Design Patterns

Article 1

Author: Douglas Minnaar

Level: Novice – Intermediate

Prerequisites:

- Understanding of Object Oriented Programming
- The examples will be demonstrated in C#.NET therefore an understanding of C#.NET code is required

Please download the code here

Summary

It is not the intent of the Design Pattern Series to focus on providing a theoretical knowledge dump of all there is to know about design patterns. There are many books that do that already. Instead, this series will focus on providing lots of practical examples. However, there will be some theory to help address important points concerning design patterns. I use the theory of design patterns mostly as a guide and instead make references to good design pattern books for more detail explanation. Think of this series as a 'Design Patterns by example' series. The target audience is that of a novice to intermediate software developer.

Intent

Before delving into the world of software design patterns, it is important to understand the reason for this article series. There are many good books available that address the topic of design patterns. However, the problem that I have experienced with many design pattern books is that they fail to provide ample examples that would allow one to better understand design patterns. Also, I have found many books to be very theoretical. I am not implying that theory is a bad thing. However, there is an issue pertaining to matters that have been theoretically over emphasized. The issue is that, it is often found by many, through experience, that when theory and practical application collide, the results are quite different from what was theorized. This is nobody's fault, just simple laws that govern the world in which we live. Many people find it easier to learn through practical example. Others find theory to be most conducive to learning. The intent of this article series is to provide some theory but more practical examples of design patterns.

The examples that have been chosen for this series are for demonstration purposes only. This means that the examples will illustrate different ways in which different design patterns can be used. It does not mean that one should rush off into ones next development job and apply these examples. These examples provide a foundation from which to build ones design pattern knowledge and understanding.

A word of caution concerning design patterns should be noted. The rule is that one should not try to fit a pattern to a scenario just for the sake of using a design pattern. One will find that when good object-oriented design principles are applied, design patterns often emerge. For many examples in this article series, I have broken the aforementioned rule by intentionally trying to fit a pattern to a scenario. That is why I emphasize that the examples are for demonstration purposes only. I do however provide some examples where design patterns are being used in the .NET framework. I have also chosen a few examples from the Microsoft Patterns & Practices Enterprise Library.

Keeping in line with the true spirit of design patterns, this series will start with the bible of design patterns, namely "Design Patterns by the Gang of Four". Once the foundation is in place, the focus will shift onto the more modern design patterns. Every so often, an anti-pattern will be thrown in for good measure. A better understanding of what this series will provide is illustrated as follows:

- **Part 1**
  - Introduction
- **Part 2**
  - GOF Design Patterns
- **Part 3**
  - Enterprise Design Patterns
- **Part 4**
  - Architectural Patterns

Part 1 - Introduction

The concept of a 'Software Design Pattern' was adopted from the idea of a building design pattern in the field of building design and architecture. For more information, refer to the book 'The Timeless Way of Building by Christopher Alexander'. Within the context of software engineering, a Design Pattern is the official term given to a design concept that encompasses the idea of reuse and communication. Simply stated, Design Patterns are tried and tested solutions to common and recurring challenges. At their core, design patterns are the manifestation of good object-oriented design principles.

Design Patterns are important to understand for the following reasons:

- They give us reuse
- Once a design pattern has emerged to solve a particular problem, that design pattern can be applied countless times on future tasks where a similar problem may arise. It can also be used by countless others who may encounter similar problems.
- The give us a common taxonomy allowing for better communication
- Design patterns allow one to refer to a design concept without explaining the full detail of that concept. How often has one heard a conversation along the lines of the following?

  Developer A: "Remember that thingy we used last time to solve a problem similar to this one."

  Team: "What thingy?"

  Developer A: "That thingy where we allowed only a single instance of an object to exist at any given time"

  Team: "Heh!?"

  Imagine the development team has reached a certain level of maturity in terms of design thinking and design patterns. The conversation would instead sound something like this.

  Developer A: "We should use a Singleton to solve this."

  Team: "A Singleton will work. And we could use a Façade for the Products sub-system."

  As one can see, a Design Pattern speaks a thousand words. It is also far more professional and efficient to speak of well documented concepts than a 'thingy'.

- They give us a higher level of abstraction in terms of analysis and design

  This allows one to think of the larger design in terms of smaller component areas. Therefore, when one looks at the bigger picture, one can visualize it in terms of its constituent parts.

- Fosters design thinking

If one is to understand object-oriented design principles, it is beneficial to understand design patterns. However, if one is to understand design patterns, one must understand object-oriented design. The two concepts (OO Design and Design Patterns) compliment each other. This will intern compliment ones development code.

Part 2 – GOF Design Patterns

The book that has profoundly influenced design pattern theory is Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides. The aforementioned four authors are commonly known as the Gang of Four (GOF). Many, including myself have referred to this book as the Bible of software design patterns. Although many have questioned the relevance of the design patterns catalogued in this book. The fact remains that it helps provide a solid foundation from which to build ones design pattern knowledge. This is where we shall start.

The GOF book catalogued 23 design patterns into 3 groups, namely Creational, Structural and Behavioural. I have adopted the intent of the design patterns as per the GOF book.

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

Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.
Structural	
Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Façade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Behavioural	
Chain Of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allow an object to alter its behaviour when it’s internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

For each of the aforementioned design patterns, I will provide (where possible) a Banking example, an Insurance example, and a Game example. This demonstrates how one might implement the aforementioned design patterns. Not all the design patterns can be demonstrated within the aforementioned examples. I will provide alternative examples in such instances.

Factory Method

Please refer to the GOF book or any other good design pattern book for more detailed information pertaining to the Factory Method.

The intent of Factory Method is to define an interface for creating an object, but allow subclasses to decide which class to instantiate. Factory Method allows a class to defer its instantiation to subclasses.

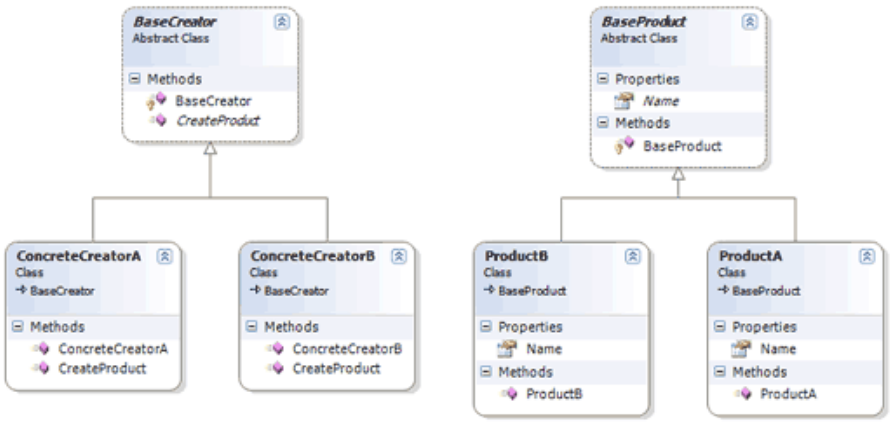
A good design principle is to use an interface driven approach. The Factory Method is helpful in this regard. This is because a Factory Method typically returns an instance of a class that implements an interface or extends an abstract class. This has the affect of minimising the affects of change in a system. The ability of not having to specify a concrete class provides the additional ability to perform dependency injection. In fact, I have used the Factory Method often in terms of performing dependency injection. Also, consider using a Factory Method in the following instances:

- When a class cannot anticipate the objects it must create
- Create an instance of a class based on input parameters (arguments) or what one provides it
- When a class is required to defer the instantiation process to its subclasses

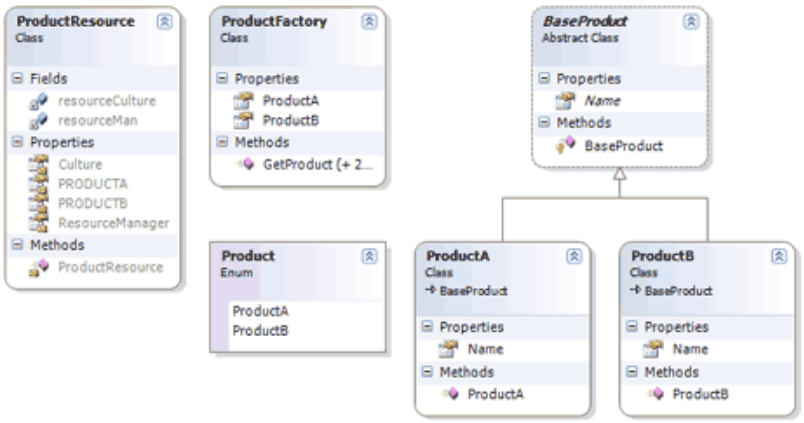
Structure

This example provides the default implementation that one would typically expect for a Factory Method. Additionally, this example demonstrates various ways in which a Factory Method can be implemented.

The following diagram illustrates a typical class diagram that one would expect for a Factory Method implementation.



An alternative to the above diagram is illustrated in the following diagram.



The diagram above illustrates the use of a Factory class that defines various Factory Methods for creating a BaseProduct.

Structure example code

Five possible ways are demonstrated in which one might apply a Factory Method implementation.

1. The default implementation as per the GOF implementation
2. Use .NET Properties as Factory Properties to create a BaseProduct
3. Create BaseProduct based on Product name
4. Create BaseProduct using a Resource file
5. Create BaseProduct using an enumeration

```
public class ProductFactory
{
  // ############################################################
  // ## Method 1 - Use Factory Properties instead of Factory Methods
  // ##
  // ## One can use properties. The advantage of this approach
  // ## is that compile time type checking is possible. Therefore,
  // ## one is guaranteed not to get a run-time invalid cast
  // ## exception. The disadvantage is that for each new type
  // ## that is added, a property must be added
  // ############################################################

  public static BaseProduct ProductA
  {
    get { return new ProductA(); }
  }

  public static BaseProduct ProductB
  {
    get { return new ProductB(); }
  }

  // ############################################################
  // ## Method 2 - Create a product based on a String input
  // ## (product name)
  // ##
  // ## The following approach is more prone to errors. The reason
  // ## for this is that one cannot be guaranteed that 'structure'
  // ## is of a correct type. Therefore, one might recieve a null
  // ## type when invoked. Another disadvantage is that the
  // ## 'Switch' statement can grow out of proportion and become
  // ## difficult to maintain. An advantage of this approach is
  // ## that only a single method is used to create structures.
  // ############################################################

  public static BaseProduct GetProduct(String product)
  {
    switch (product.ToUpper().Trim())
    {
      case ("PRODUCTA"):
        return new ProductA();
      case ("PRODUCTB"):
```

```
        return new ProductB();
      default:
        return null;
    }
  }

  // ##########################################################
  // ## Method 3 - Use a Resource file to create a Product.
  // ## The Resource file contains the Product name as a key
  // ## and the Product type as a value. Using reflection,
  // ## the Product is created based on the type specified
  // ## in the Resource file.
  // ##
  // ## In this example, the variable dummy is exactly that.
  // ## For demo purposes I wanted to keep the method names
  // ## the same. The same disadvantages and advantages as
  // ## above hold true for this method. It has further
  // ## disadvantages in that it must perform additional
  // ## resource file lookups and create types using
  // ## reflection. This can be slower as well as more
  // ## error prone. The advantage is that the types
  // ## are configurable via the resource file.
  // ##########################################################

  public static BaseProduct GetProduct(String product, bool dummy)
  {
    StringBuilder productResourceType = new StringBuilder();

    productResourceType.Append("CodeMentor.Patterns.GOF.Creational.");
    productResourceType.Append("Factory.Structure.ProductResource");

    return ResourcesUtil.GetObject(Type.GetType
(productResourceType.ToString()),
      product.ToUpper().Trim()) as BaseProduct;
  }

  // ############################################################
  // ## Method 4 - Create a Product based on an Enumeration value
  // ##
  // ## The advantage of using this approach is that the
  // ## input to the factory method is an enumeration.
  // ## The enumeration helps define a stronger contract.
  // ## Another disadvantage is that only a single method
  // ## is used to create a structure. The disadvantage
  // ## is that it uses a 'Switch' statement which can
  // ## become a maintenance issue.
  // ############################################################

  public static BaseProduct GetProduct(Product product)
  {
    switch (product)
    {
      case (Product.ProductA):
        return new ProductA();
      case (Product.ProductB):
        return new ProductB();
      default:
        return null;
    }
  }
}
```

The code that is used to create a Product using the resource file and reflection code is demonstrated as follows.

```
public static class ReflectionUtil
{
  public static object GetObject(Type type)
  {
    return Activator.CreateInstance(type);
  }
}
```

```
public static class ResourcesUtil
{
  public static ResourceManager GetResourceManager(Type resourceType)
  {
    return new ResourceManager(resourceType);
  }

  public static object GetObject(Type resourceType, String name)
  {
    object value = null;

    ResourceManager manager = GetResourceManager(resourceType);

    if (manager != null)
    {
      String typeName = manager.GetString(name);

      value = ReflectionUtil.GetObject(Type.GetType(typeName));
    }

    if (value == null)
    {
      throw new Exception(String.Format(
        "Resource not found for '{0}'", name));
    }

    return value;
  }
}
```

The resource file appears as follows.

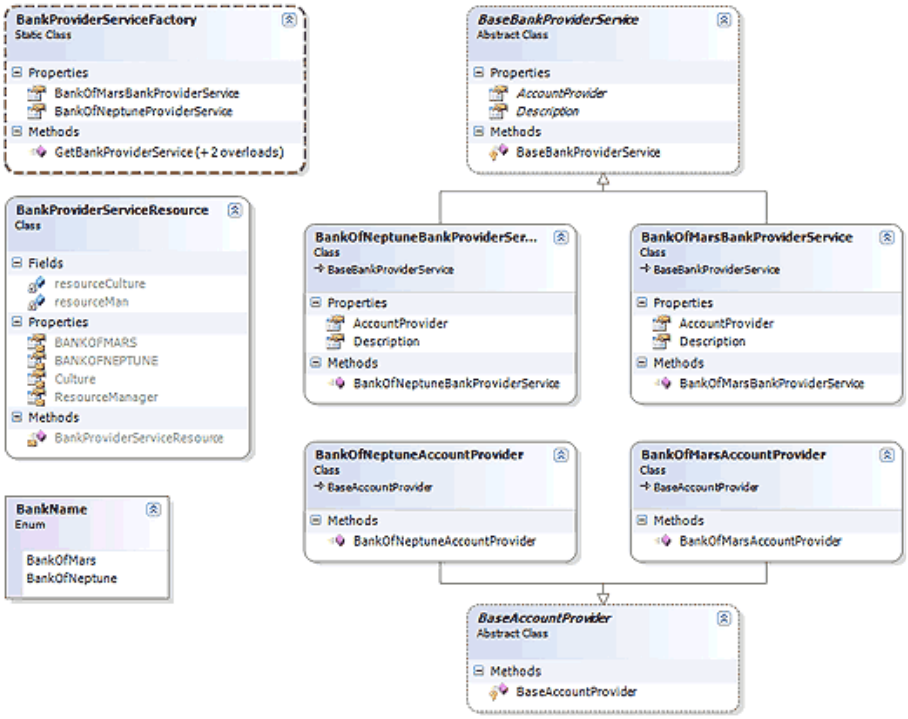
Name	Value
PRODUCTA	CodeMentor.Patterns.GOF.Creational.Factory.Structure.ProductA, CodeMentor.Patterns.GOF.Creational.Factory
PRODUCTB	CodeMentor.Patterns.GOF.Creational.Factory.Structure.ProductB, CodeMentor.Patterns.GOF.Creational.Factory

Please refer to the code in the GOF Design Patterns Solution that accompanies this article.

Solution Code: CodeMentor.Patterns.GOF

Banking Example

Use a Factory Method to create a BankProviderService. A BankProviderService is service that exposes an interface offering various providers e.g. AccountProvider. The AccountProvider Factory property on the BankProviderService intern creates the appropriate account provider. For the BankAccountProvider, instantiation is deferred to the subclasses. A factory (BankProviderServiceFactory) is used to instantiate the BankProviderService instances.

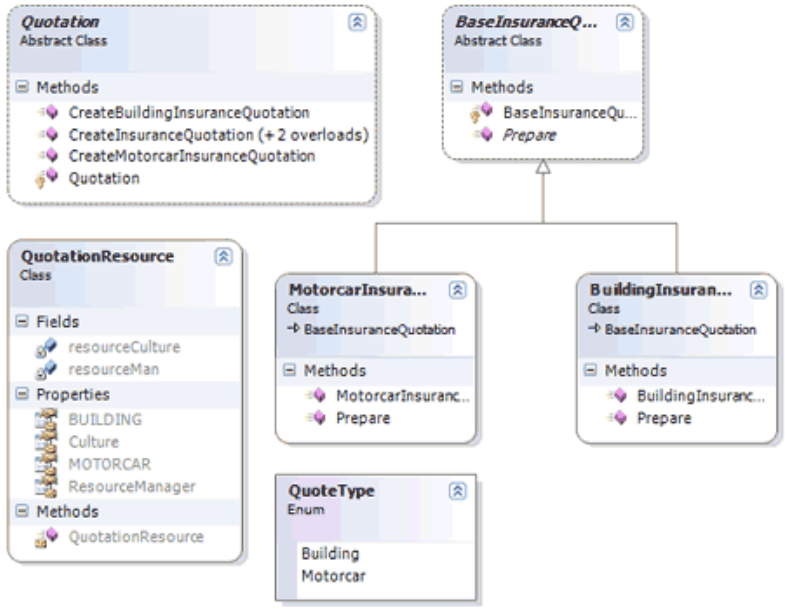


Please refer to the code in the GOF Design Patterns Solution that accompanies this article.

Solution Code: CodeMentor.Patterns.GOF

Insurance Example

Demonstrates the use of a Factory Method to create Building and Motorcar insurance quotations. Quotation exposes various Factory Methods that one might use to create an insurance quotation.

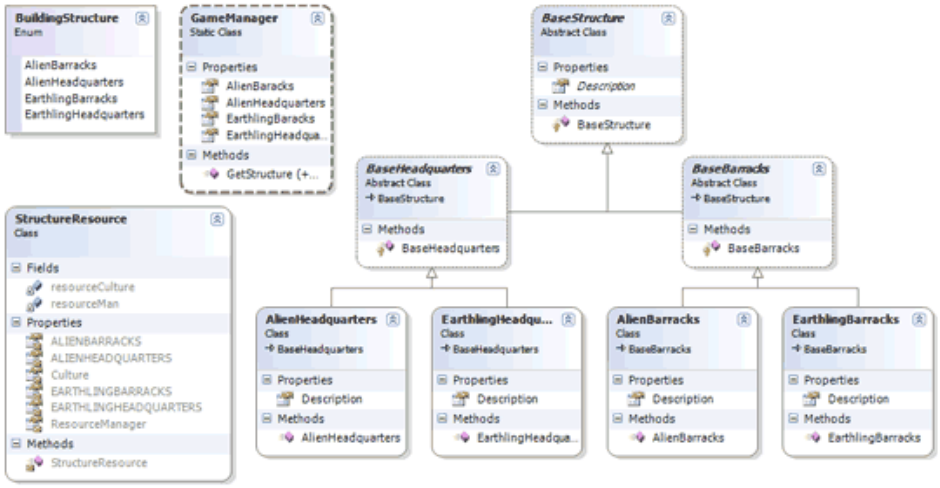


Please refer to the code in the GOF Design Patterns Solution that accompanies this article.

Solution Code: CodeMentor.Patterns.GOF

Game Example

Think of a Real Time Strategy (RTS) game. Think of the structures that one would typically use to create different units. A Factory Method may be a useful pattern in such a scenario. We have two types of structures, namely BaseHeadquarters and BaseBarracks. For each of these structures we have an accompanying earthling and alien structure. The GameManager class helps determine what structure to create by applying the concept of a Factory Method.



Please refer to the code in the GOF Design Patterns Solution that accompanies this article.

Solution Code: CodeMentor.Patterns.GOF

Miscellaneous Examples

.NET Framework

- **System.IO.File.Create() Factory Method**

  Provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects.

```
using (FileStream stream = File.Open("path", FileMode.OpenOrCreate))
{
}
```

- **System.Xml.XmlWriter.Create() Factory Method**

  Represents a writer that provides a fast, non-cached, forward-only means of generating streams or files containing XML data.

```
using (XmlWriter writer = XmlWriter.Create("outputFileName"))
{
}
```

- **System.Convert.To????**

  Converts a base data type to another base data type.

```
int result = System.Convert.ToInt32("10");
```

- **System.Activator**

  Creates an instance of the specified type using the constructor that best matches the specified parameters.

```
public static object GetObject(Type type)
{
  return Activator.CreateInstance(type);
}
```

That's it for Design Patterns Article 1. In Article 2 we will look at the Abstract Factory design pattern. The Abstract Factory will build on the examples from Article 1.

References:

- Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1995