

# Design Patterns(SE-408)

Course Teacher

Assistant Professor

Engr. Mustafa Latif

## Factory Method



# Factory Method

- *Also known as: Virtual Constructor*
- **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.



## Problem

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.

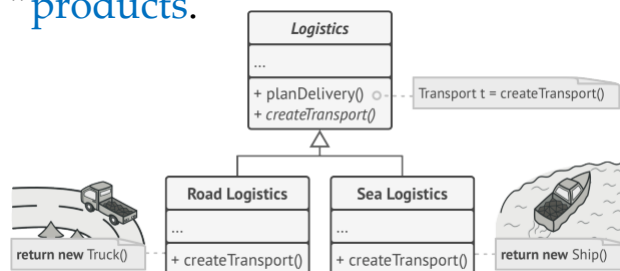
As a result, you will end up with pretty nasty code, **riddled with conditionals** that switch the app's behavior depending on the class of transportation objects.



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

## Solution

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**."

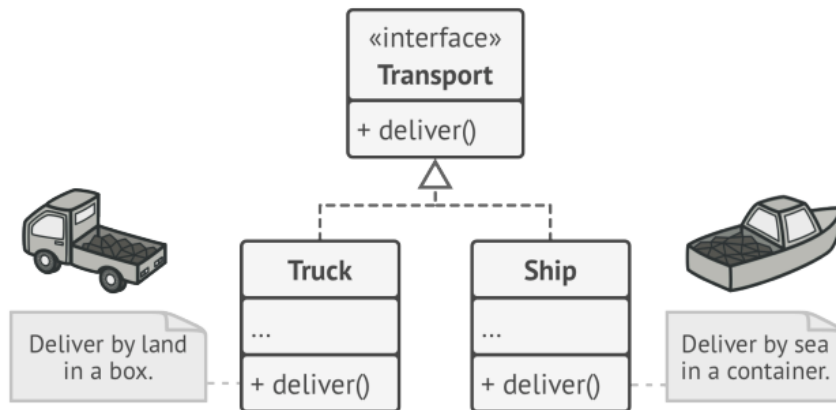


Subclasses can alter the class of objects being returned by the factory method.

# Factory Method

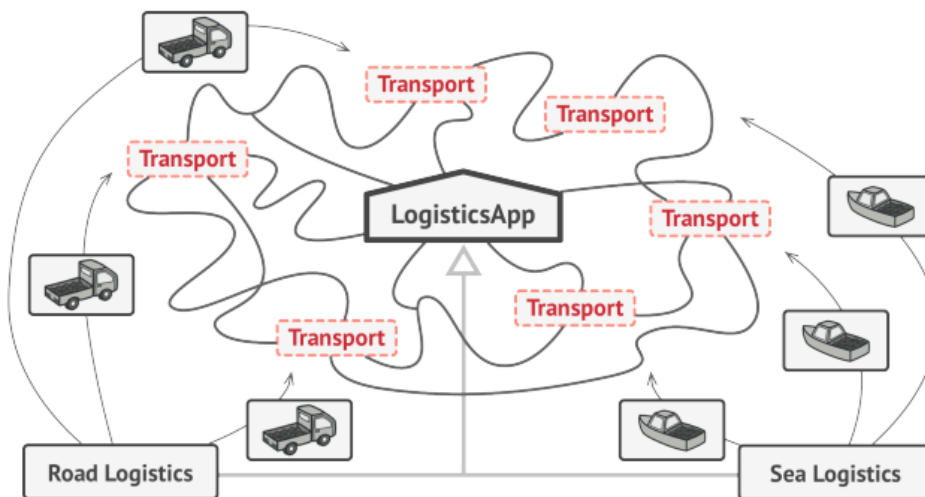
- At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.
- There's a slight limitation though: 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.

# Factory Method



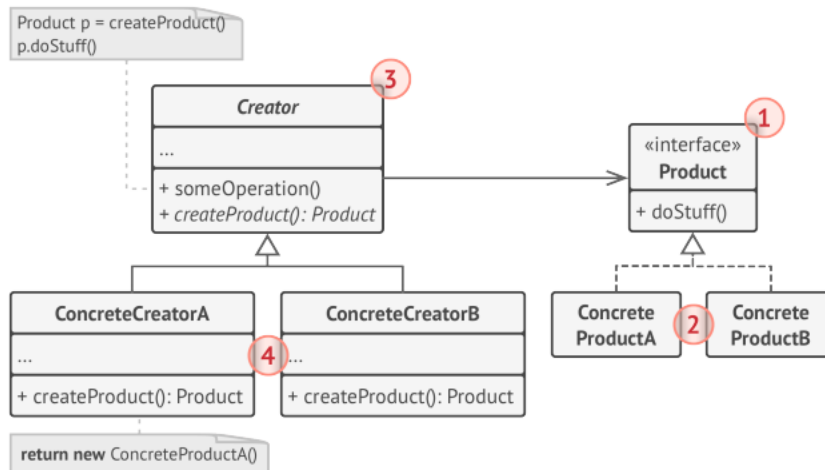
*All products must follow the same interface.*

- 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. 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*

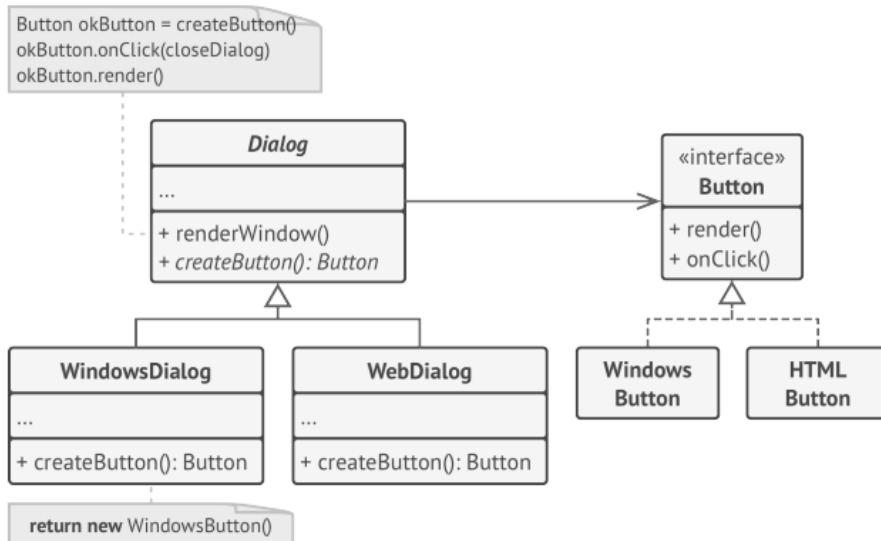
# Structure




- The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
- **Concrete Products** are different implementations of the product interface.
- 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.  
(You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.)
- 4. **Concrete Creators** override the base factory method so it returns a different type of product.
- (Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source. )


# Pseudocode

- 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 cross-platform dialog example.*


- 
- 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.

- 
- When the factory method comes into play, you don't need to rewrite the logic of the dialog for each operating system. If we declare a factory method that produces buttons inside the base dialog class, we can later create a dialog subclass that returns Windows-styled buttons from the factory method. The subclass then inherits most of the dialog's code from the base class, but, thanks to the factory method, can render Windows-looking buttons on the screen.
  - For this pattern to work, the base dialog class must work with abstract buttons: a base class or an interface that all concrete buttons follow. This way the dialog's code remains functional, whichever type of buttons it works with.






```
1 // The creator class declares the factory method that must
2 // return an object of a product class. The creator's subclasses
3 // usually provide the implementation of this method.
4 class Dialog is
5 // The creator may also provide some default implementation
6 // of the factory method.
7 abstract method createButton()
8
9 // Note that, despite its name, the creator's primary
10 // responsibility isn't creating products. It usually
11 // contains some core business logic that relies on product
12 // objects returned by the factory method. Subclasses can
13 // indirectly change that business logic by overriding the
14 // factory method and returning a different type of product
15 // from it.
```



```
16 method render() is
17 // Call the factory method to create a product object.
18 Button okButton = createButton()
19 // Now use the product.
20 okButton.onClick(closeDialog)
21 okButton.render()
22
23
24 // Concrete creators override the factory method to change the
25 // resulting product's type.
26 class WindowsDialog extends Dialog is
27 method createButton() is
28 return new WindowsButton()
29
30 class WebDialog extends Dialog is
31 method createButton() is
32 return new HTMLButton()
33
```

```
34
35 // The product interface declares the operations that all
36 // concrete products must implement.
37 interface Button is
38 method render()
39 method onClick(f)
40
41 // Concrete products provide various implementations of the
42 // product interface
43 class WindowsButton implements Button is
44 method render(a, b) is
45 // Render a button in Windows style.
46 method onClick(f) is
47 // Bind a native OS click event.
48
49 class HTMLButton implements Button is
50 method render(a, b) is
51 // Return an HTML representation of a button.
52 method onClick(f) is
53 // Bind a web browser click event.
54
```

```
55
56 class Application is
57 field dialog: Dialog
58
59 // The application picks a creator's type depending on the
60 // current configuration or environment settings.
61 method initialize() is
62 config = readApplicationConfigFile()
63
64 if (config.OS == "Windows") then
65 dialog = new WindowsDialog()
66 else if (config.OS == "Web") then
67 dialog = new WebDialog()
68 else
69 throw new Exception("Error! Unknown operating system.")
70
```



```
71 // The client code works with an instance of a concrete
72 // creator, Even though through its base interface. As long as
73 // the client keeps working with the creator via the base
74 // interface, you can pass it any creator's subclass.
75 method main() is
76 this.initialize()
77 dialog.render()
```



## Java Code Example

## How to Implement

1. Make all products follow the same interface. This interface should declare methods that make sense in every product.
2. Add an empty factory method inside the creator class. The return type of the method should match the common product interface.
3. 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. You might need to add a temporary parameter to the factory method to control the type of returned product. At this point, the code of the factory method may look pretty ugly. It may have a large switch operator that picks which product class to instantiate. But don't worry, we'll fix it soon enough.
4. Now, 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.

## How to Implement

5. If there are too many product types and it doesn't make sense to create subclasses for all of them, you can reuse the control parameter from the base class in subclasses. For instance, imagine that you have the following hierarchy of classes: the base Mail class with a couple of subclasses: AirMail and GroundMail ; the Transport classes are Plane , Truck and Train . While the AirMail class only uses Plane objects, GroundMail may work with both Truck and Train objects. You can create a new subclass (say TrainMail ) to handle both cases, but there's another option. The client code can pass an argument to the factory method of the GroundMail class to control which product it wants to receive.
6. If, after all of the extractions, the base factory method has become empty, you can make it abstract. If there's something left, you can make it a default behavior of the method.



## Pros and Cons

- You avoid tight coupling between the creator and the concrete products.
- *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.
- The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.



## Review