Abstract Factory Design Pattern

1. Abstract Factory – Introduction:

We are going to learn about abstract factory design pattern. Now if you are learning about this design pattern for the very first time, then I must warn you that first few slides may sound confusing, but don't worry, we have an example ahead that we are going to use in order to understand this design pattern in a better way. So what is the purpose of Abstract Factory and where we can use this design pattern? Whenever, in our application, we have two or more objects of two or more classes that work together, that means they are designed to work together and they form a kit/set of some objects. And whenever we have such multiple kits that can be created or multiple sets of such objects that our client needs, then we can use the abstract factory. I know it sounds a little bit confusing, but just hold on for a few seconds. The intent behind the abstract factory is to separate or isolate our client code from these objects of concrete classes forming such a set and also from the code that creates these sets of objects. So it not only separates the client from the concrete implementations of the objects that it is using, it also isolates the client from the code that creates these concrete objects.

A black box with white text

Description automatically generated

I know it sounds quite a bit confusing. It seems a lot of things are going on here. So what we are going to do is we're going to go to the example that we are going to use in order to understand this design pattern. You may have heard about strategy games that can be played on computer. 'Age of Empire' is a classic game, which is a real time strategy game. 'Civilization' series is also an example of turn based strategy games. The way these games work is that you start with a limited resource on a map and you start in a particular age. For example, you can start in a 'stone' age, 'dark' age or 'medieval' age and using your resources, you can start building units and doing research using those resources that you created. And once your research reach at a particular point or reached a particular score then you can advance to the next age. For example, you can go to the 'industrial' age or 'modern' age, so on and so forth. In this games, let's say if you are currently in a medieval age and you want to build a Naval unit, now since you are in medieval age, game will allow you to build a Galley, which is a type of ship that was used in medieval times. On the similar lines, If you want to create a Land unit or a soldier then the game would allow you to build Swordsmen. Again, this is the Land unit that was present or used in the medieval times. Now, let's say you have accumulated enough research so that you can progress or advance to the next stage. And let's say that is the industrial age. Now, if you build a Naval unit, the game will build an Ironclad or Steamboat, which was a type of ship that was used in industrial age. And on the similar lines, if you want to build a Land unit in industrial age, you would create a Rifleman because riflemen were present or used in the industrial age. So now if you look carefully, you will see that Galley and Ironclad, they both are Naval units. Swordsman and Rifleman, they both are Land units, but the Galley and Swordsman form a set(a set of units) that you can build if you are in medieval age and Ironclad and Rifleman form another set which you can build in industrial age. And this is what we mean when we are talking about these kits or sets in the abstract factory design pattern. So what Abstract Factory does is that it allows you to say that I want to build a Naval unit and the design pattern will decide or it will take care of which exact class to use, whether to use the Galley class and create a galley or whether to use Ironclad and create the ironclad or steamboat unit. So Abstract Factory really works great in these types of problems where we have set of objects that work together.



Now let's look at the UML diagram of this particular example that we just looked at. So here we have a LandUnit, which is an abstract class or let's say an interface, and there are two child classes here. We have a Swordsman and Rifleman and they both are examples of land units. On the similar lines, we have here a NavalUnit, again, an interface or an abstract class. And there are two child classes of this NavalUnit class, a Galley or an Ironclad. However, as we saw Swordsman and Galley form a set, objects of these two classes form a set, that means they are meant to work together. So if you are in medieval age, you can build Swordsman or Galley. If you are in industrial age, you can build Rifleman or an Ironclad. So these two classes or objects of these two classes form another set. And this is where our Abstract Factory comes into picture. Now let's say you are writing code for an AI(in our case, it is Client) for a strategy game. Now if you don't use Abstract Factory then your AI(in our case, it is Client) code will be littered with if-else checks and switch case ladder to determine which one of these four units you want to build. And then you have to consider all these parameters to find out which age your AI(in our case, it is Client) currently belongs to. So instead of that, what we can do here is that we can create an interface called as GameUnitFactory and we can code our entire AI(in our case, it is Client) code using this particular interface. Now this interface defines two abstract methods only. It defines a method to create a land unit[i.e., createLandUnit()] and another method to create naval unit[i.e., createNavalUnit()]. And we have two implementations for this interface. One class, called as MedievalGameUnitFactory, is a concrete class that can create all these units(i.e., land unit and naval unit) which belong in the medieval times. And another implementation, called as IndustrialAgeGameUnitFactory, can create land unit and naval unit that belong in the industrial age. So you will initialize/provide your AI(in our case, it is Client) code with instance of one of these concrete classes(i.e., IndustrialAgeGameUnitFactory or MedievalGameUnitFactory) and it can start creating land and naval units and it will be the appropriate unit for that particular age.

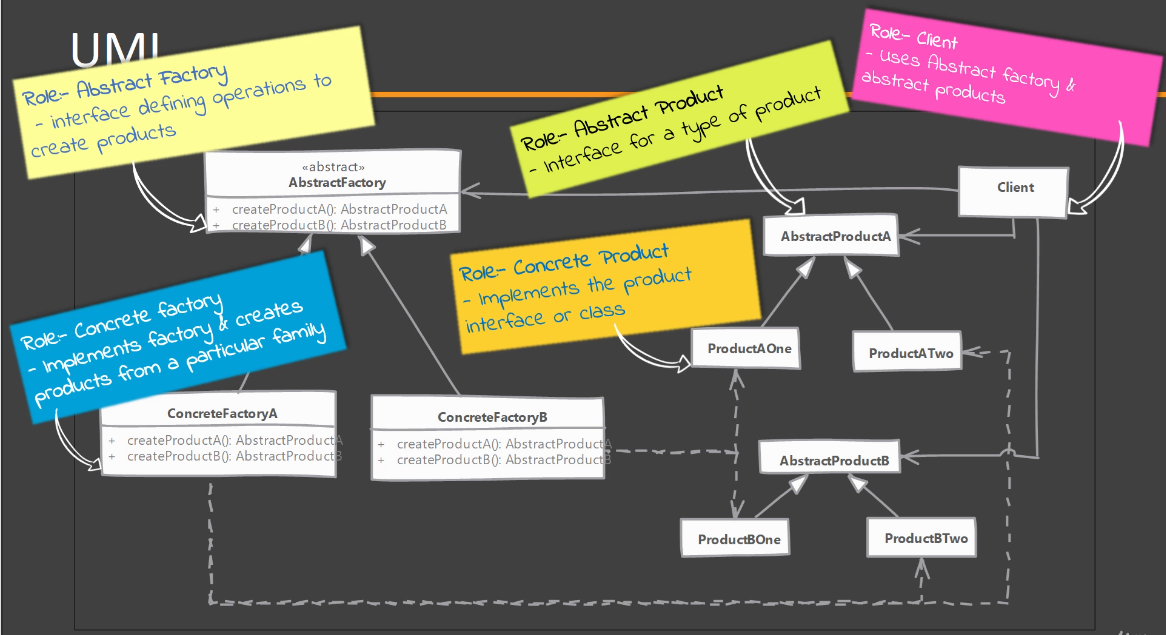
A screenshot of a computer

Description automatically generated

So what we're going to do next is we're going to see this old UML diagram which uses these abstract terms like product, concrete factory, abstract factory, etc in order to describe this abstract factory design pattern. So we are going to again go through these roles one by one. To start off, we have here an AbstractProductA. So this interface represents a type of product that can be used by our client. Then we have the actual concrete implementation of this AbstractProductA. So here we have a ProductAOne class and we can have many such concrete implementations of that interface(i.e., AbstractProductA). Then we have an interface called as AbstractFactory. Now, this interface defines methods that allow our client to create products of a particular type. So we have a method called as createProductA() which can create object of any child class of AbstractProductA. And we have another method called as createProductB() that can create object of any child class of AbstractProductB. Then we have the actual concrete implementation of this AbstractFactory and they will create objects from a particular set. Lastly, we have our Client which uses one of these concrete factories(ConcreteFactoryA and ConcreteFactoryB) in order to create its object. So this is the abstract factory design pattern.

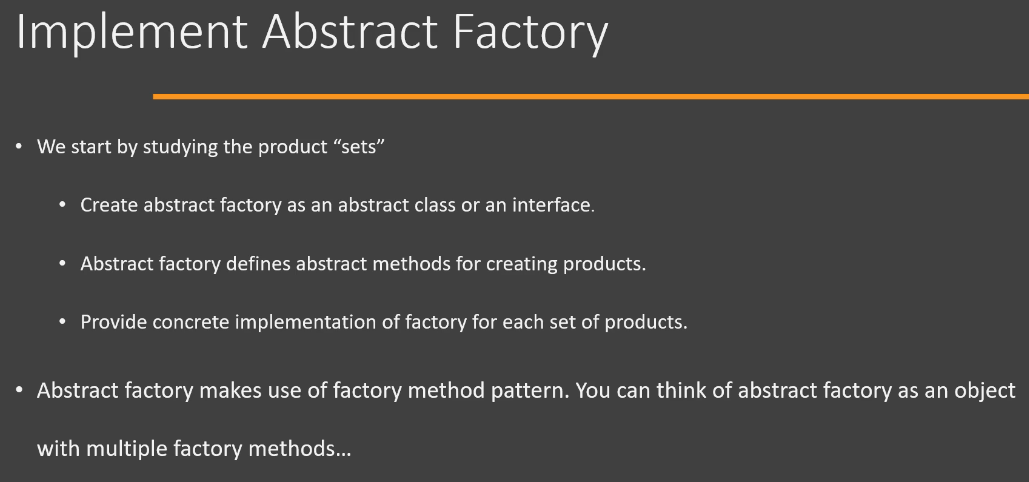
A screenshot of a computer

Description automatically generated



1. Abstract Factory – Implementation Steps:

Let's look at some of the steps that you would typically follow if you are implementing Abstract Factory. We start by studying the sets or kits or families of products that are present in our application. Once we have found out such sets, we then start by creating our abstract factory. Now, typically this will be an interface or it could be an abstract class. And inside this interface we are going to define methods that allow the client code to create objects of different types. Now the goal here is that we are going to implement this abstract factory and each concrete implementation is going to create objects which belong to same set. So once your abstract factory is defined, you can implement or create the concrete implementations that correspond to each set of such product families. Now, one thing you will note is that we are actually making use of factory method design pattern. So in case you don't remember, in factory method design pattern, we define an abstract factory method and subclasses or child classes would override or implement that method and create concrete product classes. This is exactly what is happening in Abstract Factory design pattern. However, we have an abstract class or an interface that contains these factory methods. So just wanted to point that out, that abstract factory itself uses factory method design pattern.



1. Abstract Factory – Hands-on Example UML:

Let’s look at the UML of the example that we are going to implement in Java code. Here, we are trying to implement a system where we can represent cloud resources that we can provision on different cloud providers. So, we are going to consider AWS and GCP. The point here is how you can create families of objects that are related together. So, we are going to have an Instance, which is an interface, and this is going to represent the compute resource. There are two child classes of Instance, one is Ec2Instance which is provisioned on AWS and second is GoogleComputeEngineInstance which is provisioned on GCP. In the same way, we have a Storage interface which represents the cloud storage. Again, we have two subclasses, one is S3Storage which is available on AWS and GoogleCloudStorage which is available on GCP. So, the idea here is, if you are working on AWS services and you want to provision a compute engine/machine then you would be creating Ec2Instance and S3Storage for storing. So, you can’t have an Ec2Instance and GoogleCloudStorage together because they are on different cloud service providers. Typically what you will do is, you will provide the compute engine and storage on the same cloud service provider(CSP) so that they can be connected together. ResourceFactory is our abstract factory and it is an interface having two abstract factory methods. First is createInstance(capacity) method which is going to create an object of one of the child classes of Instance. Second is createStorage(in Mib) method which is going to create an object of one of the child classes of Storage. We have two concreate implementations of our ResourceFactory( which is an abstract factory). One is GoogleCloudResourceFactory which will create resources like compute and storage on GCP and another is AwsResourceFactory which will create resources like compute and storage on AWS. In other words, AwsResourceFactory class will create objects of the classes which represents resources on AWS and similarly, GoogleCloudResourceFactory class will create objects of the classes which represents resources on GCP.

A diagram of a company

Description automatically generated

1. Abstract Factory – Implementation:

Extract abstract\_factory\_begin.zip under Creational\_design\_pattern/Abstract\_factory\_design\_pattern folder for the implementation in Java.

Now let's look at the implementation of Abstract Factory design pattern. I have implemented some of the classes that we are going to use while implementing an abstract factory. So these classes are exactly similar to what we discussed in the hands-on example UML that we saw previously.

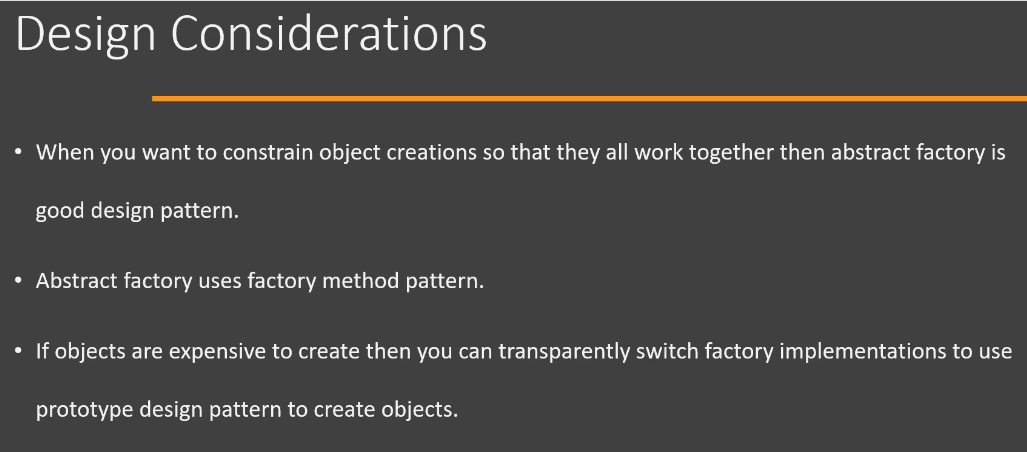
1. Abstract Factory – Implementation and Design considerations:

Let's look at some of the implementation and design considerations while using Abstract Factory. One point that you may have noticed is that we can implement our concrete factories(i.e., GoogleCloudResourceFactory and AwsResourceFactory --> in our hands-on example) as singletons because we typically need only one instance of our concrete factory and we can use that instance everywhere because they don't have state of their own. But before you do that, make sure that you are familiar with Singleton design pattern and many of its drawbacks. Now, whenever you want to add a new product type, for example, in our hands-on example, we used instance and storage, let's say we want to provide or provision a VPC then that requires changes to our abstract factory and all implementations of that abstract factory. So you should be familiar that adding a new product type requires changes to all these factory classes. In Abstract Factory, we typically provide the client code with the instance of concrete factory. So we pass on the object of one of the concrete factories and then the client code can use that factory object.

A black and white text

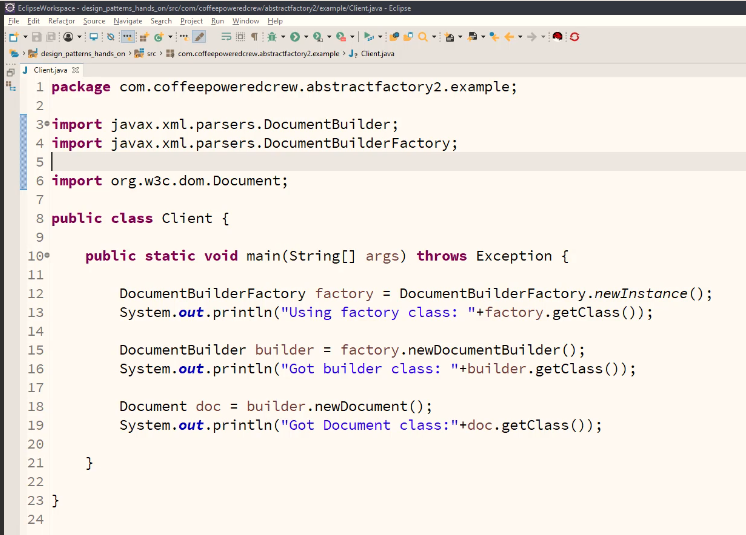
Description automatically generated

Now let's look at some of the design considerations. So whenever we have a situation that we want to constrain which type of objects are created in our application and the reason for that constraint may be that these objects can work together or they form a family or set of objects. In that case, you can use the abstract factory design pattern. Now Abstract Factory itself uses factory method design pattern. So the methods that we provided, in our hands-on example, like createInstance() and createStorage() inside our abstract factory, these are examples of factory methods which are implemented in subclasses. If the objects that we are creating in our concrete factory, if those are expensive, then we can transparently switch factory method implementations to use prototype design pattern to creates these objects. So we can use a prototype or we can use a singleton inside our abstract factory design pattern. So these are some of the points that you should remember when using the abstract factory.



1. Abstract Factory – Real world examples:

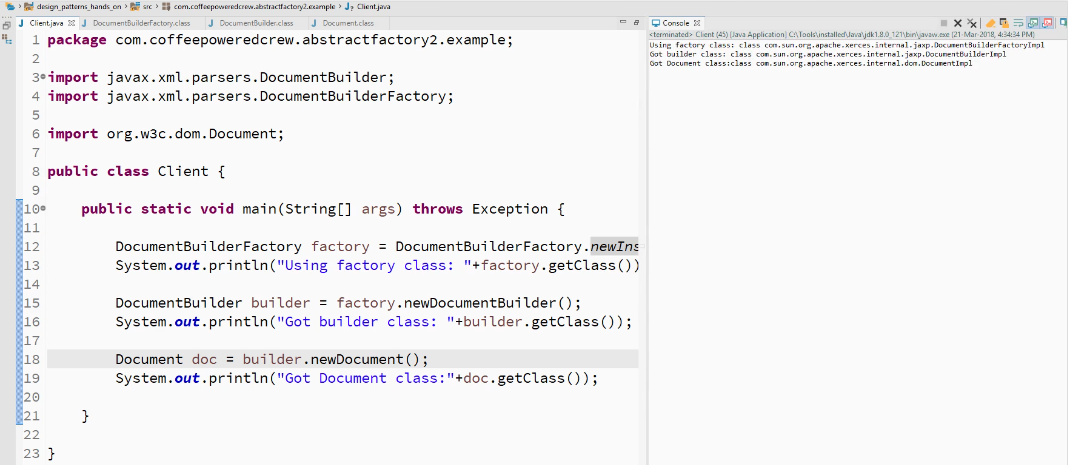
Now let's look at some real-world examples of abstract factory design pattern. So the DocumentBuilderFactory, which is part of Java API for XML, is a good example of abstract factory design pattern. However, there is one issue with this example is that this class implementation doesn't match 100% with the UML diagram that we have seen previously. Because this class has a static method called as newInstance() method, and that method returns the actual factory object that is used for creating different products(different set of objects). So maybe you are wondering if this is a static method that means it is going to have some static definition or static code which determines that these are the possible examples of factory, but that is not the case. The newInstance() method uses classpath scanning, it uses system properties and it will also check for a particular property file in order to find the actual implementation of factory. And using that information, it will instantiate that particular class and return us. So in this way we have the ability to change the actual factory that is used by our client code. And maybe, I'm sure that you are a little bit confused. So what we are going to do is we're going to jump into Java code and see what this DocumentBuilderFactory is so that you will understand how it is an example of Abstract Factory. So we have a simple class here called as Client and we have a main() method. We are only using the classes which are given to us by the Java API. We are not using any custom classes in this example. So if you look at the code, you will see that we are first going to call the newInstance() method on the DocumentBuilderFactory.



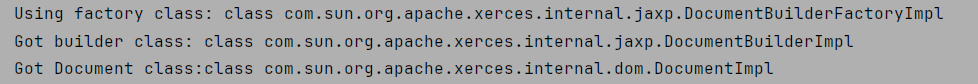
A screenshot of a computer

Description automatically generated

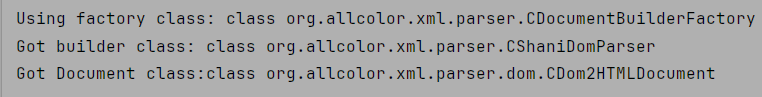
If we go into the source code of newInstance() method, you will see that this is a static method. And this is what I was talking about when we when I said that newInstance() is a static method. However, we still can change the actual factory that is being used. So DocumentBuilderFactory.newInstance() is going to return us a factory object. And using that factory object we can build a DocumentBuilder and using that DocumentBuilder we can create a Document. So DocumentBuilderFactory is an abstract class. The DocumentBuilder that we get from the factory object is also an abstract class, and the Document is actually an interface which is created by the DocumentBuilder. So right now, in my Client class, what I'm doing is I'm simply going to print the actual class that is returned to us by these methods. Okay, So we are not actually populating any data. We are only checking which objects are given to us as a result of calling these methods. So let's run our code as Java application and you will see that we have an output here.



**Output**:

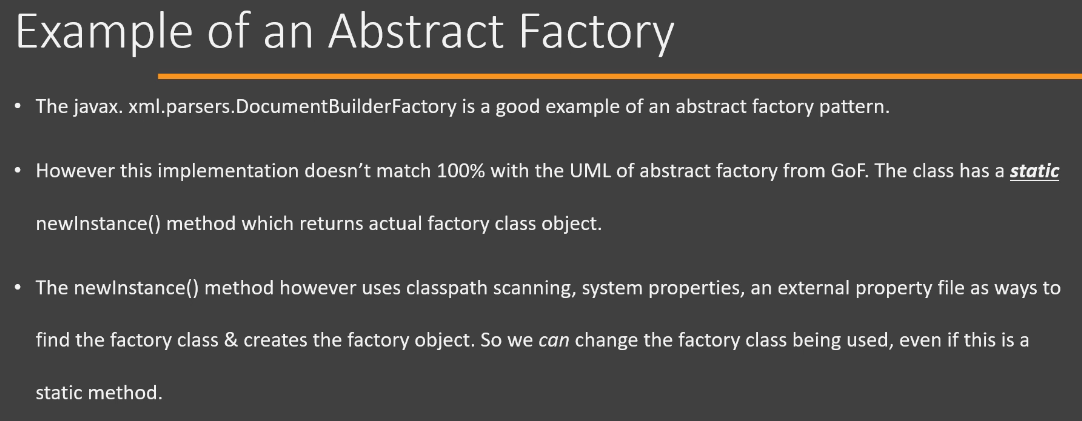


You will see that for our factory class we are getting DocumentBuilderFactoryImpl from the jaxb package. So this is the implementation of the DocumentBuilderFactory that comes with JDK. And since we are using this factory(i.e., DocumentBuilderFactoryImpl), we are going to get a DocumentBuilderImpl from the same package and then we are going to get a DocumentImpl from dom sub-package. So these are all implementations which are provided by default in Java class library. Now if we want to use a different factory, because remember in Abstract Factory design pattern we can change the factory class and we will get a different set of classes which work together. So here, DocumentBuilderImpl and DocumentImpl are the classes which work together. So they form a set or a kit. To change the factory, we have to modify the classpath of this example. Now, how to change the classpath in Eclipse? So I'm going to go to the Run Configuration Window by clicking on the Run dropdown in the upper panel and then click on Run Configuration. In the Classpath, in the User Entries(click on it), I'm going to add some jar files by clicking on Add JARs. So I'm going to use a third-party library called as 'Shani Parser' and I'm going to add these(jaxen-1.1.1-patched-shani-1.4.17.jar, shani-parser-1.4.17.jar or shani-parser-1.4.17-patched-yahp-1.3.jar, shani-xml-apis-1.4.17.jar) three jar files in my classpath. So I have simply added these three jar files in my runtime classpath and I'm going to say Apply and Run. Just remember the classes which we got before we configured the classpath. And if I run the same code again, you will see that I get a different factory.



How to change the classpath(or add JARs to your project) in IntelliJ? To add a JAR file to your project in IntelliJ IDEA, navigate to your Project Structure(File > Project Structure), go to "Modules" and then "Dependencies", where you can use the "+" button to add the JAR file directly from your file system; this effectively adds the JAR to your classpath, making its classes accessible within your project.

So this is how we can change the factory that is returned to us by the DocumentBuilderFactory.newInstance() method. And since this is an example of abstract factory design pattern, once we get a different implementation of factory(i.e., DocumentBuilderFactory in our case), whenever we create an object using that factory, we are going to get a different product(different set of objects). So, the DocumentBuilderFactory in Java API for XML is an example of abstract factory design pattern. Now, in the DocumentBuilderFactory, the way in which we configure a different factory for our client to use is slightly different. We use the classpath, or we can set a system property in order to configure our client with a particular implementation of factory class. But even so, this is an example of Abstract Factory.



1. Abstract Factory – Comparison with Factory Method:

One of the main differences between Factory and Abstract Factory patterns is the level of abstraction. The Factory pattern deals with creating objects of a single type, while the Abstract Factory pattern deals with creating objects of related types. The Factory pattern is simpler and more flexible, but the Abstract Factory pattern is more robust and consistent. Another difference is the number of classes involved. The Factory pattern usually has one Factory class and one interface for the products, while the Abstract Factory pattern has one Abstract Factory interface, multiple concrete Factory classes, and multiple interfaces for the products. The Factory pattern is easier to implement and maintain, but the Abstract Factory pattern is more scalable and extensible.

The Abstract Factory Pattern is like a factory of factories. It helps you create families of related objects without specifying their concrete classes. The Abstract Factory Pattern helps you create families of related objects without specifying their concrete classes.

To put it in a simple way, Factory is a factory of several different objects while an Abstract Factory is a factory of several different factories. You create factories to manage/create objects easier. I give the example of some shapes and colors while teaching people about it. Think about Circle, Triangle and Square shapes. You need a factory to create those objects. Then think about Red, Green and Blue colors. You need another factory to create those objects. Now, to be able to create those different factories (ShapeFactory, ColorFactory) with a proper pattern, you need another factory, which is called "AbstractFactory".

Abstract Factory pattern specializes in working with product families. Factory Method will work with only a single product hierarchy.

A screen shot of a computer screen

Description automatically generated

A black and white chart with white text

Description automatically generated

1. Abstract Factory – Pitfalls:

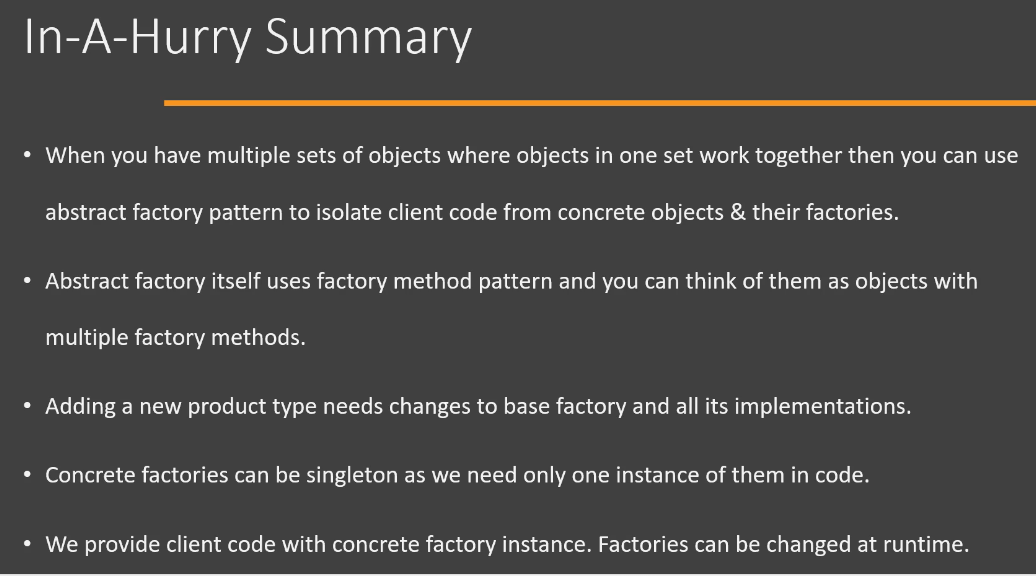
Let's look at the pitfalls of abstract factory design pattern. One thing you will notice that it is a lot more complex to implement an abstract factory than the factory method or any other creational design pattern. Next, Whenever we have a requirement to add a new product to our family of products. For example, in our hands-on example, we had a compute and a cloud storage resource. Let's say we wanted to add a network, private network or a virtual network then we need to modify all of our factory implementations as well as the base class. So this is a drawback of using abstract Factory. Now, typically it is quite difficult to visualize the need for Abstract factory when you are just starting your applications design. So this pattern typically starts out as a simple factory design pattern and then in the development, you realize that we can use the abstract factory design pattern. Now, Abstract factory design pattern, as you can see, is very specific to a problem and that problem is having the product families or having objects that work together. So it has a very specific application and typically you will not find it to be generalized to solve other problems. So these are the pitfalls of abstract factory design pattern.

A screenshot of a computer

Description automatically generated

1. Abstract Factory – Summary:

Let's look at a summary of abstract factory design pattern. So whenever we have a set of objects that work together and they form a family or set, then we can use an abstract factory to create those sets of objects. So our client code doesn't have to worry about whether it is creating the correct object or not. Our abstract factory implementations take care of creating only those objects that belong to one set or one family of products. Now Abstract Factory itself uses factory method design pattern, and you can think of Abstract Factory as a class which has multiple factory methods defined in it. Adding a new product to our existing implementation of Abstract Factory requires changes to the base class or our interface of the abstract factory as well as all of the implementations of that interface. Now concrete factories can be singleton, and the reason for that is because we are going to need only one instance of an abstract factory, and typically they don't have any state within them. We provide the client code with the concrete factory instance. And, at one point, that you might have noticed that we can change factories at runtime, so we can initialize our client with one instance or one implementation of our factory and at runtime based upon some criteria, we can switch that factory to another different instance and our client can start creating objects from a different product family.



Now let's look at the UML diagram of Abstract Factory. So we have an Instance interface here which is an example of an abstract product class. And then we have a Storage interface which is another abstract product class. We have two implementations, for Instance, Ec2Instance as well as GoogleComputeEngineInstance, for storage, Again, we have two implementations. The important point to note here is that Ec2Instance, which is a child class of Instance and S3Storage, which is a child class of Storage. These two classes or objects of these two classes work together. So these objects form one set and GoogleComputeEngineInstance and GoogleCloudStorage form another family or another set of classes or objects that work together. Then we have ResourceFactory, which is our abstract factory. And this abstract class or interface(i.e., ResourceFactory) defines factory methods to create objects of Instance and Storage. We implement our abstract factory in one or more concrete classes. So here we have AwsResourceFactory class, which provides implementations for the factory methods that are defined in our abstract factory. And these concrete implementations of factory take care that they are creating objects that belong to one single family. So AwsResourceFactory will create object of Ec2Instance class and it will, when we call createStorage() on our resource factory, create object of S3Storage. So the logic of making sure that the objects that are being created belong to same family handled in our concrete implementation of Abstract Factory.



When we say "product family" or "kit" in context of Abstract Factory design pattern, what are we referring to?

A product family has classes whose objects work with each other to achieve some functionality. These classes can belong to different inheritance hierarchy.