Decorator Pattern

-- Dilbert Comic strip --

1. We still have too many software faults. We'll miss our ship date.
2. Move the list of faults to the "future development" column and ship it.
3. 90% of this job is figuring out what to call stuff.
10¢ Design: Which classes must be included?

- **Story**
  As a user
  I want a weekend planner
  so that I can have more fun

- **Scenario**
  Given that I have made a plan
  When Saturday arrives
  Then Send me my Saturday reminders
  When Sunday arrives
  Then Send me my Sunday reminders

A. plan reminder
B. plan reminder weekend planner
C. plan reminder reminders Saturday Sunday
Discussion

- **Not B:** “weekend planner” is the app itself; don’t need a class for that. Design comes from scenarios, not Story. “Weekend” is not a stable concept (3-day weekends), or it’s possibly two things (Saturday and Sunday), if used to extract plans on the two separate days.

- **Not C:** Saturday and Sunday are instances of type/class Day, not classes themselves.

- **Best answer:** A (note the question “which classes must be included; we also need a Day class”)
Objects versus Classes

- **Object** represents a “thing”
  - person, car, date, ...
  - It’s “alive”: responds to messages

- If two objects generally **behave the same**, then are **same type**
  - Type = (abstract) class or interface

- **Recall class is a template** or **blueprint** for making objects
  - The car template is not a car, it’s a “blueprint” for a car
  - Make as many as I need on the fly

- The “Sunday” class is not a Sunday, and also lousy as a blueprint
  - Make one Sunday, but not Monday
  - And what about 18th day of month?

In OOD, group your object instances into classes.
Decorator Pattern

Can you say “composition” and “delegation”? Again???
Decorator Pattern - Problem

- Want to allow **adding new behaviors** to an object **without modifying** class

- Coffee condiments example is great
  - Soy, Mocha, Whip, ...
  - Class for each combination?
    - $2^n$ is too many!
  - And would be static – cannot change on the fly
  - Flags in Beverage?
  - Many combos are nonsense
  - Have to modify Beverage to add new options

- Clothing another example
  - Take off jacket?
  - Put on hat?
  - 2 flags or 4 classes? No!

- How do w/o mod’ing existing classes?
Decorator - Solution

So, a DarkRoast wrapped in Mocha and Whip is a Beverage and we can do anything with it, including calling its cost() method. Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

```
package headfirst.decorator.starbuzz;

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

```
package headfirst.decorator.starbuzz;

public class Mocha extends CondimentDecorator {
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + " Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

```
package headfirst.decorator.starbuzz;

public class Espresso extends Beverage {
    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

```
package headfirst.decorator.starbuzz;

public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

```
package headfirst.decorator.starbuzz;

public class DarkRoast;

beverage2 = new Mocha(beverage2);
```
UML for Coffee Decorator

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

Beverage acts as our abstract component class.

The four concrete components, one per coffee type.

Composes (HAS-A)

Extends

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

HouseBlend
- cost()

DarkRoast
- cost()

Espresso
- cost()

Decaf
- cost()

CondimentDecorator
- getBeverageDescription()

Milk
- Beverage beverage
- cost()
- getBeverageDescription()

Mocha
- Beverage beverage
- cost()
- getBeverageDescription()

Soy
- Beverage beverage
- cost()
- getBeverageDescription()

Whip
- Beverage beverage
- cost()
- getBeverageDescription()
In family of patterns called “Wrapper”

The `ConcreteComponent` is the object we’re going to dynamically add new behavior to. It extends `Component`.

Each component can be used on its own, or wrapped by a decorator.

Each decorator **HAS-A** (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

The `ConcreteDecorator` has an instance variable for the thing it decorates (the `Component` the `Decorator` wraps).

Decorators can extend the state of the component.

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.
Code a Capitalization Decorator

Example:
  hello earthlings → Hello Earthlings

class WordReader extends Reader {
  public WordReader (File file);
  public int read (char[] cbuf);  // cbuf holds word
}

You are decorating the WordReader class (or rather, it’s objects) with the capitalization capability.

Nesting (composing) a WordReader inside CapitalizationDecorator

Useful resource: static char Character.toUpperCase(char)
Capitalization Decorator

class WordReader extends Reader {
    public WordReader (File file);
    public int read (char[] cbuf); // cbuf holds word
}

class UpperReaderDecorator extends ReaderDecorator {
    Reader innerReader;
    public UpperReaderDecorator(Reader iR) {
        innerReader = iR;
    }
    public int read (char[] cbuf) {
        int length = innerReader.read(cbuf);
        cbuf[0] = Character.toUpperCase(cbuf[0]);
        return length;
    }
}

abstract class ReaderDecorator extends Reader {
    abstract public int read (char[] cbuf);
}

char[100] mybuf;
Reader myreader = new WordReader(System.in);
myreader = new UpperReaderDecorator(myreader);
int length = myreader.read(mybuf);
Open/Closed Principle

Yes, you can have it both ways

Brain surgery is not necessary when putting on a hat
Classes should be **open to extension**, but **closed to modification**

- **Extension**
  - Composition of abstract types, not concrete classes
  - Constructors and setters also take those abstract types
  - Composer delegates to the composed
  - Inheritance from those abstract classes and interfaces

- **Modification**
  - Editing – allowed for fixing bugs, but not adding behavior
  - Danger: propagation of changes to numerous dependents (subclasses, aggregators)
  - Costly invalidation of test cases (have to rewrite)

- “Closure” is really creating openings for extension
  - There is no “close” operation, but can use version control
- Also, a **constructor** that initially sets **tires field**.
- Note that KiaSoul **delegates** to Tire
- **Open to infinity of tires without modification**
OCP for Observer

- Also, Display constructor that takes Subject, and Subject register method that takes Observer
- Subject delegates to the Observer
- Open to adding observers without modification
OCP for Decorator

- Open to addition of new drinks and condiments without modification to support all combinations

- Composition of abstract types
  - Inheritance from abstract types
  - Decorator delegates to abstract superclass

Beverage acts as our abstract component class.

The four concrete components, one per coffee type.
Does this diagram indicate OCP?

A. Yes (and why?)

B. No (why not?)
Discussion

- Best answer: A. Because Client composes an abstract interface, it can accept any implementation of Target. Thus, it is possible to add new Adapter/Adaptee pairs to the design without modifying any existing classes.

- Not B. Some people noted that Adapter, a concrete class refers to a concrete class. But that is not the part of the design we need “open”. I can always code new Adapter/Adaptee pairs, if necessary.
The 80/20 Rule

"For many events, roughly 80% of the effects come from 20% of the causes." - Pareto

Therefore 20% of the effort produces 80% of the results but the last 20% of the results consumes 80% of the effort.
80/20 Rule – diminishing returns tradeoff

- Returns on engineering investment are not linear
- **80 percent of the benefit for 20% of the effort**
- Diminishing returns for effort/investment beyond
  - *(last 80% of effort gets only 20% gain)*
- Also known as Pareto Principle
- Is an $80K car 4x better than a $20K car?
80/20 rule: Project Examples

- Geofencing sounds cool...
  - ...but a bit of overkill (Porsche vs. Honda)
  - 20% better than brute force, but more effort than it’s worth

- When we prioritize our Stories, we are invoking the benefits of the 80/20 rule
  - OK, we didn’t get the recurring events done
    - Complicated! Time-consuming!
  - But we still have location-based reminders!

- JUnit is installed, working, and familiar...
  - ...and JBehave/Robotium is not
  - Yes, it’s 20% better, but cost you time you didn’t have

- Remember the 80/20 rule, diminishing returns, cost/benefit, tradeoffs, and risk in general
Take-Aways

- Be wary of mistaking instances for classes
  - A class represents a class or category of things, not one thing, which is an instance or object

- Composition over subclassing
  - Composition and delegation supercharge extensibility and reuse
  - Allows runtime extension
  - Also called aggregation

- Open/Closed Principle (OCP)
  - Open your classes to extension by following the above principles, then you don’t have to modify them!

- 80/20 rule – a shorthand for tradeoffs and risk