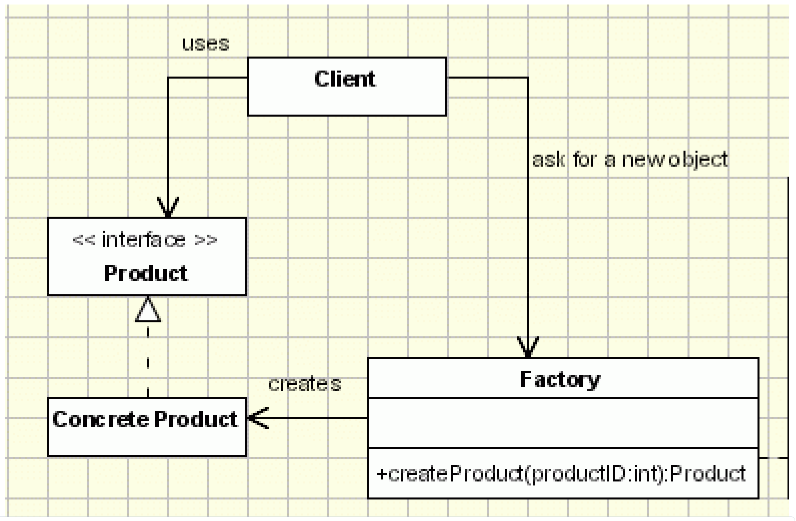
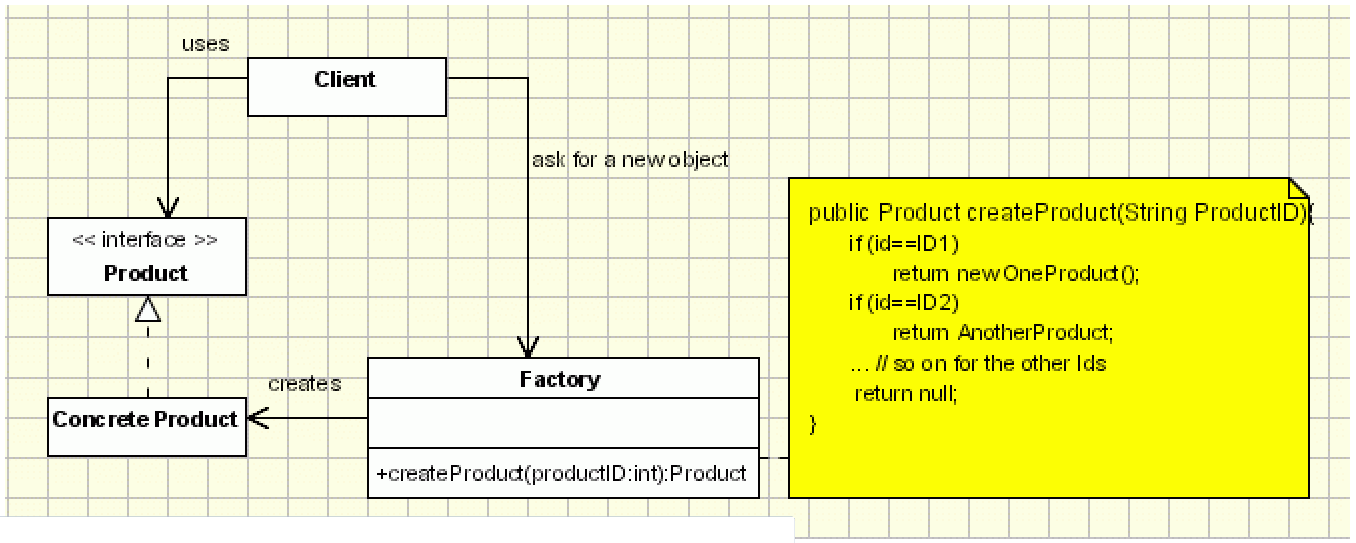
Factory Patterns

Basis of Factory Patterns

Procedural Solution

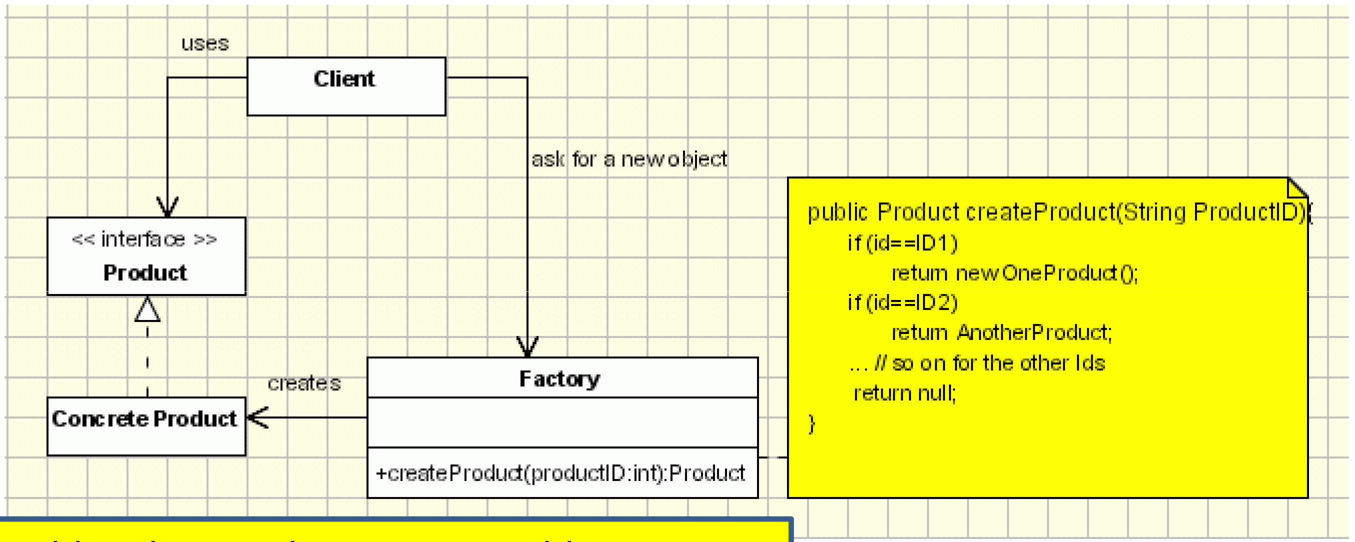


Procedural Solution



```
public Product createProduct(String ProductID){
      if (id==ID1)
            return new OneProduct();
      if (id==ID2)
            return AnotherProduct;
      ... // so on for the other Ids
       return null;
}
```

Procedural Solution



```
public Product createProduct(String ProductID){
      if (id==ID1)
            return new OneProduct();
      if (id==ID2)
            return AnotherProduct;
      ... // so on for the other Ids
       return null;
}
```

The problem here is that once we add a new concrete product call we should modify the Factory class. It is not very flexible and it violates open close principle.

Class Registration Solution

We want to have a reduced coupling between the factory and concrete products.

Since the factory should be unaware of products we have to use reflection **or** move the creation of objects **outside** of the factory to an object aware of the concrete products classes. That would be the concrete class itself.

In the later, a new abstract method is added in the **product abstract class**. Each concrete class will implement this method to create a new object of the same type as itself. We also have to change the registration method such that we'll register concrete product objects instead of Class objects.

Class Registration Solution Abstract Product

```
abstract class Product
{
        public abstract Product createProduct();
        ...
}
```

Class Registration Solution Product Factory

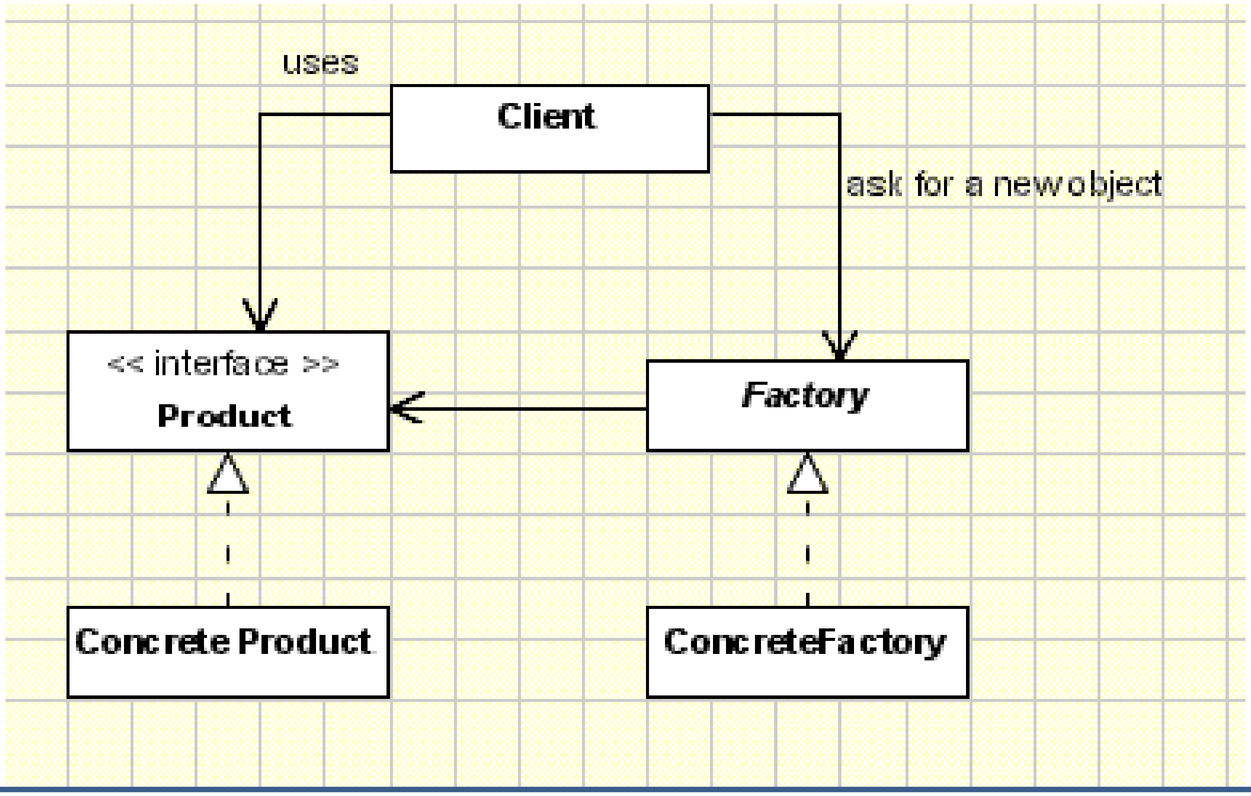
```
class ProductFactory
{
        private HashMap m_RegisteredProducts = new HashMap();

         public void registerProduct(String productID, Product p)    {
                m_RegisteredProducts.put(productID, p);
        }

        public Product createProduct(String productID){

        ((Product)m_RegisteredProducts.get(productID)).createProduct();
        }
        //Otherwise use reflection
}
```

Class Registration Solution
Real Product

```
class Product1 extends Product {
  // ...
  static
  {
  ProductFactory.instance().registerProduct("ID1", new Product1());
            }
            public Product1 createProduct()
            {
                        return new Product1();
            }
            ...
}
```

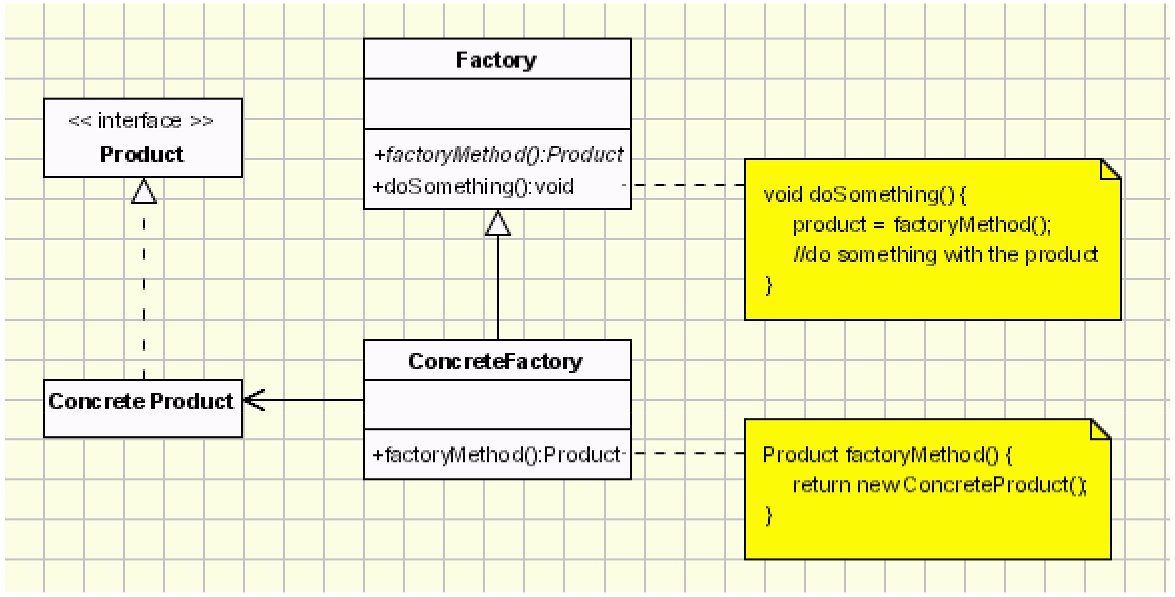
Factory Method

The most intuitive solution to avoid modifying the Factory class is to implement an interface



… or extend it:

Factory Method



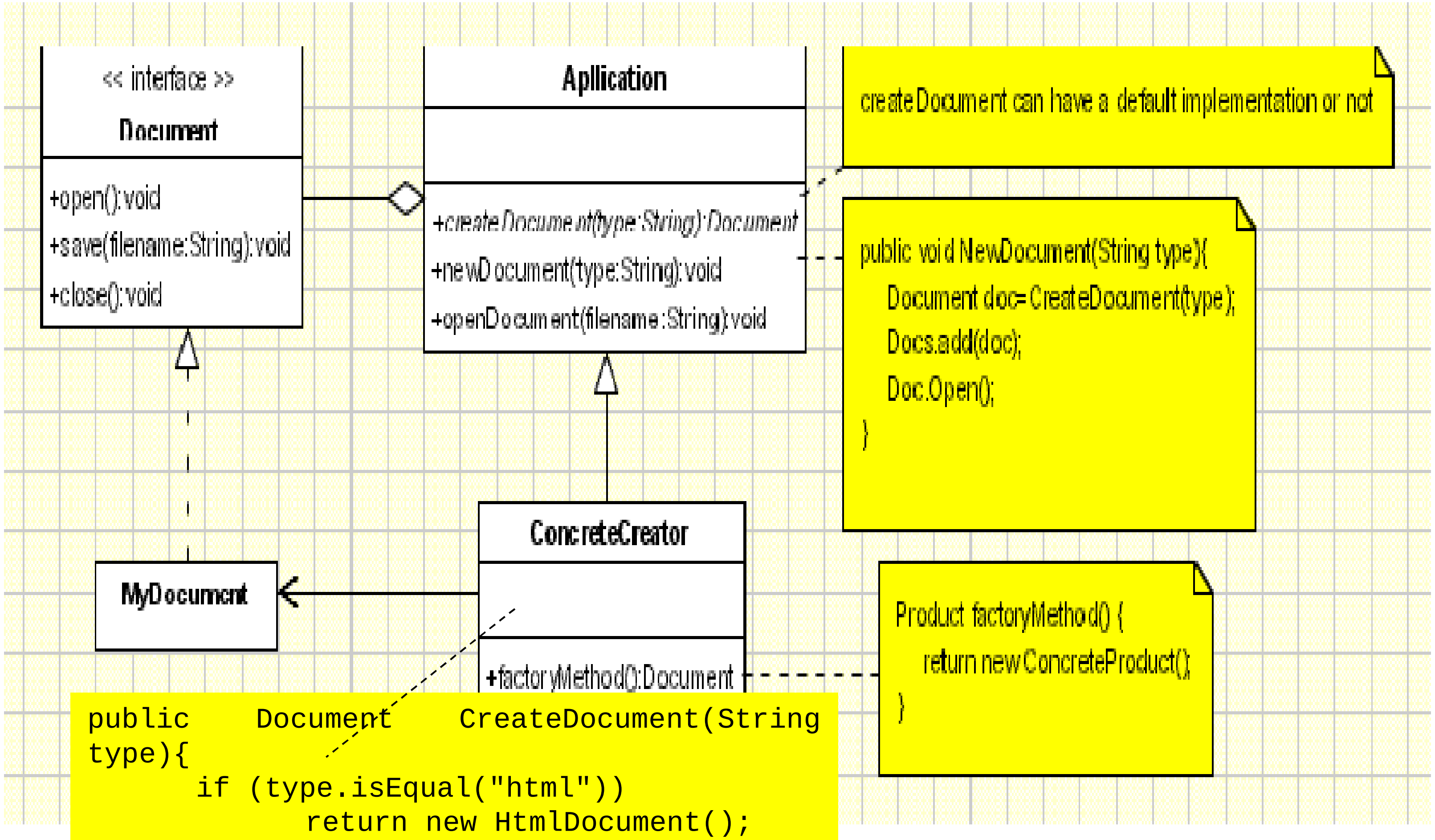
```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

There are some drawbacks over the class registration implementation and not so many advantages:

+ The derived factory method can be changed to perform additional operations when the objects are created (maybe some initialization based on some global parameters ...).
- The factory cannot be used as a singleton.
- Each factory has to be initialized before using it.
- More difficult to implement.
- If a new object has to be added a new factory has to be created.

Factory Method Example



<< interface >>
Document
+open():void
+save(filename:String):void
+close():void
Aplication
+createDocument(type:String):Document
+newDocument(type:String):void
+openDocument(filename:String):void
createDocument can have a default implementation or not
public void NewDocument(String type){
Document doc=CreateDocument(type);
Docs.add(doc);
Doc.Open();
}
ConcreteCreator
+factoryMethod():Document
MyDocument
Product factoryMethod() {
return new ConcreteProduct();
}

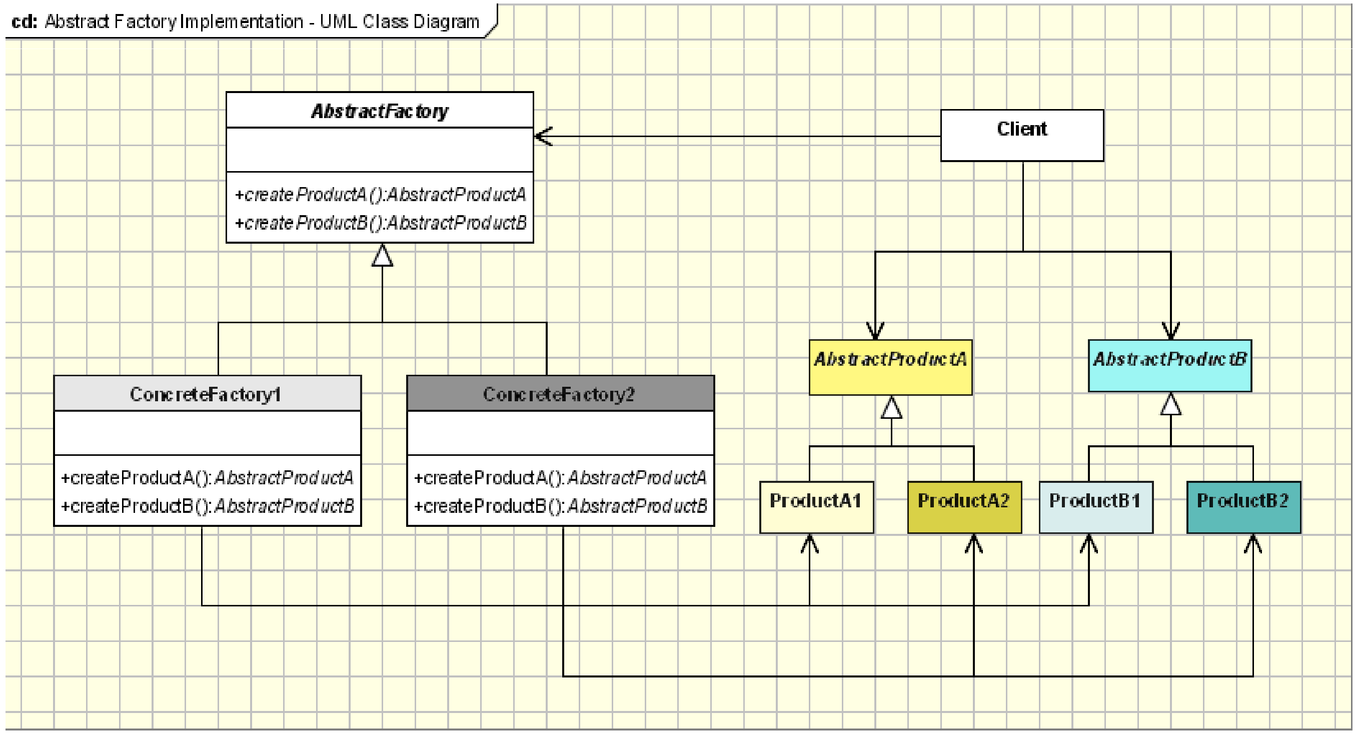
```
public    Document    CreateDocument(String
type){
        if (type.isEqual("html"))
                return new HtmlDocument();
        if (type.isEqual("proprietary"))
                return new MyDocument();
        if (type.isEqual("pdf"))
                return new PdfDocument ();
```

Factory Method Drawbacks and Benefits

- + The main reason for which the factory pattern is used is that it introduces **a separation between the application and a family of classe**s (it introduces weak coupling instead of tight coupling hiding concrete classes from the application). It provides a simple way of extending the family of products with minor changes in application code.
- + It provides **customization hooks**. When the objects are created directly inside the class it's hard to replace them by objects which extend their functionality. If a factory is used instead to create a family of objects the customized objects can easily replace the original objects, configuring the factory to create them.
- - The factory has to be used for a family of objects. If the classes doesn't extend common base class or interface they can not be used in a factory design template.

Abstract Factory

Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.



cd: Abstract Factory Implementation - UML Class Diagram
AbstractFactory
+createProductA():AbstractProductA
+createProductB():AbstractProductB
Client
ConcreteFactory1
+createProductA():AbstractProductA
+createProductB():AbstractProductB
ConcreteFactory2
+createProductA():AbstractProductA
+createProductB():AbstractProductB
AbstractProductA
AbstractProductB
ProductA1
ProductA2
ProductB1
ProductB2

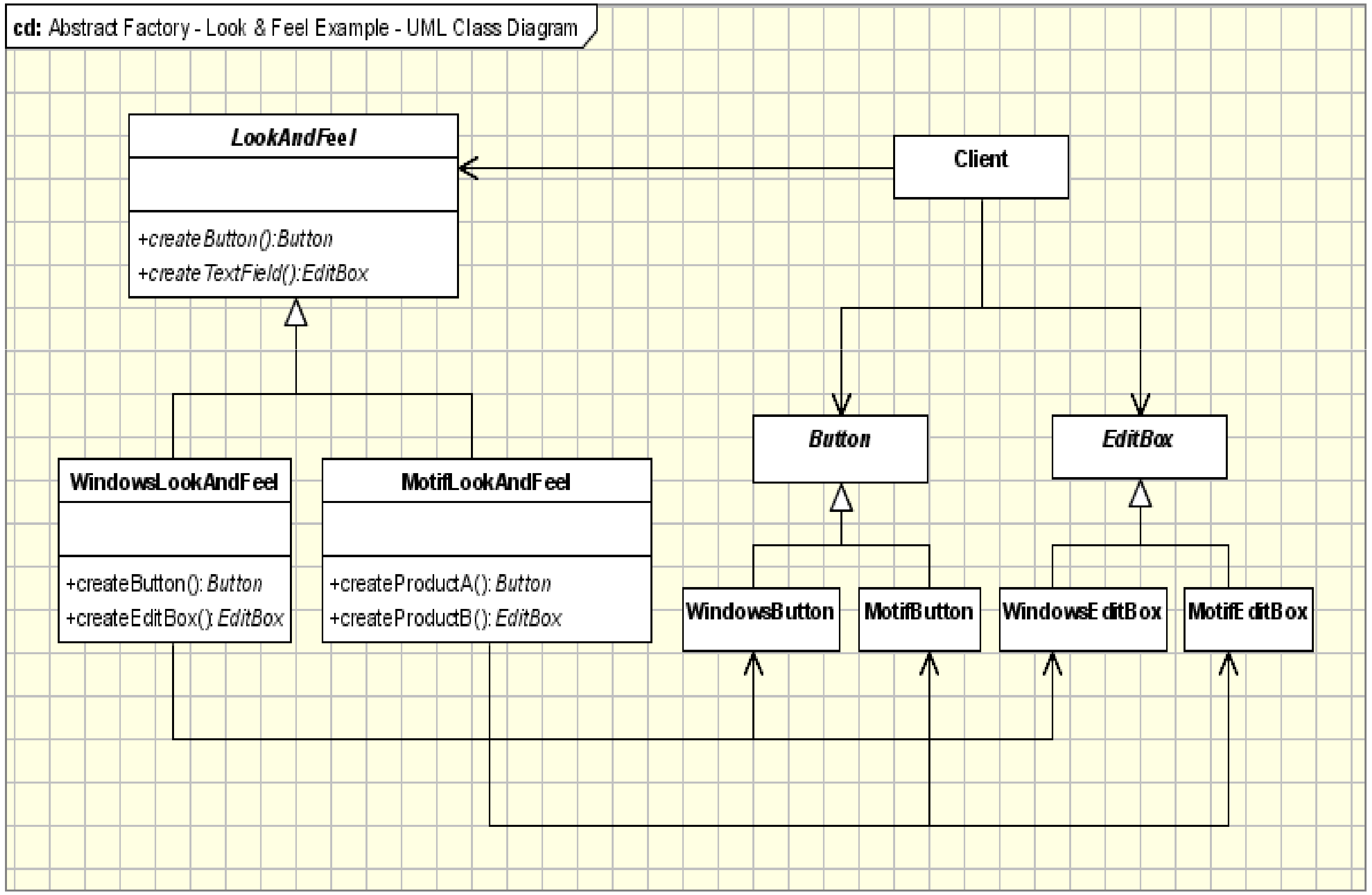
Abstract Factory

1. The fact that the factory returns an abstract pointer to the created object means that the client doesn't have knowledge of the object's type. This implies that *there is no need for including any class declarations relating to the concrete type, the client dealing at all times with the abstract type*. The objects of the concrete type, created by the factory, are accessed by the client only through the abstract interface.

Abstract Factory

2. The second implication of this way of creating objects is that when the adding new concrete types is needed, all we have to do is modify the client code and make it use a different factory, which is far easier than instantiating a new type, which requires changing the code wherever a new object is created.

Abstract Factory Example


cd: Abstract Factory - Look & Feel Example - UML Class Diagram
LookAndFeel
+createButton():Button
+createTextField():EditBox
Client
WindowsLookAndFeel
+createButton(): Button
+createEditBox(): EditBox
MotifLookAndFeel
+createProductA(): Button
+createProductB(): EditBox
Button
EditBox
WindowsButton
MotifButton
WindowsEditBox
MotifEditBox