**Introduction**

Factories are extremely popular among design patterns. I have never seen any reliable statistics on the usage of patterns but factories must be among the top three most used patterns. However, that is not to say that they are used correctly. Many developers misunderstand factories to factor out chunks of code to other classes where the factored-out code is encapsulated into a static method as follows:

|  |  |
| --- | --- |
| 1 | double cost = CostFactory.Calculate(input params); |

That is definitely NOT a factory. That rather resembles some kind of service, except that methods in services are usually not static. However, you may come across such false implementations of factories – which is probably the case of other pattern types as well.

Factories build objects. They often do this using some parameters that help them decide what kind of concrete object to build but that’s not a requirement. The return type of a factory is often some kind of abstraction, i.e. an interface or an abstract class and the factory builds a concrete implementation of the abstraction.

Why would you need such a factory? You may not know in advance which concrete type a certain class is going to use. Example: when a visitor to your route mapping web application can choose different strategies to calculate the route between two cities – fastest, cheapest, most scenic etc. – then how can you know in advance which implementation of the IRouteCalculation strategy to inject in the calculation service? (More about strategies [here](https://dotnetcodr.com/2013/04/29/design-patterns-and-practices-in-net-the-strategy-pattern/)). This is definitely a task for a factory – evaluate the client inputs and then select the appropriate strategy.

Another area where factories can help is when the creation logic of an object is so complicated that it should not be encapsulated within its constructor. .NET has several examples of that, e.g.:

|  |  |
| --- | --- |
| 1 | Guid guid = Guid.NewGuid(); |

Here we don’t have access to the setters of the Guid object and probably with a good reason. The implementation details of creating a new Guid are encapsulated in a factory method instead of the caller trying to guess the correct values of each property setter.

**Demo**

Start Visual Studio and create a new Console Application. We’ll simulate a simple application that starts and stops machines. Insert the following interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface IMachine  {          string Name { get; }      void TurnOn();      void TurnOff();  } |

We’ll now create some concrete machine objects:

Robot.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Robot : IMachine      {          public string Name          {              get { return "robot"; }          }            public void TurnOn()          {              Console.WriteLine("Robot is starting.");          }            public void TurnOff()          {              Console.WriteLine("Robot is stopping.");          }       } |

Car.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Car : IMachine      {          public string Name          {              get { return "car"; }          }            public void TurnOn()          {              Console.WriteLine("Car is starting.");          }            public void TurnOff()          {              Console.WriteLine("Car is stopping.");          }      } |

MicrowaveOven.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class MicrowaveOven : IMachine      {          public string Name          {              get { return "microwave oven"; }          }            public void TurnOn()          {              Console.WriteLine("Microwave oven is starting.");          }            public void TurnOff()          {              Console.WriteLine("Microwave oven is stopping.");          }      } |

UnknownMachine.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class UnknownMachine : IMachine      {          public string Name          {              get { return string.Empty; }          }            public void TurnOn()          {            }            public void TurnOff()          {            }      } |

UnknownMachine.cs as you see performs nothing and has no name – this is the Null Object pattern implementation of the interface. It is used instead of a null value when no suitable machine implementation has been found by the factory. I’ll write a post on that pattern as well later on.

The Main method if Program.cs looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | static void Main(string[] args)          {              string description = args[0];              IMachine machine = GetMachine(description);              machine.TurnOn();              machine.TurnOff();                Console.ReadKey();          }            private static IMachine GetMachine(string description)          {              switch (description)              {                  case "robot":                      return new Robot();                  case "car":                      return new Car();                  default:                      return new UnknownMachine();              }          } |

This is not terribly complicated I hope. We don’t know in advance which machine the user wants to start and stop so we let a private static method take care of that. Note that we haven’t yet included the microwave oven as an option. Imagine that our machine palette now includes the microwave oven. In case we want to ensure that the client can access this new machine as well we have to extend the switch statement as follows:

|  |  |
| --- | --- |
| 1  2 | case "oven":      return new MicrowaveOven(); |

This looks like a small price to pay but we violated letter ‘O’ in SOLID, i.e. the Open/Closed principle: a class is open for extension but closed for modification. Also, Program.cs must be aware of the different IMachine implementations. It is Program.cs that is made responsible for finding the correct concrete type which is not the correct approach. Ideally Program.cs should only be concerned with the IMachine interface, nothing else. Last, but not least every time we add a new IMachine implementation to our app we have to return to Program.cs and extend the GetMachine method.

With the help of the factory pattern we would like to:

* Separate out the object creation logic, i.e. relieve Program.cs of that task
* Add new implementations without breaking the Open/Closed principle
* Externalise object creation rules to a database or a configuration file: a classic example is the type of Membership object to use as defined in web.config or app.config. That is also an application of the factory pattern