BILKENT UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

CS 319 – OBJECT ORIENTED SOFTWARE ENGINEERING
SECTION 1 – GROUP 7
PROJECT TITLE: CRAZY SHOOTER
FINAL REPORT

TURAN KAAN ELGİN 21302525
ONUR UYGUR 21202165
NİHAT ARDA ŞENDUR 21101240
BENGÜ AKBOSTANCI 21100630

Contents

1. Introduction

We will implement a video game which is called Crazy Shooter. It will be an alternative version of the games called Zuma and Luxor which were created and developed for IOS, Android andDesktop. The main aim of this game will beto destroy a sequence of balls, which ismoving in a maze around a ball shooter, before itreaches to a hole. We have influenced from the following link in the project selection.

http://www.popcap.com/games/zumas-revenge/online

Crazy Shooter will have some different properties than those previous games. We have worked for creating ideas of bonuses, special ball types anddifferent level styles.

In the game, the actual goal will be to destroy the sequence by combining balls with same colors. For this goal, players will throw balls to the sequence by controlling the ball shooter. The game will becomposed of ten levels and difficulty of the game will increase level by level. Whenever players complete a level successfully, they will have a chance to play the next level. Each player will also have a unique account which keeps their progress in the game.

Crazy Shooter will bea Desktop application and flow of the game will be managed by using mouse and some keyboard shortcuts. The gamemay be improved for mobile environment in the future.

The layout of the report is as the following: At first, there is a requirement analysis section which is for determining the requirements, use-cases and user interface of the project. After that there is an analysis section which includes the identification of application domain objects and modelling the dynamic behavior of the system by sequence, state chart and activity diagrams.After analysis, there is a system design section in which solution domain objects are determined and the system is decomposed into smaller subsystems. Also in this section, design goals and key concerns about the system are determined and architecture pattern is developed. Finally, there is an object design section which contains detailed information about the signatures of methods and types of objects. Also, it includes the design patterns and object constraints of the project.

2. Requirement Analysis

2.1 Overview

Crazy Shooter is a kind of Desktop video game. It is straightforward to play and enjoyable. In the game, there is a ball sequence moving in a maze, which is composed of different colors of balls and players' purpose is to destroy the sequence by making

combinations of balls with same colors. For this purpose, they control a shooter which throws the balls from the middle of the screen. If they are not able to destroy the sequence on time, it will go through a hole and they lose. Every successful trial helps players to finish the level faster and their scores will be recorded according to the times they finish the level. Crazy Shooter provides some opportunities for players to enjoy the game more. Sometimes a coin appears in the corner of the screen and it is a gift for players for purchasing some special items from the store of the game. (See Section 2.1.4) Besides, at the beginning of the game, players encounter a menu which directsthem to create their accounts with their usernames and passwords and their names can appear in the high score table of the game. The aim is to play all levels successfully and complete them as much as fast.

2.1.1 Gameplay

Players use the mouse to play the game. Mouse is dedicated to rotate the shooter and throw the balls. Also, when a coin appears in the screen, players have to click on it to get it. However, keyboard isalso assigned for some specific purposes such as some shortcuts. For example, whenever players want to freeze the game, it will be sufficient to press the "Escape" button from the keyboard. Also, they use the menus and login screens of the game with both mouse and keyboard.

2.1.2 Leveling

Crazy Shooterhas 10 different levels. Difficulty increases level by level such that the ball sequence has more balls and move faster in each level.Playersdo not have a chance to access all levels such that whenever theycomplete one level successfully, they have a chance to play the next level. In the beginning of the game, they are able to choose any level they have come before to start with. Indeed, whenever players finisha level, they can play that level again to improve their high scores. Game has three checkpoints which are at the end of 3^{rd}, 6^{th} and 9^{th} levels. When players lose all of their lives, they start from the last checkpoint. 10^{th} level does not have a condition with lives; however it will be more difficult to play according to the first 9 levels as it is the final level.

2.1.3 Screen Elements

Balls are designed with 5 different colors. Colors are blue, red, yellow, green and white. For destroying some parts of the sequence, users must combine 3 or more balls with same color. Moreover, 3 types of special balls are assigned for private functions. They are called as

freeze ball, bomb ball,and backball. These balls have colors as other balls, but they have some identifiers like a letter or symbol on them which can be seen in the User Interface part. (Figure 2.7.16) Freezeballs have anability to freeze the sequence and timer for a while. Bomb balls can destroy more balls than the others by creating an explosion.Backballs are designed to rewinding the sequence. These types of special balls must be combined with 3 or more balls with the same color for using their abilities. If players throw them to wrong parts of the sequence, they lose their abilities. These specific balls make the game more enjoyable.

Sequences include balls with different colors. Whenever players make a successful trial, a number of balls in a sequence are diminished. If players make an unsuccessful trial,the ball they shoot is also added to the sequence, so the level becomes harder to pass.

There is a maze in the level screen which covers the ball sequence. At the end of the maze, there is a hole. If the sequence enters the hole, players lose the level.

In a level, a coin appears in the screen with a random value from 1 to 100 at random times. It remains in the screen for 5 seconds and for gaining it, players must realize it and click on it at that time interval. If they are able to take it, the game adds it to their accounts and they can use it in the game store.

2.1.4 Game Store

There is a page in the game called game store from which players are able to purchase special balls and lives by using their coins. When they purchase a special ball, game adds it to their accounts, and they become able to use it during the game whenever they want. They can use them by clicking on a shortcut and the ball of the shooter becomes that type. When they purchase a live, they can decrease their chance of falling to the last checkpoint, however lives are more expensive than the balls, because they are more valuable.

2.2 Functional Requirements

- Player can authenticate himself by using his/her id number and password.
- System creates a new account for a player by taking his/her username and password.
- System deletesPlayer’s existing account.
- Player can login as a guest without an account.
- System rotates theshooter by Player input in a level.
- System throws a ball by Player input in a level.
- System changes the ball type by Player input in a level.

- System disables the change in ball type by Player input in a level.
- System adds a coin to Player's account by Player input in a level.
- System adds an item to Player's account by Player input in the store.
- System stops the level by Player input in a level.
- System displays help screen by Player input in a level.

2.2.1 Login

There are two options for entering to the game. One of them is having an account, and the other is entering as a guest. Players are able to create their accounts by specifying their usernames and passwords, and the system gives them a unique id. Later, when they enter their ids and passwords, they can play the game, and the system saves their progress. Usernames are saving their names to the high score table as previously told. If they want to play the game one time for trial, they can choose the option of login as a guest, and the system does not save their progress and they are not able to use the store.

2.2.2 Play Game

Crazy Shooter is an alternative version of the games called Zuma and Luxor. The game is like the following: There is a shooter in the middle which has abilities to rotate and throw a ball. The purpose is to destroy a ball sequence, which is moving in a maze around the shooter, by throwing balls to it. The balls have different colors and when three or more balls with the same color come together, they are destroyed. At the end of the maze, there is a hole, and players must destroy the sequence before it reaches to the hole.

During the game, shooter will have some special balls which are bomb, back and freeze balls. Bomb balls have the ability to destroy more balls, so they have a greater impact. Back balls have the ability to make the sequence rewind for a while, so it makes it be more away from the hole. Freeze balls freeze the sequence and timer for a while such that players can destroy the sequence while it is not moving. These balls come either at random, or players can purchase them from the game store and use it whenever they want during the level.

Also during the game, a coin will appear at the corner in the screen periodically with a random value through 1 to 100. However, the chance of its value to be larger than 50 is less than the chance to get smaller values.The coin is small, and its time of appearance is about 5 seconds. If players can realize it and click on it on time, they can earn it for using in the store.

The game has 10 levels and there are checkpoints in 3^{rd}, 6^{th} and 9^{th} level. Players will have 3 lives and when they use all of them, they begin from the last checkpoint, so their progress in

the levels which are after the checkpoint is deleted. When they are starting the game, there is a level table which they can select a level to start from. However, it must be a level which they have come before in their progress. Also, there is a timer in each level for recording the time players finish it for keeping the high scores.

2.2.3 View High Scores

The system keeps the scores inversely proportional to time as (1 / time) * 10000. There is one high score table for all levels, and the rank is like the following: First, it groups the players according to their progress. (Players who have passed Level 2 are higher than the ones who have passed Level 1.) Then, it groups them according to their scores. Players are able to view the first five scores and their ranks. They can select the option from the Main Menu to see the score table.

2.2.4 View Help

Players are able to get help about the game during the level. When they select that option, a frame appears which have the instructions about the game.

2.2.5 Purchase Special Balls and Lives

There is an option called “Store” in the Main Menu which directs the players to a store from which they can purchase special balls (bomb, back and freeze) and also lives. When they purchase a ball, they can make the ball of the shooter be that kind of special ball during the game. However, they must have enough coin to buy them. Also, they can purchase lives for increasing their chances of losing, but lives are more expensive than the balls.

2.3 Nonfunctional Requirements

- Speed of sliding animations should be less than 1 ball size / second.
- Error in the magnitude of throwing angle should be less than 1 degree.
- Response time for user inputs should be less than 1 second.
- The time between concurrent activities should be less than 1 second.
- There should be shortcuts from the keyboard for confirming input, stopping level, and changing ball type during a level.

2.3.1 Proper Animations

In the game, the ball sequence moves in the maze, and the shooter has an ability to rotate. These animations should be proper such that they should occur in enough speed without any errors. Also, when shooter throws a ball, the ball must reach to the target without changing its direction. We will make these animations as proper as possible.

2.3.2 Response Time

Response time of the game for user inputs should be as fast as possible. Some examples for this are creating an account, finding an account, throwing a ball and so on. Especially in concurrent events, the time interval between the events should not be seen by the players such that updating the ball sequence and creating a sound after throwing a ball. Fulfilling this requirement is important for managing the performance criteria of the game.

2.3.3 User-friendly Interface

We will try to make the user interface of the game as user-friendly as possible such that players can be able to play the game easily. Adding some shortcuts to the game is an example for properties of the interface. User-friendliness of the interface is an important criterion for usability of the game.

2.4 Constraints

- The game must be a Desktop application.
- Implementation language must be Java.
- The game must not be an open-source product.

2.5 Scenarios

2.5.1 Login

Scenario: Player enters his id and password to the login screen and confirms his inputs. Then, System searches for the player from its database and finds him. Then it gets his score, progress and amount of coins and opens the game accordingly.

2.5.2 Create Account

Scenario: Player selects “Create Account” option from the login screen. Then, System opens the create account screen. Player enters his username and password and confirms his inputs. Then, System adds his account to the database, and it creates a confirm frame which displays the confirmation message and player id. Player presses the confirm button to confirm his account.

2.5.3 Start Game

Scenario: Player selects the "Start Game" option from the Main Menu. Then, System opens the Level Menu. Player selects the level which he wants to start. System opens the particular level and creates the screen.

2.5.4 Play Game #1

Scenario: Player selects the "Start Game" option from the Main Menu and System does the previous "Start Game" scenario. Then, System starts the timer of the level. After that, the game loop starts. In the loop, at first Player rotates the shooter with mouse for determining where it will throw the ball. Then, he throws the ball by clicking on the screen. System updates the ball sequence and sees that there is a hit, so it creates a hit sound depending on the type of ball. Then, it checks the hole for if it contains a ball or not and it sees that it is empty. It checks if there are more balls in the sequence and it sees that the sequence is not fully destroyed yet. Then, the shooter prepares the next ball to throw and the sequence slides. At the end, System updates the level screen accordingly.

2.5.5 Play Game #2

Scenario: Player selects the "Start Game" option from the Main Menu and System does the previous "Start Game" scenario. After that, the game loop starts. Then, System starts the timer of the level inside the loop since it will use a freeze ball. Player rotates the shooter. After that, Player changes the type of the ball, which is to be thrown, to a freeze ball. After updating the sequence and seeing the hit, System creates a freeze sound. It freezes the timer and updates the screen. Then, System creates a coin in the screen and gives it a random value. It starts the coin timer and after a while it checks if the timer ends or not. Player clicks on the coin and System creates a coin sound. System adds the value of the coin to the account of player and it removes the coin from the screen. Then, it returns to the beginning of the loop and starts the timer again.

Notes: There are not separate scenarios for bomb and back balls, since the only difference of them is their sounds and update of ball sequences and also the back ball does rewinding. Also in the scenarios, some activities must seem to happen at the same time such that preparing the ball of the shooter and updating the sequence. So, users must not be able to see the time intervals between them. (See Section 2.3.2)

2.6 Use Case Models

The use case diagram of the project is the following:

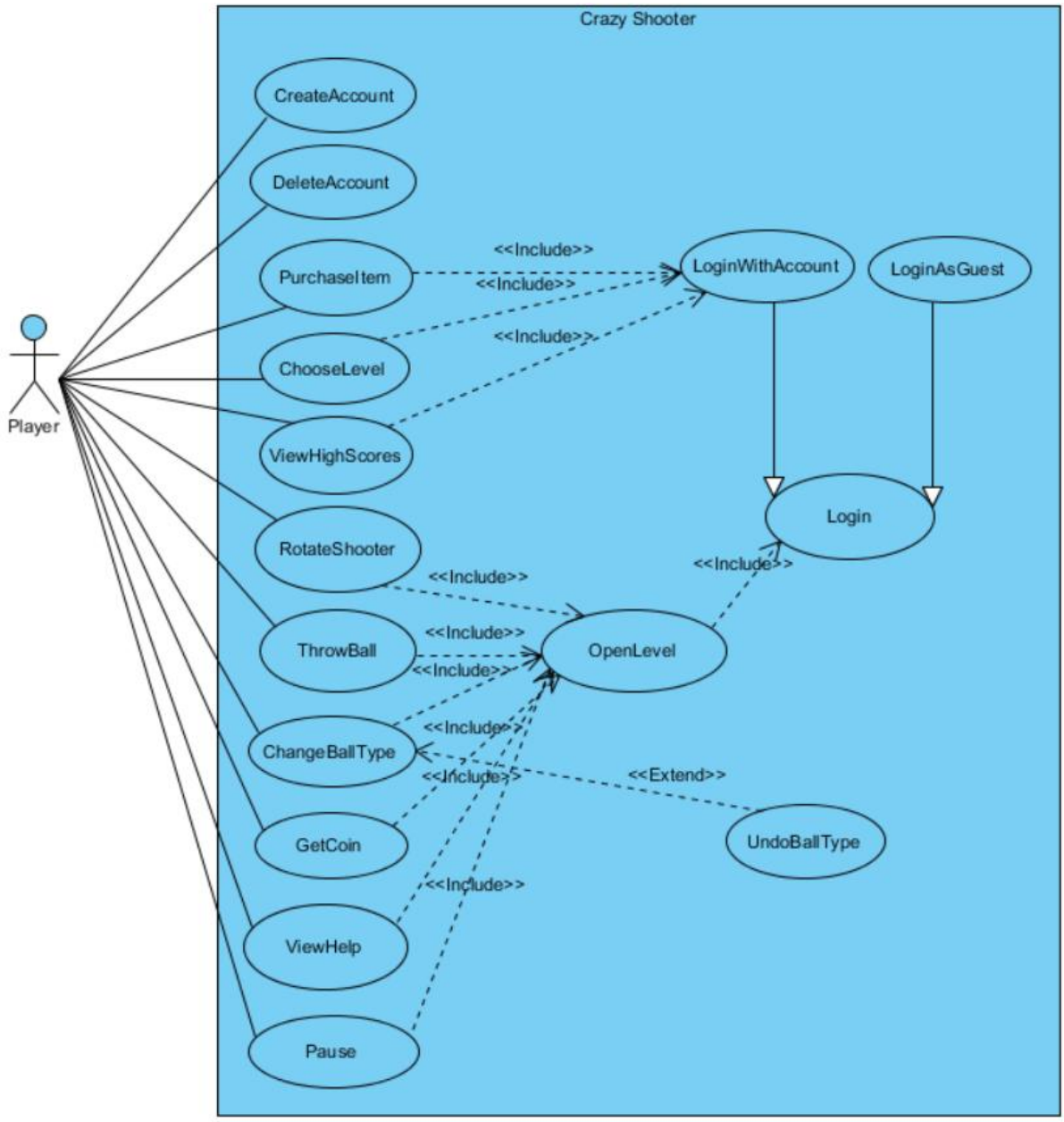


Figure 2.6.1 Use Case Diagram

2.6.1 LoginWithAccount

Use case name: LoginWithAccount

Participating actor: Player

Stakeholders and interests:

Player wants to login to the game with an existing account.

System makes Player login to the game.

Pre-condition: Player must be in the login screen.

Entry condition: System opens Main Menu.

Main flow of events:

Player enters his id and password.

System opens Level 1.

2.6.2 LoginAsGuest

Use case name: LoginAsGuest

Participating actor: Player

Stakeholders and interests:

- Player wants play the game without having an account.
- System makes Player login to the game without keeping any record.

Pre-condition: Player must be in the login screen.

Entry condition: System opens Level 1.

Main flow of events:

1. Player presses the “Login as a Guest” button in the Main Menu.
2. System opens Level 1.

2.6.3 CreateNewAccount

Use case name: CreateNewAccount

Participating actor: Player

Stakeholders and interests:

- Player wants to create a new account.
- System creates a new account.

Pre-condition: Player must be in the login screen.

Post-condition: System creates a new account.

Entry condition: System opens the create account screen.

Exit conditions:

- System creates a new account.
- System returns to login screen.

Main flow of events:

1. System displays “Create New Account” page.
2. Player enters his information.
3. System creates the account.

Alternative flows:

1. System gives warning if player gives inadequate information.
2. If player wants to go back to login page:
    a. Player selects “Back to Login Page” option.
    b. System opens the login page.

2.6.4 DeleteAccount

Use case name: DeleteAccount
Participating actor: Player
Stakeholders and interests:

- Player wants to delete his account from the system.
- System deletes Player’s account.

Pre-conditions:

- Player must have an account.
- Player must be in the login screen.

Post-condition: System deletes the account.
Entry condition: System opens the delete account screen.
Exit conditions:

- System creates the account.
- System returns to login page.

Main flow of events:

1. Player presses “Delete Account” button in the login screen.
2. System opens the delete account screen.
3. Player enters his id and password and confirms them to delete his account.
4. System deletes Player’s account.

Alternative flows:

1. System gives a warning message if the id or password is not true.
2. If player wants to go back to login page:
    a. Player selects “Back to Login Page” option.
    b. System opens the login page.

2.6.5 PurchaseItem

Use case name: PurchaseItem

Participating actor: Player

Stakeholders and interests:

- Player wants to purchase an item from the game store.
- System responds to Player’s request in the store.

Pre-condition: Player must be in Main Menu.

Post-condition: System updates Player’s account by adding the new items.

Entry condition: System opens store menu.

Exit condition: System returns to Main Menu.

Main flow of events:

1. Player presses to “Store” button in Main Menu.
2. System displays the game store.
3. Player clicks on an item which he wants to purchase.
4. System checks Player’s coin amount.
5. If there are enough coins, System gives the item to Player and takes coin from him.
   Steps 3-5 are repeated until Player wants to exit the store.
6. System returns to Main Menu.

Alternative flows:

1. If there are not enough coins, System gives a warning message.

2.6.6 ChooseLevel

Use case name: ChooseLevel

Participating actor: Player

Stakeholders and interests:

- Player wants to choose a level to play.
- System opens the particular level.

Pre-condition: Player must be in Main Menu.

Entry condition: System opens the level menu.

Exit conditions:

- System opens a level.
- System returns to Main Menu.

Main flow of events:

1. Player chooses start game option in Main Menu.
2. System opens the level menu.
3. Player selects a level.
4. System opens the level.

Alternative flows:

1. Player selects return to Main Menu option in the level menu.
    a. System returns to Main Menu.

2.6.7 ViewHighScores

Use case name: ViewHighScores

Participating actor: Player

Stakeholders and interests:

- Player wants to view the high scores and also see his rank in the game.
- System displays the high score table and Player's rank.

Pre-condition:

- System must keep the high scores in a table and also the rank of Player.
- Player must be in Main Menu.

Post-condition: System displays the high score table and Player's rank.

Entry condition: System opens high score screen.

Exit condition: System returns to the Main Menu.

Main flow of events:

1. Player presses "View High Scores" option.
2. System displays the score table.

3. Player returns to Main Menu.

2.6.8 OpenLevel

Use case name: OpenLevel

Participating actor: Player

Pre-condition: Player must have been chosen a level or login as a guest.

Entry condition: System loads the level.

Exit condition: System displays the level.

Main flow of events:

1. System opens a level.

2.6.9 RotateShooter

Use case name: RotateShooter

Participating actor: Player

Stakeholders and interests:

- Player wants to rotate the ball shooter in a level.
- System rotates the ball shooter in the specified direction.

Pre-condition: Player must be in a level.

Entry condition: System rotates the shooter.

Exit condition: System stops rotating the shooter.

Main flow of events:

1. Player moves the mouse.
2. System rotates the shooter.
3. Player stops the mouse.
4. System stops rotating the shooter.

2.6.10 ThrowBall

Use case name: ThrowBall

Participating actor: Player

Stakeholders and interests:

Player wants to throw a ball to the ball sequence in a level.

System throws the ball in the specified direction.

Pre-condition: Player must be in a level.

Entry condition: System throws the ball.

Exit condition: System updates the ball sequence depending on the ball type.

Main flow of events:

1. Player clicks on the screen.
2. System throws the ball.
3. System updates the sequence.

Alternative flows:

3A.If there is a hit, System destroys the particular part of the sequence.

3A.1. If it is a bomb ball, System destroys 4 more balls in addition to the particular part.

3A.2.If it is a back ball, System rewinds the sequence 5 units.

3A.3. If it is a freeze ball, System freezes the timer and sequence.

2.6.11 ChangeBallType

Use case name: ChangeBallType

Participating actor: Player

Stakeholders and interests:

- Player wants to change the ball type of the shooter.
- System changes the ball type.

Pre-conditions:

- Player must be in a level.
- Player must have purchased the particular ball type from the store menu.

Entry condition: System changes the ball type.

Exit condition: System displays the new ball.

Main flow of events:

1. Player presses the particular button from the keyboard.
2. System changes the ball type.

2.6.12 UndoBallType

Use case name: UndoBallType

Participating actor: Player

Stakeholders and interests:

- Player wants to undo his/her change of ball type.
- System disables the change.

Pre-conditions:

- Player must be in a level.
- Player must have changed the currentball type.

Entry condition: System changes the ball type to the old one.
Exit condition: System displays the old ball.
Main flow of events:

1. Player presses the button from the keyboard.
2. System changes the ball type to the old one.

2.6.13 GetCoin

Use case name: GetCoin
Participating actor: Player
Stakeholders and interests:

- Player wants to get a coin.
- System adds the coin to Player's balance.

Pre-conditions:

- Player must be in a level.
- There must be a coin in the screen.

Post-condition: The coin disappears.
Entry condition: System displays the coin.
Exit condition: System adds the coin to Player's balance.
Main flow of events:

1. Player clicks on the coin.
2. System adds the coin to Player's balance.
3. System makes the coin disappear.

2.6.14 ViewHelp

Use case name: ViewHelp
Participating actor: Player
Stakeholders and interests:

- Player wants help about playing the game.
- System displays the help manual in a screen

Pre-condition: Player must be in a level.
Entry condition: System displays the help screen.
Exit condition: System returns to level.
Main flow of events:

1. Player chooses the help option.
2. System displays the help screen.
3. Player closes the frame after reading the instructions.

2.6.15 Pause

Use case name: Pause
Participating actor: Player
Stakeholders and interests:

- Player wants to stop the game at any given time while the game is continuing.
- System stops the game until Player wants to continue the game.

Pre-condition: Player must be in a level.
Post-condition: System continues the game.
Entry condition: System displays pause menu.
Exit conditions:

- System returns to level.
- System returns to Main Menu.

Main flow of events:

1. System stops the level.
2. After a while, Player chooses to continue to the level.
3. System opens the level again.

Alternative flows:

1. Player chooses to return to Main Menu.
    a. System exits level and opens the Main Menu.
2. Player chooses to toggle the sound of the game.
    a. System toggles the sound.

Use cases in Main Menu and login screen have also an exit condition of quit game, but it is not included in the textual descriptions for clarity.

2.7 User Interface

2.7.1 Navigational Path

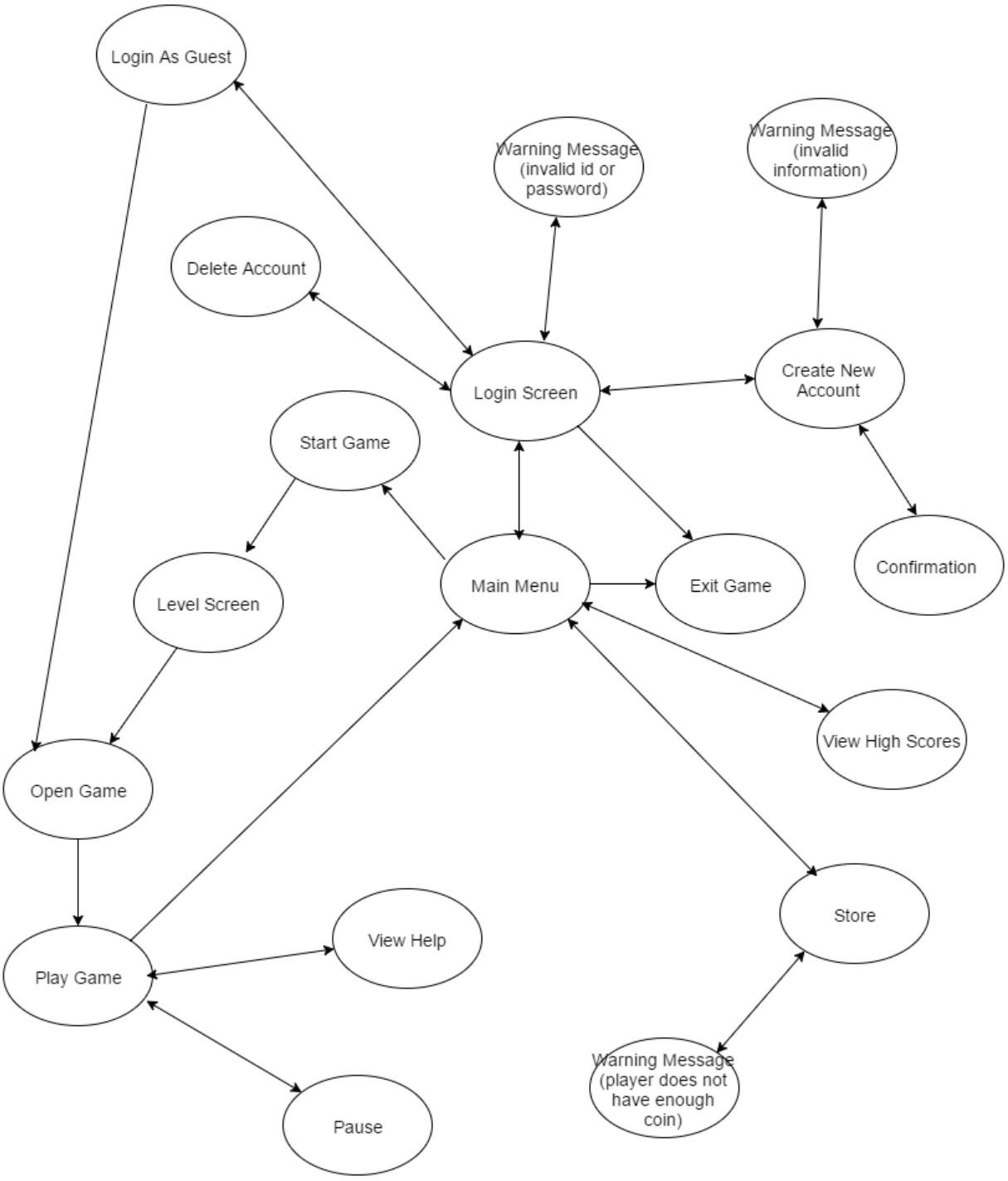


Figure 2.7.1 Navigational Path

2.7.2 Screen Mock-ups

2.7.2.1 Login

When the program begins to run, player sees the login screen. Login screen displays five options to player which are exit, delete account, create new account, login as guest and login. (Figure 2.7.2)

Figure 2.7.2 Login Screen

Exit: If Player selects exit button, the game window closes and Player exits from the application.

Delete Account: If Player selects to delete his account, he will be asked to enter his id and password to delete the existing account. (Figure 2.7.3) After confirming his account, the account will be deleted by clicking on the “Delete Account” button. Player can return to login screen by clicking “Go to Login Page” button.

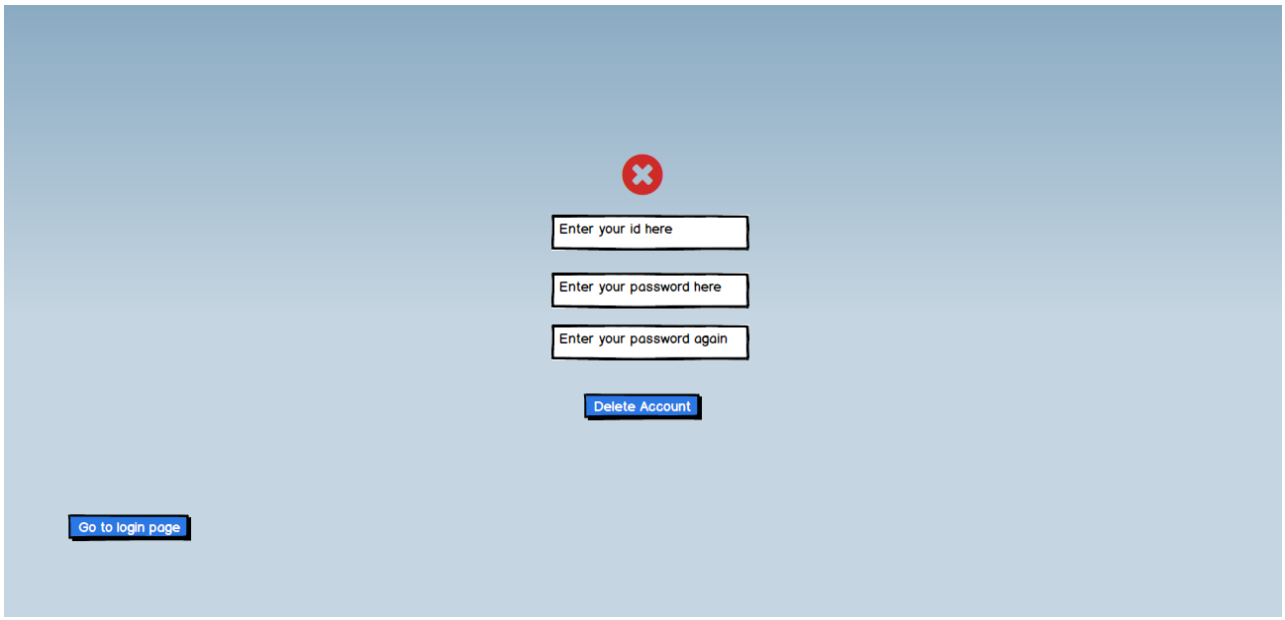


Figure 2.7.3 Delete Account Screen

Create New Account: If Player selects to create new account, he will be asked to enter his username, which will be used in the game, and password to create a new account. (Figure 2.7.4)

If Player does not fill those areas which are required to create a new account, program displays a warning message on the screen. Also same warning message appears on the screen if he does not type his password correctly for the second time. (Figure 2.7.5)

If Player fills the required areas successfully and clicks on “Create New Account” button, a confirmation message appears on the screen. (Figure 2.7.6) Player can go to login screen by clicking on “Go to Login Page” button.

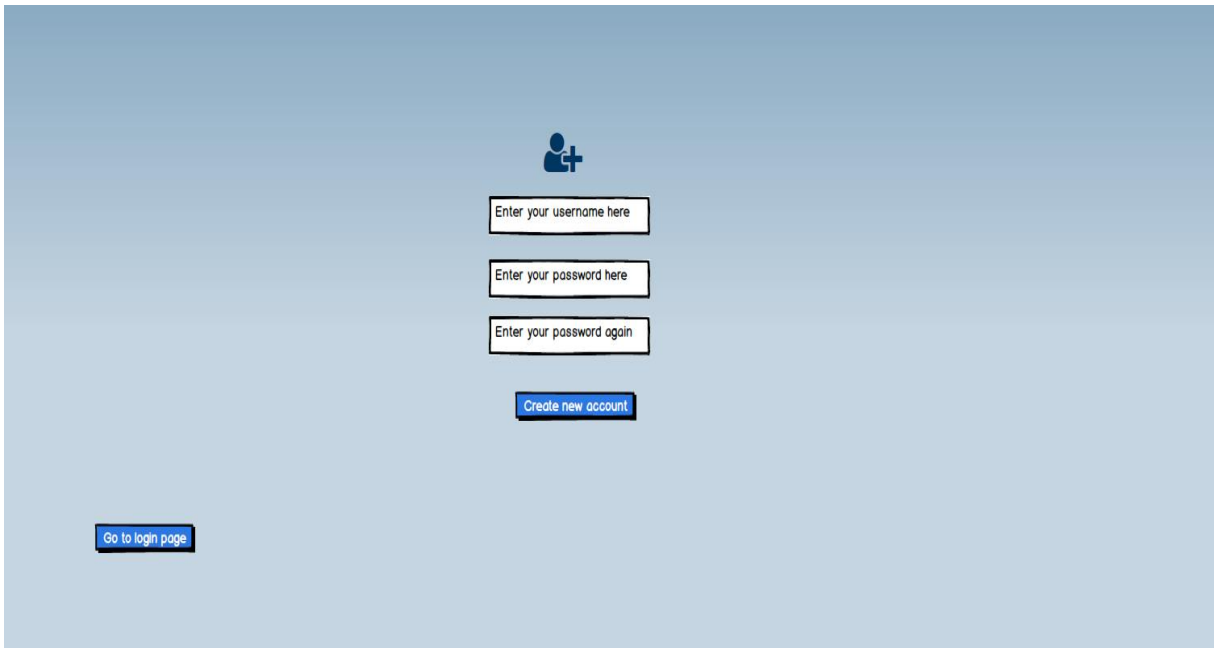


Figure 2.7.4 Create Account Screen

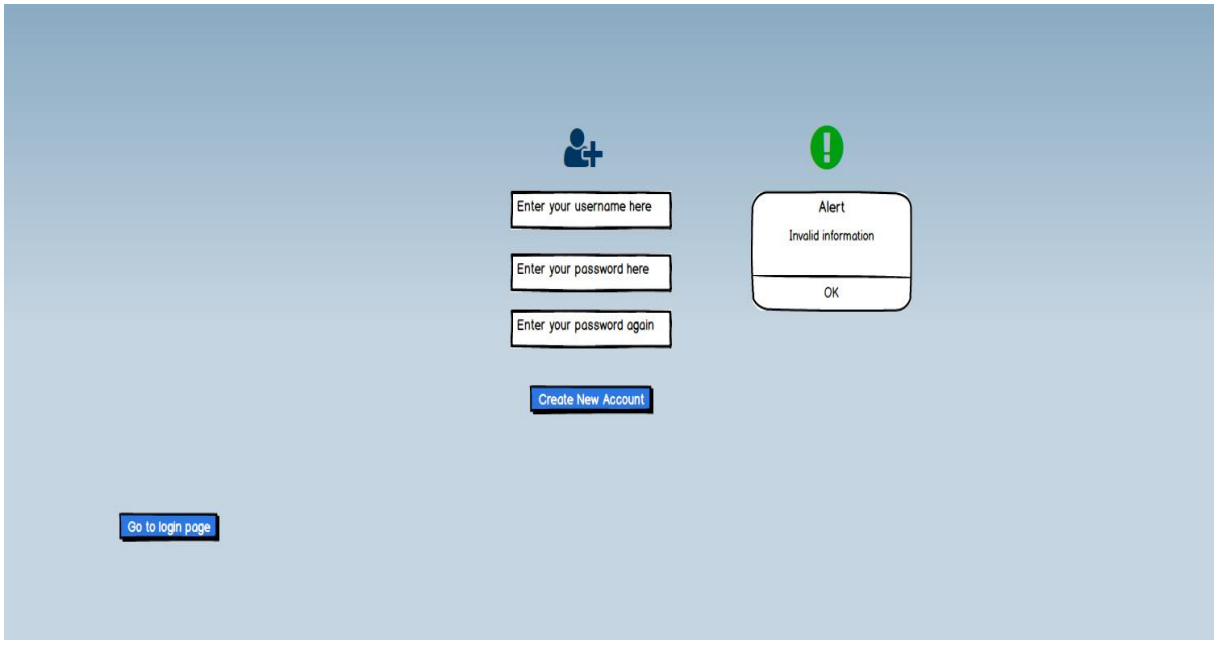


Figure 2.7.5 Warning for Create Account

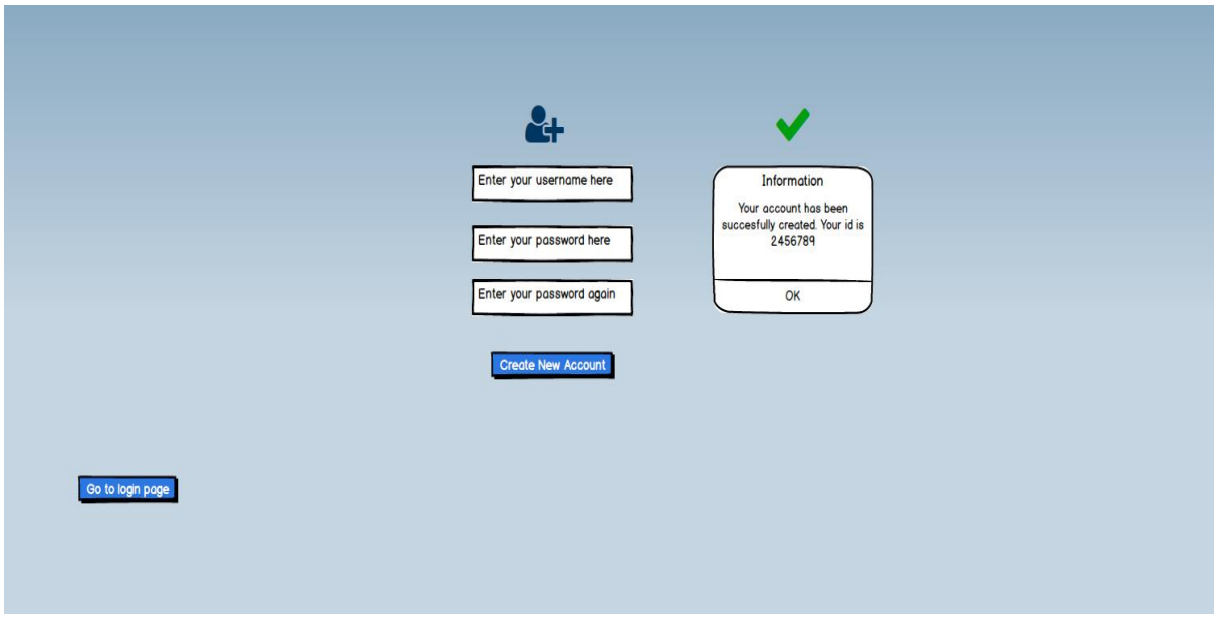


Figure 2.7.6 Confirmation Message for Create Account

Login as Guest: If Player selects login as guest, program opens the first level of the game without holding any record for the player.

Login: If Player has an account, he can login by entering his id and password on the login screen. If Player types his id or password incorrectly or leaves one of these areas empty, program displays a warning message on the screen. (Figure 2.7.7)

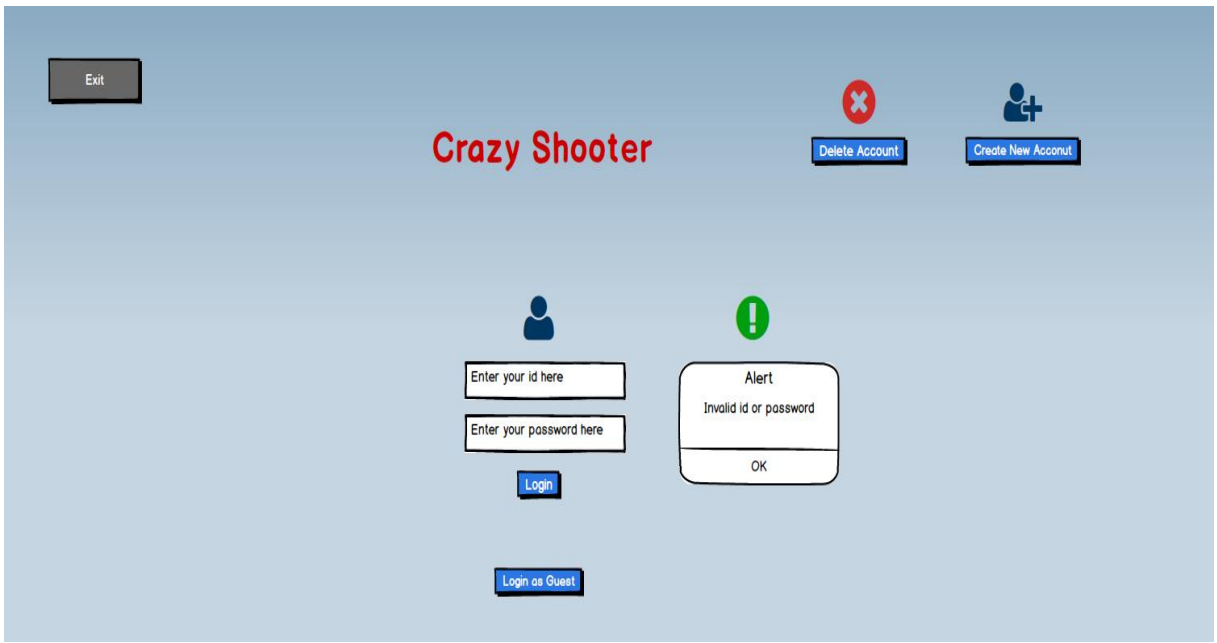


Figure 2.7.7 Warning Message for Login

2.7.2.2 Main Menu

After logging in with an account, Player sees the Main Menu screen which contains five options. (Figure 2.7.8)

Exit: "Exit" button stops the application and closes the game window.

Go to Login Page: "Go to Login Page" button makes Player go to login screen in order to log in with a different account.

View High Scores: Shows the best five scores in the game and Player's rank. Player can return to Main Menu by clicking on "Go to Main Menu" button. (Figure 2.7.9)

Store: Store contains three special balls, namely bomb ball, freeze ball and back ball, and life to purchase. Player's coin is displayed at the top of the screen. If player has enough money, the items can be bought by clicking on the "Purchase" button. (Figure 2.7.10)

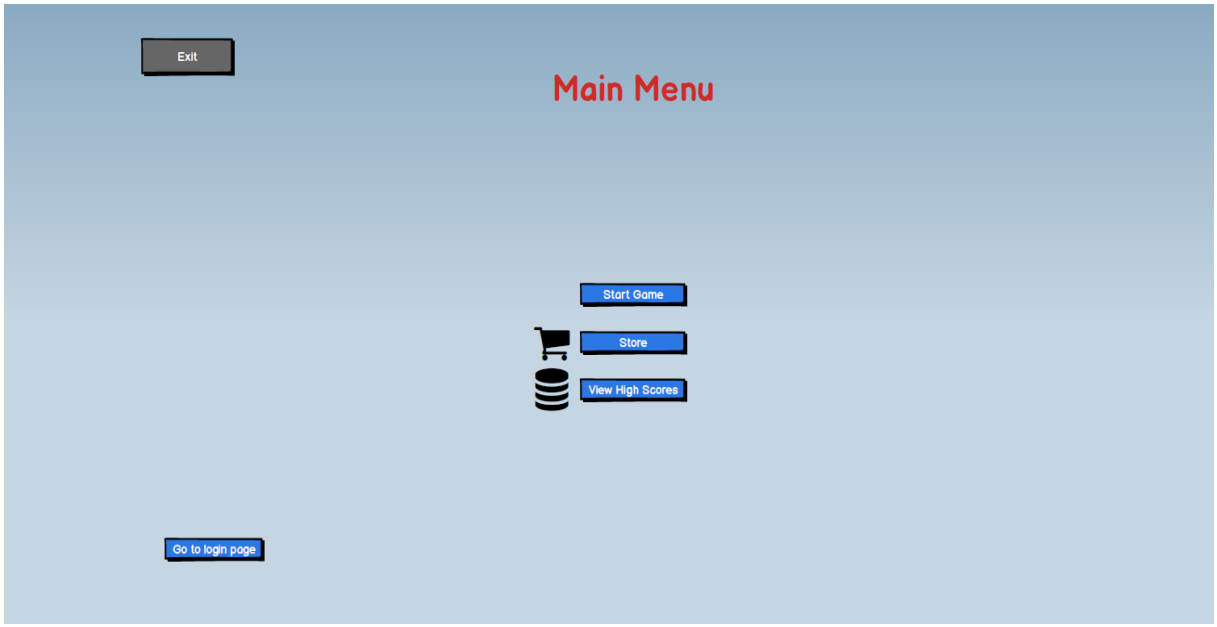


Figure 2.7.8 Main Menu

Figure 2.7.9 High Score Screen

Figure 2.7.10 Store

If Player's coin is not enough to purchase the item, a warning message appears on the screen. (Figure 2.7.11) Player can go to Main Menu by clicking on "Go to Main Menu" button.

Figure 2.7.11 Warning Message for Insufficient Coin

Level Menu: If Player selects the start game option in Main Menu, Level Menu is shown on the screen. The buttons with orange color show the completed levels by the player so far and the buttons with gray color show the levels which have not been completed yet. Player can select only the completed levels to play. (Figure 2.7.12) Player can go to Main Menu by clicking on "Go to Main Menu" button.

Figure 2.7.12 Level Menu

Level Screen: Figure 2.7.13 is a screenshot of the Zuma game which also represents our level screen with slight modifications. When Player selects the level from the Level Menu, level screen appears. Player's coin amount is shown at the top left corner of the screen. Timer and pause buttons are shown at the top right corner of the screen. (Figure 2.7.13) There will also be a shortcut for the pause option as in all games.

Pause Menu: If Player clicks on the pause button, game stops and Pause Menu appears on the screen. (Figure 2.7.14) Pause Menu has three options: Player can return to Main Menu by clicking on "Go to Main Menu" button or login page by clicking on "Go to Login page" button. Sound can be turned on and off by the switch.

View Help: Player can obtain information about the game by clicking on the "Help" button in the place of "Zuma" text in the figure at the top of the screen. If Player clicks on it, help frame appears on the screen. It contains a brief description about how to play the game. (2.7.15)

Figure 2.7.13 Level Screen

Figure 2.7.14 Pause Menu

Figure 2.7.15 View Help

2.7.2.3 Special Balls

Special balls in the game are listed below with their images in Figure 2.7.16.

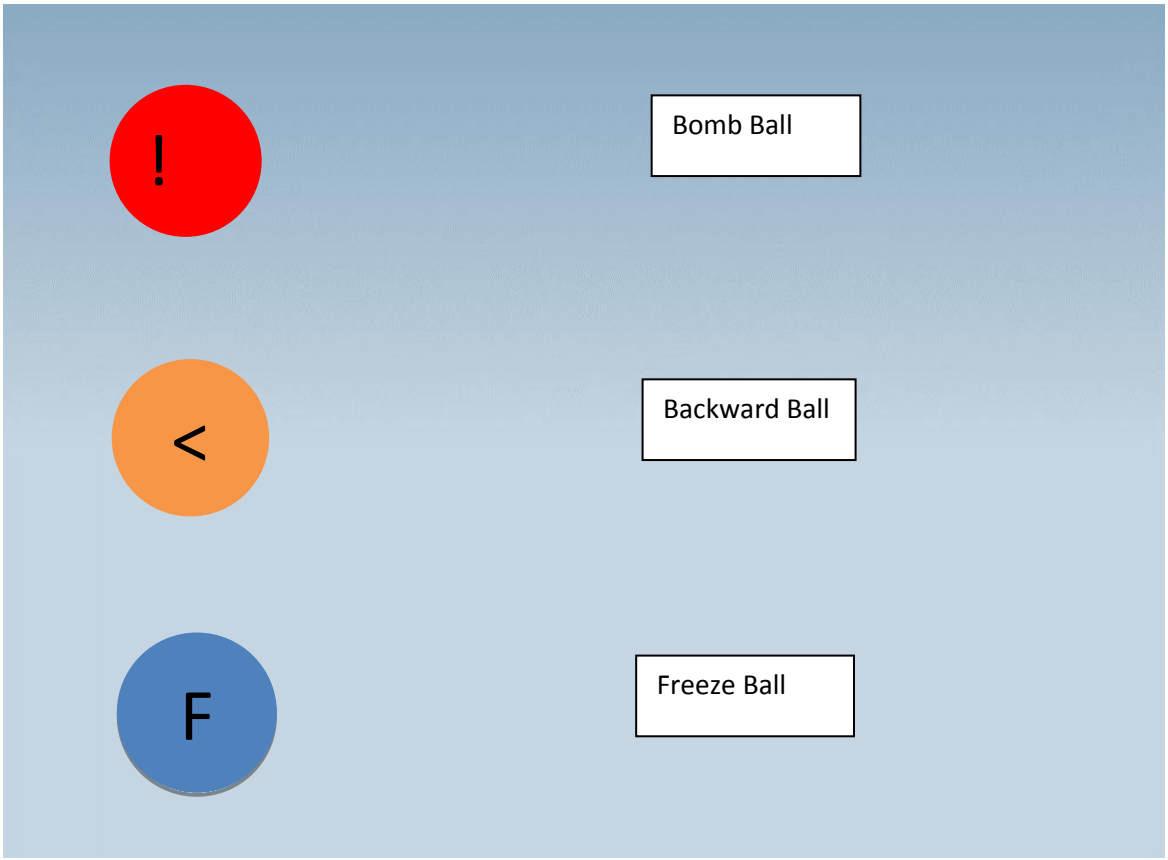


Figure 2.7.16 Special Balls

3. Analysis

3.1 Object Model

3.1.1 Domain Lexicon

There are some terms about the application domain which may be ambiguous for the readers. Their definitions are the below. Some of them were taken from the Oxford dictionary.[1]

- **Level:** Steps of the game which players have to complete individually.
- **Live:** A chance to lose a level.
- **Store:** A place in the game which players can purchase special balls and lives.

[1] http://www.oxforddictionaries.com/definition/

- **Coin:** It is for players to purchase special balls from the store.
- **Maze:** A complex network of path.
- **Checkpoint:** A point in the progresswhere players come back when they lose their lives.
- **Shortcut:** Alternative route which is shorter than other.
- **Rewinding:** Being wound back to the beginning.

3.1.2 Class Diagram

Class diagram of the project is in the next page.

3.1.2.1 Entity Classes

Entity classes represent the models in the game which are controlled by other classes. Their purpose is to provide abstractions for basic elements of the game.

There is an abstract class called ScreenElement which keeps the screen coordinates and draw operation which are the general requirements of all screen element classes. The classes which extend from it are the following: Ball, BallSequence, BallShooter, Maze, Hole, and Coin. Ball class represents a ball with a color and ball type. There are four ball types which are enumerated as the following: Plain, Bomb, Back and Freeze. Their specific properties are told in the Requirements Specification part. BallSequence class keeps a sequence of balls which moves throughout the maze and is destroyed by the players. It has a slide operation which is called in each step of the level and a rewind operation which is for back balls. BallShooter class represents the shooter which is controlled by the players to throw balls to the sequence. It also keeps a ball instance which is the ball to be thrown in the current step. It has functionalities for determining the ball type and set a random color for the ball. Maze class keeps the maze which covers the sequence. There is also a Hole class separate from the Maze for controlling if there is a ball in the hole or not in each step. Coin class represents the coins of the game which have different values and players should catch them for purchasing special balls from the store.

In addition to these, there is a Player class which holds the data of the players such as their usernames, passwords, checkpoints, and so on.

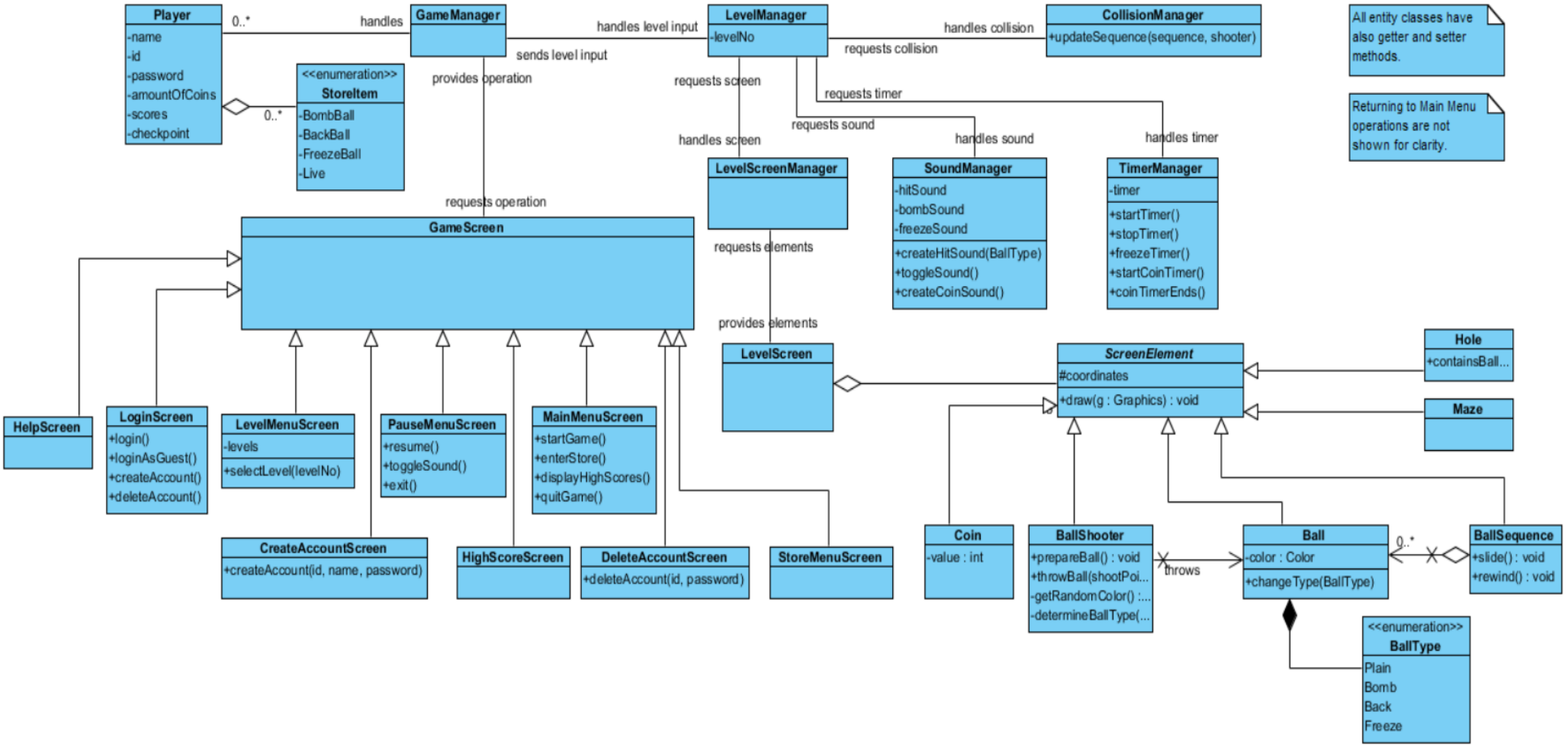


Figure 3.1.1 Class Diagram

3.1.2.2 Boundary Classes

Boundary classes are for making the interactions between the players and the game by providing user interfaces.

There is a GameScreen class which is the generalization of the screens in the game. It is for adjusting common properties of screens such as their sizes and listeners. There are ninescreens in the game: MainMenuScreen, StoreMenuScreen, PauseMenuScreen, LevelMenuScreen, LoginScreen, CreateAccountScreen, DeleteAccountScreen,HelpScreen, and HighScoreScreen.MainMenuScreen contains the options of starting the game, entering to the store, displaying high scores and quit game. StoreMenuScreen contains the items which players can purchase. LevelMenuScreen is for selecting which level player chooses for starting the game. PauseMenuScreen appears during the level which users can exit or resume the level or toggle the sound of the game. Players login to the game from theLoginScreen. They can create and delete their accounts from CreateAccountScreen and DeleteAccountScreen respectively. They can see high scores of the game and their own ranks among the scores from HighScoreScreen. For playing the game, they interact with LevelScreen. During the game, they get help about the level from HelpScreen.

3.1.2.3 Control classes

The purpose of control classes is to control the operations in the game, such as updating a view or model.

GameManageris main control class such that it handles the requests comes from the game screens by interacting with the other control classes. Also, it holds the accounts of players.

LevelManager class serves as the main level control class while playing the game and it handles the requests of LevelScreen. It interacts with three other control classes which are SoundManager, TimerManager and CollisionManager. SoundManager handles the sound effects of the game which are created in collisions. TimerManager deals with the timer of the level which flows during the level and determines the score of the player. The class also handles the time of appearance of a coin in the screen by keeping a timer for the coin. On the other hand, it has an operation for freezing the time for the freeze balls. CollisionManager determines if there is a hit or not, and it updates the ball sequence accordingly.

3.2 Dynamic Models

3.2.1 Activity Diagrams

3.2.1.1 Login

Activity diagram for “Login” use case is the following:

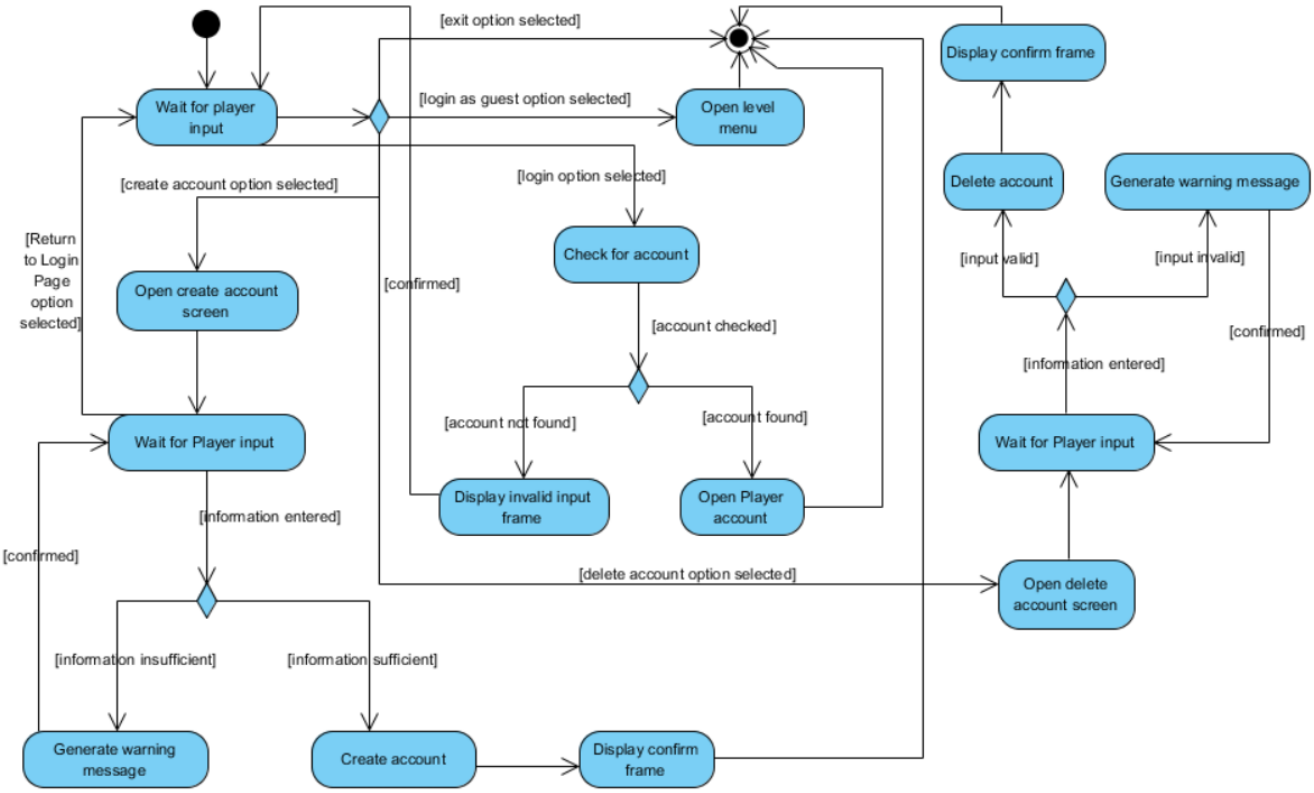


Figure 3.2.1– Activity diagram for “Login”

There are four cases:

1. If Player chooses to login directly:

System takes his id and password as inputs, checks for an account with the given id and password; then if they are valid, it opens the account. If they are invalid, it displays a warning frame and waits for an input again.

2. If Player chooses to login as a guest:

System opens the level menu directly.

3. If Player chooses to create a new account:

System takes his username and password as inputs. If Player's input is enough, System creates an account and displays a confirm frame and in that frame, it also displays Player's id. If input is not enough, System generates a warning message and waits for an input again. While System is waiting for an input, Player also has an option to return to Login page.

4. If Player choose to delete his account:

System takes his id and password as inputs and checks for an account. If they are valid, it deletes the account and displays a confirm frame. If they are not valid, it displays a warning message and waits for an input again.

3.2.1.2 Play Game

Activity diagram of "Play Game" use case is the following:

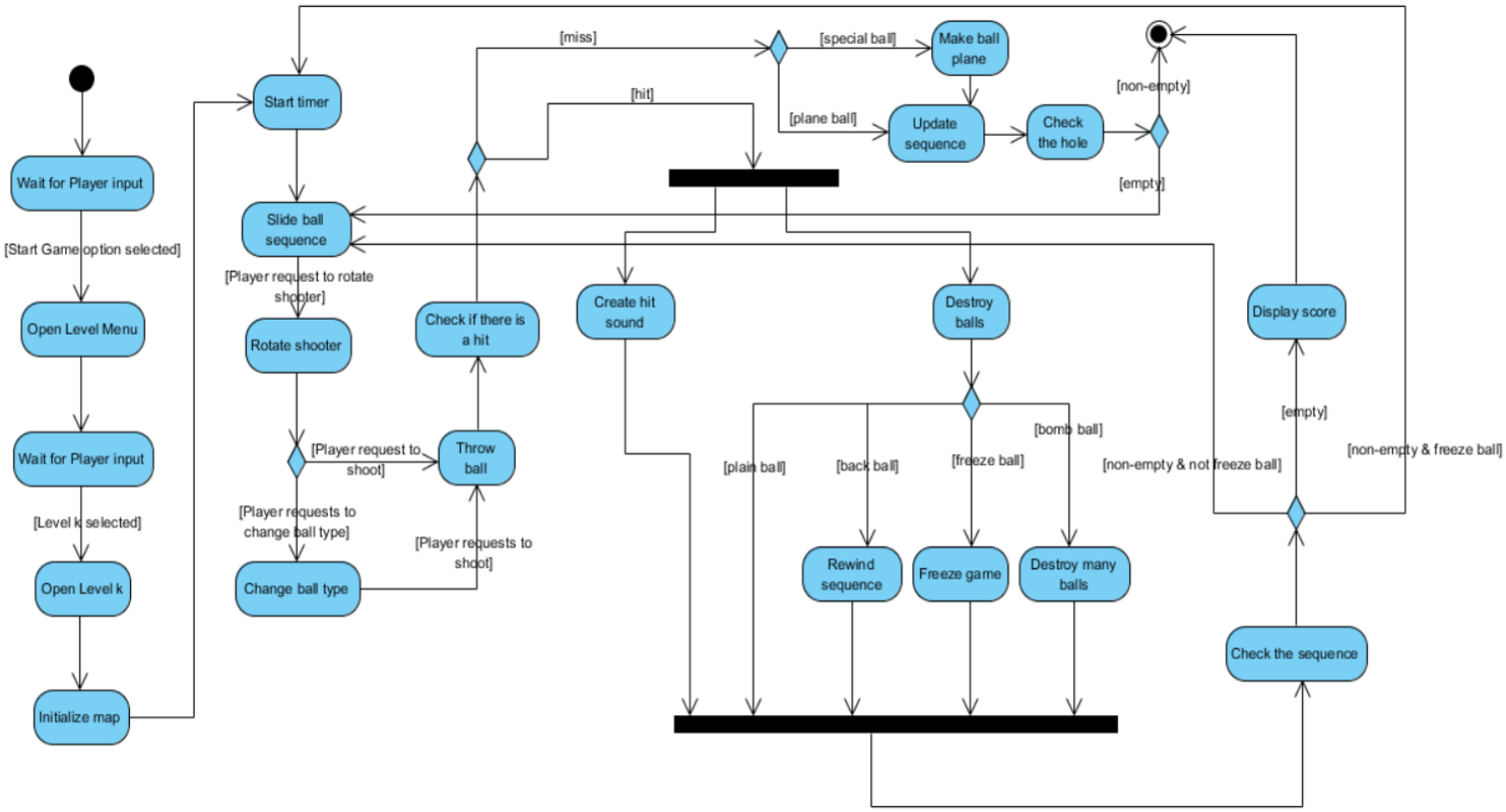


Figure 3.2.2 – Activity diagram of "Play Game"

At first, System opens a particular level which Player requests to play. Then, it initializes the level map (level screen). For triggering the level, it starts the timer and slides the ball sequence. Then, it rotates the shooter according to the player input. If Player desires to change

the ball type, System changes the ball type. System throws the ball according to the input. After that, it checks if there it is a hit or miss.

1. If it is a hit:

It concurrently creates a hit sound and updates the sequence according to the ball type. At first, it destroys the balls regularly. Then, if the ball is a back ball, it rewinds the sequence. If it is a bomb ball, it destroys more balls. If it is freeze ball, it freezes the sequence and timer. After that, it checks the sequence that if it is empty or not. If it is empty, that means the level is over, so it displays the score. If it is not empty, it loops the procedure again.

2. If it is a miss:

If the ball is a special ball, System makes the ball a plain ball. Then, after adding the ball to the sequence, it checks the hole that it contains any ball or not. If it contains any ball, that means Player loses. If it is empty, it loops the procedure again.

3.2.2 State Chart Diagram

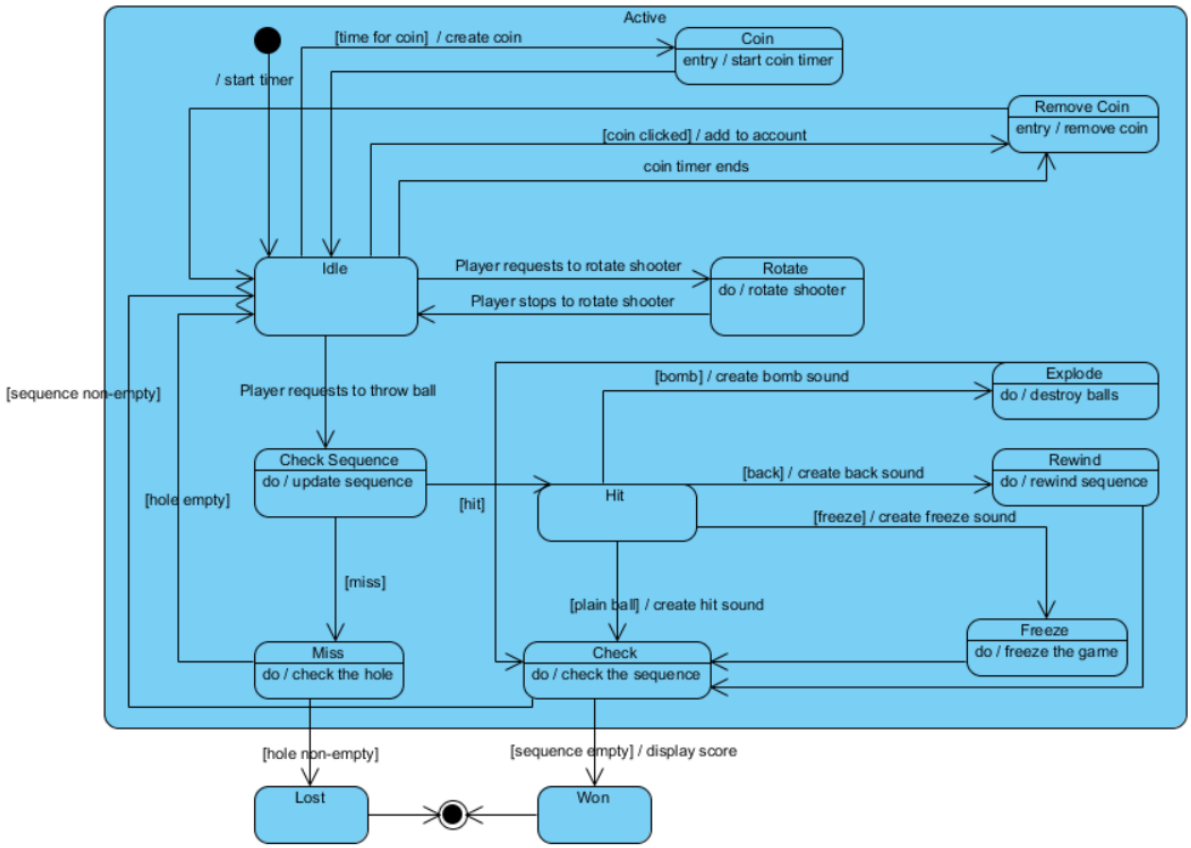


Figure 3.2.3 State Chart Diagram of LevelManager class

There are three main states: Active, Won, and Lost. System is in the Active state if the level continues. At first, it is in the Idle state. If Player requests to rotate the shooter, it goes to Rotate state and comes back when the process is done. If Player throws a ball, it goes to Check Sequence state and updates the sequence.

If there is a hit, it goes either directly to Check state for checking the sequence or firstly goes to a state belongs to one of the special balls. Also during the transition, it creates a hit sound depending on the type of ball. If the sequence is empty, it goes to Won state, if not it loops back.

If there is a miss, it checks the hole and if it contains a ball, it goes to Lost state; else, it loops back.

Also, from the Idle state, System goes to Coin state when there is time for creating a coin. As an entry condition, it starts the coin timer. From the Idle state, when Player clicks on the coin or coin timer ends, it goes RemoveCoin state and removes the coin from the screen. If there is a click; during the transition, it adds the coin to Player's account.

3.2.3 Sequence Diagrams

3.2.3.1 Create Account

The following sequence diagram corresponds to the "Create Account" scenario in the Scenarios part of the report.

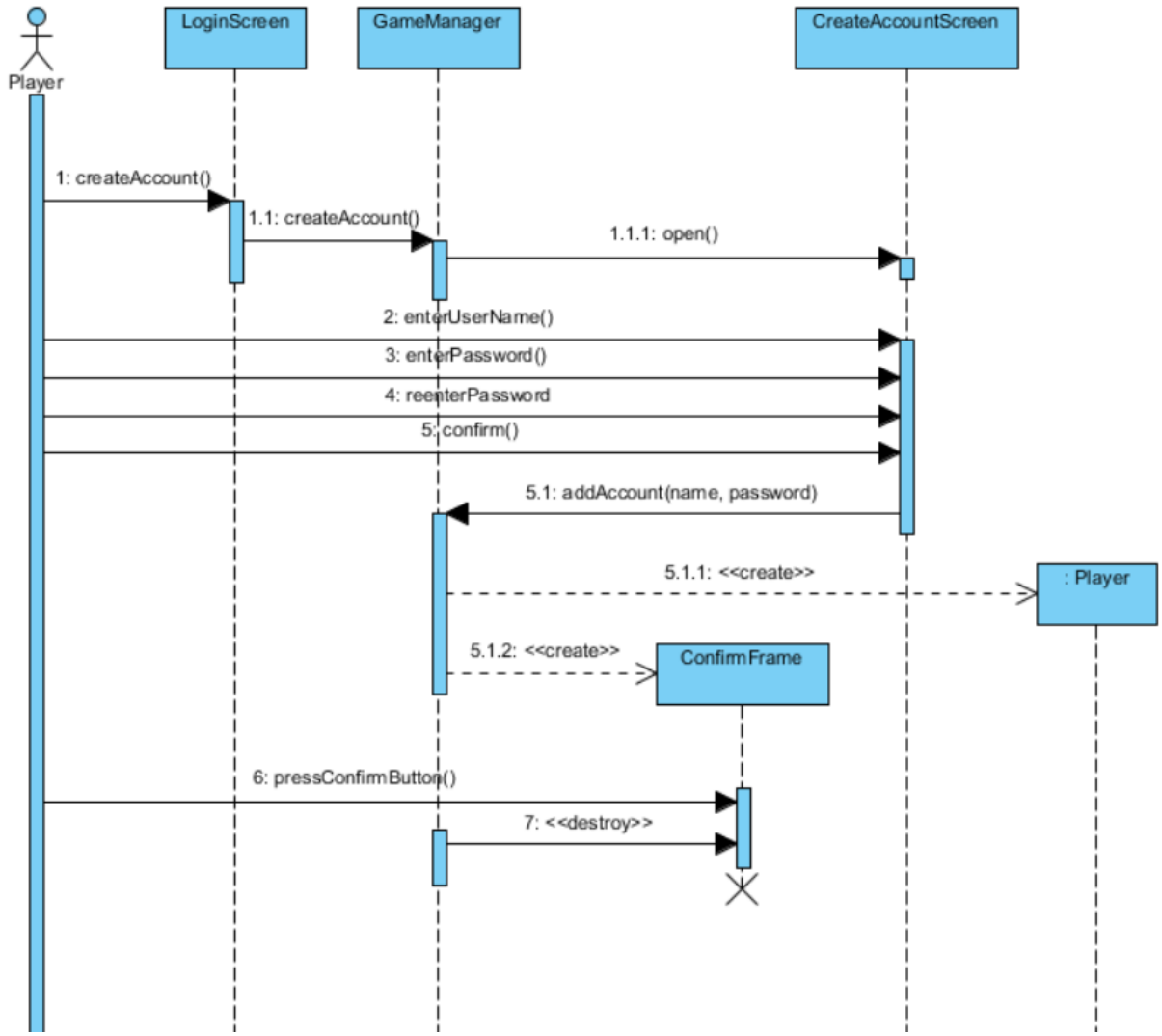


Figure 3.2.5 Sequence Diagram for "Create Account" scenario

Again, LoginScreen is the boundary class for user interaction. However, Player also interacts with CreateAccountScreen which is a boundary class too. As a control class, GameManager handles the transition between the Login screen and Create Account screen and creates a Player object for the new player. After that, GameManager creates a confirm frame which is destroyed after Player presses the confirm button.

3.2.3.2 Start Game

The following sequence diagram corresponds to the "Start Game" scenario in the Scenarios part of the report.

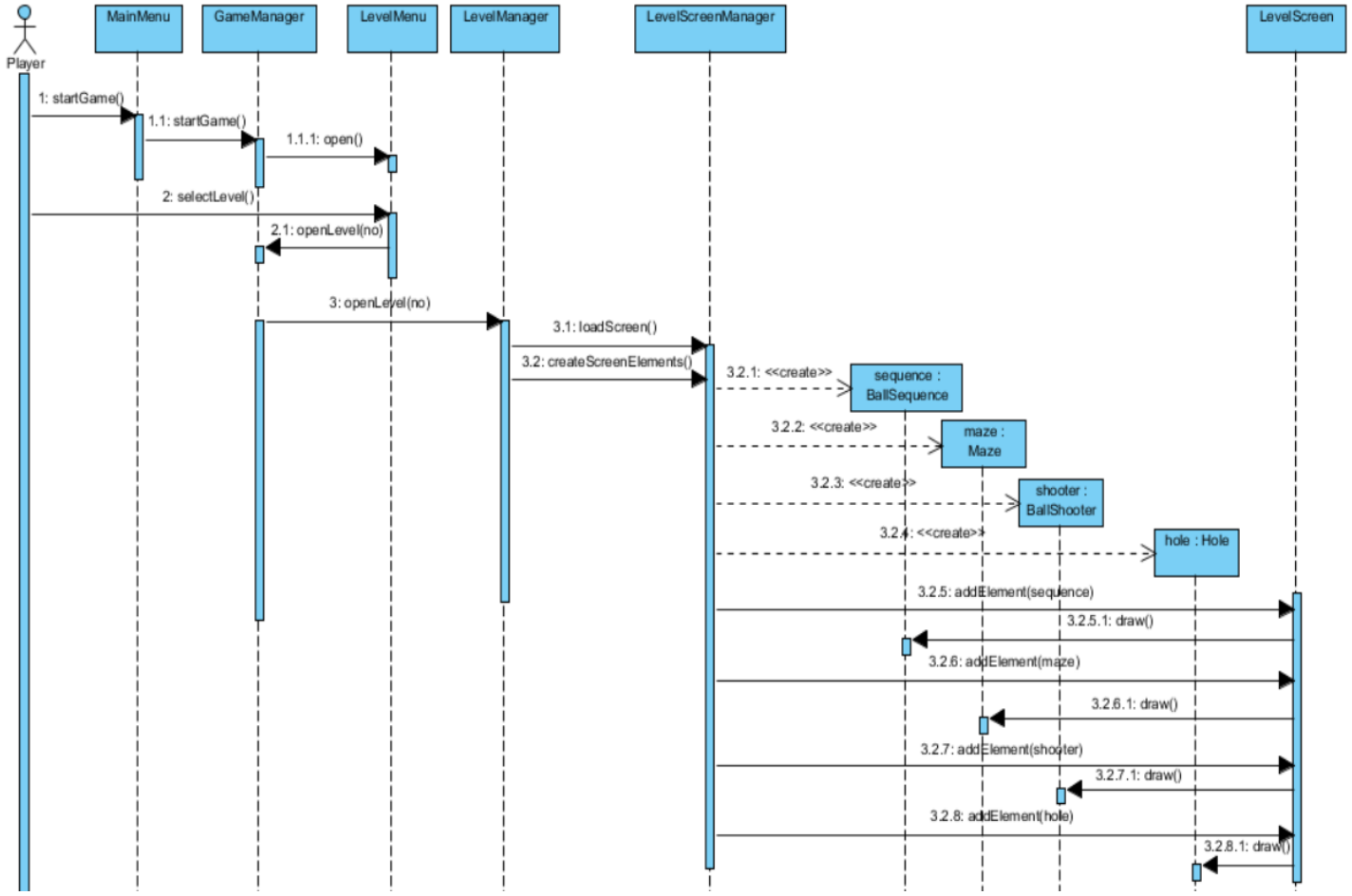


Figure 3.2.6 Sequence Diagram for "Start Game" scenario

In this scenario, MainMenu class serves as the boundary class from which Player chooses to start the game. From LevelMenu, Player selects a level to play and LevelManager opens the level by the control of GameManager. Then, LevelScreenManager initializes the LevelScreen by creating its BallSequence, Maze, and BallShooter. Then, LevelScreen class draws them to the screen. Here, LevelMenu and LevelScreen are boundary classes which Player can interact with. LevelManager and LevelScreenManager are control classes which controls the operations of boundary and entity classes. BallSequence, Maze and BallShooter are entity classes which are some basic elements of the game.

3.2.3.3 Play Game #1

The following sequence diagram corresponds to the "Play Game #1" scenario in the Scenarios part of the report.

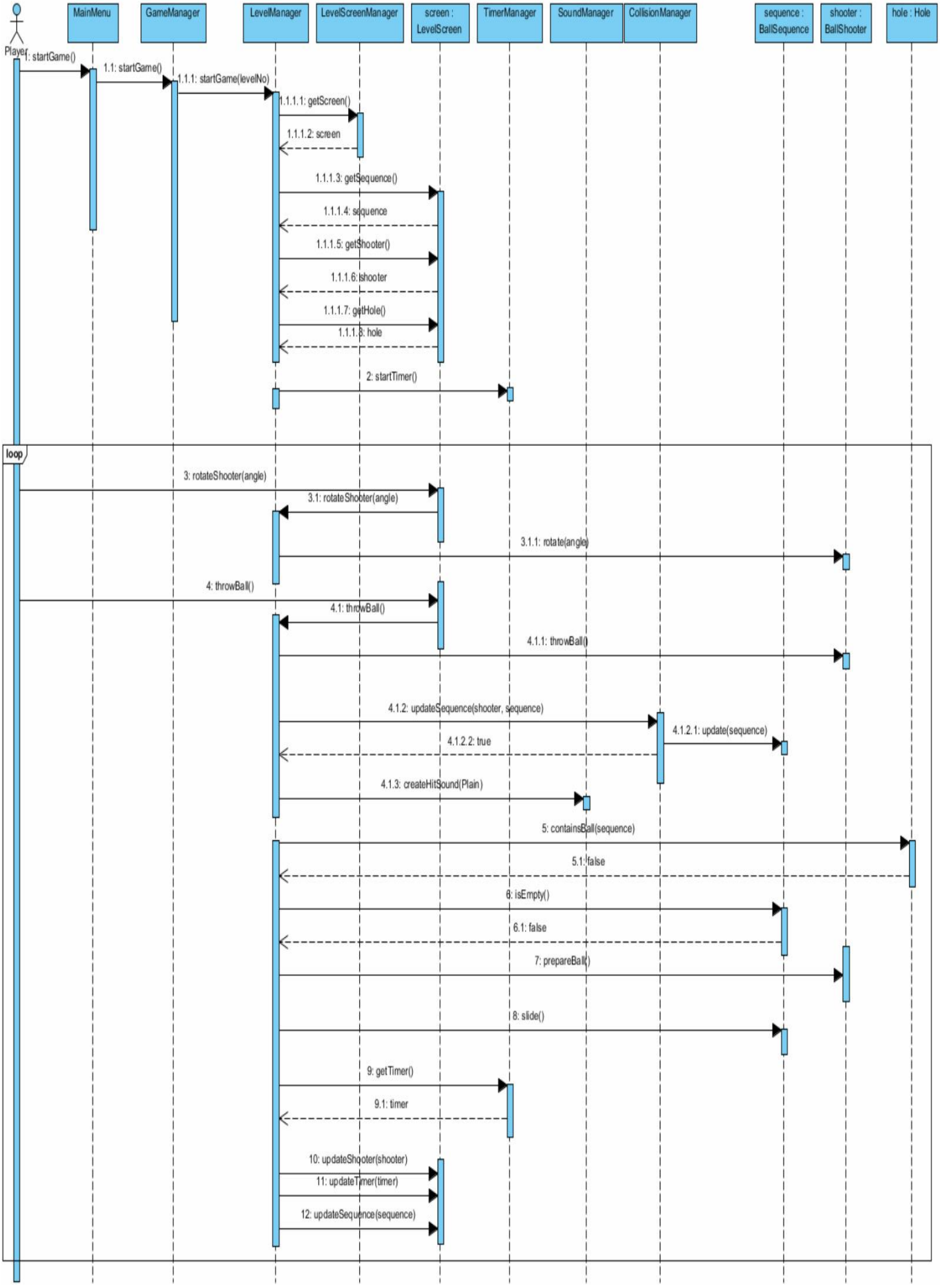
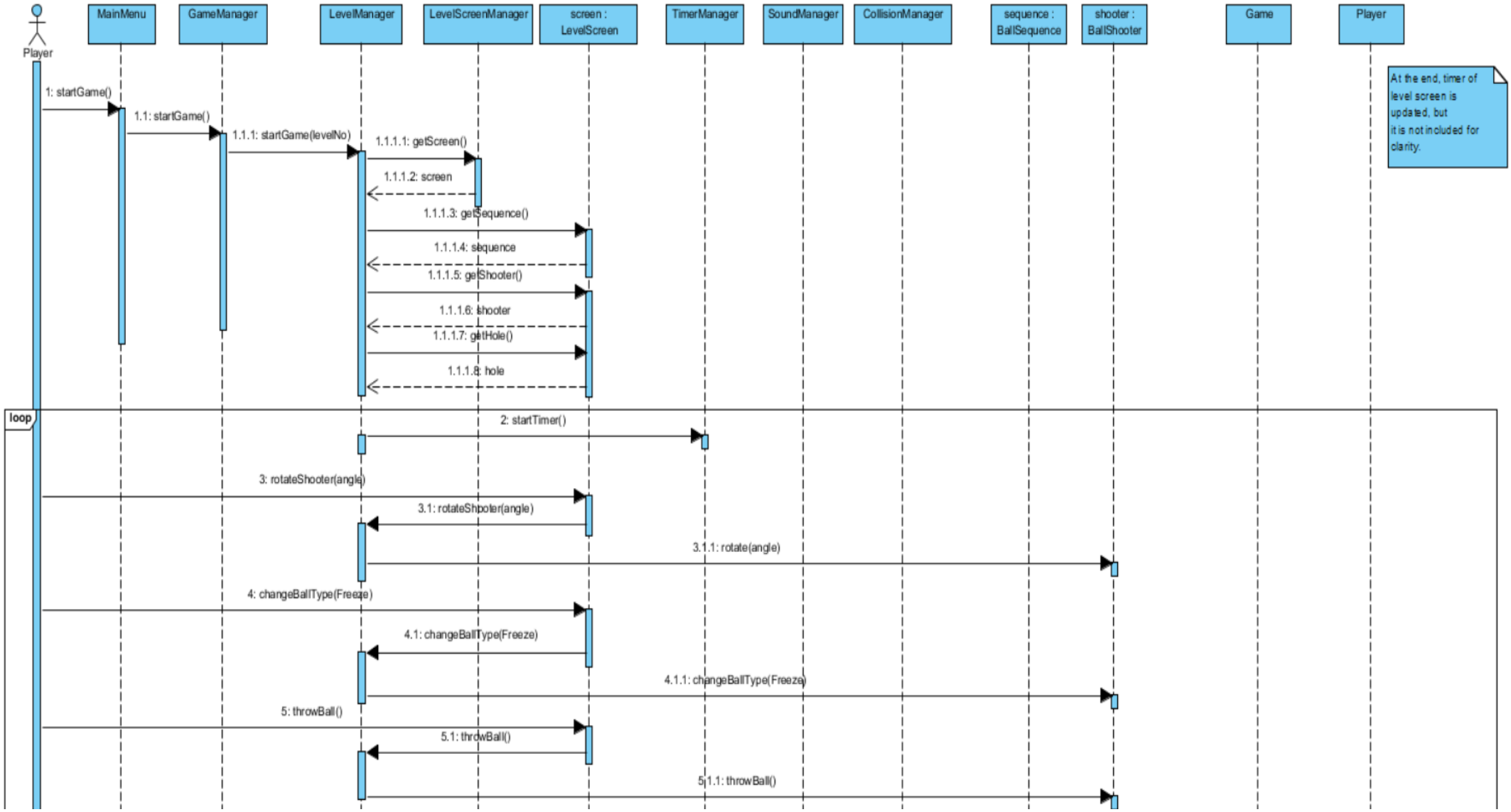


Figure 3.2.7 Sequence Diagram of "Play Game #1" scenario

After the previous “Start Game” scenario, LevelManager gets the screen elements from LevelScreen. LevelManager is the main control class for the level play. There are also three control classes associated with it: TimerManager, SoundManager, and CollisionManager. At first, TimerManager starts the timer of the level. Then, the game loop begins. Player requests to rotate shooter and throw ball by interacting with LevelScreen and LevelManager calls the methods of BallShooter for those. Then, CollisionManager updates the sequence accordingly, and if there is a hit, SoundManager creates a hit sound depending on the ball type. Then, LevelManager checks the Hole and BallSequence for determining if Player wins or loses. If the game continues, BallShooter prepares the new ball and LevelManager slides the sequence, and updates the screen elements accordingly.

3.2.3.4 Play Game #2

The following sequence diagram corresponds to the “Play Game #2” scenario in the Scenarios part of the report.


Player
MainMenu
GameManager
LevelManager
LevelScreenManager
screen : LevelScreen
TimerManager
SoundManager
CollisionManager
sequence : BallSequence
shooter : BallShooter
Game
Player
At the end, timer of level screen is updated, but it is not included for clarity.
1: startGame()
1.1: startGame()
1.1.1: startGame(levelNo)
1.1.1.1: getScreen()
1.1.1.2: screen
1.1.1.3: getSequence()
1.1.1.4: sequence
1.1.1.5: getShooter()
1.1.1.6: shooter
1.1.1.7: getHole()
1.1.1.8: hole
loop
2: startTimer()
3: rotateShooter(angle)
3.1: rotateShooter(angle)
3.1.1: rotate(angle)
4: changeBallType(Freeze)
4.1: changeBallType(Freeze)
4.1.1: changeBallType(Freeze)
5: throwBall()
5.1: throwBall()
5.1.1: throwBall()

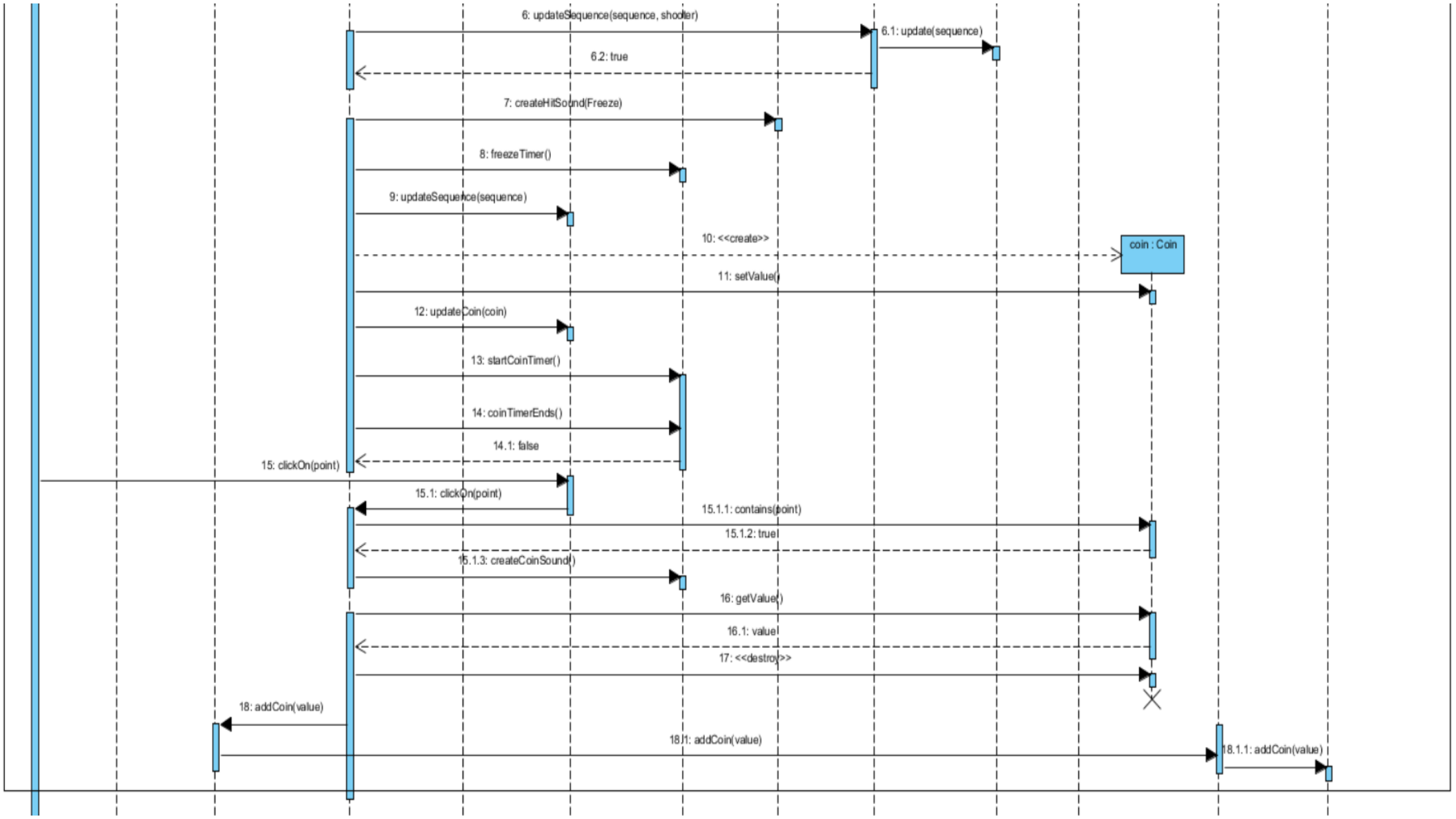


Figure 3.2.8 Sequence Diagram for "Play Game #2" Scenario

This scenario differentiates from the first one such that it is about bonus properties of the game. Since it will use freeze ball, it starts the timer inside the loop. Player requests the change the ball type from LoginScreen and LevelManager requests it from BallShooter. After updating the sequence, TimerManager freezes the time. Then, LevelManager creates a coin in LevelScreen and gives it a random value. Then, TimerManager starts the coin timer, and after a while LevelManager checks if it ends or not. Player clicks on a point, and LevelManager checks that if it belongs to the coin or not. Then, if it belongs to the coin, LevelManager gets the value of it and GameManager adds it to Player's account by using Game. Also, LevelManager destroys the coin when since it is done with it after getting its value.

4. Design

4.1 Design Goals

Identifying design goals is the basis of the system design. Before designing a system, the design goals of the system should be detected such that the design goals guide us to indicate how the system should be designed in order to function properly. Design goals can be inferred from the nonfunctional requirements. There are five different design criteria, which are end user, performance, dependability, cost and maintenance criteria.

4.1.1 Performance Criteria

Response time: Time for responding to user must be as fast as possible such that latency makes the game less enjoyable. In the animations, there are some concurrent events which should seem to happen at the same time. So, if the time interval between them is less than one second, they seem proper. Also, another response issue is about the storage system. Since there may be so many players, retrieving a particular player should be as fast as possible such as less than 1 second with a proper data structure.

4.1.2 Dependability Criteria

Security: Passwords of players are stored in a file system, so the file should be protected. For that reason, a simple encryption algorithm can be used for storing them which will be explained in the persistent data management part.

4.1.3 Maintenance Criteria

Extensibility: The game should be extensible such that developers should easily update it. For that purpose, the project should be separated into some parts which have separate functions such that updating the code will require only some of the classes to be changed. This requirement is important for the adaptability of the game for new application domain concepts.

Portability: Portability is an important criterion such that the system can have more users if it is portable among platforms. Since Java Virtual Machine is platform independent, portability of the system can be managed.

Modifiability: In the architectural pattern, subsystems should be separated from each other and layering should be used accordingly. So, dependency between classes with separate functionalities can be minimized and the system can be updated with some new functions in the future.

4.1.4 End User Criteria

Utility: The game can help people to enhance their talents because the game helps people to learn to use time efficiently and improves their reflexes, concentration, and hand-eye coordination. Also, it is a useful tool for entertainment.

Usability:

User-friendly interface: In addition to mouse inputs, there should be various shortcuts from keyboard for making the interface more user-friendly. Especially during the levels, players should not lose time for giving inputs such that time is the main criterion for scores.There should be shortcuts for entering information, changing ball type during a level, and stopping the level.

Ease of learning: Learning the game should be easy such that it can harm motivations of the players in the game. For this purpose, the system should offer a help option in order to inform player about the rules and logic of the game with clear instructions.

4.1.5 Trade-off Between Ease of Learning vs Functionality

Since one of the design goals of the project is ease of learning, there should not be too much functionality and required functionalities should be as simple as possible.Therefore, users can be able to learn all of them easily.

4.2 Sub-System Decomposition

This section covers the way of decomposing the system into almost independent parts in order to illustrate the organization of the software system and thus making it easier for any developer to understand how the system works. Subsystem decomposition is an important stepping stone on the way of developing a software system because it affects many features of the software system such as maintainability, modifiability, extensibility, efficiency and performance. Therefore, having a good subsystem decomposition increases the quality of the software system.

Figure 4.2.1 shows the simple subsystem decomposition of the system. System is divided into three subsystems namely User Interface, Game Control and Game Entity. Each of them performs different tasks of the system to ensure consistency. These subsystems are connected to each other in a way that any change made to one subsystem will not have a significant impact on other subsystems. Accordingly, this subsystem decomposition is extensible for adding new features or making any change to system. The methods and attributes of the classes will be included in the detailed diagrams.

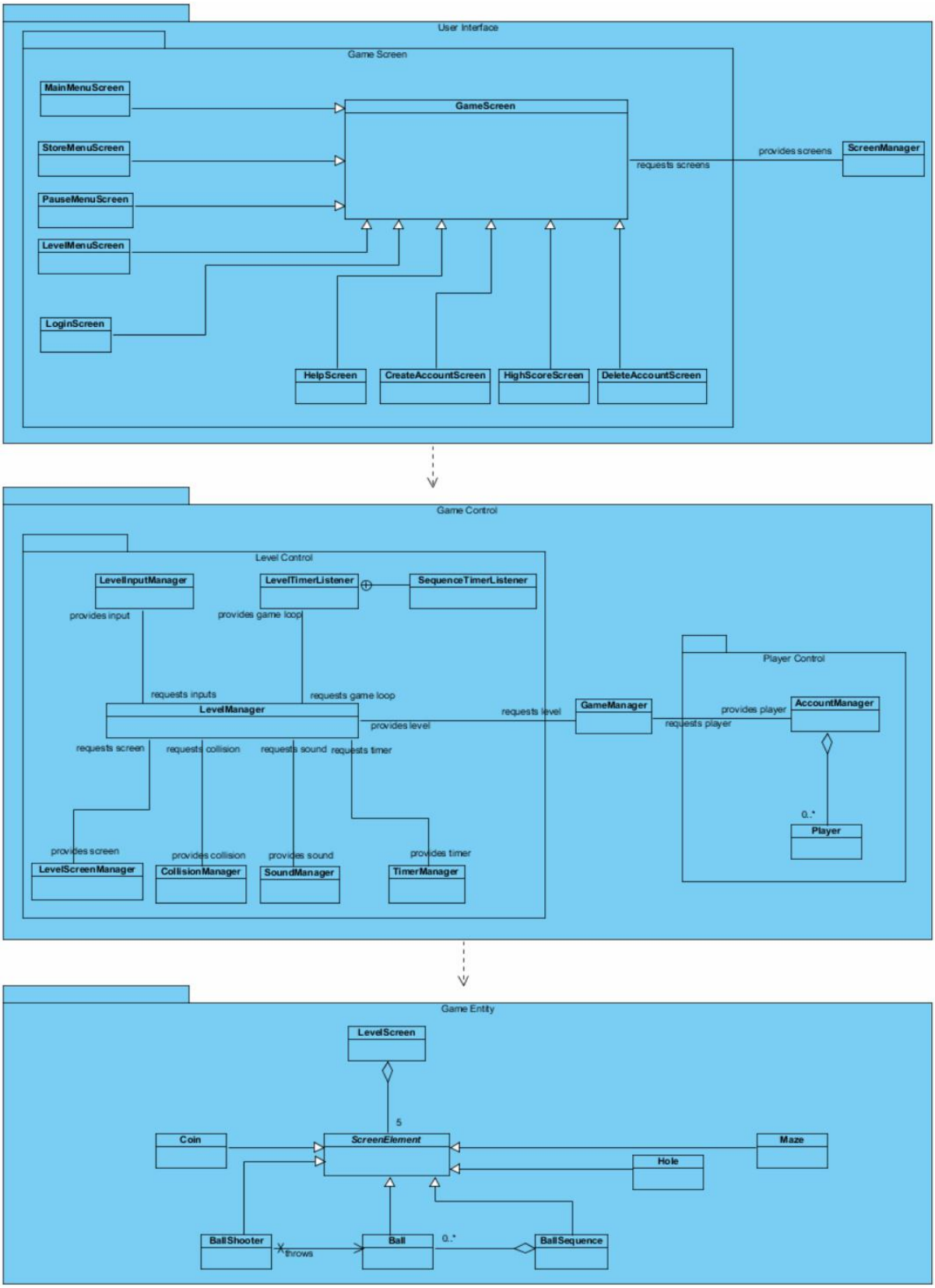


Figure 4.2.1 Package diagram

Figure 4.2.2 displays the relationships and dependencies between the subsystems and components in more detail. User Interface is connected to Game Control by the relation between Game Screen and Game Manager. Game Control is connected to Game Entity by the relation between Level Screen Manager and Level Screen. As it can be seen, there is only one coupling between each subsystem which is convenient for the minimal number of coupling rule between subsystems. In addition, the classes which perform similar tasks are gathered together in each subsystem in order to increase the number of cohesions. For example, manager classes in Game Control, entity classes in Game Entity and screen classes in User Interface are kept together. These strong associations between those classes enable them to send and request information to and from each other. In addition to these three main subsystems, there are three inner subsystems namely Game Screen, Level Control, and Player Control which provide services to other classes.

While decomposing the system into smaller subsystem, the rule of minimum number of couplings and maximum number of cohesions rule is taken into account to create a flexible and maintainable software. For that purpose, there can be transitions between the subsystems such as although Player is a model class, it is in the Game Control subsystem the reason of which is Player does not have a relationship with other model classes since it is not screen element and it has a direct relationship with AccountManager.

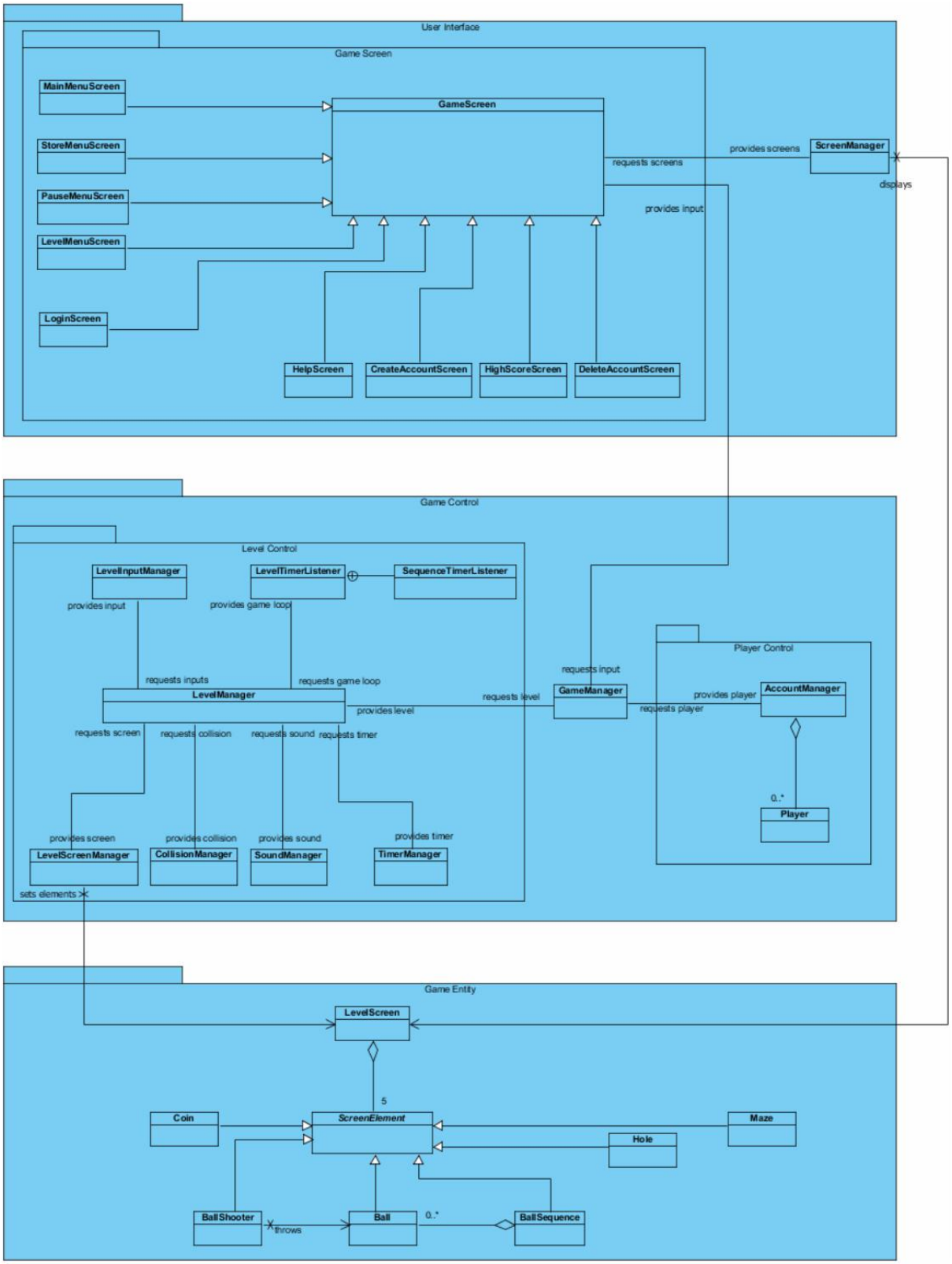


Figure 4.2.2 – Detailed Package Diagram

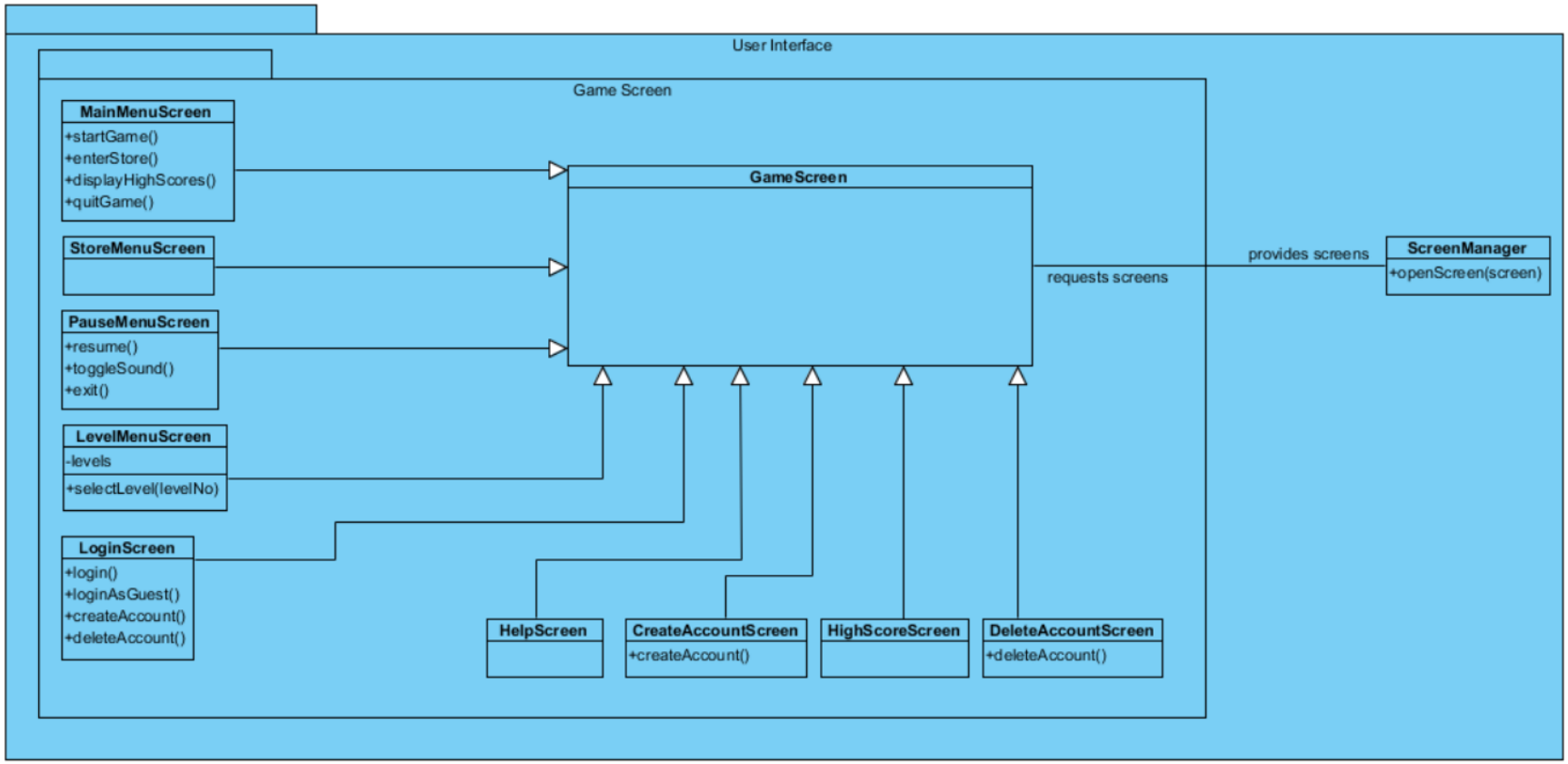


Figure 4.2.3- User Interface Subsystem

As seen in Figure 4.2.3, in the User Interface subsystem, there is another subsystem called Game Screen which contains the classes derived from GameScreenclass, which are the same classes with the analysis part. There is also ScreenManager class in the User Interface subsystem which is responsible from the transitions between the screens.

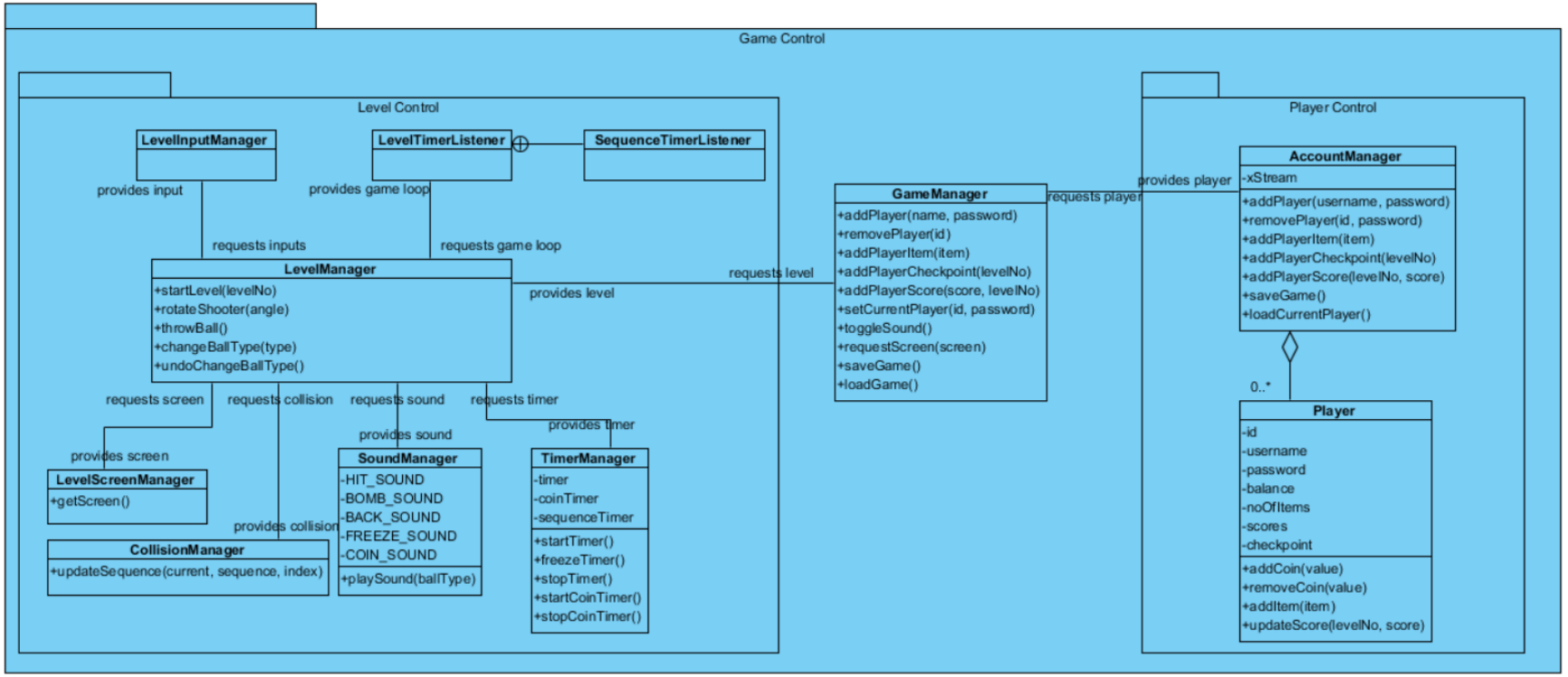


Figure 4.2.4 – Game Control Subsystem

There is a subsystem called Player Control subsystem inside the Game Control subsystem which is responsible for controlling the player operations. There is an additional solution domain class called AccountManager which holds the players qualified by unique ids. It has an instance of XStream class which is used for serializing player objects into an XML file which will be told in Persistent Data Management part. As well as the application domain classes, there are solution domain classes which are called LevelInputManager, LevelTimerListener and SequenceTimerListener. LevelInputManager'sduty is to take the inputs from players in a level. The others' duty is to provide the game loop. Listener of the sequence timer is separate because its speed depends on the level number.

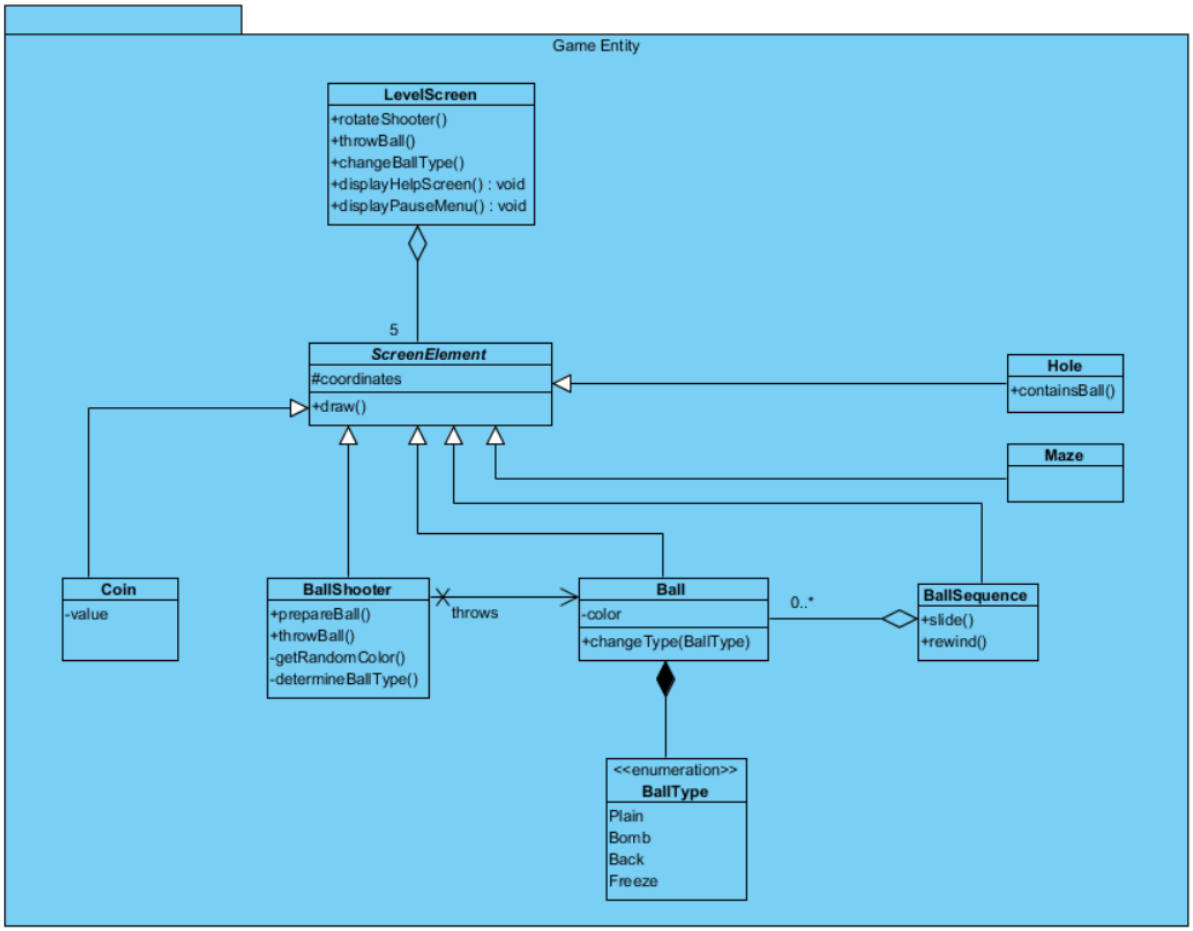


Figure 4.2.5 – Game Entity Subsystem

Game Entity subsystem includes application domain entity classes with LevelScreen which are the same with the analysis part.

4.3 Architectural Patterns

4.3.1 Layers and Partitions

The system is decomposed into three layers namely User Interface, Game Control and Game Entity. Each layer provides related services and they are structured hierarchically. The top layer is User Interface which is not used by other layers and thus has the highest hierarchy. Its main purpose is to interact with the user. The second layer is Game Control which is used by User Interface. Its main purpose is to control the flow of the game. The bottom layer is Game Entity which is used by Game Control. Its main purpose is to contain information about entity objects. This layering has open architecture which means User Interface layer can reach to Game Entity layer. However, this does not mean that interface objects can reach entity objects such that there is also an object called ScreenManager in the User Interface subsystem which should have direct association with LevelScreen class in the Game Entity subsystem. The layer decomposition is illustrated in Figure 4.3.1.

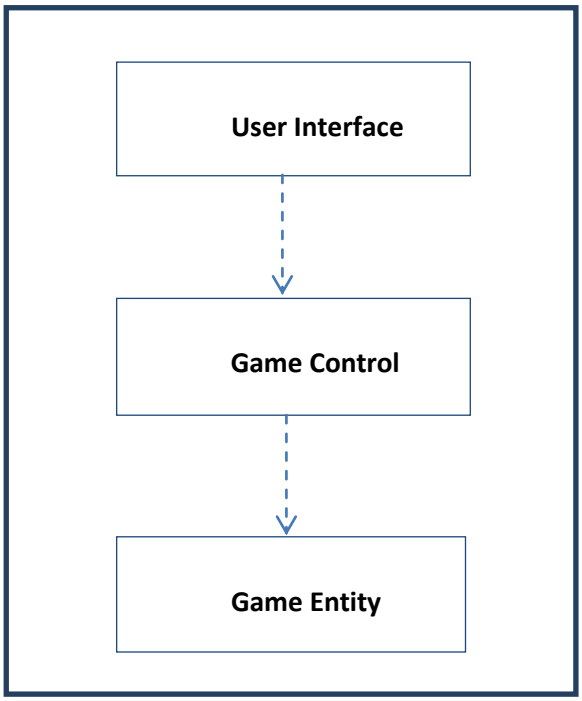


Figure 4.3.1 - Layer Decomposition

In User Interface layer, there are two partitions. One of them is Game Screen subsystem which consists of the screens of the systems. The other partition contains the ScreenManager class which handles the transitions between the screens.

Game Control layer consists of three partitions. One of them is Level Control subsystem in which there are classes responsible from the level events. Another partition is Player Control subsystem which handles the accounts of player. These two partitions provide services to the middle partition which contains the GameManager class.

4.3.2 Model-View-Controller (MVC) Architectural Style

In the system, MVC architecture style is used such that the system is divided into three layers which serve as the model, view and controller. By this style, model classes which are the application domain entities are abstracted from the view classes which are boundary classes responsible from user interaction. Controller layer is responsible from the connections of view and model and it maintains the control flow of the system.

Game Entity layer holds the application domain objects and thus represents the model part of the architecture. These domain objects are controlled and updated by Game Control layer which represents the controller part of the architecture. The interaction between user and system is accomplished by User Interface layer which represents the view part of the architecture.

As told before by the MVC style, user interface objects cannot reach the entity objects directly. The controller objects make the relationship between them. By this way, adding new view and model objects to the system become easier such that they are independent from each other.

MVC architectural style is useful for this software system because there are many views in it,such that there are many screens in the game,and they need to be handled accordingly. In MVC architectural style, it is easy to handle multiple views because any change made to views does not require a change in the models. Also, another benefit of this style is new properties can be added to models easily without impacting others much. This is also required for this system, because there are multiple screen elements and they have different properties. Also, as the functionalities increase by the new updates, properties of the players need to be changed accordingly.A disadvantage of this pattern is it slows down the program because of

the data transmissions between classes. However, extensibility and modifiability is better trade-offs for this case.

4.4 Hardware/Software Mapping

Java programming language will be used to implement the Crazy Shooter andJDK 8 will be used. For implementing the system, Eclipse Mars environment will be used as the IDE. In the game, there are various options and functionalities which are performed by mouse inputs. Also, there are some shortcuts which are activated by using keyboard buttons. Therefore, a keyboard and a mouse are needed as hardware configurations to take the inputs from the user. The software system is not a web-based application so it does not require internet connection. However, since it stores players' data in a XML file, the operating systems must support XML files.

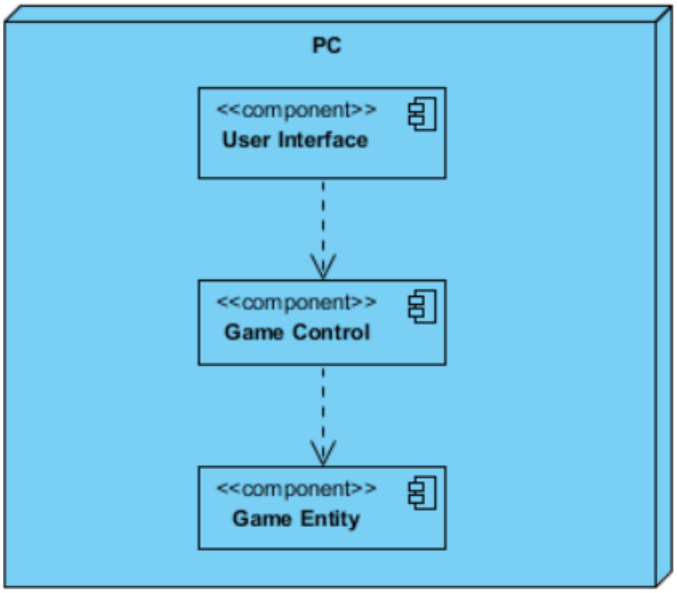


Figure 4.4.1 – Deployment Diagram of the System

Figure 4.4.1 represents the deployment diagram of the system. As told earlier, in the software part of the system, there are three subsystems which are User Interface, Game Control and Game Entity subsystems.Since there is not remote connection or server in the system, all of three software systems are allocated in the same node which is the local computer.

4.5 Addressing Key Concerns

4.5.1 Persistent Data Management

In the game, each player has an account in which his/her identity and progress are stored. Since data are used by multiple readers, but a single writer (There are multiple players, but there is one class which keeps all of them.), it is stored in a file system instead of a database.Java has a library called XStream which has a purpose for serializing the objects into a XML file and take them back. This library will be used serialize Player objects and take them back in initialization of the program. Also, game map will be created before the game starts, therefore the data of game map will be persistent during the level. Sound effects of collisions will be held in proper sound format like .aiff or .wav in hard disk drive. The image of the shooter will also be held in a proper format like .png or .gif.

4.5.2 Access Control and Security

There are two kinds of user profile in Crazy Shooter, guest and registered user. If user logs in as guest, system does not take the record of user and prevents him from selecting the desired level, purchasing item and seeing the high scores. If user logs in with an id and a password, system takes the record of him and enables him to use the additional functionalities. Since there is only one actor in the system, the authentication of it is controlled by a capability list. Before authentication with an id and password, the player can access only the functions of some of the user interface objects as shown in the following capability list. Since players can login as guest, they can reach to level screen, but they cannot reach main menu or the relatives. After they login to the system, they can use all of the user interface objects.

Actor/Object	LoginScreen	CreateAccountScreen	DeleteAccountScreen	LevelScreen	PauseMenuScreeen
Player	login() loginAsGuest() createAccount() deleteAccount()	createAccount(id, password)	deleteAccount(id, password)	rotateShooter() throwBall() changeBallType() displayHelpScreen() displayPauseMenu()	resume() toggleSound() exit()

In addition to these, since the data are stored in a file system, the file should be protected from external factors. For this purpose, the passwords will be encrypted by a very simple

encryption algorithm like a substitution cipher in which each letter will be replaced by another.

4.5.3 Global Software Control

Since this version of the system does not include multi-player option, there is no need for different threads for multiple actors. There is one thread which consists of a game loop which responds to external events (player inputs). This is an example of event-driven control. Procedure-driven is not appropriate for this system such that the sequence of inputs may vary according to the player demands. So, the main control of the levels is centralized in the LevelManager object such that the individual level controller objects are controlled by it. Also, GameManager is the main controller of the game such that level control, player control and screen transition control objects are centralized on it.

4.5.4 Boundary Conditions

4.5.4.1 Initialization

Crazy Shooter is a Desktop application and it has an executable .jar file, thus it does not require any installation.

4.5.4.2 Termination

Players can terminate the game by clicking the “Exit” button in the login screen or in the main menu. While playing the game, players can use the pause menu to return to the login screen or to main menu.

4.5.4.3 Exceptional Conditions

If for some reason collision sounds or shooter image cannot be loaded successfully to the game screen, game will still start without those resources. If the system stops functioning properly because of an unrealized error, all registered player information will be deleted, so this type condition should be avoided as much as possible.

5. Object Design

5.1 Pattern Applications

5.1.1 Façade Design Pattern

For making the boundaries of subsystems clear, the number of dependencies between them which is defined as coupling should be minimal. Façade design pattern provides this minimization by defining a simple subsystem interface which may be just one class. So, the remaining classes are encapsulated in the subsystem.

In the system of the project, Façade pattern is used between the three subsystems which are called User Interface, Game Control and Game Entity. GameManager class is the main controller class of the game which interacts with other manager classes such as AccountManager and LevelManager. It is the main source of handling the requests, which sends them to individual controller classes according to their duties. So, it represents the interface of the Game Control subsystem. The remaining classes of the Game Control subsystem are abstracted from the User Interface subsystem.GameManager has an association with the GameScreen class in the User Interface subsystem which is the generalization of the screens. It sends requests to GameManager which are taken from the players.

In the Game Entity subsystem, there is a class called LevelScreen which keeps all of the screen elements, so it provides the interface of the Game Entity subsystem. It has associations with LevelScreenManager class from Game Control subsystem and ScreenManager class from User Interface subsystem. LevelScreenManager is responsible for adjusting level screen and setting its elements.ScreenManager is the class which handles the screen transition requests, so it needs an association with LevelScreen in addition to GameScreen.

Also, there are three inner subsystems which provide services by using one class. Player Control subsystem has a class called AccountManager which GameManager interacts with and it provides the accounts of the players. Also, LevelManager class in the Level Control subsystem has an association with GameManager for taking and providing requests related to the levels. The other subsystem is the Game Screen subsystem in the User Interface package which has a GameScreen class which is the generalization of individual screens. By having

generalization, it can provide all operations of the particular subsystem to ScreenManager class which handles the transition between screens.

5.1.2 Observer Design Pattern

Observer design pattern is used for updating multiple views when a change occurs in an object. In the game, the scores of the players are kept in two places depending on their values: LevelMenu and HighScoreScreen. In LevelMenu, the best score of the player in a particular level can be seen for each level. In HighScoreScreen, the overall score of the player can be seen. So, when the score of the Player object changes, the two views must have been updated. For this purpose, Player class extends from the Observable class and LevelMenu and HighScoreScreen classes implements the Observer interface.Also, balance of the player can be seen in two places: StoreMenuScreen and LevelScreen. So, they are also observes of the Player objects. Player class is responsible for updating its observers when the score and balance of the player changes. Since the score and balance of the player can change frequently, Observer pattern is useful for this purpose.

5.1.3 Adapter Design Pattern

Adapter design pattern is used for adapting a legacy class for the use of client. In this case, the client interface and legacy classes are given and cannot be modified. In the project, it is assumed that AccountManager and Player classes cannot be modified and an adapter class called AccountManagerAdapter is used between them for encrypting and decrypting the passwords of the players. In the Player class, passwords are kept as encrypted for serializing the Player objects into an XML file. However, the system needs the plain texts such that it uses to compare them with player inputs. So, when getting players from AccountManager, the adapter class decrypts the passwords and adapts them to the system. Also, when the Player's setPassword method is called, it encrypts the password accordingly.Figure 5.1.1 represents the diagram of the adapter pattern.

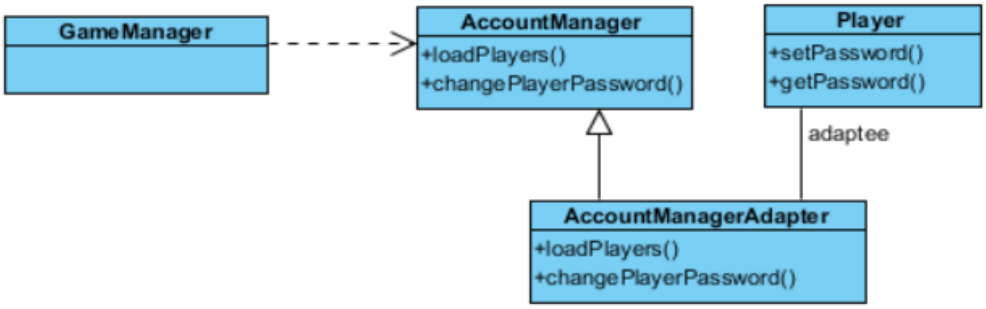


Figure 5.1.1 - Adapter Design Pattern

5.2 Class Interfaces

5.2.1 User Interface Subsystem Interface

User Interface subsystem contains the boundary objects which display the interface of the game to the players. In addition to the boundary objects, there is a control object called ScreenManager which handles the transitions between screens. GameScreen class which is the generalization of all screens interact with Game Control subsystem and ScreenManager class interacts with the Game Entity subsystem with an association with LevelScreen class.Figure 5.2.1 represents the detailed class diagram of the User Interface subsystem.

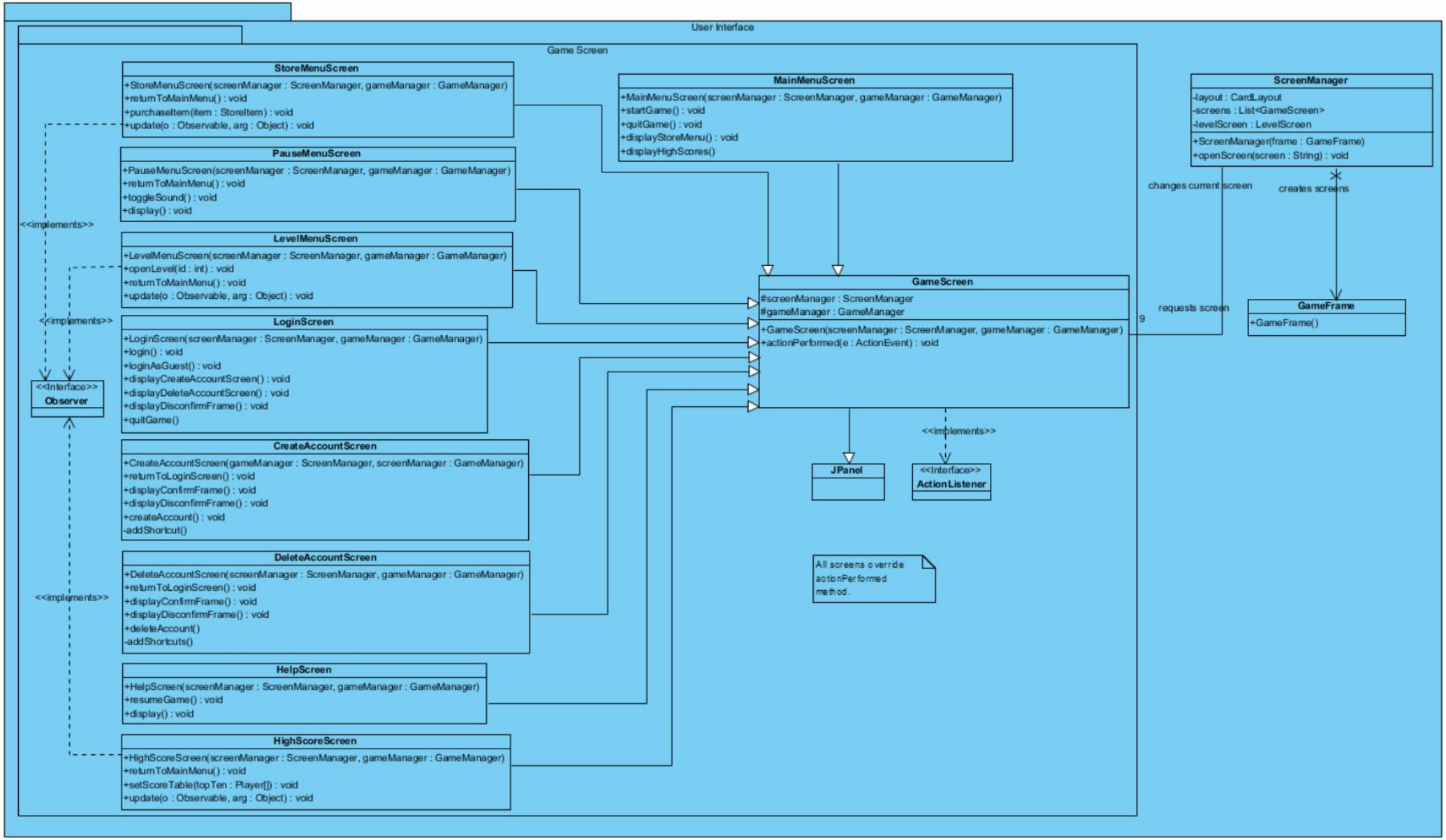


Figure 5.2.1 User Interface Subsystem

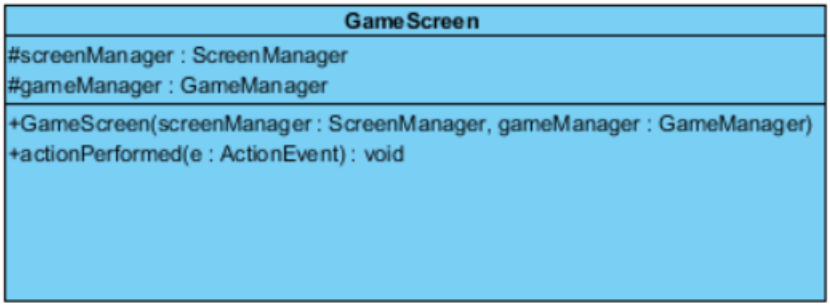
GameFrame Class

Base class: JFrame

Constructor:

- **publicGameFrame():** Instantiates the main frame of the game.

GameScreen Class



Base class: JPanel

Interface: ActionListener

Attributes:

- **protectedGameManagergameManager:** The main controller object of the game which interacts with User Interface subsystem by the GameScreen class.
- **protectedScreenManagerscreenManager:** The controller class which is responsible for the transitions between the screens.

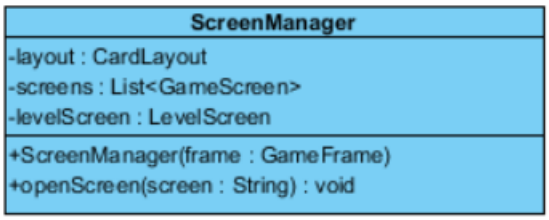
Constructor:

- **publicGameScreen(int width, int height):** Initializes the panel's size and width since they are common for all screens.

Operations:

- **public void actionPerformed(ActionEvent e):** Taken from the ActionListener interface and every screen class extending from GameScreen should overwrite it.

ScreenManager Class



Attributes:

- **privateCardLayout layout:** Main layout of the game which handles the transitions between screens.
- **private List<GameScreen> screens:** A list which contains the screens of the game. Its type is List such that the decision of the data structure can be changed later.
- **private LevelScreen levelScreen:** Level screen is also an attribute of the class.

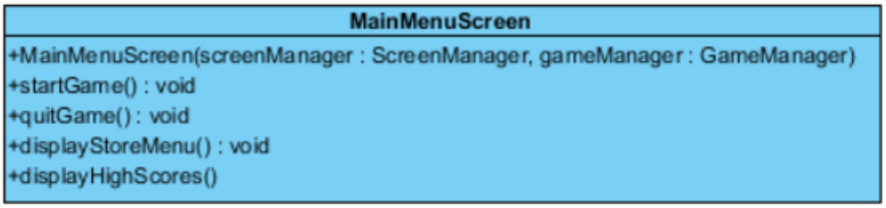
Constructor:

- **publicScreenManager(GameFrame frame):** Instantiates the frame of the game, its panels and the layout.

Operations:

- **public void openScreen(String screen):** Opens a particular screen by using the card layout according to a request from GameScreen classes. Its argument is a key value of the particular screen in the card layout.

MainMenuScreen Class



Base class: GameScreen

Constructor:

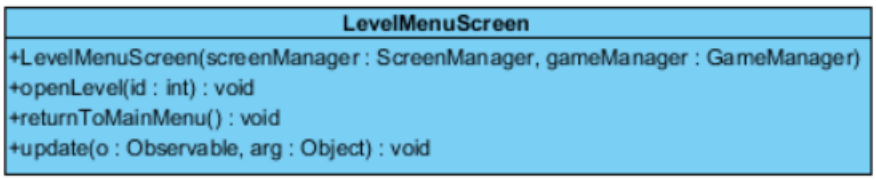
- **MainMenuScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void startGame():** Opens the level menu or a level if the player is a guest by using ScreenManager.
- **public void quitGame():** Quits the game.

- **public void displayStoreMenu():** Displays the store menu by invoking the method of ScreenManager.
- **public void displayHighScores():** Displays the high score screen by invoking the method of ScreenManager.

LevelMenuScreen Class



Base class: GameScreen

Interface: Observer

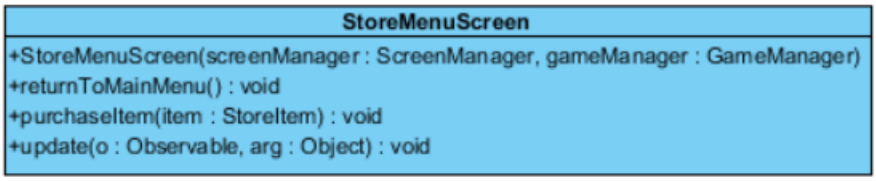
Constructor:

- **public LevelMenuScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void openLevel(int id):** Opens a particular level by ScreenManager.
- **public void returnToMainMenu():** Returns to main menu by ScreenManager.
- **public void update(Observable o, Object arg):** Updates the view of the panel when score of a player for a particular level changes.

StoreMenuScreen Class



Base class: GameScreen

Interface: Observer

Constructor:

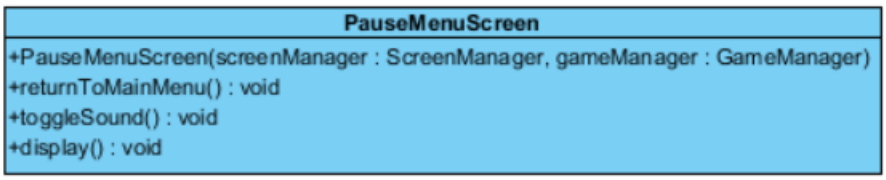
- **public StoreMenuScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void returnToMainMenu():** Returns to main menu by ScreenManager.
- **public void purchaseItem(Item item):** Adds the item to player's account by requesting it from the GameManager class.

- **public void update(Observable o, Object arg):** Updates the player balance when it is changed.

PauseMenuScreen Class



Base class: GameScreen

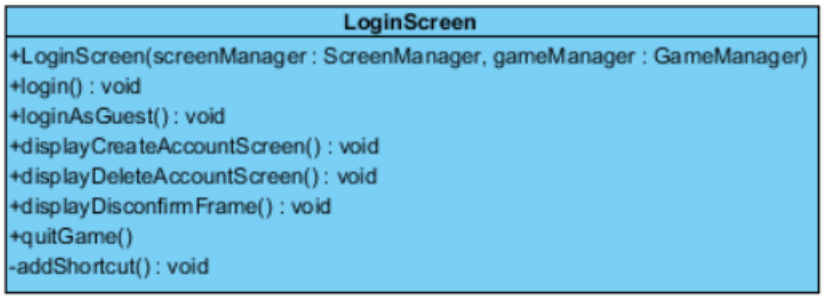
Constructor:

- **public PauseMenuScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void returnToMainMenu():** Returns to main menu by ScreenManager.
- **public void toggleSound():** Toggles the sound of the game by requesting it from GameManager.
- **public void display():** Requests from ScreenManager to display itself.

LoginScreen Class



Base class: GameScreen

Constructor:

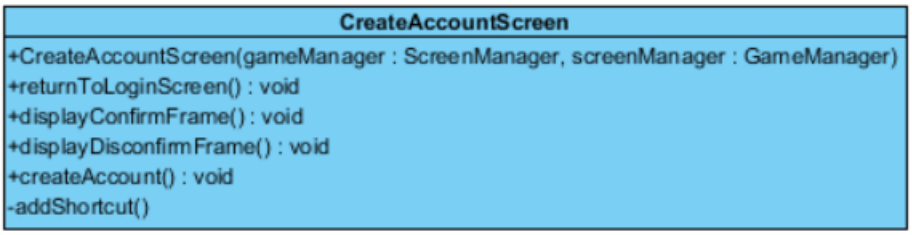
- **public LoginScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void login():** Sends login request to GameManager.
- **public void loginAsGuest():** Sends request to open the first level to GameManager.
- **public void displayCreateAccountScreen():** Sends request to display create account screen to ScreenManager.

- **public void displayDeleteAccountScreen():** Sends request to display delete account screen to ScreenManager.
- **public void displayDisconfirmFrame():** Displays a disconfirm frame if the account does not exist.
- **private void addShortcuts():** Adds the particular shortcuts to the screen.

CreateAccountScreen Class



Base class: GameScreen

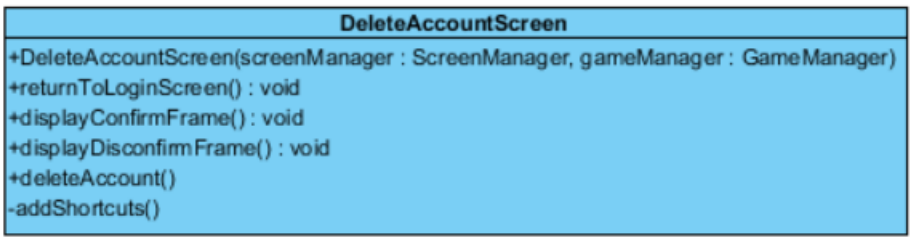
Constructor:

- **public CreateAccountScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void returnToLoginScreen():** Sends request to screen manager to return to login screen.
- **public void displayConfirmFrame():** Displays confirm frame if the account is successfully created.
- **public void displayDisconfirmFrame():** Displays disconfirm frame if the account is not successfully created.
- **public void createAccount():** Sends create account request to GameManager.
- **private void addShortcuts():** Adds the particular shortcuts to the screen.

DeleteAccountScreen Class



Base class: GameScreen

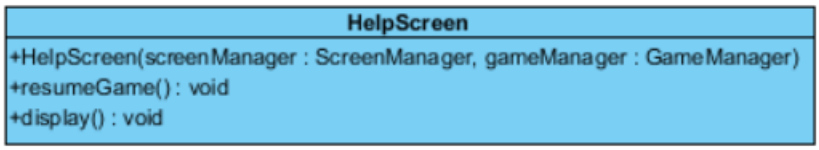
Constructor:

- **public DeleteAccountScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void returnToLoginScreen():** Sends request to screen manager to return to login screen.
- **public void displayConfirmFrame():** Displays confirm frame if the account is successfully deleted.
- **public void displayDisconfirmFrame():** Displays disconfirm frame if the account is not successfully deleted.
- **public void deleteAccount():** Sends delete account request to GameFrame.
- **private void addShortcuts():** Adds the particular shortcuts to the screen.

HelpScreen Class



Base class: GameScreen

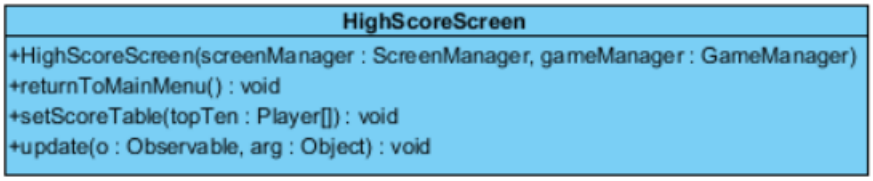
Constructor:

- **public HelpScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void resumeGame():** Sends requests to game manager to resume the game.
- **public void display():** Requests from ScreenManager to display itself.

HighScoreScreen Class



Base class: GameScreen

Interface: Observer

Constructor:

- **public HighScoreScreen(ScreenManagerscreenManager, GameManagergameManager)**

Operations:

- **public void returnToMainMenu():** Sends request to screen manager to return to main menu.

- **public void setScoreTable(Player[] topTen):** Sets the score table according to the top ten players.
- **public void update(Observable o, Object arg):** Updates the view of the panel when score of a player for a particular level changes.

5.2.2 Game Control Subsystem Interface

Game Control subsystem contains the controller classes of the game and an entity class which represents the players of the game. Its subsystem interface is represented by the GameManager class which is the main controller class of the game.GameManager interacts with User Interface subsystem and LevelScreenManager class which is responsible for adjusting the level screen interacts with GameEntity subsystem.Figure 5.2.2 represents the detailed class diagram of the Game Control subsystem.

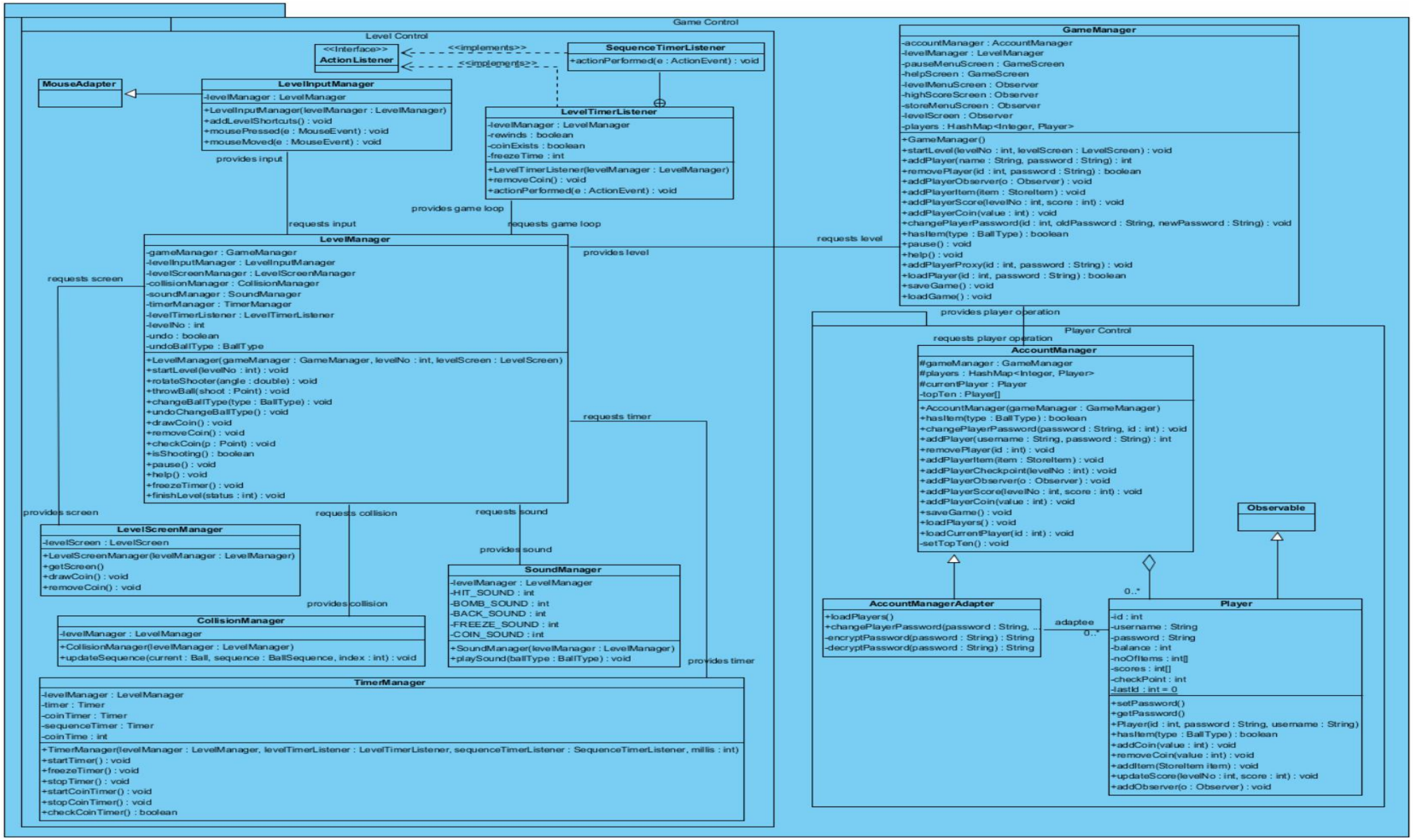
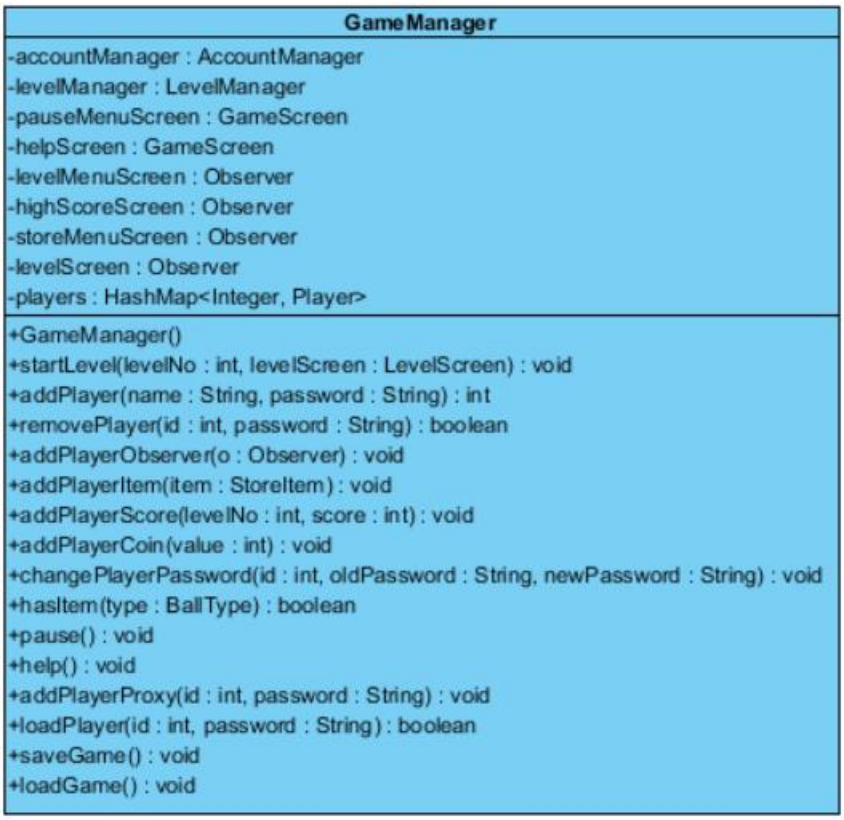


Figure 5.2.2 - Game Control Subsystem

GameManager Class



Attributes:

- **privateAccountManageraccountManager:** The object which holds the players and controls the operations of them.
- **private LevelManagerlevelManager:** The object which controls the operations in a level.
- **privateGameScreenpauseMenuScreen:** The class includes pause menu screen for handling the pause request coming from LevelManager.
- **privateGameScreenhelpScreen:** The class includes help screen for handling the pause request coming from LevelManager.
- **private Observer levelMenuScreen:**Level menu screen which is an observer for Player for including the scores.
- **private Observer highScoreScreen:**High score screen which is an observer for Player for including the scores.
- **private Observer storeMenuScreen:** Store menu screen which is an observer for Player for including the balance.
- **private Observer levelScreen:** Level screen which is an observer for Player for including the balance.
- **privateHashMap<Integer, String> players:** A hash map which matches player ids with passwords for dynamic access control. It represents a kind of protection proxy for Player objects.

Constructor:

- **publicGameManager():** Instantiates all controllers and screens.

Operations:

- **public void startLevel(intlevelNo, LevelScreen levelScreen):** Starts the particular level by creating the LevelManager object.
- **public void addPlayer(String name, String password):** Sends request to account manager to add a new player.
- **public void removePlayer(int id, String password):** Sends request to account manager to remove a player after checking access from proxy.
- **public void addPlayerItem(StoreItem item):** Sends request to account manager to add a new item to the player account.
- **public void addPlayerCheckpoint(intlevelNo):** Sends request to account manager to add a new checkpoint to the player account.
- **public void addPlayerScore(int score, intlevelNo):** Sends request to account manager to add a new score to the player account.
- **public void setCurrentPlayer(int id, String password):** Sends request to account manager to set the current player after checking access from proxy.
- **public void addPlayerObserver(Observer o):** Sends request to account manager to add an observer for player.
- **public void toggleSound():** Sends request to level manager to toggle the sound of the game.
- **public void requestScreen(GameScreen screen):** Sends request to User Interface subsystem to open a particular screen.
- **public void saveGame():** Sends request to account manager to save the player account.
- **public void loadGame():** Sends request to account manager to load the player accounts.
- **publicbooleanhasItem(Ball.BallType ball):** Sends request to account manager to check if the current player has a particular ball type or not.
- **public void pause():** Sends display request to pause menu screen.
- **public void help():** Sends display request to help screen.
- **public void changePlayerPassword(int id, String oldPassword, String newPassword):** Sends request to account manager to change the password of a player after checking access from proxy.

AccountManager Class

Attributes:

- **privateGameManagergameManager:** The main controller object of the game which all controller classes are centralized.
- **privateHashMap<Integer, Player> players:** List of players in the game matched with their ids.
- **private Player currentPlayer:** Current player of the game.
- **private Player[] topTen:** Players with top ten scores.

Constructor:

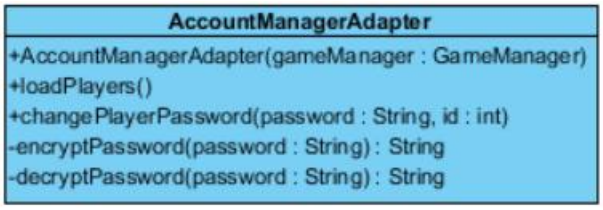
- **public AccountManager(GameManagergameManager)**

Operations:

- **public void addPlayer(String username, String password):** Adds a new player with the given username and password and the last id.
- **public void removePlayer(int id, String password):** Removes a player with the given id and password.
- **public void addPlayerItem(Item item):** Adds an item to the current player.
- **public void addPlayerCheckpoint(intlevelNo):** Adds a checkpoint to the current player.
- **public void addPlayerScore(int score, intlevelNo):** Adds a score to the current player for a particular level.
- **public void addPlayerObserver(Observer o):** Adds an observer to the current player.
- **publicbooleanhasItem(Ball.BallType type):** Checks if the current player has a particular ball type or not.

- **public void changePlayerPassword(String password, int id):** Changes the password of a player with particular id.
- **public void saveGame():** Saves the game by serializing player objects.
- **public void loadCurrentPlayer(int id):** Sets the current player as the player with a particular id.
- **public void loadPlayers():** Loads the player objects from the XML file.

AccountManagerAdapter Class



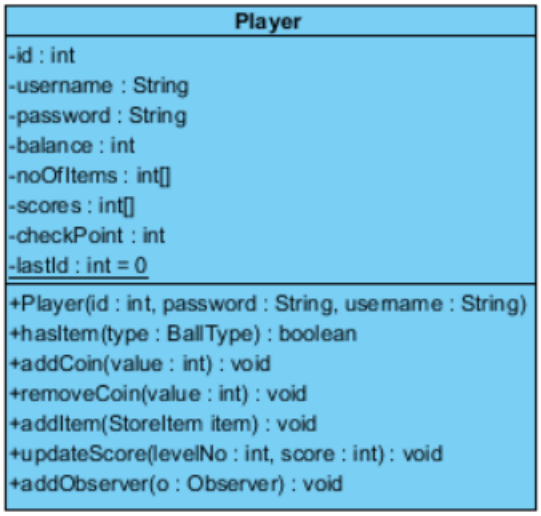
Constructor:

- **public AccountManagerAdapter(GameManagergameManager)**

Operations:

- **public void changePlayerPassword(String password, int id):** Changes the password of a player with particular id.
- **public void loadPlayers():** Loads the player objects from the XML file.
- **privateString encryptPassword(String password):** Encrypts the player password.
- **private String decryptPassword(String password):** Decrypts the player password.

Player Class



Base class: Observable

Attributes:

- **privateAccountManageraccountManager:** The controller class which is responsible for player operations.
- **private int id**
- **private String username**
- **private String password**
- **private int balance:** Number of coins which can be used for purchasing item from the store.
- **privateint[]noOfItems:**An array which keeps the amount for each item which are special balls and lives.
- **privateint[] scores:** An array which keeps the score for each level.
- **private int checkpoint:** The level number which the player has a checkpoint.
- **private static intlastId = 0:** Id of the last added player.

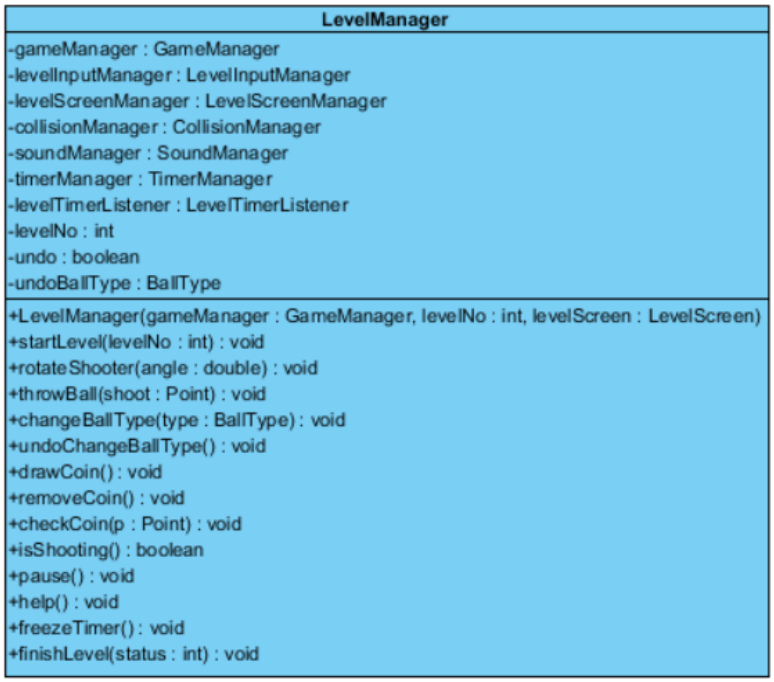
Constructor:

- **public Player(int id, String username, String password)**

Operations:

- **public void addCoin(int value):** Adds coin to the balance.
- **public void removeCoin(int value):** Removes coin from the balance.
- **public void addItem(Item item):** Adds an item to the item list.
- **public void updateScore(intlevelNo, int score):** Updates the score for a particular level.
- **public void addObserver(Observer o):** Add an observer for updates in the score.

LevelManager Class



Attributes:

- **privateGameManagergameManager:** The main controller object of the game which LevelManager class sends screen requests such as pause and help screens.
- **privateLevelScreenManagerlevelScreenManager:** The controller object which instantiates the screen elements and returns them to LevelManager.
- **privateLevelInputManagerlevelInputManager:** The object which listens to user inputs during the level.
- **privateSoundManagersoundManager:** The object which is responsible for the sound effects of the level.
- **privateTimerManagertimerManager:** The object which is responsible for controlling the timer of the level.
- **privateCollisionManagercollisionManager:** The object which is responsible for controlling the collisions between the ball and ball sequence in the level.
- **privateLevelTimerListenerlevelTimerListener:** The main timer of the level which contains the game loop.
- **privateint levelNo:** Current level number.
- **privateboolean undo:** Determines if there is an undoable operation or not which must be a change ball type operation.
- **privateBall.BallTypeundoBallType:** Keeps the previous ball type for undo operation.

Constructor:

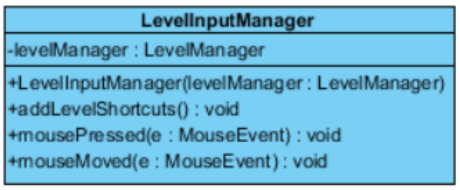
- **publicLevelManager(GameManagergameManager, intlevelNo, LevelScreen levelScreen):** Instantiates individual controller objects of levels.

Operations:

- **public void startLevel(intlevelNo):** Sends screen request to level screen manager to instantiate the level and sends timer request to timer manager for adjusting the timer according to the level number.
- **public void rotateShooter(int angle):** Sends rotation request to the shooter object which is taken from level screen manager.
- **public void throwBall():** Sends throw request to the shooter object which is taken from level screen manager.
- **public void changeBallType(Ball.BallType type):** Sends change ball type request to the shooter object which is taken from level screen manager after checking the items of the player by GameManager.
- **public void undoChangeBallType():** Undo the change ball type request and adds the item back to the player's account by GameManager.
- **public void drawCoin():**Sends request to level screen manager to draw a coin to the screen.
- **public void removeCoin():**Sends request to level screen manager to remove the coin from the screen.
- **public void checkCoin(Point p):** Sends request to level screen manager to check if the coin contains the clicked point or not and if it contains, it adds the coin to player balance.

- **publicbooleanisShooting():**Determines if the shooter is shooting or not.
- **public void pause():**Sends request to game manager to display pause menu screen.
- **public void help():** Sends request to game manager to display help screen.
- **public void freezeTimer():** Freezes the level timer.
- **public void finishLevel(int status):** Finishes the level according to status of win or lose.

LevelInputManager Class



Base class: MouseAdapter

Attributes:

- **privateLevelManagerlevelManager:** The main controller object of the levels which handles the user inputs by sending requests to entity objects taken from LevelScreenManager.

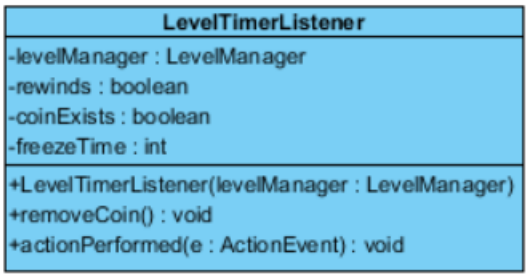
Constructor:

- **public LevelInputManager(GameManagergameManager)**

Operations:

- **public void addLevelShortcuts():** Adds the level shortcuts to level screen.
- **public void mousePressed(MouseEvent e):** Sends request to throw ball or add coin.
- **public void mouseMoved(MouseMoved e):** Sends request to rotate shooter.

LevelTimerListener Class



Interface: ActionListener

Attributes:

- **privateLevelManagerlevelManager:** The main controller object of the levels.
- **privateboolean rewinds:**Determines if the sequence is rewinding or not.
- **privatebooleancoinExists:** Determines if there is a coin in the screen or not.

- **privateintfreezeTime:**Keeps the time of freezing.

Constructor:

- **public LevelTimerListener(LevelManagerlevelManager)**

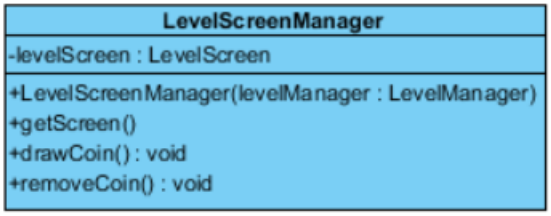
Operations:

- **public void removeCoin():**Sends request to level manager to remove the coin from the screen when the time is over.
- **public void actionPerformed(ActionEvent e):** Executes the main loop of the level.

LevelTimerListener.SequenceTimerListener Class

It is an inner class of LevelTimerListener which handles the sliding of ball sequence since its speed is different than the level speed.

LevelScreenManager Class



Attributes:

- **private LevelScreen:** Screen of the level which contains the level entities. It is the subsystem interface of Game Entity package.

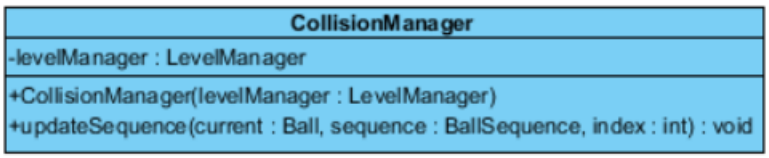
Constructor:

- **public LevelScreenManager(LevelManagerlevelManager)**

Operations:

- **public LevelScreen getScreen():** Returns the level screen.
- **public void drawCoin():** Draws a coin to the level screen.
- **public void removeCoin():** Removes the coin from the level screen.

CollisionManager Class



Attributes:

- **privateLevelManagerlevelManager:** The main controller object of levels which all individual controller objects are centralized.

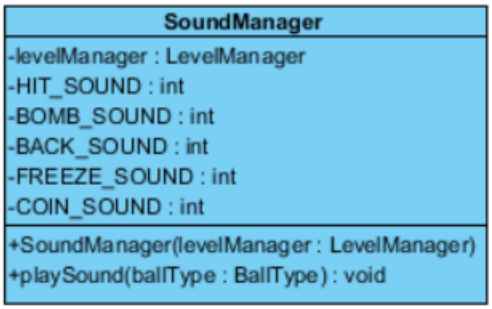
Constructor:

- **public CollisionManager(LevelManagerlevelManager)**

Operations:

- **public voidupdateSequence(Ball current, BallSequence sequence, int index):** Updates the ball sequence according to the collisions.

SoundManager Class



Attributes:

- **privateLevelManagerlevelManager:** The main controller object of levels which all individual controller objects are centralized.
- **private final int HIT_SOUND:** Sound of plain ball.
- **private final int BOMB_SOUND:** Sound of bomb ball.
- **private final int BACK_SOUND:** Sound of back ball.
- **private final int FREEZE_SOUND:** Sound of freeze ball.
- **private final int COIN_SOUND:** Sound of coin.

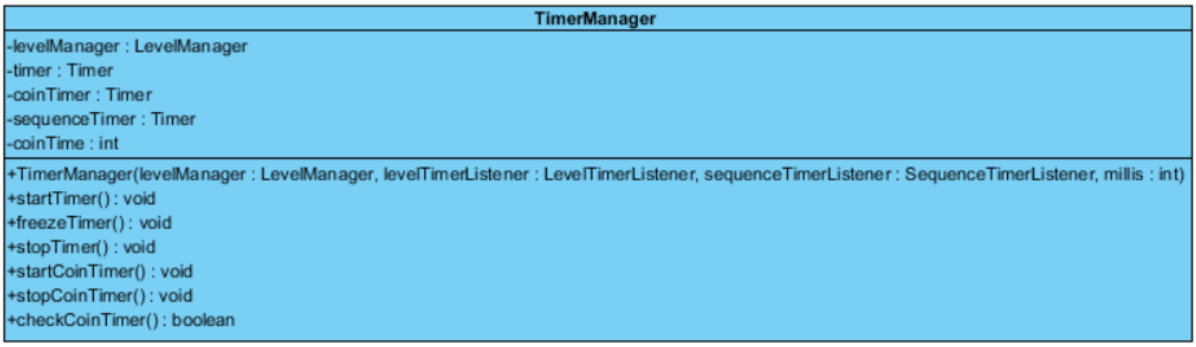
Constructor:

- **public SoundManager(LevelManagerlevelManager)**

Operations:

- **public void playSound(Ball.BallTypeballType):** Plays the sound according to the ball type. If the ball type is null, then it creates coin sound.

TimerManager Class



Attributes:

- **privateLevelManagerlevelManager:** The main controller object of levels which all individual controller objects are centralized.
- **private Timer timer:** Timer of the level.
- **private Timer coinTimer:** Timer of the appearance of coins.
- **private Timer sequenceTimer**Timer of the appearance of coins.**:**Timer which is responsible for sliding the sequence.
- **privateintcoinTime:**Time of the appearance of coins.

Constructor:

- **public TimerManager(LevelManagerlevelManager, LevelTimerListenerlevelTimerListener, LevelTimerListener.SequenceTimerListenersequenceTimerListener, intmillis)**

Operations:

- **public void startTimer():** Starts the level timer.
- **public void freezeTimer():** Freezes the level timer when freeze ball is thrown.
- **public void stopTimer():** Stops the level timer.
- **public void startCoinTimer():** Starts the coin timer.
- **public void stopCoinTimer():** Stops the coin timer.
- **publicbooleancheckCoinTimer():** Checks if the timer of coin ends or not.

5.2.3 Game Entity Subsystem Interface

Game Entity subsystem keeps the application domain objects as well as the level screen.It provides its interfaces by the LevelScreen class. The class has an association with LevelScreenManager from Game Control subsystem and ScreenManager from User Interface subsystem. Figure 5.2.3 depicts the class interfaces of Game Entity subsystem.

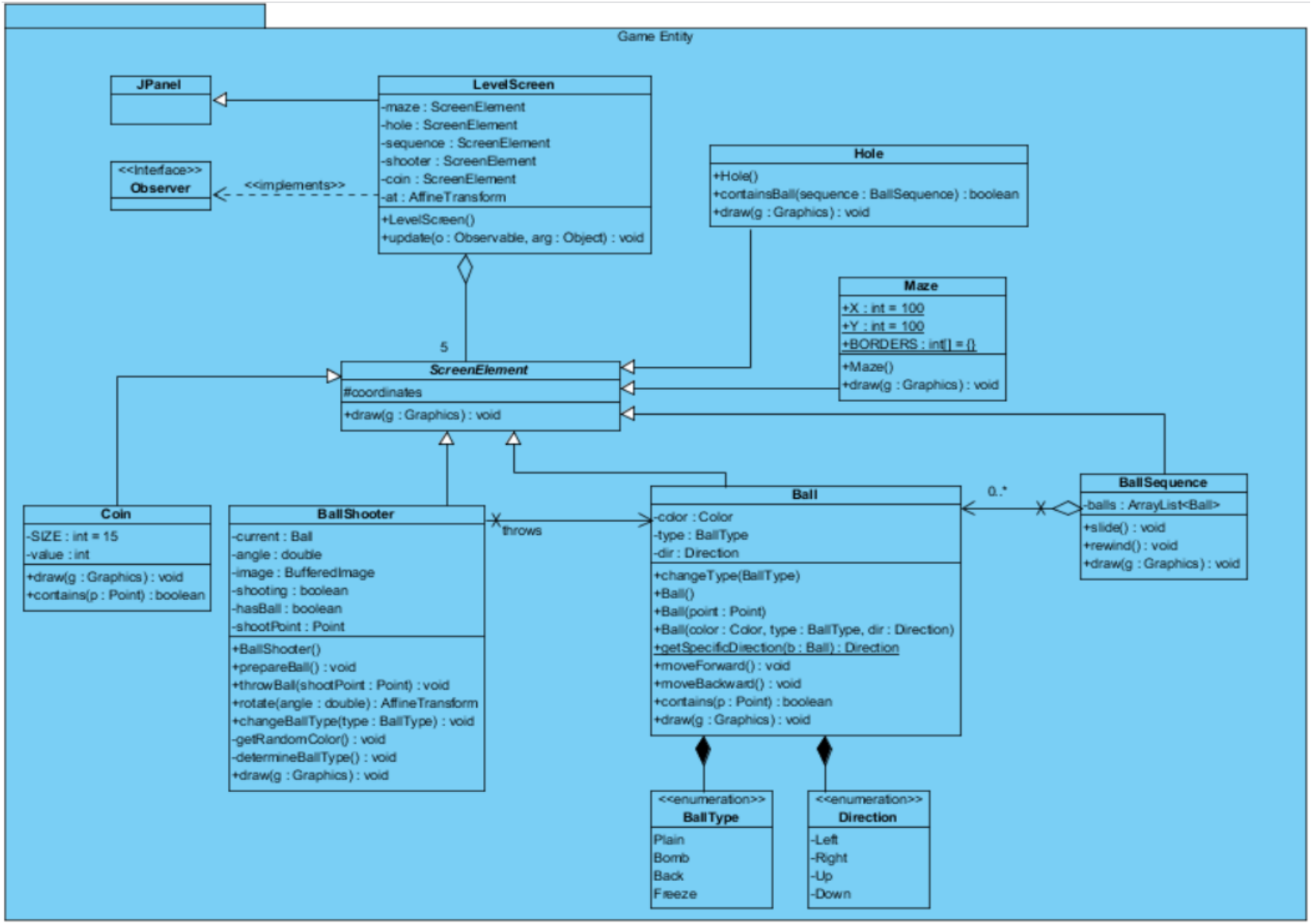
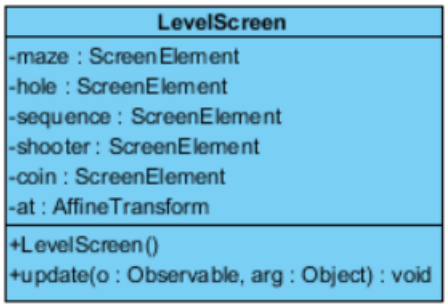


Figure 5.2.3 Game Entity Subsystem

LevelScreen Class

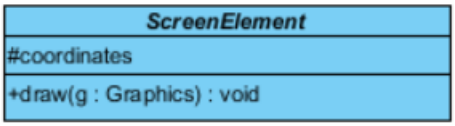


Base class: JPanel

Interface: Observer

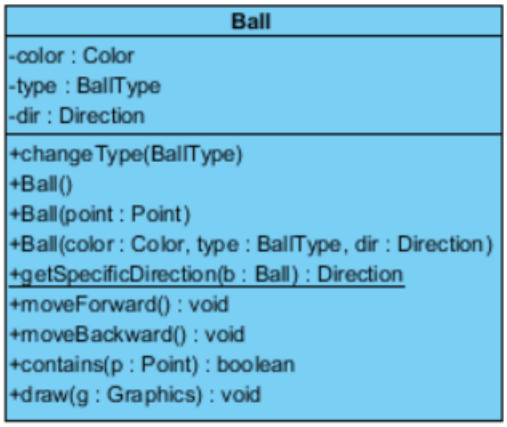
It represents the level screen and contains the screen elements. It is an observer for the players to contain their balance on the screen.

ScreenElement Class



It is an abstract class which is the generalization of all screen elements. It contains their coordinates and draw method since they are common for all.

Ball Class

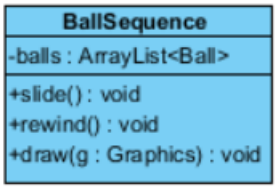


Base class: ScreenElement

Operations:

- **public static Direction getSpecificDirection(Ball b):** Returns the moving direction of a ball which is to be added to the sequence.
- **public void moveForward():**Moves forward in the maze in each loop.
- **public void moveBackward():** Moves backward in the maze in the case of back ball.

BallSequence Class

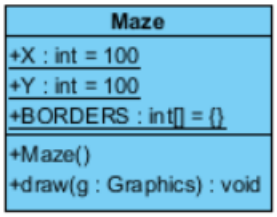


Base class: ScreenElement

Operations:

- **public void slide():** Slides the sequence in each loop.
- **public void rewind():**Rewinds the sequence in the case of back ball.

Maze Class

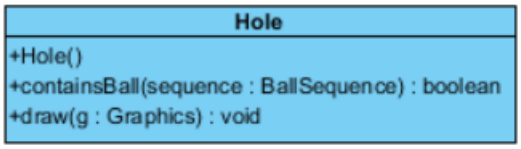


Base class: ScreenElement

Attributes:

- **public static int X = 100:** x coordinate of its upper left corner
- **public static int Y = 100:** y coordinate of its upper left corner
- **public static int[] BORDERS = {}:** The coordinates of its corners where the sequence changes its direction

Hole Class

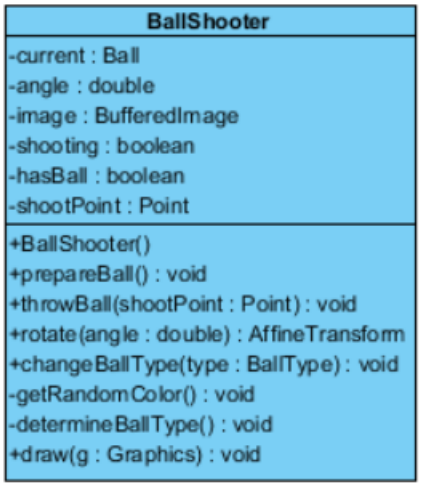


Base class: ScreenElement

Operations:

- **publicbooleancontainsBall(BallSequence sequence):** Determines if it contains any ball of the sequence or not.

BallShooter Class



Base class: ScreenElement

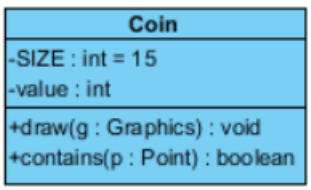
Attributes:

- **private Ball current:** The ball to be shot next.
- **private double angle:** Current angle of the shooter.
- **privateboolean shooting:** If it is shooting or not.
- **privatebooleanhasBall:** If it has a ball as current or not.
- **private Point shootPoint:** Point to be shot.

Operations:

- **public void prepareBall():** Prepares a ball to shoot in each loop.
- **public void throwBall(Point shootPoint):** Shoots a ball to the particular point.
- **publicAffineTransform rotate(double angle):** Rotates the shooter in the specified angle.
- **public void changeBallType(Ball.BallType type):** Changes the type of current ball.
- **private void getRandomColor():** Sets a random color to the current ball.
- **private void determineBallType():** Determines the type of current ball.

Coin Class



Base class: ScreenElement

Attributes:

- **private final int SIZE = 15:** Size of the coin.
- **private int value:** Amount of the coin.

Operations:

- **publicboolean contains(Point p):**Determines if the coin contains a point or not.

5.3 Specifying Contracts

This section includes the contracts of some classes in which there are important conditions. The contracts determine the pre-conditions, post-conditions and invariants of the methods such that they make their test cases easier to determine and they specify their exceptional conditions which need to be checked.

5.3.1 LevelManager Contracts

contextLevelManager::getLevelNo():int
post:result = self.levelNo

The result of the getLevelNo method is the level number which is an attribute in the class.

contextLevelManager::setLevelNo(intlevelNo):
inv:0 < levelNo
inv: levelNo <= 10
post:self.levelNo = levelNo

The level number must be between (0, 10].

contextLevelManager::startLevel(intlevelNo):
pre: not (self.levelTimerListener = null)

Level timer listener must not be null for initializing the timer of the level.

contextLevelManager::rotateShooter(double angle):
pre: not (self.levelScreenManager = null)
pre: not (self.levelScreenManager.getLevelScreen() = null)
pre: not (self.levelScreenManager.getLevelScreen().getShooter() = null)
inv: -PI <= angle
inv:angle <= PI

The angle must be between [-PI, PI].
The ball shooter in the screen must not be null for rotating it.

contextLevelManager::throwBall(Point shoot):
pre: not (self.levelScreenManager = null)
pre: not (self.levelScreenManager.getLevelScreen() = null)
pre: not (self.levelScreenManager.getLevelScreen().getShooter() = null)
inv: shoot != null
inv:shoot.x> 0
inv:shoot.x<levelScreenManager.levelScreen.width
inv: shoot.y> 0
inv:shoot.y<levelScreenManager.levelScreen.height

The point which the ball is to be shot must be in the screen.
The shooter must not be null.

contextLevelManager::updateSequence(Ball current, BallSequence sequence, int index):
inv: not (current = null)
inv: not (sequence = null)
inv: index >= 0
inv:index < sequence->size

The current ball of shooter must not be null.
The ball sequence must not be null.
The index must be within the range [0, sequence.size).

contextLevelManager::checkCoin(Point p):
pre:not (self.levelScreenManager.getLevelScreen().getCoin() = null)
inv: not (p = null)

There must be a coin in the screen.
The point must not be null.

context: LevelManager::changeBallType(Ball.BallType type):
pre: not (self.levelScreenManager.getLevelScreen().getShooter().getCurrent()= null)

The current ball of shooter must not be null.

5.3.2 GameManager Contracts

context: GameManager::startLevel(intlevelNo, LevelScreen levelScreen):
inv: not (levelScreen= null)
inv: 0 < levelNo
inv:levelNo <= 10

The level number must be in the range (0, 10].
The level screen must not be null for drawing the elements.

context: GameManager::addPlayer(String name, String password): int
pre: not (self.accountManager= null)
inv:not (name = “”)
inv:not (password = “”)

It must not already contain the particular player before.
It must contain the player after the operation.

context: GameManager::removePlayer(int id, String password): boolean
pre: not (self.accountManager= null)
pre: self.players->includes(self.key = id& self.value = password) = True
inv: not (password = “”)
post:self.players->includes(self.key = id & self.value = password) = False
post: self.players->size = self@pre.players->size - 1

It must contain the player before the operation.
It must not contain the player after the operation.
Id must be included in the hash map of player proxies.
Password must be valid.
Players’ size must be one less than before after the operation.

context: GameManager::addPlayerObserver(Observer o):
pre: self.accountManager.getPlayers().getObservers()->asSet()->includes (o) = False
post: self.accountManager.getPlayers().getObservers()->asSet()->includes (o) = True

Each player must not already include the particular observer before the operation.
Each player must include the particular observer after the operation.

context: GameManager::addPlayerProxy(int id, String password):
pre:self.players.getKeys()->asSet->includes(id) = False
inv: not (password = “”)
post:self.players->includes(self.key = id & self.value = password) = True

Id must not be included in the hash map of player proxies before the operation.
Password must not be empty.
Id and password must be included in the hash map of player proxies after the operation.

context: GameManager::loadPlayer(int id, String password):
pre:self.players.getKeys()->includes(id) =True
inv: not (password = “”)

pre:not (accountManager= null)

Id must be included in the hash map of player proxies.
Password must not be empty.
Account manager must be initialized before.

5.3.3 AccountManager Contracts

context: AccountManager::changePlayerPassword(String password):
pre: not (self.currentPlayer = null)
pre: not (self.currentPlayer.getPassword() = password)
inv: not (password = “”)
post: self.currentPlayer.getPassword() = password

The password of current player must not be the same with the new password before the operation.
The password of current player must be the same with the new password after the operation.

context: AccountManager::addPlayer(String username, String password):
pre: self.players.->includes(Player(Player.lastId, username, password)) = False
inv: not (username = “”)
inv: not (password = “”)
post: self.players.->includes(Player(Player.lastId, username, password)) = True

A player with the particular username and password must not be included before the operation.
A player with the particular username and password must be included after the operation.

context: AccountManager::removePlayer(int id):
pre:self.players.getKeys()->includes(id) = True

A player with the particular id must already exist.

context: AccountManager::addPlayerObserver(Observer o):
pre:self.players.getObservers()->asSet()->includes (o) = False
post:self.players.getObservers()->asSet()->includes (o) = True

Each player must not include the observer before the operation.
Each player must include the observer after the operation.

context: AccountManager::addPlayerCheckpoint(intlevelNo):
pre: not (self.currentPlayer = null)
pre: self.currentPlayer.getCheckpoint() < levelNo
post: self.currentPlayer.getCheckpoint() = levelNo

Current player's checkpoint must be less than the level number before the operation.
Current player's checkpoint must be equal to the level number after the operation.

context: AccountManager::addPlayerScore(intlevelNo, int score):
pre: not (self.currentPlayer.getScores.select(self.index = levelNo)= score)
inv: score > 0
inv: 0 < levelNo
inv:levelNo <= 10
post: self.currentPlayer.getScores.select(self.index = levelNo) = score

Current player's score of the particular level must not bethe score before the operation.
Current player's score of the particular level must be the score after the operation.
Score must be bigger than 0.
Level number must be in the valid range.

5.3.4 BallSequence Contracts

context: BallSequence::getBalls() Integer:
post: result = self.balls

The result must be equal the balls.

context: BallSequence::addToBalls(Ball b):
pre:self.balls.contains(b) = False
post: self.balls.contains(b) = True

Balls must not contain b before the operation.
Balls must contain b after the operation.

context: BallSequence::addToBalls(Ball b, int index):
pre: self.balls.contains(b) = False
inv: 0 <= index
inv: index <= self.balls.size
post: self.balls.contains(b) = True

Balls must not contain b before the operation.
Balls must contain b after the operation.

Index must be in the valid range.

context: BallSequence::removeBall(int index):
inv: 0 <= index
inv:index <self.balls.size

Index must be in the valid range.

context: BallSequence::slide():
pre: self.balls.size> 0

There must be at least one ball in the sequence.

context: BallSequence::rewind ():
pre: self.balls.size> 0

There must be at least one ball in the sequence.

context: BallSequence::contains(Point p) int:
pre:self.balls.size> 0
inv: not (p = null)
post: result = self.balls->select(contains(p)).getIndex()

There must be at least one ball in the sequence.
Point must not be null.
It returns the particular index which the ball in that index contains the point.

context: BallSequence::getSize() Integer:
post: result = self.balls->size

The result of the method is the size of the balls.

context BallSequence::get(int index) Ball:
inv: index>0
inv: index < self.balls->size
post: result= self.balls->select(self.index = index)

Index must be in the valid range.
The result of the method is the ball in the particular index.

contextBallSequence::draw(Graphics g):
pre: not (self.balls.point = null)
inv: not (g = null)

The coordinates of each ball must not be null.

5.3.5 BallShooter Contracts

context: BallShooter::getCurrent() Ball:
post: result = self.current

The result of getCurrent method is the current ball.

context: BallShooter::isShooting() boolean:
post: result = self.shooting

The result of isShooting method is whether the shooter is shooting or not.

context: BallShooter::setBallType():
pre: not (current = null)

Current must be initialized before initializing its type.

context: BallShooter::shoot(Point shootPoint):
pre: not (current = null)
inv: shootPoint.x> 0
inv:shootPoint.x<screenWidth
inv:shootPoint.y> 0
inv:shootPoint.y<screenHeight

The point which ball is to be shot must be in the screen.
Current must be initialized before shooting.

context: BallShooter::switchShooting():
post: self.shooting = not self@pre.shooting

Boolean value of shooting is switched.

context: BallShooter::getHasBall():
post: result = self.hasBall

It returns whether the shooter has a ball or not.

context: BallShooter::swicthHasBall():
post: self.hasBall = not self@pre.hasBall

Boolean value of hasBall is switched.

6. Conclusions and Lessons Learned

6.1 Overview of the Report

In conclusion, we organized our report in order to show how a Desktop game which is called “Crazy Shooter” will be designed and implemented. The report has four main sections which are called as Requirement Analysis, Analysis, Design and Object Design.

Requirement analysis part includes seven topics which are overview, functional requirements, nonfunctional requirements, constrains, scenarios, use case models and user interface. In this section, the necessary requirements for the game are considered and they are fulfilled. We tried to make the requirements verifiable such that there should not be an ambiguity about those for designing the system more efficiently. Moreover, the properties and rules of the game are defined in the overview part and the appearance and structure of the game are determined and supported by the screen mock-ups. Also, navigational path in the user interface section is for demonstrating the usability of the program. In addition to these, use case diagram depicts the functions of the game from the view of the external actors which are the players. In the design

The analysis section includes two topics which are object model and dynamic model. In the object model section, the class diagram is used to describe the static structure of the system. On the other hand, dynamic model part consists of activity diagrams, state chart diagram and sequence diagrams which represent the dynamic structure of the game. There are two activity diagrams and they demonstrate the dynamic behavior of use cases called “Login” and “Play Game”. In addition to those, there is a state chart diagram which models the behavior of the control class for level play called LevelManager. Five sequence diagrams indicate the dynamic behavior of scenarios for creating account, login, and starting and playing the game.

The system design section emphasizes on the solution domain of the project. In the design goal section, the goals of the system is determined and explained why they are important. In the subsystem decomposition section, the system is decomposed into three parts which have separate functionalities. Then, the hardware configurations used in the system are explained and key concerns are determined. These key concerns include persistent data management which is about managing the player accounts, access control and security which

is about authentication of the players and storage of their passwords, global software control which is about the main event handling mechanism of the software and boundary conditions which are about the initialization, termination and exceptional conditions.

Finally, there is an object design section which depicts the types of objects and signatures of methods in detail. Since it is one step before the implementation, it should include the data structures and return types of methods for mapping the models to code. Also in this part, the design patterns are determined which are some methods of ordering the classes in an efficient manner. These help to represent subsystem interfaces, handling the data transmissions between model and view objects and adapting incompatible types of objects to each other. At the end, the section includes the object constraints which help us to determine the boundary conditions of some methods.

To summarize, we organized an analysis report so that we can benefit from that while designing and implementing our project. Also by writing this report, we had a chance to shape the solution domain of the game. Then, according to the analysis report, we wrote our design report and identified the solution domain of the project. By using the contents of the report, we can prevent facing any problem and we can design the objects and implement our game more efficiently. For implementation, we designed our class interfaces and method structures as one step ahead of the design section. So by this overall report, we can start to implement our project.

6.2 Lessons Learned

By doing this project, there are many lessons to learn. At first, since this is a randomly assigned group project, we developed our skills to work with other people who may not be known before. Also in terms of the contents of the project, we developed our software development skills such that in the future, we can develop our projects in an organized way.By this project, we also learnt to deal with some software criteria such as response time, extensibility and security, and by considering these; we can update the project in the future more efficiently.In the overall sense, to understand the benefits and usage area of object-oriented development, this project was efficient for us.

References

- Bruegge, Bernd, and Dutoit, Allen H., *Object-Oriented Software Engineering,* Pearson Education Limited: 2014. Print.
- http://www.popcap.com/games/zumas-revenge/online(Development of idea)
- http://www.oxforddictionaries.com/definition/ (Domain Lexicon)