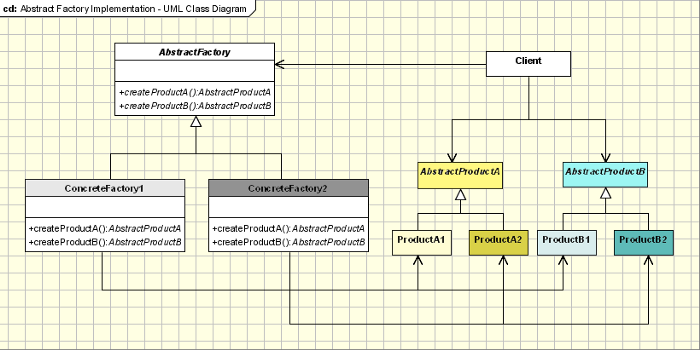
# Abstract factory Design Pattern

# Abstract Factory

**Definition**: A utility class that creates an instance of several families of classes. It can also return a factory for a certain group. The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. The Abstract Factory pattern is very similar to the Factory Method pattern. The main difference between the two is that **with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation**.

**Problem**: We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code, we do not know what class should be instantiated.

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



**Participates**:

* **AbstractFactory**: Declares an interface for operations that create abstract product objects
* **ConcreteFactory**: Implements the operations to create concrete product objects
* **AbstractProduct**: Declares an interface for a type of product object
* **ConcreteProduct**: a. Defines a product object to be created by the corresponding concrete factory, b. Implements the AbstractProduct interface
* **Client**: Uses only interfaces declared by AbstractFactory and AbstractProduct classes

C++ Code:

* Shape (AbstructProduct): Circle, Square, Ellipse, Rectangle
* Factory (AbstructFactory): SimpleShapeFactory (create Circle and Square, RobustShapeFactory (create Ellipse and Rectangle)
* Client (main function): Use Factory and Shape. Shape\* shapes[3]; shapes[0] = factory->createCurvedInstance();

**#include <iostream.h>**

**class** **Shape** {

**public**:

Shape() {

id\_ = total\_++;

}

**virtual** **void** draw() = 0;

**protected**:

**int** id\_;

**static** **int** total\_;

};

**int** Shape::total\_ = 0;

**class** **Circle** : **public** Shape {

**public**:

**void** draw() {

cout << "circle " << id\_ << ": draw" << endl;

}

};

**class** **Square** : **public** Shape {

**public**:

**void** draw() {

cout << "square " << id\_ << ": draw" << endl;

}

};

**class** **Ellipse** : **public** Shape {

**public**:

**void** draw() {

cout << "ellipse " << id\_ << ": draw" << endl;

}

};

**class** **Rectangle** : **public** Shape {

**public**:

**void** draw() {

cout << "rectangle " << id\_ << ": draw" << endl;

}

};

**class** **Factory** {

**public**:

**virtual** Shape\* createCurvedInstance() = 0;

**virtual** Shape\* createStraightInstance() = 0;

};

**class** **SimpleShapeFactory** : **public** Factory {

**public**:

Shape\* createCurvedInstance() {

**return** **new** Circle;

}

Shape\* createStraightInstance() {

**return** **new** Square;

}

};

**class** **RobustShapeFactory** : **public** Factory {

**public**:

Shape\* createCurvedInstance() {

**return** **new** Ellipse;

}

Shape\* createStraightInstance() {

**return** **new** Rectangle;

}

};

**int** **main**() {

**#ifdef SIMPLE**

Factory\* factory = **new** SimpleShapeFactory;

**#elif ROBUST**

Factory\* factory = **new** RobustShapeFactory;

**#endif**

Shape\* shapes[3];

shapes[0] = factory->createCurvedInstance(); // shapes[0] = new Ellipse;

shapes[1] = factory->createStraightInstance(); // shapes[1] = new Rectangle;

shapes[2] = factory->createCurvedInstance(); // shapes[2] = new Ellipse;

**for** (**int** i=0; i < 3; i++) {

shapes[i]->draw();

}

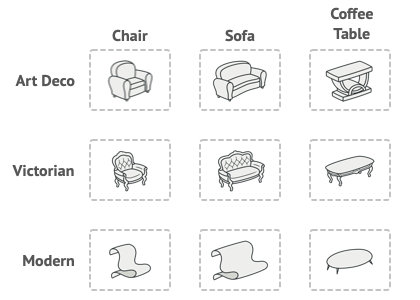
}

Details explanation:

Problem

Imagine that you’re creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say: Chair + Sofa + CoffeeTable.
2. Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.

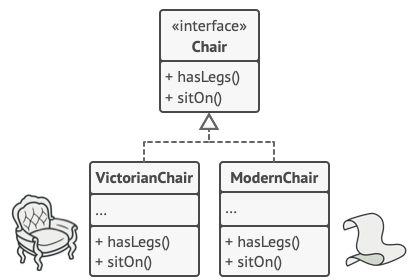


You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

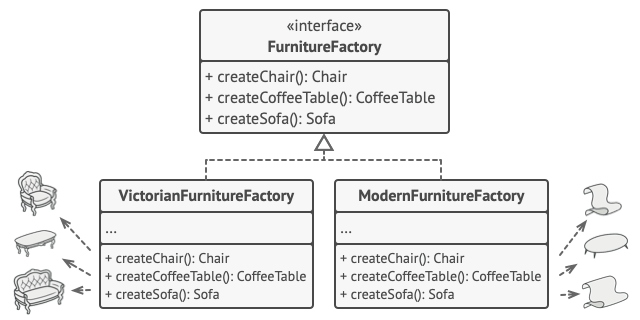
Also, you don’t want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn’t want to change the core code each time it happens.

# Solution

The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.

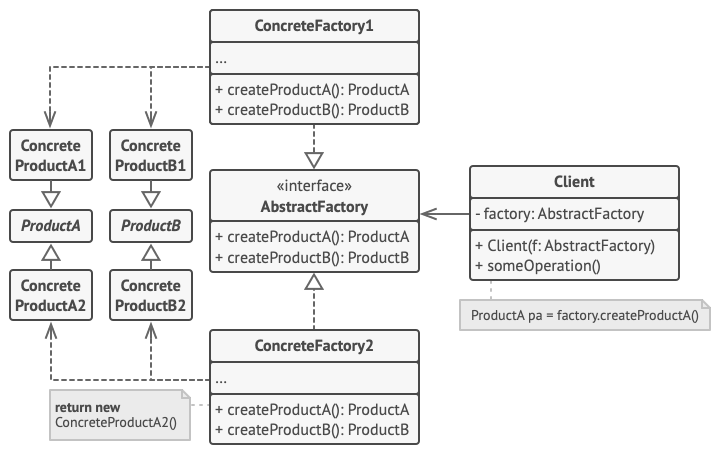


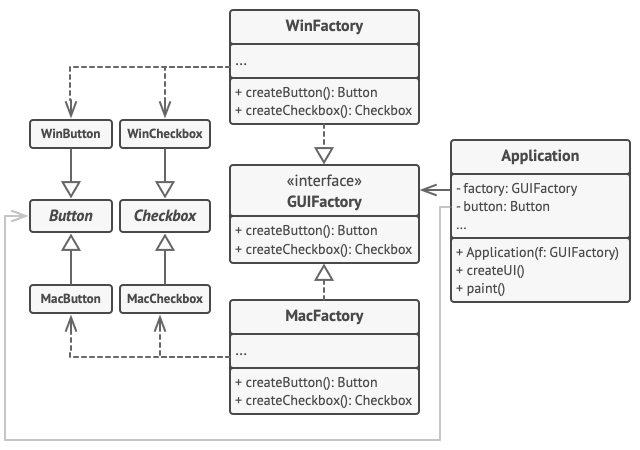
The next move is to declare the Abstract Factory—an interface with a list of creation methods for all products that are part of the product family (for example, createChair, createSofa and createCoffeeTable). These methods must return **abstract** product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on.



For each variant of a product family, we create a separate factory class based on the AbstractFactory interface. A factory is a class that returns products of a particular kind. For example, the ModernFurnitureFactory can only create ModernChair, ModernSofa and ModernCoffeeTable objects.

The client code has to work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code





# Relations with Other Patterns

* Many designs start by using [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) (less complicated and more customizable via subclasses) and evolve toward [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory), [**Prototype**](https://refactoring.guru/design-patterns/prototype), or [**Builder**](https://refactoring.guru/design-patterns/builder) (more flexible, but more complicated).
* [**Builder**](https://refactoring.guru/design-patterns/builder) focuses on constructing complex objects step by step. [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.
* [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) classes are often based on a set of [**Factory Methods**](https://refactoring.guru/design-patterns/factory-method), but you can also use [**Prototype**](https://refactoring.guru/design-patterns/prototype) to compose the methods on these classes.
* [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) can serve as an alternative to [**Facade**](https://refactoring.guru/design-patterns/facade) when you only want to hide the way the subsystem objects are created from the client code.
* You can use [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) along with [**Bridge**](https://refactoring.guru/design-patterns/bridge). This pairing is useful when some abstractions defined by Bridge can only work with specific implementations. In this case, Abstract Factory can encapsulate these relations and hide the complexity from the client code.
* [**Abstract Factories**](https://refactoring.guru/design-patterns/abstract-factory), [**Builders**](https://refactoring.guru/design-patterns/builder) and [**Prototypes**](https://refactoring.guru/design-patterns/prototype) can all be implemented as [**Singletons**](https://refactoring.guru/design-patterns/singleton).

A good example in the Java space is the JAXP API. JAXP stands for Java API for xml parsing. Using this API we can read or write and update the elements in an xml file the key class in this API is the document class that represents an xml document in memory to create a document class.

We use that document builder. So this document builder is a factory class and there is one more class document. Build a factory which is responsible for creating the document builder itself. So the document builder factory is an abstract factory because it is a factory of factories and the use case you are going to work on to give you

Another example is a DAO factory. DAO stands for data access object. We’ll learn more about it in sections later on. It simply is a class that can read write create update data we can have different types of DAO'sDAO's that deal with xml data and DAO's that deal with DB data and within xml we can have employee information department information.

Similarly within that database we can have employ information and department information. So you can see that we can have a factory to deal with these separate DOA's we can have a DB DAO factory that can give us one of these classes here when our application needs them and we can have a xml DAO factory which can give one of the classes here.

Now to get one of these factories themselves these factories will first implement a DAO abstract factory or they will extend DAO abstract factory class and we will have a producer which is responsible for creating one of these factories so abstract factory is a factory off factories.

It simply creates the factory we need. When we have multiple factories we see in our application.

# UML

