# Creational Design Pattern

## Factory

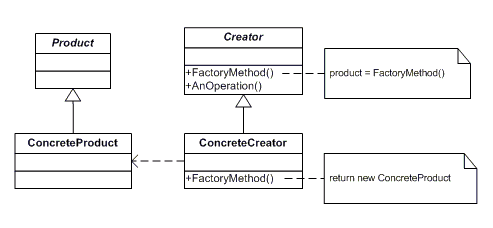
This pattern introduces loose coupling between classes which is the most important principle one should consider and apply while designing the application architecture. Loose coupling can be introduced in application architecture by programming against abstract entities rather than concrete implementations. This not only makes our architecture more flexible but also less fragile.

Think Coupling When Using NEW

When you are using “new” keyword, you are not just creating an object but under the hood you are also introducing coupling within your application. It doesn’t mean you don’t have to use it anymore as it’s the only way of instantiating an object, but we should find a way to do it more effectively because we have to cater to an important factor in software engineering (or even in life which remains constant) and that’s “CHANGE”.

Factory defines an interface for creating an object, but lets the classes that implement the interface decide which class to instantiate. Factory Pattern lets a class postpone instantiation to sub-classes.

The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.



Product : defines the interface of objects the factory method creates

ConcreteProduct : is a class which implements the Product interface.

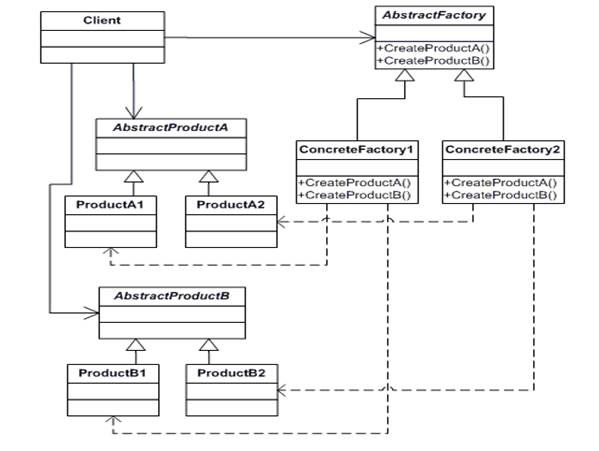
Creator : is an abstract class and declares the factory method, which returns an object of type Product. This may also define a default implementation of the factory method that returns a default ConcreteProduct object. This may call the factory method to create a Product object.

ConcreteCreator : is a class which implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct.

## Abstract Factory

We define an interface which will create families of related or dependent objects. In simple words, interface will expose multiple methods each of which will create some object. Again, here method return types will be generic interfaces. All these objects will together become part of some important functionality.

* Creates object through composition
* Produce families of products
* Concrete factories implements factory method to create product



AbstractFactory : is an interface for operations which is used to create abstract product.

ConcreteFactory : is a class which implements the AbstractFactory interface operations to create concrete products.

AbstractProduct : declares an interface for a type of product object

Product : defines a product object to be created by the corresponding concrete factory also implements the AbstractProduct interface

Client : is a class which uses AbstractFactory and AbstractProduct interfaces to create a family of related objects.

## Builder

### Background

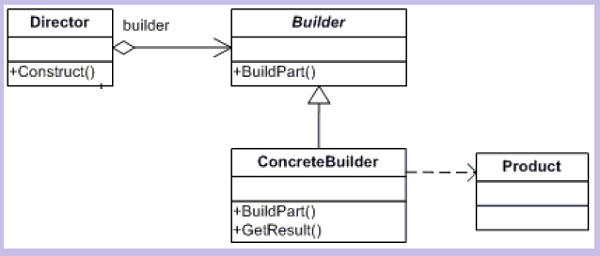
When we have an application that need to create an object which has to be constructed using many different objects, we find our client code cluttered with the details of the various Part objects that needs to be assembled together to create the resulting object.

To illustrate on above point let us take an example of mobile phone manufacturing system. Let’s assume that we have a system installed at one of the mobile phone vendors. Now the manufacturer may decide to create a phone based on parameters like Touchscreen, Operating System, Battery and Stylus. Now if we have the objects for all these parts then creation of product with any combination of above parts would lead to a very complex and unmanageable code in the client application i.e. the module that will decide what kind of phone needs to be built.

Builder pattern is meant to solve such problems. GoF defines Builder pattern as:

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

What this means is that we will have to design the system in such a way that the client application will simply specify the parameters that should be used to create the complex object and the builder will take care of building the complex object. Let us visualize the class diagram of Builder Pattern.



Now let’s see what each class in the above class diagram is meant for

ConcreteBuilder: Concrete classes that will create the complex Product. this will keep track of what Product it has created i.e. assembled what parts and this will be used by the client to get the Product object.

Builder: This is the interface for creating the actual products (it should provide the functions for creating each of the parts for any product)

Director: This is the Client code that will specify the parts needs to be put together to create the actual concrete Product.

Now if we need to create more Products only a ConcreteBuilder would be needed and rest all the codebase will remain the same. The client code can also create complex Products easily with this pattern in place.

Example



## Prototype

## Singleton

# Structural Design Patterns

## Adapter

## Bridge

## Composite

## Decorator

## Façade

## Flyweight

## Proxy

# Behavioral Design Patterns

## Chain of Responsibility

## Command

## Interpreter

## Iterator

## Mediator

## Memento

## Observer

## State

## Strategy

## Visitor

## Template Method