Design Pattern Implementation For a Game Software Architecture

Group 3:
Arda Ege İsker, Baran Özgenç, Elçin Duman, Taner Furkan Göztok

Date:
13.04.2022

Supervised By:
Prof. Dr. Mehmet Akşit

1. Introduction

a) Introduction of the report

In this part, the reason why this report is written will be explained.

This report is written to add new flexibility requirements to the Game [4] in a scenario that our group members are engineers that work for the game company and we were assigned to make the game more flexible with implementing suitable Design Patterns to the required problem.

b) Goal of the report

In this part, the goal is to write this report will be explained.

The goal of this report is to analyze the software architecture of the war game, and to define new flexibility requirements to make the game more scalable, efficient and flexible. And the main goal is to implement the Design Patterns to solve the requirement problems that we define. It is aimed that these flexibility requirements, which will be added while using the design patterns, minimize the realization costs and increase the compatibility within the software of the game. It is aimed that the added design patterns add flexibility to the commercial, quality, functional and system requirements of the project by producing the already existing components in the project, managing their behavior, or making them a structure. It is aimed to provide qualifications and contribute to the elimination of concerns. It is planned that the Design Patterns to be used will not go beyond the solution areas defined in the game report. It is also aimed to explain the reason why the patterns has been chosen in the later parts of the report, together with the presentation of their static structure and dynamic structure.

c) Methodology

In this part, the methodology that has been used to write this report will be explained.

The method of the project is the method of problem solving. In order to realize the problem solving method in the project, design patterns are used and it is aimed to meet the flexibility criteria.

In order to realize the method followed in the project, it is necessary to follow the project steps in an incremental way. The steps of this project start with identifying the problem. After the project is understood, the definition, purpose and requirements of the project are determined. With the determination of the project design, flexibility is provided in the components of the war game by using the design patterns in the project.

In order to use Design Patterns, firstly design patterns were learned by researching. These patterns were primarily created based on the book "Design Patterns: Elements of Reusable Object-Oriented Software" [1] and also benefited from various sources. Design Patterns have three categories. These are Creational Patterns, Behavioral Patterns and Structural Patterns.

The decision of which design pattern will be applied to which component of the project has been determined to meet the flexibility criterion. In the development process that followed, it was aimed to add design patterns to the project with the incremental progression method. In this direction, the design patterns have been integrated within the project in accordance with the requirements, flexibility and purpose criteria. Design patterns will be added by two different methods.

1. Patterns to be implemented on the existing architecture of the game
2. Patterns to be implemented by making changes on the project architecture

It is aimed to prove whether the design pattern applied to a certain component of the war game provides flexibility with the method called “proof of concept”.

d) Contents

In this part, topics in the following sections of the report will be introduced

As mentioned briefly in the introduction section, this report is a document that the engineers working for the game company will benefit from.

In this context, the second part of the report contains information about the raw version of the game that has not been developed by our team. In order to better understand the changes we made on the architecture, the second part contains summary information about the design of the game.

Then, in the third part of the report, the division of labor of the team members, information about the flexibility requirements, the advantages and disadvantages of the candidate patterns are included.

Each team member made 10 Design Pattern proposals and detailed five of them. The next part of the third part of the report is the part where five of these 10 patterns are detailed. This section is divided into three in itself.

- The matching between requirement problem and the design pattern
- Static class diagram of the pattern
- Dynamic application diagram of the pattern

At the end of the third section, there is information about the test scenarios and test results for each detailed pattern, the introduction of the code and its relationship with the structure of the pattern.

The fourth and last part of the report includes explanations about the difficulties faced by the team members in applying the patterns, the problem-pattern association, and the final statement about the report.

2. Introduction of the software architecture

a. Requirements

In this part, project's flexibility requirements will be introduced

In the report, there are 23 different demands and stakeholders who are the addressees of these demands. Stakeholders are obliged to meet the demands that concern them. The purpose of each request is to contribute positively to the project. There are four different types of prompts in the report. These are:

Ticari İsterler: The game must have a financial return and making the game must not exceed the total budget.

Quality Requirements: The game should have a consistent frame rate, the game's graphics should be realistic, the input from the user should reflect smoothly into the game, the network between the players should be secure, the players should not be able to use cheats.

Functional Requirements: Players can interact with in-game objects, there are enemies with artificial intelligence in the game, the game is three-dimensional, there are in-game fragmentable objects, there are multiple difficulty levels in the game, the server can keep the information of the players consistently, the game graphic can be adjusted, the game has a user interface, the game can receive inputs consistently for different platforms, and the add-ons that will come to the game can be downloaded.

System Requirements: The backend of the game is scalable, the game can run on multiple infrastructure platforms, the players can play with each other across platforms, the necessary security protocols are applied for in-game purchases, the game has a flexible structure for later features**.**

The main goal of proposing the design patterns proposed in the 3rd part of the report is to plan the problems that are desired to be solved in the war game project to add flexibility to the project by limiting these requirements. While adding design patterns, it is an important constraint that the pattern will affect and the stakeholders responsible for this demand can easily adapt to the added pattern. There is a matrix on the 21st page of the project report in order to

follow the stakeholder, while adding the design patterns, it is aimed to facilitate the work of the stakeholder related to the demand.

b. Architecture design methodology

In this part the methodology of software architecture's design methodology will be introduced.

To obtain requirements for the game a market analysis is conducted. Market analysis was done to ensure business requirements were in place. Next, the information obtained from market analysis was used to identify stakeholders. The values of stakeholders are considered in this project. In this war game project stakeholders were players, latency stakeholders, graphics experts and more.

After identifying stakeholders, requirements were derived from the perspective of stakeholders and the value proposition of the product. The concerns and problems are inferred from the requirements. How the problems can be addressed is shown in the solution domains. Concerns are also mapped to the solution domain. A list of solution domains are discussed and four of the domains are prioritized: Game Engine, Platform, Game Units, Artificial Intelligence. Solution domain description was made for each subproblem. Constraints were also determined for domains. With all these steps, an architecture was proposed.

Architecture is evaluated using the software architecture analysis method (SAAM). The resilience of the architecture was tested with various test scenarios. The scenarios that were used also changed the requirements of the game. (Synthesis based design, Domain driven design).

c. Introduction of the architecture

In this part, main and sub domains of the game's architecture will be introduced.

The architecture of the game is defined in four parts.

Game Engine

It has been deemed appropriate to use Unity or Unreal Engine game engines to make the game. DirectX is considered for the Graphics API in

the game. Unity defines different colliders based on the shape of the objects, while Unreal Engine defines different colliders based on the type of objects. Since there will be destruction of objects in the defined game, this destruction must be defined according to the object type. That's why Unreal Engine is more suitable for the game engine.

In both game engines, scripts are written to influence in-game behavior. The script language for Unity is C# and for Unreal Engine it is C++.

Various user interfaces are used to make the game more player-friendly. The most commonly used interface is the HUD (Head-Up Display) method of information conveyed to the player using visual aids as part of the game's user interface.

In Game Engine's architecture, Strategy Pattern is integrated under the physics engine to manage the collision state. The strategy design pattern is also preferred for C++ Compiler and Blueprint Compiler and is located under the programming interface. Both compilers can be used interchangeably during runtime.

Game Unit

Game Unit domain contains components such as characters, light sources, weapons, buildings, vehicles, clothes, trees and camera. As weapons and clothing can be changed by in-game purchases, the Decorator Pattern is used to make these objects flexible. A single decorator is shown in the architecture. Composite Pattern used to implement the different game unit types. For example, a game unit can be a character who is wearing some clothes and holding a weapon. This allows game units to be adaptable and extendable. A Strategy pattern is used to manage the model loaders for the different types of graphical modelling tools. The same pattern is used to manage how game units such as characters and vehicles are controlled. There are three different types of controllers. These are real player controller, artificial intelligence controller and vehicle controller. Strategy Pattern is used to set the controller.

Platform

The platform domain includes hosts client, server, database, cloud architecture components and protocols. Layered architecture is used here to flexibly adjust the traffic on the network. Event-driven architecture is preferred in order to efficiently process the events taking place on the game. Cloud architecture is preferred for horizontal scalability. It also has peer-to-peer connectivity and networking solutions where a player is the host.

One of the design patterns used within the scope of the platform is the Strategy design pattern. The cluster of players connected to the game is determined by matching algorithms. Strategy design pattern is preferred for optimal matching.

Artificial Intelligence

The artificial intelligence module is used in such a way that NPC (non-playing characters) components can interact with real user and game environments. Gaming AI is different from pure AI, and it's important to understand this difference. Pure AI aims to create an NPC that users can't beat. Game AI aims to create an NPC that's interesting and hard to beat.

Goal-Based agents have been chosen. It has been chosen because it can be easily adapted, targets and actions can be easily removed and added, and it can work well with complex data sets.

Game artificial intelligence generally consists of 4 components: Sensors, Controllers, Managers and Blackboard. Sensors is a system whose artificial intelligence analyzes the game environment and events around it and is designed as event-driven. Controller is the component that controls the game AI by considering the actions and objectives. There are 2 planners for the controller, these are GOAP (Goal Oriented Action Planning) and SHOP (Simple Hierarchical Ordered Planner) controllers. Managers communicate the NPC controlled by the game AI with other components of the game (Navigation, Weapons, Team, Target and Harm). Finally, Blackboard is responsible for controlling and regulating the communication between all the components mentioned above. Blackboard ensures efficient operation of the system by preventing excessive information transfer.

d. Sub-systems of the architecture

In this part the sub-systems of the architecture will be introduced.

There are many subsystems under the four main systems in architecture. These subsystems exist as part of the architecture to meet the game's requirements.

Physics Engine

Physics Engine is located under Game Engine. The Physics Engine is responsible for making the game look more realistic. With this engine, 3D objects in the game are subject to Newton's laws. The realization of parameters implemented in the Physics Engine can be modeled using object-oriented design patterns.

Protocols

Protocols are defined under the Platform section. There are specific protocols for client and server communication. Since this game is a multiplayer game, it needs a reliable and fast protocol. Platform-specific protocols can also be preferred on consoles such as Xbox and PlayStation.

Database

The database is defined under the Platform section. The progress and status of the game are recorded in the database. In the same way, there are user information, user rankings and data of the game unit in the databases. There is also a database for the client and server side, and it should be preferred that the database be distributed and scaled.

Transactions

There are add-ons that can be purchased in the game. These add-ons can be extra parts of the game, as well as weapons, characters, vehicles and similar products can be purchased in the game. A game-specific currency can be found in the game and purchases can be made with this currency. One of the direct purchases is to purchase as downloadable content. Downloaded content becomes immediately available in the game.

Basically, a transaction mechanism is needed to buy something in the game. This transaction mechanism can be platform-specific, as well as

applications such as Android Market or PlayStation Store. Safety should be considered in these situations.

Inputs

In order to control the game and its environment, the player provides input within the game. These inputs provide a control mechanism within the game. It is the keyboard of a desktop computer, and in console games, the controller is the input of the game. Key mapping is done in 'first-person shooter' type games and it is aimed for the player to learn which command the keys on the keyboard perform in a short time. In some games, this key mapping can be adapted according to the wishes of the person. In mobile games, this may be different.

In addition to all this, the game input and the reactions in the game need to find a quick response in the game. The sooner feedback is given in the game, the more pleasure the players get from the game. Inputs need to be processed specifically in the game.

Authentication

There are in-game purchases available in the game, and players get the items they buy. This way, if an item is special, it can only be found in the inventory of the player who bought it. This process can be achieved simply with a username - password pair.

3) Side System: Game Unit

(Author: Arda Ege İsker)

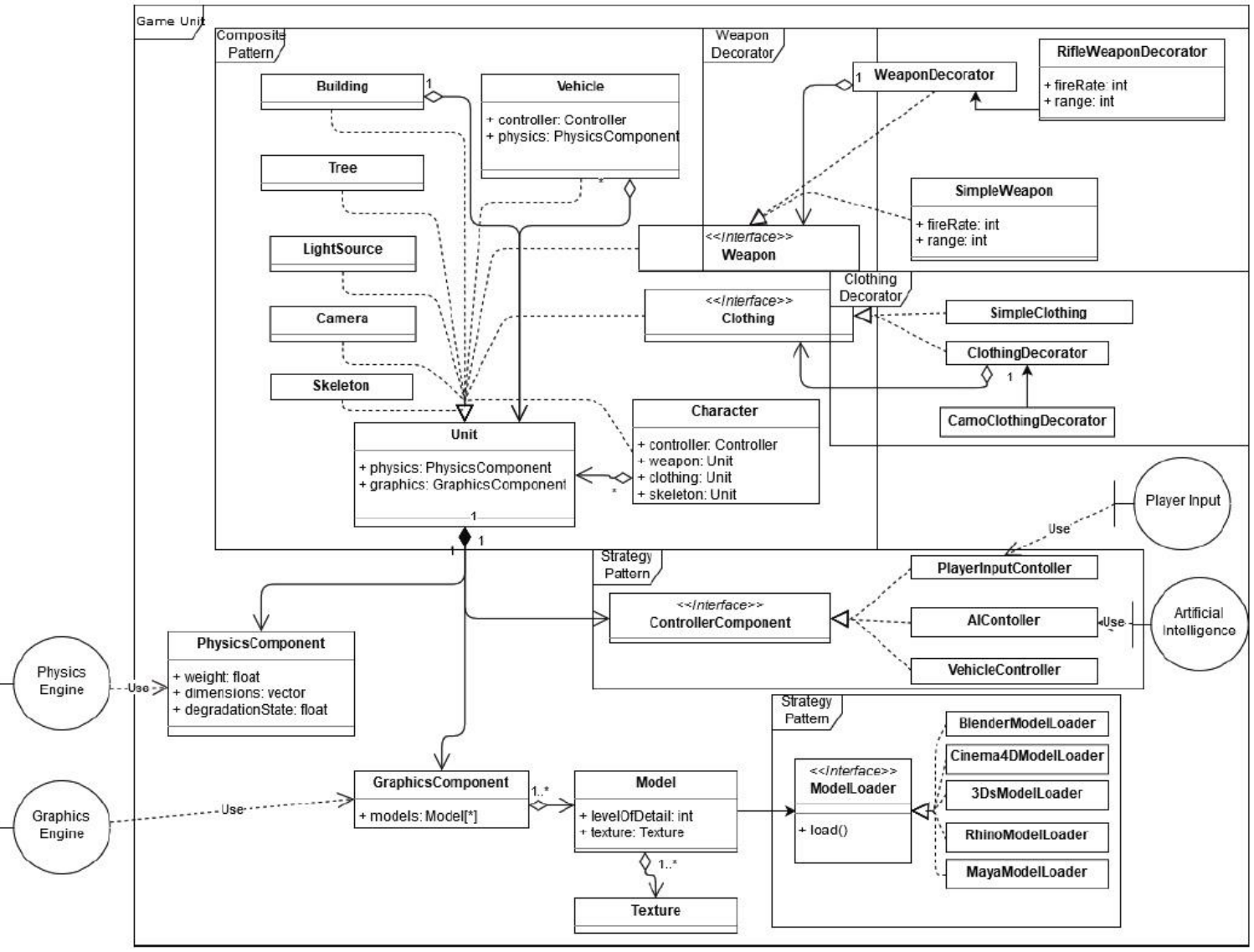
b) Sharing the domains of the game

In this part of the report, the sharing of domains amongst ourselves is going to be explained.

The game's architecture has 4 main domains. Those are Game Engine, Game Unit, Artificial Intelligence and Platform domains. Since we are four people writing this report, each one of us has chosen one domain and made a detailed analysis for the domain we chose, found flexibility requirements on our domains and we found the suitable Design Patterns to solve these specific problems. Design Patterns have been compared to each other and one Design Pattern has been selected to solve a requirement problem. Then each one of us implemented five of these Design Patterns by making their static class diagram, dynamic sequence diagram and writing the Design Patterns code in a programming language. Lastly we tested the implementation if they solve the requirement problems.

c) Flexibility Requirements

In this section, New flexibility requirements will be introduced in order to make the game more flexible, efficient and to make the game's backend more scalable. The report that we have given has been carefully examined and we came to the conclusion that it is not completely disconnected from flexibility, but there are points that can be improved. Game Unit architecture provided by game software architecture design [4] can be seen below.



Flexibility requirements will be defined on the Game Unit domain of the game based on the report that was given to us. While determining the flexibility requirements, the project's Game Unit report has been considered.. Game Unit is frequently linked to Game Engine. When the solution is defined to related requirements, some Game Engine, Artificial Intelligence and Platform components will be used.

1) When an object is destroyed, it should reflect consistently on players.

The map of the game has to be consistent for all players since this is a multiplayer game (R4 in quality requirements). Since the game has destructible objects, each environment frame that players see has to be consistent. To maintain that consistency, when an object is destroyed, players should be informed and that process should be **flexible**.

2) When players buy new clothes and weapons by in-game purchases, Decorator should be configurable.

The server should remember the game state of players. Overall progress should be stored and progress in a stage can be saved at specific moments (R14 in functional requirements). The game might have hundreds of clothes and hundreds of weapons. Also with DLC's these numbers could most probably increase. Each player can have a different cloth-weapon combination. In the game's architecture, Decorator Pattern has been used to equip players with different weapons and clothes. But when new weapons and new clothes added to the game via DLC's, Instead of using every Decorator at once, one Decorator should be used and that one Decorator should be configurable. With that, the game offers a more **flexible** and more scalable solution (R19 in functional requirements) in order to comfort the firm's back end side.

3) If a bug occurs during runtime of the game, game units should fix the bug without communicating with the server.

Game might contain some bugs during runtime. These bugs ruin the realistic view of the game world, the game should offer consistent graphics for all players (R4 in quality requirements). To overcome these bugs during runtime, game units can communicate the server and servers can give directions to fix these bugs correctly. With this solution, communication between game units and server might become a heavy load on the server network by time and extra nodes might be needed to overcome this network burden. Since the game budget is limited (R2 in business requirements) , game units should fix these bugs amongst themselves without using the network. This process should add **flexibility** to server node's load capacity.

4) Adding new maps to the game should be flexible.

This game uses platform method, via DLC's, adding new maps to the game is planned (R1 in business requirements). Newly added maps have to contain the same unit structure that all maps have. Game units are composite structures that contain buildings, trees, light sources, cameras, characters and vehicles. Also Composite pattern has used in the game architecture. Maps consist of these units.

When implementing new maps, this composite structure will be created. The creation of this composite structure should be **flexible** to save some money and time for the firm.

5) The game should reach a large audience.

This game has been planned to be playable on PC, Mobile, Xbox and PS4 hardwares. But since this is a complicated first person shooter game, all hardwares can not run the game smoothly. If the game's system requirements can be reduced without sacrificing the game's graphics quality, more players can play the game. In this way, the firm makes more profit (R1 in business requirements). Reducing the system requirements adds **flexibility** in order to draw more players to play the game.

6) Number of object types on the game unit should be scalable.

The first thing that comes to mind when discussing a structure for objects that share (some) properties is the use of inheritance. The problem with this approach is that game units that are very similar, but still different, will be different instances. This will lead to an unmanageable number of subclasses [4]. If these large number of subclasses cannot be handled efficiently, the game's backend would be more complex and hard to understand for new backend developers that started in the company. To make the game's backend scalable (R19 in system requirements), number of object types on the game should be scalable and **flexible**.

7) When the game ends, all game units should be terminated in order to relieve the game server's memory.

After players play the game on a map, the game should be over when certain conditions are met. When the game is over, all game units should be terminated because the game server holds all game units in its memory (R14 in functional requirements). This process can be **flexible** if the termination process of all game units can be done efficiently.

8) Elements of game unit that relate to each other should interact efficiently amongst themselves.

Every subpart of game unit unites on one unit object. That unit object can change due the state of the game. These subparts that make up the unit frequently interact with each other. This interaction should be made in an efficient way in order to play on higher framerate (R3, R5 in quality requirements). More efficient this interaction gets, the load on the hardware that runs the game reduces. If the interaction efficiency provided, it adds **flexibility** that the load on game server nodes reduces and client's hardware that runs the game relieves.

9) Creating same type of NPC characters should be efficient.

The game contains NPCs (non-playable characters) as enemy soldiers and ally soldiers (R9 in functional requirements). Since they are not played by some real player, the variety of NPC types is smaller compared to real time player types. Since the variety of NPC types are small and the total NPC number can be very high according to the state of the game, it should be efficient to create the same type of NPCs. This efficiency should add **flexibility** for the game's backend (R19 in system requirements).

10) Changing clothes in gameplay should be limited in order to avoid confusion.

Since this is a war game, it is played at least two sided. Opposite sided players must not have the same shade of a color in order to avoid confusion. To obtain that stability and to make the game more **flexible** on reliability, the game should provide a limited colorway on the player's clothes.

d)Possible Design Patterns to Solve Requirements

In this part, candidate design patterns to solve flexibility requirement problems will be introduced using table.

Flexibility Requirement	**Possible Design Patterns Solutions**
When an object is destroyed, it should reflect consistently on players	Observer, Publisher - Subscriber
When players buy new clothes and weapons by in-game purchases, Decorator should be configurable	Strategy, Bridge, State
If a bug occurs during runtime of the game, game units should fix the bug without communicating with the server	Chain of Responsibility, Interpreter, Memento
Adding new maps to the game should be flexible	Factory Method, Builder
The game should reach a large audience	Flyweight, Singleton
Number of object types on game unit should be scalable	Interpreter, State
When the game ends, all game units should be terminated in order to relieve the game server's memory	Visitor, Iterator
Elements of game unit that relate to each other should interact efficiently amongst themselves.	Mediator, Facade
Creating same type of NPC	Factory Method, Prototype

characters should be efficient	
Changing clothes in gameplay should be limited in order to avoid confision	State, Chain of Responsibility

In this part, candidate design patterns to solve flexibility requirement problems will be compared and one pattern will be chosen.

1) When an object is destroyed, this should reflect consistently on players.

For this problem, Observer and Publisher - Subscriber Patterns are chosen as the candidate patterns to solve the requirement.

The reason that Observer pattern selected as a candidate pattern is, players act like subscribers to the environment, when there is a destruction happened in the environment, players get a notification to update their game to keep their game consistent and every player must see the current state of the game. Observer pattern could be implemented to apply this subscription analogy.

Another candidate pattern for this problem is publisher - subscriber pattern. The problem is a real publisher and subscriber analogy. But in publisher - subscriber method the publisher and subscriber don't know about the existence of one another. There is a third component, called broker or message broker or event bus, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly[3]. Also publisher - subscriber pattern can filter the notification that will be sent. But filtering is not necessary in this problem. In this problem there is no third component in the middle between subjects and objects.

Observer Pattern has been chosen to solve this problem.

2) When players buy new clothes and weapons by in-game purchases, Decorator should be configurable.

For this problem, Strategy, Bridge and State Patterns are chosen as the candidate patterns to solve the requirement.

The reason that Strategy Pattern selected as candidate pattern is, because every equipment combination might need a specific algorithm, and that algorithm has to be changeable at runtime. Strategy pattern is suitable for this kind of problem.

The reason that Bridge Pattern selected as candidate pattern is, because Bridge Pattern splits a large class or a set of closely related classes into two separate hierarchies-abstraction and implementation-which can be developed independently of each other [2] . So that each cloth-weapon combination can be treated independently by Unit object.

The reason that State Pattern select as candidate pattern is because every cloth-weapon combination can be treated as a state so that when the state is changed, the algorithm that the cloth-weapon combination required can change.

Strategy Pattern has been chosen to solve this problem.

3) If a bug occurs during runtime of the game, game units should fix the bug without communicating with the server.

For this problem, Chain of Responsibility, Interpreter Pattern and Memento Patterns are chosen as the candidate patterns to solve the requirement.

Each game unit has two main components, these are the physics component and the graphics component. The bug might be related to each one component or both of these components.

The reason that Chain of Responsibility Pattern selected as candidate pattern is, these two components are objects in Game Unit structure, when a bug occurs on runtime, to detect if a bug occurs, these components can communicate with each other and solve their bugs with communication.

The reason that Interpreter Pattern selected as candidate pattern is, it can detect the possible bug on parse time and it can prevent the bug from happening. Preventing bugs before they happen adds more flexibility to this project, so that players experience more stable games. But implementing this pattern requires designing a suitable grammar for the game, and it might take a long time to develop.

The reason that Memento Pattern selected as a candidate pattern is because like many other games, this game consists of consecutive game states. When a bug

occurs in a state, it must be revertable to escape the bug state. Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Memento Pattern has been chosen to solve this problem.

4) Adding new maps to the game should be flexible.

For this problem, Factory Method and Builder Patterns are chosen as the candidate patterns to solve the requirement.

Since game units are composite structures, creational design patterns can add some flexibility.

The reason that Factory Method Pattern selected as candidate pattern is to avoid tight coupling between the creator and the concrete products, also with Single Responsibility Principle, creating all game units can be in just one class.

The reason that Builder Pattern selected as candidate pattern is, composite pattern is used in game unit structure, builder pattern is flexible when creating complex Composite trees because you can program its construction steps to work recursively [2]. Also Builder Pattern creates objects step-by-step, when the structure changes on runtime before creating the units, it adds extra flexibility.

Builder Pattern has been chosen to solve this problem.

5) The game should reach a large audience.

For this problem, Flyweight and Singleton Patterns are chosen as the candidate patterns to solve the requirement.

Since every interactable component of the game is a game unit, making adjustments on the game unit's structure can lower the game's system requirement.

The reason that Flyweight Pattern selected as candidate pattern is to implement shared leaf nodes of the Composite tree as Flyweights to save some RAM. So that the game requires less RAM to run. Some components of this unit structure are very similar to each other. For example, buildings and trees, light sources

and cameras are related to each other and these two fields store almost identical data across all particles. But with Flyweight Pattern, extrinsic state storage adds flexibility to total required RAM.

The reason that Singleton Pattern selected as candidate pattern is the same reason why Flyweight Pattern was selected. In order to reduce all shared states of the similar objects to just one Flyweight object. But it does not add any flexibility. And Singleton Pattern requires special treatment in a multithreaded environment like this war game so that multiple threads won't create a singleton object several times [2].

Flyweight Pattern has been chosen to solve this problem.

6) Number of object types on game unit should be scalable.

For this problem, Interpreter and State Patterns are chosen as the candidate patterns to solve the requirement.

The reason that Interpreter Pattern selected as candidate pattern is that with Interpreter Pattern, a grammar can be used to define what configurations of game units are allowed and what combinations are not allowed. This requires the people working with game units to understand the language, but if a combination is invalid, the game engine can detect this using the grammar. Simple changes such as new units can be defined in the language, while other more complex changes would only require a change in the language and grammar.

The reason that State Pattern selected as candidate pattern is because a state structure can be defined and when a subclass is added and if it makes an exception, the state structure can prevent that subclass from being created.

Interpreter Pattern has been chosen to solve this problem.

7) When the game ends, all game units should be terminated in order to relieve the game server's memory.

For this problem, Visitor and Iterator Patterns are chosen as the candidate patterns to solve the requirement.

The reason that Visitor Pattern selected as candidate pattern is because Composite Pattern has been used in the game unit architecture. Game units are composite structures that contain buildings, trees, light sources, cameras, characters and vehicles. Visitor Pattern can be used to execute a termination operation over an entire Composite tree. Since each game unit object in a different structure, separate algorithms might be needed to terminate them.

The reason that Iterator Pattern selected as candidate pattern is because iterator object can traverse elements of game unit structure without exposing its underlying representation and terminate each object it traverses.

Visitor Pattern has been chosen to solve this problem.

8) Elements of game unit that relate to each other should interact efficiently amongst themselves.

For this problem, Mediator and Facade Patterns are chosen as the candidate patterns to solve the requirement.

The reason that Mediator Pattern selected as candidate pattern is because Mediator pattern can be used to reduce dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object [2]. In the game unit architecture, all components of the game unit unites on the Unit object. Unit object can act like a mediator in this scenario.

The reason that Facade Pattern selected as candidate pattern is because a subclass of the Unit object can be a facade object that makes a simplified interface between the communication of game unit objects.

Mediator Pattern has been chosen to solve this problem.

9) Creating same type of NPC characters should be efficient.

For this problem, Factory Method and Prototype Patterns are chosen as the candidate patterns to solve the requirement.

The reason that Factory Method Pattern selected as candidate pattern is because The Factory Method can separate NPC creation code from the code that actually

uses the NPC. Therefore if new features are added via DLC's, it's easier to extend the NPC creation code independently from the rest of the code.

The reason that Prototype Pattern selected as candidate pattern is because Prototype pattern lets the game's backend copy existing NPCs easily with the clone method. With the use of Prototype Pattern, only the prototype NPC types will be created explicitly by the game's server. After the prototype NPCs are created, thousands of NPCs can be cloned based on the prototypes. Also with Prototype Pattern, Cloning objects without coupling to their concrete classes flexibility is possible.

Prototype Pattern has been chosen to solve this problem.

10) Changing clothes in gameplay should be limited in order to avoid confusion.

For this problem, State and Chain of Responsibility Patterns are chosen as the candidate patterns to solve the requirement.

The reason that State Pattern selected as candidate pattern is because each team can be represented as a state and it can have its own colorway rules on the player's clothes.

The reason that Chaşin of Responsibility Pattern selected as candidate pattern is because each player on the sam team can pass a colorway request along a chain of handlers. Upon receiving a colorway request, each handler decides either to process the colorway request as changing their clothes color to a suitable colorway or to pass it to the next handler in the chain.

State Pattern has been chosen to solve this problem.

e) Problem Description

In this part, We decided to solve the problems that comply with the flexibility requirements 1, 3, 5, 7 and 9. For this process, we will first define problems suitable for these flexibility requirements. Then we will extract the static class diagram and the application dynamic diagram to validate the implementation we will do with the Java programming language.

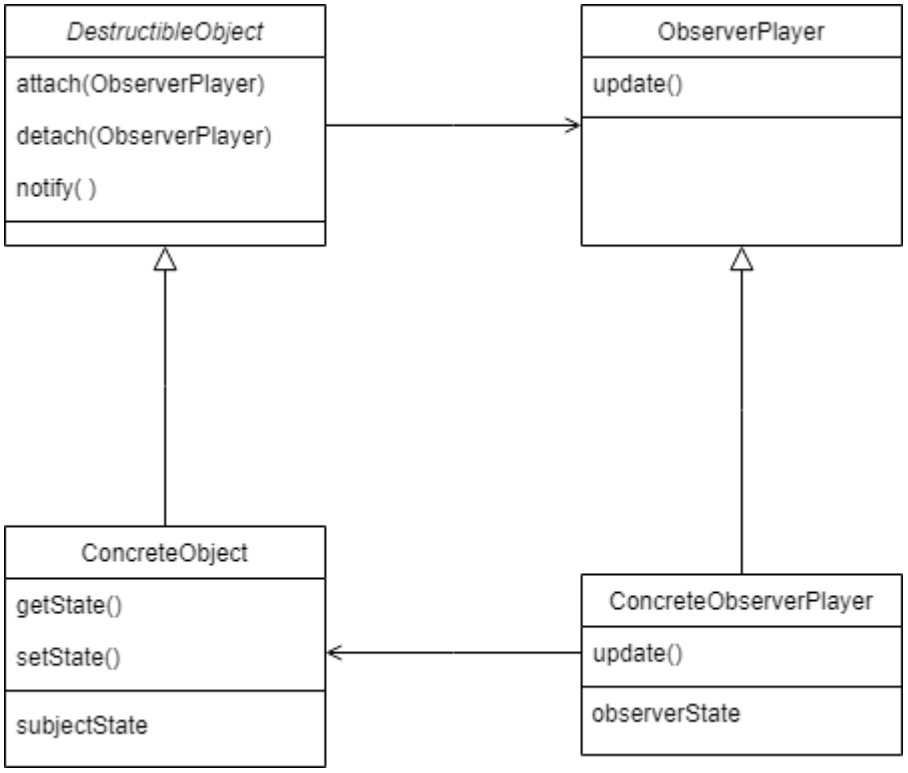
Pattern 1)

Problem Number 1: When an object is destroyed, it should reflect consistently on players

Since the game has destructible objects, each environment frame that players see has to be consistent. To acquire that, when an object is destroyed, players should get a notification to update their game state.

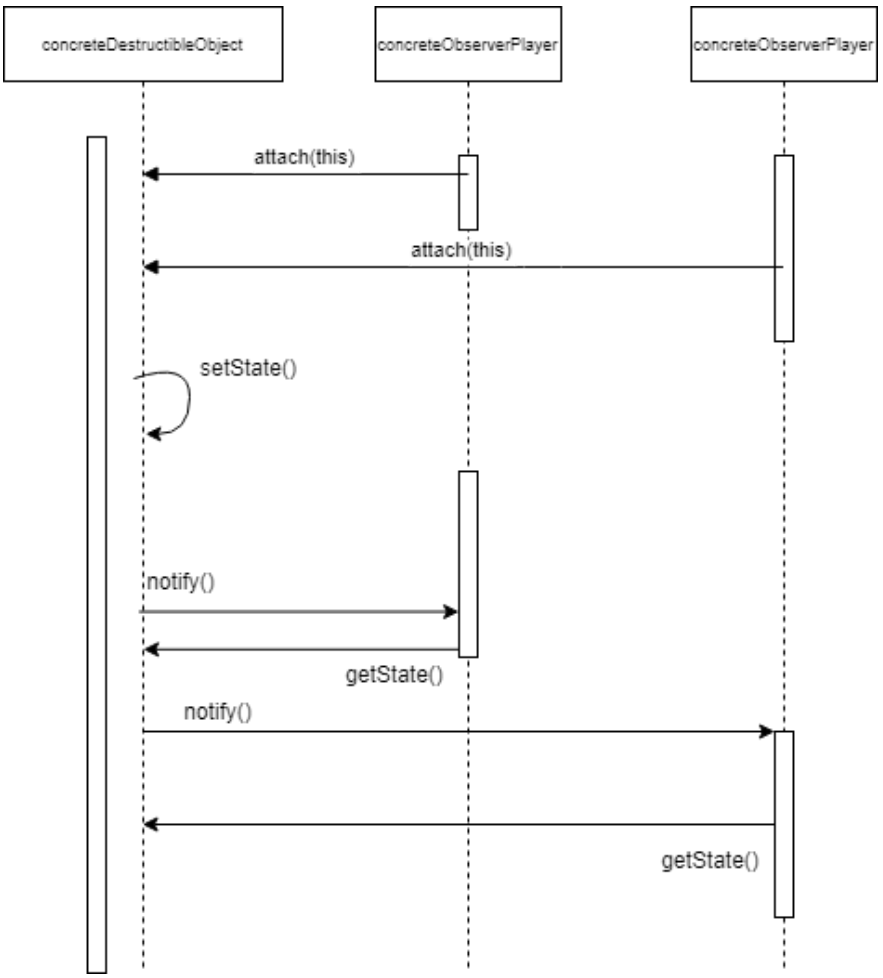
According to Design Patterns book [1], Observer Pattern intends to use one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The subject is every destructible object in the game and observers are all players. Players act like subscribers to the environment objects, when destruction happens in the environment, players get a notification to update their game to keep their game consistent and every player must see the current state of the game. Observer pattern will be implemented to apply this subscription analogy.

Static Class of Observer Pattern



The table above contains the UML class diagram for the Observer Design Pattern that will be used to solve the one-to-many communication between destructible objects and observer players. DestructibleObject and ObserverPlayer classes are Interfaces. ConcreteObject and ConcreteObserverPlayer classes are concrete implementations.Firstly the DestructibleObject issues events of interest to other objects. These events occur when the DestructibleObject changes its state or executes some destruction behaviors. DestructibleObject contains a subscription infrastructure that lets new ObserverPlayers join and current subscribers leave the list. The ObserverPlayer interface declares the notification interface.it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update. ConcreteObserverPlayers perform update their game in response to notifications issued by the DestructibleObject . All of these classes must implement the same interface so the DestructibleObject isn't coupled to concrete classes.[2]

Dynamic Structure of Application



The table above contains the sequence diagram for the Observer Design Pattern that will be used to solve the requirement problem. The UML sequence diagram shows the run-time interactions. In this example, concreteDestructibleObject acts as subject and concreteObserverPlayers are subscribed to that object. When a destruction happens to that object, it notifies its subscribers so that players can update their game (Returns are not shown).

Pattern 2)

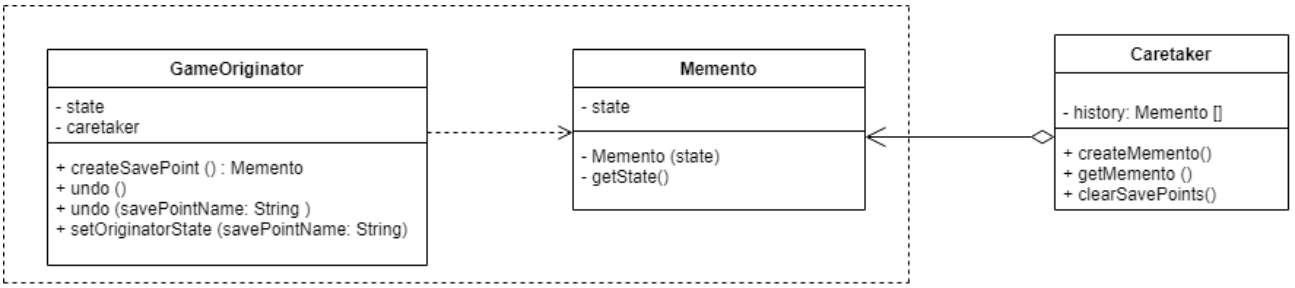
Problem Number 3: If a bug occurs during runtime of the game, game units should fix the bug without communicating with the server

Game might contain some bugs during runtime. These bugs ruin the game should offer consistent graphics for all players. To overcome these bugs during runtime, returning the previous game state of buggy state algorithm will be applied.

The Memento pattern delegates creating the state snapshots to the actual owner of that state, the originator object. Hence, instead of other objects trying to copy the editor's state from the "outside," the editor class itself can make the snapshot since it has full access to its own state.

The pattern suggests storing the copy of the object's state in a special object called memento. The contents of the memento aren't accessible to any other object except the one that produced it. Other objects must communicate with mementos using a limited interface which may allow fetching the snapshot's metadata. [2]

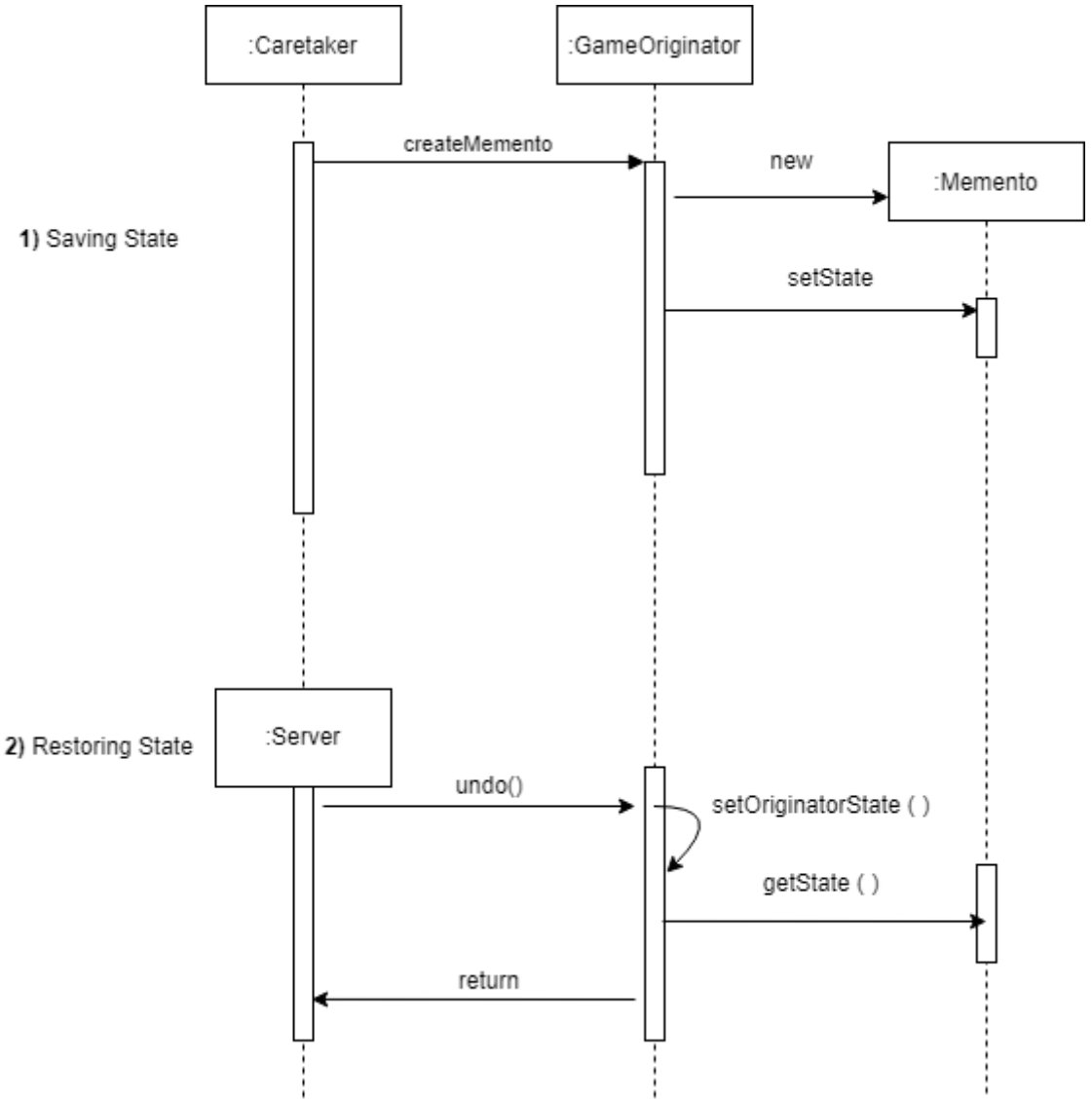
Static Class of Memento Pattern



The table above contains the UML class diagram for the Memento Design Pattern that will be used to escape from buggy game state. The GameOriginator class can produce snapshots of its own state, as well as restore its state from snapshots when needed. The Memento is a value object that acts as a snapshot of the originator's state. It's a common practice to make the memento immutable and pass it the data only once, via the constructor. Caretaker can keep track of the GameOriginator's history by storing a stack of mementos. When the GameOriginator' has to travel back in game history, the caretaker fetches the topmost memento from the stack and passes it to the originator's restore method. In this implementation, the Memento class is inside the GameOriginator. This

lets the GameOriginator access the fields and methods of the memento, even though they're declared private. On the other hand, the caretaker has very limited access to the memento's fields and methods, which lets it store mementos in a stack but not tamper with their state.

Dynamic Structure of Application



The table above contains the sequence diagram for the Memento Design Pattern that will be used to solve the requirement problem. The UML sequence diagram shows the run-time interactions. In this example, when the caretaker wants to create a memento snapshot, it makes a call to GameOriginator. GameOriginator

creates the memento and sets the game's current state. In the second scenario, when a bug occurs, Game server represented by Server make sundo call to revert the game to the latest state, GameOriginator sets its state then it makes a call to the Memento to get a state and returns that state to the server.

Pattern 3)

Problem Number 5: The game should reach a large audience.

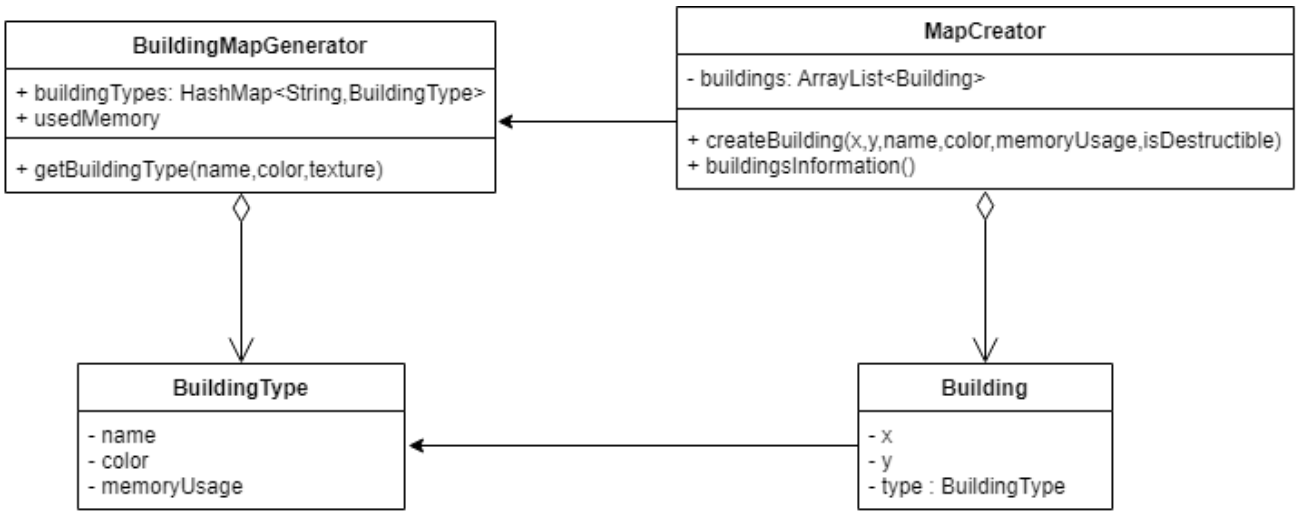
If we want to reach a larger player audience, the game's minimum system requirement should not be high. To reduce minimum system requirement, we can reduce the memory that the game uses. Each game unit is drawn to the map and the system that runs the game has to memorize all game units on its RAM. If we manage to reduce total RAM usage of the game, the minimum RAM requirement for the game will be reduced.

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object [2]. The constant data of an object is usually called the intrinsic state. It lives within the object; other objects can only read it, not change it. The rest of the object's state, often altered "from the outside" by other objects, is called the extrinsic state.

The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts. As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic [2].

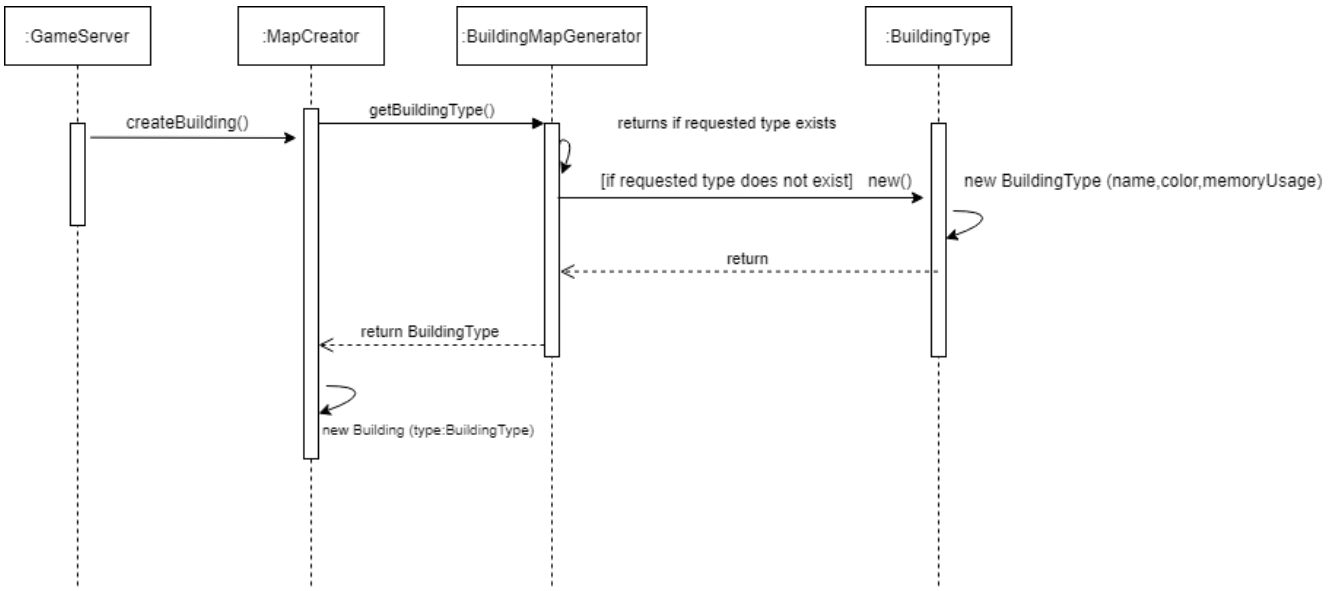
In this example, the Flyweight Pattern helps to reduce memory usage when rendering hundreds of building objects on a game map.

Static Class of Flyweight Pattern



The table above contains the UML class diagram for the Flyweight Design Pattern. BuildingType is the flyweight class contains a portion of the state of a building. These fields store values that are unique for each particular building type. Since the color data is usually big, it would be waste of memory by keeping it in each building object. Instead, we can extract color into a separate object which lots of individual building objects can reference. BuildingMapGenerator is the class that creates new BuildingTypes and chooses whether to reuse existing Flyweight or to create a new object. Building object is the contextual object contains the extrinsic part of the building state. Server can create hundreds of these since they consume little memory, just two integer coordinates and one reference field. Building class is the client of flyweight. MapCreator object contains buildings on a map. MapCreator is also the client of flyweight. Only one BuildingType is created for each type of building.

Dynamic Structure of Application



The table above contains the sequence diagram for the Flyweight Design Pattern that will be used to solve the requirement problem. Firstly, when the game server wants MapCreator to create a building, it makes createBuilding call. Then the MapCreator makes a request for the BuildingMapGenerator object to get a specific typed building. If that type of building exists on BuildingMapGenerator's cache, it returns that type of the requested building. If it does not exist on BuildingMapGenerator's cache, it makes a call to the BuildingType flyweight object to create desired building type and adds that BuildingType to its cache. BuildingMapGenerator returns the requested building type to the MapCreator and MapCreator creates such building that the game server requested.

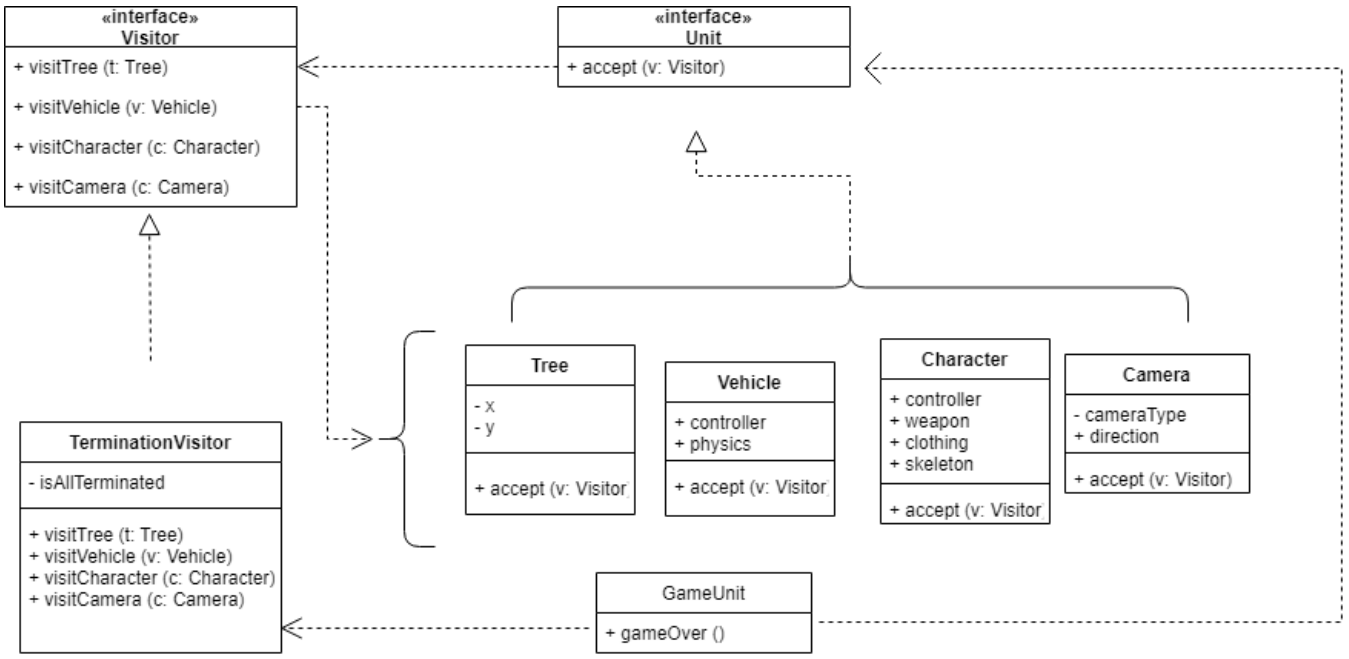
Pattern 4)

Problem Number 7: When the game ends, all game units should be terminated in order to relieve the game server's memory

Since this is an online game, the game's servers keep the game's current state on its memory. When the game ends, it is a waste of memory to keep the current game state on the server's memory. To relieve this wasted memory, all game

units should be terminated. Since game unit's structure uses Composite Pattern, a Visitor object will be used to execute termination operation over the entire Composite tree. Each subpart object of the game unit might need a different algorithm to execute before termination. For example, Tree object can easily terminated but Character object's points can be needed before termination operation. Visitor Pattern lets separate algorithms from the objects on which they operate.

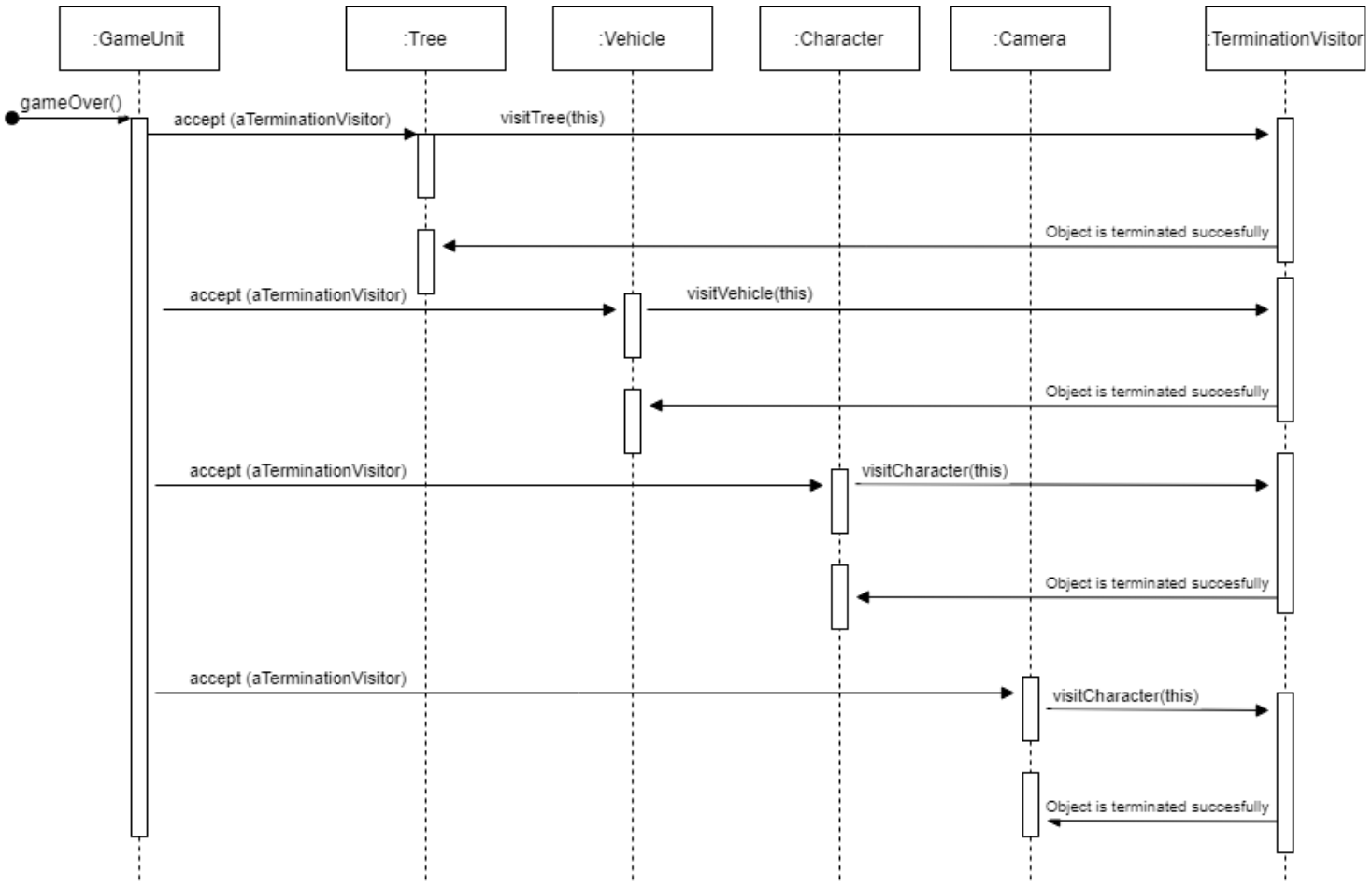
Static Class of Visitor Pattern



The table above contains the UML class diagram for the Visitor Design Pattern. The Visitor interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. TerminationVisitor implements the Visitor interface. The Unit interface declares accept method. The accept method should have one parameter declared with the type of the visitor interface. Each game unit object must implement the Unit interface. The purpose of the accept method is to redirect the call to the visitor's method corresponding to the current element class. Each unit object needs separate algorithm to terminate itself. The

GameUnit object represents the Game Unit composite tree on the game's architecture. GameUnit object makes the gameOver call when the game is over.

Dynamic Structure of Application



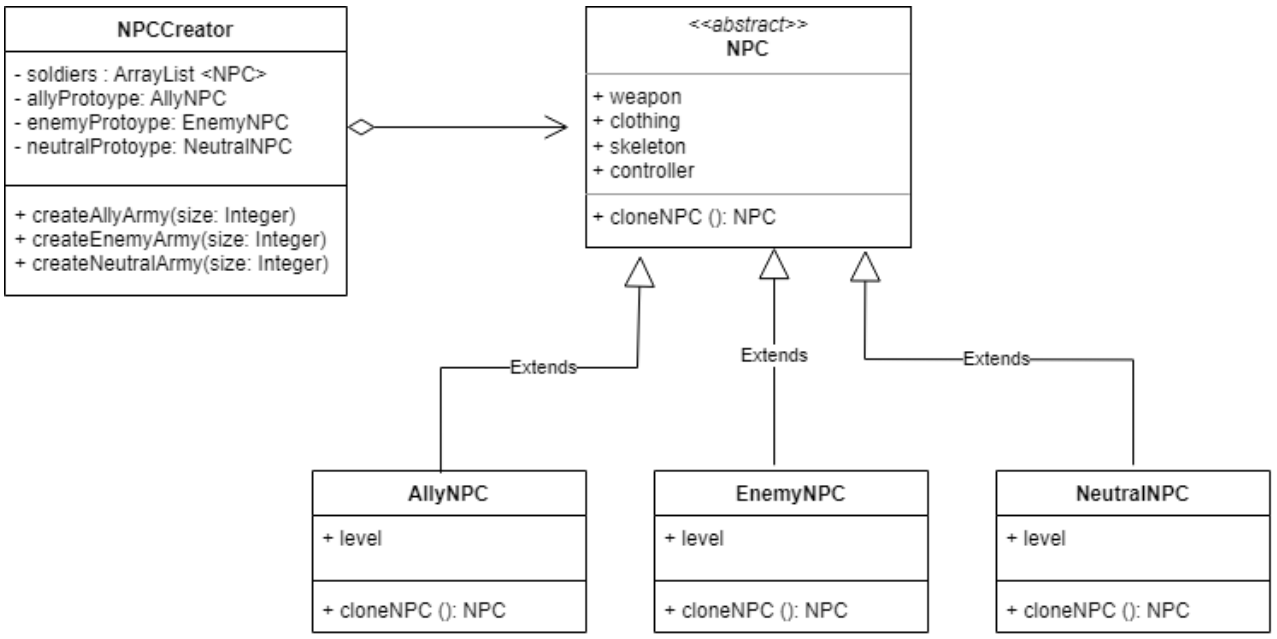
The table above contains the sequence diagram for the Visitor Design Pattern that will be used to solve the requirement problem. First, when the game is over, the game's server makes gameOver call to the GametUnit object. Then GameUnit makes accept visitor call on all Objects in the composite structure. These objects make the appropriate visit call to the TerminationVisitor visitor object. Lastly, the TerminationVisitor object terminates each game unit object the way it requires.

Pattern 5)

Problem Number 9: Creating same type of NPC characters should be efficient

The game contains a large number of NPCs. NPCs are created based on NPC types such as ally NPC, enemy NPC, neutral NPC, warpilot NPC, marine NPC, etc. Total NPC number might be high but the NPC type is limited by a number by developers due design limits. Just like the players, NPCs also have weapons, clothes, skeletons and artificial intelligence controllers. Each NPC contains substantial data because they have weapons, clothing, skeletons, and controllers. NPCs are created at the beginning of each game. Some game modes might contain thousands of NPC objects. Their creation should be serialized and should be scalable to start the game as fast as possible. To do so, the Prototype Pattern delegates the cloning process to the actual NPC objects that are being cloned.

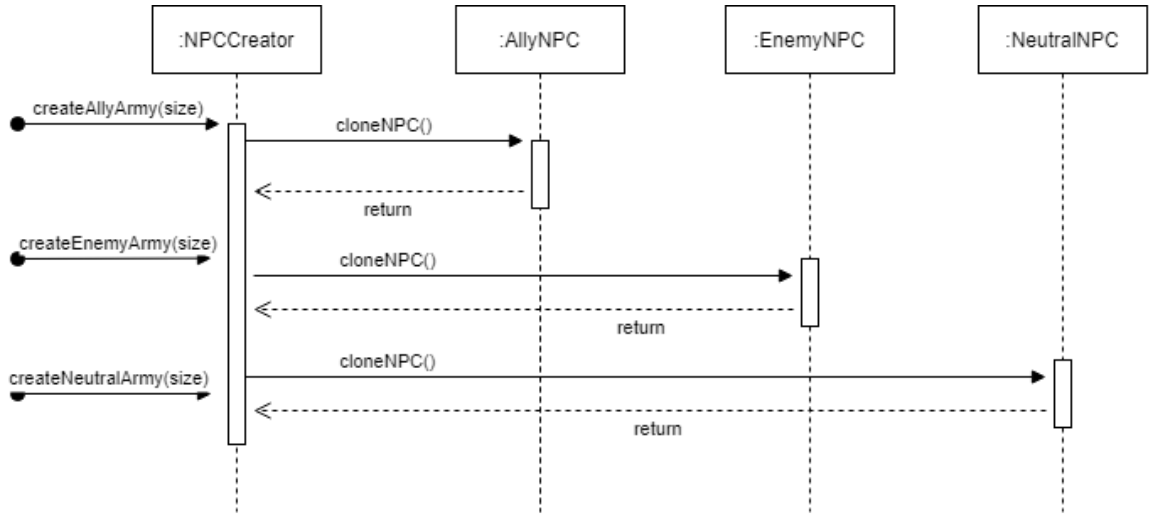
Static Class of Prototype Pattern



The table above contains the UML class diagram for the Prototype Design Pattern. The NPC abstract class declares the NPC's weapon, clothing, skeleton and controller type. NPC class also declares the cloneNPC method. AllyNPC, EnemyNPC and NeutralNPC extend the NPC class; they hold the NPC's level as

a variable. Those subclasses override the cloneNPC method. NPCCreator is the class that creates prototype of each NPC type, NPCCreator also holds each copied NPC on its soldiers list. NPCCreator has create army methods to create armies on specified type.

Dynamic Structure of Application



The table above contains the sequence diagram for the Prototype Design Pattern that will be used to solve the requirement problem. First, NPCCreator gets the createArmy call to create a specified NPC with the number of size that comes from its parameter. Then NPCCreator makes cloneNPC calls to the specified classes and clones the prototype instance variables that NPCCreator has.

f) Implementing the Design Patterns

In this part, the implementation of the design patterns to be used in the solution of the problems with the Java programming language will be emphasized. The structure of individual object classes, instance variables and methods will be examined. The problem-specific hypothetical modeling will be done in the test part.

Pattern 1) When an object is destroyed, it should reflect consistently on players (Observer Pattern)

DestructibleObject.java

```
public interface DestructibleObject {
   void attach(ConcreteObserverPlayer observer);
   void detach(ConcreteObserverPlayer observer);
   void notifyObservers();
}
```

This interface contains attach,detach and notify methods.

ObserverPlayer.java

```
public interface ObserverPlayer {
   void update(boolean isDestroyed);
}
```

This interface contains update method.

ConcreteObject.java

Instance Variables

```
public class ConcreteObject implements DestructibleObject {
   List<ConcreteObserverPlayer> observers = new
ArrayList<>();
   private String name;
   private boolean subjectState;
```

Concreteobject has its name as a string, current state value that if this object has been destroyed or not subjectState which is observed by other objects and a list which contains all observers which are subscribed before.

Constructor

```
public ConcreteObject(String name, boolean isDestroyed){
   this.name = name;
   this.subjectState = isDestroyed;
}
```

Constructor method of the ConcreteObject object needs title and initial data for creating a new Subject.

Attach and Detach

```
public void attach(ConcreteObserverPlayer observer){
   observers.add(observer);
   observer.update(subjectState);
}
public void detach(ConcreteObserverPlayer observer){

   observers.remove(observer);
}
```

The attach method allows a new observer to subscribe to the current subject. The detach method, on the other hand, breaks an existing observer's subscription to the current subject.

Notify Observers and Update

```
public void notifyObservers(){
   for(ConcreteObserverPlayer observer : observers){
       observer.update(subjectState);
   }
}
```

Setter and Getter

```
void setState(boolean isDestroyed){
   this.subjectState = isDestroyed;
   notifyObservers();
}
public boolean getState(){
   return subjectState;
```

```
}
```

With the update method, the state of the current subject can be updated. After this process, the notify method is called automatically. The notify method sends a notification to all observer players which are subscribed to that destructible subject.

ConcreteObserverPlayer.Java

The codes in the Observer.java class are given below with their explanations.

Instance Variables

```
public String name;
public ConcreteObject subject = new
ConcreteObject("Destructible Object X", false);
public boolean isSubjectDestroyed;
```

Concrete observer object has its name as a string, observed Subject object and current state of observed subject.

Constructor

```
public ConcreteObserverPlayer(String name){
   this.name = name;
   isSubjectDestroyed = false;
}
```

Construction method to initialize the name of the Observer and state of its objects current state (initially false) .

Update

```
public void update(boolean isDestroyed) {
   if (isSubjectDestroyed != isDestroyed) {
```

```
        System.out.println(name + "'s Observer Updated");
        System.out.println(name + " is updating its game
state.");

System.out.println("----------------------------------------
---");

    }
    isSubjectDestroyed = isDestroyed;
}
```

Subject notifies its subscribers after updating their data. As a result of this notification, observers update their data with this update method.

Test Scenario

In this part, Observer Design Pattern that is implemented in the Java language to solve the problem we defined will be tested. Before starting the test, we need to determine the test requirements in order to observe whether the code we have implemented meets the basic design pattern requirements, if it adds any flexibility and how successful it is in solving the problem. In this part of the report, test requirements will be determined for the test work

1. **Observers who will follow the destructible Object must be subscribers.**

   If the players in the game need the state of the destructible object's state instantly, they can subscribe to these destructible objects and be informed about instant updates with the help of the observer design pattern.

2. **Subscribers are informed as the destruction state of the destructible object has changed.**

   Players that are currently subscribed to destructible objects are notified of any updates with the help of observer design pattern

Test Results

In this section of the report, the success of the implementation we made for the solution of the problem using the Java programming language and design pattern using the test requirements we determined in the previous section will be evaluated. For this process, code fragments, the outputs of these code fragments and their evaluations will be found in the implementation.

The Java source code for testing the implementation will be shown below.

```
public class Test {
   public static void main(String[] args) {
      DestructibleSubject BuildingSubject = new DestructibleSubject("Building",false);    //Destructible
// Subject "Building" created.

       ObserverPlayer player1 = new ObserverPlayer("Player 1");
       ObserverPlayer player2 = new ObserverPlayer("Player 2");
       ObserverPlayer player3 = new ObserverPlayer("Player 3");        //Observer Players are created.
       ObserverPlayer player4 = new ObserverPlayer("Player 4");
       ObserverPlayer player5 = new ObserverPlayer("Player 5");

       BuildingSubject.attach(player1);
       BuildingSubject.attach(player2);
       BuildingSubject.attach(player3);        ///All observers attach (being subscribers) to the
//current Subject
       BuildingSubject.attach(player4);
       BuildingSubject.attach(player5);

       BuildingSubject.updateState(true);      //Building is now destroyed.
       BuildingSubject.updateState(true);      //Building is set to destroyed again, players shouldn't
                                               //react to this

       BuildingSubject.detach(player1);        //Player 1 and Player 2 are no more subscribers.
       BuildingSubject.detach(player3);

       System.out.println("\n-------------Destruction is now set to false.-------------\n");

       BuildingSubject.updateState(false);
       System.out.println("\n\n----Each player's view of the destructible building----");
       System.out.println("Is the building has been destruct on Player 1's view: " +
player1.isSubjectDestroyed);
       System.out.println("Is the building has been destruct on Player 2's view: " +
player2.isSubjectDestroyed);
       System.out.println("Is the building has been destruct on Player 3's view: " +
player3.isSubjectDestroyed);
       System.out.println("Is the building has been destruct on Player 4's view: " +
player4.isSubjectDestroyed);
       System.out.println("Is the building has been destruct on Player 5's view: " +
player5.isSubjectDestroyed);

   }
}
```

In the test code, Players and Building are in the Game Unit Part by modeling a small part of the project instead of creating the whole project in order to test the test requirements determined in the previous section.

Firstly, a ConcreteObject object to represent a destructible building is created. Then Concrete observer players are created. Then all the players are attached to the ConcreteObject object. Then building is set to destroyed state, later building is set to destroyed again to test flexibility to see if subscribed players get notification again. But they did not get a notification. Then, Player 1 and Player 3 are detached, the building is set to not destroyed state. Lastly, to see the difference between subscribers and non subscribers, the building's view for every player is printed.

Test Output:

```
Player 1's Observer Updated
Player 1 is updating its game state.
-----------------------------------------
Player 2's Observer Updated
Player 2 is updating its game state.
-----------------------------------------
Player 3's Observer Updated
Player 3 is updating its game state.
-----------------------------------------
Player 4's Observer Updated
Player 4 is updating its game state.
-----------------------------------------
Player 5's Observer Updated
Player 5 is updating its game state.
-----------------------------------------

------------Destruction is now set to false.-------------

Player 2's Observer Updated
Player 2 is updating its game state.
-----------------------------------------
Player 4's Observer Updated
Player 4 is updating its game state.
-----------------------------------------
Player 5's Observer Updated
Player 5 is updating its game state.
-----------------------------------------
```

```
----Each player's view of the destructible building----
Is the building has been destruct on Player 1's view: true
Is the building has been destruct on Player 2's view: false
Is the building has been destruct on Player 3's view: true
Is the building has been destruct on Player 4's view: false
Is the building has been destruct on Player 5's view: false

Process finished with exit code 0
```

The output of the piece of code written for the test is as above. As seen, when the first destruction occurs, every Player updated their game state, then Player 1 and Player 3 are detached, they did not get the building's new not destroyed state. At the last part of the output, Player 1 and Player 3 do not have the latest state of observed building.

Pattern 2) If a bug occurs during runtime of the game, game units should fix the bug without communicating with the server (Memento Pattern)

State.java

This class is created to represent the game's states.

Instance Variables

```
String stateDate;
String stateName;
```

These variables are representing the game's state date time and game's current state name.

Constructor

```
public State( String stateDate,String stateName){
```

```
    this.stateDate = stateDate;
    this.stateName = stateName;
 }
```

Setters and Getters

```
void setStateDate(String stateDate){
   this.stateDate = stateDate;
}
String getStateDate(){
   return stateDate;
}
```

GameOriginator.java

Instance Variables

```
private State state;
private String lastUndoSavepoint;
Caretaker careTaker;
```

It holds game's current state, the name of the game's last save state. And a Caretaker object.

Constructor

```
public GameOriginator(String stateDate,String
stateName,Caretaker careTaker){
   state = new State(stateDate,stateName);
   this.careTaker = careTaker;
   createSavepoint("INITIAL");
}
```

Setters and Getters

```
public State getState(){
   return state;
}
```

```
public void setState(State state) {
   this.state = state;
}
```

Create Savepoint

```
public void createSavepoint(String savepointName){
   careTaker.saveMemento(new
Memento(state.stateDate,state.stateName), savepointName);
   lastUndoSavepoint = savepointName;
}
```

This method creates a savepoint using the game's current state. If the current game state is buggy, it is not recommended to save that state.

Undo

```
public void undo(){
   setOriginatorState(lastUndoSavepoint);
}

public void undo(String savepointName){
   setOriginatorState(savepointName);
}

public void undoAll(){
   setOriginatorState("INITIAL");
   careTaker.clearSavepoints();
}
```

Undo methods are used to load a game state that is not buggy.

If it gets "`String savepointName`" argument, it loads a specific savepoint.

undoAll method clears all save records and sets the game state to initial state. It has been implemented to clear cache memory.

Set Originator State

```
private void setOriginatorState(String savepointName){
   Memento mem = careTaker.getMemento(savepointName);
```

```
    State dummy = mem.getState();
    this.state.stateDate = dummy.stateDate;
    this.state.stateName = dummy.stateName;
}
```

This method is used to set the Game Originator's current state.

Memento Class

Memento class is stored inside GameOriginator.java.

Instance Variable

```
private State s;
```

It holds a state to memorize.

Constructor

```
public Memento(String stateDate,String stateName){
    s = new State(stateDate,stateName);
}
```

Getter

```
public State getState(){
    return s;
}
```

Caretaker.java

Instance Variable

```
private final Map<String, Memento> gamePointStorage = new
HashMap<String, Memento>();
```

Save memento and Get Memento

```
public void createMemento(Memento memento,String
```

```
savepointName){
   System.out.println("Saving state..."+savepointName);
   gamePointStorage.put(savepointName, memento);
}

public Memento getMemento(String savepointName){
   System.out.println("Undo at ..."+savepointName);
   return gamePointStorage.get(savepointName);
}
```

Save memento is used to save current state as a memento.

Get memento is used to get a specific memento.

Clear Savepoints

```
public void clearSavepoints(){
   System.out.println("Clearing all save points...");
   gamePointStorage.clear();
}
```

Test Scenario

In this part, Memento Design pattern that is implemented in the Java language to solve the problem we defined will be tested. Before starting the test, we need to determine the test requirements in order to observe whether the code we have implemented meets the basic design pattern requirements, if it adds any flexibility and how successful it is in solving the problem. In this part of the report, test requirements will be determined for the test work

1) Game states must be stored as snapshots

Each state that has been saved should be stored in a data structure. Specific snapshots should be accessible at any point.

2) Save records must be labeled.

Each save record must be labeled with a name for further access.

3) When a bug occurs, the game's state must be revertable.

When a bug occurs during runtime of the game, the game's state should be revertible to fix the bug.

Test Results

In this section of the report, the success of the implementation we made for the solution of the problem using the Java programming language and design pattern using the test requirements we determined in the previous section will be evaluated. For this process, code fragments, the outputs of these code fragments and their evaluations will be found in the implementation.

The Java source code for testing the implementation will be shown below.

```
public class TestMementoPattern {
   public static void main(String[] args) {
       Caretaker careTaker = new Caretaker();
       GameOriginator originator = new GameOriginator("12:00", "Default Game
State", careTaker); //default game state initialized

       System.out.println("Default State: "+originator);

       originator.setState(new State("16:02", "State 1")); //game's    //state
has been changed.

       System.out.println("---------------\n"+originator);
       originator.createSavepoint("SAVE 1"); //current state saved

       originator.setState(new State("16:10", "Buggy Game State")); //The
game's current state has been set as buggy.
       System.out.println("---------------\n"+originator);

       originator.undo();    //undo method used to escape buggy state.
       System.out.println("---------------\nAfter undo: "+originator);

       originator.setState(new State("17:02", "State 3"));
```

```
        originator.createSavepoint("SAVE2");
        System.out.println("---------------\n"+originator);
        originator.setState(new State("17:11", "State 4"));
        originator.createSavepoint("SAVE3");
        System.out.println("---------------\n"+originator);
        originator.setState(new State("18:32", "State 5"));
        originator.createSavepoint("SAVE4");
        System.out.println("---------------\n"+originator);

        originator.undo("SAVE2"); // Go back to the state that has been //saved
as “SAVE2”
        System.out.println("Retrieving at: "+originator);

        originator.undoAll(); //Clears the all save records and goes back //to
the default game state.
        System.out.println("State after undo all: "+originator);
    }
}
```

In the test code, Game states are modeled as a small part of the game state instead of creating the whole game state in order to test the test requirements determined in the previous section.

Firstly, a Caretaker object to represent a debugger is created. Then Game Originator is created. Then new states added to the record, a buggy state has added and with undo method, it reverted to the old game state to test flexibility to see if Game recovers itself from bugs. Then, new states have been added. Lastly, with undoAll method all save records have been deleted.

Test Output:

Saving state...INITIAL

Default memento.State: Game's memento.State Date: 12:00, memento.State Name: Default Game memento.State

---

Game's memento.State Date: 16:02, memento.State Name: memento.State 1

Saving state...SAVE 1

Game's memento.State Date: 16:10, memento.State Name: Buggy Game memento.State

Undo at ...SAVE 1

After undo: Game's memento.State Date: 16:02, memento.State Name: memento.State 1

Saving state...SAVE2

Game's memento.State Date: 17:02, memento.State Name: memento.State 3

Saving state...SAVE3

Game's memento.State Date: 17:11, memento.State Name: memento.State 4

Saving state...SAVE4

Game's memento.State Date: 18:32, memento.State Name: memento.State 5

Undo at ...SAVE2

Retrieving at: Game's memento.State Date: 17:02, memento.State Name: memento.State 3

Undo at ...INITIAL

Clearing all save points...

memento.State after undo all: Game's memento.State Date: 12:00, memento.State Name: Default Game memento.State

Process finished with exit code 0

The output of the piece of code written for the test is as above. As seen, the game can save itself from a buggy game state. At the end of the day, all save points can be deleted to preserve the system's RAM.

Pattern 4) The game should reach a large audience. (Flyweight Pattern)

Building.java

Instance Variables

```
public class Building {
    private int x;
    private int y;
    private boolean isDestructible;
    private BuildingType type;
```

Building class contains x and y variables as its coordinates, information of if it is destructible, and the flyweight variable type.

Constructor

```
public Building(int x, int y, BuildingType type,boolean
isDestructible) {
   this.x = x;
   this.y = y;
   this.type = type;
   this.isDestructible = isDestructible;
}
```

Constructor method of the Building object needs initial data for creating a new Building.

BuildingType.java

Instance Variables

```
public class BuildingType {
   private String name;
   private String color;
   private int memoryUsage;
```

BuildingType contains name,color and memoryUsage variables. Since this is the flyweight class of the pattern, color variable represents relatively big data in real life implementations.

Constructor

```
public BuildingType(String name, String color, int memoryUsage)
{
   this.name = name;
   this.color = color;
   this.memoryUsage = memoryUsage;
}
```

Constructor method of the BuildingType object needs initial data for creating a new BuildingType.

BuildingMapGenerator.Java

Instance Variables

```
public class BuildingMapGenerator {
   public static Map<String, BuildingType> buildingTypes = new
HashMap<>();
   public static int usedMemory = 0;
```

buildingTypes variable is a HashMap for remembering existing building types as a cache. usedMemory variable is total memory used by all building types on cache.

getBuildingType

```
public static BuildingType getBuildingType(String name, String color,
int  memoryUsage) {
   BuildingType result = buildingTypes.get(name);
   if (result == null) {
       result = new BuildingType(name, color, memoryUsage);
       buildingTypes.put(name, result);
       usedMemory += memoryUsage;
   }
   return result;
}
```

This method takes a specific building type properties as its parameters and if that building type does not exist on the HashMap, it creates a new BuildingType object and adds it to its cache. This method works like Factory Method.

MapCreator.java

Instance Variables

```
public class MapCreator  {
   private List<Building> buildings = new ArrayList<>();
```

buildings ArrayList holds created buildings as a list.

createBuilding

```
public void createBuilding(int x, int y, String name, String
color, int memoryUsage,boolean isDestructible) {
   BuildingType type = BuildingMapGenerator.getBuildingType(name,
color, memoryUsage);
   Building building = new Building(x, y, type,isDestructible);
   buildings.add(building);
}
```

This method creates a new building according to information that comes from its parameter. It gets the requested building type using BuildingMapGenerator's getBuildingType method.

Test Scenario

In this part, Flyweight Design Pattern that is implemented in the Java language to solve the problem we defined will be tested. Before starting the test, we need to determine the test requirements in order to observe whether the code we have implemented meets the basic design pattern requirements, if it adds any flexibility and how successful it is in solving the problem. In this part of the report, test requirements will be determined for the test work

1. **Holding building types data in a separate class must provide efficient RAM consumage.**

   The reason is that this pattern is chosen to make more efficient RAM usage. To ensure that, total RAM usage when this pattern is implemented should be less than its old RAM usage.

2. **Holding building types as a separate class should not cause any semantic problem on map.**

   Since building types are being hold in a separate class, a merge operation is needed when creating a new building object. This operation should not cause any problem.

Test Results

In this section of the report, the success of the implementation we made for the solution of the problem using the Java programming language and design pattern using the test requirements we determined in the previous section will be evaluated. For this process, code fragments, the outputs of these code fragments and their evaluations will be found in the implementation.

The Java source code for testing the implementation will be shown below.

```
package flyweight;
import java.util.Random;

public class TestFlyweight {
   public static void main(String[] args) {
       Random rand = new Random();
       MapCreator creator = new MapCreator();
       for (int i = 0; i < 500000; i++) {

creator.createBuilding(rand.nextInt(1000),rand.nextInt(1000),
"Destructible Building",
                   "Red",32,true);

creator.createBuilding(rand.nextInt(1000),rand.nextInt(1000),
"Non-destructible Building",
                   "Blue",4,false);
       }
       creator.buildingsInformation();
```

```
    }
}
```

In the test code, only two building types are used. Those are red and blue buildings. The red building type is destructible and it consumes 32 KB of RAM. The blue building type is non-destructible and it consumes 4 KB of RAM.

Firstly, a random number generator has been used to give each building object random coordinates. Then 500000 objects for each building type, totally 1000000 buildings have been created. Lastly, buildingsInformation method has been to visualize test results. (buildingsInformation method is only written to visualize test results. It has not been added to the report.)

Test Output:

```
There are 2 types of building exist (shared memory)

There are 1000000 buildings in total.

 Flyweight Pattern Memory Usage           Without Flyweight Pattern

--------------------------------         ------------------------

(36 + 1000000) KB                        (36000000+1000000) KB

--------------------------------         ------------------------

= 1000036 KB                                   = 37000000 KB

Process finished with exit code 0
```

As seen on the results, when that pattern is used, 1000036 KB of RAM has been used, when the pattern is not used, 37000000 KB of RAM has been used. When the pattern is used , the flyweight data (32+4 = 36 KB data) is only added once. But without pattern, that data is consumed for each building object additionally. The Flyweight Pattern is %3699 more efficient on RAM consumage.

Pattern 5) When the game ends, all game units should be terminated in order to relieve the game server's memory (Visitor Pattern)

Visitor.java

```
public interface Visitor {
   public void visitTree(Tree t);
   public void visitVehicle(Vehicle v);
   public void visitCharacter(Character c);
   public void visitCamera(Camera c);
}
```

This interface contains the visitor methods.

Unit.java

```
public interface Unit {
   public void accept(TerminationVisitor v);
}
```

This interface contains the accept method.

TerminationVisitor.java

Instance Variable

```
public class TerminationVisitor implements Visitor {
   boolean isAllTerminated;
```

This variable holds the state of all game unit objects are terminated.

Constructor

```
public TerminationVisitor(boolean isAllTerminated){
   this.isAllTerminated = isAllTerminated;
}
```

Constructor method for TerminationVisitor class.

Visit methods

```
public void visitTree(Tree t){
        t.x = null;
        t.y = null;
```

```
        t = null;
        System.gc();
    }
    public void visitVehicle(Vehicle v){
        v.controller = null;
        v.physics = null;
        v = null;
        System.gc();
    }
    public void visitCharacter(Character c){
        c.controller = null;
        c.weapon = null;
        c.clothing = null;
        c.skeleton = null;
        c = null;
        System.gc();
    }
    public void visitCamera(Camera c){
        c.setCameraType(null);
        c.direction = null;
        c = null;
        System.gc();
        isAllTerminated = true;
    }
```

For each visit method, it assigns null to the object's variables and the object itself that comes from parameter. After null evaluations are finished, Java's garbage collector is called to free the memory.

Tree.java

Instance Variables

```
public class Tree implements Unit{
   String x;
   String y;
```

x and y strings are being held to represent the tree's coordinates.

Constructor

```
public Tree (String x, String y){
    this.x = x;
    this.y = y;
}
```

accept Method

```
@Override
public void accept(TerminationVisitor v) {
   v.visitTree(this);
}
```

accept method of Tree makes visitTree call to the TerminationVisitor and gives itself as parameter.

Vehicle.java

Instance Variables

```
public class Vehicle implements Unit{
    public String controller;
    public String physics;
```

controller and physics strings are being held to represent the Vehicle's reference objects.

Constructor

```
public Vehicle(String controller, String physics){
    this.controller = controller;
    this.physics = physics;
}
```

accept Method

```
@Override
public void accept(TerminationVisitor v) {
   v.visitVehicle(this);
}
```

accept method of Vehicle makes visitVehicle call to the TerminationVisitor and gives itself as parameter.

Character.java

Instance Variables

```
public class Character implements Unit{
   public String controller;
   public String weapon;
   public String clothing;
   public String skeleton;
```

controller , weapon, clothing and skeleton strings are being held to represent the Character's reference objects.

Constructor

```
public Character(String controller,String weapon, String clothing,
String skeleton){
   this.controller = controller;
   this.weapon = weapon;
   this.clothing = clothing;
   this.skeleton = skeleton;
}
```

accept Method

```
@Override
public void accept(TerminationVisitor v) {
   v.visitCharacter(this);
}
```

accept method of Character makes visitCharacter call to the TerminationVisitor and gives itself as parameter.

Camera.java

Instance Variables

```
private String cameraType;
public String direction;
```

cameraType and direction strings are being held to represent the Camera's reference objects.

Constructor

```
public Camera(String cameraType, String direction){
   this.cameraType = cameraType;
   this.direction = direction;
}
```

accept Method

```
@Override
public void accept(TerminationVisitor v) {
   v.visitCamera(this);
}
```

accept method of Camera makes visitCamera call to the TerminationVisitor and gives itself as parameter.

GameUnit.java

Instance Variables

```
public class GameUnit {
   Tree t;
   Vehicle v;
   Character c;
   Camera cam;
```

Each object type is being held as a variable.

Constructor

```
public GameUnit(Tree t, Vehicle v, Character c, Camera cam){
   this.t = t;
   this.v = v;
   this.c = c;
```

```
    this.cam =cam;
}
```

gameOver Method

```
public void gameOver(TerminationVisitor visitor){
   t.accept(visitor);
   v.accept(visitor);
   c.accept(visitor);
   cam.accept(visitor);
}
```

gameOver object makes all objects to call their accept method and make them accept them the TerminationVisitor visitor object.

Test Scenario

In this part, Visitor Design Pattern that is implemented in the Java language to solve the problem we defined will be tested. Before starting the test, we need to determine the test requirements in order to observe whether the code we have implemented meets the basic design pattern requirements, if it adds any flexibility and how successful it is in solving the problem. In this part of the report, test requirements will be determined for the test work

1. **When the game ends, all objects should be terminated and swiped from the memory.**

   The reason is that this pattern is chosen to make Visitor to execute an operation over an entire Composite tree. When the game ends, visitor should execute a termination operation to all game unit objects and free the memory that being used by these objects.

Test Results

In this section of the report, the success of the implementation we made for the solution of the problem using the Java programming language and design pattern using the test requirements we determined in the previous section will be

evaluated. For this process, code fragments, the outputs of these code fragments and their evaluations will be found in the implementation.

The Java source code for testing the implementation will be shown below.

```
public static void main(String[] args) {
   Tree tree = new Tree("500","200");
   System.out.println("Tree object is created.");
   Vehicle vehicle = new Vehicle("Vehicle Controller","VehiclePhysics");
   System.out.println("Vehicle object is created.");
   Character character = new
Character("PlayerInputController","Rifle","Camo","Soldier Skeleton");
   System.out.println("Character object is created.");
   Camera camera = new Camera("3D Camera", "South");
   System.out.println("Camera object is created.");
   GameUnit gameUnit = new GameUnit(tree,vehicle,character,camera);
   System.out.println("The game has begun...");
   // THE GAME IS BEING PLAYED AT THE MOMENT.....
   System.out.println("The game is over, TerminationVisitor is calling to
terminate objects.");
   TerminationVisitor visitor = new TerminationVisitor(false);
   gameUnit.gameOver(visitor);
   System.out.println("All Game Unit objects terminated successfully, deleted on
memory by garbage collector.");
   System.out.println("Tree Object Variables : " + tree.x);
   System.out.println("Vehicle Object Variables : " + vehicle.controller);
   System.out.println("Character Object Variables : " + character.skeleton);
   System.out.println("Camera Object Variables : " + camera.direction);
   System.out.println("Is all objects on Game Unit Terminated ? : " +
visitor.isAllTerminated);

}
```

In the test code, all the variables from Game Unit objects and their variables are modeled as a small part of the game state instead of creating the whole Game Unit Objects in order to test the test requirements determined in the previous section.

Firstly, an instance is created from Tree, Vehicle, Character and Camera classes. Then a GameUnit object is created. Then the game started, when the game is over, a TerminationVisitor object is created and then GameUnit's gameOver method is called. With that method call, TerminationVisitor visited all of the Game Unit objects and terminated them properly.

Test Output:

```
Tree object is created.

Vehicle object is created.

Character object is created.

Camera object is created.

The game has begun...

The game is over, TerminationVisitor is calling to terminate objects.

All Game Unit objects terminated successfully, deleted on memory by garbage collector.

Tree Object Variables : null

Vehicle Object Variables : null

Character Object Variables : null

Camera Object Variables : null

Is all objects on Game Unit Terminated ? : true

Process finished with exit code 0
```

As seen on the results, when the game is over, Visitor executed a termination operation over the entire Composite structure successfully and freed the memory usage that Game Unit objects used to have.

Pattern 5) Creating same type of NPC characters should be efficient (Prototype Pattern)

NPC.java

Instance Variables

```
public abstract class NPC {
    public String weapon;
    public String clothing;
```

```
    public String skeleton;
    public String controller;
```

NPC abstract class contains weapon, clothing, skeleton and controller types for the NPC object as String (holds the variables as String to model the implementation).

Constructors

```
public NPC (){ }

public NPC (NPC source){
   if (source != null){
       this.weapon = source.weapon;
       this.clothing = source.clothing;
       this.skeleton = source.skeleton;
       this.controller = source.controller;
   }
}
```

It has two constructor methods. First constructor is the empty constructor to create the first prototype object. The second constructor is the main constructor method to create clone objects that takes a prototype object as parameter.

cloneNPC

```
public abstract NPC cloneNPC();
```

It is the abstract cloning method.

AllyNPC.java

Instance variable

```
public int level;
```

It is the variable to keep the NPC's level.

Constructors

```
public AllyNPC (){ }
public AllyNPC(AllyNPC protoypeNPC) {
    super(protoypeNPC);
    if(protoypeNPC != null)
        this.level = protoypeNPC.level;
}
```

It has two constructor methods. First constructor is the empty constructor to create the first prototype object. The second constructor is the main constructor method to create clone objects that takes a prototype object as parameter.

cloneNPC

```
public NPC cloneNPC(){
    return new AllyNPC(this);
}
```

This clone method creates new AllyNPC, gives itself to constructor parameter and returns the cloned object.

EnemyNPC.java

Instance variable

```
public int level;
```

It is the variable to keep the NPC's level.

Constructors

```
public EnemyNPC (){ }
public EnemyNPC(EnemyNPC protoypeNPC) {
    super(protoypeNPC);
    if(protoypeNPC != null)
        this.level = protoypeNPC.level;
}
```

It has two constructor methods. First constructor is the empty constructor to create the first prototype object. The second constructor is the main constructor method to create clone objects that takes a prototype object as parameter.

cloneNPC

```
public NPC cloneNPC(){
   return new EnemyNPC(this);
}
```

This clone method creates new EnemyNPC, gives itself to constructor parameter and returns the cloned object.

NeutralNPC.java

Instance variable

```
public int level;
```

It is the variable to keep the NPC's level.

Constructors

```
public NeutralNPC (){ }
public NeutralNPC(NeutralNPC protoypeNPC) {
   super(protoypeNPC);
   if(protoypeNPC != null)
       this.level = protoypeNPC.level;
}
```

It has two constructor methods. First constructor is the empty constructor to create the first prototype object. The second constructor is the main constructor method to create clone objects that takes a prototype object as parameter.

cloneNPC

```
public NPC cloneNPC(){
   return new NeutralNPC(this);
}
```

This clone method creates new NeutralNPC, gives itself to constructor parameter and returns the cloned object.

NPCCreator.java

Instance Variables

```
public class NPCCreator {
    private AllyNPC allyPrototype;
    private EnemyNPC enemyPrototype;
    private NeutralNPC neutralPrototype;
    private ArrayList <NPC> soldiers = new ArrayList<>();
```

A prototype for each NPC type is being held as an instance variable. Also a soldiers list is being held to keep all cloned soldiers list.

Constructor

```
public NPCCreator(){
    allyPrototype = new AllyNPC();
    allyPrototype.weapon = "Combat Uniform";
    allyPrototype.clothing = "Regular";
    allyPrototype.skeleton = "RigidBody";
    allyPrototype.controller = "AIController";
    allyPrototype.level = 20;

    enemyPrototype = new EnemyNPC();
    enemyPrototype.weapon = "Pistol";
    enemyPrototype.clothing = "Camo Uniform";
    enemyPrototype.skeleton = "RigidBody";
    enemyPrototype.controller = "AIController";
    enemyPrototype.level = 30;

    neutralPrototype = new NeutralNPC();
    neutralPrototype.weapon = "Knife";
    neutralPrototype.clothing = "Service Uniform";
    neutralPrototype.skeleton = "RigidBody";
    neutralPrototype.controller = "AIController";
    neutralPrototype.level = 40;

}
```

Prototype objects are created in the constructor method. Their variables are easily configurable before runtime.

createArmy methods

```
public void createAllyArmy(int size){
   for (int i = 0; i<size;i++){
       soldiers.add(allyPrototype.cloneNPC());
   }
   System.out.println(size + " Ally NPC Created.");
}
public void createEnemyArmy(int size){
   for (int i = 0; i<size;i++){
       soldiers.add(enemyPrototype.cloneNPC());
   }
   System.out.println(size + " Enemy NPC Created.");
}

public void createNeutralArmy(int size){
   for (int i = 0; i<size;i++){
       soldiers.add(neutralPrototype.cloneNPC());
   }
   System.out.println(size + " Neutral NPC Created.");
}
```

createArmy methods create the size that comes from the parameter amount of NPCs of their kind. Each createArmy method calls their kind of prototype method to create armies.

Test Scenario

In this part, Prototype Design Pattern that is implemented in the Java language to solve the problem we defined will be tested. Before starting the test, we need to determine the test requirements in order to observe whether the code we have implemented meets the basic design pattern requirements, if it adds any flexibility and how successful it is in solving the problem. In this part of the report, test requirements will be determined for the test work

1. **Creating new NPCs should be scalable and efficient.**

   It should be scalable and efficient to create new NPC objects on the Game's server side.

2. **NPC's variables should be configurable easily.**

NPC variables should be configurable easily with changing just one NPC variable.

Test Results

In this section of the report, the success of the implementation we made for the solution of the problem using the Java programming language and design pattern using the test requirements we determined in the previous section will be evaluated. For this process, code fragments, the outputs of these code fragments and their evaluations will be found in the implementation.

The Java source code for testing the implementation will be shown below.

```
public static void main(String[] args) {
   NPCCreator creator = new NPCCreator();
   creator.createAllyArmy(100);
   creator.createAllyArmy(100);
   creator.createAllyArmy(300);
   creator.createAllyArmy(300);
   creator.createAllyArmy(300);
   creator.createAllyArmy(300);
   creator.createAllyArmy(300);
   creator.createAllyArmy(300);

   creator.createEnemyArmy(300);
   creator.createEnemyArmy(300);
   creator.createEnemyArmy(300);
   creator.createEnemyArmy(300);
   creator.createEnemyArmy(300);

   creator.createNeutralArmy(200);
   creator.createNeutralArmy(200);
   creator.createNeutralArmy(200);
   creator.createNeutralArmy(200);
   creator.createNeutralArmy(200);

   System.out.println("There are " + creator.soldiers.size() + " NPCs exist on the
game");
   int allyNumber = 0;
   int enemyNumber = 0;
   int neutralNumber = 0;
   for (NPC npc: creator.soldiers) {
       if (npc instanceof AllyNPC)
           allyNumber++;
       if (npc instanceof EnemyNPC)
           enemyNumber++;
```

```
        if (npc instanceof NeutralNPC)
            neutralNumber++;
    }
    System.out.println("There are " + allyNumber + " Ally NPCs exist on the game");
    System.out.println("There are " + enemyNumber + " Enemy NPCs exist on the
game");
    System.out.println("There are " + neutralNumber + " Neutral NPCs exist on the
game");

}
```

In the test code, all the variables from NPC objects are modeled as strings instead of creating the whole Weapon, Cloth, Controller and Skeleton classes in order to test the test requirements determined in the previous section.

Firstly, a NPCCreator object has been created and the prototype objects are created in the constructor method of NPCCreator. The NPC's variables can be configurable on the constructor method. Than some amount of armies created for each NPC type. Then the total amount of NPC and also total amount of NPCs for each NPC type has been printed.

```
Test Output:

100 Ally NPC Created.

100 Ally NPC Created.

300 Ally NPC Created.

300 Ally NPC Created.

300 Ally NPC Created.

300 Ally NPC Created.

300 Ally NPC Created.

300 Ally NPC Created.

300 Ally NPC Created.

300 Enemy NPC Created.

300 Enemy NPC Created.

300 Enemy NPC Created.

300 Enemy NPC Created.

300 Enemy NPC Created.
```

```
200 Neutral NPC Created.

200 Neutral NPC Created.

200 Neutral NPC Created.

200 Neutral NPC Created.

200 Neutral NPC Created.

There are 4500 NPCs exist on the game

There are 2000 Ally NPCs exist on the game

There are 1500 Enemy NPCs exist on the game

There are 1000 Neutral NPCs exist on the game

Process finished with exit code 0
```

As seen on the test results, 4500 NPC objects have been created easily with Prototype Pattern. Their types are also specified. Configuring the NPC's variables can be donewith changing their values on NPCCreator's constructor method.

4) What We Learned and Conclusions

In this section, we will explain what we learned in this course, the problems we encountered in the project and the results we reached.

a) Knowledge about Patterns

I had no knowledge of design patterns before taking this course. In this course, I learned the definition of pattern, its intended use, basic design patterns, the flexibilities provided by patterns and the downsides they bring. In addition to the theoretical knowledge written in the books, I learned how to apply patterns to real life problems, how to approach and solve problems. After this course, I think I have a strong background in design patterns. I aim to use and improve the approach and knowledge I have acquired in this course in my future technical life.

b) Challenges

- **Correlation Between Pattern and Problem**

I have encountered two different types of difficulties related to problem and pattern association. The first was that I could only think of one suitable pattern to solve the problem. However, when I made a detailed examination and reconsidered the features of all patterns, I realized that different patterns can solve the problem with different perspectives and I have to think and decide together with their advantages and disadvantages. I saw that there is actually not a single truth here, that we can use a different design pattern depending on the difference of our priorities and the sacrifices we can make. In fact, it was an enjoyable part of the job, although it was a challenge, I think it was a very developmental process as it required analytical thinking and creativity. The second problem was that I had more than one pattern in mind, but I wasn't sure which one to use. In order to overcome this problem, I tried to choose the most suitable pattern for the design, considering its advantages and disadvantages.

- **Pattern Design and Implementation**

I learned that there are no strict rules in the designs of the patterns except for certain class and interface designs. As I was trying to apply it to my own problem, I realized that apart from the structure of the pattern, its relation to the rest of the code is important. I had difficulties implementing the pattern suitable for the requirement problem.

- **Testing**

I had difficulty in determining the test requirements. Although I was able to extract the requirements for some patterns very clearly, it was too trivial for some and I couldn't increase the requirement number. However, I did not have any difficulties in writing the test code.

c) Conclusion

The main purpose of our work with this report was to provide a more flexible structure to the Game. For this study, we first had to study and analyze the main Game’s architecture. We analyzed the main architecture together, later each of us concentrated on a subdomain to share the work that will be done afterwards. Then each one of us found 10 new flexibility requirement and candidate design

patterns have been identified in order to understand that these requirements can be solved by the Design Patterns. Advantages and disadvantages of candidate design patterns were determined for each flexibility requirement and one design pattern is selected to solve each requirement problem. In order to implement the problem with the programming language in accordance with the solution, a static class diagram and application dynamic structure have been added for the design pattern. The static class structure and the application dynamic structure have been added to verify the application with the programming language. The problem that has been defined has been implemented using the Java programming language. Required code snippets and explanations are included in the report. We need to test to what extent this implementation solves the defined problem. For this reason, the test requirements that need to be tested first were determined for each implementation and tests were made according to these test requirements. The outputs of the implementation made with the programming language have been added to the test part of the report. As a result, as stated above, all the requests to be made were made, implemented and included in the report.

3) Side System: Game Engine

(Author: Taner Furkan Göztok)

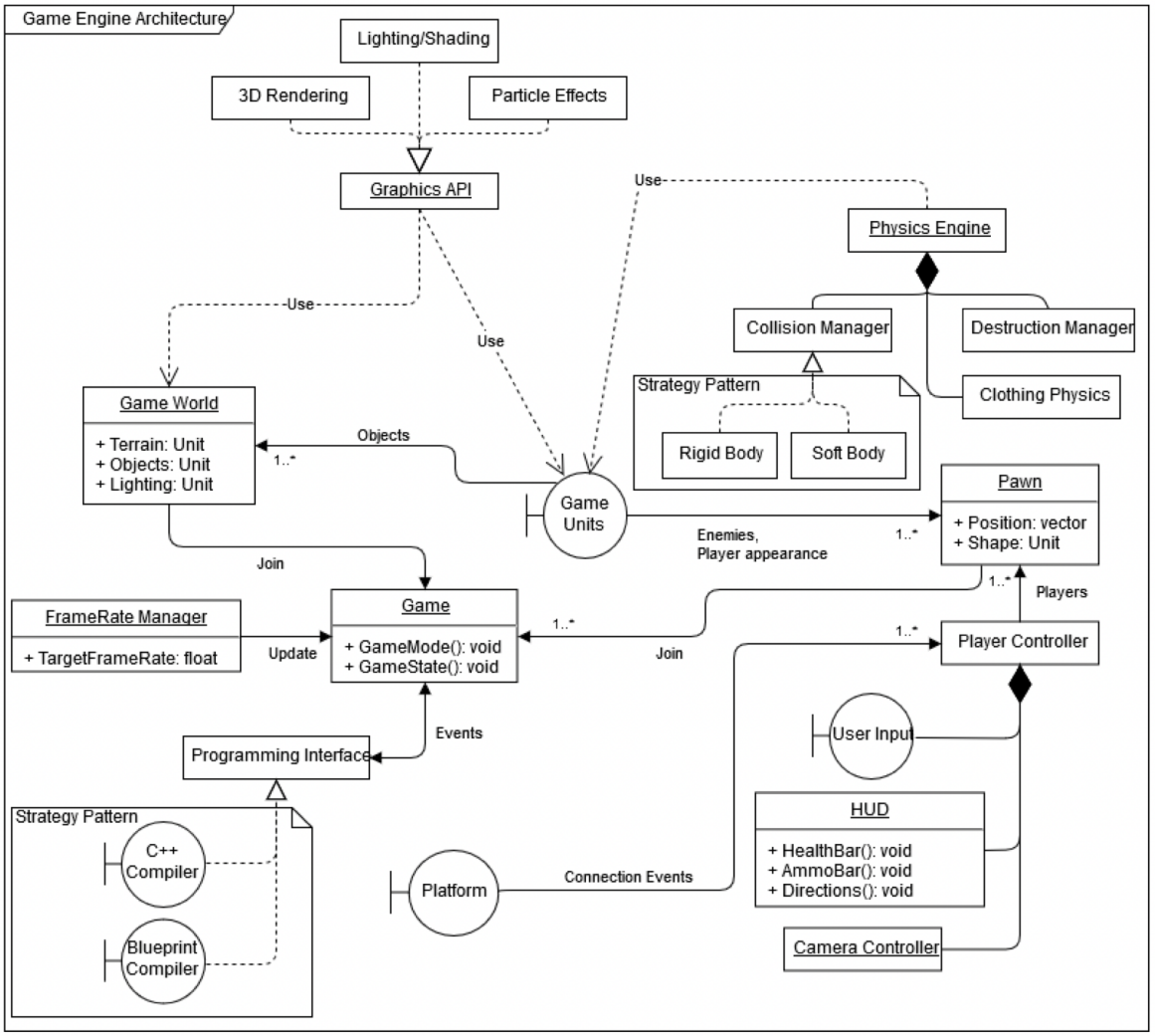
b) Sharing the domains of the game

In this part of the report, the sharing of domains amongst ourselves is going to be explained.

The game's architecture has 4 main domains. Those are Game Engine, Game Unit, Artificial Intelligence and Platform domains. Each one of us has chosen one domain and made a detailed analysis for the domain we chose, found flexibility requirements on our domains and we found the suitable Design Patterns to solve these specific problems. Design Patterns have been compared to each other and one Design Pattern has been selected to solve a requirement problem. Then each one of us implemented five of these Design Patterns by making their static class diagram, dynamic sequence diagram and writing the Design Patterns code in a programming language. Lastly we tested the implementation if they solve the requirement problems.

c) Flexibility Requirements

In this section, one will try to implement design patterns in order to increase flexibility, scalability and efficiency of Game Engine. In order to achieve those points, we studied project report and Game Engine architecture. All flexibility requirements will be divided into subparts owned by Game Engine and all the subparts will be explained below. Also, Game Engine Architecture provided by game software architecture design [4] can be seen below.



1) **Graphic load of the Game World should be calculated for possible performance optimizations.**

   In order to maintain and develop a flexibility for playability with consistent frame rate requirement R3 in Quality Requirements, there should be an algorithm which gathers all objects's graphical loads and calculates total required graphical power. This calculation will be used in order to decide whether a deletion of Game World objects are required or not to maintain consistent frame rate.

2) **When the required graphical power to maintain consistent framerate is more than the player's device's graphical power, unnecessary Game World objects should be recursively deleted.**

After being awakened to graphical limit bottleneck, to get better framerates while playing the game, unnecessary and unimportant Game World objects should be deleted or should not be shown. Logic behind this process is eliminating calculations and graphical units to a case in which players can play the game without any framerate drop. Significant point to be careful in this flexibility requirement is that deleted/not being shown objects must not have any critical place or duty in the game. For example, a group of trees in the forest which has a huge graphical load because of light and shadow calculations can be deleted but a car which can explode and damage nearby objects should not be deleted.

3) **Game HUD and interfaces should be adapted to the current controller.**

This specified feature will increase and bring flexibility to the Functional Requirement R17 which consists of being able to use preferred controllers. To increase player experience, if the game player is using the XBOX or PS4 controller on PC, all the explanations and key bindings should be shown as adapted to the current controller. For example, on the menu, “Press Enter to select” should be “Press X to select” for XBOX controller, “Press O to select” for PS4 controller. This flexibility requirement is directly connected to the third requirement.

4) **Game itself should detect players who are cheating and penalize those players.**

In order to maintain and bring flexibility to Quality Requirement R7, this flexibility requirement has been developed. The Game Engine module has Physics Engine which calculates all the physical laws of the game and processes every action for the game. In a multiplayer war game, there are rules for every player which defines all the capabilities the game player can do. For example, if movement while sprinting is defined 15km/h but a player can sprint 40km/h with some cheat which edits game physics or network, it should be detected and the player should be penalized for the cheating.

5) **Game minimap should display enemy objects proportional to their sound level.**

To increase the Player Concern C01 stated in the game development report given to us, which it indicates the concern as minimap and etc. should render in a convenient place, enemy objects should be displayed in tha game minimap regarding their size and noise level. For example, in the minimap, an enemy soldier and enemy tank should have different sizes. This arrangement should be oriented to real life in order to increase players enjoyment by developing a solution for player concern. By implementing this, the game will achieve the flexibility of changing an object's minimap resolution.

6) **In the game, when a special event occurs, every player's camera controller should change.**

To increase the Functional Requirement R13, there will be an implementation of a special event which causes every player in the game to look at a specific place. Moreover, when a nuke bomb occures with a radius of a large area, every players's camerae should turn to the bomb which leads players to acknowledge a bomb is coming and avoiding it should be better. With this pattern, all player's camera controllers can be controlled with flexibility.

7) **Players who get into a bug and can not move should be detected and respawned into a nearby location.**

Quality Requirement R5 in the game design report given to us states that the game should accommodate smooth movements after controller input. In multiplayer games, sometimes situations can happen in which players get into a state that can not move because they are stuck. Shooting, sprinting, jumping and other actions can not free them from traps. If we implement a design pattern which detects problems about moving in the game and releases players by just spawning or moving the character a few meters away, this will bring flexibility to the game to achieve solving bugs much more easily.

8) **Graphics should be handled in a much more efficient way.**

Quality Requirement R3 in the game design report which is given to us states that the game should be playable with a consistent framerate. To solve this

problem and bring flexibility to the game, graphics should be handled in an efficient way. For example, if there is a section of the game in which all players shoot and throw grenades into a structure, those graphical matrix calculations may require polynomial load because all the actions affect the others. In this case, explosions should be rendered using the same backend graphical matrix in order to solve high graphical load and bring flexibility to the game.

9) **New types of controllers should be compatible with the current controller options.**

   This specified feature will increase and bring flexibility to the Functional Requirement R17 which consists of being able to use preferred controllers. Market has different controllers for different purposes which increases player enjoyment. Helicopter pedals and sticks, fighter jet sticks, car steering wheel and muchmore controllers should be usable in the game. If a player wants to fight with a jet using his own equipment, this problem's solution will eliminate any problem regarding controller compatible issues.

10) **Game should save energy when the player is on the menu.**

   When the player is not playing the game, which means the player is on the menu or any other state which means not playing, graphical rendering should stop in order to cool down the gpu unit and save energy. This feature is developed for the game The Witcher 3 and the player's reactions were significantly good. In order to maintain the frame rate of the game and bring flexibility to the Quality Requirement R3 which states that the game should be playable with a consistent frame rate, this cooling process will bring a simple solution.

Possible Design Patterns to Solve Requirements

Flexibility Requirement	**Possible Design Patterns to Solve**
1) Graphic load of the Game World should be calculated for possible performance optimizations.	Iterator, Facade, Adapter
2) When the required graphical power to maintain consistent framerate is more than the player's device's	Visitor, Interpreter

graphical power, unnecessary Game World objects should be recursively deleted.	
3) Game HUD and interfaces should be adapted to the current controller.	Mediator
4) Game itself should detect players who are cheating and penalize those players.	Interpreter, State
5) Game minimap should display enemy objects proportional to their sound level.	Strategy, Mediator
6) In the game, when a special event occurs, every player's camera controller should change.	Observer, Command
7) Players who get into a bug and can not move should be detected and respawned into a nearby location.	Chain of Responsibility, Mediator, Observer
8) Graphics should be handled in a much more efficient way.	Flyweight, Decorator
9) New types of controllers should be compatible with the current controller options.	Adapter, Facade
10) Game should save energy when the player is on the menu	State, Interpreter

1) **Graphic load of the Game World should be calculated for possible performance optimizations.**

Reason why Iterator Pattern selected for this problem is, there is a need of traserving Game World objects and acknowledge graphical power requirement for each object. After calculations, the outcome will be used to decide whether performance improvement is required or not. Iterator pattern is widely used when there is a need of accessing an object's contents without changing its nature, and also demand support traversing for different aggregate structures.

There are two other design patterns suitable for this problem, which are Facade and Adapter. Facade design pattern eliminates complexity in subsystems and brings simple interfaces to communicate with all components. Layering subsystems and promoting independence with portability are Facade pattern's advantages as stated in [1]. However, this problem requires accessing all objects and gathering information from them, without any other dependency or operation. Facade pattern will only bring complexity to this problem if it is used. There is no need for an extra layer to communicate with all objects.

Third possible pattern is the Adapter pattern. Adapter design pattern is usable when there is a need of using an existing class without any compatibility problems. Adapter class is the main component which solves all compatibility problems among other objects. This pattern is not efficient for the problem because of its nature. To conclude, one can state that the best solution for this problem is the Iterator design pattern.

2) **When the required graphical power to maintain consistent framerate is more than the player's device's graphical power, unnecessary Game World objects should be recursively deleted.**

After the algorithm notices that there will be a performance loss due to insufficient graphical power, unnecessary Game World objects need to be deleted from the screen in order to maintain the desired stable framerate. There are lots of types of objects which could be reviewed in this process such as cars, trees etc. Visitor and Interpreter pattern could be used for this flexibility requirement.

Visitor pattern is widely used to define new operations to objects without changing the classes of the elements on which it operates[1]. An object

structure which contains many different classes and wanting to perform operations on that concrete class is Visitor pattern's one of the domains which provides a efficient solution. When there is a need for object deletion or suspending the objects from the Game World to increase performance, Visitor pattern will execute the operation by visiting all objects in the Game World. Probability of reaching the same objects in the world is nearly impossible for a FPS game, therefore objects will differ. Visitor pattern is the best solution for this flexibility requirement.

3) **Game HUD and interfaces should be adapted to the current controller.**

In this problem, in-game interfaces which users can experience such as key mapping and button naming should be capable of being easily adapted if controller changes in runtime. Runtime changes and existing loose coupling in this flexibility requirement clearly states that Mediator design pattern is the best solution. The Mediator pattern generally centralizes control over the classes it is communicating with. It promotes loose coupling by not allowing objects to call each other explicitly stated in Gamma [1].

4) **Game itself should detect players who are cheating and penalize those players.**

For the stated problem, there should be a control mechanism. For the sake of being much more understandable, we assume that in the game, players can not sprint more than 15km per hour. If a player is sprinting more than 15km per hour, then it should be detected and players must be penalized in order to maintain flexibility requirement R7. For this problem, we can track all the players and watch their speed (or any other defined rule, such as max health etc.). Or we can define an algorithm in the Game Engine which detecs any cheating action.

There are 2 possible design pattern for this requirement. Interpreter and State patterns. Interpreter design pattern is a behavioral design pattern which interprets sentences in defined language. One should define a grammar with rules and adapt the features to the grammar which can be done in the game. Adding new expressions is also for the Interpreter design pattern since there is already a grammar.

Second pattern which is the State pattern is a structural design pattern which brings flexibility by changing the algorithm in run time. A not cheating and cheating states can be added to this design pattern to detect cheating players. After comparing those 2 design pattern, Interpreter pattern is the most compatible for the stated problem and flexibility requirement.

5) **Game minimap should display enemy objects proportional to their sound level.**

   For the stated problem, Strategy and Mediator patterns are chosen as candidates to implement and bring flexibility to the problem.

   Strategy design pattern is the best pattern to implement for this problem, because every object which will be shown in the minimap will have a different specific algorithm. A person might have different methods such as sprinting, shooting or equipment that the player has will have different sound levels which need to be adjusted in the minimap. For example, If person is sprinting, it should be much more visible in the minimap than standing state. Strategy pattern is much more suitable for specific algorithm needs for every object.

   On the other hand, the Mediator design pattern includes a mediator. Mediator can become a very large and complex object if there are lots of game objects which need to be shown in the minimap. Because of this conflict, a strategy pattern is selected to solve and implement a flexibility to the problem.

6) **In the game, when a special event occurs, every player's camera controller should change.**

   For this problem, there are 2 candidate design pattern solutions which are Observer and Command design patterns. In order to solve this problem, the game must evaluate the CameraController of all the players in the game and send commands to CameraController accordingly.

   Observer is a behavioral design pattern which defines a subscription system to notify multiple objects and let them observe the current situation. Instead of the observer sending requests of the current situation, this publisher sends information to all CameraControllers. Moreover, there is a

publisher-subscription relationship. With the Observer design pattern, whenever a change needs to happen in all player's CameraController, because those CameraControllers are already subscribed to the publisher, they all look at the same point.

Command design pattern in the other hand is also a behavioral design pattern which creates objects from request that contains all the specific acknowledge about the request. This command object can be implemented to each CameraController. However, command design patterns might cause system complexity if there are lots of players in the game.

Observer design pattern is the best solution for the stated flexibility requirement, hence it is selected.

7) **Players who get into a bug and can not move should be detected and respawned into a near location.**

To solve the stated problem, there are 3 design patterns which may be implemented; Chain of Responsibility, Mediator and Observer.

Chain of Responsibility is a behavioral design pattern which passes requests through a chain of handlers. With Chain of Responsibility, we can solve any type of bug happening in the game regarding the flexibility requirement. After a request can not be handled by any of the chains, a final chain may respawn the player nearest available point. Because of its structure this design pattern is very suitable for the problem.

Mediator design pattern is also a behavioral design pattern which reduces dependencies between objects. With this pattern, direct communication between objects are restricted and this communication goes through a mediator object. Players who got stuck or got into a bug's objects can send cross messages into the game to tell why this situation happened but this may cause a significant amount of load on the game. But there are lots of bugs that can happen in the game and implementing every solution for every situation is not an efficient way with a mediator design pattern.

Lastly, Observer design pattern is also a behavioral design pattern which lets implementer to build subscription mechanisms. With this pattern, a publisher objects can watch all the players and if any bug happens, the publisher may publish a solution and let observers observe the solution. But Observer pattern is

much more suitable for bulk operations, if every solution is specific for that moment, it is not the best option for this problem.

8) **Graphics should be handled in a much more efficient way.**

   To solve this problem, Flyweight and Decorator design pattern will be examined.

   Flyweight design pattern is a structural pattern which supports sharing large numbers of objects efficiently. To solve this problem, graphical loads created by the event occuring in the similar ways should share the same structure to decrease graphical load. Flyweight design pattern will implement sharing graphical interfaces when high and same type of loads occur in the same area. Similarities between actions will make it happen to implement Flyweight pattern because all the actions are specified by the developers itself.

   Second solution is the Decorator design pattern which is also a structural pattern. Decorator patterns place objects inside a special wrapper object that contains the behaviors. With this solution, actions which had the same output for the graphical load can be wrapped. But the Decorator pattern is not fully efficient to our problem because wrapping may not be able to bring graphical efficiency or it might cause actions to lose their self behaviors.

   Flyweight design pattern is selected for this problem hence it is the best solution and it will bring flexibility to solve graphical loads.

9) **New types of controllers should be compatible with the current controller options.**

   For the stated problem, Adapter and Facade design patterns are candidates to bring flexibility.

   Adapter design pattern which is also known as Wrapper is a Structural design pattern which converts the interface of a class into another interface which clients expect. This brings a solution for incompatible interfaces. With the adapter design pattern, any different controller can be compatible.

   Facade is the second option which is also a Structural design pattern. Facade design pattern generally defines new interfaces for existing objects but the true and efficient solution here is solving incompatible situations among the controllers and making them compatible with current controller interfaces.

Because our problem is actually a mismatch and inconsistency problem, Adapter design pattern is the best solution to implement and implementation will bring flexibility to the game where any kind of adapter can be easily adapted to the game.

10) Game should save energy when the player is on the menu.

To solve the stated problem, there are 2 possible design patterns which are State and Interpreter.

State design pattern is a behavioral pattern which lets an object alter its behavior when its internal state changes. This state change can be implemented to see if the game should render the graphics or not. By implementing this solution, problem will be solved and flexibility will be implemented to the game. For any other reasons, developers can add other states easily where the game should not need to render the graphics.

Interpreter design pattern is also a behavioral pattern which given a language and grammar, interpreter class uses representation to interpret sentences given or identified language. Interpreter design pattern is much more usable when there is a state which is more compatible with the grammar backbone and it's hard to maintain an interpreter if grammars are hard to implement.

Therefore, the state design pattern is the best solution for the stated problem which brings flexibility too.

Problem Description

We have concluded to resolve the problems and bring more flexibility which comply with the flexibility requirements 3, 7, 13 17. To be able to achieve those, we will firstly define problems regarding flexibility requirements. Then, a detailed static class and application dynamic diagram will be presented in order to validate the implementations. Java is the programming language which will be used upon those implementations.

1)Problem Definition

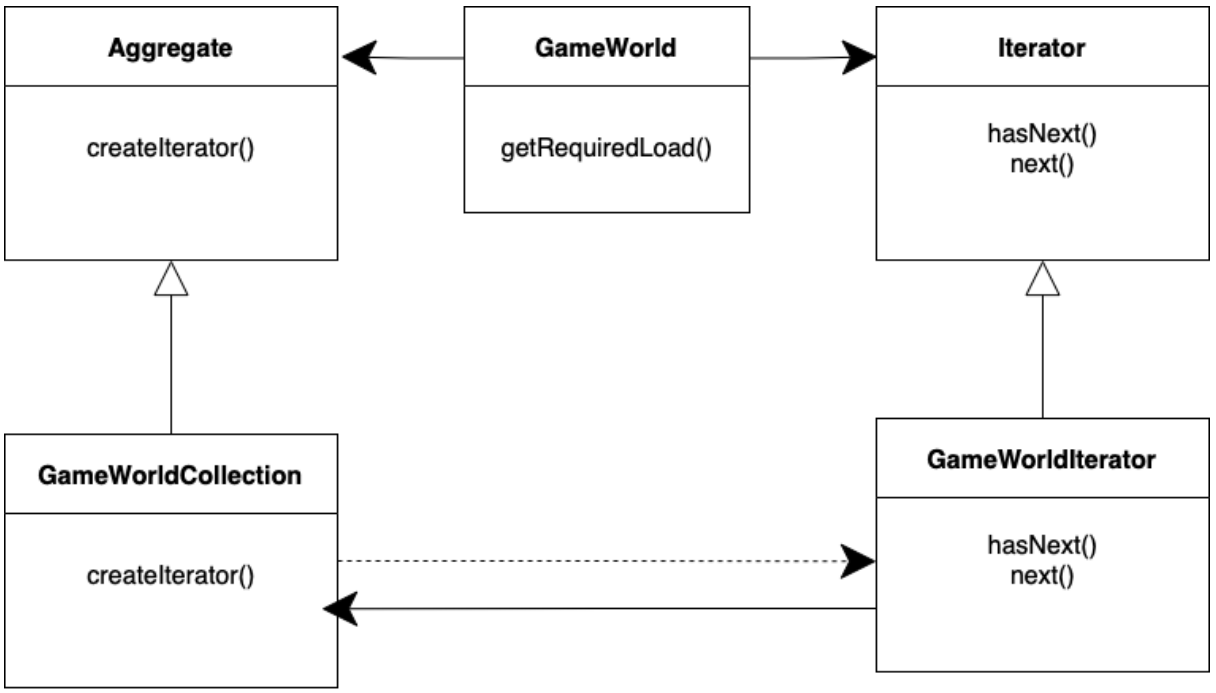
Flexibility Requirement: The game should be playable with a consistent framerate.

i.Graphic load of the Game World should be calculated for possible performance optimizations.

As stated above, the graphical load of the total Game World Object should be calculated in order to use in the future as a flexibility option. For this problem, there is a need to reach all the objects and gather their graphical loads and calculate sum all of Game World.

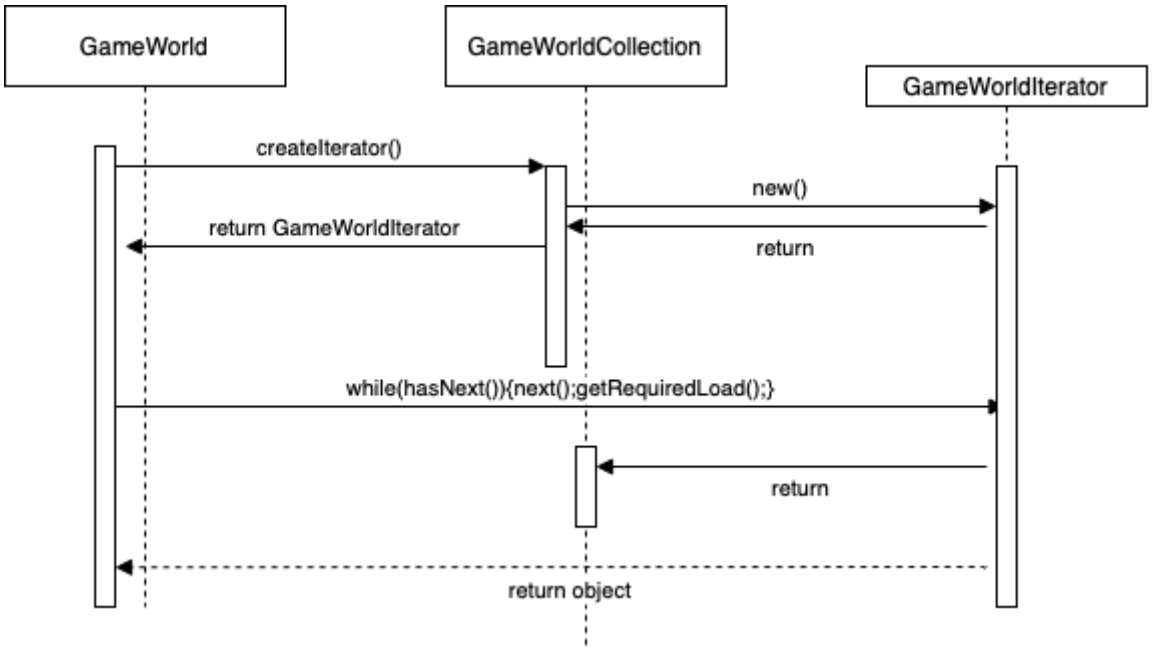
i.Static Class Structure

For the related problem, a static class diagram was created in order to solve the stated problem. Reason of the static class diagram is to conclude that both accuracy and consistency of the problem are acceptable while implementing the solution with Java programming language.



For this problem, Iterator Design Pattern is selected for the solution domain. Iterator Pattern, also known as Cursor Pattern as stated in [1], is mostly used when there is a need to access objects without exposing its internal representation and support traversing different aggregate structures such as Game World objects (trees, vehicles, containers etc). Above static class diagram for the iterator pattern to solve flexibility requirement can be seen. For every Game World object, reaching their graphical unit load will be gathered and iteratively calculated to total graphical load.

i.Dynamic Structure of Application



The structure above contains a dynamic structure diagram of iterator pattern application for the stated flexibility requirement. Structure shows run-time actions. After creating the iterator, all GameWorld objects will be iterated while there are any left objects that haven't been iterated yet and will gather their graphical required loads. This will be used for cumulative calculation of total required load.

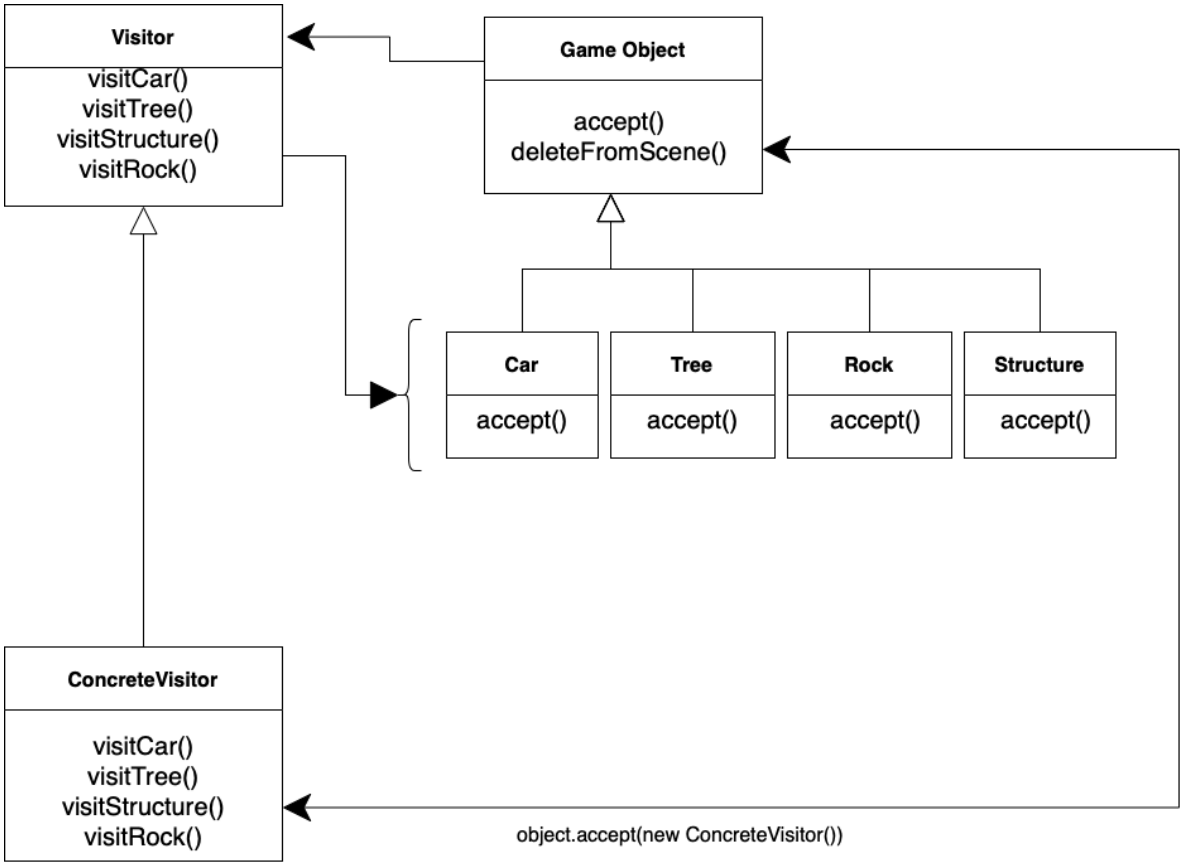
2)Problem Definition

Flexibility Requirement: The game should be playable with a consistent framerate.

ii.When the required graphical power to maintain consistent framerate is more than the player's device's graphical power, unnecessary Game World objects should be recursively deleted.

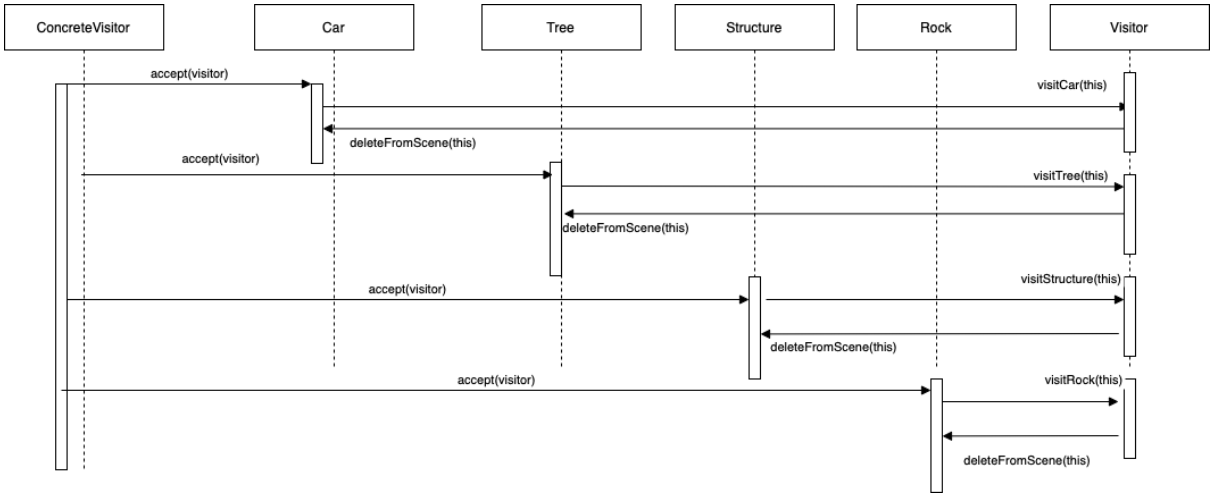
If there is a need to delete objects from the scene which have no significant role or important meaning in the Game World, it should be with stated flexibility requirement. This flexibility requirement will bring much more durable frame rate ratios.

ii.Static Class Structure



Above, static class structure for the stated flexibility requirement can be seen. Visitor design pattern will be implemented for this problem. Visitor pattern will delete objects from the scene if they have no significant role. By this method, the required total graphical load will decrease and players will play the game with much more experience since there are much less frame rate drops.

ii.Dynamic Structure of Application



The dynamic diagram of the given flexibility requirement can be seen above. Visitor pattern is understandable by seeing and following the general structure for dynamic diagrams. ConcreteVisitor can execute deletion for every game world object. If a new game object is created in the future, or changed, just adding the new object to the Visitor will be enough to keep this flexibility requirement and problem solved.

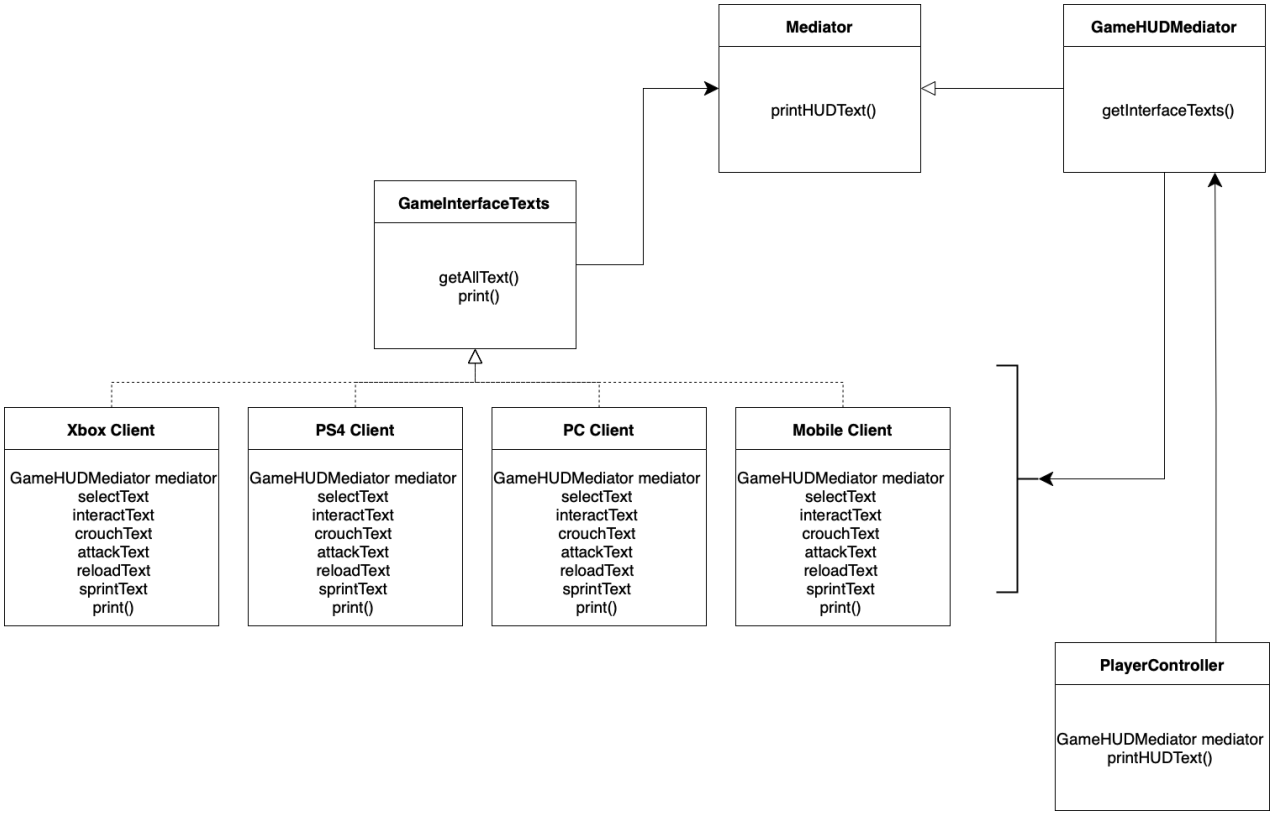
3)Problem Definition

Flexibility Requirement: The game should be controllable using the preferred input of the platform.

iii.Game HUD and interfaces should be adapted to the current controller.

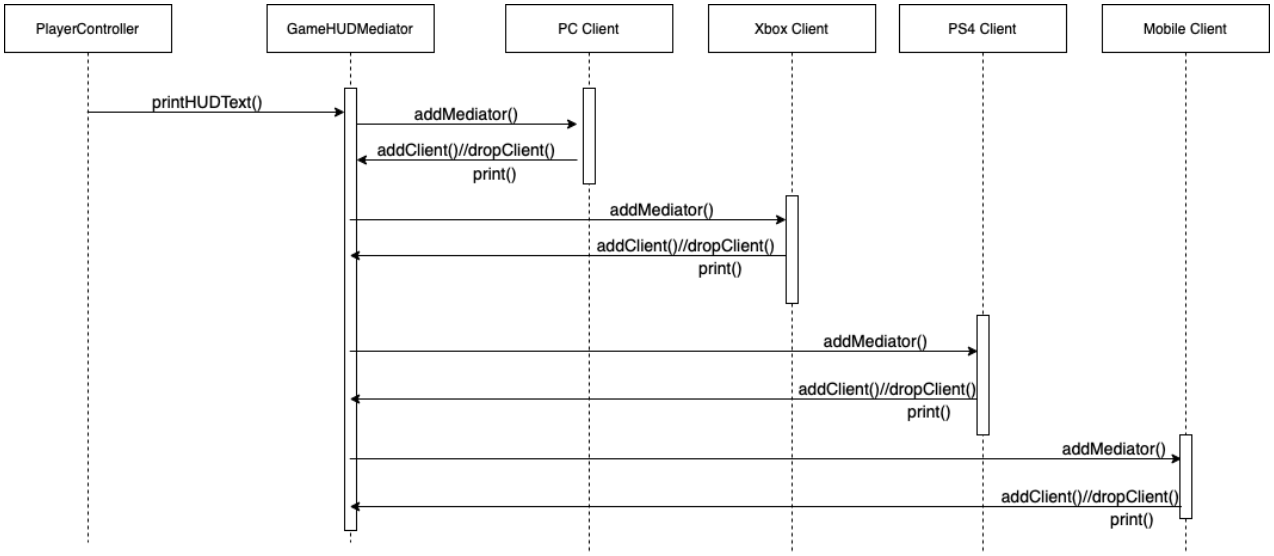
As stated in the design report of the game project, the Game Client class in the Platform architecture contains the player controller type which are PC Client, Mobile Client, Xbox Client and PS4 Client. This information passed to the Player Controller class in the Game Engine architecture which was included in this part of the report. Therefore, we can assume that the Player Controller class knows which controller type is connected and the Mediator design pattern will be implemented through this specific information.

iii.Static Class Structure



Static class diagram for the stated flexibility requirement and problem which is implemented with the mediator design pattern can be seen above. All clients which are Xbox, PS4, PC and Mobile refers to the common Mediator interface for passing their text statements which makes them indepent from one controller to another. If any new controller type needs to be added to the game, the only change that needs to be implemented is a new Client class with their specific texts in order to be printed in the game. Without changing any other object, class or hierarchy this could be easily managed. This is the flexibility which this solution brings with the Mediator design pattern.

iii.Dynamic Structure of Application



Picture above contains a sequence diagram for the stated flexibility requirement which is implemented with the Mediator design pattern.

In the game itself, there are already 4 implemented client types. Xbox, PS4, Mobile and PC. For those client types, there are specific instructions and keys to use in game to interact and play. The PlayerController class is already a working class specified in the Game Engine architecture stated in the game software architecture report.

For the sake of the flexibility requirement and not knowing the actual game itself, we assume that the PlayerController class knows which controller the user is using while playing the game. The PlayerController class can call mediators with the knowledge of the current controller type. For example, if the current controller is an Xbox controller, the PlayerController class can call a mediator with the mediator which includes text information for the Xbox controller.

4)Problem Definition

Flexibility Requirement: Players should not be able to cheat to keep games fair.

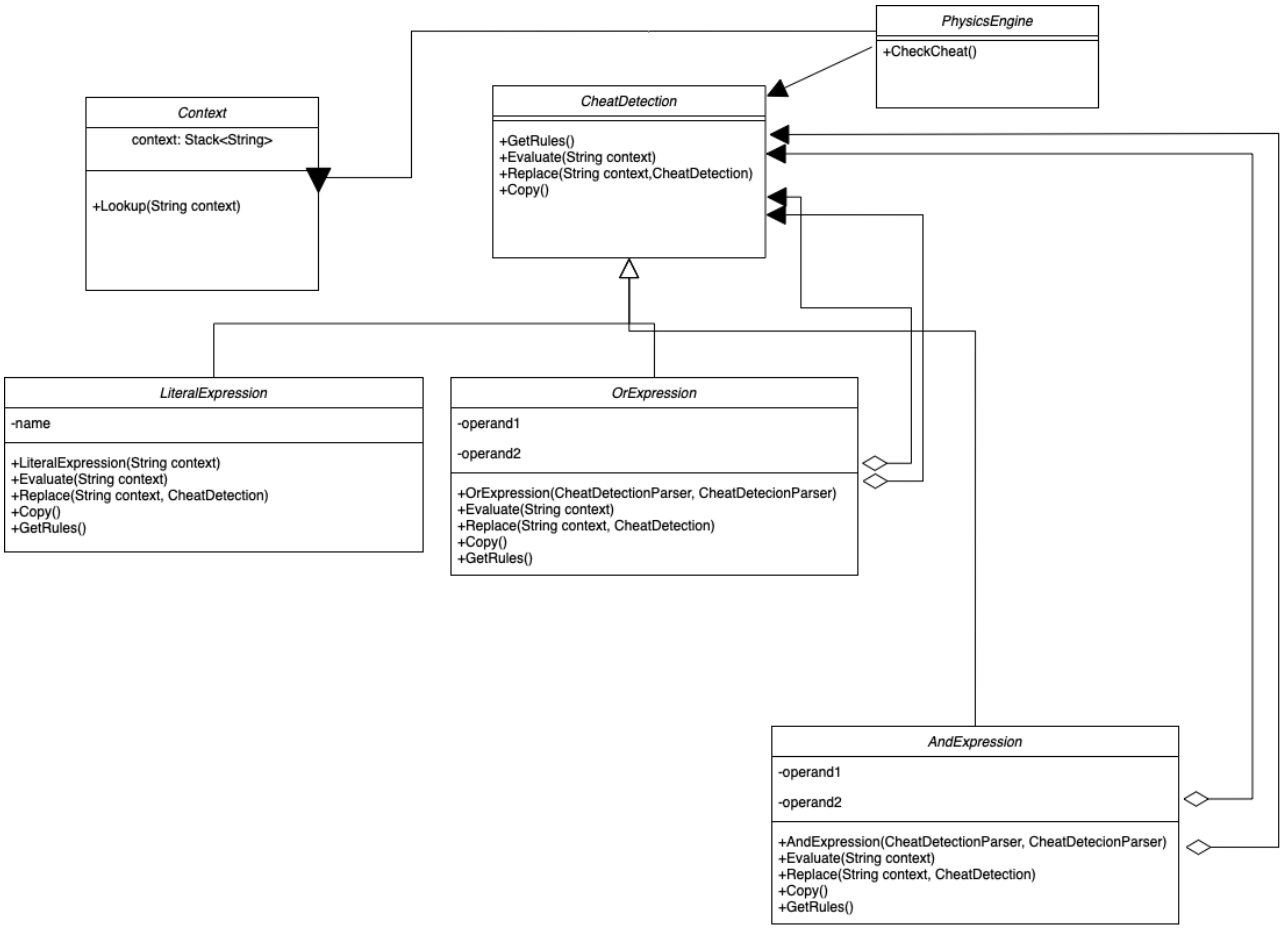
iv.Game itself should detect players who are cheating and penalize those players.

Game report states that players should not be able to cheat to keep games fair, which is a very understandable requirement. For this requirement, Interpreter

design pattern will be implemented to the game. Grammars for cheating and not cheating situations will be implemented and tested in order to prove Interpreter design pattern implementation is achieved.

Cheating situations are not stated in the game report, therefore we will interpret the Game Engine's Physics Engine and define new rules in order to prevent cheating. Those rules can easily be changed as a result of implementing a design pattern.

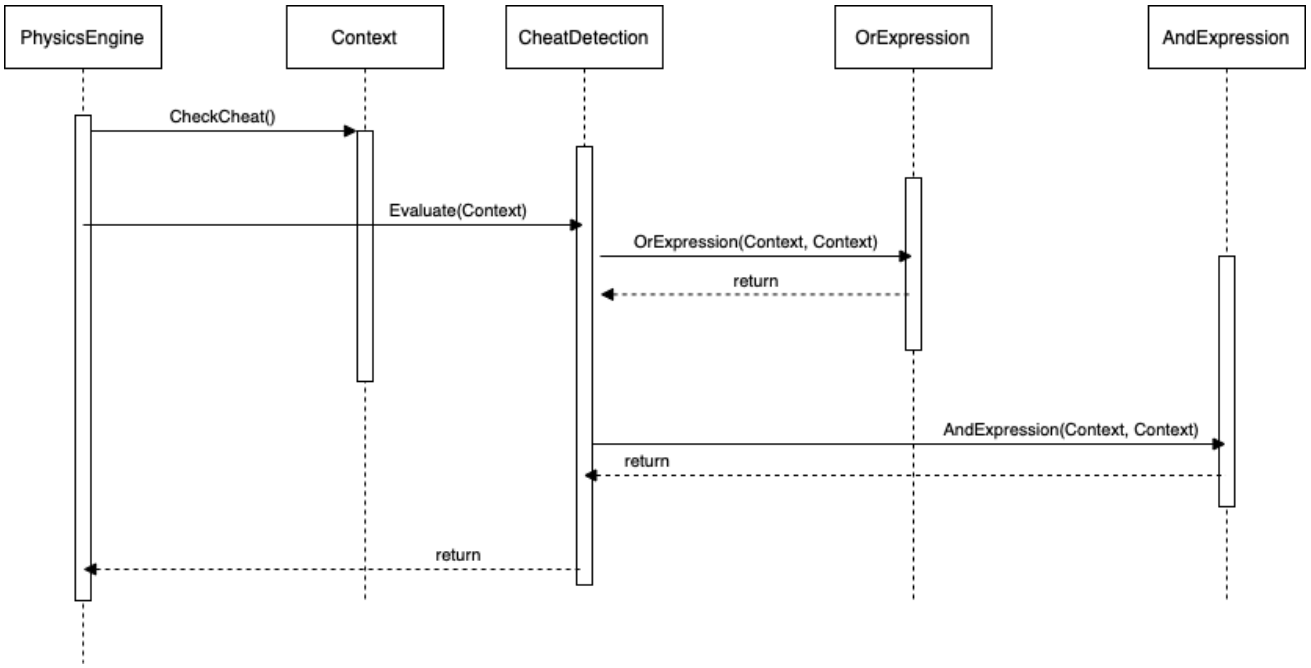
iv.Static Class Structure



Above diagram contains Interpreter design pattern implementation for the stated problem. This solution will bring flexibility to check if any player is cheating the game. Lots of other cheat rules can be easily added or modified by the Interpreter design pattern. There are 2 operants implemented, And and Bigger. These 2 operants will be used such as if a player is in a state of jumping and

shooting, it should be detected as cheating. If The player's moving speed is bigger than 15, it should also be detected. There are also methods for changing the implementation, defining a new expression.

iv.Static Class Structure



Dynamic class structure of the Interpreter design pattern for the stated flexibility requirement and problem can be seen above. PhysicEngine, which is actually a client wokes Context with CheckCheat() method. Context has all the String as a grammar which includes information of game action information such as speed, jumping state, firing state. After that, Context's are sent into BiggerExpression or AndExpression to determine if there is a cheat situation. To be able to test the Interpreter design pattern implementation, only stated expressions have been created. This pattern will bring flexibility to check cheating players. New rules can be easily adapted and modified with just creating a new expression type.

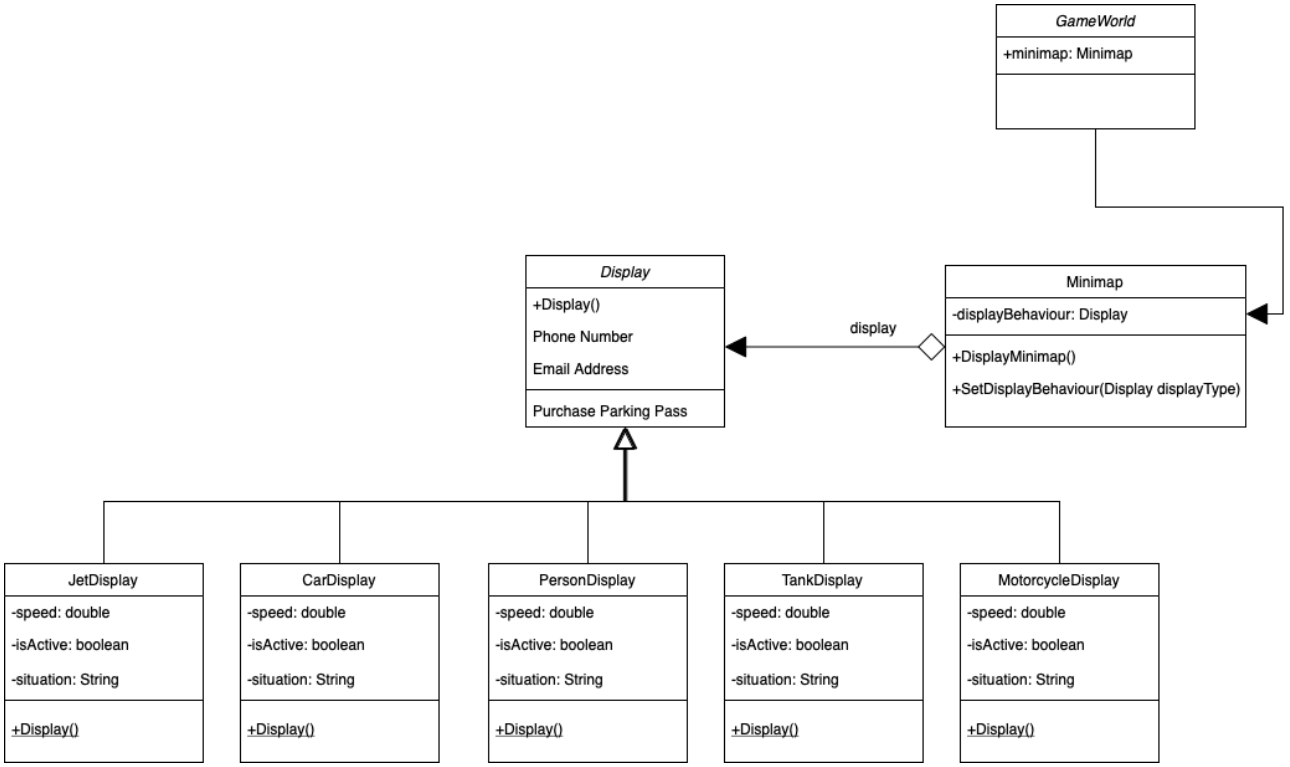
5)Problem Definition

Flexibility Requirement: There must be an intuitive interface. Minimap should render in a convenient place.

v.Game minimap should display enemy objects proportional to their sound level.

Game report suggests that the players should enjoy the game and minimap should be rendered in a convenient place. This problem is a relevant problem which requires how objects will appear in the minimap depending on their noise levels to improve the reality of the game. By implementing the Strategy design pattern, it will provide flexibility to add and change objects's minimap appearance easily.

v.Static Class Structure

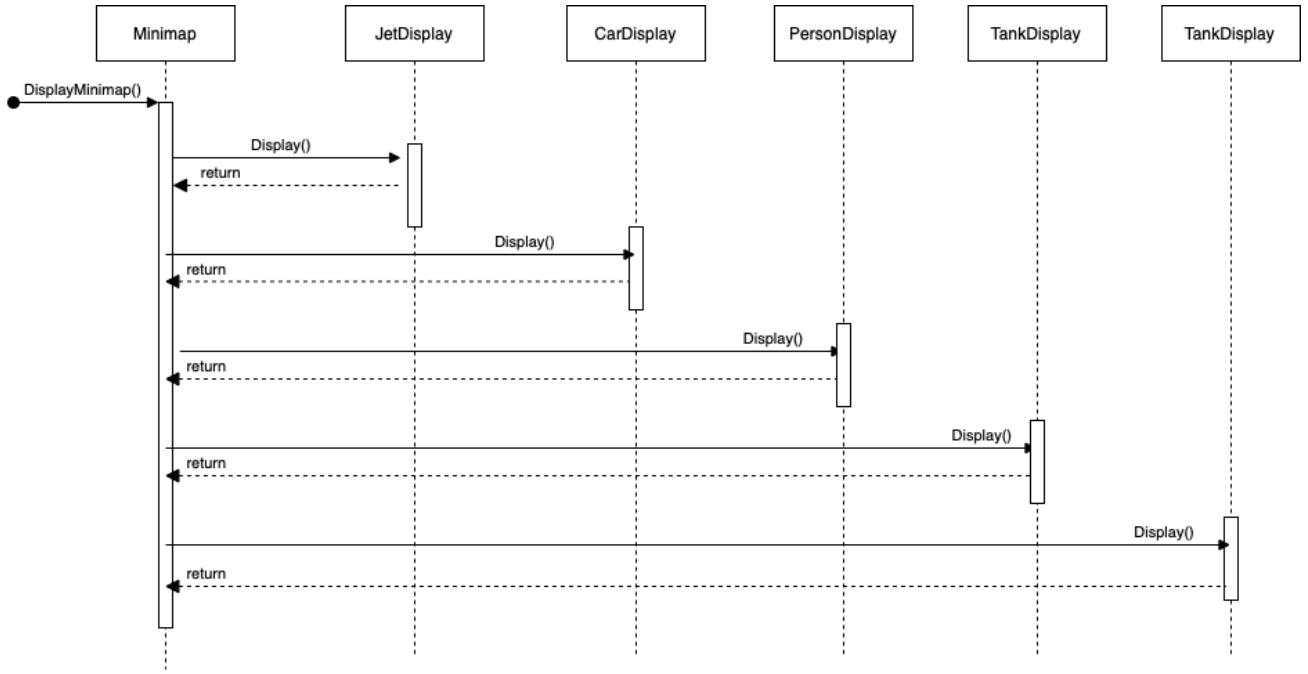


As can seen above, a static class diagram of the Strategy pattern has been implemented. GameWorld class has Minimap which displays objects in the HUD. Minimap is the Context of the Strategy pattern. Display class is the

Strategy itself. Each object which needs to be shown in the minimap has its own implementation of Display. Adding or modifying new objects to be shown in minimap is much more easy and flexible with this structure.

Minimap using Display as a Strategy will fix the problem.

v.Dynamic Class Structure



Dynamic static diagram can be seen above. Diagram shows the interactions between Objects and those interactions are solving the stated problem and bring flexibility.

Implementing The Design Pattern

In this part, we will make implementation in the Java programming language for every flexibility requirement stated in the previous part. After implementation, a test will be conducted and test results will be used in order to decide whether implementation is successfully managed or not.

i. The definition of the code and its relationship with the pattern structure

In this part, implementation of the iterator design pattern will be managed with the Java programming language to the stated flexibility requirements. After implementation, problem specific test scenarios will be executed.

Create Executions

```
private static GameWorld gameWorld = new GameWorld();
private static GameWorldCollection gameWorldCollection = new GameWorldCollection();
private static gameWorldCollectionIterator = gameWorldCollection.createIterator();
```

Above one can see the creation of classes and the iterator itself.

Iterator Interface

```
interface Iterator
{
   // indicates whether there are more elements to
   // iterate over
   boolean hasNext();
    // returns the next element
   Object next();
}
```

Iterator interface created in order to use specific iterator, not the java implemented version.

Collection Interface

```
interface Collection
{
   public Iterator createIterator();
}
```

Collection interface created in the same way Iterator interface created.

GameWorldIterator Class

```
// GameWorld iterator
class GameWorldIterator implements Iterator
{
   GameWorldObject[] gameWorldObjects;
    // maintains curr pos of iterator over the array
   int currPos = 0;
    // Constructor takes the array of GameWorld objects are
   // going to iterate over.
   public  GameWorldIterator (GameWorldObject[] gameWorldObjects)
   {
       this.gameWorldObjects = gameWorldObjects;
   }
    public Object next()
   {      // return next element in the array and increment
current position
       GameWorldObject gameWorldObject =
gameWorldObjects[currPos];
       currPos += 1;
       return gameWorldObject;
   }
    public boolean hasNext()
   {
       if (currPos >= gameWorldObjects.length ||
           gameWorldObjects[currPos] == null)
           return false;
       else
           return true;
   }
}
```

This code block presents the GameWorldIterator class which implements the Iterator interface. This can be seen in the dynamic diagram of the structure as

GameWorldIterator. hasNext and next methods are created and will be used while iterating the whole GameWorld objects.

GameWorld Class

```
class GameWorld
{
   GameWorldCollection gameWorldCollection;
    public GameWorld(GameWorldCollection gameWorldCollection)
   {
       this.gameWorldCollection = gameWorldCollection;
   }
    public void getTotalRequiredLoad()
   {
       Iterator iterator = gameWorldCollection.createIterator();
       double totalLoad = 0.0;
       System.out.println("Calculation of total required graphical
load started...");
       while (iterator.hasNext())
       {
           GameWorldObject n = (GameWorldObject)iterator.next();
           totalLoad += n.getRequiredLoad();
       }
       System.out.println("Calculation finished! Total required
graphical load is: " + totalLoad);
   }
}
```

GameWorld can be seen in dynamic diagram stated belov. Class simulated in order to maintain relationship with given iterator design pattern and execute all the required methods.

GameWorldObject Class

```
class GameWorldObject
{
   // To store object name and graphical needed load.
   String objectName;
   double graphicalLoad;
   public GameWorldObject(String objectName, double graphicalLoad)
   {
       this.objectName = objectName;
       this.graphicalLoad = graphicalLoad;
   }
   public double getRequiredLoad()
   {
       return graphicalLoad;
   }
   public String getName(){
       return objectName;
   }
}
```

GameWorldObject class created in order to represent the objects, gather their name and required graphical load per object. This double variable will be used while calculating the total required graphical load.

GameWorldCollection Class

```
class GameWorldCollection implements Collection
{
   static final int  MAX_OBJECTS = 500;
   int numberOfItems = 0;
   GameWorldObject[] gameWorldList;
    public GameWorldCollection()
   {
       gameWorldList = new GameWorldObject[MAX_OBJECTS];
        // Adding objects and testing total required load.
       addItem("Car 1", 52.0);
       addItem("Car 2", 53.0);
       addItem("Car 3", 54.0);
```

```
        addItem("Car 4", 12.0);
        addItem("Tree 1", 5.2);
        addItem("Tree 2", 4.8);
        addItem("Tree 3", 2.9);
        addItem("Tree 4", 2.2);
        addItem("Tree 5", 4.8);
        addItem("Tree 6", 6.9);
        addItem("Tree 7", 8.2);
        addItem("Tree 8", 9.8);
        addItem("Tree 9", 1.9);
        addItem("Spaceship", 26.72);
        addItem("Rock Collection 1", 13.4);
        addItem("Rock Collection 2", 15.6);
    }
     private void addItem(String str, double graphicalLoad)
    {
        GameWorldObject obj = new GameWorldObject(str,
graphicalLoad);
        if (numberOfItems >= MAX_OBJECTS)
            System.err.println("Game world is full.");
        else
        {
            gameWorldList[numberOfItems] = obj;
            numberOfItems = numberOfItems + 1;
        }
    }
     public Iterator createIterator()
    {
        return new GameWorldIterator(gameWorldList);
    }
}
```

GameWorldCollection class has the duty of creating the iterator itself, adding the game objects into the Game World and returning the iterator.

Test Requirements

For this flexibility requirement, we want to test and verify that we can reach all the objects in GameWorld, gather their graphical loads and return the total

graphical load for the GameWorld itself. This calculation will be used for making the choice of deleting GameWorld objects in order to decrease total graphical load and maintain frame rate stated in requirement 3.

Test Results

In this section of the report, we will execute the given test for the stated implementation made with the Java programming language and using the test requirements, we will determine if implementation of the iterator design pattern is successfully working with all the code fragments.

```
Calculation of total required graphical load started...
Calculation finished! Total required graphical load is: 273.42
```

The output of the test code is matching the total graphical load written above. As a result, the iterator design pattern for the problem and flexibility requirement we have implemented is working.

ii. When the required graphical power to maintain consistent framerate is more than the player's device's graphical power, unnecessary Game World objects should be recursively deleted.

In this part, implementation of the visitor design pattern will be managed with the Java programming language to the stated flexibility requirements. After implementation, problem specific test scenarios will be executed.

GameObject and GameObjectVisitor Interfaces

```
interface GameObject {
```

```
    void accept(GameObjectVisitor visitor);
    void deleteFromScene(GameObjectVisitor visitor);
}

interface GameObjectVisitor {
    void visit(Car car);

    void visit(Tree tree);

    void visit(Rock rock);

    void visit(Structure structure);

    void deleteFromScene(Tree tree);

    void deleteFromScene(Rock rock);

    void deleteFromScene(Structure structure);

    void deleteFromScene(Car car);
}
```

Above GameObject and GameObjectVisitor can be seen implemented to organize the structure. By using interface implementation on the Java programming language, we can make our flexibility requirement more reliable.

Rock, Tree, Car and Structure Classes

```
class Structure implements GameObject {
    private String status;

    public Structure(final String status) {
        this.status = status;
    }

    public String getStatus() {
        return status;
    }

    @Override
    public void accept(GameObjectVisitor visitor) {
        visitor.visit(this);
```

```
    }

    @Override
    public void deleteFromScene(GameObjectVisitor visitor) {
        visitor.deleteFromScene(this);
    }
}

class Car implements GameObject {
    private String status;

    public Car(final String status) {
        this.status = status;
    }

    public String getStatus() {
        return status;
    }

    @Override
    public void accept(GameObjectVisitor visitor) {
        visitor.visit(this);
    }

    @Override
    public void deleteFromScene(GameObjectVisitor visitor) {
        visitor.deleteFromScene(this);
    }
}

class Tree implements GameObject {
    private String status;

    public Tree(final String status) {
        this.status = status;
    }

    @Override
    public void accept(GameObjectVisitor visitor) {
        visitor.visit(this);
    }

    @Override
    public void deleteFromScene(GameObjectVisitor visitor) {
        visitor.deleteFromScene(this);

    }
    public String getStatus() {
```

```
        return status;
    }
}

class Rock implements GameObject {
    private String status;

    public Rock(final String status) {
        this.status = status;
    }

    @Override
    public void accept(GameObjectVisitor visitor) {
        visitor.visit(this);
    }

    @Override
    public void deleteFromScene(GameObjectVisitor visitor) {
        visitor.deleteFromScene(this);
    }
    public String getStatus() {
        return status;
    }
}
```

These classes are the objects which can be found in the game world itself. This is a demo implementation and the original game may have more than these objects but the real idea here is being able to reach all the objects.

GameObject Class

```
class GameObjects implements GameObject {
    private String status;

    private final List<GameObject> objects;

    public GameObjects() {
        this.objects = List.of(
                new Structure("important"), new
Structure("important"),
                new Structure("not important"), new Structure("not
important"),
                new Tree("not important"), new Rock("not
```

```
important"));
    }

    @Override
    public void accept(GameObjectVisitor visitor) {
        for (GameObject element : objects) {
            element.accept(visitor);
        }
        visitor.visit(this);
    }

    @Override
    public void deleteFromScene(GameObjectVisitor visitor) {
        for (GameObject element : objects) {
            element.accept(visitor);
        }
        visitor.deleteFromScene(this);
    }
}
```

Gameworld object is the class which is already placed in the original game. This class is a representation of creation of the game objects and inserting them into real game. After inserting, the visitor design pattern will start to visit all objects and delete the significant ones. This will bring much less graphical unit requirement for the game, hence frame rate will be much higher and stable.

Test Scenarios

For this flexibility requirement, game objects will be inserted into the game world and after that all the objects will be visited with visitor design pattern. Insignificant objects will be removed from the scene in order to increase frame rate and enjoyment of the game itself. We need to see if the unimportant objects are deleted.

```
class GameObjectDoVisitor implements GameObjectVisitor {
    @Override
    public void visit(Tree tree) {
        System.out.println("I am tree! I am " + tree.getStatus());
        if(tree.getStatus()!="important")
            System.out.println("Tree is deleted from scene!");
    }
```

```
    @Override
    public void visit(Car car) {
        System.out.println("I am car! I am " + car.getStatus());
        if(car.getStatus()!="important")
            System.out.println("Car is deleted from scene!");
    }

    @Override
    public void visit(Structure structure) {
        System.out.println("I am structure! I am " + structure.getStatus());
        if(structure.getStatus()!="important")
            System.out.println("Structure is deleted from scene!");
    }

    @Override
    public void visit(Rock rock) {
        System.out.println("I am rock! I am " + rock.getStatus());
        if(rock.getStatus()!="important")
            System.out.println("Rock is deleted from scene!");
    }

    @Override
    public void deleteFromScene(Car car) {
        if(car.getStatus()!="important")
            System.out.println("Car is deleted from scene!");
    }

    @Override
    public void deleteFromScene(Tree tree) {
        if(tree.getStatus()!="important")
            System.out.println("Tree is deleted from scene!");
    }

    @Override
    public void deleteFromScene(Rock rock) {
        if(rock.getStatus()!="important")
            System.out.println("Rock is deleted from scene!");
    }

    @Override
    public void deleteFromScene(Structure structure) {
        if(structure.getStatus()!="important")
            System.out.println("Structure is deleted from scene!");
    }

}

public class VisitorDemo {
    public static void main(final String[] args) {
        Object gameObjects = new GameObjects();
        ((GameObject) gameObjects).accept(new GameObjectDoVisitor());
    }
```

```
}
```

Test Results

```
I am structure! I am important
I am structure! I am important
I am structure! I am not important
Structure is deleted from scene!
I am structure! I am not important
Structure is deleted from scene!
I am tree! I am not important
Tree is deleted from scene!
I am rock! I am not important
Rock is deleted from scene!
```

As stated above, expected test results have been printed and we can clearly say that the visitor pattern for this flexibility requirement is working and bringing much more elasticity to the game itself.

iii. Game HUD and interfaces should be adapted to the current controller.

In this part, stated flexibility requirement will be done by implementing the Mediator design pattern with the Java programming language. For the implementation and testing, we will call every mediator and print every HUD text for every controller type just to see the Mediator design pattern is working and implemented successfully.

Mediator Interface

```
public interface Mediator {
   public void printHUDText();
}
```

Above, one can see the Mediator interface which continues the printHUDText method for gathering current controller specified HUD text.

GameInterfaceTexts Interface

```
public interface GameInterfaceTexts {
```

```
    public void print();
    public String getAllText();
}
```

GameInterfaceTexts interface is for the Clients. Every client differs from each other and in time, new clients can be added to this scheme.

GameHUDMediator Class

```
import java.util.ArrayList;
public class GameHUDMediator implements Mediator {
 private ArrayList<GameInterfaceTexts> clients;

 public GameHUDMediator() {
    clients = new ArrayList<>();
 }

 public void getInterfaceClients(GameInterfaceTexts client) {
     clients.add(client);
 }

 @Override
 public void printHUDText() {
    for(GameInterfaceTexts c: clients) {
       c.print();
    }
 }
}
```

Above, one can see the concrete implementation of the Mediator for GameHUDMediator. GameHUDMediator class gathers the HUD text information from clients with getInterfaceClients method. After calling the printHUDText method, the class shows each added client's specific HUD descriptions. It returns every client with their corresponding explanations.

Client Class

```
public class Client implements GameInterfaceTexts {
  private String selectText;
  private String interactText;
  private String crouchText;
```

```
    private String attackText;
    private String reloadText;
    private String sprintText;

    private GameHUDMediator mediator;

    public Client(String selectText, String interactText,
            String crouchText, String attackText, String reloadText,
            String sprintText, GameHUDMediator gameHUDmediator) {
        this.selectText = selectText;
        this.interactText = interactText;
        this.crouchText = crouchText;
        this.attackText = attackText;
        this.reloadText = reloadText;
        this.sprintText = sprintText;
        addMediator(gameHUDmediator);
    }

    public void addMediator(GameHUDMediator gameHUDmediator) {
        this.mediator = gameHUDmediator;
    }

    @Override
    public void print() {
            System.out.println("Selection text: " + selectText + "\n" +
            "Interact text: " + interactText + "\n" +
            "Crouch text: " + crouchText + "\n" +
            "Attack text: " + attackText + "\n" +
            "Reload text: " + reloadText + "\n" +
            "Sprint text: " + sprintText + "\n" );
    }

    @Override
    public String getAllText() {
        return "Selection text: " + selectText + "\n" +
        "Interact text: " + interactText + "\n" +
        "Crouch text: " + crouchText + "\n" +
        "Attack text: " + attackText + "\n" +
        "Reload text: " + reloadText + "\n" +
        "Sprint text: " + sprintText + "\n" ;
    }

}
```

This Client class is the concrete implementation of the GameInterfaceTexts. In the Client class, constructor has all the required text explanation of the specific actions and one GameHUDMediator. Getter and Setter methods **didn't** include code stated above for the sake of trying to eliminate the hubbub.

Each method in the Client class is very significant to keep the Mediator design pattern to implement smoothly. When a new Client is added to the game, or a user wants to change key bindings for that controller, setter methods will provide this option as a flexibility and the user may change any key binding wanted in order to increase playability. Moreover, game developers can create a new Client for a new type of controller. Users can change the controller type in run-time, if the controller type is used to control mediators by the game developers.

PlayerController Class

```
public class PlayerController {
  private GameHUDMediator mediator;

  public PlayerController(GameHUDMediator mediator) {
     this.mediator = mediator;
  }
  public void printHUDText() {
     System.out.println("HUD texts for current controller:\n");
     mediator.printHUDText();
  }
}
```

The PlayerController class is the one which communicates with the Mediator objects and obtains controller specified hud explanations.

Test Scenarios

For this flexibility requirement, client types will be inserted into the Mediator design pattern and printHUDText method will be called. After implementing each client type with controller specified HUD explanations, printHUDText

method should be able to collect all of them. This will prove that the Mediator design pattern is implemented successfully and this flexibility requirement can be used by game developers to change controller type in run-time.

First scenario: Only the Xbox client will be implemented.
Second scenario: After implementing the Xbox client, the PS4 client will be implemented.

After each scenario, all the text explanations will be called. Every implemented client should be seen printed. If we can see every implementation is printed, then we can assume that the Mediator design pattern is working.

Test Results

```
After first scenario:
Selection text: Press A to select
Interact text: Press X to interact
Crouch text: Press B to crouch
Attack text: Press RB to attack
Reload text: Press Y to reload
Sprint text: Hold RS to sprint

After second scenario:
Selection text: Press A to select
Interact text: Press X to interact
Crouch text: Press B to crouch
Attack text: Press RB to attack
Reload text: Press Y to reload
Sprint text: Hold RS to sprint

Selection text: Press X to select
Interact text: Press □ to interact
Crouch text: Press O to crouch
Attack text: Press R1 to attack
Reload text: Press △ to reload
Sprint text: Hold R3 to sprint
```

After reviewing the test results, we can clearly state that the Mediator design pattern was implemented successfully.

iv.Game itself should detect players who are cheating and penalize those players.

Stated flexibility requirement coded with the Java programming language to implement Interpreter design pattern. Static and dynamic diagrams can be seen above which explains relations and classes. Structure of the code, test cases and test result can be found below.

Context Class

```
import java.util.Stack;

public class Context {

  private Stack<String> context = new Stack<>();

  public Context() {
  }

  public Context(Stack<String> context) {
     this.context = context;
  }

  public String Lookup(Stack<String> context){
     if(context.peek()!= null){
        return context.pop();
     }
     else{
        return null;
     }
  }
}
```

Context class is the main class which meant to store all the input String itself and look into it. For the sake of simplicity, this class wasn't included in the test scenario.

CheatDetection Interface

```
public interface CheatDetection {
  public void GetRules();
  public boolean Evaluate(String context);
```

```
    public void Replace(String input, CheatDetection cheatDetection);
    public void Copy();
}
```

CheatDetection interface was implemented to keep order of Interpreter design pattern. All expressions have to include the CheatDetection rule so backbone organization of the code won't spoil.

LiteralExpression Class

```
public class LiteralExpression implements CheatDetection {

    private String name;

    public LiteralExpression(String name) {
        this.name = name;
    }

    @Override
    public boolean Evaluate(String context) {
        if(context.contains(name)){
            return true;
        }
        else{
            return false;
        }
    }

    @Override
    public void Replace(String input, CheatDetection cheatDetection) {
    }

    @Override
    public void Copy() {
        //Will implemented if needed.
    }

    @Override
    public void GetRules() {
        //Will implemented if needed.
    }

}
```

OrExpression Class

```
public class OrExpression implements CheatDetection {

  private CheatDetection operand1 = null;
  private CheatDetection operand2 = null;

  public OrExpression(CheatDetection operand1, CheatDetection operand2) {
     this.operand1 = operand1;
     this.operand2 = operand2;
  }

  @Override
  public boolean Evaluate(String context) {
     return operand1.Evaluate(context) || operand2.Evaluate(context);
  }

  @Override
  public void Replace(String input, CheatDetection cheatDetection) {
     //Will implemented if needed.
  }

  @Override
  public void Copy() {
     //Will implemented if needed.
  }

  @Override
  public void GetRules() {
     //Will implemented if needed.
  }
}
```

AndExpression Class

```
public class AndExpression implements CheatDetection {

  private CheatDetection operand1 = null;
  private CheatDetection operand2 = null;

  public AndExpression(CheatDetection operand1, CheatDetection operand2) {
     this.operand1 = operand1;
     this.operand2 = operand2;
```

```
    }

    @Override
    public boolean Evaluate(String context) {
        return (operand1.Evaluate(context) && operand2.Evaluate(context));
    }

    @Override
    public void Replace(String input, CheatDetection cheatDetection) {
    }

    @Override
    public void Copy() {
        //Will implemented if needed.
    }

    @Override
    public void GetRules() {
        //Will implemented if needed.
    }
}
```

PhysicsEngine Class

```
public class PhysicsEngine {

    public static CheatDetection andExp(){
        CheatDetection action1 = new LiteralExpression("SPRINT");
        CheatDetection action2 = new LiteralExpression("SHOOT");
        return new AndExpression(action1, action2);
    }

    public static CheatDetection orExp(){
        CheatDetection action1 = new LiteralExpression("SPRINT");
        CheatDetection action2 = new LiteralExpression("SHOOT");
        return new OrExpression(action1, action2);
    }

    public void CheckCheat(){
        CheatDetection isCheating1 = andExp();
        CheatDetection isCheating2 = orExp();

        System.out.println("Is player not cheating? " + isCheating1.Evaluate("SHOOT"));
```

```
    System.out.println("Is player not cheating? " + isCheating2.Evaluate("SHOOT"));
  }
  public static void main(String[] args) {
    PhysicsEngine physicsEngine = new PhysicsEngine();
    physicsEngine.CheckCheat();
  }
}
```

Test Scenarios

For testing the implementation of the Interpreter design pattern, we need to create simple languages such as AND and OR expressions. With the test input, we need to set the desired output to prove that the Interpreter design pattern is working accurately.

```
public static CheatDetection andExp(){
    CheatDetection action1 = new LiteralExpression("SPRINT");
    CheatDetection action2 = new LiteralExpression("SHOOT");
    return new AndExpression(action1, action2);
  }

  public static CheatDetection orExp(){
    CheatDetection action1 = new LiteralExpression("SPRINT");
    CheatDetection action2 = new LiteralExpression("SHOOT");
    return new OrExpression(action1, action2);
  }
```

This code section creates LiteralExpressions as AND and OR expressions and inserts test actions into them.

```
System.out.println("Is player not cheating? " + isCheating1.Evaluate("SHOOT"));
System.out.println("Is player not cheating? " + isCheating2.Evaluate("SHOOT"));
```

Finally, this code section will show us if the Interpreter design pattern is working or not.

Test Results

```
Is player not cheating? false
Is player not cheating? true
```

As stated above, Interpreter design pattern implementation has been successfully achieved and it bringed flexibility to the game.

v.Game minimap should display enemy objects proportional to their sound level.

Display Interface

```
public interface Display {
  public void Display();
}
```

Display interface is the main strategy for the design pattern implementation. All objects which needs to be displayed in the minimap should implement the Display interface. This structure can easily modified.

Jet Display

```
public class JetDisplay implements Display {

  private double speed;
  private boolean isActive;
  private String situation;

  public void setSpeed(double speed) {
     this.speed = speed;
  }

  public void setIsActive(boolean isActive) {
```

```
    this.isActive = isActive;
}

public JetDisplay(double speed, boolean isActive, String situation) {
    this.speed = speed;
    this.isActive = isActive;
    this.situation = situation;
}

public String getSituation() {
    return this.situation;
}

public void setSituation(String situation) {
    this.situation = situation;
}

public void print() {
    if(!isActive){
        System.out.println("Jet will not be displayed.");
    }
    else{
        System.out.println("Jet is in the mode of: " + situation + "\n");
    }

}

@Override
public void Display(){
    print();
    int a, b;
    int k = (int)Math.round(speed) / 10;
    for (a = 1; a <= k; a++) {
        for (b = 1; b <= k; b++) {
            if (a == 1 || a == k || b == 1 || b == k
                || a == b || b == (k - a + 1))
                System.out.print("*");
            else
                System.out.print(" ");
```

```
            }
            System.out.println();

        }
        System.out.println("Jet");
    }
}
```

JetDisplay is the one of the classes which implements the Display class. This can be an example for an object to be shown in the minimap by just creating a new class or modifying the display method. We can see by other classes, strategy design pattern bring flexibility.

CarDisplay

```
public class CarDisplay implements Display {

  private double speed;
  private boolean isActive;
  private String situation;

  public void setSpeed(double speed) {
     this.speed = speed;
  }

  public void setIsActive(boolean isActive) {
     this.isActive = isActive;
  }

  public CarDisplay(double speed, boolean isActive, String situation) {
     this.speed = speed;
     this.isActive = isActive;
     this.situation = situation;
  }

  public String getSituation() {
     return this.situation;
```

```
    }

    public void setSituation(String situation) {
        this.situation = situation;
    }

    public void print() {
        if(!isActive){
            System.out.println("Car will not be displayed.");
        }
        else{
            System.out.println("Car is in the mode of: " + situation + "\n");
        }

    }

    @Override
    public void Display(){
        print();
        int a, b;
        int k = (int)Math.round(speed) / 3;
        int l = k + k / 2;
        for (a = 1; a <= k; a++) {
            for (b = 1; b <= l; b++) {
                if (a == 1 || a == k || b == 1 || b == l)
                    System.out.print("*");
                else
                    System.out.print(" ");
            }

            System.out.println();
        }
        System.out.println("Car");
    }

}
```

PersonDisplay Class

```
public class PersonDisplay implements Display {
```

```
private double speed;
private boolean isActive;
private String situation;

public void setSpeed(double speed) {
   this.speed = speed;
}

public void setIsActive(boolean isActive) {
   this.isActive = isActive;
}

public PersonDisplay(double speed, boolean isActive, String situation) {
   this.speed = speed;
   this.isActive = isActive;
   this.situation = situation;
}

public String getSituation() {
   return this.situation;
}

public void setSituation(String situation) {
   this.situation = situation;
}

public void print() {
   if(!isActive){
      System.out.println("Person will not be displayed.");
   }
   else{
      System.out.println("Person is in the mode of: " + situation + "\n");
   }
}

@Override
public void Display(){
   print();
   int a, b;
   int k = (int)Math.round(speed) / 3;
   int l = k;
```

```
    for (a = 1; a <= k; a++) {
      for (b = 1; b <= l; b++) {
        if (a == 1 || a == k || b == 1 || b == l)
          System.out.print("*");
        else
          System.out.print(" ");
      }
      System.out.println();
    }
    System.out.println("Person");
  }

}
```

Those 2 classes shown above which are CarDisplay and PersonDisplay has been implemented for the sake of proving the flexibility of the pattern and provide much more test outputs.

GameWorld Class

```
public class GameWorld {
  public static void main(String[] args) {
    Minimap minimap = new Minimap();

    minimap.addDisplay(new JetDisplay(206.1, true, "Flying"));
    minimap.addDisplay(new CarDisplay(22.9, true, "Driving"));
    minimap.addDisplay(new PersonDisplay(14.9, true, "Sprinting"));
    minimap.addDisplay(new PersonDisplay(0.01, false, "Standing"));

    minimap.displayMinimap();
  }
}
```

GameWorld class implemented as a client. It controls the minimap by creating it and then adding new objects into the minimap.

Test Scenarios

By implementing the Strategy design pattern to solve stated problem, a client class which is the GameWorld class is used. GameWorld class creates a new minimap and inserts objects into it by specifying the type of the object and the information of the object. After this part, we will able to see if Strategy pattern works if the desired output achieved.

Test Results

Jet
↓ next object ↓

Car is in the mode of: Driving

Car
↓ next object ↓

Person is in the mode of: Sprinting

Person
↓ next object ↓

Person will not be displayed.
Person

As shown in the output of the strategy design pattern above, test results states that implementation is successful and everything is working. Therefore, a flexibility bringed to the game itself with this problem solution.

What We Learned and Conclusions

In this section, we will explain what we have learned and what problems we have encountered during the project and design pattern implementations.

a)Knowledge about Patterns

Before this course, I didn't know the idea of design patterns existed. With this course, I have learned the definition of pattern, design pattern and flexibility. While learning the definitions and trying to find candidate patterns for flexibility requirements, I learned general purpose and usage of the design patterns. Which design pattern is the most suitable design pattern for every problem is the one of the most important questions while implementing the process. This process made me able to pick and implement the correct design pattern to the problems.

While implementing the design pattern, I have acknowledged how relationships and interactions between classes (Static Class Diagram and Dynamic Class Diagram) works. Most important section is that design patterns bring flexibility and sustain maintainability for the code.

b)Problems Encountered During Project

- **Correlation Between Pattern and Problem**

Before this project, I have never tried to solve problems using design patterns. Therefore, I needed to adapt and learn how to approach problems by using design patterns. To achieve this, I had to change my classical kind of thinking and try to bring a solution but try to approach by flexibility requirements of the problem and trying to find which design pattern is the most suitable one. Trying to connect each pattern with the problem was significantly hard for me because of the inexperience.

- **Pattern Design and Implementation**

I had learned that designing a pattern and implementing it into a problems may have very other ways to execute. The most important step is here, following the static and dynamic class diagrams which are created by the thinking process between pattern and problem. If one can focus on those diagrams, implementations will be much easier. I had difficulties focusing between diagrams and implementation for the first try.

- **Testing**

In the testing part, I have tested the implementation I have made with the stated design patterns and problems. This was the perfect way to understand if the implementation is working or not. For my first implementations, I really struggled to write a comprehensive test case.

c)Conclusion

In the report, I have determined 10 flexibility requirements for the architecture of war game report and each of the requirements have candidate patterns which fulfills the requirement. I compared the candidate patterns and choosed the main pattern for each flexibility requirement. After choosing the stage, I described the problem in detail and showed the solution with the reason why I chose that pattern. After that stage, I also described the problem and solution with Static Class and Dynamic Class diagrams. After the implementation, I showed the code I coded with the test code. After explaining each code block's mission, I talked about the test cases and finally presented test results. By presenting the test result, I have concluded that those implementations are working. With this process, I have managed to implement a design pattern from bottom to top with the thinking, arguing and testing stages.

3) Side System: Artificial Intelligence

(Author: Elçin Duman)

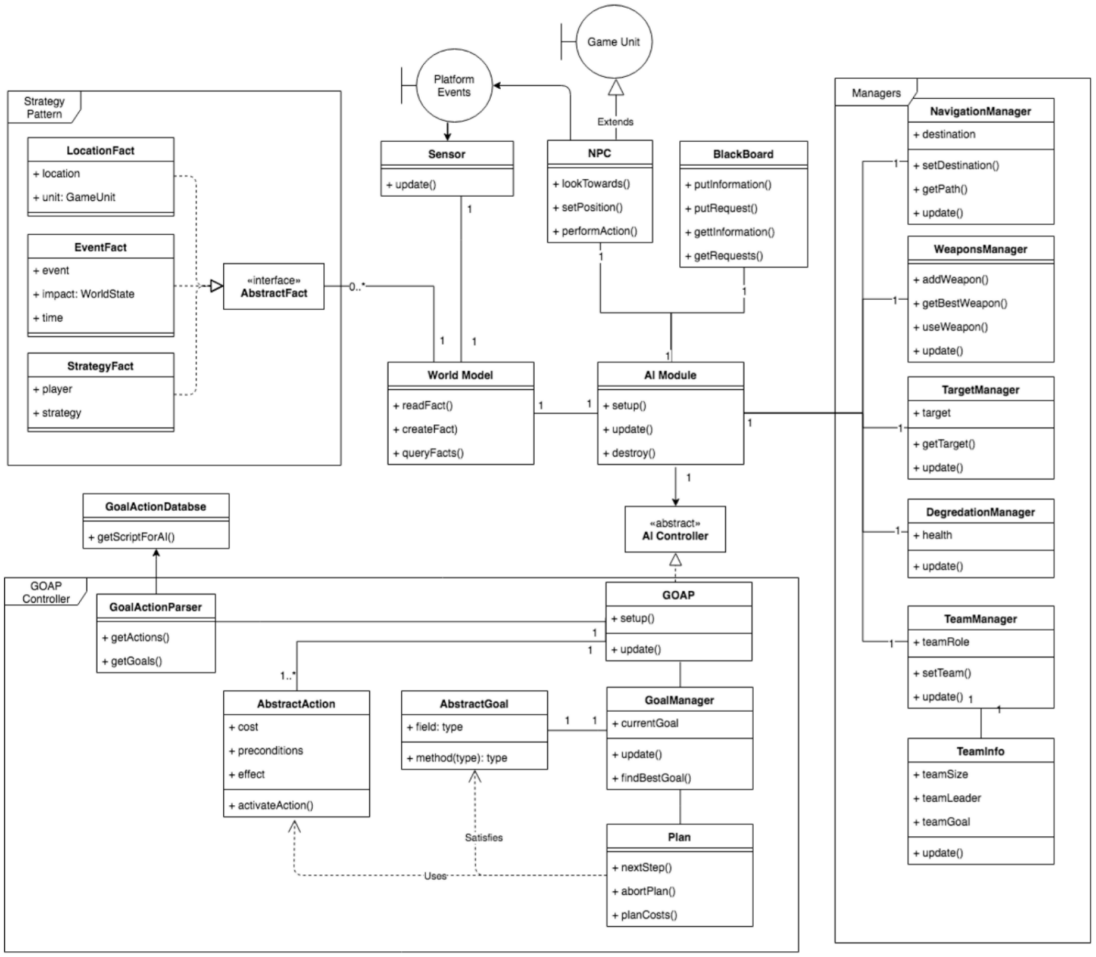
b) Sharing the domains of the game

In this part of the report, the sharing of domains amongst ourselves is going to be explained.

The game's architecture has 4 main domains. Those are Game Engine, Game Unit, Artificial Intelligence and Platform domains. Since we are four people writing this report, each one of us has chosen one domain and made a detailed analysis for the domain we chose, found flexibility requirements on our domains and we found the suitable Design Patterns to solve these specific problems. Design Patterns have been compared to each other and one Design Pattern has been selected to solve a requirement problem. Then each one of us implemented five of these Design Patterns by making their static class diagram, dynamic sequence diagram and writing the Design Patterns code in a programming language. Lastly we tested the implementation if they solve the requirement problems.

c) Flexibility Requirements

In this step, we defined the flexibility requirements of the Artificial Intelligence domain of the game. While determining the flexibility requirements, we considered the project's Artificial Intelligence report and examined the architecture of the AI subdomain of the game.



Above, you can clearly see the Artificial Intelligence architecture of the game. By examining this architecture, we are able to determine flexibility requirements and needs of the AI subdomain of the game.

1) **Variety of Non Player Character (NPC) families should be produced by the AI component of the game.**

   The report that is given to us emphasizes a single type of NPC family in the whole war game. However, in war games there should be more than one type of NPC family since a variety of NPCs can be produced by AI algorithms. NPCs can advance the experience and enjoyment from the game. It can also be challenging to deal with multiple villains or harmful NPCs.

   The war game must be able to produce multiple families of NPCs. These NPCs should be in different types such as good characters or bad characters. Some NPCs should be harmful while others must not harm our main character and their felonies. These NPCs should be generated inside of the Artificial

Intelligence subdomain of the game. NPCs should be consistent with their actions, meaning that a good character should not harm or attack our main character.

2) **Goals and actions that are represented as a language should be interpreted by the Goal Oriented Action Planning (GOAP) controller.**

   In the report, it was stated that goals and actions of characters are stored inside of a Goal Action Database. GOAP controller reaches those datas via a parser. However, this solution all by itself is not suitable for our game since it is not mentioned how the parser works in our report. Interpretations of the languages are not clearly stated.

   When a new scenario or a level is added to the system, it generates new actions and goals, however it is not mentioned how the system will work after we add new actions and goals. The interpretation of the languages should be flexible. When new levels and new scenarios are added to the game, the interpreter should work correctly with the newly added languages.

   It is clear that we are in need of an interpreter which interprets the given language. There should be a class hierarchy for the grammar of actions and goals.

3) **Goal Oriented Action Planning (GOAP) controller should update goals and multiple actions when changes are made in the AI Module.**

   Goal Oriented Action Planning (GOAP) controller is a partitioned system that has cooperating classes. In the report that was given to us states that GOAP has Action Manager and Goal Manager classes that are dependent on the controllers updates. It is obvious that these classes should maintain consistency in order to work correctly and efficiently.

   When the GOAP controller receives new information from the AI Module, we should notify our Action Manager and Goal Manager since new goals and actions will be determined by the notifications that the GOAP controller sends. In addition to all of these, the GOAP controller may have other Manager classes that manage actions and goals. Supporting broadcast communication is an important aspect for this requirement. By endorsing information exchanges with multiple classes we are able to make this requirement flexible.

4) **Coordination between Manager objects should be done effectively.**

Coordination between the manager objects have an important effect on our game play. Manager classes are defined as Navigation, Weapon, Target, Degradation and Team. All of these stores the necessary information that determines the course of the game.

In the report, Managers are expected to be independent, meaning that they would not exchange information with each other. They would directly send information to the Blackboard. However in that case, blackboard does not support the information exchange between managers. Therefore, only information supported by the blackboard can be exchanged. Coordination problems may arise due to this design, since Manager objects may communicate with each other and exchange crucial information. Manager objects behaviors' and coordinations should be separated from each other.

Information exchange and dependency problems must be solved in order to have an efficient game play. Manager objects should be coordinated accordingly. By doing so, we are expecting to have a flexible Manager system in our war game.

5) **The World Model should process different kinds of datas.**

The World Model is initially responsible for reading, querying and creating facts. However, this model receives a variety of data from Sensors, Facts and Artificial Intelligence modules. The World Model should handle datas in order to update the world of the game. This model should concrete on requesting and receiving datas from multiple sources.

6) **Sensors should handle the unwanted information that is coming from the Platform subdomain of the game.**

Sensor system should have implemented a request handling system for incoming messages from the Platform subdomain. Unwanted information or other actions may come from the Platform domain, therefore, we are in need of implementing an error or data handler system for this class. Sensors are crucial for our system since AI managers depend on the environment to choose what they will do. For this reason, we should make the sensor class flexible.

7) **The Artificial Intelligence module should update its core classes according to the changes made in other subdomains of the game.**

   The Artificial Intelligence domain updates the whole system, however it does not have any flexible component for updating different types of classes separately. We should implement a design pattern for making the AI module notify its subscribers automatically in that case. By updating the whole system, we are able to create a more flexible AI domain for the Artificial Intelligence Module.

Possible Design Patterns to Solve Requirements

Candidate patterns that may meet the requirements were determined. In this section you can find the table of possible design patterns and the comparison between those patterns.

Flexibility Requirement	**Possible Design Patterns to Solve**
Variety of Non Player Character (NPC) families should be produced by the AI component of the game.	Abstract Factory, Factory Method, Prototype
Goals and actions that are represented as a language should be interpreted by the Goal Oriented Action Planning (GOAP) controller.	Interpreter, State
Goal Oriented Action Planning (GOAP) controller should update goals and multiple actions when changes are made in the AI module.	Observer, Publisher Subscriber
Coordination between Manager objects should be done effectively.	Mediator, Facade
The World Model should process different kinds of datas.	Command, Chain of Responsibility
Sensors should handle the unwanted information that is coming from the Platform subdomain of the game.	Chain of Responsibility

The Artificial Intelligence module should update its core classes according to the changes made in other subdomains of the game.	Observer, Publisher Subscriber

1) **Variety of Non Player Character (NPC) families should be produced within the AI subdomain of the game.**

   For the first requirement Abstract Factory, Factory Method and Prototype patterns are chosen as the candidate patterns to solve the requirement.

   A variety of NPC families should be produced by the system. This can be done by the Abstract Factory or Factory Method since with the usage of these methods we are able to create related objects. However, the Factory Method does not provide the consistency that we need amongst the NPC families, therefore usage of Factory Method is not appropriate for our solution. Prototype pattern is also a creational design pattern that lets us copy existing objects with the clone method. In the prototype pattern, cloning complex objects which have circular references might be challenging. Initializing clones can be tricky when we want to initialize internal state values. In addition to all of these, managing prototypes can be difficult and we may have to use a prototype manager for this specific purpose.

   Therefore, usage of Abstract Factory pattern should be eligible for this problem. Abstract factory provides consistency among NPCs and it also makes exchanging product families easy. Since we want to have more than one product family, we can use the Abstract Factory to provide flexibility for this requirement.

2) **Goals and actions that are represented as a language should be interpreted by the Goal Oriented Action Planning (GOAP) controller.**

   For the second requirement, Interpreter and State patterns are chosen as the candidate pattern to solve the requirement.

   State pattern is a structural design pattern that aims to change the algorithm in run time. However, when using the State pattern adding new states can

sometimes be hard and it can be challenging to implement a new state.

Interpreter pattern is a behavioral design pattern that interprets sentences in a given language. It is easy to change and extend the grammar because the pattern uses classes to represent grammar rules. Adding new ways to interpret expressions is also simple, since existing expressions can be modified incrementally. Implementation of the grammar is uncomplicated and it can be automated with a parser generator. As a result of that, usage of Interpreter pattern is suitable for handling the interpretations of actions and goals.

3) **Goals Oriented Action Planning (GOAP) controller should update goals and multiple actions when new changes are made in the AI module.**

For the third requirement, Observer and Publisher-Subscriber patterns are selected as the candidate patterns to solve the requirement.

Publisher-Subscriber pattern can be implemented for this problem because it's a behavioral design pattern that is similar to Observer pattern. This pattern is mostly used in distributed systems since there can be many subscribers beyond our scope. In publisher-subscriber pattern, there may be a third component that filters all incoming updates and distributes accordingly. However, in our AI subdomain, the GOAP controller does not implement a distributed system and we do not need a third component that would filter the notifications of managers.

The Observer pattern is suitable for this kind of problem since the Observer pattern defines a one-to-many dependency between objects. When a state is changed, all the dependent objects are notified accordingly. In our situation, the GOAP controller should notify the Goal and Action Manager when an internal change is made in the AI module. These changes can affect goals and actions of players. In order to keep the game consistent, we should reflect changes in the manager classes, therefore, implementation of the Observer pattern is suitable for maintaining flexibility.

4) **Coordination between Manager objects should be done effectively.**

In this problem, the Mediator and Facade design patterns are chosen as the candidate pattern to solve the requirement.

Facade is a structural design pattern that provides a unified interface to a set of interfaces in a subsystem. Facade makes the subsystem easier to use. However, Facade patterns protocol is unidirectional. Facade objects make requests of the

subsystem classes but not vice versa [1]. It can constrain our Manager classes and how they interact with other objects, therefore, usage of the Facade pattern is not suitable for this requirement.

The Mediator is a behavioral design pattern that promotes loose coupling by keeping objects from referring to each other explicitly. Mediator enables cooperative behavior and the protocol is multidirectional. It makes object interactions independently. The Mediator pattern separates the coordination and the behavior of the object.

The Mediator pattern centralizes control. In our situation, implementing the Mediator pattern could fix the issue of dependency and communication problems between Managers. With the mediator, we may identify Manager objects independently and those objects can cooperate separately from each other. By coordinating manager objects our range of action can expand and the gameplay experience is more likely to be realistic.

5) **The World Model should process different kinds of datas.**

For this problem, Command pattern and Chain of Responsibility patterns are chosen as the candidate patterns to solve the requirement.

Chain of Responsibility is a behavioral pattern that handles the request by chaining objects. More than one object may handle a request and the handler does not know which object will handle the request. In our case, the Chain of Responsibility pattern is not suitable since there is only one method for handling the request. We could have more than one request at this point, since our game is a war game that has a dynamic structure. The World Model is always updated by sensors, artificial intelligence and fact modules, therefore, we may have more than one request at this point in the game. Receipt is also not guaranteed in this pattern.

The Command pattern can be implemented for solving this requirement. This pattern is a behavioral design pattern that supports multiple requests. Since this is a war game, we may have more than one request. In command pattern, each of the objects may perform different kinds of operations such as querying, reading or creating facts. By extending the World Models interface, we would no longer have a fixed interface problem.

In order to process different kinds of requests, the World Model should implement the Command pattern. By doing so, we would make the games world

more flexible and attainable.

6) **Sensors should handle the unwanted information that is coming from the Platform subdomain of the game.**

   For this problem, the Chain of Responsibility pattern is chosen as the candidate pattern to solve the requirement. Chain of Responsibility is a behavioral pattern that handles the request by chaining objects. More than one object may handle a request and the handler does not know which object will handle the request. In our case, it is suitable for us to have a Chain of Responsibility pattern to solve the requirement that has been given to us. By implementing Chain of Responsibility for different kinds of datas, we can make the Sensors class flexible.

7) **The Artificial Intelligence module should update its core classes according to the changes made in other subdomains of the game.**

   Observer and Publisher-Subscriber patterns are selected as the candidate patterns to solve the requirement. For this problem, we may implement the Publisher-Subscriber design pattern since we can subscribe or unsubscribe the classes easily with this method. However, the Publisher Subscriber pattern is so loosely coupled that we don't even know most of the core components. Therefore, the Observer Pattern is suitable for this requirement. As it is described in the third requirement, it is clear that we should add an Observer Pattern for the AI Module of the game.

e) Problem Description

Five of the flexibility requirements are selected to solve. In this section, you can find the patterns suitable for each problem. This section also contains the information of the reason why we implemented these selected patterns. Static and dynamic class diagrams are also shown in the sections.

e.1) Problem Definition

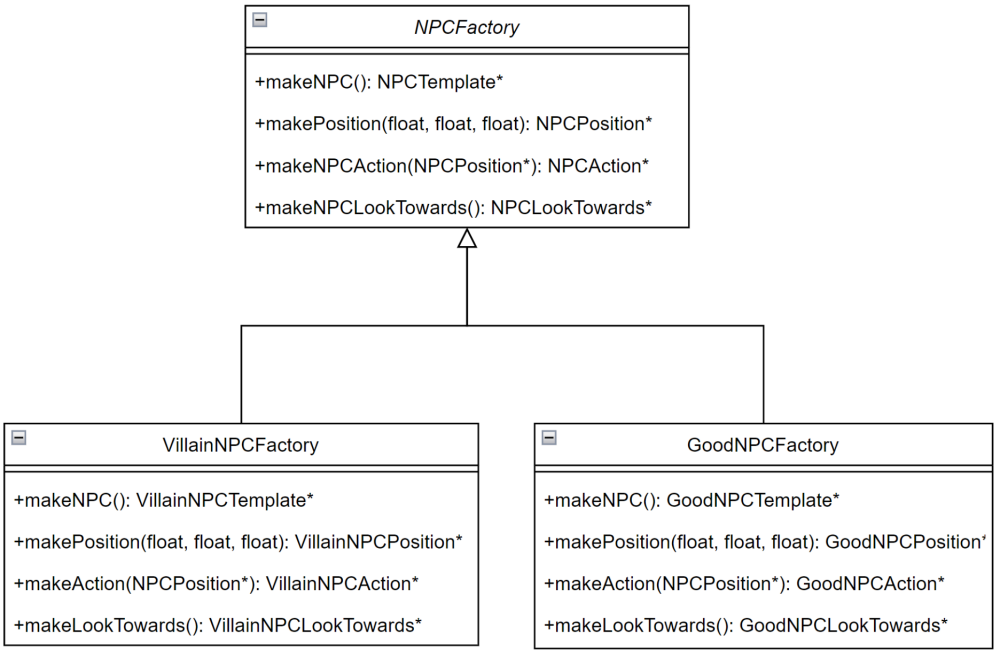
Variety of Non Player Character (NPC) families should be produced by the AI subdomain of the game.

The report that was given to us only contains one generic implementation of a NPC family. In our game, one type of NPC family is not enough since a war game should contain more than one type of NPC families in order to be successful and entertaining in the market. By implementing the abstract factory

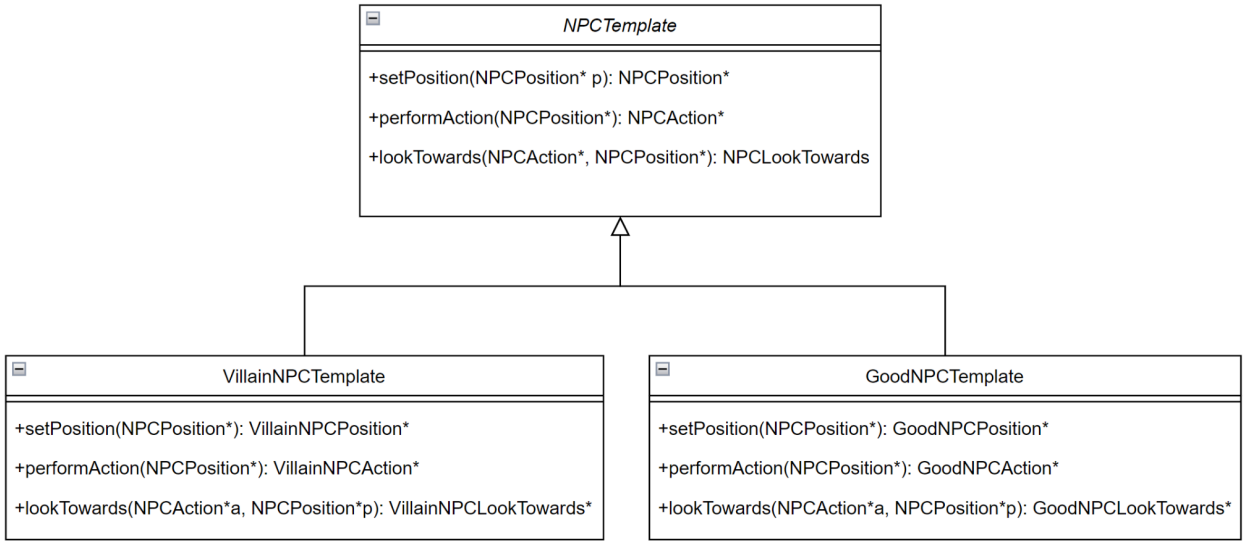
pattern, we are able to produce families of related objects without specifying their concrete classes.

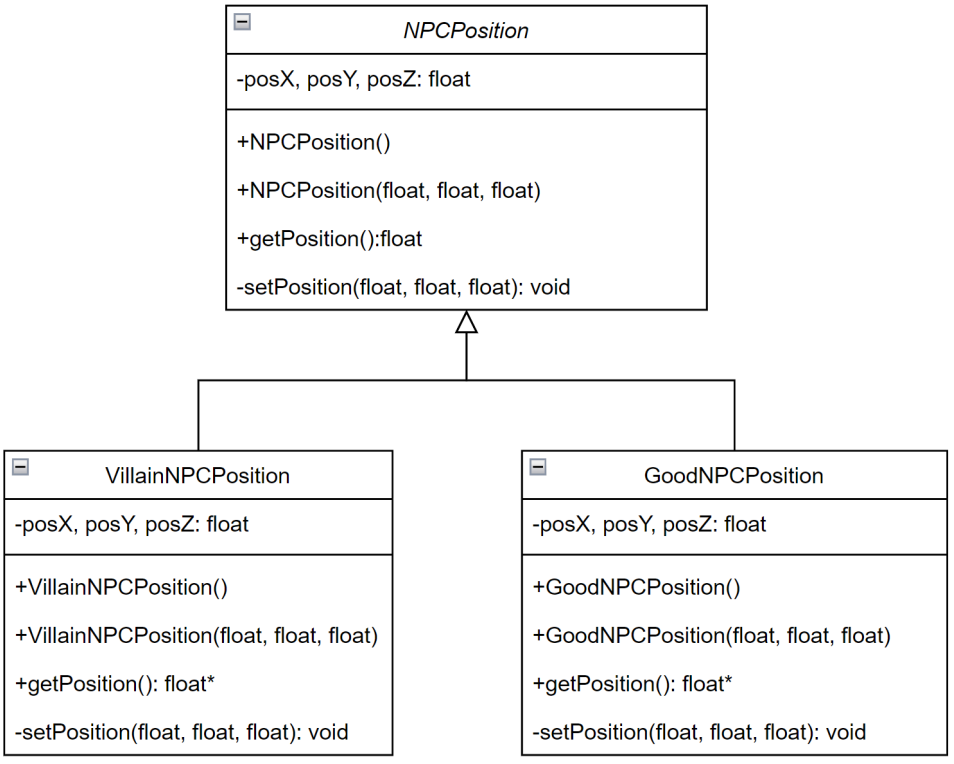
Static Class Structure

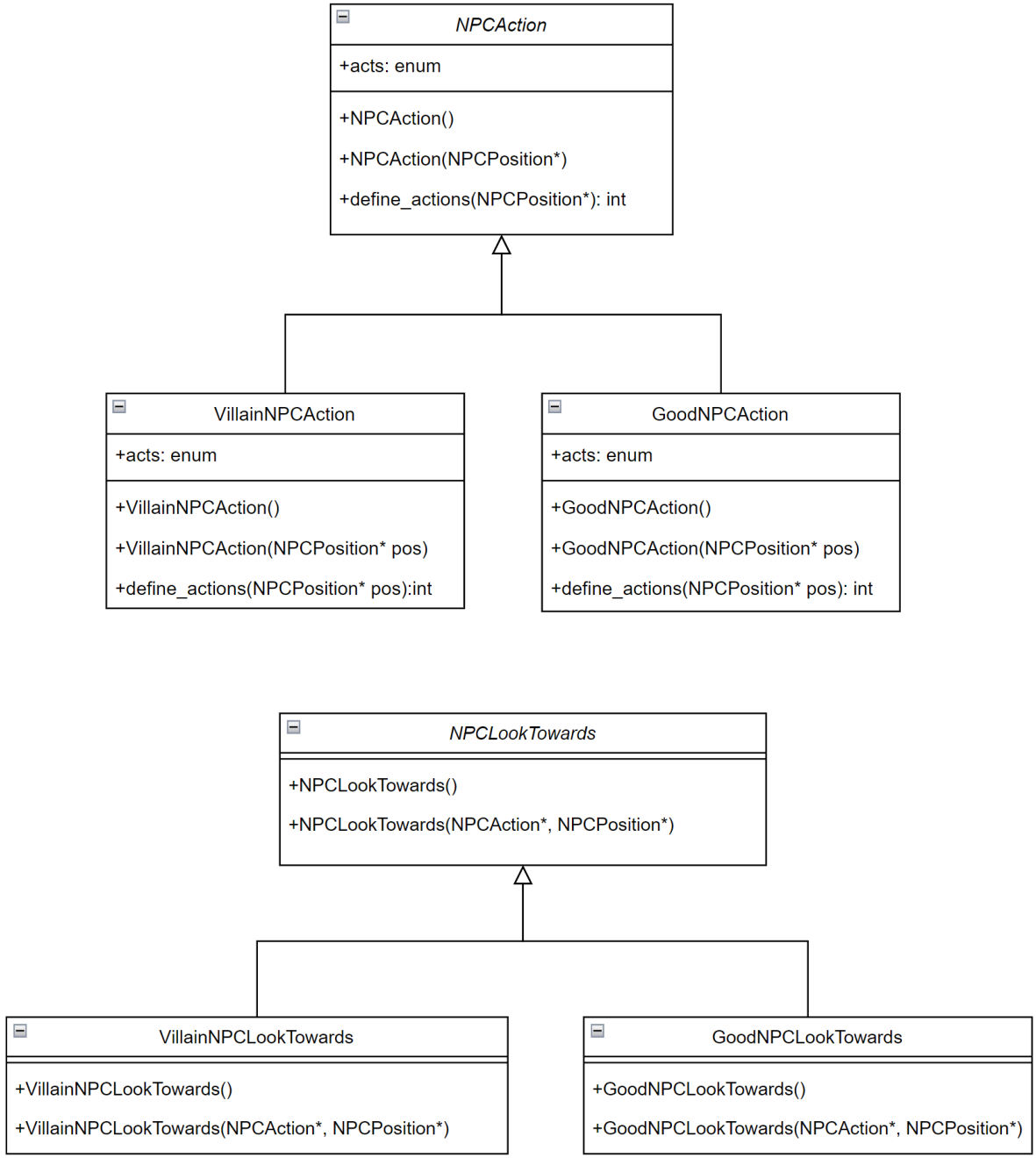
The diagrams that are created aims to check the accuracy and consistency of our solution with the usage of the Abstract Factory design pattern. C++ programming language is chosen as the implementer of the flexibility requirement.



The table above contains the UML class diagram for the Abstract Factory pattern that will be used to solve the problem. However, when implementing the Abstract Factory pattern, we should also show the product families and their hierarchies.


NPCTemplate
+setPosition(NPCPosition* p): NPCPosition*
+performAction(NPCPosition*): NPCAction*
+lookTowards(NPCAction*, NPCPosition*): NPCLookTowards
VillainNPCTemplate
+setPosition(NPCPosition*): VillainNPCPosition*
+performAction(NPCPosition*): VillainNPCAction*
+lookTowards(NPCAction*a, NPCPosition*p): VillainNPCLookTowards*
GoodNPCTemplate
+setPosition(NPCPosition*): GoodNPCPosition*
+performAction(NPCPosition*): GoodNPCAction*
+lookTowards(NPCAction*a, NPCPosition*p): GoodNPCLookTowards*

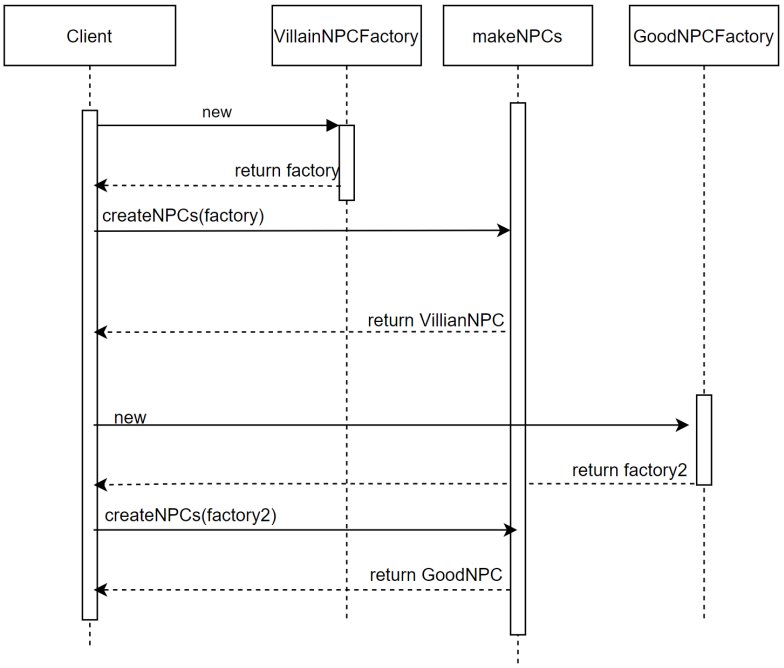

NPCPosition
-posX, posY, posZ: float
+NPCPosition()
+NPCPosition(float, float, float)
+getPosition():float
-setPosition(float, float, float): void
VillainNPCPosition
-posX, posY, posZ: float
+VillainNPCPosition()
+VillainNPCPosition(float, float, float)
+getPosition(): float*
-setPosition(float, float, float): void
GoodNPCPosition
-posX, posY, posZ: float
+GoodNPCPosition()
+GoodNPCPosition(float, float, float)
+getPosition(): float*
-setPosition(float, float, float): void



Tables above contain the UML class diagrams for NPC Template, NPC Position, NPC Action and NPC Look Towards. These diagrams represent product hierarchies and families. We can now create different types of NPCs with consistent features.

Dynamic Structure

The sequence diagram is given below to make the Abstract Factory pattern understandable by seeing the interaction of the objects.



Implementation of the Design Pattern

In this part, we are going to show the implementation of the flexibility requirements that were discussed in previous chapters. This section aims to verify the application of patterns. While implementing the design patterns, C++ programming language is chosen as the main implementer of the patterns. In this section, you can find the structure of the code, test cases and test results.

1) Variety of Non Player Characters (NPCs) should be produced by the AI subdomain of the game.

For this problem, we must create the product hierarchies and we must show the factories that are responsible for the creation of consistent object classes.

Below we have shown the NPC Factories as we defined in the NPC Factory static class diagram. When we examine the codes, we can clearly see that NPC Factory is an interface and Villain NPC Factory and Good NPC Factory inherit from the NPC Factory interface.

NPC Factory

```
class NPCFactory {
public:
	virtual NPCTemplate* makeNPC() const {
		return new NPCTemplate;
	}
	virtual NPCPosition* makePosition(float x, float y, float z)
const {
		return new NPCPosition(x, y, z);
	}
	virtual NPCAction* makeAction(NPCPosition* pos) {
		return new NPCAction(pos);
	}
	virtual NPCLookTowards* makeLookTowards(NPCAction* act) {
		return new NPCLookTowards(act);
	}
};

class VillainNPCFactory : public NPCFactory {
public:

	virtual VillainNPCTemplate* makeNPC() const override {
		return new VillainNPCTemplate;
	}
	virtual NPCPosition* makePosition(float x, float y, float z)
const override {
		return new VillainNPCPosition(x, y, z);
	}
	virtual NPCAction* makeAction(NPCPosition* pos) override {
		return new VillainNPCAction(pos);
	}
	virtual NPCLookTowards* makeLookTowards(NPCAction* act)
override{
		return new VillainNPCLookTowards;
	}
};

class GoodNPCFactory : public NPCFactory {
public:

	virtual GoodNPCTemplate* makeNPC() const override {
		return new GoodNPCTemplate;
	}
```

```
	virtual NPCPosition* makePosition(float x, float y, float z)
const override {
		return new GoodNPCPosition(x, y, z);
	}
	virtual NPCAction* makeAction(NPCPosition* pos) override {
		return new GoodNPCAction(pos);
	}
	virtual NPCLookTowards* makeLookTowards(NPCAction* act) {
		return new GoodNPCLookTowards;
	}
};
```

As we can see from above, these are factories that are responsible for creating the objects. Now, we should also show the product hierarchies of NPC Template, NPC Position, NPC Action and NPC Look towards.

Product Hierarchies

This section contains the product hierarchies of position, action, template and look towards.

```
class NPCPosition {
public:
	NPCPosition() {
		setPosition(0,0,0);
	}
	NPCPosition(float x, float y, float z) {
		setPosition(x, y, z);
	}
	virtual float* getPosition() { }
private:
	float posX, posY, posZ;
	virtual void setPosition(float x, float y, float z) { }
};

class VillainNPCPosition : public NPCPosition {
public:
	VillainNPCPosition() {
		setPosition(0, 0, 0);
	}
	VillainNPCPosition(float x, float y, float z) {
		setPosition(x, y, z);
	}
```

```
	virtual float* getPosition() override{
		float arr[3] = { posX, posY, posZ };
		return arr;
	}
private:
	float posX, posY, posZ;
	virtual void setPosition(float x, float y, float z) override
{
		posX = x;
		posY = y;
		posZ = z;
	}
};

class GoodNPCPosition : public NPCPosition {
public:
	GoodNPCPosition() {
		setPosition(0, 0, 0);
	}
	GoodNPCPosition(float x, float y, float z) {
		setPosition(x, y, z);
	}
	virtual float* getPosition() override {
		float arr[3] = { posX, posY, posZ };
		return arr;
	}
private:
	float posX, posY, posZ;
	virtual void setPosition(float x, float y, float z) override
{
		posX = x;
		posY = y;
		posZ = z;
	}
};

/*
 * NPC Action object class.
 */

class NPCAction {
public:
	Actions acts;
	NPCAction() {}
```

```
	NPCAction(NPCPosition* pos) {
	}
	virtual int define_actions(NPCPosition* pos) {
		return 0;
	}
};

class VillainNPCAction : public NPCAction {
public:
	Actions acts;
	VillainNPCAction() {

	}
	VillainNPCAction(NPCPosition* pos) {
		define_actions(pos);
		if (acts == Actions::ATTACK) {
			std::cout << "Performed action is ATTACK. " <<
std::endl;

		}
		else {
			std::cout << "Performed action is HARM. " <<
std::endl;
		}
	}
	virtual int define_actions(NPCPosition* pos) {
		float* positions = pos->getPosition();
		if (positions[0] == 20) {
			acts = Actions::ATTACK;
			return 1;
		}
		else {
			acts = Actions::HARM;
			return 5;
		}
	}
};

class GoodNPCAction : public NPCAction {
public:
	Actions act;

	GoodNPCAction() {
```

```
	}
	GoodNPCAction(NPCPosition* pos) {
		define_actions(pos);
		if (acts == Actions::HELP) {
			std::cout << "Performed action is HELP. " <<
std::endl;

		}
		else {
			std::cout << "Performed action is DO NOTHING. " <<
std::endl;
		}
	}
	virtual int define_actions(NPCPosition* pos) {
		float* positions = pos->getPosition();
		if (positions[0] == 20) {
			acts = Actions::HELP;
			return 3;
		}
		else {
			acts = Actions::DO_NOTHING;
			return 4;
		}
	}
};

/*
 * NPC Look Towards object class.
 */

class NPCLookTowards {
public:
	NPCLookTowards() {

	}
	NPCLookTowards(NPCAction* act, NPCPosition* pos) {
	}
};
class VillainNPCLookTowards : public NPCLookTowards {
public:
	VillainNPCLookTowards() {
		std::cout << "Initialized villain's look towards." <<
std::endl;
	}
```

```
	VillainNPCLookTowards(NPCAction* act, NPCPosition* pos) {

		int x = act->define_actions(pos);

		if (x==2) {
			std::cout << ("Looking towards an attack.")<<
std::endl;
		}
		else {
			std::cout << "Looking towards a harm." <<
std::endl;
		}
	}
};
class GoodNPCLookTowards : public NPCLookTowards {
public:
	GoodNPCLookTowards() {
		std::cout << "Initialized good NPC's look towards." <<
std::endl;
	}
	GoodNPCLookTowards(NPCAction* act, NPCPosition* pos) {

		int x = act->define_actions(pos);

		if (x==3) {
			std::cout << ("Looking towards a help.") <<
std::endl;
		}
		else {
			std::cout << "Nothing to look towards." <<
std::endl;
		}
	}
};

/*
 * NPC Template object class.
 */

class NPCTemplate {
public:
	NPCTemplate() {

	}
```

```
	virtual NPCPosition* setPosition(NPCPosition* p) {
		float* arr = p->getPosition();
		return new NPCPosition(arr[0], arr[1], arr[2]);
	}
	virtual NPCAction* performAction(NPCPosition* p) {
		return new NPCAction(p);
	}
	virtual NPCLookTowards* lookTowards(NPCAction* a,
NPCPosition* p) {
		return new NPCLookTowards(a,p);
	}
};

class VillainNPCTemplate : public NPCTemplate {
public:
	VillainNPCTemplate() {

	}
	virtual NPCPosition* setPosition(NPCPosition* p) override {
		std::cout << "Positioned as the villain of the game."
<< std::endl;
		std::cout << "Coordinates are: " << p->getPosition()[0]
<< " " << p->getPosition()[1] << " " << p->getPosition()[2] <<
std::endl;
		return new VillainNPCPosition(p->getPosition()[0],
p->getPosition()[1], p->getPosition()[2]);
	}
	virtual NPCAction* performAction(NPCPosition* p) override {
		return new VillainNPCAction(p);
	}
	virtual NPCLookTowards* lookTowards(NPCAction* a,
NPCPosition* p) override {
		return new VillainNPCLookTowards(a, p);
	}
};

class GoodNPCTemplate : public NPCTemplate {
public:
	GoodNPCTemplate() {

	}
	virtual NPCPosition* setPosition(NPCPosition* p) override {
		float* arr = p->getPosition();
		std::cout << "Positioned as the good character of the
```

```
game." << std::endl;
            std::cout << "Coordinates are: " << p->getPosition()[0]
<< " " << p->getPosition()[1] << " " << p->getPosition()[2] <<
std::endl;
            return new GoodNPCPosition(p->getPosition()[0],
p->getPosition()[1], p->getPosition()[2]);
      }
      virtual NPCAction* performAction(NPCPosition* pos) override {
            return new GoodNPCAction(pos);
      }
      virtual NPCLookTowards* lookTowards(NPCAction* a,
NPCPosition* p) override {
            return new GoodNPCLookTowards(a, p);
      }
};
```

Above codes are the representation of the product hierarchies.

Test Scenario

When the implementation is done, we have generated an algorithm in order to create different types of NPC families. The scenario is written in C++ programming language. We have positioned, performed and looked towards different NPCs and determined which actions that they are performing.

We should make sure that NPC classes are consistent and do not interrupt each other. In test scenarios NPC families belong to either the Good NPC class or the Villain NPC class. There are also different action sets for Good and the Villain NPCs. Good NPCs can do nothing or they can help our character, while villain NPCs can attack or harm our character. Those actions should not interfere with each other, meaning that the Good NPC should not harm or attack our character.

First, we should introduce the create NPC algorithm that uses these factories.

```
NPCTemplate* makeNPCs::createNPCs(NPCFactory* factory)
{
      NPCTemplate* aNPC = factory->makeNPC();
      NPCLookTowards* look = factory->makeLookTowards();
      NPCPosition* pos1 = factory->makePosition(20, 10, 1);
      NPCAction* action = factory->makeAction(pos1);

      aNPC->setPosition(pos1);
```

```
        aNPC->performAction(pos1);
        aNPC->lookTowards(action, pos1);

        return aNPC;
}
```

createNPCs method takes a NPCFactory as a parameter and creates a consistent NPC object.

In our first scenario, we created a Villain NPC with the VillainNPCFactory.

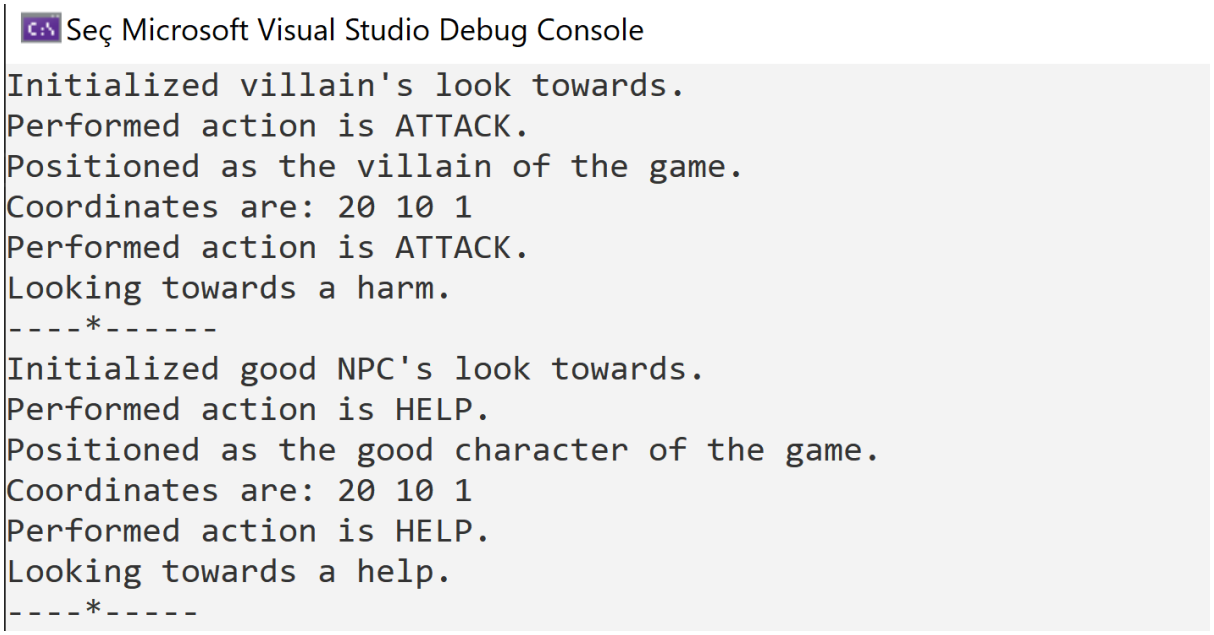
```
makeNPCs npc_game;
VillainNPCFactory factory;
npc_game.createNPCs(&factory);
```

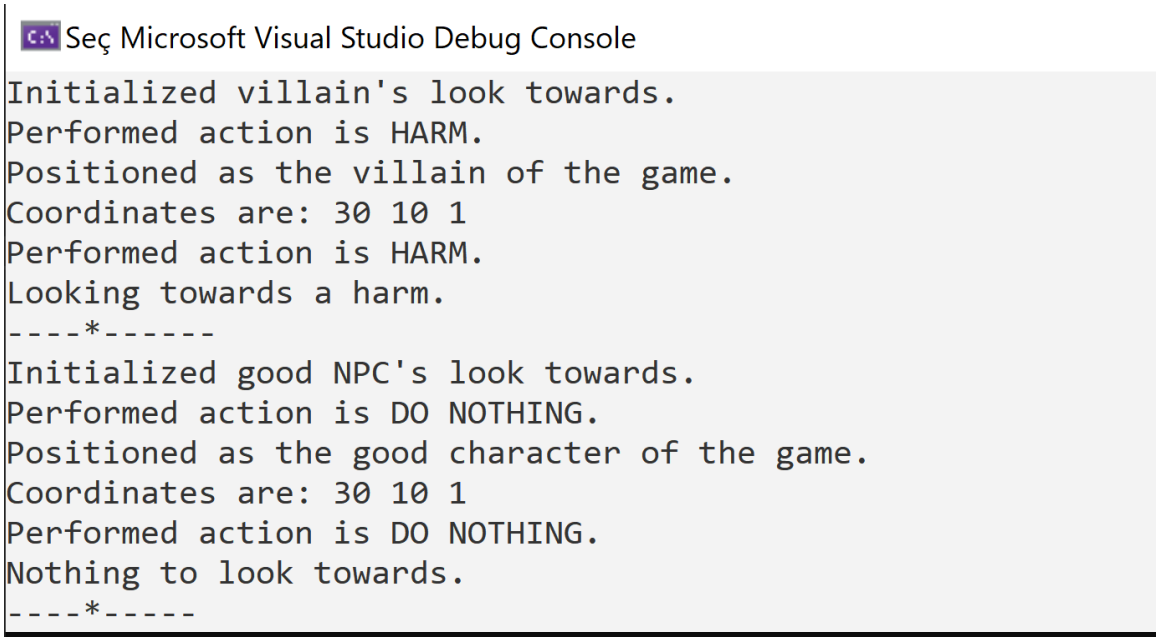
In our second scenario, we have created a Good NPC with the GoodNPCFactory with the intention of creating a consistent Good NPC.

```
GoodNPCFactory factory2;
npc_game.createNPCs(&factory2);
```

Test Results

Demo codes were shown in the test scenario. Below, you can find the test results of these demos.





Another example is done to show the different actions of NPCs. In that case, we have again created 2 different NPCs and changed their positions, therefore their actions are also changed. However, as we can see from the above Good NPCs do not harm our characters while Villains always harm or attack our characters. Good NPCs do nothing while Villains are trying to harm our characters.

By implementing Abstract Factory pattern, we are able to create different types of NPCs and define their attributes. We can say that we have satisfied the flexibility requirement needed for this problem.

Problem Definition

Goals and actions that are represented as a language should be interpreted by the Goal Oriented Action Planning (GOAP) controller.

We already know that actions and goals are a grammatical language that is being stored in Goal Action Database. The interpretations are not clearly stated in the report that was given to us, therefore, we should interpret the goals and actions with an interpreter. We can represent statements in the language as abstract syntax trees with the usage of Interpreter pattern.

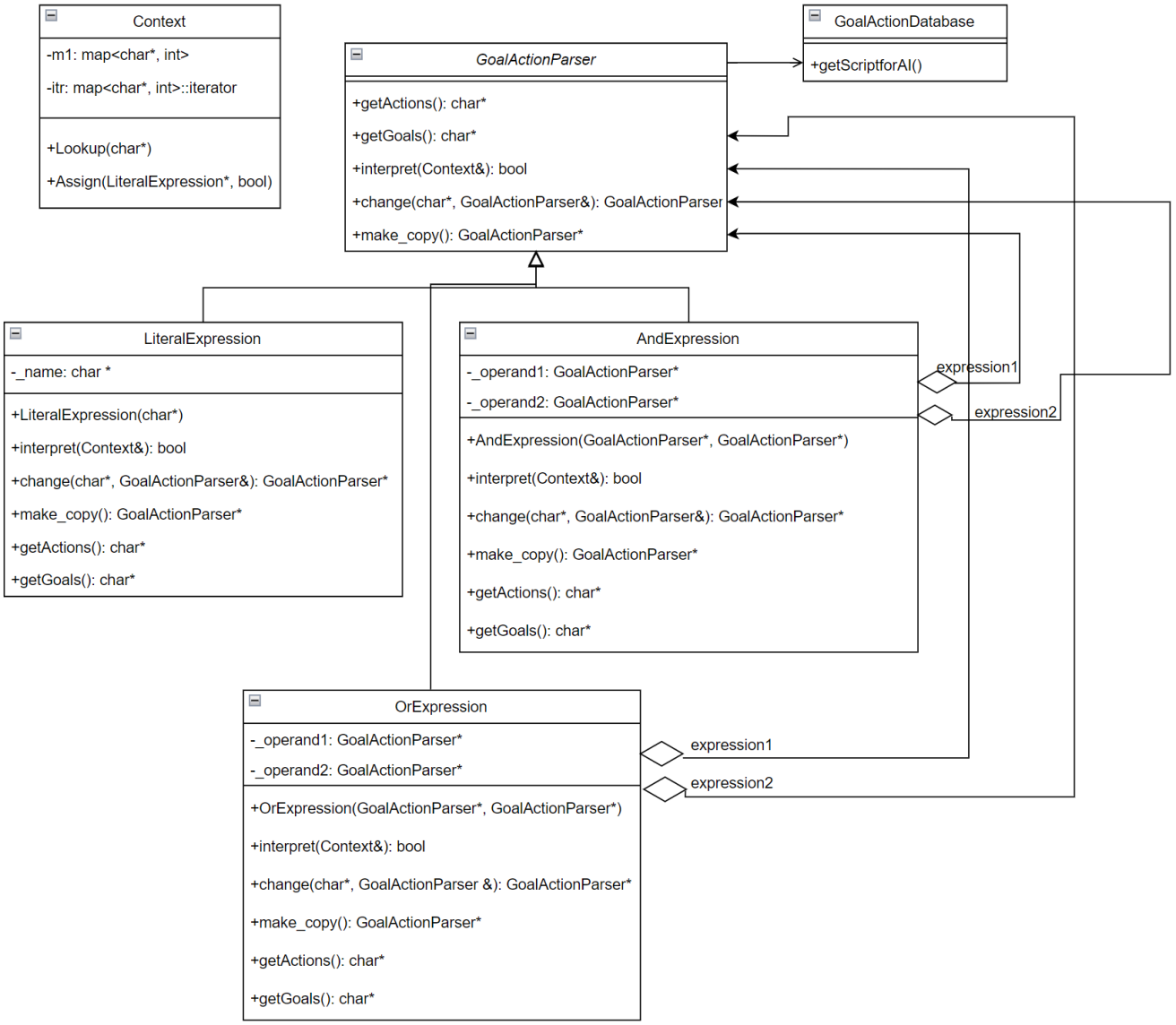
The Interpreter pattern uses a class to represent each grammar rule. This pattern also describes how to define a grammar for languages, represent sentences in the language and interpret these sentences.

Static Class Structure

The diagrams that are created aims to check the accuracy and consistency of our solution with the usage of the Interpreter design pattern. C++ programming language is chosen as the implementer of this flexibility requirement.

In the diagram below, we can see that GoalActionParser is made flexible with the Interpreter pattern. We defined two operations for Goal and Action parsing. First one is for interpreting the languages which gave actions boolean values. We also defined a method called change which produces a new expression. With the change method, we are able to use the Interpreter pattern multiple times.

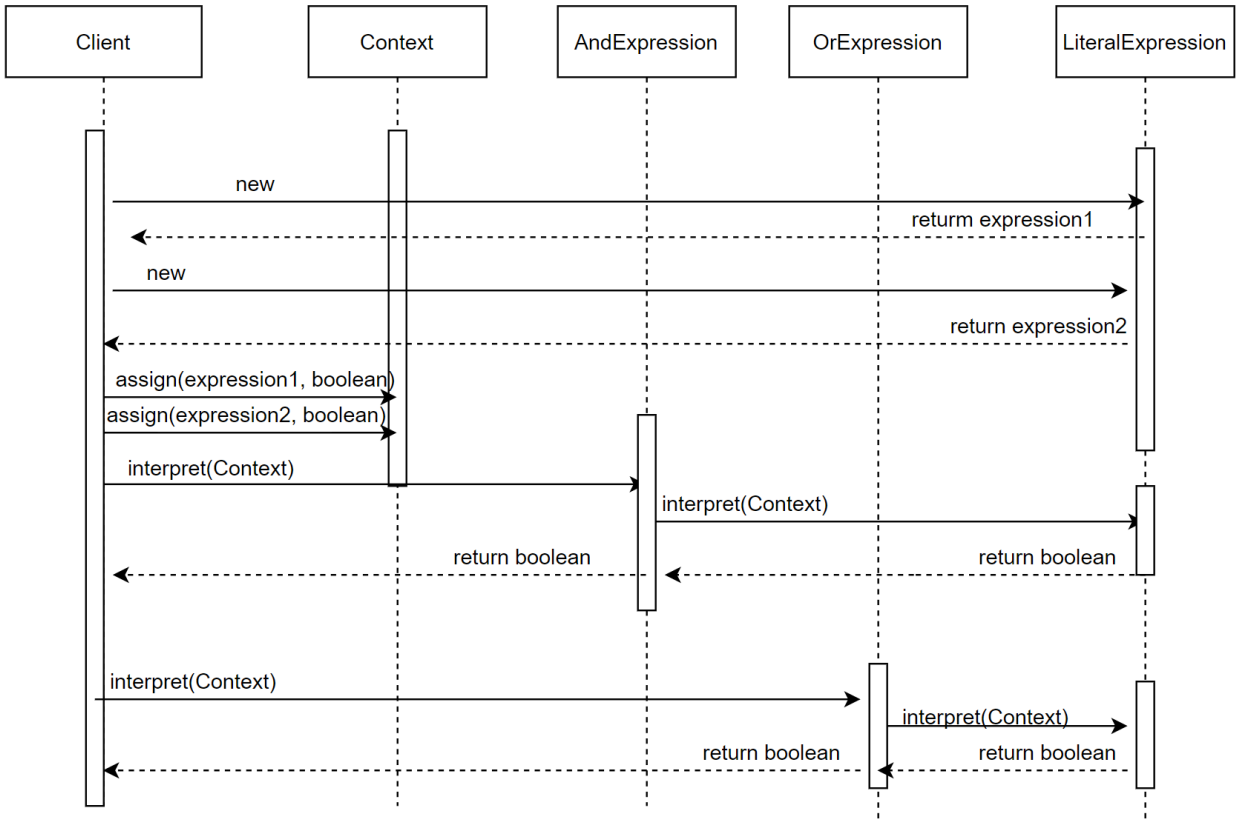
We also created Literal Expression, And Expression and Or Expression classes to represent grammar rules. As we already know, Literal Expressions are terminal symbols in the grammar. “And”, “Or” expressions are non terminal expressions which implement an interpret operation for non terminal symbols in the grammar.



The static class diagram contains the signatures of the classes. It is also shown that Goal Action Parser is connected to the Goal Action Database.

Dynamic Structure

Below, Interpreter pattern's dynamic sequence diagram is shown. In this diagram, it is clearly seen that first we are assigning our Context values, later we proceed on interpreting our contexts and evaluating the language.



Implementation of the Design Pattern

In this part, we are going to show the implementation of the flexibility requirement that is being discussed. This section aims to verify the application of Interpreter pattern. While implementing the design pattern, C++ programming language is chosen as the main implementer of the pattern. In this section, you can find the structure of the code, test cases and test results.

Goals and actions that are represented as a language should be interpreted by the Goal Oriented Action Planning (GOAP) controller.

In this specific problem, we are in need of manipulating and evaluating Actions and Goals expressions. In order to interpret these expressions we should first define the grammar of the implementation.

```
GoalActionParser ::= LiteralExpression | AndExpression | OrExpression
AndExpression ::= GoalActionParser 'and' GoalActionParser
```

```
OrExpression ::= GoalActionParser 'or' GoalActionParser
LiteralExpression := 'MOVE' | 'DO_NOTHING' | 'ATTACK' | ... | 'STOP'
```

For this specific problem, Goal Action Parser is implemented as an interface (pure virtual class) for evaluating expressions. Literal Expression, And Expression and Or Expression are subclasses of the Goal Action Parser. Literal, And, Or expression classes are inherited from the Goal Action Parser.

```
class GoalActionParser {
public:
	GoalActionParser() {

	}
	virtual bool interpret(Context& aContext) = 0;
	virtual GoalActionParser* change(const char* name,
GoalActionParser& expr) = 0;
	virtual GoalActionParser* make_copy() const = 0;
	virtual char* getActions() = 0;
	virtual char* getGoals() = 0;
};

class LiteralExpression : public GoalActionParser {
public:
	LiteralExpression() {

	}
	LiteralExpression(const char* name);

	virtual bool interpret(Context& aContext);
	virtual GoalActionParser* change(const char* name,
GoalActionParser& expr);
	virtual GoalActionParser* make_copy() const;
	virtual char* getActions();
	virtual char* getGoals();
private:
	char* _name;
};

class AndExpression : public GoalActionParser {
public:
	AndExpression() {

	}
```

```
      AndExpression(GoalActionParser* op1, GoalActionParser* op2);

      virtual bool interpret(Context& aContext);
      virtual GoalActionParser* change(const char* name,
GoalActionParser& exp);
      virtual GoalActionParser* make_copy() const;
      virtual char* getActions();
      virtual char* getGoals();
private:
      GoalActionParser* _operand1;
      GoalActionParser* _operand2;
};

class OrExpression : public GoalActionParser {
public:
      OrExpression() {

      }
      OrExpression(GoalActionParser* op1, GoalActionParser* op2);

      virtual bool interpret(Context& aContext);
      virtual GoalActionParser* change(const char* name,
GoalActionParser& exp);
      virtual GoalActionParser* make_copy() const;
      virtual char* getActions();
      virtual char* getGoals();
private:
      GoalActionParser* _operand1;
      GoalActionParser* _operand2;
};
```

The code shown above is representing the classes. Implementations are done in a separate file called “interpreter.cpp”. Before moving on with the implementation of methods, we should also show our Context class which provides a Look Up table for expressions.

```
class Context {
public:
      Context() {

      }
      bool Lookup(const char*) const;
      void Assign(LiteralExpression* exp, bool value);
```

```
private:
	map<char*, int> m1;
	map<char*, int>::iterator itr;
};
bool Context::Lookup(const char* name) const
{
	map<char*, int> m2(m1.begin(), m1.end());
	map<char*, int>::iterator itr;
	for (itr = m2.begin(); itr != m2.end(); itr++) {
		if (strcmp(itr->first, name) == 0) {
			return itr->second;
		}
	}
	return false;
}

void Context::Assign(LiteralExpression* exp, bool value) {
	char* name = exp->getActions();
	m1[name] = value;
}
```

We can now move on with the implementations of the And, Literal and Or expressions. And, Or classes have recursive calls for their operands.

```
LiteralExpression::LiteralExpression(const char* name){
	_name = _strdup(name);
}

bool LiteralExpression::interpret(Context& aContext) {
	return aContext.Lookup(_name);
}

GoalActionParser* LiteralExpression::make_copy() const {
	return new LiteralExpression(_name);
}

char* LiteralExpression::getActions()
{
	return _name;
}

char* LiteralExpression::getGoals()
{
```

```
	return _name;
}

GoalActionParser* LiteralExpression::change(const char* name,
GoalActionParser& expr) {

	if (strcmp(name, _name) == 0) {
		return expr.make_copy();
	}
	else {
		return new LiteralExpression(_name);
	}
}

/*
* Operations of AND.
*/
AndExpression::AndExpression(GoalActionParser* op1, GoalActionParser*
op2){
	_operand1 = op1;
	_operand2 = op2;
}

bool AndExpression::interpret(Context& aContext) {
	return _operand1->interpret(aContext) &&
_operand2->interpret(aContext);
}

GoalActionParser* AndExpression::make_copy() const {
	return new AndExpression(_operand1->make_copy(),
_operand2->make_copy());
}

char* AndExpression::getActions()
{
	return nullptr;
}

char* AndExpression::getGoals() {
	return nullptr;
}

GoalActionParser* AndExpression::change(const char* name,
GoalActionParser& exp) {
```

```
	return new AndExpression(_operand1->change(name, exp),
_operand2->change(name, exp));
}

/*
* Operations of OR.
*/

OrExpression::OrExpression(GoalActionParser* op1, GoalActionParser*
op2) {
	_operand1 = op1;
	_operand2 = op2;
}

bool OrExpression::interpret(Context& aContext) {
	return _operand1->interpret(aContext) ||
_operand2->interpret(aContext);
}

GoalActionParser* OrExpression::make_copy() const {
	return new OrExpression(_operand1->make_copy(),
_operand2->make_copy());
}

char* OrExpression::getActions() {
	return nullptr;
}

char* OrExpression::getGoals() {
	return nullptr;
}

GoalActionParser* OrExpression::change(const char* name,
GoalActionParser& exp) {
	return new OrExpression(_operand1->change(name, exp),
_operand2->change(name, exp));
}
```

Above codes are representing the operations and the evaluation of the expressions.

Test Scenarios

For testing the results, we have created simple languages and shown their accuracy. If needed, the implementation of other grammatical rules can be added to the “interpreter.h” file. For now, we have only AND, OR, Terminal values for the grammar. Therefore, we can create simple boolean expressions for interpreting actions and goals of the characters.

For the first test case, we have implemented a simple code which evaluates MOVE or RUN in the first case. We have assigned MOVE as false and RUN as true, therefore, it should be evaluated as 1 as expected. 1 will show the RUN action in that case.

```
GoalActionParser* exp;
Context context;
LiteralExpression* act1 = new LiteralExpression("MOVE");
LiteralExpression* act2 = new LiteralExpression("RUN");
exp = new AndExpression(act1,act2);
context.Assign(act1, false);
context.Assign(act2, true);
bool result = exp->interpret(context);
```

Another test scenario will represent the AND case for the goals. In this scenario, STOP and ATTACK goals are evaluated. For this case, STOP will be 0, and ATTACK will have the value of 1. When we interpret this expression, we would get the STOP goal.

```
GoalActionParser* exp;
Context context;
LiteralExpression* goal1 = new LiteralExpression("STOP");
LiteralExpression* goal2 = new LiteralExpression("ATTACK");
exp = new OrExpression(goal1,goal2);
context.Assign(goal1, false);
context.Assign(goal2, true);
bool result = exp->interpret(context);
```

For the third scenario, we have designed a more complex test case for the evaluation of the expression.

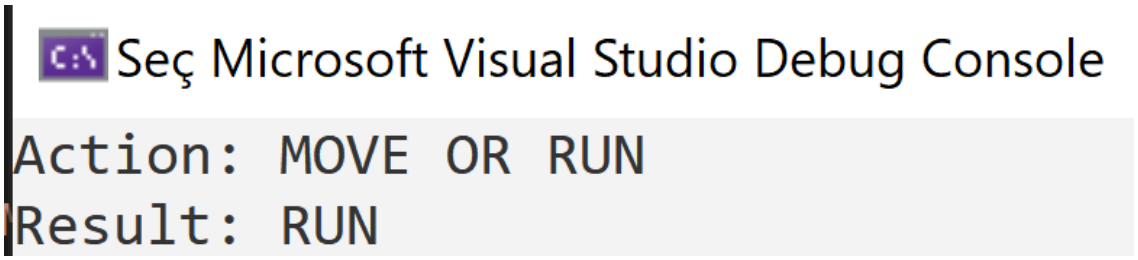
```
LiteralExpression* act1 = new LiteralExpression("MOVE");
LiteralExpression* act2 = new LiteralExpression("RUN");
```

```
        LiteralExpression* act3 = new LiteralExpression("ATTACK");
        LiteralExpression* act4 = new LiteralExpression("DO_NOTHING");

        exp = new AndExpression(new OrExpression(act1, act3), new
OrExpression(act4, act2));

        context.Assign(act1, false);
        context.Assign(act3, true);
        context.Assign(act2, true);
        context.Assign(act4, false);

        bool result = exp->interpret(context);
```

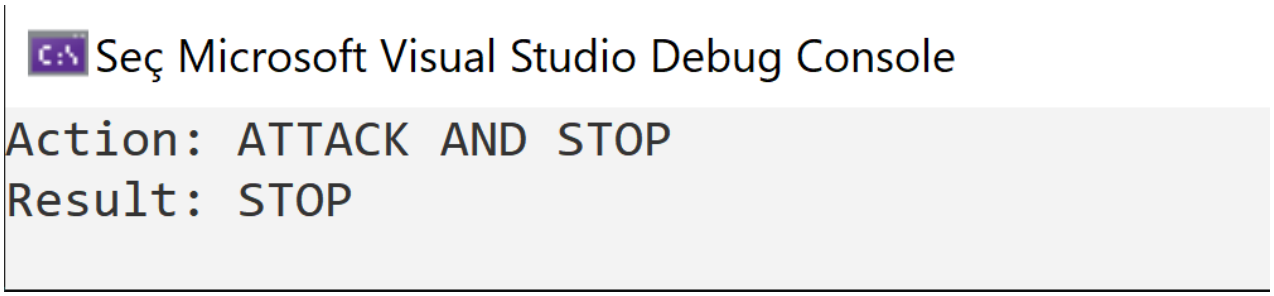
These were the tree test cases for the Interpreter pattern. Now we can move on with the test results in the next chapter.

Test Results

For the first test case, we have MOVE or RUN as the test case. MOVE given as false, RUN given as true. Result of the first test case is shown below:

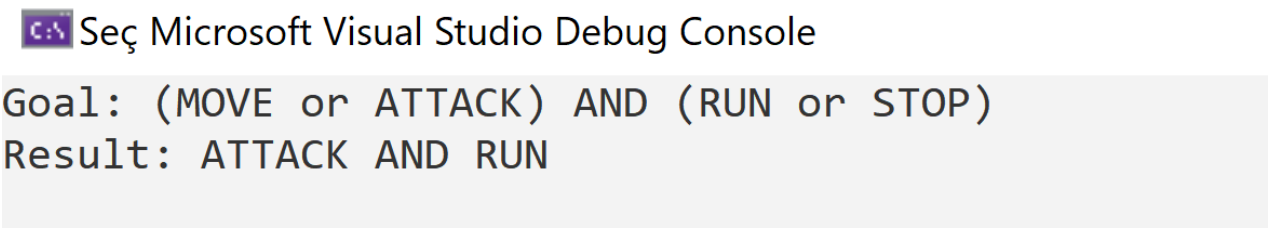


In the second test case, we have tested an AND case. In that case, STOP is given as false, ATTACK given as true. Result should be expected as STOP in that case.



In the third test case, it was given as (MOVE or ATTACK) and (RUN or

DO_NOTHING). Move, do nothing is given as false in that scenario. Attack and run is given as true. Therefore, the result should be ATTACK AND RUN.



As we can see from the test cases, the Interpreter pattern works correctly with the given scenarios. Adding new grammar rules are made flexible. Extending and changing the grammar is also made uncomplicated with the help of the abstract syntax tree structure. Moreover, the Interpreter pattern also makes it easy to evaluate expressions as seen above.

To summarize, Goal Action Parser is made flexible for interpreting goals and actions. Ultimately, we have satisfied the flexibility requirement needed for this problem.

Problem Definition

Goals Oriented Action Planning (GOAP) controller should update goals and multiple actions when changes are made in the AI module.

In order to make the game more consistent and realistic, we should update Goals and Action Manager subclasses inside of the GOAP controller. We already know that the GOAP controller receives information from the AI module of the game, therefore, we should update the manager classes inside of the controller.

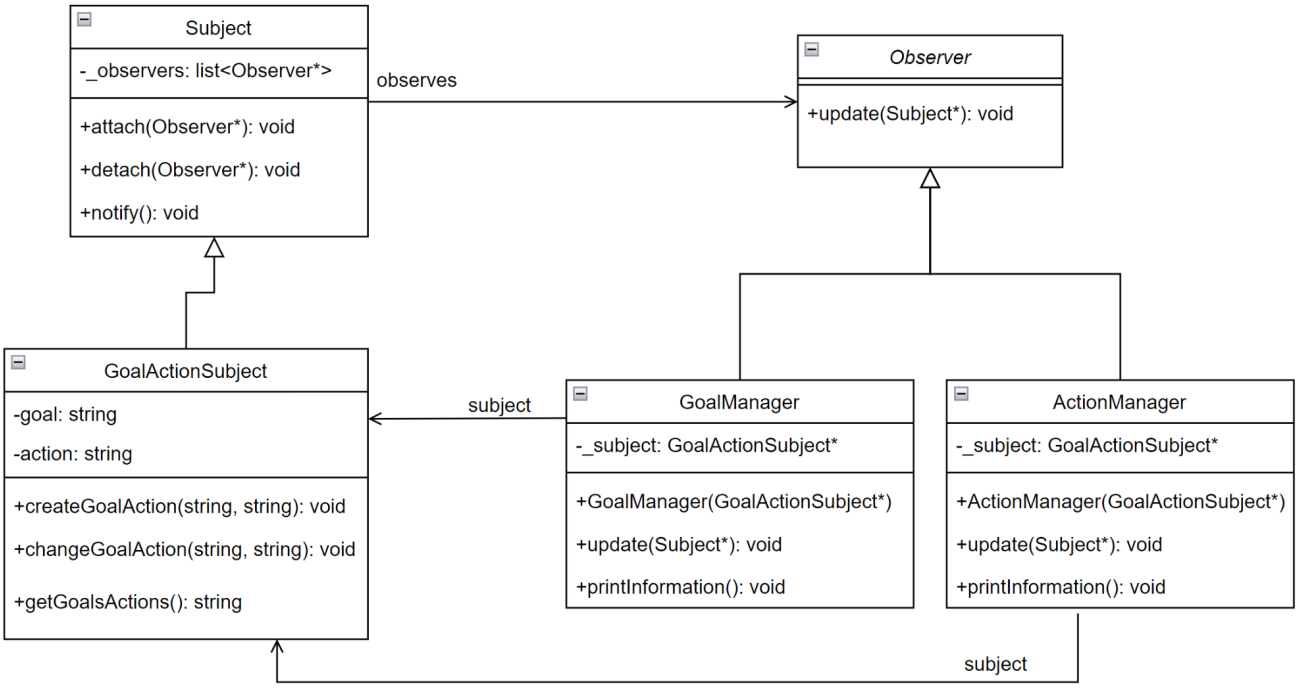
Updating the Manager classes can be done with the help of the Observer pattern. This pattern is a structural design pattern that defines a one to many relationships between objects. In order to make the GOAP controller more flexible, we should notify the Goal and Action managers in the system.

Static Class Structure

Below, the static class diagram of this problem is shown. In this diagram, it is clearly seen that Goal and Action Managers are the concrete observers. Concrete subject is defined as the Goal Action Subject.

In this diagram, the Subject knows its Observer objects. It helps us to add or remove Observer objects. Observer class defines an updating interface for objects that should be notified when the changes are made. Goal and Action Subject stores the actions and goals that the manager classes have.

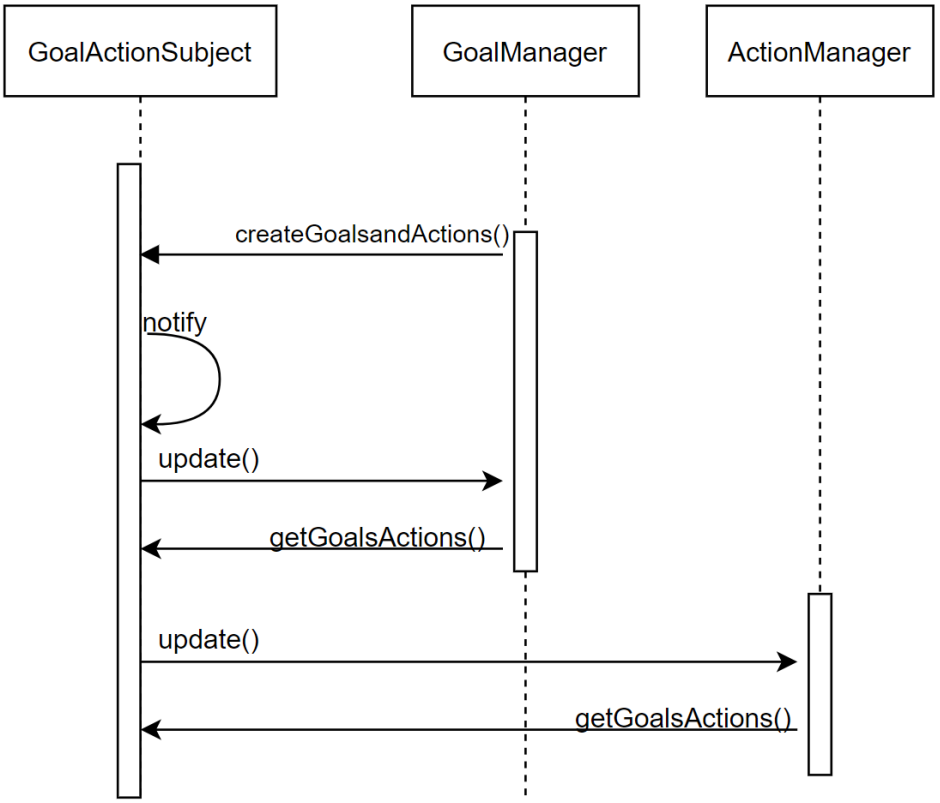
Goal Manager and Action Manager classes maintain a reference to GoalActionSubject. These classes implement the updating method of their interface to keep their states consistent.



Dynamic Class Diagram

The diagram shown below is the sequence diagram of this problem. GoalActionSubject notifies its observers when a change is made. By doing so, we are able to keep the observers' states consistent. Notify is always called by the GoalActionSubject.

After being informed of changes, Goal Manager and Action Managers can query their subject for reaching goals and actions.



Implementation of the Design Pattern

In this part, we are going to show the implementation of the flexibility requirement that is being discussed. This section aims to verify the application of the Observer pattern. While implementing the design pattern, C++ programming language is chosen as the main implementer of the pattern. In this section, you can find the structure of the code, test cases and test results.

Goals Oriented Action Planning (GOAP) controller should update goals and multiple actions when changes are made in the AI module.

For this specific problem, we should first define the Observer class. This class should contain an update method for its concrete observer subclasses.

```
class Observer {
public:
    Observer() {
```

```
    }
    virtual void update(Subject* changedSubject) = 0;
};
```

We can see our Observer class. This class provides an interface for the concrete observer classes which are shown below.

```
class GoalManager : public Observer {
public:
    GoalManager(GoalActionSubject *sub);
    ~GoalManager();
    virtual void update(Subject* changedSubject) override;
    void printInformation();
private:
    GoalActionSubject* _subject;
};

class ActionManager : public Observer {
public:
    ActionManager(GoalActionSubject *sub);
    ~ActionManager();
    virtual void update(Subject* changedSubject) override;
    void printInformation();
private:
    GoalActionSubject* _subject;
};
```

As we can see from the code, ActionManager and GoalManager maintain a reference to GoalActionSubject. We have shown the class structure of Manager objects, now we can add their implementations that are stored in the "observer.cpp" file.

```
GoalManager::GoalManager(GoalActionSubject* sub) {
    _subject = sub;
    _subject->attach(this);
}

GoalManager::~GoalManager() {
```

```
    _subject->detach(this);
}
ActionManager::ActionManager(GoalActionSubject* sub) {
    _subject = sub;
    _subject->attach(this);
}
ActionManager::~ActionManager() {
    _subject->detach(this);
}
```

As it can be seen from the code, whenever we create an Observer, we should first attach the observer to the subject. It is shown that the attachment is done when we first create the Observer object. Later, we can detach it with the deconstructor method that is defined. We can now clearly show the other methods of the Observer classes.

```
void GoalManager::printInformation() {
    std::cout << "A new action and goal has been created for Goal
Manager." << std::endl;
    std::string str = _subject->getGoalsActions();
    std::cout << str << std::endl;
}

void GoalManager::update(Subject *sub) {
    sub = _subject;
    printInformation();
}

void ActionManager::update(Subject* sub) {
    sub = _subject;
    printInformation();
}

void ActionManager::printInformation() {
    std::cout << "A new action and goal has been created for
Action Manager." << std::endl;
    std::string str = _subject->getGoalsActions();
    std::cout << str << std::endl;
}
```

Whenever we get to update the Obsevers, we are printing out their current information in order to see whether they are consistent or not.

For now, we have defined our Observers. We should also show our Subject classes which are responsible for notifying the Observers.

```
class Subject {
public:
    Subject() {

    }
    virtual void attach(Observer* obs);
    virtual void detach(Observer* obs);
    virtual void notify();
private:
    list<Observer*> _observers;
};
```

In this Subject class, we are keeping a list of observers. When an Observer is created, we first add the Observer with the attach method and later, we can remove the Observer with the detach method. Notifying the Observers is done inside of the notify method shown below.

```
void Subject::attach(Observer* obs) {
    _observers.push_back(obs);
}
void Subject::detach(Observer* obs) {
    _observers.remove(obs);
}

void Subject::notify() {
    list<Observer*>::iterator itr;
    Observer* ptr;
    for (itr = _observers.begin(); itr != _observers.end(); itr++)
{
       ptr = *itr;
       ptr->update(this);
    }
}
```

As it can be seen from above, we keep a list of observers and whenever a change is made, we notify our observer classes.

We can now move on with the concrete subject class which is defined as GoalActionSubject. This class is responsible for maintaining consistency between Observers. We are keeping the goals and actions as a string, so that we can update or create them according to the needs of clients.

```
class GoalActionSubject : public Subject {
public:
    GoalActionSubject() {

    }
    void createGoalAction(std::string goal, std::string action);
    void changeGoalAction(std::string goal, std::string action);
    std::string getGoalsActions();
private:
    std::string goal;
    std::string action;
};
```

The methods are defined as creating, changing and getting the goals and actions. This method notifies its observers when a new action or goal is created. It also notifies its observers when new changes are made. Implementation of these methods are shown below.

```
void GoalActionSubject::createGoalAction(std::string goal,
std::string action) {
    this->goal = goal;
    this->action = action;
    notify();
}

void GoalActionSubject::changeGoalAction(std::string goal,
std::string action) {
    this->goal = goal;
```

```
    this->action = action;
    notify();
    std::cout << "Goal and actions are now changed." << std::endl;
}

std::string GoalActionSubject::getGoalsActions() {
    std::string st = "Goal is: " + goal + "\nAction is: " +
action;
    return st;
}
```

As we can see from above, the implementation of the GOAPs Observer pattern is done. We have defined the creation of the goals and actions and how we retrieve them with the getGoalsActions() function. The following sections will implement the test scenarios and test results of the Observer pattern.

Test Scenario

As it can be seen from above, we have implemented the Observer pattern. We should test our patterns' accuracy by implementing test scenarios. The test scenarios that are being implemented are written in the C++ language.

First test scenario includes creating new actions and goals. When a new action and goal is created inside of our system, we should notify the Observers and see if they are being consistent or not. This test case is implemented as seen below.

```
GoalActionSubject* subject = new GoalActionSubject();
GoalManager* goalObserver = new GoalManager(subject);
ActionManager* actionObserver = new ActionManager(subject);
subject->createGoalAction("ESCAPE", "RUN");
```

As it can be seen from the test, we first created a Subject defined as GoalActionSubject. Later, we move on with the definition of Goal Observer and Action Observer. These observers are now attached to our subject.

Later we have created a new goal and action. As it can be seen from above, we now have the goal as “escape”, action as “run.”

For another test case, we should now update the actions and our goals. When we

update the action and goals, our subject should notify its observers.

```
subject->changeGoalAction("HIDE", "MOVE");
```

As it can be seen, we have now changed the goal and action. If wanted, we can move on with creating more actions and goals.

As the third test case we can delete or add observers to the system and create more actions and goals for different Goal Managers and Action Managers.

```
delete goalObserver;
subject->createGoalAction("DO NOTHING", "DO NOTHING");
```

Now that we have deleted our Goal Observer, we can now create multiple observers for goals.

```
GoalManager* g2 = new GoalManager(subject);
GoalManager* g3 = new GoalManager(subject);
GoalManager* g4 = new GoalManager(subject);

subject->createGoalAction("ATTACK", "MOVE");
```

We have created different GoalManager observer classes and assign them to the subject. All the test scenarios are done for this requirement, we can move on with the Test Results.

Test Results

In above, we have shown the test cases for the Observer pattern. Now, we can show the results of these test cases.

Test case number one was about creating new actions and goals for the action and goal manager. In the result below, it is shown that the new actions and goals have been created and the Subject has notified its observers.

Seç Microsoft Visual Studio Debug Console

```
Goal and action set has been changed for the Goal Manager.
Goal is: ESCAPE
Action is: RUN
Goal and action set has been changed for the Action Manager.
Goal is: ESCAPE
Action is: RUN
```

Action Managers and Goal Managers action and goal set has been updated accordingly. Now we can change the goal and action for this set.

```
Goal and action set has been changed for the Goal Manager.
Goal is: HIDE
Action is: MOVE
Goal and action set has been changed for the Action Manager.
Goal is: HIDE
Action is: MOVE
Goal and actions are now changed.
```

The goal and action set was Escape and Run. However we changed it to Hide and Move as it can be seen from the test result.

The third test is about deleting a manager. We have deleted the Goal Manager for this test. In the result below, it can be seen that the Action Manager has been updated but the Goal Manager is no longer a part of this system. Goal Manager has been detached from the system.

```
Goal and action set has been changed for the Goal Manager.
Goal is: ESCAPE
Action is: RUN
Goal and action set has been changed for the Action Manager.
Goal is: ESCAPE
Action is: RUN
Goal and action set has been changed for the Goal Manager.
Goal is: HIDE
Action is: MOVE
Goal and action set has been changed for the Action Manager.
Goal is: HIDE
Action is: MOVE
Goal and actions are now changed.
Goal and action set has been changed for the Action Manager.
Goal is: DO NOTHING
Action is: DO NOTHING
```

It can be seen that the Action Manager has changed as Do Nothing and Do Nothing as the goal and action pair. However, in this context, the Goal Manager is absent and it is no longer being updated.

For the fourth case, we have created multiple Goal Managers and assigned new actions and goals. It is shown in the result below.

```
Goal and action set has been changed for the Action Manager.
Goal is: ATTACK
Action is: MOVE
Goal and action set has been changed for the Goal Manager.
Goal is: ATTACK
Action is: MOVE
Goal and action set has been changed for the Goal Manager.
Goal is: ATTACK
Action is: MOVE
Goal and action set has been changed for the Goal Manager.
Goal is: ATTACK
Action is: MOVE
```

Problem Definition

Coordination between Manager objects should be done effectively.

Coordination in Manager classes has an important role in our gameplay since the Manager classes coordinate the characters and teams important tasks and information. However, these classes have dependency and coordination issues that we should solve effectively by implementing a design pattern. Managers' objects , behaviors and coordinations should be separated from each other in order to have a flexible and consistent gameplay.

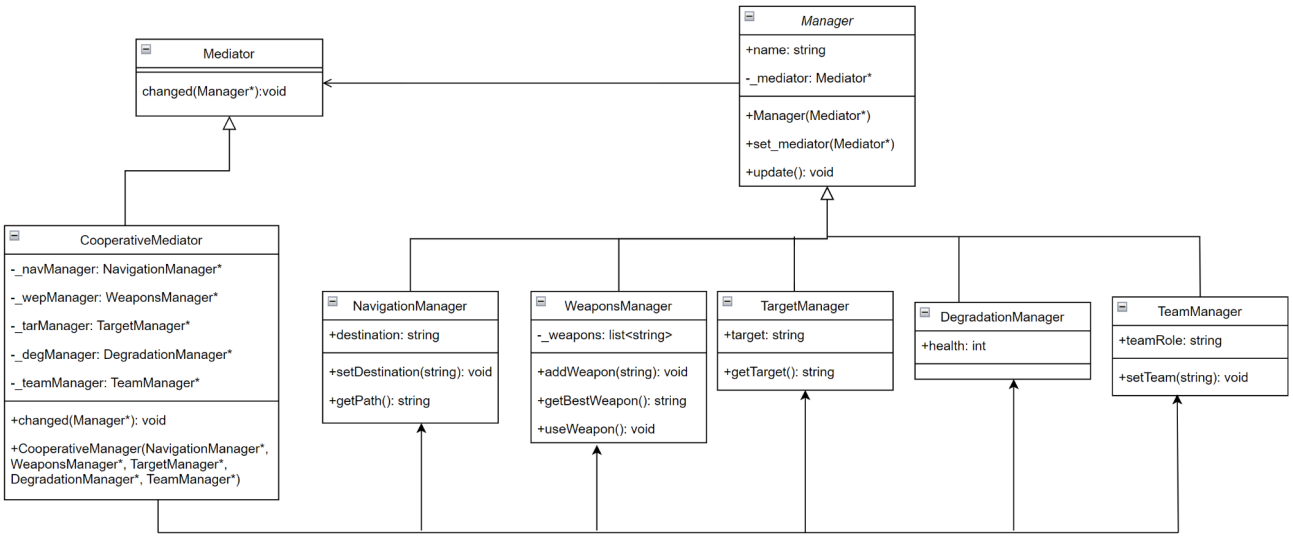
Static Class Diagram

For this problem, we have designed the class diagram in order to solve the cooperation issues in the Manager class. The Manager classes are designed to manage characters or team issues in the game.

They have a huge importance in our game since without the managers our game would not work consistently and effectively. Concrete Colleagues are defined as Navigation Manager, Weapons Manager, Target Manager, Degradation Manager

and Team Manager. These colleagues are implemented in the report that was given to us, however they need to be more flexible in order to work effectively and correctly for a consistent gameplay. These colleagues can exchange information and may be dependent on one another. These dependency and cooperation issues must be solved with the Mediator design pattern.

Concrete Mediator object is defined as Cooperative Mediator and it has instances of all the colleagues. With these instances, we can trigger or coordinate the colleagues and have different kinds of operations. We can also resolve issues around dependencies by implementing extra methods within the Cooperative Mediator class.

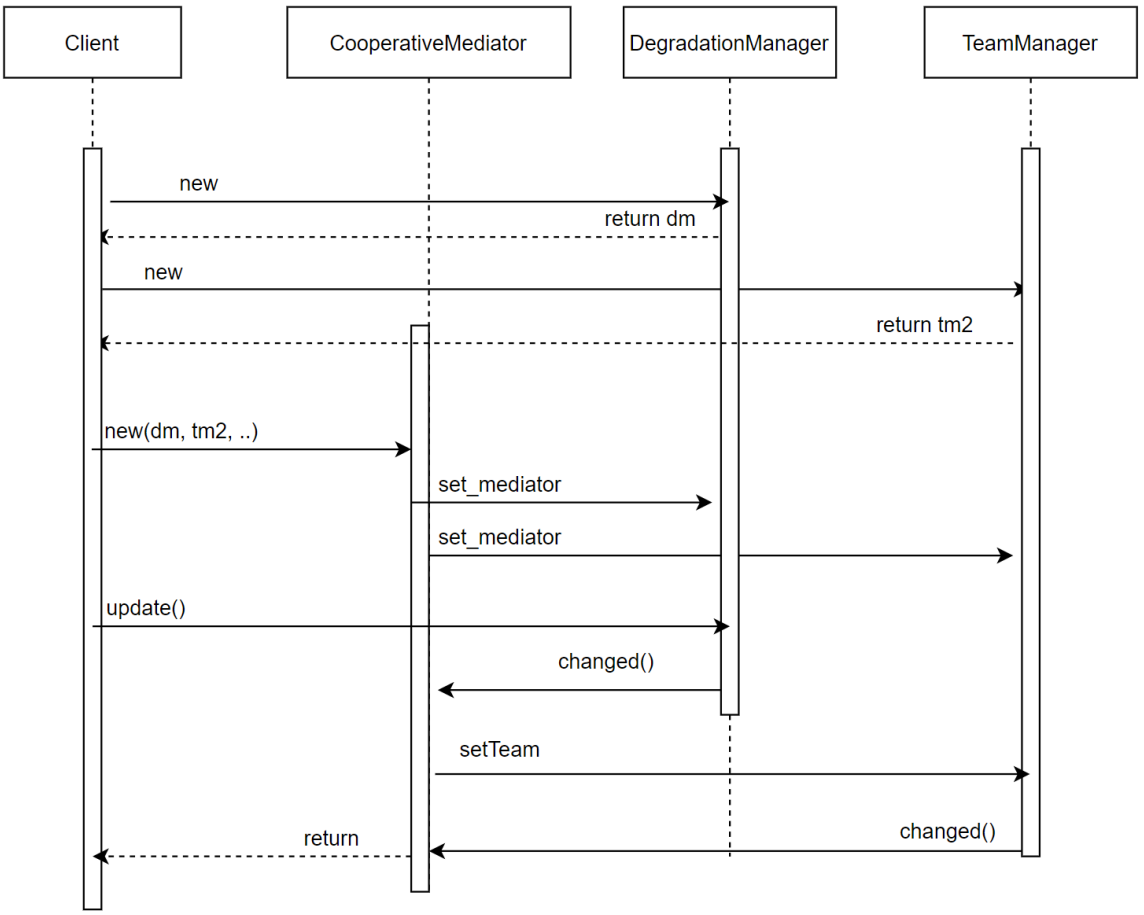


Dynamic Class Diagram

Below, we have shown the Mediator's dynamic class diagram for providing flexibility. We created this diagram to show how objects are interacting with each other and how they cooperate.

In this diagram, we have shown an example usage of the Mediator object. This usage resembles the fifth test case which is shown in the Test Results section. In this example, the client updates the Degradation Manager and the Mediator sets Teams role according to the information coming from the Degradation Manager. When the Team Role changes, Team Manager calls the change method of the Mediator. Then we return back to the client.

Below, you can examine the dynamic structure of this sample.



Implementation of the Design Pattern

In this part, we are going to show the implementation of the flexibility requirement that is being discussed. This section aims to verify the application of the Mediator pattern. While implementing the design pattern, C++ programming language is chosen as the main implementer of the pattern. In this section, you can find the structure of the code, test cases and test results. While implementing this design pattern, we have taken notes from the usage of the Mediator pattern and consider the flexibility requirements of this pattern.

Coordination between Manager objects should be done effectively.

This problem is crucial for our gameplay since we want to have a smooth and reliable game in order to be successful in the game market. Manager objects

should be reliable and dependent on one another. They should cooperate rationally.

For this problem, we should first define the Mediator interface that is a pure virtual class and has one method called as changed.

```
class Mediator { //Mediator
public:
    virtual void changed(Manager* mng) = 0;
};
```

With this in hand, we can now implement the Concrete Mediator object named Cooperative Mediator. This mediator class has the instances of colleagues.

```
class CooperativeMediator: public Mediator { //ConcreteMediator
public:

    CooperativeMediator() {

    }
    CooperativeMediator(NavigationManager* navManager,
                        WeaponsManager* wepManager,
                        TargetManager* tarManager,
                        DegradationManager* degManager,
                        TeamManager* teamManager);
    virtual void changed(Manager* mng) override;

private:
    NavigationManager* _navManager;
    WeaponsManager* _wepManager;
    TargetManager* _tarManager;
    DegradationManager* _degManager;
    TeamManager* _teamManager;
};
```

With the changed method in hand, we can now cooperate with different Manager objects when needed. We implemented actions of the concrete mediator inside of the mediator.cpp file.

```
CooperativeMediator::CooperativeMediator(NavigationManager*
```

```
navManager, WeaponsManager* wepManager, TargetManager* tarManager,
DegradationManager* degManager, TeamManager* teamManager) {
    //setting managers.
    _navManager = navManager;
    _wepManager = wepManager;
    _degManager = degManager;
    _tarManager = tarManager;
    _teamManager = teamManager;

    navManager->set_mediator(this);
    wepManager->set_mediator(this);
    tarManager->set_mediator(this);
    degManager->set_mediator(this);
    teamManager->set_mediator(this);
}

void CooperativeMediator::changed(Manager* mng) {

    std::cout << "Mediator is notified about the changes in " +
mng->name << " Manager." << std::endl;

    if (mng->name == "Navigation") {
       _navManager->getPath();
    }
    if (mng->name == "Weapons") {
       std::cout << "Best weapon chosen as: " <<
_wepManager->getBestWeapon() << std::endl;
    }
    if (mng->name == "Target") {
       _tarManager->getTarget();
    }
    if (mng->name == "Degradation") {
       std::cout << "Health has been determined as " <<
_degManager->health << std::endl;
       std::cout << "Mediator notifies the Team Manager. " <<
std::endl;
       _teamManager->setTeam("Help for injured");
       std::cout << "Team role determined as: " <<
_teamManager->teamRole << std::endl;
    }
```

```
    if (mng->name == "Team") {
      std::cout << "Team role determined as: " <<
_teamManager->teamRole << std::endl;
    }
}
```

As it can be seen from the code, we have defined the Mediators of each object. By doing so Mediator introduced itself to the colleagues. We showed that Colleagues know their mediator.

We can now define the Manager classes for this problem. Manager superclass have an update method that notifies the Mediator that something has been done inside of the Manager classes.

```
class Manager { //Collegue
public:
    string name;
    Manager() {

    }
    Manager(Mediator* med);
    void set_mediator(Mediator* med);
    virtual void update(); //it should call the notify object.
private:
    Mediator* _mediator;
};
```

This class's actions are also defined in the mediator.cpp file. We set the mediator in the constructor.

```
Manager::Manager(Mediator* med) {
    this->set_mediator(med);
}

void Manager::set_mediator(Mediator* med) {
    this->_mediator = med;
}

void Manager::update() {
```

```
    _mediator->changed(this);
}
```

Now we can define the colleagues that have been given to us in the report. Navigation Manager is responsible for navigating the characters as it can be seen from the code. Static diagram also implies that.

```
class NavigationManager : public Manager {
public:
    NavigationManager() {
       this->name = "Navigation";
    }
    string destination;
    void setDestination(string str);
    string getPath();
};
```

Navigation class is defined above. Methods are implemented for this class.

```
 void NavigationManager::setDestination(string str) {
     this->destination = str;
     std::cout << "Destination has been set as " << destination <<
 std::endl;
     this->update();
 }

 string NavigationManager::getPath() {
     string str = "Path has been determined for " + destination;
     return str;
 }
```

Now we can implement another colleague called as Weapons Manager with its implementation on methods.

```
class WeaponsManager : public Manager {
public:
    WeaponsManager() {
       this->name = "Weapons";
    }
```

```
    void addWeapon(string weapon);
    string getBestWeapon();
    void useWeapon();
private:
    list<string> _weapons;
};
void WeaponsManager::addWeapon(string weapon) {
    this->_weapons.push_back(weapon);
    std::cout << "Weapon has been added to the system." <<
std::endl;
    this->update();
}

string WeaponsManager::getBestWeapon() { //returns the first
element of the weapons array.
    return this->_weapons.front();
}

void WeaponsManager::useWeapon() {
    std::cout << "Selected weapon has been used." << std::endl;
    this->update();
}
```

Now we can implement the Target Manager and we can show this manager's methods.

```
class TargetManager : public Manager {
public:
    TargetManager() {
       this->name = "Target";
    }
    string target;
    string getTarget();
};
string TargetManager::getTarget() {
    return target;
}
```

Now, we are responsible for implementing the Degradation Manager. It has only one attribute for now.

```
class DegradationManager : public Manager {
public:
    DegradationManager() {
       this->name = "Degredation";
    }
    int health;
};
```

Now, we can implement our final manager class which is Team Manager. This Manager class manages teams and team roles.

```
class TeamManager : public Manager {
public:
    TeamManager() {
       this->name = "Team";
    }
    string teamRole;
    void setTeam(string team);
};

void TeamManager::setTeam(string team) {
    this->teamRole = team;
    std::cout << "Team has been set as " << team << " ." <<
std::endl;
    this->update();
}
```

All of the implements are done for this problem. We have shown the Mediator pattern for solving this requirement. Now, we can move on with the test scenarios for this requirement.

Test Scenarios

For this requirement, we should test the different cases of the Mediator pattern for extending the interface. We should test our patterns' accuracy by implementing test scenarios. The test scenarios that are being implemented are written in the C++ language. These scenarios should provide flexibility for our requirements.

For the first case, we have created our colleagues who are managers in our system. We also defined the concrete mediator for these managers.

```
NavigationManager* nm = new NavigationManager();
WeaponsManager* wm = new WeaponsManager();
TargetManager* tm = new TargetManager();
DegradationManager* dm = new DegradationManager();
TeamManager* tm2 = new TeamManager();
CooperativeMediator* mediator = new CooperativeMediator(nm, wm, tm,
dm, tm2);
```

Mediator introduced itself to its colleagues as we can see from the code above. We can now cooperate with these manager classes with the help of the mediator.

We have selected to trigger the Navigation manager class for the first test case.

```
std::cout << "Triggering Navigation Managers operation.\n" <<
std::endl;
nm->setDestination("Ankara");
std::cout << "\n";
```

For the second test case, we have triggered the Weapons Manager.

```
std::cout << "Triggering Weapon Managers operation.\n" <<
std::endl;
wm->addWeapon("Gun");
wm->addWeapon("Fork");
```

For the third test case, we have triggered Target Manager.

```
std::cout << "Triggering Target Managers operation.\n" <<
std::endl;
tm->target = "Target1";
tm->getTarget();
```

For the fourth case we triggered the TeamManager.

```
std::cout << "Triggering Team Managers operation.\n" << std::endl;
tm2->setTeam("Helper");
```

For the fifth case, we have triggered the Degradation Manager.

```
std::cout << "Triggering the Degradation Manager.\n" << std::endl;
dm->health = 10;
```

Test cases have been defined as above. We can now test these cases for a better understanding of the Mediator pattern.

Test Results

In the implementation section of this report, we have implemented the Command design pattern with the C++ programming language. Now, we can move on with the test results in order to show the accuracy and flexibility for this requirement.

First test scenario have been tested. As we can see from the result, mediator has been notified for the changes that have been made in the Manager class.

```
Triggering Navigation Managers operation.

Destination has been set as Ankara
Mediator is notified about the changes in Navigation Manager.
```

For the second test case we have triggered the Weapons Manager. This time, as we can see from the implementation of the CooperativeMediator, we have updated the mediator and the mediator called the getBestWeapon() function for the corresponding update.

```
Triggering Weapon Managers operation.

Weapon has been added to the system.
Mediator is notified about the changes in Weapons Manager.
Best weapon chosen as: Gun
Weapon has been added to the system.
Mediator is notified about the changes in Weapons Manager.
Best weapon chosen as: Gun
```

For the third scenario, we have triggered the Target Manager. Triggering operation was successful as we can see from the result below.

```
Triggering Target Managers operation.

Mediator is notified about the changes in Target Manager
```

For the fourth scenario, we have triggered the Team Manager and set the team role as Helper. Mediator has been notified.

```
Triggering Team Managers operation.

Team has been set as Helper .
Mediator is notified about the changes in Team Manager.
Team role determined as: Helper
```

For the fifth scenario we have triggered the Degradation Manager class. In this case, Mediator has a cooperative effect on the class. Degradation and Team classes can cooperate with each other with the help of the Mediator. When changes are made in the Degradation, Mediator object sets the team and changes the teams current role as we can see from the test result.

```
Triggering the Degradation Manager.

Mediator is notified about the changes in Degredation Manager.
Health has been determined as 10
Mediator notifies the Team Manager.
Team has been set as Help for injured .
Mediator is notified about the changes in Team Manager.
Team role determined as: Help for injured
Team role determined as: Help for injured
```

Test results are successful for our case. We have shown the implementation of the classes and their results. It is clear that Mediator provides loose coupling between objects and it is shown that we are coordinating Manager classes with the Mediator object. Mediator solves the dependency issues and provides flexibility for our Manager classes.

Problem Definition

The World Model should process different kinds of datas.

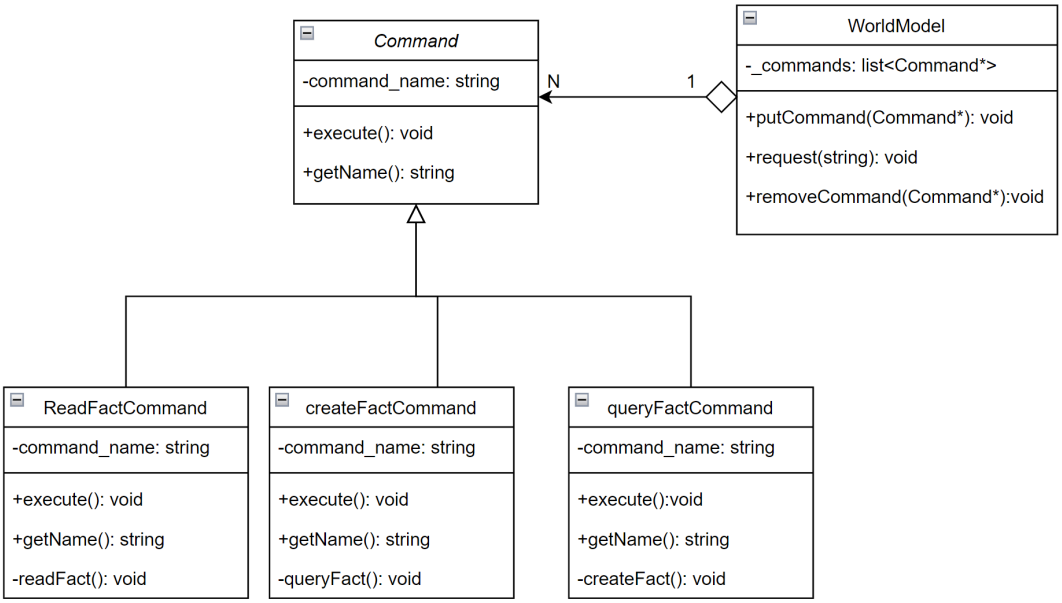
For this problem, the world model should process different kinds of datas coming towards the World Model interface. This interface receives information from Artificial Intelligence, sensors and Facts modules. The World Model should handle datas in order to update the world of the game. This model should concrete on requesting and receiving datas from multiple sources.

Static Class Diagram

For this problem, we have designed the class diagram in order to solve the World Model's fixed interface problem. In this diagram, we have created a new Command superclass for extending the World Model's interface. By doing so, we have created three different command objects which are shown below.

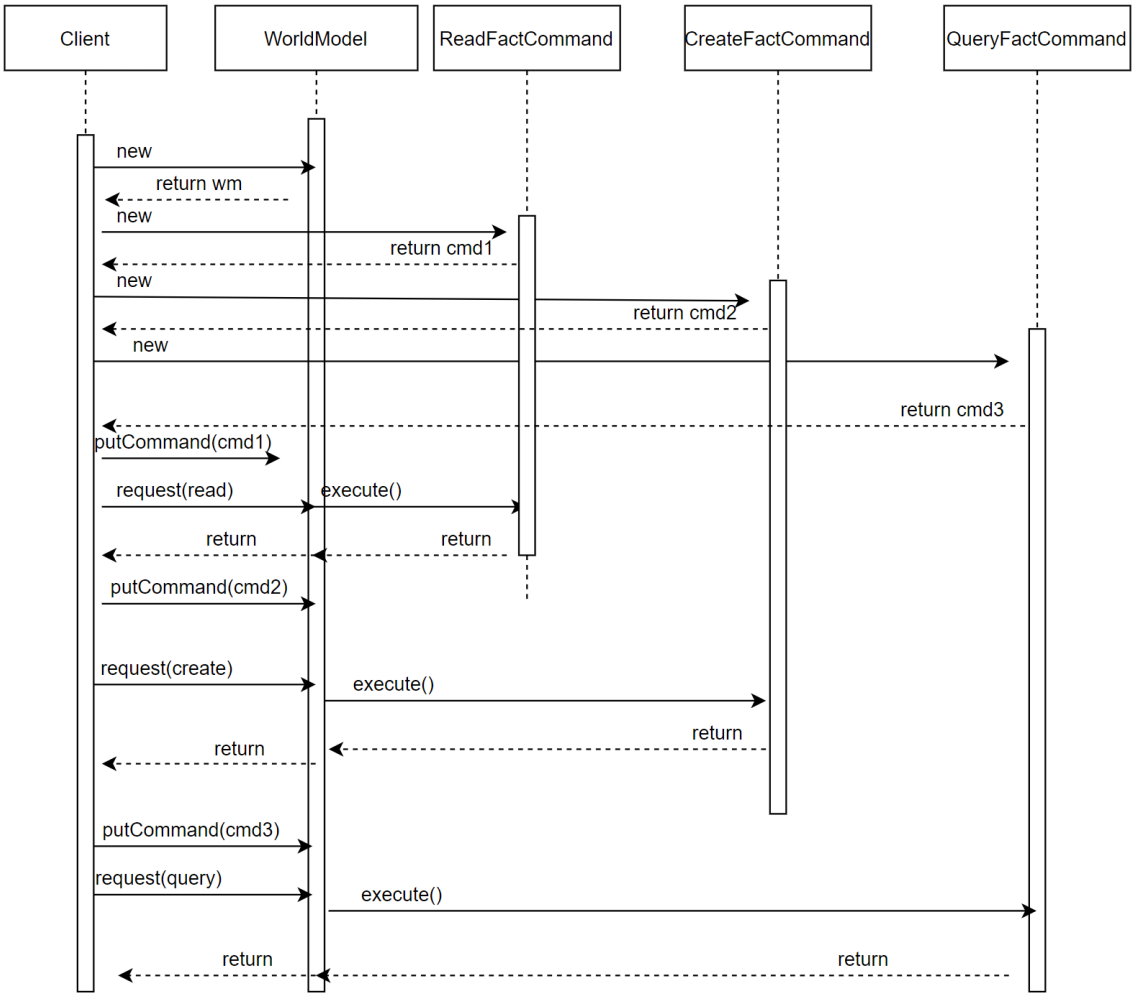
The three command objects are chosen as ReadFactCommand, QueryFactCommand and CreateFactCommand. These commands do their executions. These commands operate differently from each other, as it can be seen from their name. For example, read class reads the queries while create command creates the queries. All executions are different from each other.

Also World Model stores different kinds of commands inside of the command list. We can put commands inside of the World Model by calling the putCommand function. Other than that, we can request some data with the request method.



Dynamic Class Diagram

In this section, we have created a dynamic class diagram in order to show the interactions between classes. In this diagram, we have putCommands on the World Model and then requested some datas as a client. These requests have been answered by the execute methods of the command object classes. As seen from the diagram below, we have demonstrated a usage of the Command pattern for providing flexibility.



Implementation of the Design Pattern

In this part, we are going to show the implementation of the flexibility requirement that is being discussed. This section aims to verify the application of the Command pattern. While implementing the design pattern, C++ programming language is chosen as the main implementer of the pattern. In this section, you can find the structure of the code, test cases and test results. While implementing this design pattern, we have taken notes from the usage of the Command pattern and consider the flexibility requirements of this pattern.

The World Model should process different kinds of datas.

For implementing this module, we should first define our World Model which is corresponding to the static diagram in the static class diagram section. This WorldModel should contain putCommand, request and removeCommand functions in itself in order to process different kinds of datas. As we already

know, the WorldModel receives a lot of information from outside the Artificial Intelligence subdomain.

```
class WorldModel {
public:
    WorldModel() {

    }
    void putCommand(Command*);
    void removeCommand(Command*);
    void request(string data); //request the string
private:
    list<Command*> _commands;
};
```

The code above defines the World Model class. We should now define the methods of this World Model for the functionality of this class.

```
void WorldModel::putCommand(Command* commands) {
    _commands.push_back(commands);
}

void WorldModel::removeCommand(Command* commands) {
    _commands.remove(commands);
}

void WorldModel::request(string data) {
    list<Command*>::iterator itr;
    Command* ptr;
    int is_visited = 0;
    for (itr = _commands.begin(); itr != _commands.end(); itr++) {
      ptr = *itr;
      if (data == ptr->getName()) {
            ptr->execute();
            is_visited = 1;
      }
    }
    if (!is_visited) {
      std::cout << "Exception: Requested cannot be found." <<
```

```
std::endl;
    }
}
```

World Models functionalities are listed above. We can add, remove or request commands for this order. When we put commands, we are adding them into our list as command objects, and when requesting a command we iterate through the list.

```
class Command { // pure virtual interface for commands.
public:
    Command() {
       command_name = "command";
    }
    virtual void execute() = 0;
    virtual string getName() {
       return command_name;
    }
private:
    string command_name;
};
```

Above, we have shown the Command class which is an interface for the command objects called ReadFactCommand, CreateFactCommand and QueryFactCommand. Command class is defined as a pure virtual class, meaning that we are always overriding the execute method of this class.

Now, we can define the Command objects which are subclasses of the Command class.

```
class ReadFactCommand : public Command {
public:
    ReadFactCommand() {
       command_name = "read";
    }
    virtual void execute() override;
    virtual string getName() override;
private:
    void readFact();
    string command_name;
};
```

```
class QueryFactCommand : public Command {
public:
    QueryFactCommand() {
       command_name = "query";
    }
    virtual void execute() override;
    virtual string getName() override;
private:
    void queryFact();
    string command_name;
};

class CreateFactCommand : public Command {
public:
    CreateFactCommand() {
       command_name = "create";
    }
    virtual void execute() override;
    virtual string getName() override;
private:
    void createFact();
    string command_name;
};
```

We have shown the command classes that are residing inside of the "command.h" file. In the "command.cpp" file, we are implementing the methods of the command classes. Every execution method is different from one another as we can see from the code below.

```
void ReadFactCommand::readFact() {
    std::cout << "Fact has been read." << std::endl;
}

string ReadFactCommand::getName() {
    return command_name;
}

void ReadFactCommand::execute() {
    if (command_name == "read") {
```

```
        readFact();
    }
}
```

We have shown the ReadFact command class above. Notice that when executing, we are calling the readFact() method which is responsible for reading the facts. Other classes are responsible for different actions when the execute command is called.

```
void QueryFactCommand::queryFact() {
    std::cout << "Fact has been queried." << std::endl;
}

string QueryFactCommand::getName() {
    return command_name;
}

void QueryFactCommand::execute() {
    if (command_name == "query") {
      queryFact();
    }
}
```

When the QueryFactCommand's executed method is called we would call the queryFact() method. Same thing applies for the CreateFactCommand where we call the createFact() command for execution.

```
string CreateFactCommand::getName() {
    return command_name;
}

void CreateFactCommand::createFact() {
    std::cout << "Fact has been created." << std::endl;
}

void CreateFactCommand::execute() {
    if (command_name == "create") {
      createFact();
```

```
    }
}
```

We have shown the implementation of the Command pattern in our requirement. We can clearly see that all the executions are executing different types of methods, therefore, execution is specified by classes distinctly.

Test Scenarios

For this requirement, we should test the different cases of the command pattern for extending the interface. We should test our patterns' accuracy by implementing test scenarios. The test scenarios that are being implemented are written in the C++ language. These scenarios should provide flexibility for our requirements.

For the first scenario, we have created the Command objects. As it is seen from the requirement, there may be more than one request at a time, therefore we have created the commands distinctively.

```
WorldModel* wm = new WorldModel();
Command* cmd1 = new ReadFactCommand();
Command* cmd2 = new CreateFactCommand();
Command* cmd3 = new QueryFactCommand();
```

For this scenario, a World Model object and command objects are created. We should now put commands in an order by calling the putCommand() method of the WorldModel.

```
wm->putCommand(cmd1);
wm->putCommand(cmd2);
wm->putCommand(cmd3);
```

We have now put commands inside of our list that is being defined on the WorldModel class.

We should now start requesting “read”, “create”, “query” strings from the Commands.

```
wm->request("read");
```

```
wm->request("create");
wm->request("query");
```

We requested the datas as it can be seen from above.

Now, for another test case, we are requesting some data from the Command classes which are not defined. These Commands are not known and we have not defined such strings in this case. We should test what the Command object will do in this scenario.

```
wm->request("delete");
wm->request("action");
wm->request("something");
```

We can also remove commands from the command list. Removal is another test case that we will show.

```
wm->removeCommand(cmd1);
wm->removeCommand(cmd2);
wm->removeCommand(cmd3);

wm->request("read");
wm->request("create");
wm->request("query");
```

Now the commands have been removed from the system, therefore, we would get an error since read, create and query commands are removed from the command list.

We can add back these and find the requests.

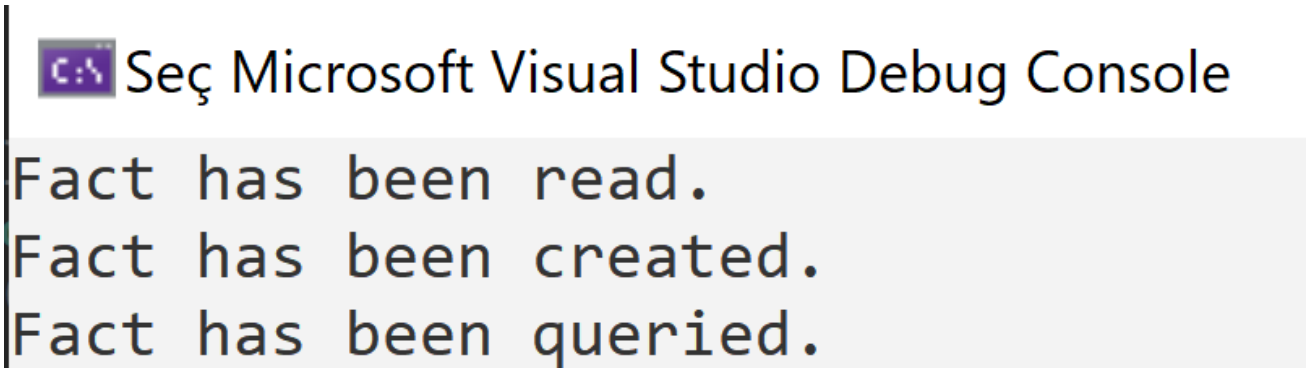
```
wm->putCommand(cmd1);
wm->putCommand(cmd2);
wm->putCommand(cmd3);
wm->request("read");
wm->request("create");
wm->request("query");
```

These are our test cases for the command pattern. For now, we can show the test results.

Test Results

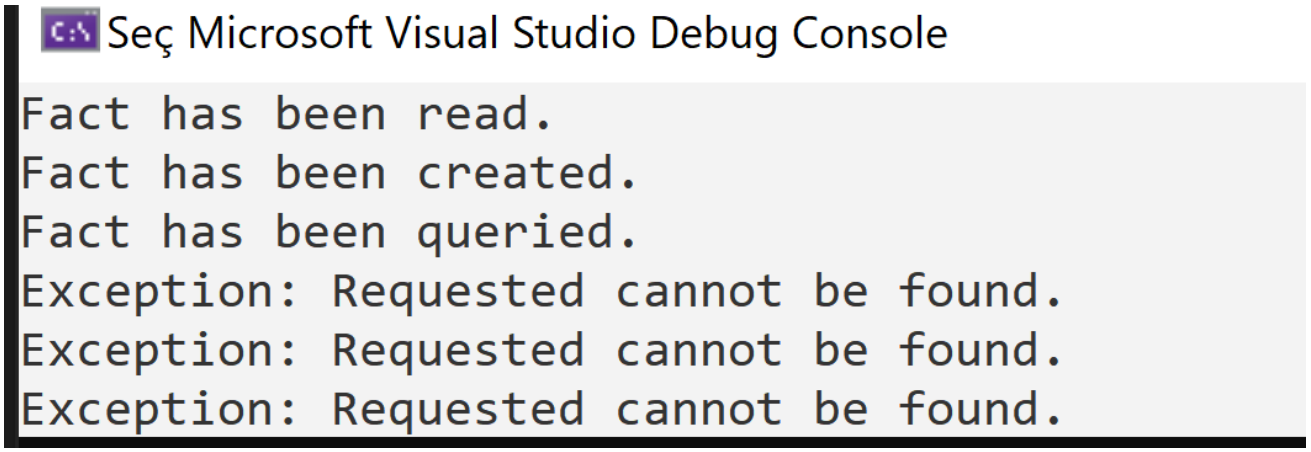
In the implementation section of this report, we have implemented the Command design pattern with the C++ programming language. Now, we can move on with the test results in order to show the accuracy and flexibility for this requirement.

For the first test case, we have put the Command objects in an array in the World Model and we get to request some data from the Command objects.



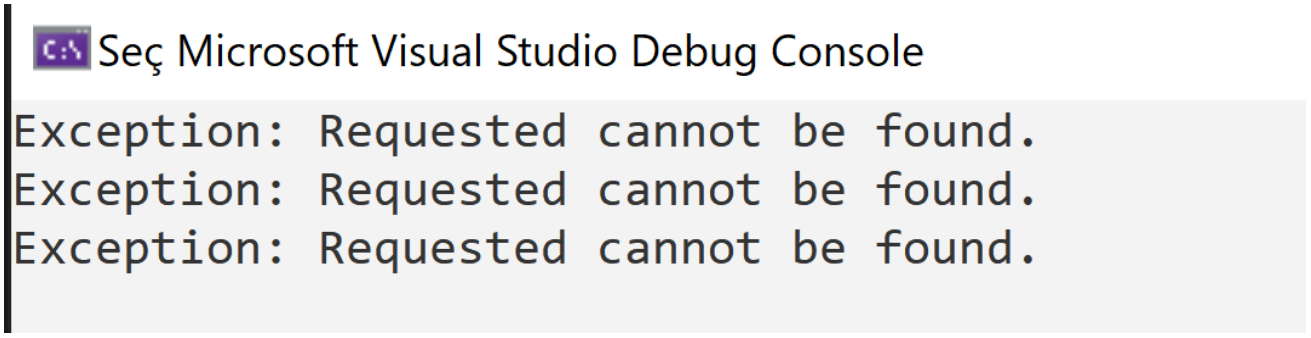
As it can be seen from the screenshot of this test case, execution methods work correctly for requesting datas. It is clear that we have put the commands in a list and later requested the strings that they identified with.

For the second test case, we requested the strings that are not identified or defined by the Command objects.



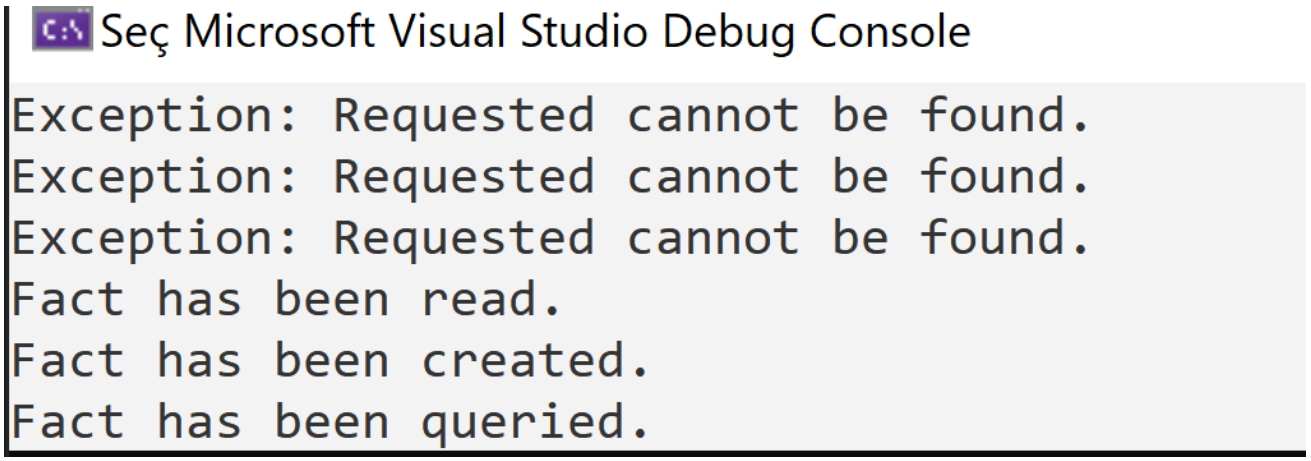
As it can be seen from above, we executed “read”, “create” and “query” commands however, we could not request “delete”, “action”, “something” datas from the Command objects since they are not defined as Command classes.

For the third scenario, we have removed the Command objects from our system. When we remove them and request “read”, “create”, “query” from the Command objects, all we get is exceptions as it can be seen from below.



Requests cannot be found since we remove command objects from the list itself.

For the fourth case, we add the requests back again in our list and request “read”, “create”, “query” from the Command objects.



This time we did not get an exception. We have successfully read, created and queried the facts that have been asked from us. We can say that we have satisfied the flexibility requirement by making the WorldModels interface flexible.

4) What We Learned and Conclusions

In this section, we are going to inform you about what we have achieved in this course and the problems that we have faced. The results of the Project will be discussed in this section of the report.

a. Knowledge about Patterns

I had very little knowledge about patterns before taking this class. With the Design Patterns course, I have gained the important thinking skills when it comes to object-oriented programming. Theoretical knowledge that has been provided to us was very helpful in this context. Learning the implementations of different solving techniques was also very helpful for my understanding in this class.

Finally, I am willing to use my knowledge skills further in my professional working life. I am also willing to learn more about different types of design patterns and apply them in problems that I have encountered.

a. Challenges

Correlation Between Pattern and Problem

I have faced different types of problems when it comes to patterns. At first, thinking about solving a pattern is not quite approachable for me since I did not know how to apply any design patterns before. However, when I studied and read about patterns, my knowledge on patterns developed and I started to think more critically about problems. It is a challenging process for me to learn some of the patterns, however, the implementation of patterns can be enjoyable and fun at times. Sometimes, choosing a pattern to solve the problem is hard for me to evaluate since I could not distinguish some patterns from others. But with enough studying, I am able to think more rationally and critically about patterns.

- **Pattern Design and Implementation**

Sometimes implementing a pattern is difficult for me. Sometimes the pattern that I had in my mind did not match quite right with the solution itself. Therefore, I learned that the rules can be flexible for patterns. We may even combine some patterns in order to solve issues.

b. Conclusion

The main goal of our work was to achieve a flexible structure in our war game. In order to achieve this goal, we have analyzed the report that was given to us. We had brainstorming about how war games work and what we can do to achieve a more flexible and consistent game.

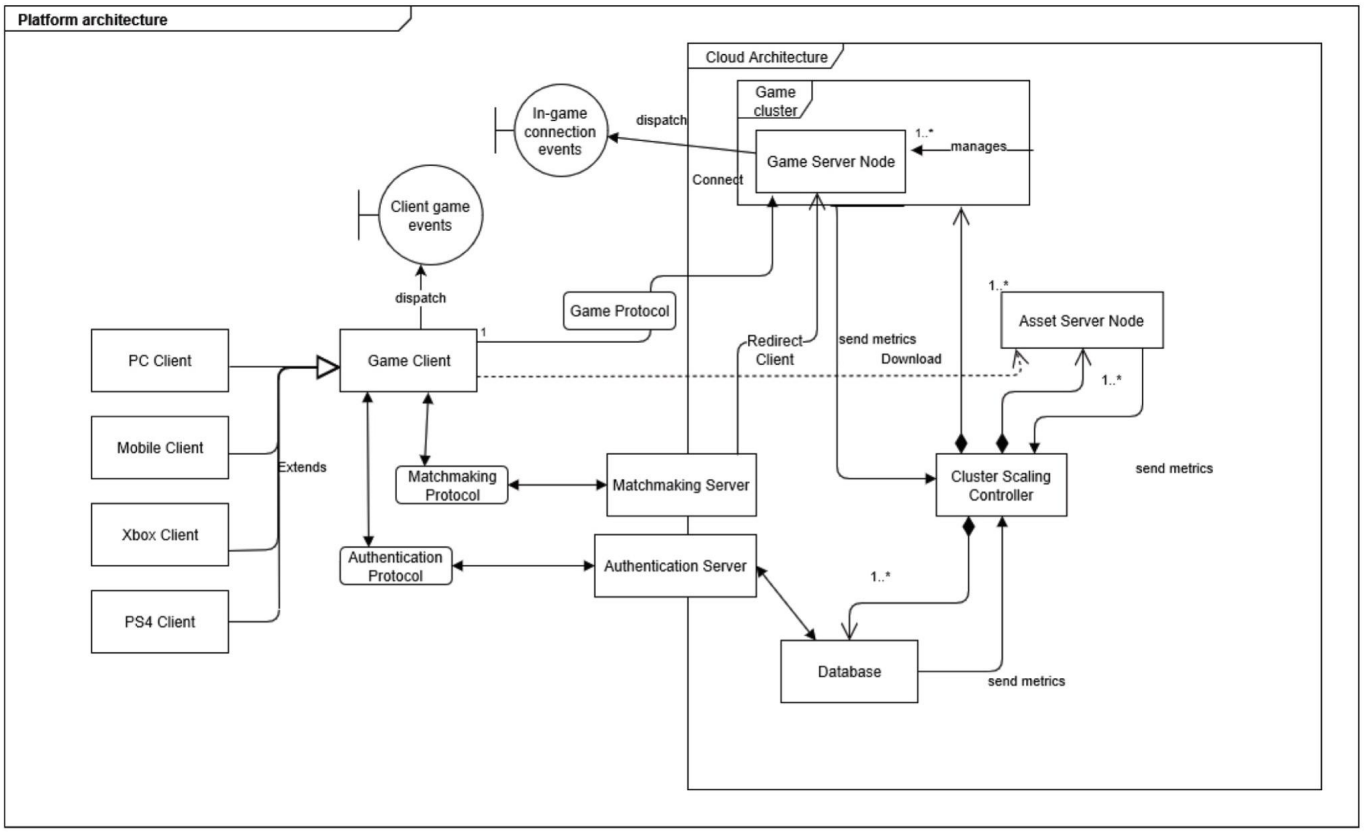
Then we began to work separately from each other and took different subdomains of the game. The main purpose is to find 10 different flexibility requirements to make the game more enjoyable and flexible in itself. When the requirements were found, we began to implement 5 of the problems that we have chosen. For this, we made static, dynamic class diagrams and wrote implementation codes. The codes were mainly written in C++ and Java. Codes and explanations are added to the report. We have also shown how the solutions work effectively with each other and how they solved the flexibility requirements that have been discussed before.

As a result of our hard work, we have maintained a flexible and more structural War Game that has been asked from us. We implemented design patterns in order to solve flexibility issues and worked on object-oriented programming very closely. We have learned more about design patterns and their implementations.

3) Side System: Platform

(Author: Baran Özgenç)

Platform architecture from the original report. (Verslag)



Sharing the domains of the game

Purpose: Introduces job sharing of domains amongst group members.

The game's architecture has four main domains. Those are Game Engine, Game Unit, Artificial Intelligence and Platform domains. Since we are four people writing this report, each one of us has chosen one domain and made a detailed analysis for the domain we chose, found flexibility requirements on our domains and we found the suitable Design Patterns to solve these specific problems. Design Patterns have been compared to each other and one Design Pattern has been selected to solve a requirement problem. Then each one of us implemented five of these Design Patterns by making their static class diagram, dynamic sequence diagram and writing the Design Patterns code in a programming language. Lastly we tested the implementation if they solve the requirement problems.

Flexibility Requirements

Purpose: This chapter introduces the flexibility requirements that the platform part of the game architecture should have then for each requirement possible design pattern recommendations are mentioned in order to achieve these requirements.

In this subpart, we obtained the flexibility requirements of the Platform domain by carely examining the Platform Architecture subpart of the report that we are given. While examining the platform architecture section of the report, we came to the conclusion that it is not completely disconnected from flexibility, but there are points that can be improved.

1) **Players should not face network-connection quality related issues such as ping, lag vs.**

   For a superior game experience, all users who connected to game servers should have low latency values. It is an commonal choice to increase server capacity but since it is unknown the possibility of how widely the servers will be placed in terms of geographically, this solution may not be sufficient enough.
   On the other hand, the report that we are given only focuses on the client-server model in the network communicating
   aspect. However network communicating methods can be enriched by implementing a P2P (peer to peer) connection as an alternative.

   With this approach, players who are physically far away from servers can be grouped separately and still have the low latency values.

2) **Game Clusters(also mentioned as servers) should follow occupancy ratio of Game Nodes (which players are connected) effectively**

   As the platform architecture from the original report states, when a player opens the game, after authenticating process, player joins to the Matchmaking Server and redirected to an arbitrary game node which helds place in servers.
   It is a vital problem for servers to monitor Game node occupancies in order to distribute new players effectively amongst nodes.

While this situation is briefly mentioned in the original report there is not a detailed description and it does not take place in the platform architecture diagrams.
However, we believe it is essential to have this feature for effective resource management.

3) **Servers should distribute network load uniformly between themselves (either communicating directly with each other or via an external manager)**

As an addition to the previous requirement, this is also quite an important requirement for resource management.
In the current architecture design, matchmaking server redirects players to the servers after running some algorithms about physical distance to server, load of server etc.
To make this process smoother, it is a good idea to implement a way that servers should communicate with each other and report the occupancy ratio of the game nodes which are held inside them.

With this approach, matchmaking servers can detect the most eligible server directly (in terms of network load) instead of checking servers separately.
As a result of that, waiting time for players (who wait in the waiting room to join game nodes) can be significantly reduced.

4) **The process of finding which node a player is in should be efficient (for dropping, or redirecting the player)**

As we said before when a player joins the game he/she is redirected to a game server by the Matchmaking server. In the game servers there are multiple game nodes which players are connected to. When a player decides to exit the game, we should determine the exact game node in the server to drop the player.

Another case is related to flexibility requirement number one. After adding the peer-to-peer connection system to the game we also should determine the game node of the player who is going to be redirected to peer-to-peer connection.

It is a good idea to implement a tool that traverses the game nodes and finds the player effectively.

5) **When expanding or removing game nodes players should be relocate between nodes efficiently**

As the original report states that there are game nodes in the game servers, and when players connect to the game servers they are attached to these game nodes. If the load of a game session becomes too much a new node will be created and new players will be attached to these new nodes.

For a better management of resources, if players exit the game and if a particular node becomes too light it should be removed and players inside that node will be relocated to the other nodes. We should do this relocating process without the loss of the player data.

Therefore we need a system that copies the player data which in the node is going to be deleted.

6) **Asset Server Node should handle dispatch process effectively**

When a user joins the game, he/she receives textures of game worlds, music and voices from Asset Server. Nowadays, since the hardware features of the systems are different, game skins are available in multiple resolutions such as low medium high quality. According to the quality of the players' devices, the server determines the textures with the appropriate resolution and dispatches it to players' devices.

As you can understand Asset Server continuously receives information from the Game Client and dispatches the appropriate asset to the player. When there are hundreds, thousands of players it is vital to handle this process effectively.

7) **Same player should not connect to the game with different game clients (e.g. player should not connect to the game with PC and PS4 Client)**

Since this is a cross platform game, players can join the server from different game clients such as PC, Mobile, Xbox or PS4. However a player should not

connect to the game servers with different clients but with the same credentials at the same time. Because in such cases it can cause awkward situations like duplicate game objects and it may crash the game.

Therefore a system must be implemented so that if a user with the same credentials tries to connect when he/she is already logged in, the system must force the player to log out from another game client.

Flexibility Requirement	Possible Design Pattern Solution
1- Players should not face network-connection quality related issues such as ping, lag vs.	Bridge,Strategy
2- Game Clusters(also mentioned as servers) should follow occupancy ratio of Game Nodes (which players are connected) effectively	Observer, Publisher & Subscriber
3-**Servers should distribute network load uniformly between themselves (either communicating directly with each other or via an external manager)**	Mediator
4- The process of finding which node a player is in should be efficient (for dropping, or redirecting the player)	Iterator, Chain of Responsibility
5- When expanding or removing game nodes players should be relocate between nodes efficiently	Prototype
6- Asset Server Node should handle dispatch process effectively	Command

7-Same player should not connect to the game with different game clients (e.g. player should not connect to the game with PC and PS4 Client)	Observer

1) **Players should not face network-connection quality related issues such as ping, lag vs.**

   As we stated before, for a superior game experience not only geographically lucky players, also unlucky players should have low latency values. And to achieve this we offered a P2P server as an alternative for the client-server model.

   With the help of a Strategy or Bridge design pattern (patterns that alter behavior during runtime) users can be relocated in between client-server or peer to peer server according to the network analytics.

   By doing this, not only players will be affected better in terms of gaming experience also we will be able to reduce network loads on the server. (This is an side effect, main focus in this flexibility requirement is for player perspective)

2) **Game Clusters(also mentioned as servers) should follow occupancy ratio of Game Nodes (which players are connected) effectively**

   As we mention in the flexibility requirements part, servers should be aware of the game nodes' occupancy level but not necessarily track them always. There are only two threshold points that matter. One for is when the game node is almost empty and one for the game node is almost full.
   Hence, we chose the observer design pattern because it is the pattern that fulfills our expectations. An observer can observe the game nodes and notify the server about occupancy level when two threshold points are passed. (These threshold points can be calculated according to the hardware adequacy of the servers.)

3) **Servers should distribute network load uniformly between themselves (either communication directly with each other or via an external manager)**

Nowadays, most of the games are being played by thousands or millions of players who are located in different places worldwide. In a market like this, distributing these players uniformly between servers is a crucial challenge. Matchmaking server (responsible for redirecting players to the game servers) and game servers should be communicating with each other all the time but since there are lots of servers, managing this communication traffic can be a complex problem.

Hence, a mediator pattern can fit here to solve this communicating complexity. Servers can communicate with each other and report their occupancy level through a mediator object. This approach is also beneficial for matchmaking server for reducing communication traffic between itself and game servers to obtain the most eligible one for player redirection.

4) **The process of finding which node a player is in should be efficient (for dropping, or redirecting the player)**

As we said before, in a game players join and leave continuously. Therefore this process should be handled delicately. When a user exits the game we should determine the player's node effectively in order to drop the player from that server.

To achieve this, we need a structure that traverses the nodes in a game server. Iterator and Chain of Responsibility is eligible for our problem. However, Chain of Responsibility loses its advantage when there are too many chain objects because implementation is going to be too complicated.

On the other hand, the Iterator design pattern is more eligible to traverse game nodes because in a server most likely there will be hundreds or maybe even thousands of nodes. In such a case the chain of responsibility will be an unstable solution.

5) **When expanding or removing game nodes players should be relocate between nodes efficiently**

As we said before, when a node is going to be deleted in a game server the players and their data should be relocated into another node. To do so, we should clone the player objects from that particular node and relocate those objects into a new node.

To achieve this process, we can use the Prototype design pattern. It allows us to copy existing player objects without being dependent on their classes. We can clone the player objects with the Prototype pattern and after that we can delete the node safely.

6) **Asset Server Node should handle dispatch process effectively**

As mentioned before, when a player connects to the game, all the assets related to the game world are received from the asset server. In a game that consists of too many players, there will be a considerable amount of traffic between the asset server and the game client. In order to convert this traffic to a more manageable way, we should restrict the asset server's interactions and add a mid level component to help the asset server.

With the help of the Command design pattern we can achieve this goal. In this renewed architecture asset server no longer communicates directly with the game client. Instead, a mid level component communicates with the game client and receives all the information from the game client. And in this component this information can be converted to a receipt and propagated to the asset server. Asset server reads the receipt from its queue and dispatches the assets to the mid level component. Component checks the received data and propagates it to the game client.

7) **Same player should not connect to the game with different game clients (e.g. player should not connect to the game with PC and PS4 Client)**

As mentioned before, to avoid a player to log in to the game with different game clients there is a trace and control mechanism is essential. We can implement this mechanism with an Observer design pattern. An observer can observe the game nodes and when a new connection request is made, it can compare the upcoming players credentials with the credentials of players who are currently in the game nodes. Then if a player is detected with the same credentials, a notification can be sent to both of the game clients of the player.

Therefore with this approach we can guarantee that a player can be connected to the game with one game client at a time.

Problem Description

In the previous section we have mentioned the flexibility requirements and possible design patterns solutions to accomplish those flexibility requirements. In this section, you can find five of the flexibility requirements that are selected to solve. We also showed which design pattern is selected to meet the specific requirement and their static and dynamic class diagram.

Problem Definitions

Purpose: This section introduces detailed descriptions, implementations of the five of the flexibility required that mentioned in the earlier section.

1) Game Clusters(also mentioned as servers) should follow occupancy ratio of Game Nodes (which players are connected) effectively

a) What was the problem, what did we choose?

Purpose: This subsection reminds what was the flexibility requirement, and which design pattern chosen to achieve it.

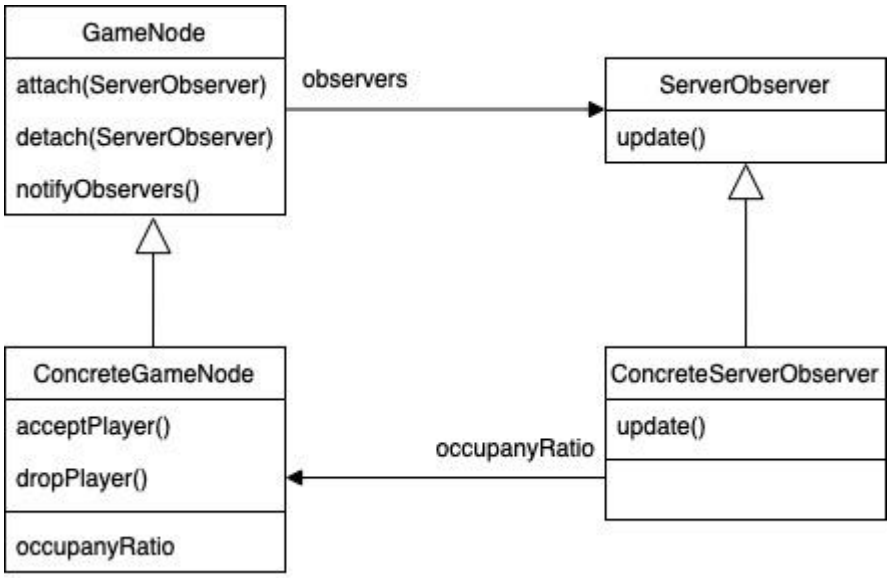
We mentioned in the previous section but as a quick reminder, when users join a game, they are redirected to GameNodes which are held in GameServer. In order to balance server load uniformly between nodes it is crucial to follow each node's current occupancy ratio.

Therefore, the selected design pattern ist the Observer Pattern to achieve this flexible requirement. The reason behind that is Observer can effectively notify the game server when it is required to be notified about node’s occupancy status. With this approach there is no need for Game Server to check nodes arbitrarily.

In the next two subsections you can find the static class diagram and dynamic sequence diagram of the Observer design pattern.
While creating these diagrams we used Gamma’s Book[1] as the main reference.

b) Static Class Structure of the Pattern

Purpose: This subsection introduces a UML diagram of the design pattern.

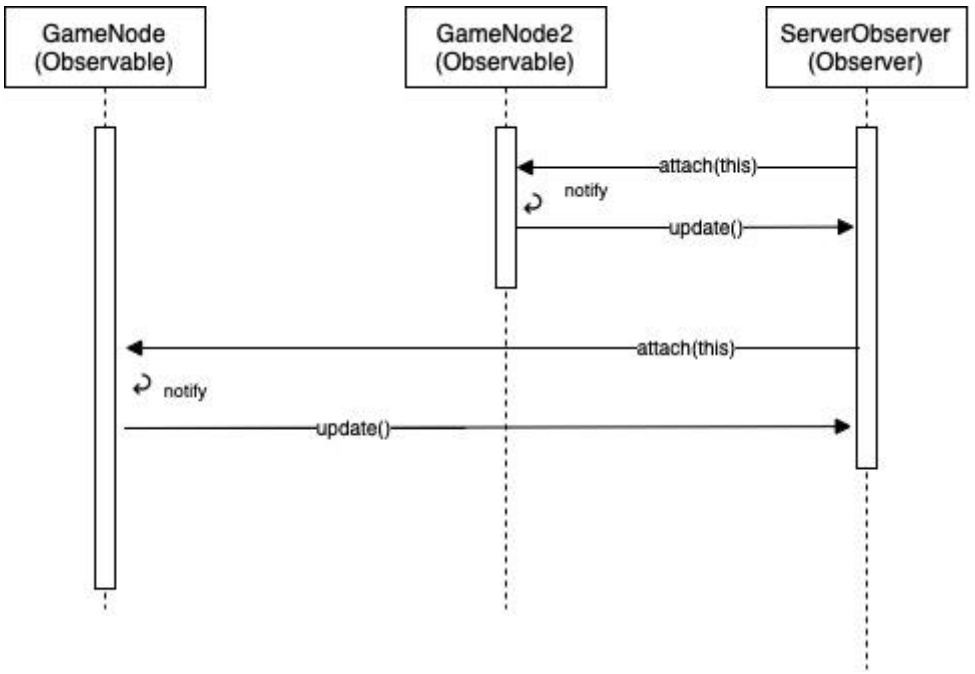


This UML diagram represents the class structure of the Observer Design Pattern. GameNode and ServerObservers classes are our Interfaces. ConcreteGameNode and ConcreteServerObserver classes are our concrete implementations. ServerObserver is our observer as its name reflects. On the other hand GameNode is the observable class.

Firstly, GameNode accepts the observers(which is ServerObserver) who want to observe itself via the attach method. And at a point if an observer decides not to observe GameNode any more it can be detached via the detach method.
Secondly, GameNode tracks a list of observers who are attached to itself and notifies them via notify method when a state change happens.

c) Dynamic Structure of the Pattern

Purpose: This subsection introduces a sequence diagram of the design pattern.



In the above UML sequence diagram you can see the run time interactions between observable and observer classes.

In the platform architecture, one server contains many game nodes. Therefore there is a need for an observer to observe these nodes. In the sequence diagram we only showed two of the observable classes but the logic stays the same no matter how many observable classes we have.

Explanation of the sequence diagram:
ServerObserver attaches to GameNode and GameNode2 and starts to observe them. When a change happens in these classes they notify the Observer via update method.

d) Introduction of code and relation between the pattern

Purpose: This subsection introduces the source code of implementation, and describes the relation between code and design pattern.

In the implementation, there are four classes directly related to the design pattern.

First, an interface GameNode class which is our observable class:

```
public interface GameNode {

   public void attach(ServerObserver observer);
   public void detach(ServerObserver observer);
   public void notifyObservers();
}
```

Observers can attach and detach to the GameNode via attach and detach methods. And notifyObservers() method is used for notifying observers who are attached.
Second, an interface ServerObserver class which is our observer class:

```
public interface ServerObserver {

   public void update(int occupancyRatio, String nodeName);
   public String getName();
}
```

Concrete implementations must implement the update() method to get the information from the observable class. And the getName() method is going to be used for distinguishability between concrete implementation objects.

Third, concrete implementation of GameNode(observable class) which is called ConcreteGameNode:
Since it is a relatively long code, it is described partly.

```
import java.util.ArrayList;
public class ConcreteGameNode implements GameNode{
   private ArrayList<ServerObserver> observers;
   private int occupancyRatio;
   private String name;

   public ConcreteGameNode(String name){
       observers = new ArrayList<ServerObserver>();
       this.name = name;
       this.occupancyRatio = 20;
   }
```

When instantiating a new node, in the constructor a new ArrayList will be created as well. Name variable is going to be used for distinguishability and the occupancy ratio should be initialized with 0 in real world use case but it is predefined as 20 for test purposes.

```
   @Override
   public void attach(ServerObserver observer) {
       observers.add(observer);
       System.out.println("Observer " + observer.getName() + " is attached to " +
this.name);
   }
```

Attach method takes an observer object as parameter and appends it to the observer list to keep track of which observers are going to be notified.

```
   @Override
   public void detach(ServerObserver observerToDelete) {
       int observerIndex = observers.indexOf(observerToDelete);
       observers.remove(observerIndex);

       System.out.println("Observer: " + "(" +  observerToDelete.getName()+ " )"
+ " is detached from " + this.name);
```

```
    }
```

Similarly, the detach method takes an observer object as a parameter and this time it removes that observer from the observer list.

```
    @Override
    public void notifyObservers() {
        for(ServerObserver o : observers) {
            o.update(occupancyRatio, this.name);
        }
    }
```

The NotifyObservers method notifies the observers as its name says. During this process it sends the name of the GameNode object and the current occupancy ratio as parameters.

```
    private void incrementNodeOccupancy(){
        this.occupancyRatio++;
        notifyObservers();
    }

    private void decrementNodeOccupancy(){
        this.occupancyRatio--;
        notifyObservers();
    }
```

Increment and decrement node occupancy methods, do what they say and call notifyObservers() method to notify them.

```
    public void acceptPlayer() {
        incrementNodeOccupancy();
        System.out.println("Player joined the game node 1. Current player count: "
+ occupancyRatio);
    }

    public void deletePlayer() {
        decrementNodeOccupancy();
        System.out.println("Player left the game node 1. Current player count: " +
occupancyRatio);
    }
}
```

Finally acceptPlayer and deletePlayer methods are used for when a player joins to GameNode or leaves from GameNode. And they trigger

incrementNodeOccupancy() and decrementNodeOccupancy() methods respectively.

e) Test Scenario

Purpose: This subsection introduces a test scenario to examine the effects of the design pattern.

As a test scenario, we have created a GameNode object as GameNode1 and two ServerObserver objects as ServerObserver1 and ServerObserver2.

After that, we tested attach and detach operations on GameNode1 with both our observers. (Note that we are trying to simulate a moment when there are already many players present in the GameNode.)

You can see the test results in the next subsection.

f) Test Results

Purpose: This subsection introduces test results of the given scenario.

Firstly, Observer1 is attached to GameNode1. It is notified at every join/leave action of players but it ignores the occupancy levels between 10 and 80 (these values can be changed).

As you can see from the output when game nodes' occupancies reaches a critical level or if they are too low then observer will print an output (In the real world case instead of printing an output, a new node will be generated or going to deleted) :

```
Observer  <Observer1> is attached to GameNode1
Observer  <Observer1> is attached to GameNode2
Player joined the game node 1. Current player count: 21
```

```
Player joined the game node 1. Current player count: 22
Player joined the game node 1. Current player count: 23
Player joined the game node 1. Current player count: 24
Player joined the game node 1. Current player count: 25
Player joined the game node 1. Current player count: 26
Player joined the game node 1. Current player count: 27
Player joined the game node 1. Current player count: 28
Player joined the game node 1. Current player count: 29
Player joined the game node 1. Current player count: 30
Player joined the game node 1. Current player count: 31
Player joined the game node 1. Current player count: 32
Player joined the game node 1. Current player count: 33
Player joined the game node 1. Current player count: 34
Player joined the game node 1. Current player count: 35
Player joined the game node 1. Current player count: 36
Player joined the game node 1. Current player count: 37
Player joined the game node 1. Current player count: 38
Player joined the game node 1. Current player count: 39
Player joined the game node 1. Current player count: 40
Player joined the game node 1. Current player count: 41
Player joined the game node 1. Current player count: 42
Player joined the game node 1. Current player count: 43
Player joined the game node 1. Current player count: 44
Player joined the game node 1. Current player count: 45
Player joined the game node 1. Current player count: 46
Player joined the game node 1. Current player count: 47
Player joined the game node 1. Current player count: 48
Player joined the game node 1. Current player count: 49
Player joined the game node 1. Current player count: 50
Player joined the game node 1. Current player count: 51
Player joined the game node 1. Current player count: 52
Player joined the game node 1. Current player count: 53
Player joined the game node 1. Current player count: 54
Player joined the game node 1. Current player count: 55
Player joined the game node 1. Current player count: 56
Player joined the game node 1. Current player count: 57
Player joined the game node 1. Current player count: 58
Player joined the game node 1. Current player count: 59
Player joined the game node 1. Current player count: 60
Player joined the game node 1. Current player count: 61
Player joined the game node 1. Current player count: 62
Player joined the game node 1. Current player count: 63
Player joined the game node 1. Current player count: 64
Player joined the game node 1. Current player count: 65
Player joined the game node 1. Current player count: 66
Player joined the game node 1. Current player count: 67
Player joined the game node 1. Current player count: 68
Player joined the game node 1. Current player count: 69
Player joined the game node 1. Current player count: 70
Player joined the game node 1. Current player count: 71
Player joined the game node 1. Current player count: 72
Player joined the game node 1. Current player count: 73
Player joined the game node 1. Current player count: 74
```

```
Player joined the game node 1. Current player count: 75
Player joined the game node 1. Current player count: 76
Player joined the game node 1. Current player count: 77
Player joined the game node 1. Current player count: 78
Player joined the game node 1. Current player count: 79
Player joined the game node 1. Current player count: 80
Observer1:
GameNode1 is loaded. Expansion needed. New node will be created.
---------------------------------------------------------------------
```

After that, we detached ServerObserver1 and instead attached a new observer called ServerObserver2. Then we simulated player drops in the game node. At this moment we expect only ServerObserver2 should be notified and the test terminated successfully.

Here are the results.

```
Observer: <Observer1> is detached from GameNode1
Observer  <Observer2> is attached to GameNode1
Observer  <Observer2> is attached to GameNode2
Player left the game node 1. Current player count: 79
Player left the game node 1. Current player count: 78
Player left the game node 1. Current player count: 77
Player left the game node 1. Current player count: 76
Player left the game node 1. Current player count: 75
Player left the game node 1. Current player count: 74
Player left the game node 1. Current player count: 73
Player left the game node 1. Current player count: 72
Player left the game node 1. Current player count: 71
Player left the game node 1. Current player count: 70
Player left the game node 1. Current player count: 69
Player left the game node 1. Current player count: 68
Player left the game node 1. Current player count: 67
Player left the game node 1. Current player count: 66
Player left the game node 1. Current player count: 65
Player left the game node 1. Current player count: 64
Player left the game node 1. Current player count: 63
Player left the game node 1. Current player count: 62
Player left the game node 1. Current player count: 61
Player left the game node 1. Current player count: 60
Player left the game node 1. Current player count: 59
Player left the game node 1. Current player count: 58
Player left the game node 1. Current player count: 57
Player left the game node 1. Current player count: 56
Player left the game node 1. Current player count: 55
Player left the game node 1. Current player count: 54
Player left the game node 1. Current player count: 53
Player left the game node 1. Current player count: 52
```

```
Player left the game node 1. Current player count: 51
Player left the game node 1. Current player count: 50
Player left the game node 1. Current player count: 49
Player left the game node 1. Current player count: 48
Player left the game node 1. Current player count: 47
Player left the game node 1. Current player count: 46
Player left the game node 1. Current player count: 45
Player left the game node 1. Current player count: 44
Player left the game node 1. Current player count: 43
Player left the game node 1. Current player count: 42
Player left the game node 1. Current player count: 41
Player left the game node 1. Current player count: 40
Player left the game node 1. Current player count: 39
Player left the game node 1. Current player count: 38
Player left the game node 1. Current player count: 37
Player left the game node 1. Current player count: 36
Player left the game node 1. Current player count: 35
Player left the game node 1. Current player count: 34
Player left the game node 1. Current player count: 33
Player left the game node 1. Current player count: 32
Player left the game node 1. Current player count: 31
Player left the game node 1. Current player count: 30
Player left the game node 1. Current player count: 29
Player left the game node 1. Current player count: 28
Player left the game node 1. Current player count: 27
Player left the game node 1. Current player count: 26
Player left the game node 1. Current player count: 25
Player left the game node 1. Current player count: 24
Player left the game node 1. Current player count: 23
Player left the game node 1. Current player count: 22
Player left the game node 1. Current player count: 21
Player left the game node 1. Current player count: 20
Player left the game node 1. Current player count: 19
Player left the game node 1. Current player count: 18
Player left the game node 1. Current player count: 17
Player left the game node 1. Current player count: 16
Player left the game node 1. Current player count: 15
Player left the game node 1. Current player count: 14
Player left the game node 1. Current player count: 13
Player left the game node 1. Current player count: 12
Player left the game node 1. Current player count: 11
Player left the game node 1. Current player count: 10
Player left the game node 1. Current player count: 9
Observer2:
GameNode1 can be deleted
------------------------------------------------------------------------
```

As a final note, you can see in the outputs actually there are two game nodes (GameNode1 and GameNode2). Observers attached to both of

them actually. But to make this section no more longer, we only showed the outputs of operations related to GameNode1.

2) **Servers should distribute network load uniformly between themselves (either communicating directly with each other or via an external manager)**

a) What was the problem, what did we choose?

Purpose: This subsection reminds what was the flexibility requirement, and which design pattern chosen to achieve it.

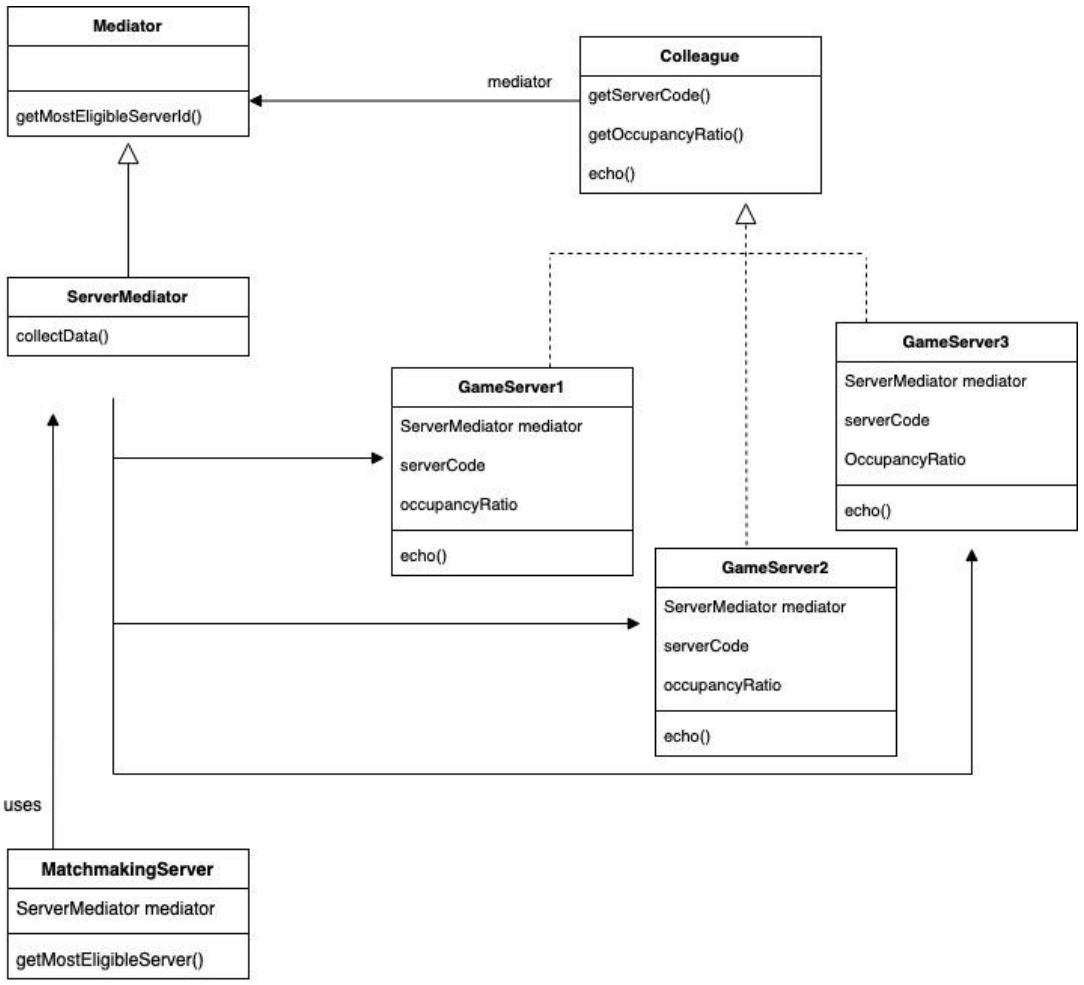
As how important balancing nodes within a server, also it is highly crucial balancing server loads between themselves. According to the original report, when a user joins the game, MatchmakingServer redirects the user to a game server but it is not detailed how this redirection process works.

However the report mentions that servers should communicate with each other via a 'Cluster Scaling Controller' to obtain current statuses and again it is not detailed how that can happen.

Therefore we chose the Mediator design pattern to meet this flexibility requirement. With this approach, when a user requests from MatchmakingServer to join a game, this server can find the most eligible game server (in terms of network load) only communicating with a mediator object rather than checking all servers.

b) Static Class Structure of the Pattern

Purpose: This subsection introduces a UML diagram of the design pattern.



This UML diagram represents the class structure of the Mediator design pattern. Mediator class is an interface for the mediator object. Colleague is the interface for classes that communicate through the mediator object.

ServerMediator is the concrete implementation of the mediator object. GameServer1, GameServer2, GameServer3 are concrete implementations of the Colleague class. Note that there can be, and likely will be more servers than three but for this UML diagram there are only three concrete implementations shown. Logic stays the same independently from concrete colleague implementation numbers.

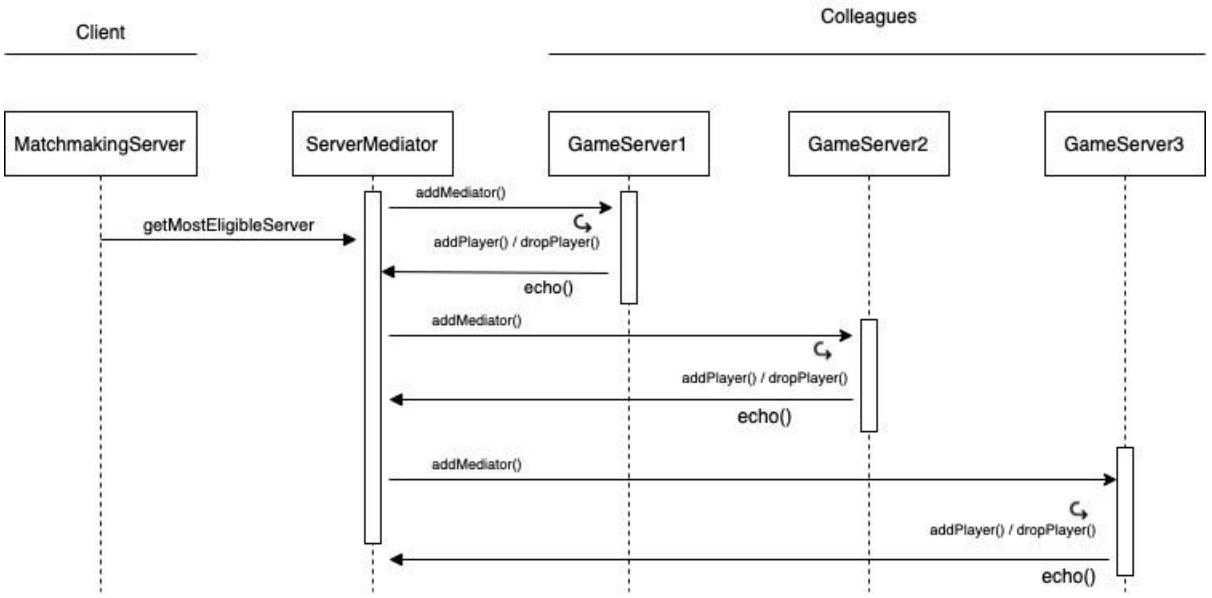
And finally MatchmakingServer is the component that asks the mediator object what is the most eligible server at the moment and redirects the user to that server.

So the logic behind this pattern is quite simple, GameServers echoes to the mediator their current occupancy ratio when an internal change of state occurs. And the mediator object calculates the most eligible game server with the informations gathered from the game servers for the MatchmakingServer.

Besides the mediator object simplifies communication between components, it also makes adding/deleting server processes easier.
No matter how many game servers will be added, it is enough to introduce that new game server to the mediator object.
And deleting a game server process is also simple. All we have to do is cut the relationship between the mediator object and the game server. Without the mediator object, we had to introduce the new server to all servers available. And while deleting a server from our system we had to update all other servers. Thus, with the mediator object we can significantly reduce the interconnections between servers.

c) Dynamic Structure of the Pattern

Purpose: This subsection introduces a sequence diagram of the design pattern.



In the above sequence diagram you can see the runtime interactions between the mediator object and its colleagues.

In the real world, there will be more than three servers but for this sequence diagram we only showed the three of them. However, logic stays the same no matter how many servers we have.

Explanation of the sequence diagram:
ServerMediator introduces itself to colleagues via addMediator() method. GameServer1, GameServer2, GameServer3 are our colleague objects. They inform the ServerMediator via echo() method when a player joins to or leaves from the game node. Contents of the echo method can differ but in our case it is mostly related to the server's current load.

ServerMediator calculates the most eligible network (it can be realized with a sorting algorithm) and stores the data within self.
After the calculation, MatchmakingServer uses the mediator object to obtain the data that which server currently has the minimum network load.

d) Introduction of code and relation between the pattern

Purpose: This subsection introduces the source code of implementation, and describes the relation between code and design pattern.

In the implementation, we have five classes directly related to the Mediator design pattern.

First, an interface for Mediator:

```
public interface Mediator {
   public int getMostEligibleServerId();
}
```

In the Mediator interface there is only one method for concrete classes to implement, which is going to be used for the Matchmaking server later.

Second, an interface for our colleagues (GameServers are colleagues in our case):

```
public interface Colleague {
   public void echo();
   public int getServerCode();
   public int getOccupancyRatio();
}
```

Third, concrete implementation of Mediator:

```
import java.util.ArrayList;
public class ServerMediator implements Mediator {
   private ArrayList<Colleague> servers;

   public ServerMediator() {
       servers = new ArrayList<>();
   }

   public void collectData(Colleague gameServer) {
       servers.add(gameServer);
   }


   @Override
```

```
    public int getMostEligibleServerId() {
        int min = 1000000;
        int minID = 0;
        for(Colleague c: servers) {
            if(c.getOccupancyRatio() < min) {
                minID = c.getServerCode();
                min = c.getOccupancyRatio();
            }
        }
        return minID;
    }
}
```

ServerMediator gathers the information from GameServers via the collectData() method. It appends the servers into an arraylist called servers. Then in the getMostEligibleServerId() method.
It compares the loads between servers and decides which one is the most eligible one. Finally it returns the server's id.

Fourth, concrete implementation of Colleague:

```
public class GameServer implements Colleague{
    private int serverCode;
    private int occupancyRatio;
    private ServerMediator mediator;

    public GameServer(int serverCode, ServerMediator serverMediator ) {
        this.occupancyRatio = 0;
        this.serverCode = serverCode;
        addMediator(serverMediator);
    }


    public void addMediator(ServerMediator mediator){
        this.mediator = mediator;
    }

    public int getServerCode() {
        return this.serverCode;
    }

    public int getOccupancyRatio() {
        return this.occupancyRatio;
    }

    public void setOccupancyRatio(int newOccupancy) {
        this.occupancyRatio = newOccupancy;
    }
```

```
    @Override
    public void echo() {
        mediator.collectData(this);
    }

    public void addPlayer() {
        this.occupancyRatio++;
        this.echo();
    }

    public void dropPlayer(){
        this.occupancyRatio--;
        this.echo();
    }
}
```

In the GameServer class, constructor has two parameters one is server code, (it is required for distinguishability) The other parameter is the mediator object that is going to be responsible for the communication between the GameServers.
Getter and setter methods are not going to be detailed because it is pretty self explanatory.

However, echo(), addPlayer() and dropPlayer() methods are quite critical for the design pattern. When an internal change of state happens in the GameServer(adding or dropping a player), the server updates the occupancy ratio of itself and calls the echo() method. This method is used for informing the Mediator object.
In the echo method, the mediator object collects the data of the object who calls itself.

And finally MatchmakingServer class, who communicates with the Mediator object and obtains the most eligible server data.

```
public class MatchmakingServer {
   private ServerMediator mediator;

   public MatchmakingServer(ServerMediator mediator) {
       this.mediator = mediator;
   }
```

```
    public int getMostEligibleServer() {
        int mostEligible = mediator.getMostEligibleServerId();
        System.out.println("***Most eligible server is currently the server with
the id: " + mostEligible);
        System.out.println("***New player will be redirected Server " +
mostEligible);
        return mostEligible;
    }
}
```

e) Test Scenarios

Purpose: This subsection introduces a test scenario to examine the effects of the design pattern.

First, we created three game servers, one mediator object and one matchmaking server.

We have set the initial occupancies with the numbers given below:
GameServer1: 30
GameServer2: 32
GamerServer3: 35

Then, we simulated that ten players wanted to join the game. Our expectation was with the help of the implemented design pattern, the matchmaking server will choose the server who has least network load. And at the end of the test all servers should have close values in terms of occupancy level.

In the next subsection you can find the test results.

f) Test Results

Purpose: This subsection introduces test results of the given scenario.

```
PLAYER 1 wants to join the game.
Current player number in Server 1: 30
Current player number in Server 2: 32
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 1
***New player will be redirected Server 1
-----------------------------------------------------------------

PLAYER 2 wants to join the game.
Current player number in Server 1: 31
Current player number in Server 2: 32
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 1
***New player will be redirected Server 1
-----------------------------------------------------------------

PLAYER 3 wants to join the game.
Current player number in Server 1: 32
Current player number in Server 2: 32
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 1
***New player will be redirected Server 1
-----------------------------------------------------------------

PLAYER 4 wants to join the game.
Current player number in Server 1: 33
Current player number in Server 2: 32
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 2
***New player will be redirected Server 2
-----------------------------------------------------------------

PLAYER 5 wants to join the game.
Current player number in Server 1: 33
Current player number in Server 2: 33
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 1
***New player will be redirected Server 1
-----------------------------------------------------------------

PLAYER 6 wants to join the game.
Current player number in Server 1: 34
Current player number in Server 2: 33
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 2
***New player will be redirected Server 2
-----------------------------------------------------------------

PLAYER 7 wants to join the game.
Current player number in Server 1: 34
Current player number in Server 2: 34
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 1
```

```
***New player will be redirected Server 1
----------------------------------------------------------------

PLAYER 8 wants to join the game.
Current player number in Server 1: 35
Current player number in Server 2: 34
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 2
***New player will be redirected Server 2
----------------------------------------------------------------

PLAYER 9 wants to join the game.
Current player number in Server 1: 35
Current player number in Server 2: 35
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 1
***New player will be redirected Server 1
----------------------------------------------------------------

PLAYER 10 wants to join the game.
Current player number in Server 1: 36
Current player number in Server 2: 35
Current player number in Server 3: 35
***Most eligible server is currently the server with the id: 2
***New player will be redirected Server 2
----------------------------------------------------------------


Process finished with exit code 0
```

As you can see, test results were successful.Mediator object evaluates the most eligible server after each player joins the game server and 10 players are distributed to the servers uniformly.

3) Players should not face network-connection quality related issues such as ping, lag vs.

a) What was the problem, what did we choose?

Purpose: This subsection reminds what was the flexibility requirement, and which design pattern chosen to achieve it.

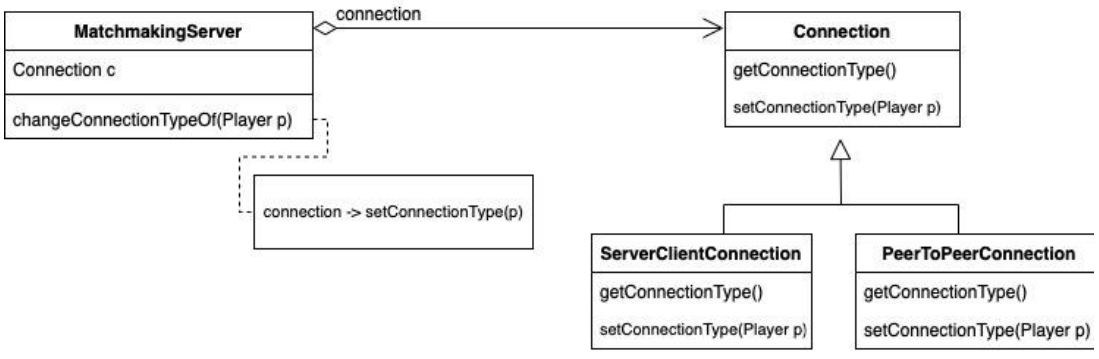
Earlier, we mentioned that players who are located far from game servers can be redirected to a peer to peer pooling system if there are enough peers.

Let's think of a scenario where a player who is relatively located far from the game servers joins the game and at some points two more players join. Let's say these three players are far from our servers but they are close to each other. Then we should drop them from the client-server model and establish a peer to peer connection between them or vice versa. (If two of the players leave the game, we should bring the other player back to our servers.)

This process should
happen seamlessly, we must be able to make these transitions during the game, therefore we choose to implement Bridge pattern to achieve this flexibility requirement because it is compulsory to make runtime implementation changes in this problem.

b) Static Class Structure of the Pattern

Purpose: This subsection introduces a UML diagram of the design pattern.



This UML diagram represents the class structure of the Bridge design pattern. In this case MatchmakingServer is our Abstraction. Connection class is an interface which corresponds as an Implementor.

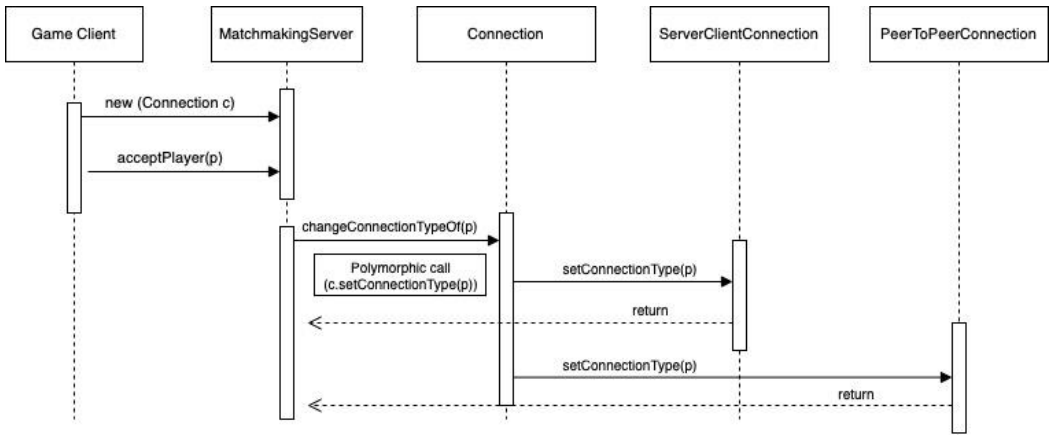
ServerClientConnection and PeerToPeerConnection classes are our concrete implementors.

MatchmakingServer has a reference to a Connection object which can alter during runtime. When a player joins the MatchmakingServer some calculations are made between player and game server's location and as a result of it, connection type of player is assigned dynamically.

This process starts within the changeConnectionTypeOf() method in MatchmakingServer, this method makes a polymorphic call and according to the calculations the connection object of the player is determined. With this flexibility, players' connection type can change seamlessly between server-client and peer-to-peer.

c) Dynamic Class Structure of the Pattern

Purpose: This subsection introduces a sequence diagram of the design pattern.



In this sequence diagram you can see the runtime interactions between our classes.

Firstly, a player joins the MatchmakingServer via acceptPlayer() method. Then MatchmakingServer compares if the player is far from the game server, if there are any other players who are close to the player who joins currently, then it produces an outcome whether the

player should connect to the game through server-client or peer-to-peer connection type.

Then it makes a polymorphic call and sets the connection type of the player. Note that if there are players close to the current player, they are grouped and redirected to the peer to peer connection. In such cases besides setting the current player's connection type as peer to peer also retroactive changes should happen and other players are dropped from server client connection for redirection.

d) Introduction of code and relation between the pattern

Purpose: This subsection introduces the source code of implementation, and describes the relation between code and design pattern.

In the implementation, there are four classes directly related to the Bridge design pattern. These are MatchmakingServer, Connection, ServerClientConnection, PeerToPeerConnection classes.
However, to make this implementation meaningful, we also added the Player class. The source code of this class will be provided in this part also but it is not going to be detailed.

First, Connection class which represents the Implementor:

```
public class Connection {
   public String getConnectionType() {
       return "not obtained";
   }
   public void setConnectionType(Player p) {
   }
}
```

Second, ServerClientConnection class which is concrete implementor:

```
public class ServerClientConnection extends Connection{
   public String getConnectionType() {
       return "Server-Client";
```

```
    }

    public void setConnectionType(Player p){
        p.setConnectionType(this);
    }
}
```

Third, PeerToPeerConnection class which is another concrete implementor:

```
public class PeerToPeerConnection extends Connection{
    public String getConnectionType() {
        return "Peer to Peer";
    }

    public void setConnectionType(Player p) {
        p.setConnectionType(this);
    }
}
```

And finally, MatchmakingServer which is abstraction:
(Since it is a relatively long code, it will be explained partly)

```
import java.util.ArrayList;
public class MatchmakingServer {
    Connection connection;
    ArrayList<Player> players;
    ArrayList<Player> playersToChangeList;
    private int latitudeOfServer;
    private int longitudeOfServer;

    public MatchmakingServer(int latitude, int longitude){
        connection = new Connection();
        players = new ArrayList<>();
        playersToChangeList = new ArrayList<>();
        this.latitudeOfServer = latitude;
        this.longitudeOfServer = longitude;
    }
```

In the constructor, a connection object is instantiated, there are two lists one for keep track of players who currently joined the game(players), one for keep track of players which are going to be redirected (playersToChangeList). Finally, the latitude and the longitude of the server is set in the constructor.

```
    public void changeImplementation(Connection c1){
        connection = c1;
    }
```

This method is used for changing the type of connection object.

```
    public void changeConnectionTypeOf(Player p){
        connection.setConnectionType(p);
    }
```

ChangeConnectionTypeOf method is used for setting the correct connection type for Player p.

```
    public boolean acceptPlayer(Player p){
        if(players.size() == 0) {
            players.add(p);
            return false;
        }
        else {
            boolean flag = false;
            players.add(p);
            if(!isPlayerFarFromServer(p))
                return false;
            else { // player is far from server
                for(int i = 0; i < players.size() - 1; i++) {
                    if(isPlayersCloseEachOther(players.get(i), p)){ // and there
are other players close to player p,
                        playersToChangeList.add(players.get(i)); // choose peer to
peer for them
                        playersToChangeList.add(p);
                        flag = true;
                    }
                }
                return flag;
            }
        }
    }
```

The acceptPlayer method accepts the player and appends them to the player list.

```
    public boolean isPlayerFarFromServer(Player p) {
        double first = (p.getLatitude() - this.latitudeOfServer) *
(p.getLatitude() - this.latitudeOfServer);
```

```
        double second = (p.getLatitude() - this.longitudeOfServer) *
(p.getLongitude() - this.longitudeOfServer);
         double ans = Math.sqrt(first + second);

        System.out.println("Player with id: " + p.getId() + "'s distance to server
is: " + ans);
        return ans > 50;
    }

    public boolean isPlayersCloseEachOther(Player p1, Player p2) {
        double first = (p1.getLatitude() - p2.getLatitude()) * (p1.getLatitude() -
p2.getLatitude());
        double second = (p1.getLongitude() - p2.getLongitude()) *
(p1.getLongitude() - p2.getLongitude());
        double ans = Math.sqrt(first + second);


        System.out.println("Distance between player with id " + p1.getId() + " and
player with id: " + p2.getId() + " is " + ans);
        return ans < 50;
    }
```

These two methods (isPlayerFarFromServer, isPlayersCloseEachOther) calculate the distances between the players and each player's distance to the game server. And they produces outcomes such as:

- Player is far from server
- Player is not far from server
- Player 1 and Player 2 are close to each other
- Player 1 and Player 2 are not close to each other

```
    public ArrayList<Player> getPlayersToChangeList(){
        return this.playersToChangeList;
    }

    public void flush(){
        this.playersToChangeList.clear();
    }
}
```

getPlayersToChangeList() method returns the list containing players which should be redirected and flush() method is used for clearing this list.

e) Test Scenario

Purpose: This subsection introduces a test scenario to examine the effects of the design pattern.

As a test scenario we created the classes mentioned before, and simulated that 4 players joined consecutively to the game server.
In order to observe that the connection object changes dynamically, we determined the positions of the players accordingly.

Our expectations were, if a player is close to the server then the player's connection type should be server-client. If there are at least two players close to each other then both their connection type should change to peer-to-peer.

You may see the results of the test scenario in the next subsection.

f) Test Results

Purpose: This subsection introduces test results of the given scenario.

We used the formula ‘distance between two points’ while calculating if the player is far from or close to the server .
If the distance between two points is below 50, we consider that two points are close to each other, otherwise they are far from each other.

And here are the rest results:

```
---------------------------------
Player <1>
Latitude: 10
Longitude: 20
Connection type: Server-Client
Internet Speed: 16
---------------------------------

Player with id: 2's distance to server is: 12.24744871391589
---------------------------------
Player <2>
Latitude: 20
Longitude: 25
Connection type: Server-Client
Internet Speed: 58
```

```
---------------------------------

Player with id: 3's distance to server is: 70.71067811865476
Distance between player with id 1 and player with id: 3 is 92.19544457292888
Distance between player with id 2 and player with id: 3 is 81.39410298049853
---------------------------------
Player <3>
Latitude: 80
Longitude: 80
Connection type: Server-Client
Internet Speed: 80
---------------------------------

Player with id: 4's distance to server is: 84.8528137423857
Distance between player with id 1 and player with id: 4 is 106.30145812734649
Distance between player with id 2 and player with id: 4 is 95.524865872714
Distance between player with id 3 and player with id: 4 is 14.142135623730951
******** CHANGED CONNECTION TYPE OF PLAYER WITH ID: 3 to Peer to Peer*********



---------------------------------
Player <3>
Latitude: 80
Longitude: 80
Connection type: Peer to Peer
Internet Speed: 80
---------------------------------

******** CHANGED CONNECTION TYPE OF PLAYER WITH ID: 4 to Peer to Peer*********
---------------------------------
Player <4>
Latitude: 90
Longitude: 90
Connection type: Peer to Peer
Internet Speed: 24
---------------------------------

Process finished with exit code 0
```

As you can see, the test produced a successful outcome. Connection type of the first player chosen server-client because at the moment there were no other players to be possible peers.
Then connection type of Player2 and Player3 chosen server-client based on their distances between to the server.

When Player4 wanted to join the game, the distance of the player calculated, according to the result, the player was far from the server. Then distance between Player4 and Player1,Player2,Player3 calculated.

Since Player4 and Player3 were close to each other, the connection type of Player4 has been set to peer-to-peer and Player3 is dropped from server and changed to peer-to-peer.

4) The process of finding which node a player is in should be efficient (for dropping, or redirecting the player)

a) What was the problem, what did we choose?

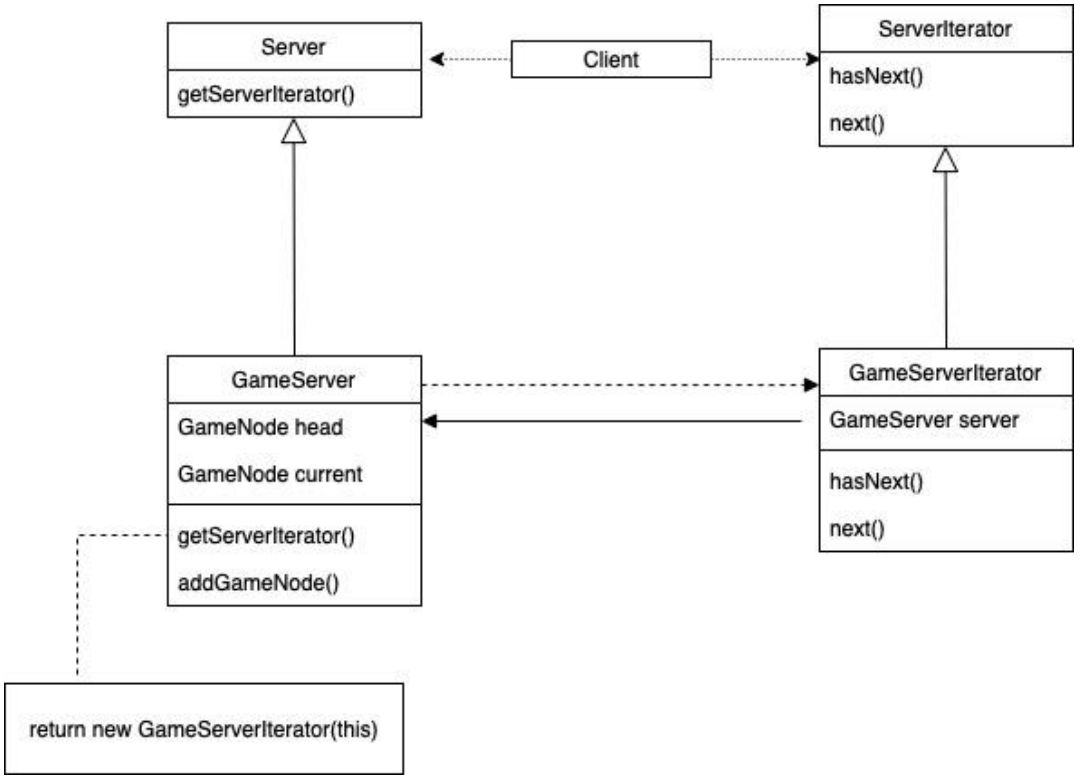
Purpose: This subsection reminds what was the flexibility requirement, and which design pattern chosen to achieve it.

Earlier, we mentioned that when a player decides to exit the game, or if a player is going to be redirected to a peer-to-peer connection, firstly he/she should be located in the server.
Servers consist of game nodes which players are attached to. It is a strong probability that in a game session there will be most likely hundreds of game nodes in a single server. Therefore we should locate the player in these game nodes effectively.

We chose the Iterator design pattern for this requirement because it allows us to traverse through game nodes without exposing its internal representation.

b) Static Class Structure of the Pattern

Purpose: This subsection introduces a UML diagram of the design pattern.



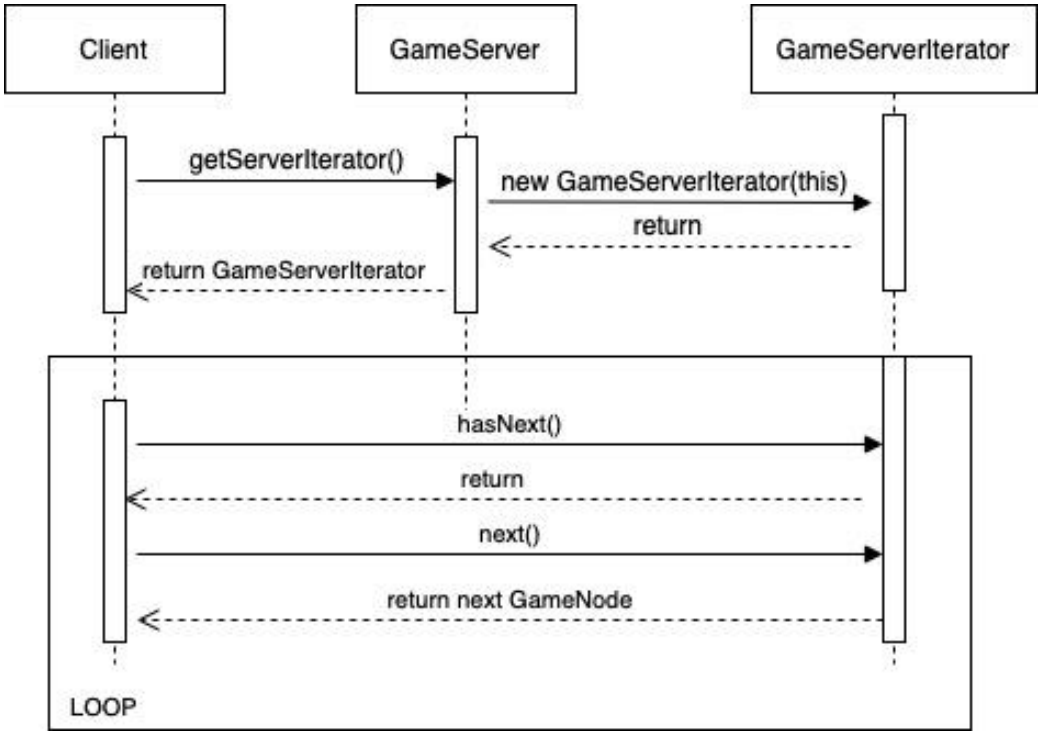
This UML diagram represents the class structure of the Iterator design pattern. We have a Server interface as an Aggregate.GameServer class implements the GameServer and it is the concrete form of the aggregate. It has a method named getServerIterator() that allows clients to create a server iterator.

You may notice there is a GameNode reference in the GameServer. In the implementation, game nodes are stored in a linked structure in the server. Therefore it was essential to have those references to keep track of where the new node will be placed.

ServerIterator class is an interface for Iterator. It has the fundamental methods of Iterator which are hasNext() and next(). Finally the GameServerIterator class is the concrete implementation of ServerIterator and it has a reference to the GameServer class.

c) Dynamic Structure of the Pattern

Purpose: This subsection introduces a sequence diagram of the design pattern.



In the above UML sequence diagram, you can see the runtime interactions between client, iterable and iterator classes.

When a client wishes to traverse the iterable collection which is GameServer in our case, it fetches an iterator via getServerIterator() method. GameServer returns an GameServerIterator object which has a reference of itself as a parameter.

After this process, the client can traverse the GameServer with the GameServerIterator via a while loop. While there is a next GameNode in the GameServer it is returned until there are no more. And the client can look for the player which is wanted, in each GameNode object.

d) Introduction of code and relation between the pattern

Purpose: This subsection introduces the source code of implementation, and describes the relation between code and design pattern.

There are four classes directly related to the Iterator design pattern. Additionally there are two more classes needed to make this implementation meaningful. These classes are GameNode and Player classes. Since they are not directly involved in the design pattern the source codes of these classes will not be provided in this section. It is enough to know that GameNode objects hold the Player objects in themselves. And GameServer consists of the GameNode objects. Basically traversal of a server is done by looking inside the nodes and searching the player inside the nodes.

Now, the classes directly involved will be detailed in the Iterator pattern.

First, Server interface:

```
public interface Server {
   ServerIterator getServerIterator();
}
```

As you can see it is pretty simple, there is only method named getServerIterator() to be overridden for the concrete implementations of Server interface.

Second, GameServer class which is an implementation of Server interface:

```
public class GameServer implements Server{
   private int size = 0;
   private GameNode cursor;
   private GameNode head;
   private GameNode curr;
```

```
private GameNode addFirst() {
    GameNode node = new GameNode();
    head = node;
    curr = node;
    cursor = node;
    size++;
    return node;
}
```

If the game server is empty, the first game node is created and the address of it returned.

```
public GameNode addGameNode() {
    if(this.size == 0)
        return addFirst();
    else {
        GameNode node = new GameNode();
        curr.setNext(node);
        curr = node;
        size++;
        return node;
    }
}
```

The addGameNode method is used for creating a new game node in the game server. Since the collection of the game nodes is a linked structure, in this method it is essential to link nodes as ‘next’ and return the address of it.

```
public GameNode first() {
    return this.head;
}
```

The first method returns the first game node.

```
public GameNode getNextNode() {
    GameNode temp = cursor;
    cursor = cursor.getNext();
    return temp;
}
```

The getNextNode() returns the next game node of the cursor object. Cursor is set to the head of the game nodes. And every next() call cursor moves to the next game node object.

```
    public int nodeNumber() {
        return this.size;
    }



    @Override
    public ServerIterator getServerIterator() {
        return new GameServerIterator(this);
    }
}
```

And getServerIterator() method returns a GameServerIterator with a self reference passed as an argument.

Third, ServerIterator class which is an interface:

```
public interface ServerIterator {
    public boolean hasNext();
    public Object next();
}
```

This interface only has two methods, which are going to be overridden by the concrete iterator class which implements ServerIterator.

Finally, GameServerIterator class, which is concrete implementation of ServerIterator:

```
public class GameServerIterator implements ServerIterator {
    private GameServer gameServer;
    private int i = 0;

    public GameServerIterator(GameServer server) {
        this.gameServer = server;
    }

    @Override
    public boolean hasNext() {
        return this.i < gameServer.nodeNumber();
    }

    @Override
    public GameNode next() {
```

```
            this.i++;
            return gameServer.getNextNode();
        }
    }
```

This class has a reference to the GameServer which is going to be iterated, and the implementation of hasNext() and next() method. hasNext() method checks if there are nodes in the game server and next() method returns the next game node in the game server.

e) Test Scenario

Purpose: This subsection introduces a test scenario to examine the effects of the design pattern.

As a test scenario, we created a game server and added four game nodes to the server. After that 400 players are created and distributed to each game node as a group of 100s. Then we created an iterator with the getServerIterator method inside the GameServer class and traversed the server with that iterator.

```
GameServer server = new GameServer(); // Create server
GameNode n1 = server.addGameNode(); // Create game node and add it to the
server
n1.setName("Node1");
for(int i = 0; i < 100; i++) {
   Player player = new Player(i);
   n1.acceptPlayer(player);
}
```

The above process is repeated four times for GameNode2, GameNode3, GameNode4 but since the code is the same only the first one is provided.

```
ServerIterator iterator = server.getServerIterator(); // create an
iterator

while(iterator.hasNext()){ // and iterate through server and check nodes
  GameNode node = (GameNode) iterator.next();
  if(node.hasPlayerWithId(id)) {
      System.out.println("Player with id” + id + found in: " +
node.getName());
```

```
            break;
        }
        else {
            System.out.println("Player could not found in the " +
node.getName());
        }
}
```

We repeated this four times and at each iteration, we looked for a player in a different node than other iterations. You may see the test results in the next subsection.

f) Test Results

Purpose: This subsection introduces test results of the given scenario.

```
First iteration:
-----------------------------
Traversing the game nodes in the server!
Player with id 32 found in: Node1
-----------------------------
Second iteration:
-----------------------------
Traversing the game nodes in the server!
Player could not found in the Node1
Player with id 159 found in: Node2
-----------------------------
Third iteration:
-----------------------------
Traversing the game nodes in the server!
Player could not found in the Node1
Player could not found in the Node2
Player could not found in the Node3
Player with id 347 found in: Node4
-----------------------------
Fourth iteration:
-----------------------------
Traversing the game nodes in the server!
Player could not found in the Node1
Player could not found in the Node2
Player with id 291 found in: Node3
-----------------------------
```

As you can see, tests produced a successful output and we find the desired player in the game nodes by traversing the server with the help of an iterator.

5) When expanding or removing game nodes players should be relocate between nodes efficiently

a) What was the problem, what did we choose?

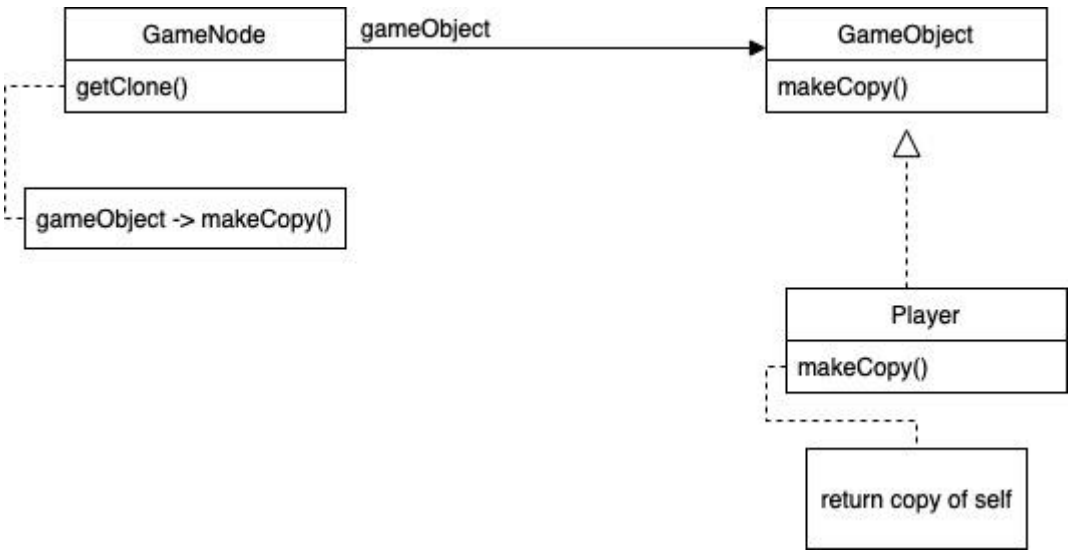
Purpose: This subsection reminds what was the flexibility requirement, and which design pattern chosen to achieve it.

As we told before, when a player joins the game, attached to the game nodes which are in game servers. If a game session becomes load heavy, a new node will be created. Similarly if players exit the game and node becomes too light in terms of load, it will be removed.

If a node is almost empty, before removing the players inside that node we should move players to the other nodes. To accomplish this we chose the Prototype pattern to clone the player objects inside the node that is going to be removed.

b) Static Class Structure of the Pattern

Purpose: This subsection introduces a UML diagram of the design pattern.

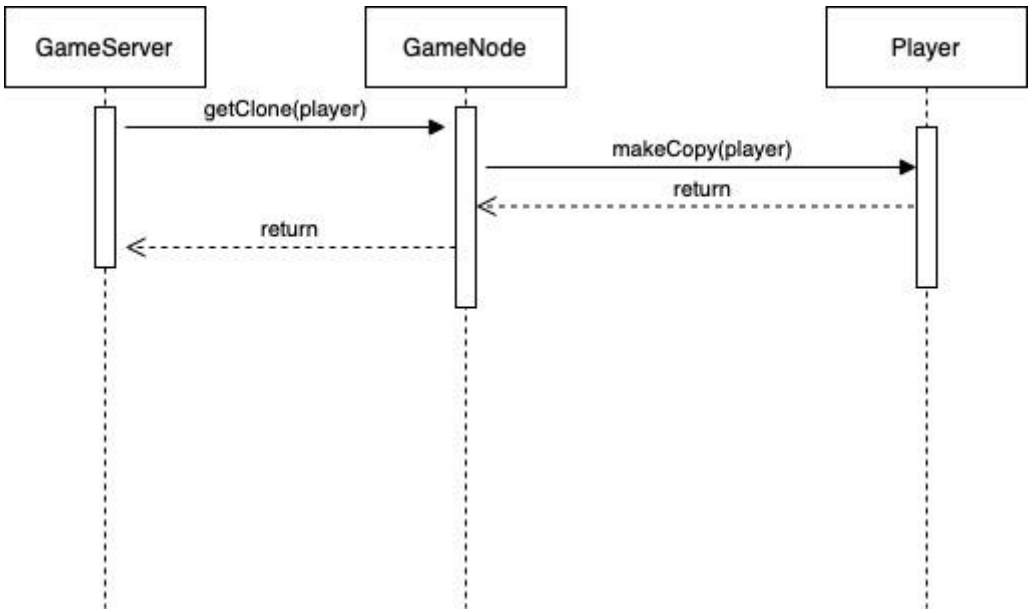


This UML diagram represents the class structure of the Prototype Design Pattern. The GameNode class has a method that is used for cloning. GameObject is an interface which has a single makeCopy() method for concrete implementations to override.

Player class is the concrete implementation of GameObject and it returns its clone via makeCopy() method.

c) Dynamic Class Structure of the Pattern

Purpose: This subsection introduces a sequence diagram of the design pattern



In the above UML sequence diagram you can see the runtime interactions between objects.

GameServer is the mechanism that decides to clone a player object because the game server handles node management such as creating or removing. When the game server makes a getClone(player) call to GameNode, GameNode forwards that request to the Player.

And player objects creates a clone of itself and returns it to the GameNode and GameNode forwards it to the GameServer. After collecting all players in the GameNode, GameServer can delete GameNode safely.

d) Introduction of code and relation between the pattern

Purpose: This subsection introduces the source code of implementation, and describes the relation between code and design pattern.

In the implementation there are three classes directly related to the design pattern.

First, the GameObject interface which has one single method makeCopy()

```
public interface GameObject extends Cloneable {
   public Object makeCopy();
}
```

Second, Player class which is a concrete implementation of the GameObject class

```
public class Player implements GameObject {
   private int id;

   public Player(int id){
       this.id = id;
   }

   public int getId(){
       return this.id;
   }

   public Player makeCopy(){
       Player playerObj = null;
       try {
           playerObj = (Player) super.clone();
       } catch (CloneNotSupportedException e) {
           e.printStackTrace();
```

```
        }
        return playerObj;
    }


public String toString(){
        return "Player with id: " + this.id;
    }
}
```

Inside the player class only method related to design pattern is the makeCopy() method that returns a clone of itself.

Third, GameNode class:

```
import java.util.ArrayList;
public class GameNode {
    private ArrayList<Player> players;
    private String name;

    public GameNode(String name){
        players = new ArrayList<>();
        this.name = name;
    }

    public void addPlayer(Player p) {
        players.add(p);
    }

    public ArrayList<Player> getPlayers() {
        return this.players;
    }

    public String getName() {
        return this.name;
    }

    public void deletePlayer(){
        if(players.size() > 0)
            players.remove(players.size() -1);
        else
            System.out.println("No players in the node!");
    }

    public GameObject getClone(GameObject obj) {
        return (GameObject) obj.makeCopy();
    }
}
```

Inside the GameNode class addPlayer(player) method is used for adding a player to the node. Similarly deletePlayer() method is used for deleting the player from that node.
And the getClone() method is the crucial method for the design pattern. It accepts a parameter and returns its clone.

e) Test Scenario

Purpose: This subsection introduces a test scenario to examine the effects of the design pattern.

As a test scenario we created a game server and added a game node. Then created 80 players and inserted players in the first game node. After that, we created another game node with 100 players and inserted players into the second game node.

Then we deleted 90 players from the second game node and it became very lightweight with only 10 players inside itself.
Finally, we cloned that 10 players and inserted them into the first game node and we deleted the second game node.

Below, you can see the source code we used to make these operations.

```
GameServer server = new GameServer();
GameNode node = new GameNode("Node1");
server.addGameNode(node);
for(int i = 0; i < 80; i++){
   Player player = new Player(i);
   node.addPlayer(player);
}

GameNode node2 = new GameNode("Node2");
for(int i = 80; i < 180; i++){
   Player player = new Player(i);
   node2.addPlayer(player);
}
server.addGameNode(node2);

for(int i = 0; i < 90; i++) {
   node2.deletePlayer();
```

```
}


for(int i = 0; i < node2.getPlayers().size(); i++) {
   Player p = (Player) node2.getClone(node2.getPlayers().get(i));
   System.out.println("Player with id " + i + " is cloned!");
   node.addPlayer(p);
   System.out.println("Player with id " + i + " redirected to Node1");
   System.out.println("--------------------");
}
server.deleteGameNode(node2);
```

f) Test Results

Purpose: This subsection introduces test results of the given scenario.

```
You may see the test results below.

There are 80 players in the Node1
--------------------
There are 100 players in the Node1
--------------------
90 players left the Node2
Current player number in the Node2 is: 10
Node2 is going to be removed from the server!
--------------------
Cloning process has started.
Player with id 0 is cloned!
Player with id 0 redirected to Node1
--------------------
Player with id 1 is cloned!
Player with id 1 redirected to Node1
--------------------
Player with id 2 is cloned!
Player with id 2 redirected to Node1
--------------------
Player with id 3 is cloned!
Player with id 3 redirected to Node1
--------------------
Player with id 4 is cloned!
Player with id 4 redirected to Node1
--------------------
Player with id 5 is cloned!
Player with id 5 redirected to Node1
--------------------
Player with id 6 is cloned!
Player with id 6 redirected to Node1
```

```
--------------------
Player with id 7 is cloned!
Player with id 7 redirected to Node1
--------------------
Player with id 8 is cloned!
Player with id 8 redirected to Node1
--------------------
Player with id 9 is cloned!
Player with id 9 redirected to Node1
--------------------
There are no players in the Node2
Game node Node2 is removed from server
Node2 is removed from the server!

Process finished with exit code 0
```

CONCLUSION

What I Have Learned?

Firstly, I have learned how to read a software design architecture and detect the problems or parts available for further development in it. Second, I think I have a good understanding of how the backend (networks-servers-databases) portion of a game operates during the game.
Third, I believe I have learned the fundamentals of design patterns that I implemented for flexibility requirements. And finally I believe I have learned how to write a suitable report in a professional way.

Challenges

Despite it being a joyful process to create this report, there were several points that led me to despair.

The first challenge was, after the work sharing, the Platform part of the architecture was my share. It was a relatively small and the most abstract architecture in the original report. There were no UML diagrams or concrete class schemas at all. Therefore it was a big challenge for me to find flexibility requirements related to Platform architecture. Hence I could only find seven flexibility requirements.

The other challenge was, sometimes it was hard to adapt the design pattern to the architecture. Even though I understood the design pattern after I read the book of Gamma, for a couple of patterns it took a long time to adapt the pattern to the architecture itself.

The last challenge is related to the one before, that is adapting UML and sequence diagrams of the pattern to architecture as well. Even if I understood the diagrams, sometimes it was hard to create the correct relation between them.

Epilogue

After implementing those design patterns, I believe the Platform part of the game architecture is much more flexible. Since we simulated an expert group in a software company, our task was highly related to the real world and because of that it was a very intense process in terms of learning. I am aware that I am not fully expertised on the design patterns however I believe it was an excellent introduction and from now on it will be a smoother process for me to learn other design patterns.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1998). *Design patterns elements of Reusable Object Oriented Software*. Addison Wesley.

[2] https://refactoring.guru/design-patterns

[3] https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

[4] Verslag