Documentation for Decorator Pattern Code

Introduction

This document provides a comprehensive overview of a Java implementation of the Decorator Pattern used to extend the functionality of a base beverage object. The Decorator Pattern allows dynamic addition of behavior to objects without modifying their structure, making it a flexible solution for various use cases in software design.

General Explanation

The code demonstrates a classic example of the Decorator Pattern applied to a beverage system. The base class Coffee implements the Beverage interface. Various decorators, such as MilkDecorator, SugarDecorator, and SyrupDecorator, extend the functionality of the base beverage object, allowing additional features (e.g., milk, sugar, syrup) to be added dynamically.

```java
package Decorator_Pattern_Code;

public interface Beverage {
    String getDescription();
    double cost();
}
```

```java
package Decorator_Pattern_Code;

public class Coffee implements Beverage {

    @Override
    public String getDescription() {
        return "Coffee";
    }

    @Override
    public double cost() {
        return 2.00; // base price for coffee
    }
}
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | F

```java
package Decorator_Pattern_Code;

public class Main {

    public static void main(String[] args) {
        // Base beverage
        Beverage beverage = new Coffee();
        System.out.println(beverage.getDescription() + " $" + beverage.cost());

        // Beverage with milk
        Beverage beverageWithMilk = new MilkDecorator(beverage);
        System.out.println(beverageWithMilk.getDescription() + " $" + beverageWithMilk.cost());

        // Beverage with milk and sugar
        Beverage beverageWithMilkAndSugar = new SugarDecorator(beverageWithMilk);
        System.out.println(beverageWithMilkAndSugar.getDescription() + " $" + beverageWithMilkAndSugar.cost());

        // Beverage with milk, sugar, and vanilla syrup
        Beverage fullyCustomizedBeverage = new SyrupDecorator(beverageWithMilkAndSugar, "Vanilla");
        System.out.println(fullyCustomizedBeverage.getDescription() + " $" + fullyCustomizedBeverage.cost());
    }
}
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | First_Proje.

```java
package Decorator_Pattern_Code;

public class MilkDecorator extends BeverageDecorator {

    public MilkDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Milk";
    }

    @Override
    public double cost() {
        return beverage.cost() + 0.50; // additional cost for milk
    }
}
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | First_Proje...

```java
package Decorator_Pattern_Code;

public class MilkDecoratorTest {

}
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | First_Proje...

```java
package Decorator_Pattern_Code;

public class SugarDecorator extends BeverageDecorator {

    public SugarDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Sugar";
    }

    @Override
    public double cost() {
        return beverage.cost() + 0.20; // additional cost for sugar
    }
}
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | First_Proje...

```java
package Decorator_Pattern_Code;

public class SyrupDecorator extends BeverageDecorator {

    private final String flavor;

    public SyrupDecorator(Beverage beverage, String flavor) {
        super(beverage);
        this.flavor = flavor;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", " + flavor + " Syrup";
    }

    @Override
    public double cost() {
        return beverage.cost() + 0.75; // additional cost for syrup
    }
}
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | First_Proje...

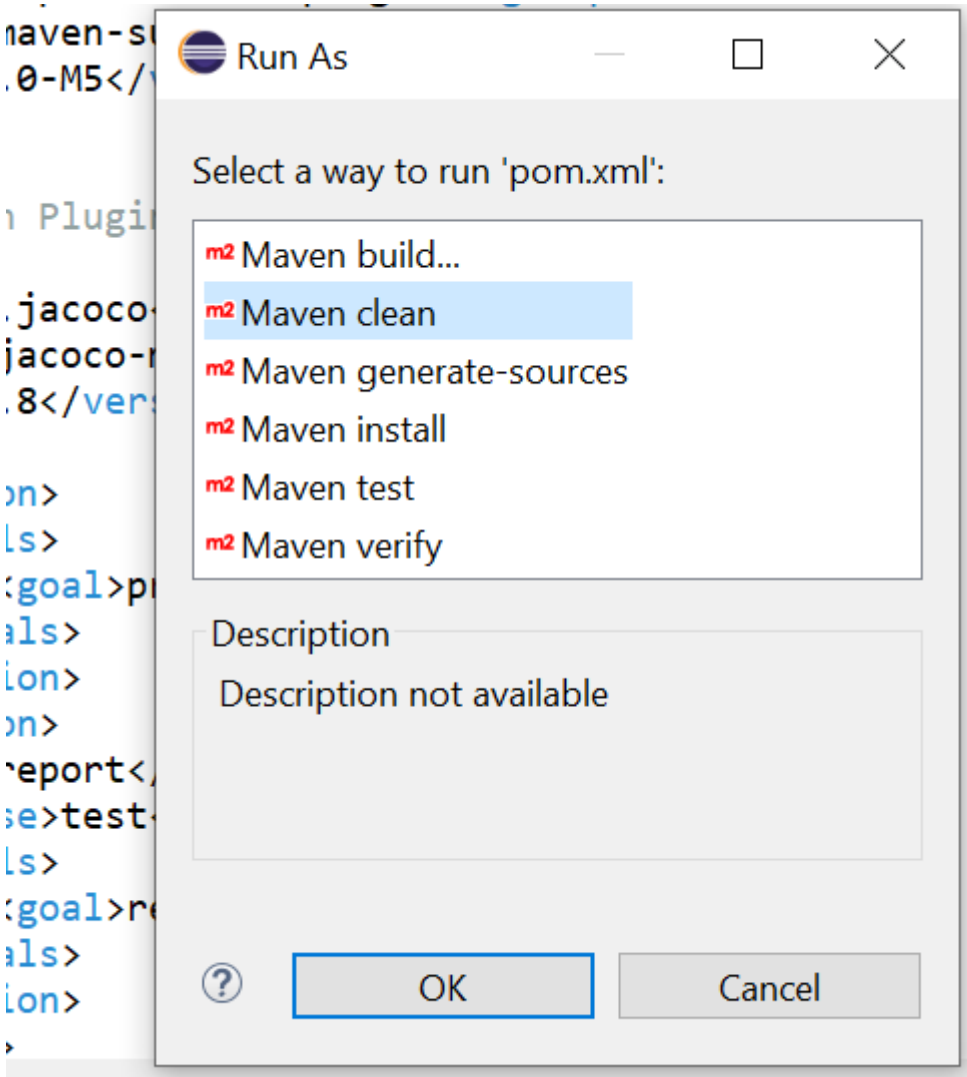
```xml
http://maven.apache.org/xsd/maven-4.0.0.xsd (xsi:schemaLocation with catalog)
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>decorator-pattern</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <maven.test.skip>false</maven.test.skip>
    </properties>

    <dependencies>
        <!-- JUnit 5 -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.9.3</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.9.3</version>
            <scope>test</scope>
```

Beverage.java | Coffee.java | Main.java | MilkDecorato... | MilkDecorato... | SugarDecora... | SugarDecora... | SyrupDecora... | First_Proje...

```xml
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
        </plugin>

        <!-- JaCoCo Maven Plugin -->
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.8</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
                <execution>
                    <id>report</id>
                    <phase>test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
  </build>
</project>
```

```xml
            <scope>test</scope>
        </dependency>

        <!-- Mockito -->
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>5.3.1</version>
            <scope>test</scope>
        </dependency>

        <!-- JaCoCo for coverage reporting -->
        <dependency>
            <groupId>org.jacoco</groupId>
            <artifactId>org.jacoco.agent</artifactId>
            <version>0.8.8</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Maven Surefire Plugin -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M5</version>
            </plugin>

```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml



Console

```
<terminated> C:\Users\HP\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< com.example:decorator-pattern >--------------------
[INFO] Building decorator-pattern 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- clean:3.2.0:clean (default-clean) @ decorator-pattern ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.281 s
[INFO] Finished at: 2024-08-31T19:12:28-06:00
[INFO] ------------------------------------------------------------------------
```

Key Classes

Beverage Interface: Defines the common interface for all beverages. It includes methods for getting the description and cost of a beverage.

Coffee Class: Implements the Beverage interface and represents a concrete beverage. It provides the base description and cost for a coffee.

BeverageDecorator Abstract Class: Abstract class that implements the Beverage interface and holds a reference to a Beverage object. It serves as the base class for all decorators.

MilkDecorator Class: A concrete decorator that adds milk to the beverage. It overrides the getDescription and cost methods to include milk in the description and adjust the cost.

SugarDecorator Class: A concrete decorator that adds sugar to the beverage. It modifies the description and cost to reflect the addition of sugar.

SyrupDecorator Class: A concrete decorator that adds syrup to the beverage. It takes an additional parameter for the syrup flavor and adjusts the description and cost accordingly.

Main Class: Demonstrates the use of decorators to build a customized beverage. It creates a base coffee, decorates it with milk, sugar, and syrup, and prints the final description and cost.

Methods Used

getDescription(): Returns a string describing the beverage and its decorations. Each decorator extends the base description with additional information.

cost(): Returns the total cost of the beverage including all decorations. Each decorator adds its specific cost to the base cost of the beverage.

Key Points

1. **Decorator Pattern:** Allows the dynamic addition of responsibilities to objects. It adheres to the Single Responsibility Principle by allowing the extension of functionality without modifying existing code.

2. **Extensibility:** The pattern provides a way to extend the functionality of a base object by composing objects, rather than using inheritance. This enhances flexibility and maintainability.

3. **Composition Over Inheritance:** Decorators leverage composition to add functionality, which is often more flexible and scalable compared to using inheritance for every variation of behavior.

4. **Maintainability:** Each decorator is a separate class, making it easy to add new types of decorations without altering the existing codebase. This promotes better maintainability and code organization.