Poznan University of Technology

Object Oriented Programming
Design Patterns - Java code examples

Jakub Piotr Hamerliński, M.Eng.

Object Oriented Programming

Agenda

01. Design Patterns - Java code examples

> "I do not apologize unless I think I’m wrong. And if you don’t like it you can leave."

Stanley Hudson

Design Patterns

Object Oriented Programming

Design Patterns

List of design patterns we will learn more about:

01. Resource acquisition is initialization aka initialism,
02. Chain of Responsibility,
03. Lazy Initialization,
04. Abstract Factory,
05. Specification,
06. Null Object,
07. Object Pool...

Object Oriented Programming

Design Patterns

And:

01. Composite,
02. Decorator (we skip it since it has been brought already),
03. Strategy,
04. Adapter,
05. Command,
06. Bridge,
07. Proxy.

Object Oriented Programming

RAII

Resource acquisition is initialization aka initialism: Resource allocation (or acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor.

Object Oriented Programming

RAII

```
1   class Permit implements Closeable {
2     private Semaphore sem;
3     Permit(Semaphore s) {
4       this.sem = s;
5     }
6     @Override
7     public void close() {
8       this.sem.release();
9     }
10    public Permit acquire() {
11      this.sem.acquire();
12      return this;
13    }
14  }
```

Object Oriented Programming

RAII

```
class Foo {
  private Semaphore sem = new Semaphore(5);
  void print(int x) throws Exception {
    try (Permit p = new Permit(this.sem).acquire()) {
      if (x > 1000) {
        throw new Exception("Too large!");
      }
      System.out.printf("x = %d", x);
    }
  }
}
```

Object Oriented Programming

Chain of Responsibility

You may transmit requests along a chain of handlers using the behavioral design pattern known as Chain of Responsibility. Each handler chooses whether to execute a request or pass it on to the handler behind it in the chain.

Benefits:

01. The sequence in which a request is handled can be changed.
02. Principle of a single responsibility. Classes that conduct operations and classes that invoke operations can be separated.
03. Principle of Open/Closed. The app can add additional handlers without causing the client code to malfunction.

Object Oriented Programming

Chain of Responsibility

```
abstract class Approve {
  protected Approve nextApprove;
  public void setNextApprove(Approve nextApprove) {
    this.nextApprove = nextApprove;
  }
  public abstract void approve(int amount);
}

class Manager extends Approve {
  public void approve(int amount) {
    if (amount <= 1000) {
      System.out.println("Manager approved $" + amount);
    } else {
      nextApprove.approve(amount);
    }
  }
}
```

Object Oriented Programming

Chain of Responsibility

```
class Director extends Approve {
  public void approve(int amount) {
    if (amount <= 10000) {
      System.out.println("Director approved $" + amount);
    } else {
      nextApprove.approve(amount);
    }
  }
}

class CEO extends Approve {
  public void approve(int amount) {
    if (amount <= 100000) {
      System.out.println("CEO approved $" + amount);
    } else {
      System.out.println("Amount too high, approval denied.");
    }
  }
}
```

Object Oriented Programming

Chain of Responsibility

```
class Main {
  public static void main(String[] args) {
    Approve manager = new Manager();
    Approve director = new Director();
    Approve ceo = new CEO();
    manager.setNextApprove(director);
    director.setNextApprove(ceo);
    manager.approve(500);
    manager.approve(5000);
    manager.approve(50000);
  }
}
```

Object Oriented Programming

Lazy Initialization

Delaying the construction of an object, computing a value, or engaging in any other computationally expensive activity until it is to be utilized for the first time is a strategy called lazy initialization. Benefits:

01. reduces startup time by spreading out complicated computations over time.
02. avoids conducting infrequent operations unless absolutely essential.

Object Oriented Programming

Lazy Initialization

```
class Response {
  private static Response response;
  private Response() {}
  public static Response instance() {
    if (response == null) {
      response = new Response();
    }
    return response;
  }
}
```

In this example, the Response class uses the Lazy Initialization design pattern to create only one instance of the class when it is needed. The instance variable is initially set to null and is only created using the new operator when the instance() method is called for the first time. This ensures that the instance is only created when it is needed, and not before.

Object Oriented Programming

Abstract Factory

A creational design pattern called Abstract Factory enables you to create families of linked things without identifying their particular classes.

```
interface Button {
  void paint();
}

class MacOSButton implements Button {
  @Override
  public void paint() {
    System.out.println("You have created MacOSButton.");
  }
}
```

Object Oriented Programming

Abstract Factory

```
1   public class WindowsButton implements Button {
2     @Override
3     public void paint() {
4       System.out.println("You have created WindowsButton.");
5     }
6   }
7
8   public interface GUIFactory {
9     Button createButton();
10  }
```

Object Oriented Programming

Abstract Factory

```
public class MacOSFactory implements GUIFactory {
  @Override
  public Button createButton() {
    return new MacOSButton();
  }
}

public class WindowsFactory implements GUIFactory {
  @Override
  public Button createButton() {
    return new WindowsButton();
  }
}
```

Object Oriented Programming

Abstract Factory

```
public class Application {
  private Button button;
  public Application(GUIFactory factory) {
    button = factory.createButton();
  }
  public void paint() {
    button.paint();
  }
}
```

Object Oriented Programming

Abstract Factory

```
public class Demo {
  private static Application configureApplication() {
    Application app;
    GUIFactory factory;
    String osName = System.getProperty("os.name").toLowerCase();
    if (osName.contains("mac")) {
      factory = new MacOSFactory();
    } else {
      factory = new WindowsFactory();
    }
    app = new Application(factory);
    return app;
  }

  public static void main(String[] args) {
    Application app = configureApplication();
    app.paint();
  }
}
```

Object Oriented Programming

Specification

The Specification design pattern is a pattern that allows for the creation of complex logical expressions, made up of simpler expressions, that can be used to filter a collection of objects. It separates the logic of filtering objects from the objects themselves, making the code more reusable and readable.

The Specification pattern defines two main components: the Specification interface and the Specification implementations. The Specification interface defines the methods that need to be implemented by the concrete classes, such as isSatisfiedBy() and and(). The concrete classes implement these methods and define the logic for filtering the objects.

Object Oriented Programming

Specification

```
interface Specification<T> {
  boolean isSatisfiedBy(T item);
  Specification<T> and(Specification<T> other);
}
class PriceSpecification implements Specification<Product> {
  private final double minPrice;
  private final double maxPrice;
  public PriceSpecification(double minPrice, double maxPrice) {
    this.minPrice = minPrice;
    this.maxPrice = maxPrice;
  }
  @Override
  public boolean isSatisfiedBy(Product item) {
    return item.currentPrice() >= minPrice && item.currentPrice() <= maxPrice;
  }
  @Override
  public Specification<Product> and(Specification<Product> other) {
    return new AndSpecification<>(this, other);
  }
}
```

Object Oriented Programming

Specification

```
class Product {
  private final double price;
  public Product(double price) { this.price = price; }
  public double currentPrice() { return price; }
}
class AndSpecification<T> implements Specification<T> {
  private final Specification<T> first;
  private final Specification<T> second;
  public AndSpecification(Specification<T> first, Specification<T> second) {
    this.first = first;
    this.second = second;
  }
  @Override
  public boolean isSatisfiedBy(T item) {
    return first.isSatisfiedBy(item) && second.isSatisfiedBy(item);
  }
  @Override
  public Specification<T> and(Specification<T> other) {
    return new AndSpecification<>(this, other);
  }
}
```

Object Oriented Programming

Null Object

The Null Object pattern is a pattern that allows for the handling of null values in an object-oriented way. Instead of using null values, the pattern uses a special Null object that implements the same interface as the actual objects, but has no implementation for its methods. This allows for a consistent way of handling null values without having to check for null values everywhere in the code.

Object Oriented Programming

Null Object

```java
interface Shape {
  void draw();
}
class Rectangle implements Shape {
  @Override
  public void draw() {
    System.out.println("Drawing a rectangle.");
  }
}
class NullShape implements Shape {
  @Override
  public void draw() {
    // Do nothing
  }
}
```

Object Oriented Programming

Null Object

```
class ShapeFactory {

  public Shape createShape(String shapeType) {
    if (shapeType == null) { return new NullShape(); }
    if (shapeType.equals("rectangle")) { return new Rectangle(); }
    return new NullShape();
  }
}
class Main {
  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();
    Shape shape1 = shapeFactory.createShape("rectangle");
    shape1.draw(); // Outputs "Drawing a rectangle."
    Shape shape2 = shapeFactory.createShape(null);
    shape2.draw(); // Does nothing
  }
}
```

Object Oriented Programming

Object Pool

The Object Pool pattern is a software creational pattern that allows for the reuse of objects that are expensive to create. Instead of creating a new object every time one is needed, the object pool maintains a pool of objects that can be reused. When a client requests an object, the pool checks if it has an available object to give, and if so, it returns it. If not, it creates a new object and adds it to the pool.

Object Oriented Programming

Object Pool

```
interface PooledObject {
  void reset();
}
class Connection implements PooledObject {
  private boolean inUse = false;
  @Override
  public void reset() {
    // code to reset the connection
  }
  public boolean used() { return inUse; }
  public void changeUsage(boolean inUse) { this.inUse = inUse; }
}
```

Object Oriented Programming

Object Pool

```
class ConnectionPool {
  private List<Connection> availableConnections = new ArrayList<>();
  private List<Connection> inUseConnections = new ArrayList<>();
  public Connection createdConnection() {
    Connection connection;
    if (availableConnections.isEmpty()) {
      connection = new Connection(); // create a new connection
    } else {
      connection = availableConnections.remove(0); // return an available connection
    }
    inUseConnections.add(connection);
    return connection;
  }
  public void releaseConnection(Connection connection) {
    connection.reset();
    inUseConnections.remove(connection);
    availableConnections.add(connection);
  }
}
```

Object Oriented Programming

Object Pool

In this example, the Connection class implements the PooledObject interface, and it defines the reset method that will be used when the connection is released back to the pool. The ConnectionPool class is responsible for maintaining a pool of connections and providing them to the client when requested. When a client requests a connection, the pool checks if there are available connections, and if not it creates a new one. When a client releases a connection, the pool resets it and adds it back to the available connections list.

Object Oriented Programming

Composite

With the use of the structural design pattern known as Composite, you can group things into tree structures and then treat those structures like individual objects. Use polymorphism and recursion to your advantage to work with complex tree structures more easily. Principle of Open/Closed. The existing code can continue to function with the object tree without being affected by the addition of additional element types.

Object Oriented Programming

Composite

```
interface Component {
  void operation();
}
class Leaf implements Component {
  @Override
  public void operation() {
    // code for leaf operation
  }
}
```

Object Oriented Programming

Composite

```
class Composite implements Component {
  private List<Component> children = new ArrayList<>();
  @Override
  public void operation() {
    for (Component child : children) {
      child.operation();
    }
  }
  public void add(Component child) {
    children.add(child);
  }
  public void remove(Component child) {
    children.remove(child);
  }
  public Component child(int index) {
    return children.get(index);
  }
}
```

Object Oriented Programming

Composite

```
class Client {
  public void someMethod() {
    Composite composite = new Composite();
    composite.add(new Leaf());
    composite.add(new Leaf());
    composite.operation();
  }
}
```

Object Oriented Programming

Composite

In this example, the Component interface defines the operation() method that must be implemented by both the Leaf and Composite classes. The Leaf class represents the leaf objects in the tree structure, and it implements the operation() method with its specific behavior. The Composite class represents the composite objects in the tree structure, and it implements the operation() method by iterating over its children and calling the operation() method on each of them. The Client class uses the composite and leaf objects in the same way, by calling the operation() method on them.

Object Oriented Programming

Strategy

The Strategy design pattern is a behavioral pattern that allows for the selection of an algorithm at runtime. It defines a family of algorithms, encapsulates each one, and makes them interchangeable. It also allows the client to choose the appropriate algorithm without having to know its implementation details.

Object Oriented Programming

Strategy

```java
interface PaymentStrategy {
  void pay(double amount);
}

class CreditCardStrategy implements PaymentStrategy {
  private String name;
  private String cardNumber;
  private String cvv;
  private String dateOfExpiry;
  public CreditCardStrategy(String name, String cardNumber, String cvv, String dateOfExpiry) {
    this.name = name;
    this.cardNumber = cardNumber;
    this.cvv = cvv;
    this.dateOfExpiry = dateOfExpiry;
  }
  @Override
  public void pay(double amount) {
    System.out.println(amount + " paid with credit/debit card");
  }
}
```

Object Oriented Programming

Strategy

```
class PayPalStrategy implements PaymentStrategy {
  private String emailId;
  private String password;
  public PayPalStrategy(String emailId, String password) {
    this.emailId = emailId;
    this.password = password;
  }
  @Override
  public void pay(double amount) {
    System.out.println(amount + " paid using PayPal.");
  }
}
```

Object Oriented Programming

Strategy

```
class ShoppingCart {
  List<Item> items;
  PaymentStrategy paymentMethod;
  public ShoppingCart(List<Item> items, PaymentStrategy paymentMethod) {
    this.items = items;
    this.paymentMethod = paymentMethod;
  }
  public void pay() {
    double totalAmount = calculateTotal();
    paymentMethod.pay(totalAmount);
  }
  private double calculateTotal() {
    double total = 0;
    for (Item item : items) {
      total += item.price();
    }
    return total;
  }
}
```

Object Oriented Programming

Strategy

In this example, the PaymentStrategy interface defines the pay method that must be implemented by the concrete classes (CreditCardStrategy and PayPalStrategy). The ShoppingCart class uses the PaymentStrategy interface to allow the client to choose the appropriate payment method at runtime, and it doesn't need to know the implementation details. It allows the client to use different Payment strategies that implements the PaymentStrategy interface and use the pay method accordingly. You can also add more Payment strategies like BankTransfer, Wallet etc. and use them in the ShoppingCart class.

Object Oriented Programming

Adapter

The Adapter design pattern is a structural pattern that allows the adaptation of an existing interface to a different one that the client expects. It helps to integrate new systems and legacy systems by allowing them to work together seamlessly.

Object Oriented Programming

Adapter

```
interface LegacySystem {
  void processData(String data);
}
class LegacySystemImpl implements LegacySystem {
  @Override
  public void processData(String data) {
    // code to process data using the legacy system
  }
}
```

Object Oriented Programming

Adapter

```
interface NewSystem {
  void process(String data);
}
class NewSystemAdapter implements NewSystem {
  private LegacySystem legacySystem;
  public NewSystemAdapter(LegacySystem legacySystem) {
    this.legacySystem = legacySystem;
  }
  @Override
  public void process(String data) {
    legacySystem.processData(data);
  }
}
```

Object Oriented Programming

Adapter

```
class Client {
  public static void main(String[] args) {
    LegacySystem legacySystem = new LegacySystemImpl();
    NewSystem newSystem = new NewSystemAdapter(legacySystem);
    newSystem.process("Some data");
  }
}
```

Object Oriented Programming

Command

The Command design pattern is a behavioral pattern that allows for the encapsulation of a request as an object, separate from the object that actually carries out the request. This allows for decoupling the objects that initiate the request from the objects that execute it.

Object Oriented Programming

Command

```
interface Command {
  void execute();
}
class Light {
  public void turnOn() {
    System.out.println("The light is on");
  }
  public void turnOff() {
    System.out.println("The light is off");
  }
}
```

Object Oriented Programming

Command

```
class TurnOnLightCommand implements Command {
  private Light light;
  public TurnOnLightCommand(Light light) {
    this.light = light;
  }
  @Override
  public void execute() {
    light.turnOn();
  }
}
```

Object Oriented Programming

Command

```
class TurnOffLightCommand implements Command {
  private Light light;
  public TurnOffLightCommand(Light light) {
    this.light = light;
  }
  @Override
  public void execute() {
    light.turnOff();
  }
}
```

Object Oriented Programming

Command

```
class RemoteControl {
  private Command command;
  public void changeCommand(Command command) {
    this.command = command;
  }
  public void pressButton() {
    command.execute();
  }
}
```

Object Oriented Programming

Command

```
public class CommandPatternDemo {
  public static void main(String[] args) {
    Light light = new Light();
    Command turnOnCommand = new TurnOnLightCommand(light);
    Command turnOffCommand = new TurnOffLightCommand(light);
    RemoteControl remote = new RemoteControl();
    remote.changeCommand(turnOnCommand);
    remote.pressButton(); // prints "The light is on"
    remote.changeCommand(turnOffCommand);
    remote.pressButton(); // prints "The light is off"
  }
}
```

Object Oriented Programming

Command

In this example, the Light class is the object that actually carries out the request, while the TurnOnLightCommand and TurnOffLightCommand classes are the objects that encapsulate the request and implement the Command interface. The RemoteControl class is the object that initiates the request, it holds a command and when the pressButton method is called, it calls the execute method of the command. This example shows how the command pattern allows for decoupling the objects that initiate the request from the objects that execute it.

Object Oriented Programming

Bridge

The Bridge design pattern is a structural pattern that allows for the separation of an abstraction from its implementation. It allows for the decoupling of the interface and the implementation, so that they can evolve independently.

Object Oriented Programming

Bridge

```
interface RemoteControl {
  void turnOn();
  void turnOff();
  void changeChannel(int channel);
}
```

Object Oriented Programming

Bridge

```
class BasicRemoteControl implements RemoteControl {
  protected TV tv;
  public BasicRemoteControl(TV tv) {
    this.tv = tv;
  }
  @Override
  public void turnOn() {
    tv.on();
  }
  @Override
  public void turnOff() {
    tv.off();
  }
  @Override
  public void changeChannel(int channel) {
    tv.tuneChannel(channel);
  }
}
```

Object Oriented Programming

Bridge

```
class AdvancedRemoteControl extends BasicRemoteControl {
  public AdvancedRemoteControl(TV tv) {
    super(tv);
  }
  public void changeVolume(int volume) {
    tv.changeVolume(volume);
  }
}
interface TV {
  void on();
  void off();
  void tuneChannel(int channel);
  void changeVolume(int volume);
}
```

Object Oriented Programming

Bridge

```
class LGTV implements TV {
  @Override
  public void on() {
    System.out.println("LG TV is turned on.");
  }
  @Override
  public void off() {
    System.out.println("LG TV is turned off.");
  }
  @Override
  public void tuneChannel(int channel) {
    System.out.println("LG TV tuned to channel: " + channel);
  }
  @Override
  public void changeVolume(int volume) {
    System.out.println("LG TV volume set to: " + volume);
  }
}
```

Object Oriented Programming

Bridge

```
class SonyTV implements TV {
  @Override
  public void on() {
    System.out.println("Sony TV is turned on.");
  }
  @Override
  public void off() {
    System.out.println("Sony TV is turned off.");
  }
  @Override
  public void tuneChannel(int channel) {
    System.out.println("Sony TV tuned to channel: " + channel);
  }
  @Override
  public void changeVolume(int volume) {
    System.out.println("Sony TV volume set to: " + volume);
  }
}
```

Object Oriented Programming

Bridge

In this example, the RemoteControl interface defines the basic operations that can be performed on a TV. The BasicRemoteControl class provides a basic implementation of these operations, delegating the actual work to the TV object. The AdvancedRemoteControl class extends the basic functionality by adding the ability to set the volume. The TV interface defines the basic operations that a TV must implement, and the LGTV and SonyTV classes are concrete implementations of this interface. The client can use the RemoteControl interface to interact with different TV implementations without having to know the details of their specific implementations.

Object Oriented Programming

Proxy

The Proxy design pattern is a structural pattern that provides a surrogate or placeholder object that controls access to the original object. The proxy object controls access to the original object, and it can add additional functionality, such as caching, logging, or security checks.

Object Oriented Programming

Proxy

```
interface Image {
  void display();
}
class RealImage implements Image {
  private final String fileName;
  public RealImage(String fileName) {
    this.fileName = fileName;
    loadFromDisk(fileName);
  }
  @Override
  public void display() { System.out.println("Displaying " + fileName); }
  private void loadFromDisk(String fileName) { System.out.println("Loading " + fileName); }
}
```

Object Oriented Programming

Proxy

```
class ImageProxy implements Image {
  private final String fileName;
  private RealImage realImage;
  public ImageProxy(String fileName) {
    this.fileName = fileName;
  }
  @Override
  public void display() {
    if (realImage == null) { realImage = new RealImage(fileName); }
    realImage.display();
  }
}
class Client {
  public static void main(String[] args) {
    Image image = new ImageProxy("test_10mb.jpg");
    image.display();
    System.out.println("");
    image.display();
  }
}
```

Object Oriented Programming

Proxy

In this example, the RealImage class is the actual object that will be displayed, and the ImageProxy class is the proxy that controls access to the RealImage object. The Client class requests the image to be displayed, but the actual image is only loaded from disk when the display() method is called on the ImageProxy object. This allows for lazy loading, where the image is only loaded when it's needed, which can be useful when working with large files or slow networks.

Additionally, the proxy can be used to add additional functionality, such as caching, logging or security checks before accessing the real image, this is the main purpose for having a proxy object.

Sources

Object Oriented Programming

Based on:

01. https://www.yegor256.com/2016/02/03/design-patterns-and-anti-patterns.html
02. https://www.yegor256.com/2017/08/08/raii-in-java.html
03. https://refactoring.guru/design-patterns

Thank you

Feel free to reach me via LinkedIn

Fin