6. Not knowing what object implementations you'll need ahead of time

Imagine that you have a code that uses a particular object type. The code that uses this object only needs to know the signatures of accessible methods and properties of the object. It doesn't care about the implementation details. Therefore, using an interface that represents the object rather than a concrete implementation of the actual object is desirable.

If you know what concrete implementation your application will be using ahead of time, you can simply apply the dependency inversion principle along with some simple dependency injection. In this case, you will know that it's always a particular implementation of the interface that will be used throughout your working code and you will only replace it with mocked objects for unit testing. But what if the concrete implementation of the interface needs to be chosen dynamically based on runtime conditions? Or what if the shape of the object needs to be defined based on input parameters in the actual place where the object is about to be used? In either of these cases, the dependency injection approach wouldn't work.

Let's look at an example of this. Imagine that you are building an application that can play audio. The application should be able to work on both Windows and Linux. And inside it, you have an interface called `IPlayer` that has standard methods that an audio player would have, such as `Play`, `Pause`, and `Stop`. It is the implementation of this interface that actually interacts with the operating system and plays the audio.

The problem is that Windows and Linux have completely different

audio architectures; therefore you cannot just have a single concrete implementation of the `IPlayer` interface that would work on both operating systems. And you won't know ahead of time what operating system your application will run on.

Suitable design patterns

Factory Method

Factory Method is a method that returns a type of object that implements a particular interface. This method belongs to a Creator object that manages the lifecycle of the object that is to be returned.

Normally, you would have several variations of the Creator object, each returning a specific implementation of the object that you want. You would instantiate a specific variation of the Creator object based on a specific condition and would then call its Factory Method to obtain the implementation of the actual object.

In our example above, you would have one version of the Creator object that returns a Windows implementation of `IPlayer` and another version that returns its Linux implementation. The Creator object will also be responsible for initializing all dependencies that your `IPlayer` implementation needs. Some code blocks in your application will check which operating system it runs on and will initialize the corresponding version of the Creator object.

Why would you want to use Factory Method

Why bother using Factory Method, if you can just initialize any objects directly? Well, here is why:

- Good use of the single responsibility principle. The Creator object is solely responsible for creating only one specific implementation type of the end object and nothing else.

- The pattern has been prescribed in such a way that it makes it easy to extend the functionality of the output object and not violate the open-closed principle.
- Easy to write unit tests, as Creational logic will be separated from the conditional logic.

Abstract Factory

Abstract Factory is a design pattern that uses multiple Factory Methods inside of the Creator object, so you can create a whole family of related objects based on a particular condition instead of just one object.

Let's change the above example slightly. Imagine that our app needs to be able to either play audio or video, so you will need to implement two separate interfaces – `IAudioPlayer` and `IVideoPlayer`. Once again, it must work on either Linux or Windows.

In this case, your Abstract Factory will have a separate method to return an implementation of `IAudioPlayer` and a separate method to return an implementation of `IVideoPlayer`. You will have a version of the Factory that is specific to Windows and another version that is specific to Linux.

It's known as Abstract Factory because it either implements an interface or extends an abstract class. It is then up to the concrete implementation of the Factory to create concrete implementations of the output objects that are both relevant to a specific condition.

Builder

Builder design pattern is similar to the Factory Method, but instead of just returning a concrete implementation of an object all at once, it builds the object step-by-step.

Let's go back to our OS-independent app that plays audio. In the Factory Method example, we had a concrete implementation of the `IPlayer` interface for each operating system. However, if we would choose to use Builder instead of a Factory Method, we would have a single concrete implementation of the interface that would act as a shell object. Let's call it `Player`. It will be this type that gets produced in all scenarios, but the concrete parameters and dependencies will be injected into it based on what kind of operating system it's running on.

For example, both Linux and Windows allow you to play audio and manipulate its volume via the command line. On Linux, it will be Bash Terminal. On Windows, it will be either cmd or PowerShell.

The principles are similar. In both cases, you would be typing commands. But the exact commands will be completely different. Plus there are likely to be separate commands for playing audio files and for manipulating audio volumes.

So, in this case, our `Player` class will simply delegate the playback of audio to operating system components that are accessible via a command line interface. It's only the actual commands that will be different. And this is where a Builder design pattern comes into play.

Builder consists of two main components – Builder and Director. Builder is a class that returns a specific object type. But it also has a number of methods that modify this object type before it gets returned. The director is a class that has a number of methods, each of which accepts a Builder class, calls methods on it with a specific set of parameters, and gets the Builder to return the finished object.

So, in our case, imagine that our Builder class for the `Player` object (which we will call `PlayerBuilder`) has the following methods: `SetPlaybackInterface` (that accepts a playback interface instance), `SetAudioVolumeInterface`(that accepts audio volume interface instance) and `BuildPlayer` (that doesn't accept any parameters). When we instantiate our `PlayerBuilder` class, we instantiate a

private instance of the `Player` class inside of it. Then, by executing `SetPlaybackInterface` and `SetAudioVolumeInterface` methods, we dynamically add the required dependencies to the instance of our `Player` class. And finally, by executing the `BuildPlayer` method, we are returning a complete instance of a `Player` object.

In this case, our Director class will have two methods, both of which accept `PlayerBuilder` as the parameter: `BuildWindowsPlayer` and `BuildLinuxPlayer`. Both of these methods will call all the methods on the `PlayerBuilder` class in the same order and both will return an instance of a `Player` class. But in the first case, the methods will be called with Windows-specific abstractions of the command line interface, it's the Linux-specific abstractions that would be injected into the `Player` instance.

However, unlike either Abstract Factory or Factory Method, Builder is not only used to build an object the implementation details of which can only be made known at runtime. It is also used to gradually build complex objects.

For example, .NET has an inbuilt `StringBuilder` class in its core `System` library. This class is used for building a string from several sub-strings, so you don't have to just keep replacing an immutable string in the same variable.

Why would you want to use Builder

- Good use of the single responsibility principle. Each Builder method has its own very specific role and there is a single method on the Director class per each condition.
- Easy to write unit tests. This is facilitated by the single responsibility principle.
- No need to have different versions of a class if only some of its implementation details may change in different circumstances.

Because an object can be built step-by-step, it is the best design

pattern to decide what the object will be if you have to adjust its properties one by one by multiple steps of conditional logic.

You can reuse the same code for different representations of the final object.