Abstract factory pattern

From Wikipedia, the free encyclopedia

The **abstract factory pattern** provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.[1] In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.[1] This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.[2]

An example of this would be an abstract factory class `DocumentCreator` that provides interfaces to create a number of products (e.g. `createLetter()` and `createResume()`). The system would have any number of derived concrete versions of the `DocumentCreator` class like `FancyDocumentCreator` or `ModernDocumentCreator`, each with a different implementation of `createLetter()` and `createResume()` that would create a corresponding object like `FancyLetter` or `ModernResume`. Each of these products is derived from a simple abstract class like `Letter` or `Resume` of which the client is aware. The client code would get an appropriate instance of the `DocumentCreator` and call its factory methods. Each of the resulting objects would be created from the same `DocumentCreator` implementation and would share a common theme (they would all be fancy or modern objects). The client would only need to know how to handle the abstract `Letter` or `Resume` class, not the specific version that it got from the concrete factory.

A **factory** is the location of a concrete class in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage and to create families of related objects without having to depend on their concrete classes.[2] This allows for new derived types to be introduced with no change to the code that uses the base class.

Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code. Additionally, higher levels of separation and abstraction can result in systems which are more difficult to debug and maintain.

Contents

Definition

The essence of the Abstract Factory Pattern is to "Provide an interface for creating families of related or dependent objects without specifying their concrete classes.".[3]

Usage

The *factory* determines the actual *concrete* type of object to be created, and it is here that the object is actually created (in C++, for instance, by the **new** operator). However, the factory only returns an *abstract* pointer to the created concrete object.

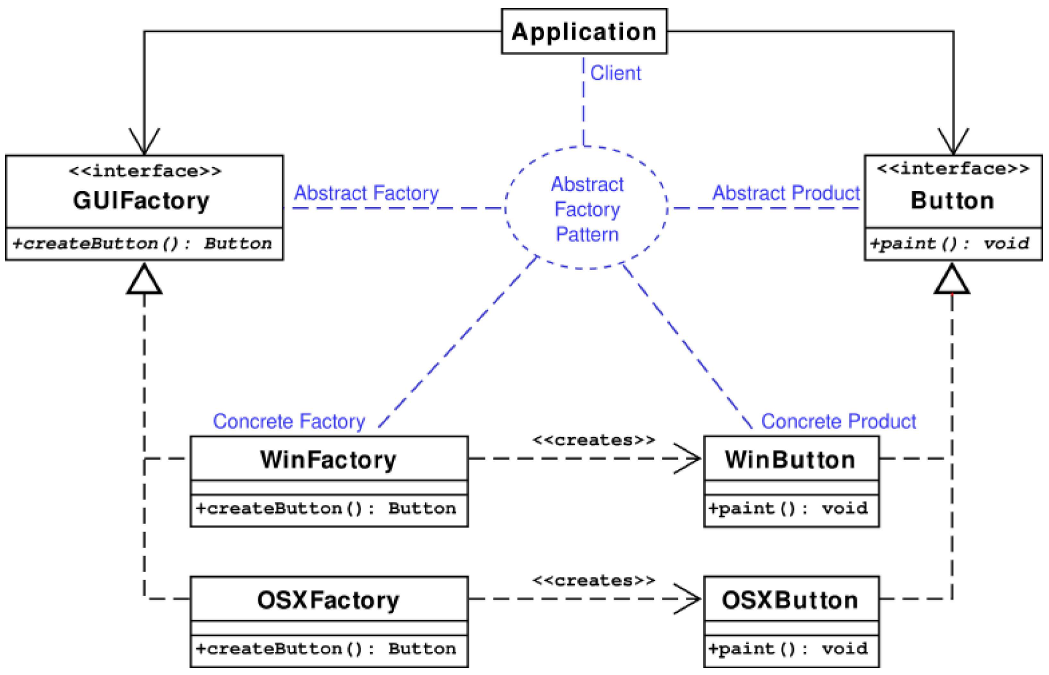
This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.[4]

As the factory only returns an abstract pointer, the client code (that requested the object from the factory) does not know — and is not burdened by — the actual concrete type of the object that was just created. However, the type of a concrete object (and hence a concrete factory) is known by the abstract factory; for instance, the factory may read it from a configuration file. The client has no need to specify the type, since it has already been specified in the configuration file. In particular, this means:

- The client code has no knowledge whatsoever of the concrete type, not needing to include any header files or class declarations related to it. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface.[5]
- Adding new concrete types is done by modifying the client code to use a different factory, a modification that is typically one line in one file. The different factory then creates objects of a *different* concrete type, but still returns a pointer of the *same* abstract type as before — thus insulating the client code from change. This is significantly easier than modifying the client code to instantiate a new type, which would require changing *every* location in the code where a new object is created (as well as making sure that all such code locations also have knowledge of the new concrete type, by including for instance a concrete class header file). If all factory objects are stored globally in a singleton object, and all client code goes through the singleton to access the proper factory for object creation, then changing factories is as easy as changing the singleton object.[5]

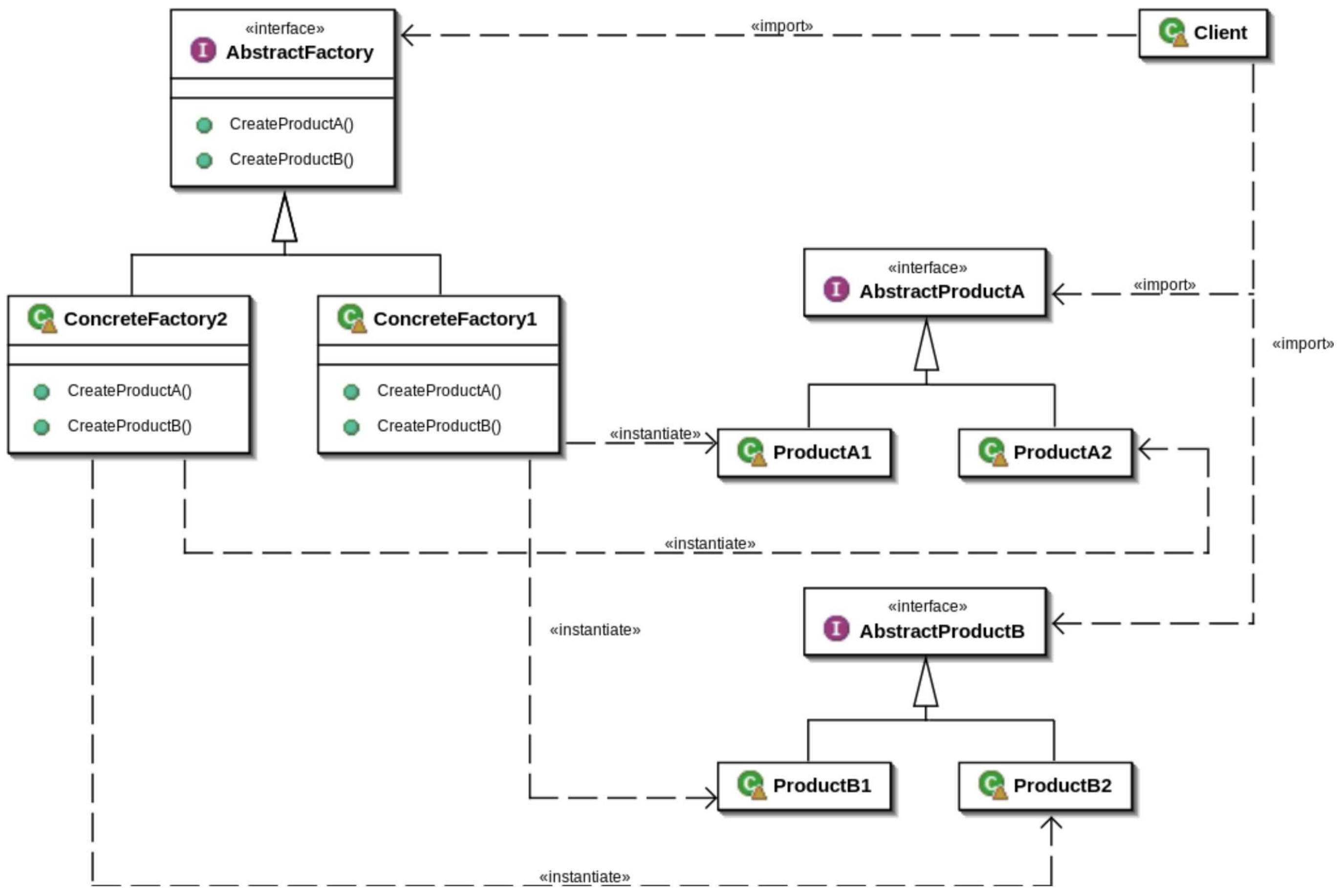
Structure

Class Diagram

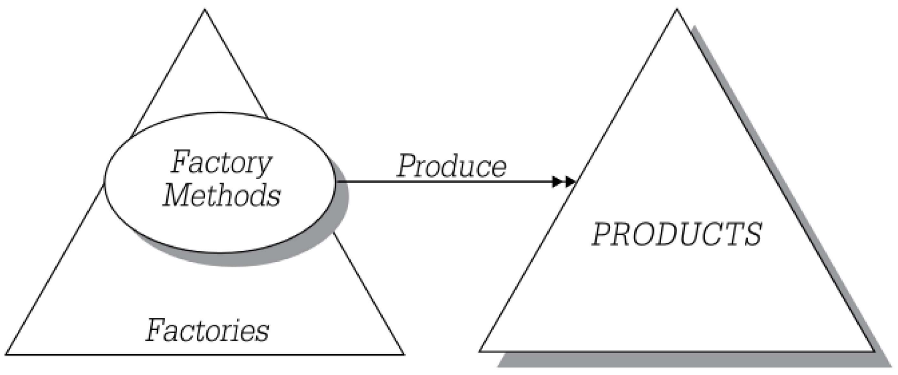


The method `createButton` on the `GuiFactory` interface returns objects of type `Button`. What implementation of `Button` is returned depends on which implementation of `GuiFactory` is handling the method call.

UML Class Diagram



Lepus3 chart (legend (http://www.lepus.org.uk/ref/legend/legend.xml))



Pseudocode

It should render a button in either a Windows style or Mac OS X style depending on which kind of factory was used. Note that the Application has no idea what kind of GUIFactory it is given or even what kind of Button that factory creates.

```
interface Button is
  method paint()

interface GUIFactory is
  method createButton()
      output:  a button
```

```
class WinFactory implementing GUIFactory is
  method createButton() is
      output:  a Windows button
    Return a new WinButton

class OSXFactory implementing GUIFactory is
  method createButton() is
      output:  an OS X button
    Return a new OSXButton

class WinButton implementing Button is
  method paint() is
    Render a button in a Windows style

class OSXButton implementing Button is
  method paint() is
    Render a button in a Mac OS X style

class Application is
  constructor Application(factory) is
      input:  the GUIFactory factory used to create buttons
    Button button := factory.createButton()
    button.paint()

Read the configuration file
If the OS specified in the configuration file is Windows, then
  Construct a WinFactory
  Construct an Application with WinFactory
else
  Construct an OSXFactory
  Construct an Application with OSXFactory
```

C# Example

```
interface IButton
{
    void Paint();
}

interface IGUIFactory
{
    IButton CreateButton();
}

class WinFactory : IGUIFactory
{
    public IButton CreateButton()
    {
        return new WinButton();
    }
}

class OSXFactory : IGUIFactory
{
    public IButton CreateButton()
    {
        return new OSXButton();
    }
}

class WinButton : IButton
{
    public void Paint()
    {
        //Render a button in a Windows style
    }
}

class OSXButton : IButton
{
    public void Paint()
    {
        //Render a button in a Mac OS X style
    }
}

class Program
{
    static void Main()
    {
        var appearance = Settings.Appearance;

        IGUIFactory factory;
        switch (appearance)
        {
            case Appearance.Win:
                factory = new WinFactory();
                break;
            case Appearance.OSX:
                factory = new OSXFactory();
                break;
            default:
                throw new System.NotImplementedException();
        }

        var button = factory.CreateButton();
        button.Paint();
    }
}
```

PHP Example

```
interface ButtonInterface
{
    public function Paint();
}

interface GUIFactoryInterface
{
    public function CreateButton();
}

class WinFactory implements GUIFactoryInterface
{
    public function CreateButton()
    {
        return new WinButton();
    }
}

class OSXFactory implements GUIFactoryInterface
{
    public function CreateButton()
    {
        return new OSXButton();
    }
}

class WinButton implements ButtonInterface
{
    public function Paint()
    {
        echo "Windows Button";
    }
}

class OSXButton implements ButtonInterface
{
    public function Paint()
    {
        echo "OSX Button";
    }
}

$appearance = "osx";

$factory = NULL;

switch ($appearance) {
    case "win":
        $factory = new WinFactory();
        break;
    case "osx":
        $factory = new OSXFactory();
        break;
    default:
        break;
}

$button = $factory->CreateButton();
$button->Paint();
```

See also

- Concrete class
- Software design pattern
- Factory method pattern
- Object creation

References

1. Freeman, Eric; Robson, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike, eds. "Head First Design Patterns" (paperback) **1**. O'REILLY: 156. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.
2. Freeman, Eric; Robson, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike, eds. "Head First Design Patterns" (paperback) **1**. O'REILLY: 162. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.
3. Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (2009-10-23). "Design Patterns: Abstract Factory". informIT. Archived from the original on 2009-10-23. Retrieved 2012-05-16. "Object Creational: Abstract Factory: Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes."
4. Veeneman, David (2009-10-23). "Object Design for the Perplexed". The Code Project. Archived from the original on 2011-09-18. Retrieved 2012-05-16. "The factory insulates the client from changes to the product or how it is created, and it can provide this insulation across objects derived from very different abstract interfaces."
5. "Abstract Factory: Implementation". OODesign.com. Retrieved 2012-05-16.

External links

- Media related to Abstract factory at Wikimedia Commons
- Abstract Factory (http://www.lepus.org.uk/ref/companion/AbstractFactory.xml) UML diagram + formal specification in LePUS3 and Class-Z (a Design Description Language)
- Abstract Factory (http://patterns.pl/abstractfactory.html) Abstract Factory implementation example

The Wikibook *Computer Science Design Patterns* has a page on the topic of: ***Abstract Factory in action***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Abstract_factory_pattern&oldid=727967992"

Categories: Software design patterns