**Abstract Factory**

# Intent

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

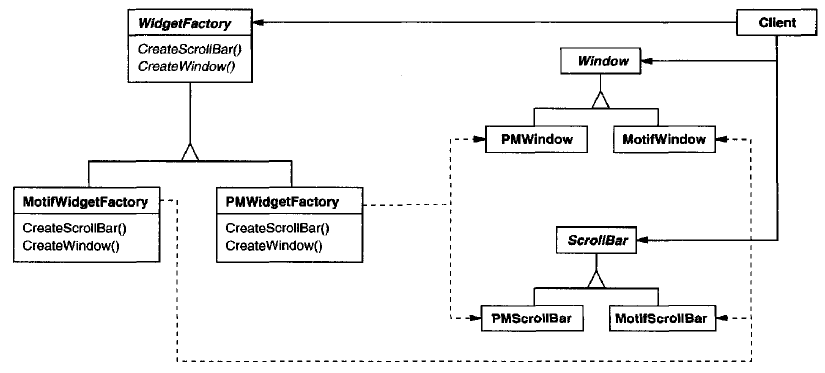
# Also Known As

Kit

# Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. To be portable across look-and-f eel standards, an application should not hard-code its widgets for a particular look and feel.

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-f eel 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.

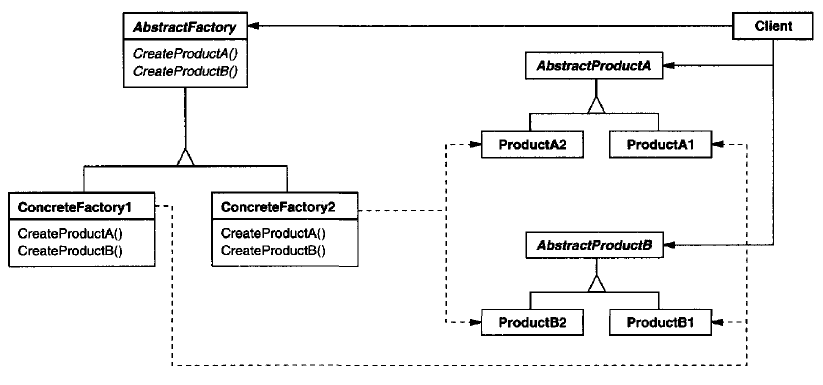


# Applicability

Use the Abstract Factory Pattern 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 th is constraint.
* you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

# Structure



The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the new operator and the concrete, platform-specific, product classes. Each "platform" is then modelled with a Factory derived class.

# Participants

**AbstractFactory (WidgetFactory)**

declares an interface for operations that create abstract product objects.

**ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)**

implements the operations to create concrete product objects.

**AbstractProduct (Window, ScrollBar)**

declares an interface for a type of product object.

**ConcreteProduct (MotifWindow, MotifScrollBar)**

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

AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

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.

# Consequences

The Abstract Factory pattern has the following benefits and liabilities:

**It isolates concrete classes.** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes.

**It makes exchanging product families easy.** The class of a concrete factory appears only once in an application - that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory.

**It promotes consistency among products.** When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.

**Supporting new kinds of products is difficult.** Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created.

# Implementation

1. **Factories as singletons:** ConcreteFactory is usually implemented as a Singleton, because an application only needs Its one instance.
2. **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.

If many product families are possible, the concrete factory can be implemented using the Prototype pattern. The concrete factory is initialized with a prototypical instance of each product in the family, and it creates a new product by cloning its prototype. The Prototype-based approach eliminates the need for a new concrete factory class for each new product family.

1. **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



# Known Uses

# 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.

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

# References

Book: Design Patterns Elements of Reusable Object-Oriented Software

Book: Head First: Design Patterns

<https://sourcemaking.com/design_patterns/abstract_factory>

# END