Software Design Patterns

Practical Activity 1 – PR1

Nicolas D'Alessandro Calderon

Design principles, analysis and architectural patterns

Question 1

A) The analysis pattern that is being applied is the **Historical Association** pattern.

This pattern is visible trough the **TokenConsumption** class acting as an association that registers "**when"** (**timestamp**) an Employee uses an AI Agent, storing the consumed tokens. This allows us to keep a historical record of the interactions between employees and AI agents over time.

In the catalog of the learning material, the application of this pattern is described as **"Adding a dimension to the association (converting the n-ary association into n+1-ary that represents time."** So, we may say that this model is a direct implementation of this pattern description but with a slight variation, given that instead of using a pure ternary association, it is using an associative class with a temporal attribute (**timestamp**) as part of its identifier.

B) An example of an analysis pattern that can be applied here is the **Quantity Pattern** to the **tokenPrice** attribute within the **AIAgent** class. According to the learning material, "*the representation of a quantity via a numerical value is a poor solution.*" So, the application of the Quantity Pattern helps to represent these quantities with units, like money, distance, etc. In the proposed model, **tokenPrice** may have different units of measures, so this pattern will allow us to encapsulate the value and the unit, being able to extend the system without changing the structure.

Applying this pattern in our example will provide various advantages, such as explicitly modeling the unit of cost and improving clarity, it will also facilitate monetary conversions allowing different monetary units for different agents. All this will result in a better semantic of the model since it is clearly represented that this attribute is a monetary value.

NOTE FOR THE PROFESSOR: To answer the next questions, and consider the confusion raised in the CAT1 regarding the DRY principle violation, I will give my answers using the supposed context where it is mentioned that **the system may evolve and incorporate new categories**. However, the usage of enums for representing categories is a very common practice and accepted (or at least I have seen this a lot in production code). Also, regarding question E the presented diagram is not showing any explicit hierarchy for the AIAgent.
Furthermore, searching in Google "does java enumns violate OCP or LSP principles?" takes to this Stack Overflow discussion where some people **declare that the usage of enums inherently violates the OCP principle** and some other even discuss the usage of the "word" used for explaining this behaviour. I didn't find any specific posture about this in the official learning material, so I wrote my answers based on these assumptions.

C) In the learning material states that according to SRP "*A class should have responsibility for only one part of the functionality provided by the software*".
In the given model, the **Employee** class seems to have at least two different responsibilities, store and manage the basic information of the employee (email and role) and determine the type of employee (developer or community manager). However, the provided diagram does not show any explicit method or associated logic that indicates multiple responsibilities, the **role** attribute is only storing a value from the **Role** enumeration, which does not by itself violate SRP.

In case this second responsibility is confirmed, considering that the statement mentions that this logic may be expanded in the future, any change in roles structure, or how billing is done for the monthly consumption calculation (which is unlikely but maybe different for each role such as the employees income calculation in the first CAT), this should be separated from the basic responsibility already mentioned, especially for compliance with the SRP principle given that this logic requires classification or categorization.

D) The learning material mentioned that the OCP principle stablish that "*A software entity (class, module, function, etc.) must be open for extension but closed for modification*." Also adding that "*One way to comply with this principle is to create abstractions around the aspects that we anticipate needing to change, so that a stable interface can be created with respect to the changes.*" This means that the usage of the enumerated **AICategory** violates the open-close principle.

Using the closed enumerated means that every time that we want to add a new category it will be necessary to modify the enumerated **AICategory** code, violating the OCP principle and promoting the possibility of runtime errors and adding difficulty in the code maintenance.

*NOTE: As mentioned before, using an enumeration in this case may seems comfortable but difficult due to the continuous grow of generative AI categories. So, maybe an option is to create a class that represent the types as objects, so new types may be created dynamically, for example, from the database.

E) The learning material mentioned that the LSP principle establish that "*the Instances of a superclass must be replaceable by instances of its subclasses without affecting the correctness of the program*". **The current model** has not explicit hierarchies for the different types of AI agents, so **it is not explicitly violating the LSP principle**. However, if new types of agents with specific behavior are implemented as it is mentioned, the code will probably need to use conditionals based on the category and not polymorphism.
If this happens, the **approach will violate the LSP principle,** since the behavior will change based on the agent type, instead of allowing each class to implement the specific behavior required.

To "apply" the LSP principle, this design should evolve to a class hierarchy where **AIAgent** is the base class and the different types of agents are subclasses, each with its specific behavior.

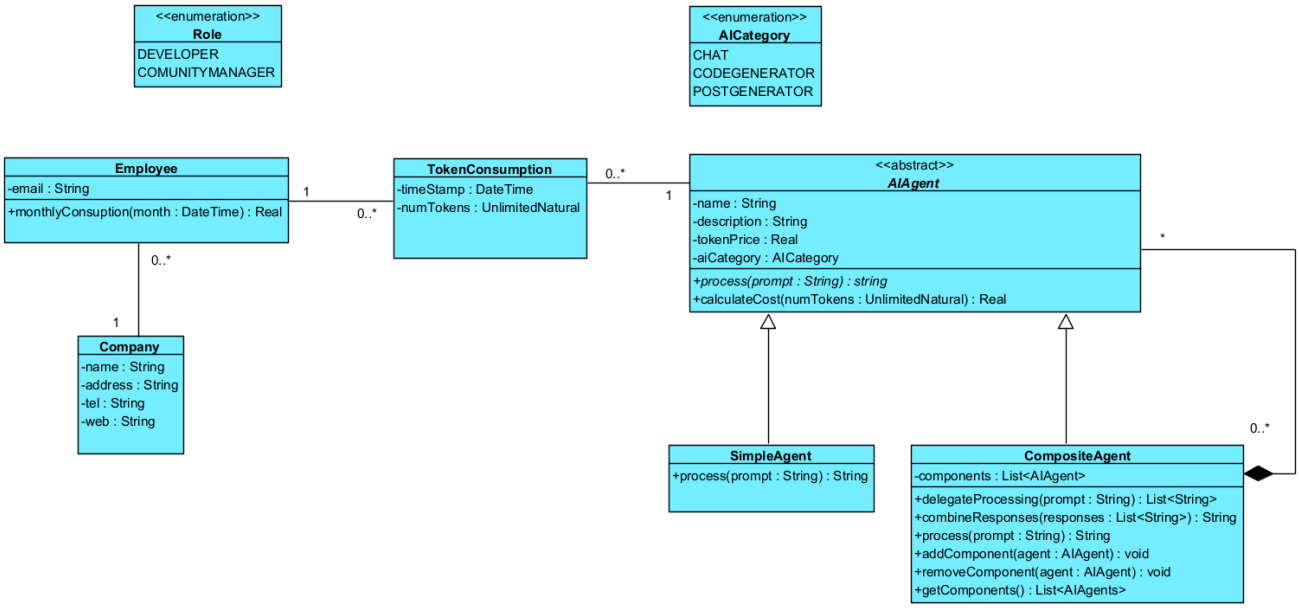
Question 2

A) To address the proposed situation, I will use the **Composite Object Pattern**.

According to the learning material, this pattern is used when "*We want to represent hierarchical structures, where an object can be composed of other objects of the same type*." Since in our case a composite AI agent could contain many AI agents whether simple or also composite, this pattern seems appropriate as it follows this exact structure description.

Applying this pattern will allow us to treat the simple agents as individual objects and the composite agents uniformly. It will also facilitate the creation of hierarchical structures using a tree form, where the composite agents may contain simple or composite agents as we explained before. Finally, we will be able to add new types of components without modifying the existent code, meeting the OCP principle.

B) Adding only this **Composite Object Pattern** approach to the base diagram of the statement, a possible UML diagram could be:



Integrity Constrains:

- A **Company** is identified by name.
- An **Employee** is identified by email.
- An **AIAGent** is identified by name.
- **TokenConsumption** is identified by **timeStamp** and the **email** of the employee.

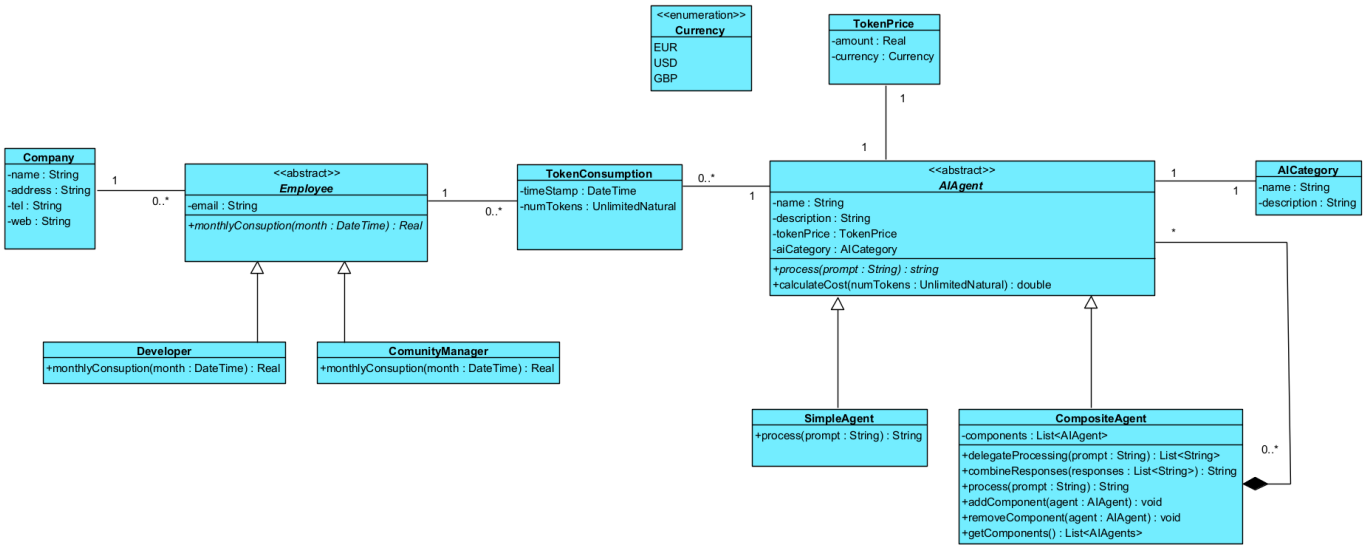
C) An operation that only makes sense for composite agents can be **delegateProcessing(prompt : String) : List<String>**. This operation is exclusive of composite agents because as it is described in the statement, only these agents can delegate the processing to multiple components in case they need it.
An example of implementation can be if this method delegates the user task by receiving the prompt and sending it to the components (whether simple or composite as well), gathering all the answers in a list. These answers are then processed by the **process** method to create the answer:

```
// Pseudocode
public abstract class AIAgent {
    public abstract String process(String prompt);
}

public class CompositeAgent extends AIAgent {
    private List<AIAgent> components;

    public CompositeAgent(List<AIAgent> components) {
        this.components = components;
    }

    public List<String> delegateProcessing(String prompt) {
        List<String> responses = new ArrayList<>();
        for (AIAgent agent : components) {
            responses.add(agent.process(prompt));
        }
        return responses;
    }

    // For example, combining the responses with a line break
    private String combineResponses(List<String> responses) {

        return String.join("\n", responses);
    }


    @Override
    public String process(String prompt) {
        List<String> componentResponses = delegateProcessing(prompt);
        return combineResponses(componentResponses);
    }
}
```

NOTE: In this model ***CompositeAgent*** *delegates the same prompt to all components just for illustrating the example with a pseudocode. However, this will be inefficient for real scenarios where each task should be delegated to the specific agent and not to all the components. Since this will probably require applying other patterns outside the scope of the practice (for example the Strategy pattern mentioned in the catalogue), I created this simple example that satisfies the requirement by showing an operation that only makes sense in composite agents.*

D) An operation that makes sense for both agents (simple and composite) can be the **process** method already included in the statement diagram
In a simple agent we will directly implement the processing such as NLP, classification, etc. and in composite agents it can orchestrate the entire process by delegating to its components, gathering the results and combining everything in a coherent answer as we see in the previous example.

*NOTE: A full UML diagram implementation, combining also some of the proposed changes or improvements proposed in question 1 could be:



Integrity Constrains:

- A **Company** is identified by name.
- An **Employee** is identified by email.
- An **AIAgent** is identified by name.
- **TokenConsumption** is identified by **timeStamp** and the **email** of the employee.
- A **TokenPrice** is identified by the **amount** and **currency.** Each **AIAgent** has exactly one **TokenPrice**. Different agents may share the same **amount** and **currency** combination, but each will have a different **TokenPrice** object.

Model Constrains:

CompositeAgent

- Is a subclass of AIAgent
- Contains a list of AIAgents
- Cannot contain itself

SimpleAgent

- Is a subclass of AIAgent
- Can't contains any other agents

All agents

- Must have a unique name

*NOTE: We keep only the **Currency** enumeration because is a finite group that is very unlikely to officially change, but even if happens, it can be easily updated. Also, we don´t need a specific behavior for each currency.

Question 3

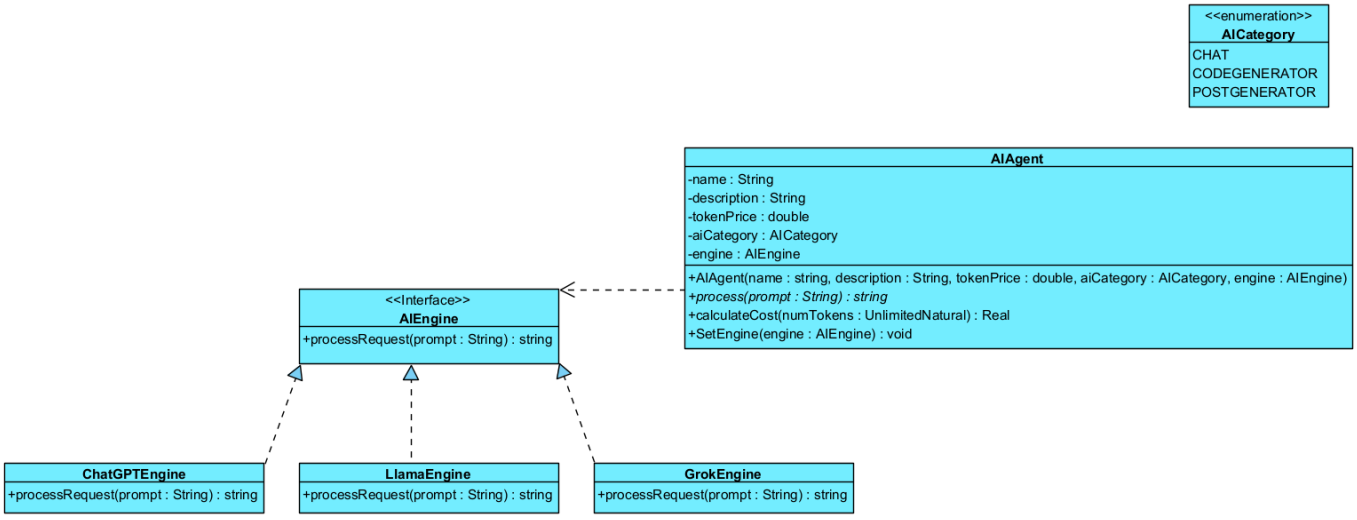
A) The following diagram shows the implementation of the **dependency injection pattern,** which allows the **AIAgent** logic to be decoupled from its concrete implementation. This architecture enables us to invoke the different AI engines through the method **processRequest** as requested in the statement.

We created the **AIEngine** interface, which defines the method **processRequest** which is also implemented by the corresponding engine classes **ChatGPTEngine**, **LlamaEngine** and **GrokEgine**.

AIAgent keeps its original attributes (**name**, **description**, **tokenprice** and **aiCategory**) but we added the attribute **engine** of type **AIEngine**, which is **externally injected through the constructor**, since we are applying the **injection by constructor** variation as requested in the statement and described in the learning material.

As in the database example of our learning material, we add the **setEngine** operation to be able to change engine at runtime.

The relationship between **AIAgent** and **AIEngine** is represented as a dependency, showing that the agent is not internally creating the engine but receiving it externally, ensuring a flexible and decoupled system.



Model Constrains:

- An **AIAgent** is identified by **name**.
- Each engine has a fix **tokenPrice**.
- We are not considering multiple model versions in this design.
- An **AIAgent** can only have one engine at a time.
- Changing engines does not affect the agent's logic.

*NOTE: To keep the consistency with the rest of the classes of the original diagram, and support the token calculations, the **tokenPrice** attribute remains in the **AIAgent.** A more realistic scenario in the future will be to assign a different token price to each engine implementation, but this model assume that the cost is being defined at agent level, and not at engine level. So, we can integrate this approach with the **TokenConsuption** and the rest of the parts of the given model domain.

B) Code Implementation:

```
// AICategory enum
public enum AICategory {
    CHAT,
    CODEGENERATOR,
    POSTGENERATOR
}

// AI Engine interface
public interface AIEngine {
    String processRequest(String prompt);
    double getTokenPrice();
}

// ChatGPT Engine implementation
public class ChatGPTEngine implements AIEngine {
    private ChatGPTAgent chatAgent;
    private String model;
    private double tokenPrice;

    public ChatGPTEngine(String model, double tokenPrice) {
        this.chatAgent = new ChatGPTAgent();
        this.model = model;
        this.tokenPrice = tokenPrice;
    }

    @Override
    public String processRequest(String prompt) {
        return chatAgent.ProcesarModeloChatGPT(prompt, model);
    }

    @Override
    public double getTokenPrice() {
        return tokenPrice;
    }
}
```

```java
// Llama Engine implementation
public class LlamaEngine implements AIEngine {
    private LlamaAgent llamaAgent;
    private String model;
    private double tokenPrice;

    public LlamaEngine(String model, double tokenPrice) {
        this.llamaAgent = new LlamaAgent();
        this.model = model;
        this.tokenPrice = tokenPrice;
    }

    @Override
    public String processRequest(String prompt) {
        return llamaAgent.InvokeModel(prompt, model);
    }

    @Override
    public double getTokenPrice() {
        return tokenPrice;
    }
}

// Grok Engine implementation
public class GrokEngine implements AIEngine {
    private GrokModel grokModel;
    private double tokenPrice;

    public GrokEngine(double tokenPrice) {
        this.grokModel = new GrokModel();
        this.tokenPrice = tokenPrice;
    }

    @Override
    public String processRequest(String prompt) {
        return grokModel.query(prompt);
    }

    @Override
    public double getTokenPrice() {
        return tokenPrice;
    }
}
```

```java
// AIAgent class modified with dependency injection by constructor and runtime engine change
public class AIAgent {
    private String name;
    private String description;
    private AICategory aiCategory;
    private AIEngine engine;

    // Constructor with dependency injection
    public AIAgent(String name, String description, AICategory aiCategory, AIEngine engine) {
        this.name = name;
        this.description = description;
        this.aiCategory = aiCategory;
        this.engine = engine;
    }

    // Method to process prompts using the injected engine
    public String process(String prompt) {
        return engine.processRequest(prompt);
    }

    // Method to change the engine at runtime
    public void setEngine(AIEngine engine) {
        this.engine = engine;
    }

    // Method to calculate the cost of the engine
    public double calculateCost(int numTokens) {
        return engine.getTokenPrice() * numTokens;
    }

}
```

C) Example where the same **AIAGent** is used with different engines without modifying the code:

```
public class Main {
    public static void main(String[] args) {

        // We create the different engine implementations
        AIEngine chatGPTEngine = new ChatGPTEngine("o4mini");
        AIEngine llamaEngine = new LlamaEngine("llama7");
        AIEngine grokEngine = new GrokEngine();

        // First agent with ChatGPT engine
        AIAgent assistant = new AIAgent(
            "AI Assistant",
            "An assistant specialized in answering questions",
            0.001,
            AICategory.CHAT,
            chatGPTEngine
        );

        // We use the agent with the ChatGPT engine
        System.out.println("Answer with ChatGPT:");
        String responseChatGPT = assistant.process("Explain the concept of
dependency injection");
        System.out.println(responseChatGPT);

        // We change the engine to Llama without modifying the AIAgent code
        assistant.setEngine(llamaEngine);

        // We use the same agent but now with the Llama engine
        System.out.println("\nAnswer with Llama:");
        String responseLlama = assistant.process("Explain the concept of
dependency injection");
        System.out.println(responseLlama);

        // We change the engine to Grok
        assistant.setEngine(grokEngine);

        // We use the same agent but now with the Grok engine
        System.out.println("\nAnswer with Grok:");
        String responseGrok = assistant.process("Explain the concept of
dependency injection");
        System.out.println(responseGrok);
```

```
        // We create a second agent with another engine...
        AIAgent codeGenerator = new AIAgent(
            "Code Generator",
            "An assistant specialized in code generation",
            0.002,
            AICategory.CODEGENERATOR,
            llamaEngine // This agent uses Llama from the start
        );

        // The system allow us to create different configurations and so on...
        System.out.println("\nCode generation with Llama:");
        String codeResponse = codeGenerator.process("Generate a function that
prints 'Hello, this is a UOC exercise!'");
        System.out.println(codeResponse);
    }
}
```

Question 4

Preliminary Analysis

I. **Identity the system architecture:** Based on the statement, the system will follow a three-layered architecture:

- **Presentation Layer:** Will implement the **MVC** pattern.
- **Domain Layer**: Will contain the business logic.
- **Technical Service Layer**: Infrastructure services.

II. **Map the events:** We will map the flow events described in the statement with its corresponding actor and layer:

Step	Actor	Description
1	System	Display the menu options.
2	User	Select the execute agent option.
3	System	Display available agents
4	User	Select the agent.
5	User	Introduce the prompt.
6	User	Confirm agent execution, press OK.
7	System	Execute the agent and return result.
8	System	Display success message.

In summary:

1. **Presentation - View (UI)** displays menu options and receive the user input (agent selection + prompt).
2. **Presentation Controller:** Receives UI events such as “execute agent”, validates the input and orchestrate the use case calling the Domain Layer.
3. **Domain**: Execute the selected agent logic based on the given prompt.
4. **Presentation Model:** Stores and manages the UI state and data needed for the presentation layer.
5. Finally, the result travels back through the same path: Domain → Controller → View → *Success Message*.

III. **Identify entities:** Based on this flow we will now define the entities involved in this model:
- Employee (user)
- AI Agent
- Prompt
- Result

IV. **Define Presentation MVC classes and responsibilities:** Based on the MVC pattern description from the learning material which mentions that the presentation responsibilities should be clearly separated, we will define the classes and its responsibilities:

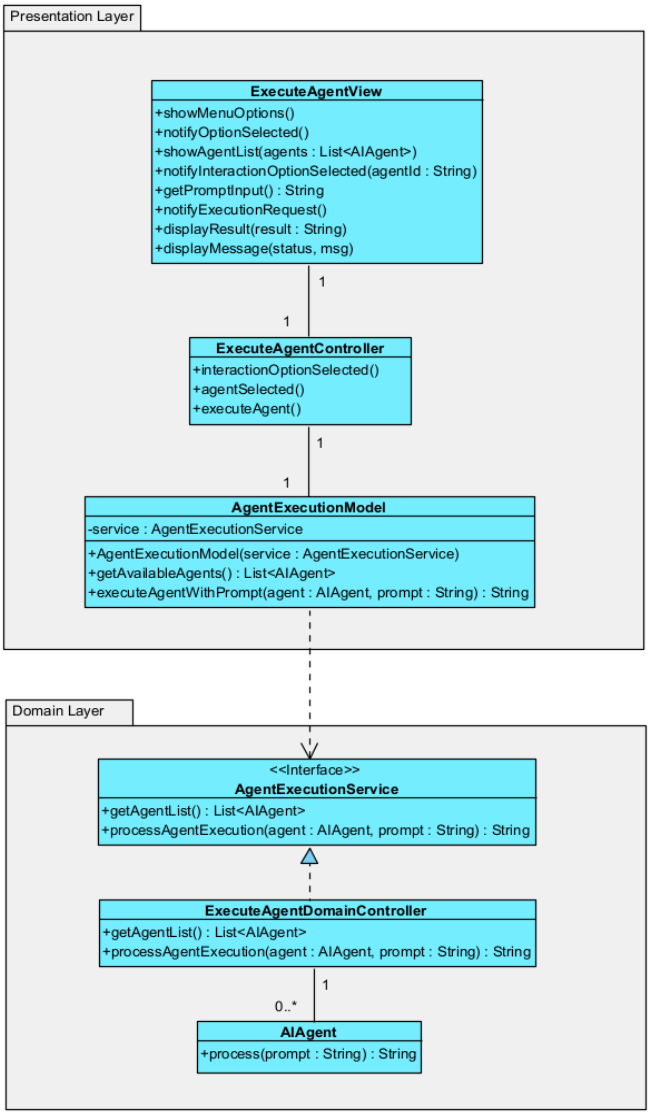
Class	Type	Responsibilities
ExecuteAgentView	View	Display menu, agents, result and gather user input.
ExecuteAgentController	Controller	Handle view events and call the domain logic.
AgentExecutionModel	Model	Keep the state (selected agent, prompt, result).

*NOTE: The view has no direct access to the model. The controller acts as an intermediary between the view and the model. The view notifies the controller which decides the action to perform.

V. **Define Domain classes**: We will separate the business logic classes from the presentation model:

Class	Type	Responsibilities
ExecuteAgentDomainController	Domain	Domain controller for the execution logic.

A) The following class diagram shows the MVC model corresponding to this case:



1. We separate the layers to better visualize the MVC pattern application in the Presentation layer and its connection to the Domain Layer. We also added the AIAgent class to be able to relate this diagram to the previous exercise.

2. To design a decoupled system, in the connection between layers, we use the same pattern learnt in the previous exercise (dependency injection by constructor). In summary, the **AgenExcecutionModel** knows “what” to do (get agents and process the execution) but it does not know “how” to do it. The dependency injected will tell this how through the logic defined in the implementation of the interface.

3. In the Presentation layer MVC pattern diagram, the view has no direct access to the model. The controller acts as an intermediary between the view and the model. The view notifies the controller which decides the action to perform.

B) Explanation of each of the methods in the class diagram:

1. Presentation Layer

1.1 ExecuteAgentView class:

Method	Description
showMenuOptions()	Shows the initial menu with the options (containing the “Use Agent” options)
notifyOptionSelected()	Notify the controller of the “Use Agent” selected option.
showAgentList(List AIAGent)	Show the list of available agents.
notifyInteractionOptionSelected(agentId)	Notify the controller of the selected agent.
getPromptInput()	Ask the user to introduce the prompt.
notifyExecutionRequest()	Informs that the user wants to execute the selected agent and prompt provided.
displayResult(result)	Shows the agent’s result.
displayMessage(status, msg)	Shows operation message (status = success/error). Succes message in our case.

1.2 ExecuteAgentController class:

Method	Description
interactionOptionSelected()	It is called when the option is selected (in our use case “use agent”) and retrieves the agents.
agentSelected()	Register the selected agent and update the model.
executeAgent()	Coordinates the execution: get the prompt, call the model and update the view.

1.3 AgentExecutionModel class:

Method	Description
AgentExecutionModel(service)	This is the constructor that receives the domain service (dependency injection).
getAvailableAgents()	Request to the domain the list of available agents.
executeAgentWithPrompt(agent, prompt)	Execute the selected agent with the indicated prompt.

2. Domain Layer

2.1 AgentExecutionService interface:

Method	Description
getAgentList()	Method to define when implemented.
processAgentExecution(agent, prompt)	Method to define when implemented.

2.2 ExecuteAgentDomainController class:

Method	Description
getAgentList()	Return the list of available agents.
processAgentExecution(agent, prompt)	Execute the agent with the prompt given and return the result

2.3 AIAgent class:

This class was added to the diagram to show how this exercise can complement the answers from question 3. It has the **process(prompt)** method which has the main logic of the agent to process the received prompt.

C) Sequence Diagram: