

[articles](#) [Q&A](#) [forums](#) [stuff](#) [lounge](#) [?](#)Search for articles, questions, 

# Factory Patterns - Simple Factory Pattern

**Snesh Prajapati**25 Sep 2016 [CPOL](#)**Rate me:**  4.88/5 (49 votes)

In this article series, we will learn about different factory design patterns. There are three kinds of factory design patterns, namely, Simple Factory Pattern, Factory Method Pattern and Abstract Factory Pattern.

[Download source code - 12.2 KB](#)

## Introduction

In this article series, we will learn about different factory design patterns. There are three kinds of factory design patterns, namely, Simple Factory Pattern, Factory Method Pattern and Abstract Factory Pattern. We will understand these three patterns in detail by learning how to implement, when to implement and subsequently we will understand the differences between those. Simple Factory Pattern is not a part of [Gang of Four \(GoF\)](#) book but Factory Method Pattern and Abstract Factory Patterns are part of this standard book.

To keep the size of articles reasonable, this article series is divided into three parts.

## Outlines

### Part 1 - Simple Factory Pattern

- What are Design Patterns
- How Design Patterns help us
- Simple Factory Pattern
- Problems without Simple Factory Pattern
- Resolution with Simple Factory Pattern
- Problem with Simple Factory Pattern

### Part 2 - Factory Method Pattern

- What is Factory Method Pattern
- Understand Definition with Example
- When to use Factory Method Pattern
- Advantages of Factory Method Pattern

### Part 3 - Abstract Factory Pattern

- What is Abstract Factory Pattern
- When to use Abstract Factory Pattern
- Understand Definition with Example
- Problem with Abstract Factory Pattern

## What are Design Patterns

As [wikipedia](#) says, "Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design"

In other words, we can say that Design Pattern provides proven solution to solve commonly occurring problems in software (generally big) applications. Implementation of a particular design pattern in any application increases flexibility, maintainability and readability of the application. *Before implementing any design pattern in the application, first we should be clear about the problem that can be solved by a particular design pattern.*

## How Design Patterns Help Us

Application can be developed without implementing design patterns. Without using design pattern, initially application development time will be less but as application functionalities grow, you will realize that it is very difficult to change, maintain or understand the design. That's where Design Pattern helps us to identify a general issue and solve it in the best possible manner.

The first and most important part while working with Design Patterns is to understand the context and exact issue in application. Once the problem is identified, then it is easy to figure out which Design Pattern should be used to solve identified problem.

## What is Simple Factory Pattern

According to [definition from wikipedia](#), Factory Pattern is "A factory is an object for creating other objects". Simple Factory Pattern is a **Factory** class in its simplest form (In comparison to Factory Method Pattern or Abstract Factory Pattern). In another way, we can say: ***In simple factory pattern, we have a factory class which has a method that returns different types of object based on given input.***

Let us understand with an example:

## Overview of Example

To understand Simple Factory Pattern practically, we take an example of Electrical Company which makes different kinds of fans, we will call it **FanFactory**. But first, we will implement this scenario without using Simple Factory Pattern, then will see the problems and how those can be solved with this Pattern.

The below program is a simple console application, in this program, **Main** method is used as client which creates a **TableFan**. In simplest implementation, we can see that the client is able to create a **TableFan** directly as per need (without any factory).

C# Copy Code

```
class Program
{
    static void Main(string[] args)
    {
        TableFan fan = new TableFan();
        fan.SwitchOn();
    }
}

class TableFan { }
```

## Problems Without Simple Factory Pattern

In the above example, we did not use any pattern and the application is working fine. But if we think about future possible changes and look closely, we can foresee the following problems with the current implementation:

1. In the current application, wherever a particular fan is needed, it is being created using concrete class. In future, if there is any change in class name/ different concrete class is proposed, you have to make change all over the application. For example, instead of having **TableFan** class, we need to introduce **BasicTableFan** (which should be used now in place of old **TableFan** class) and **PowerTableFan** classes. So any changes with respect to **Fan** classes will make it difficult to maintain the code and requires many changes at many places.
2. Currently, when client creates **TableFan** object, constructor of **TableFan** class is not taking any argument so **Fan** creation process is easy. But later **TableFan** object creation process gets changed and what if constructor of **Fan** class is expecting two objects as arguments. Then every place in code client needs to make changes wherever **Fan** object was created. For example:

C#

Copy Code

```
TableFan fan = new TableFan(Moter moter, BladType bladType);
```

3. In **TableFan fan = new TableFan(Moter moter, BladType bladType);** code when **Fan** object is created, it takes two types **Moter**, **BladType**. In the above code, the client is aware about classes and knows the object creation process as well, which should be internal detail of **FanFactory**. We should avoid exposing such object creation detail and internal classes to client application.
4. In certain cases, the lifetime management of the created objects must be centralized to ensure a consistent behavior within the application. That cannot be done if client is free to create concrete object the way he wishes. Such scenario often occurs in Cache Management and DI frameworks.
5. In C# and Java, the constructor of a class must have the same name as the name of that class. Let's say there is a need to have descriptive names for constructor like **CreateTableFanByModelNumber(int modelNumber)**. In **TableFan** class, at best, we can have a constructor like **Fan(int modelNumber)** but the name is not as descriptive as its intent. Another similar example on the issue of descriptive name is available at [this wikiedia page](#) too.

## Resolution with Simple Factory Pattern

To come out with the above problems, we can use Simple Factory Pattern because this pattern is suitable to solve the above mentioned problems. Further, we will continue with the same example and will modify the existing code.

When we implement Simple Factory Pattern, a class needs to be created which will have method to return requested instance of an object. Let's create a class called "**FanFactory**" which will implement an interface called "**IFanFactory**". This interface has a method called **IFan CreateFan(FanType type);** which takes **enum FanType** and returns respective instance of a **Fan**.

Now **Client** will not be aware about the concrete classes like **TableFan** or **CeilingFan**. Client will be using **FanType enum** and **IFan interface**. Based on passed **enum** as argument while calling "**CreateFan**" method, **FanFactory** will return the instance of desired fan. Following is the modified code of application:

C#

Shrink ▲ Copy Code

```
enum FanType
{
    TableFan,
    CeilingFan,
    ExhaustFan
}

interface IFan
{
    void SwitchOn();
    void SwitchOff();
}

class TableFan : IFan {.... }

class CeilingFan : IFan {.... }
```

```

class ExhaustFan : IFan {..... }

interface IFanFactory
{
    IFan CreateFan(FanType type);
}

class FanFactory : IFanFactory
{
    public IFan CreateFan(FanType type)
    {
        switch (type)
        {
            case FanType.TableFan:
                return new TableFan();
            case FanType.CeilingFan:
                return new CeilingFan();
            case FanType.ExhaustFan:
                return new ExhaustFan();
            default:
                return new TableFan();
        }
    }
}

//The client code is as follows:

static void Main(string[] args)
{
    IFanFactory simpleFactory = new FanFactory();
    // Creation of a Fan using Simple Factory
    IFan fan = simpleFactory.CreateFan(FanType.TableFan);
    // Use created object
    fan.SwitchOn();

    Console.ReadLine();
}

```

Now we will **confirm how Simple Factory Pattern solved all the above mentioned problems** under "Problem without implementation Simple Factory Pattern" section:

1. Now client is using interfaces and **FanFactory**, so if we change the name of concrete class of **Fan** which gets created for a given **enum** value, we need to change at only one place, that is, inside "**CreateFan**" method of **FanFactory**. Client code is not affected at all.
2. If later, the **Fan** object creation process gets changed and constructor of **Fan** class is expecting two or more objects as arguments, we need to change only inside "**CreateFan**" method of **FanFactory**. Client code is not affected at all.
3. Using **FanFactory**, all internal details will be hidden from client. So it is good in terms of abstractions and security.
4. If needed, we can write logic for lifetime management along with objects creation at centralized **FanFactory**.
5. While using **FanFactory** we can simply have methods with different and more descriptive names which would return an object of **IFan**. In our example application, **FanFactory** can have a public method **CreateTableFanByModelNumber(int modelNumber)** exposed to client.

## Problem with Simple Factory Pattern

Let's say in future, if **FanFactory** has to make a new type of fan called **WallFan** also. To adopt this new requirement, we have to change **CreateFan** method and add one **switch** case for **WallFan** type. If again new kind of **Fan** is introduced, then again one more case needs to be added. This will be a violation of Open Close Principle of **SOLID principles**. In the next article, we will see how we can overcome this violation issue with the help of Factory Method Pattern.

## Conclusion

In this article, we had a walkthrough to learn Simple Factory Pattern and its use. We understood the context of Simple Factory Pattern and how to use it to enhance maintainability of application. But it violates the Open Close Principle, so in the [next article](#), we will see how we can have better design with [Factory Method Pattern](#). Thanks for reading! Your comments and suggestions for improvement are most welcome.

## References

- [Discussion on Motivation for Simple Factory](#)
- [Corey Broderick's Blog](#)
- [Coding Geek Blog](#)
- [Article on Factory Patterns](#)

## License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

## Share

## About the Author



### Snesh Prajapati

Software Developer

India 

I am a Software Developer working on Microsoft technologies. My interest is exploring and sharing the awesomeness of emerging technologies.

## Comments and Discussions

You must [Sign In](#) to use this message board.

**Still one of the best article on factory** **\_Dhull** 1-Feb-19 2:31

Shrink ▲ Copy Code

Still one of the best article on factory

"If you can't explain it simply, you don't understand it well enough"

- Albert Einstein

Sign In · View Thread

**Bad Example** **Member 13833966** 18-May-18 7:09**Thank you Snesh i have a questions** **Member 13601376** 31-Dec-17 9:36**Thanks** **Darin Sease** 13-Oct-16 17:58

Re: Thanks

**Snesh Prajapati** 13-Oct-16 18:22**Thanks** **Recep SELLI** 7-Oct-16 1:11

Re: Thanks

**Snesh Prajapati** 9-Oct-16 19:00**Your code is one step ahead** **Stefan\_Lang** 28-Sep-16 2:35

Re: Your code is one step ahead

**Snesh Prajapati** 28-Sep-16 6:56

Re: Your code is one step ahead

**Stefan\_Lang** 29-Sep-16 0:20

Re: Your code is one step ahead

**Snesh Prajapati** 29-Sep-16 7:27

Re: Your code is one step ahead

**Vaso Elias** 3-Oct-16 4:55**Waiting for Factory Method Pattern and Abstract Factory Pattern** **DeyChandan** 27-Sep-16 23:52

Re: Waiting for Factory Method Pattern and Abstract Factory Pattern

**Snesh Prajapati** 28-Sep-16 6:58

Re: Waiting for Factory Method Pattern and Abstract Factory Pattern

**Snesh Prajapati** 2-Oct-16 16:12

Re: Waiting for Factory Method Pattern and Abstract Factory Pattern 

**DeyChandan** 2-Oct-16 21:41

Re: Waiting for Factory Method Pattern and Abstract Factory Pattern 

**Snesh Prajapati** 2-Oct-16 22:38

Re: Waiting for Factory Method Pattern and Abstract Factory Pattern 

**Snesh Prajapati** 9-Oct-16 18:54

**Questionable** 

**Member 12692295** 26-Sep-16 11:28

Re: Questionable 

**Snesh Prajapati** 26-Sep-16 16:49

**My vote of 5** 

**AlessandroF73** 26-Sep-16 5:23

Re: My vote of 5 

**Snesh Prajapati** 26-Sep-16 6:56

**My vote of 5** 









**TinTinTiTin** 25-Sep-16 21:15

Re: My vote of 5 

**Snesh Prajapati** 25-Sep-16 21:42

Refresh

1

 General  News  Suggestion  Question  Bug  Answer  Joke  Praise  Rant  Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#)

[Advertise](#)

[Privacy](#)

[Cookies](#)

[Terms of Use](#)

Layout: [fixed](#) | [fluid](#)

Article Copyright 2016 by Snesh Prajapati  
Everything else Copyright © [CodeProject](#), 1999-2021

Web04 2.8.20210930.1