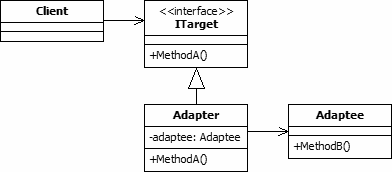
<http://www.blackwasp.co.uk/GofPatterns.aspx>

<http://www.dofactory.com/Patterns/Patterns.aspx>

The **Adapter Pattern** is a design pattern that is used to allow two incompatible types to communicate. Where one class relies upon a specific interface that is not implemented by another class, the adapter acts as a translator between the two types.

This is a structural pattern as it defines a manner for creating relationships between classes. The adapter design pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.



The adapter class will implement the ITarget interface in this case. Method A will call method B. The adapter is needed because the Adaptee class does not implement ITarget.

public class Client

{

    private ITarget \_target;

    public Client(ITarget target)

    {

        \_target = target;

    }

    public void MakeRequest()

    {

        \_target.MethodA();

    }

}

public interface ITarget

{

    void MethodA();

}

public class Adaptee

{

    public void MethodB()

    {

        Console.WriteLine("MethodB called");

    }

}

public class Adapter : ITarget

{

    Adaptee \_adaptee = new Adaptee();

    public void MethodA()

    {

        \_adaptee.MethodB();

    }

}

-------------------------------------------

The **Strategy Pattern** is a design pattern that enables an [algorithm](http://en.wikipedia.org/wiki/Algorithm)'s behavior to be selected at runtime. The strategy pattern:

* defines a family of algorithms,
* encapsulates each algorithm, and
* makes the algorithms interchangeable within that family.

Strategy lets the algorithm vary independently from clients that use it

-------------------------------------------

http://www.oodesign.com/factory-pattern.html

A **factory** is the location of a concrete class in the code at which [objects are constructed](http://en.wikipedia.org/wiki/Object_creation). 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]](http://en.wikipedia.org/wiki/Abstract_factory_pattern#cite_note-bullet_points-2) This allows for new [derived types](http://en.wikipedia.org/wiki/Subtype) to be introduced with no change to the code that uses the [base class](http://en.wikipedia.org/wiki/Base_class).

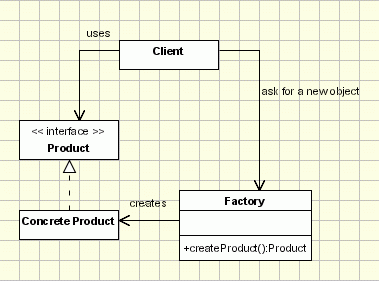
Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at [runtime](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)). However, employment of this pattern, as with similar [design patterns](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)), 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.

The Factory Design Pattern is probably the most used design pattern in modern programming languages like Java and C#. It comes in different variants and implementations.

## Intent

* creates objects without exposing the instantiation logic to the client.
* refers to the newly created object through a common interface

## Implementation

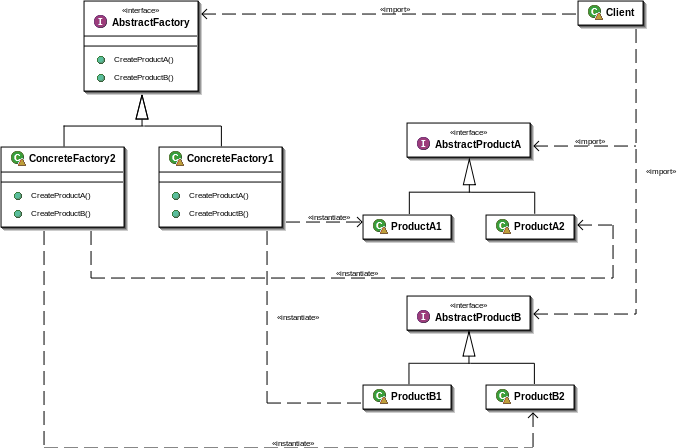


-------------------------------------------

The **abstract factory pattern** provides a way to encapsulate a group of individual [factories](http://en.wikipedia.org/wiki/Factory_object) that have a common theme without specifying their concrete classes.[[1]](http://en.wikipedia.org/wiki/Abstract_factory_pattern#cite_note-abstract_factory-1) In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic [interface](http://en.wikipedia.org/wiki/Interface_(computer_science)) of the factory to create the concrete [objects](http://en.wikipedia.org/wiki/Object_(computer_science)) that are part of the theme. The [client](http://en.wikipedia.org/wiki/Client_(computing)" \o "Client (computing)) 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]](http://en.wikipedia.org/wiki/Abstract_factory_pattern#cite_note-abstract_factory-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.

### Normally, if you use the Factory pattern, you will create objects in a Factory. The problem arises when you have multiple implementation of a given class (or classes). Now, those multiple implementations are grouped. You will use the Abstract Factory pattern when you have a factory, but you would like to group the creating of objects per group.

### UML Class Diagram[[edit](http://en.wikipedia.org/w/index.php?title=Abstract_factory_pattern&action=edit&section=5)]

[](http://en.wikipedia.org/wiki/File:Abstract_factory_UML.svg)

# Task-based Asynchronous Pattern (TAP)

The Task-based Asynchronous Pattern (TAP) is based on the [System.Threading.Tasks.Task](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) and[System.Threading.Tasks.Task<TResult>](https://msdn.microsoft.com/en-us/library/dd321424.aspx) types in the [System.Threading.Tasks](https://msdn.microsoft.com/en-us/library/system.threading.tasks.aspx) namespace, which are used to represent arbitrary asynchronous operations. TAP is the recommended asynchronous design pattern for new development.

TAP uses a single method to represent the initiation and completion of an asynchronous operation. This is in contrast to the **Asynchronous Programming Model** (APM or **IAsyncResult**) pattern, which requires Begin andEnd methods, and in contrast to the **Event-based Asynchronous Pattern (EAP),** which requires a method that has the Async suffix and also requires one or more events, event handler delegate types, and **EventArg**-derived types. Asynchronous methods in TAP include the Async suffix after the operation name; for example, GetAsyncfor a get operation. If you're adding a TAP method to a class that already contains that method name with theAsync suffix, use the suffix TaskAsync instead. For example, if the class already has a GetAsync method, use the name GetTaskAsync.

The TAP method returns either a [System.Threading.Tasks.Task](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) or a [System.Threading.Tasks.Task<TResult>](https://msdn.microsoft.com/en-us/library/dd321424.aspx), based on whether the corresponding synchronous method returns void or a type **TResult**.

Other patterns to learn:

\* Strategy  
  
\* Observer  
  
\* Decorator  
  
\* Factory  
  
\* Singleton  
  
\* Command  
  
\* Adapter and Facade  
  
\* Template  
  
\* Iterator and Composite  
  
\* State  
  
\* Proxy  
  
\* Compound Patterns  
  
\* Living better with Patterns