

Abstract Factory Pattern

Michal Moravik, SD20w2

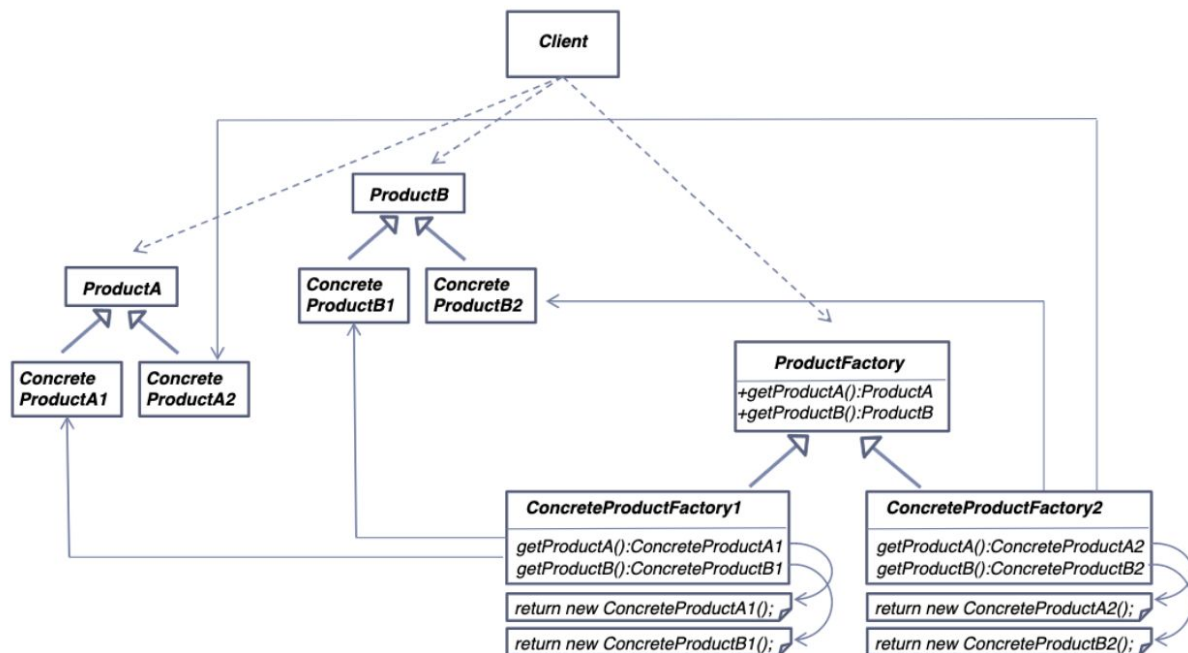
Name: Abstract Factory Pattern

Category: Creational pattern

Intent: Allows the engineer to control the creation of different factories. An abstract factory has a collection of different factory methods to create different desired concrete objects. It is basically used to instantiate different factories of objects based on condition, so it is one level up from the Factory Method Pattern.

Motivation: We want to control families (factories) of classes chosen at the runtime so we can switch between them based on the current situation. We can depend on even higher abstraction than with the factory method design pattern. We don't need to know which factory will be used during the runtime because we depend on the abstraction (thus the name "abstract factory pattern").

UML:



Implementation:

```
public interface ProductA {  
    String getType();  
}
```

```
public interface ProductB {  
    String getType();  
}
```

For product factory 1

```
public class ConcreteProductA1 implements ProductA {  
    @Override  
    public String getType() {  
        return "Hi I am product A1";  
    }  
}
```

```
public class ConcreteProductB1 implements ProductB {  
    @Override  
    public String getType() {  
        return "Hi I am product B1";  
    }  
}
```

For product factory 2

```
public class ConcreteProductA2 implements ProductA {  
    @Override  
    public String getType() { return "Hi I am product A2"; }  
}
```

```
public class ConcreteProductB2 implements ProductB {  
    @Override  
    public String getType() {  
        return "Hi I am product B2";  
    }  
}
```

Factories 1 and 2

```
public class ConcreteProductFactory1 implements AbstractProductFactory {  
    public ProductA getProductA() {  
        return new ConcreteProductA1();  
    }  
    public ProductB getProductB() {  
        return new ConcreteProductB1();  
    }  
}
```

```
public class ConcreteProductFactory2 implements AbstractProductFactory {  
    public ProductA getProductA() {  
        return new ConcreteProductA2();  
    }  
    public ProductB getProductB() {  
        return new ConcreteProductB2();  
    }  
}
```

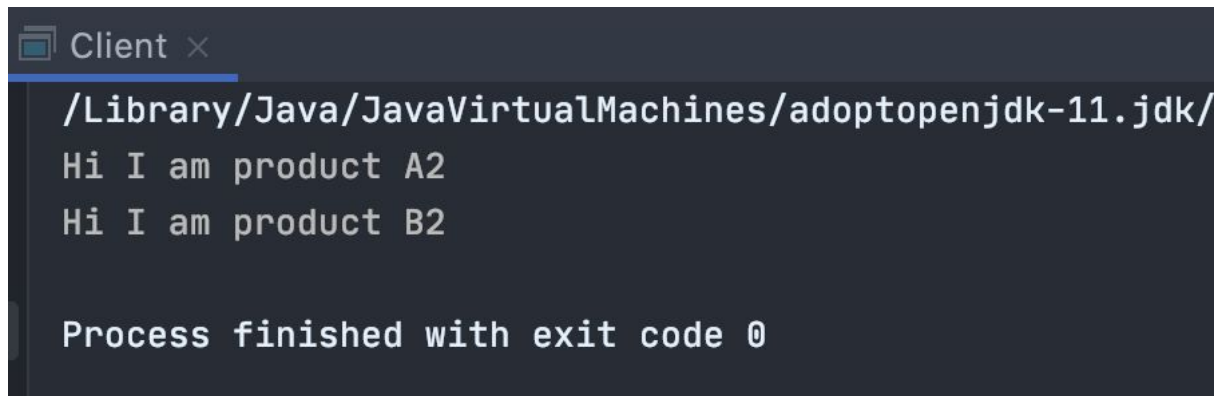
Their abstraction (this is why it is called abstract factory pattern)

```
public interface AbstractProductFactory {  
    ProductA getProductA();  
    ProductB getProductB();  
}
```

Client implementation - it is dependent only on abstraction, **we don't know which factory will be used. It will be known during the runtime.** But we can still call methods from products A and B no matter which factory will be used.

```
public class Client {  
    public static void main(String[] args) {  
        // factoryNumber can vary in different situations  
        int factoryNumber = 2;  
        AbstractProductFactory productFactory;  
  
        if (factoryNumber == 1) {  
            productFactory = new ConcreteProductFactory1();  
        } else {  
            productFactory = new ConcreteProductFactory2();  
        }  
  
        System.out.println(productFactory.getProductA().getType());  
        System.out.println(productFactory.getProductB().getType());  
    }  
}
```

The output:



```
Client x  
/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/  
Hi I am product A2  
Hi I am product B2  
  
Process finished with exit code 0
```

Consequences:

Pros:

- Creates an ultimate abstraction dependent system which means we don't need to know which factory will be used until the runtime and it will still work
- Exchanging factories (and therefore objects) is very easy
- Will bring consistency to our objects (products) and factories since they are all dependent on abstraction

Cons:

- Things can get very complex and complicated and potentially create a very hard-to-understand system
- A long process in order to create a new abstract factory (of factories)

Known uses:

- Complex systems when we don't know the families of objects in the upfront. Let's say we have two systems, Windows and Mac. Both of them can have the same Google Chrome browser installed. When we want to open a filesystem to upload a file, each of them will do it differently although both have filesystems (but different ones). Both

filesystems have then different components inside (like buttons, listViews, ...). We could use those buttons and listViews no matter which system (Windows / Mac) will be instantiated.

Related patterns:

- Factory Method Pattern