**Factory Method**

#### **Intent**

**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.

#### **Problems**

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.

#### **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.*

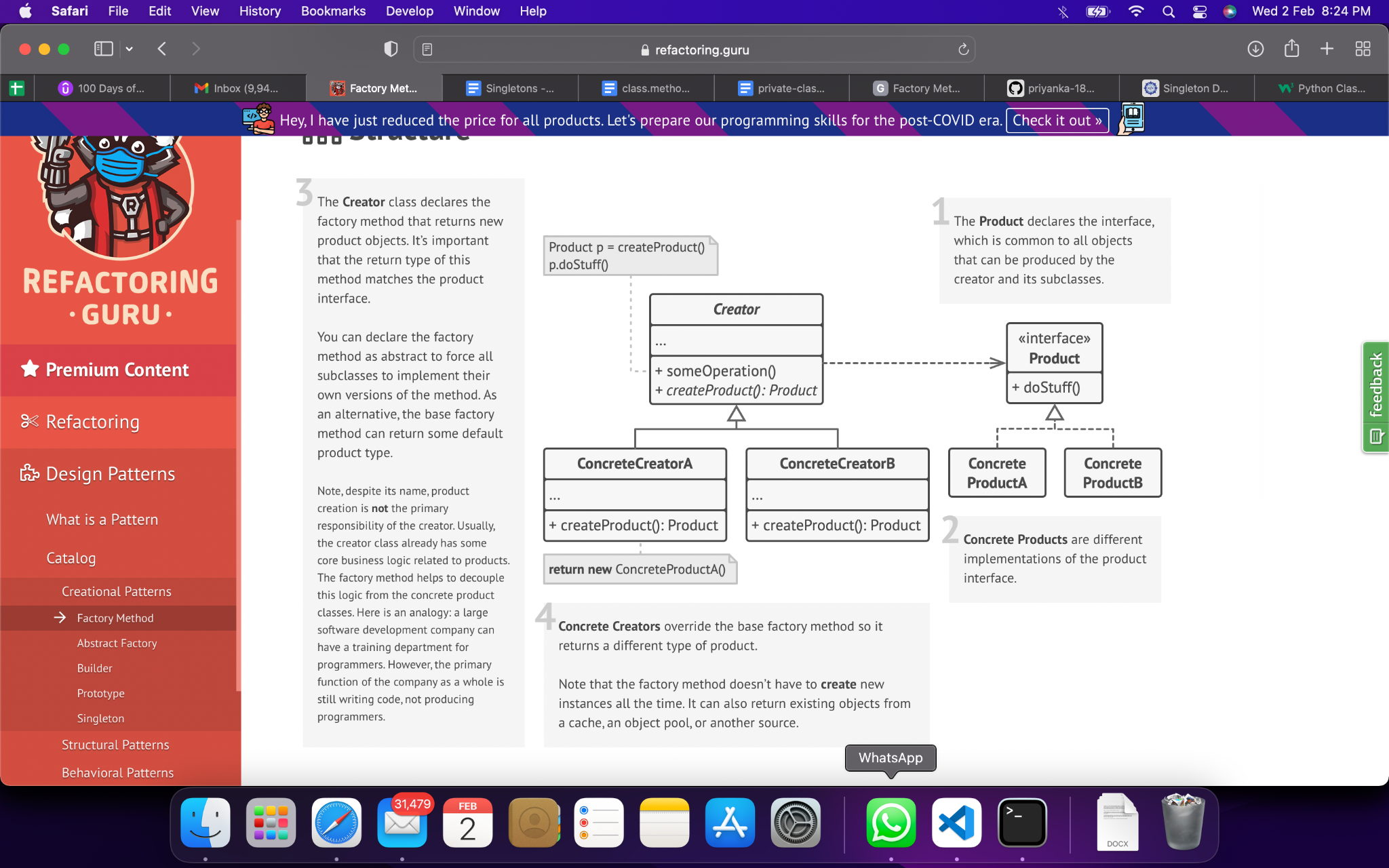
At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.

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.

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 SeaLogisticsclass 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.

#### **Diagrammatic Representation**



#### **Checklist**

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.
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 Mailclass 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 & Cons**

| You avoid tight coupling between the creator and the concrete products. | 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. |
| --- | --- |
| *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. |  |

#### **Applicability**

1. Use the Factory Method when you don’t know beforehand the exact types and dependencies of the objects your code should work with.
2. Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
3. Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

#### **Relation with Other Patterns**

1. 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).
2. [**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 you can also use [**Prototype**](https://refactoring.guru/design-patterns/prototype) to compose the methods on these classes.
3. You 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.
4. [**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.
5. [**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*.

#### **References**

1. “Factory Method.” *Refactoring.Guru*, https://refactoring.guru/design-patterns/factory-method.