COMP2511

Decorator Pattern
Adapter Pattern

Prepared by
Dr. Ashesh Mahidadia
Design Patterns

- Creational Patterns
  - Abstract Factory
  - Factory Method
  - Singleton

- Structural Patterns
  - Adapter
  - Composite
  - Decorator

- Behavioral Patterns
  - Iterator
  - Observer
  - State
  - Strategy
  - Template
  - Visitor

We plan to discuss the rest of the design patterns above in the following weeks.
Decorator Pattern
Decorator Pattern: Intent

• "Attach additional **responsibilities** to an object **dynamically**. Decorators provide a flexible alternative to sub-classing for extending functionality." [GoF]

• **Decorator design patterns** allow us to selectively add **functionality to an object** (not the class) at runtime, based on the requirements.

• Original class is **not** changed (Open-Closed Principle).

• **Inheritance** extends behaviors at **compile time**, additional functionality is bound to **all** the instances of that class for their life time.

• The **decorator** design pattern prefers a **composition** over an inheritance. Its a **structural pattern**, which provides a **wrapper** to the existing class.

• Objects can be **decorated multiple times**, in different order, due to the **recursion** involved with this design pattern. See the example in the Demo.

• Do not need to implement all possible functionality in a single (complex) class.
Decorator Pattern: Structure

- **Client**: refers to the Component interface.
- **Component**: defines a common interface for Component1 and Decorator objects
- **Component1**: defines objects that get decorated.
- **Decorator**: maintains a reference to a Component object, and forwards requests to this component object (`component.operation()`)
- **Decorator1, Decorator2, ...**: Implement additional functionality (`addBehavior()`) to be performed before and/or after forwarding a request.

See the example in the Demo.
Decorator Pattern: Structure

- Given that the decorator has the same supertype as the object it decorates,

  we can pass around a **decorated** object in place of the **original** (wrapped) object.

- The **decorator adds its own** behavior either before and/or after delegating to the object it decorates to do the rest of the job.

*From the book “Head First Design Pattern”.*
Decorator Pattern: Example

Welcome to Starbuzz Coffee

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The cost() method is abstract; subclasses need to define their own implementation.

The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast". The getDescription() method returns the description.

Each subclass implements cost() to return the cost of the beverage.
Each cost method computes the cost of the coffee along with the other condiments in the order.
Decorator Pattern: Example

Welcome to Starbuzz Coffee

Beverage acts as our abstract component class.

HouseBlend
- cost()

DarkRoast
- cost()

Espresso
- cost()

Decaf
- cost()

CondimentDecorator
- getDescription()

Milk
- Beverage beverage
- cost()
- getDescription()

Mocha
- Beverage beverage
- cost()
- getDescription()

Soy
- Beverage beverage
- cost()
- getDescription()

Whip
- Beverage beverage
- cost()
- getDescription()

The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...
Decorator Pattern: Example

Constructing a drink order with Decorators

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

The Mocha object is a decorator. Its type mirrors the object it is decorating in this case, a Beverage. (By “mirror,” we mean it is the same type.)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

Whip is a decorator, so it also mirrors DarkRoast’s type and includes a cost() method.

1. First, we call cost() on the outermost decorator, Whip.
2. Whip calls cost() on Mocha.
3. Mocha calls cost() on DarkRoast.
4. DarkRoast returns its cost, 99 cents.
5. Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.
6. Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.
Decorator Pattern: Code

```java
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Whipe: beverage.cost() is: " + beverage_cost);
    System.out.println(" - adding One Whip cost of 0.10c ");
    System.out.println(" - new cost is: " + (0.10 + beverage_cost) );
    return 0.10 + beverage_cost ;
}
```

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Mocha: beverage.cost() is: " + beverage_cost ");
    System.out.println(" - adding One Mocha cost of 0.20c ");
    System.out.println(" - new cost is: " + (0.20 + beverage_cost ) );
    return 0.20 + beverage_cost ;
}
```

Read the example code discussed/developed in the lectures, and also provided for this week.
Decorator Pattern: Java I/O Example

A text file for reading.

FileInputStream is the component that’s being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, and a few others. All of these give us a base component from which to read bytes.

FileInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds buffering behavior to a FileInputStream: it buffers input to improve performance.
Decorator Pattern: Java I/O Example

These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn’t show, like ObjectInputStream.

And finally, here are all our concrete decorators.
Decorator Pattern: Code

```java
InputStream f1 = new FileInputStream(filename);
InputStream b1 = new BufferedInputStream(f1);
InputStream lCase1 = new LowerCaseInputStream(b1);
InputStream rot13 = new Rot13(b1);

while ((c = rot13.read()) >= 0) {
    System.out.print((char) c);
}
```

Read the example code discussed/developed in the lectures, and also provided for this week
Decorator Pattern:

• Demo ...
Adapter Pattern
Adapter Pattern : Intent

- "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." [GoF]

- The adapter pattern allows the interface of an existing class to be used as another interface, suitable for a client class.

- The adapter pattern is often used to make existing classes (APIs) work with a client class without modifying their source code.

- The adapter class maps / joins functionality of two different types / interfaces.

- The adapter pattern offers a wrapper around an existing useful class, such that a client class can use functionality of the existing class.

- The adapter pattern do not offer additional functionality.
The adapter contains an instance of the class it wraps.
In this situation, the adapter makes calls to the instance of the wrapped object.
Adapter: Example

```java
class Iphone implements LightningPhone {
    private boolean connector;

    @Override
    public void useLightning() {
        connector = true;
        System.out.println("Lightning connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}
```

```java
class Android implements MicroUsbPhone {
    private boolean connector;

    @Override
    public void useMicroUsb() {
        connector = true;
        System.out.println("MicroUsb connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect MicroUsb first");
        }
    }
}
```

interface LightningPhone {
    void recharge();
    void useLightning();
}

interface MicroUsbPhone {
    void recharge();
    void useMicroUsb();
}
```
Adapter: Example

```java
public class AdapterDemo {
    static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
        phone.useMicroUsb();
        phone.recharge();
    }

    static void rechargeLightningPhone(LightningPhone phone) {
        phone.useLightning();
        phone.recharge();
    }

    public static void main(String[] args) {
        Android android = new Android();
        Iphone iphone = new Iphone();

        System.out.println("Recharging android with MicroUsb");
        rechargeMicroUsbPhone(android);

        System.out.println("Recharging iPhone with Lightning");
        rechargeLightningPhone(iphone);

        System.out.println("Recharging iPhone with MicroUsb");
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iphone));
    }
}
```

```java
class LightningToMicroUsbAdapter implements MicroUsbPhone {
    private final LightningPhone lightningPhone;

    public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}
```
Design Patterns: Discuss Differences

- Creational Patterns
  - Abstract Factory
  - Factory Method
  - Singleton

- Structural Patterns
  - Adapter
  - Composite
  - Decorator

- Behavioral Patterns
  - Iterator
  - Observer
  - State
  - Strategy
  - Template
  - Visitor

We plan to discuss the rest of the design patterns above in the following weeks; and many more other topics.
End