Abstract Factory

# Name and classification

* Name: Abstract Factory Pattern.
* Classification: Creational Pattern.

# Also known as

Kit.

# Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.

We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.

There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate widget for the look and feel. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetFactory returns a scroll bar for Presentation Manager. Clients create widgets solely through the WidgetFactory interface and have no knowledge of the classes that implement widgets for a particular look and feel. In other words, clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

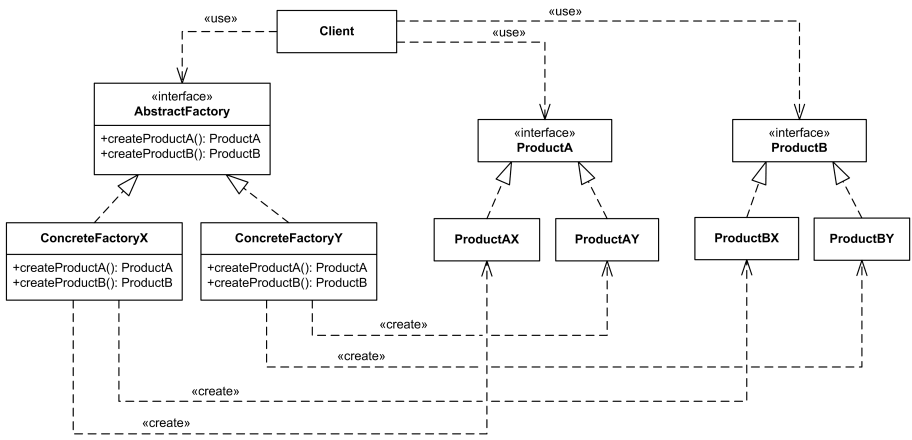
A WidgetFactory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a MotifWidgetFactory.

# Applicability

The Factory pattern should be used when:

* A system should be independent of how its products are created, composed, and represented.
* A system should be configured with one of multiple families of products.
* A family of related product objects is designed to be used together, and you need to enforce this constraint.
* You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

# Structure



# Participants

* AbstractFactory: declares an interface for operations that create abstract product objects.
* ConcreteFactory: implements the operations to create concrete product objects.
* AbstractProduct: declares an interface for a type of product object.
* ConcreteProduct: defines a product object to be created by the corresponding concrete factory, implements the AbstractProduct interface.
* Client: uses only interfaces declared by AbstractFactory and AbstractProduct classes.

# Collaborations

* Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
* AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

# Consequences

* Benefits:

+ Isolates concrete classes.

+ Makes exchanging product families easy.

+ Increase consistency among products.

* Drawback: Supporting new kinds of products is difficult.

# Implementations

Techniques for implementing the Abstract Factory pattern:

* Factories as singletons. An application typically needs only one instance of a ConcreteFactory per product family. So it's usually best implemented as a Singleton.
* Creating the products. AbstractFactory only declares an interface for creating products. It's up to ConcreteProduct subclasses to actually create them. The most common way to do this is to define a factory method for each product. A concrete factory will specify its products by overriding the factory method for each. While this implementation is simple, it requires a new concrete factory subclass for each product family, even if the product families differ only slightly.
* Defining extensible factories. AbstractFactory usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it.

A more flexible but less safe design is to add a parameter to operations that create objects. This parameter specifies the kind of object to be created. It could be a class identifier, an integer, a string, or anything else that identifies the kind of product. In fact, with this approach, AbstractFactory only needs a single "Make" operation with a parameter indicating the kind of object to create. This is the technique used in the Prototype- and the class-based abstract factories discussed earlier.

# Sample code

We will use Abstract Factory pattern to create a pizza ingredients. The factory will be responsible for creating each ingredient in the ingredient family. There are different ingredient families corresponding to each regions.

* **PizzaIngredientFactory is the interface for the factory that is going to create all our ingredients:**

**public interface** PizzaIngredientFactory {  
 **public** Dough createDough();  
 **public** Sauce createSauce();  
 **public** Cheese createCheese();  
 **public** Veggies[] createVeggies();  
 **public** Pepperoni createPepperoni();  
 **public** Clams createClam();  
}

* **NYPizzaIngredientFactory and ChicagoPizzaIngredientFactory are the concrete factories which implement PizzaIngredientFactory:**

**public class** NYPizzaIngredientFactory **implements** PizzaIngredientFactory {  
 **public** Dough createDough() {  
 **return new** ThinCrustDough();  
 }  
 **public** Sauce createSauce() {  
 **return new** MarinaraSauce();  
 }  
 **public** Cheese createCheese() {  
 **return new** ReggianoCheese();  
 }  
 **public** Veggies[] createVeggies() {  
 Veggies veggies[] = { **new** Garlic(), **new** Onion(), **new** Mushroom(), **new** RedPepper() };  
 **return** veggies;  
 }  
 **public** Pepperoni createPepperoni() {  
 **return new** SlicedPepperoni();  
 }  
 **public** Clams createClam() {  
 **return new** FreshClams();  
 }  
}

**public class** ChicagoPizzaIngredientFactory  
 **implements** PizzaIngredientFactory  
{  
 **public** Dough createDough() {  
 **return new** ThickCrustDough();  
 }  
 **public** Sauce createSauce() {  
 **return new** PlumTomatoSauce();  
 }  
 **public** Cheese createCheese() {  
 **return new** MozzarellaCheese();  
 }  
 **public** Veggies[] createVeggies() {  
 Veggies veggies[] = { **new** BlackOlives(),  
 **new** Spinach(),  
 **new** Eggplant() };  
 **return** veggies;  
 }  
 **public** Pepperoni createPepperoni() {  
 **return new** SlicedPepperoni();  
 }  
 **public** Clams createClam() {  
 **return new** FrozenClams();  
 }  
}

* **Class pizza is the client of ingredient abstract factory:**

**public class** Pizza {  
 String **name**;  
 PizzaIngredientFactory **ingredientFactory**;  
 Dough **dough**;  
 Sauce **sauce**;  
 Veggies **veggies**[];  
 Cheese **cheese**;  
 Pepperoni **pepperoni**;  
 Clams **clam**;  
 **public** Pizza (PizzaIngredientFactory ingredientFactory) {  
 **this**.**ingredientFactory** = ingredientFactory;  
 }  
 **void** prepare() {  
 System.***out***.println(“Preparing “ + **name**);  
 **dough** = **ingredientFactory**.createDough();  
 **sauce** = **ingredientFactory**.createSauce();  
 **cheese** = **ingredientFactory**.createCheese();  
 }  
 **void** bake() {  
 System.***out***.println(“Bake **for** 25 minutes at 350”);  
 }  
 **void** cut() {  
 System.***out***.println(“Cutting the pizza into diagonal slices”);  
 }  
 **void** box() {  
 System.***out***.println(“Place pizza in official PizzaStore box”);  
 }  
 **void** setName(String name) {  
 **this**.**name** = name;  
 }  
 String getName() {  
 **return name**;  
 }  
 **public** String toString() {  
 *// code to print pizza here* }  
}

* **At run time, the pizza maker can pass in different ingredient factories to make different kinds of pizza.**

# Known uses

* InterViews uses the "Kit" suffix [Lin92] to denote AbstractFactory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a LayoutKit that generates different composition objects depending on the layout desired. For example, a layout that is conceptually horizontal may require different composition objects depending on the document's orientation (portrait or landscape).
* ET++ [WGM88] uses the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example). The WindowSystem abstract base class defines the interface for creating objects that represent window system resources (MakeWindow, MakeFont, MakeColor, for example). Concrete subclasses implement the interfaces for a specific window system. At run-time, ET++ creates an instance of a concrete WindowSystem subclass that creates concrete system resource objects.

# Related patterns

* AbstractFactory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.
* A concrete factory is often a Singleton.