Slide 2: Definition

- **Speaker:** The Decorator Pattern is classified under structural design patterns. It enables the combination of multiple behaviors by wrapping an object with one or more decorators. This approach attaches additional responsibilities to an object dynamically, offering a more flexible alternative to subclassing for extending functionality. Instead of creating a subclass for every possible combination of behaviors, decorators can be stacked to achieve the desired result.

Slide 3: Motivation

- **Speaker:** The primary motivation for using the Decorator Pattern is to prevent subclass explosion. In object-oriented design, subclassing is a common technique to add functionality. However, creating a new subclass for every combination of behaviors can quickly become impractical. This leads to a large number of subclasses, which complicates maintenance and violates the Single Responsibility Principle (SRP). The Decorator Pattern provides a cleaner, more manageable way to extend functionality without proliferating subclasses.

Slide 4: How to Implement

- **Speaker:** Implementing the Decorator Pattern involves a few key steps:
    1. **Component Interface:** Define an interface for objects that can have responsibilities added to them dynamically.

       ```
       java
       Kopier kode
       public interface Tesla {
           String getDescription();
           double getCost();
       }
       ```

    2. **Concrete Component:** Create a concrete class that implements the Component interface.

       ```
       java
       Kopier kode
       public class ModelY implements Tesla {
           @Override
           public String getDescription() {
               return "Tesla Model Y";
           }

           @Override
           public double getCost() {
               return 50000;
           }
       ```

```
}
```

3. **Decorator Class:** Create an abstract decorator class that implements the Component interface and holds a reference to a Component object.

   ```
   java
   Kopier kode
   public abstract class TeslaDecorator implements Tesla {
       protected Tesla decoratedTesla;

       public TeslaDecorator(Tesla tesla) {
           this.decoratedTesla = tesla;
       }

       @Override
       public String getDescription() {
           return decoratedTesla.getDescription();
       }

       @Override
       public double getCost() {
           return decoratedTesla.getCost();
       }
   }
   ```

4. **Concrete Decorators:** Create concrete decorator classes that extend the abstract decorator class and add new behavior.

   ```
   java
   Kopier kode
   public class Color extends TeslaDecorator {
       public Color(Tesla tesla) {
           super(tesla);
       }

       @Override
       public String getDescription() {
           return decoratedTesla.getDescription() + ", Color";
       }

       @Override
       public double getCost() {
           return decoratedTesla.getCost() + 1000;
       }
   }

   public class Premium extends TeslaDecorator {
       private int level;

       public Premium(Tesla tesla, int level) {
           super(tesla);
           this.level = level;
       }

       @Override
       public String getDescription() {
   ```

```
            return decoratedTesla.getDescription() + ", Premium Level "
    + level;
        }

        @Override
        public double getCost() {
            return decoratedTesla.getCost() + 2000 * level;
        }
    }
```

- By using this approach, you can add any number of behaviors to a concrete component at runtime.

Slide 5: Pros and Cons

- **Speaker:** Let's discuss some pros and cons of the Decorator Pattern:
  - **Pros:**
    - **Flexibility:** Low coupling between decorators and the object being decorated allows for flexible behavior changes.
    - **Open/Closed Principle (OCP):** Classes are open for extension but closed for modification, allowing new functionality to be added without changing existing code.
    - **Single Responsibility Principle (SRP):** Each decorator has a specific responsibility, making the system easier to manage and understand.
    - **Code Simplification:** Reduces the need for an extensive subclass hierarchy, simplifying the codebase.
  - **Cons:**
    - **Lack of Rules:** There are no strict rules governing how decorators should be implemented, which can lead to inconsistent designs.
    - **Increased Complexity:** The use of multiple layers of decorators can increase the complexity of the code, making it harder to debug and maintain.
    - **Performance Overhead:** Each decorator adds a layer of abstraction, which can impact performance if not managed properly.

Slide 6: SOLID Principles

- **Speaker:** The Decorator Pattern aligns well with several SOLID principles:
  - **Single Responsibility Principle (SRP):** Each decorator is responsible for a specific aspect of the behavior, adhering to SRP.
  - **Open/Closed Principle (OCP):** New decorators can be added to extend functionality without altering existing code, adhering to OCP.
  - **Dependency Inversion Principle (DIP):** Both high-level components (e.g., Tesla) and low-level components (e.g., Color, Premium) depend on abstractions rather than concrete implementations.

Slide 7: Comparison with Other Patterns

- **Speaker:** Understanding how the Decorator Pattern compares to other design patterns is crucial:
  - **Decorator vs. Strategy Pattern:**
    - **Decorator:** Adds new functionality by wrapping objects. It focuses on the organization of object structures.
    - **Strategy:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It focuses on the behavior of objects.
  - **Decorator vs. Template Method Pattern:**
    - **Decorator:** Behavior is added at runtime, allowing dynamic changes.
    - **Template Method:** Defines the skeleton of an algorithm, with some steps delegated to subclasses. Changes happen at compile time.

Slide 8: Similarities and Differences

- **Speaker:** Let's delve deeper into the similarities and differences between the Decorator Pattern and other patterns:
  - **Template Method Pattern:**
    - Behavior is defined by subclasses.
    - Changes occur at compile time.
    - Suitable for situations where the algorithm's structure is fixed but steps can vary.
  - **Strategy Pattern:**
    - Allows for dynamic behavior changes at runtime.
    - Encapsulates algorithms in separate classes, making them interchangeable.
    - Ideal for scenarios where different algorithms can be applied to an object depending on the context.

  The key difference is that the Decorator Pattern focuses on adding new functionalities, while the Strategy Pattern focuses on selecting an algorithm at runtime.

Slide 9: Example in Practice

- **Speaker:** To give you a practical understanding, let's revisit the example:

  ```
  java
  Kopier kode
  Tesla myTesla = new ModelY();
  myTesla = new Color(myTesla);
  myTesla = new Premium(myTesla, 12);
  ```

  In this example, 'ModelY' is the base Tesla model. By wrapping it with the 'Color' and 'Premium' decorators, we dynamically add color and premium features to the Tesla object. This approach allows for a flexible and maintainable way to extend the functionality of the

Tesla model without altering its core structure or creating new subclasses for each combination of features.

Slide 10: Conclusion

- **Speaker:** In conclusion, the Decorator Pattern is a powerful and flexible design pattern that allows for the dynamic addition of behavior to objects. It helps maintain flexibility and adherence to design principles like OCP and SRP while avoiding the pitfalls of subclass explosion. However, it is crucial to manage its complexity and performance considerations carefully to reap the full benefits of this pattern.