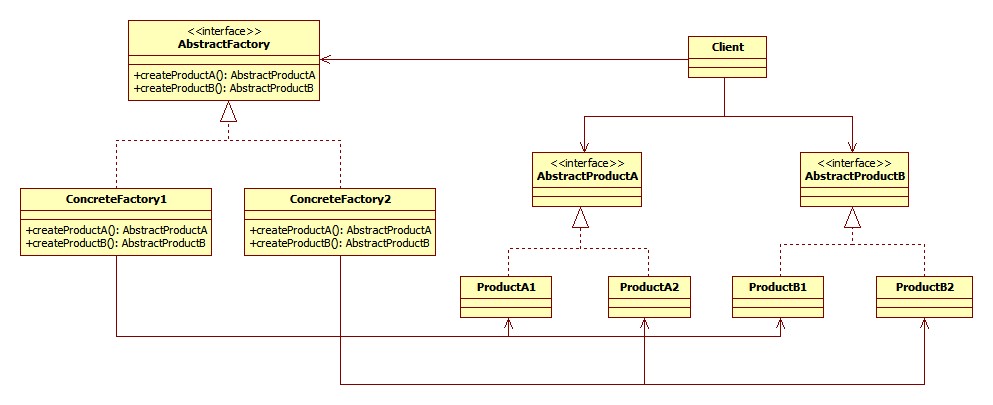
**Abstract Factory pattern**

1. **Description**

Abstract Factory pattern provides an interface for creating a family of related and dependent objects without specifying their concrete classes.

1. **Implementation**



The classes that participate to the Abstract Factory pattern are:

**AbstractFactory** - declares a interface that returns abstract products.

**ConcreteFactory** - implements AbstractFactory and is responsible for creating concrete products.

**AbstractProduct** - declares an interface for a type of product family.

**Product** - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.

**Client** - uses AbstractFactory and AbstractProduct interface to do get the product.

The AbstractFactory interface is the one that decides and creates the actual type of the concrete object and returns an abstract pointer to that object. This helps establish an encapsulation between client and objects, hiding the client from knowing anything about how the object is created and what factory it is going to use. This implies that there is no need for including any class declarations relating to the concrete objects which are accessed by the client only through the abstract interface.

Another objective of Abstract Factory pattern is that when a new concrete object type is needed, it is only required to make the client code use a different factory. This makes it much easier than instantiating a new type, which requires changing the code wherever a new object is created.

1. **Advantages and disadvantages**

**Advantages**

* It isolates construction classes from the client.
  + AbstractFactory is used to control objects creation.
  + Client gets the object through AbstractFactory interface.
* Exchanging product families is easy.
  + None of the client code breaks because it is encapsulated by AbstractFactory interface and composition usage.
  + Because the abstract factory creates a complete family of products, only one product family changes when the concrete factory is changed.
* It promotes consistency among products.
  + It is the concrete factory’s job to make sure that the right products are used together.

**Disadvantages**

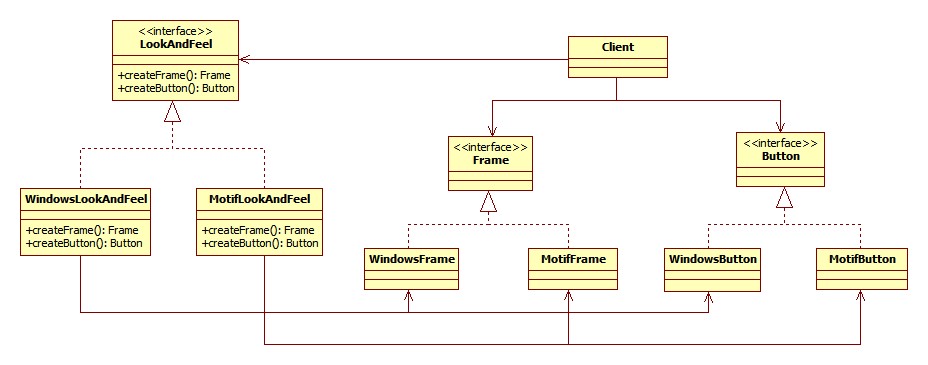
* Adding new products requires extending the abstract interface which causes all of its concrete classes to change:
  + New abstract product class is added
  + New product implementation is added
  + Abstract factory interface is extended
  + Derived concrete factories must implement the extensions
  + Client has to be extended to use the new product

**Usage**

* When the system needs to be independent of how its products are created composed and represented.
* When a family of product exists and need to be used together.
* The system need to expose product family interface not implementation.

1. **Example:** Look and Feel

A GUI framework often contains different look and feel themes such as Motif and Windows look. Each theme specifies different looks and behaviors of controls like frame and button. . In order to avoid the confusion between each type of control we define an abstract class **LookAndFeel interface which has concrete classes like WindowsLookAndFeel and MotifLookAndFeel. These classes goal is to create controls that are configured to their look and flavor.**



1. **Detailed Implementation**

As we only need a single ConcreteFactory object; people often combine it with Singleton to ensure only one instance of a family of products is created.

In addition, Prototype pattern can be used in each family of product. This help increase performance and extensibility. One reason is prototype creates new objects by cloning one of a few stored prototypes. This makes instantiate new object quickly. Another reason is when a new product is needed instead of creating it, the existing prototype is cloned. This approach eliminates the need for a new concrete factory for each new family of products.

1. **Alternative**

One can use Builder pattern instead of Abstract Factory because it has more flexibility. Instead of hiding all operation of how an object is created, Builder let client decide which attributes and behaviors an object should have when it is built. This offers the creation of complex objects and different representations of new objects. Builder pattern is suitable when it comes to encapsulate the instantiation of objects. Also, it is less error-prone as user will know what they are passing because of explicit method call. However, Builder pattern might become a very complex design for solving small problem. To sum up, it depends on what kind of system and construction that whether a design is suitably applied.

1. **Reference**

JavaRevisited 2012, ‘Builder Design pattern in Java - Example Tutorial’, 29 June, viewed 18 Nov 2012, <http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html>

Kulkarn V. 2009, ‘Understanding Factory Method and Abstract Factory Patterns’, 5 July, viewed 22 Nov 2012, <http://www.codeproject.com/Articles/35789/Understanding-Factory-Method-and-Abstract-Factory>

Microsoft 2012, ‘Creational Patterns: Prototype, Factory Method, and Singleton’, viewed 18 Nov 2012, <http://msdn.microsoft.com/en-us/library/orm-9780596527730-01-05.aspx>

OODesign 2012, ‘Abstract Factory’, OODesign.com, viewed 16 Nov 2012, <http://www.oodesign.com/abstract-factory-pattern.html>

OODesign 2012, ‘Builder Pattern’, OODesign.com, viewed 22 Nov 2012, <http://www.oodesign.com/builder-pattern.html>

Sourcemaking 2012, ‘Abstract Factory Design Pattern’, Sourcemaking – Teaching IT professionals, viewed 16 Nov 2012, <http://sourcemaking.com/design_patterns/abstract_factory>