WHAT ARE “STRUCTURAL” DESIGN PATTERNS?

- Patterns that ease the design by identifying simple ways to realize relationships between entities.
  - **Decorator** adds additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes.
  - **Adapter** “adapts” one interface for a class into one that a client expects.
  - **Facade** creates a simplified interface of an existing interface to ease usage for common tasks.
  - **Flyweight** a high quantity of objects share a common properties object to save space.
  - **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
THE DECORATOR PATTERN

• Used to modify the functionality of an object at runtime.
  • But, at the same time, other instances of the same class will not be affected by this.
  • So, an individual object gets the modified behavior.
• Attaches additional responsibilities to an object dynamically.

This is the act of “decorating” an object!

• Decorators provide a flexible alternative to inheritance for extending functionality.
  • How? By wrapping an object.

• Works on the principle that classes should be open to extension but closed to modification.
THE DECORATOR PATTERN (GOAL)

- Allow classes to be easily extended to incorporate new behavior without modifying existing code.
- What do we get if we accomplish this?
  - Designs that are
    1. resilient to change, and
    2. flexible enough to take on new functionality (usually, this is important in industry settings where clients may propose changing requirements).
5 WHAT’S WRONG WITH INHERITANCE?

• Generally, nothing. Inheritance offers a lot of flexibility!
• But, if we use inheritance to extend the behavior of an object, it’s done at compile time.
  • So this behavior is applicable to all the instances of the class.
• We can’t add new (or remove any existing) behavior at runtime.
  • This is when the decorator pattern is helpful.
THE DECORATOR PATTERN (EXAMPLE)

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 the 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.
CLASS EXPLOSION!
Each cost method computes the cost (coffee + all condiments)
HANDLING CLASS EXPLOSION

• Can’t we just use instance variables and inheritance in the superclass?
  • This way, we can keep track of the condiments.
  • Solves the “maintainability” aspect of good design.
HANDLING CLASS EXPLOSION

Can’t we just use instance variables and inheritance in the superclass?

- This way, we can keep track of the condiments.
- Solves the “maintainability” aspect of good design.

But this doesn’t solve another problem … modifiability.
public class Beverage {
    public double cost() {
    }
}

public class DarkRoast extends Beverage {
    public DarkRoast() {
        description = "Wake up, it’s a dark roast!";
    }
    public double cost() {
    }
}
• Price changes for condiments means changing existing code.

• New condiments means adding new methods and maybe even changing the cost method.
  • For Thanksgiving season, “cinnamon” could become a condiment not otherwise available.

• Some subclasses may inherit meaningless methods
  • e.g., Tea inheriting a method hasWhip()
• Customer wants Mocha with dark roast and whip
  1. Take a DarkRoast object.
  2. “Decorate” it with a Mocha object.
  3. “Decorate” it with a Whip object.
  4. Call the cost() method
     • Rely on delegation to add on the condiment costs.
• Customer wants Mocha with dark roast and whip
  1. Take a DarkRoast object.
  2. “Decorate” it with a Mocha object.
  3. “Decorate” it with a Whip object.
  4. Call the cost() method
     • Rely on delegation to add on the condiment costs.
STARBUZZ WITH DECORATOR PATTERN

Beverage acts as our abstract component class.

Beverage
- description
- getgetDescription()
- cost()
- // other useful methods

HouseBlend
- cost()

DarkRoast
- cost()

Espresso
- cost()

Decaf
- cost()

CondimentDecorator
- getgetDescription()

Milk
- Beverage beverage
- cost()
- getgetDescription()

Mocha
- Beverage beverage
- cost()
- getgetDescription()

Soy
- Beverage beverage
- cost()
- getgetDescription()

Whip
- Beverage beverage
- cost()
- getgetDescription()

The four concrete components, one per coffee type.
public abstract class Beverage {
    String description = "Unknown Beverage";
    public String getDescription() { return description; }
    public abstract double cost();
}

public class Espresso extends Beverage {
    public Espresso() { description = "Espresso"; }
    public double cost() { return 1.99; }
}

public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}

public class Mocha extends CondimentDecorator {
    Beverage beverage;
    public Mocha(Beverage beverage) { this.beverage = beverage; }
    public String getDescription() { return beverage.getDescription()+", Mocha"; }
    public double cost() { return .20 + beverage.cost(); }
}
public class StarbuzzCoffee {

    public static void main(String args[]) {

        Beverage beverage1 = new Espresso();
        System.out.println(beverage1.getDescription() + " 
        "$ + beverage1.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " 
        "$ + beverage2.cost());

        Beverage beverage3 = new Soy(
                new Mocha(
                        new Whip(
                                new HouseBlend())));
        System.out.println(beverage3.getDescription() + " 
        "$ + beverage3.cost());
    }
}
LineNumberInputStream 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 behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

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