**Abstract Factory Design Pattern – Creational**

-> Like a factory Method DP where everything is encapsulated

-> Allows us to create families of related objects without specifying their concrete type

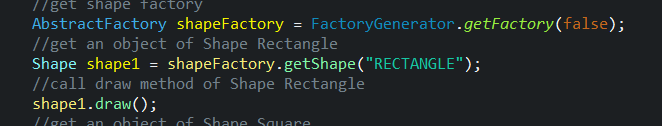
-> Use it when we want to add objects dynamically at runtime

-> We work with a factory of factories

-> An interface is responsible for creating a factory of objects without specifying their classes. Each generated factory can create the objects as it would with a Factory Method DP.

STEPS :

1. Create interface (Shape)
2. Create Concrete Classes implementing that interface(RoundedRectangle, RoundedSquare, Rectangle)
3. Create an Abstract Class to get factories for the objects(Abstract Factory)
4. Create Factory Classes extending the Abstract Factory to generate the object based on the information
5. Create a Factory Generator to generate the factory classes based on information
6. Now we use the factory generator to generator which factories we want, and we use those factories for the creation of new Objects



**Builder Design Pattern – Creational**

-> We use it to create complex objects using simple objects and using a step by step approach for each part

-> It’s useful because the creation of each part is independent of the main object. We can also have null parts

Steps:

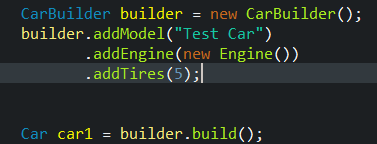
1. We create objects representing each specific part of the main object
2. We create the main object class passing as instance variables those objects created before
3. We create a Builder class for the object which will have 3 types of methods :

-> addPart(part) Methods in which the instance of the part will be added to the main object

-> buildPart(finalObject) methods in which we will have the logic on how to build the part for the finalObject

-> Build method() in which we instantiate a new main object, we call the build methods on it and return the newly built object

4) When we want to build a new object, we simply instantiate a new builder, call the addParts methods, and we simply build a new object by calling the build method. We can use the same builder for the creation of as many objects as we want.



**Factory Method Design Pattern – Creational**

-> We use it to create objects without specifying the exact class to create.

-> We use it when we want to return one of multiple possible classes that share a common superclass

-> It’s useful because we encapsulate object creation, centralize class selection code

Stepts:

1. We create an abstract class which will be the superclass of all the objects of the same category we want to create
2. We create all the classes which represent the objects by extending the abstract parent class
3. We create a factory class in which, based on certain information, the class object we want to create will be created and returned as the parent class
4. To create a new object, we instantiate an object of type parent class and we call the method from the factory for that instance, using the specific information in order to create the new subclass object.

**Prototype Design Pattern – Creational**

-> Create a new object from an existing one

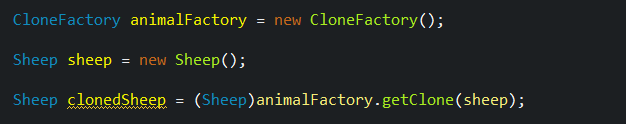
-> Allows us to add any subclass instance of a known superclass at runtime

-> It’s useful to use when there are many potential classes that we only want to use if we need to at runtime

-> It’s useful also because if the creation of an object is time or space consuming, cloning an already existing object will fasten thing dramatically

Steps :

1. Create an interface which extends cloneable so that we tell java that it’s ok to copy instances of this class, with one method called makeCopy() which returns an instance of that interface
2. We create concrete classes implementing that interface, in which we provide the implementation for makeCopy. We create a new object of that class and we call the clone method on it, after which we return it.
3. I like to create a cloneFactory with a method : getClone which has as a parameter an instance of the interface, on which we call the makeCopy method returning the clone.
4. To clone an object, we instantiate a new factory and a new object to be cloned. And create a new object of that child class by calling the method factory.getclone();



**Singleton Design Pattern – Creational**

-> This pattern involves a single class which helps to create an object while making sure that only one single object gets created.

-> This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Steps :

1. We create the singleton class and create a single instance of that object. We make the constructor private so the class cannot be instantiated. Create a method, getInstance() which returns that single instance of the object.
2. To create this class we simply say SingletonObject obj = SingletonObject.getInstance()

**Adapter Design Pattern – Structural**

-> Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.

-> The adapter class allows the use of the available interface and the target interface

-> Any class can work together as long as the adapter solves the issue that all classes must implement every method defined by the shared interface

-> The class that we already use is called the target. The class that we wish to use is called the adaptee and the class which makes the link possible is called the adapter.

Steps:

1. Create the target interface with the required abstract methods
2. Have the normal classes which implement the interface normally
3. Create the new type of class, which doesn’t implement the interface, but which we want to adapt
4. Create an adapter class which implements the target interface and which has as an instance variable an object of the class which we want to adapt. Constructor for the adapter class will also take an instance of the object we want to adapt.

Because the adapter implements the target interface, we will have to implement the abstract methods from that interface. In those implementations, we would call the appropriate methods from the instance variable(which is the object we want to adapt such that it works with out target interface)

1. To make it all work. We would create an object of type adaptee and an object of type adapter, which would be created using the adaptee in its constructor. Now we can call the methods from the target interface from the adapter, but they will behave as for an adaptee.

**Bridge Design Pattern – Structural**

-> It decouples an abstraction from its implementation so that the two can vary independently

-> What this means is that we will create a bridge interface to separate responsibilities into different abstract classes

-> It’s useful when we want to add multiple abstraction levels and each abstraction level will provide new functionalities

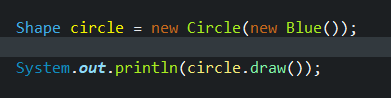
Steps :

1. Create an interface as the first abstraction(Color) and concrete classes implementing that interface(Red, Blue)
2. Create the second abstraction, an abstract class which would have a reference to the first abstraction as an instance variable. Construct the second abstraction using the first one.

Create Concrete Classes extending the second abstraction in which the methods are also linked.

public draw() : return “Draw Square” + color.drawColor().

1. To test, Initialize a new interface abstraction using the concrete class with the second abstraction.



**Composite Design Pattern – Structural**

-> It’s used when we need to treat a group of objects in a similar way to a single object

-> It’s like representing objects in a tree structure to represents parts(subtrees) or the whole hierarchy

-> Both leaves and composites respect the same interface or abstract class

-> It allowes us to represent part-whole hierarchies : componens can be divided into smaller and smaller components

Steps :

1. You create a composite type object(employee) which besides its other instance variables, also has a list of objects of its kinds(list of employees/subordinates)
2. Methods for adding, or removing employees from the list of employees/subordinates. This allows us to restructure the hierarchy.
3. Now we can create simple employees. Composite employees, which have a list of subordinates. Even higher composite employees which have a list of subordinates and some of those subordinates also have their own subordinates and so on.

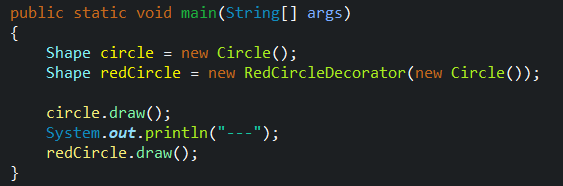
**Decorator Design Pattern – Structural**

-> Allows us to add new functionalities upon an existing object without modifying its structure

-> It acts as a wrapper to an existing class : It wraps an existing class and add functionalities

Steps:

1. We create an interface for the type of object
2. We create the type of object we want to create which implements our interface and implement its functionalities (Circle with draw method)
3. We create an AbstractDecorator which implements our interface. It will have a shape Object as instance variable which will also be initialized in its constructor. The method draw for the AbstractDecorator will call the draw method from the instance variable.
4. We create a decorator which extends the abstractDecorator and in it we override the interface methods, and add new functionalities
5. Now we can create both simple objects or decorated Objects:



**Facade Design Pattern – Structural**

-> It hides the complexities of the system

-> It provides a simplified interface (a new class) which performs many other actions(already implemented) behind the scenes

-> Useful to combine many behind the scenes operations behind one simple operation :

Big operation : Withdraw money from bank

Smaller Hidden operations : Greet the user, check if the pin is valid, check if he/she has enough money to make the withdrawal, take the money etc..

Steps :

1. We create a class which will be our facade class, which will contain as instance variables all the objects with which the hidden operations will be done
2. In the constructor of the façade object, we will instantiate all those instance variables
3. We create methods inside the façade class which will use all the hidden methods from the instance variables
4. To use it, we simply initialize a new façade object, and call its methods. We are no longer interested(or the user isn’t) with all the other logic that is going on behind the scenes, only with the façade.

**Flyweight Design Pattern – Structural**

-> Used when we want to generate a big number of very similar objects

-> In order to reduce memory usage and increase performance, objects with certain similar characteristics will share space, rather than creating new objects and allocating new space

-> Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.

Steps :

1. Create an abstract class or an interface for the object
2. Create classic object and provide its implementation
3. Create a factory for that object.

As an instance variable we will have a hashMap, using the common characteristic as the key and an object as a value.

In the method : createObject we give as a parameter that common characteristic.

We check in the hashMap if an object with that characteristic already exists. If it doesn’t we initialize a new object with that characteristic, and we put in the hashmap the object.

If it does, just return the object from the hashmap with that characteristic.

1. To create a new Object, we simply inititalize a new object using the createObject method from the factory(to which we give a new characteristic)

**Proxy Design Pattern – Structural**

-> This DP provides a class which will limit access to another class, by providing the functionality of that class

-> Useful for security reasons, or if the object is intensive to create

-> We can also use the proxy to provide access only to some part of the data from the real object, thus it’s useful when we want a simplified version of a more complex object.

Steps:

1. Create an interface with the required abstract methods
2. Create two concrete classes implementing that interface.

First one is the realObject with all its functionalities.

Second class is the proxyObject class which will have as instance variables all the instance variables of the realObject and a separate realObject.

The constructor of the proxyObject will take the same parameters as the constructor of the realObject. In the proxyObject, inherited methods from the interface, we will call the same methods from the realObject using the instance variable of the realObject.

If the realObject hasn’t been initialized yet, we initialize it here, otherwise just call the method.

1. Finally, to create a proxy object, just initialize a new proxyObject, referencing the Interface(which both the real and the proxy implement). Call the methods from the proxy

**Observer Design Pattern – Behavioral**

-> When the use has to be notified of certain changes

-> Have 2 interfaces : Subject and Observer

Subject : registerObserver, notifyObserver

Observer : update()

**Strategy Design Pattern – Behavioral**

-> It lets us define a family of algorithms and make the interchangeable

-> Thus, it’s very helpful when we need to select an algorithm, or certain behaviour at runtime

Steps:

1. Implement an interface called strategy which has a method called createStrategy(
2. Implement concrete classes(different strategies) implementing this interface in which we define the way our algorithm has to work
3. Implement a concrete class which is the context. This class will have as an instance variable an object of the strategy type, we would construct it with a strategy and it would have a method called executeStrategy which would simply call the createStrategy method of the instance variable Strategy
4. To use it, instantiate a new Context object, constructing it with the required strategy. With the constructed context, we will simply call the executeStrategy method and the chosen strategy will be run.

**State Design Pattern – Behavioral**

-> We use it when we want the behaviour of a class to change depending on its state

-> State VS Strategy :

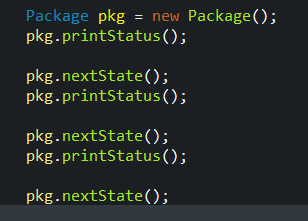
**strategy pattern defines a family of interchangeable algorithms**. Generally, they achieve the same goal, but with a different implementation, for example, sorting or rendering algorithms.

**state pattern, the behavior might change completely**, based on actual state.

-> A drawback of this pattern is the transition between the states. It has to be hardcoded.

Steps :

1. Implement a StateInterface with 3 methods : prevState, nextState and printStatus
2. Implement the object which has the possible states, not implementing the StateInterface. This object will have as an instance variable a state, getters and setters and the 3 methods from the interface in which we call the respective methods from the instance variable.
3. Implement the state. They each implement the state Interface and override the methods
4. To test it, create a new Object with states and go prev and back on the states.



**Command Design Pattern – Behavioral**

-> We want to encapsulate in an object, all the data required to perform a certain action or command

-> This data represents the method name, the object that owns that method, values for the method

-> It’s useful because it allows us to store code that will be executed at a later date

-> An object called the Invoker transfers the command to another object called Receiver and the Receiver executes the code

-> We can use it to create undo() methods also, not only execute().

-> Con : We create many little classes that store lists of commands

Steps:

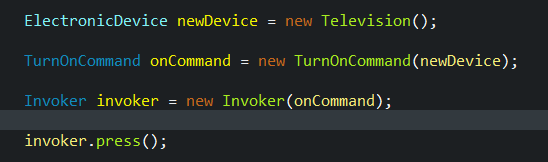
1. We create 1 interfaces

The command interface with execute and undo methods

1. We create the Receiver object, the object on which the methods will be applied(TV)
2. For each command, we implement a concrete class which implements the command interface( If we have undo, we have to create another class for the undo command)

For all the commandClasses, we have an instance variable of the receiver object. For the execute and undo methods, we call the respective methods from the receiver object.

1. We create a class called the invoker which will make the transfer of the command to the receiver. The commander is constructed using a Command instance variable. When the invokerExecute() method will be called, it will call the execute method from the command.
2. To test it, we create a new Receiver object. We create the command object that we want. And we instantiate a receiver using that command object. When we call executeReceiver() the command will be run on the Receiver.



**Chain of Responsibility Design Pattern – Behavioral**

-> a source of command objects and a series of processing objects”.

-> Send data to an object and if that object can’t use it, it sends that same data to any number of other objects that may be able to use it

-> We want to push a command and have more receivers

: Give me a 100 Lei : Give 100 ---If not possible---> give 50 -..

Steps :

1. We define an interface called chain with 2 methods :

setNextChain(Chain chainObject)

defines the next object to receive the data if the current object in the chain can’t do the operations.

solveRequest(Object on which the operations will be done)

1. We define the Object on which the operations will be done ( Numbers)
2. We define the concrete classes implementing the Chain Interface.

Each one will have as an instance variable A chain object representing the next object in the chain. We provide the implementations for the 2 methods

setNextChain(will set the next chain)

solveRequest will try to do the operations it needs or call the solveRequest method for the next in the chain

1. To test, we define all the objects in the chain, we set the nextChain object for all those. We initialize an object for the operations and we call the solveRequest method of the first chain object on that object.

**Design patterns** represent the best practices used by experienced object-oriented software developers. **Design patterns** are solutions to general problems that software developers faced during software development.