

# Abstract Factory Pattern

Abstract Factory offers the interface for creating a family of related objects  
Intent

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

Implementation

The pattern basically works as shown below, in the UML diagram:

The classes that participate to the Abstract Factory pattern are:

- AbstractFactory - declares a interface for operations that create abstract products.
  - ConcreteFactory - implements operations to create concrete products.
  - AbstractProduct - declares an interface for a type of product objects.
  - Product - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
  - Client - uses the interfaces declared by the AbstractFactory and AbstractProduct classes, without explicitly specifying their classes.
- Abstract Factory

Motivation

Modularization is a big issue in today's programming. Programmers all over the world are trying to avoid the idea of adding code to existing classes in order to make them support encapsulating more general information. Take the case of a information manager which manages phone number. Phone numbers have a particular rule on which they get generated depending on areas and countries. If at some point the application should be changed in order to support adding numbers from a new country, the code of the application would have to be changed and it would become more and more complicated.

In order to prevent it, the Abstract Factory design pattern is used. Using this pattern a framework is defined, which produces objects that follow a general pattern and at runtime this factory is paired with any concrete factory to produce objects that follow the pattern of a certain country. In other words, the Abstract Factory is a super-factory which creates other factories (Factory of factories).

Intent

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

## Implementation

The pattern basically works as shown below, in the UML diagram:

The classes that participate to the Abstract Factory pattern are:

- AbstractFactory - declares a interface for operations that create abstract products.
- ConcreteFactory - implements operations to create concrete products.
- AbstractProduct - declares an interface for a type of product objects.
- Product - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
- Client - uses the interfaces declared by the AbstractFactory and AbstractProduct classes.

The AbstractFactory class is the one that determines the actual type of the concrete object and creates it, but it returns an abstract pointer to the concrete object just created. This determines the behavior of the client that asks the factory to create an object of a certain abstract type and to return the abstract pointer to it, keeping the client from knowing anything about the actual creation of the object.

The fact that the factory returns an abstract pointer to the created object means that the client doesn't have knowledge of the object's type. This implies that there is no need for including any class declarations relating to the concrete type, the client dealing at all times with the abstract type. The objects of the concrete type, created by the factory, are accessed by the client only through the abstract interface.

The second implication of this way of creating objects is that when the adding new concrete types is needed, all we have to do is modify the client code and make it use a different factory, which is far easier than instantiating a new type, which requires changing the code wherever a new object is created.

The classic implementation for the Abstract Factory pattern is the following:

```
abstract class AbstractProductA{
    public abstract void operationA1();
    public abstract void operationA2();
}

class ProductA1 extends AbstractProductA{
    ProductA1(String arg){
        System.out.println("Hello "+arg);
    } // Implement the code here
    public void operationA1() { };
    public void operationA2() { };
}

class ProductA2 extends AbstractProductA{
```

```
ProductA2(String arg){
    System.out.println("Hello "+arg);
} // Implement the code here
public void operationA1() { };
public void operationA2() { };
}

abstract class AbstractProductB{
    //public abstract void operationB1();
    //public abstract void operationB2();
}

class ProductB1 extends AbstractProductB{
    ProductB1(String arg){
        System.out.println("Hello "+arg);
    } // Implement the code here
}

class ProductB2 extends AbstractProductB{
    ProductB2(String arg){
        System.out.println("Hello "+arg);
    } // Implement the code here
}

abstract class AbstractFactory{
    abstract AbstractProductA createProductA();
    abstract AbstractProductB createProductB();
}

class ConcreteFactory1 extends AbstractFactory{
    AbstractProductA createProductA(){
        return new ProductA1("ProductA1");
    }
    AbstractProductB createProductB(){
        return new ProductB1("ProductB1");
    }
}

class ConcreteFactory2 extends AbstractFactory{
    AbstractProductA createProductA(){
        return new ProductA2("ProductA2");
    }
    AbstractProductB createProductB(){
        return new ProductB2("ProductB2");
    }
}

//Factory creator - an indirect way of instantiating the factories
class FactoryMaker{
    private static AbstractFactory pf=null;
    static AbstractFactory getFactory(String choice){
        if(choice.equals("a")){
            pf=new ConcreteFactory1();
        }else if(choice.equals("b")){
            pf=new ConcreteFactory2();
        } return pf;
    }
}

// Client
public class Client{
    public static void main(String args[]){
        AbstractFactory pf=FactoryMaker.getFactory("a");
        AbstractProductA product=pf.createProductA();
        //more function calls on product
    }
}
```

```
}  
}
```

## Applicability & Examples

We should use the Abstract Factory design pattern when:

- the system needs to be independent from the way the products it works with are created.
- the system is or should be configured to work with multiple families of products.
- a family of products is designed to work only all together.
- the creation of a library of products is needed, for which is relevant only the interface, not the implementation, too.

## Phone Number Example

The example at the beginning of the article can be extended to addresses, too. The AbstractFactory class will contain methods for creating a new entry in the information manager for a phone number and for an address, methods that produce the abstract products Address and PhoneNumber, which belong to AbstractProduct classes. The AbstractProduct classes will define methods that these products support: for the address get and set methods for the street, city, region and postal code members and for the phone number get and set methods for the number.

The ConcreteFactory and ConcreteProduct classes will implement the interfaces defined above and will appear in our example in the form of the USAddressFactory class and the USAddress and USPhoneNumber classes. For each new country that needs to be added to the application, a new set of concrete-type classes will be added. This way we can have the EnglandAddressFactory and the EnglandAddress and EnglandPhoneNumber that are files for English address information.

## Pizza Factory Example

Another example, this time more simple and easier to understand, is the one of a pizza factory, which defines method names and returns types to make different kinds of pizza.

The abstract factory can be named AbstractPizzaFactory, RomeConcretePizzaFactory and MilanConcretePizzaFactory being two extensions of the abstract class.

The abstract factory will define types of toppings for pizza, like pepperoni, sausage or anchovy, and the concrete factories will implement only a set of the toppings, which are specific for the area and even if one topping is implemented in both concrete factories, the resulting pizzas will be different subclasses, each for the area it was implemented in.

## Look & Feel Example

Look & Feel Abstract Factory is the most common example. For example, a GUI framework should support several look and feel themes, such as Motif and Windows look. Each style defines different looks and behaviors for each type of controls: Buttons and Edit Boxes. In order to avoid the hardcoding it for each type of control we define an abstract class LookAndFeel. This calls will instantiate, depending on a configuration parameter in the application one of the concrete factories: WindowsLookAndFeel or MotifLookAndFeel. Each request for a new object will be delegated to the instantiated

concrete factory which will return the controls with the specific flavor

### Specific problems and implementation

The Abstract Factory pattern has both benefits and flaws. On one hand it isolates the creation of objects from the client that needs them, giving the client only the possibility of accessing them through an interface, which makes the manipulation easier. The exchanging of product families is easier, as the class of a concrete factory appears in the code only where it is instantiated. Also if the products of a family are meant to work together, the Abstract Factory makes it easy to use the objects from only one family at a time. On the other hand, adding new products to the existing factories is difficult, because the Abstract Factory interface uses a fixed set of products that can be created. That is why adding a new product would mean extending the factory interface, which involves changes in the AbstractFactory class and all its subclasses. This section will discuss ways of implementing the pattern in order to avoid the problems that may appear.

### Factories as singletons

An application usually needs only one instance of the ConcreteFactory class per family product. This means that it is best to implement it as a Singleton.

### Creating the products

The AbstractFactory class only declares the interface for creating the products. It is the task of the ConcreteProduct class to actually create the products. For each family the best idea is applying the Factory Method design pattern. A concrete factory will specify its products by overriding the factory method for each of them. Even if the implementation might seem simple, using this idea will mean defining a new concrete factory subclass for each product family, even if the classes are similar in most aspects.

For simplifying the code and increase the performance the Prototype design pattern can be used instead of Factory Method, especially when there are many product families. In this case the concrete factory is initiated with a prototypical instance of each product in the family and when a new one is needed instead of creating it, the existing prototype is cloned. This approach eliminates the need for a new concrete factory for each new family of products.

### Extending the factories

The operation of changing a factory in order for it to support the creation of new products is not easy. What can be done to solve this problem is, instead of a CreateProduct method for each product, to use a single Create method that takes a parameter that identifies the type of product needed. This approach is more flexible, but less secure. The problem is that all the objects returned by the Create method will have the same interface, that is the one corresponding to the type returned by the Create method and the client will not always be able to correctly detect to which class the instance actually belongs.

### Hot Points:

- AbstractFactory class declares only an interface for creating the products. The actual creation is the task of the ConcreteProduct classes, where a good approach is applying the Factory Method design pattern for each product of the family.

- Extending factories can be done by using one Create method for all products and attaching information about the type of product needed.