1. **单例模式（Singleton）**

**单例设计模式的特点：**

1.单例设计模式保证一个类只有一个实例;

2.要提供一个访问该类对象实例的全局访问点。

单例模式最重要的就是要保证一个类只有一个实例并且这个类

易于被访问。

一个全局类使得一个对象可以被访问，但是这样做却不能防止

你实例化多个对象。

**单例设计模式的实现：**

1.为了避免其它程序过多的建立该类的对象，先禁止其它程序建

立该类对象实例（将构造器私有化）。

2.为了方便其它程序访问该类的对象，只好在本类中自定义一个

对象，由1可知该对象是static的，并对外提供访问方式。

例子：对于类Singleton的实现

（1）饿汉式

class Singleton {

    //为了方便其它程序访问该类的对象，只好在本类中自定义一个,

    //该对象是static的（static类数据成员独立于一切类对象处在），并对外提供访问方式。

   private static Singleton instance=new Singleton();

   private Singleton(){} //私有化构造器，禁止其它程序建立该类对象实例

   public static Singleton getInstance() {

      return instance;

        }

 }

总结：对象预先加载，线程是安全的，在类创建好的同时对象生成，调用获得对象实例的方法反应速度快，代码简练。

（2）懒汉式

class Singleton {

     private static Singleton instance=null;    //static保证全局访问该唯一对象

     private Singleton(){}   //私有化构造器  防止构造类对象

     public static Singleton getInstance() {

      if(instance==null) {

          instance=new Singleton();

      }

      return instance;

     }

}

总结：对象延迟加载，效率高，只有在使用的时候才实例化对象，若设计不当线程会不安全，代码相对于饿汉式复杂，第一次加载类对象的时候反应不快。

**二、建造者（Builder）模式**

**模式的意图**

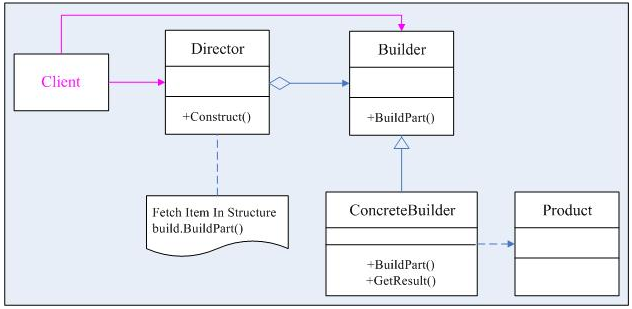
将一个复杂对象的构建过程与其表示相分离，使得同样的构建过程可以创建不同的表示

在实际生活中，有很多这样的例子：

* + 譬如一个房屋的构建。创建过程基本不变,但是门窗等组件却是易变的。

**以下情况可以使用建造者模式**

* + 需要生成的产品对象有复杂的内部结构
  + 创建复杂对象的算法稳定，或建造者模式可以强迫生成一定的顺序
  + 当构造过程允许被构造的对象有不同的表示时

**建造者模式结构图：**  


**建造者模式的结构和参与者**

1）Builder

为创建一个Product对象的各个部件指定抽象接口。

2）ConcreteBuilder

实现Builder的接口以构造和装配该产品的各个部件。

定义并明确它所创建的表示。

提供一个检索产品的接口

3）Director

构造一个使用Builder接口的对象。定义了构造产品的每个步骤，指明如何借助Builder生产产品的过程。

4）Product

表示被构造的复杂对象，包含定义组成部件的类。ConcreteBuilder创建该产品的内部表示并定义它的装配过程。

**应用举例：模拟组装一台电脑 假设电脑由主板、CPU和内存三个部件构成**

**//定义产品角色（电脑）**

//定义产品角色

public class Computer

{

   //电脑中零件的集合

   List<string> parts = new List<string>();

   //添加零件

   public void add(string part)

   {parts.Add(part);}

   //展示组装完成的电脑

   public void Show()

   { Console.WriteLine(”\n 这是一台电脑，它包括：”) ；

   foreach(string part in parts)

   {Console.Write(part);}

  }

}

定义Builder，为每一个具体的对象（存在内部变化的）指定统一的抽象接口。

//定义抽象的Builder

public abstract class Builder

{

   //添加主板

   public abstract void addMainBoard();

   //添加CPU

   public abstract void addCPU();

   //添加内存

   public abstract void addMemory();

   //获取组装好的产品

   public abstract Computer GetComputer();

}

定义具体的建造者：（由于电脑的内部变化是不同的，即虽然组装的流程一样，但是对于电脑的每个部件例如主板、CPU、内存等会根据需求发生变化，就比如性能的不同，所以，这里需要实现多个不同的具体建造者）

//具体建造者，建造一台高性能计算机

public class HighPerformanceComputer : Builder

{

   private Computer HPcomputer = new Computer();

   public override void AddMainBoard()//具体装主板的方法

   {HPComputer.add(“高档主板”);}

   public override void AddCPU()//具体装CPU的方法

   {HPComputer.add(“双核 4.0GHz”);}

   public override void AddMemory() //具体装内存的方法

   {HPComputer.add(“4GB内存”);}

   public override void GetComputer() //具体获得产品的方法

   {return HPComputer;}

}

//具体建造者，建造一台普通性能计算机

public class OrdinaryComputer : Builder

{

   private Computer Ocomputer = new Computer();

   public override void AddMainBoard()//具体装主板的方法

   {OComputer.add(“中档主板”);}

   public override void AddCPU() //具体装CPU的方法

   {OComputer.add(“P4 2.0GHz”);}

   public override void AddMemory() //具体装内存的方法

   {OComputer.add(“512MB内存”);}

   public override void GetComputer() //具体获得产品的方法

   {return OComputer;}

}

//定义指挥者类(表示了组装过程，组装过程是基本不变的)

public class Director

{

    public abstract void Construct(Builder builder)

    {

        builder.AddMainBoard();

        builder.AddCPU();

        builder.AddMemory();

   }

}

//客户端代码

static void Main(string[] args)

{

   //创建一个指挥者，用来指挥组装计算机

  Director director = new Director();

  //高性能计算机的Builder

   Builder b1 = new HighPerformanceComputer();

  //创建一台高性能计算机

   director.Construct(b1);

  //获取一台高性能计算机

   Computer HighPerformanceComputer = b1.GetComputer();

  //展示一台高性能计算机

   HighPerformanceComputer.Show();

}

**模式优点**

**（1）**客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象

（2）每一个具体建造者都相对独立，与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，扩展方便，符合开闭原则

1. 可以更加精细地控制产品的创建过程

**模式缺点**

（1）建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，不适合使用建造者模式，因此其使用范围受到一定的限制。

（2）如果产品的内部变化复杂，可能会需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加了系统的理解难度和运行成本。