Bilkent University
Department of Computer Engineering

Object-Oriented Software Engineering Project

CS319 Project: River Adventure

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.

Trade-offs

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.

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.

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 ic 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.

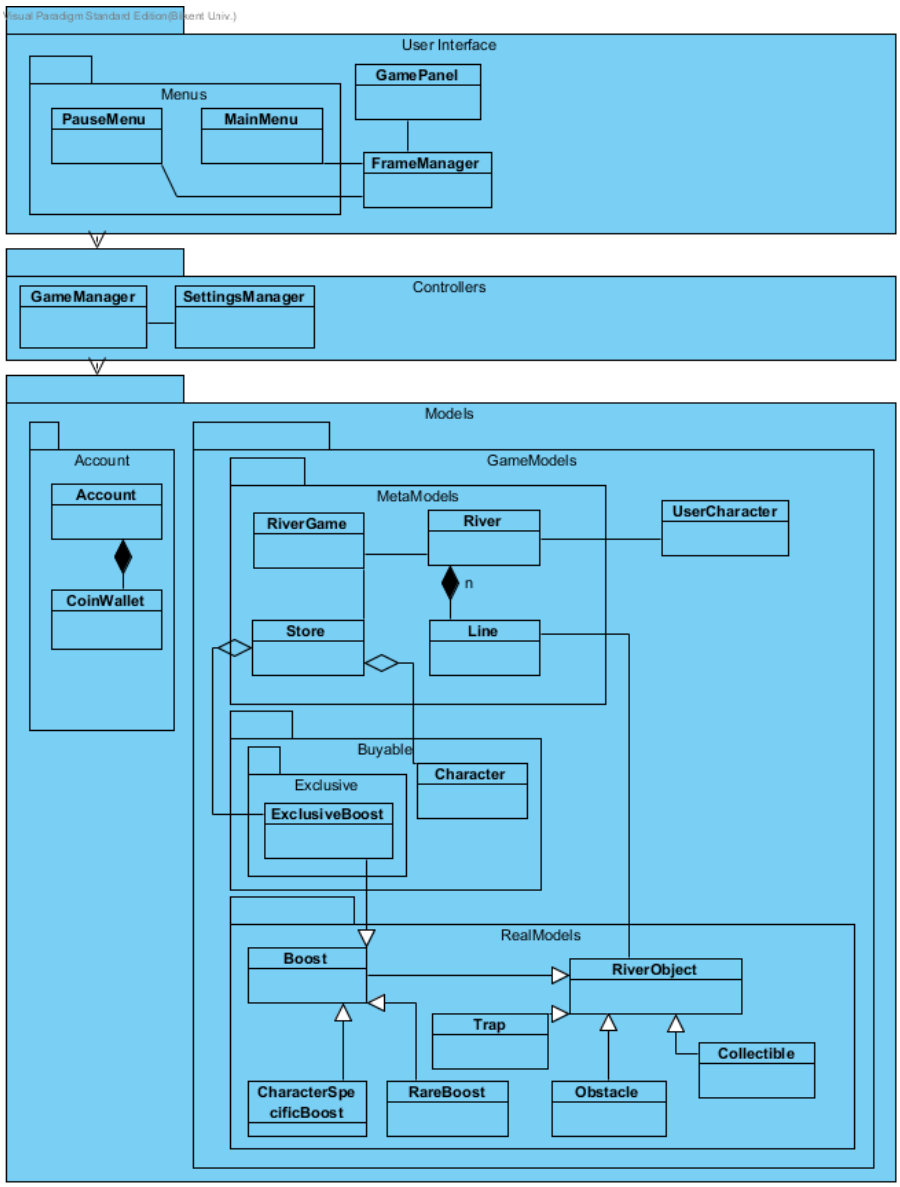

Figure 1. Subsystem Decomposition with Subsystem Details

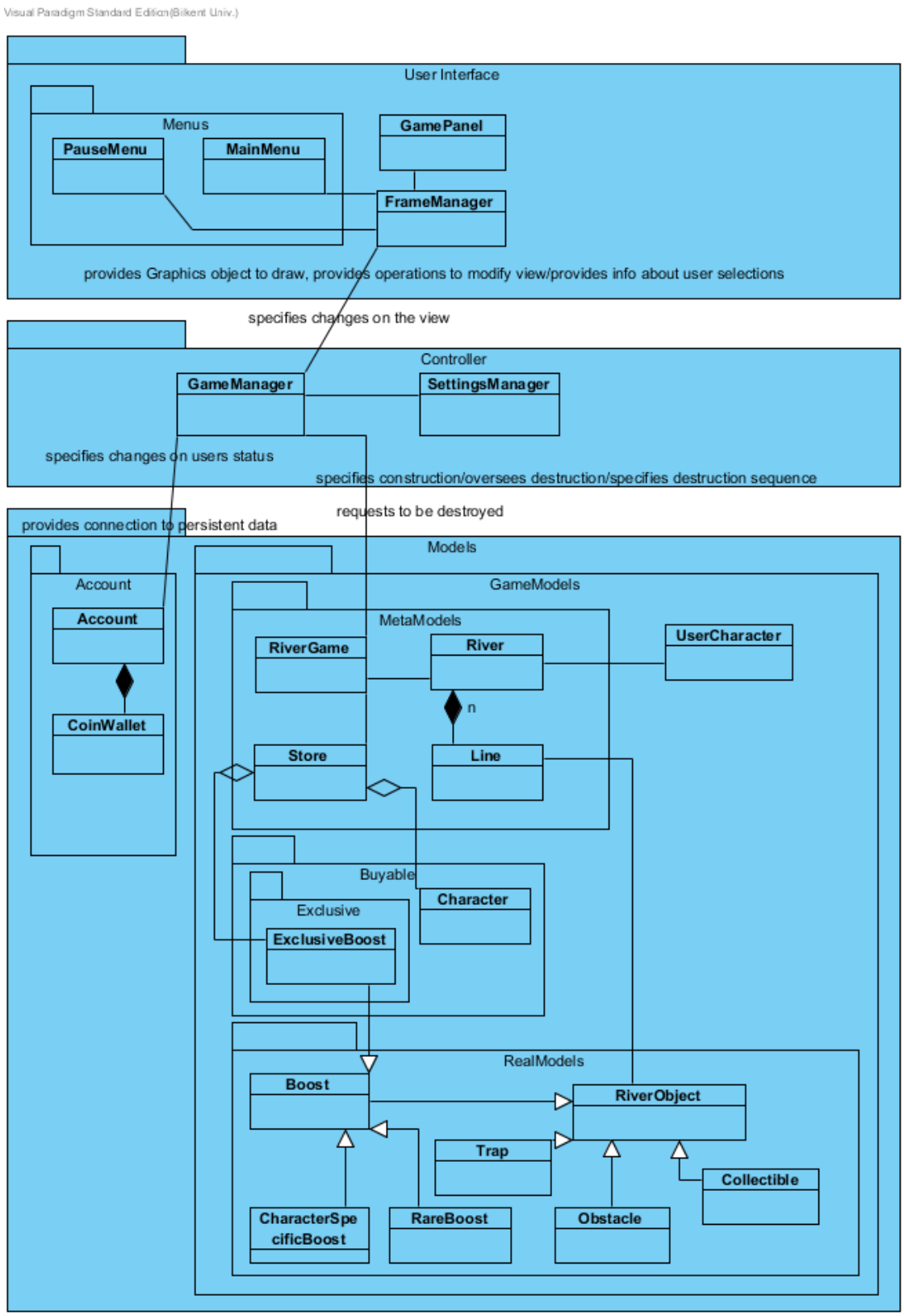

Figure 2. Subsystem Decomposition with Connection Details

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.

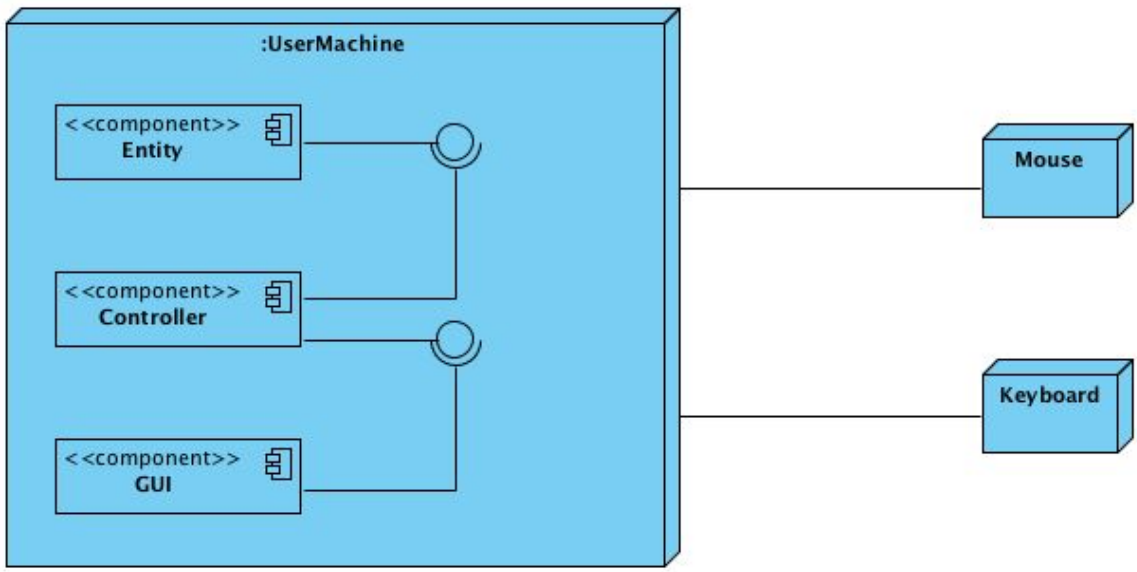

Figure 3. Deployment Diagram of River Adventure

2.3. Persistent Data Management

In River Adventure, we preferred to keep persistent data are kept as text files. The reason is because the saved data is not that large, database would be useless at the system. This would make the cost bigger and hard to maintain and develop the system. The only data needed to exist after exiting the game is consist of high scores, coin amount, settings and boost states and additional with images and sounds of the game. These are will be 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. Detailed Object Design

Object will be discovered further as parts of the related subsystems, most of the getter methods are not donated, however setter methods are not supplied as long as they are not explicitly added.

3.2. User Interface Subsystem

User Interface subsystem is composed of a subsystem named as 'Menus', and four renegade classes.

Menus:

The subsystem 'Menus' supply 'FrameManager' class with five types of menus, all of those are reimplementation of an abstract class 'Menu'. None of those classes have any

other difference from each other, besides buttons contained by them. For example Main Menu will have five buttons: Play, Settings, Help, Store, Exit; while Pause will only have : Settings, Resume, and Exit.

FrameManager:

Manages the order of menus, and sustains the communication with the Controllers subsystem. It will be instantiated directly after GameManager. Listens keyboard input through 'RiverKeyListener'.

Game Panel:

Instantiated when the user starts to play the game. Draws in-game objects.

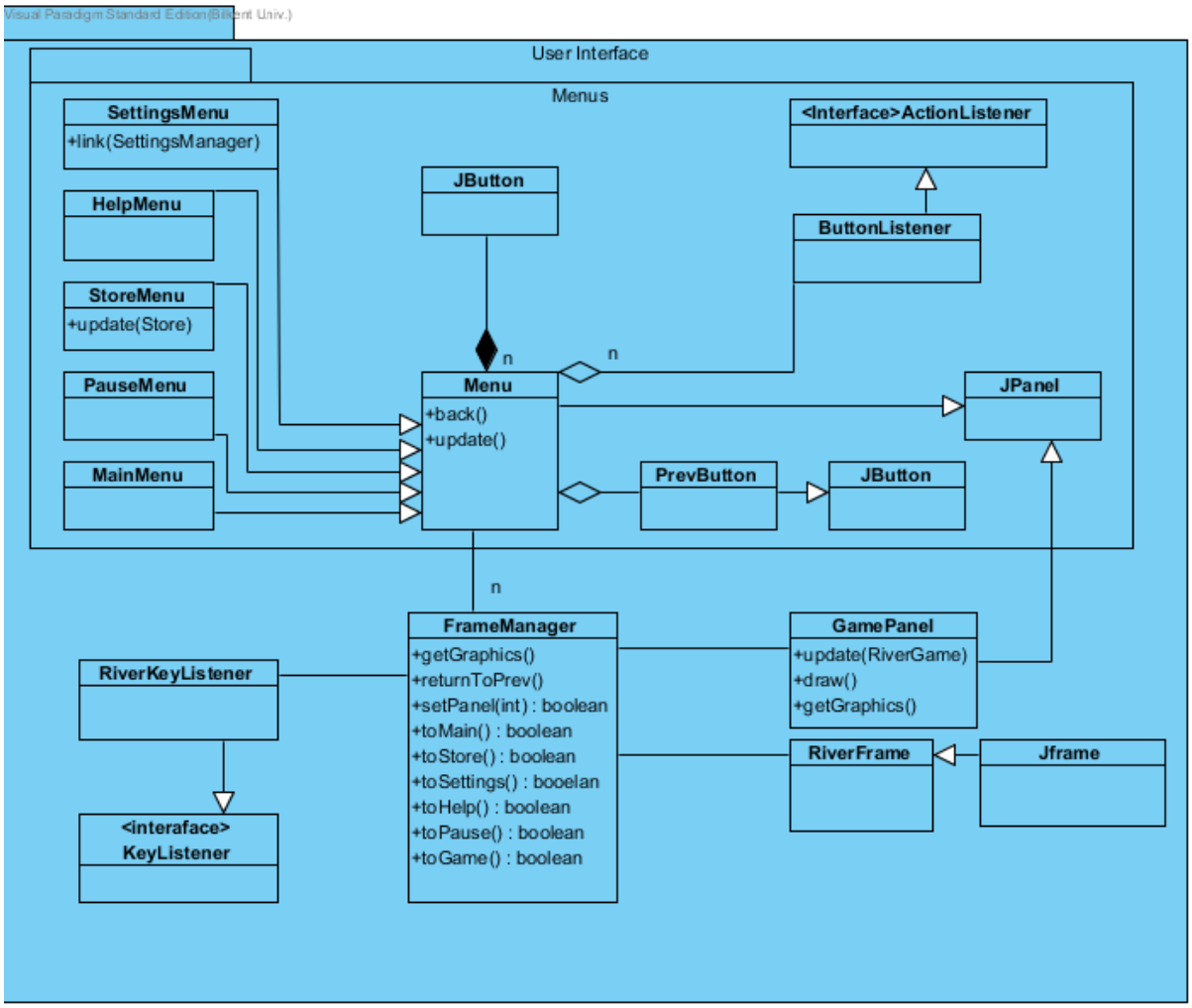

Figure 4. User Interface Subsystem Details

3.3. Controllers Subsystem

This subsystem handles the overall game state, and provides some operations to connect UI with persistent data/ models.

Game Manager:

Sustains communication with both UI subsystem and Models subsystem. Holds game state at the property 'gameState : int'. Holds a reference to an Account object, writes

onto it or receives its data. Starts initialization/destruction sequence of game models. Receives user input through UI subsystem, decides whether to start an operation or not in response to that. For example it can decide to fail a command such as pauseGame() that has been called by UI depending on the game state(granted, at the moment it should not be possible for UI to call pauseGame() at an unfavorable time). This class also calls operations on relevant models to act according to user input.

Settings Manager:

Controls the boundaries of the View Settings use case. Game Manager calls UI subsystem to link this controller with SettingsMenu UI class. Holds a string pointing to a text file which acts as a data storage.

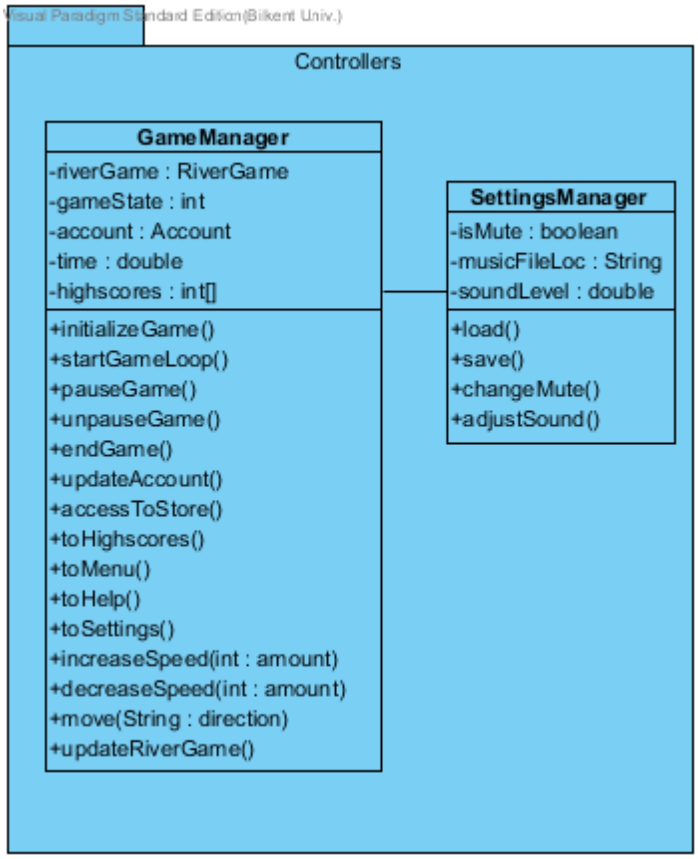

Figure 5. Controllers Subsystem Details

3.4. Models Subsystem

This subsystem contains a set of classes which are essential ontologically to the game logic.

Game Models:

Sustains communication with both UI subsystem and Models subsystem. Holds game state at the property ‘gameState : int’. Holds a reference to an Account object, writes onto it or receives its data. Starts initialization/destruction sequence of game models. Receives user input through UI subsystem, decides whether to start an operation or not in response to that. For example it can decide to fail a command such as pauseGame() that has been called by UI depending on the game state(granted, at the moment it should not be

possible for UI to call pauseGame() at an unfavorable time). This class also calls operations on relevant models to act according to user input.

- **Meta Models:** Contains a set of models, that are collection of other models as an essential attribute. Each of those define a set, which includes relation between model elements relation with each other.
    - **River Game:** Instantiated by Controller subsystem. Key Meta Model class, contains an interface for Controllers subsystem's benefit. All other model classes can be reached from here. Counts time, and informs lower models of calls from Controller subsystem.
    - **River:** Universal set for all visible game objects(like, traps boots etc.). Holds multiple Line classes, and generates/ moves/ destroys them in time. Note that this action is static and there is no way for any other class to modify this, other than the time property contained in RiverGame, or pause() call made on the RiverGame, which will be called by RiverController, which in turn stops "stream : Timer" in a RiverGame object. 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.
    - **Store:** Contains a set of visible not-ingame objects which all share the subsystem 'Buyable'. Constructed via the data from Accounts subsystem which has a persistent location to store the data.
    - **Line:** A non-visible in-game object, carries a RiverObject on an index 0 to 14. Rest of the indexes would be empty. Acts as an relational interface to make the game logic more simple. yLoc is the y-axis location of all objects contained in it, when yLoc in the Line changes all objects contained by the Line will also have the same new y-axis location.
- **Buyable:** Contains two classes that will generate objects which are to be contained in the store.Both of them contains price.
    - **Character:** Class that will specify 5 different character types to be purchased through its property name.
    - **Exclusives:** Purchasable and improvable boost types.
        - **ExclusiveBoost:** Also implements Boost abstract class. Specifies increase in effects affecting userCharacters duration.
- **RealModels:** Most ontologically obvious objects, all of those will 'swim' in the river.
    - **RiverObject:** Abstract class of all object swimming in the river. xLoc is correlated to index of a RiverObjects in an Line object. ySize is the vertical length of the object.
    - **Trap:** River objects that has malevolent effects.
    - **Obstacle:** River objects that deal damage to HP of the userCharacter.
    - **Collectable:** Stores the coin amount that will be gained in the property 'amount : int'.
    - **Boost:** Beneficial effect inducing objects.
        - **RareBoost:** Boost that turns objects onto collectables, has its own rarity modifier that changes its likeliness to be generated by a River at a Line.

        - **CharacterSpecificBoost:** Effect inducing boosts that came to be if their correlated character is being equipped, stores its type on the property ‘source : int’

Account: This subsystem acts as an interface between non-settings persistent data and the game logic.

- **Account:** This class holds address of the persistent text file as a string. After the load() operation those data stored in the variables of the class. CharState holds bought-not bought-equipped characters specified by the index. BoostStates likewise holds bought-not bought-upgraded boosts specified by the index.It stores total number of coins in the CoinWallet.
- **CoinWallet:** Holds coins, and supports add-remove coin operations.

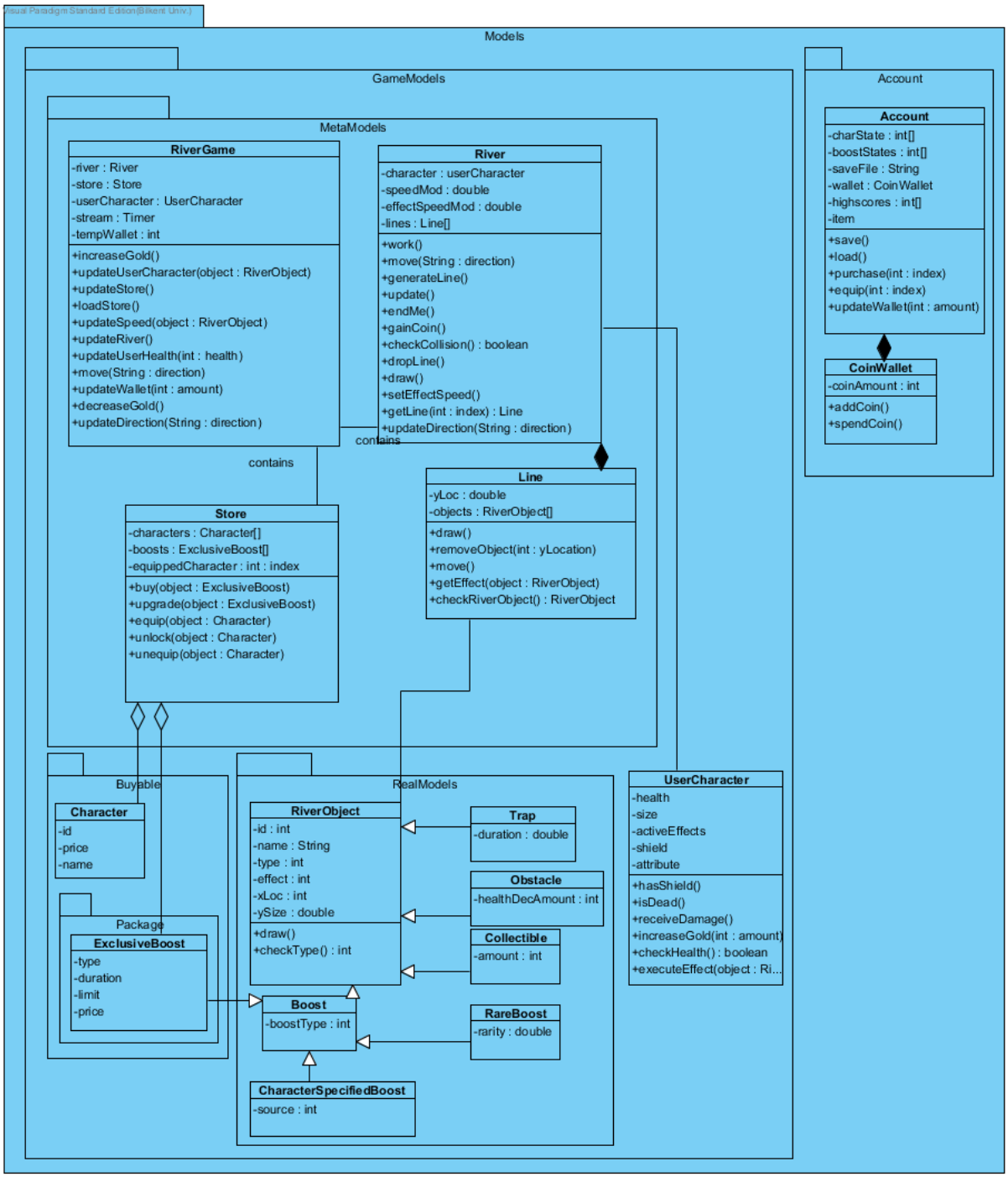

Figure 6. Models Subsystem Details