CS202 - Group 1:

Trương Như Quốc Thịnh – 18125027

Vũ Phương Anh – 18125061

Nguyễn Thành Phụng – 18125109

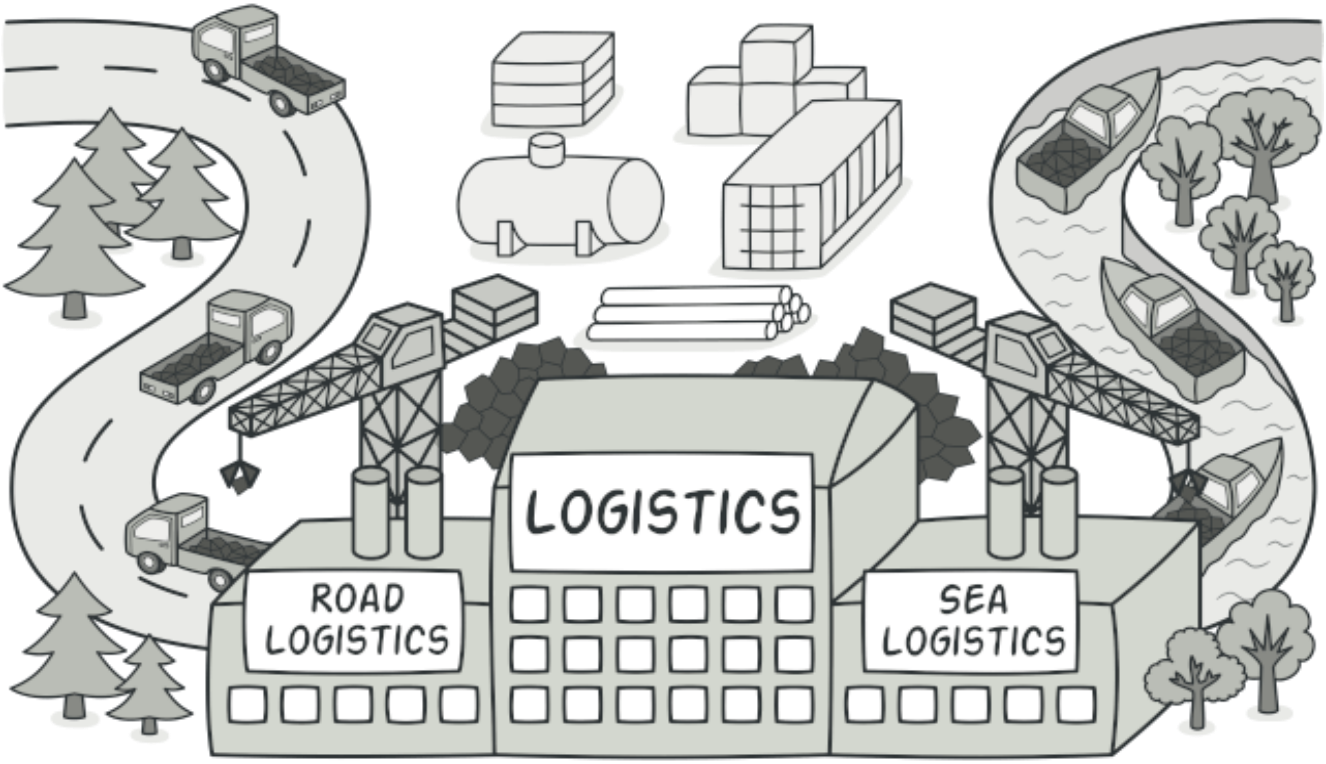
Trần Thiên Phúc – 18125137

Student Name	Student ID	% Contribution
Trương Như Quốc Thịnh	18125027	25%
Vũ Phương Anh	18125061	25%
Nguyễn Thành Phụng	18125109	25%
Trần Thiên Phúc	18125137	25%

SEMINAR REPORT

FACTORY METHOD

I. Problem:



- Without factory method:

- If users create a logistics management application.
  - The first version of app only handle transportation by trucks

→ major codes are inside the Truck class

- In case, users' app need to be updated like sea transporation companies to incorporate sea logistics
  - But adding a new class to the program isn't simple if the rest of the code is already coupled to existing classes

→After adding ships into the app would require making changes to the entire codebase.

- In another case, if users want to add more and more types of transportation to the app

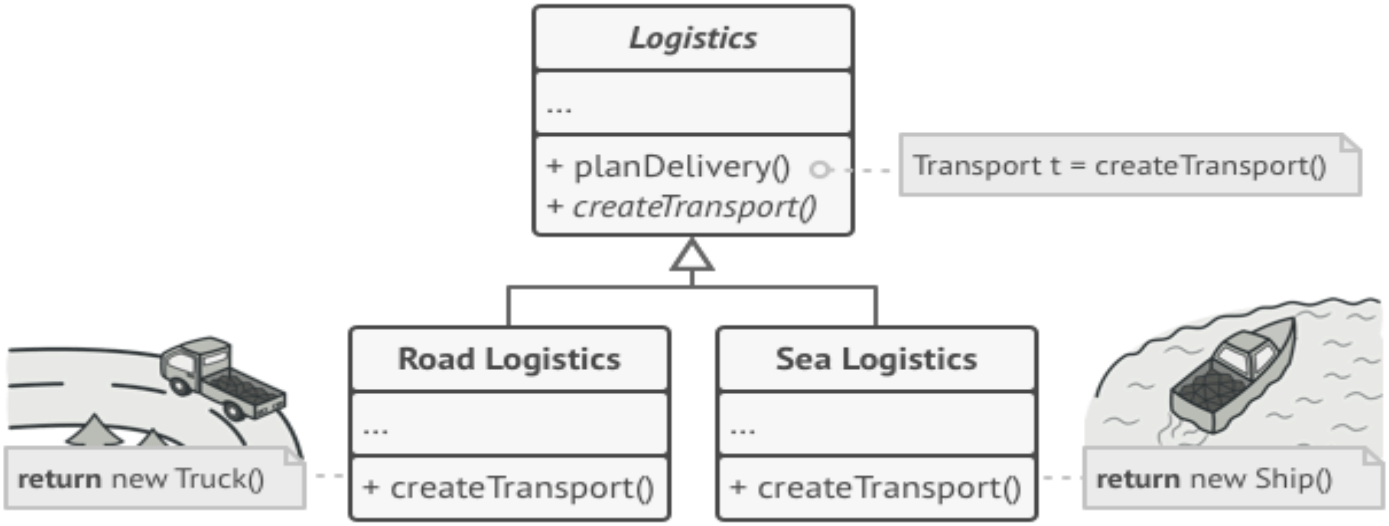
→ definitely need to make all of these changes again

❖ The disadvantage of this example is: users need to make to change a lot of code if users want to add more different class.

➢ <u>With factory method:</u>

- The Factory Method suggests that users replace direct object construction calls (using the new operator) with calls to a special factory method
  - The objects are still created via the new operator but it is called from within th e factory method.

- Objects returned by a factory are often referred to as “products”

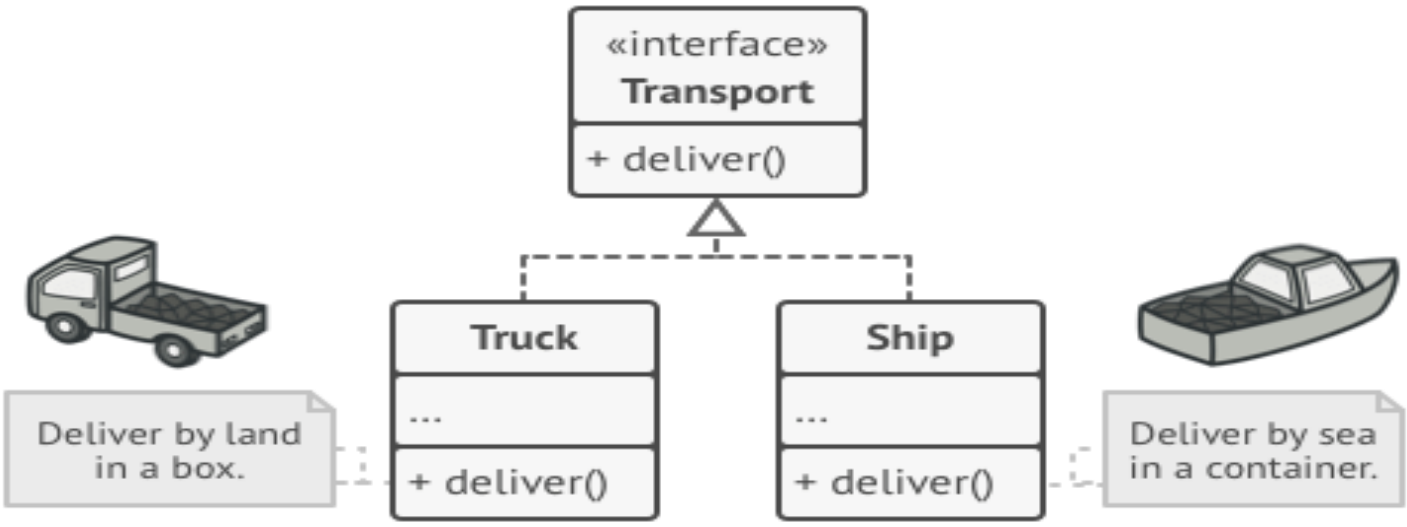


- We move the constructor call from one part of the program to another part

→ we can override the factory method in a subclass and change the class of products being created by the method

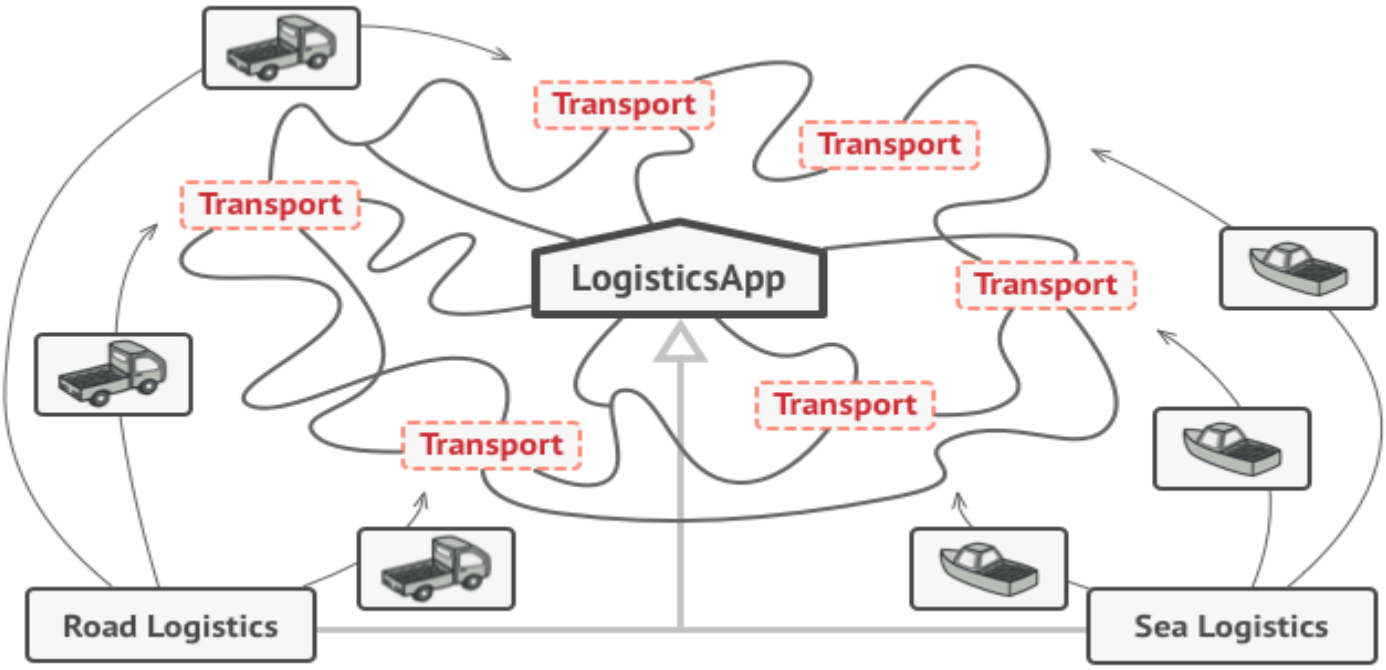
→ Subclasses can alter the class of objects being returned by the factory method.

- However:
  - Subclasses may return different types of products if these products have a common base class or interface
  - The factory method in the base class should have its return type declared as this interface



- In this instance, Truck and Ship classes should implement the Transportation interface declares a method called deliver.
    - Each class implements this method differently: Truck delivers cargo by land, Ship delivers cargo by sea.

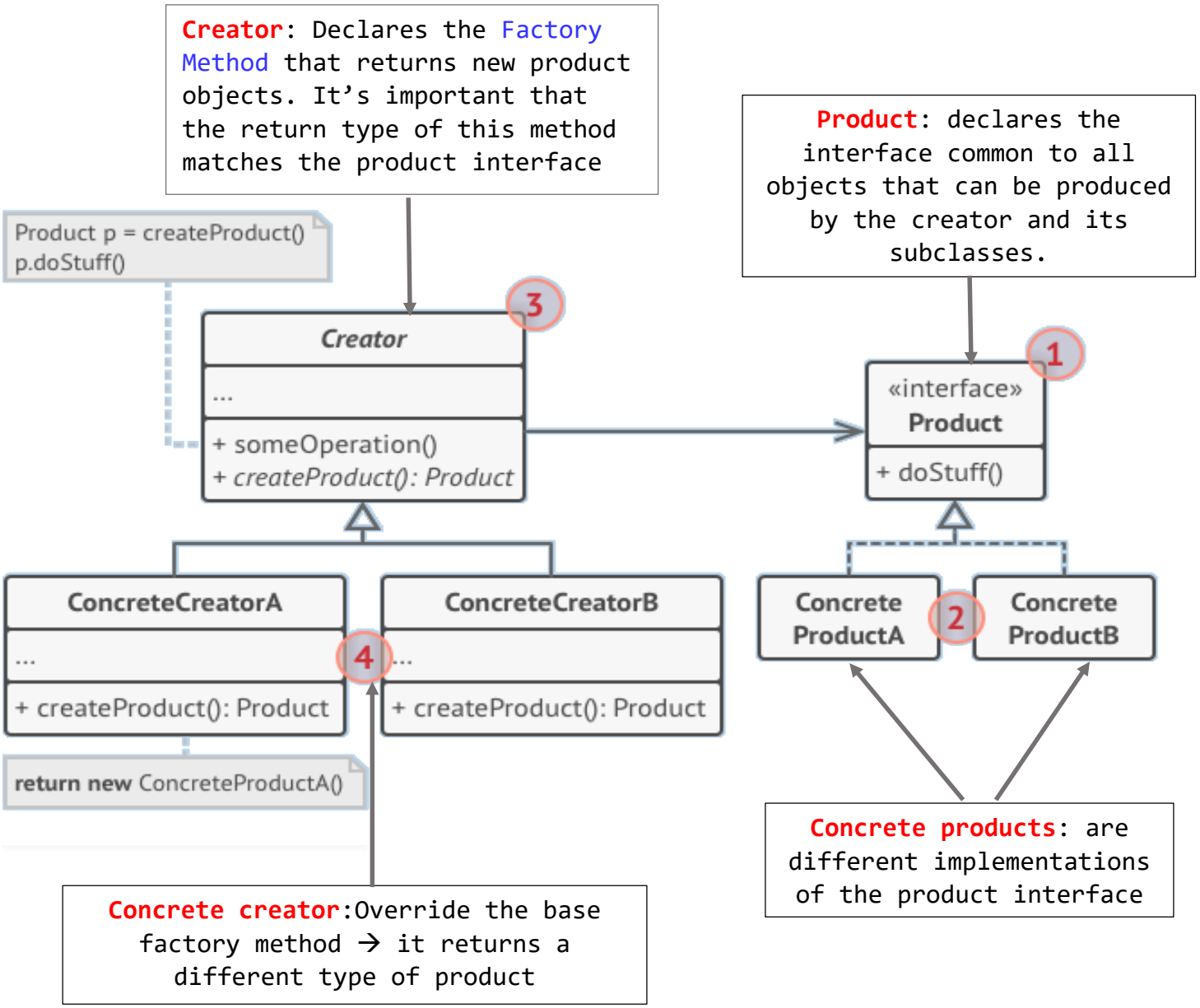
→The RoadLogistics class returns truck objects, the SeaLogistics class returns ships



- Code uses the Factory Method often called the client code which doesn't see a difference between the actual products returned by various subclasses
- The client:
    - treats all the products as abstract Transport.
    - knows that all transport objects are supposed to have the deliver method but how deliver works isn't important to the client.

→ Definition: Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

II. Structure:



III. How to implement

- Make all products follow the same interface → should declare methods that make sense in every product
- Add an empty Factory Method inside the creator class → return type of the method should match the common product interface
- In the creator's code find all references to product constructors. One by one, replace them with calls to the factory method, while extracting the product creation code into the factory method.

- Create a set of creator subclasses for each type of product listed in the Factory Method → Override the Factory Method in the subclasses and extract the appropriate bits of construction code from the base method
- If there are too many product types and it doesn't make sense to create subclasses for all of them, users can reuse the control parameter from the base class in subclasses.
- After all of the extractions, the base Factory Method has become empty, users can make it abstract. If there's something left, users can make it a default behavior of the method.

IV. Advantage and Disadvantage

Pros of Factory (Design) Pattern	Cons of Factory (Design) Pattern
+Loose coupling that helps in changing the appliction design more readily +The application is seperated from a family of classes +It makes the application more customizable	+Reduced readability due to increased abstraction +Applicable only for families of classes

V. Application:

1. When users don't know beforehand the exact types and dependencies of the objects users' code should work with

   →seperates product construction code from the code that actually uses the product

   →easier to extend the product construction code independently from the rest of the code
2. When users want to provide other users of users library or framework with a way to extend its internal components.

   → Inheritance is probably the easiest way to extend the default behavior of a library or framework.
3. When users want to save system resources by reusing existing objects instead of rebuilding them each time.

   → users often experience this need when dealing with large, resource-intensive objects such as database connections, file systems, and network resources.

VI. Relations with other patterns:

- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, but more complicated).
    - Abstract Factory classes are often based on a set of Factory Methods, but users can also use Prototype to compose the methods on these classes.
    - Iterator: users can use Factory Method along with Iterator to let collection subclasses return different types of iterators that are compatible with the collections.
    - Prototype isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, *Prototype* requires a complicated initialization of the cloned object. Factory Method is based on inheritance but doesn't require an initialization step.

- Factory Method is a specialization of Template Method. At the same time, a Factory Method may serve as a step in a large Template Method.

VII. SourceCode

```
#include<iostream>
using namespace std;

class LogisticsManagement
{
public:

   LogisticsManagement() { }
   virtual void deliver() = 0;
   virtual void printVehicle() = 0;
};
class RoadLogistics : public LogisticsManagement
{
public:
   RoadLogistics() { }
   void printVehicle() {
         cout << "This is a truck" << endl;
   }
   void deliver() { }
   ~RoadLogistics() { }
};

class SeaLogistics : public LogisticsManagement
{
public:
   SeaLogistics() { }
   void printVehicle() {
         cout << "This is a ship" << endl;
   }
   void deliver() { }
   ~SeaLogistics() { }
};

int main()
{
   RoadLogistics* truck = new RoadLogistics;
   truck->printVehicle();
   cout << endl;
   SeaLogistics* ship = new SeaLogistics;
   ship->printVehicle();
```

```
    cout << endl;
    return 0;
}
```

Output:

```
This is a truck

This is a ship
```

VIII. Another Examples:

Example 1 – Nguyễn Thành Phụng

```
#include<iostream>
using namespace std;
class PizzaFactor
{
public:
      virtual void printPizza() = 0;
};

class HamMushroomPizza : public PizzaFactory
{
public:
      HamMushroomPizza() { }
      void printPizza() { cout << "This is ham-mushroom pizza" << endl; }
      ~HamMushroomPizza() { }
};
class DeluxePizza : public PizzaFactory
{
public:
      DeluxePizza() { }
      void printPizza() { cout << "This is Deluxe pizza" << endl; }
      ~DeluxePizza() { }
};
class HawaiianPizza : public PizzaFactory
{
public:
      HawaiianPizza() { }
      void printPizza() { cout << "This is Hawaiian pizza" << endl; }
      ~HawaiianPizza() { }
};

int main()
{
      HamMushroomPizza* hammushroompizza = new HamMushroomPizza;
```

```
	hammushroompizza->printPizza();
	cout << endl;

	DeluxePizza* deluxepizza = new DeluxePizza;
	deluxepizza->printPizza();
	cout << endl;

	HawaiianPizza* hawaiianpizza = new HawaiianPizza;
	hawaiianpizza->printPizza();
	cout << endl;
	return 0;
}
```

Output:

```
This is ham-mushroom pizza

This is Deluxe pizza

This is Hawaiian pizza
```

Example 2 – Trương Như Quốc Thịnh

```
#include<iostream>
using namespace std;
class AppleFactory
{
public:
	AppleFactory() { }
	virtual void print() = 0;
	~AppleFactory() { }
};
class RedApple : public AppleFactory
{
public:
	RedApple() { }
	void print() { cout << "This is red apple" << endl; }
	~RedApple() { }
};
class GreenApple : public AppleFactory
{
public:
	GreenApple() { }
	void print() { cout << "This is green apple" << endl; }
	~GreenApple() { }
};
```

```
int main()
{
	RedApple* redapple = new RedApple;
	redapple->print();
	cout << endl;

	GreenApple* greenapple = new GreenApple;
	greenapple->print();
	cout << endl;
	return 0;
}
```

Output:

```
This is red apple

This is green apple
```

Example 3 – Trần Thiên Phúc

```
#include<iostream>
using namespace std;
class DessertFactory
{
public:
	DessertFactory() { }
	virtual void print() = 0;
	~DessertFactory() { }
};
class CupCake : public DessertFactory
{
public:
	CupCake() { }
	void print() { cout << "They are cupcake" << endl; }
	~CupCake() { }
};
class Cookies : public DessertFactory
{
public:
	Cookies() { }
	void print() { cout << "They are cookies" << endl; }
	~Cookies() { }
};

int main()
```

```
{
	CupCake* cupcake = new CupCake;
	cupcake->print();
	cout << endl;

	Cookies* cookies = new Cookies;
	cookies->print();
	cout << endl;
	return 0;
}
```

Output:

```
They are cupcake

They are cookies
```

Example 4 - Vũ Phương Anh

```
#include<iostream>
using namespace std;
enum TypeofAnimals
{
	Elephants, Tigers
};
class ZooFactory
{
public:

	ZooFactory() { }
	virtual void Eat() = 0;
	virtual void printAnimals() = 0;
};
class ElephantsFactory : public ZooFactory
{
public:
	ElephantsFactory() { }
	void printAnimals() {
		cout << "They are elephants" << endl;
	}
	void Eat() { cout << "Elephants eat sugar cane" << endl; }
	~ElephantsFactory() { }
};
class TigersFactory : public ZooFactory
```

```
{
public:
	TigersFactory() { }
	void printAnimals() {
		cout << "They are tigers" << endl;
	}
	void Eat() { cout << "Tigers eat meat" << endl; }
	~TigersFactory() { }
};

int main()
{
	ElephantsFactory* elephant = new ElephantsFactory;
	elephant->printAnimals();
	elephant->Eat();
	cout << endl;

	TigersFactory* tiger = new TigersFactory;
	tiger->printAnimals();
	tiger->Eat();


	return 0;
}
```

Output:

```
They are elephants

Elephants eat sugar cane


They are tigers

Tigers eat meat
```

ABSTRACT FACTORY

I. Problem:

- Without Abstract Factory:

+If users creating a furniture shop simulator → users will have 3 class represent:

+A family of related products: Chair + Sofa + CoffeeTable

+Several variants of this family (art deco, victorian, modern)

→Need a way to create individual furniture objects → they can match other objects of the same family

-More than that, users won't wanna change existing code when adding new products or families of products to the program.And furniture vendors update their catalogs very often and users, definitely, won't want to change users' core code each time changing.

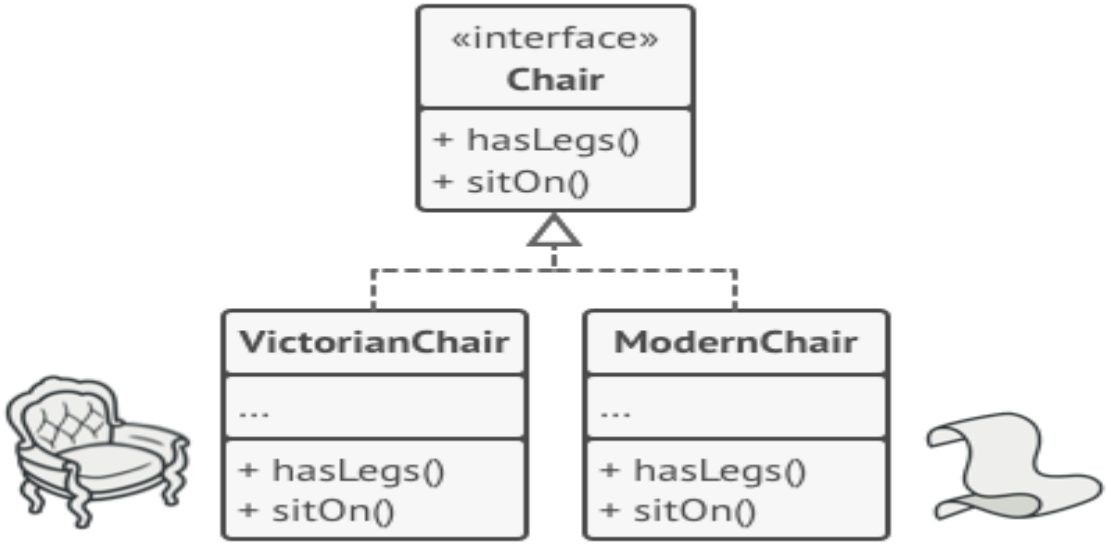
- ❖ The disadvantage of this example: The users have to change codebase when adding new classes.

- With abstract factory:

The Abstract Factory pattern suggests:

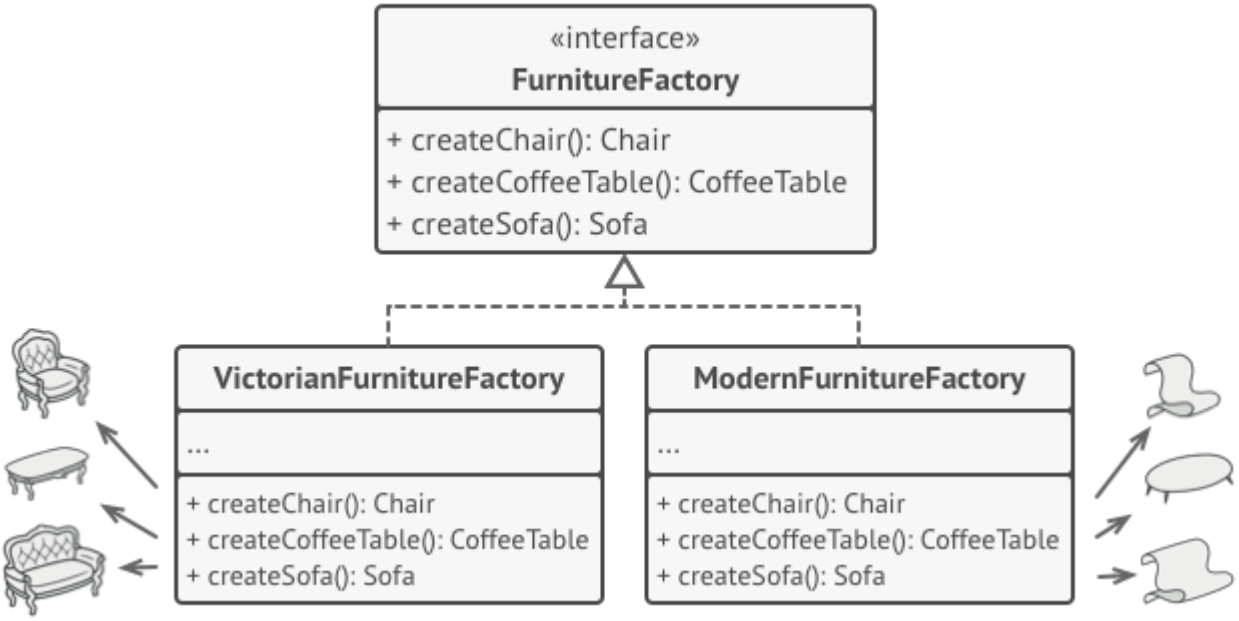
+explicitly declare interfaces for each distinct product (chair or sofa or coffeetable) of the product family

+make all variants of products follow those interface.



+declare the Abstract Factory – an interface with a list of creation methods (createChair or CreateSofa or createCoffeeTable) for all products that are part of product family

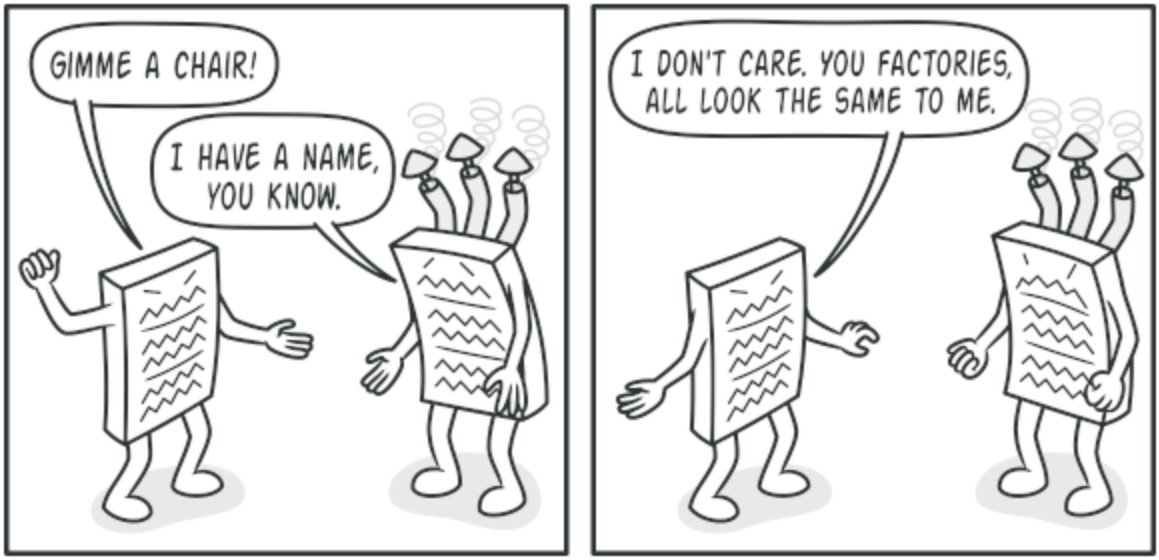
→Must return abstract product types represented by the interfaces (Chair or Sofa or CoffeeTable)



-Each concrete factory corressponds to a specific product variant

-For each variant of a product family → create a seperate factory class based on AbstractFactory interface (a factory is a class returns products of a particular kind)

+ Example: ArtDecoFurnitureFactory can only create ArtDecoChair, ArtDecoSofa, ArtDecoCoffeeTable objects



The clients don’t have to be aware of the factory’s class nor does it matter what kind of objects it gets → the clients must treat all chairs in the same manner using the abstract interface.

The application creates a concrete factory object at the initialization → app must select the factory type depending on the configuration or the environment settings.

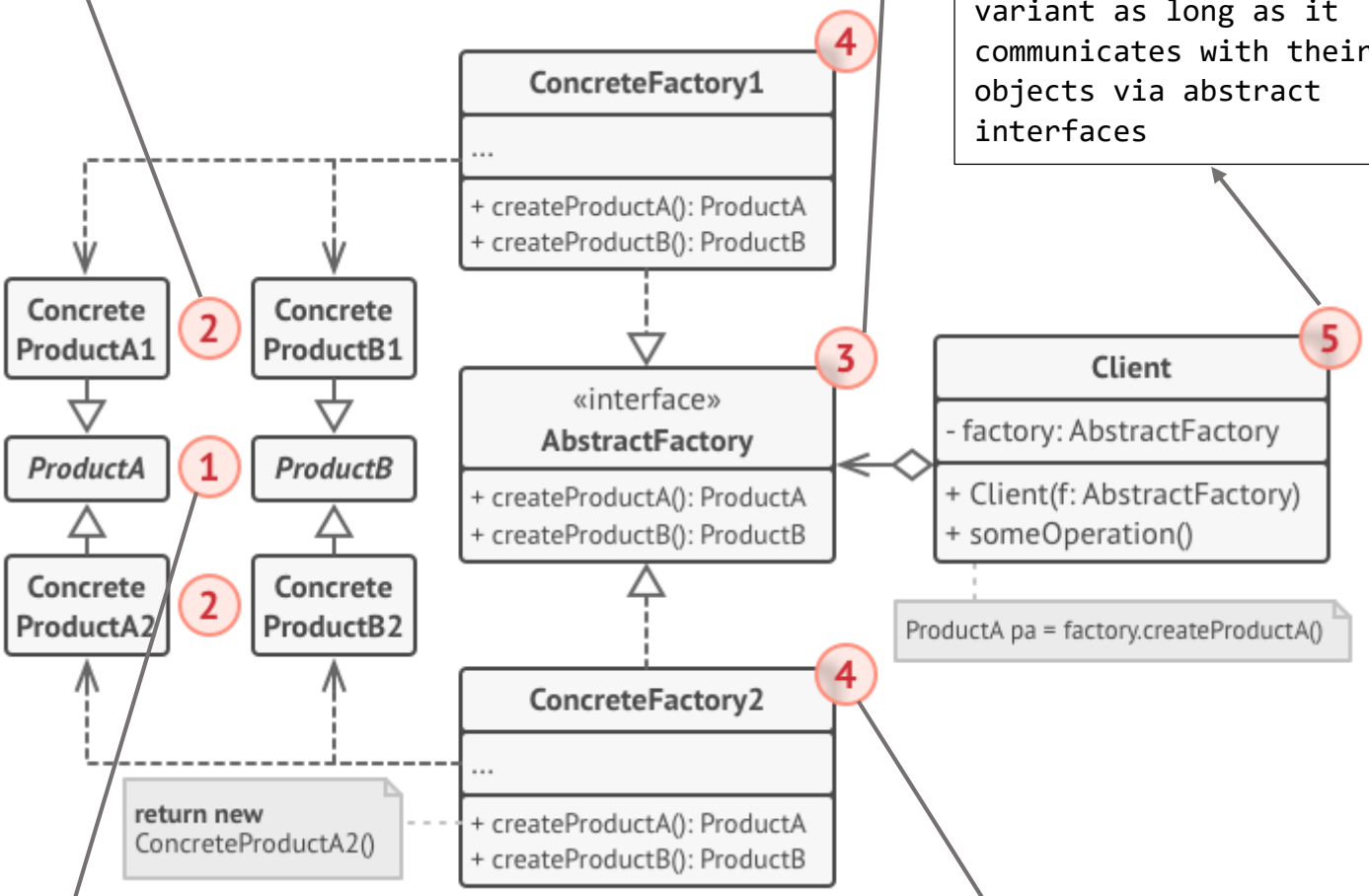
→Definition: An Abstract Factory is a creational design patterns let users product families of related objects without specifying their concrete classes

II. Structure

Concrete products: various implementations of abstract products. Each abstract product must be implement in all given variants

Abstract factory: declares a set of methods for creating each of the abstract products

Client: work with any concrete factory/product variant as long as it communicates with their objects via abstract interfaces



Abstract products: Declares interface for a set of distinct but related products which make up a product family

Concrete factories: implement creation methods of the abstract factory. Each concrete factory corressponds to a specific variant of products + creates those product variants

III. How to implement:

- Map out a matrix of distinct product types versus variants of these products
- Declare abstract product interfaces for all product types → make all concrete product classes implement these classes
- Declare the abstract factory interface with a set of creation methods for all abstract products
- Create factory initialization code in the app → instantiate one of the concrete factory classes depend on the application configuration or the current environement → pass factory object to all classes that construct products

1. Scan through the code and find all the direct calls to product constructors → replace them with calls to the approiate creationg method

IV. Applicability

- When the users' code needs to work with various familiies of relatied products. However, users don't want it to depend on the concrete classes of those products – they might be unknown beforehand or users simply want to allow for future extensibility

→Abstract Factory provides an interface for creating objects from each class of the product family

V. Advantage and Disadvantage:

Pros of Abstract Factory Pattern	Cons of Abstract Factory Pattern
• Can be sure that the products the users are getting from a factory are compatible with each other • Users avoid tight coupling between concrete products and client code • Single Responsibility Principle: the user can extract the product	• A lot of new interface and classes are introduced along with pattern → The code may become more complicated than it should be

creation code into 1 place only → making the code easier • Open/Closed Principle: The user can introduce new variants of products without breaking existing client code	

VI. Relations with Other patterns

- Factory Method: Many designs start by using Factory Method (less complicated and more customizable via subclasses) and envolve toward Abstract Factory, Prototype, or Builder (more flexible, but more complicated).
- Builder: focuses on constructing complex objects step by step. 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.
- Prototype: classes are often based on a set of Factory Methods, but you can also use Prototype to compose the methods on these classes.
- Facade: Abstract Factory can serve as an alternative to Facade when you only want to hide the way the subsystem objects are created from the client code.
- Bridge: You can use Abstract Factory along with 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, Builders and Prototypes can all be implemented as Singletons.

VII. Sourcecode

```
#include<iostream>
using namespace std;
class TypeofFurniture
{
```

```
class TypeofFurniture
public:
   virtual void sitOn() = 0;
   virtual void hasLeg() = 0;
};

class Chair : public TypeofFurniture
{
public:
   void sitOn() { cout << "There are 1 sit" << endl; }
   void hasLeg() { cout << "There are 2 legs" << endl; }
};
class Sofa : public TypeofFurniture
{
public:
   void sitOn() { cout << "There are 3 sits" << endl; }
   void hasLeg() { cout << "There are 4 legs" << endl; }
};
class CoffeeTable : public TypeofFurniture
{
public:
   void sitOn() { cout << "There are 6 sits" << endl; }
   void hasLeg() { cout << "There are 4 legs" << endl; }
};

class FurnitureFactory {
public:
   virtual Chair* createChair() = 0;
   virtual Sofa* createSofa() = 0;
   virtual CoffeeTable* createCoffeeTable() = 0;
};
class ArtDecoFurnitureFactory : public FurnitureFactory
{
public:
   Chair* createChair()
   {
         cout << "This is art deco chair" << endl;
         return new Chair;
   }
   Sofa* createSofa()
   {
         cout << "This is art deco sofa" << endl;
         return new Sofa;
   }
   CoffeeTable* createCoffeeTable()
```

```
    {
        cout << "This is art deco coffee table" << endl;
        return new CoffeeTable;
    }
};
class ModernFurnitureFactory : public FurnitureFactory
{
public:
    Chair* createChair()
    {
        cout << "This is mordern chair" << endl;
        return new Chair;
    }
    Sofa* createSofa()
    {
        cout << "This is mordern sofa" << endl;
        return new Sofa;
    }
    CoffeeTable* createCoffeeTable()
    {
        cout << "This is mordern coffee table" << endl;
        return new CoffeeTable;
    }
};
class VictorianFurnitureFactory : public FurnitureFactory
{
public:
    Chair* createChair()
    {
        cout << "This is victorian chair" << endl;
        return new Chair;
    }
    Sofa* createSofa()
    {
        cout << "This is victorian sofa" << endl;
        return new Sofa;
    }
    CoffeeTable* createCoffeeTable()
    {
        cout << "This is coffee table" << endl;
        return new CoffeeTable;
    }
};

int main()
```

```
{
    cout << "               Art Deco Furniture                " << endl;
    ArtDecoFurnitureFactory* furniture1 = new ArtDecoFurnitureFactory;
    furniture1->createChair()->hasLeg();
    furniture1->createChair()->sitOn();
    cout << endl;
    furniture1->createSofa()->hasLeg();
    furniture1->createSofa()->sitOn();
    cout << endl;
    furniture1->createCoffeeTable()->hasLeg();
    furniture1->createCoffeeTable()->sitOn();
    cout << endl;

    cout << "               Modern Furniture                " << endl;
    ModernFurnitureFactory* furniture2 = new ModernFurnitureFactory;
    furniture2->createChair()->hasLeg();
    furniture2->createChair()->sitOn();
    cout << endl;
    furniture2->createSofa()->hasLeg();
    furniture2->createSofa()->sitOn();
    cout << endl;
    furniture2->createCoffeeTable()->hasLeg();
    furniture2->createCoffeeTable()->sitOn();
    cout << endl;

    cout << "               Victorian Furniture                " << endl;
    VictorianFurnitureFactory* furniture3 = new VictorianFurnitureFactory;
    furniture3->createChair()->hasLeg();
    furniture3->createChair()->sitOn();
    cout << endl;
    furniture3->createSofa()->hasLeg();
    furniture3->createSofa()->sitOn();
    cout << endl;
    furniture3->createCoffeeTable()->hasLeg();
    furniture3->createCoffeeTable()->sitOn();
    cout << endl;
    cout << endl;

    return 0;
}
```

Output:

```
               Art Deco Furniture
This is art deco chair
There are 2 legs
```

```
This is art deco chair
There are 1 sit

This is art deco sofa
There are 4 legs
This is art deco sofa
There are 3 sits

This is art deco coffee table
There are 4 legs
This is art deco coffee table
There are 6 sits

              Modern Furniture
This is mordern chair
There are 2 legs
This is mordern chair
There are 1 sit

This is mordern sofa
There are 4 legs
This is mordern sofa
There are 3 sits

This is mordern coffee table
There are 4 legs
This is mordern coffee table
There are 6 sits

              Victorian Furniture
This is victorian chair
There are 2 legs
This is victorian chair
There are 1 sit

This is victorian sofa
There are 4 legs
This is victorian sofa
There are 3 sits

This is coffee table
There are 4 legs
This is coffee table
```

```
There are 6 sits
```

VIII. Another Examples:

Example 1 – Trương Như Quốc Thịnh

```
#include<iostream>
using namespace std;
class TypeofCake
{
public:
	TypeofCake() { }
	virtual void printCake() = 0;
	~TypeofCake() { }
};
class Souffle : public TypeofCake
{
public:
	Souffle() { };
	void printCake() { }
	~Souffle() { }
};
class Cupcake : public TypeofCake
{
public:
	Cupcake() { };
	void printCake() { }
	~Cupcake() { }
};
class FruitTart : public TypeofCake
{
public:
	FruitTart() { };
	void printCake() { }
	~FruitTart() { }
};
class CakeFactory
{
public:
	CakeFactory() { }
	virtual Souffle* createSouffle() = 0;
	virtual Cupcake* createCupcake() = 0;
	virtual FruitTart* createFruitTart() = 0;
	~CakeFactory() { }
```

```
};
class SmallCake : public CakeFactory
{
public:
	Souffle* createSouffle()
	{
		cout << "This is small souffle" << endl;
		return new Souffle;
	}
	Cupcake* createCupcake()
	{
		cout << "This is small cupcake" << endl;
		return new Cupcake;
	}
	FruitTart* createFruitTart()
	{
		cout << "This is small fruit tart" << endl;
		return new FruitTart;
	}
};
class MediumCake : public CakeFactory
{
public:
	Souffle* createSouffle()
	{
		cout << "This is medium souffle" << endl;
		return new Souffle;
	}
	Cupcake* createCupcake()
	{
		cout << "This is medium cupcake" << endl;
		return new Cupcake;
	}
	FruitTart* createFruitTart()
	{
		cout << "This is medium fruit tart" << endl;
		return new FruitTart;
	}
};
class BigCake : public CakeFactory
{
public:
	Souffle* createSouffle()
	{
		cout << "This is big souffle" << endl;
```

```
            return new Souffle;
      }
      Cupcake* createCupcake()
      {
            cout << "This is big cupcake" << endl;
            return new Cupcake;
      }
      FruitTart* createFruitTart()
      {
            cout << "This is big fruit tart" << endl;
            return new FruitTart;
      }
};
int main()
{
      cout << "                                    Small Cake" << endl;
      SmallCake* smallCake = new SmallCake;
      smallCake->createSouffle();
      smallCake->createCupcake();
      smallCake->createFruitTart();
      cout << endl;

      cout << "                                    Medium Cake" << endl;
      MediumCake* mediumCake = new MediumCake;
      mediumCake->createSouffle();
      mediumCake->createCupcake();
      mediumCake->createFruitTart();
      cout << endl;

      cout << "                                    Big Cake" << endl;
      BigCake* bigCake = new BigCake;
      bigCake->createSouffle();
      bigCake->createCupcake();
      bigCake->createFruitTart();
      cout << endl;

      return 0;
}
```

Output:

```
                              Small Cake
This is small souffle
This is small cupcake
This is small fruit tart
```

```
                                Medium Cake
This is medium souffle
This is medium cupcake
This is medium fruit tart

                                Big Cake
This is big souffle
This is big cupcake
This is big fruit tart
```

Example 2 – Trần Thiên Phúc

```
#include<iostream>
using namespace std;
class TypeofJuice
{
public:
	TypeofJuice() { }
	virtual void printJuice() = 0;
	~TypeofJuice() { }
};
class GrapeJuice : public TypeofJuice
{
public:
	GrapeJuice() { }
	void printJuice() { }
	~GrapeJuice() { }
};
class OrangeJuice : public TypeofJuice
{
public:
	OrangeJuice() { }
	void printJuice() { }
	~OrangeJuice() { }
};
class AppleJuice : public TypeofJuice
{
public:
	AppleJuice() { }
	void printJuice() { }
	~AppleJuice() { }
};
class JuiceFactory
{
```

```
public:
	JuiceFactory() { }
	virtual GrapeJuice* createGrapeJuice() = 0;
	virtual OrangeJuice* createOrangeJuice() = 0;
	virtual AppleJuice* createAppleJuice() = 0;
	~JuiceFactory() { }
};
class createSmallJuice  : public JuiceFactory
{
public:
	GrapeJuice* createGrapeJuice()
	{
		cout << "This is small grape juice" << endl;
		return new GrapeJuice;
	}
	OrangeJuice* createOrangeJuice()
	{
		cout << "This is small orange juice" << endl;
		return new OrangeJuice;
	}
	AppleJuice* createAppleJuice()
	{
		cout << "This is small apple juice" << endl;
		return new AppleJuice;
	}
};
class createMediumJuice : public JuiceFactory
{
public:
	GrapeJuice* createGrapeJuice()
	{
		cout << "This is medium grape juice" << endl;
		return new GrapeJuice;
	}
	OrangeJuice* createOrangeJuice()
	{
		cout << "This is medium orange juice" << endl;
		return new OrangeJuice;
	}
	AppleJuice* createAppleJuice()
	{
		cout << "This is medium apple juice" << endl;
		return new AppleJuice;
	}
};
```

```
class createBigJuice : public JuiceFactory
{
public:
	GrapeJuice* createGrapeJuice()
	{
		cout << "This is big grape juice" << endl;
		return new GrapeJuice;
	}
	OrangeJuice* createOrangeJuice()
	{
		cout << "This is big orange juice" << endl;
		return new OrangeJuice;
	}
	AppleJuice* createAppleJuice()
	{
		cout << "This is big apple juice" << endl;
		return new AppleJuice;
	}
};
int main()
{
	cout << "                                    Small Juice                    " <<
endl;
	createSmallJuice* smalljuice = new createSmallJuice;
	smalljuice->createGrapeJuice();
	smalljuice->createOrangeJuice();
	smalljuice->createAppleJuice();
	cout << endl;

	cout << "                                    Medium Juice                    "
<< endl;
	createMediumJuice* mediumjuice = new createMediumJuice;
	mediumjuice->createGrapeJuice();
	mediumjuice->createOrangeJuice();
	mediumjuice->createAppleJuice();
	cout << endl;

	cout << "                                    Big Juice                  " <<
endl;
	createBigJuice* bigjuice = new createBigJuice;
	bigjuice->createGrapeJuice();
	bigjuice->createOrangeJuice();
	bigjuice->createAppleJuice();
	cout << endl;
	return 0;
```

```
}
```

Output:

```
                                    Small Juice
This is small grape juice
This is small orange juice
This is small apple juice

                                    Medium Juice
This is medium grape juice
This is medium orange juice
This is medium apple juice

                                    Big Juice
This is big grape juice
This is big orange juice
This is big apple juice
```

Example 3 – Nguyễn Thành Phụng

```
#include<iostream>
using namespace std;
class TypeofElectronicDevice
{
public:
	TypeofElectronicDevice() { }
	virtual void printDevice() = 0;
	~TypeofElectronicDevice() { }
};
class IPhone : public TypeofElectronicDevice
{
public:
	IPhone() { }
	void printDevice() { cout << "This is iphone" << endl; }
	~IPhone() { }
};
class MacBook : public TypeofElectronicDevice
{
public:
	MacBook() { }
	void printDevice() { cout << "This is macbook" << endl; }
	~MacBook() { }
};
```

```
class IPad : public TypeofElectronicDevice
{
public:
	IPad() { }
	void printDevice() { cout << "This is ipad" << endl; }
	~IPad() { }
};
class DeviceFactory
{
public:
	virtual IPhone* createIphone() = 0;
	virtual IPad* createIpad() = 0;
	virtual MacBook* createMacBook() = 0;
};
class createPro : public DeviceFactory
{
public:
	IPhone* createIphone()
	{
		cout << "This is Iphone pro" << endl;
		return new IPhone;
	}
	IPad* createIpad()
	{
		cout << "This is Ipad pro" << endl;
		return new IPad;
	}
	MacBook* createMacBook()
	{
		cout << "This is MacBook pro" << endl;
		return new MacBook;
	}
};
class createPromax : public DeviceFactory
{
public:
	IPhone* createIphone()
	{
		cout << "This is Iphone pro max" << endl;
		return new IPhone;
	}
	IPad* createIpad()
	{
		cout << "This is Ipad pro max" << endl;
		return new IPad;
```

```
	}
	MacBook* createMacBook()
	{
		cout << "This is MacBook pro max" << endl;
		return new MacBook;
	}
};

int main()
{
	cout << "                                Pro Device" << endl;
	createPro* proDevice = new createPro;
	proDevice->createIphone();
	proDevice->createIpad();
	proDevice->createMacBook();
	cout << endl;
	cout << "                                Pro Max Device" << endl;
	createPromax* promaxDevice = new createPromax;
	promaxDevice->createIphone();
	promaxDevice->createIpad();
	promaxDevice->createMacBook();
	cout << endl;
}
```

Output:

```
                              Pro Device
This is Iphone pro
This is Ipad pro
This is MacBook pro

                              Pro Max Device
This is Iphone pro max
This is Ipad pro max
This is MacBook pro max
```

Example 4 – Vũ Phương Anh

```
#include<iostream>
using namespace std;
class Animal
{
public:
	virtual void Eat() = 0;
	virtual void Drink() = 0;
```

```
};
class Dog : public Animal
{
public:
	void Eat() {
		cout << "Eat dog food" << endl;
	}
	void Drink() {
		cout << "Drink dog milk" << endl;
	}
};
class Cat : public Animal
{
public:
	void Eat() {
		cout << "Eat cat food" << endl;
	}
	void Drink() {
		cout << "Drink cat milk" << endl;
	}
};
class AnimalFactory
{
public:
	AnimalFactory() { }
	virtual Cat* createCat() = 0;
	virtual Dog* createDog() = 0;
	~AnimalFactory() { }
};
class WhiteAnimalFactory : public AnimalFactory
{
public:
	WhiteAnimalFactory() { };
	Cat* createCat() {
		cout << "This is white cat" << endl;
		return new Cat;
	}
	Dog* createDog() {
		cout << "This is white dog" << endl;
		return new Dog;
	}
	~WhiteAnimalFactory() { }
};
class BlackAnimalFactory : public AnimalFactory
{
```

```
public:
	BlackAnimalFactory() { }
	Cat* createCat() {
		cout << "This is black cat" << endl;
		return new Cat;
	}
	Dog* createDog() {
		cout << "This is black dog" << endl;
		return new Dog;
	}
	~BlackAnimalFactory() { }
};
int main()
{
	cout << "                                        White Animal
" << endl;
	WhiteAnimalFactory* whiteAnimal = new WhiteAnimalFactory;
	whiteAnimal->createCat()->Eat();
	whiteAnimal->createCat()->Drink();
	cout << endl;
	whiteAnimal->createDog()->Eat();
	whiteAnimal->createDog()->Drink();
	cout << endl;

	cout << "                                        Black Animal
" << endl;
	BlackAnimalFactory* blackAnimal = new BlackAnimalFactory;
	blackAnimal->createCat()->Eat();
	blackAnimal->createCat()->Drink();
	cout << endl;
	blackAnimal->createDog()->Eat();
	blackAnimal->createDog()->Drink();
	cout << endl;

	return 0;
}
```

Output:

```
                                        White Animal
This is white cat
Eat cat food
This is white cat
Drink cat milk
```

This is white dog
Eat dog food
This is white dog
Drink dog milk

Black Animal

This is black cat
Eat cat food
This is black cat
Drink cat milk

This is black dog
Eat dog food
This is black dog
Drink dog milk