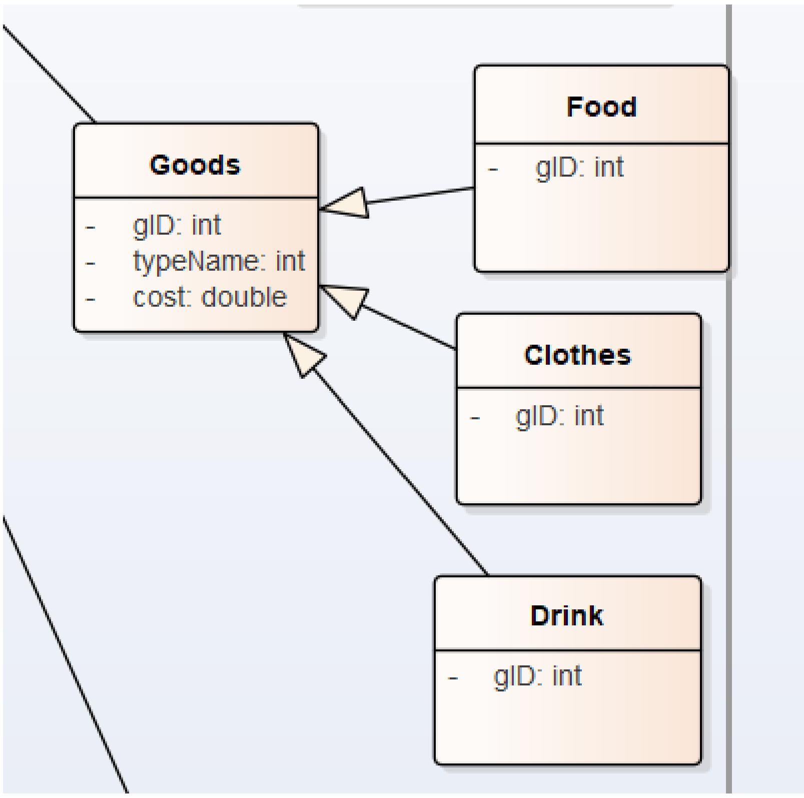
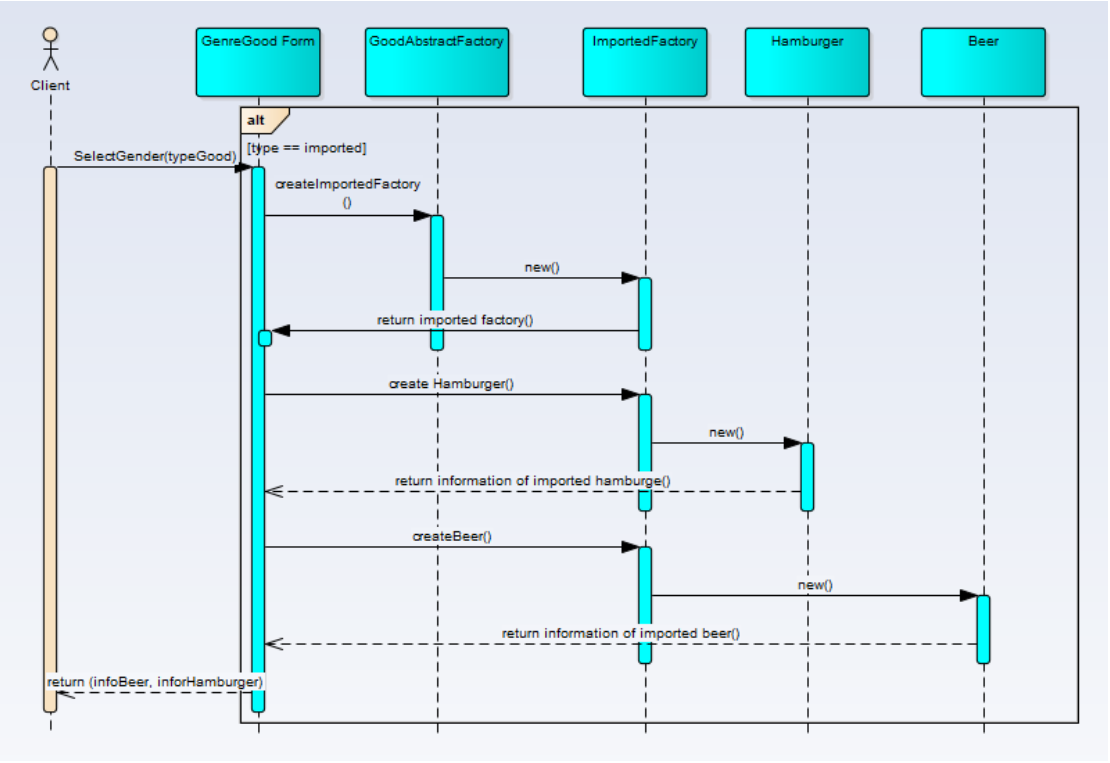
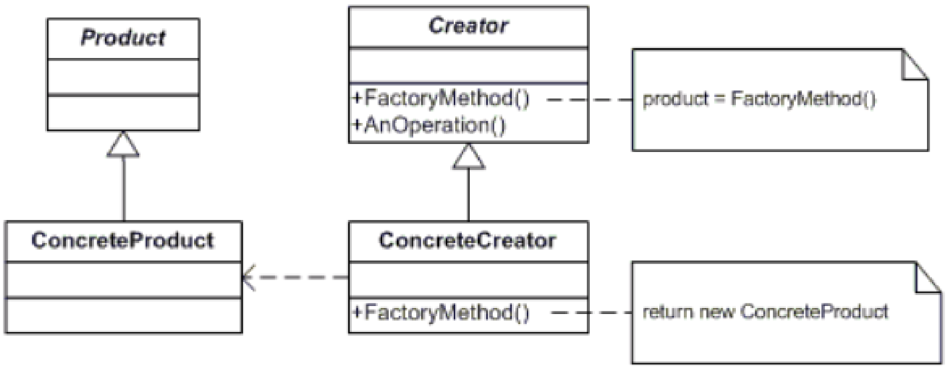
Object-oriented software design (design pattern)

1. Factory

Name of function: Get Goods




Client
GenreGood Form
GoodAbstractFactory
ImportedFactory
Hamburger
Beer
alt
[type == imported]
SelectGender(typeGood)
createImportedFactory
()
new()
return imported factory()
create Hamburger()
new()
return information of imported hamburge()
createBeer()
new()
return information of imported beer()
return (infoBeer, inforHamburger)


Product
Creator
+FactoryMethod()
+AnOperation()
product = FactoryMethod()
ConcreteProduct
ConcreteCreator
+FactoryMethod()
return new ConcreteProduct

ConcreteProduct

This is a class that implements the Product interface.

Product

This defines the interface of objects the factory method creates.

Creator

This is an abstract class and declares the factory method, which returns an object of type Product.

This may also define a default implementation of the factory method that returns a default ConcreteProduct object.
This may call the factory method to create a Product object.

ConcreteCreator

This is a class that implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct.

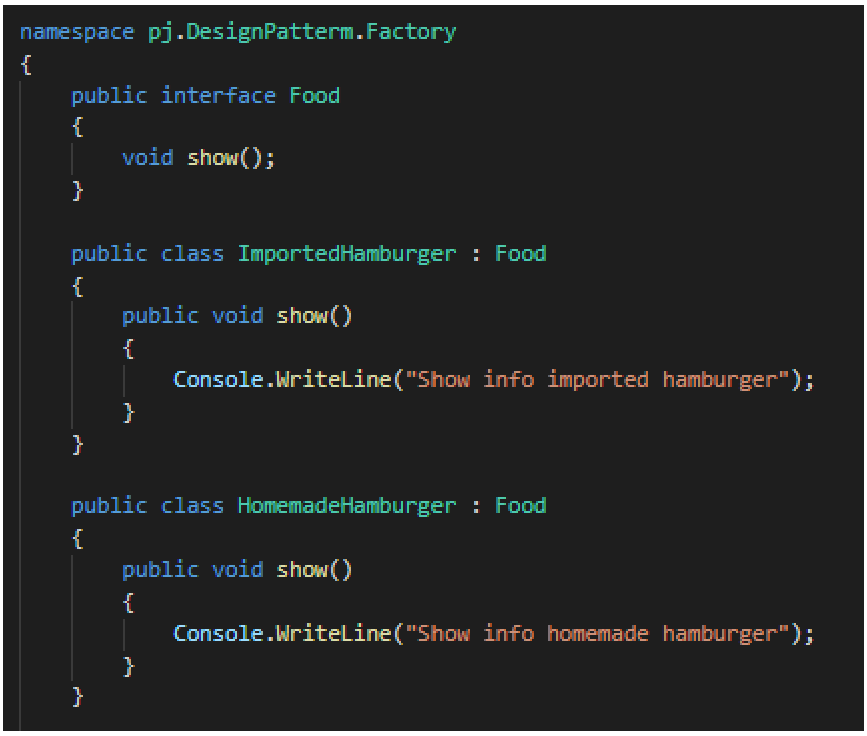
Reason:

Because it is not known exactly what order the user will order, each time a search is made, it is made available so that the search will appear

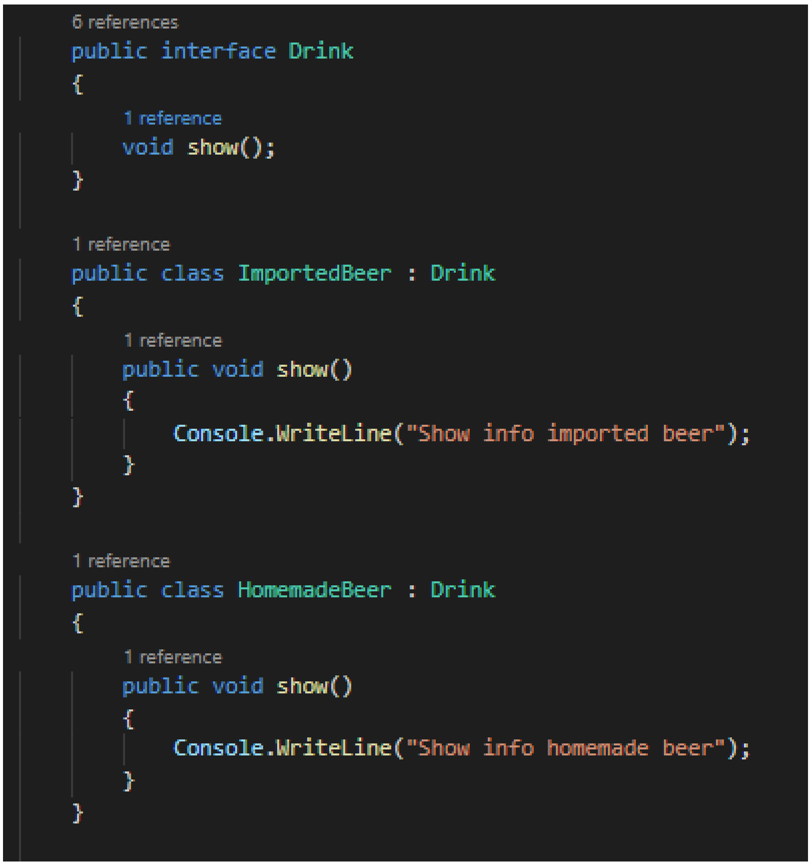
Characteristic:

- Creates objects without exposing the instantiation logic to the client.
- Refers to the newly created object through a common interface

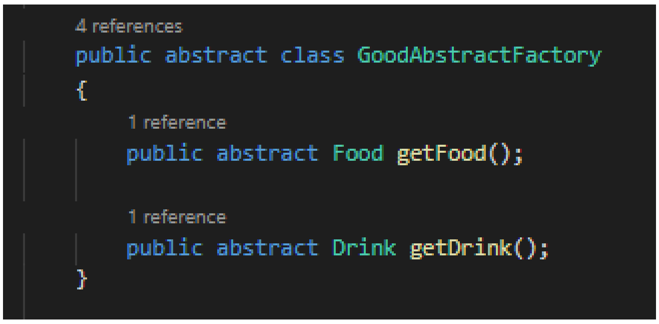
C# Codes:



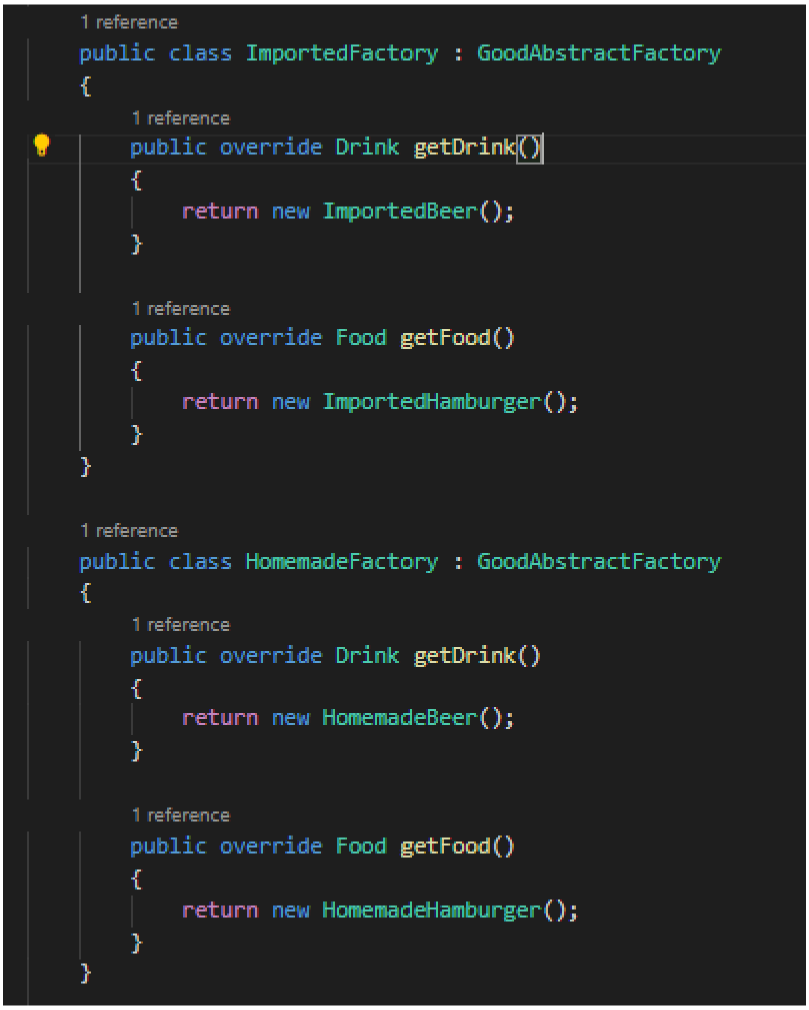
```
namespace pj.DesignPatterm.Factory
{
    public interface Food
    {
        void show();
    }

    public class ImportedHamburger : Food
    {
        public void show()
        {
            Console.WriteLine("Show info imported hamburger");
        }
    }

    public class HomemadeHamburger : Food
    {
        public void show()
        {
            Console.WriteLine("Show info homemade hamburger");
        }
    }
```



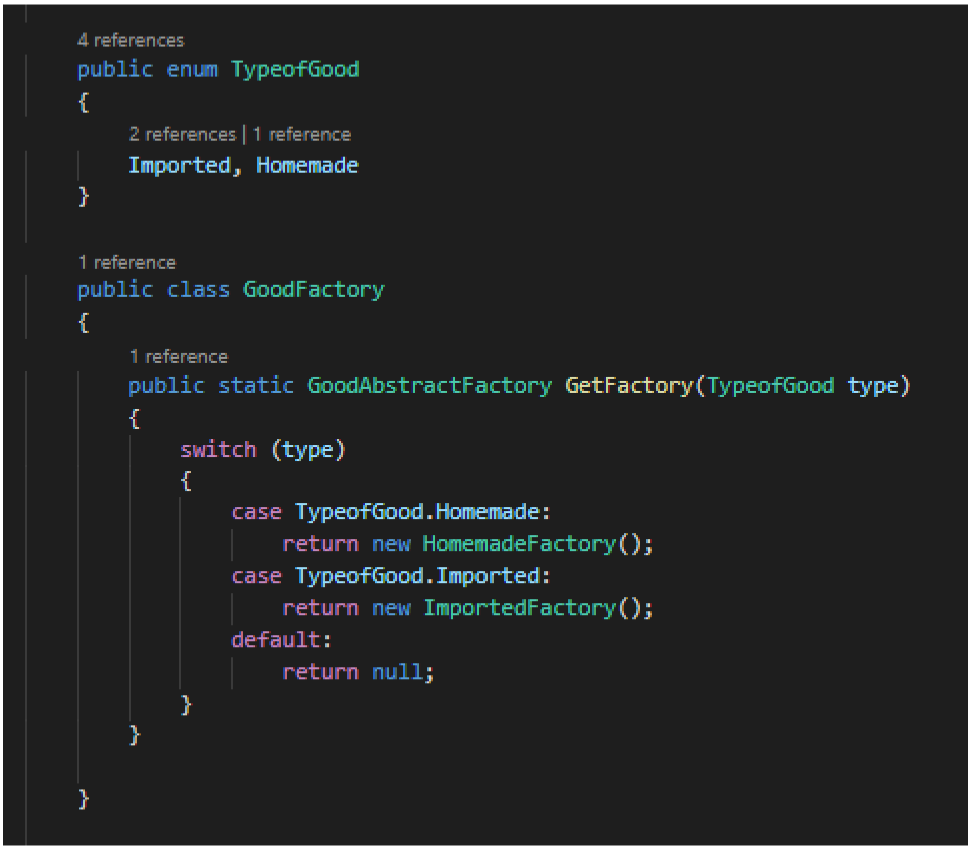
```
    6 references
    public interface Drink
    {
        1 reference
        void show();
    }

    1 reference
    public class ImportedBeer : Drink
    {
        1 reference
        public void show()
        {
            Console.WriteLine("Show info imported beer");
        }
    }

    1 reference
    public class HomemadeBeer : Drink
    {
        1 reference
        public void show()
        {
            Console.WriteLine("Show info homemade beer");
        }
    }
```



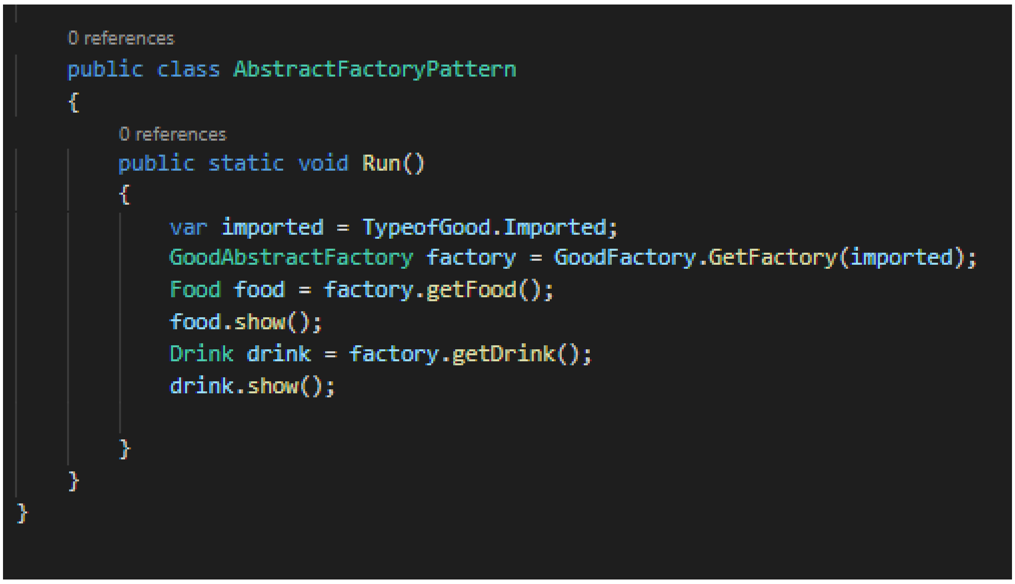
```
4 references
public abstract class GoodAbstractFactory
{
    1 reference
    public abstract Food getFood();

    1 reference
    public abstract Drink getDrink();
}
```



```
1 reference
public class ImportedFactory : GoodAbstractFactory
{
    1 reference
    public override Drink getDrink()
    {
        return new ImportedBeer();
    }

    1 reference
    public override Food getFood()
    {
        return new ImportedHamburger();
    }
}

1 reference
public class HomemadeFactory : GoodAbstractFactory
{
    1 reference
    public override Drink getDrink()
    {
        return new HomemadeBeer();
    }

    1 reference
    public override Food getFood()
    {
        return new HomemadeHamburger();
    }
}
```



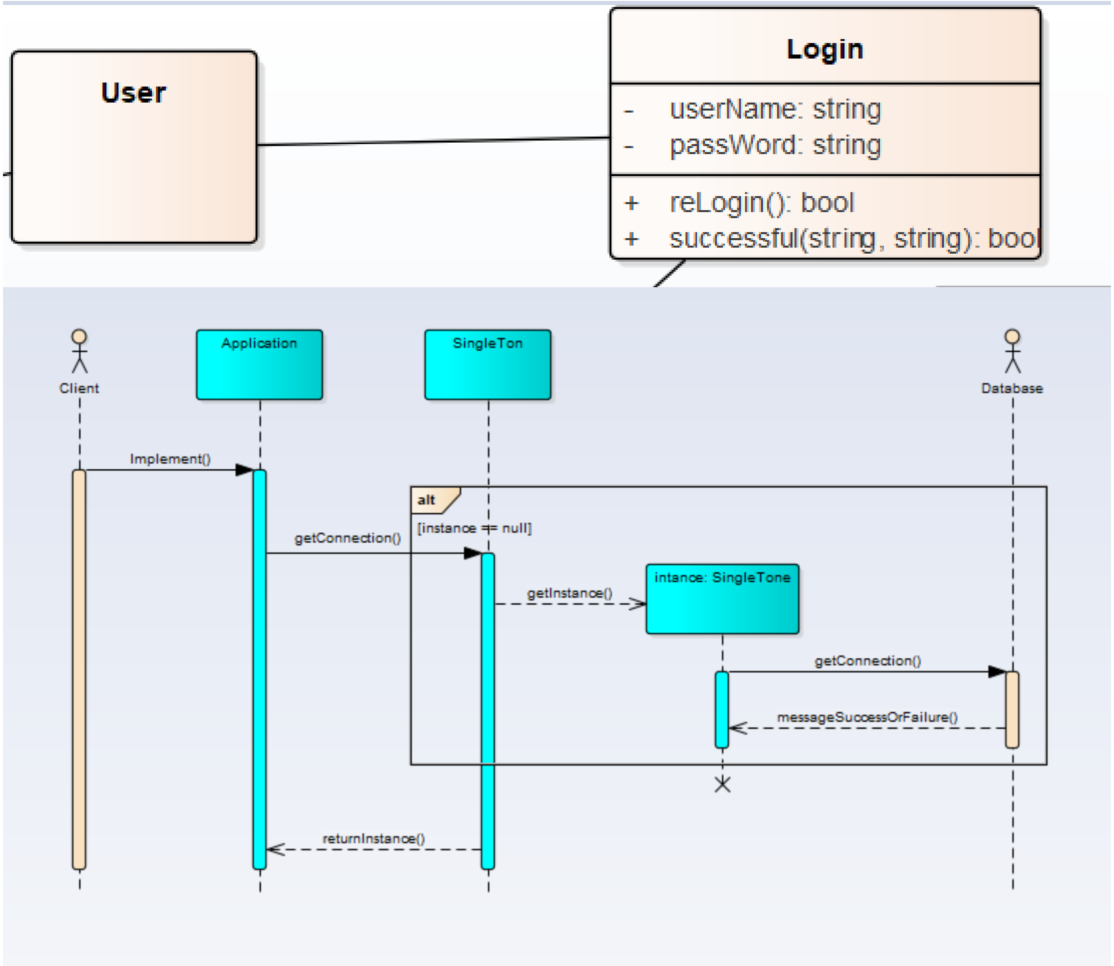
```
    4 references
    public enum TypeofGood
    {
        2 references | 1 reference
        Imported, Homemade
    }

    1 reference
    public class GoodFactory
    {
        1 reference
        public static GoodAbstractFactory GetFactory(TypeofGood type)
        {
            switch (type)
            {
                case TypeofGood.Homemade:
                    return new HomemadeFactory();
                case TypeofGood.Imported:
                    return new ImportedFactory();
                default:
                    return null;
            }
        }

    }
```



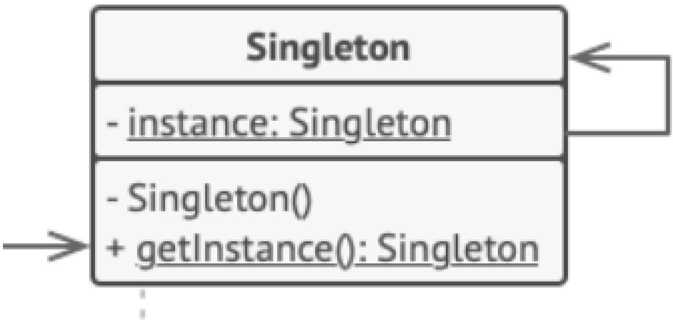
```
    0 references
    public class AbstractFactoryPattern
    {
        0 references
        public static void Run()
        {
            var imported = TypeofGood.Imported;
            GoodAbstractFactory factory = GoodFactory.GetFactory(imported);
            Food food = factory.getFood();
            food.show();
            Drink drink = factory.getDrink();
            drink.show();

        }
    }
}
```

2. Singleton

Name of function: Get connection



After login successfully, I will use singleton pattern for getting information of user from database



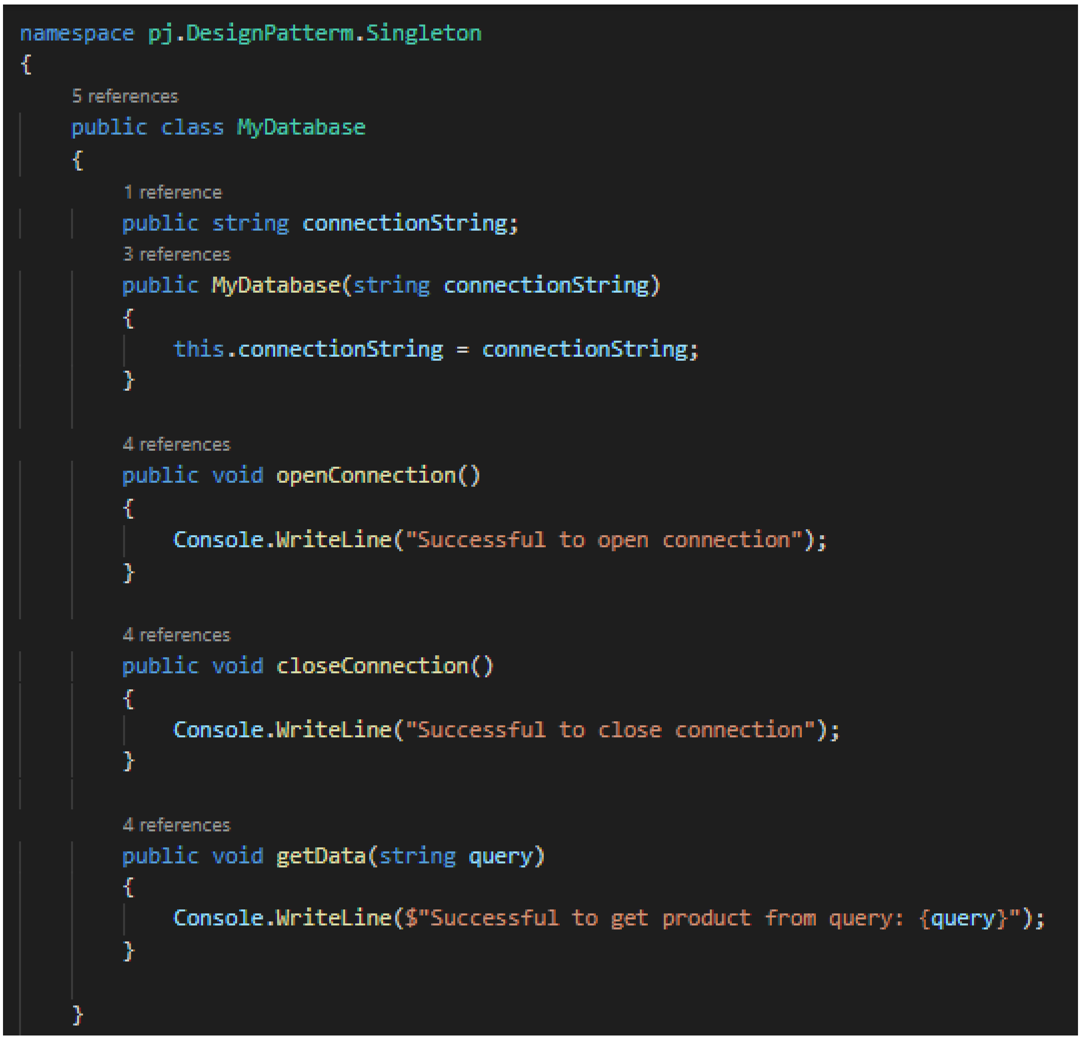
Reason:

In stead of using connection string to connect to database in many time and get user information, that way can save many time. I use this pattern to get some necessary information of user and save it in User class and public all of information.

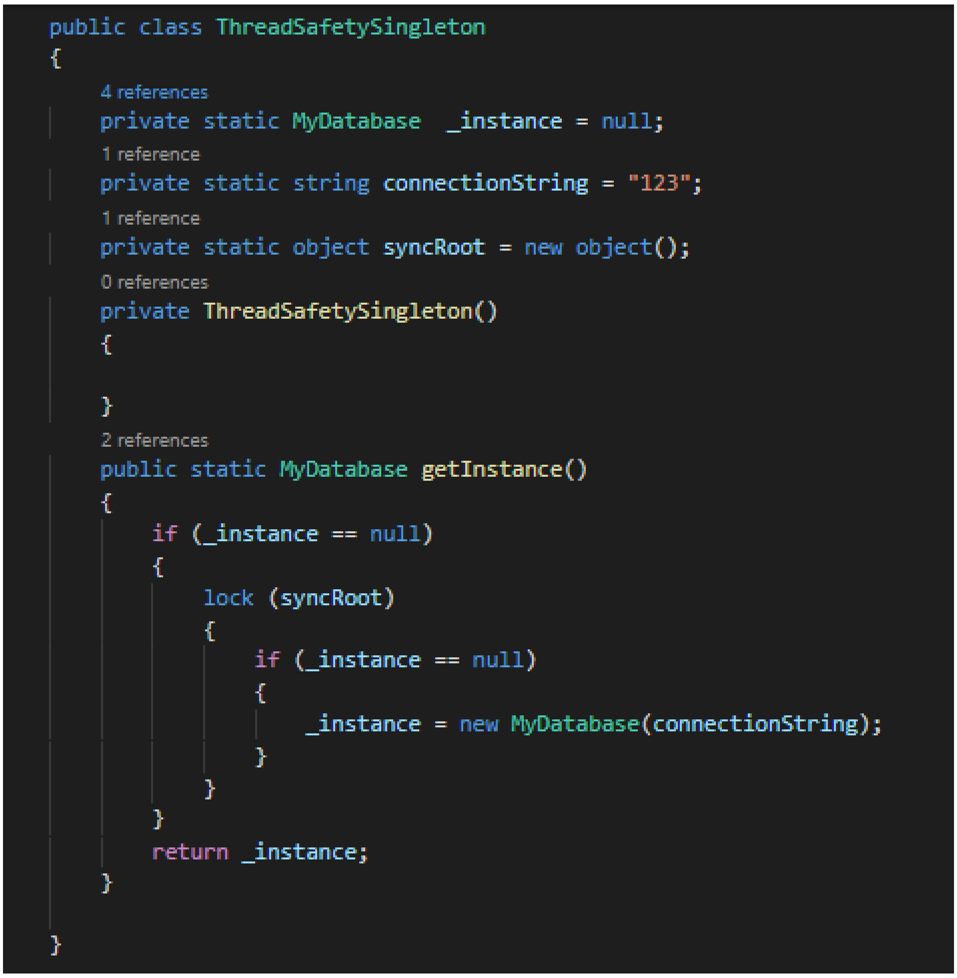
Characteristic:

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.

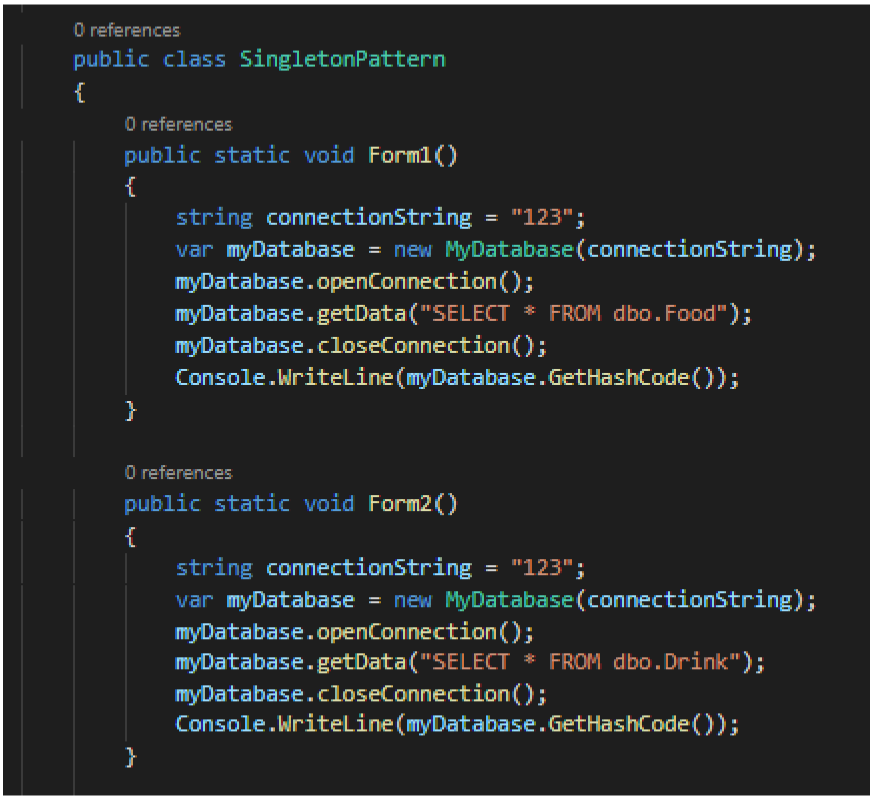
C# codes:



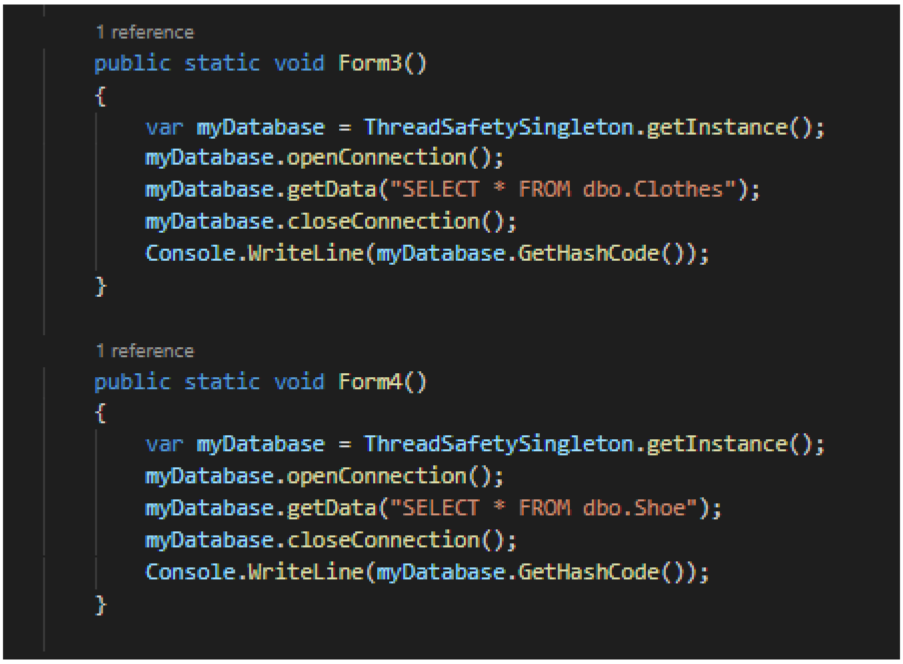
```
namespace pj.DesignPatterm.Singleton
{
    5 references
    public class MyDatabase
    {
        1 reference
        public string connectionString;
        3 references
        public MyDatabase(string connectionString)
        {
            this.connectionString = connectionString;
        }

        4 references
        public void openConnection()
        {
            Console.WriteLine("Successful to open connection");
        }

        4 references
        public void closeConnection()
        {
            Console.WriteLine("Successful to close connection");
        }

        4 references
        public void getData(string query)
        {
            Console.WriteLine($"Successful to get product from query: {query}");
        }

    }
```



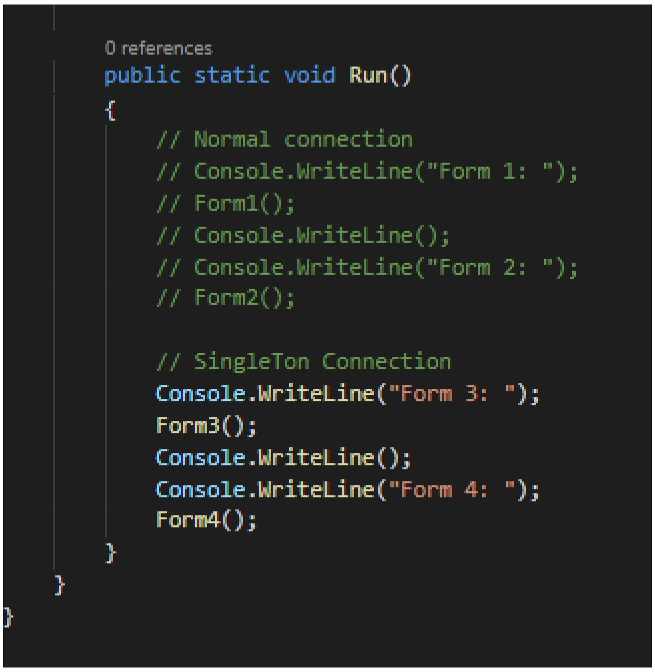
```
public class ThreadSafetySingleton
{
    4 references
    private static MyDatabase  _instance = null;
    1 reference
    private static string connectionString = "123";
    1 reference
    private static object syncRoot = new object();
    0 references
    private ThreadSafetySingleton()
    {

    }
    2 references
    public static MyDatabase getInstance()
    {
        if (_instance == null)
        {
            lock (syncRoot)
            {
                if (_instance == null)
                {
                    _instance = new MyDatabase(connectionString);
                }
            }
        }
        return _instance;
    }

}
```



```
    0 references
    public class SingletonPattern
    {
        0 references
        public static void Form1()
        {
            string connectionString = "123";
            var myDatabase = new MyDatabase(connectionString);
            myDatabase.openConnection();
            myDatabase.getData("SELECT * FROM dbo.Food");
            myDatabase.closeConnection();
            Console.WriteLine(myDatabase.GetHashCode());
        }

        0 references
        public static void Form2()
        {
            string connectionString = "123";
            var myDatabase = new MyDatabase(connectionString);
            myDatabase.openConnection();
            myDatabase.getData("SELECT * FROM dbo.Drink");
            myDatabase.closeConnection();
            Console.WriteLine(myDatabase.GetHashCode());
        }
```



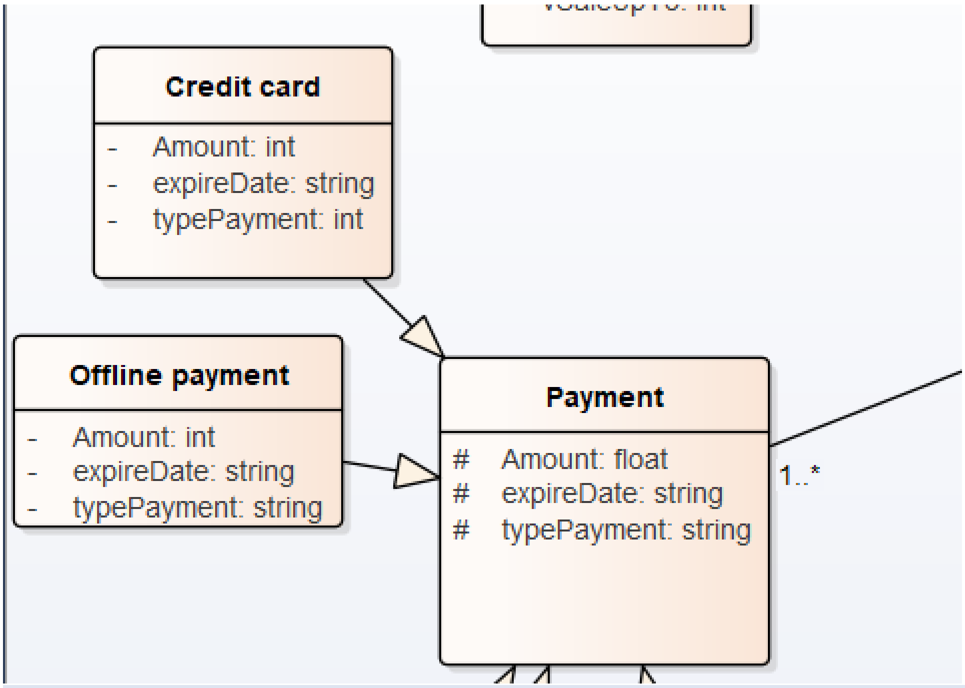
```
        1 reference
        public static void Form3()
        {
            var myDatabase = ThreadSafetySingleton.getInstance();
            myDatabase.openConnection();
            myDatabase.getData("SELECT * FROM dbo.Clothes");
            myDatabase.closeConnection();
            Console.WriteLine(myDatabase.GetHashCode());
        }

        1 reference
        public static void Form4()
        {
            var myDatabase = ThreadSafetySingleton.getInstance();
            myDatabase.openConnection();
            myDatabase.getData("SELECT * FROM dbo.Shoe");
            myDatabase.closeConnection();
            Console.WriteLine(myDatabase.GetHashCode());
        }
```

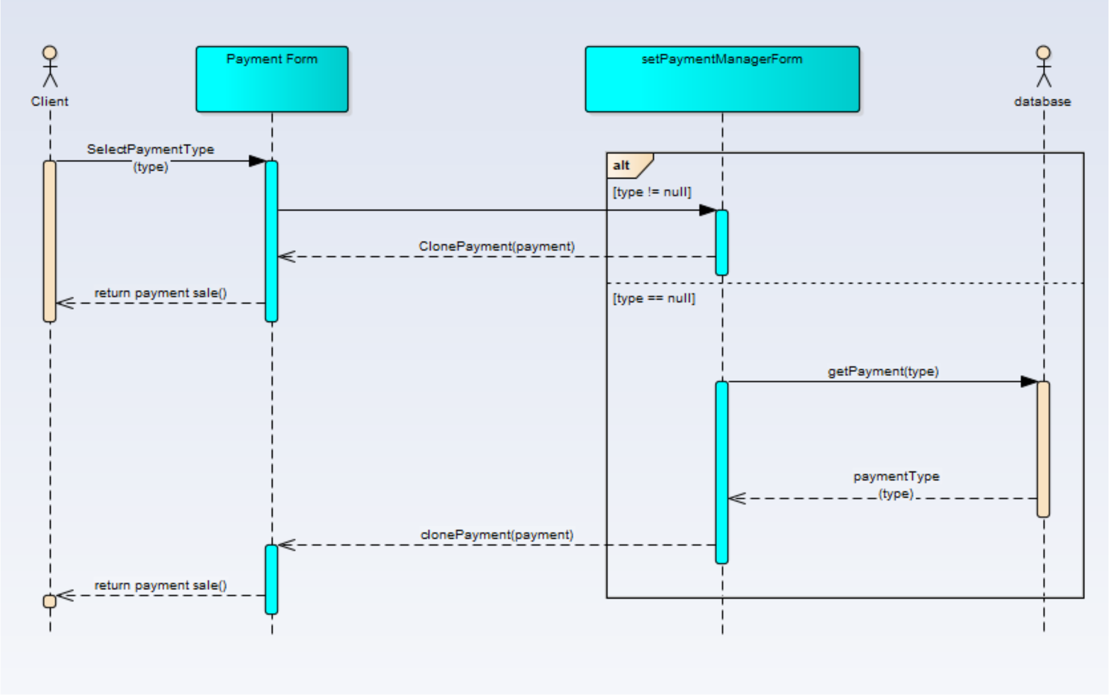


```
        0 references
        public static void Run()
        {
            // Normal connection
            // Console.WriteLine("Form 1: ");
            // Form1();
            // Console.WriteLine();
            // Console.WriteLine("Form 2: ");
            // Form2();

            // SingleTon Connection
            Console.WriteLine("Form 3: ");
            Form3();
            Console.WriteLine();
            Console.WriteLine("Form 4: ");
            Form4();
        }
    }
}
```

3. Prototype

Name of function: Payment





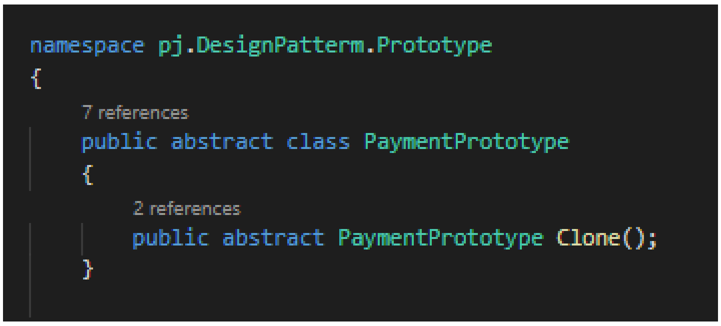
Reason:

Instead of creating 2 method (Offlinepayment, CreditCardPayment) which is not use Prototype pattern. You may write the same code like code in Payment class. Your code will loop and it is not good to maintain. So I decided to use Prototype pattern in this case

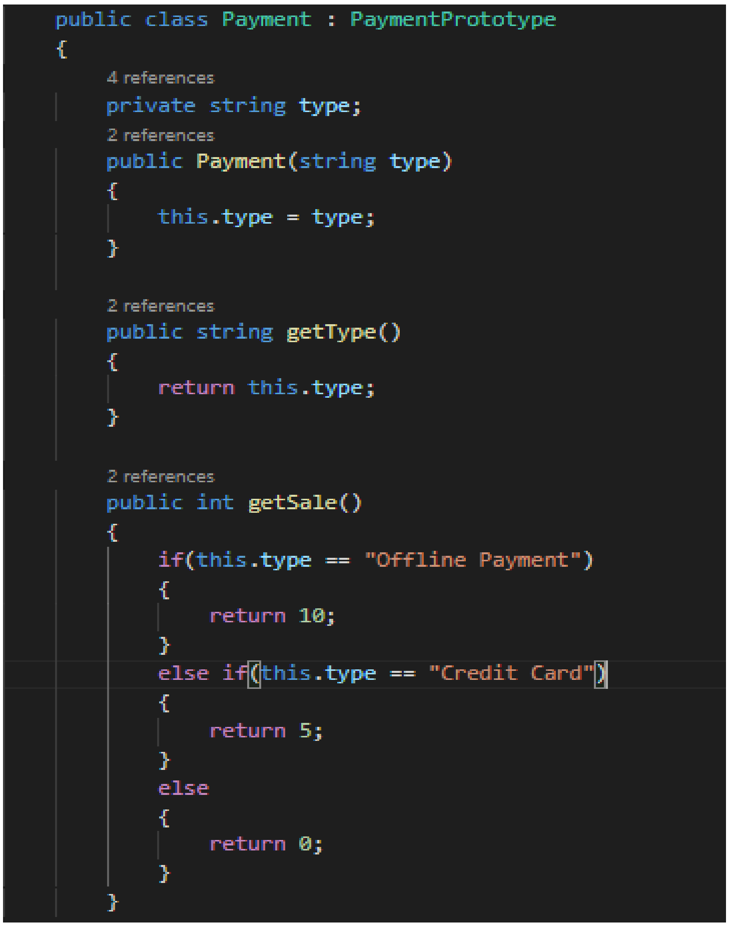
Characteristic:

To be used to create objects from a prototype object, by copying the properties of that object.

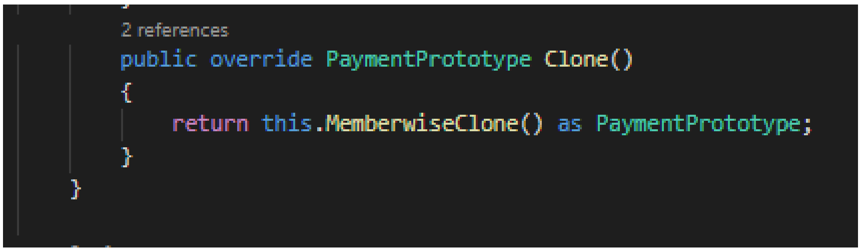
C# Code:



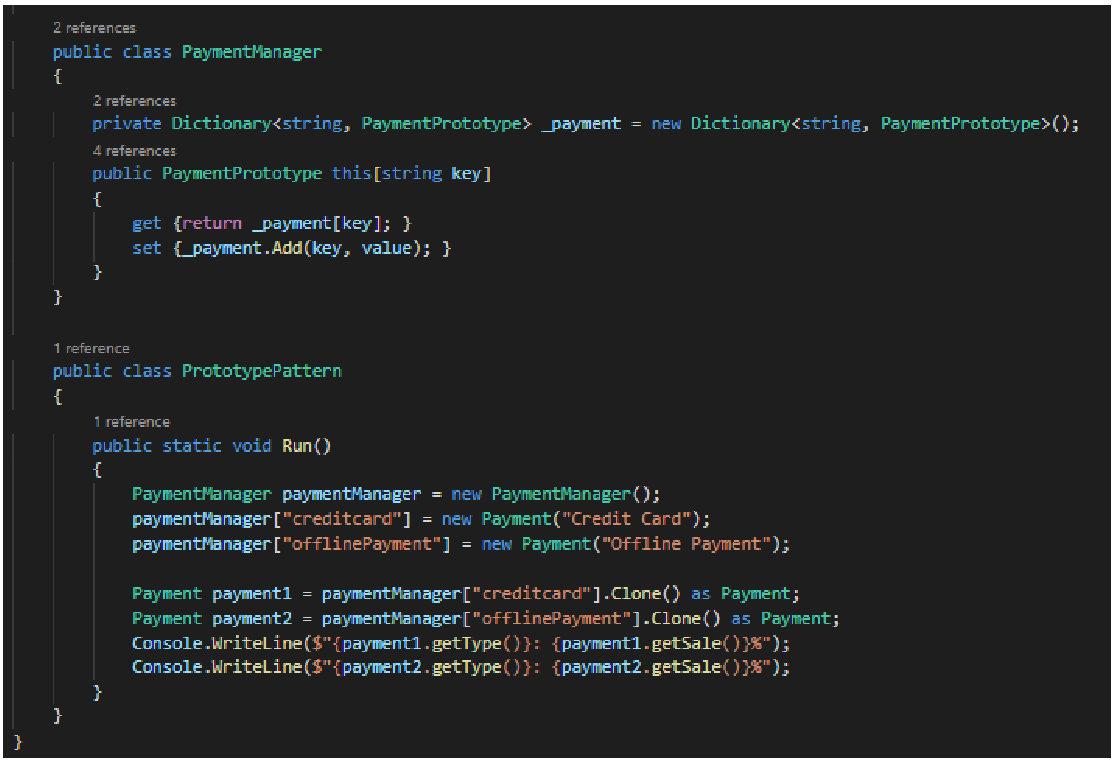
```
namespace pj.DesignPatterm.Prototype
{
    7 references
    public abstract class PaymentPrototype
    {
        2 references
        public abstract PaymentPrototype Clone();
    }
```



```
    public class Payment : PaymentPrototype
    {
        4 references
        private string type;
        2 references
        public Payment(string type)
        {
            this.type = type;
        }

        2 references
        public string getType()
        {
            return this.type;
        }

        2 references
        public int getSale()
        {
            if(this.type == "Offline Payment")
            {
                return 10;
            }
            else if(this.type == "Credit Card")
            {
                return 5;
            }
            else
            {
                return 0;
            }
        }
```



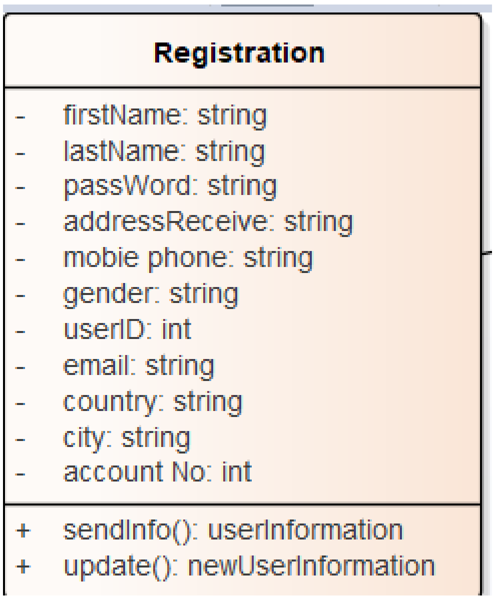
```
        2 references
        public override PaymentPrototype Clone()
        {
            return this.MemberwiseClone() as PaymentPrototype;
        }
    }
```

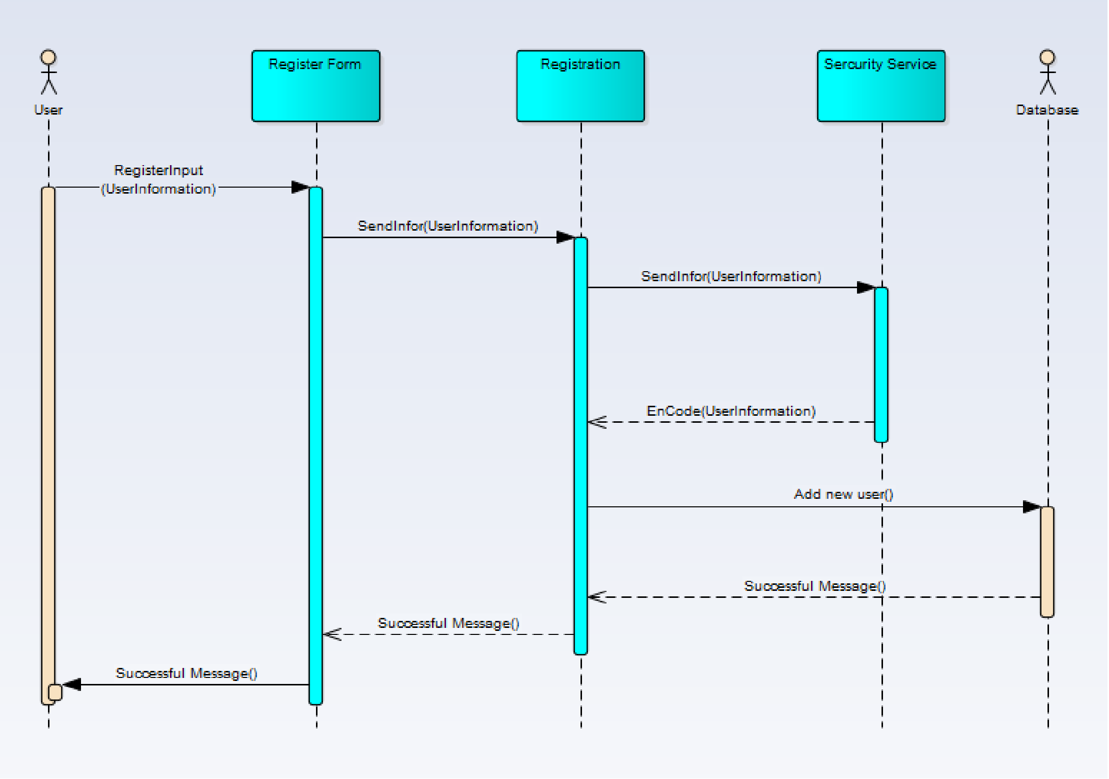


```
    2 references
    public class PaymentManager
    {
        2 references
        private Dictionary<string, PaymentPrototype> _payment = new Dictionary<string, PaymentPrototype>();
        4 references
        public PaymentPrototype this[string key]
        {
            get {return _payment[key]; }
            set {_payment.Add(key, value); }
        }
    }

    1 reference
    public class PrototypePattern
    {
        1 reference
        public static void Run()
        {
            PaymentManager paymentManager = new PaymentManager();
            paymentManager["creditcard"] = new Payment("Credit Card");
            paymentManager["offlinePayment"] = new Payment("Offline Payment");

            Payment payment1 = paymentManager["creditcard"].Clone() as Payment;
            Payment payment2 = paymentManager["offlinePayment"].Clone() as Payment;
            Console.WriteLine($"{payment1.getType()}: {payment1.getSale()}%");
            Console.WriteLine($"{payment2.getType()}: {payment2.getSale()}%");
        }
    }
}
```

4. Adapter

Name of function: Register





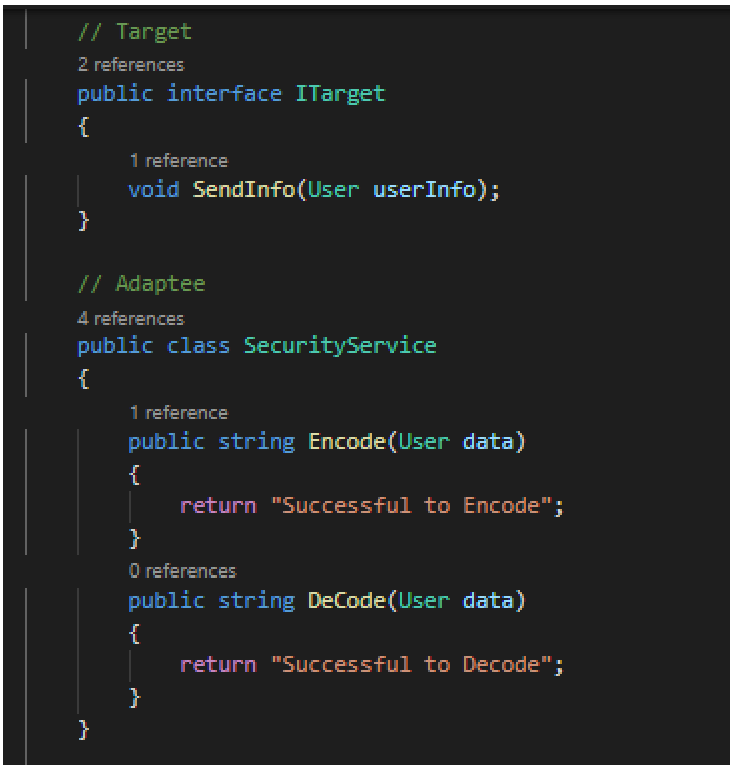
Reason:

Other class which is need to send information or update user information can use ITarget interface. This can help you control easily your code.

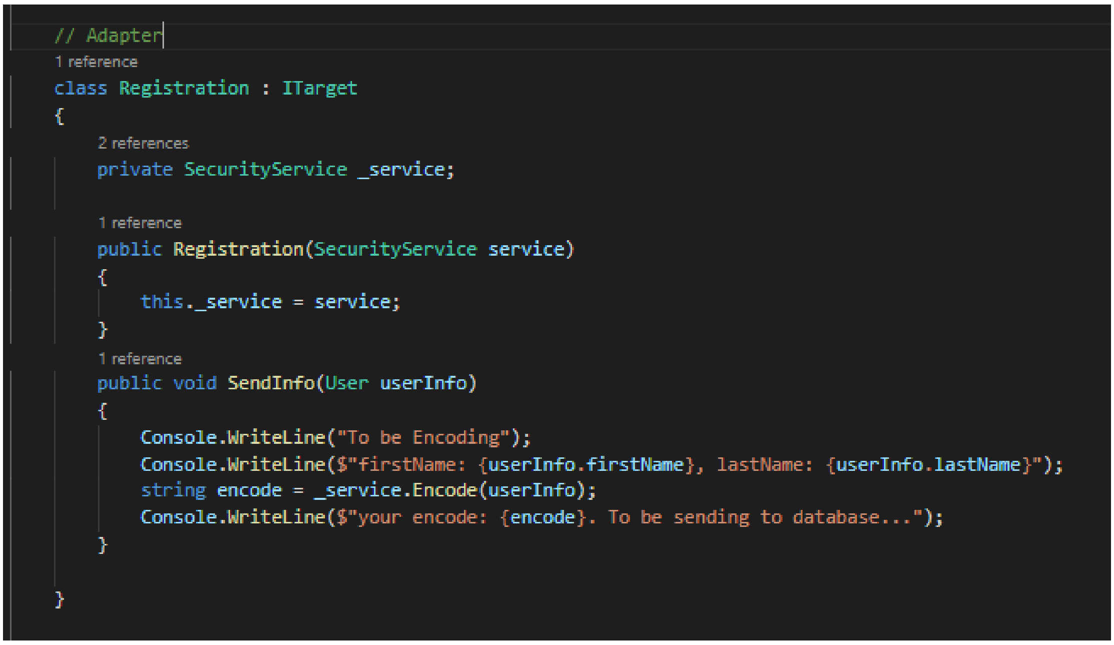
Characteristic:

Adapter is recognizable by a constructor which takes an instance of a different abstract/interface type. When the adapter receives a call to any of its methods, it translates parameters to the appropriate format and then directs the call to one or several methods of the wrapped object.

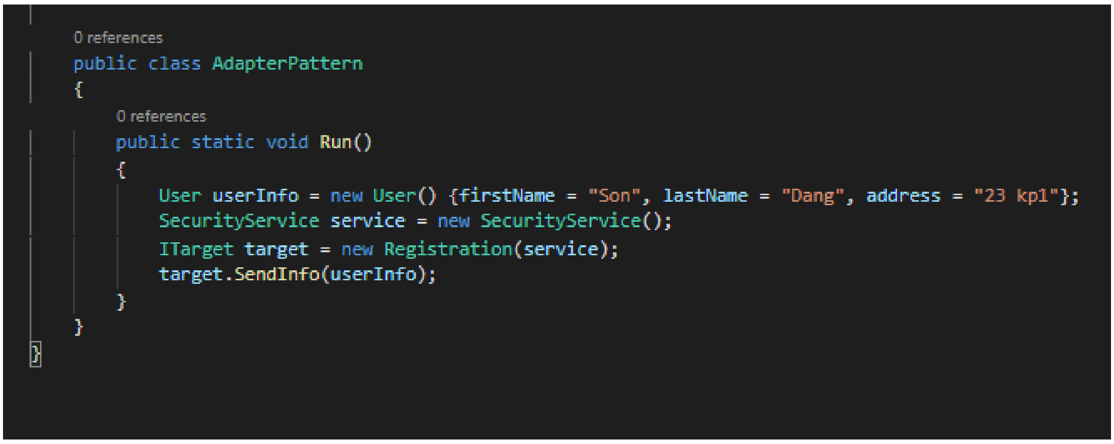
C# Code:



```
// Target
2 references
public interface ITarget
{
    1 reference
    void SendInfo(User userInfo);
}

// Adaptee
4 references
public class SecurityService
{
    1 reference
    public string Encode(User data)
    {
        return "Successful to Encode";
    }
    0 references
    public string DeCode(User data)
    {
        return "Successful to Decode";
    }
}
```



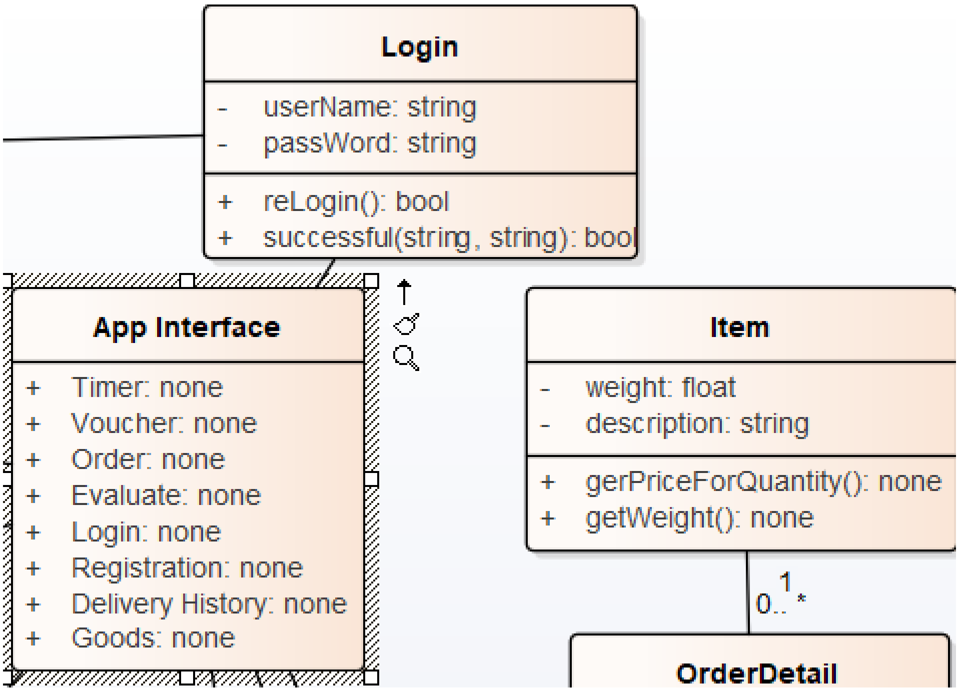
```
// Adapter
1 reference
class Registration : ITarget
{
    2 references
    private SecurityService _service;

    1 reference
    public Registration(SecurityService service)
    {
        this._service = service;
    }
    1 reference
    public void SendInfo(User userInfo)
    {
        Console.WriteLine("To be Encoding");
        Console.WriteLine($"firstName: {userInfo.firstName}, lastName: {userInfo.lastName}");
        string encode = _service.Encode(userInfo);
        Console.WriteLine($"your encode: {encode}. To be sending to database...");
    }

}
```

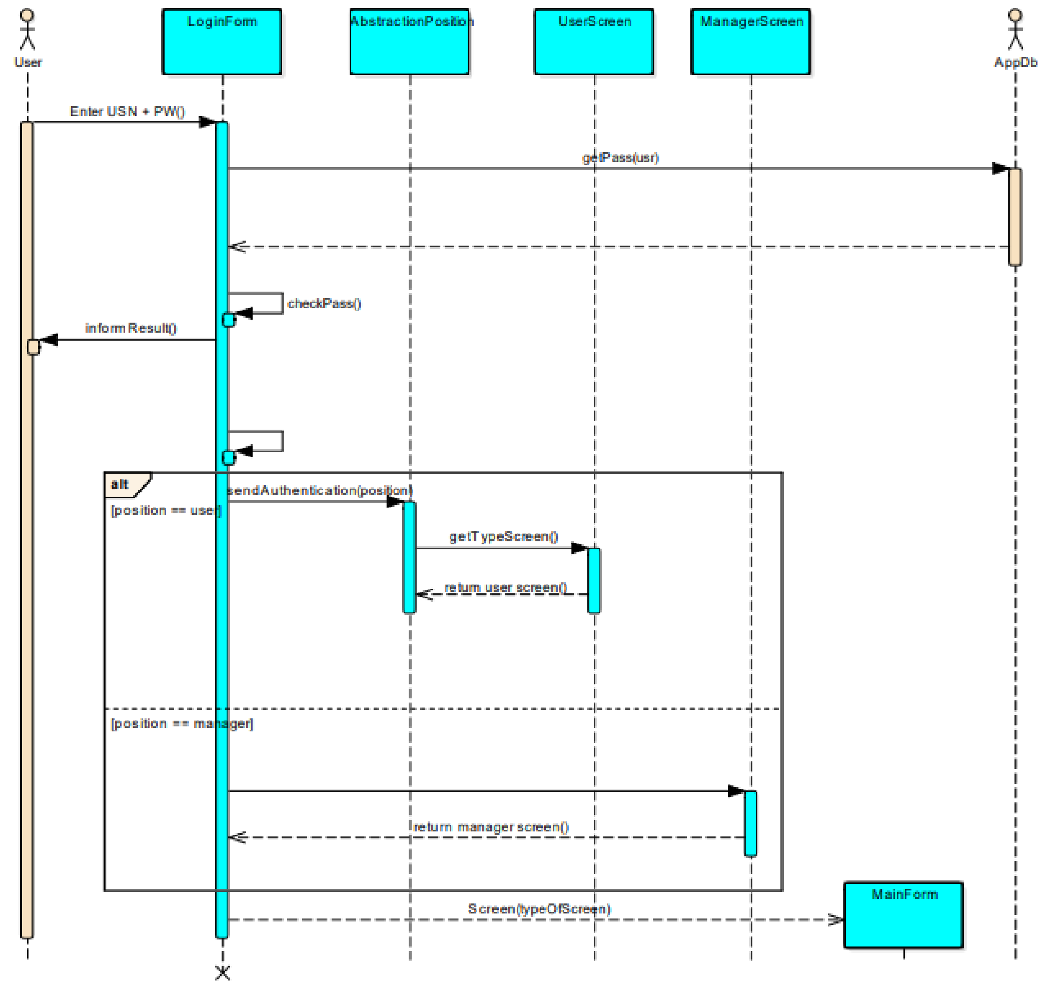


```
0 references
public class AdapterPattern
{
    0 references
    public static void Run()
    {
        User userInfo = new User() {firstName = "Son", lastName = "Dang", address = "23 kp1"};
        SecurityService service = new SecurityService();
        ITarget target = new Registration(service);
        target.SendInfo(userInfo);
    }
}
}
```

5. Bridge

Name of function: Login





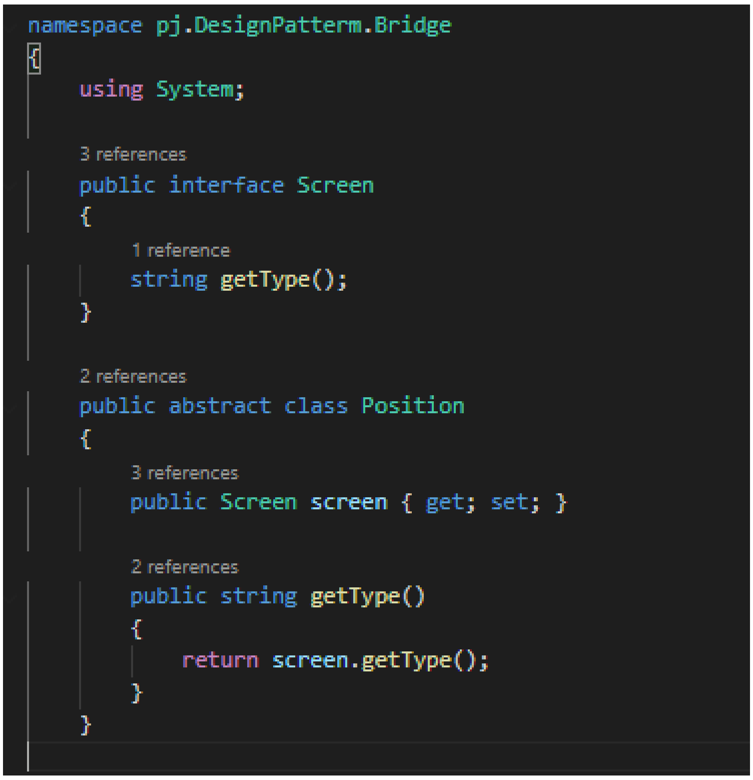
Reason:

After login user and manage will have different screen because they are different position, so bridge in this case is suitable.

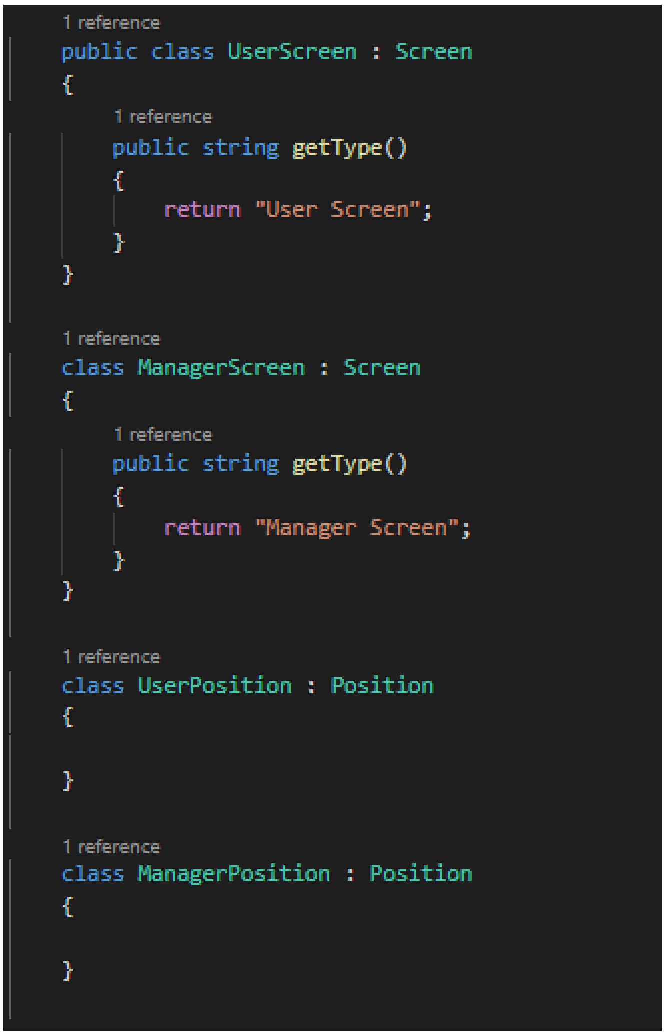
Characteristic:

Bridge can be recognized by a clear distinction between some controlling entity and several different platforms that it relies on.

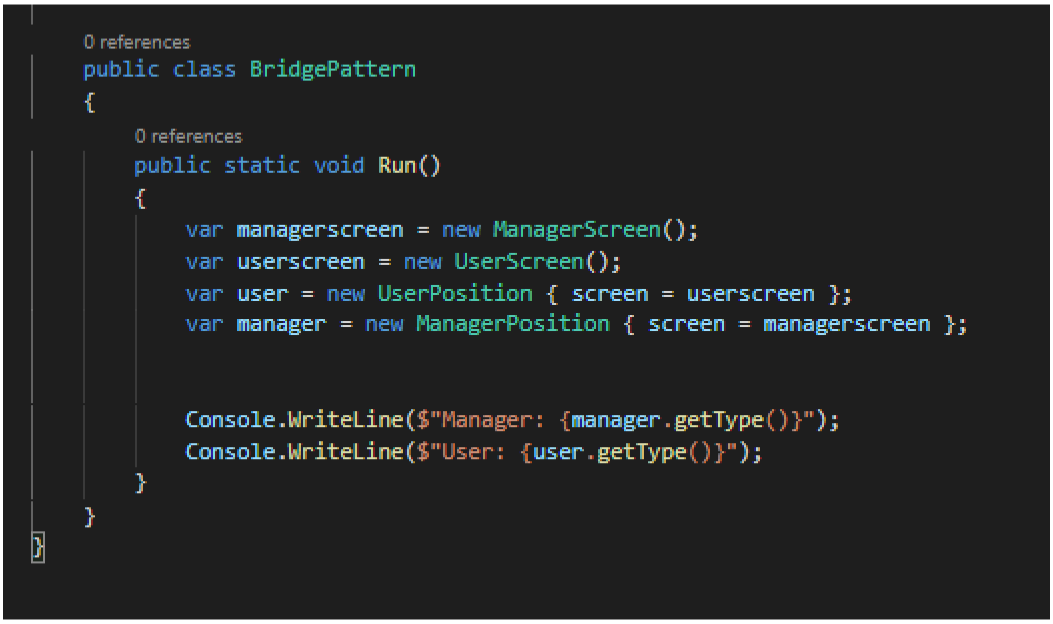
C# Code:



```
namespace pj.DesignPatterm.Bridge
{
    using System;

    3 references
    public interface Screen
    {
        1 reference
        string getType();
    }

    2 references
    public abstract class Position
    {
        3 references
        public Screen screen { get; set; }

        2 references
        public string getType()
        {
            return screen.getType();
        }
    }
```



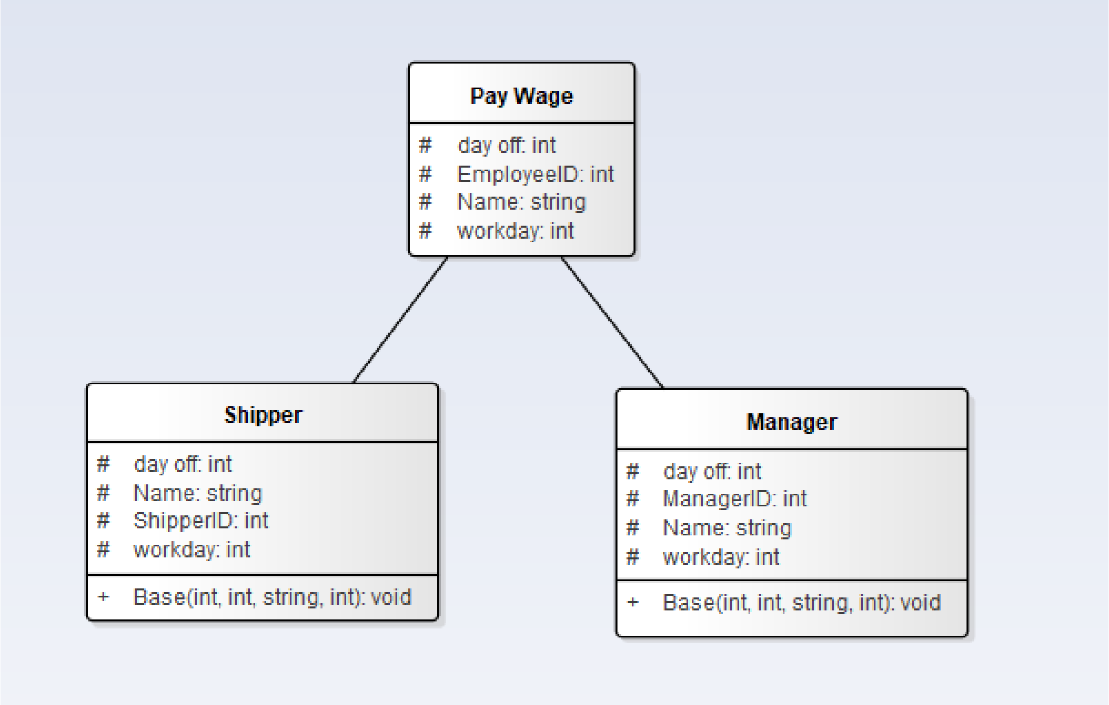
```
1 reference
public class UserScreen : Screen
{
    1 reference
    public string getType()
    {
        return "User Screen";
    }
}

1 reference
class ManagerScreen : Screen
{
    1 reference
    public string getType()
    {
        return "Manager Screen";
    }
}

1 reference
class UserPosition : Position
{

}

1 reference
class ManagerPosition : Position
{

}
```



```
    0 references
    public class BridgePattern
    {
        0 references
        public static void Run()
        {
            var managerscreen = new ManagerScreen();
            var userscreen = new UserScreen();
            var user = new UserPosition { screen = userscreen };
            var manager = new ManagerPosition { screen = managerscreen };


            Console.WriteLine($"Manager: {manager.getType()}");
            Console.WriteLine($"User: {user.getType()}");
        }
    }
}
```

6. Visitor

Name of function: Pay Wage



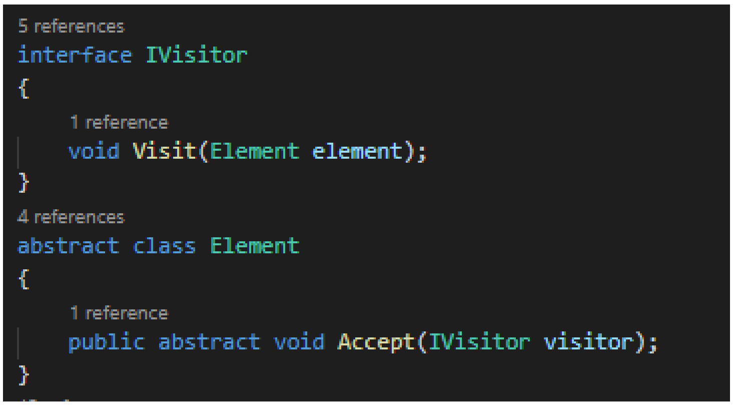
Reason:

With the features of the visitor to help implement the payroll function for employees, we decided to choose the visitor pattern

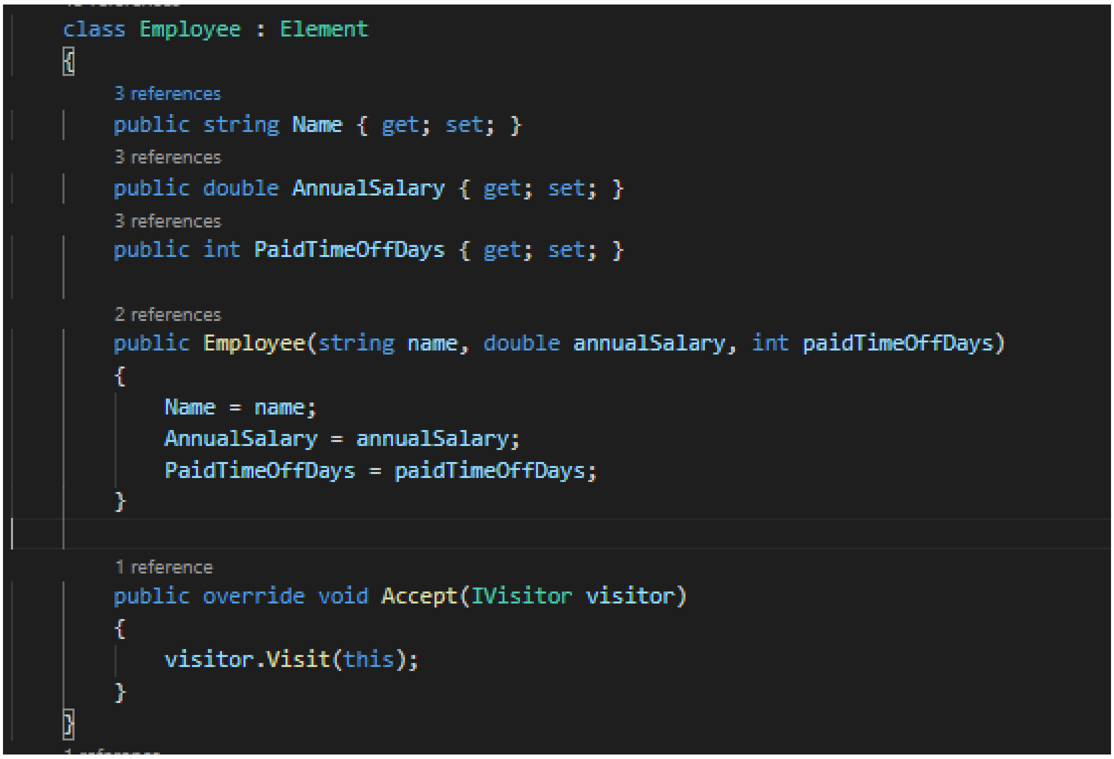
Characteristic:

- The visitor pattern or visitor design pattern is a pattern that will separate an algorithm from the object structure on which it operates. It describes a way to add new operations to existing object structures without modifying the structures themselves.
- This characteristic makes visitor patterns a way to implement the open/closed principle (OCP).

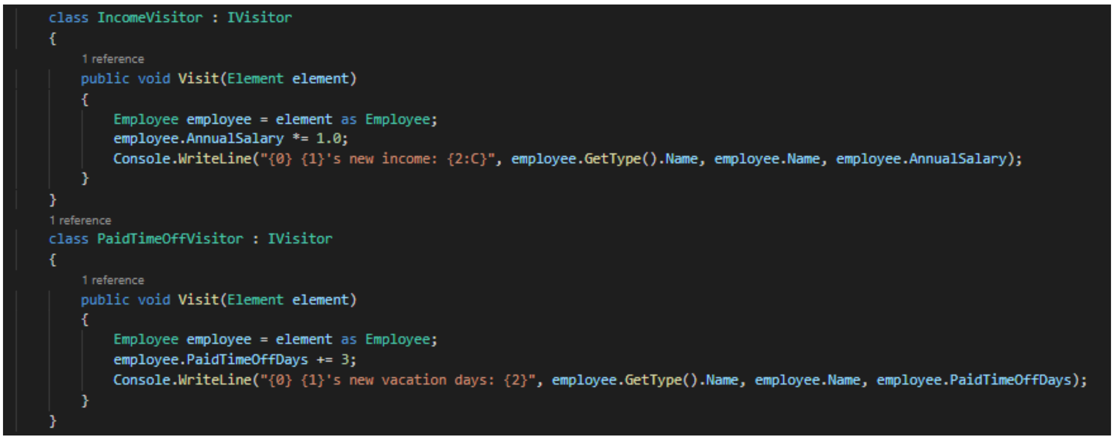
C# Code:



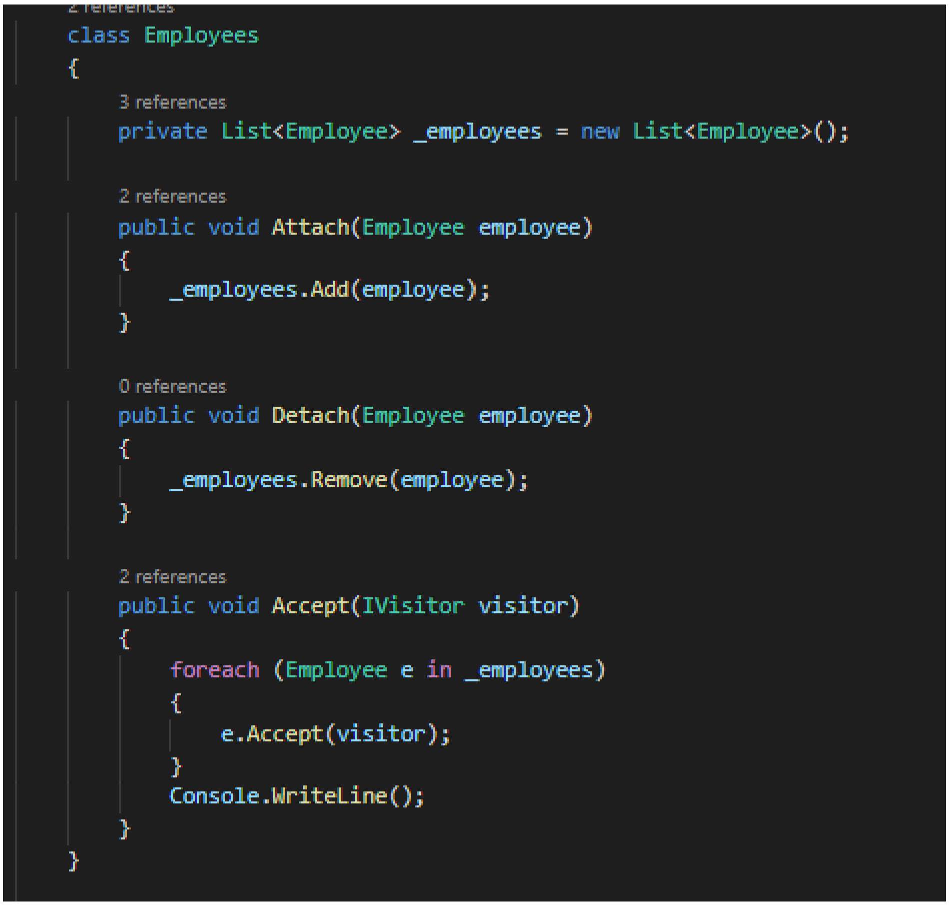
```
5 references
interface IVisitor
{
    1 reference
    void Visit(Element element);
}
4 references
abstract class Element
{
    1 reference
    public abstract void Accept(IVisitor visitor);
}
```



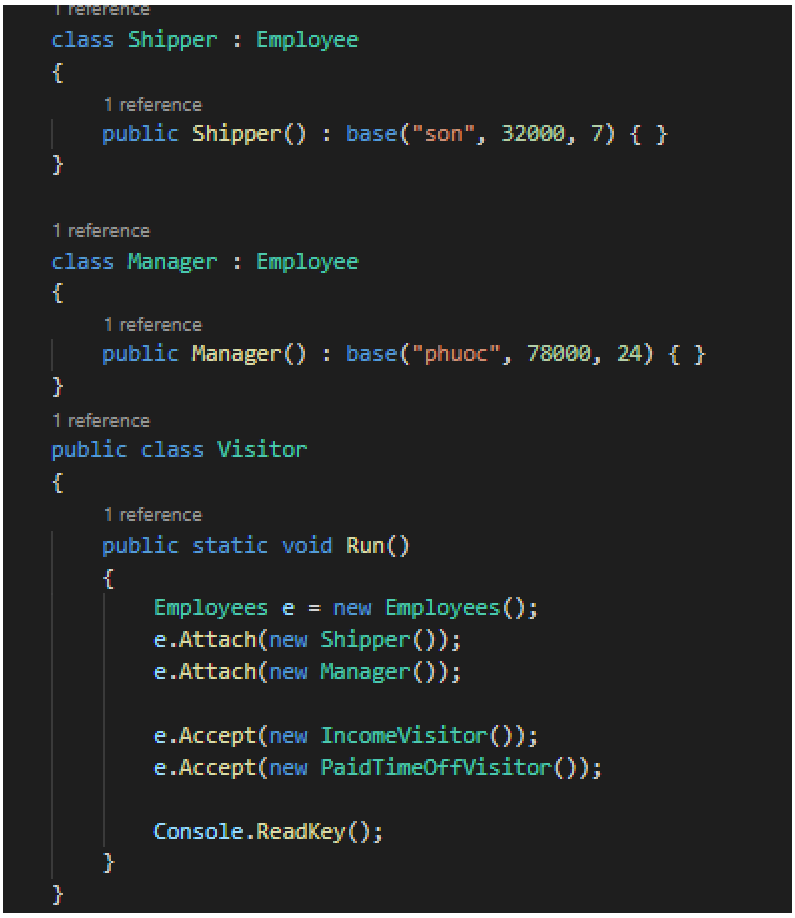
```
class Employee : Element
{
    3 references
    public string Name { get; set; }
    3 references
    public double AnnualSalary { get; set; }
    3 references
    public int PaidTimeOffDays { get; set; }

    2 references
    public Employee(string name, double annualSalary, int paidTimeOffDays)
    {
        Name = name;
        AnnualSalary = annualSalary;
        PaidTimeOffDays = paidTimeOffDays;
    }

    1 reference
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```



```
class IncomeVisitor : IVisitor
{
    1 reference
    public void Visit(Element element)
    {
        Employee employee = element as Employee;
        employee.AnnualSalary *= 1.0;
        Console.WriteLine("{0} {1}'s new income: {2:C}", employee.GetType().Name, employee.Name, employee.AnnualSalary);
    }
}
1 reference
class PaidTimeOffVisitor : IVisitor
{
    1 reference
    public void Visit(Element element)
    {
        Employee employee = element as Employee;
        employee.PaidTimeOffDays += 3;
        Console.WriteLine("{0} {1}'s new vacation days: {2}", employee.GetType().Name, employee.Name, employee.PaidTimeOffDays);
    }
}
```



```
2 references
class Employees
{
    3 references
    private List<Employee> _employees = new List<Employee>();

    2 references
    public void Attach(Employee employee)
    {
        _employees.Add(employee);
    }

    0 references
    public void Detach(Employee employee)
    {
        _employees.Remove(employee);
    }

    2 references
    public void Accept(IVisitor visitor)
    {
        foreach (Employee e in _employees)
        {
            e.Accept(visitor);
        }
        Console.WriteLine();
    }
}
```



```
1 reference
class Shipper : Employee
{
    1 reference
    public Shipper() : base("son", 32000, 7) { }
}

1 reference
class Manager : Employee
{
    1 reference
    public Manager() : base("phuoc", 78000, 24) { }
}
1 reference
public class Visitor
{
    1 reference
    public static void Run()
    {
        Employees e = new Employees();
        e.Attach(new Shipper());
        e.Attach(new Manager());

        e.Accept(new IncomeVisitor());
        e.Accept(new PaidTimeOffVisitor());

        Console.ReadKey();
    }
}
```