Design Patterns: Modelling in UML and Implementation Using C#

Dr Jie Zhao, Dunstan Thomas Consulting

Summary

Design Patterns are proven design solutions to some common problems we face in software development. The use of design patterns has been a significant movement in object oriented design (OOD) and greatly helps us to deliver high quality, easy to maintain software products faster at lower cost. This article employs the standard software system modelling language UML (Unified Modelling Language) and the .NET prime programming language C# to demonstrate the application of the abstract factory design pattern.

Why Design patterns?

Design patterns are reusable assets that encapsulate the OOD principles such as low coupling and high cohesion and embody design knowledge with regards to identifying class-object collaboration, responsibility distribution among the objects, etc. Very often designers are overwhelmed by the options available; the experienced designers make good designs by applying reusable and flexible design solutions, so called design patterns.

Design Pattern Classification

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organise them. Classification helps to learn the patterns in the catalogue faster and it can direct efforts to find other appropriate patterns as well.

Three categories [1] as follows are:

1. Creational
   - Concerns the process of object creation. Creational Patterns encapsulate knowledge about who creates the instances, what gets created, how it gets created, and when, making the design more flexible and promote reuse.
2. Structural
   - Deals with the composition of classes and objects
3. Behavioural
   - Characterises the way in which classes or objects interact and distribute responsibilities

How to describe Design Patterns

In this article we will adopt the following structure to describe a design pattern:

1. **Pattern Name** Conveys the essence of the pattern succinctly.
2. **Intent** Short description about what particular design problem the pattern addresses.
3. **Solution** UML class and sequence mode.
4. **Application** Using an example to illustrate how to apply the pattern, the UML design model and C# implementation are provided.

Abstract Factory[1]

Pattern Name: Abstract Factory

Intent: Provide an interface for clients to use to create families of related or dependent objects without using their constructors directly.

Solution: As shown in Figure 1, an abstract class AbstractFactory provides the abstract methods as the interfaces of the subclasses that, for example, Factory1 and Factory2 implement the abstract methods defined in the parent class. Several abstract classes capture the common characteristics of a range of products, such as AbstractProductA, AbstractProductB, which serve as the base classes for all different product classes. Client class only depends on the abstract classes, there are no direct or hard-code relationship between client class and the concrete product classes. Figure 2 uses UML sequence diagram to illustrate the sequence of massage calls among objects when concrete class

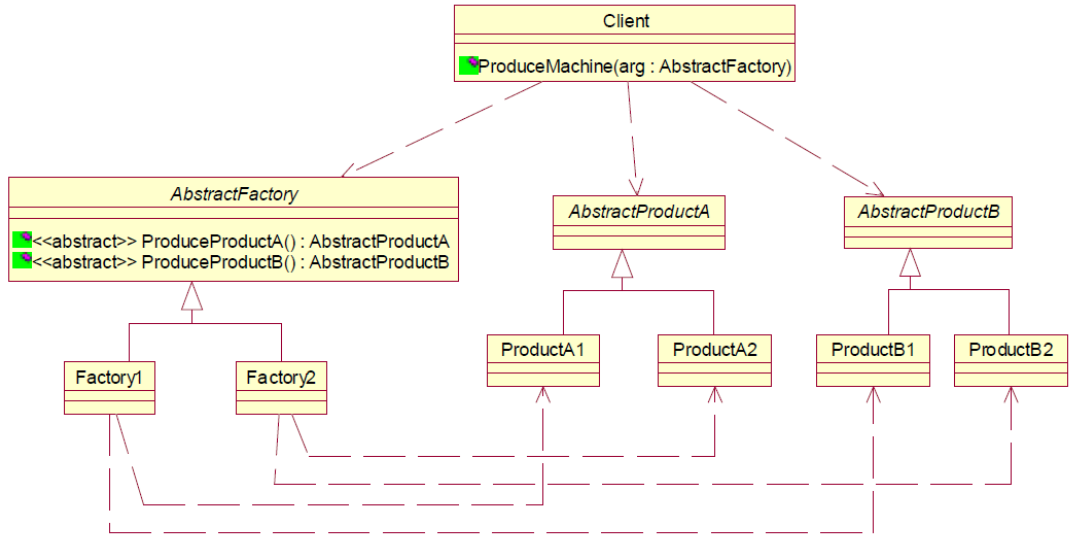

Figure 1. Abstract Factory Design Pattern Class Diagram

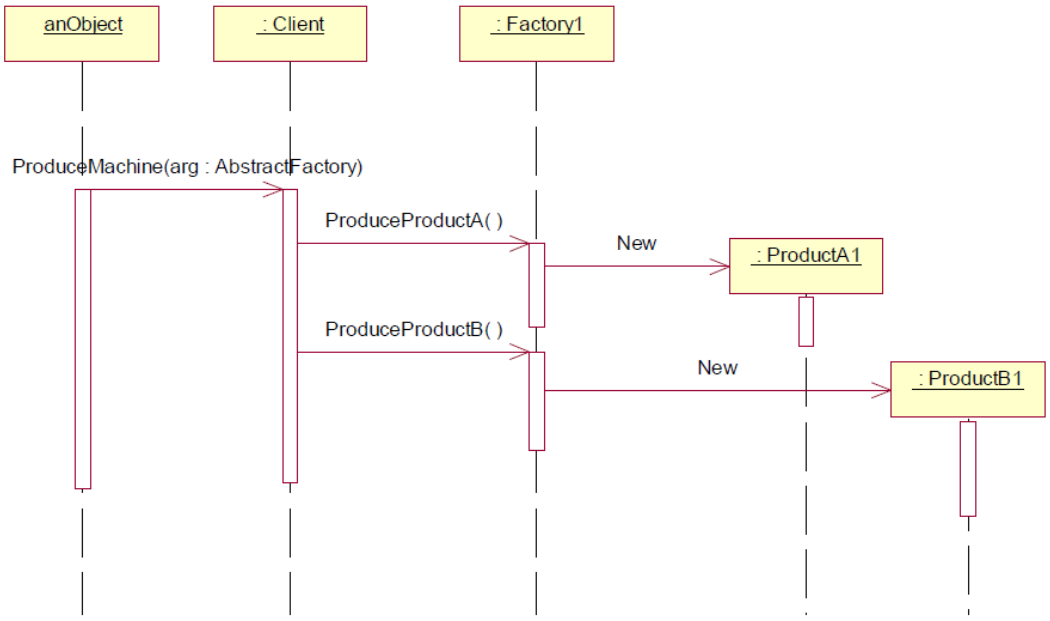

Figure 2. Abstract Factory Design Pattern Sequence Diagram

Application: The scenario is that a customer requests to purchase a computer, to isolate the customer from the concrete product classes, abstract factory pattern is applied.

Design model for the application using abstract factory pattern is shown in Figure 3 and Figure 4. PCManufacturer abstract factory class encapsulates the methods of the production of a computer. Two concrete subclasses Dell, Compaq implements the abstract methods in PCManufacturer. The Chassis and Monitor are abstract classes for which the concrete classes DellMonitor, CompaqMonitor, DellChassis, and CompaqChassis are derived. Computer class has composition relationship with Monitor and Chassis. DellComputer and CompaqComputer are subclasses of Computer class. Figure 4 shows Dell class as a concrete factory class to produce a computer for a customer.

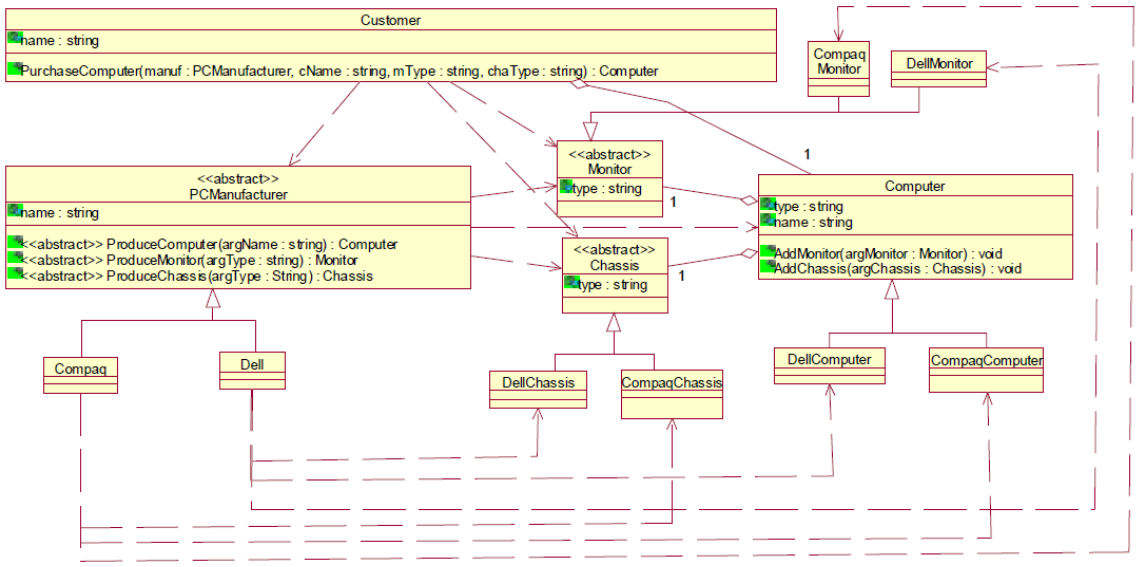

Figure 3. An Example of Using Abstract Factory Pattern

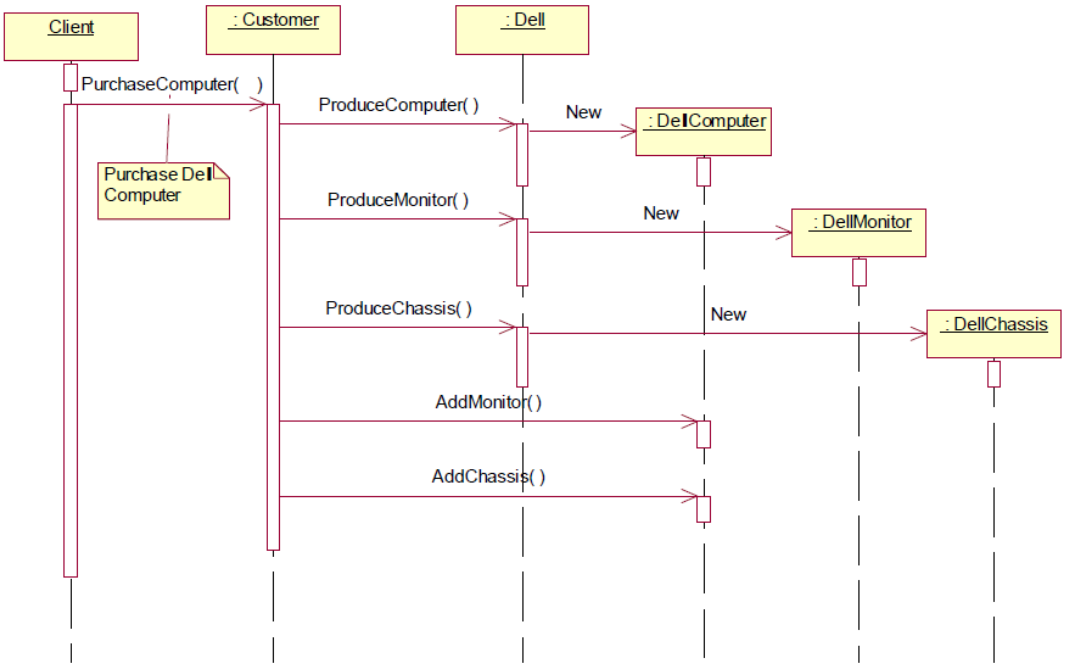

Figure 4. Make a Dell Computer Sequence Diagram

C# Implementation: The complete VS.NET project can be downloaded here. The following parts are the source codes for some of the base classes, concrete classes and Customer class.

PCManufacturer.cs

```
using System;
namespace AbstractFactory
{
abstract class PCManufacturer
{
```

```
private string name;
#region constructors and properties
#endregion
public abstract Computer ProduceComputer(string argName);
public abstract Monitor ProduceMonitor(string argType);
public abstract Chassis ProduceChassis(string argType);
}
}
```

Dell.cs

```
using System;
namespace AbstractFactory
{
class Dell : PCManufacturer
{
public Dell():base("Dell")
{
}
public override Computer ProduceComputer(string argName)
{
return new DellComputer(argName);
}
public override Monitor ProduceMonitor(string argType)
{
return new DellMonitor(argType);
}
public override Chassis ProduceChassis(string argType)
{
return new DellChassis(argType);
}
}
}
```

Computer.cs

```
using System;
namespace AbstractFactory
{
/// <summary>
/// Equivalen to one of AbstractProduct
/// </summary>
class Computer
{
private string type;
private string name;
private Monitor monitor;
private Chassis chassis;
#region constructors and properties for private data fields
#endregion
public void AddMonitor(Monitor argMonitor)
{
monitor = argMonitor;
}
```

```
public void AddChassis(Chassis argChassis)
{
chassis = argChassis;
}
}
}
```

DellComputer.cs

```
using System;
namespace AbstractFactory
{
class DellComputer : Computer
{
public DellComputer(string argName):base("Dell", argName)
{
Console.WriteLine("Make Dell Computer {0}", argName);
}
}
}
```

Customer.cs

```
using System;
namespace AbstractFactory
{
class Customer
{
private string name;
private Computer computer;
public Customer(string argName)
{
name = argName;
}
public void PurchaseComputer(PCManufacturer manuf, string cName, string
mType, string chaType)
{
Computer com = manuf.ProduceComputer(cName);
Monitor mon = manuf.ProduceMonitor(mType);
Chassis cha = manuf.ProduceChassis(chaType);
com.AddMonitor(mon);
com.AddChassis(cha);
computer = com;
}
}
}
```

Reference

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-**Oriented Software, Addison-Welsley, 1995**