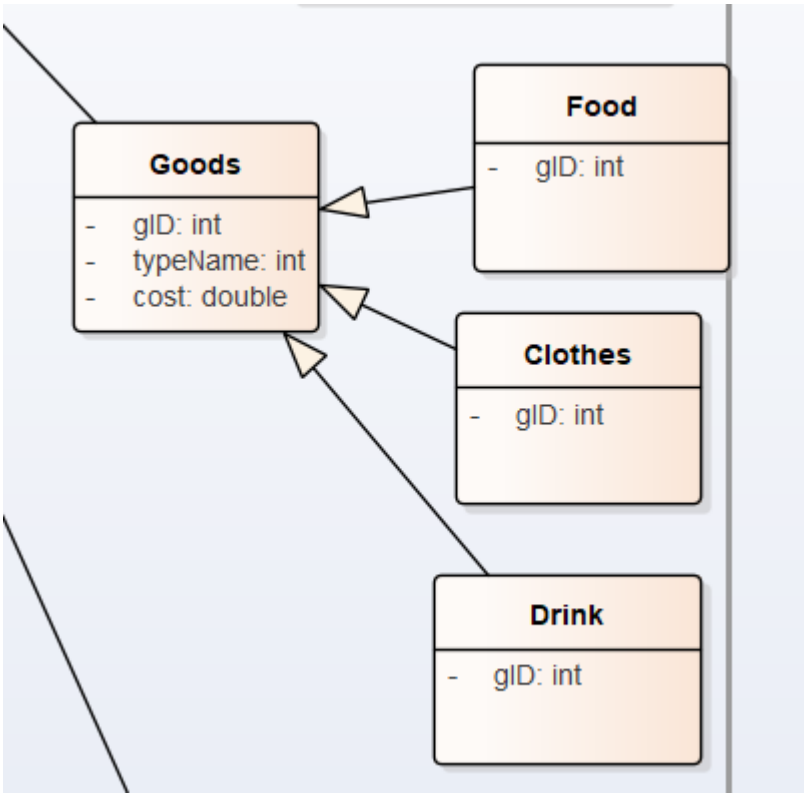
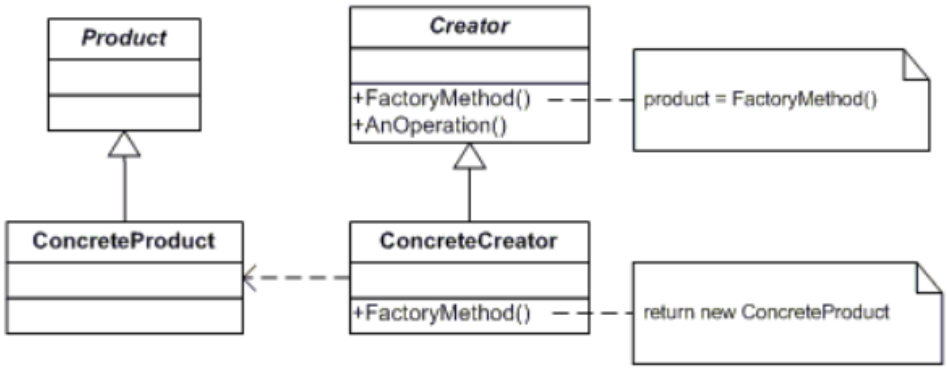
Object-oriented software design (design pattern)

1. Factory

Name of function: Get Goods





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:

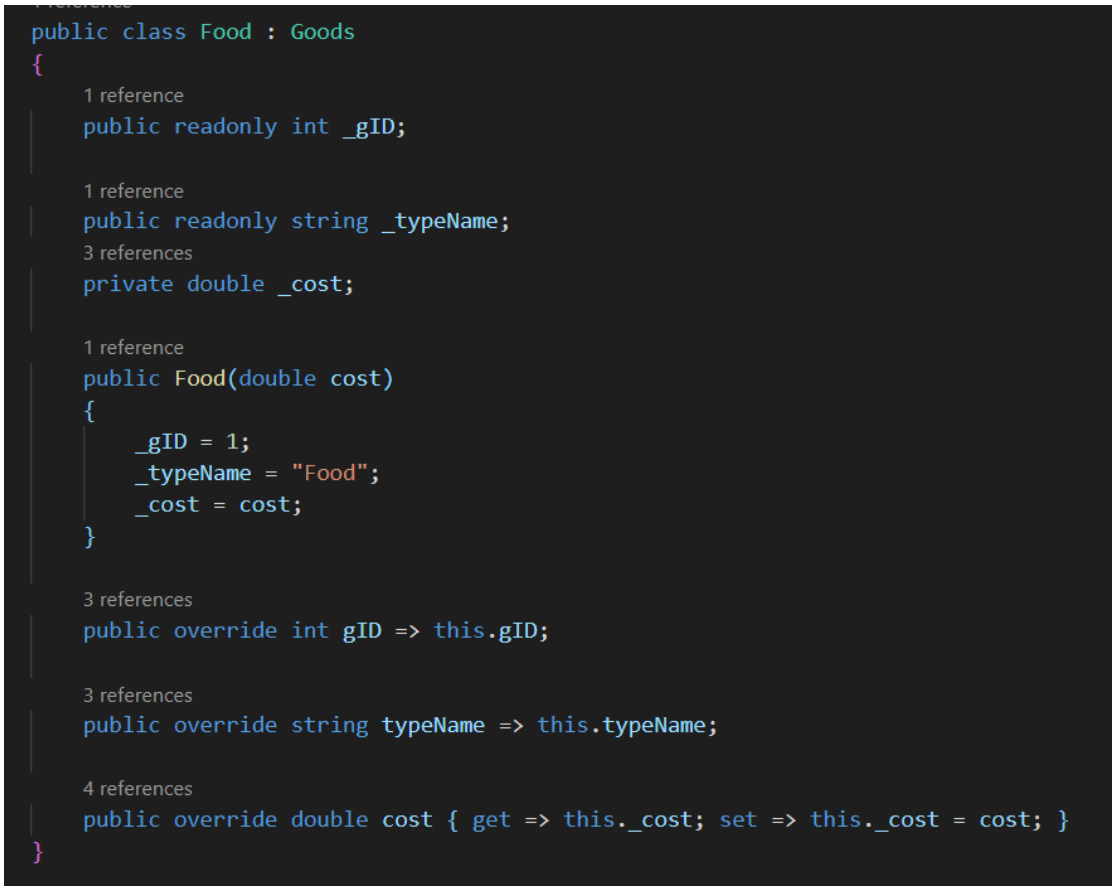
- Creates objects without exposing the instantiation logic to the client.
- Refers to the newly created object through a common interface

C# Codes:

```
0 references
static void Main(string[] args)
{
    GoodsFactory factory = null;
    int id = 1;
    string type = "Food";

    if(id == 1 && type == "Food")
    {
        factory = new FoodFactory(51000);
    }
    else if(id == 2 && type == "Clothes")
    {
        factory = new ClothesFactory(23000);
    }
    else if(id == 3 && type == "Drinks")
    {
        factory = new DrinkFactory(12344000);
    }
    Console.WriteLine($"{factory.GetFoods().cost}");
}
```

```
// Product
7 references
public abstract class Goods
{
    3 references
    public abstract int gID { get; }
    3 references
    public abstract string typeName { get; }
    4 references
    public abstract double cost { get; set; }
}
```

```
// ConcreteProduct
1 reference
public class Food : Goods ···

1 reference
public class Clothes : Goods ···

1 reference
public class Drink : Goods ···
```



```
public class Food : Goods
{
    1 reference
    public readonly int _gID;

    1 reference
    public readonly string _typeName;
    3 references
    private double _cost;

    1 reference
    public Food(double cost)
    {
        _gID = 1;
        _typeName = "Food";
        _cost = cost;
    }

    3 references
    public override int gID => this.gID;

    3 references
    public override string typeName => this.typeName;

    4 references
    public override double cost { get => this._cost; set => this._cost = cost; }
}
```

```
1 reference
public class Clothes : Goods
{
    1 reference
    public readonly int _gID;

    1 reference
    public readonly string _typeName;
    3 references
    private double _cost;

    1 reference
    public Clothes(double cost)
    {
        _gID = 2;
        _typeName = "Clothes";
        _cost = cost;
    }

    3 references
    public override int gID => this.gID;

    3 references
    public override string typeName => this.typeName;

    4 references
    public override double cost { get => this._cost; set => this._cost = cost; }
}
```

```
1 reference
public class Clothes : Goods
{
    1 reference
    public readonly int _gID;

    1 reference
    public readonly string _typeName;
    3 references
    private double _cost;

    1 reference
    public Clothes(double cost)
    {
        _gID = 2;
        _typeName = "Clothes";
        _cost = cost;
    }

    3 references
    public override int gID => this.gID;

    3 references
    public override string typeName => this.typeName;

    4 references
    public override double cost { get => this._cost; set => this._cost = cost; }
}
1 reference
public class Drink : Goods
{
    1 reference
    public readonly int _gID;

    1 reference
    public readonly string _typeName;
    3 references
    private double _cost;

    1 reference
    public Drink(double cost)
    {
        _gID = 3;
        _typeName = "Drink";
        _cost = cost;
    }

    3 references
    public override int gID => this.gID;

    3 references
    public override string typeName => this.typeName;

    4 references
    public override double cost { get => this._cost; set => this._cost = cost; }
}
```

```
// Creator
4 references
public abstract class GoodsFactory
{
    1 reference
    public abstract Goods GetFoods();
}
```

```
// ConcreteCreator
1 reference
public class FoodFactory : GoodsFactory···

1 reference
public class ClothesFactory : GoodsFactory···

1 reference
public class DrinkFactory : GoodsFactory···
```

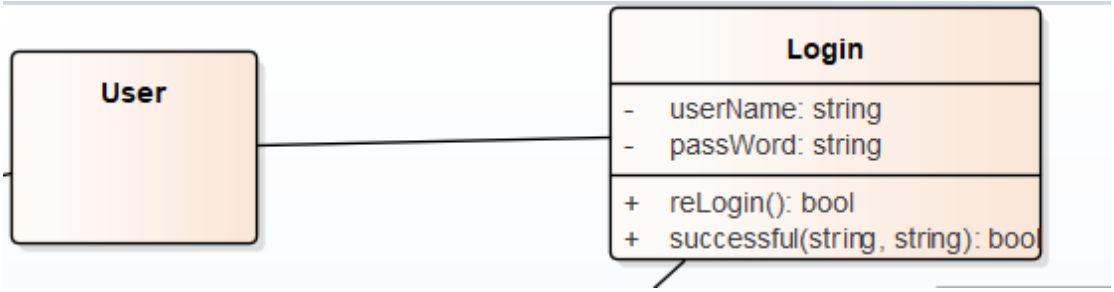
```
public class FoodFactory : GoodsFactory
{
    2 references
    private double _cost;

    1 reference
    public FoodFactory(double cost)
    {
        _cost = cost;
    }
    1 reference
    public override Goods GetFoods()
    {
        return new Food(_cost);
    }
}
```

```
1 reference
public class ClothesFactory : GoodsFactory
{
    2 references
    private double _cost;

    1 reference
    public ClothesFactory(double cost)
    {
        _cost = cost;
    }
    1 reference
    public override Goods GetFoods()
    {
        return new Clothes(_cost);
    }
}
```

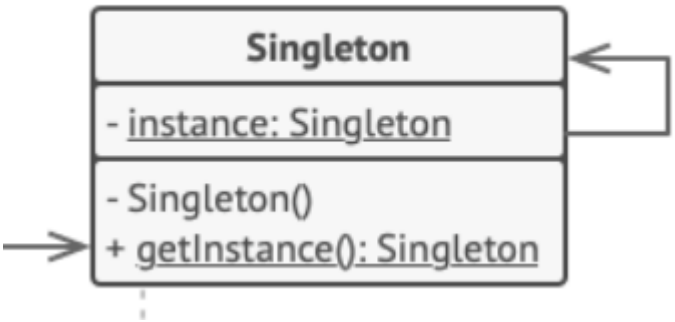
```
1 reference
public class DrinkFactory : GoodsFactory
{
    2 references
    private double _cost;

    1 reference
    public DrinkFactory(double cost)
    {
        _cost = cost;
    }
    1 reference
    public override Goods GetFoods()
    {
        return new Drink(_cost);
    }
}
```

2. Singleton

Name of function: Login



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:

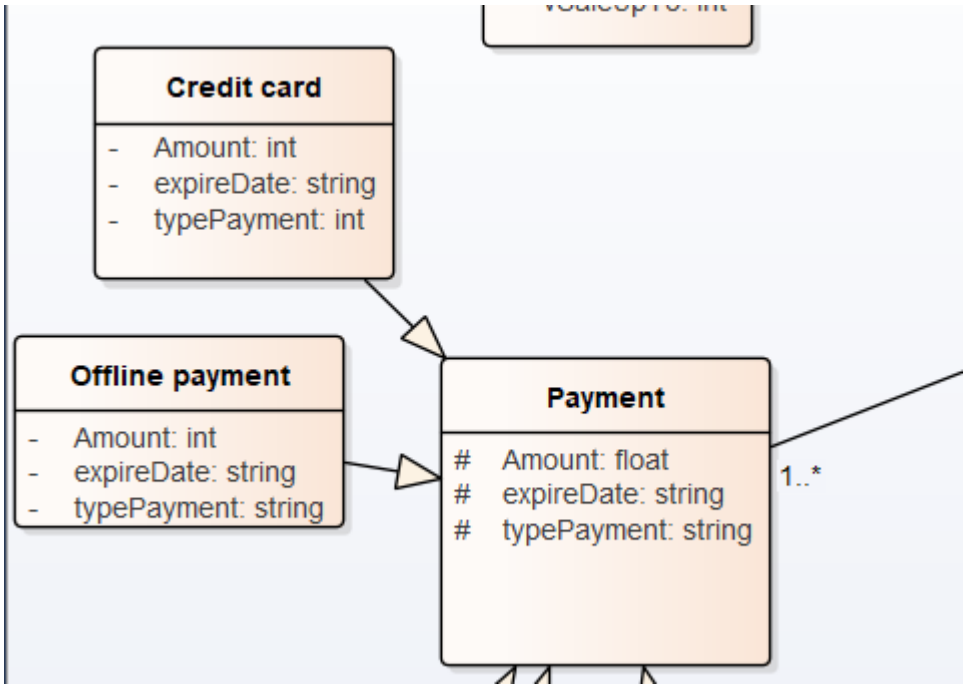
```
7 references
public class User
{
    3 references
    public string firstName { get; set; }
    3 references
    public string lastName { get; set; }
    3 references
    public string address { get; set; }
}
1 reference
public class UserInfomation
{

    1 reference
    public static User getUserInformation(string _userName)
    {
        if(_userName == "son")
        {
            return new User {firstName = "Son", lastName = "Dang", address = "23 pk1"};
        }
        else if(_userName == "phuoc")
        {
            return new User {firstName = "Phuoc", lastName = "Pham", address = "none"};
        }
        else if(_userName == "phu")
        {
            return new User {firstName = "Phu", lastName = "Nguyen", address = "none"};
        }
        return null;
    }

}
```

```
public class ThreadSafetySingleton
{
    4 references
    private static User _instance = null;
    1 reference
    private static object syncRoot = new object();
    0 references
    private ThreadSafetySingleton()
    {

    }
    2 references
    public static User getInstance(string userName)
    {
        if(_instance == null)
        {
            lock(syncRoot)
            {
                if(_instance == null)
                {
                    _instance = UserInfomation.getUserInformation(userName);
                }
            }
        }
        return _instance;
    }

}
```

```
public static void Main()
{
    User user = ThreadSafetySingleton.getInstance("son");
    Console.WriteLine(user.GetHashCode());
    user = ThreadSafetySingleton.getInstance("son");
    Console.WriteLine(user.GetHashCode());
}
```

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:

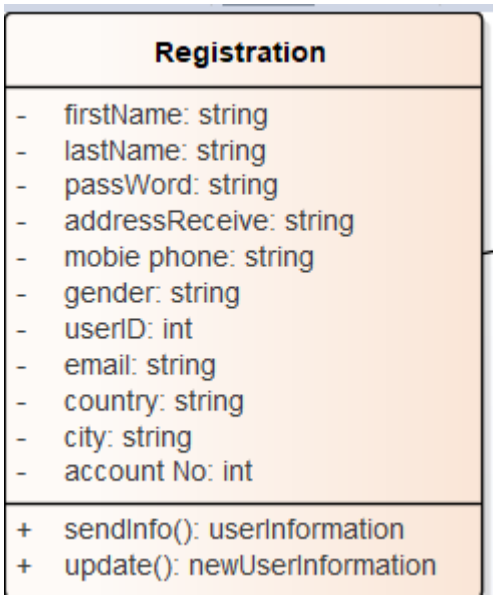
```
4 references
public class Payment
{
    2 references
    protected double amount { get; set; }
    1 reference
    protected string expireDate { get; set; }
    2 references
    protected string typePayment { get; set; }

    2 references
    public Payment(double amount, string expireDate, string typePayment)
    {
        this.amount = amount;
        this.expireDate = expireDate;
        this.typePayment = typePayment;
    }

    0 references
    public double getAmount()
    {
        return this.amount;
    }

    1 reference
    public string getExpireDate()
    {
        return this.getExpireDate();
    }

    0 references
    public string getTypePayment()
    {
        return this.typePayment;
    }
}
```

```
0 references
public class OfflinePayment : Payment
{
    0 references
    public OfflinePayment(double amount, string expireDate) : base(amount, expireDate, "Offline payment")
    {

    }
}

0 references
public class CreditCardPayment : Payment
{
    0 references
    public CreditCardPayment(double amount, string expireDate) : base(amount, expireDate, "Credit Card")
    {

    }
}
```

```
public static void Run()
{
    CreditCardPayment card = new CreditCardPayment(200, "23/5/2021");
    Console.WriteLine(card.getTypePayment());
    Console.WriteLine(card.getAmount());
    Console.WriteLine(card.getExpireDate());
}
```

4. Adapter

Name of function: Registration



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:

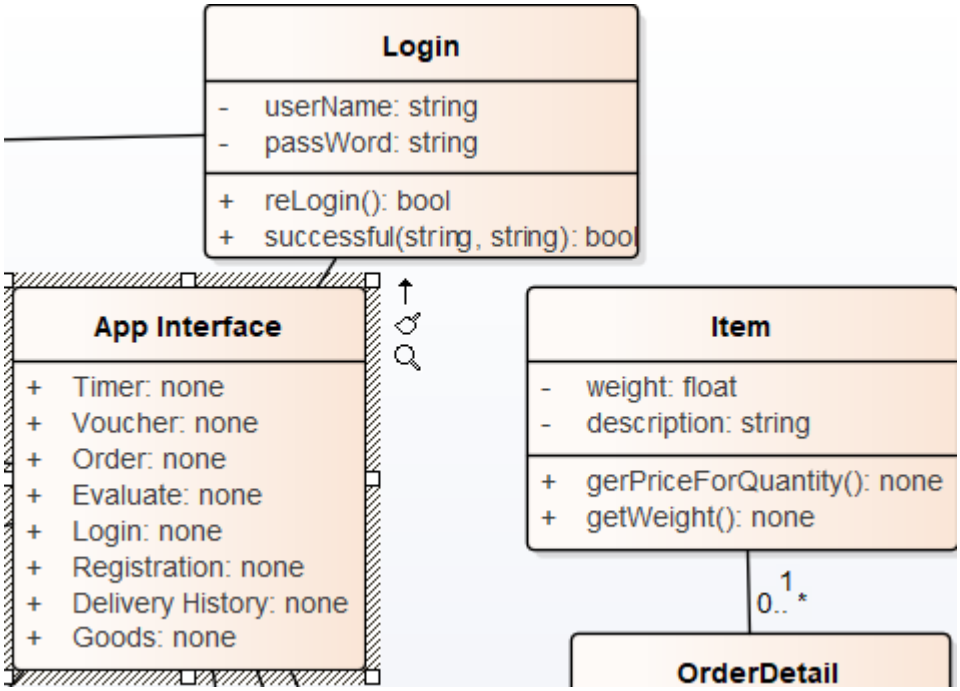
```
public interface ITarget
{
    1 reference
    public string SendInfo();
    0 references
    public string update();
}
```

```
class Registration : ITarget
{
    1 reference
    private User _info;

    1 reference
    public Registration(User info)
    {
        this._info = info;
    }
    1 reference
    public string SendInfo()
    {
        return "Sending successful to database";
    }

    0 references
    public string update()
    {
        return "Updated successfull into database";
    }
}
```

```
public class Adapter
{
    1 reference
    public static void Run()
    {
        User info = new User() {firstName = "Son", lastName = "Dang", address = "23 kp1"};
        ITarget target = new Registration(info);

        Console.WriteLine(target.SendInfo());
    }
}
```

5. Bridge

Name of function: Decentralization



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.

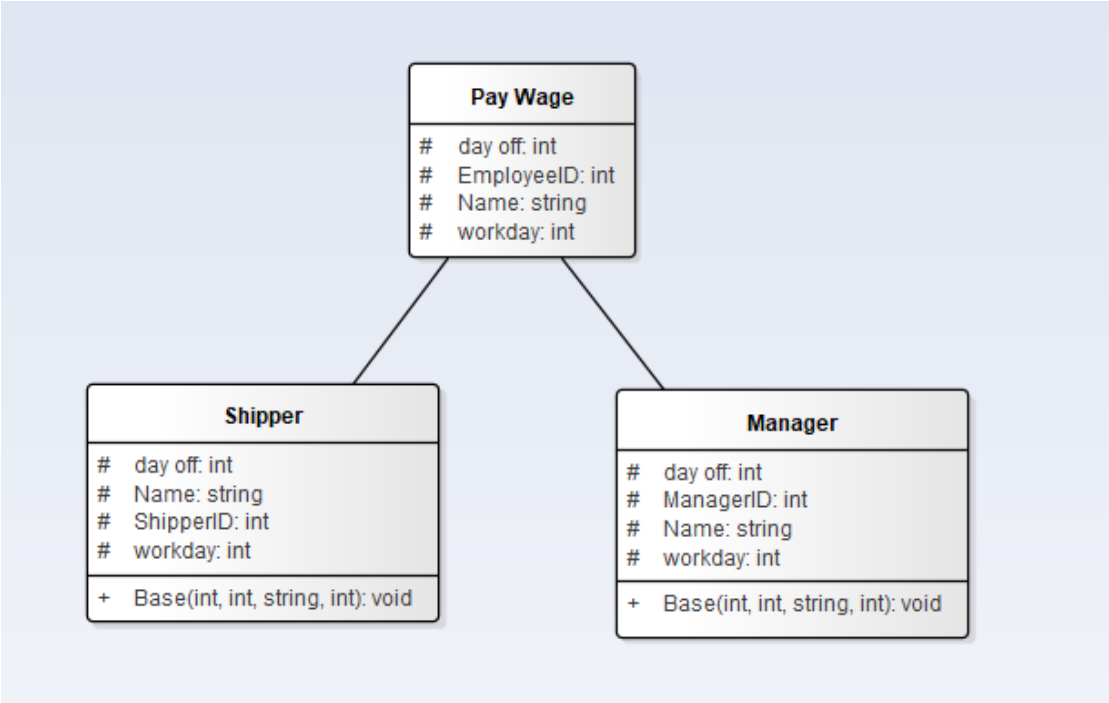
C# Code:

```
5 references
public interface IIImplementation
{
    1 reference
    string DecentralizationPosition();
}

1 reference
public class UserImplementationLogin : IIImplementation
{
    1 reference
    public string DecentralizationPosition()
    {
        return "this is screen for user";
    }
}

1 reference
public class ManagerImplementationLogin : IIImplementation
{
    1 reference
    public string DecentralizationPosition()
    {
        return "this is screen for manager";
    }
}
```

```
public class Abstraction
{
    2 references
    protected IIImplementation _implementation;
    3 references
    public Abstraction(IIImplementation implementation)
    {
        this._implementation = implementation;
    }

    2 references
    public virtual string Decentralization()
    {
        return $"Abstract: Base operation with: {_implementation.DecentralizationPosition()}";
    }
}

0 references
public class ExtendedAbstraction : Abstraction
{
    0 references
    public ExtendedAbstraction(IIImplementation implementation) : base(implementation)
    {

    }

    2 references
    public override string Decentralization()
    {
        return $"ExtendedAbstraction: Extended operation with:\n {base.Decentralization()}";
    }
}

2 references
public class Client
{
    2 references
    public void ClientCode(Abstraction abstraction)
    {
        Console.WriteLine(abstraction.Decentralization());
    }
}

public class Bridge
{
    1 reference
    public static void Run()
    {
        Client client = new Client();
        Abstraction abstraction = new Abstraction(new UserImplementationLogin());
        client.ClientCode(abstraction);

        abstraction = new Abstraction(new ManagerImplementationLogin());
        client.ClientCode(abstraction);
    }
}
```

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();
    }
}
```