Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Design Patterns

(IT069IU)

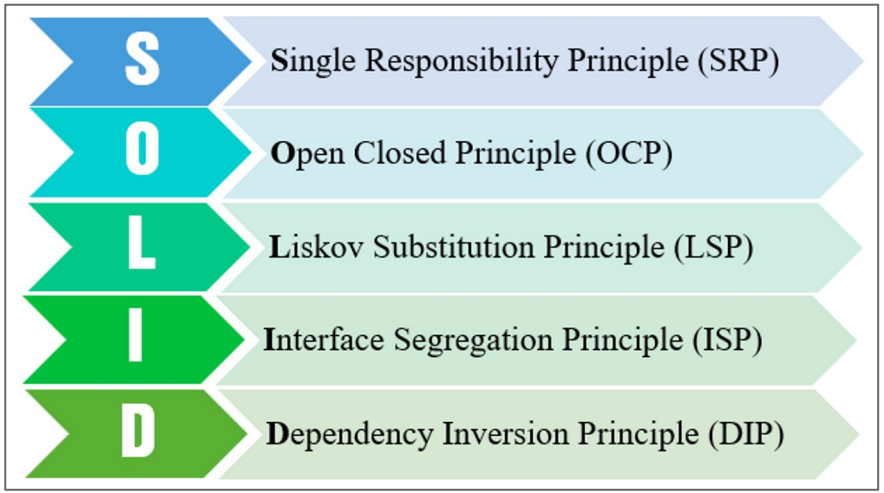
Le Duy Tan, Ph.D.

ldtan@hcmiu.edu.vn

leduytanit.com

Previously,

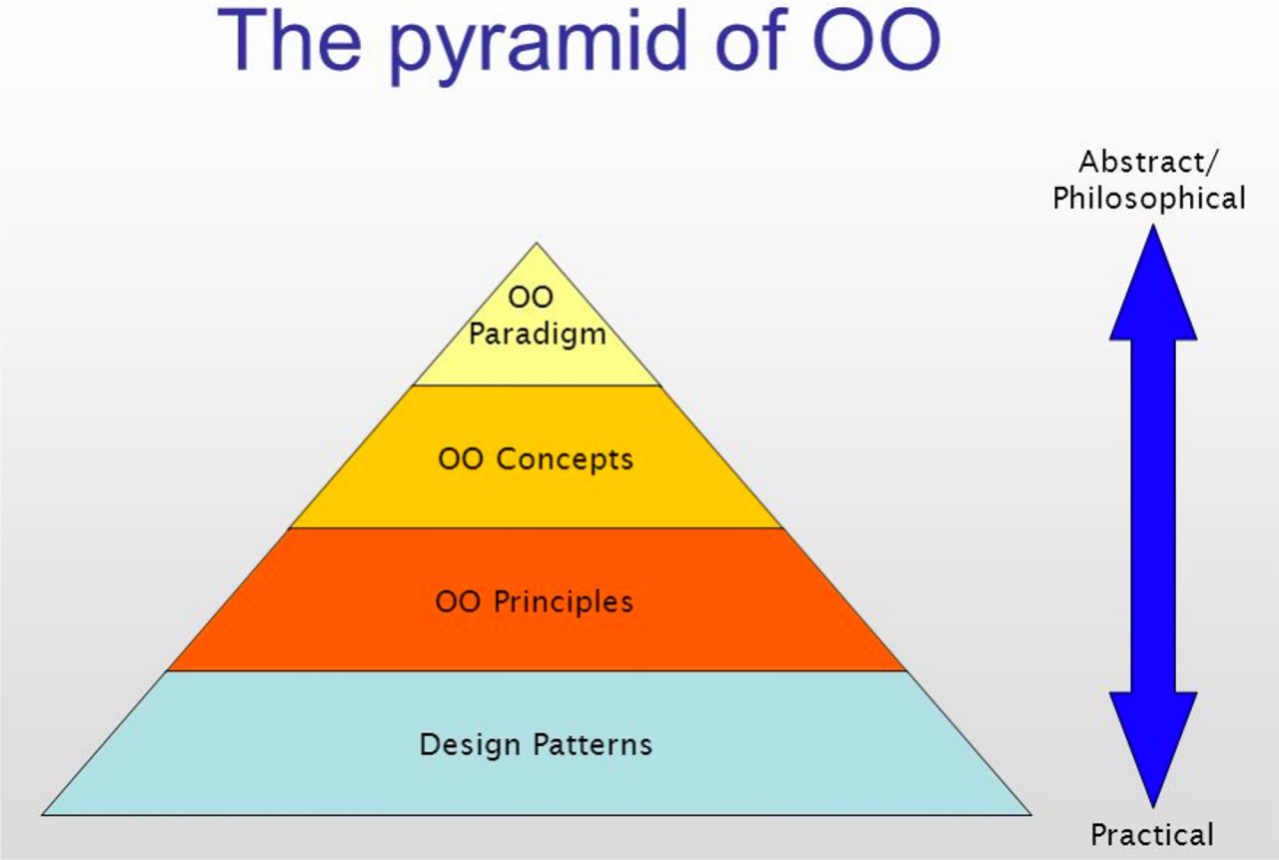
- Object Oriented Design Principles: SOLID
  - S: Single responsibility
  - O: Open/closed principle
  - L: Liskov substitution principle
  - I: Interface segregation principle
  - D: Dependency inversion principle



Agenda

- Design Patterns:
  - Creational
    - Singleton
    - Factory Method
  - Structural
  - Behavioral

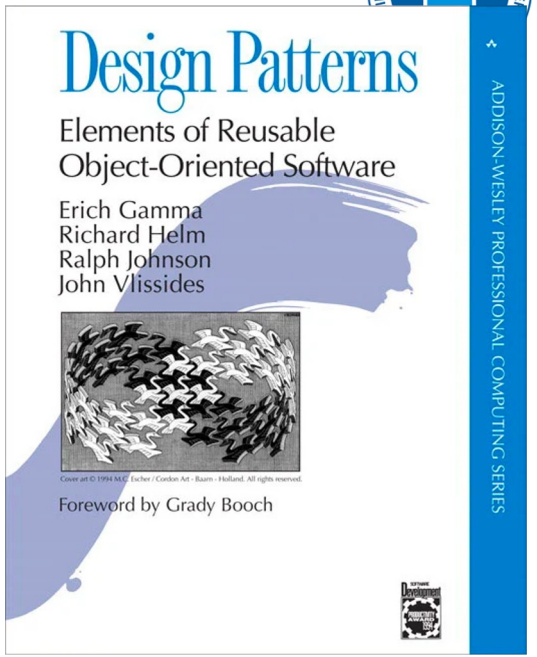
OOP Concepts vs OOP Principles vs Design Pattern




Design Patterns

Design Patterns

- A design pattern is a **solution** to a commonly occurring **problem** in **software design**.
- These patterns are mostly “evolved” rather than “discovered”. A lot of learning, by lots of professional, have been summarized into these design patterns.
- None of these patterns force you any specific implementation; they are just **guidelines to solve a particular problem** – in a particular way for particular contexts. Code implementation is your responsibility.
- Was first **introduced** by the **Gang of Four**:
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides



Types of Design Patterns

There are three categories of these patterns:

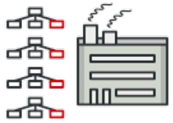
- **Creational Patterns**
  - Focused on the way objects are created.
- **Structural Patterns**
  - Focused on creating objects into larger structures which gives developers the advantage to keep groups of children together and also abstract objects to simplify applications.
- **Behavioral Patterns**
  - Focused on combining algorithms and responsibilities together which concerned with communication between objects.

Many Design Patterns of Each Type

Creational	Structural	Behavioral
1. Singleton	6. Adapter	13. Template Method
2. Factory	7. Composite	14. Mediator
3. Abstract Factory	8. Proxy	15. Observer
4. Builder	9. Flyweight	16. Strategy
5. Prototype	10. Facade	17. Command
	11. Bridge	18. State
	12. Decorator	19. Visitor
		20. Iterator
		21. Interpreter
		22. Memento
		23. Chain Of Responsibility

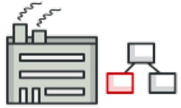
Creational Design Pattern

These design patterns focused on the way objects are created:

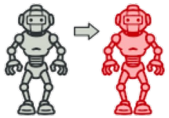


- **Abstract Factory** (⭐⭐⭐): Lets you produce families of related objects without specifying their concrete classes.

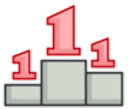
- **Builder**(⭐⭐⭐): Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



- **Factory Method**(⭐⭐⭐): Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



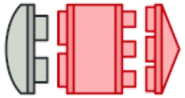
- **Prototype**(⭐⭐): Lets you copy existing objects without making your code dependent on their classes.



- **Singleton**(⭐⭐): Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Structural Design Pattern

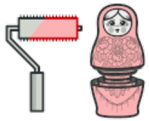
These design patterns focused on creating objects into larger structures:



- **Adapter**(⭐⭐⭐): Allows objects with incompatible interfaces to collaborate.

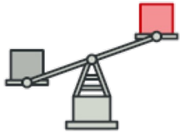
- **Bridge**(⭐): Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

- **Composite**(⭐⭐): Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

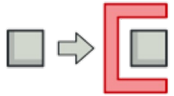


- **Decorator**(⭐⭐): Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

- **Facade**(⭐⭐): Provides a simplified interface to a library, a framework, or any other complex set of classes.



- **Flyweight**(⭐): Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



- **Proxy**(⭐): Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

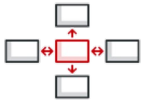
Behavioral Design Pattern

These design patterns focused on the communication between objects.:

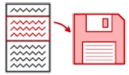
- **Chain of responsibility(⭐):** Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

- **Command(⭐⭐⭐)**: Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

- **Iterator(⭐⭐⭐)**: Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

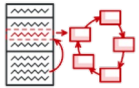


- **Mediator(⭐⭐)**: Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

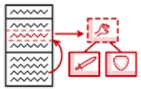


- **Memento(⭐)**: Lets you save and restore the previous state of an object without revealing the details of its implementation.

- **Observer(⭐⭐⭐)**: Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



- **State(⭐⭐)**: Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



- **Strategy(⭐⭐⭐)**: Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

- **Template method(⭐⭐):** Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

- **Visitor(⭐)**: Lets you separate algorithms from the objects on which they operate.

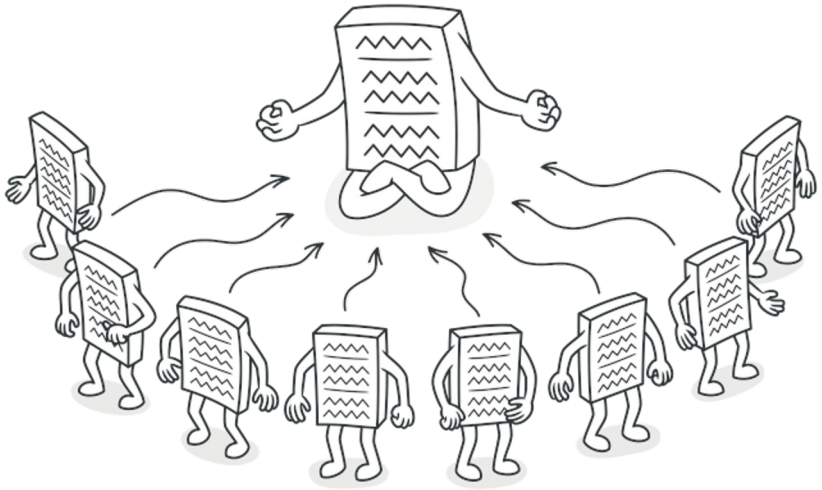
Which is best?

- **There is no best design pattern**! The best solution depends on the pattern that works for you at any given moment.
- Sometimes you need to implement one design pattern for one context while using another for another context.
- Try out as many as you can and discover the many solutions that others have supplied to us already!

Singleton

Singleton: What?

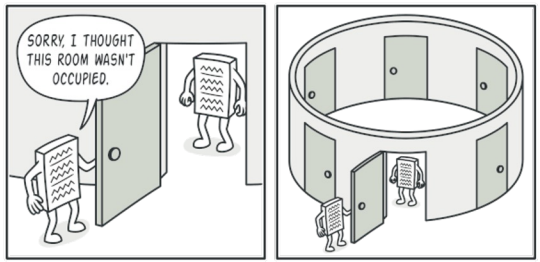
- Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- Singleton is a creational design pattern that lets you **ensure that a class has only one instance**, while **providing a global access point to this instance**.



Singleton: Why?

The Singleton pattern solves two problems:

- 1. Ensure that a class has just a single instance. Why would anyone want to control how many instances a class has?
  - The most common reason for this is to control access to some shared resource—for example, a database or a file.
  - Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created. So The singleton object is initialized only when it's requested for the first time.
- 2. Provide a global access point to that instance.
  - Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.



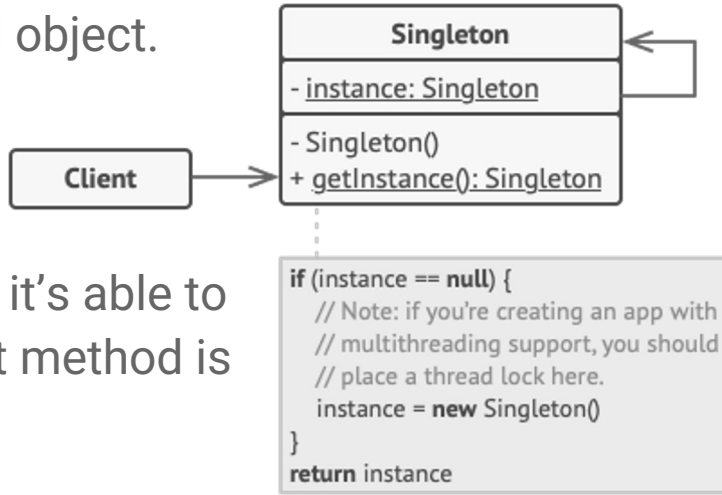
Clients may not even realize that they're working with the same object all the time.

Singleton: How?

All implementations of the Singleton have these two steps in common:

Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.



If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

Singleton: Example

SingleObject.java

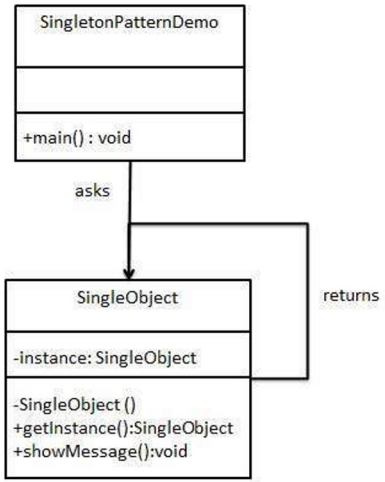
```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that
   //this class cannot be instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Error: The constructor SingleObject()
      //is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Output

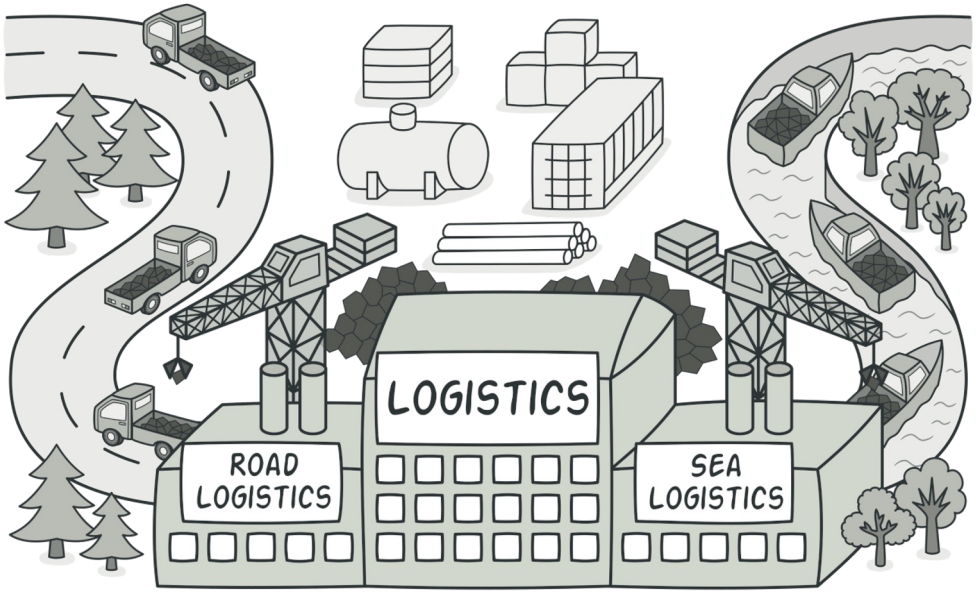
```
Hello World!
```



Factory Method

Factory Method: What?

- 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.



Factory Method: Why?

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.

After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

Great news, right? But how about the code? At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

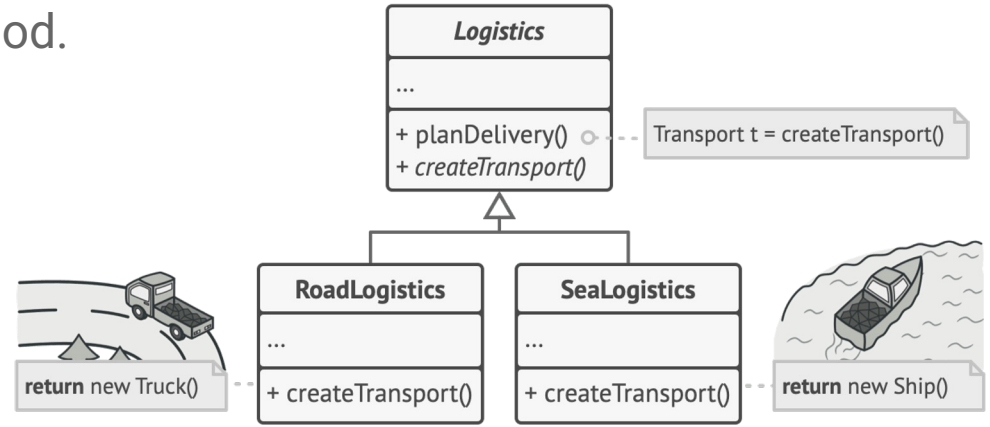
As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Factory Method: How (Part 1)?

The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. Don't worry: the objects are still created via the new operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as products.

now you can override the factory method in a subclass and change the class of products being created by the method.

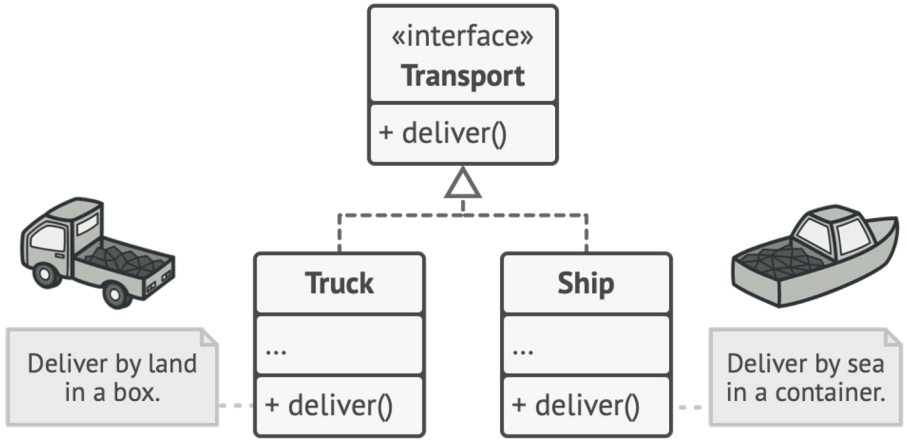


Subclasses can alter the class of objects being returned by the factory method.

Factory Method: How (Part 2)?

There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

For example, both **Truck** and **Ship** classes should implement the **Transport** interface, which declares a method called **deliver**. Each class implements this method differently: trucks deliver cargo by land, ships deliver cargo by sea.

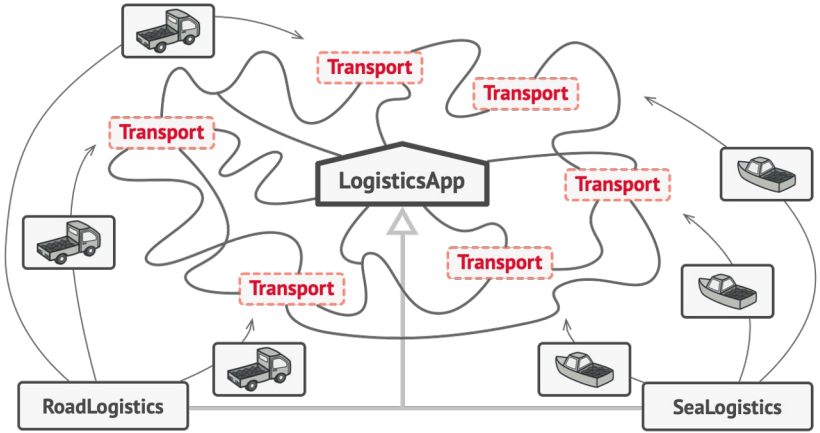


All products must follow the same interface.

Factory Method: How (Part 3)?

The factory method in the **RoadLogistics** class returns truck objects, whereas the factory method in the **SeaLogistics** class returns ships.

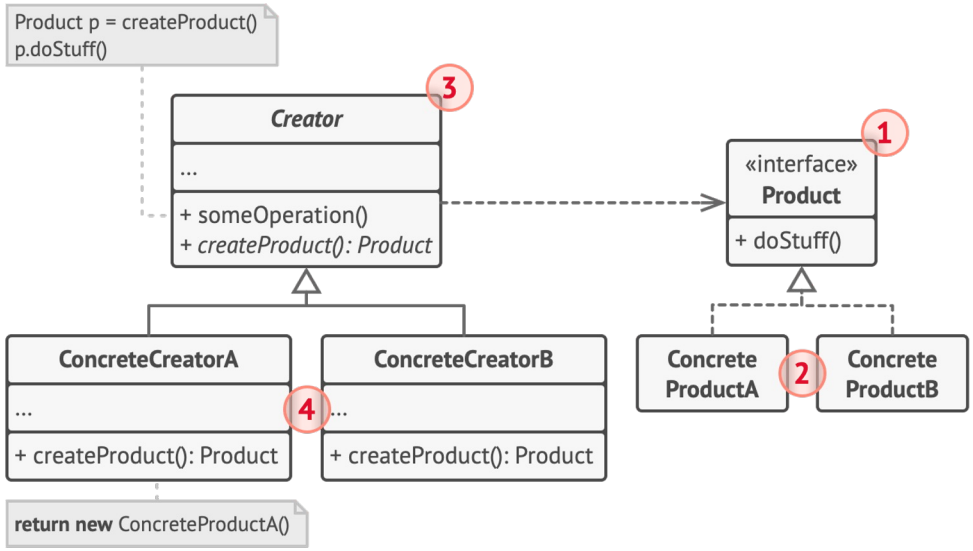
The code that uses the factory method (often called the client code) doesn't see a difference between the actual products returned by various subclasses. The client treats all the products as abstract **Transport**. The client knows that all transport objects are supposed to have the **deliver** method, but exactly how it works isn't important to the client.



As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.

Factory Method: Steps

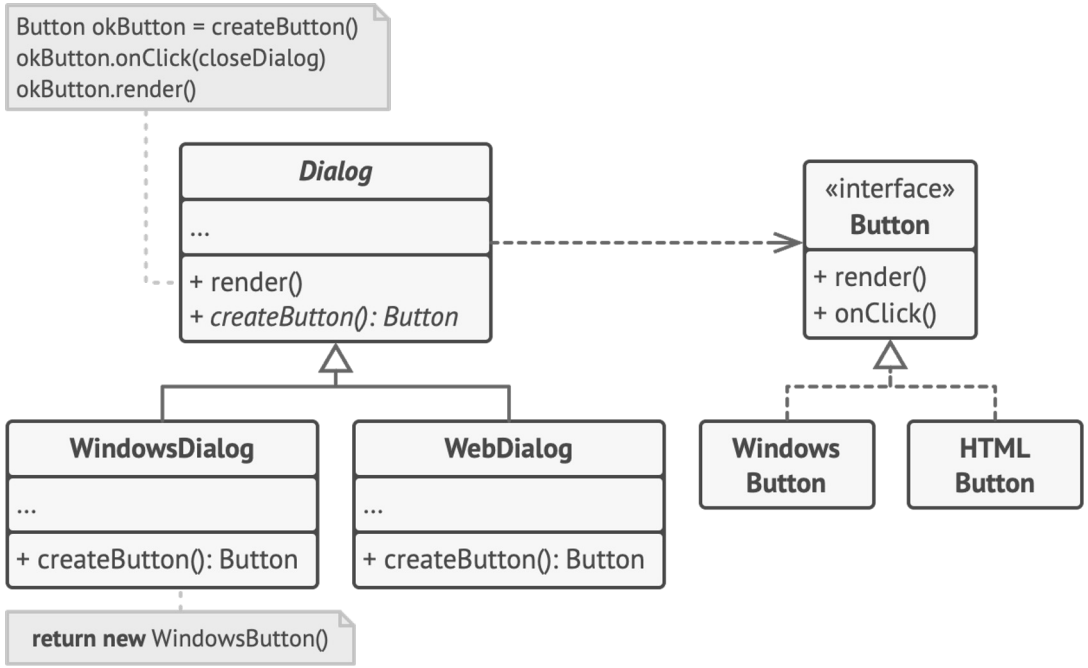
1. The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2. Concrete Products are different implementations of the product interface.
3. The Creator class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.
   a. You can declare the factory method as `abstract` to force all subclasses to implement their own versions of the method.
4. Concrete Creators override the base factory method so it returns a different type of product.



Factory Method: Example

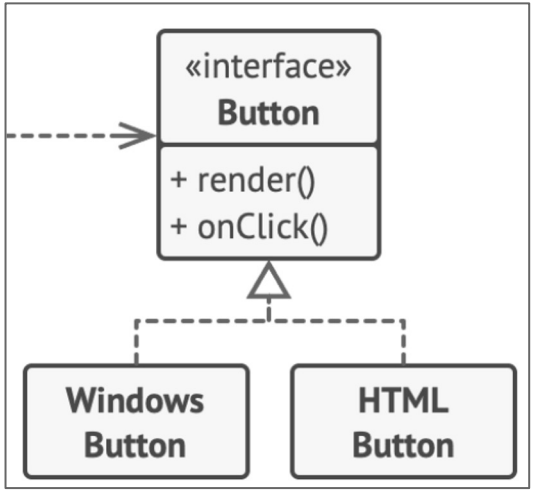
- This example illustrates how the Factory Method can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes.

  - The base dialog class uses different UI elements to render its window. Under various operating systems, these elements may look a little bit different, but they should still behave consistently. A button in Windows is still a button in Linux.



Factory Method: Button

```java
public interface Button {
    public void render();
    public void onClick(String action);
}
```

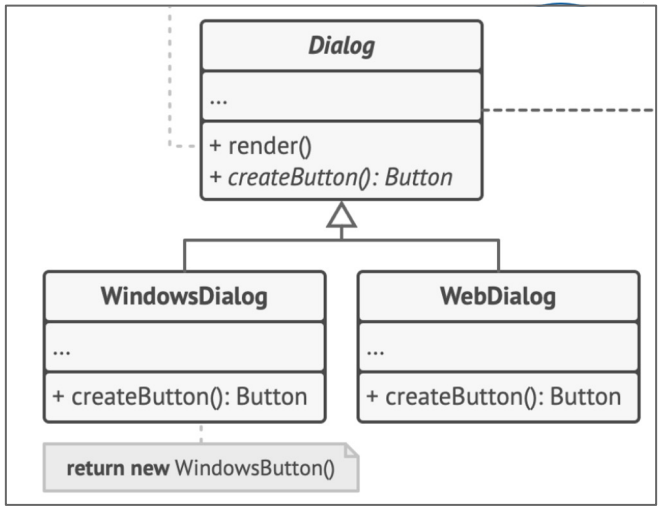



```java
public class WindowsButton implements Button{
    @Override
    public void render() {
        // Render a button in Windows style
        System.out.println("Here is a Windows button!");
    }

    @Override
    public void onClick(String action) {
        // Bind a native OS click event.
        System.out.println("A click in Windows detected!");
        System.out.println(action);
    }
}
```

```java
public class HTMLButton implements Button{
    @Override
    public void render() {
        // Render a button in HTML Web style
        System.out.println("Here is a HTML button on Web!");
    }

    @Override
    public void onClick(String action) {
        // Bind a web browser click event.
        System.out.println("A click in web browser detected!");
        System.out.println(action);
    }
}
```

Factory Method: Dialog Factory

```
public abstract class Dialog {
    public abstract Button createButton();

    public void render(){
        Button okButton = createButton();
        okButton.render();
        okButton.onClick( action: "Close the dialog!");
    }
}
```




```
public class WindowsDialog extends Dialog{
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}
```

```
public class WebDialog extends Dialog{

    @Override
    public Button createButton() {
        return new HTMLButton();
    }
}
```

Factory Method: TestApplication

```java
public class MyApplication {
    Dialog myDialog;

    public MyApplication() {
        String configOS = "Windows";

        if (configOS.equals("Windows")){
            myDialog = new WindowsDialog();
        } else if (configOS.equals("Web")){
            myDialog = new WebDialog();
        } else {
            // You need to throws error/exception
            // for unknown OS config
        }
    }

    public static void main(String[] args) {
        MyApplication myApp = new MyApplication();
        myApp.myDialog.render();
    }
}
```

Output for “Windows”:

```
Here is a Windows button!
A click in Windows detected!
Close the dialog!
```

Output for “Web”:

```
Here is a HTML button on Web!
A click in web browser detected!
Close the dialog!
```

Factory Method: Final Thought

- The Factory Method separates product construction code from the code that actually uses the product. Therefore it’s easier to extend the product construction code independently from the rest of the code.
- For example, to add a new product type to the app, you’ll only need to create a new creator subclass and override the factory method in it.

Recap

- Design Patterns:
  - Creational
    - Singleton
    - Factory Method
  - Structural
  - Behavioral

Reference

If you want to learn more about design patterns, please go through these links:

- https://refactoring.guru/design-patterns
- https://www.journaldev.com/1827/java-design-patterns-example-tutorial
- https://www.javatpoint.com/design-patterns-in-java
- https://github.com/iluwatar/java-design-patterns

Thank you for your listening!

“Live as if you were to die tomorrow.
Learn as if you were to live forever!”

Mahatma Gandhi