

HOME »» DESIGN PATTERN »» DESIGN PATTERN: FACTORY PATTERNS

Design Pattern: factory patterns

by Christophe | updated: March 31, 2016 | posted: June 11, 2015

12 Comments

6

3

Factories are one of the key **creational patterns** that each developer should know. They are the main component of many advanced patterns. For a long time I had trouble with the different types of factory patterns. Moreover, it's difficult to find information about these types in the same article. This article is about 4 types of factory patterns:

- the **factory method pattern**,
- the **abstract factory pattern**,
- the **static factory method**,
- the **simple factory** (also called factory).

The factory method pattern was described in the book "*Design Patterns: Elements of Reusable Object-Oriented Software*" by the "**Gang of Four**". The first time I read this pattern, I misinterpreted it with the static one which was described by **Joshua Bloch** – one of the main architects of the Java APIs – in his book "*Effective Java*". The simple factory (sometimes called factory) is informal but appears many times on the net. The last one, the abstract factory pattern, was also described in the book of the "Gang of Four" and is a broader concept of factory method pattern.

In this post, I'll explain why factories are useful then I'll present each type with real examples from famous Java

frameworks or Java APIs. I'll use Java code to implement factories but if you don't know Java you'll still be able to understand the idea. Moreover, I'll use UML to describe the patterns formally.

Contents [hide]

- 1 Anti-pattern
- 2 The need for factories
 - 2.1 Control over instantiation
 - 2.2 Loose coupling
 - 2.3 Encapsulation
 - 2.4 Disambiguation
- 3 The factory patterns
 - 3.1 static factory method
 - 3.1.1 Real examples
 - 3.2 simple factory
 - 3.3 factory method pattern
 - 3.3.1 Real examples
 - 3.4 The abstract factory
 - 3.4.1 Real examples
- 4 Conclusion

Anti-pattern

Though this article is about factory patterns, using patterns just for using patterns is worst than never using them. This behaviour is an anti-pattern. Indeed, most patterns make the code more difficult to understand. Most of the time, I don't use factories. For example:

- When I code alone at home/work, I avoid using them.
- For small projects that won't change a lot I also avoid factories.
- For medium to large projects involving multiple developers using the same code I find them useful.

I always think of factories as a **tradeoff** between their **advantages** (will see them in the next parts) and the **readability** and **comprehension** of the code.

The main objective of factories is to instantiate objects. But why not directly create objets with constructor calls ?


For simple use cases, there is no need to use a factory. Let's look at this code.

```
public class SimpleClass {
    private final Integer arg1;
    private final Integer arg2;

    SimpleClass(Integer arg1, Integer arg2) {
        this.arg1 = arg1;
        this.arg2 = arg2;
    }

    public Integer getArg1(){
        return arg1;
    }

    public Integer getArg2(){
        return args;
    }
}
...
public class BusinessClassXYZ {
    public static void someFunction(){
        SimpleClass mySimpleClass = new SimpleClass(1,
            // some stuff
        );
    }
}
```



In this code, SimpleClass is a very simple class with a state, no dependencies, no polymorphism and no business logic. You could use a factory to create this object but it would double the amount of code. Therefore, it would make the code more difficult to understand. **If you can avoid using factories do it**, you'll end up with simpler a code!

But, you'll often encounter more complex cases when writing large applications that require many developers and many code changes. For these complex cases, the advantages of factories overtake their drawbacks.

The need for factories

Now that I warned you about the use of factories, let's see why they are so powerful and therefore used in most projects.

Control over instantiation

A common use case with enterprise applications is to limit the number of instances of a class. How would you manage to have only one (or 2, or 10) instance of a class because it

consumes a resource like a socket, or a database connection, or a file system descriptor or whatever?


With the constructor approach, it would be difficult for different functions (from different classes) to know if an instance of a class already exists. And, even if there was an instance, how could a function get this instance? You could do that by using shared variables that each function would check but

- it would link the behaviour of all the functions that needs to instantiate the same class since they are using and modifying the same shared variables,
- multiple parts of the code would have same logic to check if the class has already be instantiated which would lead to code duplication (very bad!).

Using a static factory method, you could easily do that:

```
public class Singleton {
    private static final Singleton INSTANCE = new S
    private Singleton(){}

    public static Singleton getInstance(){
        return INSTANCE;
    }
    ...
}
...
public class ClassXXX{
    ...
    public static void someFunctionInClassXXX(){
        Singleton instance = Singleton.getInstance();
        //some stuff
    }
}
...
public class ClassYYY{
    ...
    public static void someFunctionInClassYYY(){
        Singleton instance = Singleton.getInstance();
        //some stuff
    }
}
```



In this code, we're using a factory that limits the number of instance of the class Singleton to one. By limiting the number of objects we're creating a pool of instances and this **Pool Pattern** is based on a factory.

Note: Instead of limiting the number of instances, we could


have modified the way an instance is created (for example by using a prototype pattern instead of creating a new object from scratch each time).

Loose coupling

Another advantage of factories is the **loose coupling**.

Let's assume you write a program that computes stuff and needs to write logs. Since it's a big project, one of your mates codes the class that writes the logs into a filesystem (the class `FileSystemLogger`) while you're coding the business classes. Without factories, you need to instantiate the `FileSystemLogger` with a constructor before using it:

```
public class FileSystemLogger {  
    ...  
    public void writeLog(String s) {  
        //Implementation  
    }  
}  
...  
public void someFunctionInClassXXX(some parameters)  
    FileSystemLogger logger= new FileSystemLogger(som  
    logger.writeLog("This is a log");  
}
```



But what happens if there is a sudden change and you now need to write logs in a database with the implementation `DatabaseLogger`? Without factories, you'll have to modify all the functions using the `FileSystemLogger` class. Since this logger is used everywhere you'll need to modify hundreds of functions/classes whereas **using a factory you could easily switch from one implementation to another** by modifying only the factory:

```
//this is an abstraction of a Logger  
public interface ILogger {  
    public void writeLog(String s);  
}  
  
public class FileSystemLogger implements ILogger {  
    ...  
    public void writeLog(String s) {  
        //Implementation  
    }  
}  
  
public class DatabaseLogger implements ILogger {  
    ...  
    public void writeLog(String s) {  
        //Implementation  
    }  
}
```

```

    }
}

public class FactoryLogger {
    public static ILogger createLogger() {
        //you can choose the logger you want
        // as long as it's an ILogger
        return new FileSystemLogger();
    }
}

//////////some code using the factory
public class SomeClass {
    public void someFunction() {
        //if the logger implementation changes
        //you have nothing to change in this code
        ILogger logger = FactoryLogger.createLogger();
        logger.writeLog("This is a log");
    }
}

```

If you look at this code, you can easily change the logger implementation from `FileSystemLogger` to `DatabaseLogger`. You just have to modify the function `createLogger()` (which is a factory). **This change is invisible for the client (business) code** since the client code use an **interface** of logger (`ILogger`) and the choice of the logger implementation is made by the factory. By doing so, you're creating a loose coupling between the implementation of the logger and the parts of codes that uses the logger.

Encapsulation

Sometimes, using a factory improves the **readability** of your code and **reduces** its **complexity** by **encapsulation**.

Let assume you need to use a business Class `CarComparator` that compares 2 cars. This class needs a `DatabaseConnection` to get the features of millions of cars and a `FileSystemConnection` to get a configuration file that parametrizes the comparison algorithm (for example: adding more weight to the fuel consumption than the maximum speed).

Without a factory you could code something like:

```

public class DatabaseConnection {
    DatabaseConnection(some parameters) {
        // some stuff
    }
    ...
}

```

```

public class FileSystemConnection {
    FileSystemConnection(some parameters) {
        // some stuff
    }
    ...
}

public class CarComparator {
    CarComparator(DatabaseConnection dbConn, FileSystem
        // some stuff
    }

    public int compare(String car1, String car2) {
        // some stuff with objets dbConn and fsConn
    }
}
...
public class CarBusinessXY {
    public void someFunctionInTheCodeThatNeedsToComp
        DatabaseConnection db = new DatabaseConnection
        FileSystemConnection fs = new FileSystemConnec
        CarComparator carComparator = new CarComparato
        carComparator.compare("Ford Mustang", "Ferrari I
    }
}
...
}

public class CarBusinessZY {
    public void someOtherFunctionInTheCodeThatNeedsT
        DatabaseConnection db = new DatabaseConnection
        FileSystemConnection fs = new FileSystemConnec
        CarComparator carComparator = new CarComparato
        carComparator.compare("chevrolet camaro 2015", '
    }
}
...
}

```

This code works but you can see that in order to use the comparison method, you need to instanciate

- a DatabaseConnection,
- a FileSystemConnection,
- then a CarComparator.

If you need to use the comparison in multiple functions, you will have to **duplicate your code** which means if the construction of the CarComparator changes, you will have to modify all the duplicated parts. The use of a factory could **factorize** the code and **hide the complexity** of the construction of the CarComparator class.

```

...
public class Factory {
    public static CarComparator getCarComparator() {
        DatabaseConnection db = new DatabaseConnection
        FileSystemConnection fs = new FileSystemConnec
        CarComparator carComparator = new CarComparato
    }
}

```

```

    }
}
//////////////////////////////////////////some code using the fac
public class CarBusinessXY {
    public void someFunctionInTheCodeThatNeedsToComp
        CarComparator carComparator = Factory.getCarCor
        carComparator.compare("Ford Mustang", "Ferrari I
    }
}
...
}
...
public class CarBusinessZY {
    public void someOtherFunctionInTheCodeThatNeedsT
        CarComparator carComparator = Factory.getCarCor
        carComparator.compare("chevrolet camaro 2015", '
    }
}
...
}

```

If you compare both codes, you can see that using a factory:

- Reduces the number of line of code.
- Avoid code duplication.
- Organise the code: the factory has the **reponsability** to build a CarComparator and the business class just uses it.

The last point is important (in fact, they're all important!) because a it's about **separation of concerns**. A business class shouldn't have to know how to build a complex object it needs to use: the business class needs to focus only on business concerns. Moreover, it also increases the division of work among the developers of the same project:

- One works on the CarComparator and the way it's created.
- Others work on business objects that use the CarComparator.

Disambiguation

Let's assume that you have a class with multiple constructors (with very different behaviors). How can you be sure that you won't use the wrong constructor by mistake?

Let's look at the following code:

```

class Example{
    //constructor one
    public Example(double a, float b) {
        //...
    }
}

```



```

    }
    //constructor two
    public Example(double a) {
        //...
    }
    //constructor three
    public Example(float a, double b) {
        //...
    }
}

```

Though constructor one and two doesn't have the same number of arguments, you can quickly fail to choose the right one, especially at the end of a busy day using the nice autocomplete from your favorite IDE (I've been there). It's even more difficult to see the difference between constructor one and constructor three. This example looks like a fake one but I saw it on legacy code (true story !).

The question is, how could you implement different constructors with the same type of parameters (while avoiding a dirty way like the constructors one and three) ?

Here is a clean solution using a factory:

```

class Complex {
    public static Complex fromCartesian(double real, double imag) {
        return new Complex(real, imag);
    }

    public static Complex fromPolar(double rho, double theta) {
        return new Complex(rho * Math.cos(theta), rho * Math.sin(theta));
    }

    private Complex(double a, double b) {
        //...
    }
}

```

In this example, using a factory adds a description of what the creation is about with the factory method name: you can create a Complex number from cartesian coordinates or from polar coordinates. In both cases, you know exactly what the creation is about.

The factory patterns

Now that we saw the pros and cons of factories, let's focus on the different types of factory patterns.

I'll present each factory from the simplest to the most abstract. If you want to use factories, keep in mind that the

simpler the better.

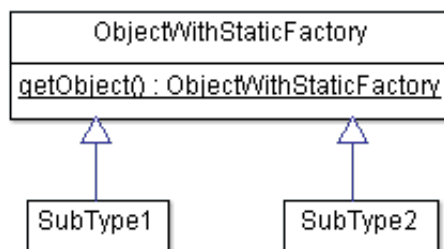
static factory method

Note: If you read this article and don't know a lot about Java, a static method is a class method.

The static factory method was described by Joshua Bloch in "Effective Java":

"A class can provide a public static factory method, which is simply a static method that returns an instance of the class."

In other words, instead of using a constructor to create an instance, a class could provide a static method that returns an instance. If this class has subtypes, the static factory method can return a type of the class or its subtypes. Though I hate UML, I said in the beginning of the article that I'd use UML to give a formal description. Here it is:



In this diagram, the class `ObjectWithStaticFactory` has a static factory method (called `getObject()`). This method can instantiate any type of class `ObjectWithStaticFactory`, which means a type `ObjectWithStaticFactory` or a type `SubType1` or a type `SubType2`. Of course, this class can have other methods, properties and static factory methods.

Let's look at this code:

```
public class MyClass {
    Integer a;
    Integer b;

    MyClass(int a, int b){
        this.a=a;
        this.b=b;
    };
};
```

```

public static MyClass getInstance(int a, int b)
    return new MyClass(a, b);
}

public static void main(String[] args){
    //instanciation with a constructor
    MyClass a = new MyClass(1, 2);
    //instanciation with a static factory method
    MyClass b = MyClass.getInstance(1, 2);
}

```

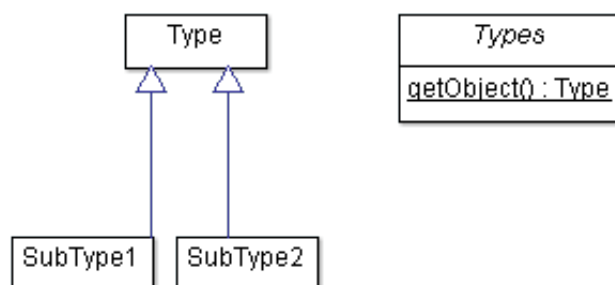
This code shows 2 ways to create an instance of MyClass:

- a static factory method getInstance() inside MyClass
- a constructor of MyClass

But this concept can go deeper. What if a class with a static factory method could instantiate another class? Joshua Bloch described this possibility:

"Interfaces can't have static methods, so by convention, static factory methods for an interface named Type are put in a noninstantiable class (Item 4) named Types."

Here is the associated UML:



In this case the factory method getObject is inside an abstract class name Types. The factory method can create instances of class Type or any subtype of class Type (SubType1 or SubType2 in the diagram). The getObject() method can have parameters so that it returns a SubType1 for a given parameter and a SubType2 otherwise.

Let's go back in Java and assume that we have 2 classes: Ferrari and Mustang that implements an interface Car. The static factory method can be put in an abstract class named "CarFactory" (using Joshua Bloch's conventions the name of the class should be "Cars" but I don't like it):

```

//////////the products
public interface Car {
    public void drive();
}

public class Mustang implements Car{
    public void drive() {
        // some stuff
    }
    ...
}

public class Ferrari implements Car{
    public void drive() {
        // some stuff
    }
    ...
}

////////// the factory
public abstract class CarFactory{
    public static Car getCar() {
        // choose which car you want
    }
}
...
//////////some code using the factory
public static void someFunctionInTheCode(){
    Car myCar = CarFactory.getCar();
    myCar.drive();
}

```

The power of this pattern compared to the other factory patterns is that you don't need

- to instantiate the factory in order to use it (you'll understand what I mean in a few minutes),
- the factory to implement an interface (same comment).

It's easy to use but works only for languages that provides class methods (i.e the static java keyword).

Note: **When it comes to factories, many posts on the net are wrong**, like this post on stackoverflow that was upvoted 1.5k times. The problem with the given examples of factory method patterns is that they are static factory methods. If I quote Joshua Bloch:

"a static factory method is not the same as the Factory Method pattern from *Design Patterns* [Gamma95, p. 107]. The static factory method described in **this item has no direct equivalent in Design Patterns.**"

If you look at the stackoverflow post, only the last example (the URLStreamHandlerFactory) is a factory method pattern by the GoF (we'll see this pattern in a few minutes)

Real examples

Here are some examples of static factory methods in Java frameworks and Java APIs. Finding examples in the Java APIs is very easy since Joshua Bloch was the main architect for many Java APIs.

Logging frameworks

The java logging frameworks slf4j, logback and log4j use an abstract class, LoggerFactory. If a developer wants to write logs, he needs to get an instance of Logger from the static method getLogger() of LoggerFactory.

The Logger implementation returned by getLogger() will depend on the of getLogger() implementation (and also the configuration file written by the developer that is used by getLogger()).

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example{

    public void example() {
        //we're using the static factory method to get our
        Logger logger = LoggerFactory.getLogger(Example.class);

        logger.info("This is an example.");
    }
}
```

Note: the name of the factory class and its static factory method is not exactly the same whether you're using slf4j or log4j or slf4j.

Java String class

The String class in Java represents a string. Sometimes, you need to get a string from a boolean or an integer. But String doesn't provide constructors like String(Integer i) or String(Boolean b). Instead, it provides multiple static factory methods String.valueOf(...).

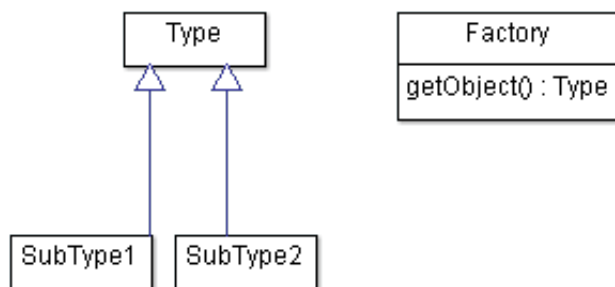
```
int i = 12;
String integerAsString = String.valueOf(i);
```

simple factory

This pattern is not a “real one” but I’ve seen it many times on the Internet. It doesn’t have a formal description but here is mine: A simple factory (or factory) is a tool

- whose job is to create/instanciate objects,
- and is neither a factory method pattern (we’ll see this pattern after),
- nor an abstract factory pattern (same comment).

You can see it has a generalization of the static factory pattern but this time the factory can be instantiated (or not) because the “factory method” is not a class method (but it can). For a Java developer using the simple factory in its non-static form is rare. So, **this pattern is most of the time equivalent to the static one**. Here is an UML for the non-static form:



In this case the factory method getObject() is inside a class named Factory. The factory method is not class method so, you need to instantiate the Factory before using it. The factory method can create instances of class Type or any of its subtypes .

Here is the previous example from the static factory method but this time I instantiate the factory before using it

```
//////////the products
public interface Car {
    public void drive();
}

public class Mustang implements Car{
    public void drive() {
        // some stuff
    }
    ...
}
```

```

}

public class Ferrari implements Car{
    public void drive() {
        // some stuff
    }
    ...
}
////////////////////////The factory
public class CarFactory{
    //this class is instantiable
    public CarFactory(){
        //some stuff
    }
    public Car getCar() {
        // choose which car you want
    }
}
...
////////////////////////some code using the factory
public static void someFunctionInTheCode(){
    CarFactory carFactory = new CarFactory();
    Car myCar = carFactory.getCar();
    myCar.drive();
}

```

As you see, this time I need to instantiate the Factory in order to use it. I didn't find real example in java since **it's better to use a static factory method than a simple factory**. Still, you can use this pattern in its non-static form if your factory method needs some instances to work. For example, if you need a database connection, you could first instantiate your factory (that would instantiate the database connection) and then use the factory method that requires this connection. Personally, in this case I'd still use a static factory with lazy initialization (and a pool of database connections).

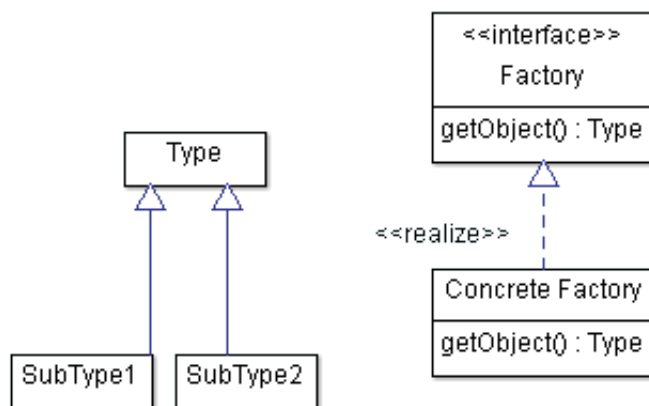
Tell me if you know a Java framework that uses a simple factory in its non-static form.

factory method pattern

The factory method pattern is a more abstract factory. Here is the definition of the pattern given by the "Gang of Four":

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses"

Here is a simplified UML diagram of the factory method pattern:



This diagram looks like the simple factory one (in its non-static form). The only (and **BIG !**) difference is the interface Factory:

- the Factory represents the “interface for creating an object”. It describes a factory method: getObject().
- the ConcreteFactory represents one of the “subclasses [that] decide which class to instantiate”. Each ConcreteFactory has its own implementation of the factory method getObject().

In the diagram getObject() has to return a Type (or its subtypes). Which means that one concrete factory could return a SubType1 whereas another could return a SubType2.

Why using a factory method pattern instead of a simple factory?

Only when your code requires multiple factory implementations. This will force each factory implementation to have the same logic so that a developer that uses one implementation can easily switch to another one without wondering how to use it (since he just has to call the factory method that has the same signature).

Since this is abstract, let's go back to the car example. It's not a great example but I use it so that you can see the difference with the simple factory, (we'll see the real examples to understand the power of this pattern):

//////////the products


```

public interface Car {
    public void drive();
}

public class Mustang implements Car{
    public void drive() {
        // some stuff
    }
    ...
}

public class Ferrari implements Car{
    public void drive() {
        // some stuff
    }
    ...
}

////////// the factory
//the definition of the interface
public interface CarFactory{
    public Car getCar() {}
}

//the real factory with an implementation of the get
public class ConcreteCarFactory implements CarFact
    //this class is instantiable
    public CarFactory(){
        //some stuff
    }
    public Car getCar() {
        // choose which car you want
        return new Ferrari();
    }
}

...
//////////some code using the factory
public static void someFunctionInTheCode(){
    CarFactory carFactory = new ConcreteCarFactory();
    Car myCar = carFactory.getCar();
    myCar.drive();
}

```

If you compare this code with the simple factory, I added this time an interface (*CarFactory*). The real factory (*ConcreteCarFactory*) implements this interface.

As I said this is not a great example because **in this example you shouldn't use a factory method pattern since there is only one concrete factory**. It would be useful only if I have multiple implementations like *SportCarFactory*, *VintageCarFactory*, *LuxeCarFactory*, *CheapCarFactory* In this case, a developer could easily switch from one implementation to another since the factory method is always `getCar()`.

Real examples

Java API

In java, a common example is the `iterator()` function in the collection API. Each collection implements the interface `Iterable<E>`. This interface describes a function `iterator()` that returns an `Iterator<E>`. An `ArrayList<E>` is a collection. So, it implements the interface `Iterable<E>` and its factory method `iterator()` that returns a subclass of `Iterator<E>`

```
//here is a simplified definition of an iterator from
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}

//here comes the factory interface!
public interface Iterable<T> {
    Iterator<T> iterator();
}

//here is a simplified definition of ArrayList from
//you can see that this class is a concrete factory
//a factory method iterator()
//Note : in the real Java source code, ArrayList is a
//AbstractList which is the one that implements the
public class ArrayList<E> {
    //the iterator() returns a subtype and an "anonymous"
    public Iterator<E> iterator()
    {
        return new Iterator<E>()
        {
            //implementation of the methods hasNext(), next()
        }
    }
    ...
}
```

And here is a standard use of the `ArrayList`

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Example {
    public static void main(String[] args){
        //instantiation of the (concrete factory) ArrayList
        List<Integer> myArrayList = new ArrayList<>();
        //calling the factory method iterator() of ArrayList
        Iterator<Integer> myIterator = myArrayList.iterator();
    }
}
```

I showed an `ArrayList` but I could have shown a `HashSet`, a `LinkedList` or a `HashMap` since they are all part of the collection API. **The strength of this pattern is that you**

don't need to know what type of collections you're using, each collection will provide an Iterator through the factory method iterator().

Another good example is the stream() method in the new Java 8 collection API.

Spring

The Spring Framework is based on a factory method pattern. The ApplicationContext implements the BeanFactory Interface. This interface describes a function Object getBean(param) that returns an Object. This example is interesting because, every Class in java are derived from Object. So, this factory can return an instance of any class (depending on the parameters).

```
public class Example{  
    public static void main(String[] args) {  
        //creation of the BeanFactory  
        ApplicationContext context = new ClassPathXmlAppL  
        //creation totally different type of objets with tl  
        MyType1 objectType1 = context.getBean("myType1");  
        MyType2 objectType2 = context.getBean("myType2");  
    }  
}
```

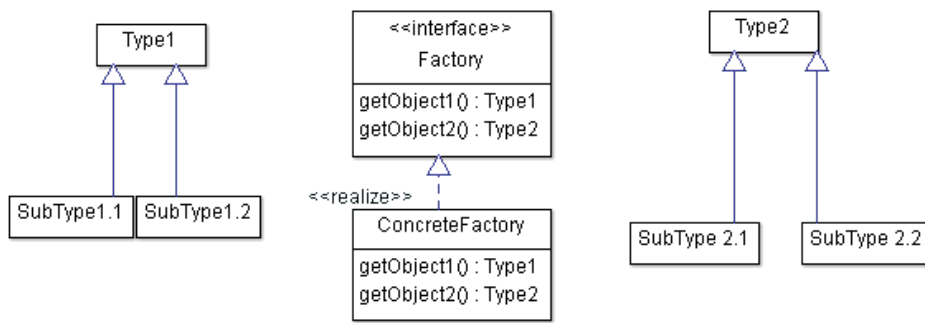
The abstract factory

Here comes the fat one ! This factory was described by the Gang of Four with the following sentence:

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes"

If you don't understand this sentence, don't worry it's normal. It's not called **abstract** factory for nothing!

If it can help you, I see the abstract factory pattern as a generalization of the factory method pattern except this time the factory interface have **multiple factory methods that are related**. When I say related, I mean conceptually linked so that they form a "family" of factory methods. Let's look at the UML diagram to see the difference with the factory method pattern:



- Factory is an interface that defines **multiple factory methods**, 2 in our cases: getObject1() and getObject2(). Each method creates a type (or its subtypes).
- ConcreteFactory implements the Factory interface and therefore has its own implementation of getObject1() and getObject2(). Now imagine 2 concrete factories: one could return instances of SubType1.1 and SubType2.1 and the other SubType1.2 and SubType2.2

Since this is very abstract, let's go back to the CarFactory example.

With the factory method pattern, the factory interface had just one method, getCar(). An abstract factory could be an interface with 3 factory methods: getEngine(), getBody() and getWheel(). You could have multiple concrete factories:

- SportCarFactory that could return instances of PowerfulEngine, RaceCarBody and RaceCarWheel
- CheapCarFactory that could return instance of WeakEngine, HorribleBody and RottenWheel

If you want to build a sport car, you'll need to instantiate a SportCarFactory then use it. And if you want to build a cheap car, you'll need to instantiate a CheapCarFactory then use it.

The 3 factory methods of this abstract factory are related. They all belong to the car production concept. Of course, the factory methods can have parameters so that they return different types. For example getEngine(String model) factory from SportCarFactory could return a Ferrari458Engine or a FerrariF50Engine or a Ferrari450Engine or ... depending on the parameter.

Here is the same example in java (with only the SportCarFactory and 2 factory methods).

```
//////////the different products
public interface Wheel{
    public void turn();
}

public class RaceCarWheel implements Wheel{
    public void turn(){
        // some stuff
    }
    ...
}

public interface Engine{
    public void work();
}

public class PowerfulEngine implements Engine{
    public void work(){
        // some stuff
    }
    ...
}

//////////the factory
public interface CarFactory{
    public Engine getEngine();
    public Wheel getWheel();
}

public class SportCarFactory implements CarFactory
{
    public Engine getEngine(){
        return new PowerfulEngine();
    }
    public Wheel getWheel(){
        return new RaceCarWheel();
    }
}

//////////some code using the factory
public class SomeClass {
    public void someFunctionInTheCode(){
        CarFactory carFactory = new SportCarFactory();
        Wheel myWheel= carFactory.getWheel();
        Engine myEngine = carFactory.getEngine();
    }
}
```

This factory is not an easy one. **So, when should you use it ?**

~~NEVER!!!!~~ Hum, difficult to answer. I see this factory pattern as a way to organise code. If you end up with many factory method patterns in your code and you see a common theme between some of them, you can gather this group with an abstract factory. I'm not a big fan of the "let's use an abstract

factory because we could need one in the future” because this pattern is very abstract. I prefer building simple things and **refactor** them after if needed.

Yet, a common use case is when you need to create a user interface with different look and feel. This example was used by the Gang of Four to present this pattern. This UI will need some products like a window, a scroll bar, buttons ... You can create a Factory with a concret factory for each look and feel. Of course, this example was written before the Internet era, now you could have one component and modify its look and feel using CSS (or some scripting languages) even for desktop applications. Which means a static factory method is most of the time enough.

But if you still want to use this pattern, here are some use cases from the GoF:

"a system should be configured with one of multiple families of products"

"you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations"

Real examples

Most DAO (Data Access Object) frameworks use an abstract factory to specify the basic operations a concrete factory should do. Though the names of the factory methods depend on the framework, it's often closed to:

- `createObject(...)` or `persistObject(...)`
- `updateObject(...)` or `saveObject(...)`
- `deleteObject(...)` or `removeObject(...)`
- `readObject(...)` or `findObject(...)`

For each type of object you manipulate, you'll need a concrete factory. For example, if you need to manage people, houses, and contracts using a database. I'll have a `PersonFactory`, a `HouseFactory` and a `ContractFactory`.

The `CrudRepository` from Spring is a good example of an abstract Factory.

If you want Java code, you can look for JPA, Hibernate or SpringData tutorials.

Conclusion

I hope that you now have a good idea of the different types of factory patterns and when to use them. Though I said it many times in this article, keep in mind that most of the time a factory make the code more complex/abstract. Even if you know the factories (if you don't, read this article again!), what about your co-workers? Yet, when working on medium/large applications it's worth using factories.

I struggled a long time with the factories to understand which is which and an article like this one would have helped me. I hope this article was understandable and that I didn't write too many mistakes. Feel free to tell me if something you read bothered you so that I can improve this article.



6

3

Design Pattern design pattern factory java

Related Posts

Design pattern: singleton, prototype and builder

How does a HashMap work in JAVA

References in JAVA

Memory optimisation: Custom Set

Design Pattern: Liskov's Substitution Principle (LSP)

«« Leonard Susskind's Quantum Mechanics course Design pattern: singleton, prototype and builder »»