Bilkent University

Department of Computer Engineering

CS 319 Object - Oriented Software Engineering Project

Mafia:TCoS - Mafia:TheCity of Sin

Design Report - Low Level Design

Project Group 2.A:

Başak Melis Öcal, Gökcan Değirmenci, Gökberk Aktulay, Sinan Öndül

Supervisor & Course Instructor: **Prof. Dr. Uğur Doğrusöz**

Submitted at April 8, 2017

Contents

1. Introduction

1.1 Purpose of the System

Mafia:TCoS is a system appealing to the users who want to play an easy, entertaining, text based Role Playing Game with its simple structure and versatile gameplay. Compared to the alternative text based RPG's, Mafia has more intuitive gameplay and different game mechanics. The game has senseful learning-curve for users who want to play a challenging strategy game without taking too much time to learn and master. The game will be distinguishable from alternatives with its map system and unique submenus. Mafia requires to think strategically and aims to improve users' strategic thinking.

1.2 Design Goals

Before composing the system it is important for us to identify and overview the design goals of the system, in order to see what qualities/quantities that our system should focus on. Many of our design goals inherit from non-functional requirements that we have stated in the analysis stage of our project. Important design goals of our design for the system are stated and described below:

Usability
Our game should be user friendly, in order to be entertaining and easy to use. The players shouldn't be spending time getting to know how menus and submenus work. We will provide easily understandable interfaces for doing desired operations and navigating through the menus. We have decided that our game will get inputs from mouse and keyboard which will not be challenging for an average computer user.

Robustness

The program should be safe-guarded against invalid or malicious inputs. No matter how extraordinary the provided input was, our program should not segfault or gives the common NullPointerException. A good practice, is that each and every function/method would check its argument before executing inner logic. A solid exception handling structure and should be implemented.

Reliability

The program should be reliable, means that it produces expected results and behaviors every time when user interacts(even commercial software products do not guarantee that, so it is just our expectation and motivation to design our software). User should not be faces with errors all the time. For instance, if user wants to change his profile picture, he should be able to do it by clicking the "Change/Upload" button, and then the flow of events should be instantiated in background. Then user should be able to successfully upload/change his photo, as expected, without facing errors or different kind of outputs.

Security

The system should avoid malicious attacks and game resource hack programs which alters the user's assets, achievements, etc. So we decided not to store the user assets and user credentials on the Client side, rather we store them on the Cloud. Client only makes calculations on the gathered data from the server. Also, user passwords must be stored after encrypted and the whole connection between Client and the Server must be use secure line such as HTTPS protocol.

Extendibility

During our weekly meetings with the project group, there were many good ideas that would take too much of our time for a term project although would make our game superior. In general, a game should receive updates and new features in order to be attractive to the players. We came up with many ideas including an internet connection for multiplayer, more precise ratios for crimes, drug manufacturing with different variables and many more. Our game will be easily extendible in order to be easy to add new functionalities in the future. As the subsystems are designed in a way that only the part to be edit will be changed. Minimum coupling of our MVC pattern is also the main supporter of the extendibility goal.

Portability

Portability is important in order for the game to reach more players. We will implement the game on Java platform therefore it will be compatible with almost all of the platforms out there, thanks to the JVM. With creating a single .jar file, our game are gonna be reachable and executable on every platform that installed the JVM. Also, as an example, our game will be easily portable to handheld devices, if desired.

Traceability & Readability

The design of the system should not be over-complicated and its implementation should not has some kind of spaghetti code structure and therefore causes not understandable code. It should be easily understandable, readable by other developers, testers, supervisors and product managers. Every component and subsystem of the system should have traceable and meaningful implementation. This design goal highly improves the maintainability of the software and helps us to not being cursed by future maintainers or developers of the legacy codebase. While writing the code and designing the classes/methods, we need to keep in mind that: one of the the future developers/maintainers of our software would be a serial-killer.

Modifiability

Our system will be constructed in order to be easily modifying the existing functionalities of the game. In order to do this, we will avoid coupling the subsystems as much as possible in order to prevent side effects of a desired change. We also try to separate the implementation of the backend(Server) from the frontend(Client). This will allow us to modify the backend without changing the frontend code or vice versa.

Transparency
User of the program should be able to interact with the software easily without knowing the background of the internal architecture.The change in internal parts of the software should not affect the user interface, therefore user should not be aware of about that particular internal changes. For this purpose, RESTful Service is used instead of a direct access to database. With the service provided, neither the client nor the server side will be affected from any code changes in the other end.

2. Software Architecture

2.1 Architectural Patterns

For the system design, Client/server architectural style with an embedded MVC pattern on the client side is chosen.

2.1.1 Client/Server

As the main architectural style prompts, the system is decomposed into two main subsystems: client and server. A communication between the main application and server is provided by sending HTTP, *PUT*,*GET,POST,DELETE* requests to RESTful Service. Requests for updating the information such as name, money and inventory update etc. are sent by client to service periodically aiming to provide continuous updates. The server retrieves account related information from the database and sends it back as a response to the request.

2.1.2 Model/View/Controller

MVC architectural style is chosen to organize the code in the client side. Front end of our software is divided into three main subsystems which are model, view and controller. With its basic features, model represents the behaviour and state of objects, view manages user interface and controller handles various functionalities of the system.

<u>Reasons for deciding on this architectural style:</u>

- MVC pattern provides the minimum coupling and maximum cohesion to our subsystems when classes are considered.
- Each subsystem is developed in such a way that allows them to remain almost unaffected from revisions/edits in the code of other subsystems. Thus, only the part to be edit will be changed.

- In terms of the practicality and time efficiency it provides during the development process, MVC pattern supports the extendibility goal of our software. Time efficiency also fits to Waterfall software development technique.

- MVC decomposition also reduces the complexity of our code.

- Minimum dependency of subsystems supports the reuse of logics across applications.

2.2 Subsystem Decomposition

Client

As mentioned before, client is the front end of our software that is in contact with the user via getting the user input with a customized user interface. Clients are responsible from sending HTTP(S) requests to the RESTful Service periodically intending to provide a continuous data flow. The reason behind choosing to interact with RESTful Service rather than directly the database is providing the independence of database management classes when a change occurs in the code of client.

2.2.1 Model Subsystem

Naturally, our Model Subsystem maintains the domain knowledge by representing the current state of the game. Generally entity objects related with crime types, their subclasses and behaviors are hold by this subsystem. Any change in the current state of the game will be transferred to the Controller and Network Subsystems via subscribe/notify protocol.

2.2.2 View Subsystem

Basically, View is the subsystem in which all the user interface of the application is managed. View is responsible from getting the input from the user for the client to make requests and apply necessary changes to the UI whenever a state change occurs in the Model. Classes for map management, menus, input management, sound management are all hold by this subsystem intending to provide a more interactive experience to the user.

2.2.3 Controller Subsystem

All of the management classes which are responsible from managing the sequence of interactions in the game logic are grouped under the Controller Subsystem. It handles all of the functionalities and calculations that affect the state of the Model Subsystem with the GameEngine class it involves.

2.2.4 Network Subsystem

Most of the client functionality will be managed from Network Subsystem. HTTP(S) requests from client to RESTful service will be sent via Network in pre-defined periods to the server. When the server retrieves the requested data from the database, it will be manipulated if requested and sent as a response to the request. The response coming from the REST service will be used as attributes after parsing for the state and behavior changes of the objects which are hold by Model Subsystem.

Server

Whenever a request is made by the client, RESTful service does authentication by interaction with the related database through the server(however we may use Realtime Database service such as Google Cloud Platform and then we may not need to query the database manually). Retrieved data is returned as a JSON response to client by the REST service. Server functionality is provided in a way that it runs on various networking environments and operating systems. As RESTful Service is used as a middleware, server will not be affected from any revisions in the client’s code. Thus, server can be distributed without critical changes in its code.

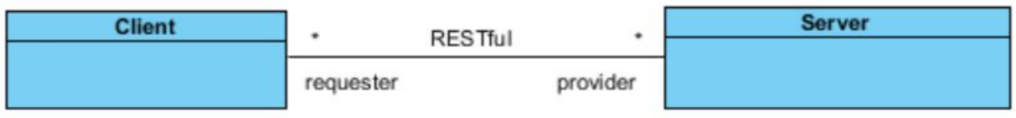


Figure 1. UML class diagram of client/server architecture: Features of the classes are left empty, as they will be decided during the low level design.

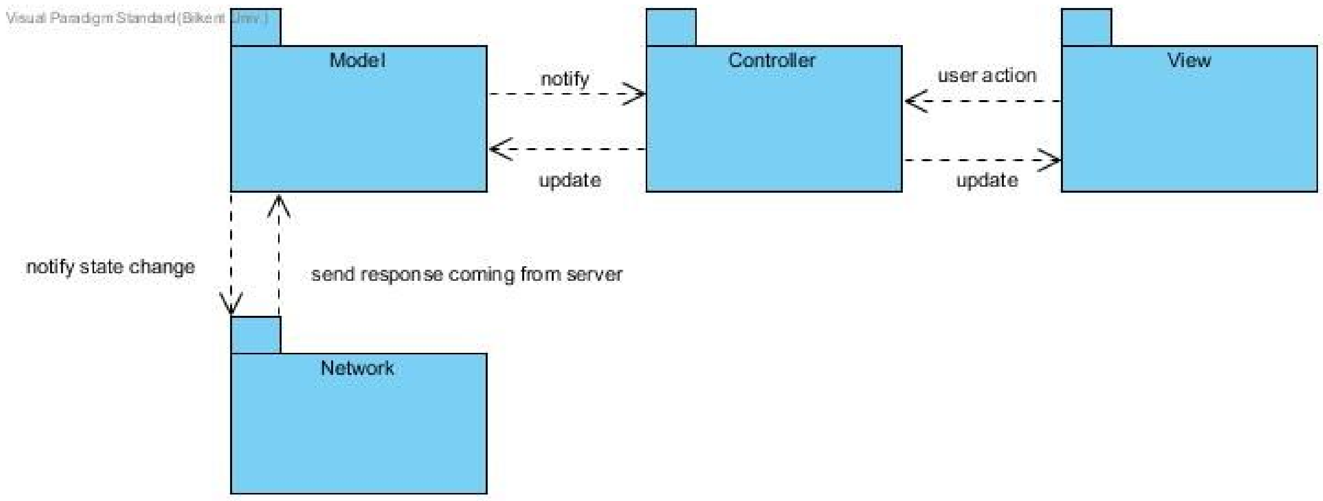


Figure 2. Subsystem decomposition diagram of front end(client): MVC architectural pattern is used with an extra network subsystem aiming to reduce the complexity of the database connection.

2.3 Hardware/Software Mapping

- Mafia: TCoS will be developed by using Java programming language and the latest development kit, which is JDK8. Playing the game requires a Java compiler in user's computers.
- Users require PC, monitor, mouse and keyboard for playing the game. Monitor which is connected to the PC will be used for displaying the game, mouse will be used for the main control of the game. Keyboard will be used during the login process for entering the credentials.
- Mafia: TCoS will use .wav for sound and .png for images. Thus, PC that will run the game should support .wav and .png.
- There should be fast and reliable Internet connection in the user's machine.
- Account information of the players, game's current state, assets they own, will be stored in a relational database such as MySQL or non-relational one like MongoDB. REST service will be used for the communication between client and database as a middleware. It will be implemented in either Spring Framework or Node.js.

2.4 Persistent Data Management

Our game is built on some core features such as real-signup-login, reliable game engine calculations and persistent game and user account state. So we need to store the data of the user and have great control on the user's game session. But while trying to accomplish those tasks: design of the system also need to be easy-to-implement and robust. Therefore, we decided to consume RESTful Service in our Client code. The Client code which is written in Java, tries to send requests over HTTP(S) to RESTful Service.

Then, RESTful Service will interact with the secured Database and manipulate the DB if necessary. Database includes detailed user account info, user's assets and the latest game state. Rather than Client interacts with the Database directly, we choose to position a Middleware, in this case REST Service, between the Client and the Database. The REST Service should be implemented in Spring Framework or Node.js and interacts with a relational database such as MySQL or non-relational database such as MongoDB. However, our team has got Plan B, if we encounter a chaotic development situation, we may choose a trending *Serverless* approach and consume REST APIs of the Realtime Database services such as *Google Cloud Platform* or *Firebase*: rather than writing the REST Service from scratch. https://firebase.google.com.

The Network Controller of the Client interacts with the RESTful Service by sending payloads containing HTTP *PUT*,*GET*,*POST*,*DELETE* requests. Then RESTful Service interacts with the Relational Database(MySQL or PostgreSQL) or NRDB: manipulates the specific data if needed and return JSON response to Client. Then, Client parses that JSON data and maps its values to corresponding Java objects.

The static resources of the game such as icons, sound clips, sound effects, background images should be stored in Client side not in the database. User uploaded profile image should be stored on some *Cloud Storage* service like *Amazon S3* and then the URI of that uploaded photo can be appended to POST or PUT payload when hitting the RESTful service.

2.5 Access Control and Security

Mafia:TCoS will use **HTTPS** network connection as a security procedure while sending/getting requests from the server. When creating a new account or signing into an existing one, we are planning to use Base64 encryption to store and retrieve user profile credentials, instead of using plain text.
We also use real kind of **authentication** to control the accesses over the game. The flow of authentication scheme will be as follows:

- The client will send user account credentials (username and password) to the server.
- The server will check the credentials and will create a token.
- The server will store the generated token in some storage along with an user id (uid).
- The server then sends the token to the client.
- The client stores that token and uses it when sending requests to the user.

For example:

https://somerestserver.com/$uid=hu3f92e/photo?auth=[TOKEN]

- In each request, the server uses token to perform authentication and authorization.
    - If token is valid, server accepts the request.
    - If token is invalid, server rejects the request.

2.6 Boundary Conditions

2.6.1 Initialization

Initialization of the game begins when the .jar file is double clicked. User shouldn’t have any problem with the initialization process if Java Runtime Environment exists in the PC. Main menu UI will be provided to the user when the game starts. User should sign up or login in order to play. The details of the UI will be designed and implemented considering the analysis report.

2.6.2 Termination

Leaving the game requires player to click ‘X’ button on the left corner of the screen which will result in the termination of the game. User shouldn’t have errors related with termination. As the game has the auto-save option which is provided by updating the database periodically, current state of the game, player’s account information etc. will be updated in the db before the termination.

2.6.3 Failure

As Mafia:TcoS will be implemented in Java, game can not operate without a Java Runtime Environment. Incorrect credentials during the login process or entering the existing credentials during sign up prevent user to play the game until the correct/available credentials are entered. When the connection between the server and client fails, system tries to reconnect. Loss of internet connection also results in reconnection trial.

3. Subsystem Services

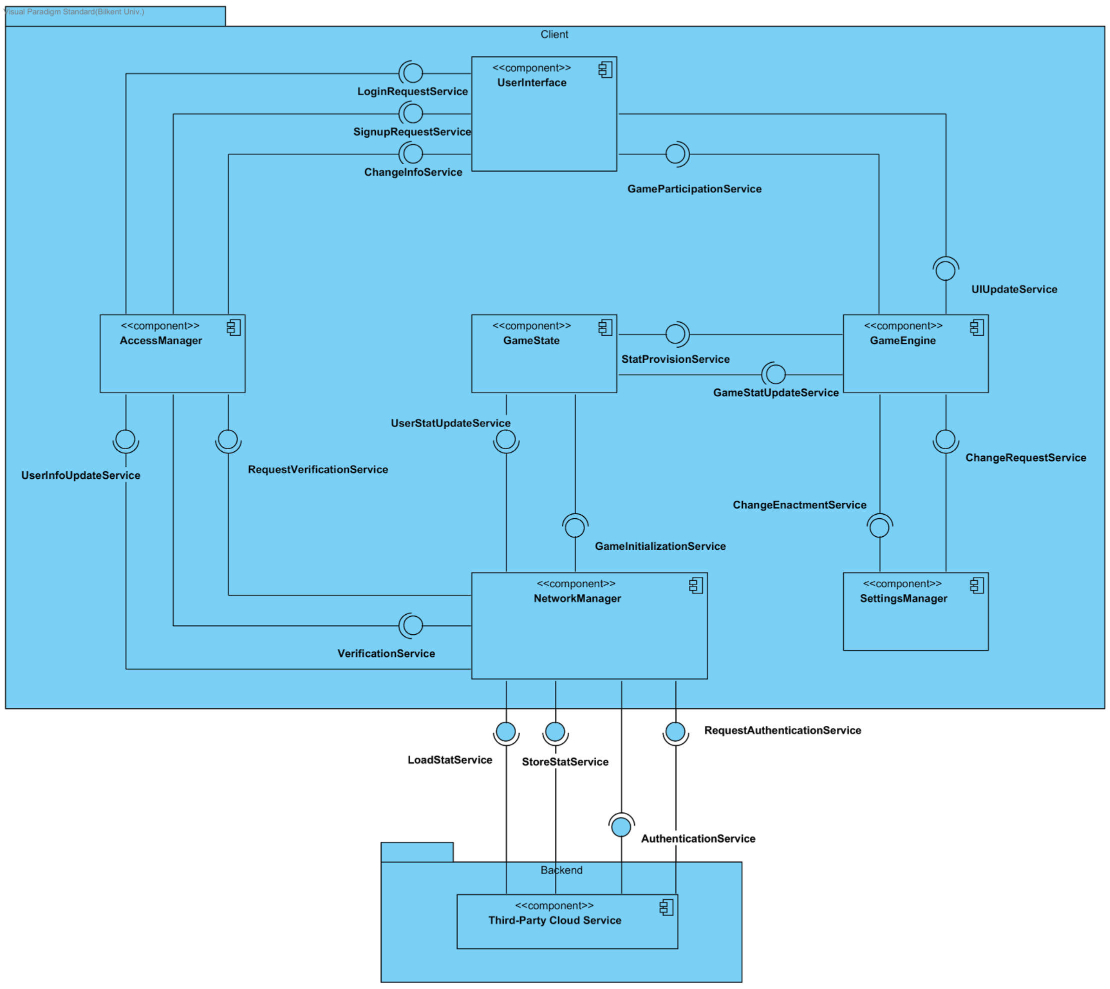


Figure 3. Subsystem Services Diagram

As mentioned before, the proposed architecture style is client/server for the software in which MVC pattern is embedded to the front-end. Naturally, View is responsible from any aspect of the UI, model maintains the domain knowledge, controller manages the main logic and network is responsible from the communication with the server. The services they provide to other subsystems which are shown in the UML component diagram are explained as follows:

3.1 UserInterface subsystem services:

LoginRequestService: Sends AccessManager login information and requests it to make a login attempt.

SignupRequestService: Sends AccessManager signup information and requests it to create an account.

ChangeInfoService: Asks the AccessManager to change user information with the new input.

GameParticipationService: Relays actions to GameEngine.

3.2 AccessManager:

RequestVerificationService: Sends NetworkManager user information and asks for verification.

UserUpdateInfo: Sends NetworkManager user information and asks that information to replace existing one or is added to a fresh account.

3.3 NetworkManager:

VerificationService: Tells AccessManager whether user credentials are correct.

LoadStatService: Retrieves stored data from the Third-Party Cloud Service.

StoreStatService: Stores data to the Third-Party Cloud Service.

Request Authentication: Sends the user credentials to the Third-Party Cloud Service and asks it to authenticate its server-side.

GameInitializationService: Sends GameState data necessary to initialize the game.

3.4 GameState:

UserStatUpdateService: Sends NetworkManager user-specific in-game resources and asks for old data to be updated.

StatProvisionService: Provides GameEngine with player stats needed in calculations.

3.5 GameEngine:

GameStatUpdateService: Updates player stats in GameState.

UIUpdateService: Modifies the UI as needed.

ChangeRequestService: Passes change requests the SettingsManager.

3.6 SettingsManager:

ChangeEnactmentService: Passes approved change requests back to GameEngine and asks for appropriate implementations.

4. Low-level Design

4.1 Object Design Trade-Offs

- **Delivery time vs. Bug-free Software**

The project must catch the "Project Demo" deadline date which is 8-9 May 2017. So, we have roughly 1 month to come up with a functional "Minimum Valuable Product (MVP)". This time period is really short to develop a high-quality desktop game which is bug-free and also functional. We have two choices: either we will deliver the game on time with known-bugs(includes serious bugs too) and publish patches later on or we will request more time to deliver the game with fewer bugs. As we have no chance to choose the second option and we also must rely on the somewhat old software development technique *Waterfall*; delivery-time will be our main priority instead of the bug-free software.

- **Delivery time vs. Features**

By choosing catching the delivery time over functionality, we decide to do whatever we are able to do in limited time. Some of the minor features of the game may not be implemented, we cannot say which ones yet. However, by choosing delivery time over features, our software tries to guarantee a robust, reliable, solid-working end product with its core features implemented.

- **Usability (User Experience) vs. Complexity**

User Experience will be our main concern. So we decided to reduce complex interactions and then integrate a smooth user experience. Rather than implementing immersive animations, glorious screen transitions and much more, we focus on more simple UI/UX core features and try to make them more intuitive and easy to use.

- **Portability vs. Speed Efficiency**

The project relies on Java and its frameworks, so basically it runs on JVM. We know that by choosing Java as a development language we omitted the fact that our code must first compile on JVM and then the JVM will translate that compiled code to machine code. Finally, machine code will be run on the CPU and RAM. That process took much longer time compared to non-JVM programming languages. But the advantageous part of that is, JVM runs on almost every OS and platform. So our game will be platform-independent and portable, but relatively slow. If we would use, for example, C++ to write our game and its engine, the code will be compiled and run directly on the CPU, without stopping by the JVM.

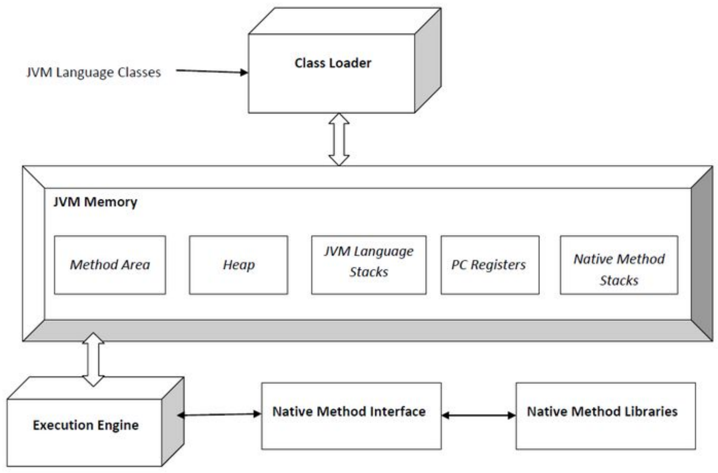


Figure 4. Overview of Java Virtual Machine [1]

4.2 Final Object Design

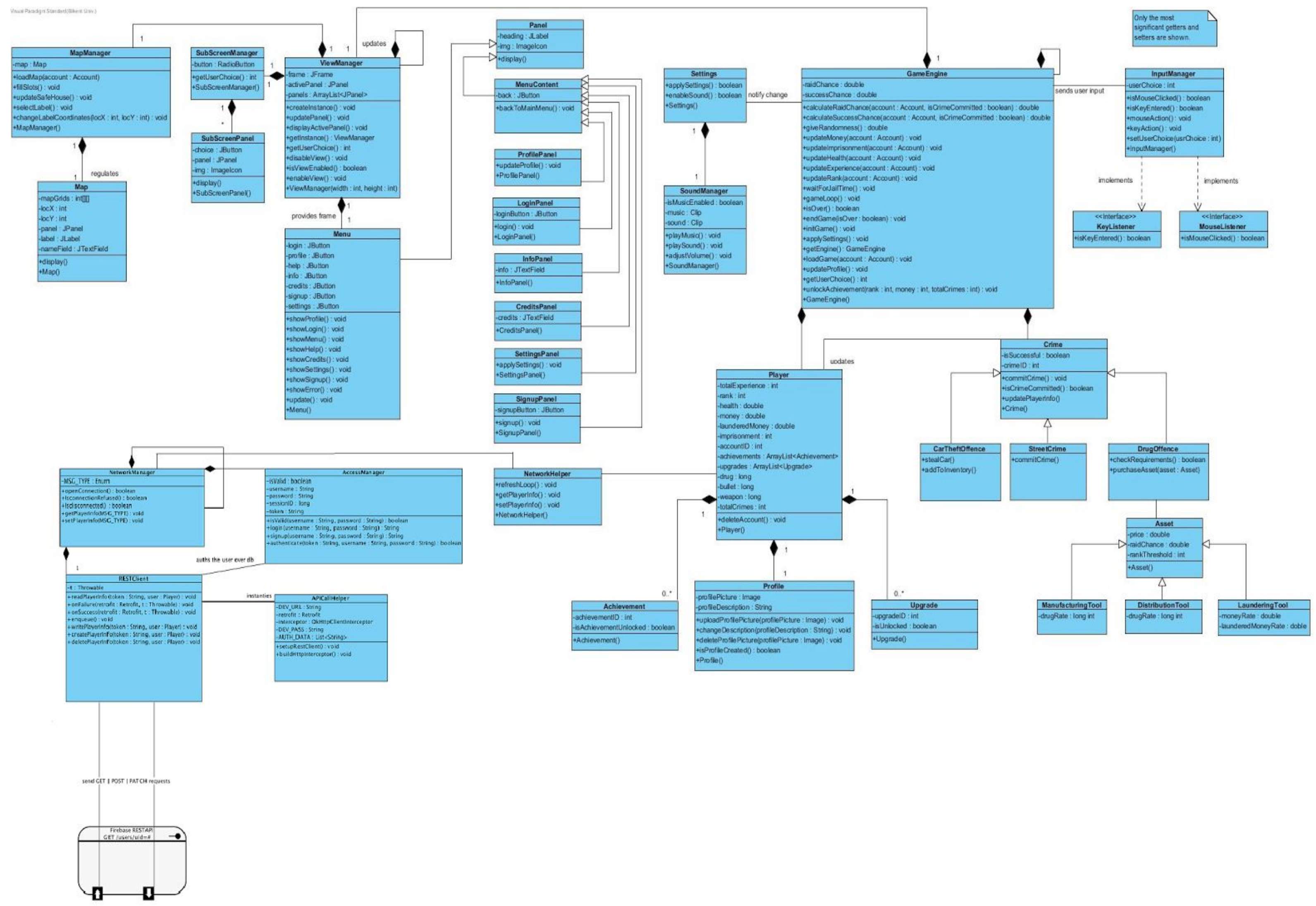


Figure 5. Final Object Design Diagram

4.2.1 Use of Inheritance

In our system, entity objects are mainly designed by using inheritance. CarTheftOffence, StreetCrime and DrugOffence have some common methods and attributes which involve commitCrime(), isSuccessful, crimeID etc. Thus, these classes are inherited from a superclass called Crime. Also, inheritance is used for classifying Asset class' objects. Specialized attributes and methods of the classes are demonstrated in each class.

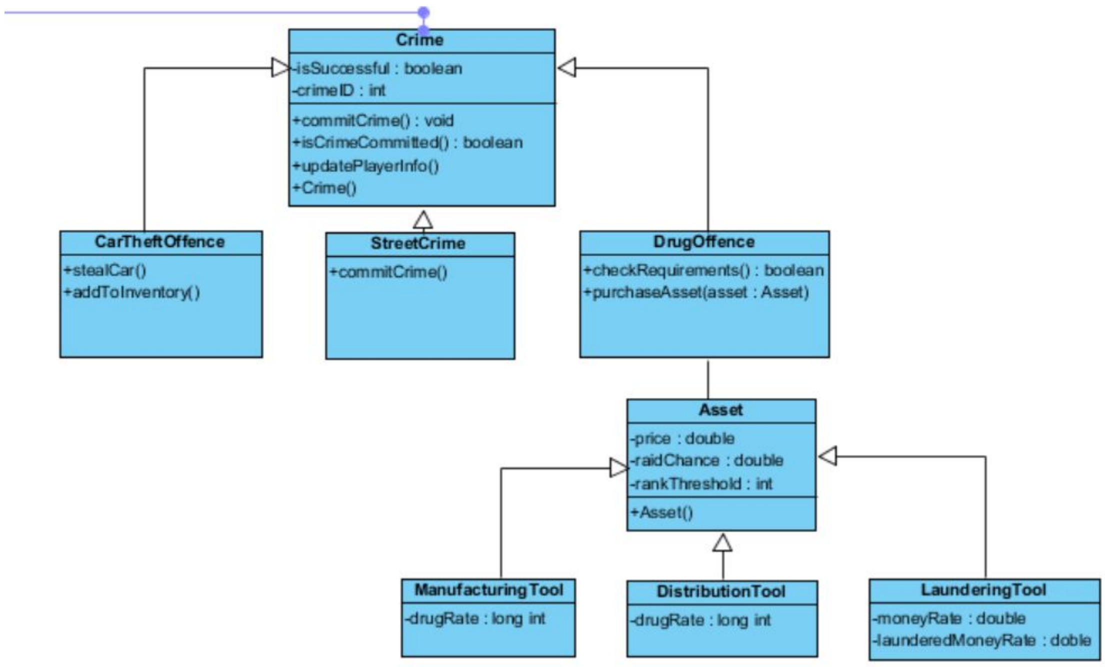


Figure 6. Diagram showcasing an example of inheritance

4.2.2 Information Hiding

Reliable systems requires the parts of the classes which shouldn't be accessed by the users to be kept private. If only the owner class needs to reach that attribute, it's kept private. If it's need to be reached by other classes attribute of the class is kept protected.

4.2.3 Facade Pattern

Our system's front-end is designed according to the Model - Controller - View architectural style as this style best fits with our design goals such as modifiability and extendibility. To further serve these purposes, we decided to support our design by using facade pattern. Each subsystem has its own facade class which acts as an interface which is responsible from the communication with other subsystems. A modification requires only the facade class to be changed as external subsystems only send data to the facade class. This utilization resulted in less coupling and provided an easy and convenient way for modifications and extensions. Our system has GameEngine, ViewManager, NetworkManager facade classes which represent Controller, View and Network subsystems consecutively.

4.2.4 Singleton Pattern

Since more than one instance of GameEngine, ViewManager and NetworkManager classes will result in a totally new process with its classes, singleton technique is used for preventing the duplication of them. Design returns a reference to the only GameEngine instance when it’s tried to be reached by a method.

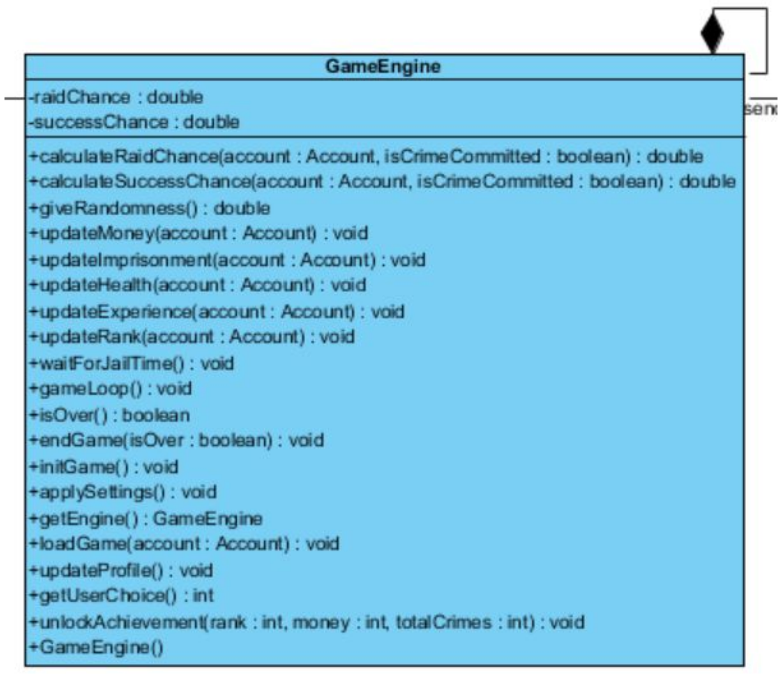

GameEngine
-raidChance : double
-successChance : double
+calculateRaidChance(account : Account, isCrimeCommitted : boolean) : double
+calculateSuccessChance(account : Account, isCrimeCommitted : boolean) : double
+giveRandomness() : double
+updateMoney(account : Account) : void
+updateImprisonment(account : Account) : void
+updateHealth(account : Account) : void
+updateExperience(account : Account) : void
+updateRank(account : Account) : void
+waitForJailTime() : void
+gameLoop() : void
+isOver() : boolean
+endGame(isOver : boolean) : void
+initGame() : void
+applySettings() : void
+getEngine() : GameEngine
+loadGame(account : Account) : void
+updateProfile() : void
+getUserChoice() : int
+unlockAchievement(rank : int, money : int, totalCrimes : int) : void
+GameEngine()

Figure 7. Game Engine Class

4.3 Packages

4.3.1 View Package

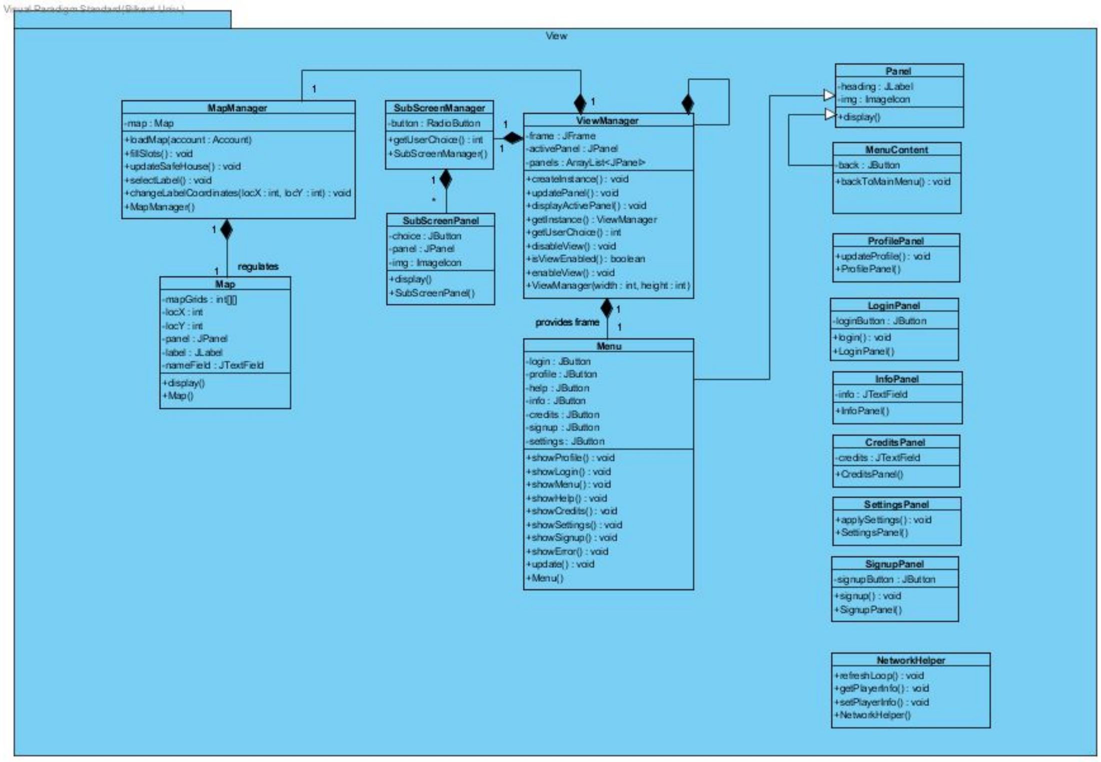
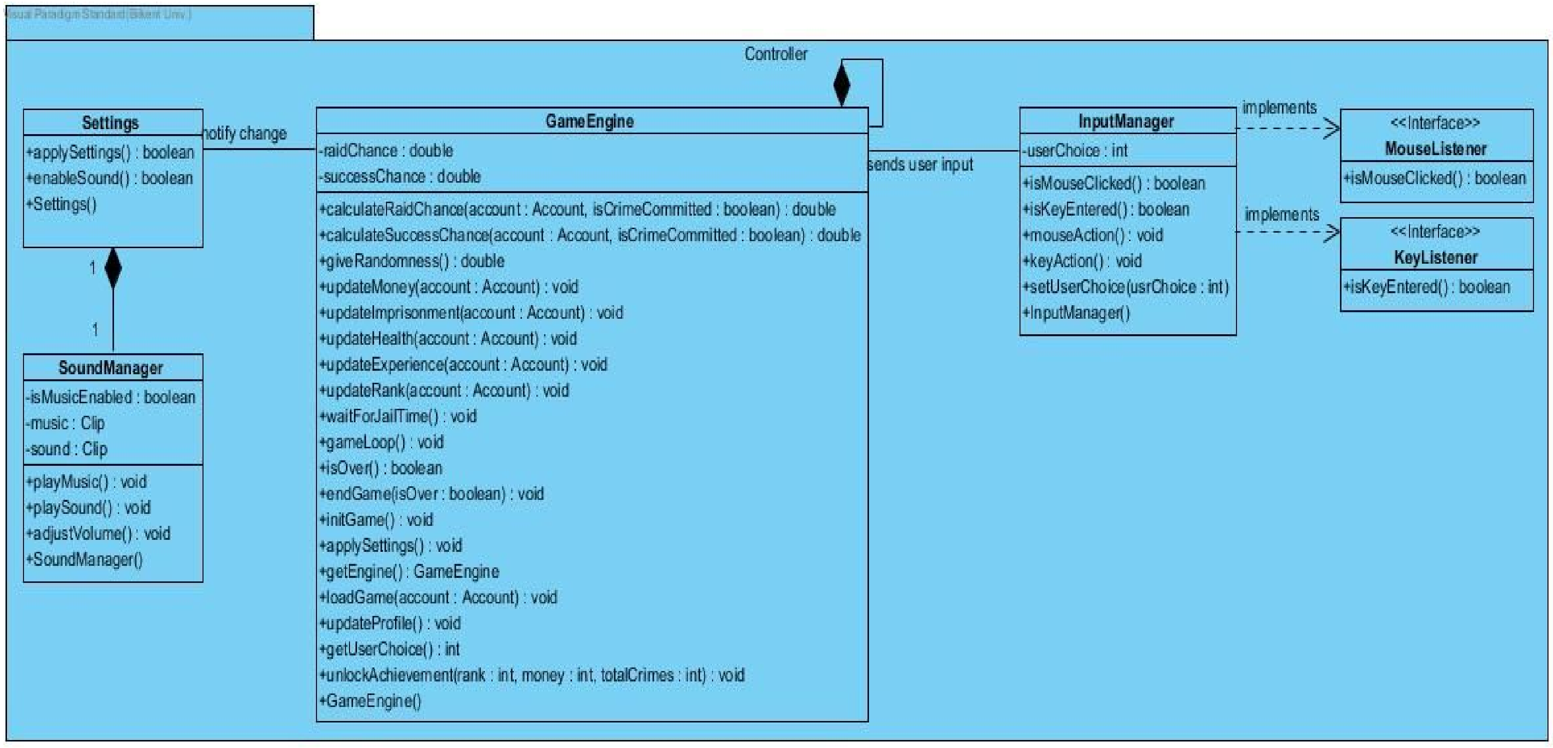


Figure 8. View Package Diagram

View Package contains UI elements and components that make up the view of the program. Various panels and a Menu class to traverse between them will be stored in this package. The Map class is used in drawing the grid with additional panels, the map upon which the actual game will be played. Choosing territories from the map will prompt the SubScreen panel. MapManager and SubScreenManager classes respectively connect Map and SubscreenPanel classes to the ViewManager, the facade class of the View Subsystem, which in turns connects the rest of the package to the Controller Package.

4.3.2 Controller Package

Controller package is responsible from the game logic and shelters GameEngine, InputManager, Settings, SoundManager classes as well as KeyListener and MouseListener interfaces. GameEngine does all of the calculations necessary for the continuity of the game. It interacts with the facade classes of the View and Model subsystems for making the necessary updates based on the state changes. Also, continuous checks are made in this class for the updates and ending the game. InputManager implements the KeyListener and MouseListener interfaces. It gets the inputs from the user and identifies them according to its context by checking the activePanel. Transfers the user choice to the GameEngine and also invokes necessary methods for generating responses to the user's actions. Settings is responsible from the game related adjustments such as sound. SoundManager plays the sounds/music if the isMusicEnabled option is kept 'on' by the user in the settings.



isual Paradigm Standard (Bilkent Univ.)
Controller
Settings
+applySettings() : boolean
+enableSound() : boolean
+Settings()
notify change
1
1
SoundManager
-isMusicEnabled : boolean
-music : Clip
-sound : Clip
+playMusic() : void
+playSound() : void
+adjustVolume() : void
+SoundManager()
GameEngine
-raidChance : double
-successChance : double
+calculateRaidChance(account : Account, isCrimeCommitted : boolean) : double
+calculateSuccessChance(account : Account, isCrimeCommitted : boolean) : double
+giveRandomness() : double
+updateMoney(account : Account) : void
+updateImprisonment(account : Account) : void
+updateHealth(account : Account) : void
+updateExperience(account : Account) : void
+updateRank(account : Account) : void
+waitForJailTime() : void
+gameLoop() : void
+isOver() : boolean
+endGame(isOver : boolean) : void
+initGame() : void
+applySettings() : void
+getEngine() : GameEngine
+loadGame(account : Account) : void
+updateProfile() : void
+getUserChoice() : int
+unlockAchievement(rank : int, money : int, totalCrimes : int) : void
+GameEngine()
sends user input
InputManager
-userChoice : int
+isMouseClicked() : boolean
+isKeyEntered() : boolean
+mouseAction() : void
+keyAction() : void
+setUserChoice(usrChoice : int)
+InputManager()
implements
implements
<<Interface>>
MouseListener
+isMouseClicked() : boolean
<<Interface>>
KeyListener
+isKeyEntered() : boolean

Figure 9. Controller Package Diagram

4.3.3 Model Package

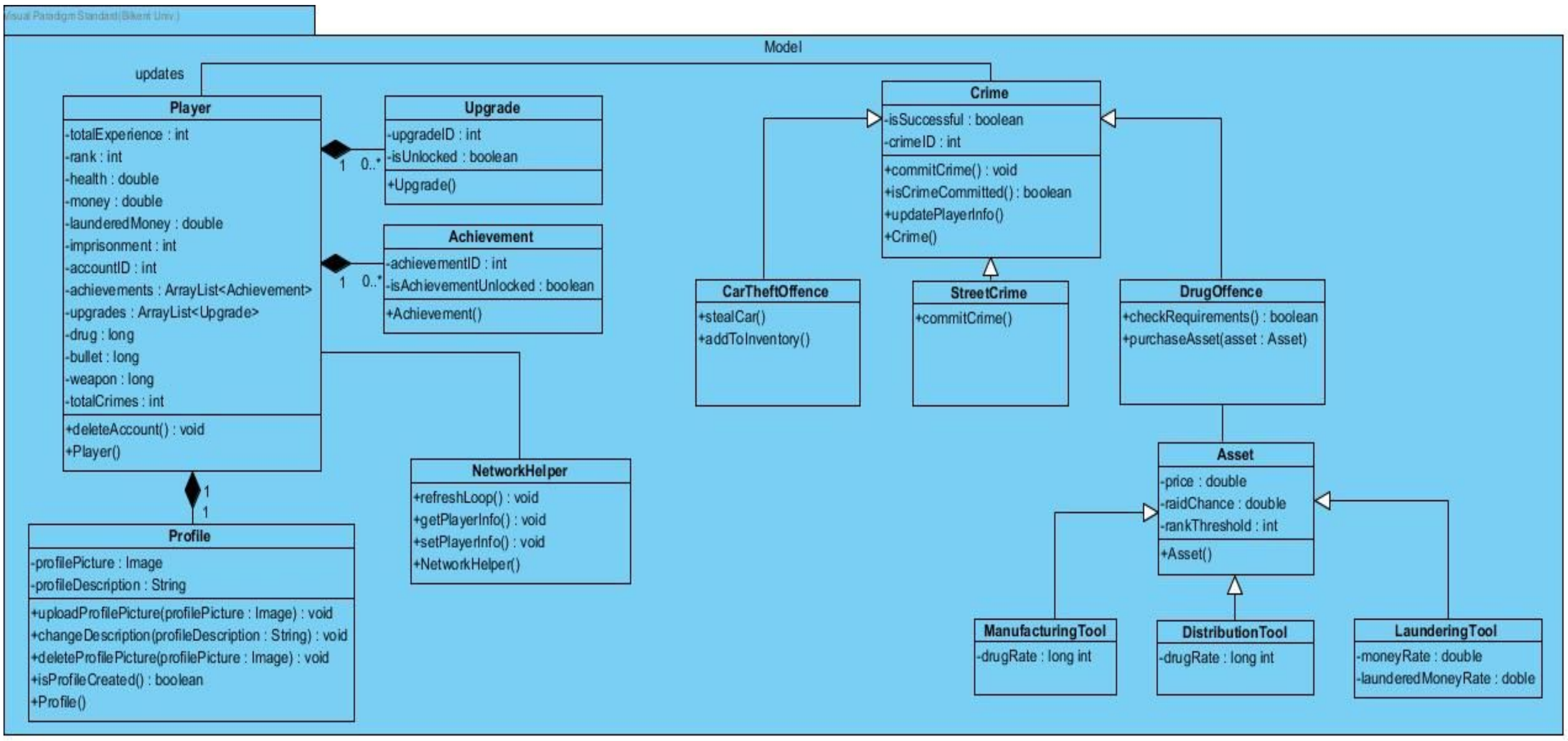


Figure 10. Model Package Diagram

Model class holds all the entity objects of the system. The package is composed of Player, Profile, Upgrade, Achievement classes for holding the account related information and Crime, CarTheftOffence, StreetCrime, DrugOffence and their subclasses for the functioning of the crime attempts. Player class keeps all the information related with the user account which can be sent to GameEngine from itself or Network from NetworkHelper. Profile provides customization to the user. Crime class is the other class which is interacting with the Controller subsystem via GameEngine.

Data to the database is sent by the NetworkHelper class which provides a loop for the continuous update of the database. This looper method invoke the Network’s Façade class in every 30 seconds to update/refresh Modal’s attributes.

4.3.4 Network Package

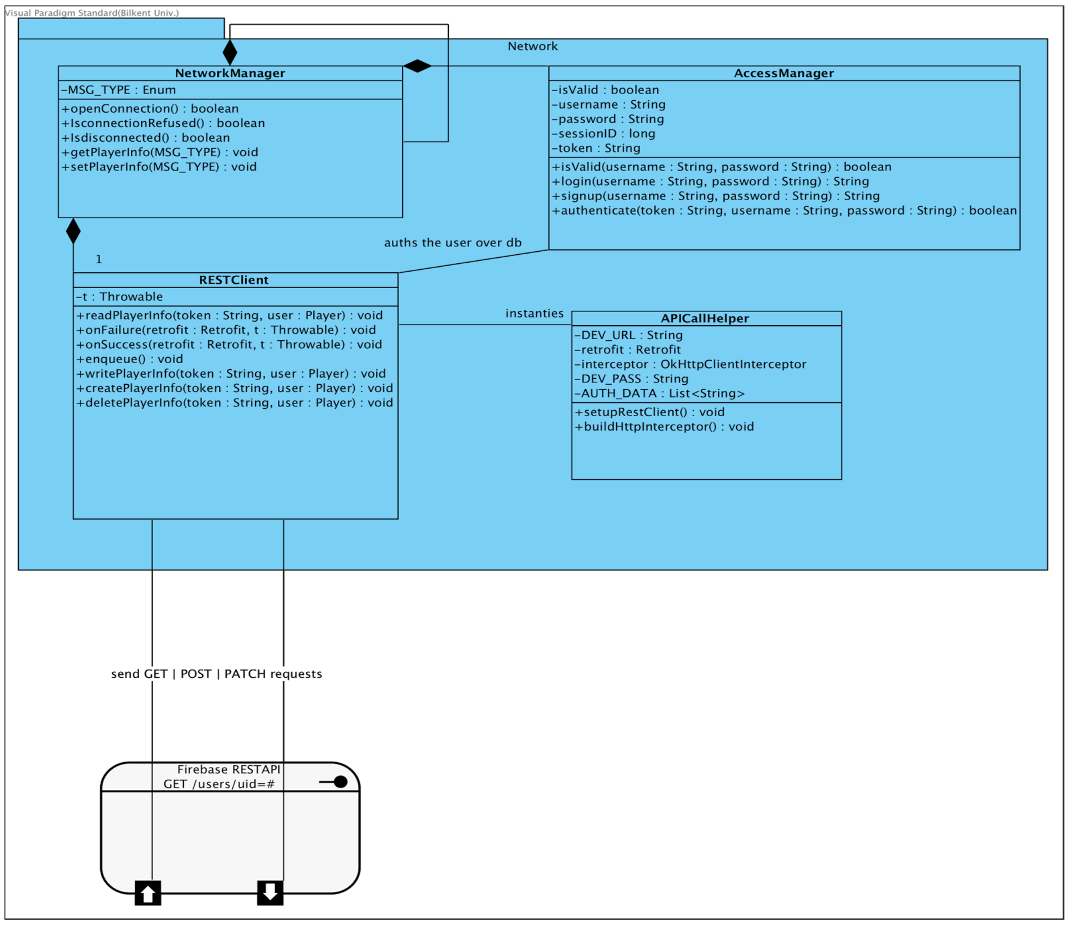

Visual Paradigm Standard(Bilkent Univ.)
Network
NetworkManager
-MSG_TYPE : Enum
+openConnection() : boolean
+IsconnectionRefused() : boolean
+Isdisconnected() : boolean
+getPlayerInfo(MSG_TYPE) : void
+setPlayerInfo(MSG_TYPE) : void
AccessManager
-isValid : boolean
-username : String
-password : String
-sessionID : long
-token : String
+isValid(username : String, password : String) : boolean
+login(username : String, password : String) : String
+signup(username : String, password : String) : String
+authenticate(token : String, username : String, password : String) : boolean
auths the user over db
1
RESTClient
-t : Throwable
+readPlayerInfo(token : String, user : Player) : void
+onFailure(retrofit : Retrofit, t : Throwable) : void
+onSuccess(retrofit : Retrofit, t : Throwable) : void
+enqueue() : void
+writePlayerInfo(token : String, user : Player) : void
+createPlayerInfo(token : String, user : Player) : void
+deletePlayerInfo(token : String, user : Player) : void
instanties
APICallHelper
-DEV_URL : String
-retrofit : Retrofit
-interceptor : OkHttpClientInterceptor
-DEV_PASS : String
-AUTH_DATA : List<String>
+setupRestClient() : void
+buildHttpInterceptor() : void
send GET | POST | PATCH requests
Firebase RESTAPI
GET /users/uid=#

Figure 11. Network Package Diagram

Network Package ultimately handles the core **persistent data** management functionalities of the game. Mafia:TCos has been designed to be playable from different computers, different connections and from different environments as long as you remember your login credentials and if your OS supports JVM. We use a Façade Design Pattern by using a Façade class on the entry point of our Network package. Our Façade and also Singleton Class which interacts with the Model layer is **NetworkManager**. In Network Package the **RESTClient** class handles sending HTTP(S) requests to Firebase REST API. **APICallHelper** class is a helper class as its name suggests, which handles the API routes and sets what kind of responses we want from the requests(like JSON or XML, but we choose JSON). **AccessManager** class helps to signup a user, login a user, create an authentication token and validate the user credentials along with the cross-checking the token's genuinity.

4.3.5 Server (REST Service)

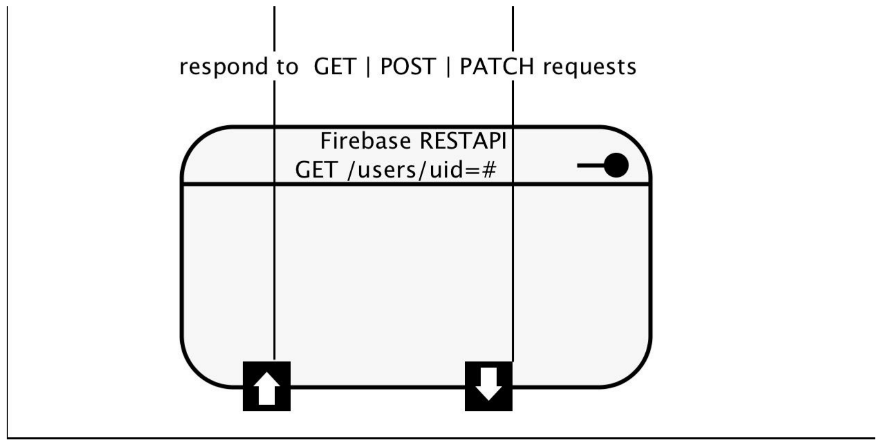
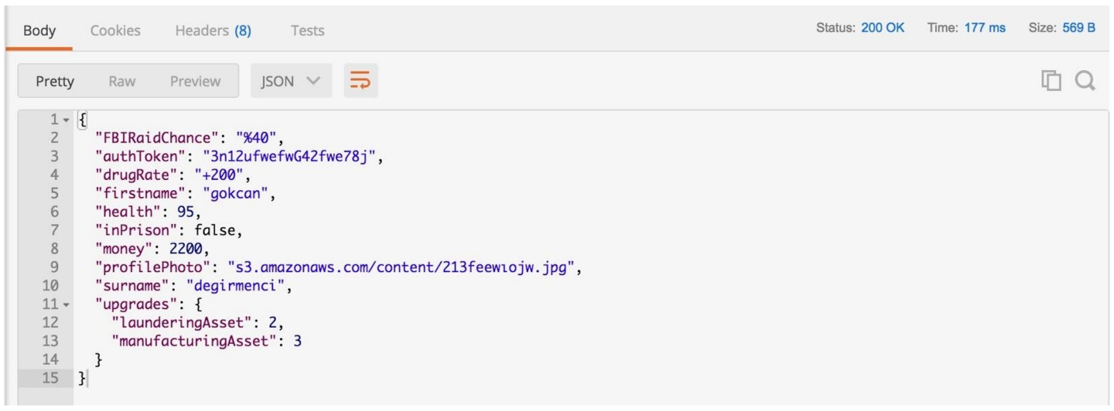


Figure 12.

We are using a Client-Server architecture on top of the game. Server-side of the architecture is basically a RESTful Web Service. We had two choices: rather we implement our very own REST Service from scratch with Spring Framework and MySQL Database or consume one of the popular 3rd party Cloud Service provider's REST APIs. As we have only 1 month to come up with a proper implementation and MVP version of the Mafia:TCoS, we have decided to go through with the second option. Hence, we will build the Realtime Database on the Google **Firebase** platform and consume its dedicated REST API. We create only one superior table which will be named "*Users*" on the Firebase **Realtime Database** and hit the API Endpoints from the Client-side of the game.



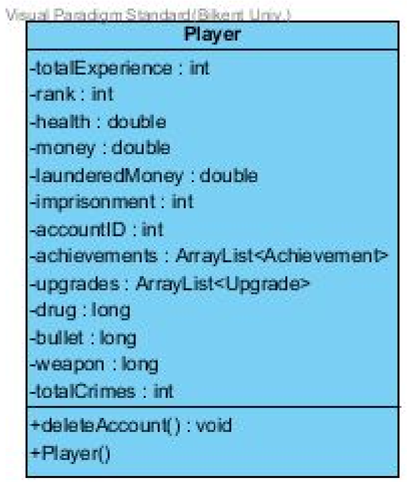
```
{
  "FBIRaidChance": "%40",
  "authToken": "3n12ufwefwG42fwe78j",
  "drugRate": "+200",
  "firstname": "gokcan",
  "health": 95,
  "inPrison": false,
  "money": 2200,
  "profilePhoto": "s3.amazonaws.com/content/213feewıojw.jpg",
  "surname": "degirmenci",
  "upgrades": {
    "launderingAsset": 2,
    "manufacturingAsset": 3
  }
}
```

Figure 13. *3* Early-prototype of JSON response obtained from the GET request to specific user endpoint.*

4.4 Class Interfaces

4.4.1 Model Classes:

Player:



<u>Constructor:</u>

public Player(): Creates an instance of a player.

<u>Attributes:</u>

private int totalExperience: The total experience a player has culminated.

private int rank: Player’s attained rank.

private double health: Player’s remaining health.

private double money: Player’s money vulnerable to be seized in a raid/audit.

private double launderedMoney: Player’s money that has been laundered and now legally safe. When making purchases, laundered money used after dirty money has been thoroughly expended.

private int imprisonment: The number of times the player has been incarcerated.

private int accountID: ID of the account to which the player is affiliated with.

private ArrayList<Achievement> achievements: A list of all the achievements the player has earned.

private ArrayList<Upgrade> upgrades: A list of all the upgrades the player has purchased.

private long drug: The quantity of opiates under the proprietary of the player.

private long bullet: The number of bullets at the player’s disposal.

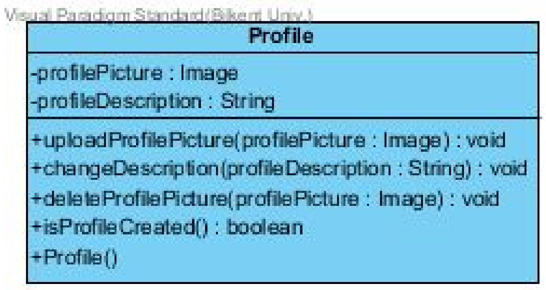
private long weapon: The number of bullets in the player’s arsenal.

private int totalCrimes: The number of crimes the player has committed so far.

<u>Methods:</u>

public void deleteAccount(): deletes the account related to the player.

Profile:


Visual Paradigm Standard(Bilkent Univ.)
Profile
-profilePicture : Image
-profileDescription : String
+uploadProfilePicture(profilePicture : Image) : void
+changeDescription(profileDescription : String) : void
+deleteProfilePicture(profilePicture : Image) : void
+isProfileCreated() : boolean
+Profile()

<u>Constructor:</u>

public Profile(): Creates a profile for the related Player.

<u>Attributes:</u>

private image profilePicture: The picture that is the avatar of this profile.

private String profileDescription: A description of the profile.

<u>Methods:</u>

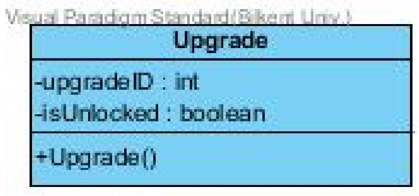
public void uploadProfilePicture(Image profilePicture): Adds a picture to the profile.

public void changeDescription(profileDescription : String): Changes description.

public void deleteProfilePicture(Image profilePicture): Deletes the profile picture.

public boolean isProfileCreated(): Queries whether the profile is created.

Upgrade:



<u>Constructor:</u>

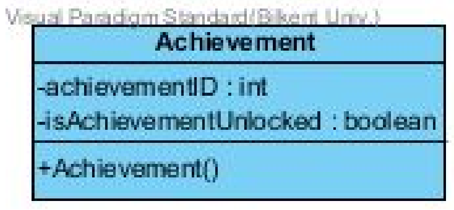
public Upgrade(): Instantiates an upgrade.

<u>Attributes:</u>

private int upgradeID: Numerical identification of the upgrade.

private boolean isUnlocked: Returns whether the upgrade is unlocked.

Achievement:


Visual Paradigm Standard(Bilkent Univ.)
Achievement
-achievementID : int
-isAchievementUnlocked : boolean
+Achievement()

<u>Constructor:</u>

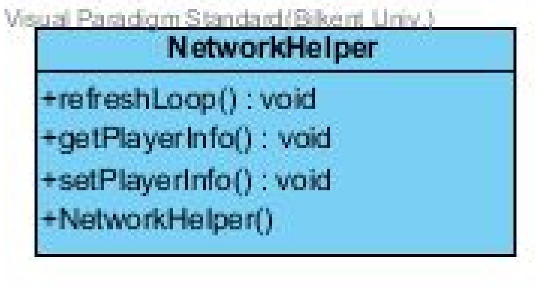
public Achievement(): Instantiates an achievement.

<u>Attributes:</u>

private int achievementID: Numerical identification of the achievement.

private boolean isAchievementUnlocked: Returns whether the achievement is unlocked.

NetworkHelper:



<u>Constructor:</u>

public NetworkHelper(): Instantiates a network helper.

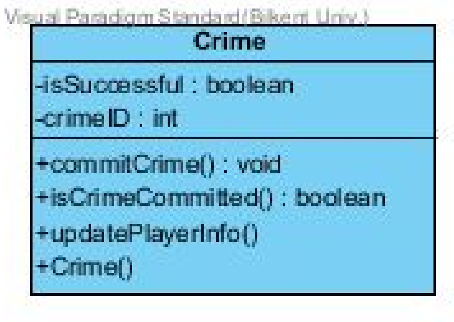
<u>Methods:</u>

public void refreshLoop(): In a continuous loop that loops in a predefined period, the method invokes getPlayerInfo() method and sends data to the network subsystem to update the database periodically.

public void getPlayerInfo(): Gets all of the player related information and assign them to new variables aiming to write them to database,

public void setPlayerInfo(): Sets player information by using the data coming from the database.

Crime:



Constructor:

public Crime(): Instantiates a crime.

Attributes:

private boolean isSuccessful: Whether the crime is successful.

private int crimeID: Numerical identification of the crime.

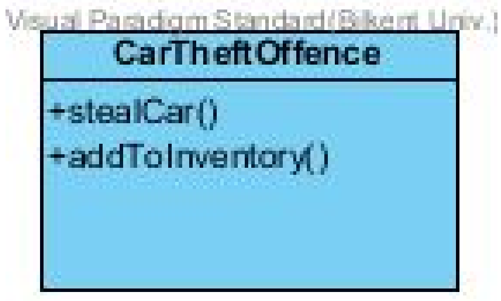
Methods:

public void commitCrime(): Attempts to commit the crime. This method is overwritten in CarTheftOffence and StreetCrime classes. Invokes update methods and waitForJailTime() method in the GameEngine depending on the result of the crime.

public boolean isCrimeCommitted(): Returns whether the crime is committed successfully.

public void updatePlayerInfo(): Updates player information such as money, health, rank etc. based on what was expended during the crime and what was earned.

CarTheftOffence:

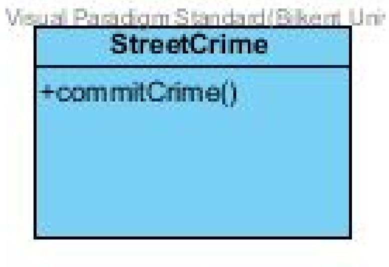


Methods:

public void stealCar(): Depending on the calculations made in GameEngine, a car will be stolen if it's successful.

public void addToInventory(): Stolen car will be added to the player’s inventory.

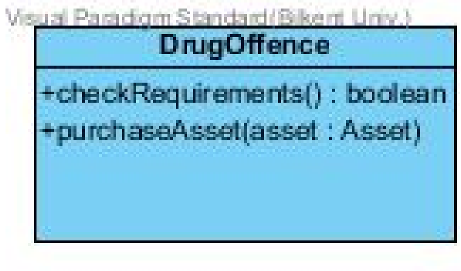
StreetCrime:



Methods:

public void commitCrime(): Street crimes are monetary crimes and will happen at the end of a figurative dice roll. giveRandomness() method of the GameEngine will manipulate the risk of the crime and result the crime.

DrugOffence:

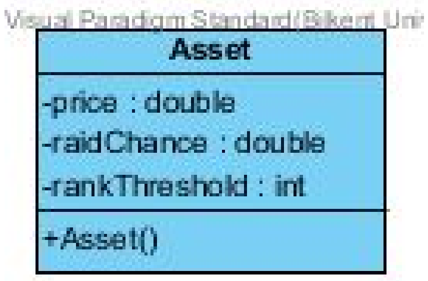

Visual Paradigm Standard(Bilkent Univ.)
DrugOffence
+checkRequirements() : boolean
+purchaseAsset(asset : Asset)

Methods:

public boolean checkRequirements(): Checks to see if the player is eligible to purchase the merchandise.

public void purchaseAsset(Asset asset): Player acquires sovereignty over the asset in exchange for money.

Asset:



Constructor:

public Asset(): Instantiates an asset.

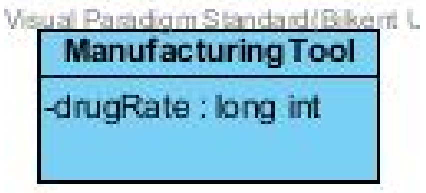
Attributes:

private double price: The price of the asset.

private double raidChance: Acquired assets may increase or decrease the raid chance.

private int rankTreshold: Different assets require certain minimum ranks before they can be purchased.

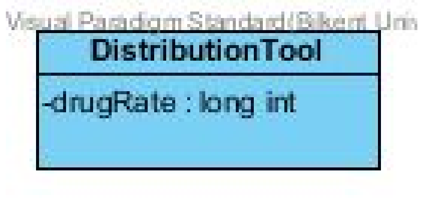
ManufacturingTool:



Attributes:

private long drugRate: Represents the drug amount produced in an hour. Manufacturing tools increase drug production in different rates.

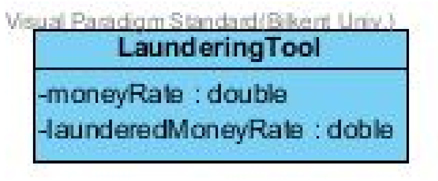
DistributionTool:



Attributes:

private long drugRate: Represents the drug amount sold in an hour. Has a negative value because distributed drug no longer belongs to the player.

LaunderingTool:


Visual Paradigm Standard(Bilkent Univ.)
LaunderingTool
-moneyRate : double
-launderedMoneyRate : doble

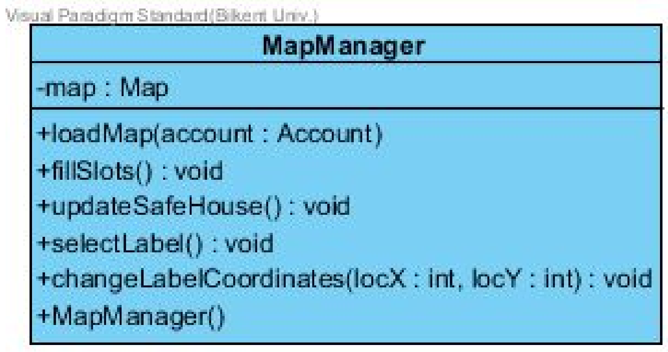
Attributes:

private double moneyRate: Has a negative value because laundered money is no longer considered as regular money.

private double launderedMoneyRate: Represents the launderedMoney earned in an hour.

4.4.2 View Classes:

MapManager:


Visual Paradigm Standard(Bilkent Univ.)
MapManager
-map : Map
+loadMap(account : Account)
+fillSlots() : void
+updateSafeHouse() : void
+selectLabel() : void
+changeLabelCoordinates(locX : int, locY : int) : void
+MapManager()

Constructor:

public MapManager(): Creates an instance of the MapManager.

Attributes:

private Map map(): Holds a map object.

Methods:

private loadMap(account:Account) : Loads the saved game's map of the related account.

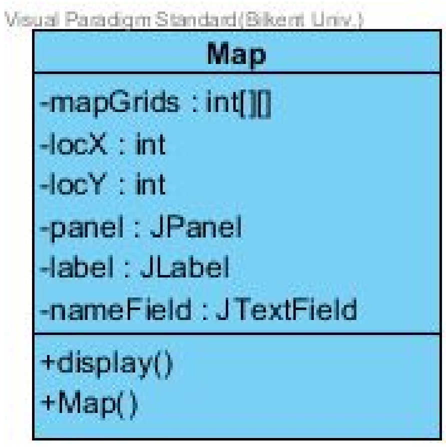
private void fillSlots(): Fills the slots of the map whenever an update occurs in the safehouse or changeLabelCoordinates() method is invoked.

private void updateSafeHouse(): Updates the safehouse whenever a an asset is purchased, a car is stolen, weapons/bullets are bought or an upgrade is purchased for any of them.

private void selectLabel(): Invoked whenever user clicks on the labels on the map. Invokes the method for demonstrating the related subscreen.

private void changeLabelCoordinates(locX: int, locY: int): In order to provide a dynamic map experience to the user, map label's places will be changed periodically in a loop.

Map:



<u>Constructor:</u>

public Map(): Creates an instance of the map.

<u>Attributes:</u>

private int[][] mapGrids: Each slot defined by a multiple array holds the type of the label as an integer. Different integers are assigned for street crime, drug, steal car, upgrades and sell labels.

private int locX: Represents the x-axis location of the grid.

private int locY: Represents the y-axis location of the grid.

private JPanel panel: Main panel of the map.

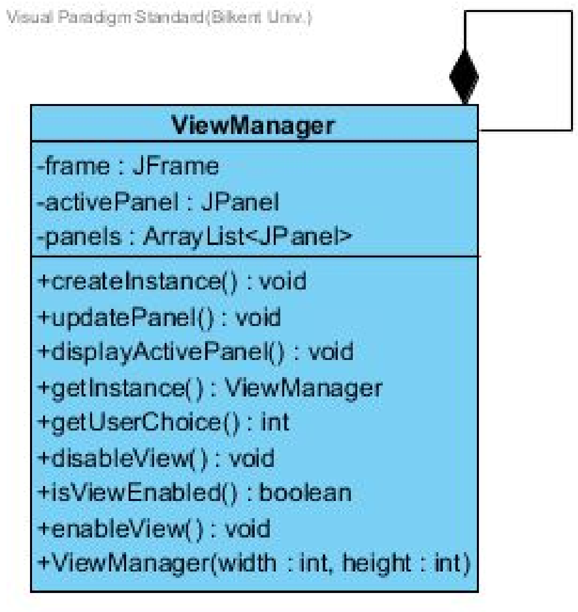
private JLabel label: Labels are used for the different submenu options on the map.

private JTextField nameField: Used for the title of the subscreen.

<u>Methods:</u>

public void display(): Demonstrates the map whenever invoked.

ViewManager:


Visual Paradigm Standard(Bilkent Univ.)
ViewManager
-frame : JFrame
-activePanel : JPanel
-panels : ArrayList<JPanel>
+createInstance() : void
+updatePanel() : void
+displayActivePanel() : void
+getInstance() : ViewManager
+getUserChoice() : int
+disableView() : void
+isViewEnabled() : boolean
+enableView() : void
+ViewManager(width : int, height : int)

<u>Constructor:</u>

public ViewManager(width : int, height : int): Creates an instance of the ViewManager. Provides the main frame in the size of the screen.

<u>Attributes:</u>

private JFrame frame: The main frame which is in the size of the screen.

private JPanel activePanel: It's the active panel which is displayed at that moment.

private ArrayList<Jpanel> panels: Stores the panel instances.

<u>Methods:</u>

private void updatePanel(): Invokes the related update methods of the panels depending on the context it's invoked by the

GameEngine.

private void displayActivePanel(): Demonstrates the currently active panel on the screen.

private void getInstance(): Returns an instance of the ViewManager.

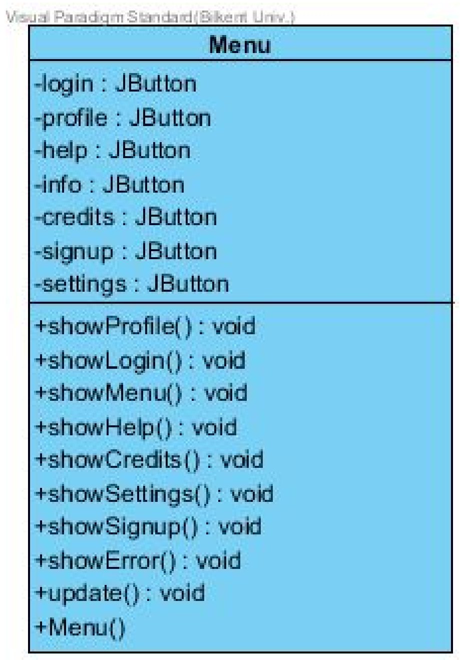
private int getUserChoice(): Gets the user choice from the GameEngine which is previously obtained by the InputManager.

private void disableView(): Disables the view when the waitForJailTime() method is invoked after an unsuccessful crime.

private boolean isViewEnabled(): Returns if the view is enabled as a boolean value after the jail time.

private void enableView(): Enables the view after the jail time.

Menu:



<ins>Constructor:</ins>

public Menu(): Creates an instance of the Menu.

Attributes:

private JButton login: Button for starting the authentication process.

private JButton profile: Button for switching to the profile panel.

private JButton help: Button for switching to the help panel.

private JButton info: Button for switching to the info panel.

private JButton credits: Button for switching to the credits panel.

private JButton signup: Button for switching to the signup panel.

private JButton settings: Button for switching to the settings panel.

Methods:

private void showProfile(): Invokes display() method of the ProfilePanel.

private void showLogin(): Invokes display() method of the LoginPanel.

private void showMenu(): Invokes display() method of the Menu.

private void showHelp(): Invokes display() method of the InfoPanel.

private void showCredits(): Invokes display() method of the CreditsPanel.

private void showSettings(): Invokes display() method of the SettingsPanel.

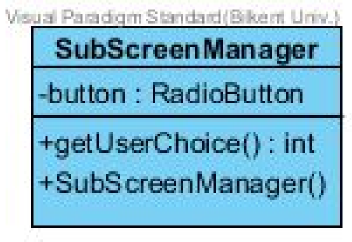
private void showSignUp(): Invokes display() method of the

SignupPanel.

private void showError(): Prints an error message to the screen whenever a panel can't be displayed.

private void update(): Invokes the updates methods of the editable MenuContent subclasses.

SubScreenManager Class:



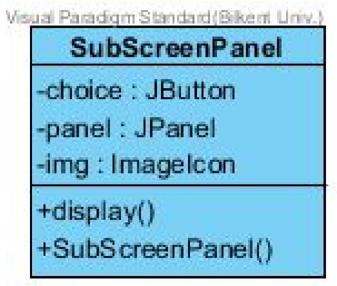
Constructor:

public SubScreenManager(): Creates an instance of the SubScreenManager.

Methods:

public void getUserChoice(): Gets the user choice from the active subscreen's JButtons.

SubScreenPanel Class:



Constructor:

public SubScreenPanel(): Creates an instance of the

SubScreenPanel.

Attributes:

private JButton choice: Represent choice.

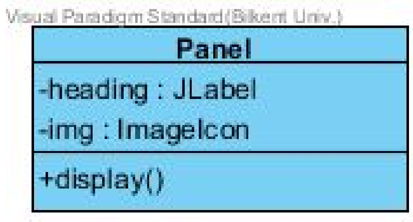
private JPanel panel: Main panel for the subscreen.

private ImageIcon img: Image of the related subscreen.

Methods:

private display(): Displays the subscreen whenever it's invoked.

Panel:



Attributes:

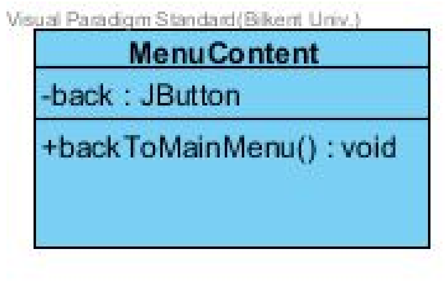
private JLabel heading: Provides heading for its subclasses.

private ImageIcon img: Provides image for its subclasses.

Methods:

private display(): Provides a display method for its subclasses. Displays the related panel.

MenuContent Class:



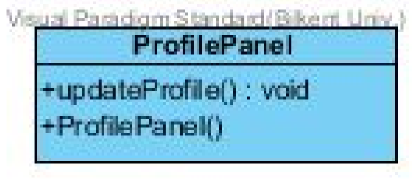
Attributes:

private JButton back: Button for navigating back to Main menu.

Methods:

private void backToMainMenu(): Method for navigating back to main menu.

ProfilePanel:



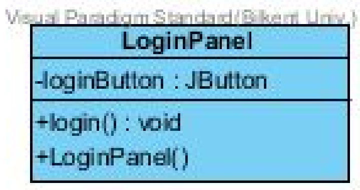
Constructor:

public ProfilePanel(): Creates an instance of the ProfilePanel.

Methods:

private void updateProfile(): A method for updating profile with new credentials.

LoginPanel:



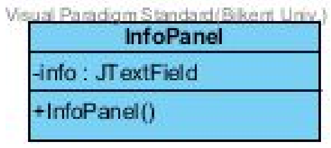
Constructor:

public LoginPanel(): Creates an instance of the LoginPanel.

Attributes:

private JButton button: Login button which is used after the credentials of the player is entered.

InfoPanel:



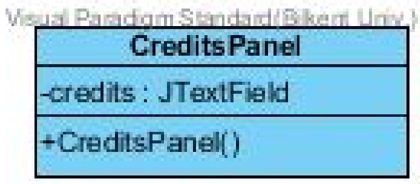
Constructor:

public InfoPanel(): Instantiates InfoPanel.

Attributes:

private JTextField info: Presents all of the information about the game as a text.

CreditsPanel:



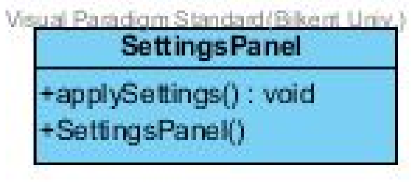
Constructor:

public CreditsPanel(): Instantiates CreditsPanel.

Attributes:

private JTextField credits: Presents the credits as a text.

SettingsPanel:



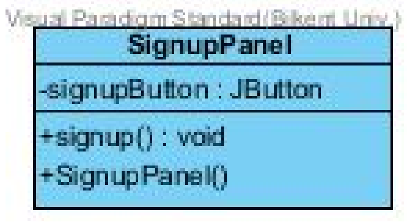
Constructor: Instantiates SettingsPanel.

SettingsPanel()

Methods:

private void applySettings(): Method for saving new settings.

SignUpPanel:



Constructor:

public SignUpPanel(): Constructor for creating a SignUpPanel instance.

Attributes:

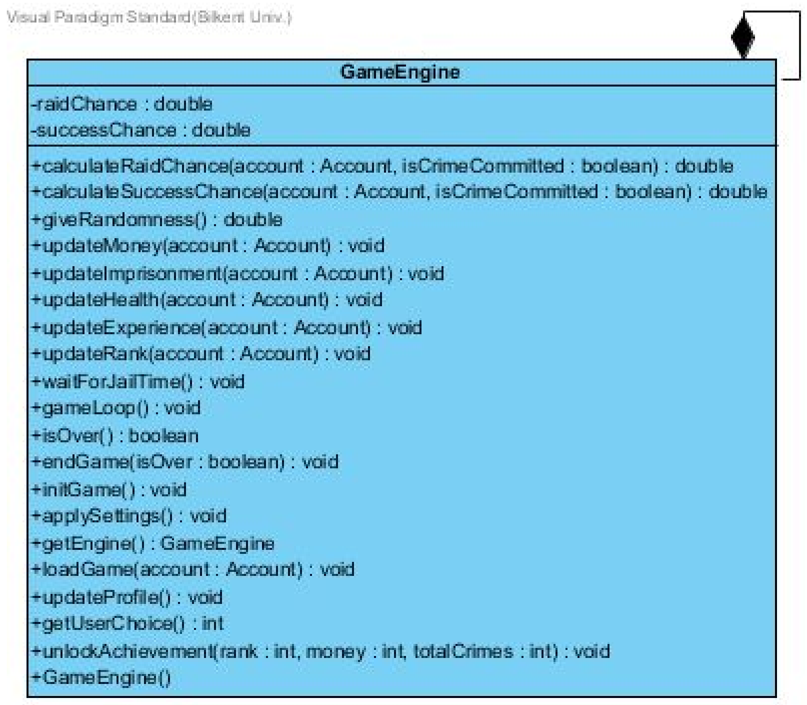
private JButton signupButton: Button for sign up.

<u>Methods:</u>

private void signup(): Method for showing the screen after the signup process.

4.4.3 Controller Classes:

GameEngine:



```
GameEngine
-raidChance : double
-successChance : double
+calculateRaidChance(account : Account, isCrimeCommitted : boolean) : double
+calculateSuccessChance(account : Account, isCrimeCommitted : boolean) : double
+giveRandomness() : double
+updateMoney(account : Account) : void
+updateImprisonment(account : Account) : void
+updateHealth(account : Account) : void
+updateExperience(account : Account) : void
+updateRank(account : Account) : void
+waitForJailTime() : void
+gameLoop() : void
+isOver() : boolean
+endGame(isOver : boolean) : void
+initGame() : void
+applySettings() : void
+getEngine() : GameEngine
+loadGame(account : Account) : void
+updateProfile() : void
+getUserChoice() : int
+unlockAchievement(rank : int, money : int, totalCrimes : int) : void
+GameEngine()
```

<u>Constructor:</u>

public GameEngine(): Instantiates the game engine.

<u>Attributes:</u>

private double raidChance: Probability of getting raided.

private double successChance: Probability of successfully completing the crime.

Methods:

public double calculateRaidChance(Account account, boolean isCrimeCommitted): Calculates the probability of getting raided. Calculations are based on laundered money, drug amount, drug distribution/manufacturing rates of the player. giveRandomness() method is invoked for varying percentage in a small range.

public double calculateSuccessChance(Account account, boolean isCrimeCommitted): Calculates the probability of successfully finishing the crime. Calculates the probability of getting raided. Calculations are based on the rank, health, experience and bullet amount of the player. giveRandomness() method is invoked for varying percentages between the crimes.

public double giveRandomness(): Method is invoked for providing changing percentages for successChance and raidChance. Also, method is invoked for resulting the crime attempt.

public void updateMoney(Account account): Updates player's money periodically and also after every successful crime attempt.

public void updateImprisonment(Account account): Increments imprisonments as the player goes to prison.

public void updateHealth(Account account): Updates player's health after every crime attempt.

public void updateExperience(Account account): Increments player's experience as crimes are being committed.

public void updateRank(Account account): Updates player's rank by considering the crimes committed, upgrades bought and achievements unlocked.

public void waitForJailTime(): If the crime failed, player needs to wait before taking further actions. Provides screen view to be disabled for a predefined period such as 30 sec.

public void gameLoop(): Provides the game to continue continuously in a loop.

public boolean isOver(): Returns true if all of the health of the player is used up or player’s imprisonment exceeded 20 times.

public void endGame(): Terminates the game after checking if the requirements are met for ending the game.

public void initGame(): Initializes the game.

public void applySettings(): Applies changes made to the settings.

public GameEngine getEngine(): Returns game engine.

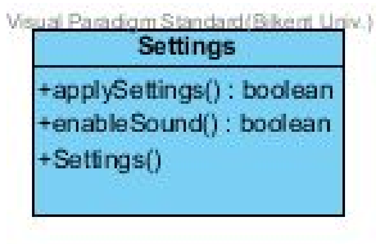
public void loadGame(Account account): Sets game state with saved values for the related account.

public void updateProfile(): Updates profile whenever user requests a change. This method is invoked after InputManager gets the necessary update.

public int getUserChoice(): Gets interpreted user input.

public void unlockAchievement(int rank, int money, int totalCrimes): Unlocks achievements when their prerequisites are met such as rank, experience, money etc.

Settings:


Visual Paradigm Standard(Bilkent Univ.)
Settings
+applySettings() : boolean
+enableSound() : boolean
+Settings()

<ins>Constructor:</ins>

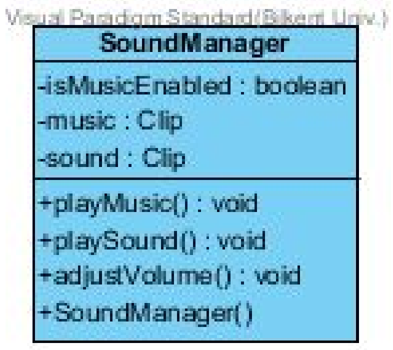
public Settings(): Instantiates settings.

Methods:

public boolean applySettings(): Applies changes.

public boolean enableSound(): Enables sound.

SoundManager:



Constructor:

public SoundManager():

Attributes:

private boolean isMusicEnabled: Holds the boolean value, if it is true SoundManager invokes the playMusic method.

private Clip music: Holds the music in a Clip object.

private Clip sound: Holds the sound effects in a Clip object.

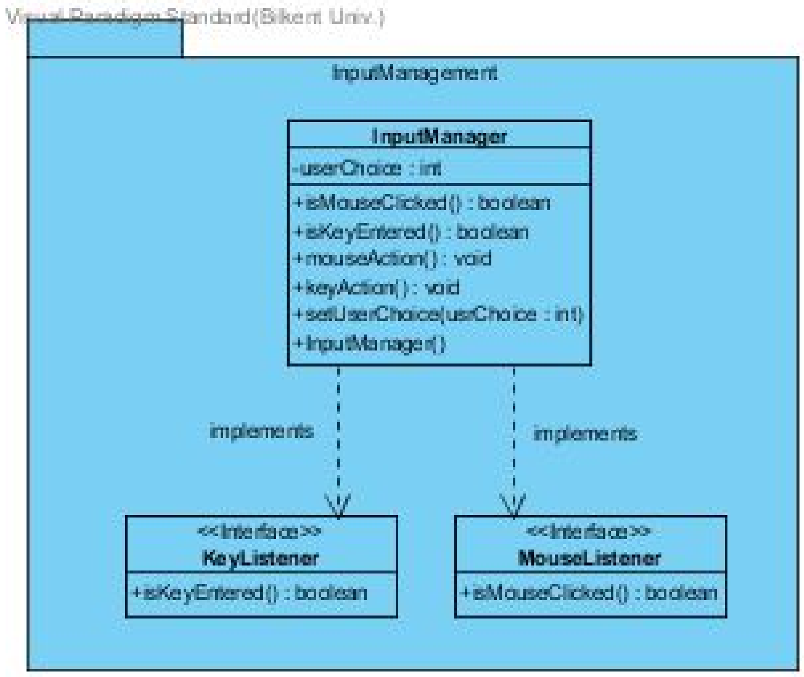
Methods:

public void playMusic(): Invokes the OS's utility functions and starts to play music.

public void playSound(): Invokes the OS's utility functions and starts to play sound effects.

public void adjustVolume(): Changed the volume of the music/sound according to the user input.

InputManager:


Visual Paradigm Standard(Bilkent Univ.)
InputManagement
InputManager
-userChoice : int
+isMouseClicked() : boolean
+isKeyEntered() : boolean
+mouseAction() : void
+keyAction() : void
+setUserChoice(usrChoice : int)
+InputManager()
implements
implements
<<Interface>>
KeyListener
+isKeyEntered() : boolean
<<Interface>>
MouseListener
+isMouseClicked() : boolean

<u>Constructor:</u>

public InputManager(): Creates an instance of the InputManager.

<u>Attributes:</u>

private int userChoice: User choice that's obtained after a click to the label or buttons.

<u>Methods:</u>

public boolean isMouseClicked(): Returns true if the mouse is clicked.

public boolean isKeyEntered(): Returns true if a key is entered.

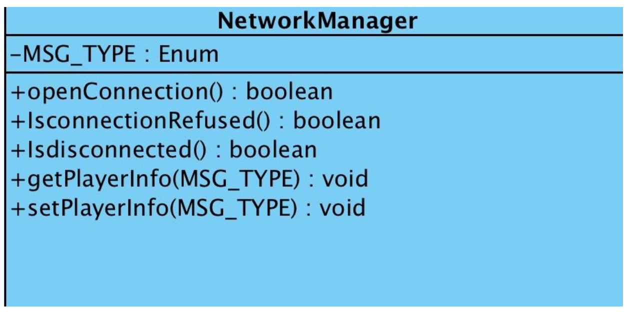
public void mouseAction(): Generates the necessary response by after a click is made by the user. After checking the activePanel, invokes the necessary methods depending on the context (activePane).

public void keyAction(): Generates the necessary response after a key is entered by the user. After checking the activePanel, invokes the necessary methods depending on the context (activePane).

public void setUserChoice(int userChoice): Sets the user choice by getting the input from the labels/buttons.

4.4.4 Network Classes:

- **NetworkManager (Façade Class)**


NetworkManager
-MSG_TYPE : Enum
+openConnection() : boolean
+IsconnectionRefused() : boolean
+Isdisconnected() : boolean
+getPlayerInfo(MSG_TYPE) : void
+setPlayerInfo(MSG_TYPE) : void

Attributes:

private enum MSG_TYPE; Message type helps to determine which attributes of the Player class will be getted or setted.

Methods:

public boolean openConnection() This method calls appropriate RESTClient class' methods to open connection to REST Service.

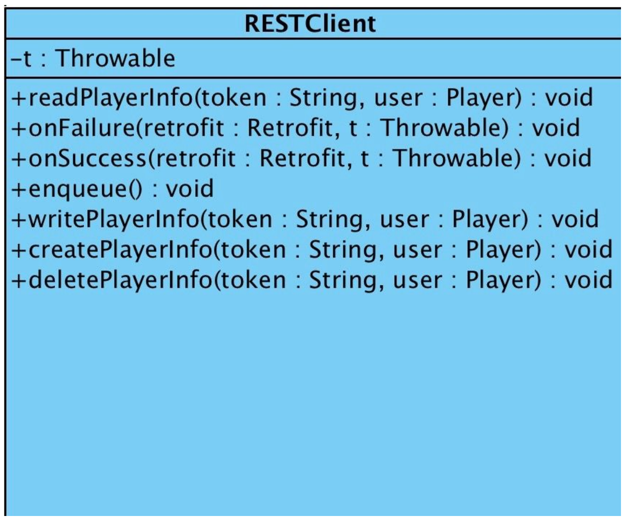
public boolean isConnectionRefused() This method is a helper method

public boolean isDisconnected() This method is a helper method

public void getPlayerInfo(MSG_TYPE) This method calls the appropriate RESTClient class' methods to get/modify player's info.

public void setPlayerInfo(MSG_TYPE) This method calls the appropriate RESTClient class' methods to update/create player's info.

- **RESTClient**



RESTClient
-t : Throwable
+readPlayerInfo(token : String, user : Player) : void
+onFailure(retrofit : Retrofit, t : Throwable) : void
+onSuccess(retrofit : Retrofit, t : Throwable) : void
+enqueue() : void
+writePlayerInfo(token : String, user : Player) : void
+createPlayerInfo(token : String, user : Player) : void
+deletePlayerInfo(token : String, user : Player) : void

<u>Attributes:</u>

private Throwable t : When encountering an issue we throw the t.

<u>Methods:</u>

public readPlayerInfo(String token, Player user) **:** When getting the user's info persistently we use this method. It sends GET requests over HTTP(S) protocol with the help of Retrofit annotations.

public void writePlayerInfo(String token, Player user) : When updating/modifying player's info we use this method. It sends PATCH requests over HTTP(S) protocol with the help of Retrofit annotations.

public void createPlayerInfo(String token, Player user) If user signups(or logins for the first time) we create a row on "users" table with the initial values. It sends PUT requests over HTTP(S) protocol with the help of Retrofit annotations.

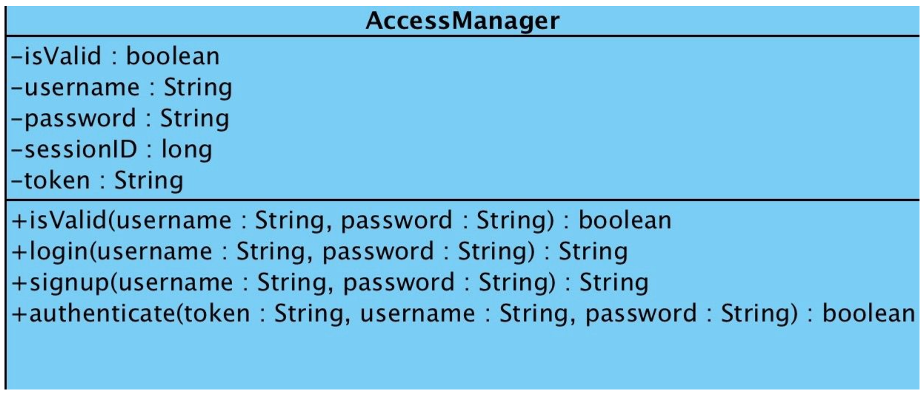
public void deletePlayerInfo(String token, Player user): If user wants to destroy his account we can delete(without modifying) all its info with this method. It sends DELETE requests over HTTP(S) protocol with the help of Retrofit annotations.

public void enqueue() : When we send the requests, we use async approach; hence we need to line up the requests with the enqueue.

public void onSuccess() : If the connection and CRUD operations are successful we parse the response's body over here.

public void onFailure(); If the connection or CRUD operations are unsuccessful we throw an error over here.

- **AccessManager**


AccessManager
-isValid : boolean
-username : String
-password : String
-sessionID : long
-token : String
+isValid(username : String, password : String) : boolean
+login(username : String, password : String) : String
+signup(username : String, password : String) : String
+authenticate(token : String, username : String, password : String) : boolean

private boolean isValid;

private String username : User's "username"

private String password : User's "password"

private final static String token : Authentication token will be used when sending requests.

private long sessionID : sessionID determines the user's Away-From-Keyboard(AFK) time.

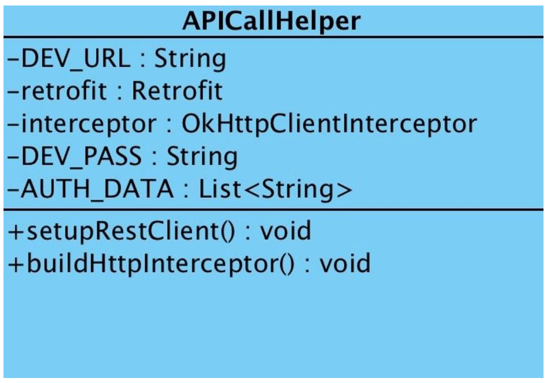
public boolean isValid(String username, String password) : This method lookups the entire user table and checks if the taken username and password is valid or not(previously used or not).

public String login(String username, String password) : This method make the user login with the given credentials and returns a token.

public String signup(String username, String password) : This method help the user signup with the given credentials and returns a token.

public boolean authenticate(String token, String username, String password) : This method checks the token's validity and compares the token with the user's login credentials. If it is false, user cannot send requests and therefore gets an error message.

- **APICallHelper**


APICallHelper
-DEV_URL : String
-retrofit : Retrofit
-interceptor : OkHttpClientInterceptor
-DEV_PASS : String
-AUTH_DATA : List<String>
+setupRestClient() : void
+buildHttpInterceptor() : void

private String DEV_URL : the REST API's route URL

private Retrofit retrofit : Retrofit instance for making the HTTP connections. ***[2]**

private OkHttpInterceptor interceptor : OkHttp Library support instance.

private String DEV_PASS : the REST API's route password

private List<String> AUTH_DATA : This list contains the DEV_URL, DEV_PASS and Token. When making(requesting) the connections we make use of these data.

public void setupRestClient(); This method setups the Rest Client with using the DEV_URL and DEV_PASS.

public void buildHttpInterceptor() : This method specify the connection timeout, read timeout, write timeout values and builds the interceptor with the help of the okHttp Library.

References

[1] "Java Virtual Machine." *Wikipedia*. N.p., n.d. Web. 21 Mar. 2017. <https://en.wikipedia.org/wiki/Java_virtual_machine>.

[2] Retrofit lib: http://square.github.io/retrofit/

[3] https://firebase.google.com/docs/reference/rest/database/