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

<http://www.codeproject.com/Articles/33530/Design-Pattern-Examples-in-C>

<https://csharpdesignpatterns.codeplex.com/>

**Design Patterns**

Шаблоните за дизайн представляват формален механизъм за документиране на решения на често случващи се софтуерни проблеми. За пръв път шаблони за дизайн въвежда Кристофър Александър през 1977 г. Неговите са свързани с архитектурата, а впоследствие са адаптирани и към софтуерния дизайн. През последните години интересът към софтуерните шаблони за дизайн расте изключително много.

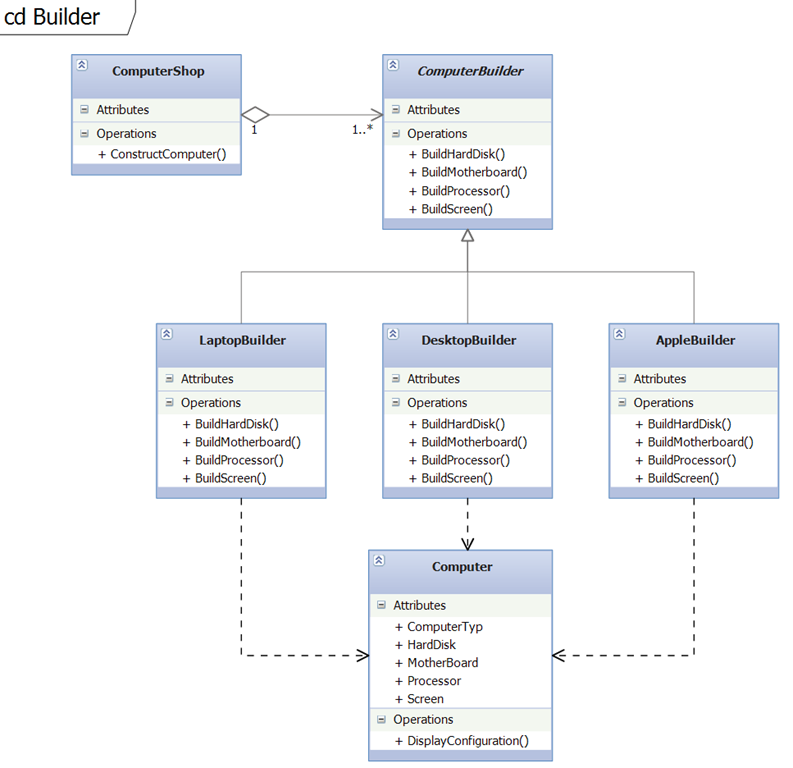
Шаблоните за дизайн най-често се използват като решения на някои софтуерни проблеми. Те, разбира се, не са единственото решение, и в никакъв случай не трябва да се изключват другите възможности. Софтуерната архитектура и рефакторирането също намират място.

Singleton

**Singleton е шаблон за създаване на обекти които ограничава създаването на клас и в рамките на този клас и позволява на потребителя да ползва само статичните методи които могат да се викат чрез единствената инстанция на класа.  
Прилага се когато даденият клас не трябва да може да буде имплементиран а само да бъде използван. Примери за такива класове са в C# са (Math, Console и други).  
Класът се имплементира като се използва ключовата дума sealed (която забранява класа да се наследява) и конструкторът който е задължителен се прави private за да може да бъде достъпен само в рамките на този клас.**

1. **Builder Pattern**

Този шаблон е шаблон за създаване (Creational Pattern). Предоставя възможност за разкачена репрезентация и създаване на обектите. Използва се рядко; най-използваем е, когато обектите се състоят от множество части, които трябва да се конструират.



Директорът (ComputerShop) имплементира метод, който е отговорен за последователността от стъпки при процеса на създаване на обекта. Той използва абстрактен строителен (builder) клас като входен параметър и разчита на създаването изцяло на него.

Абстрактният строителен клас дефинира интерфейса, който всички наследници ще използват реално за създаването на обекта.

|  |
| --- |
| public class ComputerShop     {         public void ConstructComputer(ComputerBuilder computerBuilder)         {             computerBuilder.BuildMotherboard();             computerBuilder.BuildProcessor();             computerBuilder.BuildHardDisk();             computerBuilder.BuildScreen();         }     }       public abstract class ComputerBuilder     {         public Computer Computer { get; set; }           public abstract void BuildMotherboard();         public abstract void BuildProcessor();         public abstract void BuildHardDisk();         public abstract void BuildScreen();     } |

Конкретните имплементации съдържат частите, необходими за създаването на обектите.

|  |
| --- |
| public class LaptopBuilder : ComputerBuilder     {         public LaptopBuilder()         {             Computer = new Computer(ComputerTyp.Laptop);         }           public override void BuildMotherboard()         {             Computer.MotherBoard = "DELL MotherBoard";         }           public override void BuildProcessor()         {             Computer.Processor = "Intel Core 2 Duo";         }           public override void BuildHardDisk()         {             Computer.HardDisk = "250GB";         }           public override void BuildScreen()         {             Computer.Screen = "15.4-inch (1280 x 800)";         }     }       public class DesktopBuilder : ComputerBuilder     {         public DesktopBuilder()         {             Computer = new Computer(ComputerTyp.Desktop);         }           public override void BuildMotherboard()         {             Computer.MotherBoard = "Asus P6X58D Premium";         }           public override void BuildProcessor()         {             Computer.Processor = "Intel Xeon 7500";         }           public override void BuildHardDisk()         {             Computer.HardDisk = "2TB";         }           public override void BuildScreen()         {             Computer.Screen = "21 inch (1980 x 1200)";         }     }       public class AppleBuilder : ComputerBuilder     {         public AppleBuilder()         {             Computer = new Computer(ComputerTyp.Apple);         }           public override void BuildMotherboard()         {             Computer.MotherBoard = "iMac G5 PowerPC";         }           public override void BuildProcessor()         {             Computer.Processor = "Intel Core 2 Duo";         }           public override void BuildHardDisk()         {             Computer.HardDisk = "320GB";         }           public override void BuildScreen()         {             Computer.Screen = "24 inch (1980 x 1200)";         }     } |

Крайният обект съдържа всички части, които са създадени от конкретните строителни класове. Те могат много да се различават едни от други в зависимост от конкретните имплементации.

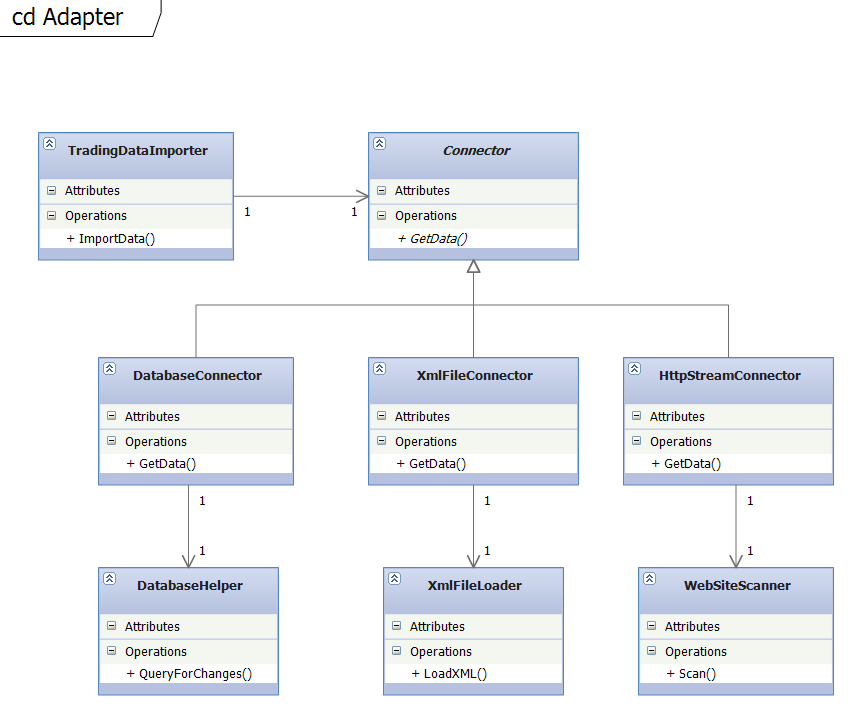
|  |
| --- |
| public class Computer     {         private ComputerTyp \_computerTyp;           public string MotherBoard { get; set; }         public string Processor { get; set; }         public string HardDisk { get; set; }         public string Screen { get; set; }           public Computer(ComputerTyp computerTyp)         {             \_computerTyp = computerTyp;         }           public void DisplayConfiguration()         {             string message;               message = string.Format("Computer: {0}", \_computerTyp);             Console.WriteLine(message);               message = string.Format("Motherboard: {0}", MotherBoard);             Console.WriteLine(message);               message = string.Format("Processor: {0}", Processor);             Console.WriteLine(message);               message = string.Format("Harddisk: {0}", HardDisk);             Console.WriteLine(message);               message = string.Format("Screen: {0}", Screen);             Console.WriteLine(message);               Console.WriteLine();         }     }       public enum ComputerTyp     {         Laptop,         Desktop,         Apple     } |

За тестване:

|  |
| --- |
| private static void Builder()     {         ComputerShop computerShop = new ComputerShop();         ComputerBuilder computerBuilder;           computerBuilder = new LaptopBuilder();         computerShop.ConstructComputer(computerBuilder);         computerBuilder.Computer.DisplayConfiguration();           computerBuilder = new DesktopBuilder();         computerShop.ConstructComputer(computerBuilder);         computerBuilder.Computer.DisplayConfiguration();           computerBuilder = new AppleBuilder();         computerShop.ConstructComputer(computerBuilder);         computerBuilder.Computer.DisplayConfiguration();         Console.ReadKey();     } |

1. **Adapter Pattern**

Този шаблон е структурен (Structural Pattern). Той прави сравнение между интерфейси на класове с други интерфейси. Използва се често и е лесен за имплементация. Използва се, когато има нужда класовете да работят заедно, когато те имат несъвместими имплементирани интерфейси.



Класът TradingDataImporter работи като клиент, който използва други класове със съществуваща връзка между интерфейсите.

|  |
| --- |
| public class TradingDataImporter     {         public void ImportData(Connector connector)         {             connector.GetData();         }     } |

Абстрактният клас Adapter дефинира интерфейса, който класът клиент може да използва.

|  |
| --- |
| public abstract class Connector     {         public abstract void GetData();     }       public class DatabaseConnector : Connector     {         public override void GetData()         {             var databaseHelper = new DatabaseHelper();             databaseHelper.QueryForChanges();         }     }       public class XmlFileConnector : Connector     {         public override void GetData()         {             var xmlfileLoader = new XmlFileLoader();             xmlfileLoader.LoadXML();         }     }       public class HttpStreamConnector : Connector     {         public override void GetData()         {             var websiteScanner = new WebSiteScanner();             websiteScanner.Scan();         }     }      public class DatabaseHelper     {         public void QueryForChanges()         {             Console.WriteLine("Database queried.");         }     }       public class WebSiteScanner     {         public void Scan()         {             Console.WriteLine("Web sites scanned.");         }     }       public class XmlFileLoader     {         public void LoadXML()         {             Console.WriteLine("Xml files loaded.");         }     } |

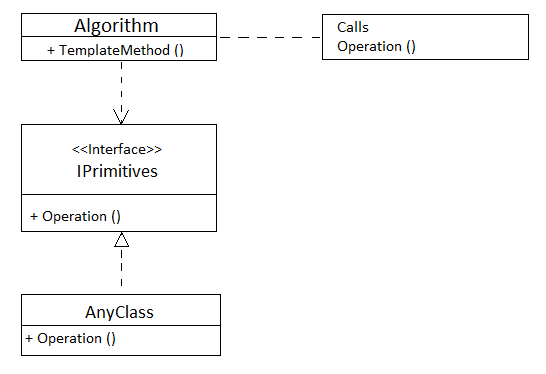
За тестване:

|  |
| --- |
| public static void Adapter()     {         var tradingdataImporter = new TradingDataImporter();           Connector databaseConnector = new DatabaseConnector();         tradingdataImporter.ImportData(databaseConnector);           Connector xmlfileConnector = new XmlFileConnector();         tradingdataImporter.ImportData(xmlfileConnector);           Connector httpstreamConnector = new HttpStreamConnector();         tradingdataImporter.ImportData(httpstreamConnector);                       Console.ReadKey();     } |

1. **Template Method Pattern**

Този шаблон е поведенчески (Behavioral Pattern). Позволява алгоритмите да съотнасят конкретните стъпки към съответните подкласове. Самата структура на алгоритъма не се променя, но малки и определени части се обработват на друго място.

UML-диаграмата показва един алгоритмичен клас, който имплементира интерфейса IPrimitives, за да се свързва с методи дефинирани, в който и да е друг клас.



Algorithm – клас, който включва TemplateMethod

TemplateMethod – метод, който разпределя някои части от неговия алгоритъм към други класове

IPrimitives – интерфейс, който дефинира операциите, които TemplateMethod ще разпределя

|  |
| --- |
| interface IPrimitives  {  string Operation1( );  string Operation2( );  }  class Algorithm  {  public void TemplateMethod(IPrimitives a)  {  string s =  a.Operation1( ) +  a.Operation2( );  Console.WriteLine(s);  }  }  class ClassA : Iprimitives  {  public string Operation1( )  {  return "ClassA:Op1 ";  }  public string Operation2( )  {  return "ClassA:Op2 ";  }  }  class ClassB : Iprimitives  {  public string Operation1( )  {  return "ClassB:Op1 ";  }  public string Operation2( )  {  return "ClassB.Op2 ";  }  }  class TemplateMethodPattern  {  static void Main( )  {  Algorithm m = new Algorithm( );  m.TemplateMethod(new ClassA( ));  m.TemplateMethod(new ClassB( ));  }  } |

Този шаблон е много използван в комбинация със Strategy Pattern. Всяка програма, която иска да разпределя задачите на други класове, би трябвало да използва този шаблон. В C# има широко използване в предефинирани интерфейси като IComparable.