Design Patterns

Software Design Pattern can be defined as a **software template** or a description to solve a problem that occurs in multiple instances while designing a Software Application or a Software Framework.

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. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

The credit of Design Patterns goes to the **Gang of Four.** Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides were the authors of the book on the Java Design Patterns. According to these authors design patterns are primarily based on the following principles of object orientated design.

* Program to an interface not an implementation
* Favor object composition over inheritance

**Structure of a Design Pattern**

**Structure** of any Design Pattern is considered to be highly **organised.** They are often documented in the form of a **template** such that, the users can **visually identify** the problem and instantaneously find the solution to it, based on the **relationship** between **classes** and **objects.** The Design Patterns Template described by the original authors of the Design Patterns is as follows:

|  |  |
| --- | --- |
| **Term** | **Description** |
| **Pattern Name** | Describes the essence of the pattern in a short, but expressive name. |
| **Intent** | Describes what the Pattern Does |
| **Also, known as** | List any synonyms of the pattern |
| **Motivation** | Provides an example of the problem and describes the solution to it |
| **Applicability** | Lists the situations where the pattern is applicable |
| **Structure** | Set of diagrams of the classes and objects that depict the pattern |
| **Participants** | Describes the classes and objects that participate in the design |
| **Collaborations** | Description of how classes and objects used in the pattern interact with each other. |
| **Consequences** | A description of the results, side effects, and trade-offs caused by using the pattern. |

Java Design Patterns are divided into **Four Categories** and each of those are further classified as below:



* **Creational Design Patterns**are concerned with the method of creating Objects.
* **Structural Design Patterns**deal with the composition of classes and objects which form larger structures.
* **Behaviour Design Patterns**are concerned with the responsibility and interaction between the objects.
* **JEE Design Patterns**are concerned with providing solutions to the Java EE-based software applications and frameworks.

Why Design Patterns?

* Design patterns can speed up the development process by providing tested, proven development paradigms
* Effective software design requires considering issues that may not become visible until later in the implementation
* Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns
* Often, people only understand how to apply certain software design techniques to certain problems
* These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a problem
* In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs

**Creational design patterns**

Creational design patterns are concerned with**the way of creating objects.** These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

But everyone knows an object is created by using new keyword in java. For example:

StudentRecord s1=**new** StudentRecord();

Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

* They reduce complexities and instability by creating objects in a controlled manner
* The new operator is often considered harmful as it scatters objects all over the application
* Over time it can become challenging to change an implementation because classes become tightly coupled
* Creational Design Patterns address this issue by decoupling the client entirely from the actual initialization process

Types of creational design patterns

There are following 6 types of creational design patterns.

* Factory Method Pattern
* Abstract Factory Pattern
* Singleton Pattern
* Prototype Pattern
* Builder Pattern
* Object Pool Pattern

Factory Method Pattern

A normal factory produces goods; a software factory produces objects. And not just that — it does so without specifying the exact class of the object to be created. To accomplish this, objects are created by calling a factory method instead of calling a constructor.

Usually, object creation in Java takes place like so:

*SomeClass someClassObject = new SomeClass();*

The problem with the above approach is that the code using the **SomeClass**’s object, suddenly now becomes dependent on the concrete implementation of **SomeClass**. There’s nothing wrong with using **new** to create objects but it comes with the baggage of tightly coupling our code to the concrete implementation class, which can occasionally be problematic.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

#### **Advantage of Factory Design Pattern**

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### **Usage of Factory Design Pattern**

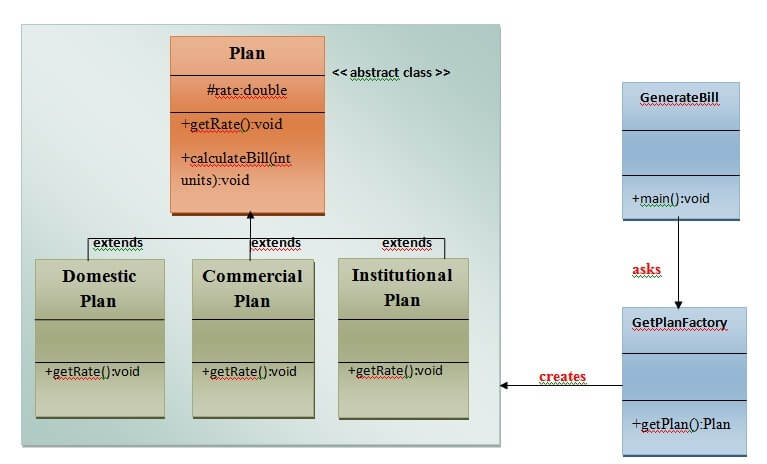
* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

## **Implementation**

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.

*FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.





Abstract Factory Method

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

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.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

#### **Advantage of Abstract Factory Pattern**

* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

#### **Usage of Abstract Factory Pattern**

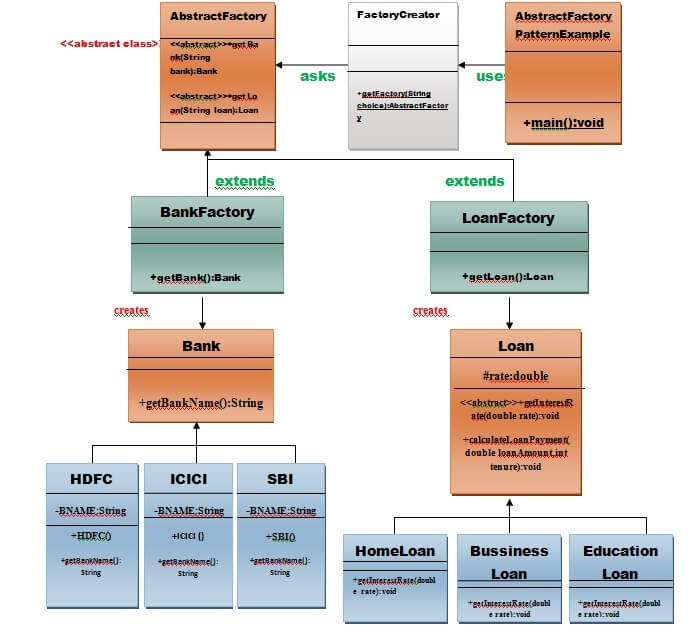
* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

Implementation

We are going to create a Shape interface and a concrete class implementing it. We create an abstract factory class AbstractFactory as next step. Factory class ShapeFactory is defined, which extends AbstractFactory. A factory creator/generator class FactoryProducer is created.

AbstractFactoryPatternDemo, our demo class uses FactoryProducer to get a AbstractFactory object. It will pass information (CIRCLE / RECTANGLE / SQUARE for Shape) to AbstractFactory to get the type of object it needs.





Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

#### **Advantage of Prototype Pattern**

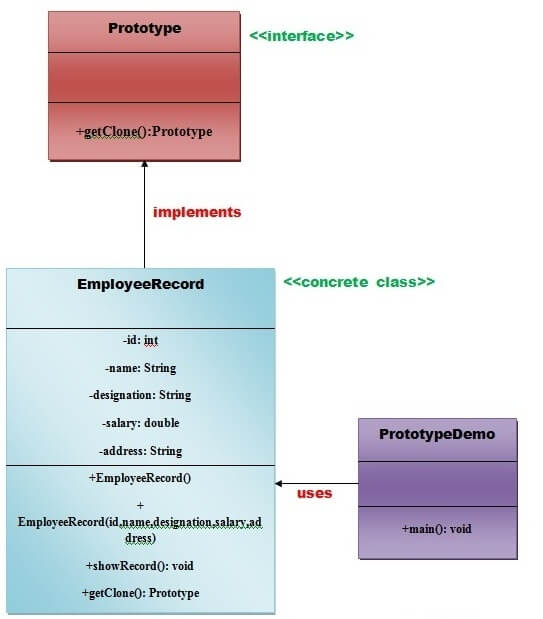
The main advantages of prototype pattern are as follows

* It reduces the need of sub-classing.
* It hides complexities of creating objects.
* The clients can get new objects without knowing which type of object it will be.
* It lets you add or remove objects at runtime.

#### **Usage of Prototype Pattern**

* When the classes are instantiated at runtime.
* When the cost of creating an object is expensive or complicated.
* When you want to keep the number of classes in an application minimum.
* When the client application needs to be unaware of object creation and representation.

## **Implementation**



Builder Design Pattern

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

#### **Advantage of Builder Design Pattern**

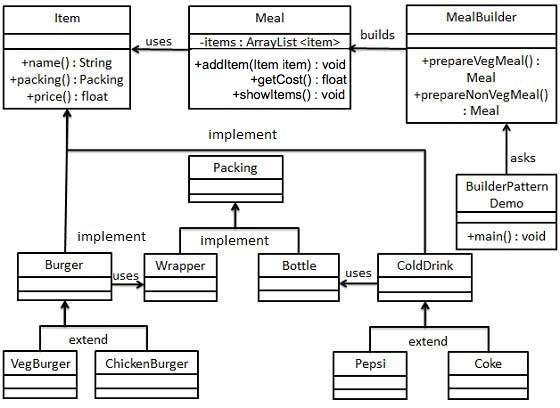
* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects

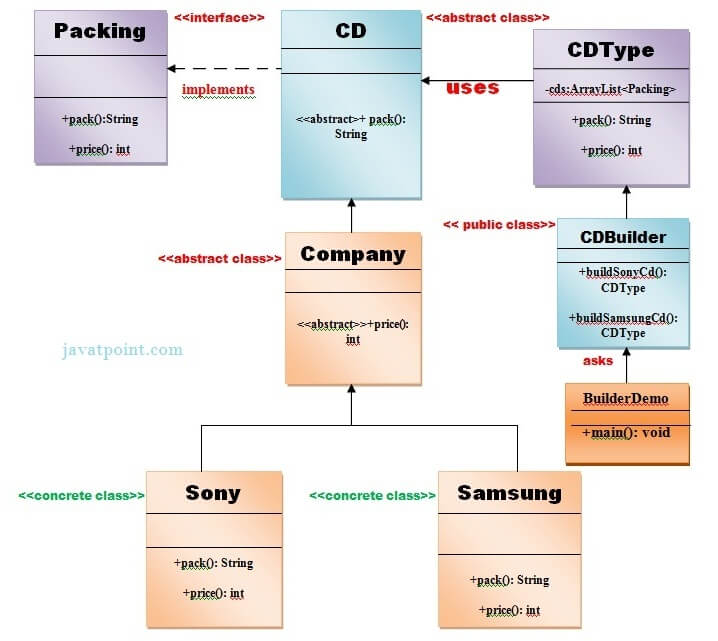
## **Implementation**

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*.





# Structural design patterns

**Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns **simplifies the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

## **Types of structural design patterns**

* Adapter Pattern
* Bridge Pattern
* Composite Pattern
* Decorator Pattern
* Façade Pattern
* Flyweight Pattern
* Proxy Pattern

**Composite Pattern**

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

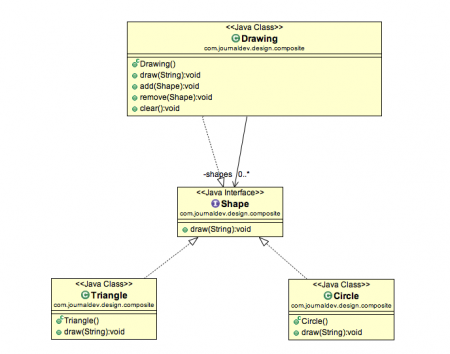
A Composite Pattern says that just **"allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects".**

#### **Advantage of Composite Design Pattern**

* It defines class hierarchies that contain primitive and complex objects.
* It makes easier to you to add new kinds of components.
* It provides flexibility of structure with manageable class or interface.

#### **Usage of Composite Pattern**

It is used:

* When you want to represent a full or partial hierarchy of objects.
* When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.
* 

**Composite Pattern Base Component** - Composite pattern base component defines the common methods for leaf and composites. We can create a class Shape with a method draw(String fillColor) to draw the shape with given color

**Composite Design Pattern Leaf Objects** Composite design pattern leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

**Composite object** A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

# Decorator Pattern

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as **Wrapper.**

Decorator pattern allows a user to add new functionality to an existing object without altering its structure.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

#### **Advantage of Decorator Pattern**

* It provides greater flexibility than static inheritance.
* It enhances the extensibility of the object, because changes are made by coding new classes.
* It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

#### **Usage of Decorator Pattern**

It is used:

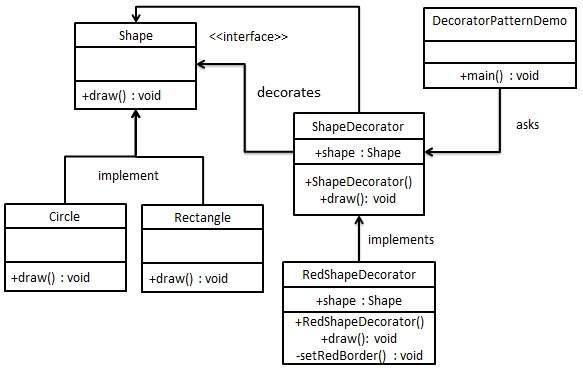
* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

## **Implementation**

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. We will then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable.

*RedShapeDecorator* is concrete class implementing *ShapeDecorator*.

*DecoratorPatternDemo*, our demo class will use *RedShapeDecorator* to decorate *Shape* objects



**Behavioral Design Patterns**

Behavioral design patterns are concerned with **the interaction and responsibility of objects.**

In these design patterns,**the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.**

That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

#### **There are 12 types of structural design patterns:**

1. Chain of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern
12. Null Object

Strategy Pattern

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

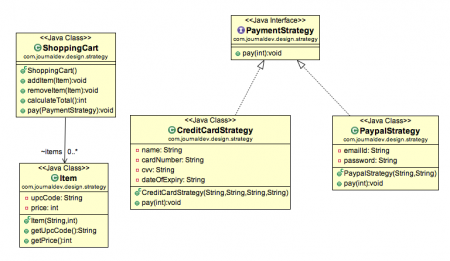
Strategy pattern is also known as **Policy Pattern**. We define multiple algorithms and let client application pass the algorithm to be used as a parameter.

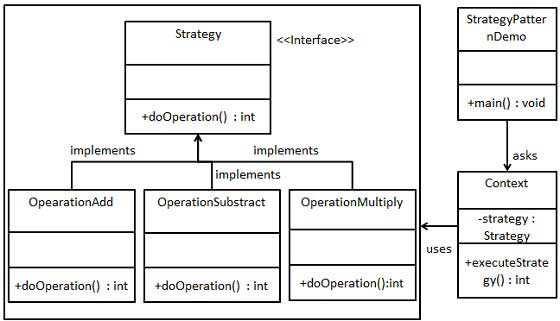
## **Benefits:**

* It provides a substitute to subclassing.
* It defines each behavior within its own class, eliminating the need for conditional statements.
* It makes it easier to extend and incorporate new behavior without changing the application.

## **Usage:**

* When the multiple classes differ only in their behaviors.
* It is used when you need different variations of an algorithm.





## Visitor Design Pattern

Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

### Visitor Pattern Benefits

The benefit of this pattern is that if the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.

Another benefit is that adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

For example, think of a Shopping cart where we can add different type of items (Elements). When we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern. Let’s implement this in our example of visitor pattern.

