Sandeep Singh

8 Followers About Follow

Factory Design Pattern

Sandeep Singh Mar 28, 2020 · 3 min read

There are 3 types of factory design pattern.

1. Simple factory
2. Factory method
3. Abstract factory

Basically, *Simple factory* is not a design pattern. So, we will not talk about this rather we will cover its relevance while studying the factory method and in this post, I will only be explaining the *factory* method.

Now, what factory method is?

Factory Method is a creational design pattern that provides an interface for object creation. It says when you want to instantiate or create an object (let's encapsulate that instantiation so that we can make it uniform across places) then call that *factory* and that *factory* is responsible for instantiating it appropriately.

Here, basically we are abstracting the construction of the object away from the place that is actually using it and the *factory* is responsible for object creation.

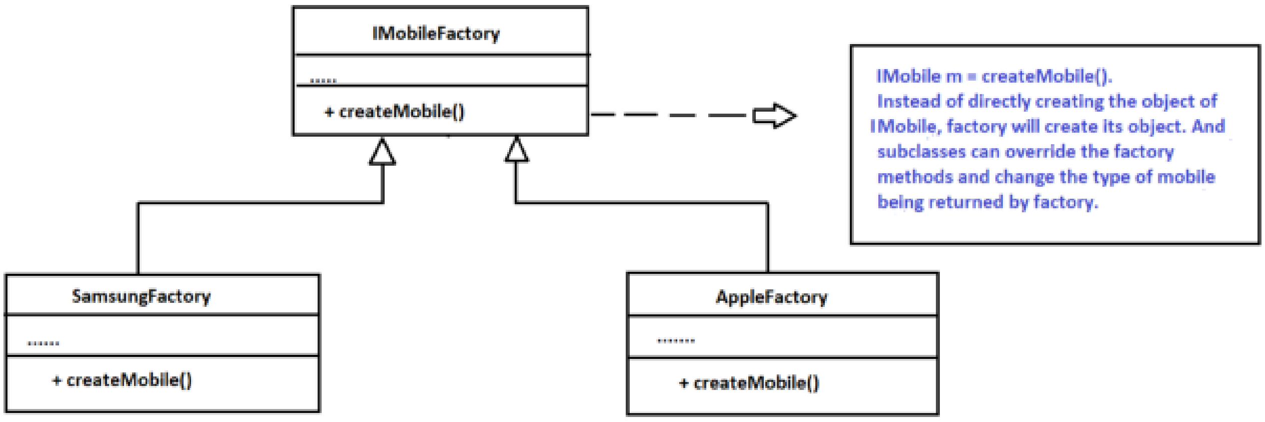
What problem does it solve?

Imagine you are creating a mobile manufacturing system. Initially, the system only manufactures **Samsung** mobiles. So most of your code resides in **Samsung** class.

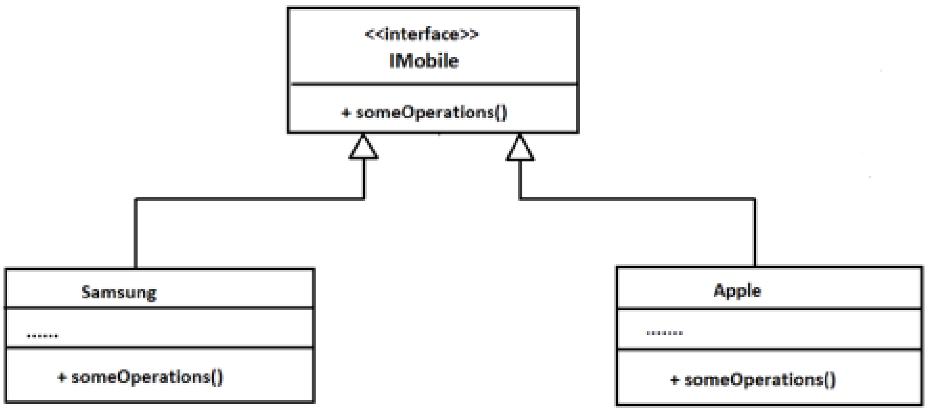
At present, most of your code is coupled to the **Samsung** class, adding **Apple** into the system would require making changes to the entire codebase. As a result, we will end up with conditions that switch the system's behavior depending on the class of manufacturing objects.

Solution

According to *factory* method design pattern, direct object construction calls (using the **new** operator) replaced by calls to a special *factory* method, and within the *factory* method, objects are still being created by the **new** operator.


IMobileFactory
......
+ createMobile()
IMobile m = createMobile().
Instead of directly creating the object of IMobile, factory will create its object. And subclasses can override the factory methods and change the type of mobile being returned by factory.
SamsungFactory
....
+ createMobile()
AppleFactory
....
+ createMobile()

Subclasses can alter the class of objects being returned by the factory method.



Samsung and Apple will implement 'IMobile' interface.

‘AppleFactory’. These concrete classes control object creation, both these factories will implement ***createMobile*** method. The factory method in **SamsungFactory** will return **Samsung** object whereas the factory method in **AppleFactory** will return **Apple** object.

> *In Simple factory pattern, ‘MobileExecutioner’ will return the instance of ‘IMobile’ instead of ‘IMobileFactory’.*

Sample Code

```
public interface IMobile{
  void Operation();
}
public class Samsung : IMobile
{
  public void Operation()
  {
    Console.WriteLine(“Samsung Mobile”);
  }
}
public class Apple : IMobile
{
  public void Operation()
  {
     Console.WriteLine(“Apple Mobile”);
  }
}
// IMobile Factory
public interface IMobileFactory
{
  IMobile createMobile();
}
// IMobile Factory concrete implementation
public class SamsumgFactory : IMobileFactory
{
  public IMobile createMobile()
  {
    new Samsung();
  }
}
// IMobile Factory concrete implementation
public class AppleFactory : IMobileFactory
{
  }
}
public class MobileExecutioner
{
  public static IMobileFactory Instance(string type)
  {
    switch (type)
    {
      case “Samsung”:
      {
       return new SamsumgFactory();
      }
      case “Apple”:
      {
        return new AppleFactory();
      }
      default:
        return new SamsumgFactory();
    }
  }
}
class MainApp
{
/// <summary>
/// Entry point into console application.
/// </summary>
  static void Main()
  {
    string type = string.Empty;
    //Reading user input i.e. type of mobile (SAMSUNG OR APPLE)
    type = Console.Readline();
    IMobileFactory m = MobileExecutioner.Instance(type);
    m.createMobile().Operation();
  }
}
```

But imagine if we have a lot of factory classes, which is quite common in large projects. That would lead to a quite big switch case statement which is quite unreadable. So to improve it, we can create an enum to specify the type of object and execute the appropriate factory.

```
public enum MobileType{
```

Get started Open in app

```

public class MobileExecutioner
{
  private readonly Dictionary<MobileType, IMobileFactory> _factories;
  public MobileExecutioner()
  {
    _factories = new Dictionary<MobileType, IMobileFactory>();

    foreach (Actions action in Enum.GetValues(typeof(Actions)))
      {
        var factory = (IMobileFactory)Activator.CreateInstance(Type.GetType(Enum.GetName(typeof

         _factories.Add(action, factory);
      }

   }
}
```

FactoryMethodImproved.cs hosted with ♡ by GitHub view raw

Conclusion (When to use Factory Method Pattern)

- When you don't know beforehand the exact type of object your application code use.
- With the *factory* method pattern, instead of rebuilding the object every time, you can reuse the existing ones.
- The main reason for which the factory method pattern is used is that it introduces a separation between the application and a family of classes. It provides a simple way of extending the family of products with minor changes in application code.

Credits

- Factory Method Pattern — Wikipedia
- Head First Design Pattern

Design Patterns Web Development Factory Design Pattern