**DECORATOR DESIGN PATTERN** (Important)

When to use:

* Dynamically add or modify behavior of objects without altering their structure.
* Avoid subclass explosion by providing a flexible alternative.
* Extend functionality without changing existing code.
* Compose objects with different behavior variants at runtime.

Further Explanation: <https://www.youtube.com/watch?v=w6a9MXUwcfY&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=6>

The Decorator Design Pattern is like adding layers of wrapping paper around a gift box. Each layer adds something extra, but the gift box remains the same underneath.

In programming, it's used to add new features to objects dynamically without changing their original structure. Let's say you have a simple object, like a plain cake. With the Decorator Pattern, you can "decorate" this cake with different toppings or decorations without modifying its core.

For example, you can have a basic Cake object representing the plain cake. Then, you can create decorator classes for each additional feature you want to add, like frosting, sprinkles, or fruit. These decorators wrap around the original cake object, adding their own functionality.

So, the Decorator Pattern allows you to build objects with flexible combinations of features, just like how you can customize a cake with various toppings.

**Decorator Design Pattern**

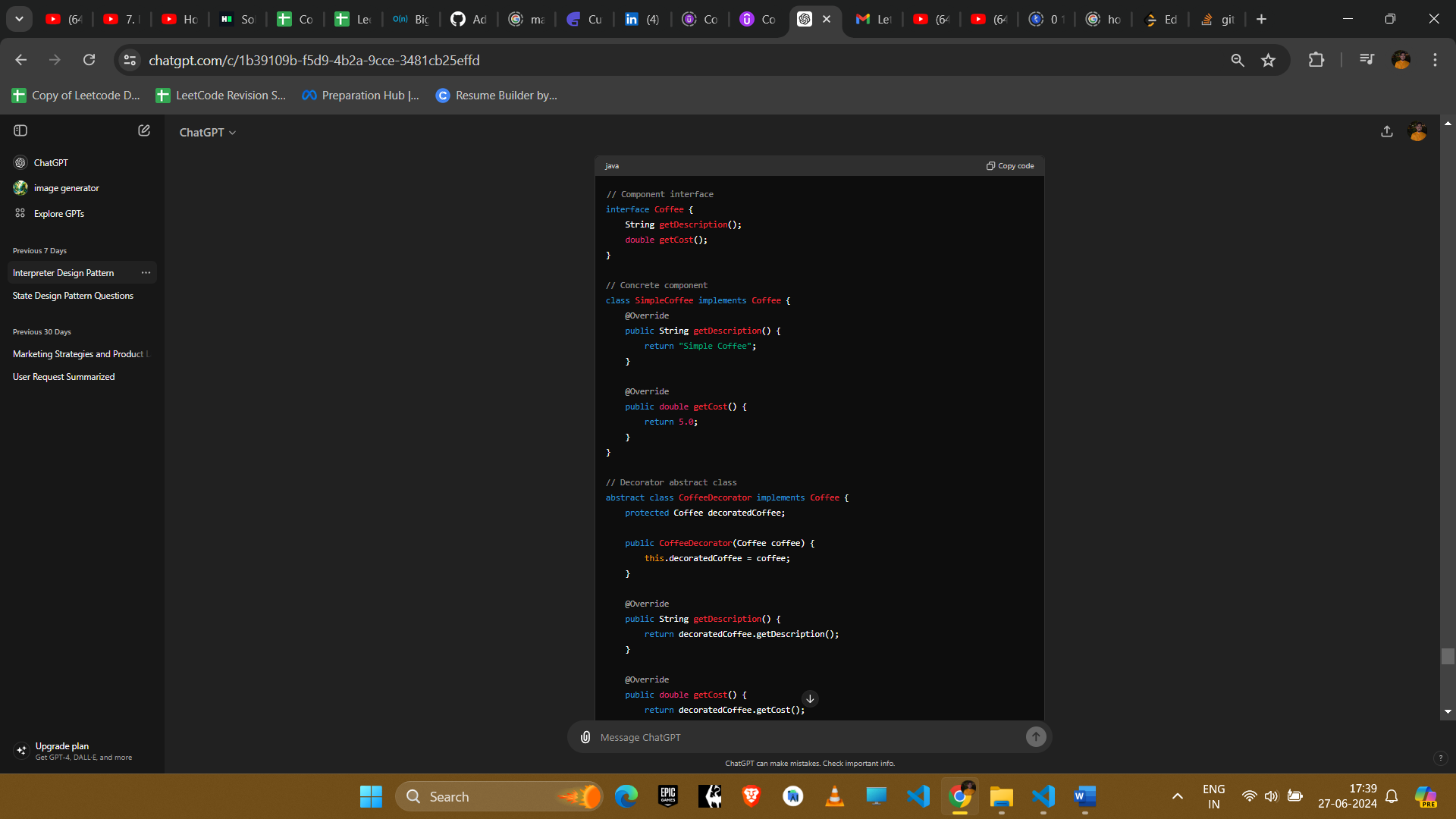
The Decorator design pattern is a structural pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. This pattern involves a set of decorator classes that are used to wrap concrete components. Decorators provide a flexible alternative to subclassing for extending functionality.

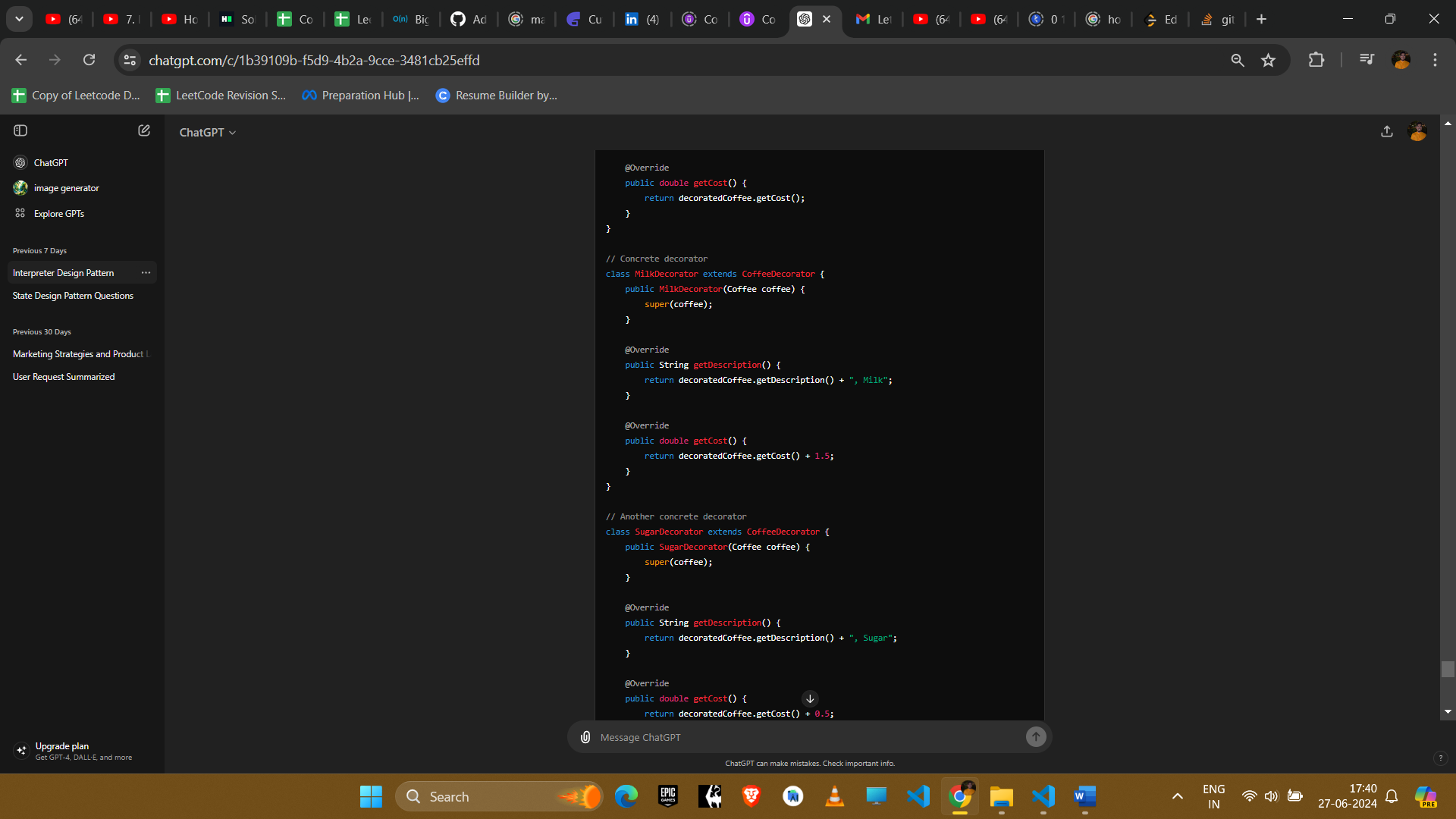
**Definition**

The Decorator pattern enables you to add responsibilities to objects at runtime. Instead of altering an object's structure, you create a set of decorator classes that are used to wrap concrete components. Each decorator class implements the same interface as the component it decorates, ensuring that the decorated objects can be used interchangeably with the undecorated ones.

**Example**

Consider a simple example where we have a Coffee interface and different types of coffee that can be decorated with additional features like milk, sugar, etc.





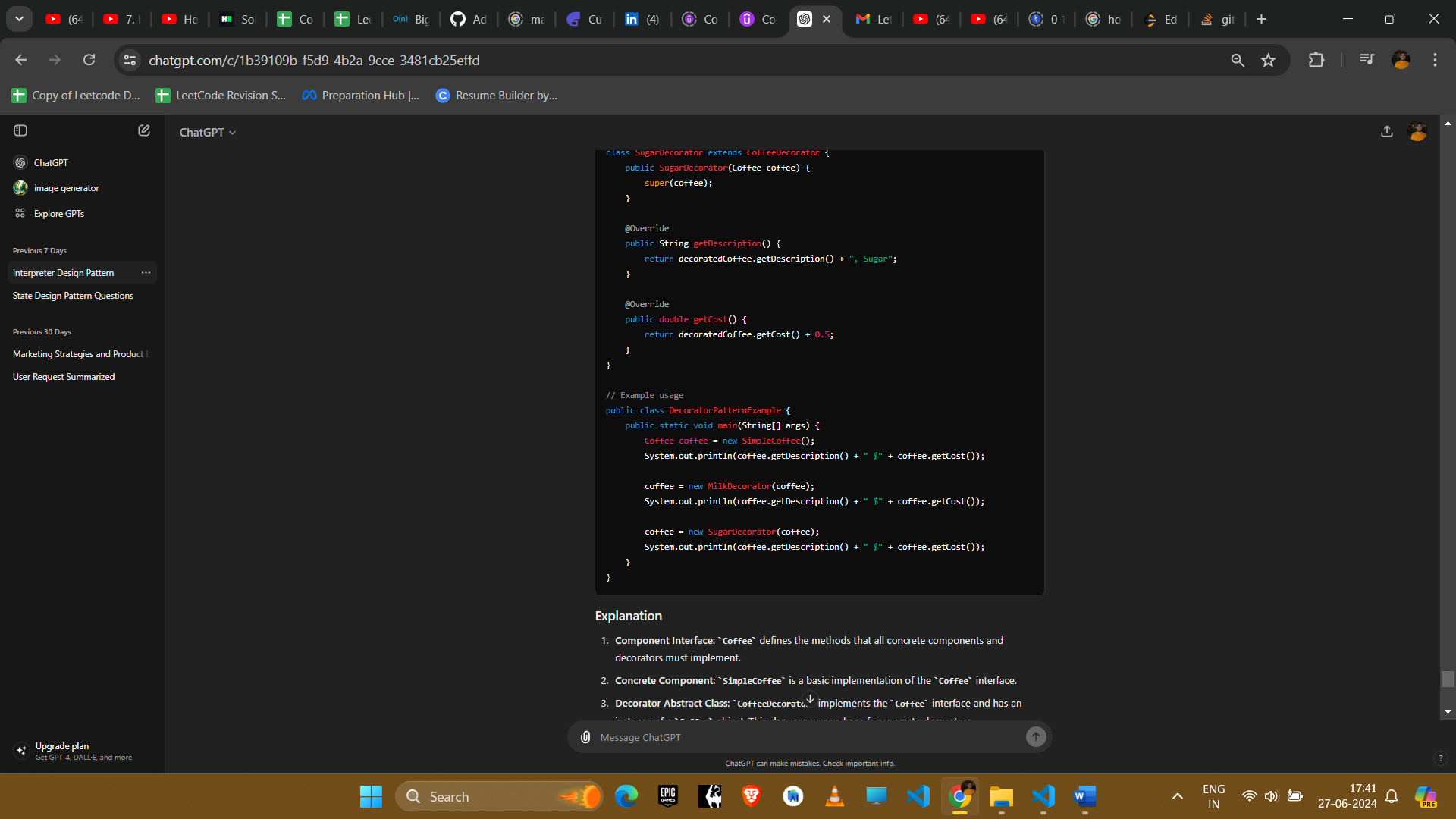
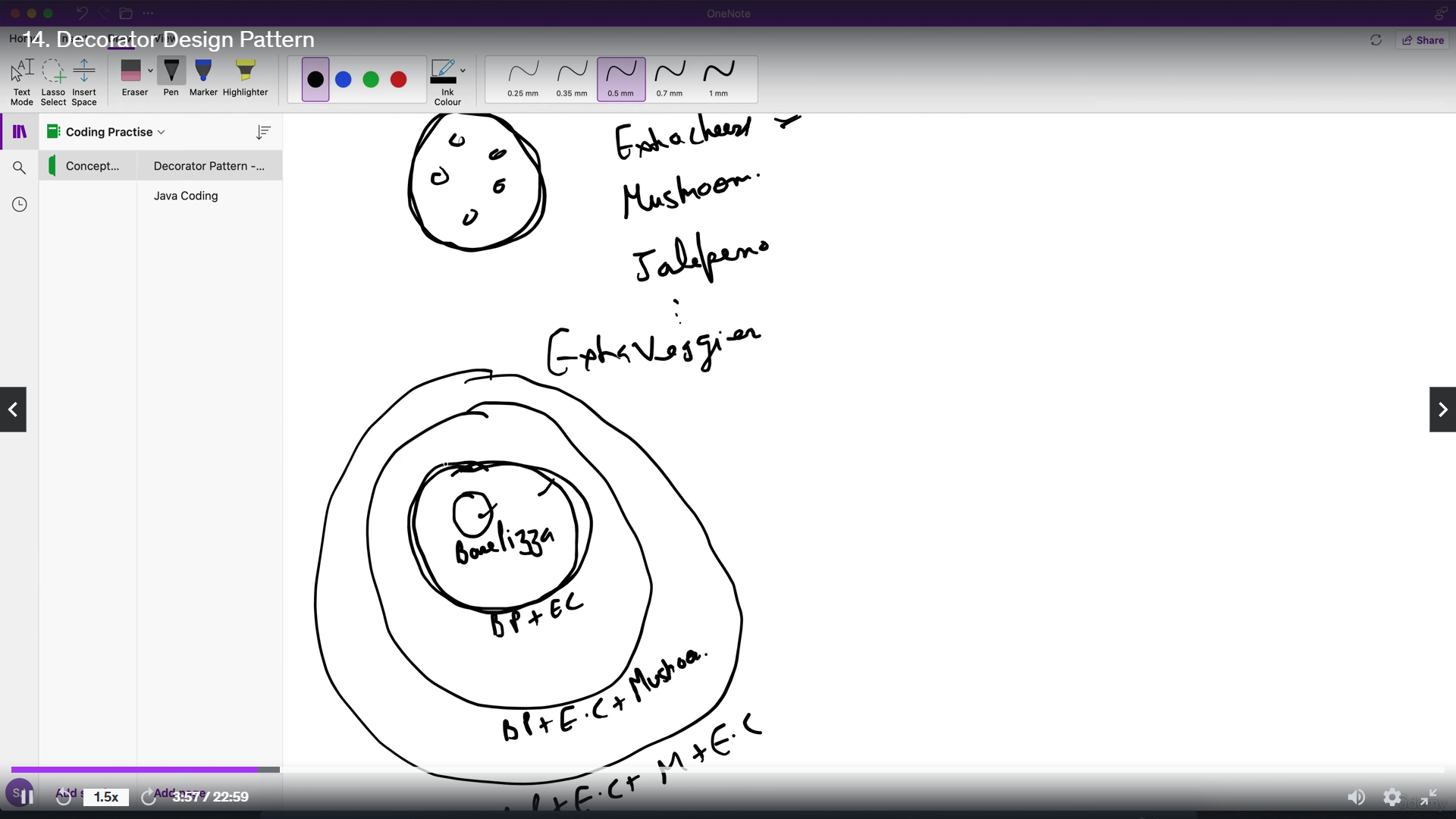


Image illustrates the layer wrapping considering pizza as an example



**Explanation**

1. **Component Interface**: Coffee defines the methods that all concrete components and decorators must implement.
2. **Concrete Component**: SimpleCoffee is a basic implementation of the Coffee interface.
3. **Decorator Abstract Class**: CoffeeDecorator implements the Coffee interface and has an instance of a Coffee object. This class serves as a base for concrete decorators.
4. **Concrete Decorators**: MilkDecorator and SugarDecorator extend CoffeeDecorator and add their own behavior while still adhering to the Coffee interface.

**Example Uses in Amazon Interviews**

1. **UI Component Customization**
   * **Scenario**: Adding dynamic features to UI components like buttons, panels, etc.
   * **Implementation**: Use decorators to add scrollbars, borders, and other UI features at runtime.
2. **Logging**
   * **Scenario**: Adding logging functionality to different parts of an application without changing their code.
   * **Implementation**: Decorate existing classes with logging decorators to log method calls and parameters.
3. **Data Compression and Encryption**
   * **Scenario**: Adding compression or encryption to data streams.
   * **Implementation**: Use decorators to wrap data streams with compression or encryption features.
4. **Extending Core Functionality**
   * **Scenario**: Adding features to core classes without altering their implementation.
   * **Implementation**: Decorate core classes with additional features as needed (e.g., adding caching to data retrieval methods).

**Conclusion**

The Decorator pattern is a powerful tool for extending the behavior of objects dynamically and flexibly. It is especially useful when you need to add responsibilities to individual objects without affecting others, making it a popular choice in many scenarios requiring runtime flexibility and clean code separation.