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 or energy. In the exercise 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). 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" violates or apply for explaining this behaviour. I didn't find any specific posture about this in the official learning material.

C) The `Employee` class is not compliant with the SRP principle. This principle says that "*A class should have responsibility for only one part of the functionality provided by the software*", and in our example model, the `Employee` class has 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). This second responsibility requires classification or categorization logic that should be separated from the basic responsibility already mentioned, especially considering that this logic may change or be expanded in the future, for instance a change in roles structure, or how billing is done for the monthly consumption calculation, that maybe is different based on the role as in the example of the salary in the first CAT, etc.

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.*" So, 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 by promoting the possibility of runtime errors, adding difficulty in the code maintenance.

*Note: As mentioned before, using an enumeration in this case may seems comfortable but difficult due to the continuously growing of generative AI categories. So maybe an option can be to create a class that represent the types as objects because new types may be created dynamically.

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, however if new types of agents with a specific behavior are implemented as it is mentioned, the code will probably need to use conditional based on the category and not polymorphism.

This approach violates the LSP principle since the behavior will change based on the agent type, instead of allowing each class to implement the specific behavior required.

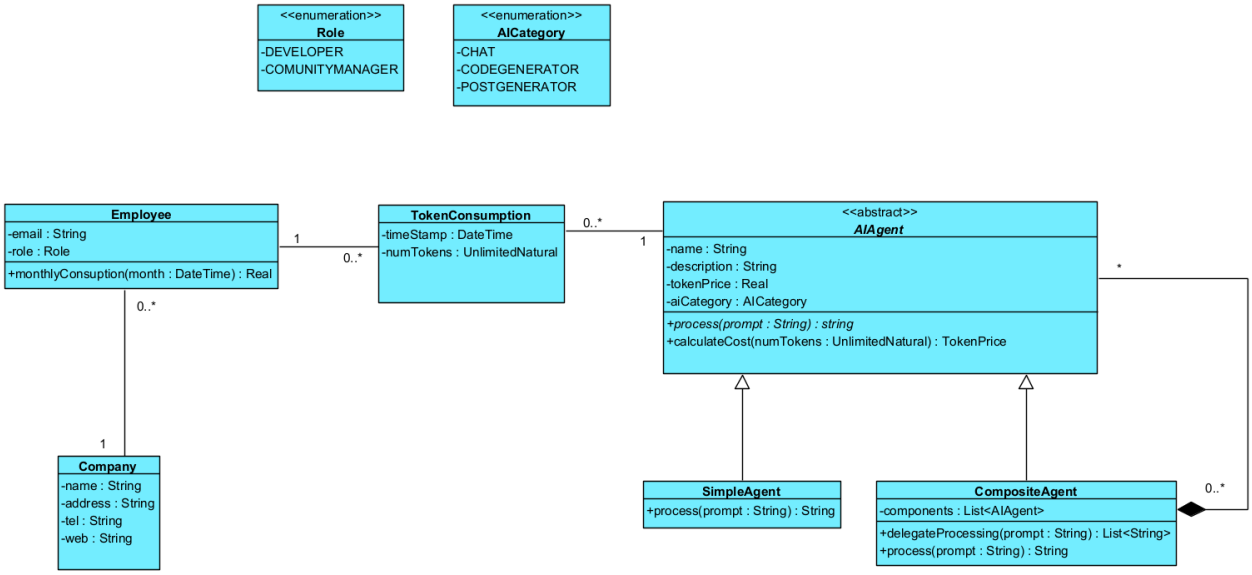
To "apply" the LSP principle, this design should evolve to a class hierarchy where **AIAgent** is the base class and the different types of agents are subclasses, each with its specific behavior.

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 AIAgent could contain many AIAgents whether simple or composite, this pattern seems appropriate as it follows this exact structure.

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 hierarchy 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 to the base diagram I will show this UML diagram:



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)

1. 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 corresponding 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:

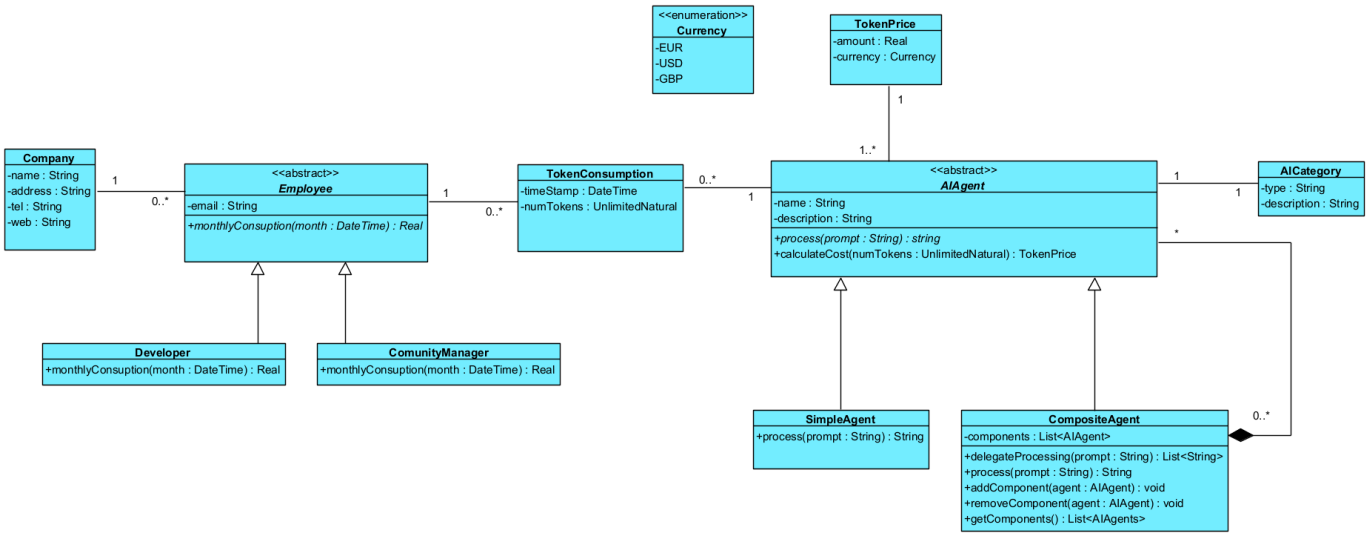
```
// Pseudocode
public List<String> delegateProcessing(String prompt) {
    List<String> responses = new ArrayList<>();
    for (AIAgent component : components) {
        responses.add(component.process(prompt));
    }
    return responses;
}
```

2. 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 will directly implement the processing such as NLP, classification, etc. and in composite agents it will orchestrate the entire process by delegating to it components, gathering the results and combining everything in a coherent answer.

```
// Pseudocode for SimpleAgent
public String process(String prompt) {
    // Implementation of specialized processing
    return processWithSpecializedFunction(prompt);
}

// Pseudocode for CompositeAgent
public String process(String prompt) {
    // Delegate processing to components
    List<String> componentResponses = delegateProcessing(prompt);
    // Combine responses into a coherent result
    return combineResponses(componentResponses);
}
```

A full UML diagram implementation combining 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.

`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

Question 3

TODO:

Question 4

TODO: