**Factory Method**

#### **Intent**

**Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

#### **Problems**

Imagine that you’re creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say: Chair + Sofa + CoffeeTable.
2. Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.

You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

Also, you don’t want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn’t want to change the core code each time it happens.

#### **Solution**

The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.

The next move is to declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (for example, createChair, createSofaand createCoffeeTable). These methods must return abstract product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on.

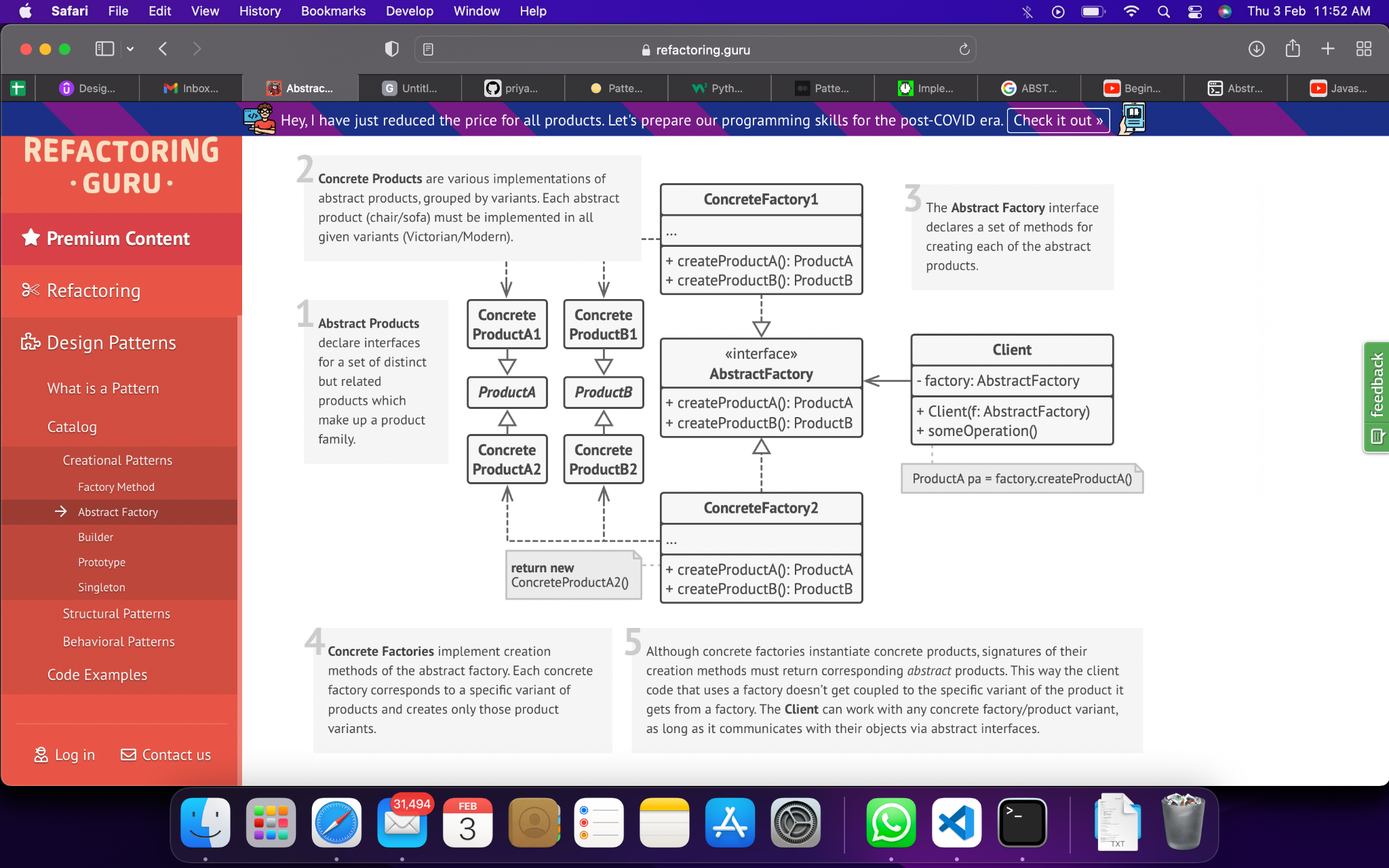
Now, how about the product variants? For each variant of a product family, we create a separate factory class based on the AbstractFactory interface. A factory is a class that returns products of a particular kind. For example, the ModernFurnitureFactory can only create ModernChair, ModernSofa and ModernCoffeeTable objects.

The client code has to work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

Say the client wants a factory to produce a chair. The client doesn’t have to be aware of the factory’s class, nor does it matter what kind of chair it gets. Whether it’s a Modern model or a Victorian-style chair, the client must treat all chairs in the same manner, using the abstract Chair interface. With this approach, the only thing that the client knows about the chair is that it implements the sitOn method in some way. Also, whichever variant of the chair is returned, it’ll always match the type of sofa or coffee table produced by the same factory object.

There’s one more thing left to clarify: if the client is only exposed to the abstract interfaces, what creates the actual factory objects? Usually, the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration or the environment settings.

#### **Diagrammatic Representation**



#### **Checklist**

1. Map out a matrix of distinct product types versus variants of these products.
2. Declare abstract product interfaces for all product types. Then make all concrete product classes implement these interfaces.
3. Declare the abstract factory interface with a set of creation methods for all abstract products.
4. Implement a set of concrete factory classes, one for each product variant.
5. Create factory initialization code somewhere in the app. It should instantiate one of the concrete factory classes, depending on the application configuration or the current environment. Pass this factory object to all classes that construct products.
6. Scan through the code and find all direct calls to product constructors. Replace them with calls to the appropriate creation method on the factory object.

#### **Pros & Cons**

| You can be sure that the products you’re getting from a factory are compatible with each other. | The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern. |
| --- | --- |
| You avoid tight coupling between concrete products and client code. |  |
| *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support. |  |
| *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code. |  |

#### **Applicability**

1. Use the Abstract Factory when your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

#### **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. [**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.
3. [**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.
4. [**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.
5. 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.
6. [**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).

#### **References**

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