Recitation 10

Principles of Software

Brandon Rozek
rozekb@rpi.edu

Rensselaer Polytechnic Institute, Troy, NY, USA

March 2022

Outline

One Thing:

- Design Patterns

Design Patterns

- A design pattern is a solution to a design problem that commonly occurs in software development.
- Many design patterns have been proposed, we'll go over a few today.
- Many satisfy and are inspired from SOLID.

Design Pattern Categories

There are three categories for design patterns:

- Creational Patterns: Control the creation of objects by setting various criterion.
- Structural Patterns: Combines different classes in order to create larger structures with new functionality.
- Behavioral Patterns: Identifies and realizes common communication patterns between objects.

Creational Patterns

- Builder
- Factory
- Singleton
- Prototype

Builder

This design pattern separates the construction of a complex object from its representation.

```
Channel c = new ChannelBuilder()
    .rxFrequency(14.54)
    .txFrequency(15.2)
    .modulation("FSK")
    .gain(5)
    .build();
```

Builder Source

```
public ChannelBuilder rxFrequency(float freq) {
    this.rxFrequency = freq;
    return this;
}
public Channel build() {
    if (this.rxFrequency == null) { throw new
        IllegalStateException(""); }
    return new Channel(this);
}
```

Factory

This allows you to create objects of a certain supertype without knowing the exact subclass.

```
Scanner userInput = new Scanner(System.in);
System.out.println("Which␣dwelling␣do␣you␣desire?");
String userDream = userInput.nextLine();

DwellingFactory df = new DwellingFactory();
Dwelling d = df.makeDwelling(userDream);
```

Factory Source

```
public abstract class Dwelling { /* ... */ }
public class Apartment extends Dwelling { /* ... */ }
public class House extends Dwelling { /* ... */ }
```

```
public class DwellingFactory {
    public Dwelling makeDwelling(String name) {
        if (name == "Apartment") {
            return new Apartment();
        }
        // ...
        return null; // Default
    }
}
```

Singleton

Ensure a class has only one instance, and provide a global point of access to it.

```
Logger log = Logger.getInstance();
log.write("Testing␣logger");
```

Singleton Source

```
public class Logger {
    private static Logger instance = new Logger();
    private Logger() {}
    public static Logger getInstance() {
        return instance;
    }
    /* ... */
}
```

Questions

1. Do the order of method calls before `build` matter for a builder?
2. What is the benefit of templating over factories?
3. How would you edit the singleton source if you didn't want to eagerly create the instance object?

Structural Patterns

- Adapter / Wrapper / Translator
- Decorator
- Flyweight
- Bridge
- Composite
- Facade
- Proxy

Adapter

```
interface SupportsHDMI { /* ... */ }
interface SupportsUSBC { /* ... */ }

class Projector implements SupportsHDMI { /* ... */ }
class Laptop implements SupportsUSBC { /* ... */ }

class USBCToHDMIAdapter implements SupportsHDMI {
    public USBCToHDMIAdapter(SupportsUSBC laptop) {
        /* ... */
    }
    /* ... */
}
```

Decorator

Adds behavior dynamically to an individual object without affecting the behavior of other objects from the same class.

```
Grapics watermarkedImage = new WatermarkDecorator(new
    Image("profile.png"));
watermarkedImage.draw();
```

Decorator Source

```
public interface Graphics {
   void draw();
}
class WatermarkDecorator implements Graphics {
   private final Graphics graphicsToDecorate;
   @Override
   public void draw() {
       graphicsToDecorate.draw();
       this.drawWatermark();
   }
   /* ... */
}
```

Flyweight

```
class Registry {
    private HashMap<String, Person> people;

    public Person findByName(String name) {
        if (!people.containsKey(name)) {
            people.put(name, Person(name));
        }
        return people.get(name);
    }
}
```

Questions

1. Why would you want an adapter as opposed to using a producer in the class?
2. The decorator example shows the watermark being drawn after the method is ran. Does decorators support running it before?
3. Name a couple benefits of the flyweight design pattern

Behavioral Patterns

- Mediator
- Observer
- Visitor
- Chain of responsibility
- Command
- Iterator
- Interpreter
- Memento
- Strategy

Mediator

Instead of having objects interact directly, a mediator encapsulates how they interact.

```
public class Buyer { /* ... */ }
public class Seller { /* ... */ }

public class Escrow {
    /* ... */
    public void receive_item(Item i) {
        self.item = i;
        if (this.money >= this.asking_price) {
            this.facilitate_transfer();
        }
    }
}
```

Observer

Very popular pattern in *event-driven frameworks*. Allows objects to *subscribe* to certain changes in the system.

```
class Character {
    /* ... */
    void update(String key) {
        if (key == "UP") /* ... */
    }
}
/* ... */
player = Character();
loop.addObserver(player);
```

Observer Source

```
class EventLoop {
    public interface Observer { void update(String e); }
    private final ArrayList<Observer> obs = new
        ArrayList<>();
    private void notifyObservers(String event) {
        obs.forEach(observer -> observer.update(event));
    }
    public void addObserver(Observer observer) {
        obs.add(observer);
    }
    public static void main(String[] args) {
        while (this.app.running()) {
            if (this.keyPressed()) {
                this.notifyObservers(this.keyCode);
            }
        }
    }
}
```

Visitor

Separates out new functionality to a different class.

```
// Create a list of documents we want to export
ArrayList<Document> ds = new ArrayList<>();
ds.add(new Lease());
ds.add(new Agreement());

PDFExporter exporter = new PDFExporter();

// Export each document to PDF
ds.forEach((d) -> d.accept(exporter));
```

Visitor Source

```
public interface Doc {
    public void accept(DocVisitor v);
}
public class Lease implements Document {
    @Override
    public void accept(DocVisitor v) { v.visit(this); }
}
public interface DocVisitor {
    public void visit(Lease l);
    public void visit(Agreement a);
}
public class PDFExporter implements DocVisitor {
    @Override
    public void visit(Lease l) { /* ... */ }
}
```

Questions

1. What is the biggest danger of the mediator pattern?
2. Is the observer pattern a one-to-one, many-to-one, or a many-to-many relationship?
3. What are the differences between the visitor pattern and overloading?

Any Questions?