Abstract Factory

Nengbao Liu
Terence Zhao

Intent

“Provides an interface for creating families of related or dependent objects without specifying their concrete classes.”

- Gamma et. al (Design Patterns)

Motivation

Imagine we built a Sandwich Robot to work at a local hamburger restaurant.

Our SandwichBot 1.0 has been programmed to combine a sesame bun, a grilled beef patty, a slice of cheddar cheese, a piece of lettuce, and a squirt of ketchup to make a sandwich.

Let's say that one day, SandwichBot is reassigned to a nearby hotdog stand. For SandwichBot to now make hotdogs, we would need to reprogram it to recognize a new set of ingredients. Since updating SandwichBot is costly, it should ideally be able to make sandwiches regardless of the particular ingredients available.

In a similar fashion, if our application deals with "creating families of related or dependent objects" (sandwich ingredients), we should not hard-code their creation to specific concrete classes (i.e. hamburger ingredients).

SandwichBot 2.0

AbstractFactory (SandwichFactory)
Declares an interface for ingredient operations that create abstract ingredient objects

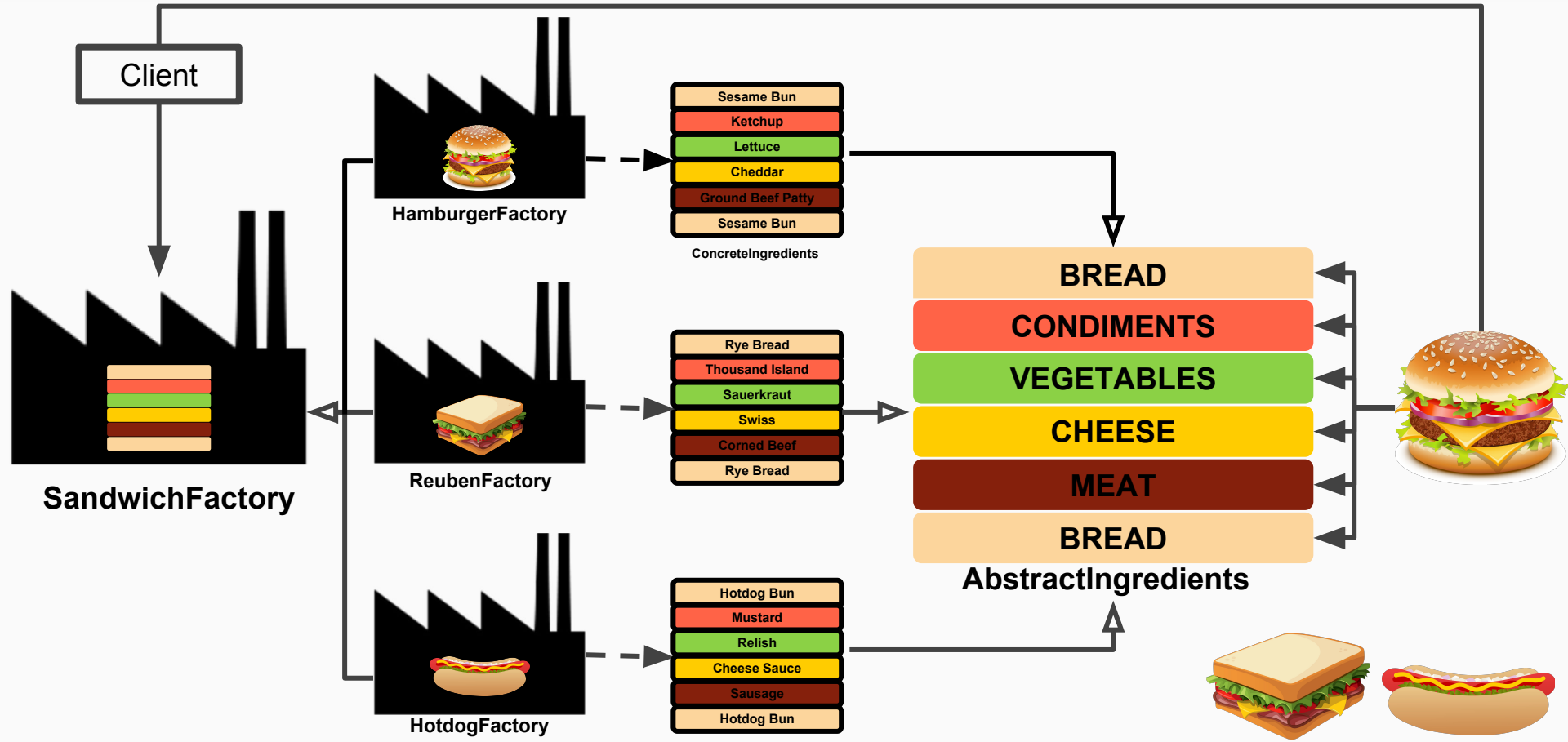
ConcreteFactory (HamburgerFactory, ReubenFactory, HotdogFactory, etc.)
Implements the ingredient operations to create concrete ingredient objects

AbstractIngredient (Bread, Meat, Cheese, Vegetables, and Condiments)
Declares an interface for a type of ingredient object

ConcreteIngredient (Rye, Corned Beef, Swiss Cheese, Sauerkraut, Thousand Island dressing)
Implements the AbstractIngredient interface and defines an ingredient object to be created by the corresponding concrete sandwich factory.

Client (SandwichBot)
Uses only interfaces declared by the AbstractFactory and AbstractIngredient classes and remains unaware of the concrete classes it's using.
Thus, SandwichBot2.0 will be able to create any number of different sandwiches since it stays independent of the specific ingredients needed and only commits to an ingredient interface.

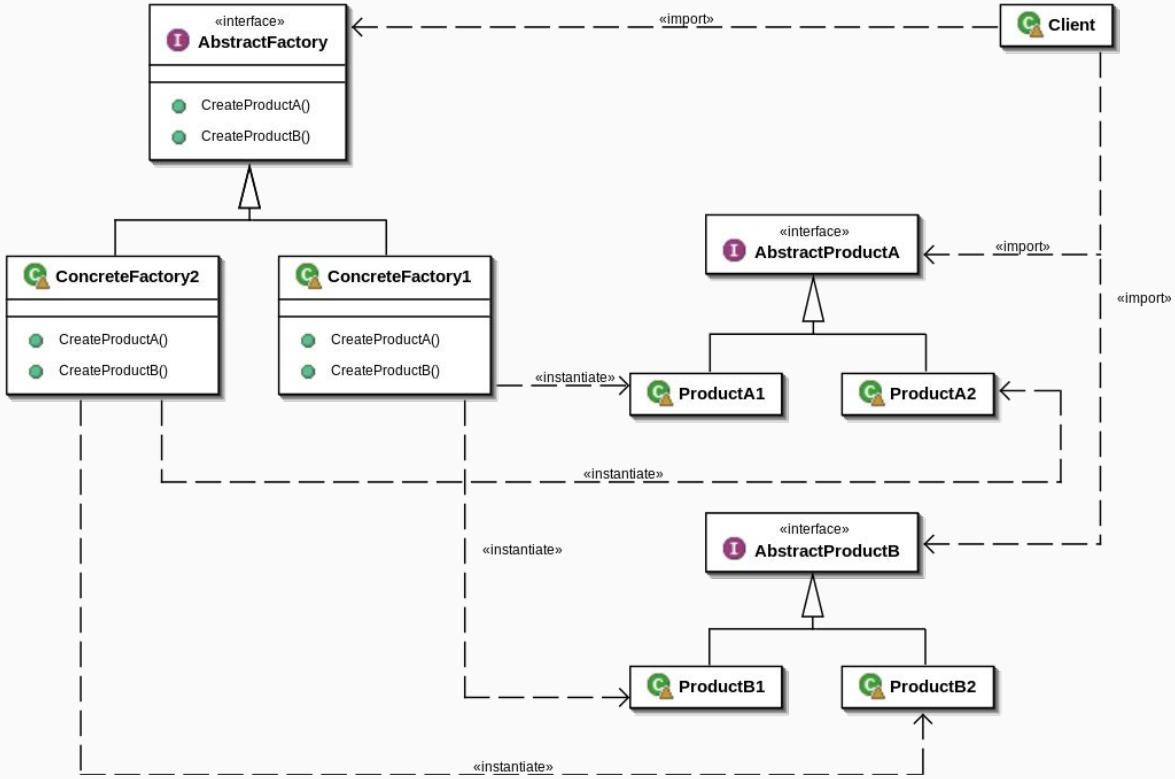
UML: Sandwich Example



UML Diagram

The Abstract Factory pattern is useful when:

1. An application should be independent of how its objects are created, composed, and represented.
2. An application should be configured with one of multiple families of related or dependent objects.
3. You want to constrain an application to use a family of related objects that were designed to be used together.
4. You want to provide a class library of objects in which you reveal just their interfaces, not their implementations.



Benefits

Isolates concrete classes:
Since clients manipulate instances through their abstract interfaces, product class names are isolated in the implementation of the concrete factory.

Makes exchanging product families easy:
By simply changing the concrete factory an application uses, we can change the entire family of products all at once.

Promotes consistency among products:
Makes it easy for an application to enforce using only product objects from the same family.

Consequences

Supporting new kinds of products is difficult:
Since the AbstractFactory interface fixes the set of products that can be created, supporting new kinds of products requires extending the factory interface.This involves changing the AbstractFactory class and all of its subclasses.

Defining extensible factories requires a tradeoff:
A more flexible but less safe design is to add a parameter to the operations that create objects allowing the type of object created to be specified. Unfortunately, the client will not be able to differentiate the classes of the products returned or use sub-class specific operations unless it tries to downcast, which is not always feasible or safe.

Implementation Techniques

Factories as singletons:
Typically only one instance of a ConcreteFactory per product family is needed
Creating products via Factory Method:
We can define a factory method for each product to be overridden in a concrete factory when specifying its products, however, doing so requires a new concrete factory subclass for each product family, even if the product families are only slightly different.
Creating products via Prototype:
If many product families are possible, the concrete factory can be implemented using the Prototype pattern. The concrete factory is initialized with the prototypical instance of each product in the family and creates a new product by cloning its prototype, eliminating the need for a new concrete factory class for each new product family.

Code Example