Design patterns are proven solutions to common problems faced during software development. They can be categorized into three types: Creational, Structural, and Behavioral. Below are some of the most widely used design patterns in each category, along with brief descriptions and examples in Java.

**Creational Patterns**

1. **Singleton**
   * **Description**: Ensures that a class has only one instance and provides a global point of access to it.
   * **Example**:

java

* 
* public class Singleton {
* private static Singleton instance;
* private Singleton() {}
* public static Singleton getInstance() {
* if (instance == null) {
* instance = new Singleton();
* }
* return instance;
* }
* }

 **Factory Method**

* **Description**: Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.
* **Example**:

java

* 
* interface Product {
* void use();
* }
* class ConcreteProductA implements Product {
* public void use() {
* System.out.println("Using Product A");
* }
* }
* class ConcreteProductB implements Product {
* public void use() {
* System.out.println("Using Product B");
* }
* }
* abstract class Creator {
* public abstract Product factoryMethod();
* }
* class ConcreteCreatorA extends Creator {
* public Product factoryMethod() {
* return new ConcreteProductA();
* }
* }
* class ConcreteCreatorB extends Creator {
* public Product factoryMethod() {
* return new ConcreteProductB();
* }
* }

 **Abstract Factory**

* **Description**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
* **Example**:

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();
* }
* }
* interface Button {
* void paint();
* }
* class WinButton implements Button {
* public void paint() {
* System.out.println("Windows Button");
* }
* }
* class MacButton implements Button {
* public void paint() {
* System.out.println("Mac Button");
* }
* }
* interface Checkbox {
* void paint();
* }
* class WinCheckbox implements Checkbox {
* public void paint() {
* System.out.println("Windows Checkbox");
* }
* }
* class MacCheckbox implements Checkbox {
* public void paint() {
* System.out.println("Mac Checkbox");
* }
* }

 **Builder**

* **Description**: Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.
* **Example**:

java

* 
* public class Product {
* private String partA;
* private String partB;
* public void setPartA(String partA) {
* this.partA = partA;
* }
* public void setPartB(String partB) {
* this.partB = partB;
* }
* }
* abstract class Builder {
* protected Product product = new Product();
* public abstract void buildPartA();
* public abstract void buildPartB();
* public Product getResult() {
* return product;
* }
* }
* class ConcreteBuilder extends Builder {
* public void buildPartA() {
* product.setPartA("Part A");
* }
* public void buildPartB() {
* product.setPartB("Part B");
* }
* }
* class Director {
* public void construct(Builder builder) {
* builder.buildPartA();
* builder.buildPartB();
* }
* }

 **Prototype**

* **Description**: Specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype.
* **Example**:

java

* + abstract class Prototype implements Cloneable {
  + public Prototype clone() throws CloneNotSupportedException {
  + return (Prototype) super.clone();
  + }
  + }
  + class ConcretePrototype extends Prototype {
  + private String field;
  + public void setField(String field) {
  + this.field = field;
  + }
  + public String getField() {
  + return field;
  + }
  + }

**Structural Patterns**

1. **Adapter**
   * **Description**: Converts the interface of a class into another interface clients expect, allowing classes to work together that couldn't otherwise because of incompatible interfaces.
   * **Example**:

java

* 
* interface Target {
* void request();
* }
* class Adaptee {
* public 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();
* }
* }

 **Bridge**

* **Description**: Decouples an abstraction from its implementation so that the two can vary independently.
* **Example**:

java

* 
* interface Implementor {
* void operationImpl();
* }
* class ConcreteImplementorA implements Implementor {
* public void operationImpl() {
* System.out.println("Implementation A");
* }
* }
* class ConcreteImplementorB implements Implementor {
* public void operationImpl() {
* System.out.println("Implementation B");
* }
* }
* abstract class Abstraction {
* protected Implementor implementor;
* public Abstraction(Implementor implementor) {
* this.implementor = implementor;
* }
* public abstract void operation();
* }
* class RefinedAbstraction extends Abstraction {
* public RefinedAbstraction(Implementor implementor) {
* super(implementor);
* }
* public void operation() {
* implementor.operationImpl();
* }
* }

 **Composite**

* **Description**: Composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.
* **Example**:

java

* 
* interface Component {
* void operation();
* }
* class Leaf implements Component {
* public void operation() {
* System.out.println("Leaf");
* }
* }
* class Composite implements Component {
* private List<Component> children = new ArrayList<>();
* public void add(Component component) {
* children.add(component);
* }
* public void remove(Component component) {
* children.remove(component);
* }
* public void operation() {
* for (Component child : children) {
* child.operation();
* }
* }
* }

 **Decorator**

* **Description**: Attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality.
* **Example**:

java

* 
* interface Component {
* void operation();
* }
* class ConcreteComponent implements Component {
* public void operation() {
* System.out.println("ConcreteComponent");
* }
* }
* abstract class Decorator implements Component {
* protected Component component;
* public Decorator(Component component) {
* this.component = component;
* }
* public void operation() {
* component.operation();
* }
* }
* class ConcreteDecorator extends Decorator {
* public ConcreteDecorator(Component component) {
* super(component);
* }
* public void operation() {
* super.operation();
* System.out.println("ConcreteDecorator");
* }
* }

 **Facade**

* **Description**: Provides a unified interface to a set of interfaces in a subsystem, making the subsystem easier to use.
* **Example**:

java

* 
* class Subsystem1 {
* public void operation1() {
* System.out.println("Subsystem1 operation1");
* }
* }
* class Subsystem2 {
* public void operation2() {
* System.out.println("Subsystem2 operation2");
* }
* }
* class Facade {
* private Subsystem1 subsystem1 = new Subsystem1();
* private Subsystem2 subsystem2 = new Subsystem2();
* public void operation() {
* subsystem1.operation1();
* subsystem2.operation2();
* }
* }

 **Flyweight**

* **Description**: Uses sharing to support large numbers of fine-grained objects efficiently.
* **Example**:

java

* 
* interface Flyweight {
* void operation(String extrinsicState);
* }
* class ConcreteFlyweight implements Flyweight {
* private String intrinsicState;
* public ConcreteFlyweight(String intrinsicState) {
* this.intrinsicState = intrinsicState;
* }
* public void operation(String extrinsicState) {
* System.out.println("Intrinsic State = " + intrinsicState + ", Extrinsic State = " + extrinsicState);
* }
* }
* class FlyweightFactory {
* private Map<String, Flyweight> flyweights = new HashMap<>();
* public Flyweight getFlyweight(String key) {
* if (!flyweights.containsKey(key)) {
* flyweights.put(key, new ConcreteFlyweight(key));
* }
* return flyweights.get(key);
* }
* }

 **Proxy**

* **Description**: Provides a surrogate or placeholder for another object to control access to it.
* **Example**:

java

* + interface Subject {
  + void request();
  + }
  + class RealSubject implements Subject {
  + public void request() {
  + System.out.println("RealSubject request");
  + }
  + }
  + class Proxy implements Subject {
  + private RealSubject realSubject;
  + public void request() {
  + if (realSubject == null) {
  + realSubject = new RealSubject();
  + }
  + realSubject.request();
  + }
  + }

**Behavioral Patterns**

1. **Chain of Responsibility**
   * **Description**: Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
   * **Example**:

java

* 
* abstract class Handler {
* protected Handler successor;
* public void setSuccessor(Handler successor) {
* this.successor = successor;
* }
* public abstract void handleRequest(int request);
* }
* class ConcreteHandler1 extends Handler {
* public void handleRequest(int request) {
* if (request < 10) {
* System.out.println("Handler1 handled request: " + request);
* } else if (successor != null) {
* successor.handleRequest(request);
* }
* }
* }
* class ConcreteHandler2 extends Handler {
* public void handleRequest(int request) {
* if (request >= 10 && request < 20) {
* System.out.println("Handler2 handled request: " + request);
* } else if (successor != null) {
* successor.handleRequest(request);
* }
* }
* }
* // Client
* class Client {
* public static void main(String[] args) {
* Handler handler1 = new ConcreteHandler1();
* Handler handler2 = new ConcreteHandler2();
* handler1.setSuccessor(handler2);
* handler1.handleRequest(5);
* handler1.handleRequest(15);
* handler1.handleRequest(25);
* }
* }

 **Command**

* **Description**: Encapsulates a request as an object, thereby letting you parameterize clients with queues, requests, and operations.
* **Example**:

java

* 
* interface Command {
* void execute();
* }
* class ConcreteCommand implements Command {
* private Receiver receiver;
* public ConcreteCommand(Receiver receiver) {
* this.receiver = receiver;
* }
* public void execute() {
* receiver.action();
* }
* }
* class Receiver {
* public void action() {
* System.out.println("Receiver action");
* }
* }
* class Invoker {
* private Command command;
* public void setCommand(Command command) {
* this.command = command;
* }
* public void executeCommand() {
* command.execute();
* }
* }
* // Client
* class Client {
* public static void main(String[] args) {
* Receiver receiver = new Receiver();
* Command command = new ConcreteCommand(receiver);
* Invoker invoker = new Invoker();
* invoker.setCommand(command);
* invoker.executeCommand();
* }
* }

 **Interpreter**

* **Description**: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
* **Example**:

java

* 
* interface Expression {
* boolean interpret(String context);
* }
* class TerminalExpression implements Expression {
* private String data;
* public TerminalExpression(String data) {
* this.data = data;
* }
* public boolean interpret(String context) {
* return context.contains(data);
* }
* }
* class OrExpression implements Expression {
* private Expression expr1;
* private Expression expr2;
* public OrExpression(Expression expr1, Expression expr2) {
* this.expr1 = expr1;
* this.expr2 = expr2;
* }
* public boolean interpret(String context) {
* return expr1.interpret(context) || expr2.interpret(context);
* }
* }
* class AndExpression implements Expression {
* private Expression expr1;
* private Expression expr2;
* public AndExpression(Expression expr1, Expression expr2) {
* this.expr1 = expr1;
* this.expr2 = expr2;
* }
* public boolean interpret(String context) {
* return expr1.interpret(context) && expr2.interpret(context);
* }
* }
* // Client
* class InterpreterPatternDemo {
* public static void main(String[] args) {
* Expression isJava = new TerminalExpression("Java");
* Expression isPython = new TerminalExpression("Python");
* Expression isProgrammingLanguage = new OrExpression(isJava, isPython);
* System.out.println(isProgrammingLanguage.interpret("Java")); // true
* System.out.println(isProgrammingLanguage.interpret("JavaScript")); // false
* }
* }

 **Iterator**

* **Description**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
* **Example**:

java

* 
* interface Iterator {
* boolean hasNext();
* Object next();
* }
* interface Container {
* Iterator getIterator();
* }
* class NameRepository implements Container {
* private String[] names = {"John", "Jane", "Jack", "Jill"};
* public Iterator getIterator() {
* return new NameIterator();
* }
* private class NameIterator implements Iterator {
* int index;
* public boolean hasNext() {
* return index < names.length;
* }
* public Object next() {
* if (this.hasNext()) {
* return names[index++];
* }
* return null;
* }
* }
* }
* // Client
* class IteratorPatternDemo {
* public static void main(String[] args) {
* NameRepository namesRepository = new NameRepository();
* for (Iterator iter = namesRepository.getIterator(); iter.hasNext(); ) {
* String name = (String) iter.next();
* System.out.println("Name : " + name);
* }
* }
* }

 **Mediator**

* **Description**: Defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
* **Example**:

java

* 
* interface Mediator {
* void sendMessage(String message, Colleague colleague);
* }
* abstract class Colleague {
* protected Mediator mediator;
* public Colleague(Mediator mediator) {
* this.mediator = mediator;
* }
* public abstract void receiveMessage(String message);
* }
* class ConcreteColleague1 extends Colleague {
* public ConcreteColleague1(Mediator mediator) {
* super(mediator);
* }
* public void receiveMessage(String message) {
* System.out.println("Colleague1 received: " + message);
* }
* }
* class ConcreteColleague2 extends Colleague {
* public ConcreteColleague2(Mediator mediator) {
* super(mediator);
* }
* public void receiveMessage(String message) {
* System.out.println("Colleague2 received: " + message);
* }
* }
* class ConcreteMediator implements Mediator {
* private ConcreteColleague1 colleague1;
* private ConcreteColleague2 colleague2;
* public void setColleague1(ConcreteColleague1 colleague1) {
* this.colleague1 = colleague1;
* }
* public void setColleague2(ConcreteColleague2 colleague2) {
* this.colleague2 = colleague2;
* }
* public void sendMessage(String message, Colleague colleague) {
* if (colleague == colleague1) {
* colleague2.receiveMessage(message);
* } else {
* colleague1.receiveMessage(message);
* }
* }
* }
* // Client
* class MediatorPatternDemo {
* public static void main(String[] args) {
* ConcreteMediator mediator = new ConcreteMediator();
* ConcreteColleague1 colleague1 = new ConcreteColleague1(mediator);
* ConcreteColleague2 colleague2 = new ConcreteColleague2(mediator);
* mediator.setColleague1(colleague1);
* mediator.setColleague2(colleague2);
* colleague1.receiveMessage("Hi from Colleague1");
* colleague2.receiveMessage("Hi from Colleague2");
* }
* }

 **Memento**

* **Description**: Captures and externalizes an object's internal state so that the object can be restored to this state later.
* **Example**:

java

* 
* class Memento {
* private String state;
* public Memento(String state) {
* this.state = state;
* }
* public String getState() {
* return state;
* }
* }
* class Originator {
* private String state;
* public void setState(String state) {
* this.state = state;
* }
* public String getState() {
* return state;
* }
* public Memento saveStateToMemento() {
* return new Memento(state);
* }
* public void getStateFromMemento(Memento memento) {
* state = memento.getState();
* }
* }
* class Caretaker {
* private List<Memento> mementoList = new ArrayList<>();
* public void add(Memento state) {
* mementoList.add(state);
* }
* public Memento get(int index) {
* return mementoList.get(index);
* }
* }
* // Client
* class MementoPatternDemo {
* public static void main(String[] args) {
* Originator originator = new Originator();
* Caretaker caretaker = new Caretaker();
* originator.setState("State #1");
* originator.setState("State #2");
* caretaker.add(originator.saveStateToMemento());
* originator.setState("State #3");
* caretaker.add(originator.saveStateToMemento());
* originator.setState("State #4");
* System.out.println("Current State: " + originator.getState());
* originator.getStateFromMemento(caretaker.get(0));
* System.out.println("First saved State: " + originator.getState());
* originator.getStateFromMemento(caretaker.get(1));
* System.out.println("Second saved State: " + originator.getState());
* }
* }

 **Observer**

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

java

* 
* import java.util.ArrayList;
* import java.util.List;
* interface Observer {
* void update();
* }
* class Subject {
* private List<Observer> observers = new ArrayList<>();
* private int state;
* public int getState() {
* return state;
* }
* public void setState(int state) {
* this.state = state;
* notifyAllObservers();
* }
* public void attach(Observer observer) {
* observers.add(observer);
* }
* private void notifyAllObservers() {
* for (Observer observer : observers) {
* observer.update();
* }
* }
* }
* class ConcreteObserver implements Observer {
* private Subject subject;
* public ConcreteObserver(Subject subject) {
* this.subject = subject;
* this.subject.attach(this);
* }
* public void update() {
* System.out.println("State changed: " + subject.getState());
* }
* }
* // Client
* class ObserverPatternDemo {
* public static void main(String[] args) {
* Subject subject = new Subject();
* new ConcreteObserver(subject);
* new ConcreteObserver(subject);
* subject.setState(1);
* subject.setState(2);
* }
* }

 **State**

* **Description**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
* **Example**:

java

* 
* interface State {
* void doAction(Context context);
* }
* class StartState implements State {
* public void doAction(Context context) {
* System.out.println("Player is in start state");
* context.setState(this);
* }
* public String toString() {
* return "Start State";
* }
* }
* class StopState implements State {
* public void doAction(Context context) {
* System.out.println("Player is in stop state");
* context.setState(this);
* }
* public String toString() {
* return "Stop State";
* }
* }
* class Context {
* private State state;
* public void setState(State state) {
* this.state = state;
* }
* public State getState() {
* return state;
* }
* }
* // Client
* class StatePatternDemo {
* public static void main(String[] args) {
* Context context = new Context();
* StartState startState = new StartState();
* startState.doAction(context);
* System.out.println(context.getState().toString());
* StopState stopState = new StopState();
* stopState.doAction(context);
* System.out.println(context.getState().toString());
* }
* }

 **Strategy**

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

java

* 
* interface Strategy {
* int doOperation(int num1, int num2);
* }
* class OperationAdd implements Strategy {
* public int doOperation(int num1, int num2) {
* return num1 + num2;
* }
* }
* class OperationSubtract implements Strategy {
* public int doOperation(int num1, int num2) {
* return num1 - num2;
* }
* }
* class OperationMultiply implements Strategy {
* public int doOperation(int num1, int num2) {
* return num1 \* num2;
* }
* }
* class Context {
* private Strategy strategy;
* public Context(Strategy strategy) {
* this.strategy = strategy;
* }
* public int executeStrategy(int num1, int num2) {
* return strategy.doOperation(num1, num2);
* }
* }
* // Client
* class StrategyPatternDemo {
* public static void main(String[] args) {
* Context context = new Context(new OperationAdd());
* System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
* context = new Context(new OperationSubtract());
* System.out.println("10 - 5 = " + context.executeStrategy(10, 5));
* context = new Context(new OperationMultiply());
* System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));
* }
* }

 **Template Method**

* **Description**: Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
* **Example**:

java

* 
* abstract class Game {
* abstract void initialize();
* abstract void startPlay();
* abstract void endPlay();
* // template method
* public final void play() {
* initialize();
* startPlay();
* endPlay();
* }
* }
* class Cricket extends Game {
* void initialize() {
* System.out.println("Cricket Game Initialized! Start playing.");
* }
* void startPlay() {
* System.out.println("Cricket Game Started. Enjoy the game!");
* }
* void endPlay() {
* System.out.println("Cricket Game Finished!");
* }
* }
* class Football extends Game {
* void initialize() {
* System.out.println("Football Game Initialized! Start playing.");
* }
* void startPlay() {
* System.out.println("Football Game Started. Enjoy the game!");
* }
* void endPlay() {
* System.out.println("Football Game Finished!");
* }
* }
* // Client
* class TemplatePatternDemo {
* public static void main(String[] args) {
* Game game = new Cricket();
* game.play();
* game = new Football();
* game.play();
* }
* }

 **Visitor**

* **Description**: Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
* **Example**:

java

* + interface ComputerPart {
  + void accept(ComputerPartVisitor computerPartVisitor);
  + }
  + class Keyboard implements ComputerPart {
  + public void accept(ComputerPartVisitor computerPartVisitor) {
  + computerPartVisitor.visit(this);
  + }
  + }
  + class Monitor implements ComputerPart {
  + public void accept(ComputerPartVisitor computerPartVisitor) {
  + computerPartVisitor.visit(this);
  + }
  + }
  + class Computer implements ComputerPart {
  + ComputerPart[] parts;
  + public Computer() {
  + parts = new ComputerPart[] { new Keyboard(), new Monitor() };
  + }
  + public void accept(ComputerPartVisitor computerPartVisitor) {
  + for (ComputerPart part : parts) {
  + part.accept(computerPartVisitor);
  + }
  + computerPartVisitor.visit(this);
  + }
  + }
  + interface ComputerPartVisitor {
  + void visit(Keyboard keyboard);
  + void visit(Monitor monitor);
  + void visit(Computer computer);
  + }
  + class ComputerPartDisplayVisitor implements ComputerPartVisitor {
  + public void visit(Keyboard keyboard) {
  + System.out.println("Displaying Keyboard.");
  + }
  + public void visit(Monitor monitor) {
  + System.out.println("Displaying Monitor.");
  + }
  + public void visit(Computer computer) {
  + System.out.println("Displaying Computer.");
  + }
  + }
  + // Client
  + class VisitorPatternDemo {
  + public static void main(String[] args) {
  + ComputerPart computer = new Computer();
  + computer.accept(new ComputerPartDisplayVisitor());
  + }
  + }

These examples provide a comprehensive overview of various design patterns and their implementations in Java. They serve as a good starting point for understanding how to apply design patterns to solve common software design problems.