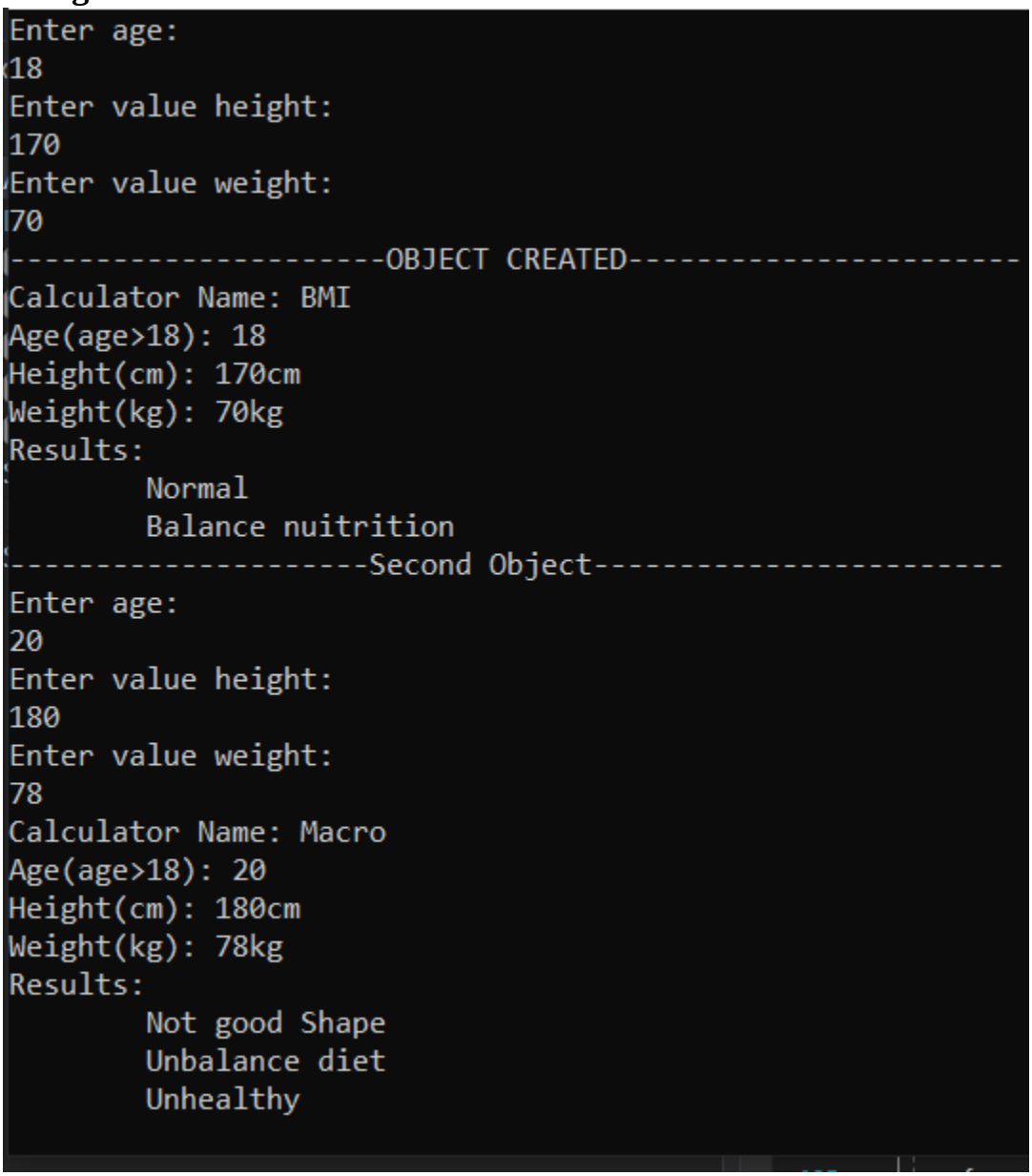
1. **Builder Pattern**

Builder pattern is an in the Creational pattern. Builder pattern is the sample design object is being create to build the double object by how to use the simple object and using the next to step, build up the object to set up with other object.

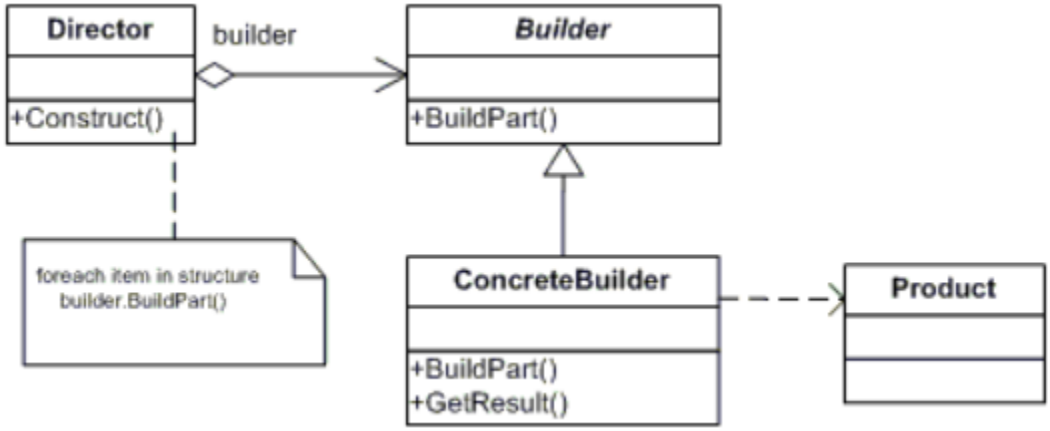
Using for BMI and Macro Calculator



```
static void Main(string[] args)
{
    var CalculatorCreator = new CalculatorCreator(new BMIBuilder());
    CalculatorCreator.CreateCalculator();
    var Calculator = CalculatorCreator.GetCalculator();
    Console.WriteLine("----------------------OBJECT CREATED-----------------------");
    Calculator.ShowInfo();
    Console.WriteLine("----------------------Second Object-----------------------");
    CalculatorCreator = new CalculatorCreator(new MacroBuilder());
    CalculatorCreator.CreateCalculator();Calculator = CalculatorCreator.GetCalculator();
    Calculator.ShowInfo();
    Console.ReadKey();
}
```

Reason: Clear code and easy to understand. Save time and better control in program because this pattern does not depend on initialization variable when both my calculator has the same structure(name/age/height/weight/results). Therefore, choosing this pattern help us to build our calculator.

UML Diagram



Classes and objects that participate in this pattern include:

- **Builder (CalculatorBuilder)** implements an abstract interface by creating a part of the Product object

```
public interface ICalCulateBuilder
{
    void SetModel();
    void SetAge(string age);
    void SetValueB(string valueB);
    void SetValueA(string valueA);
    void SetResults(int tage, int tvalueA, int tvaluaB);

    Calculator GetCalculator();
}
```

- **ConcreteBuilder (BMIBuilder, MacroBuilder)**

- structuring and concatenating parts of a product by implementing the Builder interface
- redefine and document the details it creates
- Provide an interface that can return the details of the generated product

```
public class BMIBuilder : ICalCulateBuilder
{
    Calculator objCalculator = new Calculator();
    public void SetModel()
    {
        objCalculator.Model = "BMI";
    }

    public void SetAge(string age)
    {
        objCalculator.Age = age;
    }

    public void SetValueB(string valueB)
    {
        objCalculator.Weight = valueB + "kg";
    }

    public void SetValueA(string valueA)
    {
        objCalculator.Height = valueA + "cm";
    }

    public void SetResults(int age, int valueA, int valueB)
    {
```

```
/// The 'ConcreteBuilder2' class
/// </summary>
public class MacroBuilder : ICalCulateBuilder
{
    Calculator objCalculator = new Calculator();
    public void SetModel()
    {
        objCalculator.Model = "Macro";
    }

    public void SetAge(string age)
    {
        objCalculator.Age = age.ToString();
    }

    public void SetValueB(string valueB)
    {
        objCalculator.Weight = valueB + "kg";
    }

    public void SetValueA(string valueA)
    {
        objCalculator.Height = valueA +"cm";
    }

    public void SetResults(int age,int valueA, int valueB)
```

- **Director (Calculator)** creates object using Builder interface

```
/// The 'Director' class
/// </summary>
public class CalculatorCreator
{
    private readonly ICalCulateBuilder objBuilder;

    public CalculatorCreator(ICalCulateBuilder builder)
    {
        objBuilder = builder;
    }

    public void CreateCalculator()
    {
        Console.WriteLine("Enter age:");
        string age = Console.ReadLine();
        Console.WriteLine("Enter value height:");
        string valueA = Console.ReadLine();
        Console.WriteLine("Enter value weight:");
        string valueB = Console.ReadLine();
        int tage=0;

        int tvalueA=1;
        int tvalueB=0;
        if (age != "" || valueA != "" || valueB != "")
        {
            tage = Convert.ToInt32(age);

            tvalueA = Convert.ToInt32(valueA) / 100;

            tvalueB = Convert.ToInt32(valueB);
        }
        objBuilder.SetModel();
        objBuilder.SetAge(age);
        objBuilder.SetValueA(valueA);
        objBuilder.SetValueB(valueB);
        objBuilder.SetResults(tage,tvalueA,tvalueB);
    }

    public Calculator GetCalculator()
    {
        return objBuilder.GetCalculator();
```

- **Product (Calculator)** is the complex object created. ConcreteBuilder builds the product details internally and defines the concatenation handling, including classes that define the details, and the interfaces for piecing together the parts that produce the final result.

```
/// The 'Product' class
/// </summary>
public class Calculator
{
    public string Model { get; set; }
    public string Age { get; set; }
    public string Weight { get; set; }
    public string Height { get; set; }
    public List<string> Results { get; set; }

    public Calculator()
    {
        Results = new List<string>();
    }

    public void ShowInfo()
    {
        Console.WriteLine("Calculator Name: {0}", Model);
        Console.WriteLine("Age(age>18): {0}", Age);
        Console.WriteLine("Height(cm): {0}", Height);
        Console.WriteLine("Weight(kg): {0}", Weight);
        Console.WriteLine("Results:");
        foreach (var accessory in Results)
        {
            Console.WriteLine("\t{0}", accessory);
        }
    }
}
```

2. Abstract Pattern

Abstract Factory pattern is like a super factory, to create other factories (to create objects). This super machine is also known as the factory of factories. This design pattern is a type of creational pattern, and is one of the best and most effective ways to create objects. In the Abstract Factory pattern, an interface is responsible for creating a factory for similar objects without specifying their class. Each generated factory can expose different factory objects (Factory Pattern). Frequency of use: Very high

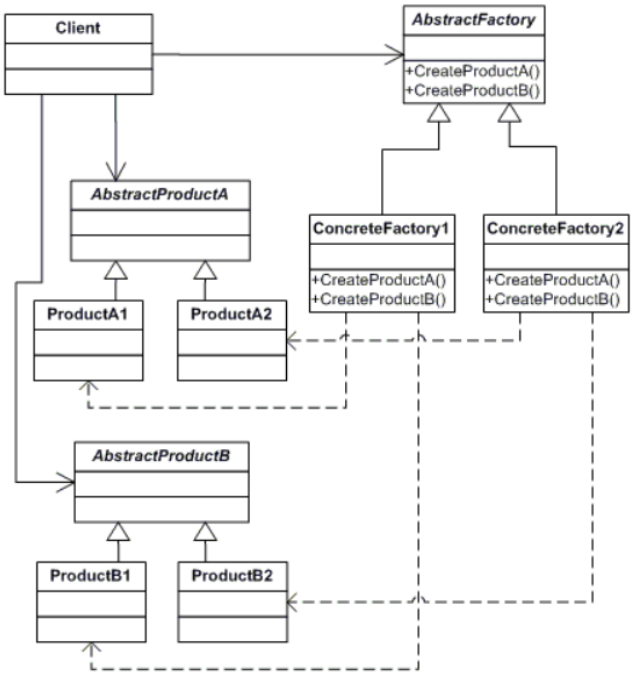
Using Abstract Pattern for My Favorite Exercise

```
C:\Users\DELL\Desktop\ConsoleApp2\ConsoleApp2\bin\Debug\netcoreapp3.1\ConsoleApp2.exe
User1 love the Excercise1
User2 love the Excercise2
```

```
public static void Main()
{
    // Create and run the Excerise

    ContinentFactory Excecise = new ExceciseFactory();
    UsersWorld world = new UsersWorld(Excecise);
    world.RunFavorExcerciseChain();

    // Create and run the UsersWorld

    ContinentFactory Users = new UsersFactory();
    world = new UsersWorld(Users);
    world.RunFavorExcerciseChain();

    // Wait for user input

    Console.ReadKey();
}
```

Reason: With the Abstract Pattern, creating objects with related and unique properties promotes object-oriented abstraction, while the code looks clean and scientific. Our function has objects which have the same structure with some unique characteristic so abstract pattern is appropriate for this case.

UML Diagram



The components

- **AbstractFactory (ContinentFactory - Continent factory)** declares an interface for functions to create abstract products

```
abstract class ContinentFactory

{
    public abstract Excercise CreateFavoriteExcercise();
    public abstract User CreateUsers();
}
```

- **ConcreteFactory (ExcerciseFactory, UsersFactory)** implements function from interface to create specific product objects

```
/// <summary>

/// The 'ConcreteFactory1' class

/// </summary>

class ExceciseFactory : ContinentFactory

{
    public override Excercise CreateFavoriteExcercise()
    {
        return new Excercise1();
    }
    public override User CreateUsers()
    {
        return new User1();
    }
}
```

```
/// <summary>

/// The 'ConcreteFactory2' class

/// </summary>

class UsersFactory : ContinentFactory

{
    public override Excercise CreateFavoriteExcercise()
    {
        return new Excercise2();
    }
    public override User CreateUsers()
    {
        return new User2();
    }
}
```

- **AbstractProduct (User, Excercise)** declares an interface for the product type

```
/// <summary>

/// The 'AbstractProductA' abstract class

/// </summary>

abstract class Excercise

{
}

/// <summary>

/// The 'AbstractProductB' abstract class

/// </summary>

abstract class User

{
    public abstract void FavoriteExcecise(Excercise h);
}
```

- **Product (User1, User2, Excercise1, Excercise2)**
    - Shows product objects created by ConcreteFactory
    - implement the Abstract Product interface

```
/// <summary>

/// The 'ProductA1' class

/// </summary>

class Excercise1 : Excercise

{
}

/// <summary>

/// The 'ProductB1' class

/// </summary>

class User1 : User

{
    public override void FavoriteExcecise(Excercise h)
    {
        // FavoriteExcecise Excercise1

        Console.WriteLine(this.GetType().Name +
          " love the " + h.GetType().Name);
    }
}
```

```
/// <summary>

/// The 'ProductA2' class

/// </summary>

class Excercise2 : Excercise

{
}

/// <summary>

/// The 'ProductB2' class

/// </summary>

class User2 : User

{
    public override void FavoriteExcecise(Excercise h)
    {
        // FavoriteExcecise Excercise2

        Console.WriteLine(this.GetType().Name +
          " love the " + h.GetType().Name);
    }
}
```

- **Client (UsersWorld)** uses interfaces declared by classes AbstarctFactory and AbstractProduct

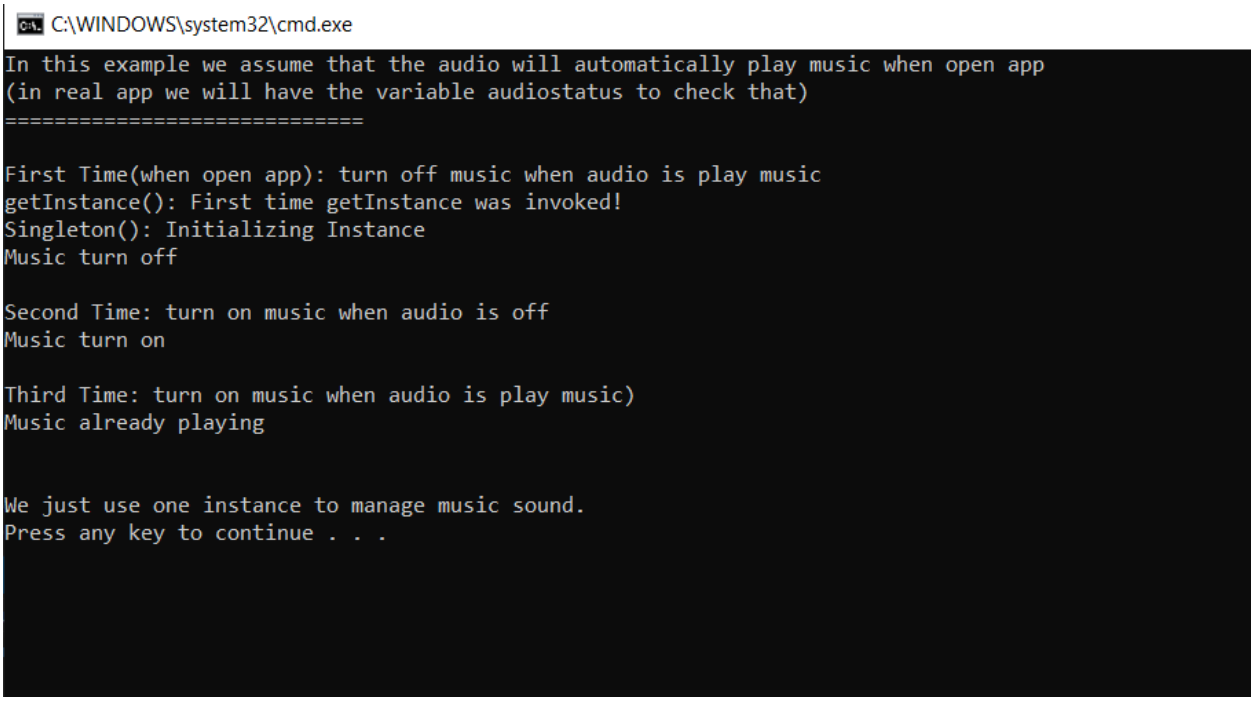
```
/// <summary>

/// The 'Client' class

/// </summary>

class UsersWorld

{
    private Excercise _Excercise;
    private User _User;

    // Constructor

    public UsersWorld(ContinentFactory factory)
    {
        _User = factory.CreateUsers();
        _Excercise = factory.CreateFavoriteExcercise();
    }

    public void RunFavorExcerciseChain()
    {
        _User.FavoriteExcecise(_Excercise);
    }
}
```

3. Singleton Pattern

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

Reason: The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to make sure there is no more than one instance; it does it by instantiating itself and in the same time it provides a global point of access to that instance. By doing it, the singleton class ensures the same instance can be used from everywhere, preventing direct invocation of the singleton constructor. In our project, we have function play music so that we can use singleton to responsible one instance on and off music.

Using Singleton Pattern for Sound and Music



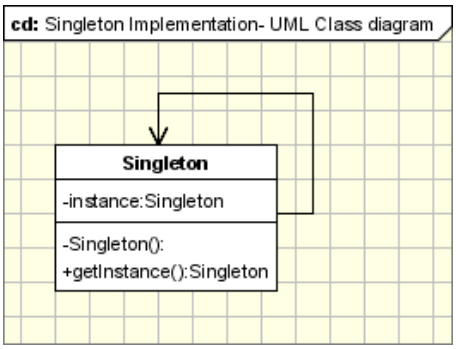

```
static void Main(string[] args)
{
    Console.WriteLine("In this example we assume that the audio will automatically play music when open app " +
        "\n(in real app we will have the variable audiostatus to check that)" +
        "\n==========\n");
    Console.WriteLine("First Time(when open app): turn off music when audio is play music");
    bool audiostatus = true;
    MusicManager.getInstance().PlayMusic(false, audiostatus);
    Console.WriteLine("\nSecond Time: turn on music when audio is off");
    audiostatus = false;
    MusicManager.getInstance().PlayMusic(true, audiostatus);
    Console.WriteLine("\nThird Time: turn on music when audio is play music)");
    audiostatus = true;
    MusicManager.getInstance().PlayMusic(true, audiostatus);
    Console.WriteLine("\n\nWe just use one instance to manage music sound.");
}
```

```
public class MusicManager
{
    private static MusicManager instance;

    //private bool audioSource; //=true: isplaying, =false:is not playing (audioSource.isPlaying in Unity)
    1 reference
    private MusicManager()
    {
        Console.WriteLine("Singleton(): Initializing Instance");
    }

    3 references
    public static MusicManager getInstance()
    {
        if (instance == null)
        {
            Console.WriteLine("getInstance(): First time getInstance was invoked!");
            instance = new MusicManager();
        }
        return instance;
    }

    3 references
    public void PlayMusic(bool play, bool audioSource)
    {
        if (play == true)
        {
```

```
        3 references
        public void PlayMusic(bool play, bool audioSource)
        {
            if (play == true)
            {
                if (audioSource == false)// audio is not playing
                {
                    //audioSource.Play();
                    Console.WriteLine("Music turn on ");
                }
                else
                {
                    Console.WriteLine("Music already playing ");
                }
            }
            else
            {
                if (audioSource == true) // audio is playing
                {
                    //audioSource.Stop();
                    Console.WriteLine("Music turn off");
                }
                else
                {
                    Console.WriteLine("Music already off ");
                }
            }
        }
    }
```

Reason: To manage music, we only need one instance because when an instance is on, only that instance is allowed to turn off, not another instances.

UML Diagram



The implementation involves a static member in the **Singleton** class which keeps the reference to the instance, a private constructor and a static public method that returns the static member reference.

The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is responsible for creating its class unique instance in case it is not created yet and to return that instance.

4. Observer

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object.

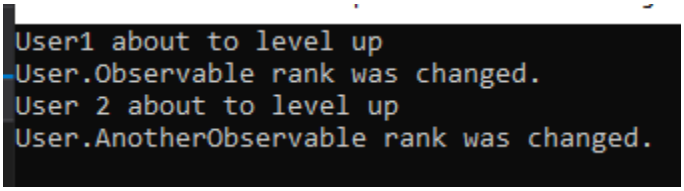
Using Observer Pattern for LeaderBoard

```
internal static class Program
{
    private static void Main()
    {
        var observable = new Observable();
        var anotherObservable = new AnotherObservable();
        using (IObserver observer = new Observer(observable))
        {
```

```
            observable.DoSomething();
            observer.Add(anotherObservable);
            anotherObservable.DoSomething();
        }

        Console.ReadLine();
    }
}

internal interface IObservable
{
    event EventHandler SomethingHappened;
}

internal sealed class Observable : IObservable
{
    public event EventHandler SomethingHappened;
    public void DoSomething()
    {
        var handler = this.SomethingHappened;

        Console.WriteLine("User1 about to level up");
        if (handler != null)
        {
            handler(this, EventArgs.Empty);
        }
    }
}

internal sealed class AnotherObservable : IObservable
{
    public event EventHandler SomethingHappened;

    public void DoSomething()
    {
        var handler = this.SomethingHappened;

        Console.WriteLine("User 2 about to level up");
        if (handler != null)
        {
```

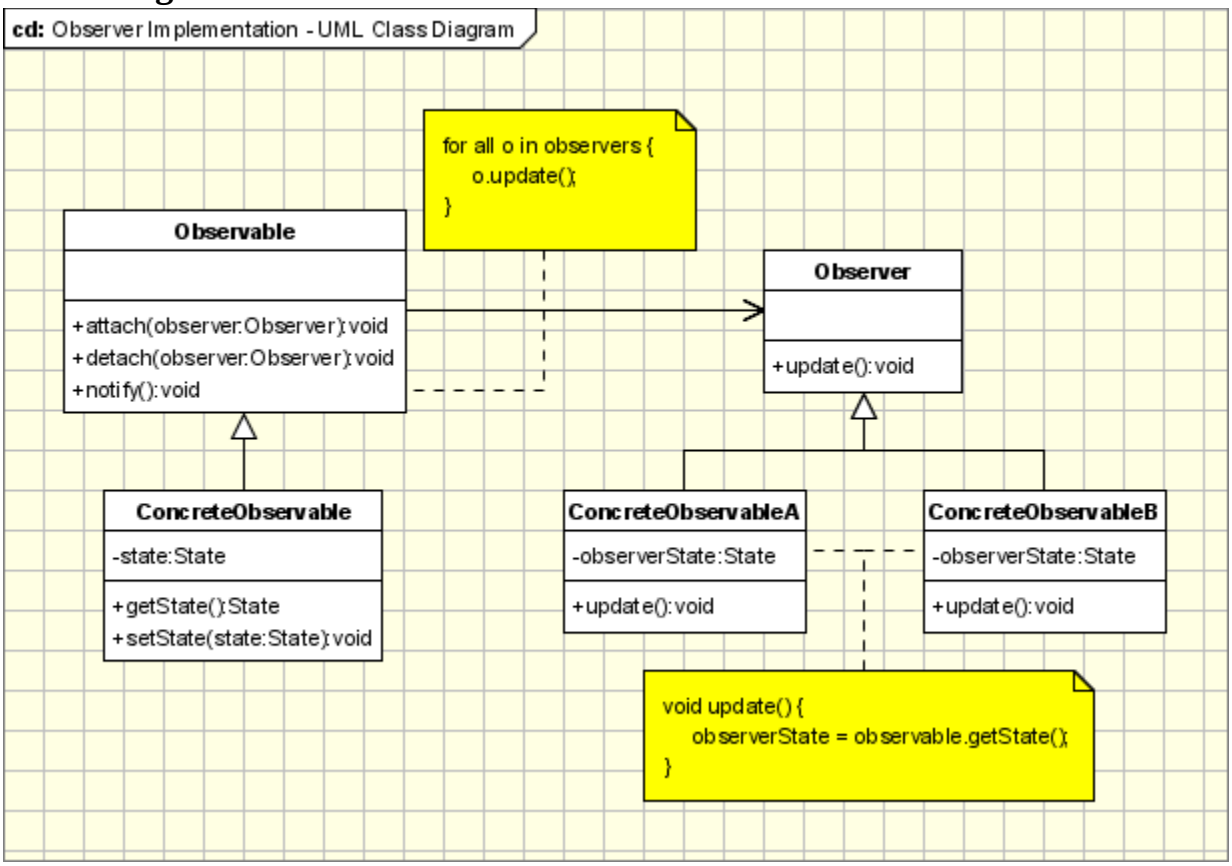
```
            handler(this, EventArgs.Empty);
        }
    }
}

internal interface IObserver : IDisposable
{
    void Add(IObservable observable);
    void Remove(IObservable observable);
}

internal sealed class Observer : IObserver
{
    private readonly Lazy<IList<IObservable>> observables =
        new Lazy<IList<IObservable>>(() => new List<IObservable>());

    public Observer()
    {
    }

    public Observer(IObservable observable) : this()
    {
        this.Add(observable);
    }

    public void Add(IObservable observable)
    {
        if (observable == null)
        {
            return;
        }
        lock (this.observables)
        {
            this.observables.Value.Add(observable);
            observable.SomethingHappened += HandleEvent;
        }
    }

    public void Remove(IObservable observable)
    {
```

```
        if (observable == null)
        {
            return;
        }
        lock (this.observables)
        {
            observable.SomethingHappened -= HandleEvent;
            this.observables.Value.Remove(observable);
        }
    }

    public void Dispose()
    {
        for (var i = this.observables.Value.Count - 1; i >= 0; i--)
        {
            this.Remove(this.observables.Value[i]);
        }
    }

    private static void HandleEvent(object sender, EventArgs args)
    {
        Console.WriteLine(sender + " rank was changed.");
    }
}
```



Reason: We have created an abstract class Observer and a concrete class Subject that is extending class Observer and implement it to the LeaderBoard function in our project, this will allow an auto-update to the user rank position in the board.

UML Diagram



5. Visitor

Visitor design pattern is one of the behavioral design patterns. It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

The visitor pattern consists of two parts:

- a method called Visit() which is implemented by the visitor and is called for every element in the data structure
- visitable classes providing Accept() methods that accept a visitor
- Client : The Client class is a consumer of the classes of the visitor design pattern. It has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.
- Visitor : This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.

- ConcreteVisitor : For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.
- Visitable : This is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.
- ConcreteVisitable : These classes implement the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

Using Visitor Pattern for Friendlist

```
namespace User
{
   namespace VisitorDesignPattern
   {
      public interface IElement
      {
         void Accept(IVisitor visitor);
      }
   }
   namespace VisitorDesignPattern
   {
      public class FriendAccout : IElement
      {
         public string FriendAccoutName { get; set; }

         public FriendAccout(string name)
         {
            FriendAccoutName = name;
         }

         public void Accept(IVisitor visitor)
         {
            visitor.Visit(this);
         }
      }
   }
   namespace VisitorDesignPattern
   {
```

```
    public interface IVisitor
    {
        void Visit(IElement element);
    }
}

namespace VisitorDesignPattern
{
    public class User : IVisitor
    {
        public string Name { get; set; }
        public User(string name)
        {
            Name = name;
        }

        public void Visit(IElement element)
        {
            FriendAccout FriendAccout = (FriendAccout)element;
            Console.WriteLine("User: " + this.Name + " did a check up on " +
FriendAccout.FriendAccoutName);
        }
    }
}

namespace VisitorDesignPattern
{
    class MainUser : IVisitor
    {
        public string Name { get; set; }
        public MainUser(string name)
        {
            Name = name;
        }
        public void Visit(IElement element)
        {
            FriendAccout FriendAccout = (FriendAccout)element;
            Console.WriteLine("MainUser: " + this.Name + " like something on "
                    + FriendAccout.FriendAccoutName + " account page");
        }
```

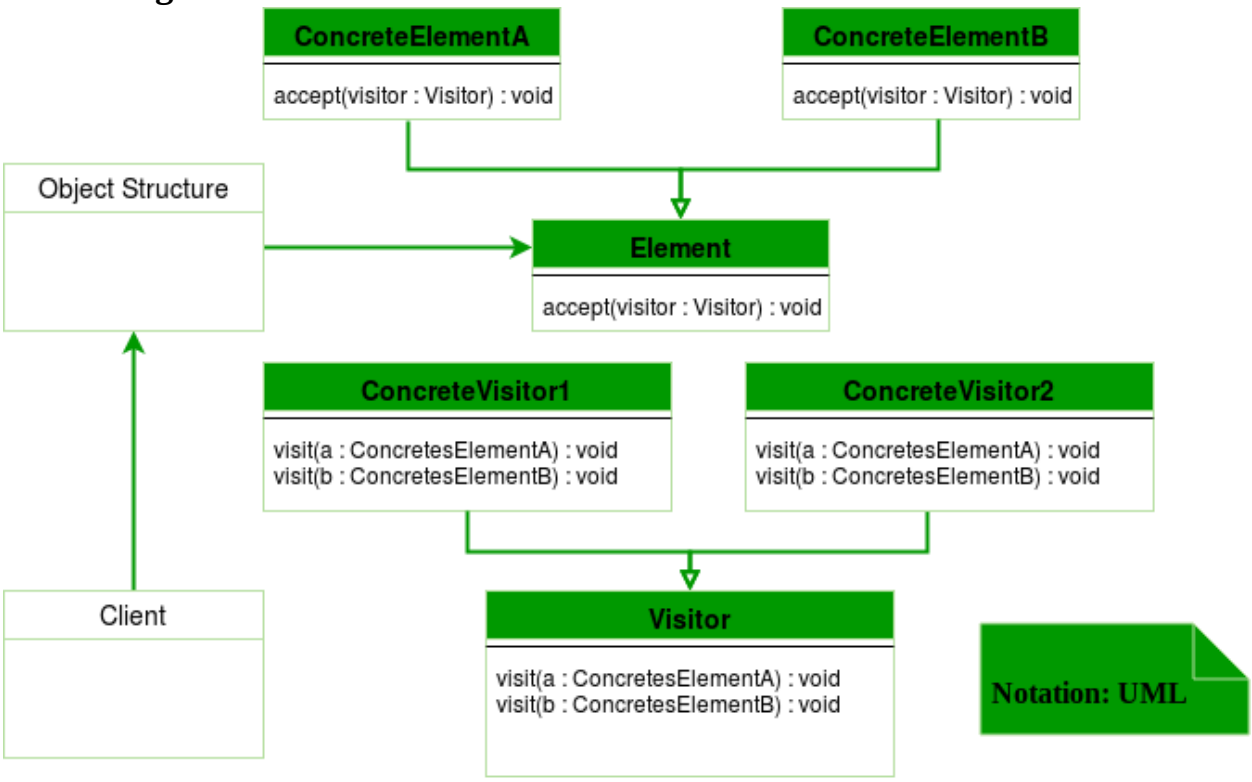
```
    }
}

namespace VisitorDesignPattern
{
    public class Friendlist
    {
        private static List<IElement> elements;
        static Friendlist()
        {
            elements = new List<IElement>
        {
            new FriendAccout("Friend2"),
            new FriendAccout("Friend3"),
            new FriendAccout("Friend4")
        };
        }
        public void PerformOperation(IVisitor visitor)
        {
            foreach (var FriendAccout in elements)
            {
                FriendAccout.Accept(visitor);
            }
        }
    }
}

namespace VisitorDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Friendlist Friendlist = new Friendlist();
            var visitor1 = new User("User 01");
            Friendlist.PerformOperation(visitor1);
            Console.WriteLine();
            var visitor2 = new MainUser("John");
            Friendlist.PerformOperation(visitor2);
            Console.Read();
```

```
        }
    }
}
```

```
User: User 01 did a check up on Friend2
User: User 01 did a check up on Friend3
User: User 01 did a check up on Friend4

MainUser: John like something on Friend2 account page
MainUser: John like something on Friend3 account page
MainUser: John like something on Friend4 account page
```

Reason: In the project, we will implement Visitor pattern to the Friendlist option, this will allow a user to check up on a group of accounts and perform some action on these accounts.

UML Diagram



Contents

1. Builder Pattern 1

Using for BMI and Macro Calculator 1

Reason 2

UML Diagram 2
2. Abstract Pattern 6
Using Abstract Pattern for My Favorite Exercise 6
Reason 7
UML Diagram 7
3. Singleton Pattern 12
Reason 12
Using Singleton Pattern for Sound and Music 13
Reason 14
UML Diagram 15
4. Observer 15
Using Observer Pattern for LeaderBoard 15
Reason 18
UML Diagram 19
5. Visitor 19
Using Visitor Pattern for Friendlist 20
Reason 23
UML Diagram 23