**NHÓM 10 – NHN**

|  |  |
| --- | --- |
| Huỳnh Đại Nhân | 18125013 |
| Phạm Quốc Huy | 18125008 |
| Mai Hoàng Nhi | 18125015 |

**ABSTRACT FACTORY**

1. **DEFINITION**

As Factory Method Pattern, Abstract Factory Pattern is one of Creational patterns.

Considered as another layer of abstraction over factory pattern.

Work around a super factory to create other factories.

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes.

1. **PROBLEM**

Imagine that you’re drawing many shapes. Your code consists of classes that represent:

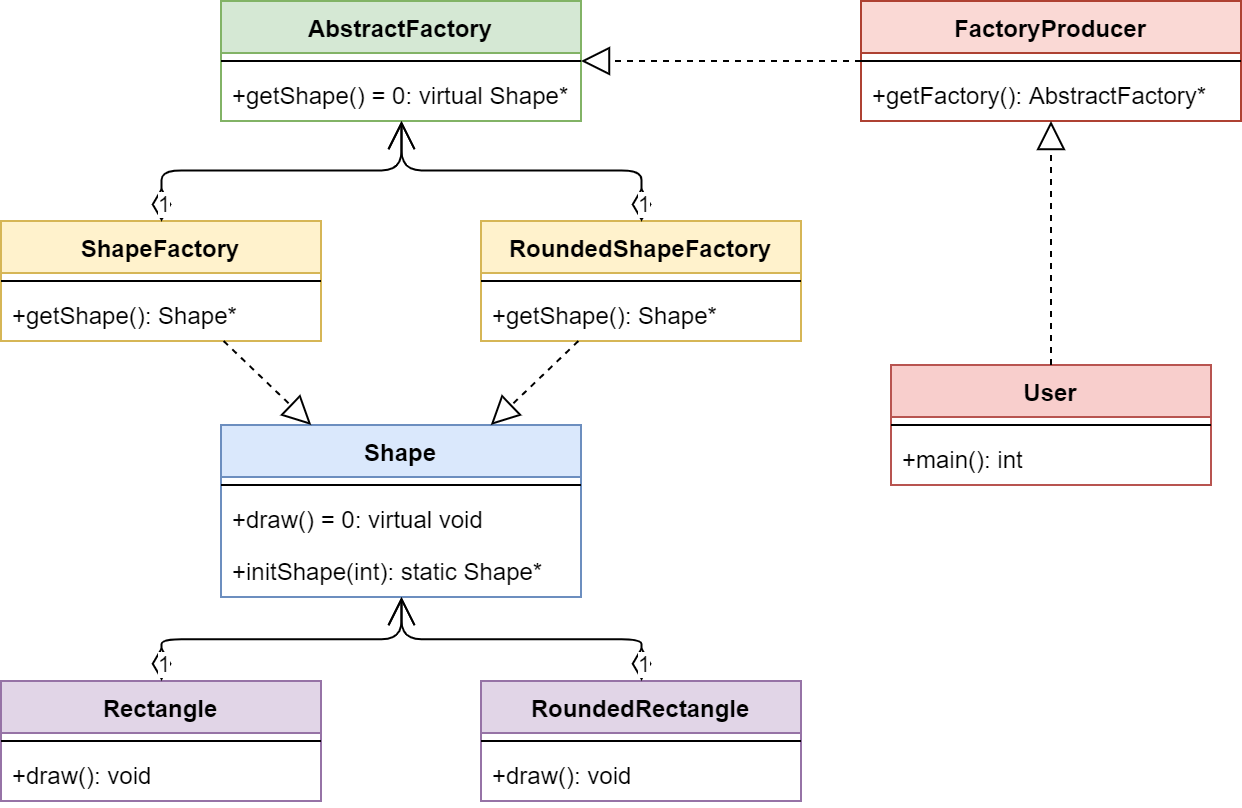
1. Shape, RoundedShape

2. Shape and RoundedShape are also have Square, Rectangle,... You need to creat individual objects so that they match other objects of the same type. Also, you don't want to change existing code when adding new shape or type of shape

1. **SOLVE**
2. **Normal**

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <string>  using namespace std;  class Shape  {  private:  public:  void draw();  };  class RoundedRectangle : public Shape  {  private:  public:  void draw();  };  class Rectangle : public Shape  {  private:  public:  void draw();  };  void Shape::draw()  {  #ifdef RECTANGLE  Shape \*shape = new Rectangle;  #else  Shape \*shape = new RoundedRectangle;  #endif  shape->draw();  }  void RoundedRectangle::draw()  {  cout << "Rounded Rectangle" << endl;  }  void Rectangle::draw()  {  cout << "Rectangle" << endl;  }  int main()  {  Shape \*shape = new Shape();  shape->draw();  return 0;  } |

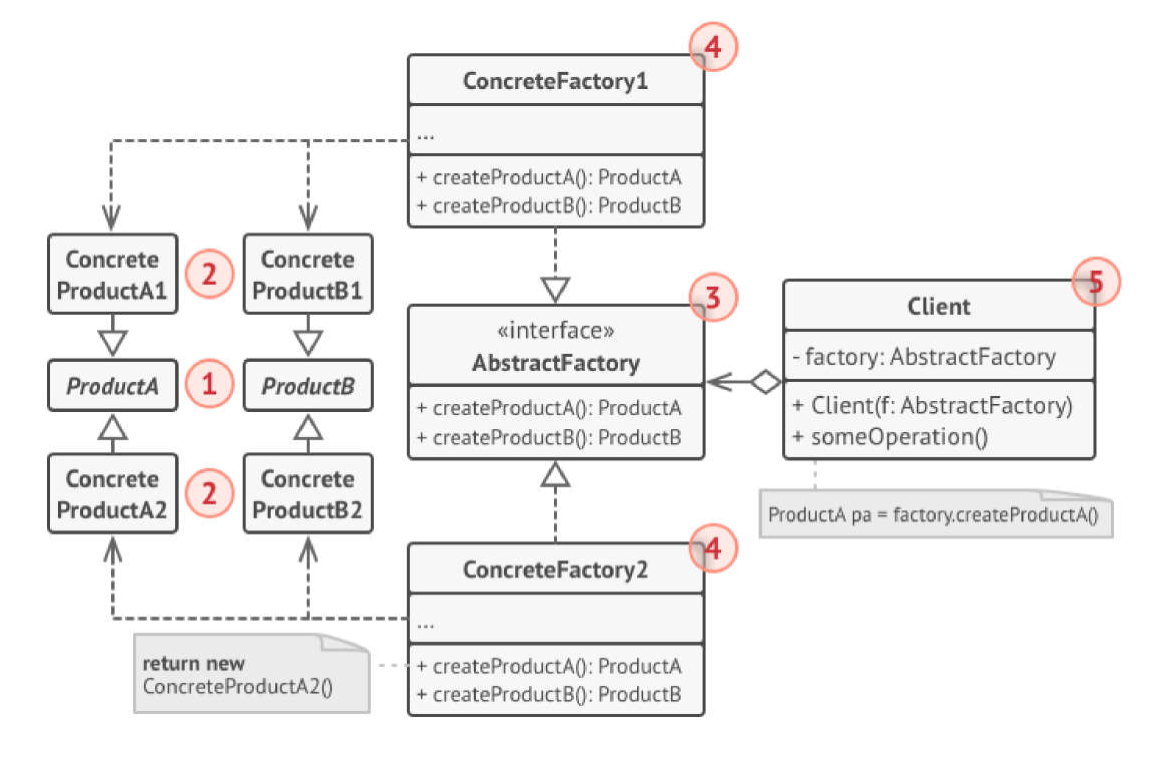
1. **Abstract Factory**



|  |
| --- |
| #include <iostream>  #include <fstream>  #include <string>  #include <vector>  using namespace std;  class AbstractFactory  {  private:  public:  virtual Shape \*getShape() = 0;  };  class FactoryProducer  {  private:  public:  AbstractFactory \*getFactory();  };  class Rectangle : public Shape  {  private:  public:  void draw();  };  class RoundedRectangle : public Shape  {  private:  public:  void draw();  };  class RoundedShapeFactory : public AbstractFactory  {  private:  public:  Shape \*getShape();  };  class Shape  {  private:  public:  static Shape\* initShape(int);  virtual void draw() = 0;  };  class ShapeFactory : public AbstractFactory  {  private:  public:  Shape \*getShape();  };  AbstractFactory \*FactoryProducer::getFactory()  {  int type;  do  {  cout << "1: Rectangle, 2: Rounded Rectangle" << endl;  cin >> type;  } while (type < 1 || type > 2);  if (type == 1)  return new ShapeFactory;  else if (type == 2)  return new RoundedShapeFactory;  return nullptr;  }  void RoundedRectangle::draw()  {  cout << "Rounded Rectangle" << endl;  }  Shape \*Shape::initShape(int type)  {  if (type == 1)  return new Rectangle;  else if (type == 2)  return new RoundedRectangle;  else  return nullptr;  }  Shape \*ShapeFactory::getShape()  {  return Shape::initShape(1);  }  int main()  {  FactoryProducer FP;  AbstractFactory \*a = FP.getFactory();  Shape \*s1 = a->getShape();  s1->draw();  FactoryProducer FP1;  AbstractFactory \*a1 = FP1.getFactory();  Shape \*s2 = a1->getShape();  s2->draw();  return 0;  } |

1. **STRUCTURE**

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



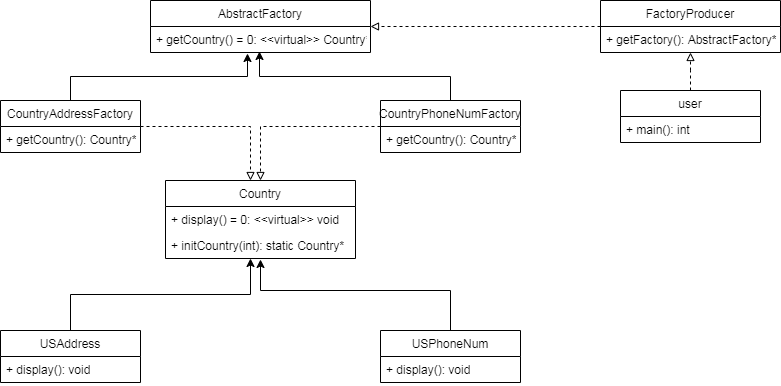
1. **ADVANTAGES & DISADVANTAGES**

|  |  |
| --- | --- |
| **ADVANTAGE** | **DISADVANTAGE** |
| * Ensure that all products you receive from a factory are compatible. * Cut down the dependence between creator and concrete products. * Create products into somewhere in the program 🡪 easy to track and manipulate. * Freely add many new products to the program without changing the existing code base. | * Code will be more complicated and must use a lot of classes. |

1. **SOME MORE PROBLEM/ EXAMPLE**

* **Phone Number Example**

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



* **Pizza Factory Example**

Another example, this time more simple and easier to understand, is the one of a pizza factory, which defines method names and returns types to make different kinds of pizza. The abstract factory can be named AbstractPizzaFactory, RomeConcretePizzaFactory and MilanConcretePizzaFactory being two extensions of the abstract class. The abstract factory will define types of toppings for pizza, like pepperoni, sausage or anchovy, and the concrete factories will implement only a set of the toppings, which are specific for the area and even if one topping is implemented in both concrete factories, the resulting pizzas will be different subclasses, each for the area it was implemented in.

1. **DISTINGUISH BETWEEN ABSTRACT FACTORY AND FACTORY METHOD**
2. **SAME**

* They are all factory pattern.
* Use to decrease the dependence of program on concrete setting (with 2 difference ways).
* Encapsulate the process creating object to make the program independent.

1. **DIFFERENCE**

|  |  |
| --- | --- |
| **FACTORY METHOD** | **ABSTRACT FACTORY** |
| * Create products, objects based on inheritance. If you want to use Factory Method to create objects, you have to extend a class and override create Factory Method function, so that Factory Method will create one object. * Idea: using sub-class to create expected object. It makes program independent with concrete types. * If you want to add more products to group of products, just need one method. * Factory Method use FactoryMethod function to create concrete product that user is expect, user just have to call function without kwowing what creates. | * Use objects to create products. * Create products, objects due to combining objects. * Idea : create abstract type to create another group of projects. Then, sub classes of abstract type will define way to create these products. To apply this idea, you need to create an instance of a sub class (this instance is a factory) and put it to necessary place in code. Therefore, as Factory Method, using factory in Abstract Factory will be independent on concrete products. * One disadvantage is all products in common will be grouped. So that, when you need to add a product to group of products that Abstract Factory can be create, user must change all sub classes (factories). * Potentiality to create products with different type. * Abstract Factory use many Factory Method functions in the Factory Method way to create objects inside own factories. Sub factories use Factory Method to create corresponding products. In this case, Factory Method is basicly used to create products. |

1. **Use Abstract Factory in which case? And when should use Factory Method?**

|  |  |
| --- | --- |
| **FACTORY METHOD** | **ABSTRACT FACTORY** |
| * When you need to create one product to make program independent with concrete class. * When you don’t know if you need which sub classes in the future. | * When you create many products at the same * When you want to ensure that don’t need to know concrete class when using. |

1. **REFERENCE**

<https://sourcemaking.com/design_patterns/abstract_factory>

<https://www.geeksforgeeks.org/abstract-factory-pattern/>

<https://www.oodesign.com/abstract-factory-pattern.html>

Book: Head First – Design Pattern