Bilkent University
Department of Computer Engineering

Object-Oriented Software Engineering Project

CS319 Project: River Adventure

Final Project Design Report

Nazlı Özge Uçan, Meder Kutbidin Uulu, Aras Heper, Hande Teke

Course Instructor: Uğur Doğrusöz
TA: İstemi Bahçeci

Table of Contents

Table of Figures

1. Introduction

1.1. Purpose of the System

In the game River Adventure, the player tries to survive as long as he can while swimming through the river, avoiding obstacles and collecting boosts and coins. Additionally, the player can access to the store, buy, upgrade various boosts and characters and can play with them during the game. River Adventure is game aimed for fun. Therefore, the purpose of our system is to design it easy enough for all age groups to enjoy and to design it reliable on performance so that the player wouldn't worry about his progress and lose fun.

1.2. Design Goals

User-friendliness:

The River Adventures application has to be that simple that everyone could understand it easily. The design of the game should be attractive, and should have proper naming on every button so that users of the game should not have any problem to understand them. If there is any ambiguity, a user should have an option to go to help page, learn the concepts of the game and its instructions.

Extensibility:

When the player plays the game many times, it gets so mundane and give up playing the game again. Therefore, to keep our players in our game, we have to add new levels to the game, change the backgrounds of some game screens, add new boosts or change some game rules. In order to achieve these features, our methods should be for general cases that we could implement them with much less coding. Otherwise we need to develop them from scratch just to add bit of new features.

Reliability:

The game should be bug-free and consistent to any boundary conditions. If there is a power-loss or any other problems occur not related to the game, then there should not be data loss so that the user could continue the game where she/he left off. The game should be tested in every stage while developing in order not to have unexpected crashes. Additionally, after the completion of the development, it should be given to few people to check if there is any bug.

High-performance:

As the aim of the game is to entertain its players, it should be as fast as possible, otherwise the player does not enjoy the game. Moreover, the game should move the objects smoothly and the animations should be handled without any pause. To achieve this, all the necessary images will be loaded to the memory before the game starts.

2. Software Architecture

2.1. Subsystem Decomposition

River Adventure system is decomposed to three parts: User Interface, Controller and Models. These parts have a runtime dependency. User Interface subsystem is the system that links the user to the game by means of view components. Game Control subsystem is the one with the main control functions. This subsystem is called by means of User Interface subsystem and runs the game. On the other hand, third subsystem is the Game Model. This one contains all the classes which interacts the game data so Game Control calls this subsystem when any game data is needed.

Closed Architecture

In River Adventure, each subsystem can access only the next one's services. Which means User Interface can access the Controller and Controller can access the Models only. This type of architecture may decrease the efficiency. On the other hand, by this way, subsystems are more independent of each other so that the complete architecture becomes more maintainable.

Coupling and Coherence of Subsystems

The main purpose of the existence of subsystems is creating the coherence. This means, each subsystem contains all the classes that services the related operations in the system. For example, User Interface contains all the GUI components that makes the user can interact with the system by means of its view. Therefore, all the classes are gathered to serve this purpose such as Frame Manager, Menu which have the dependencies with a bunch of listener, panel, frame and button classes. Unifying these kind of services in one subsystem to serve similar purposes provides the coherence to the system.

On the other hand, system has another achievement which is low coupling. Hence, each subsystem can access the other one by means of one or two classes so that subsystems mostly do not know about interfaces of other layers. By this way, each subsystem has a freedom to act independent from others. This means, changes in one subsystem does not affect others so that the system cannot be affected easily from the changes in one subsystem.

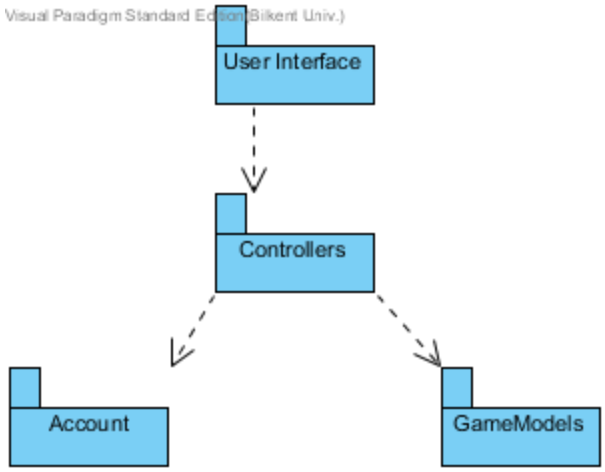

Figure 1. Subsystem Decomposition with Subsystem Dependency Details

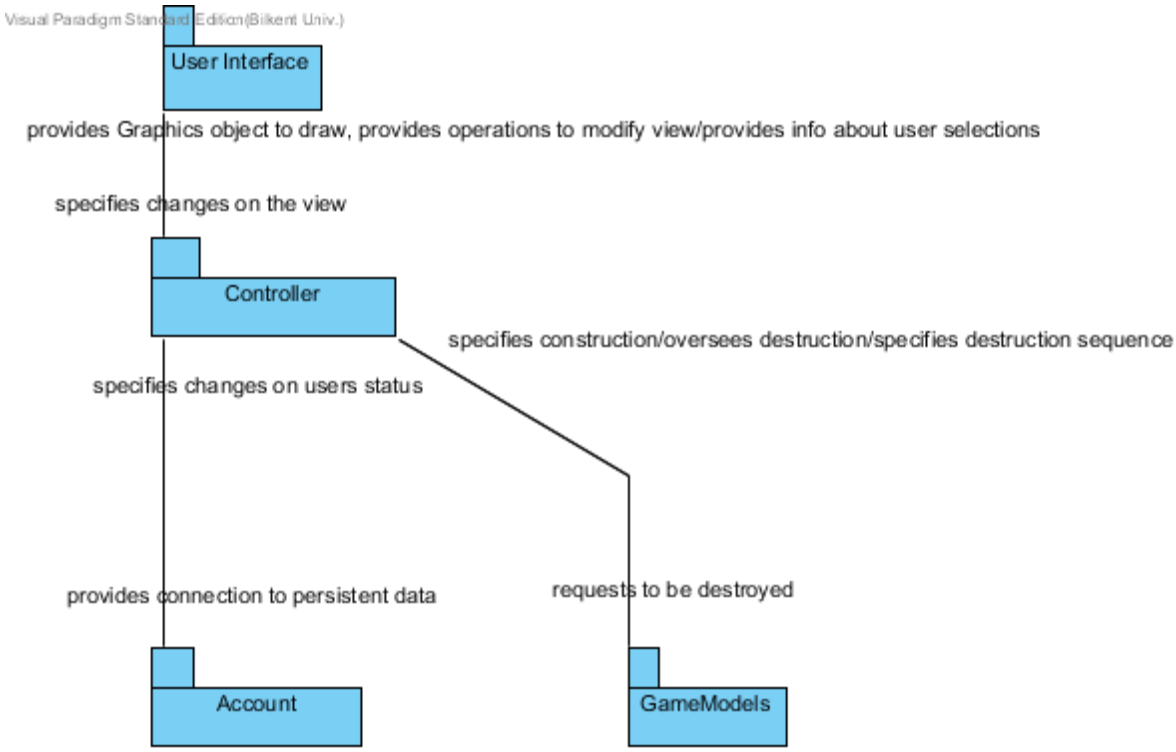

Figure 2. Subsystem Decomposition with Connection Details

2.2. Hardware/Software Mapping

The River Adventure game is a desktop application and the game is played offline so no internet connection is required. The game is single player game, in order to play no extra hardware or software is needed.

The game is not so complicated and do not require much memory, therefore device memory is enough for the game to be executed.

The computation rate is not too demanding for the single processor, so only one processor is required to maintain a steady state load.

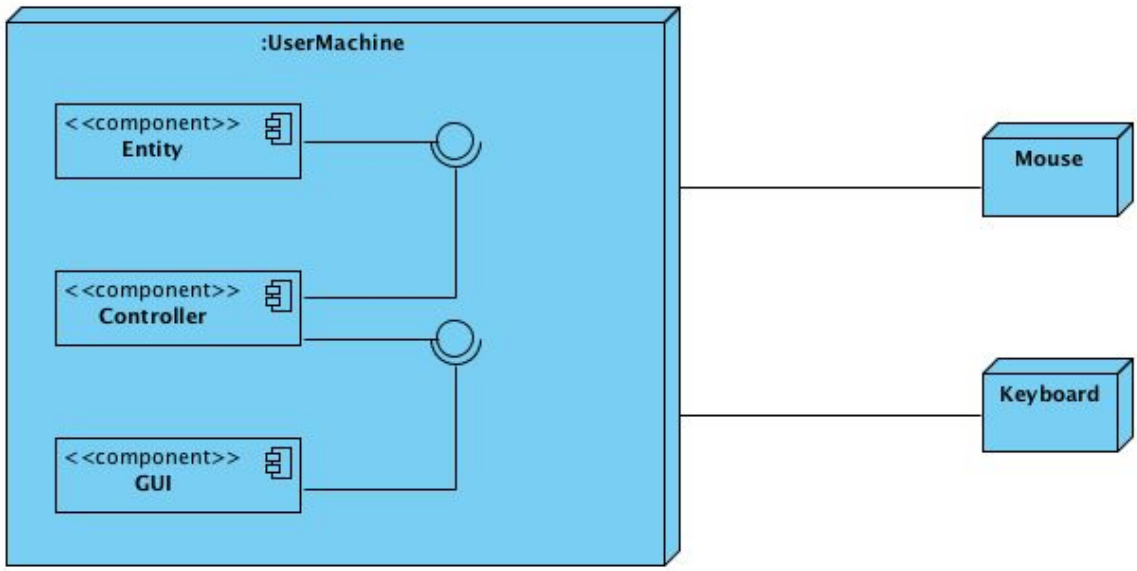

Figure 3. Deployment Diagram of River Adventure

2.3. Persistent Data Management

In River Adventure, we preferred to keep the persistent data as text files. The reason is because the saved data is not large and using database would be unnecessary for the system since it would make the cost bigger and hard to maintain and develop the system. The only data needed to exist after exiting the game consists of high scores, wallet as coin amount, unlocked boosts, characters, boost states and settings additional with images and sound files of the game. These are provided by a single writer of the file. Therefore, In River Adventure, most of the entities are not needed to be persistent.

2.4. Access control and Security

River Adventure can be played by anyone who has the application, there are no restrictions. Therefore, it doesn't require any authentication process. As stated earlier, the system doesn't require any internet connection so, the data of the user; collections, highscores and sound settings, etc., are stored offline. Since the game doesn't require any internet connection and can only played by a single player at a time, security is not a subject to concern for the game.

2.5. Boundary conditions

Initialization
In order to start the game, the player only needs to execute .jar file. It doesn't require any additional setups.

Termination
In order to terminate the game, the player can click the exit button (x button) on the frame in any screen. However, in order to terminate the gameplay, the player can either click exit button on the frame, which will close the entire game or he can finish the game which will eventually lead him to Highscores table and then to the Main Menu or he can click Pause button on the top left corner and then click Exit button to return to the Main Menu. For the third case, the player will lose all the progress he made.

Failure
If there is a problem about loading previous data of the user, such as his characters, boosts, upgrades or the data of the game such as settings, the game starts without the loading data, as it is initially opened.

3. Subsystem Services

3.1. User Interface Subsystem

This subsystem includes menus package that consist of Pause menu and Main Menu. Also Game Panel and Frame Manager takes place in this subsystem. This part of River Adventure is in charge of connecting the user to the game. This means the interface of the game that the user would face during the game process is created and managed in this subsystem. Java`s GUI components are used for this purpose. The interaction between Game Manager and Frame Manager provides the updates in graphics.

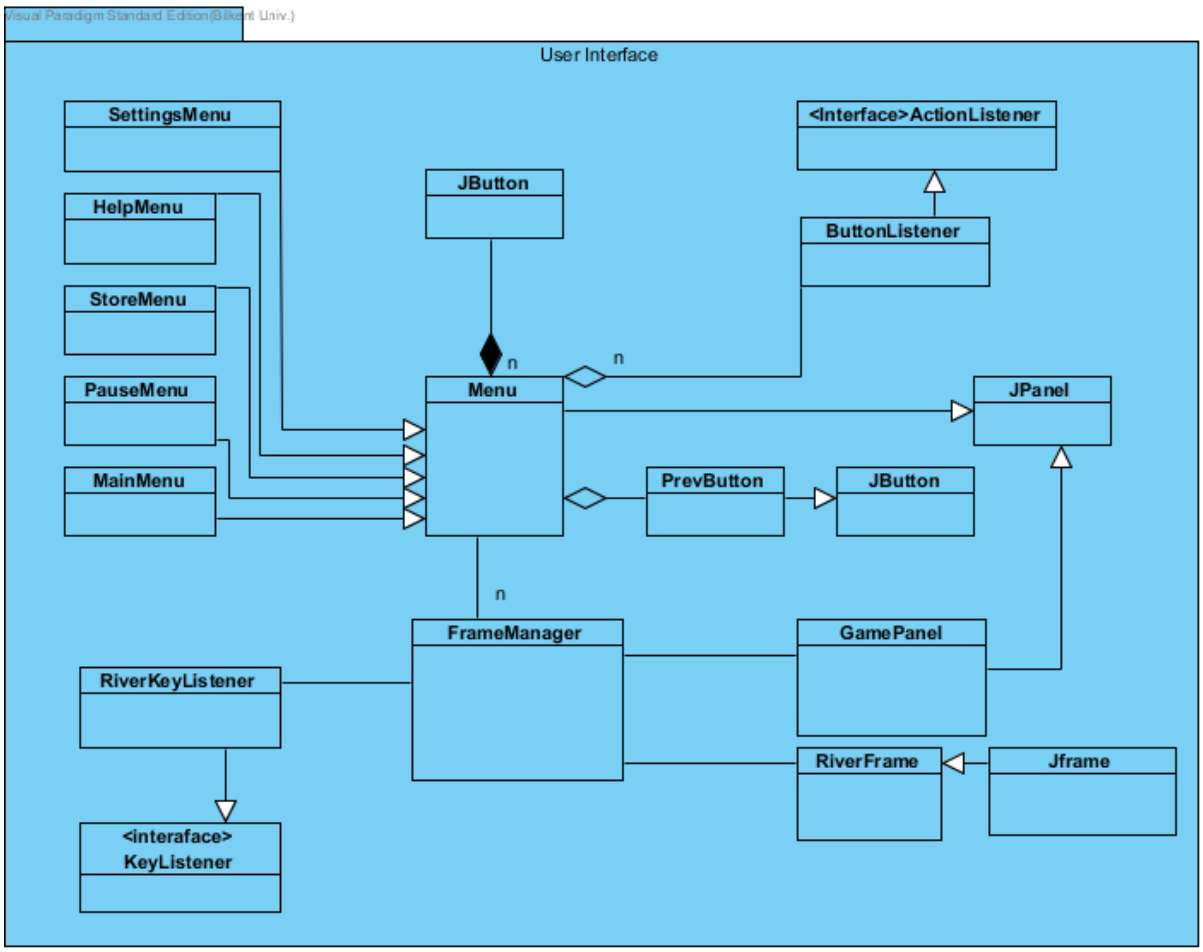

Figure 4. User Interface Subsystem Details

3.2. Controller Subsystem

Controller Subsystem includes Game Manager and Settings Manager. Main controls of the game that makes the total program exist and run takes place in this subsystem. This means Controller Subsystem contains the services that every game is obliged to handle. It takes the needed data from Models Subsystem and updates game's properties. Then by means of its own interaction between User Interface Subsystem, reflects them to the user.

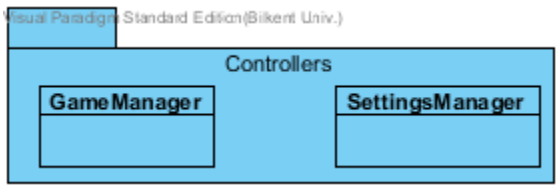

Figure 5. Controllers Subsystem Details

3.3. Account Model Subsystem

This subsystem contains Account and CoinWallet. Coin Wallet class puts the data of total amount of coins. That means the interaction between persistent data occurs in this subsystem. The account class contains other properties of the user's own game and manage them. It updates the values in text files, make the purchase, equip actions happen.

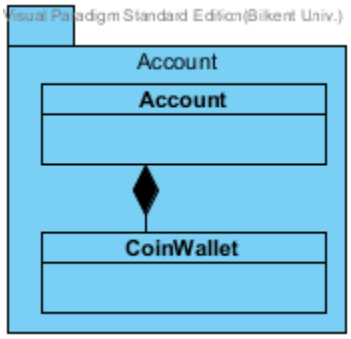

Figure 6: Account Subsystem Details

3.4. GameModels Model Subsystem

This Subsystem has three different parts. first one is the MetaModels that contains the RiverGame, River, Store and Line classes. Line is the place manages the game process in terms of objects that user would face while playing. It interacts with the River class which is the upper class of it. In River class, Line objects are generated. All the updates while the time flows, happens in this place. In Store class, the properties of the user are specified by means of the interactions between persistent data. As a final destination, RiverGame unifies other three classes by updating each property of River and Store classes. Therefore RiverGame class is the one that provides game's main functions that makes River Adventure is one unique game.

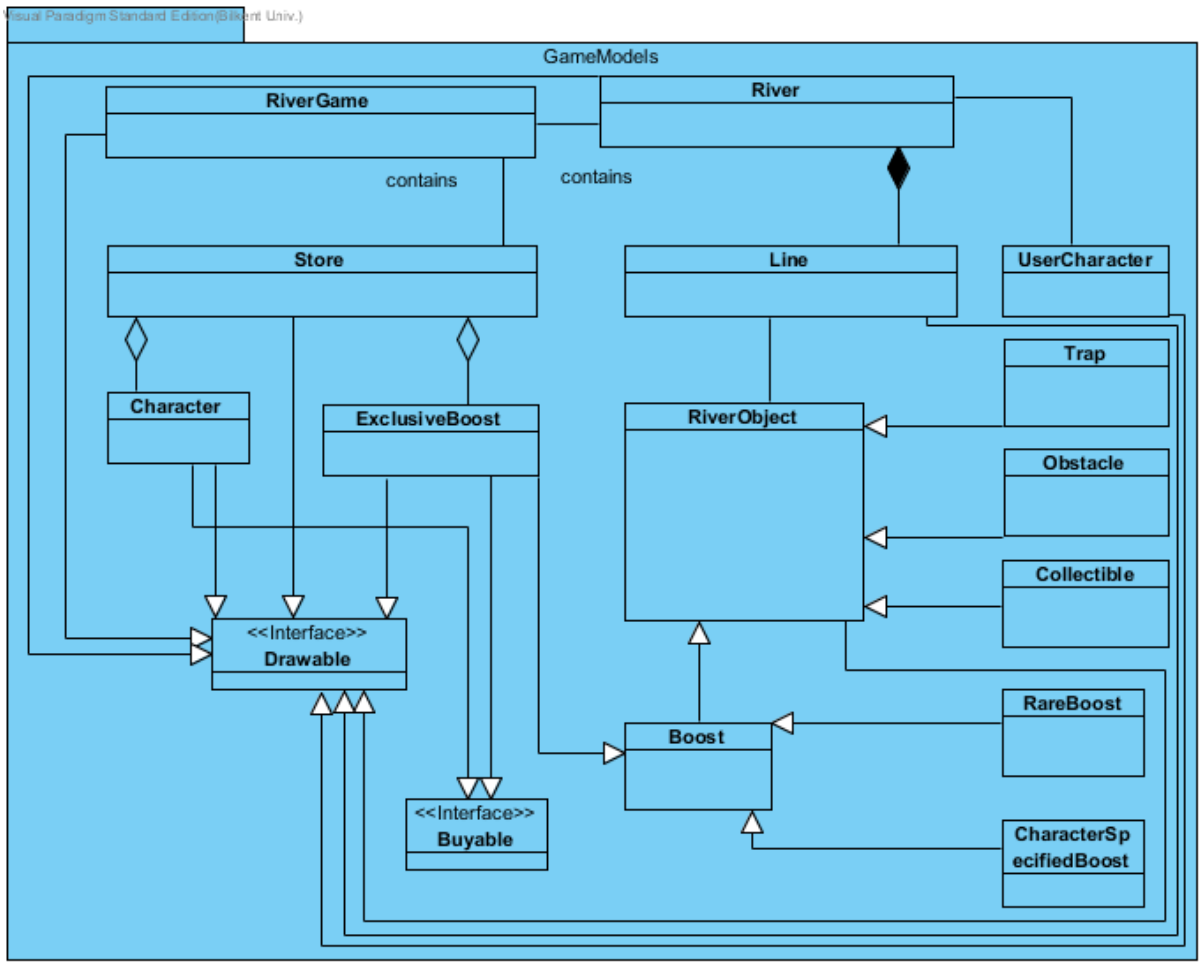

Figure 7. GameModels Subsystem Details

4. Low-Level Design

4.1. Object Design Trade-offs

Buy vs. Build

In the game that we are developing, we are not going to use any kind of physics engines or any other outsourced libraries other than those provided by standard Java. The system of the game is not complicated, so there is no point to buy any kind of software or part of a system for this game, therefore the game will be built by ourselves from scratch. Additionally, the images of the characters, obstacles, boosts and the design of the game overall will be designed by our team members.

Durability vs. Platform Dependence

The technology is developing so fast: old platforms are updating to the newer platforms, so in order not to face with platform incompatibility problem, we try to make the

game as few dependent from platform as possible. The game will be developed in Java programming language, and all the users can play the game just by having Java SDK. The game will not be using any native platform built-in functionalities or tools except keyboard and mouse.

Functionality vs. Ease of Use

There are many unknown applications exist which are much more functional than those simple popular applications. The reason is because people like to use simple apps, because they don't want to be distracted from many other functionalities the app contains and want to focus on main purpose of the app. Therefore, even though there are many functions that can be added to The River Adventure, for the sake of easiness those functions should be omitted.

Functionality vs. Cost

In further development of the game, there can be added more functionalities that costs more but makes the game more fun. The game can be a multiplayer game. In this case, the game will cost more, because in order to provide this functionality the game has to have the multiplayer server side to data exchange. Also, we can add more fancy boosts, and levels, which are not free, but for this one, we should have game payment system that increases the cost. However, at this stage, the game is a single player game, and all the boosts are free, so no need to buy a multiplayer server and game payment system.

High-performance vs. Memory

As we are developing a game, it is crucial for the game to have a high-performance. However, in order to achieve a high-performance, we sacrificed a memory, because before the game starts, all necessary data are loaded to the memory: sounds, images, etc.

4.2. Final Object Design

In River Adventure, Model-View-Controller is applied as an architectural pattern; Model - View - and Controllers strictly separated to different subsystems, with only the exception of 'Drawable' game models specifying certain procedures by their draw() method.

We integrated façade pattern into River Adventure; procedures of draw(), movement with move() are initiated from the façade methods sharing the same name at the RiverGame class, which initiates those procedures at lower classes. Similarly it is applied in all other non specific class procedures at RiverGame class, and further down below depending on whether they have reference to other objects that have significance to the said procedure. This allowed us more space to have a messy code at lower levels.

4.3. Packages

In River Adventure there are 3 main packages; UserInterface, Controllers and Models.

UserInterface Package: Inside of this package, there is GamePanel class, FrameManager class, RiverKeyListener class, RiverFrame class and Menus package.

Menus Package: It is located in UserInterface Package. It contains Menu class, PrevButton class, ButtonListener class, SettingsMenu class, HelpMenu class, StoreMenu class, PauseMenu class and MainMenu class.

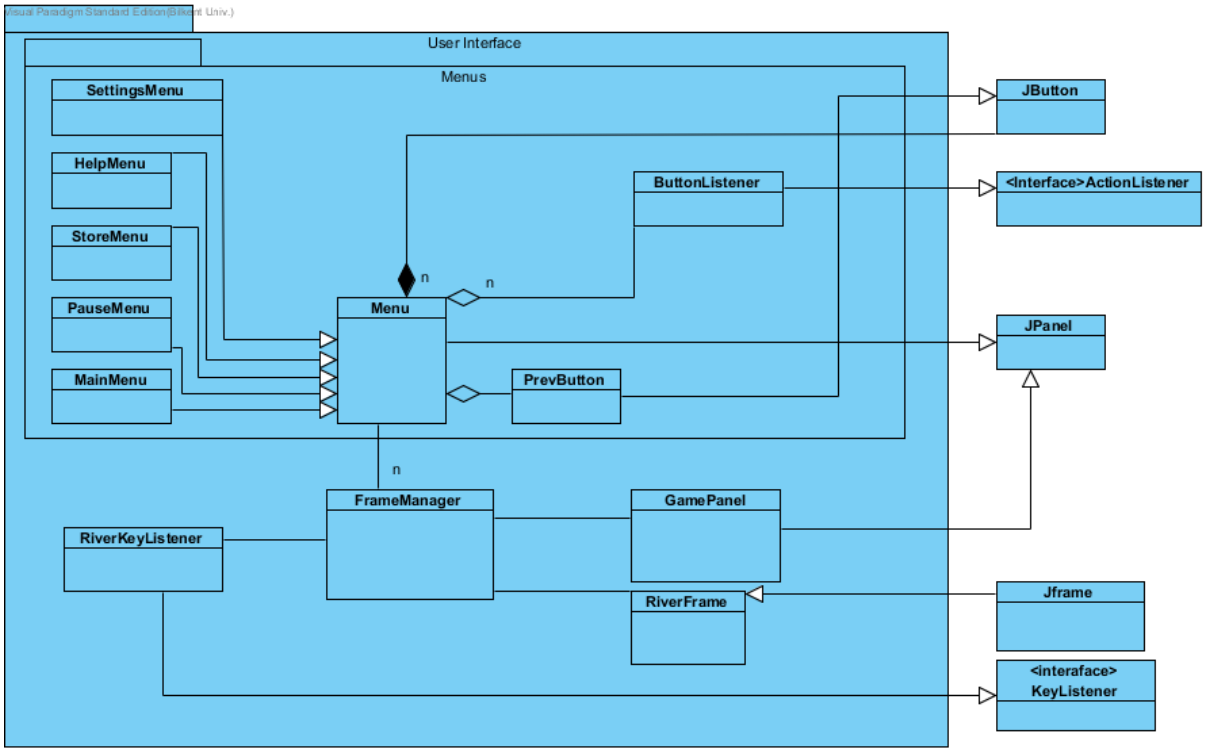

Figure 8 . User Interface Package

Controllers Package: Inside of this package, there is GameManager class and SettingsManager class.

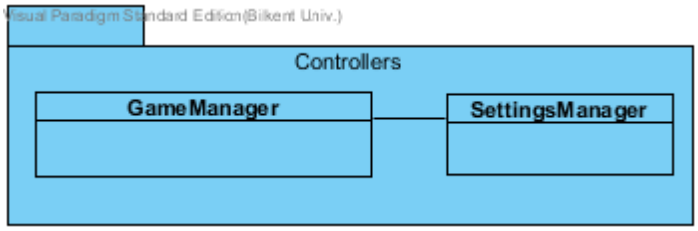

Figure 9. Controllers Package

Models Package: Inside of this package there are Account and GameModels packages.

Account Package: Inside of this package there are Account and CoinWallet classes.

GameModels Package: Inside of this package there are Drawable interface, UserCharacter class, MetaModels package, Buyable package and RealModels package.

MetaModels Package: Inside of this package there is RiverGame, River, Line and Store classes.

Buyable Package: Inside of this package there is Character, ExclusiveBoost classes and Buyable interface.
RealModels Package: Inside of this package there is RiverObject, Trap, Obstacle, Collectible, Boost, CharacterSpecifiedBoost and RareBoost.

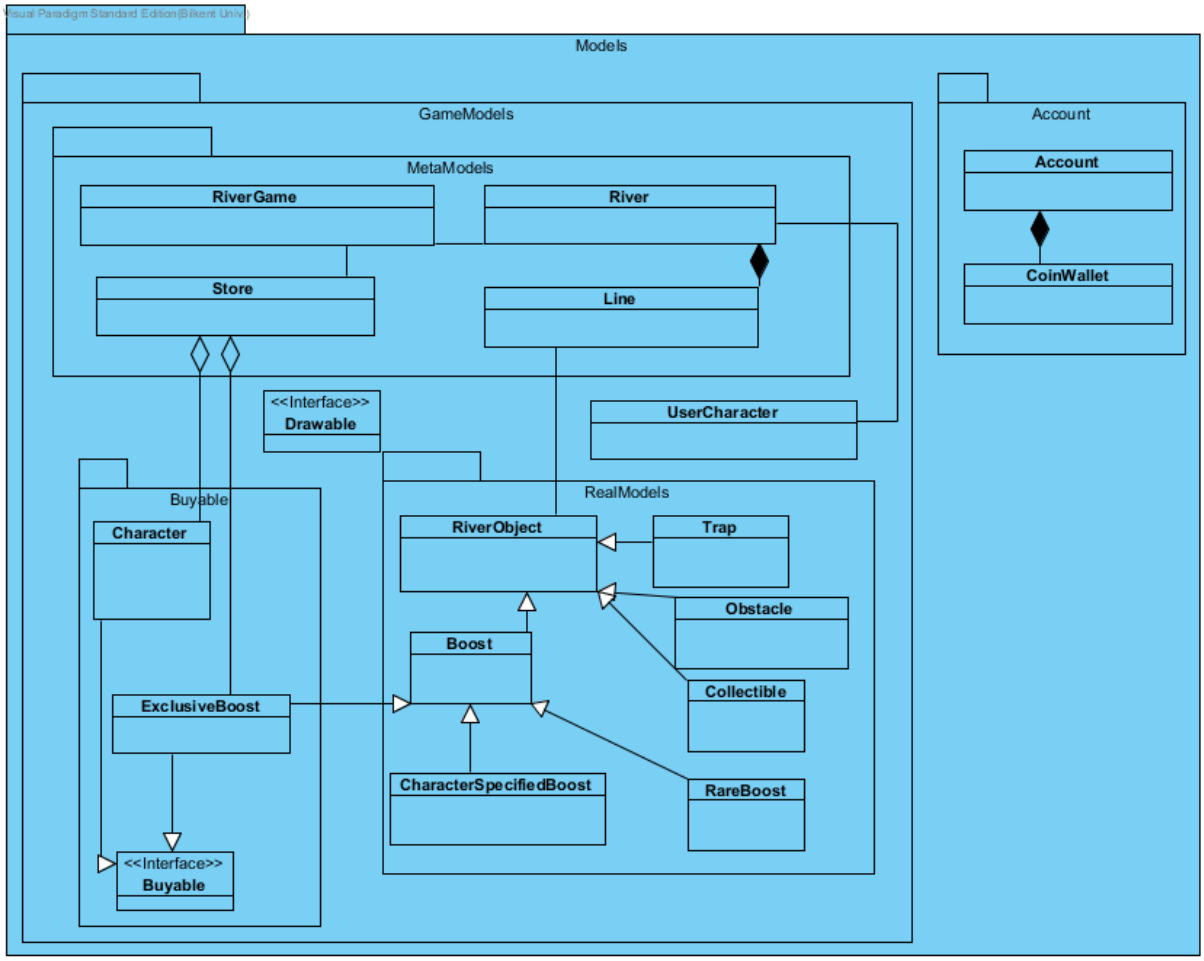

Figure 10. Models Package

4.4. Class Interfaces

User Interface Subsystem Objects:

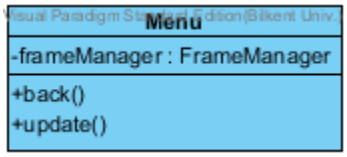

Figure 11. Menu Class

Menu: Menu is an abstract class that is implemented by all menus. It contains a reference to the instance of FrameManager class. Its methods:

- back(): this method calls FrameManager's returnToPrev() method which will goes back to the previous menu.
- update(): abstract method that is called by other menus.

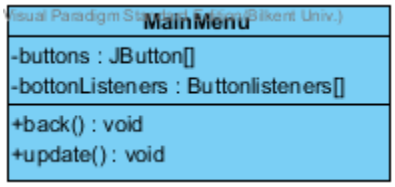

Figure 12. Main Menu Class

MainMenu: MainMenu extends Menu class and overrides its methods. It has an attribute "buttons" that is type list. It holds the buttons in the main menu. Its methods:

- back(): calls FrameManager' returnToPrev().
- update(): reconstructing UI elements according to the game state.

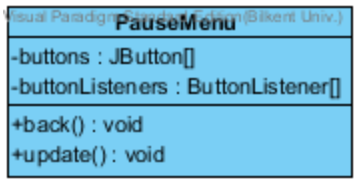

Figure 13. Pause Menu Class

PauseMenu: PauseMenu extends Menu class and overrides its methods. It has an attribute "buttons" that is type list. It holds the buttons in the main menu. Its methods:

- back(): calls FrameManager' returnToPrev().
- update(): reconstructing UI elements according to the game state.

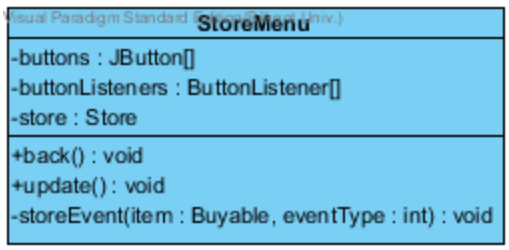

Figure 14: Store Menu Class

StoreMenu: StoreMenu extends Menu class and overrides its methods. It has an attribute "buttons" that is type list. It holds the buttons in the main menu. Its methods:

- back(): calls FrameManager' returnToPrev().
- update(): reconstructing UI elements according to the game state.

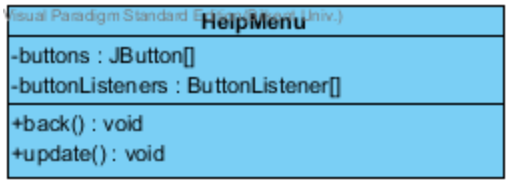

Figure 15. Help Menu Class

HelpMenu: HelpMenu extends Menu class and overrides its methods. It has an attribute "buttons" that is type list. It holds the buttons in the main menu. Its methods:

- back(): calls FrameManager' returnToPrev().
- update(): reconstructing UI elements according to the game state.

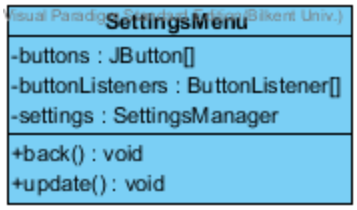

Figure 16. Settings Menu Class

SettingsMenu: SettingsMenu extends Menu class and overrides its methods. It has an attribute "buttons" that is type list. It holds the buttons in the main menu. Its methods:

- back(): calls FrameManager' returnToPrev().
- update(): reconstructing UI elements according to the game state.

JButton: Java's class to support UI buttons.

ActionListener: Java's action listener interface.

ButtonListener: Implements ActionListener for every button.

JPanel: Java's panel class. Our menus reimplements this class.

JButton: Java's button class. Our buttons inside menus reimplements this class.

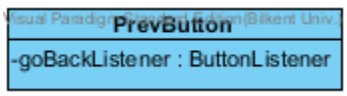

Figure 17. PrevButton Class

PrevButton: This class extends JButton. It contains listener to call back() of the parental menu.

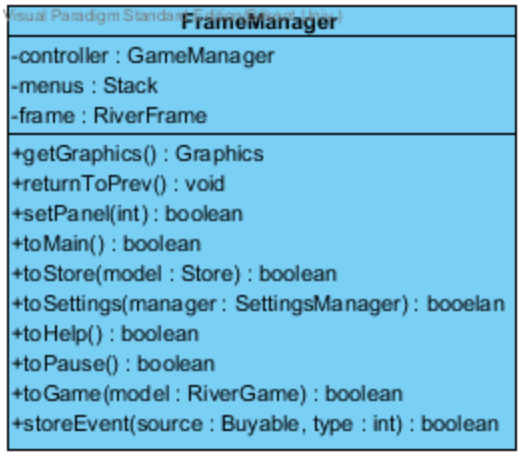

Figure 18. FrameManager Class

FrameManager: This class manages the menus and key effects. It has an attribute "menuStack" that contains the stack of menus to used by UI elements. It has several methods:

- getGraphics(): returns the graphics object.
- toGame(riverGame: Object): It takes a RiverGame object and creates a game panel with it.
- returnToPrev(): Updates the UI by opening the previous menu.
- toMain(): Updates the UI by opening the main screen.
- toStore(): Updates the UI by opening the store screen.
- toSettings(): Updates the UI by opening the settings screen.
- toHelp(): Updates the UI by opening the help screen.
- toPause(): Updates the UI by opening the pause screen.

JFrame: Java's frame class. Our frames reimplements this class.

Figure 19. RiverFrame Class

RiverFrame: Extends JFrame. This class is the frame of the river

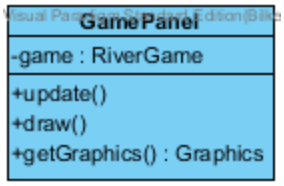

Figure 20. GamePanel Class

GamePanel: This class extends JPanel. This class is the place where all the objects come together and make a fully working game.
Its methods:

- update(riverGame: RiverGame): Updates all objects' position within a specified time.

- draw(): This draw method calls all other objects' draw methods appropriately, by the help of polymorphism.
- getGraphics(): Calls the Java's Graphics class.

Controller Subsystem Objects:

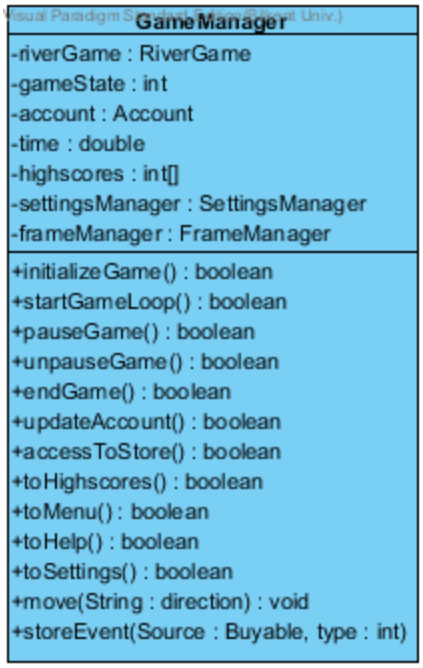

Figure 21. GameManager Class

GameManager: GameManager class informs UI components about models. It has an instance of FrameManager and SettingsManager. Its attributes:

- riverGame: it is a reference to the RiverGame object. This is the only connection of the game model.
- gameState: it is the general state of the game i.e. atMenu, atGame, atPause, atSettings, atPauseAndSettings etc. It has an int value for each state.
- account: It is a reference to the instance of Account class.
- time: It holds the lifetime of the game process as double.
- highScores: This is a list of integers that contains top 'n' plays.

Its methods:

- initializeGame(): It creates a new RiverGame object, connects the riverGame attribute to it, calls the riverGame object's related methods and calls the FrameManager's toGame() method.
- startGameLoop(): It starts the timer.
- pauseGame(): Stops the timer and updates the UI by calling the FrameManager's toPause() method.
- unpauseGame(): Restarts the timer and updates the UI.
- endGame(): Stops the timer and updates the UI.

- updateAccount(): Whenever the player hits some boosts, the game should keep track of them, therefore after each collision with a boost, this method is being called, which calls *updateWallet()* method of Account.
- accessToStore(): Updates UI by calling the FrameManager's toStore() method.
- toHighscores(): Starts the process to screen scores at the end of the game.
- toMenu(): Updates UI by calling the FrameManager's toMain() method.
- toHelp(): Updates UI by calling the FrameManager's toHelp() method.
- toSettings(): Updates the UI by calling the FrameManager's toSettings(settingsManager: Object) method.
- move(direction: String): Parameter direction could be either "left" or "right". Calls move method of the RiverGame object.

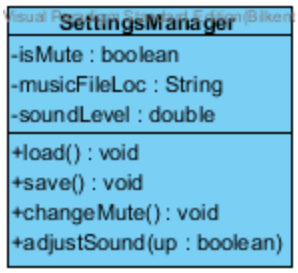

Figure 22. SettingsManager Class

SettingsManager: SettingsManager provides methods to UI for persistent data management about settings. Its attributes:

- isMute: Type is boolean. Holds the information about mute. If it is mute, true; false otherwise.
- musicFileLoc: Type is String. Holds the location of the music file.
- soundLevel: Type is double. Defines the level of the sound.

Its methods:

- load(): Loads persistent data and updates the isMute and soundLevel attribute.
- save(): Saves the current data to persistent data.
- changeMute(): Change isMute to the opposite state.
- adjustSound(up: boolean): It takes a boolean parameter to check if the sound increases or decreases (true for increase, false for decrease) and updates the soundLevel attribute.

Account Model Subsystem Objects:

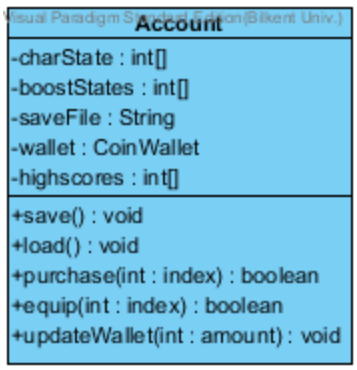

Figure 23. Account Class

Account: It holds the persistent data of the user.

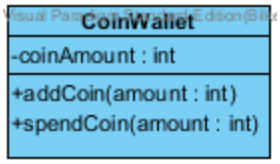

Figure 24. CoinWallet Class

CoinWallet:

- addCoin(amount: int): Increments coinAmount by speficied amount.
- spendCoin(amount: int): Decrements coinAmount by speficied amount.

GameModels Model Subsystem Objects:

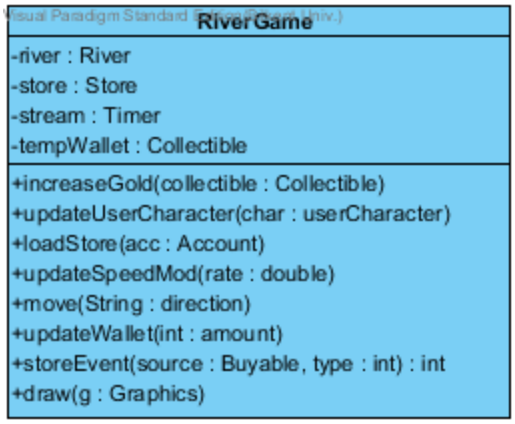

Figure 25. RiverGame Class

RiverGame: This class maintains the communication between GameManager and River and Store classes. It has an instance of the River class and Store class.
Its attributes:

- stream: Timer object observing OS time to initiate game loops.
- river : The river object.
- store : The store object.
- tempWallet : holds the amount of coin that is gained during the game as a collectible value.

Its methods:

- increaseGold(collectible: Collectible): This method takes a collectible object as parameter and updates the tempWallet accordingly.
- updateUserCharacter(char: Character): Sets the current character that the player plays with during game.
- loadStore(acc: Account) : Updates store with account info.
- updateSpeedMod(rate : double):

- move(direction: String): Parameter direction could be either “left” or “right”. Calls move method of the River object.
- updateWallet(amount: int): Updates the amount of coin of the account. Amount could be either negative or positive.
- addGold(amount : int) : Updates temp wallet, negative values won’t be accepted.
- endMe() : called by River at the end of the game.
- draw(g : Graphics): Implements Drawable.

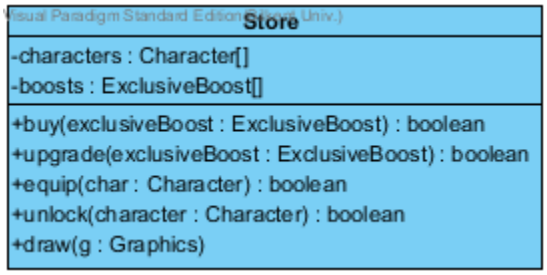

Figure 26. Store Class

Store: The Store class holds the characters and boosts that the player can purchase from the store. Its attributes:

- characters: The list of all characters that the player can purchase/equip/unequip.
- boosts: The list of all boosts that the player can unlock/upgrade.

Its methods:

- buy(exclusiveBoost: ExclusiveBoost): This method takes the exclusive boost as the parameter and updates the wallet and the state of the boost accordingly.
- upgrade(exclusiveBoost: ExclusiveBoost): This method calls the upgrade method of the exclusive boost.
- equip(char: Character): This method sets the isEquipped attribute of the character as true and assigns it as the current character.
- unlock(character: Character): Sets the isUnlocked attribute of the character to true.
- draw(g : Graphics): Implements Drawable.

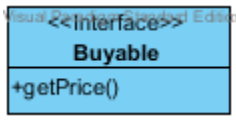

Figure 27. Buyable Class

Buyable: Interface for characters and exclusive boosts.
Signature method:

- getPrice() : return the cost.

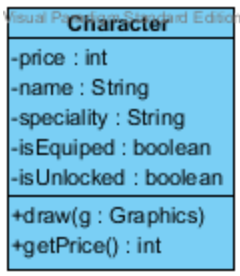

Figure 28. Character Class

Character: The Character class is the class of characters that the player can see/buy/equip in the store. Its attributes:

- name: The type is String. It is the name of the object.
- price: The type is int. It is the price of the object.
- speciality: The type is String. It is the description of the object.
- isEquipped: The type is boolean. If the character is equipped this becomes true, otherwise it is false.
- isUnlocked: The type is boolean. If the character is unlockes this becomes true, otherwise it is false.

Its methods:

- draw(g : Graphics): Implements Drawable.
- getPrice() : returns the cost of the objects, type is int.

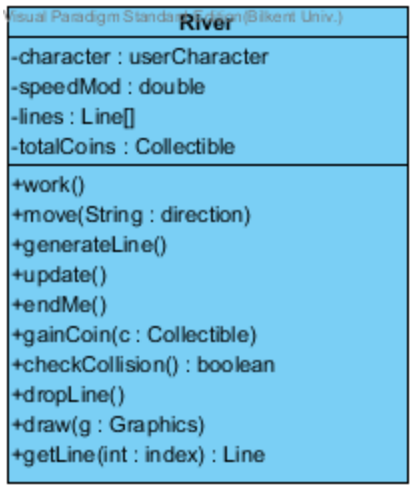

Figure 29. River Class

River: Universal set for all visible game objects. Holds multiple Line classes, and generates/ moves/ destroys them in time. This class also self-checks collision after a player movement or time tick. Generation/move/destruction speed changes by the “speedMod:double” and “effectSpeedMod:double”, which river adjusts with the effects affecting the user and game time. Its attributes:

- character: The instance of UserCharacter class.
- speedMod: The type is double. Defines the speed of the game which is in fact the ratio per loop that Line objects are generated, and moved.

- lines: A list of line objects. Whenever a new line is generated with river objects on it, it is added to the list.
- totalCoins: Total amount of coins that is gained through game. Type is Collectible.

Its methods:

- work(): Method called in a single standard game loop.
- move(direction: String): Updates the position of the character.
- generateLine(): Creates Line objects and updates “lines”.
- update(): It updates the positions of the objects within a specified time.
- endMe(): Is called by UserCharacter when users hp is zero.
- gainCoin(): Updates the totalCoins whenever the character collides with a collectible.
- checkCollision(): Checks if there is a collision between the character and the river objects on the line. If there is collision, handles it accordingly.
- dropLine(): Deletes the line after it reaches to lower end of the river.
- draw(g : Graphics): Implements Drawable, calls draw()s on lower objects.
- getLine(index: int): returns the index of the line from lines.

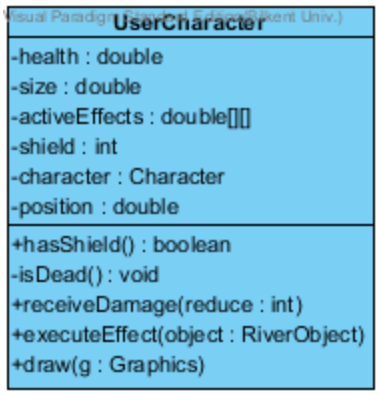

Figure 30. UserCharacter Class

UserCharacter: This class defines the user character in the game. Its attributes:

- character: The instance of the Character class.
- health: The health attribute holds the dynamic health of the user character.
- size: The size attribute holds the dynamic size of the user character.
- activeEffects: This is a list of effects on the user character.
- shield: Shield attribute is type int and is the number of shields that the user character has. If it doesn’t have any shield, the initial value is 0. However for character “the Beaver”, the initial value is 3. For each shield it gains, the shield attribute updates itself according to the type of the shield.
- position: The position attribute is type double. It is the location of the user character along the bottom of the river.

Its methods:

- hasShield(): Checks if the user character has shield or not. Returns true if the shield attribute is greater than 0, returns false otherwise.
- isDead(): calls endMe() of the River object.

- receiveDamage(reduce: int): If there is a collision between the user character and an obstacle, this method is called. It updates the health of the user character according to the reduce parameter.
- executeEffect(riverObject: RiverObject): If there is an effect of the river object on the user character, this method is called. It updates the user character accordingly.
- draw(g : Graphics): Implements drawable.

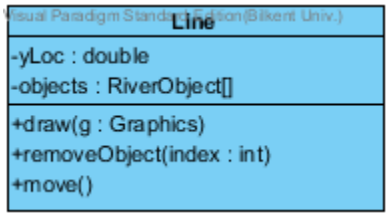

Figure 31. Line Class

Line: The instances of Line class are the non-visible game objects. The river has multiple line objects and each line object holds river objects. Its attributes:

- yLoc: Type of the yLoc is double. It is the location of the line in the y-axis along the river.
- objects: Each line object has a list of river objects that is stored in this attribute.

Its methods:

- draw(): Implements drawable. Calls lower objects' draw()s.
- removeObject(index: int): This method removes the object on the line of the specific index.
- move(): This method updates the y-location of the line.

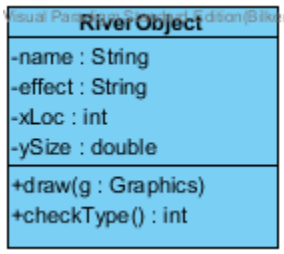

Figure 32. RiverObject Class

RiverObject: An abstract father class for all river objects. It holds the basic information for each river object. Its attributes:

- name: Type is String. It is the name of the object. It is initially null, assigned for each seperate object.
- effect: The type is String. It is the effect of the object. It is initially null, assigned for each seperate object.
- xSize: The x-dimension of the object. It is type int.
- ySize: The y-dimension of the object. It is type int.

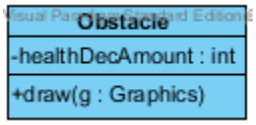

Figure 33. Obstacle Class

Obstacle: An obstacle is an in-game object that the player can see in the river. It is a child class of RiverObject. If it collides with the player the specified amount of health is reduced from the user character according to the type of the trap. Its attributes:

- healthDecAmount: Type is int. If the user character collides with an obstacle that amount of health is reduced.

Its Methods:

- draw(g : Graphics): Implements drawable.

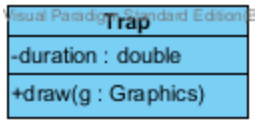

Figure 34. Trap Class

Trap: A trap is an in-game object that the player can see in the river. It is a child class of RiverObject. Each trap has a special effect on the game/user character. Its attributes:

- duration: Type is int. It specifies the amount of time that the trap adds to its correlated effect.

Its Methods:

- draw(g : Graphics): Implements drawable.

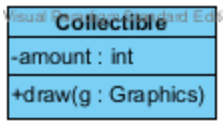

Figure 35. Collectible Class

Collectible: A collectible is an in-game object that the player can see in the river. It is a child class of RiverObject. Its attributes:

- amount: Type is int. It specifies the amount of coin of the collectible object.

Its Methods:

- draw(g : Graphics): Implements drawable.

Figure 36. Boost Class

Boost: A boost is also a father class of ExclusiveBoost, CharacterSpecifiedBoost and RareBoost. It is an abstract class. Its methods:

- draw(g : Graphics): Implements drawable. This is an abstract method that will be implemented in each child class.

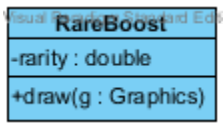

Figure 37. RareBoost Class

RareBoost: Boosts that turns objects onto collectables, has its own rarity modifier that changes its likelines. Its attributes:

- rarity: Type is int. Defines how rare that the boost is generated.

Its methods:

- draw(g : Graphics): Implements drawable.

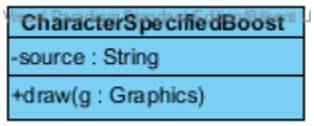

Figure 38. CharacterSpecificBoost Class

CharacterSpecifiedBoost: Boosts that shows up during the game if and only if the according character is unlocked. Its attributes:

- source: Type is String. It is the source character of the boost.

Its methods:

- draw(g : Graphics): Implements drawable.

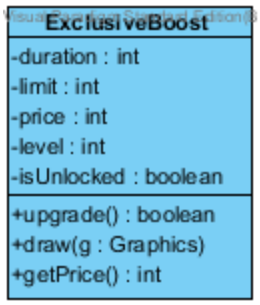

Figure 39. ExclusiveBoost Class

ExclusiveBoost: Boosts that the player can buy from the store. Its attributes:

- duration: Type is int. It determines the duration of the boost. 0 for boosts that don't require duration.
- limit: Type is int. It determines the tolerance of the boost. 0 for boosts that don't require limit.
- price: Type is int. It is the price of the boost.
- level: Type is int. This is the level of the boost.
- isUnlocked(): Type is boolean. Determines if the boost is unlocked. If true, unlocked; false otherwise.

Its methods:

- upgrade(): Increments level attribute by one.

- draw(g : Graphics): Implements drawable.
- getPrice(): Returns the cost of this item.

5. Glossary

OS - Operating System