Group 1:

Trương Như Quốc Thịnh – 18125027

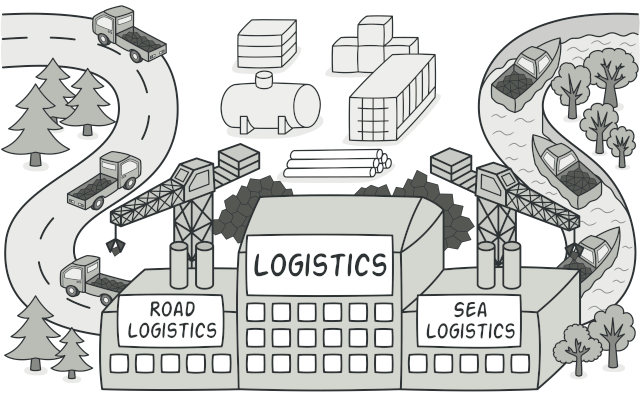
Vũ Phương Anh – 18125061

Nguyễn Thành Phụng – 18125109

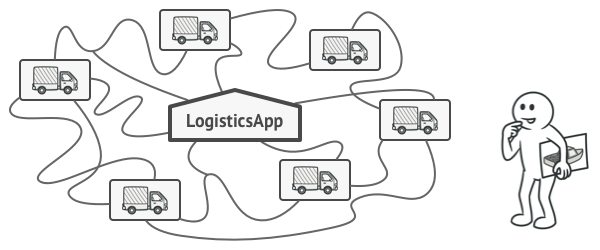
Trần Thiên Phúc – 18125137

SEMINAR REPORT

FACTORY METHOD

1. **Problem:**

* Without factory method:



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

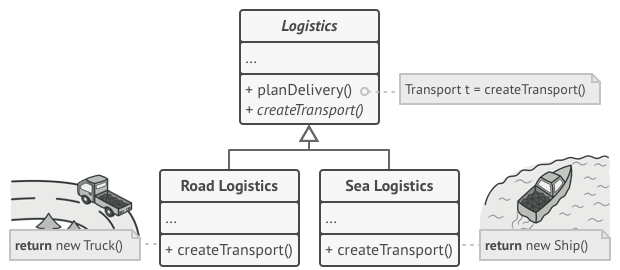
🡪 major codes are inside the Truck class

* 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

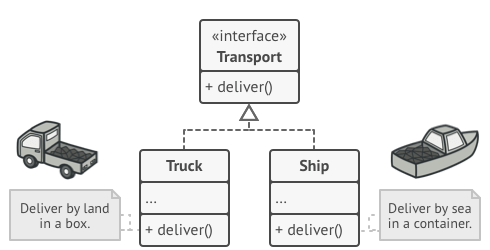
* The disadvantage of this example is: users need to make to change a lot of code if users want to add more different class.
* With factory method:
* The Factory Method suggests that users replace direct object construction calls (using the new operator) with calls to a special factory method
  + The objects are still created via the new operator but it is called from within the 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

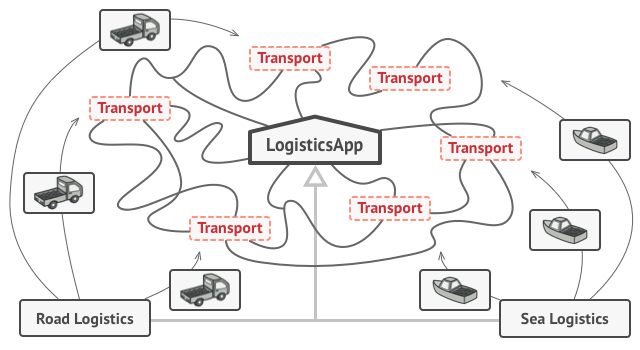
🡪 Subclasses can alter the class of objects being returned by the factory method.

* 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



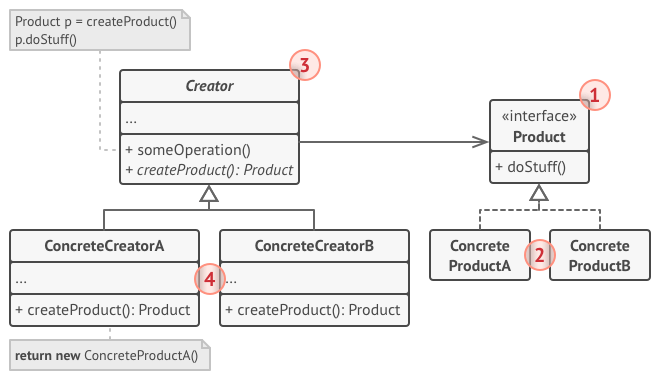
* Code uses the Factory Method often called the client code which doesn’t see a difference between the actual products returned by various subclasses
* The client:
  + treats all the products as abstract Transport.
  + knows that all transport objects are supposed to have the deliver method but how deliver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

1. **Structure:**

**Product**: declares the interface common to all objects that can be produced by the creator and its subclasses.

**Creator**: Declares the Factory Method that returns new product objects. It’s important that the return type of this method matches the product interface



**Concrete products**: are different implementations of the product interface

**Concrete creator**:Override the base factory method 🡪 it returns a different type of product

1. **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.

1. **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 |

1. **Application:**
2. 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

1. 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.

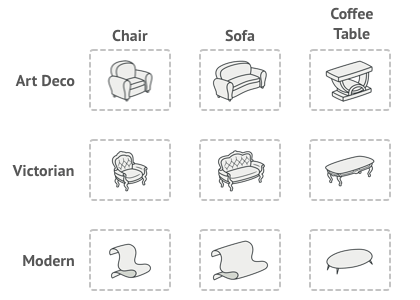
1. When users want to save system resources by reusing existing objects instead of rebuilding them each time.

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

1. **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).
* [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 users can also use [Prototype](https://refactoring.guru/design-patterns/prototype) to compose the methods on these classes.
* Iterator: users can use [Factory Method](https://refactoring.guru/design-patterns/factory-method) along with [Iterator](https://refactoring.guru/design-patterns/iterator) to let collection subclasses return different types of iterators that are compatible with the collections.
* [Prototype](https://refactoring.guru/design-patterns/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](https://refactoring.guru/design-patterns/factory-method) is based on inheritance but doesn’t require an initialization step.
* [Factory Method](https://refactoring.guru/design-patterns/factory-method) is a specialization of [Template Method](https://refactoring.guru/design-patterns/template-method). At the same time, a Factory Method may serve as a step in a large Template Method*.*

ABSTRACT FACTORY

1. **Problem:**

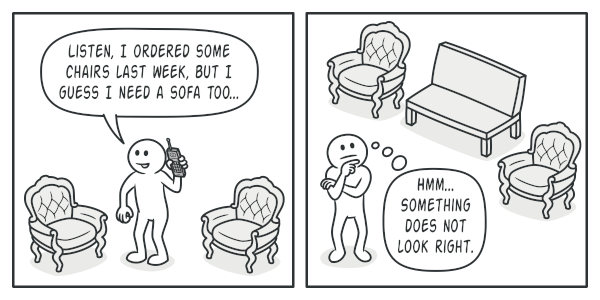
* Without Abstract Factory:

+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



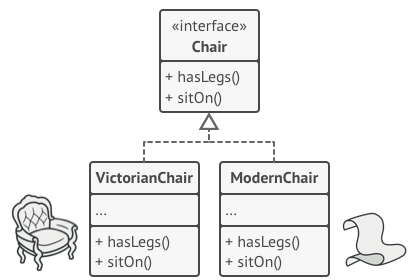
-More than that, users won’t wanna change existing code when adding new products or families of products to the program.And furniture vendors update their catalogs very often and users, definitely, won’t want to change users’ core code each time changing.

* The disadvantage of this example: The users have to change codebase when adding new classes.
* With abstract factory:

The Abstract Factory pattern suggests:

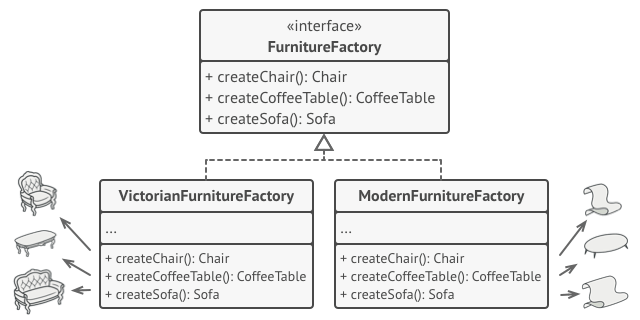
+explicitly declare interfaces for each distinct product (chair or sofa or coffeetable) of the product family

+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

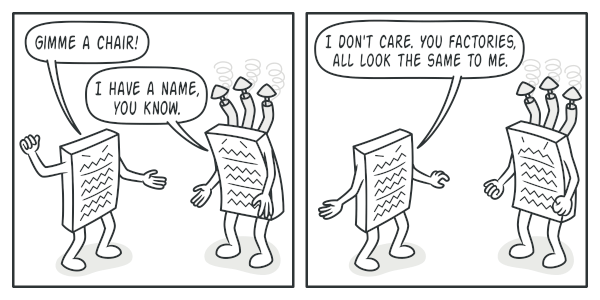
🡪Must return abstract product types represented by the interfaces (Chair or Sofa or CoffeeTable)



-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

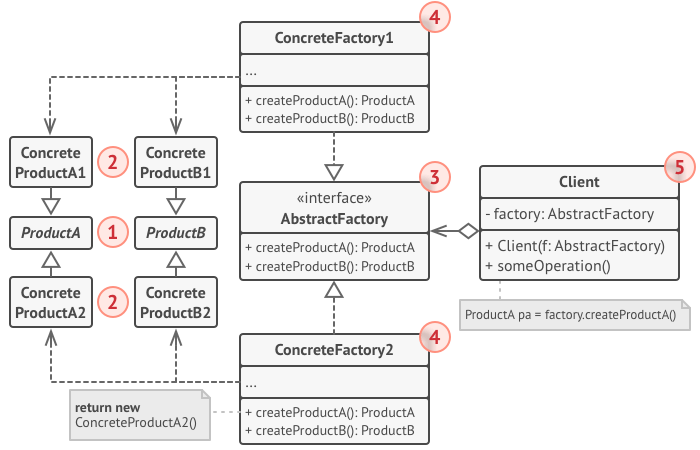


The clients don’t have to be awaew of the factory’s class nor does it matter what kind of objects it gets 🡪 the clients must treat all chairs in the same manner using the abstract interface.

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

1. **Structure**



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

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

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

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

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

1. **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
2. **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

1. **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 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 | * A lot of new interface and classes are introduced along with pattern 🡪 The code may become more complicated than it should be |

1. **Relations with Other patterns**

* Factory Method: Many designs start by using [Factory Method](https://refactoring.guru/design-patterns/factory-method) (less complicated and more customizable via subclasses) and envolve 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.
* [Prototype](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.
* Facade: [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.
* Bridge: 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).

Question 1 : The ***factory method pattern*** is an \_\_\_\_\_\_\_\_ [design pattern](http://quiz.thefullwiki.org/Design_pattern_(computer_science)) to implement the concept of [factories](http://quiz.thefullwiki.org/Factory_(software_concept)).

A. Object (computer science)

B. Unified Modeling Language

C. **Object-oriented programming**

D. Programming paradigm

Question 2 : \_\_\_\_\_\_\_\_ Improving the Design of Existing Code

A. **Refactoring**

B. Object-oriented programming

C. Extreme Programming

D. Software architecture

Question 3 : \_\_\_\_\_\_\_\_\_ is a pattern often implemented using *factory methods*

A. Factory method pattern

B. **Abstract factory pattern**

C. Decorator pattern

D. Design pattern (computer science)

Question 4 : Cohen, Tal; Gil, Joseph (2007).

“*Better Contruction with Factories* “ ? \_\_\_\_\_\_\_ .

A. HTML

B. **Portable Document Format (PDF)**

C. OpenDocument

D. Office Open XML

Question 5 : As an example, when \_\_\_\_\_\_\_\_ are created from two real numbers the real numbers can be interpreted as Cartesian or polar coordinates, but using factory methods, the meaning is clear.

A. Split-complex number

B. Field (mathematics)

C. Vector space

D. **Complex number**

C# Example a program to read [**image files**](http://www.thefullwiki.org/Image_file) and make [**thumbnails**](http://www.thefullwiki.org/Thumbnail) out of them.

public interface ImageReader {

public DecodedImage getDecodedImage();

}

public class GifReader implements ImageReader {

public DecodedImage getDecodedImage() {

return decodedImage;

}

}

public class JpegReader implements ImageReader {

// ....

}

//Each time the program reads an image it needs to create a //reader of the appropriate type based on some information in //the file. This logic can be encapsulated in a factory method:

public class ImageReaderFactory {

public static ImageReader getImageReader(InputStream is) {

int imageType = determineImageType(is);

switch(imageType) {

case ImageReaderFactory.GIF:

return new GifReader(is);

case ImageReaderFactory.JPEG:

return new JpegReader(is);

// etc.

}

}