**Design Patterns**

A design pattern describes a problem which is common and then describes the solution to that problem

Design patterns can be divided into 3 categories

1) Creational : Patterns deal with creation of objects

2) Behavioral: Patterns deal with how classes and objects communicates

3) Structural: Patterns deal with the composition of classes or objects

**Singleton (Creational)**

Use when only one instance of a class is required

**Why not just use static variables/methods?**

* We might not have enough information to instantiate every singleton at static initialization time(Class laod time).
* In future if you want to allow more than one instance of the Class, May be a limited number of instances, There will not be a major change in the code

Foreg. You want an object holding details of web application properties which can be used through out the application

**How to do?**

Make the constructor private so that object can’t be instantiated outside the class and keep a static field in the class which will hold the object which can be used for access outside the class, You can give access to the instance by using static method

**Usecase**

Application properties will not be changed through out the application uptime so a singleton object of application class can be used whereever required in the application

Below example keeps values at compilation time to keep it simple else values can be read from a properties file and can be populated in the fields at a later time

public class MyApplication {  
 private static MyApplication *instance*;  
 private int webserverPort =8080;  
 private String driverClass="com.mysql.cj.jdbc.Driver";  
 private String databaseName="productdb";  
 private String dataSourceUrl="jdbc:mysql://localhost:3306/productdb";  
 private RunMode runMode=RunMode.*Development*;  
  
 public int getWebserverPort() {  
 return webserverPort;  
 }  
  
 public String getDatabaseName() {  
 return databaseName;  
 }  
  
 public String getDataSourceUrl() {  
 return dataSourceUrl;  
 }  
  
 public static MyApplication getInstance(){  
 return new MyApplication();  
 }  
  
 public String getDriverClass() {  
 return driverClass;  
 }  
  
 public RunMode getRunMode() {  
 return runMode;  
 }  
  
 public static enum RunMode{  
 *Development*,  
 *Production*  
}  
}

/\*

Using the Singleton below

\*/

public class MyApplicationUsage {  
 public static void main(String args[]) {  
 MyApplication app = MyApplication.*getInstance*();  
 int port = app.getWebserverPort();  
 String driver = app.getDriverClass();  
 MyApplication.RunMode runMode = app.getRunMode();  
 System.*out*.println("port="+port+" driver="+driver+" runmode="+runMode);  
  
 }  
}

**Builder (Creational)**

**Problem**

* The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
* The construction process must allow different representations for the objects which are getting constructed.
* Many attributes that exist in class that are optional and you want to send as arguments repeatedly
* Encapsulates the way a complex object is constructed.

**Solution-How to?**

Extract the object construction code out of its own class and move it to separate objects called builders

Execute series of these steps on a builder object

public class HolidayPlanner { private String hotel; private BusType busType; private LocalTime dinnerTime; private LocalTime lunchTime; private LocalTime breakFastTime; private LocalDate startDate; private LocalDate endDate; public String getHotel() { return hotel; } public BusType getBusType() { return busType; } public LocalTime getDinnerTime() { return dinnerTime; } public LocalTime getLunchTime() { return lunchTime; } public LocalTime getBreakFastTime() { return breakFastTime; } public LocalDate getStartDate() { return startDate; } public LocalDate getEndDate() { return endDate; } public static class HolidayPlannerBuilder { private HolidayPlanner planner; public HolidayPlannerBuilder() { this.planner=new HolidayPlanner(); } public HolidayPlannerBuilder(LocalDate startDate, LocalDate endDate) { this.planner=new HolidayPlanner(); planner.startDate = startDate; planner.endDate = endDate; } public HolidayPlannerBuilder setHotel(String hotel) { planner.hotel = hotel; return this; } public HolidayPlannerBuilder setBusType(BusType busType) { planner.busType = busType; return this; } public HolidayPlannerBuilder setDinnerTime(LocalTime dinnerTime) { planner.dinnerTime = dinnerTime; return this; } public HolidayPlannerBuilder setLunchTime(LocalTime lunchTime) { planner.lunchTime = lunchTime; return this; } public HolidayPlannerBuilder setBreakFastTime(LocalTime breakFastTime) { planner.breakFastTime = breakFastTime; return this; } public HolidayPlannerBuilder setStartDate(LocalDate startDate) { planner.startDate = startDate; return this; } public HolidayPlannerBuilder setEndDate(LocalDate endDate) { planner.endDate = endDate; return this; } public HolidayPlanner build() { return planner; } } public static enum BusType { *REGULAR*, *DELUX*}}

/\*\*

Using HolidayPlanner

\*\*/

public class HolidayPlannerUsage {  
 public static void main(String[] args) {  
 LocalDate startDate=LocalDate.*of*(2023,7,8);  
 LocalDate endDate=LocalDate.*of*(2023,7,10);  
  
 HolidayPlanner planner=new HolidayPlanner.HolidayPlannerBuilder(startDate,endDate)  
 .setBusType(HolidayPlanner.BusType.*DELUX*)  
 .build();  
 System.*out*.println("start date="+planner.getStartDate()+"end date="+planner.getEndDate()+" bus type="+planner.getBusType());  
 }  
}

**Prototype(Creational)**

**Problem**

You want to hide the complexity of making new instances from the client.

Or You want to work on an object which requires details of an existing object

**Solution-How to?**

Create a clone of an existing object, In java implement Cloneable

class Product implements Cloneable {  
  
 private String name;  
 private double price;  
 private String category;  
 private String manufacturedBy;  
 public Product(String name, double price, String category, String manufacturedBy){  
 this.category=category;  
 this.name=name;  
 this.price=price;  
 this.manufacturedBy=manufacturedBy;  
 }  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public double getPrice() {  
 return price;  
 }  
 public void setPrice(double price) {  
 this.price = price;  
 }  
 public String getCategory() {  
 return category;  
 }  
  
 public void setCategory(String category) {  
 this.category = category;  
 }  
  
 public String getManufacturedBy() {  
 return manufacturedBy;  
 }  
  
 public void setManufacturedBy(String manufacturedBy) {  
 this.manufacturedBy = manufacturedBy;  
 }  
  
 @Override  
 protected Product clone() throws CloneNotSupportedException {  
 return (Product) super.clone();  
 }  
}

/\*\*

copy of a product can be created at runtime like below and changes can be made to the copy

\*\*/

class PrototypeUsage {  
 public static void main(String[] args)throws Exception {  
 Product product=new Product("galaxy",20000,"phone","samsung");  
 Product copy= product.clone();  
 System.*out*.println(copy.getName()+", "+copy.getPrice());  
 }  
}

**Factory Method(Creational)**

**Problem**

* When you have a lot of a classes of same type and you don’t know which one to use till runtime
* When you use new keyword you are creating instance of a Concrete class which means we have tied to a concrete class and tying ourself to concrete class reduces flexibility, we should be dependent on interface and not classes.

If your code is written to an interface, then it will work with any new classes implementing that interface

through polymorphism. However, when you have code that makes use of lots of concrete classes, You are looking for trouble because that code may have to be changed as new concrete classes are added.

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

public interface IComputer {  
 void boots();  
 void works();  
 void shutdown();  
}

public abstract class AbstractComputer implements IComputer{  
  
 private String os;  
 private String processor;  
  
 public AbstractComputer(String os, String processor){  
 this.os=os;  
 this.processor=processor;  
 }  
  
 public String getOs() {  
 return os;  
 }  
  
 public void setOs(String os) {  
 this.os = os;  
 }  
  
 public String getProcessor() {  
 return processor;  
 }  
  
 public void setProcessor(String processor) {  
 this.processor = processor;  
 }  
}

public class WindowComputer extends AbstractComputer{  
 public WindowComputer(String os, String processor){  
 super(os, processor);  
 }  
 @Override  
 public void boots() {  
 System.*out*.println(getOs()+"-"+getProcessor() +"Window computer boots");  
 }  
  
 @Override  
 public void works() {  
 System.*out*.println(getOs()+"-"+getProcessor() +"window computer works");  
 }  
  
 @Override  
 public void shutdown() {  
 System.*out*.println(getOs()+"-"+getProcessor()+"-"+ "window computer shutdown");  
 }  
}

public class MacComputer extends AbstractComputer{  
  
 public MacComputer(String os, String processor){  
 super(os, processor);  
 }  
  
 @Override  
 public void boots() {  
 System.*out*.println(getOs()+"-"+getProcessor()+"-" +"mac boots");  
 }  
  
 @Override  
 public void works() {  
 System.*out*.println(getOs()+"-"+getProcessor()+"-mac works");  
 }  
  
 @Override  
 public void shutdown() {  
 System.*out*.println(getOs()+"-"+getProcessor()+"-mac shutdowns");  
 }  
}

public interface IComputerFactory {  
  
 IComputer newComputer();  
}

public class MacComputerFactory implements IComputerFactory{  
 @Override  
 public IComputer newComputer() {  
 return new MacComputer("macos 11","M2");  
 }  
}

public class WindowComputerFactory implements IComputerFactory{  
 @Override  
 public IComputer newComputer() {  
 return new WindowComputer("windows 11","intel i7");  
 }  
}

public class ComputerUsage {  
 public static void main(String[] args) {  
 IComputerFactory windowsFactory=new WindowComputerFactory();  
 IComputerFactory macFactory=new MacComputerFactory();  
  
 IComputer windowsComputer=windowsFactory.newComputer();  
 IComputer macComputer=macFactory.newComputer();  
 windowsComputer.boots();  
 windowsComputer.works();  
 windowsComputer.shutdown();  
  
 macComputer.boots();  
 macComputer.works();  
 macComputer.shutdown();  
 }  
}

**AbstractFactory(Creational)**

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

**Problem**

We started with FactoryMethod Imagine but in future or we realized that we need to take care of families of dependent objects

Foreg.

Mac has its own ecosystem of devices lightning cable , mac-charger

Windows has its own ecosystem of devices like data cable, charger

**Proxy (Structural)**

Provides a surrogate/placeholder/substitute for another object in order to control and manage access

**Problem**

Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large

images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened.

**Types of Proxy**

virtual proxy: Creates expensive objects on demand. The ImageProxy described is an example of such a proxy

Protection proxy: Controls access to the original object. Protection proxies are useful when objects should have different access rights

Remote proxy: Responsible for representing the object located remotely. Talking to the real object might involve marshalling and unmarshalling of data and talking to the remote object

interface Image {  
 byte[] getData();  
}

RealImage implements Image {  
 private byte[] data;  
  
 private String fileName;  
  
 public RealImage(String fileName) {  
 this.fileName = fileName;  
 }  
  
 protected byte[] loads() {  
 System.*out*.println("image is loaded");  
 this.data = new byte[0];  
 return data;  
 }  
  
 @Override  
 public byte[] getData() {  
 return data;  
 }  
  
}  
public class Proxy implements Image{  
 private RealImage realImage;  
 private String fileName;  
 public Proxy(String fileName){  
 this.fileName=fileName;  
 }  
  
 @Override  
 public byte[] getData(){  
 if(realImage!=null){  
 return realImage.getData();  
 }  
 realImage=new RealImage(fileName);  
 return realImage.loads();  
 }  
  
}

/\*\*

Client will use Proxy instead of directly using RealImage

\*\*/

public class ProxyUse {  
 public static void main(String[] args) {  
 Proxy proxy=new Proxy("me.jpg");  
 proxy.getData();  
 proxy.getData();  
 proxy.getData();  
 }  
}

**Adapter Pattern(Structural)**

* Convert an existing interface into the one which the client expects
* Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces

**Problem:** You have an Application which is following a Contract/Interface which is used by an existing client

There is a new Client that you want to support but it wants to use different contract which your Application is not supporting as it is following old contract which is followed by Old Client

class OldClient{  
 private IExistingContract existing;  
  
 public OldClient(IExistingContract contract){  
 this.existing=contract;  
 }  
  
 public void printPerson(){  
 Person person= existing.getPerson();  
 System.*out*.println(person.getId()+","+person.getName()+person.getPhoneNumber());  
 }  
  
}

public class Application implements IExistingContract {  
 @Override  
 public Person getPerson() {  
 Person person=new Person();  
 person.setId(1);  
 person.setName("scooby dog");  
 person.setPhoneNumber("12345");  
 return person;  
 }  
}

public class NewClient {  
  
 private IExpectedContract expected;  
  
 public NewClient(IExpectedContract expected){  
 this.expected=expected;  
 }  
  
 public void printDetails(){  
 PersonDto person =expected.getPersonDetails();  
 System.*out*.println(person.getId()+","+person.getName());  
 }  
  
}

class AdapterUsage {  
 public static void main(String[] args) {  
 IExistingContract contract=new Application();  
 OldClient oldClient=new OldClient(contract);  
 oldClient.printPerson();  
 NewClient newClient=new NewClient(new NewToOldAdapter(contract));  
 newClient.printDetails();  
 }  
}

**Decorator (Structural)**

* Used to attach additional responsibilities to an object dynamically.
* Allow a user to add new functionality to an existing object without changing the original class
* Composition over an inheritance

**How to Do?**

Create an interface.

Create concrete classes implementing the same interface.

Create an abstract decorator class implementing the above same interface.

Create a concrete decorator class extending the above abstract decorator class.

Now use the concrete decorator class created above to decorate interface objects

public interface IComponent {  
 void renders();  
}

public abstract class ComponentDecorator implements IComponent {  
 private IComponent decoratedComponent;  
  
 public ComponentDecorator(IComponent component) {  
 this.decoratedComponent = component;  
 }  
  
 public IComponent getDecoratedComponent(){  
 return decoratedComponent;  
 }  
  
 @Override  
 public void renders() {  
 decoratedComponent.renders();  
 }  
}

/\*

box composing original component

\*/  
public class BoxDecorator extends ComponentDecorator{  
 public BoxDecorator(IComponent component) {  
 super(component);  
 }  
  
 @Override  
 public void renders() {  
 System.*out*.println("box boundaries starts");  
 super.renders();  
 System.*out*.println("box boundaries stops");  
 }  
}

/\*

circle composing box

\*/

public class CircleOutSideBoxDecorator extends ComponentDecorator{  
 public CircleOutSideBoxDecorator(IComponent component) {  
 super(component);  
 }  
  
 @Override  
 public void renders() {  
 System.*out*.println("circle starts");  
 super.renders();  
 System.*out*.println("circle finishes");  
  
 }  
}

**Facade (Structural)**

* Makes complex subsystems easier to use by providing a simple interface
* A facade not only simplifies an interface, it decouples a client from a subsystem of components
* Adds one more level of abstraction to hide internal complexity

**Problem**

* You want to decouple client from subsystems
* Your subsystems have complex apis exposed which is not required for the client

**Observer (Behavior)**

The Observer Pattern deﬁnes a one-to-many dependency between objects so that when one

object changes state, all of its dependents are notiﬁed and updated automatically.

**Problem**

We want to create a Weather system in a way that when new weather update is received, The clients subscribed for the Weather update are updated automatically to display new update

We want a chat system where Chat clients are updated for the new chat message received automatically **,** Chat clients do not have to keep asking whether a new chat message is available

interface IStockObserver { void update(Stock stock);}

class SmallStockDisplay implements IStockDisplay, IStockObserver { @Override public void update(Stock stock) { display(stock); } @Override public void display(Stock stock) { System.*out*.println("Stock value " + stock.getName() + "-" + stock.getValue()); }}

class BigStockDisplay implements IStockDisplay, IStockObserver{ @Override public void update(Stock stock) { display(stock); } @Override public void display(Stock stock) { System.*out*.println("Big Stock value "+stock.getName()+"-"+stock.getValue()); }}

interface IStockObservable { void add(IStockObserver observer); void remove(IStockObserver observer); void notifyObservers();}

class StockObservable implements IStockObservable{ private List<IStockObserver> observers=new ArrayList<>(); private Stock stock; @Override public void add(IStockObserver observer) { observers.add(observer); } void updateStock(Stock stock){ this.stock=stock; notifyObservers(); } @Override public void remove(IStockObserver observer) { observers.remove(observer); } @Override public void notifyObservers() { observers.stream().forEach(observer->observer.update(stock)); }}