1, gameObject o2): boolean

pre: gameField.updateGameField()

post: negativeBrick.slowDownBall()

when ball collide with a negative brick it will lose speed.

7. Context GameManager::startGameLoop(): void

pre: initilizeGame()

post: createPower()

after the game initialized game will starts and the powers are randomly created.

8. Context GameManager::isCollide(gameObject o1, gameObject o2): Boolean

pre: gameField.updateGameField()

post: gameField.removeObject(o2)

after the ball hit a power, the power will be removed from the map.

9. Context GameFieldManager::setGameFieldTexture(image): void

post: gameField.texture = image

player can define the background of their gameplay area.

10. Context GameFieldManager::updateGameField(): void

pre: brick.isAlive() = false

post: gameField.removeObject(brick)

after the ball consume all lives of a brick, the brick will be removed from map.

11.Context GameFieldManager::applyPower(power : int)
pre: isTherePower() : boolean
Before adding the power, it should check whether there is a power or not.

12.Context GameFieldManager::generatePowerUp() : void
post: isTherePower() : boolean
After generating the power, system should check if it is still remaining or not.

13. Context GameManager::isCollide(gameObject o1, gameObject o2): Boolean

pre: gameField.updateGameField()

post: ball.reflectball()

after the ball hit a regular brick it change direction.

14. Context GameManager::applySetting

post: gameField.loadBricks(Brick[])

before the game the player choose a set of pre organize brikcs.

15. Context PositivePower::speedPaddle() : void
pre: speedUp != 0
post: stick.changeSpeed(stick.speed+speedUp) & speedUp = 0
If a speedUp power gained and sticks speed is not increased yet, increase the stick speed and set speedUp to the 0

16. Context PositivePower::enlargePaddle() : void
pre: enlargeHeight != 0
post stick.changeSize (stick.height + enlargeHeight, stick.width) & enlargeHeight = 0;
After changing the height of the paddle, make enlargeHeight 0.

17. Context PositivePower::enlargePaddle() : void
pre: enlargeWidth != 0
post stick.changeSize (stick.height, stick.width+ enlargeWidth) & enlargeWidth = 0;
After changing the height of the paddle, make enlargeHeight 0.

18. Context NegativePower::slowPaddle() : void
pre: slowDown != 0
post: stick.changeSpeed(stick.speed-slowDown) & slowDown = 0
If a slowDown power gained and sticks speed is not decreased yet,set the new speed and set slowDown to 0

19. Context NegativePower::shrinkPaddle() : void
pre: shrinkHeight != 0
post stick.changeSize (stick.height - shrinkHeight, stick.width) & shrinkHeight = 0;
If shrinkHeight is not zero, change the height of the paddle, make shrinkHeight 0.

20. Context NegativePower::shrinkPaddle() : void
pre: shrinkWidth != 0
post stick.changeSize (stick.height, stick.width - shrinkWidth) & shrinkWidth = 0;
If shrinkHeight is not zero, change the width of the paddle, make shrinkWidth 0.

21. Context FastBrick::speedUpBall() : void
pre: speedUp != 0
post: ball.changeSpeed(ball.speed + speedUp) & speedUp = 0;
After ball hit the fastbrick speedUp should change, then speed of ball should change and lastly speedUp should be 0 again.

22. Context NegativeBrick::slowDownBall() : void
pre: slowDown != 0
post: ball.changeSpeed(ball.speed - slowDown) & slowDown = 0;
After ball hit the negative brick slowDown should change, then speed of ball should change and lastly slowDown should be 0 again.

23. Context ScreenManager::backToPreviousMenu(): boolean

Pre: previousPane != null

Post: currentPane == previousPane

When back button is pressed in a menu, previous menu should not be null. After that previousPane should become currentPane which indicates that it is succesfully loaded.

24. Context GameObject::update(int x, int y): void

Pre: x < GAME_WIDTH && x >= 0 && y < GAME_HEIGHT && y > 0

Post: axisX == x && axisY==Y

25.Context GameManager::resumeGame(): void

Pre: isPaused == true

Post: isPaused == false

Running game cannot be resumed. To be resumed pause menu should be shown and game should be paused.

26. Context Brick::isAlive() : boolean
Pre:GameFieldManager.detectBrickBallCollisions()
Post: update(x: int, y: int)
If there is a collision with a brick, check if its still alive or not and
update

6. Conclusion and Lessons Learned

In our project report there are 3 main parts: Project Analysis section, Project Design section and Object Design section.

The Analysis process was long and challenging because he had to define our project purpose and set the limits for the project. We stated our project overview and requirements. Having these thing before starting the implementation should help us at the implementation part but it was very hard to decide on requirements without having any process on project code. We tried to stick with the user interface style that we mentioned at the report. Use cases and scenarios were the most useful parts of this section because they are explanation of what process is going to be like. With the help of the scenarios we drew the sequence diagrams. Sequence diagrams were hard to draw since we had no experience on them and we had to draw them without having any code. Class diagrams were challenging too, not

because of drawing or deciding interactions between classes, because of we didn't know exactly which classes we needed. We started with the classes that we think we needed, then removed some of them, added new ones.

In the second part, Project Design, we started with design goals, design goals helped us to focus on things that we had determined as important. After determining design goals, we decomposed system into subsystems. At the decomposition we care about high cohesion. After subsystem decomposition architectural patterns are applied and hierarchy and relation between subsystems are denoted. At last key concerns are addressed as persistent data management, access control and security, global software control and boundary conditions. In the boundary conditions we specified how to start the program, how to determine it and what to do if an error happens.

At the last part, Object Design, we determined our design patterns. We tried to choose suitable patterns for our program. After that we explained the classes in our design and their interfaces. Finally, we specified contracts using OCL.

In the project, we practiced what we have been thought in the lectures. We gained experience about drawing UML diagrams and producing code from UML diagrams. Most importantly we had experience about working as a group, things that we experienced are not given in the classes. If we look from that perspective, this project is very valuable and important for us.