Abstract Factory Design Pattern in Java

JULY 15, 2018 BY PANKAJ — 28 COMMENTS

Welcome to Abstract Factory Design Pattern in java example. Abstract Factory design pattern is one of the Creational patterns. Abstract Factory pattern is almost similar to **Factory Pattern** except the fact that its more like factory of factories.

Table of Contents [hide]

Abstract Factory

If you are familiar with **factory design pattern in java**, you will notice that we have a single Factory class. This factory class returns different subclasses based on the input provided and factory class uses if-else or switch statement to achieve this.

In the Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class. Then an Abstract Factory class that will return the sub-class based on the input factory class. At first, it seems confusing but once you see the implementation, it's really easy to grasp and understand the minor difference between Factory and Abstract Factory pattern.

Like our factory pattern post, we will use the same superclass and sub-classes.

Abstract Factory Design Pattern Super Class and Subclasses

`Computer.java`

```java
package com.journaldev.design.model;

public abstract class Computer {

    public abstract String getRAM();
    public abstract String getHDD();
    public abstract String getCPU();

    @Override
    public String toString(){
        return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();
    }
}
```

`PC.java`

```
package com.journaldev.design.model;

public class PC extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }
```

Server.java

```
package com.journaldev.design.model;


public class Server extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public Server(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }
    @Override
    public String getHDD() {
        return this.hdd;
```

Factory Class for Each subclass

First of all we need to create a Abstract Factory interface or **abstract class**.

`ComputerAbstractFactory.java`

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public interface ComputerAbstractFactory {

        public Computer createComputer();

}
```

Notice that `createComputer()` method is returning an instance of super class `Computer`. Now our factory classes will implement this interface and return their respective sub-class.

`PCFactory.java`

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;

public class PCFactory implements ComputerAbstractFactory {

        private String ram;
        private String hdd;
        private String cpu;

        public PCFactory(String ram, String hdd, String cpu){
                this.ram=ram;
                this.hdd=hdd;
                this.cpu=cpu;
        }
        @Override
        public Computer createComputer() {
                return new PC(ram,hdd,cpu);
        }

}
```

Similarly we will have a factory class for `Server` subclass.

`ServerFactory.java`

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.Server;

public class ServerFactory implements ComputerAbstractFactory {

        private String ram;
        private String hdd;
        private String cpu;

        public ServerFactory(String ram, String hdd, String cpu){
                this.ram=ram;
                this.hdd=hdd;
                this.cpu=cpu;
        }

        @Override
        public Computer createComputer() {
                return new Server(ram,hdd,cpu);
        }
```

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

`ComputerFactory.java`

```
package com.journaldev.design.abstractfactory;
```

```
import com.journaldev.design.model.Computer;

public class ComputerFactory {

        public static Computer getComputer(ComputerAbstractFactory factory){
                return factory.createComputer();
        }
}
```

Notice that its a simple class and `getComputer` method is accepting `ComputerAbstractFactory` argument and returning `Computer` object. At this point the implementation must be getting clear.

Let's write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

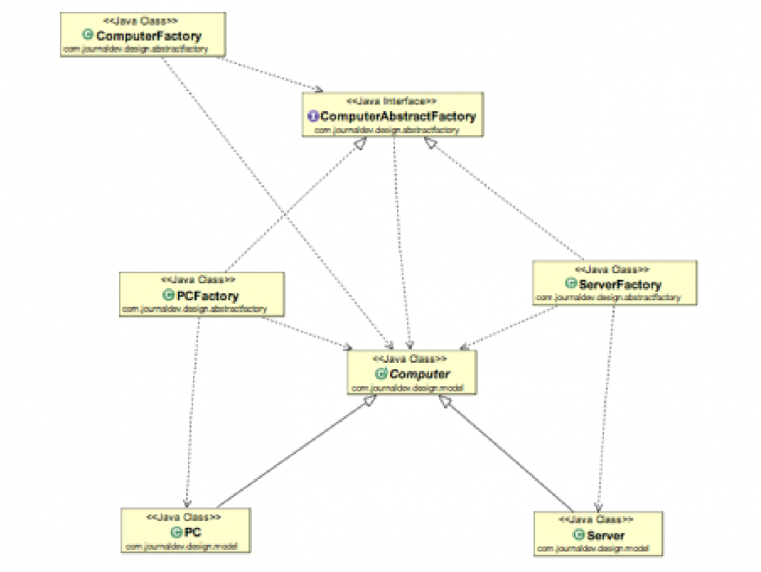
`TestDesignPatterns.java`

```
package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;

public class TestDesignPatterns {

        public static void main(String[] args) {
                testAbstractFactory();
        }

        private static void testAbstractFactory() {
                Computer pc = 
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 
GB","500 GB","2.4 GHz"));
                Computer server = 
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new 
ServerFactory("16 GB","1 TB","2.9 GHz"));
                System.out.println("AbstractFactory PC Config::"+pc);
                System.out.println("AbstractFactory Server Config::"+server);
```

Output of the above program will be:

```
AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

Here is the class diagram of abstract factory design pattern implementation.


<<Java Class>>
ComputerFactory
com.journaldev.design.abstractfactory
<<Java Interface>>
ComputerAbstractFactory
com.journaldev.design.abstractfactory
<<Java Class>>
PCFactory
com.journaldev.design.abstractfactory
<<Java Class>>
ServerFactory
com.journaldev.design.abstractfactory
<<Java Class>>
Computer
com.journaldev.design.model
<<Java Class>>
PC
com.journaldev.design.model
<<Java Class>>
Server
com.journaldev.design.model

Abstract Factory Design Pattern Benefits

- Abstract Factory design pattern provides approach to code for interface rather than implementation.
- Abstract Factory pattern is "factory of factories" and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
- Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

Abstract Factory Design Pattern Examples in JDK

- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

Abstract Factory Design Pattern Video Tutorial

I recently uploaded a video on YouTube for abstract factory design pattern. In the video, I discuss when and how to implement an abstract factory pattern. I have also discussed what is the difference between the factory pattern and abstract factory design pattern.

Subscribe to my YouTube Channel YouTube 6K

You can download the examples code from my GitHub Project.

« PREVIOUS
Factory Design Pattern in Java

NEXT »
Builder Design Pattern in Java

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

Andrii says
JULY 14, 2018 AT 11:36 AM
Hello,
There is a small typo – "Banefits"
Regards,
Andrii
Reply

Pankaj says
JULY 15, 2018 AT 8:45 AM
Thanks buddy, corrected it.
Reply

Ashish Doneriya says
MARCH 29, 2018 AT 9:52 PM
If in the end the user has to write new PCFactory() and new ServerFactory() then what is the use of creating all those interfaces. means instead of PCFactory and ServerFactory he could use new PC() and new Server()
Reply

Harut says
MAY 13, 2018 AT 10:34 PM
Totally agree with you Not a good sample of abstract factory
Reply

Abbin Varghese says
MAY 14, 2018 AT 7:08 AM
As mentioned in the benifits, I think the idea is the possibility to extend..
"can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory."
Reply

abc says
FEBRUARY 25, 2018 AT 10:49 PM
good
Reply

kai says
DECEMBER 7, 2017 AT 8:56 PM
Your article and your example is very good and useful.
I used some your example on my blog: https://stackjava.com and I place back link to your page.
Because my students can not read english language..
Please let me know if you feel bothered I will remove it.
Thanks!
Reply

Jacek Feliksiak says
JUNE 21, 2017 AT 1:32 AM
I think it is worth to add one more abstract product and bunch of end products which extends this one (two sets of products). Than it makes sens.
https://en.wikipedia.org/wiki/Abstract_factory_pattern#/media/File:Abstract_factory_UML.svg
Reply

Valentino says
MARCH 26, 2017 AT 10:54 PM
Hi Pankaj. A question. In your TestDesignPatterns , it looks like the client directly knows about PCFactory and ServerFactory. Is this correct? Isn't this a kind of coupling between the client and the factory classes?
Reply

Kailash Singh says
MARCH 17, 2017 AT 7:44 AM
This is no correct abstract factory implementation, here client knows about the concrete factories and just passing to consumer. Client shouldn't know about concrete factories or concrete computer implementations.
Abstract factory only should know the actual concrete factories to be used.
Reply

Rajesh KSV says
JANUARY 19, 2017 AT 10:47 PM
Good Article.
But if you aren't still convinced why this pattern is to be used, we can refer to another example.
Say you are writing application code independent of OS . Like Google Chrome Code. Now there will be a factory class for Windows OS to generate Windows Buttons, Menus etc. Similarly there will be another factory class for Linux OS to generate its buttons, menus. Now application code instead of coding to any of these factory classes will code to its interface – AbstractFactoryClass for simplicity
This class is used more than to prevent if/else in factory pattern
Reply

RazorEdge says
FEBRUARY 26, 2017 AT 11:54 AM
Good Example
Reply

Vinay Kallat says
APRIL 15, 2016 AT 3:47 AM
I do not feel the If – else logic is actually irradicated. Sorry, to spoil the fun. When i first read it, I felt at last something that explains a difference between Factory and abstract factory. The if-else is removed from the Factory code as in other examples you find here and many other link http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm. The if-else logic is still present in the factory class.
However, this seems to b worse. Actually the responsibility has shifted to client side.
Imagine, at compile time you do not what the client requests and the code would look like.
String computerType= getRequestFromClient();
Computer remoteComputer = null;

//Now this is where the actual if else comes in.
if(computerType.equals("PC")){
remoteComputer = ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));
} else if(computerType.equals("SERVER")){
remoteComputer = ComputerFactory.getComputer(new ServerFactory("16 GB","1 TB","2.9 GHz"))
}
//And remember this code block would be present everywhere you want the remoteComputer object from the factory.. This was the reason why this code was moved to Factory to not let client worry about it..
Else why to go with so much of Fuzz just type in:
if(computerType.equals("PC")){
remoteComputer = new PCFactory("2 GB","500 GB","2.4 GHz");
} else if(computerType.equals("SERVER")){
remoteComputer = new Server("16 GB","1 TB","2.9 GHz"))
}
//This looks much cleaner. This is why I feel som many people are confused with Design patterns because everyone just explain the way they like it.
Reply

Udi Reshef says
DECEMBER 3, 2017 AT 8:41 AM
In this example you are right. It seems redundant to use this pattern. better use simple factory. because the code is very simple indeed.
The base of this example was build first to explain simple factory pattern:
https://www.journaldev.com/1392/factory-design-pattern-in-java
But I think there is a place for abstract factory pattern where your instantiations are very complicated,
too complicated and ugly for a single factory and too complicated for the UI to comprehends..
Let's say you would like to have an option to make another brand of objects to instantiate – and for sure it will be too much for a single factory class to have this brand instantiate as well as the previous brands.
Imagine there is a detail sophisticated info needed in order to make the correct server class object, and in this object you need to know exactly which parameters to tune and how to tune them. Also lets say this is a brand not a single class.. let's say there is a family of 20 kind of severs,
and also 20 kind of PC`s..
In the special factory for PC we will have them differentiate and get the exact PC to instantiate and also how to instantiate it . we will know that according to input from the net (let's say what color is available in the online store) , and from other applications and services running in background (the UI is not aware of all this details).
And maybe tomorrow we will have another family of let's say "laptops" to instantiate.
So the UI will have the "if else" to know if the user want a "server", "pc" or "laptop" – but the factories classes will decide exactly which server, pc or laptop to build (and how to tune it – what values to set to its parameters , or to send to its contractor) according to many parameters that the UI is not

aware of.
All of this will be too much for a single factory class.
Reply

bala says
SEPTEMBER 17, 2015 AT 11:28 PM
clean explanation sir .can u provide pdf tutorial sir
Reply

Badri says
JUNE 7, 2014 AT 9:58 AM
Hi Pankaj,
I am not able to understand Abstract Factory Design Pattern perfectly.Can you please help me ??
In ComputerFactory class, getComputer() method is taking ComputerAbstractFactory as argument.Are you sure this is correct ??
In your client program i.e. TestDesignPatterns.java,
To create pc, the code is " Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));".
It means client have access to class PCFactory.
When client has access to class PCFactory, then pc can be created without using ComputerFactory class.
PCFactory factory = new PCFactory("2 GB","500 GB","2.4 GHz");
Computer pc = factory.createComputer();
What is the use of ComputerFactory class when client have direct access to PCFactory or ServerFactory???
Best Regards,
Badri
Reply

Shiva says
JUNE 13, 2014 AT 10:39 AM
You can follow this technical example… (But follow explanation as mentioned by Pankaj – its awesome)
http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
Thanks,
Shiva
Reply

Badri says

JUNE 16, 2014 AT 10:15 AM

@Shiva:
Thanks for the reply.
I gone through the link
http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm.
My actual concern for posting the question is:
Pankaj sir is highlighting this sentence "In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class."
The example shown in http://www.tutorialspoint.com/ is using **if else block to decide which factory to return.**
When I am implementing this pattern,should
1.I use if – else block to decide which factory to return.
OR
2.I take AbstractFactory(like ComputerAbstractFactory) as argument and just delegate the call (as Pankaj explained in this example).
Which one is correct approach ??
In case I am using second approach , how should I avoid from using the below code:
PCFactory factory = new PCFactory("2 GB","500 GB","2.4 GHz");
Computer pc = factory.createComputer();
Best Regards,
Badri

Reply

Peter Yan says

OCTOBER 23, 2014 AT 7:36 PM

Each factory is a plugin, in real project, which concrete factory to use is configurable in configure file rather than in code.
For client, what it need is one kind of products from one factory at one time.
If we put all kinds of products in one factory, there must be a lot of if/else conditions. One day, If client want replace with another kind of product. we must go back and modify this factory.
On the other hand, if we are using abstract factory, all we need to do is to add a new concrete factory. and change configuration for factory.

Reply

Badri says

JUNE 14, 2014 AT 12:14 AM

Hi Pankaj,

Can you kindly respond to my previous comment??

Best Regards,

Badri

Reply

Vishwas says

JANUARY 16, 2014 AT 5:50 AM

One thing is nagging me in your example.

Your factory should be without parameters and creatComputer should accept the parameters, currently multile calls to createComputer will create same computer instead of ones with different configuration.

Existing methods:

public ServerFactory(String ram, String hdd, String cpu){

public Computer createComputer() {

Should be changed to these methods with following signature.

public ServerFactory()

public Computer createComputer(String ram, String hdd, String cpu)

Factory should be without parameters and createComputer should accept parameters.

Reply

Pankaj says

JANUARY 16, 2014 AT 6:18 PM

No its fine because Abstract Factory pattern is "Factory of Factories", what you are telling is Factory pattern where we have a single factory and based on user inputs, different objects are created.

Reply

Praveen says

JULY 8, 2013 AT 1:45 PM

Hi,

Is there any pdf file available for all these design patters?

Regards,

Praveen

Reply

Pankaj says
JULY 9, 2013 AT 3:21 AM
I am planning to provide a PDF tutorial for all the java design patterns in a future post.
Reply

Vikram Bakhru says
APRIL 5, 2014 AT 1:36 PM
Hi Pankaj,
Thanks for the wonderful tutorial. Is the pdf available now. I couldn't find it on your website. Can you please provide me the direct URL if it is already available.
Regards,
Vikram
Reply

Shiva says
JUNE 13, 2014 AT 10:40 AM
Thanks a lot ..
Please provide pdf kind of doc ..
Thanks,
Shiva
Reply

Madhusudhan Takkella says
APRIL 19, 2017 AT 11:03 AM
This example is pretty much good. But, the only modification i can suggest is, from client program, instead of instantiating the factory classes for PC and Server directly, we can have one more factory which provides instance of PC and Server based on some input like If "PC" -> create instance of PCFactory else if it is "Server" -> create instance of ServerFactory.
Reply

Naveen goyal says
FEBRUARY 13, 2018 AT 5:47 AM
That is tight coupling and the example showed above is less coupled. Tightly coupled classes can create future modification issues. and also The way you are telling to instantiate the object. It is used in factory design pattern

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

☐

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

GET IT ON Google Play

DESIGN PATTERNS TUTORIAL

- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial