Abstract Factory Pattern

This lesson details the working of yet another creational design pattern, that allows us to create a family of products in a flexible and extensible manner.

What is it ?

In the previous lesson, we learned the factory method pattern. We saw how we were able to model the variants of the F-16 using the factory method. But there are numerous airplanes other than F-16 that we'll need to represent. Say the client buys a Boeing-747 for the CEO to travel and now wants your software to provide support for this new type of aircraft.

The abstract factory pattern solves the problem of creating *families of related products*. For instance, F-16 needs an engine, a cockpit, and wings. The Boeing-747 would require the same set of parts but they would be specific to Boeing. Any airplane would require these three *related* parts but the parts will be plane and vendor specific. Can you see a pattern emerge here? We need a framework for creating the related parts for each airplane, a family of parts for the F-16, a family of parts for the Boeing-747 so on and so forth.

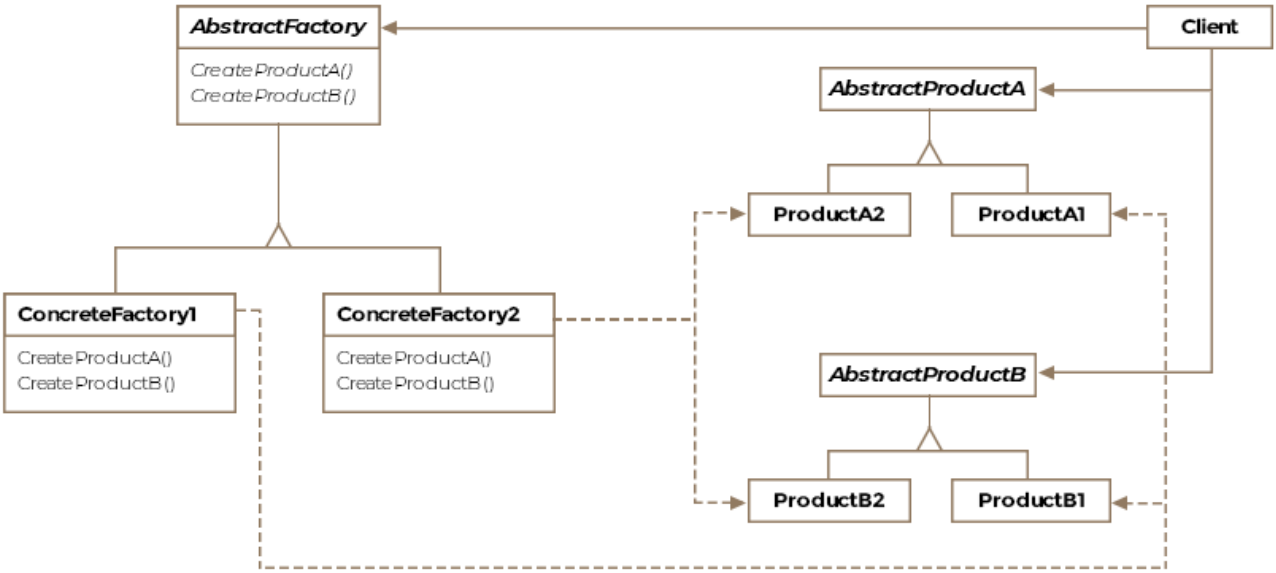
Formally, the abstract factory pattern is defined as ***defining an interface to create families of related or dependent objects without specifying their concrete classes***.

Class Diagram

The class diagram consists of the following entities

- **Abstract Factory**
- **Concrete Factory**

- **Concrete Factory**
- **Abstract Product**
- **Concrete Product**
- **Client**



Class Diagram

Example

An abstract factory can be thought of as a super factor or a factory of factories. The pattern achieves the creation of a family of products without revealing concrete classes to the client. Let's consider an example. Say, you are creating a simulation software for the aviation industry and need to represent different aircraft as objects. But before you represent an aircraft, you also need to represent the different pieces of an aircraft as objects. For now let's pick three: the cockpit, the wings, and the engine. Now say the first aircraft you want to represent is the mighty F-16. You'll probably write three classes one for each piece specific to the F-16. In your code you'll likely consume the just created three classes as follows:

```
    public void main() {
        F16Cockpit f16Cockpit = new F16Cockpit();
        F16Engine f16Engine = new F16Engine();
```

```
        F16Engine f16Engine = new F16Engine();
        F16Wings f16Wings = new F16Wings();

        List<F16Engine> engines = new ArrayList<>();
        engines.add(f16Engine);
        for (F16Engine engine : engines) {
            engine.start();
        }
    }
```

This innocuous looking snippet can cause severe headaches down the road if your simulation software takes off and you need to expand it to other aircraft. Below is a list of what is wrong with the above snippet:

- The concrete classes for the three parts have been directly exposed to the consumer.
- F-16 has several variants with different engines and say in future if you want to return engine object matching to the variant, you'll need to subclass `F16Engine` class and that would necessitate a change in the consumer snippet too.
- The List in the code snippet is parametrized with a concrete class, in future if you add another aircraft engine object then the new engine can't be added to the list, even though engines for all aircraft are somewhat similar.

We'll fix these issues one by one and see how the abstract factory pattern would emerge.

Code to an interface not to an implementation

One of the fundamental principles of good object orientated design is to hide the concrete classes and expose interfaces to clients. An object responds to a set of requests, these requests can be captured by an interface which the object's class implements. The client should know what requests an object responds to rather than the implementation.

In our example, we can create an interface `IEngine` which exposes the

In our example, we can create an interface `IEngine`, which exposes the method `start()`. The F16Engine class would then change like so:

```
public interface IEngine {

    void start();
}

public class F16Engine implements IEngine {

    @Override
    public void start() {
        System.out.println("F16 engine on");
    }
}
```

With the above change see how the corresponding consumer code changes

```
    public void main() {
        IEngine f16Engine = new F16Engine();
        List<IEngine> engines = new ArrayList<>();
        engines.add(f16Engine);
        for (IEngine engine : engines) {
            engine.start();
        }
    }
```

Suddenly the consumer code is free of the implementation details of what class implements the F-16 engine and works with an interface. However, we would still like to hide the `new F16Engine()` part of the code. We don't want the consumer to know what class we are instantiating. This is discussed next.

Creating a factory

Instead of new-ing up objects in client code, we'll have a class responsible for manufacturing the requested objects and returning them to the client. We'll call this class `F16Factory` since it can create the various parts of the F16 aircraft and deliver them to the requesting client. The class would

take the following shape.

```
public class F16Factory {

    public IEngine createEngine() {
        return new F16Engine();
    }

    public IWings createWings() {
        return new F16Wings();
    }

    public ICockpit createCockpit() {
        return new F16Cockpit();
    }
}
```

Suppose we pass in the `F16Factory` object in the constructor to the client code and it would now create objects like so:

```
    public void main(F16Factory f16Factory) {
        IEngine f16Engine = f16Factory.createEngine();
        List<IEngine> engines = new ArrayList<>();
        engines.add(f16Engine);
        for (IEngine engine : engines) {
            engine.start();
        }
    }
```

Note how this setup allows us the freedom to change the concrete class representing the F16Engine as long as it commits to the `IEngine` interface. We can rename, enhance or modify our class without causing a breaking change in the client. Also note that by just differing the factory class passed into the client constructor, we are able to provide the client with the same parts for a completely new aircraft. This is discussed next.

Factory of Factories

Wouldn't it be great if we could use the same client snippet for other aircraft such as Boeing747 or a Russian MiG-29? If we could have all the

factories being passed into the client agree to implement the

`createEngine()` method, then the client code will keep working for all kinds of aircraft factories. But all the factories must commit to a common interface whose methods they'll implement and this common interface will be the **abstract factory**.

Implementation

Let's start with an interface that would define the methods the factories for different aircraft would need to implement. The client code is written against the abstract factory but composed at runtime with a concrete factory.

```
public interface IAircraftFactory {

    IEngine createEngine();

    IWings createWings();

    ICockpit createCockpit();
}
```

Note that we mean a Java abstract class or a Java interface when referring to "interface". In this instance, we could have used an abstract class if there were a default implementation for any of the products. The create methods don't return concrete products rather interfaces to decouple the factory consumers from the concrete implementation of parts.

The formal definition of the abstract factory pattern says abstract factory pattern defines an interface for creating families of related products without specifying the concrete classes. Here the `IAircraftFactory` is *that* interface in the formal definition and note how its create methods are not returning concrete parts but rather interfaces that'll be implemented by the concrete parts' classes.

Next let's define our factories for the two aircraft.

```
public class F16Factory implements IAircraftFactory {

    @Override
    public IEngine createEngine() {
        return new F16Engine();
    }

    @Override
    public IWings createWings() {
        return new F16Wings();
    }

    @Override
    public ICockpit createCockpit() {
        return new F16Cockpit();
    }
}

public class Boeing747Factory implements IAircraftFactory {

    @Override
    public IEngine createEngine() {
        return new Boeing747Engine();
    }

    @Override
    public IWings createWings() {
        return new Boeing747Wings();
    }

    @Override
    public ICockpit createCockpit() {
        return new Boeing747Cockpit();
    }
}
```

The concrete factories will be responsible for creating F-16 or Boeing specific engine, cockpit and wings. Each part has a corresponding product interface that we don't list for brevity's sake. The interfaces representing the parts would be:

- `IEngine`

- `ICockpit`
- `IWings`

All the **create** methods are actually factory methods that have been overridden. Indeed, the factory method pattern is utilized when implementing the abstract factory pattern. For the sake of brevity, we have skipped listing the concrete classes for engine, wings, and cockpit.

In the previous lesson, we created a class for F-16 which included a public method `fly()`. This method internally invoked the `makeF16()` method and after the aircraft was manufactured, it invoked the `taxi()` method before printing a fly statement. In our scenario, all aircrafts are expected to follow the same pattern. They first get manufactured, then taxi on the runway and then fly away. We can thus create a class for an aircraft that does these three tasks. Note, how we aren't creating separate classes to represent the two aircraft i.e. the F-16 and Boeing-747 rather a single `Aircraft` class that can represent both.

```
// Incomplete skeleton of the class.
public class Aircraft {

    IEngine engine;
    ICockpit cockpit;
    IWings wings;

    protected Aircraft makeAircraft() {
        //TODO: provide implementation
    }

    private void taxi() {
        System.out.println("Taxing on runway");
    }

    public void fly() {
        Aircraft aircraft = makeAircraft();
        aircraft.taxi();
        System.out.println("Flying");
    }
```

```
}
```

For now we'll keep the `makeAircraft` method empty. Let's first see how a client will request F-16 and Boeing-747 objects.

```
public class Client {

    public void main() {

        // Instantiate a concrete factory for F-16
        F16Factory f16Factory = new F16Factory();

        // Instantiate a concrete factory for Boeing-747
        Boeing747Factory boeing747Factory = new Boeing747Factory();

        // Lets create a list of all our airplanes
        Collection<Aircraft> myPlanes = new ArrayList<>();

        // Create a new F-16 by passing in the f16 factory
        myPlanes.add(new Aircraft(f16Factory));

        // Create a new Boeing-747 by passing in the boeing factory
        myPlanes.add(new Aircraft(boeing747Factory));

        // Fly all your planes
        for (Aircraft aircraft : myPlanes) {
            aircraft.fly();
        }

    }
}
```

We'll need to add a constructor to our `Aircraft` class, which will store the passed-in factory object and create the aircraft parts using the factory. **Just by composing the aircraft object with a different factory we are able to get a different aircraft**. The complete version of the aircraft class would look like below:

```
public class Aircraft {

    IEngine engine;
    ICockpit cockpit;
    IWings wings;
    IAircraftFactory factory;
```

```
    public Aircraft(IAircraftFactory factory) {
        this.factory = factory;
    }

    protected Aircraft makeAircraft() {
        engine = factory.createEngine();
        cockpit = factory.createCockpit();
        wings = factory.createWings();
        return this;
    }

    private void taxi() {
        System.out.println("Taxing on runway");
    }

    public void fly() {
        Aircraft aircraft = makeAircraft();
        aircraft.taxi();
        System.out.println("Flying");
    }
}
```

The client just needs to instantiate the right factory and pass it in. The consumer or client of the factory is the `Aircraft` class. We could have created an interface `IAircraft` to represent all the aircraft that the class `Aircraft` in turn would implement but for our limited example it's not necessary.

The resulting code is easily extensible and flexible.

To tie our current example with the example discussed in the factory method pattern lesson, we have the option of either subclassing the `F16Factory` further to create factories for the **A** and **B** variants of F-16. We could also parametrize the existing `F16Factory` factory to take in an enum specifying the variant model and accordingly return the right part in a switch statement.

Other Examples

- The abstract factory is particularly useful for frameworks and toolkits that work on different operating systems. For instance, if your library provides fancy widgets for the UI, then you may need a family of products that work on MacOS and a similar family of products that work on Windows. Similarly, themes used in IDE can be another example. If your IDE supports light and dark themes then it may use the abstract factory pattern to create widgets that belong to the light or dark theme just by varying the concrete factory that creates the widgets.
- `javax.xml.parsers.DocumentBuilderFactory.newInstance()` will return you a factory.
- `javax.xml.transform.TransformerFactory.newInstance()` will return you a factory.

Caveats

- It might appear to the naive reader that the factory method pattern and the abstract factory pattern are similar. The difference between the two lies in their motivations. The factory method pattern is usually responsible for creating a single product whereas an abstract factory pattern creates entire families of related products. Furthermore, in the factory method pattern, we use inheritance to create more specialized products whereas, in an abstract factory pattern, we practice object composition by passing in factories that are consumed to create the desired products.
- In our aircraft example, we can add a new aircraft simply by creating a concrete factory for it. However, note that if a helicopter is added

to the fleet and requires a part that an aircraft doesn't have, then we'll need to extend the `IAircraftFactory` interface with another create method for the part required only by the helicopter. This will cascade the change to existing factories that'll need to return null since the new component isn't part of the jets.

- Concrete factories can be best represented as a singleton object