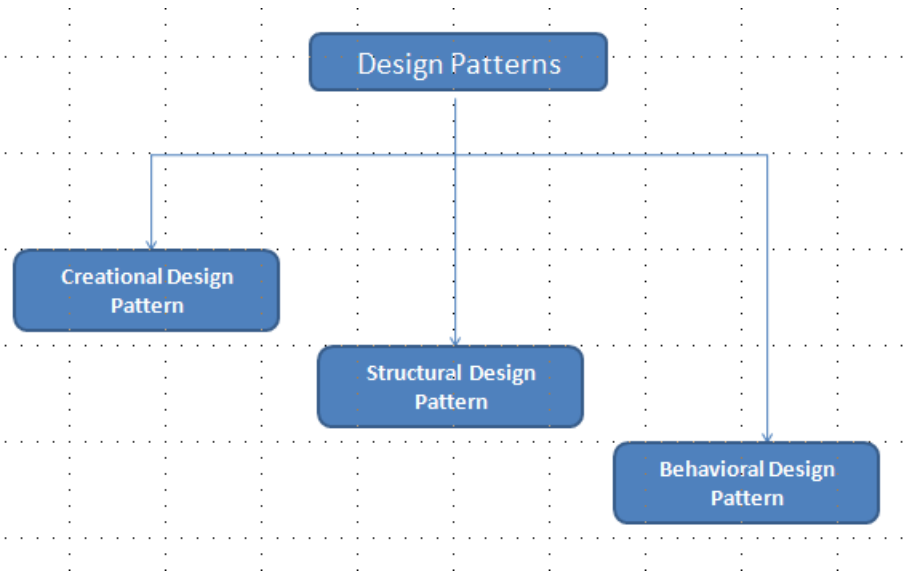
DESIGN PATTERNS

PURPOSE OF DESIGN PATTERNS

- ✓ Design patterns are programming language independent strategies for solving the common object-oriented design problems. (i.e)a design pattern represents an idea, not a particular implementation.
- ✓ By using the design patterns you can make your code more flexible, reusable and maintainable.
- ✓ To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

Categorization:



In this document, we will discuss about Creational Design Patterns.

The sub categories of Creational Design Patterns are:

- ➢ Factory Method Pattern.
- ➢ Abstract Factory Pattern.
- ➢ Singleton Pattern.
- ➢ Prototype Pattern.
- ➢ Builder Pattern.

FACTORY METHOD PATTERN

- In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.
- The idea is to use a static member-function which creates & returns instances, hiding the details of class modules from user.
- Factory Method Pattern says that ***just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate***.
- The Factory Method Pattern is also known as Virtual Constructor.

EXAMPLE:

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

Step 1 : Create an interface *Shape.java*

```
public interface Shape {
  void draw();
}
```

Step 2 : Create concrete classes implementing the same interface.

```
public class Rectangle implements Shape {     //Rectangle.java
  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}

public class Square implements Shape {                    //Square.java
  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}

public class Circle implements Shape {                    //Circle.java
  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}
```

Step 3 : Create a Factory to generate object of concrete class based on given information.

```
public class ShapeFactory {                    // ShapeFactory.java
  //use getShape method to get object of type shape
  public Shape getShape(String shapeType){
    if(shapeType == null){
      return null;
    }
    if(shapeType.equalsIgnoreCase("CIRCLE")){
      return new Circle();
    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();
    } else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }
    return null;
  }
}
```

Step 4 : Use the Factory to get object of concrete class by passing an information such as type.

```
public class FactoryPatternDemo {                    //FactoryPatternDemo.java
  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();
    //get an object of Circle and call its draw method.
    Shape shape1 = shapeFactory.getShape("CIRCLE");
    //call draw method of Circle
    shape1.draw();
    //get an object of Rectangle and call its draw method.
    Shape shape2 = shapeFactory.getShape("RECTANGLE");
    //call draw method of Rectangle
    shape2.draw();
    //get an object of Square and call its draw method.
    Shape shape3 = shapeFactory.getShape("SQUARE");
    //call draw method of square
    shape3.draw();
  }
}
```

OUTPUT :

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

ABSTRACT FACTORY METHOD

- In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes.
- Each generated factory can give the objects as per the Factory pattern.
- Abstract Factory Pattern says that ***just define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.***
- An Abstract Factory Pattern is also known as Kit.

EXAMPLE

Let's take an example, Suppose we want to build a global car factory. If it was factory design pattern, then it was suitable for a single location. But for this pattern, we need multiple locations and some critical design changes.

We need car factories in each location like IndiaCarFactory, USACarFactory and DefaultCarFactory. Now, our application should be smart enough to identify the location where it is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling wrong factory for a particular location.

Here we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to user. This is exactly the problem, which abstract factory pattern is used to solve.

```
enum CarType
{
   MICRO, MINI, LUXURY
}
abstract class Car  //Car.java
 {
          Car(CarType model, Location location)
          {
          this.model = model;
          this.location = location;
           }
          abstract void construct();
          CarType model = null;
          Location location = null;

          CarType getModel()
          {
           return model;
           }
          void setModel(CarType model)
          {
          this.model = model;
          }
```

```
        Location getLocation()
        {
        return location;
         }
        void setLocation(Location location)
        {
         this.location = location;
        }

        @Override
        public String toString()
        {
        return "CarModel - "+model + " located in "+location;
        }
}

class LuxuryCar extends Car // LuxuryCar.java
{
        LuxuryCar(Location location)
        {
            super(CarType.LUXURY, location);
            construct();
        }
        @Override
        protected void construct()
        {
            System.out.println("Connecting to luxury car");
         }
}

class MicroCar extends Car  //MicroCar.java
{
        MicroCar(Location location)
        {
            super(CarType.MICRO, location);
            construct();
         }
        @Override
        protected void construct()
        {
            System.out.println("Connecting to Micro Car ");
        }
}

class MiniCar extends Car //MiniCar.java
{
        MiniCar(Location location)
        {
           super(CarType.MINI,location );
            construct();
        }
```

```
        @Override
        void construct()
        {
            System.out.println("Connecting to Mini car");
         }
}

enum Location
{
 DEFAULT, USA, INDIA
}

class INDIACarFactory  // INDIACarFactory.java
{
    static Car buildCar(CarType model)
    {
        Car car = null;
        switch (model)
        {
        case MICRO:
           car = new MicroCar(Location.INDIA);
           break;

        case MINI:
           car = new MiniCar(Location.INDIA);
           break;

        case LUXURY:
           car = new LuxuryCar(Location.INDIA);
           break;

        default:
        break;
        }
        return car;
    }
}

class DefaultCarFactory  // DefaultCarFactory.java
{
   public static Car buildCar(CarType model)
   {
      Car car = null;

      switch (model)
      {
         case MICRO:
           car = new MicroCar(Location.DEFAULT);
           break;
```

```
            case MINI:
               car = new MiniCar(Location.DEFAULT);
               break;

            case LUXURY:
               car = new LuxuryCar(Location.DEFAULT);
               break;

               default:
               break;

         }
         return car;
      }
}


class USACarFactory // USACarFactory.java
{
   public static Car buildCar(CarType model)
   {
      Car car = null;
      switch (model)
      {
         case MICRO:
            car = new MicroCar(Location.USA);
            break;

         case MINI:
            car = new MiniCar(Location.USA);
            break;

         case LUXURY:
            car = new LuxuryCar(Location.USA);
            break;

            default:
            break;

      }
      return car;
   }
}
```

```
class CarFactory //carFactory.java
{
   private CarFactory()
   {
   }
   public static Car buildCar(CarType type)
   {
      Car car = null;
      // We can add any GPS Function here which
      // read location property somewhere from configuration
      // and use location specific car factory
      // Currently  INDIA is used for Location
      Location location = Location.INDIA;

      switch(location)
      {
         case USA:
            car = USACarFactory.buildCar(type);
            break;

         case INDIA:
            car = INDIACarFactory.buildCar(type);
            break;

         default:
            car = DefaultCarFactory.buildCar(type);

      }
      return car;

   }
}

class AbstractDesign  // AbstractDesign .java
{
   public static void main(String[] args)
   {
      System.out.println(CarFactory.buildCar(CarType.MICRO));
      System.out.println(CarFactory.buildCar(CarType.MINI));
      System.out.println(CarFactory.buildCar(CarType.LUXURY));
   }
}
```

OUTPUT **:**

```
Connecting to Micro Car
CarModel - MICRO located in INDIA
Connecting to Mini car
CarModel - MINI located in INDIA
Connecting to luxury car
CarModel - LUXURY located in INDIA
```

SINGLETON PATTERN

- Sometimes we need to have only one instance of our class.
- For example there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.
- Singleton Pattern says that ***just define a class that has only one instance and provides a global point of access to it*.**
- There are two forms of singleton design pattern

1) **Early Instantiation:** creation of instance at load time.

```
class A{
        private static A obj=new A();//Early, instance will be created at load time
        private A(){}

        public static A getA(){
        return obj;
        }

        public void doSomething(){
        //write your code
        }
}
```

2) **Lazy Instantiation:** creation of instance when required.

```
 class A{
        private static A obj;
        private A(){}

        public static A getA(){
        if (obj == null){
        synchronized(Singleton.class){
        if (obj == null){
        obj = new Singleton(); //instance will be created at request time
        } }  }
  return obj;
 }

 public void doSomething(){
 //write your code    } }
```

EXAMPLE :

Step 1 : **Create a Singleton Class.** *SingleObject.java*

```
public class SingleObject {
//create an object of SingleObject
private static SingleObject instance = new SingleObject();
//make the constructor private so that this class cannot be
//instantiated
private SingleObject(){}
//Get the only object available
 public static SingleObject getInstance(){
return instance; }
public void showMessage(){
System.out.println("Hello World!");
} }
```

Step 2 : Get the only object from the singleton class. *SingletonPatternDemo.java*

```
public class SingletonPatternDemo {
        public static void main(String[] args) {
           SingleObject object = new SingleObject();
           //Get the only object available
           SingleObject object = SingleObject.getInstance();
           //show the message
           object.showMessage();
} }
```

OUTPUT :

```
Hello World!
```

PROTOTYPE DESIGN PATTERN

- The concept is to copy an existing object rather than creating a new instance from scratch.
- The existing object acts as a prototype and contains the state of the object.
- The newly copied object may change same properties only if required.
- Prototype Pattern says that ***cloning of an existing object instead of creating new one and can also be customized as per the requirement***.

EXAMPLE :

We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.

PrototypPatternDemo, our demo class will use *ShapeCache* class to get a *Shape* object.

Step 1 : Create an abstract class implementing *Clonable* interface. *Shape.java*

```
public abstract class Shape implements Cloneable {

  private String id;
  protected String type;

  abstract void draw();

  public String getType(){
    return type;
  }
  public String getId() {
    return id;
  }
  public void setId(String id) {
    this.id = id;
  }
  public Object clone() {
    Object clone = null;

    try {
      clone = super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return clone;
  }
}
```

Step 2 : Create concrete classes extending the above class.

```
public class Rectangle extends Shape {   // Rectangle.java

  public Rectangle(){
    type = "Rectangle";
  }
  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}

public class Square extends Shape { // Square.java

  public Square(){
    type = "Square";
  }
  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}

public class Circle extends Shape { //Circle.java

  public Circle(){
    type = "Circle";
  }
  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}
```

Step 3 : Create a class to get concrete classes from database and store them in a *Hashtable*.

```
public class ShapeCache { // ShapeCache.java
  private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();
  public static Shape getShape(String shapeId) {
    Shape cachedShape = shapeMap.get(shapeId);
    return (Shape) cachedShape.clone();
  }

  // for each shape run database query and create shape
  // shapeMap.put(shapeKey, shape);
  // for example, we are adding three shapes

  public static void loadCache() {
    Circle circle = new Circle();
    circle.setId("1");
```

```
    shapeMap.put(circle.getId(),circle);

    Square square = new Square();
    square.setId("2");
    shapeMap.put(square.getId(),square);

    Rectangle rectangle = new Rectangle();
    rectangle.setId("3");
    shapeMap.put(rectangle.getId(), rectangle);
  }
}
```

Step 4 : *PrototypePatternDemo* uses *ShapeCache* class to get clones of shapes stored in a *Hashtable*.

```
public class PrototypePatternDemo { //PrototypePatternDemo.java
  public static void main(String[] args) {
    ShapeCache.loadCache();

    Shape clonedShape = (Shape) ShapeCache.getShape("1");
    System.out.println("Shape : " + clonedShape.getType());

    Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
    System.out.println("Shape : " + clonedShape2.getType());

    Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
    System.out.println("Shape : " + clonedShape3.getType());
  }
}
```

OUTPUT:

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

BUILDER DESIGN PATTERN

- Builder pattern aims to separate the construction of a complex object from its representation so that the same construction process can create different representations.
- It is used to construct a complex object step by step and the final step will return the object.
- The process of constructing an object should be generic so that it can be used to create different representations of the same object.
- Builder Pattern says that **construct a complex object from simple objects using step-by-step approach** .

EXAMPLE:

We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle. We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*. *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.

Step 1 : Create an interface Item representing food item and packing.

```
public interface Item { //Item.java
  public String name();
  public Packing packing();
  public float price();
}
public interface Packing { //Packing.java
  public String pack();
}
```

Step 2 : Create concrete classes implementing the Packing interface.

```
public class Wrapper implements Packing {  // Wrapper.java
  @Override
  public String pack() {
    return "Wrapper";
  }
}

public class Bottle implements Packing { //Bottle.java
  @Override
  public String pack() {
    return "Bottle";
  }
}
```

Step3 : Create abstract classes implementing the item interface providing default functionalities.

```
public abstract class Burger implements Item { //Burger.java
  @Override
  public Packing packing() {
    return new Wrapper();
  }
  @Override
  public abstract float price();
}
```

```
public abstract class ColdDrink implements Item { //ColdDrink.java
    @Override
    public Packing packing() {
    return new Bottle();
    }
    @Override
    public abstract float price();
}
```

Step 4 : Create concrete classes extending Burger and ColdDrink classes

```
public class VegBurger extends Burger { // VegBurger.java
  @Override
  public float price() {
    return 25.0f;
  }
  @Override
  public String name() {
    return "Veg Burger";
  }
}

public class ChickenBurger extends Burger { // ChickenBurger.java
  @Override
  public float price() {
    return 50.5f;
  }
  @Override
  public String name() {
    return "Chicken Burger";
  }
}

public class Coke extends ColdDrink { //Coke.java
  @Override
  public float price() {
    return 30.0f;
  }
  @Override
  public String name() {
    return "Coke";
  }
}
public class Pepsi extends ColdDrink { //Pepsi.java
  @Override
  public float price() {
    return 35.0f;
  }
  @Override
  public String name() {
    return "Pepsi";
  }
```

```
}
```

Step 5 : Create a Meal class having Item objects defined above.

```
public class Meal { //Meal.java
        private List<Item> items = new ArrayList<Item>();

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

        public float getCost(){
           float cost = 0.0f;

          for (Item item : items) {
             cost += item.price();
        }
         return cost;
  }

  public void showItems(){

    for (Item item : items) {
      System.out.print("Item : " + item.name());
      System.out.print(", Packing : " + item.packing().pack());
      System.out.println(", Price : " + item.price());
    }
  }
}
```

Step 6 : Create a MealBuilder class, the actual builder class responsible to create Meal objects.

```
public class MealBuilder { //MealBuilder.java

  public Meal prepareVegMeal (){
    Meal meal = new Meal();
    meal.addItem(new VegBurger());
    meal.addItem(new Coke());
    return meal;
  }

  public Meal prepareNonVegMeal (){
    Meal meal = new Meal();
    meal.addItem(new ChickenBurger());
    meal.addItem(new Pepsi());
    return meal;
  }
}
```

Step 7 : BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

```
public class BuilderPatternDemo { // BuilderPatternDemo.java
  public static void main(String[] args) {

    MealBuilder mealBuilder = new MealBuilder();

    Meal vegMeal = mealBuilder.prepareVegMeal();
    System.out.println("Veg Meal");
    vegMeal.showItems();
    System.out.println("Total Cost: " + vegMeal.getCost());

    Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
    System.out.println("\n\nNon-Veg Meal");
    nonVegMeal.showItems();
    System.out.println("Total Cost: " + nonVegMeal.getCost());
  }
}
```

OUTPUT :

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0


Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```