Pattern in .NET Core

Andrei Ignat

Contents

Patterns Book - examples from .NET Core source code

This book is a collection of patterns from the .NET Core source code. The patterns are presented in a way that makes them easy to understand and use in your own projects. Each pattern is accompanied by a detailed explanation and example code that demonstrates how it can be implemented in practice.

Pattern: AbstractFactory

Read online at https://ignatandrei.github.io/patterns/docs/patterns/AbstractFactory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Purpose of .NET implementation

You want to create commands for any specific database type in order to obtain data from a database. This means you can switch between different databases (e.g., SQL Server, MySQL, PostgreSQL) without changing the core logic of your application. The factory will provide the appropriate concrete implementation of the DBConnection for the database in use.. By using an Abstract Factory, your application can remain agnostic of the specific type of database it is interacting with.

Example in .NET :

AbstractFactory

```
using Microsoft.Data.SqlClient;
using System.Data.Common;

namespace AbstractFactory;
internal class AbstractFactoryDemo
{
    public static void Demo()
    {
        //create DbConnection factory by using the instance of SqlConnection
        DbConnection connection = new SqlConnection();
        //create DbCommand instance by using the instance of SqlConnection
        DbCommand command = connection.CreateCommand();
        //really, the DBCommand is a SqlCommand
        SqlCommand? sqlCommand = command as SqlCommand;
        //check if the DbCommand is a SqlCommand
        Console.WriteLine($"DbCommand is SqlCommand: {sqlCommand != null}");
    }
}
```

See Source Code for Microsoft implementation of AbstractFactory

SourceCode DbConnection : https://source.dot.net/#System.Data.Common/System/Data/Common/DbConnection.cs

Learn More

Wikipedia : https://en.wikipedia.org/wiki/Abstract_factory_pattern

Homework

Imagine you want to produce loggers. You have a logger that logs to a file and a logger that logs to a console and a Nothing Logger - a logger that does nothing. Implement an abstract factory that will allow you to create a logger factory that will create a logger that logs to a file or to a console or nothing.

Pattern: Adapter

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Adapter

Adapter design pattern allows the interface of an existing class to be used as another interface.It is often used to make existing classes work with others without modifying their source code.

Purpose of .NET implementation

You want to transfer data from a database Command to a DataTable. The SQLiteDataAdapter serves as an adapter between the SQLiteCommand object (which represents a SQL command or stored procedure to execute against a SQLite database) and the DataTable object (which represents in-memory data in a tabular format).

Examples in .NET :

SQLiteDataAdapter

```
namespace Adaptor;
internal class SQLiteDataAdapterDemo
{
    /// <summary>
    ///Adaptee  - Command
    ///Target  - DataTable
    ///Adapter  - SqlDataAdapter
    ///Target Method - Fill(Dataset instance)
    /// </summary>
    /// <returns></returns>
    public static async Task MainSqliteAdapterAsync()
    {
        //Target: Creates a DataTable instance to hold the data fetched from the database.
        var dataFormats = new DataTable();
        Console.WriteLine(dataFormats.Rows.Count);
        Console.WriteLine(dataFormats.Columns.Count);
        using (var con = new SQLiteConnection())
```

```
        {
            con.ConnectionString = "Data Source=CatalogRo.sqlite3";
            await con.OpenAsync();

            using (var cmd = new SQLiteCommand())
            {
                // Adaptee: Sets the SQL command text to fetch all records from the 'Format
                cmd.CommandText = "select * from Format";
                cmd.Connection = con;
                using (var adapt = new SQLiteDataAdapter(cmd))
                {
                    // Adapter: Fills the DataTable (Target) with data fetched using the SQ
                    adapt.Fill(dataFormats);
                }
            }

        }
        Console.WriteLine(dataFormats.Rows.Count);
        Console.WriteLine(dataFormats.Columns.Count);
    }
}
```

EncodingAdapter

```
namespace Adaptor;
internal class EncodingAdapterDemo
{
    /// <summary>
    ///Adaptee  - string
    ///Target  - bytes
    ///Adapter  - encoding
    ///Target Method - GetBytes
    /// </summary>
    public static void AdapterStringByte()
    {
        var url = "http://msprogrammer.serviciipeweb.ro";
        Encoding e = new ASCIIEncoding();
        var b = e.GetBytes(url);

        Console.WriteLine($"from {e.EncodingName} number bytes {b.Length}");

        e = new UTF32Encoding();
        b = e.GetBytes(url);
        Console.WriteLine($"from {e.EncodingName} number bytes {b.Length}");

    }
```

```
}
```

See Source Code for Microsoft implementation of Adapter

SourceCode AsciiEncoding : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Text/ASCIIEncoding.cs

SourceCode SqliteDataAdapter : https://github.com/mono/mono/blob/9bb01f57a126dab35f070ce238457931e9814c33/mcs/class/Mono.Data.Sqlite/Mono.Data.Sqlite_2.0/SQLiteDataAdapter.cs#L20

SourceCode UTF32Encoding : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Text/UTF32Encoding.cs

Learn More

C2Wiki : http://wiki.c2.com/?AdapterPattern

dofactory : http://www.dofactory.com/net/Adapter-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-Adapter

Wikipedia : https://en.wikipedia.org/wiki/Adapter_pattern

Homework

iPhone 7 does not have a headphone jack. Implement an adapter that will allow you to use your old headphones , that have jack, with the iPhone 7.

Pattern: Builder

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Builder

The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

Purpose of .NET implementation

You want to let the developer construct a SqlConnectionString. The SqlConnectionStringBuilder class provides a way to construct connection strings for SQL Server databases. Instead of requiring the developer to construct a connection string in one go, potentially leading to mistakes or omissions, SqlConnectionStringBuilder allows for the step-by-step construction of a connection string. This can help to ensure that all necessary parameters are included and that the connection string is correctly formatted. Once all necessary parameters have been set, the ConnectionString property of the SqlConnectionStringBuilder object can be used to retrieve the constructed connection string.

Examples in .NET :

SqlConnectionStringBuilder

```
namespace Builder;
internal class ConnectionStringDemo
{
    public static void ConnectionString()
    {
        //start example 2
        SqlConnectionStringBuilder build = new ();
        build.DataSource = ".";
        build.InitialCatalog = "MyDatabase";
        build.ConnectTimeout = 30;
        // Outputs the constructed connection string to the console. This demonstrates the
        Console.WriteLine(build.ConnectionString);
        //end example 2
    }
}
```

UriBuilder

```
namespace Builder;
static class UriBuilderDemo
{
    public static void UriMod()
    {
        //start example 1
        var uri = new Uri("https://msprogrammer.serviciipeweb.ro/category/friday-links/");
        var b = new UriBuilder(uri);
        //changing part
        b.Scheme = "http";
        //now we have http://msprogrammer.serviciipeweb.ro/category/friday-links/
        Console.WriteLine(b.Uri);
        //changing part
        b.Path = "2018/03/05/design-patterns-class/";
        //now we have http://msprogrammer.serviciipeweb.ro/2018/03/05/design-patterns-class/
        Console.WriteLine(b.Uri);
        //end example 1
    }
}
```

See Source Code for Microsoft implementation of Builder

SourceCode SqlConnectionStringBuilder : https://referencesource.microsoft.com/#System.Data/fx/src/data/System/Data/SqlClient/SqlConnectionStringBuilder.cs

SourceCode UriBuilder : https://source.dot.net/#System.Private.Uri/System/UriBuilder.cs

Learn More

C2Wiki : http://wiki.c2.com/?BuilderPattern

dofactory : http://www.dofactory.com/net/Builder-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-Builder

Wikipedia : https://en.wikipedia.org/wiki/Builder_pattern

Homework

Imagine that you have a logger that logs to a file and to a console. Implement a builder that will allow you to create a logger with different configurations. For example, you can set the log level, the log format, and the log destination.

Pattern: Chain

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Chain

Chain of responsibility pattern allows an object to send a command without knowing what object will receive and handle it.Chain the receiving objects and pass the request along the chain until an object handles it.

Purpose of .NET implementation

You want to pass the exception to the possible handlers / catch blocks in the all functions in the call stack. Exception bubbling in .NET exemplifies the Chain of Responsibility pattern by allowing exceptions to be passed along a chain of potential handlers (catch blocks) until one is found that can handle the exception. This can be useful when you want to ensure that exceptions are handled at the appropriate level of abstraction, rather than being caught and handled at a lower level where they may not be fully understood or properly addressed. By allowing exceptions to bubble up the call stack, you can ensure that they are handled in a consistent and appropriate manner, regardless of where they occur in the code. This mechanism decouples the thrower of the exception from the handlers, providing a flexible and dynamic way of managing errors that occur during runtime.

Example in .NET :

Chain

```
namespace Chain;
```

```
public static class ChainDemo
{
    public static int SecondException()
    {
        try
        {
            // Calls 'FirstException' method which is known to throw an exception.
            FirstException();
            return 5;
        }
        catch (Exception ex)
        {
            // Throws a new exception, chaining the caught exception 'ex' as the inner excep
            // This adds context to the exception, indicating it originated from 'SecondExc
            throw new Exception($"from {nameof(SecondException)}", ex);
        }
    }
    static int FirstException()
    {

        throw new ArgumentException("argument");
    }
}
```

See Source Code for Microsoft implementation of Chain

SourceCode ApplicationBuilderExtensions.UseMiddleware : https://source.dot.net/#Microsoft.AspNetCore.Builder/ApplicationBuilderExtensions.cs

SourceCode UseMiddlewareExtensions.UseMiddleware : https://source.dot.net/#System.Private.CoreLib/src/System/Object.CoreCLR.cs

Learn More

C2Wiki : https://wiki.c2.com/?ChainOfResponsibilityPattern

dofactory : http://www.dofactory.com/net/chain-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-chain

Wikipedia : https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

Homework

Implement a middleware in ASP. NET Core that intercepts the exception and logs it to the database. The middleware should be able to pass the exception to the next middleware in the chain.

Pattern: Decorator

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Decorator

Decorator allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.

Purpose of .NET implementation

Stream is a perfect example of the Decorator pattern. Imagine you want to write a text to a file, but you want to add some additional functionality(or not, at will) to the stream, such as compression or encryption.. By using the Decorator pattern, you can easily compose streams with different behaviors to create custom stream objects that meet specific requirements. Each stream class focuses on a single responsibility. FileStream handles file I/O, CryptoStream handles encryption and decryption, and GZipStream handles compression and decompression. This makes the classes easier to understand, test, and maintain..

Example in .NET :

Decorator

```
namespace Decorator;
internal class DecoratorDemo
{
    public static void Stream_Crypto_Gzip()
    {
        string nameFile = "test.txt";
        if (File.Exists(nameFile))
            File.Delete(nameFile);
        byte[] data = ASCIIEncoding.ASCII.GetBytes("Hello World!");
        // Creates a FileStream (the ConcreteComponent in the Decorator pattern context).
        using (var stream = new FileStream(nameFile, FileMode.OpenOrCreate, FileAccess.Write
        {
            //stream.Write(data, 0, data.Length);
            //return;

            var cryptic = new DESCryptoServiceProvider();

            cryptic.Key = ASCIIEncoding.ASCII.GetBytes("ABCDEFGH");
            cryptic.IV = ASCIIEncoding.ASCII.GetBytes("ABCDEFGH");
            // Decorates the FileStream with a CryptoStream (the first Decorator).
            using (var crStream = new CryptoStream(stream, cryptic.CreateEncryptor(), Crypto
            {
                // Further decorates the CryptoStream with a GZipStream (the second Decorato
```

```
                using (var gz = new GZipStream(crStream, CompressionLevel.Optimal))
                {
                    gz.Write(data, 0, data.Length);
                }

            }
        }
    }
}
```

See Source Code for Microsoft implementation of Decorator

SourceCode CryptoStream : https://source.dot.net/#System.Security.Cryptography/System/Security/Cryptography/CryptoStream.cs

SourceCode GZipStream : https://source.dot.net/#System.IO.Compression/System/IO/Compression/GZipStream.cs

Learn More

C2Wiki : http://wiki.c2.com/?DecoratorPattern

dofactory : http://www.dofactory.com/net/decorator-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-decorator

Wikipedia : https://en.wikipedia.org/wiki/Prototype_pattern

Homework

1.Create a LoggingDbConnectionDecorator class that adds logging functionality to a DbConnection object.. This class should log the details of the operations performed on the DbConnection (like opening a connection, closing a connection, executing a command, etc.) to a log file or console.. 2.Your task is to model a coffee shop ordering system using the Decorator design pattern. The base component will be a coffee, and you will create decorators for adding milk, sugar, and chocolate.. The coffee should be able to display the condiments in a Display method and calculate the price of the coffee with milk, sugar, and chocolate.

Pattern: Facade

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Facade

Facade is is an object that provides a simplified interface to a larger body of code, such as a class library.

Purpose of .NET implementation

You are the creator of EFCore that provides ORM capabilities to the developers. You want also to have a simplified interface to interact with the underlying database and provide simple methods like EnsureCreated(), BeginTransaction(), etc. The Facade pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use by providing a single entry point for common operations.

Example in .NET :

Facade

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;

namespace Facade;
internal class FacadeDemo
{
    public static void ExecuteSql()
    {
        MyDbContext cnt = new();

        DatabaseFacade dbFacade = cnt.Database;
        //calling the facade for create the database
        dbFacade.EnsureCreated();
        //calling the facade for begin a transaction
        dbFacade.BeginTransaction();
    }
}

public class MyDbContext:DbContext
{

}
```

See Source Code for Microsoft implementation of Facade

SourceCode DatabaseFacade : https://github.com/dotnet/efcore/blob/3163cb9a0677f94bd986dcdbe3d6026d4f743c73/src/EFCore/Infrastructure/DatabaseFacade.cs#L15

Learn More

C2Wiki : http://wiki.c2.com/?FacadePattern

dofactory : http://www.dofactory.com/net/facade-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-facade

Wikipedia : https://en.wikipedia.org/wiki/Facade_pattern

Homework

Implement a Facade that will allow you to display a question in a MessageBox with a single method call in a console application and return yes/no as a result.

Pattern: Factory

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Factory

A factory is a function or method that returns objects of a varying prototype or class from some method call, which is assumed to be new.

Purpose of .NET implementation

For getting data from Web, you can have a HttpWebRequest or FtpWebRequest. The type of the request depends on the protocol you want to use : HTTP or FTP. You want to make easier for the developer to create the appropriate request object based on the string that starts with the protocol. So you can have the Factory pattern method : WebRequest.Create.

Example in .NET :

Factory

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.Internal;
using System.Data.Common;
using System.Globalization;
using System.Net;
using System.Web.Mvc;

namespace Factory;
internal class FactoryDemo
{
    public static void DemoWebRequest()
    {

        //WebRequest.Create is a factory - can create HttpWebRequest or FtpWebRequest
        HttpWebRequest hwr = (HttpWebRequest)WebRequest.Create("http://www.yahoo.com");

    }
    public static void DemoConvert()
```

```
    {
        string value = "1,500";
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");

        Console.WriteLine(Convert.ToDouble(value));

        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR");
        Console.WriteLine(Convert.ToDouble(value));

    }
    static void RegisterControllerFactory()
    {
        ControllerBuilder.Current.SetControllerFactory(new MyControllerFactory());
    }
}

//default controller factory is a factory of controllers
class MyControllerFactory : System.Web.Mvc.DefaultControllerFactory
{

    public override IController CreateController(System.Web.Routing.RequestContext requestCo
    {
        if (controllerName == "andrei")
            return null;//maybe controller not found

        return base.CreateController(requestContext, controllerName);
    }
}
```

See Source Code for Microsoft implementation of Factory

SourceCode Convert.ToDouble : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Convert.cs

SourceCode DefaultControllerFactory : https://source.dot.net/#Microsoft.AspNetCore.Mvc.Core/Controllers/DefaultControllerFactory.cs

SourceCode WebRequest.Create : https://referencesource.microsoft.com/#System/net/System/Net/WebRequest.cs

Learn More

C2Wiki : http://wiki.c2.com/?FactoryPattern

dofactory : http://www.dofactory.com/net/factory-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab

=readme-ov-file#-factory

Hammer Factories : https://www.danstroot.com/posts/2018-10-03-hammer-factories

Wikipedia : https://en.wikipedia.org/wiki/Factory_pattern

Homework

You want to create multiple types of drinks(water, tea, coffee). With an IDrink interface create a factory method (with a parameter) to create a drink.

Pattern: FluentInterface

Read online at https://ignatandrei.github.io/patterns/docs/patterns/FluentInterface

Fluent interface allows you do have method chaining.

Purpose of .NET implementation

The methods of a service collection instance is called multiple times(e.g. for .AddSingleton). You want to ensure that the programmer can write code that is easy to write, easy to read, and easy to maintain. So from each method you return the instance of the service collection, so you can chain the methods.

Example in .NET :

FluentInterface

```
using Microsoft.Extensions.DependencyInjection;
using System.Data;
using System.Data.Common;

namespace FluentInterface;
internal static class FluentInterfaceDemo
{
    public static ServiceCollection AddServices(this ServiceCollection sc)
    {
        //just for demo, does not make sense
        sc
            .AddSingleton<IComparable>((sp) =>
            {
                //does not matter
                return 1970;
            })
            .AddSingleton<IComparable<Int32>>((sp) =>
            {
```

```
                //does not matter
                return 16;
            });
        //this way you can chain the calls , making a fluent interface
        return sc;


    }
}
```

See Source Code for Microsoft implementation of FluentInterface

SourceCode Microsoft.Extensions.DependencyInjection.ServiceCollectionServiceExtensions.AddSingleton : https://source.dot.net/#Microsoft.Extensions.DependencyInjection.Abstractions/ServiceCollectionServiceExtensions.cs

Learn More

Wikipedia : https://en.wikipedia.org/wiki/Software_design_pattern

Homework

Implement a class person that you can see the first name and last name as fluent interface.

Pattern: Flyweight

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Flyweight

Flyweight pattern is used to reduce the memory and resource usage for complex models containing a large number of similar objects.

Purpose of .NET implementation

String is costly as allocation in .NET, so you want to reuse the same string in the memory. The string.Intern method is used to retrieve a reference to a string from the intern pool, which is a table of unique strings maintained by .NET. If the string you are trying to intern is already in the intern pool, the method returns a reference to the string in the intern pool instead of creating a new string object.

Example in .NET :

Flyweight

```
using System.Text;
```

```
namespace Flyweight;
internal class FlyweightDemo
{
    public static void Demo()
    {
        var str = "Andrei Ignat";
        var str2 = string.Intern(str);
        var str3 = new StringBuilder("Andrei").Append(" Ignat").ToString();
        Console.WriteLine($"str == str2: Value {str==str2} Reference {Object.ReferenceEquals
        Console.WriteLine($"str == str3: Value {str==str3} Reference {Object.ReferenceEquals

    }
}
```

See Source Code for Microsoft implementation of Flyweight

SourceCode String.Intern : https://source.dot.net/#System.Private.CoreLib/src/System/Object.CoreCLR.cs

Learn More

Wikipedia : https://en.wikipedia.org/wiki/Flyweight_pattern

Homework

Make an exchange rate system. The symbol and names of the currency are the same for all the currencies. The exchange rate is different for each currency. Implement a flyweight that will allow you to create a currency with a symbol and a name and to get the exchange rate for the currency.

Pattern: IOC

Read online at https://ignatandrei.github.io/patterns/docs/patterns/IOC

Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework.It's a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework.

Purpose of .NET implementation

You want people to work with the file system . Possibly without a file or a directory that exists, but just with implementation. You want to be able to test the code without the file system. So you want to abstract the file system. See file system abstraction in the links below.

Examples in .NET :

IOC

```
namespace IOC;
public class NotificationService
{
    private readonly IMessageService _messageService;

    public NotificationService(IMessageService messageService)
    {
        _messageService = messageService;
    }

    public void SendNotification(string message)
    {
        _messageService.SendMessage(message);
    }
}
public interface IMessageService
{
    void SendMessage(string message);
}
```

DI

```
namespace IOC;
public class SMSService : IMessageService
{
    public void SendMessage(string message)
    {
        Console.WriteLine("Sending SMS: " + message);
    }
}

public class EmailService : IMessageService
{
    public void SendMessage(string message)
    {
        Console.WriteLine("Sending email: " + message);
    }
}
```

See Source Code for Microsoft implementation of IOC

SourceCode ServiceCollection : https://source.dot.net/#Microsoft.Extensions

.DependencyInjection.Abstractions/ServiceCollection.cs

Learn More

dofactory : http://www.dofactory.com/net/InversionOfControl-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-InversionOfControl

File Providers in ASP.NET Core : https://learn.microsoft.com/en-us/aspnet/core/fundamentals/file-providers?view=aspnetcore-8.0

Homework

Implement a simple IoC container that will allow you to register and resolve dependencies. The container should be able to resolve dependencies by type and by name.

Pattern: Iterator

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Iterator

Iterator design pattern allows to traverse a container and access the container's elements.

Purpose of .NET implementation

Any collection should be able to loop through its elements. So the iterator pattern could be retrieved almost anywhere in Array, IEnumerable , Hashset, List, and more.

Example in .NET :

DirectoryEnumerable

```
namespace Iterator;
internal class DirectoryEnumerableDemo
{
    public static void DirectoryEnumerableFiles(int nrMaxFiles)
    {
        var count = 0;
        //what if we called Directory.GetFiles
        foreach (var file in Directory.EnumerateFiles(@"c:\","*.*",SearchOption.AllDirectori
        {
            count++;
            if (count > nrMaxFiles)
                break;
            Console.WriteLine(file);
        }
```

```
    }
}
```

See Source Code for Microsoft implementation of Iterator

SourceCode Directory.GetFiles : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/IO/Directory.cs

SourceCode IEnumerable : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Collections/IEnumerable.cs

SourceCode IEnumerator : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Collections/IEnumerator.cs

Learn More

C2Wiki : https://wiki.c2.com/?IteratorPattern

dofactory : http://www.dofactory.com/net/Iterator-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-Iterator

Wikipedia : https://en.wikipedia.org/wiki/iterator_pattern

Homework

With the Yield keyword implement a function that return an IEnumerable of generic int that will return the first 10 numbers of the Fibonacci sequence.

Pattern: Lazy

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Lazy

Lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

Purpose of .NET implementation

You want to access one object that is difficult to create. But you do not know when will be created. You can use the Lazy pattern to create the object only when it is needed. The Lazy of generic T class provides a way to defer the creation of an object until it is actually needed, allowing you to avoid the cost of creating the object until it is actually required.

Example in .NET :

Lazy

```
namespace Lazy;
internal class LazyDemo
{
    public DateTime dateTimeConstructClass =DateTime.Now;

    public Lazy<DateTime> DateTimeLazy = new(() =>
    {
        Console.WriteLine("Lazy<DateTime> is being initialized ONCE!");
        return DateTime.Now;
    });
}
```

See Source Code for Microsoft implementation of Lazy

SourceCode _defaultLaunchProfile = new Lazy of LaunchProfile : https://github.com/dotnet/project-system/blob/ebc15f3e0fa644bc96b3a7d19b0595bab9d0ab7d/src/Microsoft.VisualStudio.ProjectSystem.Managed/ProjectSystem/Debug/LaunchSettingsProvider.cs#L39

SourceCode Lazy : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Lazy.cs

SourceCode Lazy of IStreamingFindUsagesPresenter : https://github.com/dotnet/roslyn/blob/d89c824648207390f5be355a782048812ba5f91e/src/VisualStudio/Core/Def/Progression/GraphNavigatorExtension.cs#L27

Learn More

C2Wiki : https://wiki.c2.com/?LazyInstantiationPattern

Wikipedia : https://en.wikipedia.org/wiki/Lazy_initialization

Homework

Implement a lazy initialization of a logger that logs to a file and to a console. The logger should be created only when it is needed.

Pattern: NullObject

Read online at https://ignatandrei.github.io/patterns/docs/patterns/NullObject

Instead of returning null , use an object which implements the expected interface, but whose method body is empty.

Purpose of .NET implementation

You want to log data into the application (with an ILogger interface). You do not want to verify if the logger is null or not before logging. You can use the

Null Object pattern to provide a default implementation of the ILogger interface that does nothing when its methods are called.

Examples in .NET :

NullLogger

```
namespace NullObject;
internal class LogWithData
{
    ILogger _logger;
    public LogWithData(ILogger? logger=null)
    {
        // If logger is null, use NullLogger.Instance
        // This is the Null Object pattern in action
        _logger = logger ?? NullLogger.Instance;
    }
    public void DoWork(string message)
    {
        // Even if _logger is NullLogger.Instance, this line won't throw a null reference ex
        // because NullLogger.Instance is a non-functional implementation of ILogger
        _logger.LogInformation($"start work with {message}");
    }
}
```

EmptyFolder

```
using System;
using System.IO;

namespace NullObject;
internal class EmptyFolderDemo
{
    public static void DemoWithCreateNewFolder()
    {
        //start example 1
        var env = Environment.CurrentDirectory;
        var guid = Guid.NewGuid().ToString("X");
        var fldEmpty = Path.Combine(env, guid);
        //create empty folder
        Directory.CreateDirectory(fldEmpty);
        var files= Directory.GetFiles(fldEmpty);
        Console.WriteLine($"files.Length:{files.Length}");
        //end example 1
```

```
    }
}
```

See Source Code for Microsoft implementation of NullObject

SourceCode Directory.GetFiles : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/IO/Directory.cs

Learn More

C2Wiki : https://wiki.c2.com/?NullObject

dofactory : http://www.dofactory.com/net/NullObject-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-NullObject

Wikipedia : https://en.wikipedia.org/wiki/Null_object_pattern

Homework

When retrieving data(e. g. a Person with ID =-1) from a database , return a NullObject instead of null. How you will verify that the object is a NullObject?.

Pattern: Observer

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Observer

Observer pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Purpose of .NET implementation

You want to see when properties of an object are modified, to know how to react in the GUI. You can use the Observer pattern to notify the GUI when the properties of the object are modified.

Example in .NET :

Observer

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Observer;

/// <summary>
/// INotifyPropertyChanged is an interface that provides a mechanism for the object to noti
```

```
/// </summary>
public class Person: INotifyPropertyChanged
{
    private string name=string.Empty;
    public string Name
    {
        get => name;
        set
        {
            if (name == value) return;
            name = value;
            OnPropertyChanged();
        }
    }

    public event PropertyChangedEventHandler? PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
internal class ObserverDemo
{
    public static void Demo()
    {
        Person person = new ();
        //subscribe to the event to observe the changes
        person.PropertyChanged += (sender, args) =>
        {
            var p = sender as Person;
            Console.WriteLine($"Property {args.PropertyName} changed to {p?.Name}");
        };
        person.Name = "Andrei Ignat" ;
    }
}
```

See Source Code for Microsoft implementation of Observer

SourceCode INotifyPropertyChanged : https://source.dot.net/#System.Objec
tModel/System/ComponentModel/INotifyPropertyChanged.cs

Learn More

Wikipedia : https://en.wikipedia.org/wiki/Observer_pattern

Homework

Imagine you have a logger that logs to a file and to a console. Implement an observable logger that will allow you to subscribe to the logger and to be notified when the logger logs a message.

Pattern: Prototype

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Prototype

It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

Purpose of .NET implementation

If you want to clone an object (that has members and methods), the easy way is to copy the members into a new instance. But if you have a complex object, you may want to use the Prototype pattern. You then provide MemberwiseClone is a shallow copy for the members of an object. However, the Prototype pattern is not used very often in .NET, because the ICloneable interface is not very useful.

Example in .NET :

ICloneable

```
using System;

namespace Prototype;
class Parent : ICloneable
{
    public int Age { get; set; }
    public Child MyChild { get; set; }

    public object Clone()
    {
        //TODO: serialize + unserialize
        //with System.Text.Json.JsonSerializer

        var p = ShallowClone();
        //TODO: clone the child
        var c = new Child();
        c.Age = this.MyChild.Age;

        p.MyChild = c;
        return p;
    }
```

```
    public Parent ShallowClone()
    {
        return this.MemberwiseClone() as Parent;
    }
}
```

See Source Code for Microsoft implementation of Prototype

SourceCode Object.MemberwiseClone : https://source.dot.net/#System.Private.CoreLib/src/System/Object.CoreCLR.cs

Learn More

C2Wiki : https://wiki.c2.com/?PrototypePattern

dofactory : http://www.dofactory.com/net/Prototype-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-Prototype

Wikipedia : https://en.wikipedia.org/wiki/Prototype_pattern

Homework

Imagine that you have a cow farm and you want to create a new cow. Implement a prototype that will allow you to clone a cow. The cow should have a name and a weight.

Pattern: Singleton

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Singleton

Singleton pattern restricts the instantiation of a class to one object.It is used when you want to have one instance of a class that is shared across the application.

Purpose of .NET implementation

You have a factory to create objects. You want to have a single point of creation for all the objects. You can use the Singleton pattern to ensure that only one instance of the factory is created and that all requests for object creation are handled by that single instance.

Example in .NET :

Singleton

```
using System.Data.OleDb;
```

```
namespace Singleton;
/// <summary>
///
///sealed class Singleton
///{
///    private Singleton() { }
///    public static readonly Singleton Instance = new Singleton();
///}
///
/// </summary>
internal class SingletonDemo
{
    public static void GetFactory()
    {
        //cannot do new
        //OleDbFactory factory=new OleDbFactory();
        //get singleton instance
        OleDbFactory factory = OleDbFactory.Instance;
    }
}
```

See Source Code for Microsoft implementation of Singleton

SourceCode OleDbFactory.Instance : https://referencesource.microsoft.com/#System.Data/fx/src/data/System/Data/OleDb/OleDbFactory.cs

Learn More

C2Wiki : https://wiki.c2.com/?SingletonPattern

dofactory : http://www.dofactory.com/net/singleton-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-singleton

Wikipedia : https://en.wikipedia.org/wiki/Singleton_pattern

Homework

Implement a singleton that will allow you to create a single instance of a logger that logs to a file and to a console.

Pattern: Strategy

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Strategy

Strategy pattern allows a client to choose from a family of algorithms at run-time.It is used when the client expects to have multiple algorithms and wants to choose one of them at runtime.

Purpose of .NET implementation

You want to have a function that sort objects based on a specific criteria. You want to let the developer to provide the sort criteria. You want also allow the sorting behavior (the strategy) to be selected at runtime. You can use the Strategy pattern to let developer define the sorting criteria and the strategy to be used at runtime.

Example in .NET :

Strategy

```
using System;
using System.Collections.Generic;

namespace Strategy;
internal class StrategyDemo
{
    public static void SortWithDifferentStrategies()
    {
        List<int> al = new ();
        al.Add(102);
        al.Add(201);
        // Strategy 1: Sorts the list in ascending order.
        al.Sort((x, y) => x.CompareTo(y));

        for (int i = 0; i < al.Count; i++)
        {
            Console.WriteLine(al[i]);
        }

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

        // Strategy 2: Sorts the list in descending order.
        al.Sort((y, x) => x.CompareTo(y));
        for (int i = 0; i < al.Count; i++)
        {
            Console.WriteLine(al[i]);
        }
        Console.WriteLine("---------------");
        // Strategy 3: Sorts the list based on the last digit of each number.
        al.Sort((x, y) => LastDigit(x).CompareTo(LastDigit(y)));
        for (int i = 0; i < al.Count; i++)
        {
            Console.WriteLine(al[i]);
        }
```

```
        var array = al.FindAll(it => it > 10);


    }

    static int LastDigit(int x)
    {
        return x % 10;
    }

}
```

See Source Code for Microsoft implementation of Strategy

SourceCode Array.Sort : https://source.dot.net/#System.Private.CoreLib/src/libraries/System.Private.CoreLib/src/System/Array.cs

Learn More

C2Wiki : https://wiki.c2.com/?StrategyPattern

dofactory : http://www.dofactory.com/net/strategy-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-strategy

Wikipedia : https://en.wikipedia.org/wiki/Strategy_pattern

Homework

Image you want to serialize classes to XML,JSON and CSV. Implement a strategy that will allow you to choose between XML , JSON and CSV serialization at runtime.

Pattern: Visitor

Read online at https://ignatandrei.github.io/patterns/docs/patterns/Visitor

Visitor pattern is a way of separating an algorithm from an object structure on which it operates.A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures.

Purpose of .NET implementation

Roslyn is a syntax analyzer for C#. You want to provide a way for other to analyze/modify the syntax of a C# code. You can use the Visitor pattern to traverse the syntax tree and perform operations on the nodes of the tree. The Visitor pattern allows you to separate the algorithm from the data structure,

making it easier to add new operations to the syntax tree without modifying the existing classes.

Example in .NET :

Visitor

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace Visitor;
internal class VisitorDemo
{
    public static void VisitMethods()
    {
        var Code = @"""
using System;
namespace Test1
{
    class Program
    {
        static void Main(string[] args)
        {
              var dt=DateTime.Now;
              Console.WriteLine(dt);
        }
     }
}

""";
        var tree = CSharpSyntaxTree.ParseText(Code);

        var node = tree.GetRoot();

        MethodVisiting LG = new MethodVisiting();
        //start visiting
        var sn = LG.Visit(node);

    }
}
public class MethodVisiting : CSharpSyntaxRewriter
{
    public override SyntaxNode? VisitMethodDeclaration(MethodDeclarationSyntax node)
    {
        if (node.Body == null || node.Body.Statements.Count == 0)
```

```
            return base.VisitMethodDeclaration(node);

        var parent = node.Parent as ClassDeclarationSyntax;

        if (parent == null)
            return base.VisitMethodDeclaration(node);

        var nameMethod = node.Identifier.Text;
        var nameClass = parent.Identifier.Text;
        Console.WriteLine($"visiting {nameMethod} from {nameClass}");

        return base.VisitMethodDeclaration(node);

    }
}
```

See Source Code for Microsoft implementation of Visitor

SourceCode CSharpSyntaxRewriter : https://github.com/dotnet/roslyn/blob/cecdb802007a49e346215c0afdce354d6c111112/src/Compilers/CSharp/Portable/Syntax/CSharpSyntaxRewriter.cs#L17

Learn More

C2Wiki : http://wiki.c2.com/?VisitorPattern

C2Wiki : https://wiki.c2.com/?VisitorPattern

dofactory : http://www.dofactory.com/net/visitor-design-pattern

DPH : https://github.com/kamranahmedse/design-patterns-for-humans?tab=readme-ov-file#-visitor

Wikipedia : https://en.wikipedia.org/wiki/Visitor_pattern

Homework

Implement a visitor that will allow you to calculate the total price of a shopping cart. The shopping cart should contain items with a price and a quantity. Visit every item and make the sum.