Design patterns

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

What are design patterns?

- Well described solutions to object oriented software development problems
  - Address recurring problems
  - A template for solving problems
- Utilize best developed practices by experience of software experts over the years
  - Most patterns have been evolved and refined over time
  - Well proven solutions
- Each design pattern describes a problem, solution, and results of the solution if it is applied
- Design patterns are suggestions/guidelines

Benefits of using design patterns

- Well specified industry standard solutions
- Code reusability, maintainability, and portability
- Code is easier to understand and is consistent across systems
  - Less training for new engineers

Major types of design patterns

- Creational design patterns
  - Provide templates for how to instantiate an object in the best way for a specific problem
- Structural design patterns
  - Provide templates for different ways to design and create a system structure
    - E.g. inheritance and composition
- Behavioral design patterns
  - Provide templates for the best ways to design and implement interactions between objects and systems
- Other/miscellaneous
  - Other templates that cannot be classified under the first three major classes

Types of creational design patterns

- Singleton pattern
- Factory pattern
- Abstract factory pattern
- Builder pattern
- Prototype pattern

Singleton pattern

- Enables an application to have one and only one instance of a class per JVM
- Restricts the instantiation of a class to a single object
- Serialization of singletons needs special consideration
  - Deserializing a singleton can create new instance of a class
  - To overcome this issue we need to provide implementation of *readResolve()* method
    - Can just return the instance of the class

Ways to implement singleton pattern

- Eager initialization
  - Instance of a class is created before its use is required
  - Drawback is that objects are created irrespective of their need, taking up resources that may not be necessary during runtime
- Lazy initialization
  - Delaying the instantiation of an object until the first time it is used
  - Drawback is that there could be issues with this method within multi-threaded programs
- Static block initialization
  - Static blocks are executed upon class initialization before a constructor is even called
  - Drawback is that not all static fields in the class maybe needed but they will all be initialized
- Bill Pugh solution
  - Bill Pugh was influential in development of Java memory model
  - Initialization on demand
  - Using inner static helper class for initialization
  - Most commonly used method
- Enum singleton
  - Java automatically ensures that enum values are initialized only once
  - Enum values are globally accessible
  - Drawback is that enums are not flexible enough as a data structure to achieve extended functionality

Examples: ways to implement singleton pattern

```
public class Test {
    private static final EagerSingleton myObject = new EagerSingleton();

    private Test(){};

    public static EagerSingleton getSingletonInstance(){
        return myObject;
    }
}
```

```
public class Test {
    private static LazySingleton myObject = null;

    private Test(){};

    public static LazySingleton getSingletonInstance(){
        if(myObject == null){
            myObject = new LazySingleton();
        }
        return myObject;
    }
}
```

```
public class Test {
    private static StaticBlockSingleton myObject;

    private Test(){};

    static{
        try{
            myObject = new StaticBlockSingleton();
        }catch(Exception e){
            throw new RuntimeException("Exception occured");
        }
    }

    public static StaticBlockSingleton getSingletonInstance(){
        return myObject;
    }
}
```

```
public class Test {
    private Test(){};

    private static class SingletonHelper{
        private static final BillPughSingleton myObject = new BillPughSingleton();
    }

    public static BillPughSingleton getSingletonInstance(){
        return SingletonHelper.myObject;
    }
}
```

```
public enum EnumSingleton {
    myObject;

    public static void myMethod(){
        ...
    }
}
```

Implementation of *readResolve()* method:

```
protected Object readResolve() {
    return getSingletonInstance();
}
```

Factory pattern

- This design pattern is used when complex object creation is needed and there is a need to centralized coordination
- Object instantiation is delegated to a “factory” class
  - Factory methods can be made static
- Separation of object creation responsibility and the “client” class/system
- One factory class returns various class/sub-class instances

Example: using factory design pattern

```java
class Animal{
    private String type;
    private Double weight;

    public Animal(String t, Double w){
        this.type = t;
        this.weight = w;
    }
}

class Dog extends Animal{
    public Dog(String t, Double w){super(t, w);}
}

class Cat extends Animal{
    public Cat(String t, Double w){super(t, w);}
}

class AnimalFactory {
    public static Animal createAnimal(String objectType, String type, Double weight){
        if("Dog".equalsIgnoreCase(objectType)) return new Dog(type, weight);
        else if("Cat".equalsIgnoreCase(objectType)) return new Cat(type, weight);

        return null;
    }
}

public class Test {
    public static void main(String args[]){
        Animal dog1 = AnimalFactory.createAnimal("Dog", "Dog, german shepherd", 50.0);
        Animal dog2 = AnimalFactory.createAnimal("Dog", "Dog, pug", 25.0);
        Animal cat1 = AnimalFactory.createAnimal("Cat", "Cat, tabby cat", 18.5);

        System.out.println(dog1);
        System.out.println(dog2);
        System.out.println(cat1);
    }
}
```

Both Dog and Cat inherit from Animal

Factory class whose only purpose is to instantiate Animal objects

Public static method that can be called without creating an instance of AnimalFactory

Abstract factory pattern

- This design pattern is used when additional abstraction level is needed in addition to the simpler factory design pattern
- Usually implemented in a way where a single factory class is responsible for instantiating/returning a single class/type

Example: using abstract factory design pattern

Interface representing abstract animal factory

Two factory classes that implement abstract factory interface, one for each type of object to create

A single animal factory class that is responsible for creating all animals using specified factory

Create animal objects by creating new instances of specialized animal factories

```
class Animal{
    private String type;
    private Double weight;

    public Animal(String t, Double w){
        this.type = t;
        this.weight = w;
    }
}

class Dog extends Animal{
    public Dog(String t, Double w){super(t, w);}
}

class Cat extends Animal{
    public Cat(String t, Double w){super(t, w);}
}

interface AnimalAbstractFactory {
    public Animal createAnimal();
}

class DogFactory implements AnimalAbstractFactory {
    private String type;
    private Double weight;

    public DogFactory(String t, Double w){
        this.type = t;
        this.weight = w;
    }

    @Override
    public Animal createAnimal() {
        return new Dog(this.type, this.weight);
    }
}

class CatFactory implements AnimalAbstractFactory {
    private String type;
    private Double weight;

    public CatFactory(String t, Double w){
        this.type = t;
        this.weight = w;
    }

    @Override
    public Animal createAnimal() {
        return new Cat(this.type, this.weight);
    }
}

class AnimalFactory {
    public static Animal createAnimal(AnimalAbstractFactory factory){
        return factory.createAnimal();
    }
}

public class Test {
    public static void main(String args[]){
        Animal dog1 = AnimalFactory.createAnimal(new DogFactory("Dog, german shepherd", 50.0));
        Animal dog2 = AnimalFactory.createAnimal(new DogFactory("Dog, pug", 25.0));
        Animal cat1 = AnimalFactory.createAnimal(new CatFactory("Cat, tabby cat", 18.5));

        System.out.println(dog1);
        System.out.println(dog2);
        System.out.println(cat1);
    }
}
```

Builder pattern

- This design process is used when we want to create many different objects by the same building process
- Builder object solves issues with factory and abstract factory patterns, especially when many optional parameters and inconsistent states are involved
- Builder class usually provides a method to return the instantiated object

Example: using builder design pattern

```
class Animal {
    private String type;
    private Double weight;
    // optional fields:
    private Boolean bushyTail;
    private String name;

    private Animal(AnimalBuilder ab){
        this.type = ab.type;
        this.weight = ab.weight;
        this.bushyTail = ab.bushyTail;
        this.name = ab.name;
    }

    public static class AnimalBuilder {
        private String type;
        private Double weight;
        // optional fields:
        private Boolean bushyTail;
        private String name;

        public AnimalBuilder(String t, Double w){
            this.type = t;
            this.weight = w;
        }

        // setters for optional fields:
        public AnimalBuilder setBushyTail(Boolean val) {
            this.bushyTail = val;
            return this;
        }

        public AnimalBuilder setName(String val) {
            this.name = val;
            return this;
        }

        public Animal createAnimal(){
            return new Animal(this);
        }
    }
}

public class Test {
    public static void main(String args[]){
        Animal dog1 = new Animal.AnimalBuilder("Dog, german shepherd", 50.0).setBushyTail(true).setName("Sally").createAnimal();
        Animal dog2 = new Animal.AnimalBuilder("Dog, pug", 25.0).setBushyTail(false).setName("Barky").createAnimal();
        Animal cat1 = new Animal.AnimalBuilder("Cat, tabby cat", 18.5).setBushyTail(true).setName("Tinker").createAnimal();

        System.out.println(dog1);
        System.out.println(dog2);
        System.out.println(cat1);
    }
}
```

Note that the constructor is private and accepts AnimalBuilder instance as input parameter

Inner static class that serves the purpose of a builder/factory

Setters for the optional fields; notice that they return self

Set the optional fields during the object initialization

Prototype pattern

- This design pattern is used when an application needs to create a large number of instances of the same class
  - In the same or similar state
- Usually used when object creation is costly in terms of resources
  - Make copies of an object at a lower cost
  - Strongly recommended to implement deep copy in the class we need many copies of

Example: using prototype pattern

```
class Animal implements Cloneable{
    private String type;
    private Double weight;

    public Animal(String t, Double w){
        this.type = t;
        this.weight = w;
    }

    public Object clone() throws CloneNotSupportedException{
        return new Animal(this.type, this.weight);
    }

    public String toString(){return this.type+", "+this.weight+" lbs";}
}

public class Test {
    public static void main(String args[]){
        Animal animal1 = new Animal("Dog, german shepherd", 50.0);

        try{
            Animal animal2 = (Animal) animal1.clone();
            System.out.println(animal1);
            System.out.println(animal2);
        }catch(CloneNotSupportedException e){
            System.out.println("This type does not support cloning");
        }
    }
}
```

Must implement Clonable interface

Must implement *clone()* method

Deep copy of an object

Output:

```
Dog, german shepherd, 50.0 lbs
Dog, german shepherd, 50.0 lbs
```

Must catch *CloneNotSupportedException* at some point

Summary of **creational design patterns**

Creational design pattern	Main takeaways
Singleton	Describes ways to force only a single instance of an object in a given JVM
Factory	Describes a way to use a factory class to create instances of objects of many related types
Abstract factory	Allows an additional level of abstraction for the factory pattern
Builder	Similar to the factory method but allows for more flexibility when many optional parameters are available during instantiating an object
Prototype	Describes a way to create many copies of an object from initially created prototype object

Types of structural design patterns

- Adapter pattern
- Composite pattern
- Proxy pattern
- Flyweight pattern
- Facade pattern
- Bridge pattern
- Decorator pattern

Adapter pattern

- Adapter design pattern converts the interface of a class into another interface to fit expectations/requirements/needs of the user class/system
- Two ways to achieve adapter pattern
  - Class adapter – achieved through inheritance
  - Object adapter – achieved through composition

Example: using adapter pattern to enhance functionality

Using class adapter:

```
class Animal{
    private String type;
    private Double weight;

    public Animal(String t, Double w){
        this.type = t;
        this.weight = w;
    }

    // getters:
    public String getType(){return this.type;}
    public Double getWeight(){return this.weight;}
}

class Dog extends Animal {
    public Dog(String t, Double w){
        super(t, w);
    }

    public void bark(){System.out.println("Bark bark");}
}

public class Test {
    public static void main(String args[]){
        Dog animal1 = new Dog("Dog, german shepherd", 50.0);
        animal1.bark();

        Animal animal2 = new Dog("Dog, pug", 25.0);
        ((Dog)animal2).bark();

        Animal animal3 = new Animal("Dog, pug", 27.0);
        //animal3.bark();
    }
}
```

We use inheritance

Dog class is an adapter in both cases

Dog is an Animal so we can do this casting;
note that we cannot do this object adapter on the right

Using object adapter:

```
class Animal{
    private String type;
    private Double weight;

    public Animal(String t, Double w){
        this.type = t;
        this.weight = w;
    }

    // getters:
    public String getType(){return this.type;}
    public Double getWeight(){return this.weight;}
}

class Dog {
    private Animal dog;

    public Dog(String t, Double w){
        this.dog = new Animal(t, w);
    }

    public void bark(){System.out.println("Bark bark");}
}

public class Test {
    public static void main(String args[]){
        Dog animal1 = new Dog("Dog, german shepherd", 50.0);
        animal1.bark();

        Animal animal3 = new Animal("Dog, pug", 27.0);
        //animal3.bark();
    }
}
```

We use composition

Example: using adapter pattern to simplify API

```
interface Animal {
    public void walk();
    public void run();
    public void makeSound();
    public void sleep();
    public void introduce();
}

class BasicAnimal implements Animal{
    private String sound;

    public BasicAnimal(String s){this.sound = s;}
    public void makeSound(){System.out.println(this.sound);}

    // empty implementations of methods we do not need:
    public void walk(){};
    public void run(){};
    public void sleep(){};
    public void introduce(){};
}

class FairyTaleAnimal extends BasicAnimal {
    public FairyTaleAnimal(String s){super(s);}
}

public class Test {
    public static void main(String args[]){
        Animal animal1 = new FairyTaleAnimal("Meow");
        animal1.makeSound();
    }
}
```

BasicAnimal class is an adapter class for Animal interface, allowing any of its children to not have to implement many methods specified by the Animal API

Empty implementations for methods specified by Animal interface

FairyTaleAnimal is an Animal but does not have to implement methods specified by Animal interface by using an adapter class

Composite pattern

- This design pattern helps combining objects into hierarchical structures
  - Part-whole relationships
- Allows treatment of individual and composite objects in a uniform manner
- E.g. each individual shape has a method *draw()* but a canvas with multiple shape also has a method *draw()*

Example: using composite pattern

Each instance of Animal has method makeSound()

Children of Animal have makeSound() functionality

Herd is an Animal composite class

Herd now also has makeSound() functionality

To the user, both individual objects as well as the composite object has the same makeSound() functionality

```
import java.util.*;

class Animal {
    private String type;
    private String sound;

    public Animal(String t, String s){
        this.type = t;
        this.sound = s;
    }

    public void makeSound(){System.out.println(this.sound);}
}

class Horse extends Animal {
    public Horse(String t){
        super(t, "Neigh");
    }
}

class WildHog extends Animal {
    public WildHog(String t){
        super(t, "Oink");
    }
}

class Herd {
    private List<Animal> animals = new ArrayList<Animal>();

    public void add(Animal a){
        this.animals.add(a);
    }

    public void makeSound() {
        for(Animal a : animals){
            a.makeSound();
        }
    }
}

public class Test {
    public static void main(String args[]){
        Animal animal1 = new Horse("Mustang");
        animal1.makeSound();

        Animal animal2 = new WildHog("Pig");
        animal2.makeSound();

        Herd herd = new Herd();
        herd.add(new Horse("Mustang"));
        herd.add(new Horse("Morab"));
        herd.add(new Horse("Lokai"));
        herd.add(new Horse("Mustang"));
        herd.makeSound();
    }
}
```

Proxy pattern

- Using this design pattern we provide a proxy object to deal with the object of interest without providing direct access to this object
  - Surrogate object
  - E.g. lazy loading
- We use this pattern when we want to provide controlled access and/or functionality

Example: using proxy pattern

```java
class ClassWithFunctionality {
    public void importantFunction(){
        System.out.println("This is a call to the method that does the work");
    }
}

class ProxyClass extends ClassWithFunctionality {
    public void importantFunction(){
        System.out.println("This call is to proxy class, delegating to real implementation now");
        super.importantFunction();
    }
}

public class Test {
    public static void main(String args[]){
        ClassWithFunctionality proxyObject = new ProxyClass();
        proxyObject.importantFunction();
    }
}
```

The class that has the implementation of interest

Class that is a proxy class, inherits from the class above and delegates the work to the actual implementation

We make an instance of a proxy class and make the method call

Output:

```
This call is to proxy class, delegating to real implementation now
This is a call to the method that does the work
```

Flyweight pattern

- We use this pattern when we need to provide object use in multiple contexts
  - Flyweight objects provide independent use in different contexts
- Utilizes object sharing
- Usually used when we need to conserve resources
  - Can reduced load on memory
  - Useful in mobile and embedded systems

Example: using flyweight pattern

Generic pen

Separate thick and thin pen classes

```java
import java.util.HashMap;

enum BrushSize {
    THIN, THICK
}

interface Pen {
    public void setColor(String color);
    public void draw(String content);
}

class ThickPen implements Pen {
    final BrushSize brushSize = BrushSize.THICK;
    private String color = null;

    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public void draw(String content) {
        System.out.println("Drawing THICK content in color : " + color);
    }
}

class ThinPen implements Pen {
    final BrushSize brushSize = BrushSize.THIN;
    private String color = null;

    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public void draw(String content) {
        System.out.println("Drawing THIN content in color : " + color);
    }
}
```

Here draw() is customized (overriden) based on the pen type

Pen factory class

```java
class PenFactory {
    private static final HashMap<String, Pen> pensMap = new HashMap<>();

    public static Pen getThickPen(String color) {
        String key = color + "-THICK";

        Pen pen = pensMap.get(key);

        if(pen != null) {
            return pen;
        } else {
            pen = new ThickPen();
            pen.setColor(color);
            pensMap.put(key, pen);
        }

        return pen;
    }

    public static Pen getThinPen(String color) {
        String key = color + "-THIN";

        Pen pen = pensMap.get(key);

        if(pen != null) {
            return pen;
        } else {
            pen = new ThinPen();
            pen.setColor(color);
            pensMap.put(key, pen);
        }

        return pen;
    }
}

public class Test {
    public static void main(String args[]){
        Pen yellowThinPen = PenFactory.getThickPen("YELLOW");
        yellowThinPen.draw("Hello World !!");

        Pen blueThinPen = PenFactory.getThinPen("BLUE");
        blueThinPen.draw("Hello World !!");
    }
}
```

This example is adopted from https://howtodoinjava.com/

Facade pattern

- This design pattern provides a unified interface for a number of classes/interfaces/subsystems for easier use
- Facade is a higher level interface
  - Analogous to giving the same front/face/facade to multiple buildings
- E.g. we can provide a facade class/interface to access a number of different databases/sources
  - To the user it appears that we access all databases in the same way

Example: using facade pattern

Class that deals with MySQL database

```
import java.sql.Connection;

class MySqlHelper {
    public static Connection getMySqlDBConnection(){
        //get MySql DB connection using connection parameters
        return null;
    }

    public void generateMySqlPDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

    public void generateMySqlHTMLReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }
}

class OracleHelper {
    public static Connection getOracleDBConnection(){
        //get Oracle DB connection using connection parameters
        return null;
    }

    public void generateOraclePDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

    public void generateOracleHTMLReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

}
```

Class that deals with Oracle database

Facade class that helps generate report from various sources and in various formats

```
class HelperFacade {
    public static enum DBTypes{
        MYSQL,ORACLE;
    }

    public static enum ReportTypes{
        HTML,PDF;
    }

    public static void generateReport(DBTypes dbType, ReportTypes reportType, String tableName){
        Connection con = null;
        switch (dbType){
        case MYSQL:
            con = MySqlHelper.getMySqlDBConnection();
            MySqlHelper mySqlHelper = new MySqlHelper();
            switch(reportType){
            case HTML:
                mySqlHelper.generateMySqlHTMLReport(tableName, con);
                break;
            case PDF:
                mySqlHelper.generateMySqlPDFReport(tableName, con);
                break;
            }
            break;
        case ORACLE:
            con = OracleHelper.getOracleDBConnection();
            OracleHelper oracleHelper = new OracleHelper();
            switch(reportType){
            case HTML:
                oracleHelper.generateOracleHTMLReport(tableName, con);
                break;
            case PDF:
                oracleHelper.generateOraclePDFReport(tableName, con);
                break;
            }
            break;
        }

    }
}
```

A look at how we make it work without a facade helper class

```
public class Test {
    public static void main(String args[]){
        String tableName="Employee";

        //generating MySql HTML report and Oracle PDF report without using Facade
        Connection con = MySqlHelper.getMySqlDBConnection();
        MySqlHelper mySqlHelper = new MySqlHelper();
        mySqlHelper.generateMySqlHTMLReport(tableName, con);

        Connection con1 = OracleHelper.getOracleDBConnection();
        OracleHelper oracleHelper = new OracleHelper();
        oracleHelper.generateOraclePDFReport(tableName, con1);
```

Much easier with a facade helper class

```
        //generating MySql HTML report and Oracle PDF report using Facade
        HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL, HelperFacade.ReportTypes.HTML, tableName);
        HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE, HelperFacade.ReportTypes.PDF, tableName);
    }
}
```

This example is borrowed from www.journaldev.com

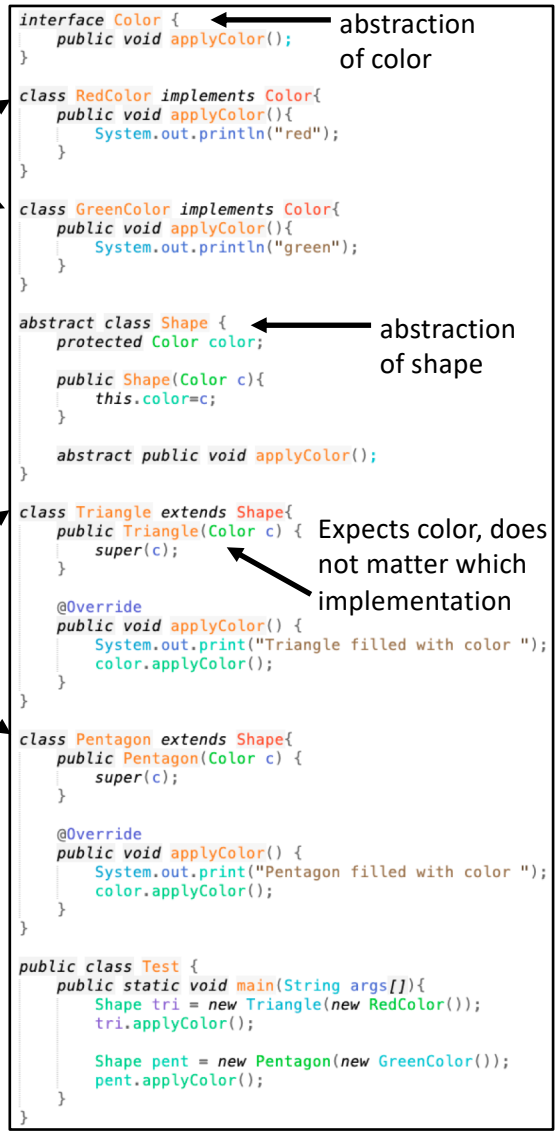
Bridge pattern

- This design pattern allows decoupling class abstraction and class implementation
  - Easier to maintain and provides flexibility for each to change at different times
  - Helps promote “loose coupling”
- Allows to hide implementation details from the user class/system

Example: using bridge pattern

Concreate implementations of colors

Concreate implementations of shapes



This example is borrowed from www.journaldev.com

Decorator pattern

- This design pattern allows adding functionality to an object on top of the interface of its class
  - Extends the behavior of an object
  - Decorator class is the one that implements these additions
  - Lets us use the class with new functionality without changing the interface of the base class

Example: using decorator pattern

Abstract car

Basic car, with no additional functionality

Decorator class that is a car

”Decorated” car classes, inherit from the decorator class

Note the nested instantiation of sports and sports luxury car objects

```
interface Car {
    public void assemble();
}

class BasicCar implements Car {
    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }
}

class CarDecorator implements Car {
    protected Car car;

    public CarDecorator(Car c){
        this.car=c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }
}

class SportsCar extends CarDecorator {
    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}

class LuxuryCar extends CarDecorator {
    public LuxuryCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}

public class Test {
    public static void main(String args[]){
        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println();

        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
        sportsLuxuryCar.assemble();
        System.out.println();
    }
}
```

Output:

```
Basic Car. Adding features of Sports Car.
Basic Car. Adding features of Luxury Car. Adding features of Sports Car.
```

This example is borrowed from www.journaldev.com

Summary of **structural design patterns**

Design pattern	Main takeaways
Adapter	Describes a way to convert interface of a class/interface into a different interface
Composite	Describes a pattern of providing the same functionality for individual objects as well as composite objects
Proxy	Describes a way to restrict access to a class functionality by employing a proxy object
Flyweight	Describes how to design classes so objects can be used in different contexts
Facade	Describes how to provide uniform interface for multiple objects
Bridge	Describes how to decouple abstraction and implementation
Decorator	Describes how to “decorate” an object with additional functionalities and responsibilities without needing to expand the original class interface

Types of behavioral design patterns

- Template method pattern
- Mediator pattern
- Chain of responsibility pattern
- Observer pattern
- Strategy pattern
- Command pattern
- State pattern
- Visitor pattern
- Iterator pattern
- Memento pattern
- Interpreter pattern

Template method pattern

- This design pattern defines steps for implementing multi-step algorithms
  - Sequential steps
  - Method stub
  - Optionally provides default implementation
    - Usually common for all or most sub-classes
  - Optionally defers implementation to sub-classes

Example: using template method pattern

```java
abstract class House {
    public final void buildhouse() {
        constructBase();
        constructRoof();
        constructWalls();
        constructWindows();
        constructDoors();
        paintHouse();
        decorateHouse();
    }

    public abstract void decorateHouse();
    public abstract void paintHouse();
    public abstract void constructDoors();
    public abstract void constructWindows();
    public abstract void constructWalls();

    private final void constructRoof() {
        System.out.println("Roof has been constructed.");
    }

    private final void constructBase() {
        System.out.println("Base has been constructed.");
    }
}
```

instructions for how to build an abstract house

Every house has the same roof and base

Concrete implementation of a house

```java
class ConcreteWallHouse extends House {
    @Override
    public void decorateHouse() {
        System.out.println("Decorating Concrete Wall House");
    }
    @Override
    public void paintHouse() {
        System.out.println("Painting Concrete Wall House");
    }
    @Override
    public void constructDoors() {
        System.out.println("Constructing Doors for Concrete Wall House");
    }
    @Override
    public void constructWindows() {
        System.out.println("Constructing Windows for Concrete Wall House");
    }
    @Override
    public void constructWalls() {
        System.out.println("Constructing Concrete Wall for my House");
    }
}

class GlassWallHouse extends House {
    @Override
    public void decorateHouse() {
        System.out.println("Decorating Glass Wall House");
    }

    @Override
    public void paintHouse() {
        System.out.println("Painting Glass Wall House");
    }

    @Override
    public void constructDoors() {
        System.out.println("Constructing Doors for Glass Wall House");
    }

    @Override
    public void constructWindows() {
        System.out.println("Constructing Windows for Glass Wall House");
    }

    @Override
    public void constructWalls() {
        System.out.println("Constructing Glass Wall for my House");
    }
}
```

Concrete implementation of a house

```java
public class Test {
    public static void main(String args[]){
        House house = new ConcreteWallHouse();
        house.buildhouse();

        System.out.println("Concrete Wall House constructed successfully");

        System.out.println();

        house = new GlassWallHouse();
        house.buildhouse();

        System.out.println("Glass Wall House constructed successfully");
    }
}
```

inherited from abstract house

Abstract house only specifies instructions for how to build the house; individual types of houses choose the implementation that is appropriate for them

Output:

```
Base has been constructed.
Roof has been constructed.
Constructing Concrete Wall for my House
Constructing Windows for Concrete Wall House
Constructing Doors for Concrete Wall House
Painting Concrete Wall House
Decorating Concrete Wall House
Concrete Wall House constructed successfully

Base has been constructed.
Roof has been constructed.
Constructing Glass Wall for my House
Constructing Windows for Glass Wall House
Constructing Doors for Glass Wall House
Painting Glass Wall House
Decorating Glass Wall House
Glass Wall House constructed successfully
```

This example is borrowed from www.howtodoinjava.com

Mediator pattern

- This design pattern provides a way to encapsulate interactions between two objects
  - Usually provides a third object that hides implementation of such interaction
  - Helps with “loose coupling”
    - Objects do not need to know how to interact with each other
- Mediator object acts as an intermediary between two objects
- E.g. a chat room might have interactions between a number of participants

Example: using mediator pattern

Mediator class is responsible for all communications between two objects

```java
interface AbstractMediator {
    public void registerRunway(Runway runway);
    public void registerFlight(Flight flight);
    public boolean isLandingOk();
    public void setLandingStatus(boolean status);
}

class Mediator implements AbstractMediator {
    private Flight flight;
    private Runway runway;
    public boolean land;

    public void registerRunway(Runway runway) {
        this.runway = runway;
    }

    public void registerFlight(Flight flight) {
        this.flight = flight;
    }

    public boolean isLandingOk() {
        return land;
    }

    @Override
    public void setLandingStatus(boolean status) {
        land = status;
    }
}

interface Command {
    void land();
}
```

Output:

```
Ready for landing.
Landing permission granted.
Successfully Landed.
```

```java
interface Command {
    void land();
}

class Flight implements Command {
    private Mediator mediator;

    public Flight(Mediator mediator) {
        this.mediator = mediator;
    }

    public void land() {
        if (mediator.isLandingOk()){
            System.out.println("Successfully Landed.");
            mediator.setLandingStatus(true);
        }else {
            System.out.println("Waiting for landing.");
        }
    }

    public void getReady() {
        System.out.println("Ready for landing.");
    }
}

class Runway implements Command {
    private Mediator mediator;

    public Runway(Mediator mediator) {
        this.mediator = mediator;
        mediator.setLandingStatus(true);
    }

    @Override
    public void land() {
        System.out.println("Landing permission granted.");
        mediator.setLandingStatus(true);
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        Mediator mediator = new Mediator();
        Flight sparrow101 = new Flight(mediator);
        Runway mainRunway = new Runway(mediator);

        mediator.registerFlight(sparrow101);
        mediator.registerRunway(mainRunway);
        sparrow101.getReady();
        mainRunway.land();
        sparrow101.land();
    }
}
```

Note there is no direct communication between the flight and the runway; all communication is conducted through the mediator object

This example is borrowed from www.geeksforgeeks.org

Chain of responsibility pattern

- In this design pattern we can link objects and their responsibilities in a linked behavior structure
  - Similar to a chain
- E.g. request from a client object is passed on to a chain of objects to be handled
  - Responsibility for deciding whether to forward on the request and which objects to forward the request to lies with the current object in the chain
- Helps with “loose coupling”

Example: using chain of responsibility pattern

```java
class Currency {
    private int amount;

    public Currency(int amt){this.amount=amt;}
    public int getAmount(){return this.amount;}
}

interface DispenseChain {
    void setNextChain(DispenseChain nextChain);
    void dispense(Currency cur);
}
```

Abstract currency dispenser

For each type of currency we have a different dispenser type

```java
class Dollar50Dispenser implements DispenseChain {
    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 50){
            int num = cur.getAmount()/50;
            int remainder = cur.getAmount() % 50;
            System.out.println("Dispensing "+num+" $50 note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}

class Dollar20Dispenser implements DispenseChain{
    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 20){
            int num = cur.getAmount()/20;
            int remainder = cur.getAmount() % 20;
            System.out.println("Dispensing "+num+" $20 note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}

class Dollar10Dispenser implements DispenseChain {
    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 10){
            int num = cur.getAmount()/10;
            int remainder = cur.getAmount() % 10;
            System.out.println("Dispensing "+num+" $10 note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        int amount = 130;

        DispenseChain c1 = new Dollar50Dispenser();
        DispenseChain c2 = new Dollar20Dispenser();
        DispenseChain c3 = new Dollar10Dispenser();

        c1.setNextChain(c2);
        c2.setNextChain(c3);

        c1.dispense(new Currency(amount));
    }
}
```

We can link currency dispensers, from largest to smallest bills, handing over dispensing responsibility to the next currency dispenser once we are done dispensing current type of currency

Output:

```
Dispensing 2 $50 note
Dispensing 1 $20 note
Dispensing 1 $10 note
```

This example is adopted from www.journaldev.com

Observer pattern

- Also called “publish-subscribe pattern”
- In this design pattern we are interested in observing the state changes of an object and notifying all its dependents about this change
  - Observer object is the object that is watching for a change
  - Subject object is the object being watched

Example: using observer pattern

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

interface Subject {
    //methods to register and unregister observers
    public void register(Observer obj);
    public void unregister(Observer obj);

    //method to notify observers of change
    public void notifyObservers();

    //method to get updates from subject
    public Object getUpdate(Observer obj);
}

interface Observer {
    //method to update the observer, used by subject
    public void update();

    //attach with subject to observe
    public void setSubject(Subject sub);
}
```

Output:

```
Obj1: No new message
Message Posted to Topic:New Message
Obj1: Consuming message:New Message
Obj2: Consuming message:New Message
Obj3: Consuming message:New Message
```

```java
class MyTopic implements Subject {
    private List<Observer> observers;
    private String message;
    private boolean changed;
    private final Object MUTEX= new Object();

    public MyTopic(){
        this.observers=new ArrayList<>();
    }

    @Override
    public void register(Observer obj) {
        if(obj == null) throw new NullPointerException("Null Observer");
        synchronized (MUTEX) {
        if(!observers.contains(obj)) observers.add(obj);
        }
    }

    @Override
    public void unregister(Observer obj) {
        synchronized (MUTEX) {
        observers.remove(obj);
        }
    }

    @Override
    public void notifyObservers() {
        List<Observer> observersLocal = null;
        //synchronization is used to make sure any observer
        //registered after message is received is not notified
        synchronized (MUTEX) {
            if (!changed)
                return;
            observersLocal = new ArrayList<>(this.observers);
            this.changed=false;
        }

        for (Observer obj : observersLocal) {
            obj.update();
        }
    }

    @Override
    public Object getUpdate(Observer obj) {
        return this.message;
    }

    //method to post message to the topic
    public void postMessage(String msg){
        System.out.println("Message Posted to Topic:"+msg);
        this.message=msg;
        this.changed=true;
        notifyObservers();
    }
}
```

```java
class MyTopicSubscriber implements Observer {
    private String name;
    private Subject topic;

    public MyTopicSubscriber(String nm){
        this.name=nm;
    }

    @Override
    public void update() {
        String msg = (String) topic.getUpdate(this);
        if(msg == null){
            System.out.println(name+": No new message");
        }else
        System.out.println(name+": Consuming message:"+msg);
    }

    @Override
    public void setSubject(Subject sub) {
        this.topic=sub;
    }
}

public class Test {
    public static void main(String args[]){
        //create subject
        MyTopic topic = new MyTopic();

        //create observers
        Observer obj1 = new MyTopicSubscriber("Obj1");
        Observer obj2 = new MyTopicSubscriber("Obj2");
        Observer obj3 = new MyTopicSubscriber("Obj3");

        //register observers to the subject
        topic.register(obj1);
        topic.register(obj2);
        topic.register(obj3);

        //attach observer to subject
        obj1.setSubject(topic);
        obj2.setSubject(topic);
        obj3.setSubject(topic);

        //check if any update is available
        obj1.update();

        //now send message to subject
        topic.postMessage("New Message");
    }
}
```

observer class

observer objects

This example is borrowed from www.journaldev.com

Strategy pattern

- Also called “policy pattern”
- In this design pattern we allow the user object to choose at runtime which implementation of a given functionality/algorithm to use
  - Our object provides multiple implementations of an algorithm for a specific task
  - The client object can choose between these implementations

Example: using strategy pattern

```java
import java.text.DecimalFormat;
import java.util.*;

interface PaymentStrategy {
    public void pay(int amount);
}

class CreditCardStrategy implements PaymentStrategy {
    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;

    public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){
        this.name=nm;
        this.cardNumber=ccNum;
        this.cvv=cvv;
        this.dateOfExpiry=expiryDate;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount +" paid with credit/debit card");
    }
}

class PaypalStrategy implements PaymentStrategy {
    private String emailId;
    private String password;

    public PaypalStrategy(String email, String pwd){
        this.emailId=email;
        this.password=pwd;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using Paypal.");
    }
}
```

Abstract payment strategy

Specific payment strategy with credit card

pay() method implementation is specific to this strategy

Specific payment strategy with paypal

```java
class Item {
    private String upcCode;
    private int price;

    public Item(String upc, int cost){
        this.upcCode=upc;
        this.price=cost;
    }

    public String getUpcCode() {
        return upcCode;
    }

    public int getPrice() {
        return price;
    }
}

class ShoppingCart {
    //List of items
    List<Item> items;

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

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

    public void removeItem(Item item){
        this.items.remove(item);
    }

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

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

Shopping cart item

Shopping cart

```java
public class Test {
    public static void main(String args[]){
        ShoppingCart cart = new ShoppingCart();

        Item item1 = new Item("1234",10);
        Item item2 = new Item("5678",40);

        cart.addItem(item1);
        cart.addItem(item2);

        //pay by paypal
        cart.pay(new PaypalStrategy("exampleemail@example.com", "pwd"));

        //pay by credit card
        cart.pay(new CreditCardStrategy("Will Smith",
            "1234567890123456", "786", "12/15"));
    }
}
```

Paying can accept any payment strategy

Output:

```
50 paid using Paypal.
50 paid with credit/debit card
```

This example is borrowed from www.journaldev.com

Command pattern

- In this design pattern we break up program logic into logical units we call commands
  - The “invoker” object calls a method on a “receiver” object to perform a given command
    - Often we use a command object, which is called on by invoker and which makes the call to the receiver
- Request-response model
- Helps with “loose coupling”

Example: using command pattern

```
interface FileSystemReceiver {
    void openFile();
    void writeFile();
    void closeFile();
}

class UnixFileSystemReceiver implements FileSystemReceiver {
    @Override
    public void openFile() {
        System.out.println("Opening file in unix OS");
    }

    @Override
    public void writeFile() {
        System.out.println("Writing file in unix OS");
    }

    @Override
    public void closeFile() {
        System.out.println("Closing file in unix OS");
    }
}

class WindowsFileSystemReceiver implements FileSystemReceiver {
    @Override
    public void openFile() {
        System.out.println("Opening file in Windows OS");
    }

    @Override
    public void writeFile() {
        System.out.println("Writing file in Windows OS");
    }

    @Override
    public void closeFile() {
        System.out.println("Closing file in Windows OS");
    }
}
```

Generic OS and file operations

Different implementations for each OS type

File operations are implemented differently for different OS types

```
interface Command {
    void execute();
}

class OpenFileCommand implements Command {
    private FileSystemReceiver fileSystem;

    public OpenFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }

    @Override
    public void execute() {
        this.fileSystem.openFile();
    }
}

class CloseFileCommand implements Command {
    private FileSystemReceiver fileSystem;

    public CloseFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }

    @Override
    public void execute() {
        this.fileSystem.closeFile();
    }
}

class WriteFileCommand implements Command {
    private FileSystemReceiver fileSystem;

    public WriteFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }

    @Override
    public void execute() {
        this.fileSystem.writeFile();
    }
}
```

Different file operations represent commands, regardless of the OS type

Generic command invoker

```
class FileInvoker {
    public Command command;

    public FileInvoker(Command c){
        this.command=c;
    }

    public void execute(){
        this.command.execute();
    }
}

class FileSystemReceiverUtil {
    public static FileSystemReceiver getUnderlyingFileSystem(){
        String osName = System.getProperty("os.name");
        System.out.println("Underlying OS is: "+osName);
        if(osName.contains("Windows")){
            return new WindowsFileSystemReceiver();
        }else{
            return new UnixFileSystemReceiver();
        }
    }
}

public class Test {
    public static void main(String args[]){
        //Creating the receiver object
        FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();

        //creating command and associating with receiver
        OpenFileCommand openFileCommand = new OpenFileCommand(fs);

        //Creating invoker and associating with Command
        FileInvoker file = new FileInvoker(openFileCommand);

        //perform action on invoker object
        file.execute();

        WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
        file = new FileInvoker(writeFileCommand);
        file.execute();

        CloseFileCommand closeFileCommand = new CloseFileCommand(fs);
        file = new FileInvoker(closeFileCommand);
        file.execute();
    }
}
```

Utility class responsible to recognizing the OS type

Automatically detect the OS type

Output:

```
Underlying OS is:Mac OS X
Opening file in unix OS
Writing file in unix OS
Closing file in unix OS
```

This example is borrowed from www.journaldev.com

State pattern

- This design pattern allows changing object's behavior when its state changes
  - The object's change in behavior will often appear almost as a change in class
- Behavior is dependent on the internal state of the object

Example: using state pattern

Output:

```
vibration...
vibration...
silent...
silent...
silent...
```

Can be Vibration or Silent type

Initialize to vibration type

Calls alert() method on whichever MobileAlertState is the current state

Types of MobileAlertState

Alert behaves differently depending on whether the silent is on or not

```
interface MobileAlertState {
    public void alert(AlertStateContext ctx);
}

class AlertStateContext {
    private MobileAlertState currentState;

    public AlertStateContext() {
        currentState = new Vibration();
    }

    public void setState(MobileAlertState state) {
        currentState = state;
    }

    public void alert() {
        currentState.alert(this);
    }
}

class Vibration implements MobileAlertState {
    @Override
    public void alert(AlertStateContext ctx) {
        System.out.println("vibration...");
    }
}

class Silent implements MobileAlertState {
    @Override
    public void alert(AlertStateContext ctx) {
        System.out.println("silent...");
    }
}

public class Test {
    public static void main(String args[]){
        AlertStateContext stateContext = new AlertStateContext();
        stateContext.alert();
        stateContext.alert();
        stateContext.setState(new Silent());
        stateContext.alert();
        stateContext.alert();
        stateContext.alert();
    }
}
```

This example is borrowed from www.geeksforgeeks.org

Visitor pattern

- In this design pattern we want the ability to perform similar operations on related/similar types of objects
  - We can create a new object which implements the operation logic
- E.g. shopping cart can contain all types of objects
  - When we go to checkout, total price is calculated
  - This calculation logic can be moved into a visitor object out of each object type in the cart

Example: using visitor pattern

```
interface ItemElement {
    public int accept(ShoppingCartVisitor visitor);
}

class Book implements ItemElement {
    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn) {
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice() {
        return price;
    }

    public String getIsbnNumber() {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}

class Fruit implements ItemElement {
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm) {
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg() {
        return pricePerKg;
    }

    public int getWeight() {
        return weight;
    }

    public String getName() {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }

}
```

Book and Fruit are types of shopping cart items

Shopping cart visitor implementation

Method visit() is overloaded so we can call it with any type of item

```
interface ShoppingCartVisitor {
    int visit(Book book);
    int visit(Fruit fruit);
}

class ShoppingCartVisitorImpl implements ShoppingCartVisitor {
    @Override
    public int visit(Book book) {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50) {
            cost = book.getPrice()-5;
        } else{
            cost = book.getPrice();
        }

        System.out.println("Book ISBN: "+book.getIsbnNumber() + " cost = "+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit) {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }
}

public class Test {
    public static void main(String args[]){
        ItemElement[] items = new ItemElement[]{
                                new Book(20, "1234"),
                                new Book(100, "5678"),
                                new Fruit(10, 2, "Banana"),
                                new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items){
            sum = sum + item.accept(visitor);
        }
        return sum;
    }
}
```

Output:

```
Book ISBN: 1234 cost = 20
Book ISBN: 5678 cost = 95
Banana cost = 20
Apple cost = 25
Total Cost = 160
```

This example is borrowed from www.geeksforgeeks.org

Iterator pattern

- This design pattern allows accessing individual objects in an aggregate object sequentially without consideration for the type of object each element is
- Allows us to traverse through an aggregate object
- E.g. Java collections framework is an example of iterator design pattern

Example: using iterator pattern

A linked list of String names

An inner class that implements iterating over the list

Create list of names

Iterate over the list

```
interface Iterator {
   public boolean hasNext();
   public Object next();
}

interface Container {
   public Iterator getIterator();
}

class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {
      int index;

      @Override
      public boolean hasNext() {
         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {
         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}

public class Test {
    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);
       }
    }
}
```

Output:

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

This example is borrowed from www.tutorialspoint.com

Memento pattern

- Also called “snapshot pattern”
- Allows reversing/restoring the state of an object to a previous state
- Allows taking a snapshot of an object state for later usage
- Usually implemented with two objects
  - Originator – the object whose state needs to be saved/restored
  - Caretaker – an inner private memento class responsible for saving/restoring the state

Example: using memento pattern

```java
class Article {
    private long id;
    private String title;
    private String content;

    public Article(long id, String title) {
        super();
        this.id = id;
        this.title = title;
    }

    public ArticleMemento createMemento() {
        ArticleMemento m = new ArticleMemento(id, title, content);
        return m;
    }

    public void setContent(String c){this.content = c;}

    public void restore(ArticleMemento m) {
        this.id = m.getId();
        this.title = m.getTitle();
        this.content = m.getContent();
    }

    @Override
    public String toString() {
        return "Article [id=" + id + ", title=" + title + ", content=" + content + "]";
    }
}

final class ArticleMemento {
    private final long id;
    private final String title;
    private final String content;

    public ArticleMemento(long id, String title, String content) {
        super();
        this.id = id;
        this.title = title;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public String getContent() {
        return content;
    }
}
```

Memento class has the same fields as the original class

Setup an article

Create a snapshot

Change the contents

Restore the article

```java
public class Test {
    public static void main(String args[]){
        Article article = new Article(1, "My Article");
        article.setContent("ABC");
        System.out.println(article);

        ArticleMemento memento = article.createMemento();

        article.setContent("123");
        System.out.println(article);

        article.restore(memento);
        System.out.println(article);
    }
}
```

Output:

```
Article [id=1, title=My Article, content=ABC]
Article [id=1, title=My Article, content=123]
Article [id=1, title=My Article, content=ABC]
```

After restoring the content is the original one

This example is borrowed from www.howtodoinjava.com

Interpreter pattern

- This design pattern allows representing grammatical structures
- Allows building a language grammar from which sentences can be evaluated
  - Provides an interpreter to deal with this grammar
- E.g. Java compiler interprets java code into byte code so JVM can understand it
- E.g. Google translator interprets one language sentence into another language

Example: using interpreter pattern

Output:

```
28 in Binary= 11100
28 in Hexadecimal= 1c
```

Types of expressions

This class performs interpreting

Create two expressions

Interpret expressions

Interpretation is based on the context of expression

```
class InterpreterContext {
    public String getBinaryFormat(int i){
        return Integer.toBinaryString(i);
    }

    public String getHexadecimalFormat(int i){
        return Integer.toHexString(i);
    }
}

interface Expression {
    String interpret(InterpreterContext ic);
}

class IntToBinaryExpression implements Expression {
    private int i;

    public IntToBinaryExpression(int c){
        this.i=c;
    }

    @Override
    public String interpret(InterpreterContext ic) {
        return ic.getBinaryFormat(this.i);
    }
}

class IntToHexExpression implements Expression {
    private int i;

    public IntToHexExpression(int c){
        this.i = c;
    }

    @Override
    public String interpret(InterpreterContext ic) {
        return ic.getHexadecimalFormat(i);
    }
}

class InterpreterClient {
    public InterpreterContext ic;

    public InterpreterClient(InterpreterContext i){
        this.ic = i;
    }

    public String interpret(String str){
        Expression exp = null;

        if(str.contains("Hexadecimal")){
            exp=new IntToHexExpression(Integer.parseInt(str.substring(0,str.indexOf(" "))));
        }else if(str.contains("Binary")){
            exp=new IntToBinaryExpression(Integer.parseInt(str.substring(0,str.indexOf(" "))));
        }else return str;

        return exp.interpret(ic);
    }
}

class Test {
    public static void main(String args[]){
        String str1 = "28 in Binary";
        String str2 = "28 in Hexadecimal";

        InterpreterClient ec = new InterpreterClient(new InterpreterContext());
        System.out.println(str1+"= "+ec.interpret(str1));
        System.out.println(str2+"= "+ec.interpret(str2));
    }
}
```

This example is adopted from www.journaldev.com

Summary of **behavioral design patterns**

Design patters	Main takeaways
Template method	Describes a design pattern for providing functionality templates
Mediator	Describes a way to encapsulate interactions between objects using a mediator object
Chain of responsibility	Describes a way to chain responsibilities between objects, similar to a linked chain
Observer	Describes how to implement publish-subscribe type of applications
Strategy	Describes how to provide diverse implementations of the same functionality in order to allow the user object to use a particular implementation depending on context
Command	Describes a way to implement request-response type of application
State	Describes a way to implement different behaviors depending on the internal state of the object
Visitor	Describes a design in which the same operations can be performed on related objects
Iterator	Describes how to implement iterable data structure
Memento	Describes a way to create immutable snapshots of objects
Interpreter	Describes a way to implement translators

To recap

- We looked at three major classes of design patters
  - Creational design patters
    - Deal with how you instantiate and create objects
  - Structural design patterns
    - Deal with how you structure your application
  - Behavioral design patterns
    - Deal with how your program behaves

Concluding remarks

- Design patterns are important guidelines for how to design your application to achieve a specific objectives
- Design patterns are only recommendations/suggestions
- Design patterns have evolved over time and utilize expertise and experience of industry experts
- Some design patterns are not independent
  - Some have similar objectives and some have similar solutions but they solve different problems