Software design patterns are established solutions to common design problems that software developers face. They are templates or best practices for designing software architectures and solving issues in a way that is both effective and reusable. Here’s a detailed overview of some of the most important design patterns, categorized into three main types: \*\*Creational\*\*, \*\*Structural\*\*, and \*\*Behavioral\*\*.

### 1. \*\*Creational Design Patterns\*\*

Creational patterns focus on how objects are created. They abstract the instantiation process, making it more flexible and efficient.

- \*\*Singleton\*\*

- \*\*Purpose\*\*: Ensures that a class has only one instance and provides a global point of access to it.

- \*\*Example\*\*: A configuration manager that reads configuration settings from a file.

- \*\*Example Code (Java)\*\*:

```java

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

```

- \*\*Factory Method\*\*

- \*\*Purpose\*\*: Defines an interface for creating objects, but allows subclasses to alter the type of objects that will be created.

- \*\*Example\*\*: A document creation application where the type of document (Word, PDF, etc.) is decided at runtime.

- \*\*Example Code (Java)\*\*:

```java

abstract class Document {

abstract void create();

}

class WordDocument extends Document {

@Override

void create() {

System.out.println("Creating a Word document.");

}

}

class DocumentFactory {

public Document createDocument(String type) {

if (type.equals("Word")) {

return new WordDocument();

}

// Additional types can be added here

return null;

}

}

```

- \*\*Abstract Factory\*\*

- \*\*Purpose\*\*: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

- \*\*Example\*\*: Creating user interfaces with different themes (light mode, dark mode).

- \*\*Example Code (Java)\*\*:

```java

interface GUIFactory {

Button createButton();

Checkbox createCheckbox();

}

class WinFactory implements GUIFactory {

public Button createButton() { return new WinButton(); }

public Checkbox createCheckbox() { return new WinCheckbox(); }

}

class MacFactory implements GUIFactory {

public Button createButton() { return new MacButton(); }

public Checkbox createCheckbox() { return new MacCheckbox(); }

}

```

- \*\*Builder\*\*

- \*\*Purpose\*\*: Separates the construction of a complex object from its representation so that the same construction process can create different representations.

- \*\*Example\*\*: Building a complex meal with different combinations of dishes.

- \*\*Example Code (Java)\*\*:

```java

class Meal {

private String mainCourse;

private String drink;

public void setMainCourse(String mainCourse) { this.mainCourse = mainCourse; }

public void setDrink(String drink) { this.drink = drink; }

}

abstract class MealBuilder {

protected Meal meal = new Meal();

public abstract void buildMainCourse();

public abstract void buildDrink();

public Meal getMeal() { return meal; }

}

class VegMealBuilder extends MealBuilder {

public void buildMainCourse() { meal.setMainCourse("Vegetarian Burger"); }

public void buildDrink() { meal.setDrink("Lemonade"); }

}

```

- \*\*Prototype\*\*

- \*\*Purpose\*\*: Creates new objects by copying an existing object, known as the prototype.

- \*\*Example\*\*: Copying objects with default settings for a new configuration.

- \*\*Example Code (Java)\*\*:

```java

interface Prototype {

Prototype clone();

}

class ConcretePrototype implements Prototype {

@Override

public Prototype clone() {

return new ConcretePrototype();

}

}

```

### 2. \*\*Structural Design Patterns\*\*

Structural patterns focus on how objects and classes are composed to form larger structures.

- \*\*Adapter (or Wrapper)\*\*

- \*\*Purpose\*\*: Allows incompatible interfaces to work together.

- \*\*Example\*\*: Adapting a legacy system interface to a new system.

- \*\*Example Code (Java)\*\*:

```java

interface Target {

void request();

}

class Adaptee {

void specificRequest() {

System.out.println("Specific request.");

}

}

class Adapter implements Target {

private Adaptee adaptee;

public Adapter(Adaptee adaptee) { this.adaptee = adaptee; }

public void request() { adaptee.specificRequest(); }

}

```

- \*\*Decorator\*\*

- \*\*Purpose\*\*: Adds new functionality to an object without altering its structure.

- \*\*Example\*\*: Adding scroll bars to a window.

- \*\*Example Code (Java)\*\*:

```java

interface Window {

void draw();

}

class SimpleWindow implements Window {

public void draw() {

System.out.println("Drawing a simple window.");

}

}

abstract class WindowDecorator implements Window {

protected Window decoratedWindow;

public WindowDecorator(Window decoratedWindow) { this.decoratedWindow = decoratedWindow; }

public void draw() { decoratedWindow.draw(); }

}

class ScrollableWindow extends WindowDecorator {

public ScrollableWindow(Window decoratedWindow) { super(decoratedWindow); }

public void draw() {

super.draw();

System.out.println("Adding scroll bars.");

}

}

```

- \*\*Composite\*\*

- \*\*Purpose\*\*: Allows clients to treat individual objects and compositions of objects uniformly.

- \*\*Example\*\*: A file system where files and directories are treated similarly.

- \*\*Example Code (Java)\*\*:

```java

interface Component {

void operation();

}

class Leaf implements Component {

public void operation() {

System.out.println("Leaf operation.");

}

}

class Composite implements Component {

private List<Component> children = new ArrayList<>();

public void add(Component component) { children.add(component); }

public void operation() {

for (Component child : children) {

child.operation();

}

}

}

```

- \*\*Facade\*\*

- \*\*Purpose\*\*: Provides a simplified interface to a complex subsystem.

- \*\*Example\*\*: A simplified API for a complex library.

- \*\*Example Code (Java)\*\*:

```java

class SubsystemA {

void operationA() { System.out.println("Subsystem A operation."); }

}

class SubsystemB {

void operationB() { System.out.println("Subsystem B operation."); }

}

class Facade {

private SubsystemA a = new SubsystemA();

private SubsystemB b = new SubsystemB();

public void performOperation() {

a.operationA();

b.operationB();

}

}

```

- \*\*Bridge\*\*

- \*\*Purpose\*\*: Decouples an abstraction from its implementation so that the two can vary independently.

- \*\*Example\*\*: Drawing different shapes (circle, square) in different colors.

- \*\*Example Code (Java)\*\*:

```java

interface DrawingAPI {

void drawCircle(int x, int y, int radius);

}

class ConcreteDrawingAPI1 implements DrawingAPI {

public void drawCircle(int x, int y, int radius) {

System.out.println("Drawing API 1: Circle at (" + x + ", " + y + ") with radius " + radius);

}

}

class Circle {

private int x, y, radius;

private DrawingAPI drawingAPI;

public Circle(int x, int y, int radius, DrawingAPI drawingAPI) {

this.x = x; this.y = y; this.radius = radius; this.drawingAPI = drawingAPI;

}

public void draw() {

drawingAPI.drawCircle(x, y, radius);

}

}

```

- \*\*Proxy\*\*

- \*\*Purpose\*\*: Provides a surrogate or placeholder for another object.

- \*\*Example\*\*: A proxy that manages access to a resource-heavy object.

- \*\*Example Code (Java)\*\*:

```java

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) { this.filename = filename; }

public void display() { System.out.println("Displaying " + filename); }

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) { this.filename = filename; }

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

```

### 3. \*\*Behavioral Design Patterns\*\*

Behavioral patterns focus on communication between objects and how responsibilities are distributed.

- \*\*Chain of Responsibility\*\*

- \*\*Purpose\*\*: Passes a request along a chain of potential handlers until one of them handles it.

- \*\*Example\*\*: A help desk where requests are escalated through different levels.

- \*\*Example Code (Java)\*\*:

```java

abstract class Handler {

private Handler next;

public void setNext(Handler next) { this.next = next; }

public void handleRequest(int request) {

if (next != null) {

next.handleRequest(request);

}

}

}

class ConcreteHandlerA extends Handler {

public void handleRequest(int request) {

if (request < 10) {

System.out.println("Handler A handled request " + request);

} else {

super.handleRequest(request);

}