Bilkent University

Department of Computer Engineering

CS319 - OBJECT ORIENTED SOFTWARE ENGINEERING SPRING 2015 – 2016 FALL

Break Bricks

Final Report

Group 14 | Section 01
Berire Gündüz
Kemal Büyükkaya
Kaan Akıncı
Barış Çelik

07.12.2015

1. Introduction

Break Bricks is actually very well-known arcade game that has a many different versions. In our project we will develop our own version of this game which enriched by many power-ups, penalties and different kinds of bricks to break. For the implementation of the game Java, which is an object-oriented language, will be used.

The purpose of this game, like many other versions of it, will be to break as many bricks as possible in restrictive conditions. In our version these restrictive conditions are limited time and limited amount of lives that a player can have through a game. The game will be a desktop application and will be controlled by left and right buttons on the keyboard.

We aim to design a consistent game that abides by OOP fundamentals which properly models the real-life objects, interactions and events.

In this report two main categories about project analysis are included, requirement analysis and system analysis.

2. Requirement Analysis

2.1. Overview

Break Bricks is a ball-and-paddle arcade game. Like its many versions, it is quite easy to play and fun at the same time. Basically there 30 different kinds of bricks and the player tries to destroy each brick by sending a ball through them by the help of the paddle at the bottom. Every hit from the ball either destroys a brick or just crack it depending on the type of the brick or/and the ball. To complete the game the player should destroy all the bricks, in limited time and without running out of lives that given at the start of the game. At the end of each game the point that player had won will be added to the scores list so that the list can be used to detect the highest score ever received.

2.1.1. Gameplay

Gameplay is aimed to be easy as possible because of that a player can simply control the paddle on the play screen by just two buttons, left button and right button, on the keyboard. The aim of the player should be steer the paddle, to make paddle go left player have to use left button and to make it go right player have to use right button, so that the ball bounces off from the paddle towards a selected brick. The ball can bounce off from the walls as well however player should be careful about not to dropping the ball below the paddle. In case of missing the ball the player will be losing a life.

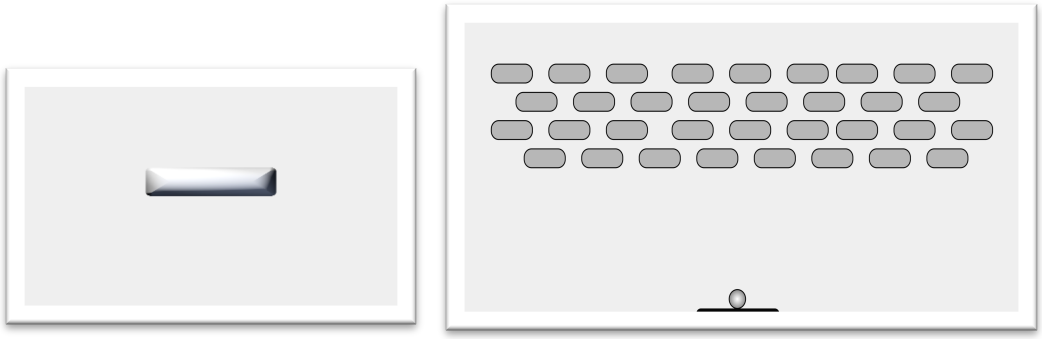
2.1.2 Challenge of the Game

Although we didn't design different levels for this game there is an indirect challenge to make a player keep wanting to play. That is the list of scores that gained through other rounds of the game. By showing the high score in the high scores menu which can be accessed by clicking the "High Scores" button at main menu, we aimed to create feeling of competition for players. As the score that gained through a game depends on power-ups that gained by luck, as the power-ups randomly distributed among bricks, and the amount of time plus life that is left after game is over, in each game player has a chance to have better score than ever.

2.1.3. Ball & Paddle

The paddle is the only tool that the player can control; so the size and usage of it are important to take control over the game. While a short kind of paddle would lose the ball to the ground, a long kind of paddle might have to catch penalties. Also the player might catch some power-ups that effect the paddle to destroy the bricks by its own. The size and the usage of the ball is also important. While a small kind of ball would make you cannot see the ball, a ball with increased strength might destroy more than one brick at the same time.

2.1.4. Brick Types



The arena basically consists of 30 bricks that consist of 3 kinds of bricks, like glass brick that can be broken by 1 hit, normal brick that can be broken by two hit and steel brick that can be broken by 3 hit. In addition to the breakability of bricks there will be other properties that will be randomly distributed among bricks which are penalties and power-ups.

Glass brick: Takes one hit to break it.

Normal brick: Takes two hits to break it.

Steel brick: Takes three hits to break it.

2.1.5. Power-ups and Penalties

The power-ups and penalties are the essential part of the game to make it so fun. The player will never know if there is a coming power-up or a penalty when the ball hits a brick. Also the player cannot know exactly what type of power-up or penalty coming. Some of them would be rare and not be coming as much as the other ones and that makes this kind of power-ups interesting. The power-ups and the penalties are:

Faster paddle: Makes paddle move faster.

Slower paddle: Makes paddle move slower.

Extra life: It gives one extra life to the player.

Super ball: Increases the power of the ball so that it can break bricks easily.

Weak ball: Decreases the power of the ball so that it takes more hit then usual to break a brick.

PointsX2: One that multiplies the score with 2 per brick for a limited time

Extra Time: Gives additional time to player.

Time Penalty: Shortens the time limit.

Instant death: It takes one life from the player.

Random power-up: It gives a random power-up/down.

Rebel paddle: Paddle goes to opposite direction, when you move it to right or left.

2.1.6. Settings

The settings of the game can be changed in the Options Menu which can be accessed by clicking on “Options” button at the main menu. The User can change the volume, the bar and the ball’s color and texture from one of the pre-determined one’s. Any changes that are saved by the user will be stored until user change the options again.

2.1.7. Victory Conditions

In order to win the game, the user has to break all the bricks in a limited time without running out of lives. To get the highest score in the game, a player should provide victory conditions in shorter amount of time than other trials. In addition to time element if a player was able to gain more power-ups than other trials, which depends on how lucky the player is, then again the player can have the highest score among other scores.

2.2. Functional Requirements

Break Bricks is a Java client-based, arcade game with one player. Break bricks is a very well-known game that player uses a paddle to break bricks and gains points depending on the total number of the bricks that have been broken.

Users can launch the game using the desktop shortcut to the executable. After launching, users can choose to go to options menu, high-scores screen, help screen, credits screen and play screen.

If the user clicks on the “Options” button, the options menu will be opened and user will be able to control sound effects by on and off choices, customize the paddle or ball colors depending to their own wish. Then user will be able to go back to the main menu by clicking “Back” button.

If the user clicks on the “High Scores” button, the high scores menu that displays the list of high scores and the highest score ever received, will be opened. Again player will be able to go back to main menu by “Back” button.

As by clicking “Help” button a screen which gives detailed information about gameplay will be displayed by clicking “Credits” button a screen that displays information about our team will be shown. In both case user will be able to go back to main menu by clicking “Back” button.

After user starts the game by pressing “Play” button, system prepares the arena for the user. The arena basically consists of 30 bricks that consist of 3 kinds of bricks, like glass brick that can be broken by 1 hit, normal brick that can be broken by two hit and steel brick that can be broken by 3 hit. In addition to the breakability of bricks there will be other properties that will be randomly distributed among bricks such as power ups (one that multiplies the score with 2 per brick for a limited time, one that speeds up the paddle and the one that increases the strength of the ball etc.) and penalties (one that slows down the paddle etc.).

Users will have a total Score point, which is a sum of all the points that user wins through the game. This total score will be shown on the top right of the screen until the game is over. In our version of the game player starts with 3 lives and there is also a time restriction (five minutes). The amount of lives and time left will be shown in play screen too. By pressing ESC user will be able to pause the game and from the pause screen user can choose whether to continue to game, to go back to main menu or to restart the game.

During the game play there is three conditions to game to end. One is when the player press the ESC and exits from the game, other is when player is run out of time and the last one is

when the player run out of lives. When player cannot hit the ball with paddle and causes the ball to fall under the paddle, he/she will lost a life. So the user's goal is to block the ball and give it the appropriate direction to break some bricks. While playing the game, the user will use the arrow keys to move their paddle right and left. User should not be able to move their paddle up and down for the sake of simplicity. The user can see their in-game score on the screen. Total score will be updated after every time when a brick is broken.
When the game is over user will be shown a new screen that displays Total Score and rank among the other high scores. This screen will also enable user to go back to main menu or to restart the game by appropriate buttons.

2.3. Nonfunctional requirements

2.3.1. Usability

• As break brick game has a lots of different versions user will be easily accustomed to our version of the game.
• Any user should be able to start a new game with less than 3 mouse clicks.
• Gameplay is sufficiently straightforward and very simple so that users can get used to it quickly.

2.3.2. Reliability

• If system crashes in the middle of a task, which means during a game, no prior progress is lost due to this inconvenience.
• When a game is completed without any failure, system saves the score to the list of high scores and sorts the list in decreasing order so that the score at the top of the list will be shown as the highest score.

2.3.3. Performance

• System should respond to user's command no longer than 50 milliseconds.

2.3.4. Supportability

• The system will be able to be developed, such new attributes and functionalities can be added without changing the core structure of the system. Changes imposed by new technologies and bug fixes will be easy to implement.

2.4. Constraints

2.4.1. Implementation

• The system must be implemented in Java because it is a widely used programming language that is perfectly suitable for object-oriented software implementation which is needed for this game which includes a lot of objects and interactions among them.

• Some of the graphic objects will be designed using Adobe® Photoshop CS4.

• Any system that has Java libraries installed in itself will be able to run the game executable.

• The system must be implemented in IntelliJ IDEA since it has functionalities that ease the coding process. In addition to that for the source code management GitHub will be used.

2.4.2. Interface

• Game graphics will be designed in order to be appealing.

• Graphical computations should be executed with an onboard GPU.

• System should run properly in any display supporting a 1920x1080 resolution.

• User will be able to play game easily on the screen. The components of the game such as paddle, ball and different kinds of bricks etc. will be easy to distinguish and well designed. In addition to that, because the game will be played on the screen by using keyboard, the graphical user interface components such as buttons will be easily clickable and recognizable.

2.4.3. Legal

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

2.5. Scenarios

- **Scenario Name:** PlayerHitsBall

Participating Actors: Kemal: Player

Flow of Events:

- Kemal wants to save the ball in order to not to lose.
- Kemal presses the movement keys to move the paddle to the direction of the ball.
- Program responds and moves the paddle as long as he presses.
- The ball and paddle collides.
- When they collide program detects this collision and changes the direction of the ball by 90 ^{o}.
- Ball moves away with the calculation of the collision and program shows the result of this collision.

- **Scenario Name:** BreakBrick

Participating Actors: Kemal: Player

Flow of Events:

- Kemal hits the ball with his paddle.
- After Kemal hits the ball with paddle, with certain amount of time passed, ball collides with a brick object.
- Program catches the collision and makes the ball bounce off, similar to paddle-ball collision.
- When the calculations are done the brick disappears and number of bricks is decreased.
- Ball moves to its new direction and his score is updated.

- **Scenario Name:** DropPowerUp

Participating Actors: Kemal: Player

Flow of Events:

- When the ball that Kemal hits breaks a brick, program rolls a number and if the number is a hit, then the destroyed brick creates a powerup object.
- This object moves directly down to the bottom threshold line.
- Kemal catches it before the powerup touches the threshold line, Kemal's paddle will possess that random powerup.
-

- **Scenario Name:** LoseLife

Participating Actors: Kemal: Player

Flow of Events:

- Ball comes at Kemal's paddle.
- Kemal can't time the movement and his paddle misses the ball.
- Ball moves past the paddle line.
- Program catches the collision with ball and threshold line.

- Program deletes the ball and decrements the lives and score he has.

- **Scenario Name:** NoLivesLeft

Participating Actors: Kemal: Player

Flow of Events:

- Kemal misses the ball and lost a life.
- Program detects that Kemal has no lives left.
- Program stops the game and presents the game over message.
- Program asks for a high score name.
- Kemal sees his high score in the list, and then clicks OK.

- **Scenario Name:** TimeUp

Participating Actors: Kemal: Player

Flow of Events:

- Kemal tries to break all the bricks.
- But the timer goes to 00:00 and he can't finish the level in the given time.
- Program stops the game and defines Kemal as loser by default.
- His score is recorded and game over message is presented.

- **Scenario Name:** PauseGame

Participating Actors: Kemal: Player

Flow of Events:

- Kemal needs to do his project, so he needs to exit the game.
- He pushes the escape button and program detects this key.
- The program pauses the game loop until Kemal decides to click the resume game button or Exit to Menu button.
- Kemal presses Exit to Menu button and quits the game.

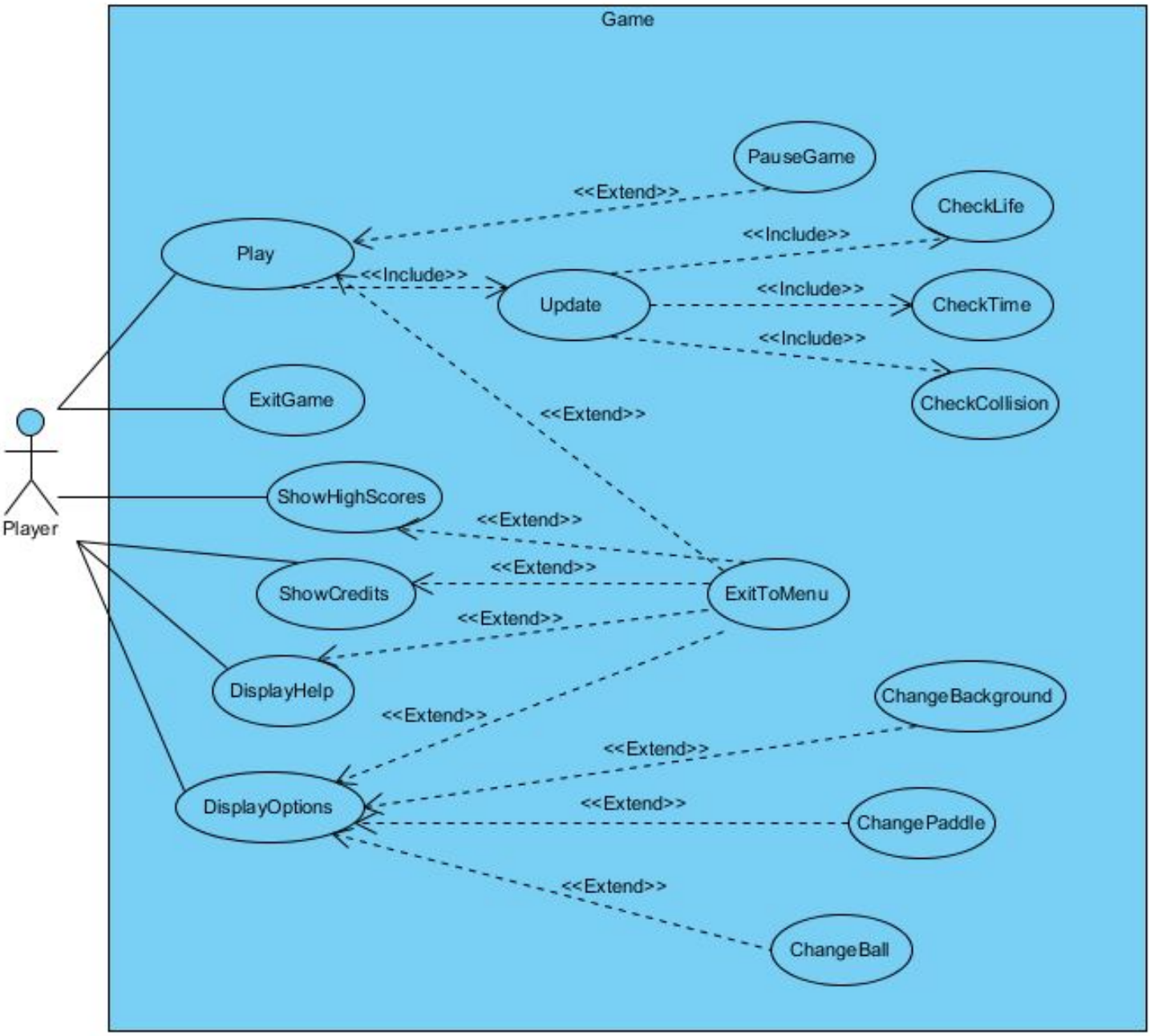
- **Scenario Name:** Customize

Participating Actors: Kemal: Player

Flow of Events:

- Kemal wants to change the appearances of the ball or paddle or background.
- He clicks the Options menu which opens the options and customization menu.
- In here he can turn the sound on or off and change the textures of the objects in game.
- After Kemal is done with customizations, he clicks the Back button which redirects him back to the previous menu.

2.6. Use Case Models



- **Scenario Name:** Play

Participating Actors: Initiated by Player

Flow of Events:

  - Program launches and the main menu appears.

- Player clicks Play.
  - Program opens the game at default settings. If the player changed the texture of the paddle or ball, they are set. Otherwise default textures are used. Brick objects are placed. Paddle and ball are placed. Score is set to 0 and Time is set for the according level.
  - Ball starts to move towards the player
- Player presses right or left arrow keys to navigate the paddle.
  - Program moves the player's paddle right or left and moves the ball. If ball collides with paddle or brick it bounces back. If a brick is broken program increases the score.
- Player breaks a brick with the ball and a powerup drops.
  - Program randomly generates a powerup object and starts moving it downwards. If the player's paddle collides with it, paddle will possess the attribute of the powerup. Else, program destroys that powerup.
- Player couldn't catch the ball, thus lost a life.
  - Program decreases the score of player and checks if player has any lives left. If player has lives left, ball's position is reset and paddle's position and powerups' are reset. Else program prompts game over message and asks for high score. Program stores the data in the txt file, sort it and open HighScoresMenu.
- Player destroys all the bricks.
  - Program checks if all the bricks are destroyed. After confirmation program prompts victory message and asks for high score. Program stores the data in the txt file, sort it and open HighScoresMenu.
- Player runs out of time.
  - Program detects that the time counter reached 00:00 and pauses game. Then prompts game over message and asks for high score. Program stores the data in the txt file, sort it and open HighScoresMenu.

Entry Condition: Player clicks Play.

Exit Condition: Player wins/loses/exits to menu.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** Update

Participating Actors: Initiated by Player

Flow of Events:

- Player clicks play and starts the game.
    - Program calculates ball's and paddle's next position, checks for collision and redraws the result in game loop
- Player ends the game.

Entry Condition: This use case is included in Play. It is initiated when the player clicks Play.

Exit Condition: Player pauses/loses/wins the game.

Quality Requirements: The program's response time should be 0.01 second.

- **Scenario Name:** CheckLife

Participating Actors: Initiated by Player

Flow of Events:

- Player is in game.
- Player misses the ball and loses a life
    - Program checks if the player has any lives left in game loop. If none left program prompts game over message and terminates.
- Player has no more lives left and loses the game.

Entry Condition: This use case is included in Update. It is initiated when the player clicks Play and Update is in use.

Exit Condition: Player pauses/loses/wins the game.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** CheckTime

Participating Actors: Initiated by Player

Flow of Events:

- Player is in game.
    - Program checks if the player has time left in game loop. If none left program prompts game over message and terminates.
- Player clicks ExitToMenu and returns to main menu.

Entry Condition: This use case is included in Update. It is initiated when the player clicks Play and Update is in use.
Exit Condition: Player pauses/loses/wins the game.
Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** CheckCollision

Participating Actors: Initiated by Player
Flow of Events:
- Player is in game.
    - Program checks if the ball hit the paddle or any bricks or bottom threshold line in game loop. If hit update the score and calculate next state of ball.
- Player exits the game.

Entry Condition: This use case is included in Update. It is initiated when the player clicks Play and Update is in use.
Exit Condition: Player pauses/loses/wins the game.
Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** PauseGame

Participating Actors: Initiated by Player
Flow of Events:
- Player is in game.
- Player presses the escape key.
    - Program gets out of game loop until player presses ResumeGame.
- Player presses ResumeGame.
    - Program gets back into the game loop and continues to update objects.

Entry Condition: This use case is extension of Play. It is initiated when the player clicks Play.

Exit Condition: Player clicks ResumeGame or ExitToMenu.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** ShowHighScores

Participating Actors: Initiated by Player

Flow of Events:

- Player finishes a game and enters high score.
- In main menu player clicks ShowHighScores.
    - Program opens up high scores and displays them to the player.
- Player clicks ExitToMenu.

Entry Condition: Player clicks ShowHighScores.

Exit Condition: Player clicks ExitToMenu.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** ShowCredits

Participating Actors: Initiated by Player

Flow of Events:

- Player launches the game.
    - Program greets the player with main menu.
- Player clicks on ShowCredits.
    - Program opens up the credits and displays the text.
- Player clicks ExitToMenu.

Entry Condition: Player clicks ShowCredits.

Exit Condition: Player clicks ExitToMenu.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** DisplayHelp

Participating Actors: Initiated by Player

Flow of Events:

- Player launches the game.
    - Program greets the player with main menu.
- Player clicks on DisplayHelp.
    - Program opens up the help window and displays the controls of the game.
- Player clicks ExitToMenu.

Entry Condition: Player clicks DisplayHelp.

Exit Condition: Player clicks ExitToMenu.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** DisplayOptions

Participating Actors: Initiated by Player

Flow of Events:

- Player launches the game.
    - Program greets the player with main menu.
- Player clicks on DisplayOptions.
    - Program opens up the options menu.
- Player changes some options and click ExitToMenu.

Entry Condition: Player clicks DisplayOptions.

Exit Condition: Player clicks ExitToMenu.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** ChangeBackground

Participating Actors: Initiated by Player

Flow of Events:

- Player is on the DisplayOptions menu.
- Player clicks on the background icon.
    - Program shows possible background images and gives player a choice to choose from these images.
- Player chooses one image and clicks on it.
    - Program sets the chosen image to be the new image of the background.

Entry Condition: Player clicks on the background icon on DisplayOptions menu.

Exit Condition: Player chooses a background image.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** ChangePaddle

Participating Actors: Initiated by Player

Flow of Events:

- Player is on the DisplayOptions menu.
- Player clicks on the paddle icon.
    - Program shows possible paddle textures and gives player a choice to choose from these images.
- Player chooses one texture and clicks on it.
    - Program sets the chosen image to be the new texture of the paddle.

Entry Condition: Player clicks on the paddle icon on DisplayOptions menu.

Exit Condition: Player chooses a paddle image.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** ChangeBall

Participating Actors: Initiated by Player

Flow of Events:

- Player is on the DisplayOptions menu.
- Player clicks on the ball icon.
    - Program shows possible ball textures and gives player a choice to choose from these images.
- Player chooses one texture and clicks on it.
    - Program sets the chosen image to be the new texture of the ball.

Entry Condition: Player clicks on the ball icon on DisplayOptions menu.

Exit Condition: Player chooses a ball image.

Quality Requirements: The program's response time should be 0.1 second.

- **Scenario Name:** ExitToMenu

Participating Actors: Initiated by Player

Flow of Events:

- Player is in an ongoing game or in any menu component.
- Player clicks on the ExitToMenu button.
    - Program destroys every object and clears any progress. Program saves the options.

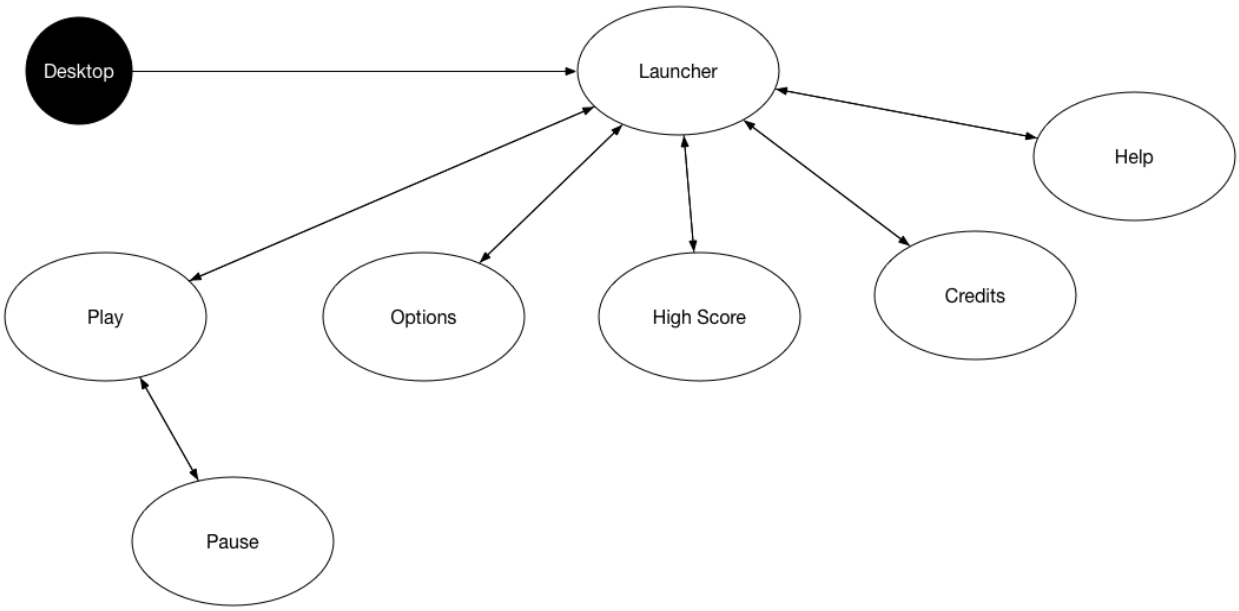
Entry Condition: Player clicks ExitToMenu.

Exit Condition: Player returns to previous menu.

Quality Requirements: The program's response time should be 0.1 second.

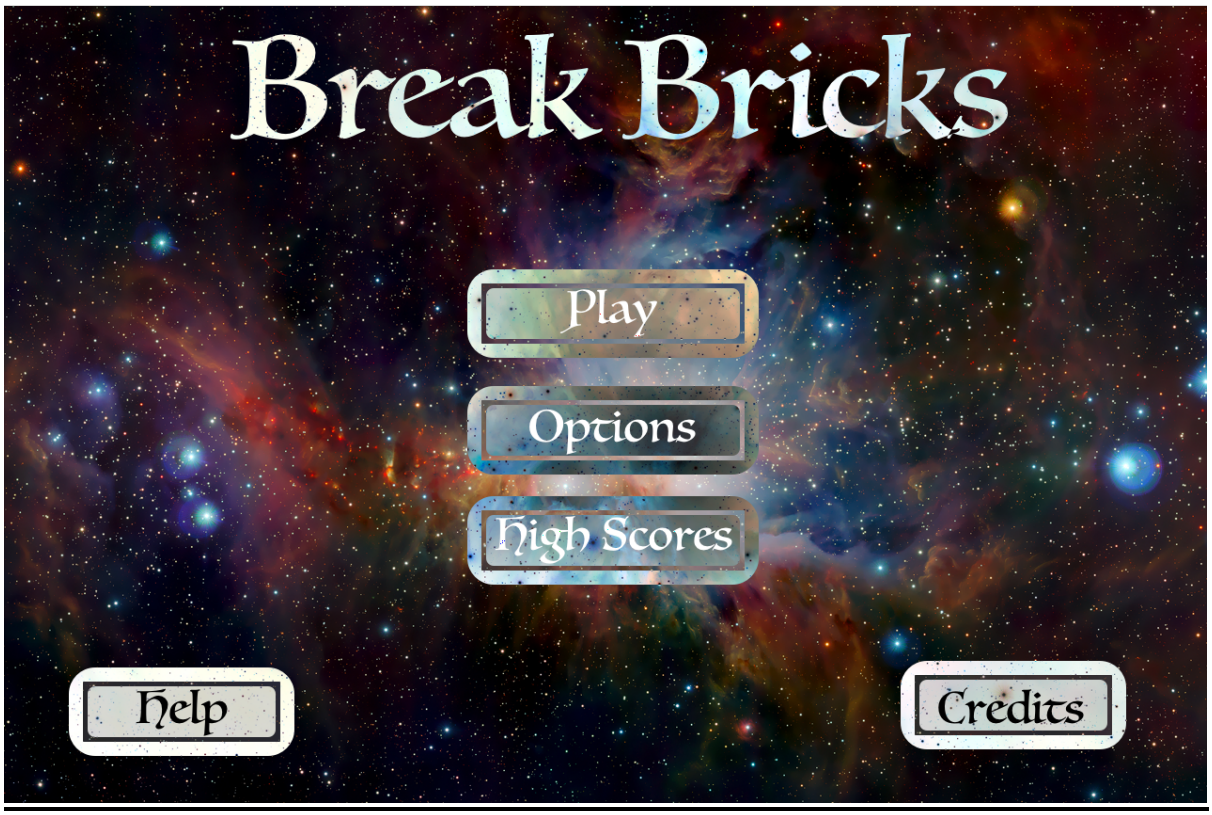
2.7. User Interface

2.7.1. *Navigational Path*



2.7.2. *Screen Mockups*

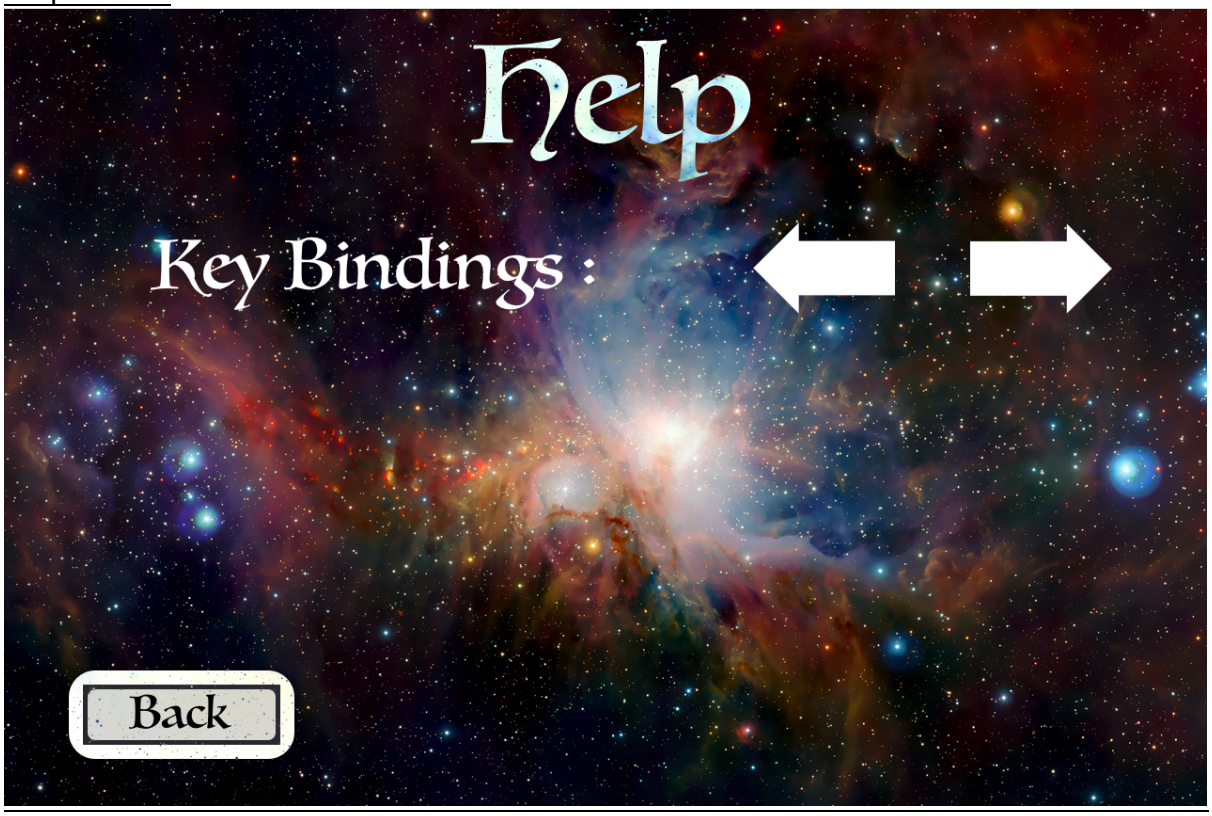
Main Menu:



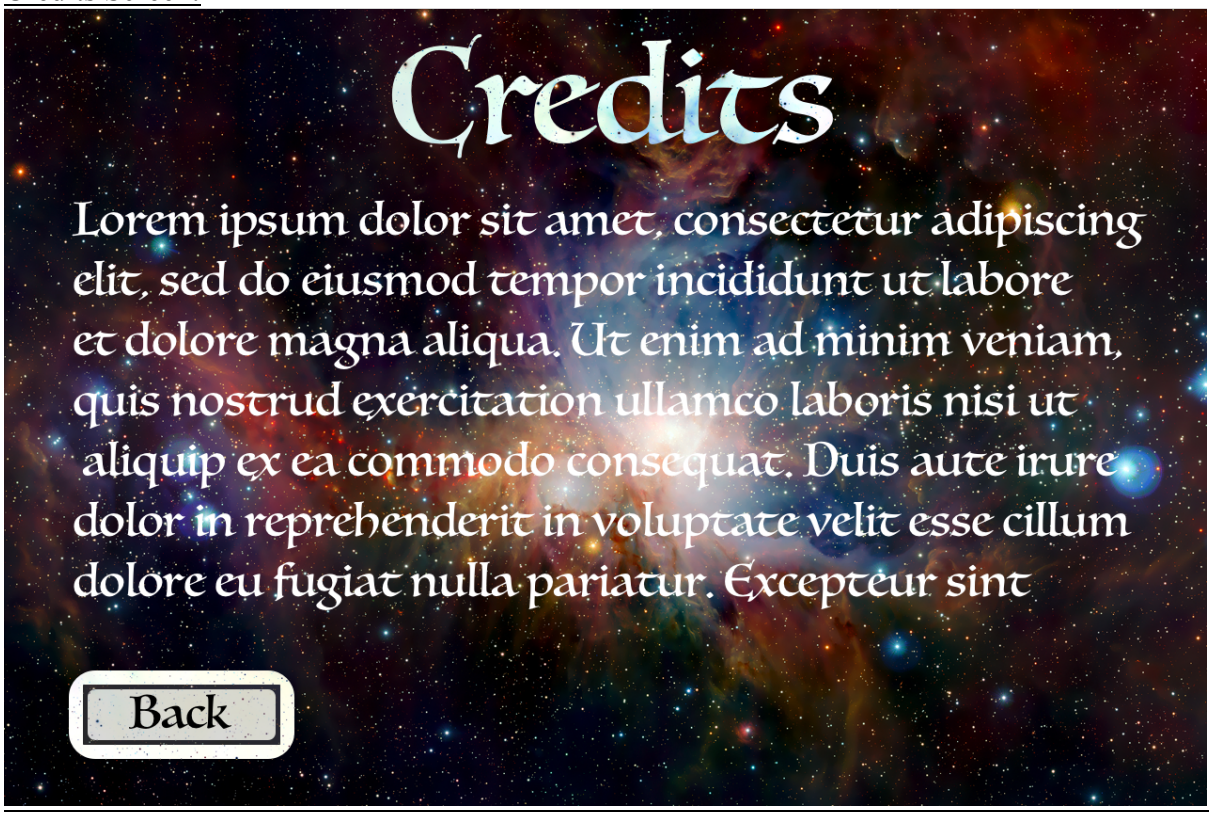
Options Menu:

High Scores Table:

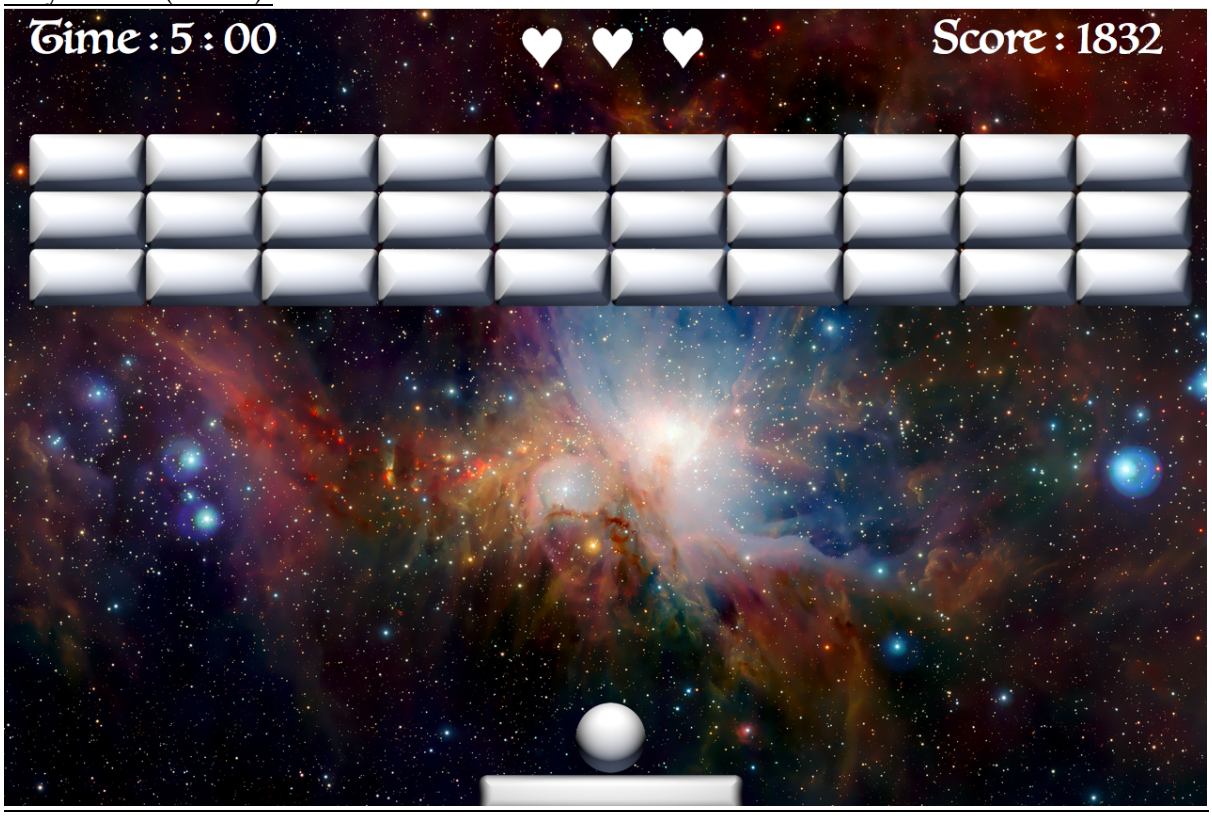
Help Screen:



Credits Screen:



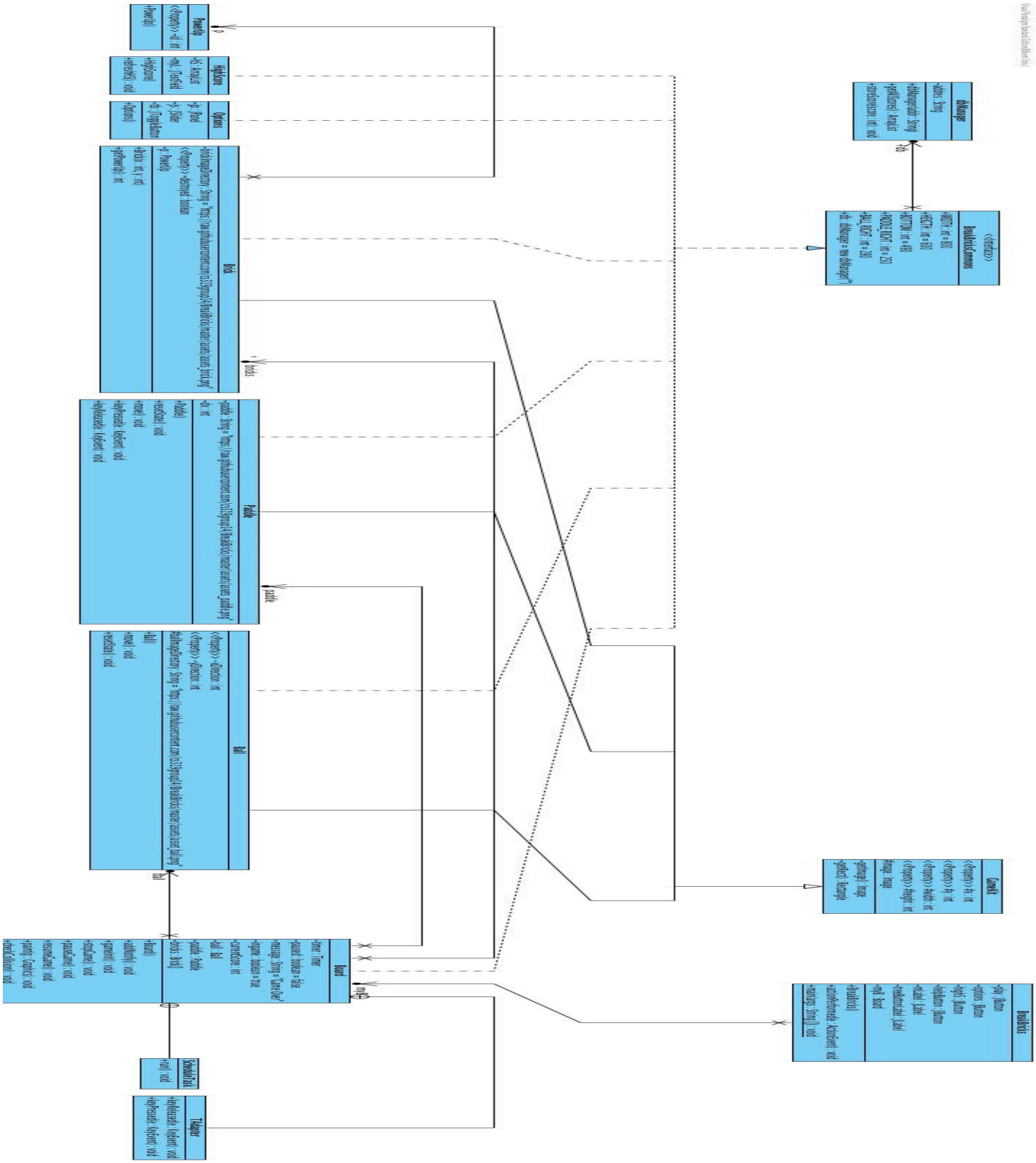
Play Screen (Arena):



3. System Analysis

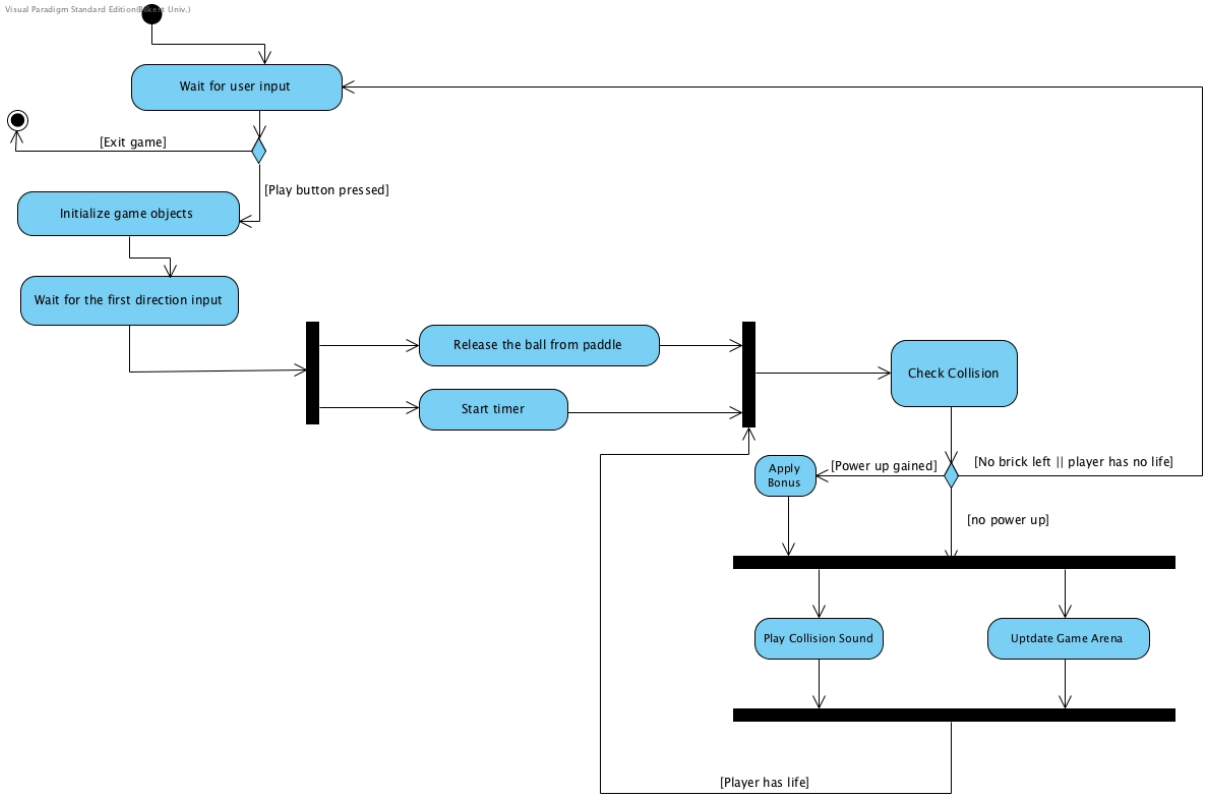
3.1. Object Model

3.1.1 Class Diagram



3.2. Dynamic Model

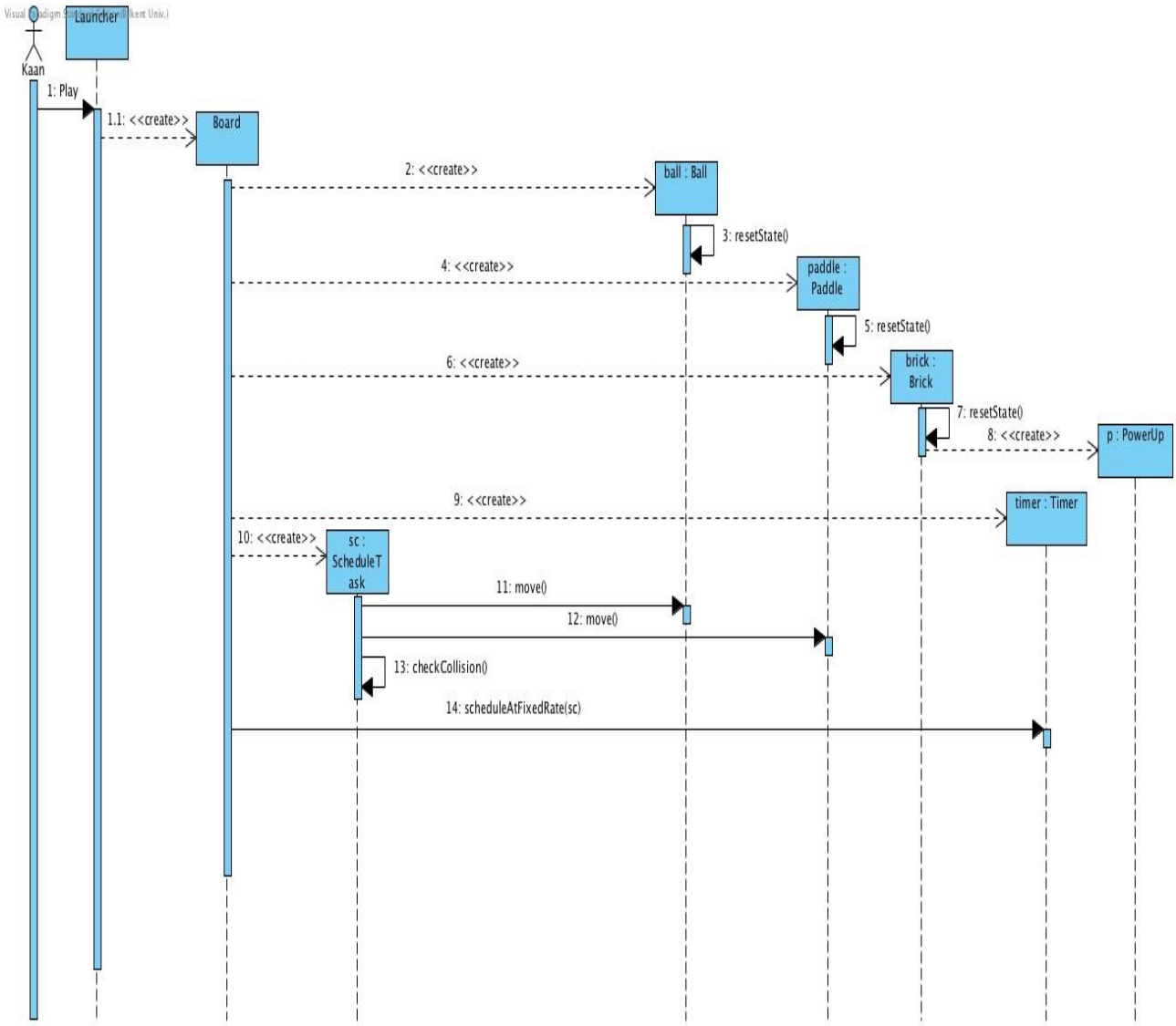
3.2.1. State Chart



3.2.2. Sequence Diagram

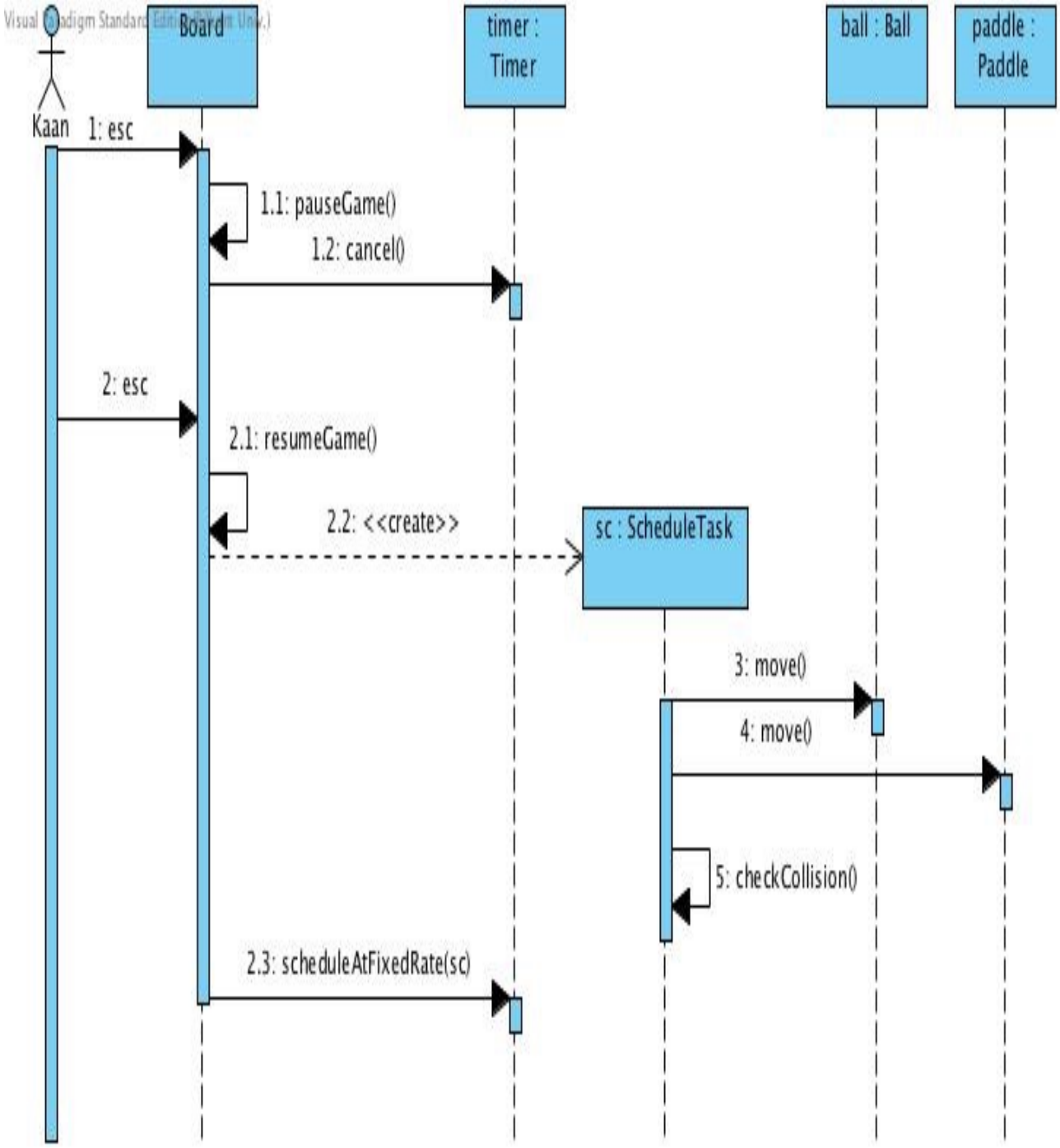
Start Game:

Scenario: Player Kaan presses Play button from launcher. Launcher calls Board object. Board object creates Ball, Paddle objects and also creates a Brick array. each Brick object creates a PowerUp object. PowerUp objects can have good , bad or none affect to the game. Then it resets the starting positions of the objects. Then it creates a Timer to periodically refresh the game. In order to do this it creates a ScheduleTask object. which controls movement of the ball and the paddle and also checks if the ball has collided.



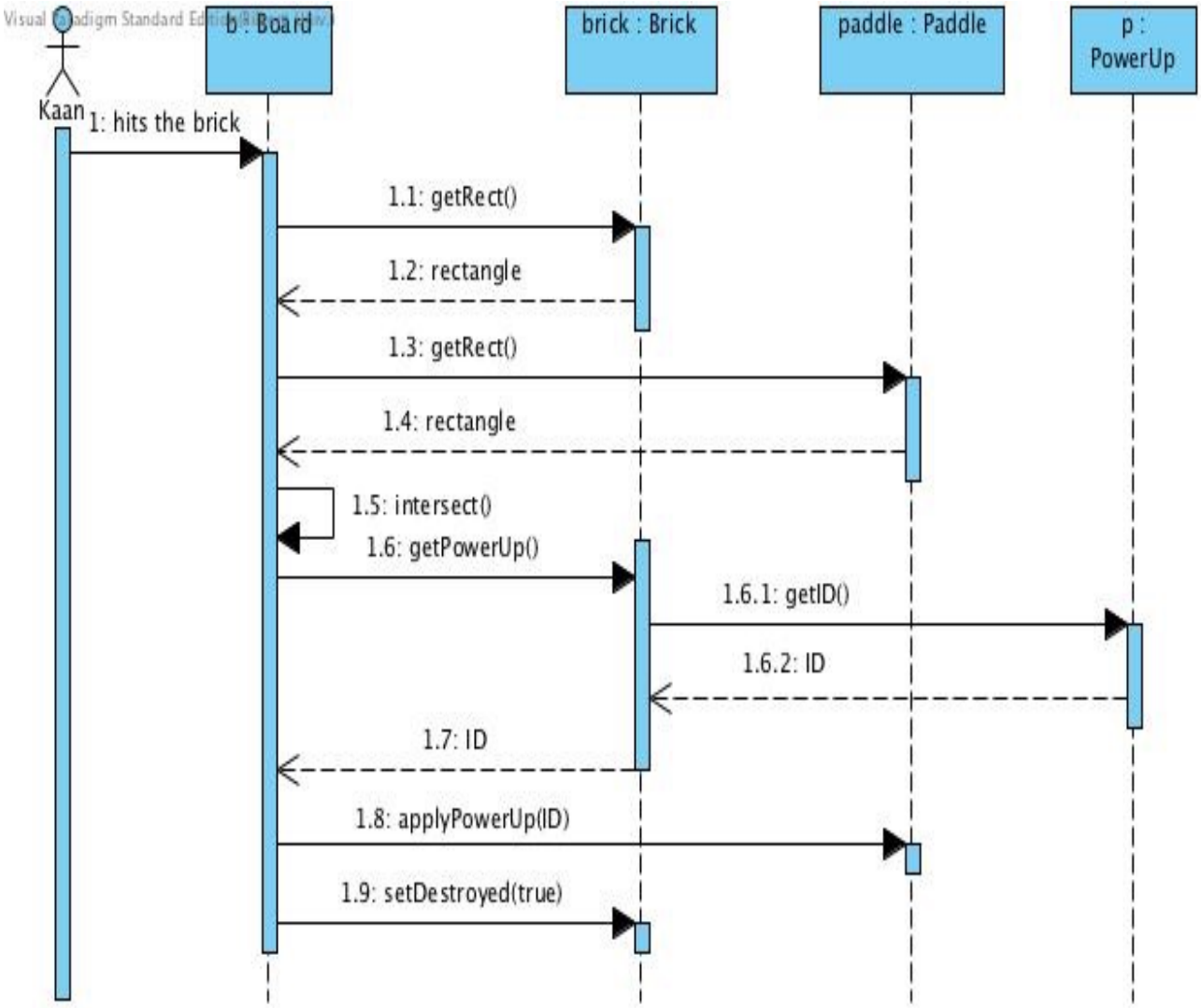
Pause Game:

Scenario: Kaan presses play button then previous diagram occurs. Then while playing the game he presses pause button and game pauses. Then after a delay Kaan presses esc again to resume the game.



Power Up:

Scenario: Kaan presses the play button and the game begins. The first diagram occurs then Kaan hits a brick system checks for any collision if yes then it looks for power up. There is always a power up which is determined from its id. The id 0 is for no power up and others for other power ups. In this game power up automatically gained and no need to pick it up. In this scenario we assume that we get fast paddle power up. The board takes the id of the power up from brick. Then it determines which one is it and sends the id to corresponding object. (For example if power up is about brick, it sends the ID to brick or if it is about paddle, it sends it to the paddle.)



4. Design

Break Bricks is a game that simply designed for users to have fun while improving their reflexes and evaluation skills like many other games. While trying to break bricks or sometimes trying to avoid breaking some other bricks, a player must control paddle effectively in a limited time and because of this reason it can be said that this game can be used to increase players' attention span. Besides, as players will be able to see best scores that have ever received from Highest Scores page, it is aimed to grab attention of the player by creating a virtual experience of challenge. In addition to that, speaking biologically, while trying to get highest score players will activate many regions of their brain such as pleasure circuit by the help of increased rate of dopamine hormone. For further readings see References.

4.1. Design Goals

Before forming this game, it is crucial to detect its designing goals so that it can be shaped in the way that it is most beneficial, efficient and easy to build for as many people as possible. To find and create our own designing goals we used many resources and our analysis report. Here we explained some design goals of our project:

4.1.1. Robustness and Reliability

One of the important goals of our project is to make it a system that works smoothly which means while the user plays the game there should be no interruption. So the program should be bug-free and it should cope with errors during execution without causing crash in whole system. For that, in every level of code writing many tests will be performed and adequate and unambiguous messages that informs coder and user about errors will be appointed. To detect almost all possible errors that may occur during run-time we will use main error finding principles which are *paranoia, stupidity, dangerous implements, cannot happen.* According to these principles we will test on breakability of code, how code handles with abnormal inputs, safety (provided with encapsulation) of code and code's behavior while facing with impossible cases.

4.2.2. Performance

Performance is very important in sense of to keep player playing our game. According to many researches reasonable load times, smooth animations, and responsive interaction gives user a sense of interaction and immersion, whereas slow load times frustrate users, and choppy animation and interaction quickly makes an experience awkward and disconcerting.

a. Memory Usage

We put memory usage's importance after the importance of response time and usability. However, this doesn't mean that we don't give the required importance to it. As memory usage is also related to response time very closely we will choose least memory allocating functions and systems in our game. To make memory allocation easy for us we used Java, which has automatic garbage collection and information system about memory leaks, to implement our code.

b. Response Time

For many games response time is very important to keep players attention we also think that it is one of the most important design goals. The game will give quick responses to user's inputs. It will run at least 40 fps in order to provide smoothness in the movements at game-play screen.

4.2.3. Usability

Usability basically means a system's potential to accomplish the goals of the user. As we mentioned before benefits' of users is very important for us so we aim to make our game easy to understand, easy to play and as attractive as to keep player playing. For this we used user friendly interface that enriched with attractive graphics. To make game easy to understand we used Help Screen that can be reached from Main Menu and to make it changeable according to users' own wish we added Options Menu too. To grab attention of user without confusing them we used clear and pleasant screen graphics. We colorized brick types, ball, paddle, power-ups and punishments that fall from bricks suitably to make them easy to distinguish from each other. In addition to that we kept game-play as easy as possible, a player just uses <- and -> keys to move paddle, so that users will attached to game.

4.2.4. Maintenance

a. Extendibility

Another goal of this game should be related to its usage in future. As in future we or other coders may want to add new features to the game it should be extendible. To make it extendible as possible first code it must be well-structured and easy to understand. In addition to that any change in system's behavior (what system does) should not fully change source code (how system works) itself. To provide that Gray-Box Extensibility can be used.

b. Portability

Portability is an important issue for many games as well as ours. As we want to make this game playable on many platforms as possible we used Java, which gets around compatibility limitations by inserting its virtual machine between the application and the platform that application will run on, to implement our code. With the help of Java our game will run on any platform that JRE (Java Runtime Environment) is installed.

c. Modifiability

Making our game modifiable is also related to composing a well-structured system. By creating efficient sub-systems and lowering the dependency among these sub-systems, any coder who wants to modify or change any part of this game will be able to fulfill his/her aim easily. For example; if a coder wants to modify movements of paddle all he/she have to do is to some addition or subtraction on the sub-system that responsible of paddle's movement and modifications that he/she had made will not affect the how the rest of the system works.

4.2.5 Trade Offs

Main question that should be asked at the beginning of any trade off should be "Who will benefit most". In our game the answer of this question is "The User". Because of that reason it can be said that while making decisions about design goals we always preferred the cases that user benefit most.

a. Memory Usage vs Usability

According to our statement "User should benefit most" usability is more crucial than memory usage. That means we choose to use extra memory to make this game easy to play and understand. So we put extra screen that named Options and Help that are connected to Main Menu. With this choice while we used extra memory we also increased the usability of this game.

b. Maintenance vs Usability

As our main intention was to be sure that users have the most benefit among the people who interacts with the game we choose usability over maintenance. That means we can sacrifice well structure of our code and make its implementation complex to increase its usability more. However we will always seek the fine line between maintenance and usability.

c. Performance vs Memory Usage

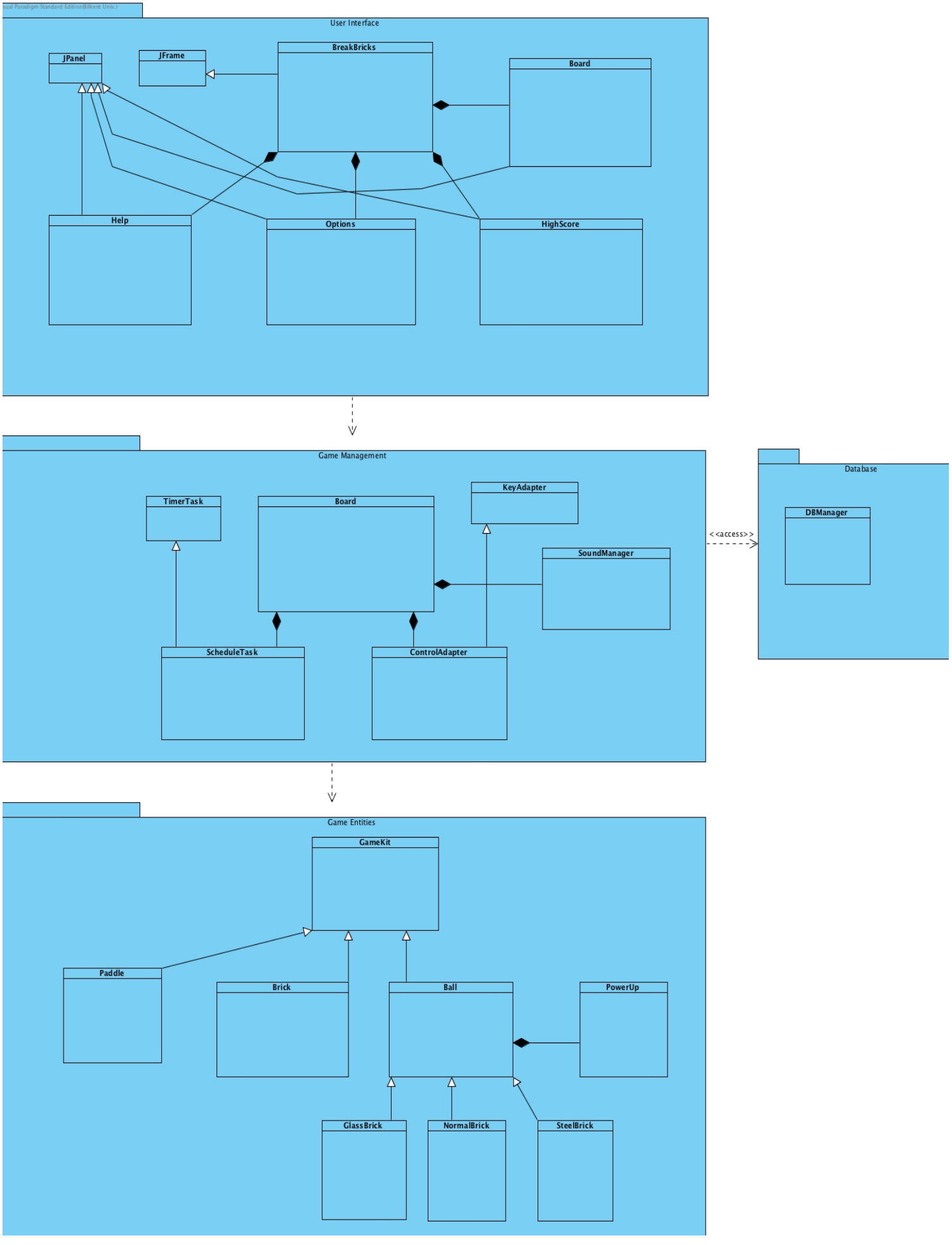
We preferred Java to implement this game. By choosing Java we won simplicity for memory management and lost performance speed compared to C++/ C. In order to reach the optimum game performance, rather than trying to minimize the memory usage, we separated game into many objects. So we give up on memory in order to increase performance and also reliability.

4.2 Sub-System Decomposition

Break Bricks is an arcade game with interactive gameplay. There could be many different approaches to design an interactive game; nevertheless, the most convenient object-oriented design pattern is model-view-controller, formally MVC design pattern.

MVC constructs the main decomposition of classes into subsystems, which may have internal subsystem separation inside. The system is decomposed into three main subsystems: Game Entities, User Interface, Game Management. Game Entities encloses entity objects that are designed to maintain and encapsulate the data, which is responsible application domain knowledge. These entity objects are models of real-world entities and they are designed after analysis of application domain. Game entities are responsible for encapsulating the persistent information which is tracked and used by the system regularly and it provides information to other subsystems. User Interface is basically responsible for rendering the user interface and displaying the relevant game entities with their current states during gameplay. User interface mainly consist of panels and frames, which render and display the user interface. Since user interface is designed to provide interaction between system and users, we can name it as presentation level of the system. User interface needs data from model objects constantly to render graphics and it also should respond according to the inputs provided by the use. To do so, view needs services from another subsystem, which corresponds to Game Management subsystem in our design. Game management is the intermediate connection between user interface and game entities. Therefore, it serves to user interface and game entities as a common interface.

While constructing the subsystems we aimed to meet our system goals. Thus, tried to design an efficient software system with high cohesion and loosely coupling. This will provide the system design with flexibility for any possible change.


User Interface
JPanel
JFrame
BreakBricks
Board
Help
Options
HighScore
Game Management
TimerTask
Board
KeyAdapter
SoundManager
ScheduleTask
ControlAdapter
<<access>>
Database
DBManager
Game Entities
GameKit
Paddle
Brick
Ball
PowerUp
GlassBrick
NormalBrick
SteelBrick

4.3. Architectural Patterns

4.3.1. Layers

In Break Bricks we used closed architectural style. In order to do that we separated our design in 3 layers. These are User Interface, Game Management and Game Entities. User Interface is only responsible to collect data from the user and transmit it to the Game Management. Also User Interface refresh itself according to the data that gathered from Game Management. User Interface can only access to the Game Management layer. Game Management is responsible from the game logic. Game Management layer can only access to the Game Entities layer. Game Entities layer is only responsible for to combine all needed entity objects.

4.3.2. Model View Controller

In our architectural style, we divide it into 3 subsystems. In order to isolate the domain knowledge from user, we used a controller part between user interface and game entities. We grouped the domain objects into Game Entities part. Also Game Entities and domain objects are only controlled by Game Management. The classes that are responsible for user interaction is grouped under User Interface part. User Interface part only communicates with Game Management.

4.4. Hardware/Software Mapping

Break Bricks will be implemented by using Java because of many reasons that explained before and we will use latest JDK. For Java integrated development environment (IDE) for developing our code, we will use İntelliJ IDEA because of its user friendly functions. As hardware configuration, we just need basic keyboard (<- and -> buttons to move paddle) and mouse for accessing different menus from Main Menu. Since this game aims portability and it is quite basic in sense of information processing it will be running on every platform that have JRE (Java Runtime Environment). Because Java's 2D graphics library (Graphics2D class) uses GPU acceleration when available, playing the game on platforms that have GPU would make it game-play smoother. For storage issues, since we will have .txt based structures to form high score list, the operating system should support .txt file formats.

4.5. Addressing Key Concerns

4.5.1. Persistent Data Management

All of the game data, such as high scores or game preferences will be stored in a database. We chose a database to store all the data because that way players' can't just enter the file location and change the high scores like it was a text file. This way we can store much information safe and secure. This system will not be complex, since we store the data only when options are changed or when the game ended to get the high score stored.

There will be only one modifier for the database and that will be the system. Player will only be asked to enter a high score or change the options, which will be the only cases where system modifies the database with player's preferences. There will be no problem about concurrency when modifying the game files since the game is single player. Also database system will not have any human interaction and every action will be done by the system.

4.5.2. Access Control and Security

Only actor in this game is player, and except the database system, player can access every feature of the game. There is no login or validation to limit player's privileges. The database system handles all the security of the players. If the player changes the ball's texture or background texture it will be automatically stored in the database, thus making that preference persistent. This table shows an example of in game access control mechanism for player:

	Player	System
In game	movePaddle()	update()
After game	enterHighScore()	updateDatabase()

Table XX: simple examples of access control mechanism

Our only concern on security is that our game is not file encrypted, hence all the classes can be seen in the file directory. But players will not be able to change the source code since it will not be open to edit.

4.5.3. Global Software Control

The game should be able to use explicit event-driven controls such as moving the paddle and react to player's input as fast as possible, and also make calculations such as calculating ball's movement, at the same time. The control mechanism is provided with an update method which is checked in every tick in the game loop.

The control mechanism has an embed procedure-driven mechanism into it so that even if there are no player input, game still refreshes itself and keeps the flow of events which will be implemented into and handled by the update() method. This update () method resides in the Board class and game will regularly update itself with or without player interaction.

Event-driven control flow allows the game to be highly sensitive against the status of the game objects. By providing this control structure, input will be the key of control and the controller mechanism will be simpler which causes the system to update itself with every input from player.

Our game will not have any concurrency issues since the control mechanism will be centralized, so there will be no performance reduction due to the single-control mechanism. But the drawback of this mechanism is a bottleneck, since it is a centralized system. This bottleneck will be solved by layers of controllers like: collision detect and schedule task and these controllers will interact with each other in a maintainable way.

4.5.4 Boundary Conditions

Initialization

Break Bricks does not require any installation instead it uses ".jar" (Java Archive) executable file to open our game.

Termination

Break Bricks can be terminated by simply pressed the "x" key. If the game is terminated during playing the game all data and current score will be lost. Also at the options menu after some change has been made, if the "x" button has been pressed before saving changes all data will be lost.

Error

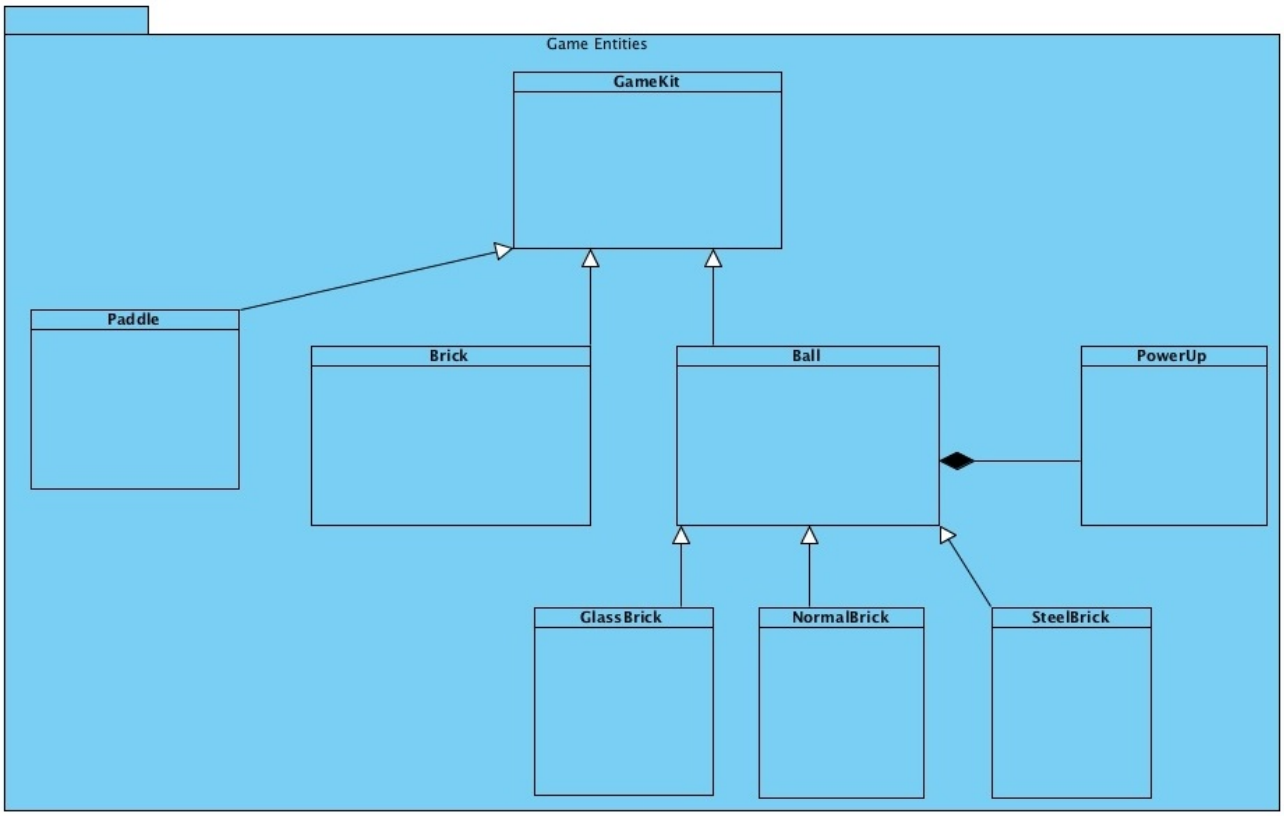
If Break Bricks encounters an error, some sound and image files could not be loaded and the buttons won't work during the game. If Break Bricks is forced to shut down by any reason, the unsaved data such as current game score will be lost.

5. Object Design

5.1. Pattern applications

5.1.1. Façade Pattern

Façade pattern subsystem should provide a common interface for external classes to use, in order to interact with the subsystem in a secure and proper way. In our design, we used Façade pattern for GameKit class. GameKit class provides an interface to all Game Entity subsystem. With GameKit class we are able to encapsulate the game entities by making them reachable only by the functions and attributes that they provide to other subsystems of the whole program.



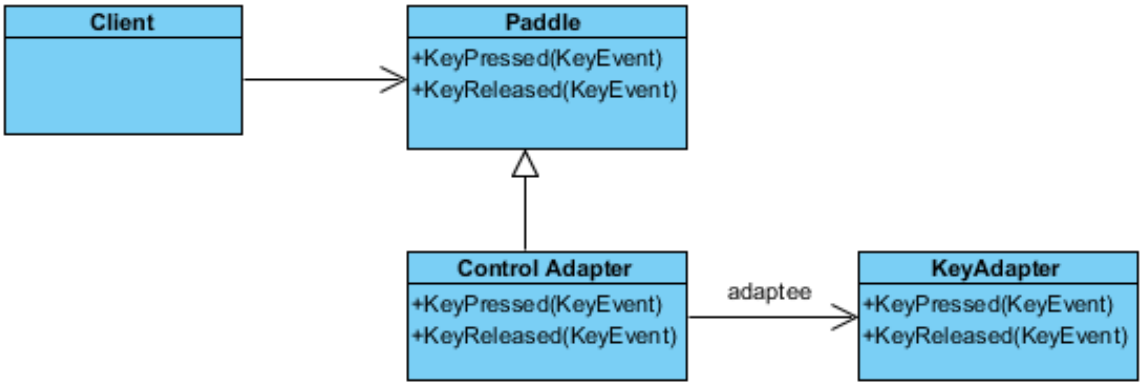
Game Entity subsystem

Various components used in the game itself such as ball, paddle or brick. They are extending the GameKit class so that we can encapsulate these classes and reach them only by function. Since every game entity is encapsulated by this class, there are no other ways to access any other Game Entity subsystem objects from any other subsystem. For example, if a Game Management subsystem member wants to interact with a Game Entity subsystem member, it will always interact with a GameKit object with some predefined functions. Game Entity Subsytem Table shows us this relation in between the classes.

5.1.2. Adapter Pattern

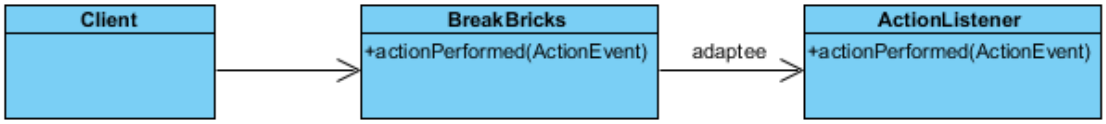
Adapter pattern enables developers to use the functionalities of any already existing class with a modified interface that suits the needs of the client. We used this pattern in our project because we needed this pattern for the adapter class.

KeyAdapter is used by the adapter class to execute one of the most critical functionality which is movement of the Paddle game entity. Paddle relies on key movements and only way to get these key movements is through ControlAdapter which adapts KeyAdapter. Here is the adapter relation of movement system:



Movement adapters and adapter relationship

Similarly, ActionListener is used by the BreakBricks class to navigate through the menu of the game. Graphical user interface relies on this adapter class to listen to the client's input and navigate the client to his/her choice. With the actionPerformed method program detects client's input and execute the necessary operations to provide the client with the desired functionality. Here is the diagram for this adapter pattern:



Menu operations relationship

5.1.3. Observer Pattern

Observer pattern helps observe any change of a state or property of subject. Our program has observer pattern on two classes. BreakBricks class observes the flow and decides when to launch the menu and direct the client to desired functionality. Board class has lots of observation to do since it contains the core of the game.

Board class observes the current state of the game and may decide to stop the game according to the input from client. stopGame, pauseGame and resumeGame methods are executed by the observer on demand and provide the functionality of time control in game. The observer is also responsible for updating the game status so it is in charge of paintComponent, drawObject and checkCollision methods.

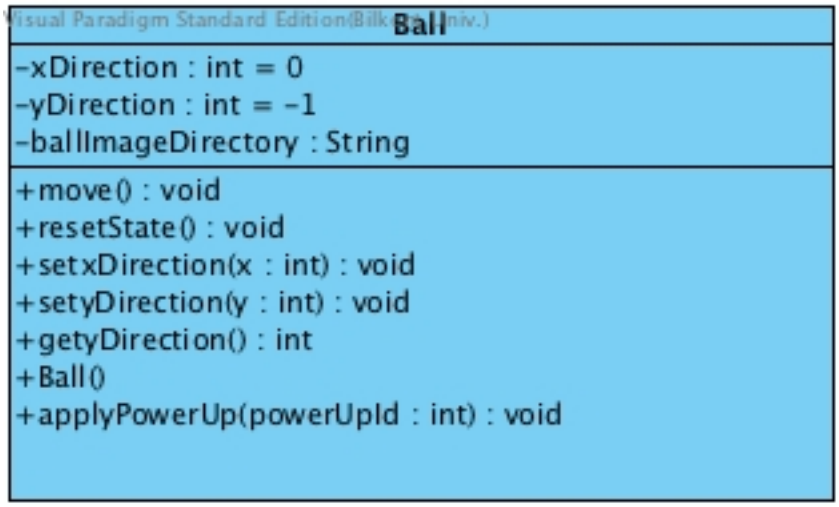
To further examine the behavior of the observer Board class, following scenario can be assumed to demonstrate the observation. If the ball hits a brick, observer detects this collision via checkCollision and removes the brick from the brick array which means the game now needs an update so it stops displaying the destroyed brick. By using paintComponent method observer checks if the game is running or paused and if the game is

running, observer calls drawObjects method which updates the screen. This observer needs to be fast since collision checks and repainting methods must have instant effect, just like movement methods. In a sense, Board class iterates through all the game entities in a tick and checks whether there is a collision or an event happening and act accordingly, then update the state of the game.

5.2. Class Interfaces

5.2.1 Model

Ball Class:



extends GameKit implements BreakBrickCommons

Attributes:

xDirection: direction of movement for the x-axis.

yDirection: direction of movement for the y-axis.

ballImageDirectory: URL of the image

Constructors:

Ball(): creates a ball object with default and database based parameters.

Methods:

move(): determines the location of ball with respect to xDirection and yDirection.

resetState(): resets the location of ball to the default x and y values.

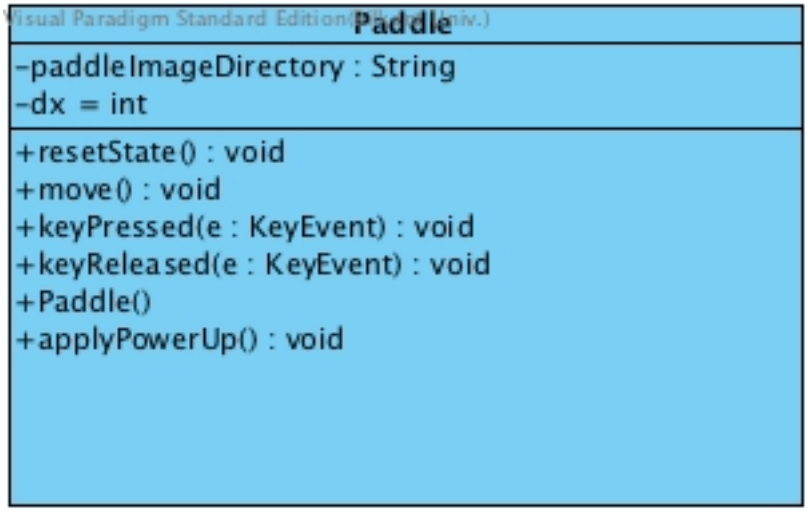
setxDirection(): sets a new value for xDirection variable.

setyDirection(): sets a new value for yDirection variable.

getyDirection(): returns yDirection variable.

applyPowerUp (powerUpId: int): applies the specified powerUp.

Paddle Class:



extends GameKit implements BreakBricksCommons

Attributes:

paddleImageDirectory: URL of the image

dx: denotes the displacement of the paddle.

Constructors:

Paddle(): creates a paddle object with default and database based parameters.

Methods:

resetState(): resets the location of paddle to the default x and y values.

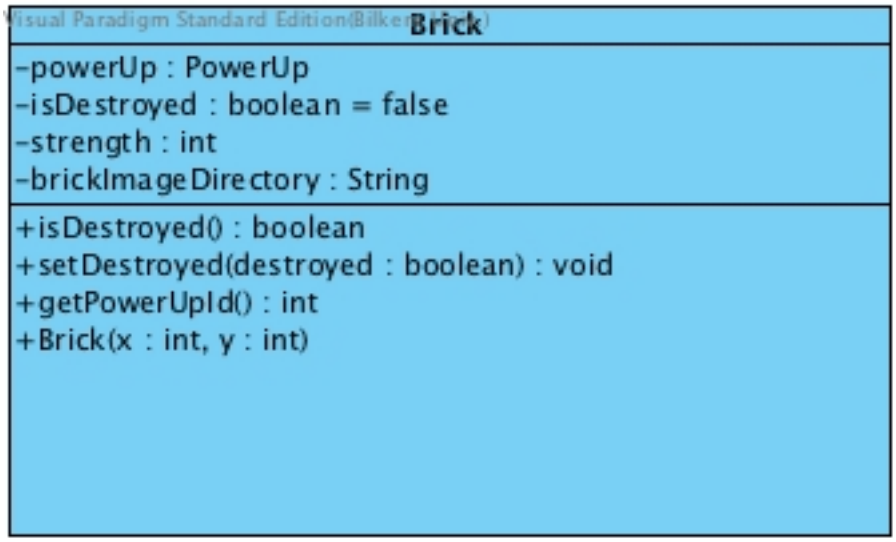
move(): determines new location of paddle with respect to dx variable.

keyPressed(e:KeyEvent): changes dx value to make the movement of paddle possible.

keyReleased(e:KeyEvent): changes dx value to 0 to stop the movement of paddle.

applyPowerUp (powerUpId: int): applies the specified powerUp.

Brick Class:



extends GameKit implements BreakBricksCommons

Attributes:

poweUp: special power up that brick has

isDestroyed: denotes current state of brick

strength: varies according to type of brick

brickImageDirectory: URL of the image

Constructros:

Brick(int x, int y): creates a brick object with specified location

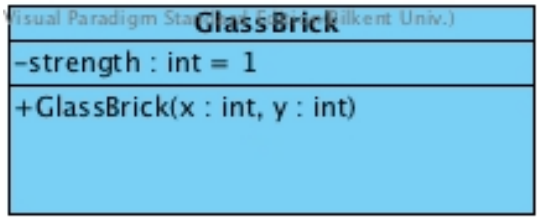
Methods:

isDestroyed(): returns isDestroyed boolean variable

setDestroyed(boolean destroyed): changes isDestroyed with provided destroyed value.

getPowerUpId(): returns id of power up that brick holds.

GlassBrick Class:



extends Brick

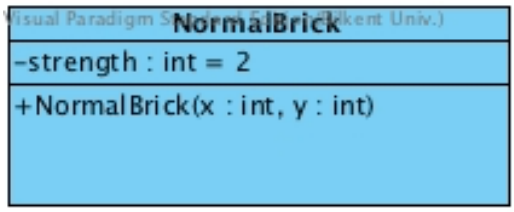
Attributes:

strength: has default value 1

Constructros:

GlassBrick(int x, int y): calls super(x,y)

Normal Brick Class:



extends Brick

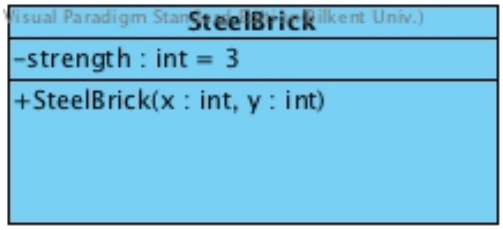
Attributes:

strength: has default value 2

Constructros:

NormalBrick(int x, int y): calls super(x,y)

SteelBrick Class:



extends Brick

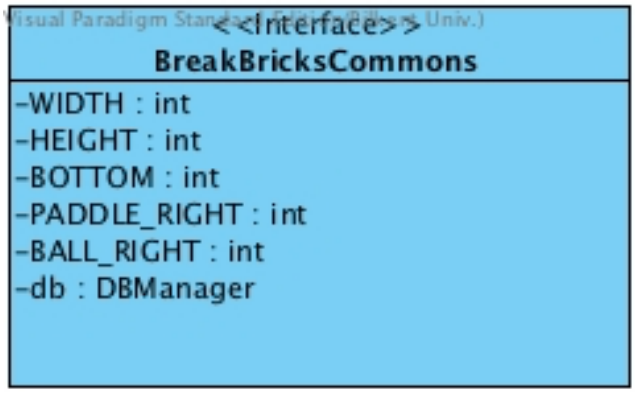
Attributes:

strength: has default value 1

Constructros:

GlassBrick(int x, int y): calls super(x,y)

BreakBricksCommons Interface:



Attributes:

WIDTH: common width for the JFrames

HEIGHT: common height for the JFrames

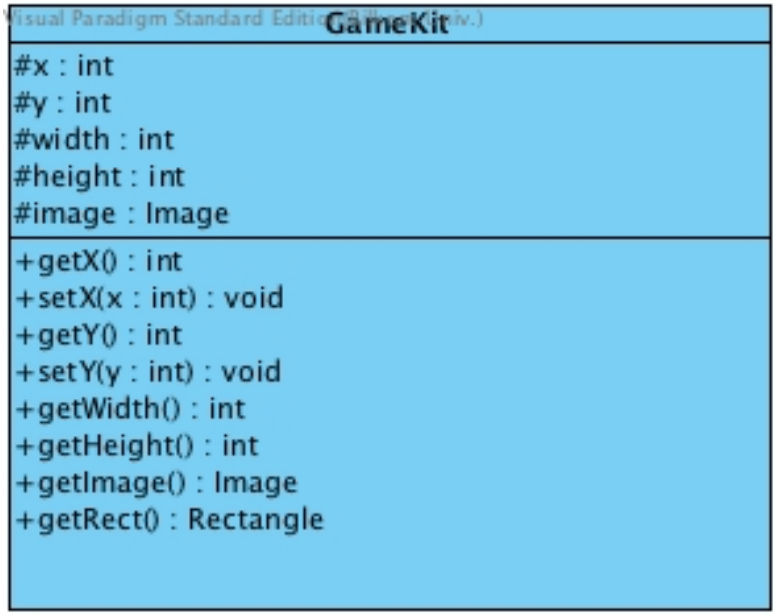
BOTTOM: common bottom value for collision calculations

PADDLE_RIGHT: boundary for the paddle object

BALL_RIGHT: boundary for the ball object

db: common database manager

GameKit Class:



Attributes:

x: location of game entity with respect to x-axis

y: location of game entity with respect to y-axis

width: width of game entity in terms of pixels

height: height of game entity in terms of pixels

image: texture of game entity

Methods:

getX(): returns location of game entity with respect to x-axis.

setX(): sets location of game entity with respect to x-axis

getY(): returns location of game entity with respect to y-axis.

setY(): sets location of game entity with respect to y-axis

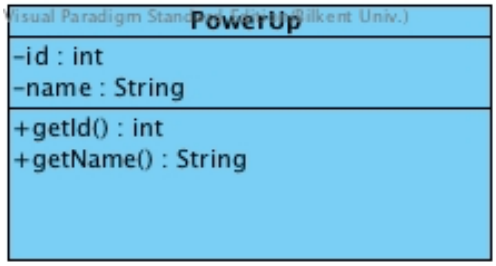
getWidth(): returns width of game entity in terms of pixels

getHeight(): returns height of game entity in terms of pixels

getImage(): returns texture of game entity

getRect(): returns a rectangle objet to make intersects calculations easy

PowerUp Class:



Attributes:

id: unique value to specify power up

name: name of power up

Methods:

getId(): returns id of power up

getName(): returns name of power up

5.2.2. View

BreakBricks Class:

extends JPanel implements ActionListener

Attributes:

play: The button for starting the game and switching to game interface.

options: The button for switching to options interface.

highS: The button for switching to high scores interface.

helpButton: The button for switching to help interface.

mLabel: The label that holds the components of the menu.

treeButtonLabel: The label that holds the options,highS and play buttons of the menu.

myB: The board object.

Constructors:

BreakBricks(): Initialises the menu interface and adds action listener to the buttons of the menu.

Methods:

actionPerformed(): The action listener for the menu objects.

Board Class:

extends JPanel implements BreakBricksCommons

Attributes:

timer: The timer for updating the game.

paused: If true, the game is paused.

ingame: If true, the game is still playing, otherwise the game is finished.

ball: The ball object in the game.

paddle: The paddle object in the game.

bricks[]: The brick objects in the game.

currentScore: The current score of the player.

Constructors:

Board(): Initialises the ball, paddle, timer, bricks ,resets the score.

Methods:

getPaddle(): Returns the paddle.

addNotify(): Tries to start the gameInit method.

gameInıt(): Initialises the game.

stopGame(): Stops the game.

pauseGame(): Pauses the game.

resumeGame(): Resumes the game.

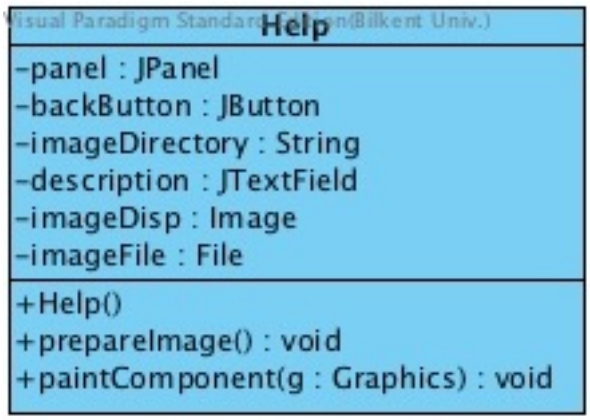
paintComponent(): Paints the components.

drawObjects(): Draws the ball , paddle and the bricks.

gameFinished(): Finishes the game and shows the end-game screen.

checkCollision(): Checks the collusions of the objects.

Help Class:



extends JPanel

Attributes:

panel: Panel for holding the components.

backButton: Returns to the menu.

imageDirectory: The directory of the image.

description: Stores the description.

imageDisp: Stores the image.

imageFile: Required to get the image.

Constructors:

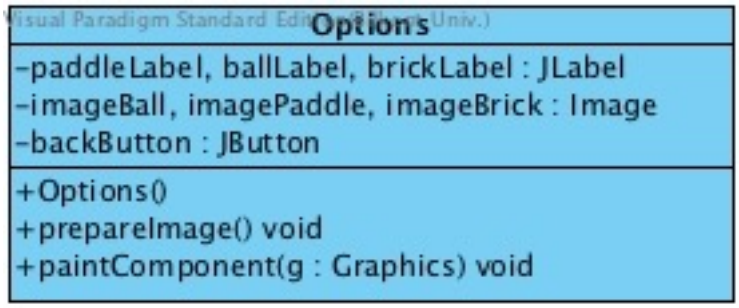
Help(): Initialises the view and index of the interface.

Methods:

prepareImage(): Gets the image from the directory and prepares the image.

paintComponent(): Paints the components of the interface.

***Options Class*:**



extends JPanel implements BreakBricksCommons

Attributes:

paddleLabel: Label for holding the images of the paddle patterns.

brickLabel: Label for holding the images of the brick patterns.

ballLabel: Label for holding the images of the ball patterns.

backButton: Returns to the menu.

imageBall: The image of the ball pattern.

imagePaddle: The image of the paddle pattern.

imageBrick:The image of the brick pattern.

Constructors:

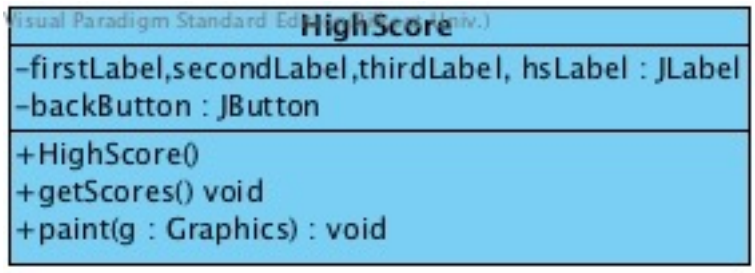
Options(): Initialises the view and index of the interface.

Methods:

prepareImage(): Gets the image from the directory and prepares the image.

paintComponent(): Paints the components of the interface.

HighScore Class:



extends JPanel implements BreakBricksCommons

Attributes:

firstLabel: Displays the highest score.

secondLabel: Displays the second highest score.

thirdLabel: Displays the third highest score.

hsLabel: Displays the title “High Scores” title.

backButton: Returns to the menu.

Constructors:

HighScore(): Initialises the view and index of the interface.

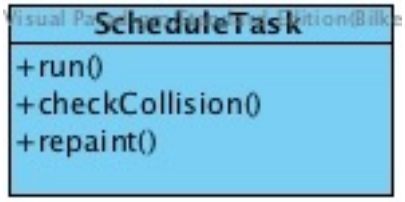
Methods:

getScores(): Gets the stored scores.

paint(): Paints the components of the interface.

5.2.3. Controller

ScheduleTask Class:



extends TimerTask

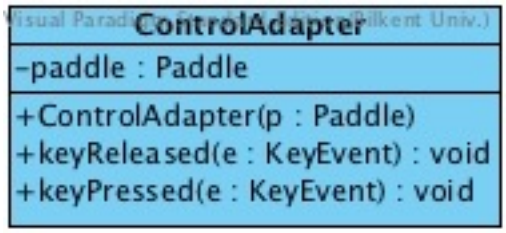
Methods:

run(): Moves the paddle and the ball.

checkCollision(): Checks the collusions.

repaint(): Repaints the components.

ControlAdapter Class:



extends KeyAdapter

Attributes:

paddle: The paddle from the game.

Constructors:

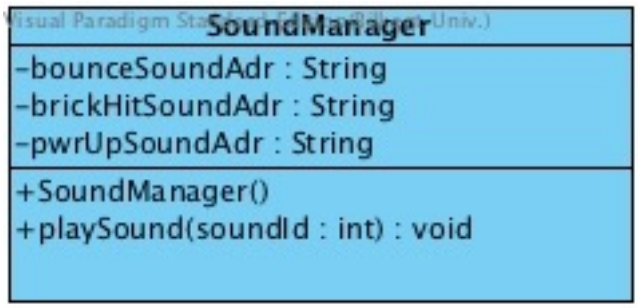
ControlAdapter(): Gets and initialises the paddle.

Methods:

keyReleased(): Detects the key presses and moves the paddle accordingly.

keyPressed(): Detects the key presses and moves the paddle accordingly.

soundManager Class:



Attributes:

bounceSoundAdr: The address of bounce sound file.

brickHitSoundAdr: The address of bounce sound file.

pwrUpSoundAdr: The address of bounce sound file.

Constructors:

SoundManager(): Initialises the sound files and loads them.

Methods:

playSound(): Plays the corresponding sound to the given id.

5.3. Specifying Contracts

1. **Paddle cannot be null**

```
context Board.ScheduleTask.run() inv:
If(paddle != null)  paddle.move()
else                System.out.println("Error null paddle")
```

2. **Ball cannot be null**

```
context Board.ScheduleTask.run() inv:
If(ball != null)
    ball.move()
 else
    System.out.println("Error null paddle")
```

3. **Number of lives must be in between 0 to 3**

```
context Board.stopGame() pre:
livesLeft < 3
livesLeft >= 0
```

4. **Game loop is in ScheduleTask which is extending Java's TimerTask**

```
context Board.ScheduleTask.run() pre:
checkCollision()
repaint()
```

5. **Every game entity extends GameKit**

```
context Ball pre:
public class Ball extends GameKit implements BreakBricksCommons

context Brick pre:
public class Brick extends GameKit implements BreakBricksCommons

context Paddle pre:
public class Paddle extends GameKit implements BreakBricksCommons
```

6. **When a brick gets destroyed, count is incremented by one**

```
context Board.checkCollision() post:
if(Brick[i].isDestroyed)    j++
```

7. Total number of bricks is capped at 30

```
context Board.Board inv:
bricks = new Brick[30];
```

8. A brick can get hit if it is visible

```
context Board.checkCollision() post:
if(!brick[i].isDestroyed){
    ….
    if(brick[i].getStrength() == 0) brick[i].setDestroyed(true);
}
```

9. Paint component method renders game entities with high quality graphics options

```
context Board.paintComponent(Graphics g) pre:
g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);
```

10. Timer of the thread starts after the game arena is completely rendered

```
context BreakBricks.actionPerformed(ActionEvent e) post:
 public void addNotify() {
       super.addNotify();
       try {
          gameInit();
       } catch (IOException e) {
          e.printStackTrace();
       }
       timer.scheduleAtFixedRate(new ScheduleTask(), 1000, 6);
   }
```

11. Board sends a request to get focus on current window when it became visible

```
    context BreakBricks.actionPerformed(ActionEvent e) post:
   if (e.getSource() == play) {
          …
          myB.setVisible(true);
          myB.setFocusable(true);
          myB.requestFocus();
          myB.repaint();
          repaint();
   }
```

12. Size of the frames is common for every frame of the program

```
context BreakBricksCommons
public interface BreakBricksCommons {
   int WIDTH = 1920;
   int HEIGTH = 1080;
}
```

13. End of the game is notified to client via String

```
context Board.gameFinished(Graphics2D g2d) pre:
g2d.drawString(message, BreakBricksCommons.WIDTH - metr.stringWidth(message)) /
                    2, BreakBricksCommons.WIDTH / 2);
```

14. Paddle can reset its position when ball is dropped

```
context Paddle.resetState() inv:
x = 200;
y = 560;
```

15. Ball can reset its position when ball is dropped

```
context Ball.resetState() inv:
x = 240;
y = 520;
```

16. Ball can move according to its prior state

```
context Ball.move() post:
x += xDirection;
y += yDirection;
if (x = 0)                  setxDirection(1)
if (x = BALL_RIGHT)   setxDirention(-1)
if (y = 0)                  setyDirection(1)
```

17. All of the menu buttons must be visible to click

```
context BreakBricks.BreakBricks() inv:
play.setVisible(true);
options.setVisible(true);
highS.setVisible(true);
helpButton.setVisible(true);
```

18. All menu buttons must be linked to an actionListener

```
context BreakBricks.BreakBricks() inv:
helpButton.addActionListener(this);
play.addActionListener(this);
options.addActionListener(this);
highS.addActionListener(this);
```

19. Ball can not go beyond boundaries

```
context Ball.move() post:
      if (x == 0) {
         setxDirection(1);
      }

      if (x == BALL_RIGHT) {
         setxDirection(-1);
      }
      if (y == 0) {
         setyDirection(1);
      }
```

20. Paddle can not go beyond boundaries

```
context Paddle.move() post:
 if (x <= 2)
      x = 2;
 if (x >= BreakBricksCommons.PADDLE_RIGHT)
      x = BreakBricksCommons.PADDLE_RIGHT;
```

6. Conclusions and Lessons Learned

Break Bricks is actually very well-known arcade game which in our version will be enriched by many power-ups, penalties and different kinds of bricks to break. For the implementation of the game Java, which is an object-oriented language, will be used.

The purpose of this game, like many other versions of it, will be to break as many bricks as possible in restrictive conditions. In our version these restrictive conditions are limited time and limited amount of lives that a player can have through a game. The game will be a desktop application and will be controlled by left and right buttons on the keyboard.

As many different objects such as 30 different bricks and their relationship among each other requires deep awareness about the course's main topic, we thought this game can be a great tool to understand and apply what we learned in this course. We aim to design a consistent game that abides by OOP fundamentals which properly models the real-life objects, interactions and events.

We learned very important lessons through development of this project. At first through writing analysis report we learned using UML tools and diagrams to represent components of the project, then we learned generating a functional model using scenarios and producing use cases, transforming functional model into class model and dynamic model to represent the behavior and structure of the system in a real-world situation, decomposing a

given model into meaningful subsystems where it produces a neat and organized relationship chain, deciding on the interactions and dependencies of subsystems and learning about architectural styles and design patterns.

While writing design report of our project we learned the importance of detecting designing goals before starting to write the code, so that it can be shaped in the way that it is most beneficial, efficient and easy to build for as many people as possible.

In the end we completed object design and implementation. We made minor changes on our initial system decomposition in design stage and implemented the project according to our overall reports and decisions.

To sum up, with this project we learned about principles and stages of object-oriented software development, object-oriented software modeling with Unified Modeling Language and some tools for object-oriented development.

7. References

Arpaci-Dusseau, Andrea C., and Remzi H. Arpaci-Dusseau. "Information and Control in Gray-box Systems." Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles - SOSP '01. Print.

"Java's Three Types of Portability." JavaWorld. Web. 20 Nov. 2015. Link:http://www.javaworld.com/article/2076944/java-s-three-types-of-portability.html

"Overview of Design Goals." Microsoft's Design Goals. Web. 20 Nov. 2015. Link:https://msdn.microsoft.com/en-us/library/aa292476(v=vs.71).aspx

"Video Games Can Activate the Brain's Pleasure Circuits." Psychology Today. Web. 20 Nov. 2015. Link: https://www.psychologytoday.com/blog/the-compass-pleasure/201110/video-games-can-activate-the-brains-pleasure-circuits-0

Zyp, Kris. "Memory Consumption: The Externality of Programming."SitePen Blog, 17 Mar. 2015. Web. 20 Nov. 2015. Link: https://www.sitepen.com/blog/2015/03/17/memory-consumption-the-externality-of-programming/