STRUCTURAL DESIGN PATTERNS

CSE 219 (COMPUTER SCIENCE III)
STONY BROOK UNIVERSITY
Travelers know this problem only too well!

- Electronics from one country may not fit power outlet in another country.
- What you need is an “adapter”
THE ADAPTER PATTERN

• Converts the interface of one class into another.
  • Often, it is used to fit an interface to client’s expectations.

• Lets classes work together that would otherwise have incompatible interfaces.

• Say you have an existing system, and you need to work with a vendor library in that system.
  • The vendor interface is different from what you were using before, but you don’t want to change your existing system.
THE ADAPTER PATTERN

Adapter Visualized
Think of a standard Windows driver.

- The existing system uses a driver via an interface.
- But then, there are hardware upgrades, and new hardware uses a different interface.
- The adapter comes in handy!

The existing system “HAS-A” old interface.

- Adapter implements old interface and “HAS-A” new interface.
- Existing system calls old interface methods on adapter, which in turn forwards them to the new interface implementations.
5 ADVANTAGES OF THE ADAPTER PATTERN

• The client is decoupled from the implemented interface.

• If the interface changes over time (think of newer versions) …
  • the adapter encapsulates the changes so that the client can remain completely oblivious to the modifications
  • i.e., the client code doesn’t need any changes in order to work with a different, newer, interface.
public interface Duck { void quack(); void walk(); }

public class MallardDuck implements Duck {
    @Override
    public void quack() {
        System.out.println("Quack... quack...");
    }

    @Override
    public void walk() {
        System.out.println("Walking duck ...");
    }
}

public class Main {
    public static void main(String... args) {
        System.out.println("Duck: ");
        Duck duck = new MallardDuck();
        test(duck);
    }

    static void test(Duck duck) {
        duck.quack();
        duck.walk();
    }
}
public class Turkey {
    public void walk() { System.out.println("Walking turkey ..."); }
    public void gobble() { System.out.println("Gobble gobble!"); }
}

public class TurkeyAdapter implements Duck {
    private Turkey turkey;
    public TurkeyAdapter(Turkey turkey) { this.turkey = turkey; }

    @Override
    public void quack() {
        turkey.gobble();
    }

    @Override
    public void walk() { turkey.walk(); }
}
public class Main {
    public static void main(String... args) {
        System.out.println("Fake Duck: ");
        Duck x = new TurkeyAdapter(new Turkey());
        test(x);
    }

    static void test(Duck duck) {
        duck.quack();
        duck.walk();
    }
}
THE FAÇADE PATTERN

- Provides a unified interface to a set of interfaces in a subsystem.
- The façade defines a higher-level interface that makes the subsystem easier to use
  - by employing the principle of least knowledge.
  - This is also called the Law of Demeter, a design principle for OOP. It allows for loose coupling:
    - Each unit has only limited knowledge about a few other “friend” units.
    - Each unit talks only to friends, not strangers!
    - The communication is essentially on a need-to-know basis.
THE PRINCIPLE OF LEAST KNOWLEDGE

- If you don’t need to know the details of a certain set of communication details
  - just deal with the high-level methods or controls.
EXAMPLE USE CASE: WATCHING A MOVIE

1. Turn on the projector
2. Set projector input to DVD
3. Select wide screen mode on projector
4. Turn on the audio amplifier
5. Set amplifier input to DVD
6. Set amplifier setting to surround sound
7. Set amplifier volume to medium (say, 5 or 6)
8. Turn on DVD player
9. Turn on the popcorn popper
10. Dim the lights
EXAMPLE USE CASE: WATCHING A MOVIE

Now let’s do this programatically:

```java
lights.dim();
popper.on();
popper.pop();
projector.on();
projector.setInput(dvd);
projector.setScreenMode(wide);
amp.on();
amp.setInput(dvd);
amp.setSurroundSound();
amp.setVolume(5);
dvd.on();
dvd.play();
```

There are six different classes involved!

A lot of methods being called before FINALLY playing the movie!

And when the movie is over, you have to all again, in reverse!!

If you upgrade to a better system, you will most probably have to learn a different procedure.
EXAMPLE USE CASE: WATCHING A MOVIE

• With the façade pattern, it’s just
  • watchMovie();
  • endMovie();

• The client (i.e., you, the viewer) has only “friend”, a class called
  • HomeTheaterFacade

• This class manages all those subsystem components.
  • Keeps the client simple.

• The components (e.g., the amplifier) can be upgraded without affecting the client.
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    ...
    public HomeTheaterFacade(Amplifier amp, Tuner tuner, 
                               DvdPlayer dvd, CdPlayer cd, 
                               Projector projector, Screen screen, 
                               TheaterLights lights, ...) {
        ...
    }
    public void watchMovie(Movie movie) { ... }
    public void endMovie() { ... }
}
public void watchMovie(Movie movie) {
    lights.dim(10);
    screen.down();
    projector.on();
    projector.setScreenMode(wide);
    amp.on();
    amp.setInput(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() { ... }
THE FLYWEIGHT PATTERN

• Use sharing to support a large number of objects efficiently.
  • Reduces the number of objects created by reusing pre-existing objects of similar kind.
  • Typically, objects are stored for reuse.
  • New object are created when the objects that is needed is not found to have a similar one in storage.
  • Clearly, this can drastically reduce the memory footprint of an application.
OBJECT STORAGE

• Flyweights are stored in a Factory repository.
• The client doesn’t create flyweights directly.
  • Instead, requests them from a factory.
• Most modern web browsers use the flyweight idea
  • When a page is loaded, the browser traverses through all images in that page, downloads them and stores them in its internal cache.
  • For pages already loaded, a flyweight object is created.
  • This has some unique information (e.g., position)
  • But everything else is referenced to the cached data.
FLYWEIGHT EXAMPLE 2: CIRCLES

• We will draw several circles in different locations using only 5 objects.
  • We will use 5 colors to distinguish between a truly new object and a pre-existing one.
public interface Shape { void draw(); }

public class Circle implements Shape {
    private String color;
    private int x;
    private int y;
    private int radius;

    public Circle(String color) { this.color = color; }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public void setRadius(int radius) { this.radius = radius; }

    @Override
    public void draw() {
        System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius");
    }
}
Create a factory to generate object of concrete class
public class FlyweightPatternDemo {
    private static final String colors[] = {"Red", "Green", "Blue", "White", "Black"};

    public static void main(String... args) {
        for(int i = 0; i < 20; ++i) {
            Circle circle = (Circle) ShapeFactory.getCircle(getRandomColor());
            circle.setX(getRandomX());
            circle.setY(getRandomY());
            circle.setRadius(100);
            circle.draw();
        }
    }

    private static String getRandomColor() {
        return colors[(int) (Math.random() * colors.length)];
    }

    private static int getRandomX() {
        return (int) (Math.random() * 100);
    }

    private static int getRandomY() {
        return (int) (Math.random() * 100);
    }
}

Use the factory to get object of concrete class

• in this example, we are using color