Bilkent University

Department of Computer Engineering

CS 319 Object - Oriented Software Engineering Project

Mafia:TCoS - Mafia:The City of Sin

Design Report - High Level Design

Project Group 2.A:

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

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

Submitted at March 25, 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.

1.3 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 choice: either we deliver the game on time with known-bugs(includes serious bugs too) and publish patches later on or we 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.

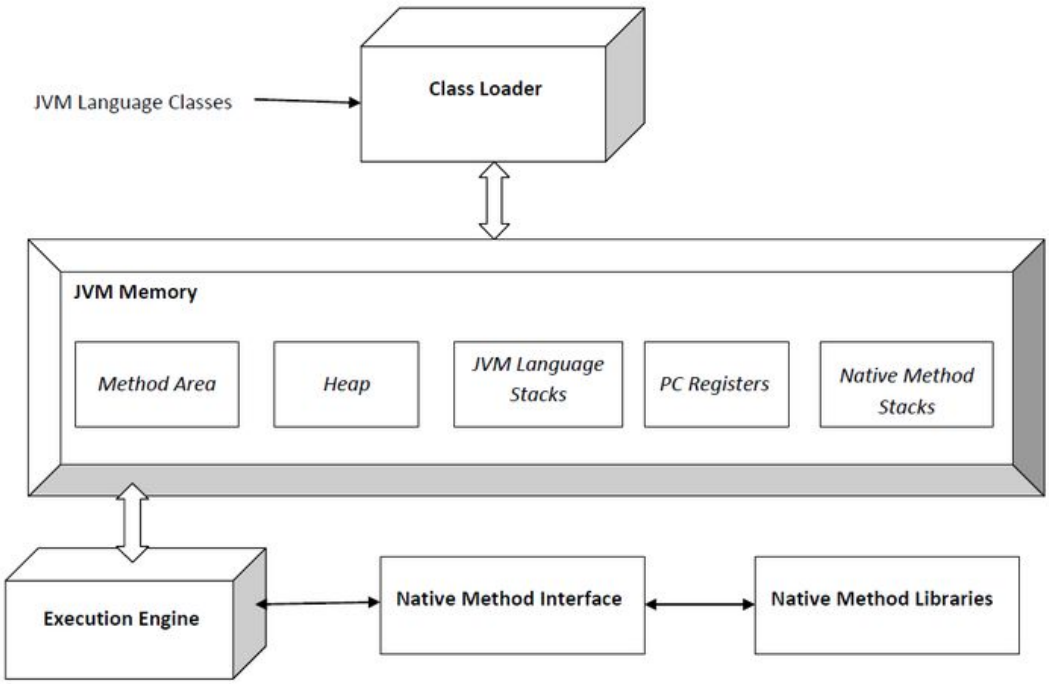

Figure 1. Overview of Java Virtual Machine [1]

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.

Reasons for deciding on this architectural style:

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

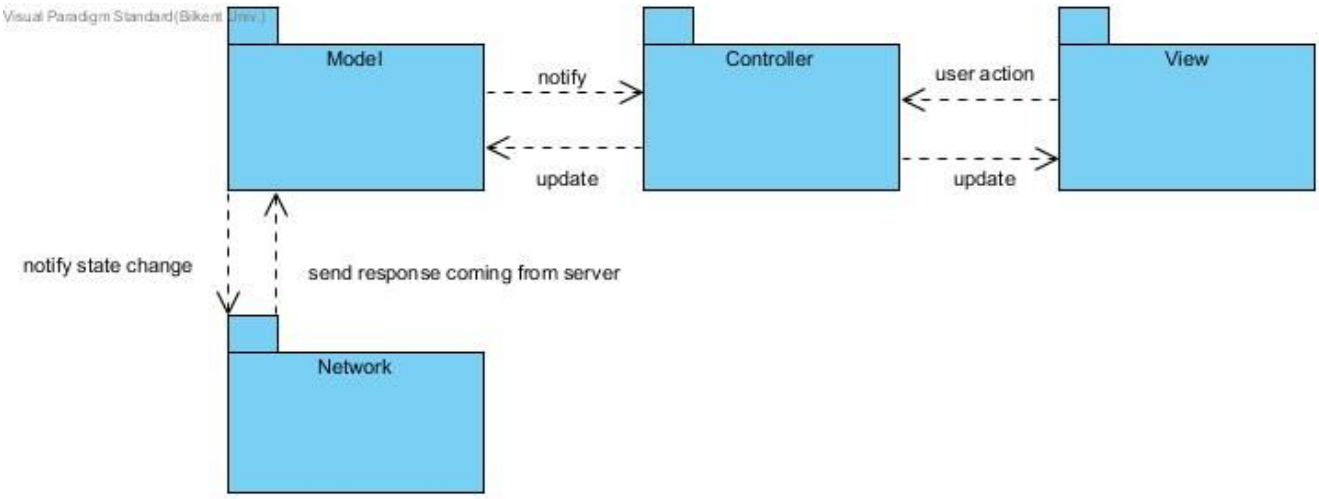

Figure 3. Subsystem decomposition diagram of front end(client): MVC architectural pattern is used with an extra network subsystem aiming to reduce the complexity of the database connection.

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

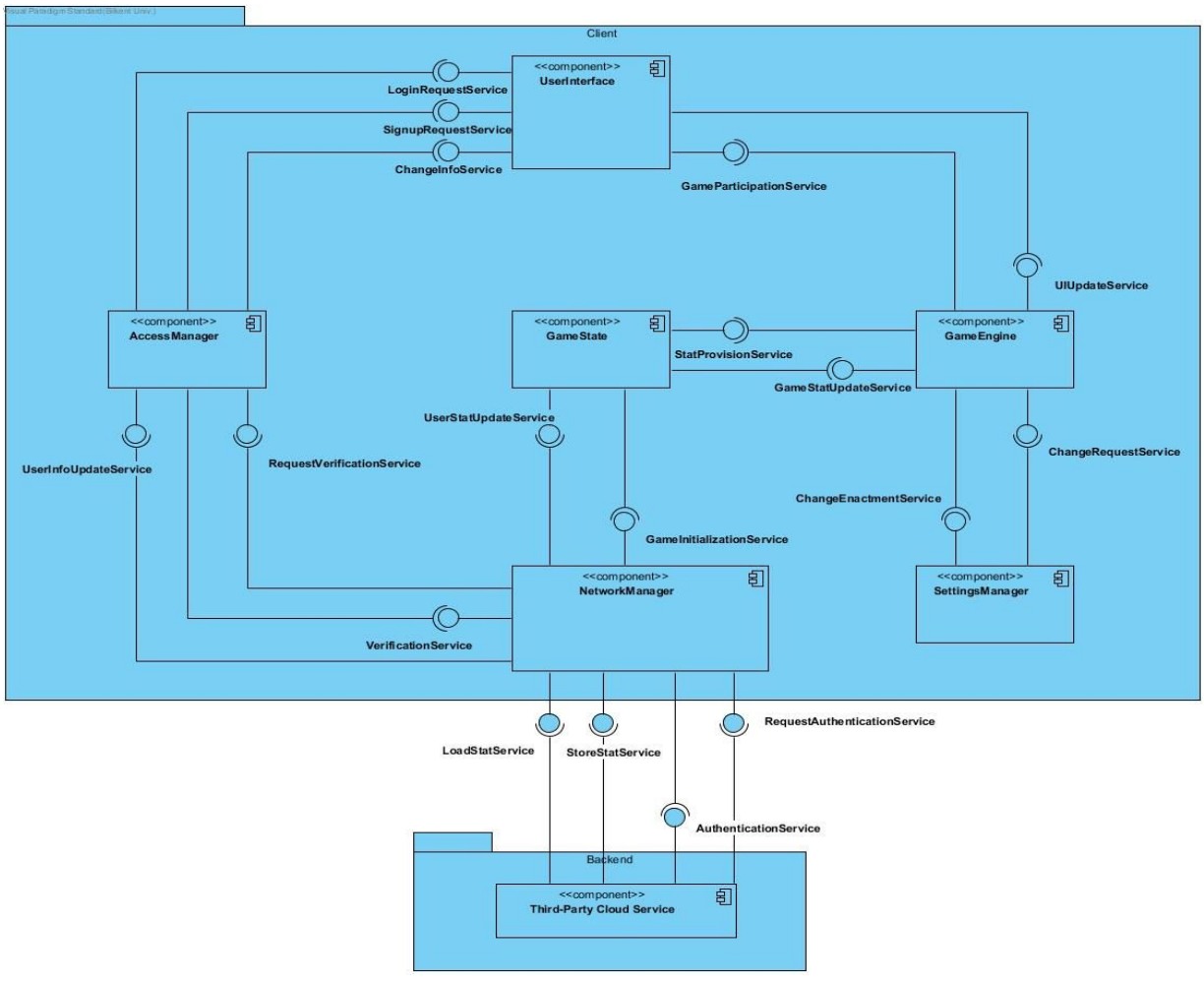

Figure 4. Subsystem Services Diagram

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.

References

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