Final Report Conquerrors Of Mars

(v. 0.1.0)

Author
federico.bonetti2@studio.unibo.it

December 14, 2024

This is a real-time strategy game where players establish their civilization on Mars, starting from a simple base.

Players will compete against each other to steal resources or defend their own bases to maintain control.

The main actions players can perform include: collecting materials from Mars, spending materials to construct buildings (e.g., military bases, material collectors), upgrading buildings (e.g., evolving the headquarters from level 1 to level 2) and training troops to attack other players' bases.

The game should continue running even when players are offline, which necessitates a server-client architecture. Additionally, a small database is needed to store progress in case the server goes down or needs to be restarted. I also plan to develop a minimalistic graphical user interface (GUI) to ensure user interaction is simple and intuitive.

1 Goal/s of the project

The main goal is to ensure that the game continues running smoothly through all possible unforeseen events over a long period of time. Players should be able to play, disconnect, and reconnect without issues, as the game keeps progressing even when they are offline.

In a real-time strategy game, the game continues operating in the background, allowing players to reconnect, review the current state of their base and make strategic decisions based on long-term gameplay.

1.1 Usage scenarios

The objective of this game is to entertain players who enjoy choosing strategies to gradually improve their base over time. The ideal way to play is to log in daily for a short period to check on the base's status and select upgrades, allowing it to progress primarily while you're offline, so you see the benefits each time you return.

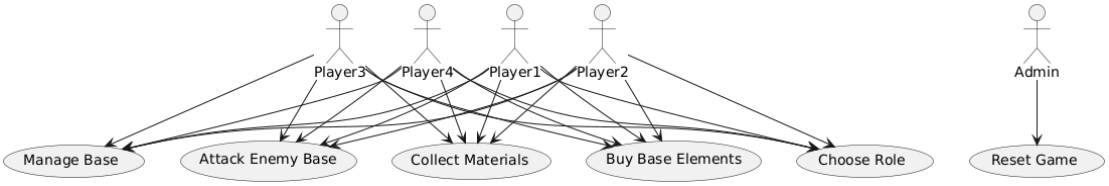

Figure 1: Use case UML

1.2 Definition of Done

As mentioned earlier, the primary goal is to ensure the game runs correctly over time. The requirements that guarantee the proper progression of the game are:

- The server must not allow two players to have the same role.
- The server must collect materials and track how many soldiers have completed their training (based on the number of material collectors and military bases present at the base) and send this information to the clients.
- The server should allow players to acquire new material collectors or military bases by spending materials.
- The server should allow players to upgrade the level of material collectors or military bases present at the base by spending materials (provided their next level does not exceed the level of the headquarters).
- The server, if any soldiers have been sent on an expedition, should update their movements over time and send this information to the clients.
- The server, if any soldiers reach their destination and a battle begins, should send the battle results (materials stolen or gained) to the clients.

From the player's perspective, they should be able to perform the following actions:

- The player should be able to choose their role at the beginning of the game.
- The player should be able to switch between their base view and the world view.
- The player should be able to purchase a material collector or military base if they have enough materials.
- The player should be able to upgrade the level of a material collector or military base present at the base if they have enough materials and the next level does not exceed the level of the headquarters.
- The player should be able to send soldiers on an expedition after deciding how many to send.

To test the game work correctly, it's been choosen 2 principal way:

- Write some Junit test of the principal requirements.
- Do a long gameplay from two clients to see if the game work correctly over time.

2 Background and link to the theory

Because the game must continue running while players are offline, a client-server architecture was selected. This approach centralizes control, ensuring consistent synchronization and persistence of the game state.

The server acts as the central component of the architecture, responsible for managing all interactions, coordinating the flow of the game, and maintaining the overall consistency of the system. The clients, on the other hand, are designed to interact with the server, sending specific requests and receiving updates about the game state. This centralization simplifies synchronization, as all game logic is handled in a single location, reducing the complexity of distributed interactions.

The architecture is based on well-defined components (clients and server) and connectors (TCP sockets) that mediate communication between them. This design ensures modularity, allowing the system to scale and evolve as new requirements emerge. By separating the responsibilities of the server and clients, the architecture adheres to the principles of distributed systems, such as separation of concerns and loose coupling between components. The choice of TCP as the communication protocol guarantees reliable data transmission, a critical feature for maintaining synchronization in a real-time multiplayer game.

To achieve its goals, the system employs a hybrid architectural style, combining elements of event-driven, request-response, and time-based patterns. Each pattern addresses specific challenges inherent to a multiplayer game:

- **Event-Driven Pattern:** The event-driven pattern enables asynchronous communication between the server and clients. In this model, the server pushes updates, such as troop movements and resource collections, to the clients whenever a significant event occurs. This ensures that clients are kept up-to-date with the game state without needing to constantly poll the server. By decoupling the generation and consumption of events, the system achieves spatial and temporal disconnection, a key advantage of the event-driven approach.
- **Request-Response Pattern:** The request-response pattern is used for client-initiated actions. For example, when a player chooses to upgrade an element or purchase a new base component, the client sends a request to the server, which processes the action and sends back a response indicating success or failure. This synchronous interaction guarantees immediate feedback for actions that require direct confirmation from the server, ensuring a smooth and interactive user experience.

- **Time-Based Pattern:** The time-based pattern ensures periodic synchronization of the game state. The server sends updates at fixed intervals, such as every minute, to report on the collection of resources or the completion of troop training. This guarantees that even if a client temporarily disconnects, it can resynchronize with the server upon reconnection, ensuring consistency across all players.

By combining these patterns, the architecture is both reactive and interactive. The event-driven and time-based patterns handle most updates asynchronously, ensuring high responsiveness and scalability. Meanwhile, the request-response pattern provides synchronous interactions for critical client-initiated actions, striking a balance between automation and user control.

The client-server architecture was chosen over alternative styles, such as peer-to-peer or data-centered architectures, due to its inherent simplicity and robustness. Peer-to-peer architectures, while useful for decentralized systems, would have introduced significant challenges in terms of synchronization, fault tolerance, and maintaining a persistent game state. In contrast, the client-server model centralizes the game logic, allowing the server to act as a single source of truth. This centralization ensures that the game can continue running even when players are temporarily disconnected, as the server maintains the game state independently of the clients.

The server's centralized role also simplifies the management of scalability and fault tolerance. With all game logic handled on the server, additional clients can be supported with minimal changes to the architecture. The modular nature of the design means that each component can evolve independently; for example, additional factions or game mechanics can be introduced without disrupting the core communication protocols.

Furthermore, the architecture aligns with the principles of distributed systems by separating computation and communication. The server handles computationally intensive tasks, such as processing game logic and resolving conflicts, while the clients focus on user interaction and display. This separation ensures efficient use of resources and avoids overloading the clients, particularly in scenarios involving multiple simultaneous connections.

In conclusion, this hybrid architecture provides a robust foundation for the multiplayer game. The combination of event-driven, request-response, and time-based patterns ensures responsiveness, scalability, and synchronization. The client-server model's simplicity and centralization make it well-suited for managing the complexities of real-time multiplayer interactions, providing a solid basis for future extensions and enhancements.

3 Requirements Analysis

3.1 Requirements list

Functional requirements

- The player should be able to choose their role at the beginning of the game.
- The server must not allow two players to have the same role.

- The server must collect materials and track how many soldiers have completed their training (based on the number of material collectors and military bases present at the base) and send this information to the clients.
- The server should allow players to acquire new material collectors or military bases by spending materials.
- The server should allow players to upgrade the level of material collectors or military bases present at the base by spending materials (provided their next level does not exceed the level of the headquarters).
- The server, if any soldiers have been sent on an expedition, should update their movements over time and send this information to the clients.
- The server, if any soldiers reach their destination and a battle begins, should send the battle results (materials stolen or gained) to the clients

Non functional requirements

- The server should continue working in a long time without errors or problems.

3.2 Top-down analysis

The client-server architecture is essential for ensuring that the game can run continuously over time, even when players are offline. By having a central server managing the game state, including materials, troops, and the actions of each player, the system allows the game world to persist independently of the players' connections.

This means that, even if a player disconnects or is inactive for a period, the server continues to handle updates, progress, and interactions in the game world. For example, while a player is offline, the server can keep updating the state of materials gathered, troops trained, and soldiers on missions, ensuring that when the player logs back in, they will find their progress preserved.

This ensures the game's world is dynamic and evolving continuously, allowing for long-term play without the need for players to be always connected.

Moreover, the event-driven interaction pattern supports the long duration of gameplay by allowing the server to send updates to clients periodically or in response to game events, such as changes in materials or the completion of tasks.

This approach ensures that the game remains active and responsive over time, even when no direct player action is required. For example, the server may periodically push updates about the state of resources or troops, keeping players informed about their progress and the game's evolution in real time.

This real-time feedback loop helps create a dynamic, persistent game world, which is crucial for a game designed to be played over extended periods.

The use of sockets for communication, particularly TCP or WebSockets, is another key factor in supporting long gameplay duration. Sockets allow for continuous, low-latency communication between the server and the clients, which is essential for real-time updates during the gameplay. Additionally, socket-based communication ensures

that data is exchanged in a reliable and persistent manner, without the need for constant reconnections, thus supporting long, uninterrupted gaming sessions. The server can push updates to the clients as soon as an event occurs (such as a soldier finishing training), and clients can request data or send commands whenever needed, providing a smooth, uninterrupted experience.

In terms of scalability and future-proofing, the client-server architecture also facilitates the ability to expand the game as more players join. The server can handle an increasing number of simultaneous clients by scaling horizontally (adding more servers or resources), while maintaining the integrity and performance of the game state.

This scalability ensures that the game can accommodate growing player bases over time, a vital requirement for maintaining long-term engagement. Furthermore, with the event-driven and time-based interaction models, the server can handle complex interactions and periodic updates across a large number of players efficiently, making sure the system remains responsive even as the player base grows.

Finally, the client-server model allows for easier maintenance and updates. Developers can update the server-side game logic, add small new features, and apply patches without disrupting the players' experience (Instead, for relevant features, it is also necessary to update the client.). Clients simply connect to the updated server, which guarantees that all players are using the same version of the game.

This centralized control makes it easier to manage the game's evolution over time, ensuring that new content or fixes are deployed smoothly, further supporting the game's long-term viability.

Overall, the client-server architecture, event-driven patterns, and socket-based communication provide a robust foundation for a game designed to run over long periods. These choices ensure that the game remains dynamic, scalable, and responsive, offering a persistent experience that can adapt to a growing player base and deliver continuous updates and improvements.

This structure is crucial for maintaining player engagement and ensuring the game's longevity.

4 Design

4.1 Structure (Domain Entities)

The main entities in the system that need to be modeled to reflect the domain include:

- **Base**: The base represents the central structure of each player and serves as the core for managing resources and troops. Each base consists of 4 main sub-entities:
  - **Headquarter**: The central headquarters defines the maximum level achievable for any other element within the base. The level of the Headquarter is the upper limit for upgrading other structures and troops. This link between the level of the Headquarter and the other elements ensures consistent and centralized management of progress in the game.

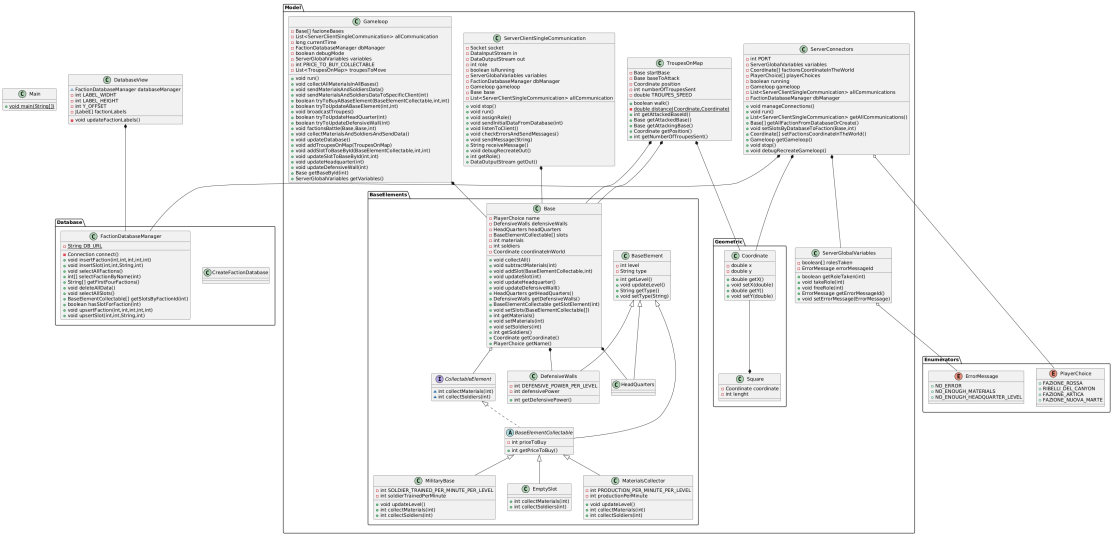

Figure 2: Server Class UML

  - **Military Base**: A structure dedicated to training soldiers. Each soldier trained is a resource that can be used for missions. The training capacity of the base depends on the level of the Headquarter, which, in turn, influences the speed and quantity of soldiers produced.
  - **Material Collector**: A structure that collects resources from the game world. The resources collected are used to build or upgrade other base entities. The quantity of resources gathered is linked to the base's productivity and the management of resources by the system.
  - **Defensive Walls**: Defensive walls that reduce the damage taken by the base during enemy attacks, increasing the protection of the base. The effectiveness of these walls is determined by their upgrade level, which depends on available resources and the management capabilities of the base.
  - **Empty Slot**: An available slot in the base where the player can choose to build either a Military Base or a Material Collector. The management of empty slots is directly connected to the update system and available resources, influencing the player's strategy.

- **Troops in the World**: These entities represent the military units that move across the game world. Troops are sent on missions to enemy bases to perform tasks or attacks. Their state (position, health, etc.) is continuously updated by the server, which constantly monitors the interactions between the troops and other entities in the game.

The system is designed to maintain centralized control through the server, which manages operations and ensures that domain entities (such as the base, troops, and

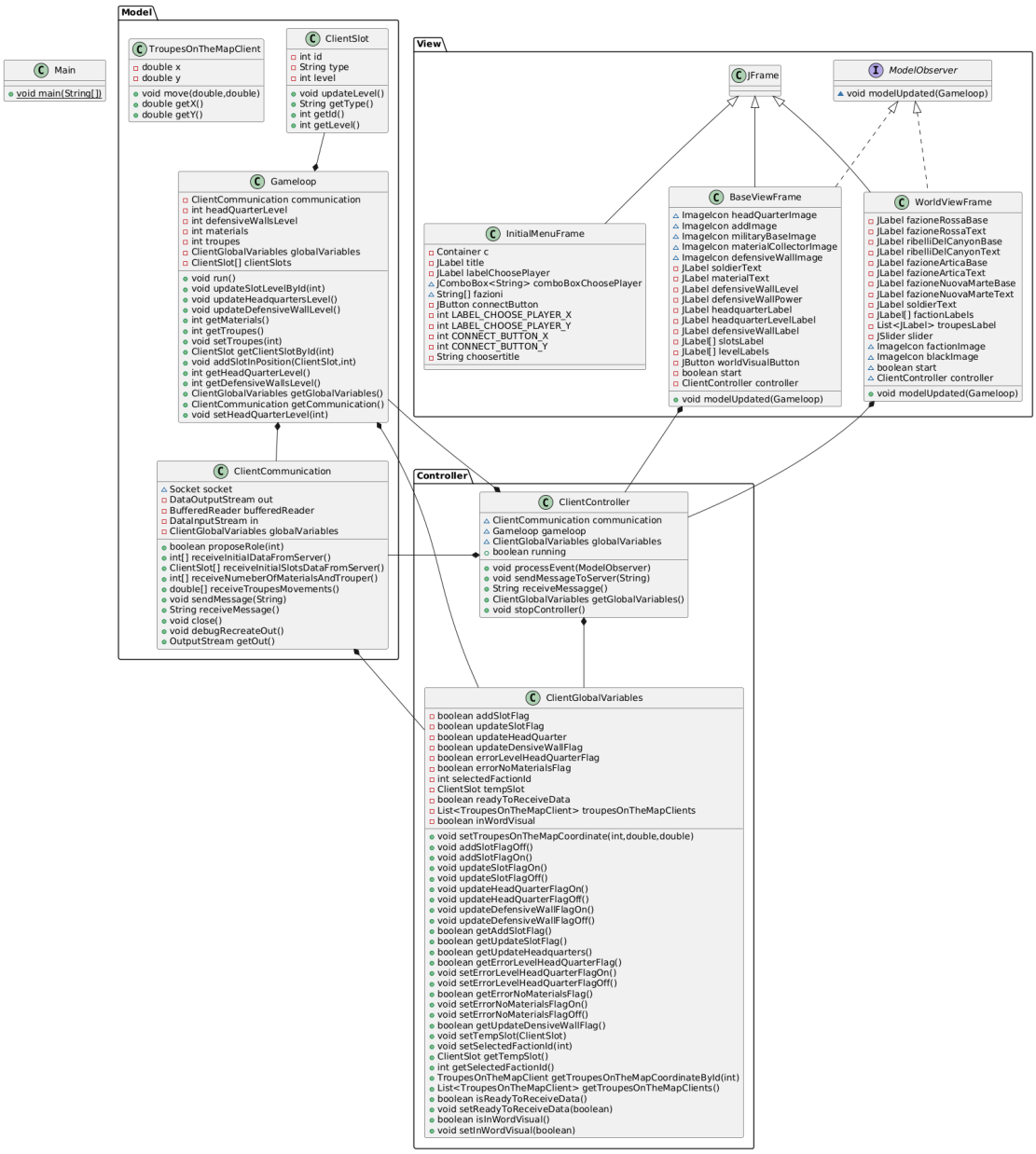

Figure 3: Client Class UML

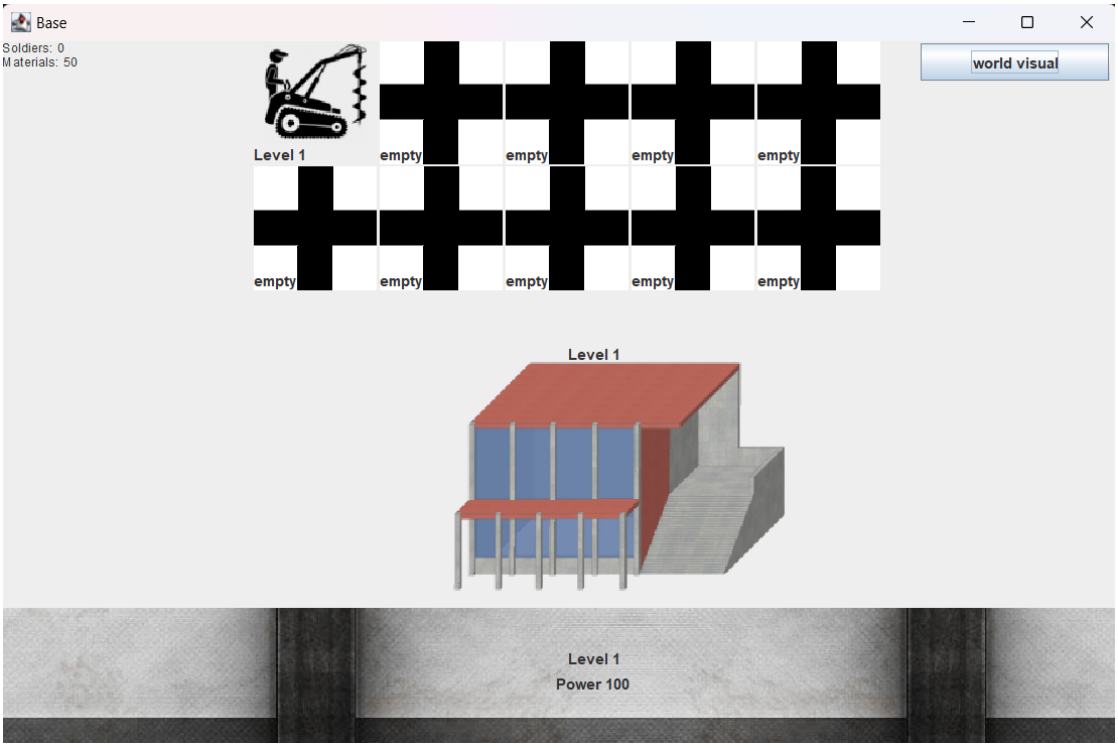

Figure 4: How the base looks at the beginning

structures) are always synchronized between the server and clients. Any interaction related to the creation, upgrading, or movement of entities in the game is the result of an operation managed by the server.

The interaction between entities in the system is tightly linked to the game logic. For example, when a player builds a new structure in their base, this behavior is not just a decision of the player, but a consequence of the update system and the resources that the server keeps track of. The synchronization of entities, such as troops in motion or resource production, is managed periodically by the server, which sends updates to the clients at regular intervals.

This separation between the system that manages the entities and the game logic allows for efficient resource management, while continuous updates allow the game to evolve over time without requiring players to worry about directly managing updates or synchronization.

4.2 Interaction

In this system, communication between the entities is managed by the server, which controls the flow of data and ensures synchronization across the entire game. The server handles the core logic, while the client acts as the interface for player interaction. The system architecture allows for dynamic communication, where both the player and the

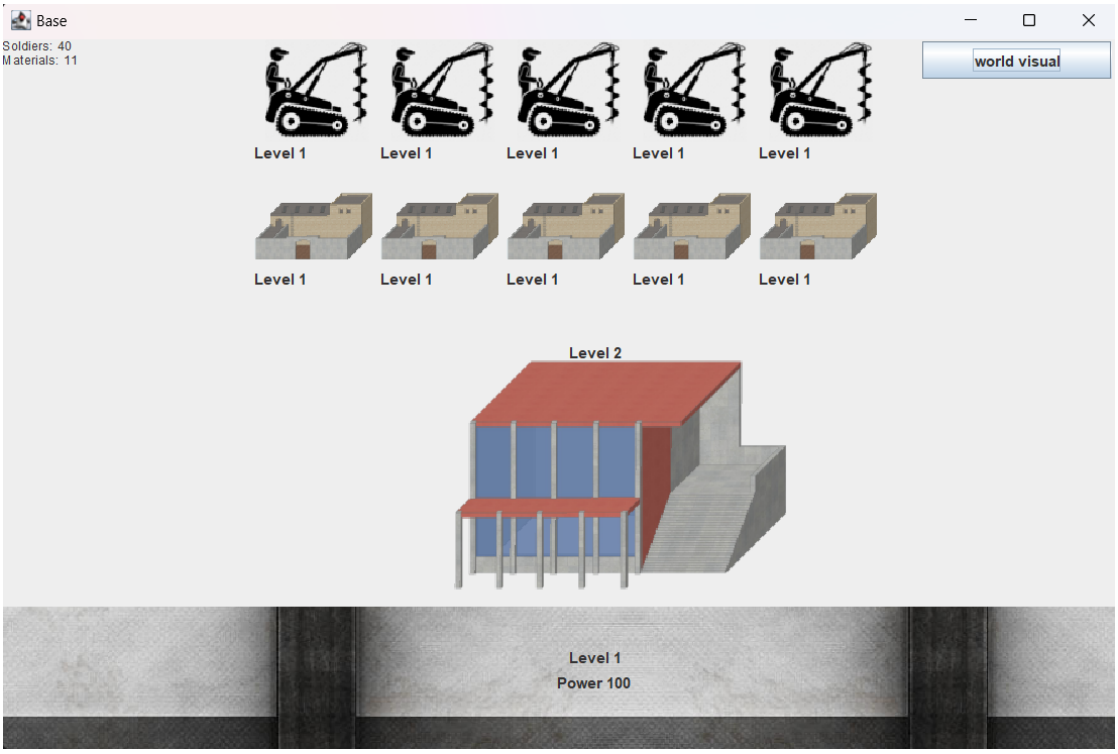

Figure 5: How the base could look after playing a little bit

server can initiate actions that impact the game state.

- **Base**: The base is composed of the following elements:
  - **Headquarter**: The Headquarter is responsible for setting the maximum level achievable for other elements in the base. The server manages the logic of upgrading the Headquarter and updates the client with the latest state after each modification.
  - **Military Base**: The Military Base is managed by the server and produces soldiers over time. The server tracks the number of soldiers trained and periodically sends updates to the client with the current number of available soldiers. When a player clicks on the Military Base to upgrade it, the client sends a request to the server to update the base's state, provided the player has enough resources.
  - **Material Collector**: The Material Collector is also managed by the server and collects resources at regular intervals. The server tracks the amount of resources gathered and updates the client with the total amount available in the base. When a player clicks on the Material Collector to upgrade it, the client sends a request to the server to upgrade it if the player has enough resources.
  - **Defensive Walls**: The server manages the updates for the defensive walls' effectiveness. These updates are sent to the client based on the current level of the walls and the resources available. If the player clicks on the walls to upgrade them, the client sends a request to the server to handle the action and update the state accordingly.
  - **Empty Slot**: The Empty Slot is managed by the server. When the player clicks on an empty slot, the client sends a request to the server to build an element in that slot. The player is prompted to choose between building a Military Base or a Material Collector. Once the player makes the choice, the client sends the request to the server. If the base has enough resources, the server deducts the necessary amount and updates the base's state accordingly.
- **Base in the World**: This element displays the locations of all bases on the game map. If the player clicks on their own base, they are returned to the "Base View." If the player clicks on an enemy base and has selected a number of soldiers greater than one, the client sends a request to the server to attack the enemy base. This results in the generation of troops on the world map, which will be sent to the target base for an attack.
- **Troops in the World**: The server continuously manages the state of troops, tracking their position, health, and actions. If the player deploys troops, the server updates their state and sends information to the client, ensuring that the game state is consistent across all connected clients.

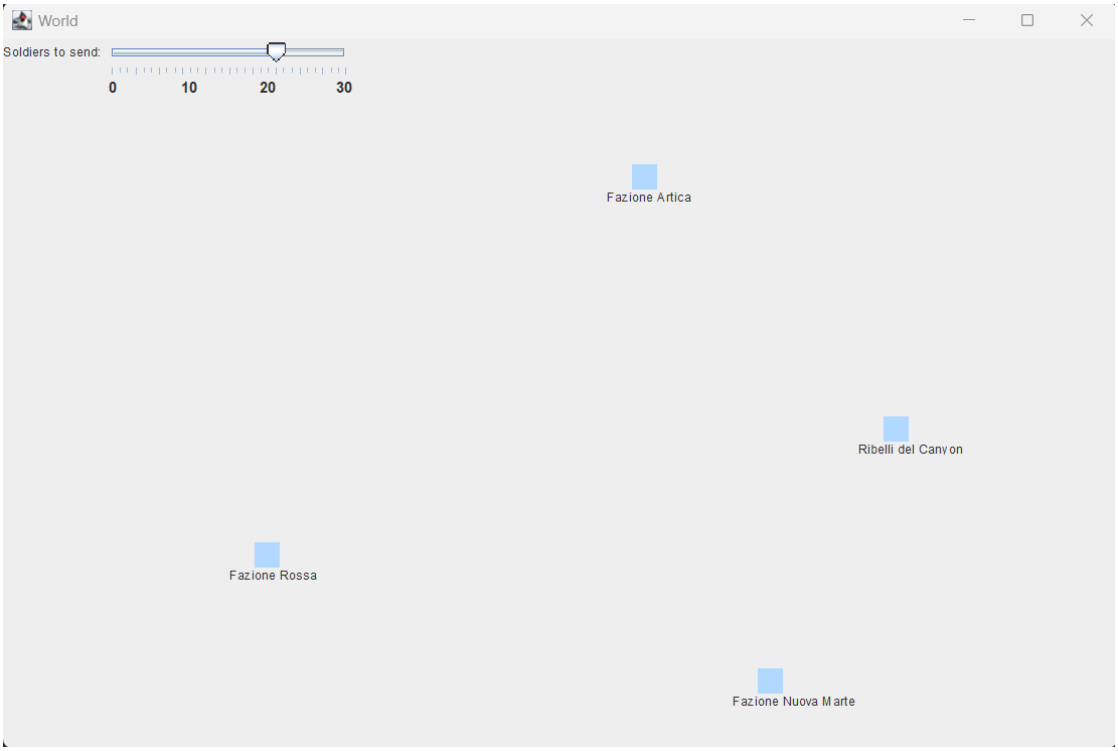

Figure 6: This is the world visual

The communication flow in the system ensures that both server-side events and client-side actions are managed efficiently. The client sends commands to the server, which processes these requests and updates the game state accordingly. The server then sends back the updated game state to all connected clients. This communication is essential to maintaining the consistency of the game world, where all entities (such as resources, troops and bases) must be synchronized across clients.

Periodically, the server also sends updates to the client regarding changes in the game state, such as resource collection, troop movements, and other dynamic changes. These updates are sent as messages to all connected clients, with the content depending on the visual they are currently in. In the **Base View**, clients receive updates about their trained soldiers and collected materials, ensuring they can manage their resources effectively. In the **World Visual**, clients instead receive updates about the coordinates and movements of all troops in the game world. This selective messaging ensures that clients are only sent relevant data based on their current context.

In this system, the server plays a central role in managing interactions and maintaining the integrity of the game state. The client, on the other hand, acts as the interface for the player, facilitating their interactions with the system by sending requests and displaying the latest state of the game. The hybrid approach of both request-response and event-driven communication ensures that the system remains responsive and consistent, regardless of the frequency or nature of interactions.

4.3 Behaviors

The system is built on a centralized architecture, where the **server** plays a pivotal role in managing the game state and ensuring consistency across all connected **clients**. The server continuously tracks the status of all entities in the game, processes client requests, and periodically sends updates to ensure synchronization. Clients act as the player's interface to the system, sending commands to the server and displaying the latest state of the game based on the information received.

Server Behavior: The server manages all core operations within the system. It processes incoming requests from clients, such as upgrading elements, training soldiers, or initiating attacks, and ensures the game state reflects these actions. Periodically, the server broadcasts updates to clients based on their current view. For clients in the **Base View**, updates include details about trained soldiers and collected resources. For clients in the **World Visual**, the server sends the positions and movements of troops across the game map. The server also enforces rules, such as limiting upgrades based on the Headquarter level, and resolves interactions like battles between troops or resource allocation.

Client Behavior: Clients facilitate player interaction by sending requests to the server and displaying the resulting updates. Each client operates in one of two primary views:

- In the **Base View**, players can manage their base by interacting with elements such as the Headquarter, Military Base, Material Collector, and Defensive Walls.

Commands like upgrades or purchases are sent to the server for processing, and the client displays the updated state once the server confirms the action.

- In the **World Visual**, players can view the map of all bases and troop movements. Players can command troops to attack enemy bases by clicking on them, which sends a request to the server to deploy troops. The server manages the deployment and updates all clients about the troop movements and outcomes of battles.

Each entity in the system operates based on its role, with behaviors managed centrally by the server. The server ensures all entities are synchronized and reflects their states to the clients.

Base: The Base acts as the central structure for each player, containing and organizing elements like the Headquarter, Military Base, Material Collector, Defensive Walls, and Empty Slots. While the Base itself does not perform actions directly, the server manages its overall state, including resources and levels.

Headquarter: The Headquarter determines the maximum level for all other elements in the Base. The server ensures that no upgrades exceed this limit. When a player attempts to upgrade the Headquarter, the client sends a request to the server, which validates the action and processes it if sufficient resources are available.

Military Base: The Military Base trains soldiers over time. The server determines the number of soldiers produced based on the level of the Military Base and periodically updates the client with the current number of available soldiers.

Material Collector: The Material Collector gathers resources for the Base. The server calculates the amount of resources collected over a fixed interval, influenced by the Material Collector's level. Updates about the total resources available are sent periodically to the client.

Defensive Walls: Defensive Walls provide protection for the Base, reducing the damage from enemy attacks. The server calculates their effectiveness based on their level and updates the client when changes occur.

Base in the World: The Base in the World represents the locations of all bases on the game map. The server tracks the coordinates of every base and periodically sends this information to clients. If a player clicks on their own base, the client switches to the **Base View**. Clicking on an enemy base sends a request to the server to initiate an attack if sufficient troops are selected.

Troops in the World: Troops represent military units deployed on missions. The server manages their positions, movements, and interactions with other entities. When troops are sent to attack, their behavior is controlled by the server, which updates all clients with their status and the results of their interactions.

By centralizing the management of behaviors on the server, the system maintains consistency and ensures smooth gameplay across all clients. The hybrid communication model, combining request-response for player actions and periodic updates for synchronization, allows the system to remain responsive while providing real-time updates.

4.4 Architecture

The architecture of the game is based on a client-server model, ensuring centralized control of game state and continuous gameplay, even when players are offline. This design separates responsibilities into distinct modules, each handling specific aspects of the system.

The **Server Module** acts as the core of the architecture, managing the entire game state, including Base levels, resources, troops, and battles. It processes commands from clients, updates the game state, and sends updates back to the clients. The server ensures consistency and fairness in the game.

The **Client Module** provides the player interface, allowing users to interact with the game by sending commands to the server (e.g., training soldiers, upgrading buildings). It receives updates from the server and visualizes the current game state.

Data flows between the client and server through a well-defined protocol over a TCP connection. Commands from the client are sent as requests to the server, which processes them and responds with updates. In addition, the server periodically sends updates to ensure that the client always has the latest state of the game.

4.5 Corner cases

- **Faults detection:** Both the client and the server continuously expect messages from each other during the interaction. If the connection is closed unexpectedly, the recipient (client or server) detects the anomaly by receiving an invalid or malformed message. This mechanism ensures quick detection of communication failures and prevents inconsistent states.

- **Recovery strategies:** The server constantly saves all game state data to a database (every minutes). In the event of a server crash, this database allows the server to restore the latest state upon restarting, minimizing the impact on the gameplay experience. If a client disconnects unexpectedly, its role is released, enabling a new client to join the game and assume that role. Slots or other in-game resources tied to the base remain intact and are not released to maintain consistency.

- **Error messages:** Clear and informative error messages are sent to clients when invalid actions are attempted or unexpected errors occur. For instance, if a player tries to upgrade a building beyond the Base's level, the client receives a specific error message explaining the issue.

- **Graceful shutdown:** When shutting down, the server ensures all active game states are saved to the database, and ongoing client-server communications are terminated cleanly. Clients are notified about the shutdown to avoid confusion and allow them to reconnect once the server is back online.

5 Salient implementation details

Listing 1: The client propose the role

```
    public boolean proposeRole(int roleId) throws IOException,
        InterruptedException {
        sendMessage(""+roleId);
        in = new DataInputStream(new BufferedInputStream(socket.getInputStream()))
            ;

        String line = in.readUTF();
        System.out.println("Received: " + line);

        if(line.equals("ok")){
            return true;
        } else {
            sendMessage("end communication");
            close();
            return false;
        }
    }
```

Listing 2: The server assign the role if it is possible

```
public void assignRole() throws IOException, InterruptedException {
        String line = in.readUTF();
        System.out.println("Server received: " + line);
        if (variables.getRoleTaken(Integer.parseInt(line))) {
            sendMessage("no");
            System.out.println("role not empty");
            role = -1;
        } else {
            variables.takeRole(Integer.parseInt(line));
            sendMessage("ok");
            System.out.println("role assigned");
            role = Integer.parseInt(line);

            if(!dbManager.hasSlotForFaction(role)){
                dbManager.insertSlot(role, 0,"MaterialCollector",1);
                gameloop.getBaseById(role).addSlot(new MaterialsCollector(1), 0);
                System.out.println("this is a new faction, added material
                    collector lv 1");
            }

            Thread.sleep(10);
            sendInitialDataFromDatabase(role);
            System.out.println("sent faction data");
        }
    }
```

One of the most important aspects of the implementation is the communication model between the client and the server. The system uses TCP sockets to establish a reliable communication channel, ensuring consistent and real-time updates for the players. To prevent data loss and ensure smooth communication, each message sent between the client and the server is acknowledged, and both sides handle unexpected disconnections gracefully. This approach guarantees robustness in a multiplayer environment.

The server continuously saves game state data to a relational database, which allows recovery in case of failure. Each player's progress, resources, and troop movements are

stored, enabling the server to reload the last consistent state after a crash or restart. This database-driven approach provides fault tolerance and contributes to the overall stability of the system.

Listing 3: Server manage the request of buying an element

```
    public boolean tryToBuyABaseElement(BaseElementCollectable slotToAdd, int
        slotId, int baseId){
        if(fazioneBases[baseId].getMaterials() >= PRICE_TO_BUY_COLLECTABLE){
            System.out.println("prezzo: " + slotToAdd.getPriceToBuy() + " ,
                materiali disponibili: " + fazioneBases[baseId].getMaterials());
            fazioneBases[baseId].subtractMaterials(PRICE_TO_BUY_COLLECTABLE);
            addSlotToBaseById(slotToAdd, slotId, baseId);
            return true;
        }
        variables.setErrorMessage(ErrorMessage.NO_ENOUGH_MATERIALS);
        return false;
    }
```

Listing 4: The action Listener of a slot

```
public void addActionListenerToSlotById(int id){
                slotsLabel[id].addMouseListener(new MouseAdapter() {
                    @Override
                    public void mouseClicked(MouseEvent e) {
                        if(slotsLabel[id].getIcon().equals(addImage)) {
                            String type = "";
                            String[] options = {"Material Collector", "Military
                                Base"};
                            int choice = JOptionPane.showOptionDialog(null,
                                    "Cosa vuoi aggiungere?",
                                    "Seleziona un'opzione",
                                    JOptionPane.DEFAULT_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null, options, options[0]);
                            if (choice == JOptionPane.CLOSED_OPTION) {
                                return;
                            }
                            if (choice == 0) {
                                type = "MaterialCollector";
                            } else if (choice == 1) {
                                type = "MilitaryBase";
                            }
                            controller.sendMessageToServer("add " + type);
                            ClientSlot tempSlot = new ClientSlot(id, type, 1);
                            controller.getGlobalVariables().setTempSlot(tempSlot);
                            tryToBuyCollectableMessages(id);
                        } else {
                            String type = "";
                            if(slotsLabel[id].getIcon().equals(
                                materialCollectorImage)) {
                                type = "MaterialCollector";
                            } else {
                                type = "MilitaryBase";
                            }
                            String[] options = {"Si", "No"};
                            int choice = JOptionPane.showOptionDialog(null,
                                    "Vuoi aggiornare l'elemento al costo di " +
                                        Math.pow(5, getLevelFromLabelLevel(
                                        levelLabels[id])+1) + " materiali",
                                    "vuoi aggiornare l'elemento?",
```

```
                                    JOptionPane.DEFAULT_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null, options, options[0]);
                            if (choice == JOptionPane.CLOSED_OPTION) {
                                return;
                            }
                            if (choice == 0) {
                                controller.sendMessageToServer("update " + type);
                                ClientSlot tempSlot = gameloop.getClientSlotById(
                                    id);
                                controller.getGlobalVariables().setTempSlot(
                                    tempSlot);
                                tryToUpdateCollectableMessages(id);
                            }
                        }
                    }
                });

            }
```

Another interesting aspect of the implementation is the real-time management of game elements, such as military bases, material collectors, and troops. These entities interact with each other dynamically, and their states are constantly updated both based on player input and automatic time-based events.

Listing 5: The server broadcast the coordinate of all the troupes

```
    public void broadcastTroupes(){
        for(int troupesWalkId = 0; troupesWalkId < troupesToMove.size();
            troupesWalkId++) {
            for (int i = 0; i < allCommunication.size(); i++) {
                allCommunication.get(i).sendMessage("troupes walking");
                allCommunication.get(i).sendMessage("" + troupesWalkId);
                allCommunication.get(i).sendMessage("" + troupesToMove.get(
                    troupesWalkId).getPosition().getX());
                allCommunication.get(i).sendMessage("" + troupesToMove.get(
                    troupesWalkId).getPosition().getY());
            }
        }
    }
```

However, to prevent communication conflicts between the client and the server, the client only receives soldier movement data when in the **"World View"** mode. This ensures that the client's view of the base does not interfere with the ongoing communication between the client and the server, which could result in conflicting data being sent or processed.

While viewing the base, the client is not receiving or sending updates related to troop movements, as this is only active when the player is observing the world map where such data is relevant. The decision of whether to send a specific type of message, based on the view the client is currently in (**Base View** or **World View**, as explained earlier), depends on the boolean variable `inWorldVisual`. This variable is part of the `ClientGlobalVariables` class and is managed by the `Gameloop` class.

The `inWorldVisual` variable acts as a semaphore, enabling or disabling the transmission of certain types of data based on the player's view. This ensures a clean separation of concerns and prevents communication issues.

The project also makes use of design patterns like **Observer** and **Command** to handle communication between the server and the client.

The **Observer** pattern allows the client to receive updates automatically whenever the server state changes, while the **Command** pattern is used to encapsulate player actions and their corresponding server-side operations.

6 Validation

To evaluate the correctness and compliance of the produced software with the specified requirements, a set of automated tests was developed to focus on key functionalities and interactions between the client and server. These tests are designed to verify both the expected behaviors and edge cases for the game mechanics.

Unit testing frameworks, such as JUnit, were used to ensure that the individual components work as expected. The tests are organized around core features, including role assignment, material collection, and the interaction between clients and the server.

Some of the tests implemented include:

- **First Client Gets Role**: This test ensures that the first client successfully receives a role and the server properly handles role assignments.
- **Different Clients with Different Roles**: This test verifies that when two clients connect, they are assigned different roles without conflicts.
- **Base Can Collect Materials**: This test checks if the base is capable of collecting materials and updating its resource count over time.
- **Server and Client Material Synchronization**: This test ensures that the material count is correctly synchronized between the client and the server after material collection.
- **Client Can Buy Material Collector**: This test validates that the client can purchase a material collector if sufficient resources are available.
- **Base Can Buy Military Base and Train Soldiers**: This test ensures that a base can purchase a military base and train soldiers, verifying the progression system.

In addition to the automated tests, a long gameplay session lasting approximately two hours was conducted. During this session, two clients were connected simultaneously, and all the main aspects of the game were tested. This provided a comprehensive validation of the gameplay dynamics, ensuring that the game functions as expected over extended periods. The session also confirmed that the system maintained consistent behavior and proper synchronization between clients and the server, even as the game state evolved.

Each of the automated tests evaluates specific game mechanics and ensures that the system behaves as expected in both normal and edge-case scenarios. The tests are

designed to run automatically, and their results are analyzed to identify any discrepancies or failures.

In total, the test suite includes N tests, out of which X tests passed successfully, with a passing rate of $Y\%$. The coverage of the tests ensures that the main functionalities are thoroughly tested, while also providing insight into possible areas for further improvement.

These automated tests are complemented by manual testing to ensure the user experience aligns with the design requirements. The validation process helps confirm that the system meets both functional and non-functional requirements, with a focus on reliability and correctness.

7 Deployment Instructions

To deploy and launch the produced software artifacts, follow these steps:

- **Clone the Repository and Build the Project:**
  - Clone the project's repository to a local machine.
  - Use Gradle to build the project. Ensure Gradle is installed or use the provided Gradle Wrapper (`gradlew.bat` for Windows or `./gradlew` for Unix systems).
  - build the project

    ```
    ./gradlew build
    ```

- **Running the Server:**
  - Start the server using the Gradle command:

    ```
    ./gradlew RunServer
    ```

  - The server initializes the game environment, including the database for saving player dat¡a and game state.
  - To reset the game state, use the server's provided reset functionality:
    - Access the database view available in the server interface.
    - Confirm the reset operation through a security prompt. This action deletes all stored game data and resets the game environment.
    - Restart the server. It will recreate the database.
- **Running the Client:**
  - Each player must start the client application using the Gradle command:

    ```
    ./gradlew RunClient
    ```

  - Upon launching, the player is prompted to choose a role:

    - Roles include specific identifiers such as "Fazione Rossa", "Ribelli del Canyon", "Fazione Artica" and "Fazione Nuova Marte".
    - If the chosen role is already taken, the player must select another available role.
    - Once a valid role is selected, the player gains access to the corresponding game functionality.
  - The client also provides a text box for entering the server's IP address:
    - By default, the IP is set to `localhost`.
    - Players can modify the IP address in the IP's text box if the server is hosted on a different machine.

Listing 6: build.gradle.kts file which contains gradle's function "RunServer" and "RunClient"

```
plugins {
    id("java")
    id("application")
}

group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.xerial:sqlite-jdbc:3.36.0.3")

    testImplementation(platform("org.junit:junit-bom:5.9.1"))
    testImplementation("org.junit.jupiter:junit-jupiter")
}

tasks.test {
    useJUnitPlatform()
}

tasks.register<JavaExec>("RunServer") {
    mainClass.set("org.example.src.server.Main")
    classpath = sourceSets["main"].runtimeClasspath
}

tasks.register<JavaExec>("RunClient") {
    mainClass.set("org.example.src.client.Main")
    classpath = sourceSets["main"].runtimeClasspath
}
```

- **Configuration Notes:**
  - If a player disconnects unexpectedly, their role becomes available for other players.
  - To ensure proper functionality, maintain a stable network connection between clients and the server.

- The build configuration includes custom Gradle tasks:
  - `RunServer`: Starts the server application.
  - `RunClient`: Starts the client application.

  These tasks are defined in the `build.gradle.kts` file and simplify the deployment process.

8 Usage Examples

The following examples demonstrate how to use the produced software artifacts in various scenarios, covering all the main functionalities of the system.

- **Starting the Game:**
  - Launch the server.
  - Each player launches the client application.
  - Upon connection, players are prompted to choose a role. For example:
    - Player 1 selects "Fazione Rossa."
    - Player 2 selects "Ribelli del Canyon."
  - If a role is already taken, the player is asked to choose another available role. For instance:
    - Player 3 tries to select "Fazione Rossa" it is already taken. He then select "Fazione Artica".
- **Managing a Base:**
  - Upon connecting to a base for the first time:
    - If the base has never been used before, the system automatically creates a *Material Collector* in the first visible slot (out of 10 total available). This ensures the player can immediately start collecting resources and participate in the game.
    - Example: Player 4, as "Fazione Nuova Marte" connects to an unused base. A *Material Collector* is automatically created in the first slot.
  - Players can then manage their base by:
    - Selecting an empty slot to construct a new element. Upon selection, the system prompts the player to choose between building a *Material Collector* or a *Military Base*.
    - Upgrading existing elements to increase productivity or troop training capacity.
    - Monitoring their base's current status, including materials, soldiers, and defenses.

- Example: Player 1, as "Fazione Rossa," selects an empty slot, chooses to build a *Military Base*, and spends 5 materials to complete the construction.

- **Attacking Another Base:**
  - Players can send troops to enemy bases to attack and reduce their resources or defenses.
  - Example: Player 2, as "Ribelli del Canyon," selects 15 soldiers from their base and sends them to attack Player 1's "Fazione Rossa" base. The server processes the battle a nd updates the status for all players.

- **Resetting the Game:**
  - The administrator can reset the game using the reset functionality provided by the server interface.
  - Example: The admin accesses the reset view, confirms the reset via a security prompt, and all game data is cleared, readying the system for a new session.

9 Conclusions

This project successfully implemented a multiplayer strategy game where up to four players can interact by managing bases, collecting resources, and engaging in battles.

The system was designed with a robust client-server architecture, ensuring that game state synchronization between all connected clients and the server occurs seamlessly. Players are given the option to choose one of four roles, each representing a distinct faction, and are assigned to a base where they can construct and upgrade elements to gain strategic advantages. To streamline gameplay, unused bases automatically initialize with a *Material Collector*, allowing players to start participating immediately.

Key features such as resource collection, troop training, and base defense were implemented with scalability in mind. Special attention was given to error handling and recovery mechanisms to ensure a stable gaming experience. For example, unexpected client disconnections do not affect the overall game, as roles are dynamically reassignable, and server crashes are mitigated by constant data synchronization with a database.

Extensive testing was carried out, including automated unit tests and manual gameplay sessions, to validate the correctness and performance of the system. These tests confirmed that the application behaves as intended, even under prolonged usage and with multiple simultaneous clients.

The project provides clear deployment and usage instructions, making it accessible to users even in a fresh environment. The reset functionality for the administrator adds further flexibility, allowing for quick reinitialization of the game session when needed.

This work demonstrates a comprehensive approach to software design, from initial requirements gathering to final validation, achieving a functional and engaging multiplayer game experience.

9.1 Future Works

While the project successfully implements the core gameplay mechanics and provides a stable multiplayer experience, there are several aspects that could be expanded or refined in the future:

- Expanding the game to support more than 4 players simultaneously. This would require adjustments to the server-side architecture and gameplay mechanics to ensure smooth functionality with increased connections.
- Implement a more professional graphical interface, including adding a background and ensuring a more uniform and visually appealing design.
- Allowing players to place more than 10 *Material Collectors* or *Military Bases* in their base, removing the restriction of predefined slots.
- Adding the option for players to customize their faction name instead of selecting from predefined factions. This would enhance personalization and engagement in the game.
- **Dynamic Map Generation:** The game currently uses a static map. Implementing a dynamic map generation system would add variety and replayability to each session.
- **Mobile and Cross-Platform Support:** Extending the game to mobile platforms or ensuring seamless cross-platform play between desktop and mobile users would broaden the player base.

9.2 What did I learned

This project was a profound learning experience in designing and implementing architectures that combine *event-driven programming* with *request-response communication*. These two paradigms, though distinct, were crucial in shaping the interactions between the client and server in a distributed system. I gained practical experience in understanding their strengths and challenges, particularly in maintaining a seamless flow of information.

The event-driven model was essential for handling asynchronous updates, such as resource collection or troop movements, which occur independently of direct player input. This approach provided a dynamic and reactive system where the server could push updates to clients without waiting for explicit requests. On the other hand, the request-response paradigm was employed for synchronous actions, like purchasing elements or assigning roles, ensuring immediate feedback and clear control over user actions.

One of the most significant takeaways was the complexity of combining these two models effectively. Ensuring that events triggered by the server did not conflict with client requests required careful synchronization and prioritization mechanisms. For instance, I had to address scenarios where simultaneous troop movements and player actions could

lead to race conditions or inconsistent game states. Resolving these conflicts provided hands-on experience in managing communication flows in distributed systems.

Additionally, the project highlighted the importance of designing scalable and resilient communication protocols. By implementing structured message handling on both the client and server sides, I was able to minimize errors and ensure robust fault tolerance. Debugging communication issues and implementing recovery mechanisms for unexpected disconnections further strengthened my understanding of reliable system design.

Ultimately, this project offered invaluable insights into building systems that blend different architectural paradigms while addressing their inherent challenges. The practical experience gained here will be instrumental in future endeavors involving complex, distributed, and interactive systems.