**Принцип подстановки Лисков** (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 году, выглядело следующим образом:

*Если для каждого объекта o1 типа S существует объект o2 типа T, такой, что для любой программы P, определенной в терминах T, поведение P не изменяется при замене o2 на o1, то S является подтипом T.*

То есть иными словами класс S может считаться подклассом T, если замена объектов T на объекты S не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так:

**Должна быть возможность вместо базового типа подставить любой его подтип**.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

Проблему, с который связан принцип Лисков, наглядно можно продемонстрировать на примере двух классов Прямоугольника и Квадрата. Пусть они будут выглядеть следующим образом:

class Rectangle

{

    public virtual int Width { get; set; }

    public virtual int Height { get; set; }

    public int GetArea()

    {

        return Width \* Height;

    }

}

class Square : Rectangle

{

    public override int Width

    {

        get

        {

            return base.Width;

        }

        set

        {

            base.Width = value;

            base.Height = value;

        }

    }

    public override int Height

    {

        get

        {

            return base.Height;

        }

        set

        {

            base.Height = value;

            base.Width = value;

        }

    }

}

Как правило, квадрат представляют как частный случай прямоугольника - те же прямые углы, четыре стороны, только ширина обязательно равна высоте. Поэтому в классе Квадрат у одного свойства устанавливаются сразу и ширина, и высота:

set

{

    base.Height = value;

    base.Width = value;

}

На первый взгляд вроде все правильно, классы предельно простые, всего два свойства, и, казалось бы, сложно где-то ошибиться. Однако представим ситуацию, что в главной программе у нас следующий код:

class Program

{

    static void Main(string[] args)

    {

        Rectangle rect = new Square();

        TestRectangleArea(rect);

        Console.Read();

    }

    public static void TestRectangleArea(Rectangle rect)

    {

        rect.Height = 5;

        rect.Width = 10;

        if (rect.GetArea() != 50)

            throw new Exception("Некорректная площадь!");

    }

}

С точки зрения прямоугольника метод TestRectangleArea выглядит нормально, но не с точки зрения квадрата. Мы ожидаем, что переданный в метод TestRectangleArea объект будет вести себя как стандартный прямоугольник. Однако квадрат, будучи в иерархии наследования прямоугольником, все же ведет себя не как прямоугольник. В итоге программа вывалится в ошибку.

Иногда для выхода из подобных ситуаций прибегают к специальному хаку, который заключается в проверке объекта на соответствие типам:

public static void TestRectangleArea(Rectangle rect)

{

    if(rect is Square)

    {

        rect.Height = 5;

        if (rect.GetArea() != 25)

            throw new Exception("Неправильная площадь!");

    }

    else if(rect is Rectangle)

    {

        rect.Height = 5;

        rect.Width = 10;

        if (rect.GetArea() != 50)

            throw new Exception("Неправильная площадь!");

    }

}

Но такая проверка не отменяет того факта, что с архитектурой классов что-то не так. Более того такие решения только больше подчеркивают проблему несовершенства архитектуры. И проблема заключается в том, что производный класс Square не ведет себя как базовый класс Rectangle, и поэтому его не следует наследовать от данного базового класса. В этом и есть практический смысл принципа Лисков. Производный класс, который может делать меньше, чем базовый, обычно нельзя подставить вместо базового, и поэтому он нарушает принцип подстановки Лисков.

Существует несколько типов правил, которые должны быть соблюдены для выполнения принципа подстановки Лисков. Прежде всего это правила контракта.

Контракт представляет собой некоторый интерфейс базового класса, некоторые соглашения по его использованию, которым должен следовать класс-наследник. Контракт задает ряд ограничений или правил, и производный класс должен выполнять эти правила:

* Предусловия (Preconditions) не могут быть усилены в подклассе. Другими словами подклассы не должны создавать больше предусловий, чем это определено в базовом классе, для выполнения некоторого поведения

Предусловия представляют набор условий, необходимых для безошибочного выполнения метода. Например:

public virtual void SetCapital(int money)

{

    if (money < 0)

        throw new Exception("Нельзя положить на счет меньше 0");

    this.Capital = money;

}

Здесь условное выражение служит предусловием - без его выполнения не будут выполняться остальные действия, а метод завершится с ошибкой.

Причем объектом предусловий могут быть только общедоступные свойства или поля класса или параметры метода, как в данном случае. Приватное поле не может быть объектом для предусловия, так как оно не может быть установлено из вызывающего кода. Например, в следующем случае условное выражение не является предусловием:

private bool isValid = false

public virtual void SetCapital(int money)

{

    if (isValid == false)

        throw new Exception("Валидация не пройдена");

    this.Capital = money;

}

Теперь, допустим, есть два класса: Account (общий счет) и MicroAccount (мини-счет с ограничениями). И второй класс переопределяет метод SetCapital:

class Account

{

    public int Capital { get; protected set; }

    public virtual void SetCapital(int money)

    {

        if (money < 0)

            throw new Exception("Нельзя положить на счет меньше 0");

        this.Capital = money;

    }

}

class MicroAccount : Account

{

    public override void SetCapital(int money)

    {

        if (money < 0)

            throw new Exception("Нельзя положить на счет меньше 0");

        if (money > 100)

            throw new Exception("Нельзя положить на счет больше 100");

        this.Capital = money;

    }

}

В этом случае подкласс MicroAccount добавляет дополнительное предусловие, то есть усиливает его, что недопустимо. Поэтому в реальной задаче мы можем столкнуться с проблемой:

class Program

{

    static void Main(string[] args)

    {

        Account acc = new MicroAccount();

        InitializeAccount(acc);

        Console.Read();

    }

    public static void InitializeAccount(Account account)

    {

        account.SetCapital(200);

        Console.WriteLine(account.Capital);

    }

}

С точки зрения класса Account метод InitializeAccount() вполне является работоспособным. Однако при передаче в него объекта MicroAccount мы столкнемся с ошибкой. В итоге пинцип Лисков будет нарушен.

* Постусловия (Postconditions) не могут быть ослаблены в подклассе. То есть подклассы должны выполнять все постусловия, которые определены в базовом классе.

Постусловия проверяют состояние возвращаемого объекта на выходе из функции. Например:

public static float GetMedium(float[] numbers)

{

    if (numbers.Length == 0)

        throw new Exception("длина массива равна нулю");

    float result = numbers.Sum() / numbers.Length;

    if(result <0)

        throw new Exception("Результат меньше нуля");

    return result;

}

Второе условное выражение здесь является постусловием.

Рассмотрим пример нарушения принципа Лисков при ослаблении постусловия:

class Account

{

    public virtual decimal GetInterest(decimal sum,  int month, int rate)

    {

        // предусловие

        if (sum < 0 || month >12 || month <1 || rate <0)

            throw new Exception("Некорректные данные");

        decimal result = sum;

        for (int i = 0; i < month; i++)

            result += result \* rate  / 100;

        // постусловие

        if (sum >= 1000)

            result += 100; // добавляем бонус

        return result;

    }

}

class MicroAccount : Account

{

    public override decimal GetInterest(decimal sum, int month, int rate)

    {

        if (sum < 0 || month > 12 || month < 1 || rate < 0)

            throw new Exception("Некорректные данные");

        decimal result = sum;

        for (int i = 0; i < month; i++)

            result += result \* rate /100;

        return result;

    }

}

В качестве постусловия в классе Account используется начисление бонусов в 100 единиц к финальной сумме, если начальная сумма от 1000 и более. В классе MicroAccount это условие не используется.

Теперь посмотрим, с какой проблемой мы можем столкнуться с данными классами:

class Program

{

    public static void CalculateInterest(Account account)

    {

        decimal sum = account.GetInterest(1000, 1, 10); // 1000 + 1000 \* 10 / 100 + 100 (бонус)

        if(sum!=1200) // ожидаем 1200

        {

            throw new Exception("Неожиданная сумма при вычислениях");

        }

    }

    static void Main(string[] args)

    {

        Account acc = new MicroAccount();

        CalculateInterest(acc); // получаем 1100 без бонуса 100

        Console.Read();

    }

}

Исходя из логики класса Account, в методе CalculateInterest мы ожидаем получить в качестве результата числа 1200. Однако логика класса MicroAccount показывает другой результат. В итоге мы приходим к нарушению принципа Лисков, хотя формально мы просто применили стандартные принципы ООП - полиморфизм и наследование.

* Инварианты (Invariants) — все условия базового класса - также должны быть сохранены и в подклассе

Инварианты - это некоторые условия, которые остаются истинными на протяжении всей жизни объекта. Как правило, инварианты передают внутреннее состояние объекта. Например:

class User

{

    protected int age;

    public User(int age)

    {

        if(age<0)

            throw new Exception("возраст меньше нуля");

        this.age = age;

    }

    public int Age

    {

        get { return age; }

        set

        {

            if (value < 0)

                throw new Exception("возраст меньше нуля");

            this.age = value;

        }

    }

}

Поле age выступает инвариантом. И поскольку его установка возможна только через конструктор или свойство, то в любом случае выполнение предусловия и в конструкторе, и в свойстве гарантирует, что возраст не будет меньше 0. И данное объектоятельство сохранит свою истинность на протяжении всей жизни объекта User.

Теперь рассмотрим, как здесь может быть нарушен принцип Лисков. Пусть у нас будут следующие два класса:

class Account

{

    protected int capital;

    public Account(int sum)

    {

        if(sum<100)

            throw new Exception("Некорректная сумма");

        this.capital = sum;

    }

    public virtual int Capital

    {

        get { return capital; }

        set

        {

            if (value < 100)

                throw new Exception("Некорректная сумма");

            capital = value;

        }

    }

}

class MicroAccount : Account

{

    public MicroAccount(int sum) : base(sum)

    {

    }

    public override int Capital

    {

        get { return capital; }

        set

        {

            capital = value;

        }

    }

}

С точки зрения класса Account поле не может быть меньше 100, и в обоих случаях, где идет присвоение - в конструкторе и свойстве это гарантируется. А вот производный класс MicroAccount, переопределяя свойство Capital, этого уже не гарантирует. Поэтому инвариант класса Account нарушается.

Во всех трех вышеперечисленных случаях проблема решается в общем случае с помощью абстрагирования и выделения общего функционала, который уже наследуют классы Account и MicroAccount. То есть не один из них наследуется от другого, а оба они наследуются от одного общего класса.

Таким образом, принцип подстановки Лисков заставляет задуматься над правильностью построения иерархий классов и применения полиморфизма, позволяя уйти от ложных иерархий наследования и делая всю систему классом более стройной и непротиворечивой.