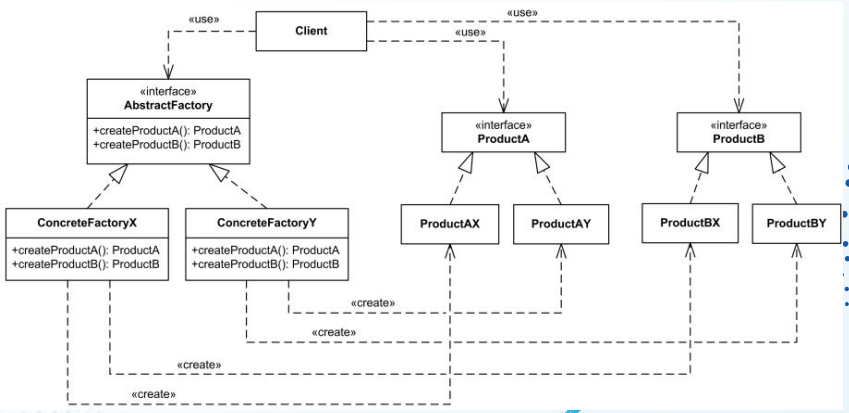
Design Pattern: Factory

So what is it?

- Abstract Factories declares an interface for the creation of Abstract Products.
- Concrete Factory objects carry out the creation by implementing an Abstract Factory's interface.
- Abstract Products declare interfaces for types of products.
- Products define a product object, created by the corresponding Concrete Factory and used by the user through the Abstract interface.
- Clients interact with the Abstract Factories and Products, and has no knowledge of which factory is producing the products.

UML Provided by:
https://www.uml-diagrams.org/design-pattern-abstract-factory-uml-class-diagram-example.html



Pros and Cons

Pros

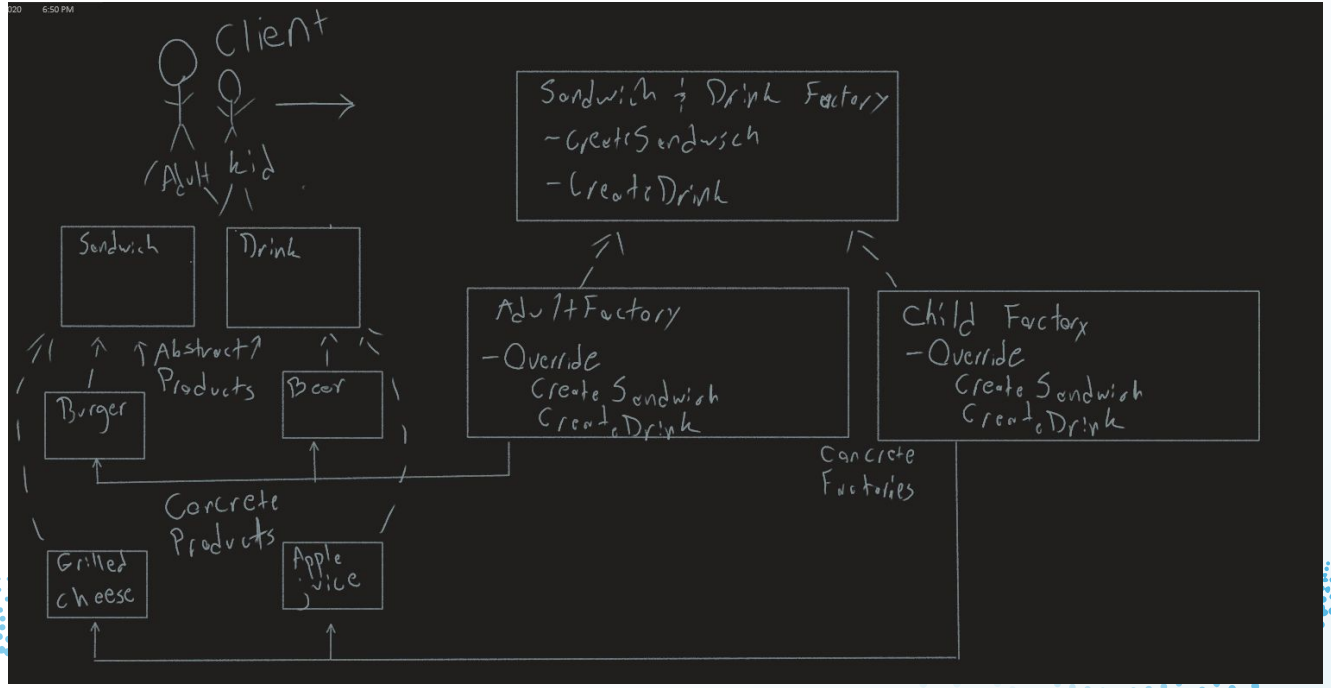
- Allows for interchanging families of concrete classes without changing the code that uses them.
- Avoids tight coupling of concrete products and “client code”.
- Follows Single Responsibility Principle by extracting product creation code into one place.
- Follows Open/Closed Principle, new variants of products can be created w/o breaking existing client code.
- Facilitates dependency injection.

Cons

- Often used when it isn’t necessary, if you only have one factory there’s no point.
- When used unnecessarily, code can become overly complicated with new interfaces and classes.

Let's get real

Link to Code: https://github.com/Niccoryan0/Abstract-Factory-Example



Example adapted from: https://www.exceptionnotfound.net/abstract-factory-pattern-in-csharp/

What about Factory Method?

- Factory Method is a *Class* Creational Pattern that utilizes a specific method that can be overridden by children for creating new objects.
  - Shown in the top of image by class B extending class A, it can call doSomething() but now the object created is different than if called by class A.
- Abstract Factory (an *Object* Creational Pattern) creates concrete Factory Objects based on a Factory Interface, the factories perform the same action but will create different concrete “Product” objects (generally based on Abstract Products) based on the factory being used at the time.
  - Shown in the bottom of image by class A requiring a concrete factory in it’s constructor, which will determine what Foo object it produces.

```
class A {
    public void doSomething() {
        Foo f = makeFoo();
        f.whatever();
    }

    protected Foo makeFoo() {
        return new RegularFoo();
    }
}

class B extends A {
    protected Foo makeFoo() {
        //subclass is overriding the factory method
        //to return something different
        return new SpecialFoo();
    }
}
```

And here is an abstract factory in use:

```
class A {
    private Factory factory;

    public A(Factory factory) {
        this.factory = factory;
    }

    public void doSomething() {
        //The concrete class of "f" depends on the concrete class
        //of the factory passed into the constructor. If you provide a
        //different factory, you get a different Foo object.
        Foo f = factory.makeFoo();
        f.whatever();
    }
}

interface Factory {
    Foo makeFoo();
    Bar makeBar();
    Aycufcn makeAmbiguousYetCommonlyUsedFakeClassName();
}

//need to make concrete factories that implement the "Factory" interface here
```

share improve this answer follow

edited Jun 20 at 9:12 Community ♦ 1 • 1

answered Apr 21 '11 at 5:39 Tom Dalling 20.9k • 4 • 55 • 77

Thanks Tom!

Wait, what is it again?

- *Object* Creational design pattern
- Useful for creating objects that are in related or dependent but different families without the need to rely on concrete implementations, two factories can create the “same” object in two different ways.
- Facilitates the use of Dependency Injection and other architectures.
- Used heavily in C# and Java, potentially even overused. We’ll get to that.
- Being an extremely common pattern, even if you don’t use it you will almost definitely see it/work with it at some point.

The end!

Any Questions?