CEN206 Object-Oriented Programming

Week-10 (OO Design Patterns - Advanced Concepts)

Spring Semester, 2024-2025

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX

OO Design Patterns - Advanced Concepts

Outline

- More Creational Patterns
- More Structural Patterns
- More Behavioral Patterns
- Anti-Patterns
- Design Pattern Selection Criteria

Singleton Pattern

Ensures a class has only one instance and provides a global point of access to it.

```java
public class Singleton {
    // Private static instance
    private static Singleton instance;

    // Private constructor to prevent instantiation
    private Singleton() {}

    // Public method to get the instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Thread-safe version
    public static synchronized Singleton getThreadSafeInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Double-checked locking
    public static Singleton getDCLInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

```java
// Product
class Pizza {
    private String dough;
    private String sauce;
    private String topping;

    public void setDough(String dough) { this.dough = dough; }
    public void setSauce(String sauce) { this.sauce = sauce; }
    public void setTopping(String topping) { this.topping = topping; }

    @Override
    public String toString() {
        return "Pizza with " + dough + " dough, " + sauce + " sauce, and " + topping + " topping";
    }
}

// Abstract Builder
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() { return pizza; }
    public void createNewPizza() { pizza = new Pizza(); }

    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}

// Concrete Builder
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("cross"); }
    public void buildSauce() { pizza.setSauce("mild"); }
    public void buildTopping() { pizza.setTopping("ham and pineapple"); }
}

// Director
class Cook {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pizzaBuilder) {
        this.pizzaBuilder = pizzaBuilder;
    }

    public Pizza getPizza() { return pizzaBuilder.getPizza(); }

    public void constructPizza() {
        pizzaBuilder.createNewPizza();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

Adapter Pattern

Allows incompatible interfaces to work together by wrapping an instance of one class into an adapter that conforms to another class's interface.

```java
// Target interface
interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Adaptee interface
interface AdvancedMediaPlayer {
    void playVlc(String fileName);
    void playMp4(String fileName);
}

// Concrete Adaptee
class VlcPlayer implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file: " + fileName);
    }

    @Override
    public void playMp4(String fileName) {
        // Do nothing
    }
}

// Adapter
class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer = new VlcPlayer();
        }
        // Add other players as needed
    }

    @Override
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer.playVlc(fileName);
        }
        // Handle other formats
    }
}

// Client
class AudioPlayer implements MediaPlayer {
    private MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {
        // Built-in support for mp3
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing mp3 file: " + fileName);
        }
        // MediaAdapter provides support for other formats
        else if (audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")) {
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        } else {
```

Decorator Pattern

Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

```java
// Component interface
interface Coffee {
    double getCost();
    String getDescription();
}

// Concrete Component
class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1.0;
    }

    @Override
    public String getDescription() {
        return "Simple coffee";
    }
}

// Abstract Decorator
abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
}

// Concrete Decorator
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 0.5;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", milk";
    }
}

// Usage
// Coffee myCoffee = new SimpleCoffee();
// myCoffee = new MilkDecorator(myCoffee);
// System.out.println(myCoffee.getDescription() + " $" + myCoffee.getCost());
```

Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```java
import java.util.ArrayList;
import java.util.List;

// Observer interface
interface Observer {
    void update(String message);
}

// Subject
class Subject {
    private final List<Observer> observers = new ArrayList<>();
    private String state;

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
        notifyAllObservers();
    }

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }
}

// Concrete Observer
class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}

// Usage
// Subject subject = new Subject();
// Observer observer1 = new ConcreteObserver("Observer 1");
// Observer observer2 = new ConcreteObserver("Observer 2");
```

Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

```java
// Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
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(int amount) {
        System.out.println(amount + " paid with credit card");
    }
}

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(int amount) {
        System.out.println(amount + " paid using PayPal");
    }
}

// Context
class ShoppingCart {
    private List<Item> items;

    public ShoppingCart() {
        this.items = new ArrayList<Item>();
    }

    public void addItem(Item item) {
        this.items.add(item);
    }

    public int calculateTotal() {
        int sum = 0;
        for (Item item : items) {
            sum += item.getPrice();
        }
        return sum;
    }

    public void pay(PaymentStrategy paymentStrategy) {
        int amount = calculateTotal();
        paymentStrategy.pay(amount);
    }
}
```

Anti-Patterns

Anti-patterns are common solutions to recurring problems that tend to be ineffective and risky.

Common Anti-Patterns

- **God Object**: A class that knows or does too much
- **Spaghetti Code**: Unstructured and difficult-to-maintain code
- **Singleton Abuse**: Overusing the Singleton pattern
- **Golden Hammer**: Using a familiar solution regardless of the problem
- **Reinventing the Wheel**: Creating custom solutions when standard ones exist
- **Premature Optimization**: Optimizing before identifying bottlenecks
- **Copy-Paste Programming**: Duplicating code instead of reusing it

Design Pattern Selection Criteria

When choosing a design pattern, consider:

1. **Problem Context**: What specific problem are you trying to solve?
2. **Pattern Consequences**: What are the trade-offs of using this pattern?
3. **Alternative Patterns**: Are there other patterns that could address this problem?
4. **Implementation Language**: Some patterns are more natural in certain languages
5. **Team Familiarity**: Is your team familiar with the pattern?
6. **Maintainability**: Will the pattern make the code more maintainable?
7. **Performance Concerns**: Will the pattern impact performance?

Applying Patterns in Real Projects

Best Practices

- Don't force design patterns where they don't fit
- Start simple, refactor to patterns when needed
- Document why you chose a particular pattern
- Consider the entire system, not just individual components
- Pattern combinations can be more powerful than individual patterns
- Test pattern implementations thoroughly

Secure Design Patterns

Security should be a fundamental consideration in software design.

Key security design patterns include:

- **Secure Factory**: Centralize object creation with security checks
- **Secure Proxy**: Control access to sensitive objects
- **Secure Singleton**: Ensure secure access to single instances
- **Intercepting Validator**: Validate all input through central validators

More security controls: https://www.cisecurity.org/controls/cis-controls-list https://www.amazon.com/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199

Gang of Four (GoF) Design Patterns Book:

https://www.amazon.com/gp/product/0201633612/

SOLID Principles Resources:

- https://www.monterail.com/blog/solid-principles-cheatsheet-printable
- https://www.monterail.com/hubfs/PDF content/SOLID_cheatsheet.pdf
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

Liskov Substitution Principle Examples:

https://code-examples.net/en/q/a476f2

Dependency Injection Resources:

- http://www.dotnet-stuff.com/tutorials/dependency-ınjection/understanding-and-implementing-inversion-of-control-container-ioc-container-using-csharp
- https://stackify.com/dependency-injection/

Next Week

We'll continue with UML and UMPLE, focusing on modeling our designs and generating code from models.