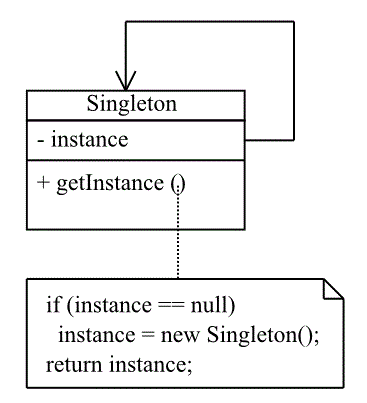
# Introduction to Design Patterns

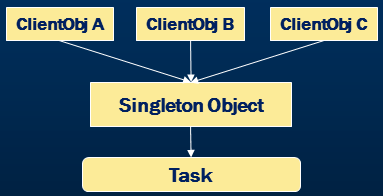


**In this tutorial we will discuss**  
1. What are Design Patterns  
2. Evolution of Design Patterns   
3. The three categories of design patterns    
  
In many interviews you might have encountered lot of questions related to interfaces, abstract classes, delegates and other features related to object oriented programming along with the design pattern related questions.    
  
  
  
Design patterns solutions are evolved from the  features of object oriented programming.    
  
Once you understand the design patterns, it makes you very comfortable in attending interviews as well as applying these features with confidence in your projects or applications.  
    
Implementing design patterns in the applications are proven and tested. Writing the code aligning with design patterns will make your applications Reliable,Scalable and Maintainable.    
  
  
  
**What are Design Patterns**   
Design patterns are reusable solutions to the problems that we encounter in the day to day programming. They are generally targeted at solving the problems of object generation and integration. In other words, Design patterns acts as templates which can be applied to the real-world programming problems.  
  
**History and evolution of design Patterns**  
The four authors of the book famously know as Gang of four are the ones who brought the concepts of design patterns in their book “Elements of reusable Object-Oriented software” .   
   
Gang of Four has divided the book into two parts with first part explaining about the pros and cons of object oriented programming and the second part describes the evolution of 23 classic software design patterns.  
   
For more details, please refer to the following wikipedia article  
<https://en.wikipedia.org/wiki/Design_Patterns>  
  
**Types of Design Patterns**  
Gang of Four have categorised the design patterns in to 3 types based on different problems encountered in the real world applications. They are Creational, Structural and Behavioural.  
   
**Creational design patterns :**These patterns deal with object creation and initialization. Creational pattern gives the program more flexibility in deciding which objects need to be created for a given case.  
   
**Examples of Creational design patterns category :**Singleton , Factory and Abstract Factory etc.  
  
**Structural design patterns :**This pattern deals with class and object composition. In simple words, This pattern focuses on decoupling interface, implementation of classes and its objects.   
  
**Examples of Structural design patterns category :**Adapter,  Facade and Bridge etc.  
  
**Behavioural design patterns :**These patterns deal with communication between Classes and objects.   
  
**Examples of Behavioural design patterns :** Chain of Responsibility, Command and Interpreter etc.  
   
For understanding design patterns better it is very important to have basic knowledge about the following object oriented concepts

* Abstraction
* Inheritance
* Polymorphism
* Encapsulation
* Interfaces
* Classes
* Abstract classes

In Part 2 we will focus on the Singleton design pattern in detail with a simple example.

### Singleton Design Pattern

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)   
  
**In this tutorial we will discuss**   
1. What is Singleton Design Pattern  
2. Singleton as Creational Pattern  
3. Implementation Guidelines  
4. How do we implement a Singleton class   
  
  
This is continuation to [Part 1](https://www.youtube.com/watch?v=rI4kdGLaUiQ) of [Design Patterns Tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd). So please watch [Part 1](https://www.youtube.com/watch?v=rI4kdGLaUiQ) before proceeding.   
  
**Singleton Pattern** belongs to **Creational type pattern**. As discussed in our previous video, Gang of four have defined five design patterns that belongs to creational design type category. Singleton is one among them and the rest are Factory, Abstract Factory, Builder and Prototype patterns. As the name implies, creational design type deals with object creation mechanisms. Basically, to simplify this, creational pattern explain us the creation of objects in a manner suitable to a given situation.   
  
  
Singleton design pattern is used when we need to ensure that only one object of a particular class is Instantiated. That single instance created is responsible to coordinate actions across the application.   
  
   
  
If you look at the illustrated diagram above you will see different objects trying to invoke an object instantiated as singleton. This single instance of the object is responsible to invoke underneath methods or events.  
  
**Advantages and Guidelines for Singleton implementation.**  
  
Concurrent access to the resource is well managed by singleton design pattern.  
  
As part of the Implementation guidelines we need to ensure that only one instance of the class exists by declaring all constructors of the class to be private.  Also, to control the singleton access we need to provide a static property that returns a single instance of the object.  
  
**Singleton Class Implementation Example**

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            /\*

             \* Assuming Singleton is created from employee class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

            /\*

             \* Assuming Singleton is created from student class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

            Console.ReadLine();

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed ensures the class being inherited and

     \*  object instantiation is restricted in the derived class

     \*/

    public sealed class Singleton

    {

        private static int counter = 0;

        /\*

         \* Private property initilized with null

         \* ensures that only one instance of the object is created

         \* based on the null condition

         \*/

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton();

                return instance;

            }

        }

        /\*

         \* Private constructor ensures that object is not

         \* instantiated other than with in the class itself

         \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

Why is singleton class sealed

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)  
[Part 2 - Singleton Design Pattern](https://www.youtube.com/watch?v=YGGg9ecy0K4) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern_13.html)   
  
**In this tutorial we will discuss**

* A quick Recap of Singleton version which we have discussed in the previous session
* We will focus on why we used Sealed keyword
* Why we need to seal the singleton class

**In our previous video we discussed**

* Why we need to create a singleton class and how we can apply singleton design pattern to a class
* We have changed the class to restrict external instantiation by changing the public constructor to private and then we provided a public property to access this class
* We have proved that adding Private constructor will prevent us instantiating a new class
* We have further sealed down this class to avoid any inheritance

You might be wondering why we need to seal the class when a private constructor is present.    
  
Let’s first remove the sealed keyword and check that. Let’s create another class called DerivedSingleton and Inherit the singleton class. Let's compile the code and look at that it has thrown an error saying Singleton is inaccessible due to its protection level. This error is because of private constructor.  
  
Now you might be thinking that when a private constructor is restricting the inheritance then why we need to apply sealed keyword to the class.   
  
Let’s just move this new class inside the Singleton class. By moving this class inside the Singleton class it has now become nested or child class of the main singleton class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed restricts the inheritance

     \*/

    public class Singleton

    {

        private static int counter = 0;

        private static object obj = new object();

        /\*

        \* Private constructor ensures that object is not

        \* instantiated other than with in the class itself

        \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton();

                return instance;

            }

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

        /\*

         \* By removing sealed keyword we can inherit the singleton and instantiate multiple objects

         \* This violates singleton design principles.

         \*/

        public class DerivedSingleton : Singleton

        {

        }

    }

}   
  
**What is a nested class?**   
A class with in another class is called a nested class.   
  
Now that we have moved the derived class to nested class lets compile the program and check. Look at that we are able to compile this successfully.   
  
Now, let’s switch to main program and access the nested class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            /\*

            \* Assuming Singleton is created from student class

            \* we refer to the GetInstance property from the Singleton class

            \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

            /\*

            \* Assuming Singleton is created from employee class

            \* we refer to the GetInstance property from the Singleton class

            \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

            Console.WriteLine("-------------------------------------");

            /\*

             \* Instantiating singleton from a derived class. This violates singleton pattern principles.

             \*/

            Singleton.DerivedSingleton derivedObj = new Singleton.DerivedSingleton();

            derivedObj.PrintDetails("From Derived");

            Console.ReadLine();

        }

    }

}  
  
Lets run the program. Look at that the counter value has incremented to 2 proving that we are able to create multiple instances of the singleton using the nested derived class.  
  
This violates the principle of singleton.  Let’s go back to the Singleton and make the class as sealed. Let’s compile the program  
  
Look at that we have got an error when we compile the program saying we cannot derive a sealed class. With this we have proved that private constructor helps in preventing any external instantiations of objects and sealed will prevent the class inheritances.  
  
In the next session we will discuss how to handle thread safety in singleton as the current version can create multiple instances in multi-threaded environments.

Thread Safety in Singleton

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)  
[Part 2 - Singleton Design Pattern](https://www.youtube.com/watch?v=YGGg9ecy0K4) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern_13.html)   
[Part 3 - Why is singleton class sealed](https://www.youtube.com/watch?v=LypTOnfkfvA) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed_19.html)  
  
**In this tutorial we will discuss**

* Lazy Initialization in Singleton
* How to use Multithreads in Singleton
* How to implement a Thread Safe singleton class

**Lazy Initialization in Singleton :** GetInstance Property is responsible for the Singleton Instance creation. Singleton object is not instantiated until and unless **GetInstance** is invoked. Hence, there is a delay in instance creation till the GetInstance is accessed. This Delay in Instance creation is called Lazy Initialization.   
  
  
  
  
**How to use Multithreads in Singleton :**The lazy initialization works perfectly well when we invoke the GetInstance in a Single threaded approach. However, there is a chance that we may end up creating multiple instances when multiple threads invoke the GetInstance at the same time.  
  
This Thread racing situation causes thread safety issues in Singleton Initialization and further the current code ends up in creating multiple instances of Singleton objects in memory.  
  
To achieve and replicate multiple threads accessing GetInstance, We have modified the main program by using Parallel.Invoke method of .NET Framework 4.0.  Please refer to Main program code below for more details.  
  
**How to implement a Thread Safe singleton class :** Locks are the best way to control thread race condition and they help us to overcome the present situation. Please refer to the Singleton.cs code for lock checks and double check locking.  
  
For more details on double check locking please refer to the below article  
<https://en.wikipedia.org/wiki/Double-checked_locking>

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Parallel.Invoke(

                () => PrintStudentdetails(),

                () => PrintEmployeeDetails()

                );

            Console.ReadLine();

        }

        private static void PrintEmployeeDetails()

        {

            /\*

             \* Assuming Singleton is created from employee class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

        }

        private static void PrintStudentdetails()

        {

            /\*

                         \* Assuming Singleton is created from student class

                         \* we refer to the GetInstance property from the Singleton class

                         \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed restricts the inheritance

     \*/

    public sealed class Singleton

    {

        private static int counter = 0;

        private static readonly object obj = new object();

        /\*

        \* Private constructor ensures that object is not

        \* instantiated other than with in the class itself

        \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                {

                    lock (obj)

                    {

                        if (instance == null)

                            instance = new Singleton();

                    }

                }

                return instance;

            }

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

### Lazy vs Eager loading in Singleton

**Suggested Videos**  
[Part 2 - Singleton Design Pattern](https://www.youtube.com/watch?v=YGGg9ecy0K4) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/singleton-design-pattern_13.html)  
[Part 3 - Why is singleton class sealed](https://www.youtube.com/watch?v=LypTOnfkfvA) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed_19.html)   
[Part 4 - Thread Safety in Singleton](https://www.youtube.com/watch?v=QWrcOmLWi_Q) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton_5.html)  
  
In this tutorial we will discuss the **difference between Lazy Initialization and Eager Initialization**  
  
**Lazy Initialization :**The lazy initialization of an object improves the performance and avoids unnecessary computation till the point the object is accessed. Further, it reduces the memory footprint during the startup of the program. Reducing the memory print will help faster loading of the application.   
  
  
  
**Non-Lazy or Eager Loading :** Eager loading is nothing but to initialize the required object before it’s being accessed.  Which means, we instantiate the object and keep it ready and use it when we need it. This type of initialization is used in lower memory footprints. Also, in eager loading, the common language runtime takes care of the variable initialization and its thread safety. Hence, we don’t need to write any explicit coding for thread safety.   
  
  
  
**Singleton with Lazy keyword (.NET 4.0) :** Lazy keyword provides support for lazy initialization. In order to make a property as lazy, we need to pass the type of object to the lazy keyword which is being lazily initialized.   
  
By default, Lazy<T> objects are thread-safe.  In multi-threaded scenarios, the first thread which tries to access the Value property of the lazy object will take care of thread safety when multiple threads are trying to access the Get Instance at the same time.   
  
Therefore, it does not matter which thread initializes the object or if there are any thread race conditions that are trying to access this property.

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Parallel.Invoke(

                () => PrintStudentDetails(),

                () => PrintEmployeeDetails()

            );

            Console.ReadLine();

        }

        private static void PrintEmployeeDetails()

        {

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

        }

        private static void PrintStudentDetails()

        {

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

    public sealed class Singleton

    {

        private static int counter = 0;

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static readonly Lazy<Singleton> instance =

new Lazy<Singleton>(()=>new Singleton());

        public static Singleton GetInstance

        {

            get

            {

                return instance.Value;

            }

        }

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

Static Class vs Singleton

**Suggested Videos**  
[Part 3 - Why is singleton class sealed](https://www.youtube.com/watch?v=LypTOnfkfvA) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/why-is-singleton-class-sealed_19.html)  
[Part 4 - Thread Safety in Singleton](https://www.youtube.com/watch?v=QWrcOmLWi_Q) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton_5.html)   
[Part 5 - Lazy vs Eager loading in Singleton](https://www.youtube.com/watch?v=xk-AKHUCdGc) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton_12.html)  
  
**Differences between Singleton and static classes**

1. Static is a keyword and Singleton is a design pattern
2. Static classes can contain only static members
3. Singleton is an object creational pattern with one instance of the class
4. Singleton can implement interfaces, inherit from other classes and it aligns with the OOPS concepts
5. Singleton object can be passed as a reference
6. Singleton supports object disposal
7. Singleton object is stored on heap
8. Singleton objects can be cloned

**Static class example - Temperature Converter**  
We are pretty sure that the formulas for foreign heat to Celsius conversion and vice versa will not change at all and hence we can use static classes with static methods that does the conversion for us. Please refer to the below code for more details.  
  
  
  
**Real world usage of Singleton :** Listed are few real world scenarios for singleton usage

1. Exception/Information logging
2. Connection pool management
3. File management
4. Device management such as printer spooling
5. Application Configuration management
6. Cache management
7. And Session based shopping cart are some of the real world usage of singleton design pattern

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace StaticDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            double celcius = 37; double fahrenheit = 98.6;

            Console.WriteLine("Value of {0} celcius to fahrenheit is {1}",

                celcius, Converter.ToFahrenheit(celcius));

            Console.WriteLine("Value of {0} fahrenheit to celcius is {1}",

                fahrenheit, Converter.ToCelcius(fahrenheit));

            Console.ReadLine();

        }

    }

}

**Converter.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace StaticDemo

{

    public static class Converter

    {

        public static double ToFahrenheit(double celcius)

        {

            return (celcius \* 9 / 5) + 32;

        }

        public static double ToCelcius(double fahrenheit)

        {

            return (fahrenheit - 32) \* 5 / 9;

        }

    }

}

Exception Logging using Singleton Design Pattern

**Suggested Videos**  
[Part 4 - Thread Safety in Singleton](https://www.youtube.com/watch?v=QWrcOmLWi_Q) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/thread-safety-in-singleton_5.html)  
[Part 5 - Lazy vs Eager loading in Singleton](https://www.youtube.com/watch?v=xk-AKHUCdGc) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/lazy-vs-eager-loading-in-singleton_12.html)   
[Part 6 - Static Class vs Singleton](https://www.youtube.com/watch?v=zlg4jCY2g4o) - [Text](http://csharp-video-tutorials.blogspot.com/2017/06/static-class-vs-singleton.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/static-class-vs-singleton_17.html)  
  
  
  
In this tutorial we will discuss how to create a simple employee web application using ASP.NET MVC and we will create a custom logger library using Singleton design pattern which logs exceptions to an external file   
  
  
  
Logger Library

**ILog.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Logger

{

    public interface ILog

    {

        void LogException(string message);

    }

}

**Log.cs**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Logger

{

    public sealed class Log : ILog

    {

        private Log()

        {

        }

        private static readonly Lazy<Log> instance = new Lazy<Log>(() => new Log());

        public static Log GetInstance

        {

            get

            {

                return instance.Value;

            }

        }

        public void LogException(string message)

        {

            string fileName = string.Format("{0}\_{1}.log", "Exception", DateTime.Now.ToShortDateString());

            string logFilePath = string.Format(@"{0}\{1}", AppDomain.CurrentDomain.BaseDirectory, fileName);

            StringBuilder sb = new StringBuilder();

            sb.AppendLine("----------------------------------------");

            sb.AppendLine(DateTime.Now.ToString());

            sb.AppendLine(message);

            using (StreamWriter writer = new StreamWriter(logFilePath, true))

            {

                writer.Write(sb.ToString());

                writer.Flush();

            }

        }

    }

}

Create and MVC Application and Create EmployeePortal DB with Employee Table

**Employee Table**

CREATE TABLE [dbo].[Employee] (

    [Id]             INT          IDENTITY (1, 1) NOT NULL,

    [Name]           VARCHAR (50) NOT NULL,

    [JobDescription] VARCHAR (50) NOT NULL,

    [Number]         VARCHAR (50) NOT NULL,

    [Department]     VARCHAR (50) NOT NULL,

    PRIMARY KEY CLUSTERED ([Id] ASC)

);

Generate Model using ADO.Net entity model generator using the above Table. Post generation, Add an Employee controller and use generated model which further creates views for Employee which facilitates CRUD operations on the employee.

**Sample EmployeeController.cs**

using Logger;

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.Entity;

using System.Linq;

using System.Net;

using System.Web;

using System.Web.Mvc;

using Web.Models;

namespace Web.Controllers

{

    public class EmployeesController : Controller

    {

        private ILog \_ILog;

        private EmployeePortalEntities db = new EmployeePortalEntities();

        public EmployeesController()

        {

            \_ILog = Log.GetInstance;

        }

        protected override void OnException(ExceptionContext filterContext)

        {

            \_ILog.LogException(filterContext.Exception.ToString());

            filterContext.ExceptionHandled = true;

            this.View("Error").ExecuteResult(this.ControllerContext);

        }

        // GET: Employees

        public ActionResult Index()

        {

            return View(db.Employees.ToList());

        }

        // GET: Employees/Details/5

        public ActionResult Details(int? id)

        {

            if (id == null)

            {

                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

            }

            Employee employee = db.Employees.Find(id);

            if (employee == null)

            {

                return HttpNotFound();

            }

            return View(employee);

        }

        // GET: Employees/Create

        public ActionResult Create()

        {

            return View();

        }

        // POST: Employees/Create

        // To protect from overposting attacks, please enable the specific properties you want to bind to, for

        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.

        [HttpPost]

        [ValidateAntiForgeryToken]

        public ActionResult Create([Bind(Include = "Id,Name,JobDescription,Number,Department")] Employee employee)

        {

            if (ModelState.IsValid)

            {

                db.Employees.Add(employee);

                db.SaveChanges();

                return RedirectToAction("Index");

            }

            return View(employee);

        }

        // GET: Employees/Edit/5

        public ActionResult Edit(int? id)

        {

            if (id == null)

            {

                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

            }

            Employee employee = db.Employees.Find(id);

            if (employee == null)

            {

                return HttpNotFound();

            }

            return View(employee);

        }

        // POST: Employees/Edit/5

        // To protect from overposting attacks, please enable the specific properties you want to bind to, for

        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.

        [HttpPost]

        [ValidateAntiForgeryToken]

        public ActionResult Edit([Bind(Include = "Id,Name,JobDescription,Number,Department")] Employee employee)

        {

            if (ModelState.IsValid)

            {

                db.Entry(employee).State = EntityState.Modified;

                db.SaveChanges();

                return RedirectToAction("Index");

            }

            return View(employee);

        }

        // GET: Employees/Delete/5

        public ActionResult Delete(int? id)

        {

            if (id == null)

            {

                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

            }

            Employee employee = db.Employees.Find(id);

            if (employee == null)

            {

                return HttpNotFound();

            }

            return View(employee);

        }

        // POST: Employees/Delete/5

        [HttpPost, ActionName("Delete")]

        [ValidateAntiForgeryToken]

        public ActionResult DeleteConfirmed(int id)

        {

            Employee employee = db.Employees.Find(id);

            db.Employees.Remove(employee);

            db.SaveChanges();

            return RedirectToAction("Index");

        }

        protected override void Dispose(bool disposing)

        {

            if (disposing)

            {

                db.Dispose();

            }

            base.Dispose(disposing);

        }

    }

}

* Run the application and all the exceptions will be logged under the file created by the logger class library.
* This Proves that singleton design pattern comes handy in the situations where we need to have a single instance of the object.
* Now, to consider another example, we can design Cache Management to use and leverage on Singleton design pattern as we can handle reads and writes to external caches using the Single Cache instance object.

# Dependency injection and IOC

**Inversion of Control (инверсия управления)** — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.  
**Dependency Injection (внедрение зависимостей)** — это одна из реализаций этого принципа (помимо этого есть еще [Factory Method](http://en.wikipedia.org/wiki/Factory_pattern), [Service Locator](http://en.wikipedia.org/wiki/Service_locator_pattern)).  
**IoC-контейнер** — это какая-то библиотека, фреймворк, программа если хотите, которая позволит вам упростить и автоматизировать написание кода с использованием данного подхода на столько, на сколько это возможно. Их довольно много, пользуйтесь тем, чем вам будет удобно, я продемонстрирую все на примере [Ninject](http://www.ninject.org/).

## Constructor injection

interface IChild

{

void Help(string message);

}

class Parent

{

IChild child = null;

public Parent(IChild child)

{

this.child = child;

}

public void Notify(string message)

{

if (child == null)

{

//decide which child parent should choose

}

else

{

child.Help(message);

}

}

}

class Child1 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 1 -"+ message);

}

}

class Child2 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 2 -" + message);

}

}

class Program

{

static void Main(string[] args)

{

Child1 child1 = new Child1();

Child2 child2 = new Child2();

Parent parent = new Parent(child1);

parent.Notify("Azaza");

}

}

## Method injection

interface IChild

{

void Help(string message);

}

class Parent

{

public void Notify(IChild child, string message)

{

if (child == null)

{

//decide which child parent should choose

}

else

{

child.Help(message);

}

}

}

class Child1 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 1 -"+ message);

}

}

class Child2 : IChild

{

public void Help(string message)

{

Console.WriteLine("I am child 2 -" + message);

}

}

## Property injection

class Parent

{

public IChild Child { set; get; }

public void Notify(string message)

{

if (Child== null)

{

//decide which child parent should choose

}

else

{

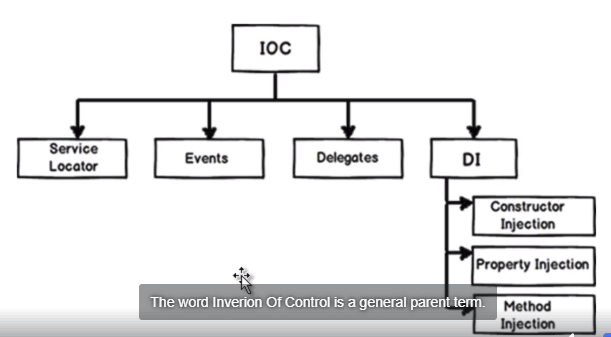
Child.Help(message);

}

}

}

## IOC



interface IDal

{

void Add();

}

class Customer

{

IDal dal = null;

public Customer(IDal dal)

{

this.dal = dal;

}

public void Add()

{

dal.Add();

}

}

class SqlServer : IDal

{

public void Add()

{

}

}

class Oracle : IDal

{

public void Add()

{

}

}

class Program

{

static void Main(string[] args)

{

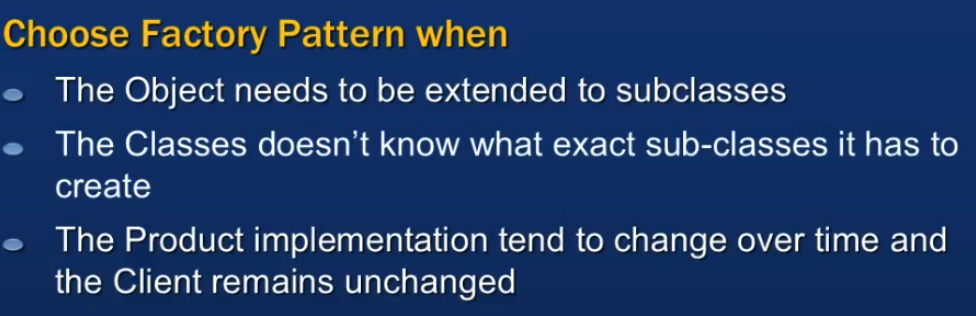
Customer customer = new Customer(new SqlServer());

customer.Add();

}

}

# Factory



enum FanType

{

TableFan,

CeilingFan,

ExhaustFan

}

interface IFan

{

void SwitchOn();

void SwitchOff();

}

class TableFan : IFan { }

class CeilingFan : IFan {}

class ExhaustFan : IFan { }

interface IFanFactory

{

IFan CreateFan(FanType type);

}

class FanFactory : IFanFactory

{

public IFan CreateFan(FanType type)

{

switch (type)

{

case FanType.TableFan:

return new TableFan();

case FanType.CeilingFan:

return new CeilingFan();

case FanType.ExhaustFan:

return new ExhaustFan();

default:

return new TableFan();

}

}

}

class Program

{

static void Main(string[] args)

{

IFanFactory simpleFactory = new FanFactory();

// Creation of a Fan using Simple Factory

IFan fan = simpleFactory.CreateFan(FanType.TableFan);

// Use created object

fan.SwitchOn();

Console.ReadLine();

}

}

### What is Simple Factory?

Simple **Factory** is a class with a public static **method** which will actually do the object creation task according to the input it gets.

Hide   Copy Code

public class Course**Factory**

{

public static AbstractCourse CreateCourse(string ScheduleType)

{

AbstractCourse objCourse = null;

switch(ScheduleType)

{

case "N":

objCourse = new NetCourse();

break;

case "J":

objCourse = new JavaCourse();

break;

*//Add more case conditions here for VC++ and C++*

}

objCourse.CreateCourseMaterial();

objCourse.CreateSchedule();

return objCourse;

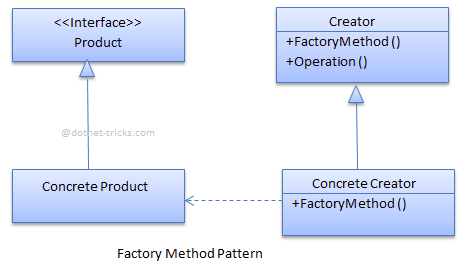
}

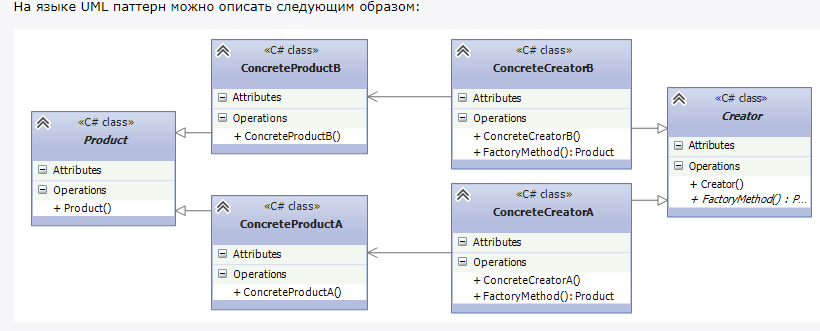
}

Advantages of the Approach 2

* Whenever new course will be introduced, the one which gets changed is **factory** not client code.
* As **Factory** is class any one can use it who have access. In short Course logic now can be reused.

# Factory method





**Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов

|  |  |
| --- | --- |
| **Плюсы:** | Создание объектов, независимо от их типов и сложности процесса создания. |
| **Минусы:** | Даже для одного объекта необходимо создать соответствующую фабрику, что увеличивает код. |

Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

### Когда надо применять паттерн

* Когда заранее неизвестно, объекты каких типов необходимо создавать
* Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
* Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

На языке UML паттерн можно описать следующим образом:

public abstract class AbstractCourse**Factory**

{

public AbstractCourse CreateCourse(string ScheduleType)

{

AbstractCourse objCourse = this.GetCourse(ScheduleType);

objCourse.CreateCourseMaterial();

objCourse.CreateSchedule();

return objCourse;

}

public abstract AbstractCourse GetCourse(string ScheduleType);

}

*Please note GetCourse****method****is****abstract***

Now everyone (every **factory**) will easily override this GetCourse **method**.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

public class CorporateCourse**Factory**:AbstractCourse**Factory**

{

public override AbstractCourse GetCourse(string ScheduleType)

{

switch(ScheduleType)

{

case "N":

return new CNetCourse();

case "J":

return new CJavaCourse();

default:

return null;

}

}

}

public class OnlineCourse**Factory** : AbstractCourse**Factory**

{

public override AbstractCourse GetCourse(string ScheduleType)

{

switch(ScheduleType)

{

case "N":

return new ONetCourse();

case "J":

return new OJavaCourse();

default:

return null;

}

}

}

abstract class Developer

{

public string Name { get; set; }

public Developer(string n)

{

Name = n;

}

// фабричный метод

abstract public House Create();

}

// строит панельные дома

class PanelDeveloper : Developer

{

public PanelDeveloper(string n) : base(n)

{ }

public override House Create()

{

return new PanelHouse();

}

}

// строит деревянные дома

class WoodDeveloper : Developer

{

public WoodDeveloper(string n) : base(n)

{ }

public override House Create()

{

return new WoodHouse();

}

}

abstract class House

{ }

class PanelHouse : House

{

public PanelHouse()

{

Console.WriteLine("Панельный дом построен");

}

}

class WoodHouse : House

{

public WoodHouse()

{

Console.WriteLine("Деревянный дом построен");

}

}

class Program

{

static void Main(string[] args)

{

Developer dev = new PanelDeveloper("ООО КирпичСтрой");

House house2 = dev.Create();

dev = new WoodDeveloper("Частный застройщик");

House house = dev.Create();

Console.ReadLine();

}

}

# Abstract factory

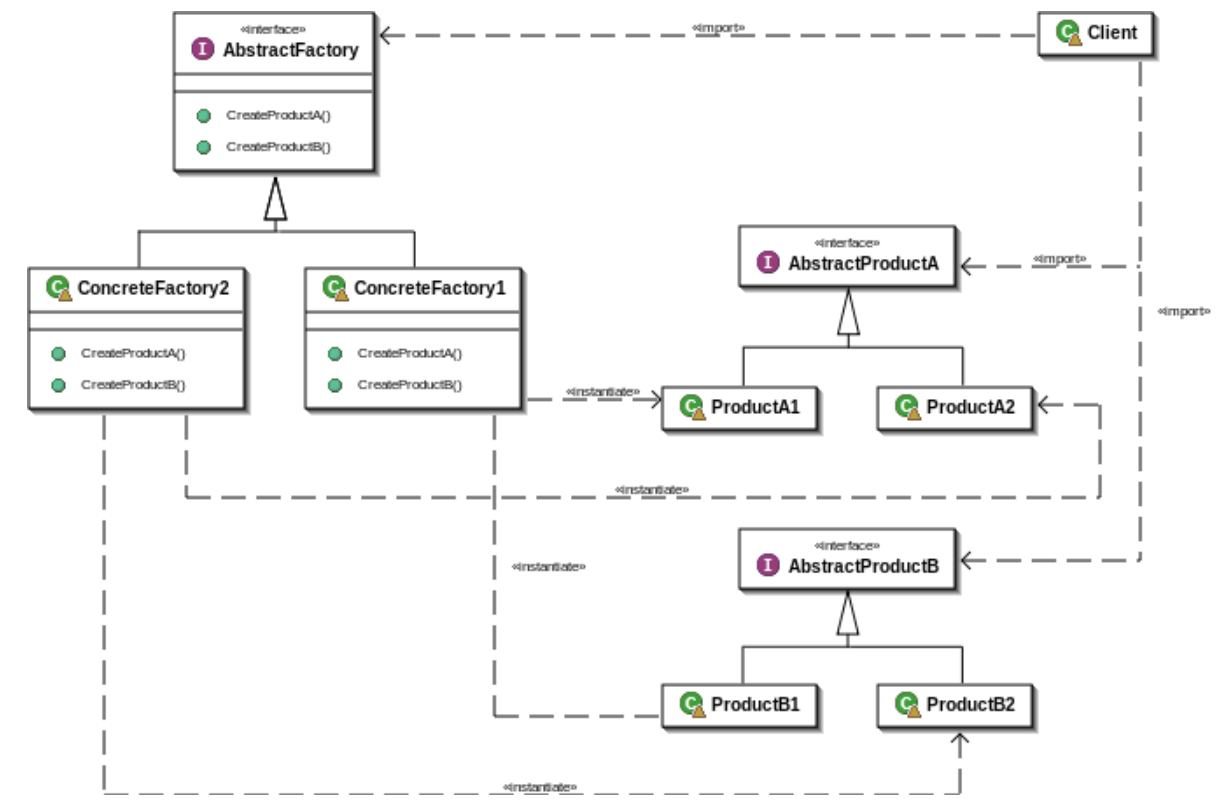
Паттерн "Абстрактная фабрика" (Abstract Factory) предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

### Когда использовать абстрактную фабрику

* Когда система не должна зависеть от способа создания и компоновки новых объектов
* Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

С помощью UML абстрактную фабрику можно представить следующим образом:

**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



|  |  |
| --- | --- |
| **Применяется в случаях:** | * Когда программа должна быть независимой от процесса и типов создаваемых новых объектов. * Когда необходимо создать семейства или группы взаимосвязанных объектов исключая возможность одновременного использования объектов из разных этих наборов в одном контексте[[1]](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B0%D1%8F_%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)#cite_note-1). |
| **Плюсы:** | * изолирует конкретные классы; * упрощает замену семейств продуктов; * гарантирует сочетаемость продуктов. |
| **Минусы:** | * сложно добавить поддержку нового вида продуктов. |

Create two interface called IFan and ITubelight.

Hide   Copy Code

interface IFan

{

void SwithOn();

}

interface ITubelight { }

Create Two concreate classes as given below by implementing IFan and ITubelight. Implementation of interfaces will be according to Indian electrical equipment standard.

Hide   Copy Code

class IndianFan : IFan { }

class IndianTubelight : ITubelight { }

Create an IElectrical**Factory** interface, this interface is actual **Abstract** **Factory** which will create families of related objects.

Hide   Copy Code

interface IElectrical**Factory**

{

IFan GetFan();

ITubelight GetTubeLight();

}

Create an IndianElectical**Factory** class and inherit it from IElectrical**Factory** interface. It will implementation of two methods called GetFan and will GetTubeLight. GetFan returns object of IndianFan class as IFan interface is implemented by IndianFan class. Similarly GetTubeLight will return the object of IndianTubelight class.

Hide   Copy Code

class IndianElectrical**Factory** : IElectrical**Factory**

{

public IFan GetFan()

{

return new IndianFan();

}

public ITubelight GetTubeLight()

{

return new IndianTubelight();

}

}

So far IndianElectical**Factory** class is ready to create IndianFan and IndianTubelight. Now let’s see how client can create them.

Hide   Copy Code

static void Main(string[] args)

{

IElectrical**Factory** electrical**Factory** = new IndianElectrical**Factory**();

IFan fan = electrical**Factory**.GetFan();

fan.SwithOn();

Console.ReadKey();

}

As we discussed above section, company wants to setup new electrical company in US and company name will be USElectrical**Factory**. In the same application we will add two new classes called USFan and USTubelight by implementing required interfaces. Here implementation of IFan and ITubeLight interfaces will be according to US electrical equipment standard.

Hide   Copy Code

class USFan : IFan { }

class USTubelight : ITubelight { }

Finally create USElectrical**Factory** class which is also inherited from IElectrical**Factory**.

Hide   Copy Code

class USElectrical**Factory** : IElectrical**Factory**

{

public IFan GetFan()

{

return new USFan();

}

public ITubelight GetTubeLight()

{

return new USTubelight();

}

}

Now if client wants to get USElectrical equipment then this can be done just by doing one change in client code, as shown below:

Hide   Copy Code

*//IElectrical****Factory*** *electrical****Factory*** *= new IndianElectrical****Factory****();*

IElectrical**Factory** electrical**Factory** = new USElectrical**Factory**();

public interface IProcessor

{

void PerformOperation();

}

public interface IHardDisk { void StoreData(); }

public interface IMonitor { void DisplayPicture();}

public class ExpensiveProcessor : IProcessor

{

public void PerformOperation()

{

Console.WriteLine("Operation will perform quickly");

}

}

public class CheapProcessor : IProcessor

{

public void PerformOperation()

{

Console.WriteLine("Operation will perform Slowly");

}

}

public class ExpensiveHDD : IHardDisk

{

public void StoreData()

{

Console.WriteLine("Data will take less time to store");

}

}

public class CheapHDD : IHardDisk

{

public void StoreData()

{

Console.WriteLine("Data will take more time to store");

}

}

public class HighResolutionMonitor : IMonitor

{

public void DisplayPicture()

{

Console.WriteLine("Picture quality is Best");

}

}

public class LowResolutionMonitor : IMonitor

{

public void DisplayPicture()

{

Console.WriteLine("Picture quality is Average");

}

}

**Factory** Code will be as follows

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

public interface IMachine**Factory**

{

IProcessor GetRam();

IHardDisk GetHardDisk();

IMonitor GetMonitor();

}

public class HighBudgetMachine : IMachine**Factory**

{

public IProcessor GetRam() { return new ExpensiveProcessor(); }

public IHardDisk GetHardDisk() { return new ExpensiveHDD(); }

public IMonitor GetMonitor() { return new HighResolutionMonitor(); }

}

public class LowBudgetMachine : IMachine**Factory**

{

public IProcessor GetRam() { return new CheapProcessor(); }

public IHardDisk GetHardDisk() { return new CheapHDD(); }

public IMonitor GetMonitor() { return new LowResolutionMonitor(); }

}

*//Let's say in future...Ram in the LowBudgetMachine is decided to upgrade then*

*//first make GetRam in LowBudgetMachine Virtual and create new class as follows*

public class AverageBudgetMachine : LowBudgetMachine

{

public override IProcessor GetRam()

{

return new ExpensiveProcessor();

}

}

Hide   Copy Code

public class ComputerShop

{

IMachine**Factory** category;

public ComputerShop(IMachine**Factory** \_category)

{

category = \_category;

}

public void AssembleMachine()

{

IProcessor processor = category.GetRam();

IHardDisk hdd = category.GetHardDisk();

IMonitor monitor = category.GetMonitor();

*//use all three and create machine*

processor.PerformOperation();

hdd.StoreData();

monitor.DisplayPicture();

}

}

Hide   Copy Code

*//Client Code*

IMachine**Factory** **factory** = new HighBudgetMachine();*// Or new LowBudgetMachine();*

ComputerShop shop = new ComputerShop(**factory**);

shop.AssembleMachine();

using System;

using System.Collections.Generic;

using System.Collections.Specialized;

using System.IO;

using System.Linq;

using System.Net;

using System.Text;

using System.Threading.Tasks;

namespace TrainCode

{

//абстрактный класс - оружие

abstract class Weapon

{

public abstract void Hit();

}

// абстрактный класс движение

abstract class Movement

{

public abstract void Move();

}

// класс арбалет

class Arbalet : Weapon

{

public override void Hit()

{

Console.WriteLine("Стреляем из арбалета");

}

}

// класс меч

class Sword : Weapon

{

public override void Hit()

{

Console.WriteLine("Бьем мечом");

}

}

// движение полета

class FlyMovement : Movement

{

public override void Move()

{

Console.WriteLine("Летим");

}

}

// движение - бег

class RunMovement : Movement

{

public override void Move()

{

Console.WriteLine("Бежим");

}

}

// класс абстрактной фабрики

abstract class HeroFactory

{

public abstract Movement CreateMovement();

public abstract Weapon CreateWeapon();

}

// Фабрика создания летящего героя с арбалетом

class ElfFactory : HeroFactory

{

public override Movement CreateMovement()

{

return new FlyMovement();

}

public override Weapon CreateWeapon()

{

return new Arbalet();

}

}

// Фабрика создания бегущего героя с мечом

class VoinFactory : HeroFactory

{

public override Movement CreateMovement()

{

return new RunMovement();

}

public override Weapon CreateWeapon()

{

return new Sword();

}

}

// клиент - сам супергерой

class Hero

{

private Weapon weapon;

private Movement movement;

public Hero(HeroFactory factory)

{

weapon = factory.CreateWeapon();

movement = factory.CreateMovement();

}

public void Run()

{

movement.Move();

}

public void Hit()

{

weapon.Hit();

}

}

class Program

{

static void Main(string[] args)

{

Hero elf = new Hero(new ElfFactory());

elf.Hit();

elf.Run();

Hero voin = new Hero(new VoinFactory());

voin.Hit();

voin.Run();

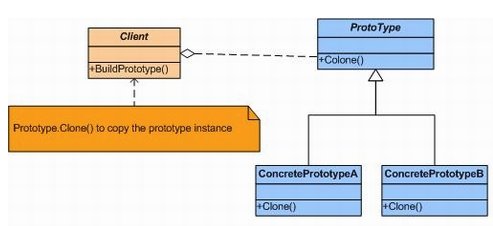
Console.ReadLine();

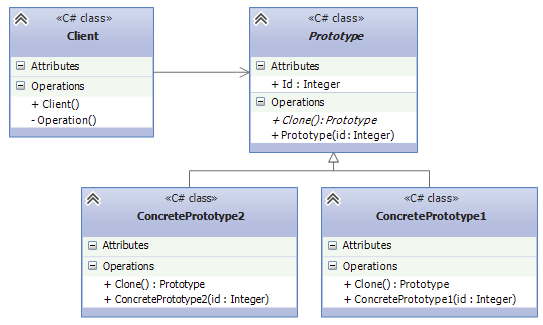
}

}

}

# Prototype





Паттерн Прототип (Prototype) позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть по сути данный паттерн предлагает технику клонирования объектов.

Когда использовать Прототип?

* Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения
* Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)
* Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.
* Створює глибоку копію комплексної ієрархії об'єктів
* Зменшення навантаження ініціалізації
* Багаторазові можливості - оптимізовані зусилля щодо кодування
* Спрощений процес копіювання об'єктів:
* Додаткові витрати на розробку
* Потребує уважності і точності при глибокому копіюванні

class Client

{

void Operation()

{

Prototype prototype = new ConcretePrototype1(1);

Prototype clone = prototype.Clone();

prototype = new ConcretePrototype2(2);

clone = prototype.Clone();

}

}

abstract class Prototype

{

public int Id { get; private set; }

public Prototype(int id)

{

this.Id = id;

}

public abstract Prototype Clone();

}

class ConcretePrototype1 : Prototype

{

public ConcretePrototype1(int id)

: base(id)

{ }

public override Prototype Clone()

{

return new ConcretePrototype1(Id);

}

}

class ConcretePrototype2 : Prototype

{

public ConcretePrototype2(int id)

: base(id)

{ }

public override Prototype Clone()

{

return new ConcretePrototype2(Id);

}

}

interface IFigure

{

IFigure Clone();

void GetInfo();

}

class Rectangle : IFigure

{

int width;

int height;

public Rectangle(int w, int h)

{

width = w;

height = h;

}

public IFigure Clone()

{

return new Rectangle(this.width, this.height);

}

public void GetInfo()

{

Console.WriteLine("Прямоугольник длиной {0} и шириной {1}", height, width);

}

}

class Circle : IFigure

{

int radius;

public Circle(int r)

{

radius = r;

}

public IFigure Clone()

{

return new Circle(this.radius);

}

public void GetInfo()

{

Console.WriteLine("Круг радиусом {0}", radius);

}

}

class Program

{

static void Main(string[] args)

{

IFigure figure = new Rectangle(30, 40);

IFigure clonedFigure = figure.Clone();

figure.GetInfo();

clonedFigure.GetInfo();

figure = new Circle(30);

clonedFigure = figure.Clone();

figure.GetInfo();

clonedFigure.GetInfo();

Console.Read();

}

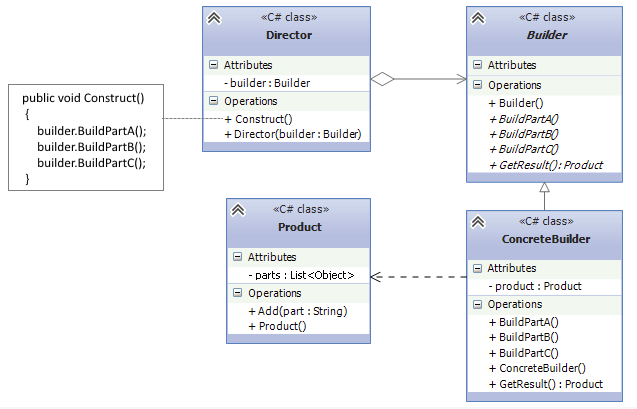
}

# Builder

Строитель (Builder) - шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

### Когда использовать паттерн Строитель?

* Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой
* Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания



* дозволяє змінювати внутрішнє представлення продукту;
* ізолює код, який реалізує конструювання та подання;
* дає більш тонкий контроль над процесом конструювання.
* Потрібно створити окремий ConcreteBuilder для кожного окремого продукту.

class Director

{

Builder builder;

public Director(Builder builder)

{

this.builder = builder;

}

public void Construct()

{

builder.BuildPartA();

builder.BuildPartB();

builder.BuildPartC();

}

}

abstract class Builder

{

public abstract void BuildPartA();

public abstract void BuildPartB();

public abstract void BuildPartC();

public abstract Product GetResult();

}

class Product

{

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

public void Add(string part)

{

parts.Add(part);

}

}

class ConcreteBuilder : Builder

{

Product product = new Product();

public override void BuildPartA()

{

product.Add("Part A");

}

public override void BuildPartB()

{

product.Add("Part B");

}

public override void BuildPartC()

{

product.Add("Part C");

}

public override Product GetResult()

{

return product;

}

}

class Program

{

static void Main(string[] args)

{

Builder builder = new ConcreteBuilder();

Director director = new Director(builder);

director.Construct();

Product product = builder.GetResult();

Console.Read();

}

}

//мука

class Flour

{

// какого сорта мука

public string Sort { get; set; }

}

// соль

class Salt

{ }

// пищевые добавки

class Additives

{

public string Name { get; set; }

}

class Bread

{

// мука

public Flour Flour { get; set; }

// соль

public Salt Salt { get; set; }

// пищевые добавки

public Additives Additives { get; set; }

public override string ToString()

{

StringBuilder sb = new StringBuilder();

if (Flour != null)

sb.Append(Flour.Sort + "\n");

if (Salt != null)

sb.Append("Соль \n");

if (Additives != null)

sb.Append("Добавки: " + Additives.Name + " \n");

return sb.ToString();

}

}

// абстрактный класс строителя

abstract class BreadBuilder

{

public Bread Bread { get; private set; }

public void CreateBread()

{

Bread = new Bread();

}

public abstract void SetFlour();

public abstract void SetSalt();

public abstract void SetAdditives();

}

// пекарь

class Baker

{

public Bread Bake(BreadBuilder breadBuilder)

{

breadBuilder.CreateBread();

breadBuilder.SetFlour();

breadBuilder.SetSalt();

breadBuilder.SetAdditives();

return breadBuilder.Bread;

}

}

// строитель для ржаного хлеба

class RyeBreadBuilder : BreadBuilder

{

public override void SetFlour()

{

this.Bread.Flour = new Flour { Sort = "Ржаная мука 1 сорт" };

}

public override void SetSalt()

{

this.Bread.Salt = new Salt();

}

public override void SetAdditives()

{

// не используется

}

}

// строитель для пшеничного хлеба

class WheatBreadBuilder : BreadBuilder

{

public override void SetFlour()

{

this.Bread.Flour = new Flour { Sort = "Пшеничная мука высший сорт" };

}

public override void SetSalt()

{

this.Bread.Salt = new Salt();

}

public override void SetAdditives()

{

this.Bread.Additives = new Additives { Name = "улучшитель хлебопекарный" };

}

}

class Program

{

static void Main(string[] args)

{

// содаем объект пекаря

Baker baker = new Baker();

// создаем билдер для ржаного хлеба

BreadBuilder builder = new RyeBreadBuilder();

// выпекаем

Bread ryeBread = baker.Bake(builder);

Console.WriteLine(ryeBread.ToString());

// оздаем билдер для пшеничного хлеба

builder = new WheatBreadBuilder();

Bread wheatBread = baker.Bake(builder);

Console.WriteLine(wheatBread.ToString());

Console.Read();

}

}

}

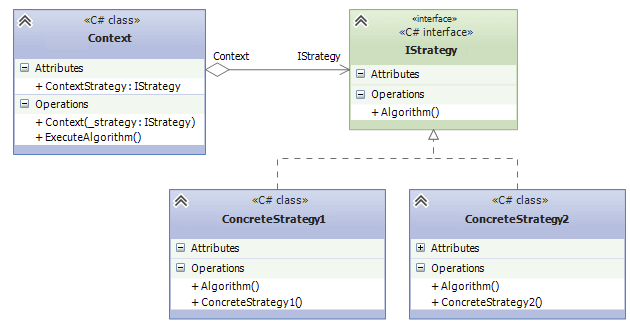
# Strategy

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

### Когда использовать стратегию?

* Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
* Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
* Когда необходимо менять поведение объектов на стадии выполнения программы
* Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации
* Можливість позбутися умовних операторів.
* Клієнт може вибирати найбільш влучну стратегію залежно від вимог щодо швидкодії і пам'яті.

1. Збільшення кількості об'єктів.
2. Клієнт має знати особливості реалізацій стратегій для вибору найбільш вдалої.



interface IMovable

{

void Move();

}

class PetrolMove : IMovable

{

public void Move()

{

Console.WriteLine("Перемещение на бензине");

}

}

class ElectricMove : IMovable

{

public void Move()

{

Console.WriteLine("Перемещение на электричестве");

}

}

class Car

{

protected int passengers; // кол-во пассажиров

protected string model; // модель автомобиля

public Car(int num, string model, IMovable mov)

{

this.passengers = num;

this.model = model;

Movable = mov;

}

public IMovable Movable { private get; set; }

public void Move()

{

Movable.Move();

}

}

class Program

{

static void Main(string[] args)

{

Car auto = new Car(4, "Volvo", new PetrolMove());

auto.Move();

auto.Movable = new ElectricMove();

auto.Move();

Console.Read();

}

}

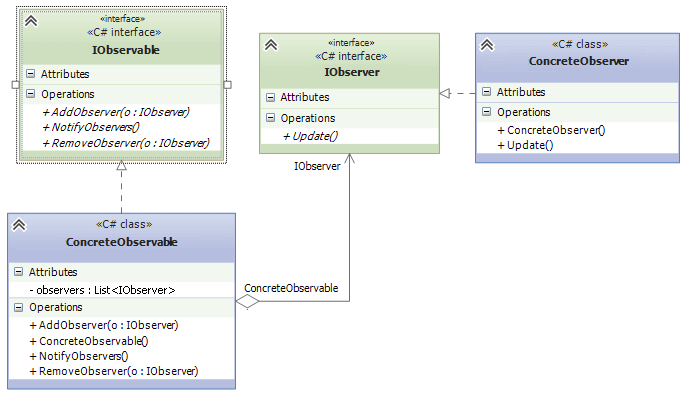
}

# Observer

Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

### Когда использовать паттерн Наблюдатель?

* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.



* Він підтримує принцип вільного зв'язку між об'єктами, які взаємодіють один з одним
* Дозволяє ефективно передавати дані іншим об'єктам, без будь-яких змін у класах Subject або Observer
* Спостерігачі можуть бути додані / видалені в будь-який момент часу
* Інтерфейс Observer повинен бути впроваджений ConcreteObserver, який передбачає успадкування. Композиції для композиції немає, оскільки інтерфейс Observer може бути екземплятором.
* Якщо це неправильно реалізовано, спостерігач може додати складність і призвести до ненавмисних проблем із продуктивністю.
* У програмному застосуванні повідомлення іноді можуть бути невибагливими і призвести до умов перегонів або непослідовності.

interface IObserver

{

void Update(Object ob);

}

interface IObservable

{

void RegisterObserver(IObserver o);

void RemoveObserver(IObserver o);

void NotifyObservers();

}

class Stock : IObservable

{

StockInfo sInfo; // информация о торгах

List<IObserver> observers;

public Stock()

{

observers = new List<IObserver>();

sInfo = new StockInfo();

}

public void RegisterObserver(IObserver o)

{

observers.Add(o);

}

public void RemoveObserver(IObserver o)

{

observers.Remove(o);

}

public void NotifyObservers()

{

foreach (IObserver o in observers)

{

o.Update(sInfo);

}

}

public void Market()

{

Random rnd = new Random();

sInfo.USD = rnd.Next(20, 40);

sInfo.Euro = rnd.Next(30, 50);

NotifyObservers();

}

}

class StockInfo

{

public int USD { get; set; }

public int Euro { get; set; }

}

class Broker : IObserver

{

public string Name { get; set; }

IObservable stock;

public Broker(string name, IObservable obs)

{

this.Name = name;

stock = obs;

stock.RegisterObserver(this);

}

public void Update(object ob)

{

StockInfo sInfo = (StockInfo)ob;

if (sInfo.USD > 30)

Console.WriteLine("Брокер {0} продает доллары; Курс доллара: {1}", this.Name, sInfo.USD);

else

Console.WriteLine("Брокер {0} покупает доллары; Курс доллара: {1}", this.Name, sInfo.USD);

}

public void StopTrade()

{

stock.RemoveObserver(this);

stock = null;

}

}

class Bank : IObserver

{

public string Name { get; set; }

IObservable stock;

public Bank(string name, IObservable obs)

{

this.Name = name;

stock = obs;

stock.RegisterObserver(this);

}

public void Update(object ob)

{

StockInfo sInfo = (StockInfo)ob;

if (sInfo.Euro > 40)

Console.WriteLine("Банк {0} продает евро; Курс евро: {1}", this.Name, sInfo.Euro);

else

Console.WriteLine("Банк {0} покупает евро; Курс евро: {1}", this.Name, sInfo.Euro);

}

}

class Program

{

static void Main(string[] args)

{

Stock stock = new Stock();

Bank bank = new Bank("ЮнитБанк", stock);

Broker broker = new Broker("Иван Иваныч", stock);

// имитация торгов

stock.Market();

// брокер прекращает наблюдать за торгами

broker.StopTrade();

// имитация торгов

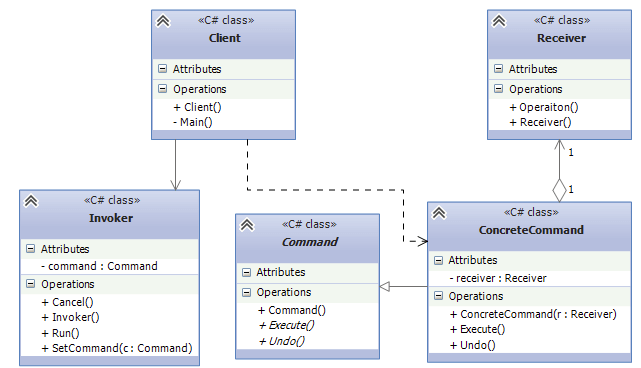
stock.Market();

Console.Read();

}

}

# Command



* Відокремлює класи, які викликають операцію від об'єкта, який вміє виконувати операцію
* Дозволяє створювати послідовність команд за допомогою системи черги
* Розширення для додавання нової команди є простими і можуть бути виконані без зміни існуючого коду
* Ви також можете визначити систему відкату з командним шаблоном, наприклад, у прикладі майстра, ми можемо написати метод відкату
* Збільшення кількості класів для кожної окремої команди

interface ICommand

{

void Execute();

void Undo();

}

// Receiver - Получатель

class TV

{

public void On()

{

Console.WriteLine("Телевизор включен!");

}

public void Off()

{

Console.WriteLine("Телевизор выключен...");

}

}

class TVOnCommand : ICommand

{

TV tv;

public TVOnCommand(TV tvSet)

{

tv = tvSet;

}

public void Execute()

{

tv.On();

}

public void Undo()

{

tv.Off();

}

}

// Invoker - инициатор

class Pult

{

ICommand command;

public Pult() { }

public void SetCommand(ICommand com)

{

command = com;

}

public void PressButton()

{

command.Execute();

}

public void PressUndo()

{

command.Undo();

}

}

class Program

{

static void Main(string[] args)

{

Pult pult = new Pult();

TV tv = new TV();

pult.SetCommand(new TVOnCommand(tv));

pult.PressButton();

pult.PressUndo();

Console.Read();

}

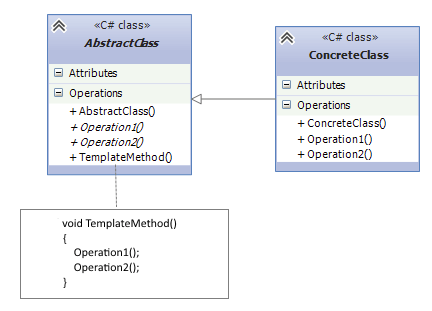
}

# Template method

Шаблонный метод (Template Method) определяет общий алгоритм поведения подклассов, позволяя им переопределить отдельные шаги этого алгоритма без изменения его структуры.

### Когда использовать шаблонный метод?

* Когда планируется, что в будущем подклассы должны будут переопределять различные этапы алгоритма без изменения его структуры
* Когда в классах, реализующим схожий алгоритм, происходит дублирование кода. Вынесение общего кода в шаблонный метод уменьшит его дублирование в подклассах.



* Немає копіювання коду.
* Лише кілька методів потрібно перевизначити.
* Гнучкість дозволяє підкласам вирішити, як здійснювати кроки в алгоритмі.
* Налагодження та розуміння послідовності алгоритму може іноді бути складною.
* Підтримка структури шаблонів може бути проблемою, оскільки зміни на будь-якому рівні (низький рівень або високий рівень) можуть перешкоджати реалізації.

abstract class Education

{

public void Learn()

{

Enter();

Study();

PassExams();

GetDocument();

}

public abstract void Enter();

public abstract void Study();

public virtual void PassExams()

{

Console.WriteLine("Сдаем выпускные экзамены");

}

public abstract void GetDocument();

}

class School : Education

{

public override void Enter()

{

Console.WriteLine("Идем в первый класс");

}

public override void Study()

{

Console.WriteLine("Посещаем уроки, делаем домашние задания");

}

public override void GetDocument()

{

Console.WriteLine("Получаем аттестат о среднем образовании");

}

}

class University : Education

{

public override void Enter()

{

Console.WriteLine("Сдаем вступительные экзамены и поступаем в ВУЗ");

}

public override void Study()

{

Console.WriteLine("Посещаем лекции");

Console.WriteLine("Проходим практику");

}

public override void PassExams()

{

Console.WriteLine("Сдаем экзамен по специальности");

}

public override void GetDocument()

{

Console.WriteLine("Получаем диплом о высшем образовании");

}

class Program

{

static void Main(string[] args)

{

School school = new School();

University university = new University();

school.Learn();

university.Learn();

Console.Read();

}

}

}

# Iterator

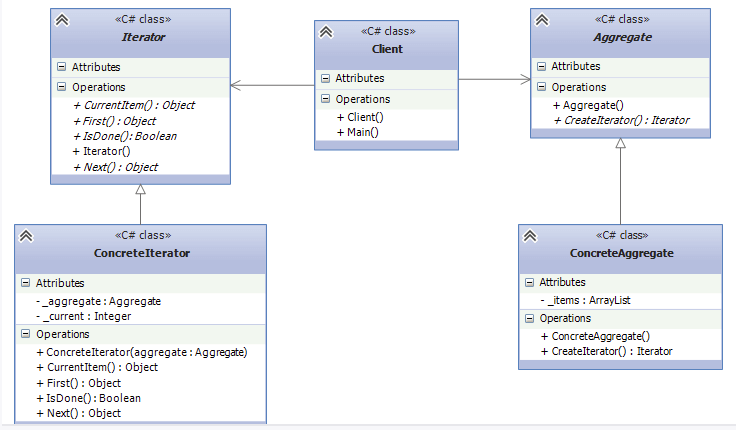
Паттерн Итератор (Iterator) предоставляет абстрактный интерфейс для последовательного доступа ко всем элементам составного объекта без раскрытия его внутренней структуры.

Наверное, всем программистам, работающим с языком C#, приходилось иметь дело с циклом foreach, который перебирает объекты в массиве или коллекции. При этом встроенных классов коллекций существует множество, и каждая из них отличается по своей структуре и поведению.

Ключевым моментом, который позволяет осуществить перебор коллекций с помощью foreach, является применения этими классами коллекций паттерна итератор, или проще говоря пары интерфейсов IEnumerable / IEnumerator

### Когда использовать итераторы?

* Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
* Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
* Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта



* Немає копіювання коду.
* Лише кілька методів потрібно перевизначити.
* Гнучкість дозволяє підкласам вирішити, як здійснювати кроки в алгоритмі.
* Налагодження та розуміння послідовності алгоритму може іноді бути складною.
* Підтримка структури шаблонів може бути проблемою, оскільки зміни на будь-якому рівні (низький рівень або високий рівень) можуть перешкоджати реалізації.

class Reader

{

public void SeeBooks(Library library)

{

IBookIterator iterator = library.CreateNumerator();

while (iterator.HasNext())

{

Book book = iterator.Next();

Console.WriteLine(book.Name);

}

}

}

interface IBookIterator

{

bool HasNext();

Book Next();

}

interface IBookNumerable

{

IBookIterator CreateNumerator();

int Count { get; }

Book this[int index] { get; }

}

class Book

{

public string Name { get; set; }

}

class Library : IBookNumerable

{

private Book[] books;

public Library()

{

books = new Book[]

{

new Book{Name="Война и мир"},

new Book {Name="Отцы и дети"},

new Book {Name="Вишневый сад"}

};

}

public int Count

{

get { return books.Length; }

}

public Book this[int index]

{

get { return books[index]; }

}

public IBookIterator CreateNumerator()

{

return new LibraryNumerator(this);

}

}

class LibraryNumerator : IBookIterator

{

IBookNumerable aggregate;

int index = 0;

public LibraryNumerator(IBookNumerable a)

{

aggregate = a;

}

public bool HasNext()

{

return index < aggregate.Count;

}

public Book Next()

{

return aggregate[index++];

}

}

class Program

{

static void Main(string[] args)

{

Library library = new Library();

Reader reader = new Reader();

reader.SeeBooks(library);

Console.Read();

}

}

}

class Client

{

public void Main()

{

Aggregate a = new ConcreteAggregate();

Iterator i = a.CreateIterator();

object item = i.First();

while (!i.IsDone())

{

item = i.Next();

}

}

}

abstract class Aggregate

{

public abstract Iterator CreateIterator();

public abstract int Count { get; protected set; }

public abstract object this[int index] { get; set; }

}

class ConcreteAggregate : Aggregate

{

private readonly ArrayList \_items = new ArrayList();

public override Iterator CreateIterator()

{

return new ConcreteIterator(this);

}

public override int Count

{

get { return \_items.Count; }

protected set { }

}

public override object this[int index]

{

get { return \_items[index]; }

set { \_items.Insert(index, value); }

}

}

abstract class Iterator

{

public abstract object First();

public abstract object Next();

public abstract bool IsDone();

public abstract object CurrentItem();

}

class ConcreteIterator : Iterator

{

private readonly Aggregate \_aggregate;

private int \_current;

public ConcreteIterator(Aggregate aggregate)

{

this.\_aggregate = aggregate;

}

public override object First()

{

return \_aggregate[0];

}

public override object Next()

{

object ret = null;

\_current++;

if (\_current < \_aggregate.Count)

{

ret = \_aggregate[\_current];

}

return ret;

}

public override object CurrentItem()

{

return \_aggregate[\_current];

}

public override bool IsDone()

{

return \_current >= \_aggregate.Count;

}

}

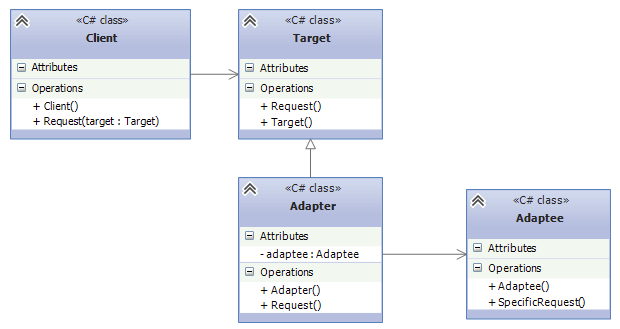
# Adapter

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

### Когда надо использовать Адаптер?

* Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям
* Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

Формальное определение паттерна на UML выглядит следующим образом:



* Допомагає досягти багаторазового використання та гнучкості.
* Клас клієнта не ускладнюється необхідністю використовувати інший інтерфейс і може використовувати поліморфізм для обміну між різними реалізаціями адаптерів.
* Всі запити пересилаються, тому спостерігається незначне збільшення накладних витрат.
* Іноді багато адаптацій потрібні по ланцюгу адаптера, щоб досягти потрібного типу.

class Client

{

public void Request(Target target)

{

target.Request();

}

}

// класс, к которому надо адаптировать другой класс

class Target

{

public virtual void Request()

{ }

}

// Адаптер

class Adapter : Target

{

private Adaptee adaptee = new Adaptee();

public override void Request()

{

adaptee.SpecificRequest();

}

}

// Адаптируемый класс

class Adaptee

{

public void SpecificRequest()

{ }

}

interface ITransport

{

void Drive();

}

// класс машины

class Auto : ITransport

{

public void Drive()

{

Console.WriteLine("Машина едет по дороге");

}

}

class Driver

{

public void Travel(ITransport transport)

{

transport.Drive();

}

}

// интерфейс животного

interface IAnimal

{

void Move();

}

// класс верблюда

class Camel : IAnimal

{

public void Move()

{

Console.WriteLine("Верблюд идет по пескам пустыни");

}

}

// Адаптер от Camel к ITransport

class CamelToTransportAdapter : ITransport

{

Camel camel;

public CamelToTransportAdapter(Camel c)

{

camel = c;

}

public void Drive()

{

camel.Move();

}

}

class Program

{

static void Main(string[] args)

{

// путешественник

Driver driver = new Driver();

// машина

Auto auto = new Auto();

// отправляемся в путешествие

driver.Travel(auto);

// встретились пески, надо использовать верблюда

Camel camel = new Camel();

// используем адаптер

ITransport camelTransport = new CamelToTransportAdapter(camel);

// продолжаем путь по пескам пустыни

driver.Travel(camelTransport);

Console.Read();

}

}

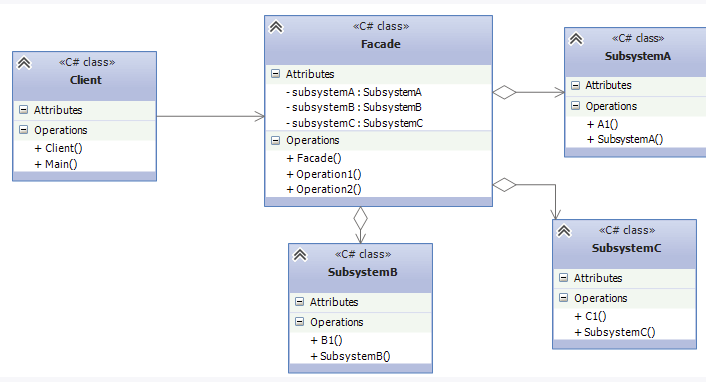
}

# Façade

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

### Когда использовать фасад?

* Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.
* Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.
* Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.



* Допомагає досягти багаторазового використання та гнучкості.
* Клас клієнта не ускладнюється необхідністю використовувати інший інтерфейс і може використовувати поліморфізм для обміну між різними реалізаціями адаптерів.
* Всі запити пересилаються, тому спостерігається незначне збільшення накладних витрат.
* Іноді багато адаптацій потрібні по ланцюгу адаптера, щоб досягти потрібного типу.

class SubsystemA

{

public void A1()

{ }

}

class SubsystemB

{

public void B1()

{ }

}

class SubsystemC

{

public void C1()

{ }

}

class Facade

{

SubsystemA subsystemA;

SubsystemB subsystemB;

SubsystemC subsystemC;

public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc)

{

subsystemA = sa;

subsystemB = sb;

subsystemC = sc;

}

public void Operation1()

{

subsystemA.A1();

subsystemB.B1();

subsystemC.C1();

}

public void Operation2()

{

subsystemB.B1();

subsystemC.C1();

}

}

class Program

{

static void Main(string[] args)

{

Facade facade = new Facade(new SubsystemA(), new SubsystemB(), new SubsystemC());

facade.Operation1();

facade.Operation2();

Console.Read();

}

}

// текстовый редактор

class TextEditor

{

public void CreateCode()

{

Console.WriteLine("Написание кода");

}

public void Save()

{

Console.WriteLine("Сохранение кода");

}

}

class Compiller

{

public void Compile()

{

Console.WriteLine("Компиляция приложения");

}

}

class CLR

{

public void Execute()

{

Console.WriteLine("Выполнение приложения");

}

public void Finish()

{

Console.WriteLine("Завершение работы приложения");

}

}

class VisualStudioFacade

{

TextEditor textEditor;

Compiller compiller;

CLR clr;

public VisualStudioFacade(TextEditor te, Compiller compil, CLR clr)

{

this.textEditor = te;

this.compiller = compil;

this.clr = clr;

}

public void Start()

{

textEditor.CreateCode();

textEditor.Save();

compiller.Compile();

clr.Execute();

}

public void Stop()

{

clr.Finish();

}

}

class Programmer

{

public void CreateApplication(VisualStudioFacade facade)

{

facade.Start();

facade.Stop();

}

}

class Program

{

static void Main(string[] args)

{

TextEditor textEditor = new TextEditor();

Compiller compiller = new Compiller();

CLR clr = new CLR();

VisualStudioFacade ide = new VisualStudioFacade(textEditor, compiller, clr);

Programmer programmer = new Programmer();

programmer.CreateApplication(ide);

Console.Read();

}

}

}

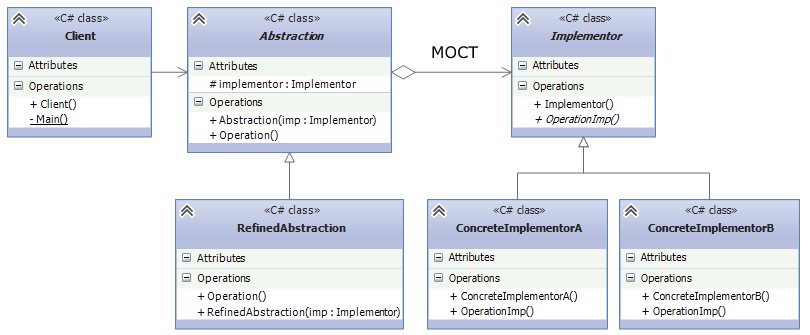
# Bridge

Мост (Bridge) - структурный шаблон проектирования, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.

Даже если мы отделим абстракцию от конкретных реализаций, то у нас все равно все наследуемые классы будут жестко привязаны к интерфейсу, определяемому в базовом абстрактном классе. Для преодоления жестких связей и служит паттерн Мост.

### Когда использовать данный паттерн?

* Когда надо избежать постоянной привязки абстракции к реализации
* Когда наряду с реализацией надо изменять и абстракцию независимо друг от друга. То есть изменения в абстракции не должно привести к изменениям в реализации



* Від'єднання абстракції від реалізації
* Зменшення кількості підкласів
* Чистий код і зменшенням розміру виконуваного файлу
* Інтерфейс і реалізація можуть варіюватися самостійно
* Покращена розширюваність - абстракції та впровадження можуть бути розширені самостійно
* Підвищує складність.
* Подвійна спрямованість - це матиме невеликий вплив на продуктивність.
* **Abstraction**: определяет базовый интерфейс и хранит ссылку на объект Implementor. Выполнение операций в Abstraction делегируется методам объекта Implementor
* **RefinedAbstraction**: уточненная абстракция, наследуется от Abstraction и расширяет унаследованный интерфейс
* **Implementor**: определяет базовый интерфейс для конкретных реализаций. Как правило, Implementor определяет только примитивные операции. Более сложные операции, которые базируются на примитивных, определяются в Abstraction
* **ConcreteImplementorA** и **ConcreteImplementorB**: конкретные реализации, которые унаследованы от Implementor
* **Client**: использует объекты Abstraction

abstract class Abstraction

{

protected Implementor implementor;

public Implementor Implementor

{

set { implementor = value; }

}

public Abstraction(Implementor imp)

{

implementor = imp;

}

public virtual void Operation()

{

implementor.OperationImp();

}

}

abstract class Implementor

{

public abstract void OperationImp();

}

class RefinedAbstraction : Abstraction

{

public RefinedAbstraction(Implementor imp)

: base(imp)

{ }

public override void Operation()

{

}

}

class ConcreteImplementorA : Implementor

{

public override void OperationImp()

{

}

}

class ConcreteImplementorB : Implementor

{

public override void OperationImp()

{

}

}

class Program

{

static void Main(string[] args)

{

Abstraction abstraction;

abstraction = new RefinedAbstraction(new ConcreteImplementorA());

abstraction.Operation();

abstraction.Implementor = new ConcreteImplementorB();

abstraction.Operation();

Console.Read();

}

}

interface ILanguage

{

void Build();

void Execute();

}

class CPPLanguage : ILanguage

{

public void Build()

{

Console.WriteLine("С помощью компилятора C++ компилируем программу в бинарный код");

}

public void Execute()

{

Console.WriteLine("Запускаем исполняемый файл программы");

}

}

class CSharpLanguage : ILanguage

{

public void Build()

{

Console.WriteLine("С помощью компилятора Roslyn компилируем исходный код в файл exe");

}

public void Execute()

{

Console.WriteLine("JIT компилирует программу бинарный код");

Console.WriteLine("CLR выполняет скомпилированный бинарный код");

}

}

abstract class Programmer

{

protected ILanguage language;

public ILanguage Language

{

set { language = value; }

}

public Programmer(ILanguage lang)

{

language = lang;

}

public virtual void DoWork()

{

language.Build();

language.Execute();

}

public abstract void EarnMoney();

}

class FreelanceProgrammer : Programmer

{

public FreelanceProgrammer(ILanguage lang) : base(lang)

{

}

public override void EarnMoney()

{

Console.WriteLine("Получаем оплату за выполненный заказ");

}

}

class CorporateProgrammer : Programmer

{

public CorporateProgrammer(ILanguage lang)

: base(lang)

{

}

public override void EarnMoney()

{

Console.WriteLine("Получаем в конце месяца зарплату");

}

}

class Program

{

static void Main(string[] args)

{

// создаем нового программиста, он работает с с++

Programmer freelancer = new FreelanceProgrammer(new CPPLanguage());

freelancer.DoWork();

freelancer.EarnMoney();

// пришел новый заказ, но теперь нужен c#

freelancer.Language = new CSharpLanguage();

freelancer.DoWork();

freelancer.EarnMoney();

Console.Read();

}

}

}