**Abstract Factory Design Pattern**

Abstract Factory design pattern is used to create families of objects.

**Intent**

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

The intent is to provide an interface through which families of related or dependent objects can be created. Also while creation, the concrete classes of those objects should not be specified.

**Problem**

* Instantiation of families of related objects.

Instantiation of objects in application will require hard coding of classes which will make it difficult to manage and modify later. Also the objects are related to a family and will be used together, so the change of family will be cumbersome. There is another problem, when we try to add new family and new product, we may end up creating objects based on multiple scenarios using if, switch as the number of families and their objects grow.

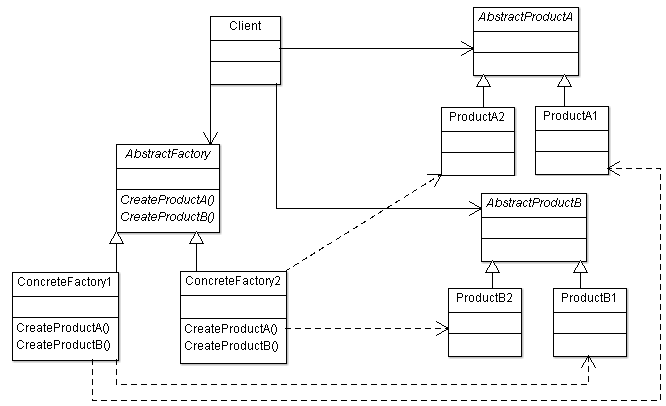
**Solution**

* Come up with a way to create the family of objects.
* Separate the instantiation of objects from use.

**Where it is applicable?**

* We have to create set of objects which are going to work together.
* Supporting different families like system, OS or different versions of product.
* Creation of products has to be separated from use.
* Interface has to be provided to outside world; and there is a need to hide implementation like in case of libraries, frameworks, plug-ins.

**Structure**



**Participant classes**

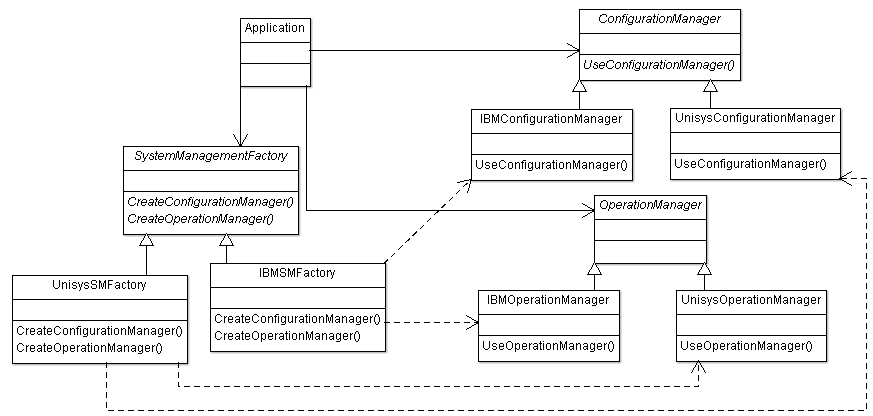
* **AbstractFactory** provides interface for creating product objects.
* **ConcreteFactory** classes derived from the AbstractFactory implements the methods for creating product objects. Each concrete class represents one family, so we can say ConcreteFactory1 has the methods to create objects for Series 1 product objects and ConcreteFactory2 has the methods to create objects for Series 2 product objects.
* **AbstractProduct** classes provides interface for operations on Products.
* **Product** classes are concrete classes derived from AbstractProduct classes to implement the methods for Product.
* **Client** uses the Factory and Product.

**How they work together?**

* ConcreteFactory creates the family of products. ConcreteFactory1 class methods are creating series 1 products and ConcreteFactory2 class methods are creating series 2 products.
* So you can see client does not have any information of concrete product classes for creating objects as it is in method of ConcreteFactory class. Also each concrete factory is creating set of objects for a family and Client is using these objects, so here the instantiation is separated from use.
* So if tomorrow we have to replace the family then it will be easy as the concrete factory is used only at one place in application and we will get another set of product for a family by changing the concrete factory.
* How about adding of a new product. This looks complicated as AbstractFactory and ConcreteFactory will require adding method for each product, so can we think of adding methods dynamically, may be but that will require separate discussion.
* If you see here each family requires one ConcreteFactory, so ConcreteFactory can be implemented as Singleton.

**Example**

We want to create set of objects to manage the system. The families are Unisys and IBM.



SystemManagementFactory is the abstract class which provides interface for creating set of objects for a family. UnisysSMFactory is the concrete class which implements methods for creating objects of Unisys family. IBMSMFactory is the concrete class which implements methods for creating objects of IBM family. Application uses the ConfigurationManager and OperationManager for a particular family.