Инкапсуляция, абстракция, наследование, полиморфизм, SOLID

Будет ~~больно~~ весело

Инкапсуляция

Инкапсуляция в объектно-ориентированном программировании (ООП) - принцип, который объединяет данные и методы, работающие с ними, в единое целое.

Суть: все данные и функции, необходимые для работы объекта, сосредоточены внутри него самого. Объект сам управляет своим состоянием и своими действиями. Внешнему коду доступны только те элементы объекта, которые явно определены как открытые — обычно это отдельные методы или атрибуты. Если данные объекта нужно изменить или получить, это делается через специальные методы, прямой доступ к внутренним данным ограничивается.

Инкапсуляция (будь проще)

Принцип "чёрного ящика". Мы жмем на педаль газа, и нам нет разницы, как там работает инжектор (и прочие автомобильные штучки). Все сложности спрятаны внутри, наружу торчит только простой интерфейс (педали и руль). Защита от самого себя из будущего, когда ты забыл как устроен класс.

Абстракция

Абстракция в объектно-ориентированном программировании (ООП) - придание объекту характеристик, которые отличают его от всех объектов, чётко определяя его концептуальные границы.

Абстракция позволяет **выделить главные и наиболее значимые свойства предмета** и отбросить второстепенные характеристики и реализацию.

Абстракция (будь еще проще)

Принцип «мне неинтересно, как ты это делаешь, просто сделай». Мы создаём контракт (abstract class Shape), который говорит: «Любая фигура обязана уметь рисоваться и считать площадь».

А как именно - решает каждый конкретный наследник. Нельзя создать «просто фигуру», которая ничего не умеет.

Наследование

Наследование в объектно-ориентированном программировании (ООП) - принцип, который позволяет создавать иерархии классов. Дочерние классы могут наследовать данные и методы от родительских классов.

При этом дочерние классы также могут расширять функциональность родительских - иметь собственные поля и методы.

Наследование помогает **избежать лишнего дублирования кода.**

Наследование (чуточку проще)

Принцип «не изобретай велосипед». Сделал «Транспортное средство», а потом на его основе «Машину» и «Мотоцикл», добавив им уникальные детали, но не переписывая с нуля про колёса и двигатель.

Полиморфизм

Полиморфизм в объектно-ориентированном программировании (ООП) - возможность объектов разных классов использовать один и тот же интерфейс (методы с одинаковым названием), но при этом реализовывать их по-разному.

Слово «полиморфизм» происходит из греческого языка и буквально означает «много форм».

Полиморфизм (да кто такой этот проще)

Одна кнопка - разные действия. Убийца if-else. Ты говоришь «Рисуйся» (shape.Draw()) любому объекту, и разницы нет, круг это или квадрат - он сам знает, как себя нарисовать.

SOLID

SOLID (сокр. от англ. single responsibility, open–closed, Liskov substitution, interface segregation и dependency inversion) в программировании — мнемонический акроним, введённый Майклом Фэзерсом (Michael Feathers) для первых пяти принципов, названных Робертом Мартином в начале 2000-х, которые означали **5 основных принципов объектно-ориентированных проектирования и программирования**.

Код - это не слова, это решения проблем. А эти термины - названия для самых лучших решений, которые придумали до вас

S – SRP (Single Responsibility Principle / Принцип единственной ответственности)

Проблема, которую решает: Класс-швейцарский-нож, который ломается от любого чиха. Вся логика написана в одном классе.

Суть: Один класс - одна, задача. Класс User хранит данные юзера. Класс UserDbSaver сохраняет его в базу. Не следует мешать все в одну кучу.

O – OCP (Open/Closed Principle / Принцип открытости/закрытости)

Проблема, которую решает: Страх трогать старый код, потому что всё может сломаться.

Суть: Пишем код, как Лего. Мы должны иметь возможность прицепить новую деталь (расширить функционал), но не должен для этого ломать или переплавлять старые детали (изменять рабочий код).

L – LSP (Liskov Substitution Principle / Принцип подстановки Барбары Лисков)

Проблема, которую решает: Наследники, которые притворяются родителями, но ведут себя неадекватно.

Суть: Если ты меняешь родителя на его потомка, программа не должна этого заметить и упасть. Потомок должен полностью соответствовать «контракту» поведения родителя.

I – ISP (Interface Segregation Principle / Принцип разделения интерфейса)

Проблема, которую решает: Жирные, бестолковые интерфейсы, которые заставляют реализовывать ненужные методы. Интерфейсы должны быть маленькими и специфическими, а не обширными и обобщёнными.

Суть: Лучше много маленьких и конкретных интерфейсов (IClickable, IDrawable), чем один большой (IAllMightyGodObject). Не заставляй класс реализовывать то, что ему не нужно.

D – DIP (Dependency Inversion Principle / Принцип инверсии зависимостей)

Проблема, которую решает: Жестко спаянный код, который невозможно тестировать и менять. Модули высокого уровня должны зависеть от абстракций, а не от модулей низкого уровня.

Суть: Зависеть от абстракций (интерфейсов, абстрактных классов), а не от конкретики. Твой модуль не должен знать, что он работает именно с MySqlDatabase. Он должен знать, что работает с чем-то, что реализует IDatabase.

Это все

Спасибо за внимание

От спагетти-кода до SOLID'ного блюда

- Общая идея: Мы пишем убогий графический редактор. На каждом шаге сталкиваемся с sheet-кодом и рефакторим его, используя принципы ООП и SOLID, чтобы показать, как они могут использоваться в реальной жизни.

Начало

```
Ссылок: 6
public class Shape
{
    Ссылок: 2
    public Shape(string type = "None")
    {
        this.type = type;
    }
    private string type;
    Ссылок: 2
    public string Type
    {
        get
        {
            return type;
        }
    }

    Ссылок: 0
    public int X { get; set; }
    Ссылок: 0
    public int Y { get; set; }

    // Данные для ВСЕХ фигур в одном месте. У круга нет ширины, но поле есть.
    Ссылок: 0
    public int Radius { get; set; }
    Ссылок: 0
    public int Width { get; set; }
    Ссылок: 0
    public int Height { get; set; }
}
```

Начало

```
// ГЛАВНЫЙ ВРАГ. Этот метод — позор нашего проекта.
Ссылок: 0
public void DrawAllShapes(List<Shape> shapes)
{
    foreach (var shape in shapes)
    {
        if (shape.Type == "rectangle")
        {
            // Логика отрисовки прямоугольника...
            Console.WriteLine($"Рисую прямоугольник в {shape.X},{shape.Y} размером {shape.Width}x{shape.Height}");
        }
        else if (shape.Type == "circle")
        {
            // Логика отрисовки круга...
            Console.WriteLine($"Рисую круг в {shape.X},{shape.Y} радиусом {shape.Radius}");
        }
        // А ТЕПЕРЬ ДОБАВЬ СЮДА ТРЕУГОЛЬНИК, УМНИК!
        // А ПОТОМ ЗВЕЗДУ! И ЭЛЛИПС! УДАЧИ НЕ ОШИБИТЬСЯ,
        // СЛУЧАЙНО УДАЛИВ СКОБКУ В БЛОКЕ ДЛЯ ПРЯМОУГОЛЬНИКА.
    }
}
```

Начало

```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Shape(type: "Circle");
    Shape Circle_another = new Shape(type: "Cricle");
    Shape Rectangle = new Shape(type: "Rectangle");

    List<Shape> shapes = new List<Shape>();
    shapes.Add(Circle);
    shapes.Add(Circle_another);
    shapes.Add(Rectangle);
}
```

Инкапсуляция

- Проблема: В коде выше данные (тип, координаты) и логика (отрисовка) разделены. Любой *художник* может залезть и поменять shape.Type с «Circle» на «Labubu» и всё сломать.
- Решение: Создаём классы Circle и Rectangle. Прячем данные (координаты, радиус) в private поля. Доступ - через public свойства и методы.

Инкапсуляция

```
// Класс отвечает ТОЛЬКО за круг. Никаких Width/Height.
Ссылок: 1
public class Circle
{
    // Данные спрятаны. Их нельзя просто так изменить извне.
    private int _x, _y, _radius;

    Ссылок: 0
    public Circle(int x, int y, int radius)
    {
        _x = x; _y = y; _radius = radius;
    }

    // Вместо прямого доступа к полям - метод. Он часть этого класса.
    Ссылок: 0
    public void Draw()
    {
        Console.WriteLine($"Рисую круг в {_x},{_y} радиусом {_radius}");
    }
}
```

Инкапсуляция

```
// Класс отвечает ТОЛЬКО за прямоугольник. Никакого Radius.
Ссылок: 1
public class Rectangle
{
    private int _x, _y, _width, _height;

    Ссылок: 0
    public Rectangle(int x, int y, int width, int height)
    {
        _x = x; _y = y; _width = width; _height = height;
    }

    Ссылок: 0
    public void Draw()
    {
        Console.WriteLine($"Рисую прямоугольник в {_x},{_y} размером {_width}x{_height}");
    }
}
```

Инкапсуляция

```
Ссылок: 0
static void Main(string[] args)
{
    object Circle = new Circle(0,0, radius: 15);
    object Rectangle = new Rectangle(0,0, width: 10, height: 15);

    List<object> shapes = new List<object>();
    shapes.Add(Circle);
    shapes.Add(Rectangle);
}
```

Инкапсуляция

```
// РЕДАКТОР СТАЛ ЕЩЁ УРОДЛИВЕЕ, НО ЭТО НЕОБХОДИМЫЙ ШАГ!
Ссылок: 0
public class GraphicEditor
{
    // Мы больше не можем использовать общий список, и это вскрывает новую проблему.
    // Теперь мы вынуждены проверять КОНКРЕТНЫЙ ТИП.
    Ссылок: 0
    public void DrawAllShapes(List<object> shapes) // Пришлось использовать object!
    {
        foreach (var shape :object in shapes)
        {
            if (shape is Rectangle rect)
                rect.Draw();
            else if (shape is Circle circle)
                circle.Draw();
        }
    }
}
```

Наследование

- Проблема: в наших классах Circle и Rectangle есть общие свойства: координаты, цвет, метод Draw(). Мы дублируем код.
- Решение: создаем базовый класс Shape. Выносим в него всё общее. Circle и Rectangle теперь наследуются от Shape.
- Суть: «Мы убрали дублирование. Теперь, если захотим добавить всем фигурам, например, прозрачность, мы сделаем это в одном, месте – класс Shape»

Наследование

```
Ссылок: 2
public class Shape
{
    // protected - видно нам и нашим наследникам.
    protected int _x, _y;
}
```

Наследование

```
// Circle теперь НАСЛЕДНИК Shape. Он получает _x и _y "по наследству".
Ссылок: 3
public class Circle : Shape
{
    private int _radius;

    Ссылок: 1
    public Circle(int x, int y, int radius)
    {
        _x = x; // Поле из базового класса!
        _y = y; // И это тоже!
        _radius = radius;
    }
    Ссылок: 1
    public void Draw() => Console.WriteLine($"Рисую круг в {_x},{_y} радиусом {_radius}");
}
```

Наследование

```
// Rectangle - тоже наследник.
Ссылок: 3
public class Rectangle : Shape
{
    private int _width, _height;

    Ссылок: 1
    public Rectangle(int x, int y, int width, int height)
    {
        _x = x;
        _y = y;
        _width = width;
        _height = height;
    }
    Ссылок: 1
    public void Draw() => Console.WriteLine($"Рисую прямоугольник в {_x},{_y} размером {_width}x{_height}");
}
```

Наследование

```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Circle(0,0, radius: 15);
    Shape Rectangle = new Rectangle(0,0, width: 10, height: 15);

    List<Shape> shapes = new List<Shape>();

    shapes.Add(Circle);
    shapes.Add(Rectangle);
}
```

Наследование

```
// Редактор стал чуть чище, но главная проблема осталась
Ссылок: 0
public class GraphicEditor
{
    // Теперь мы можем работать с общим списком List<Shape>! Это уже победа.
    Ссылок: 0
    public void DrawAllShapes(List<Shape> shapes)
    {
        foreach (var shape in shapes)
        {
            // НО ЭТОТ IF-ELSE ВСЁ ЕЩЁ ЗДЕСЬ!
            if (shape is Rectangle rect)
                rect.Draw();
            else if (shape is Circle circle)
                circle.Draw();
        }
    }
}
```

Полиморфизм

- Проблема: Наш *фантастический-if-else* всё ещё жив.
- Решение: В базовый класс Shape добавляем public virtual void Draw(). В наследниках Circle и Rectangle делаем public override void Draw().

Полиморфизм

```
Ссылок: 7
public class Shape
{
    protected int _x, _y;

    // Виртуальный метод, который могут переопределить наследники.
    Ссылок: 2
    public virtual void Draw()
    {

    }
}
```

```
Ссылок: 1
public override void Draw() => Console.Writ
```

There is no suitable method for override

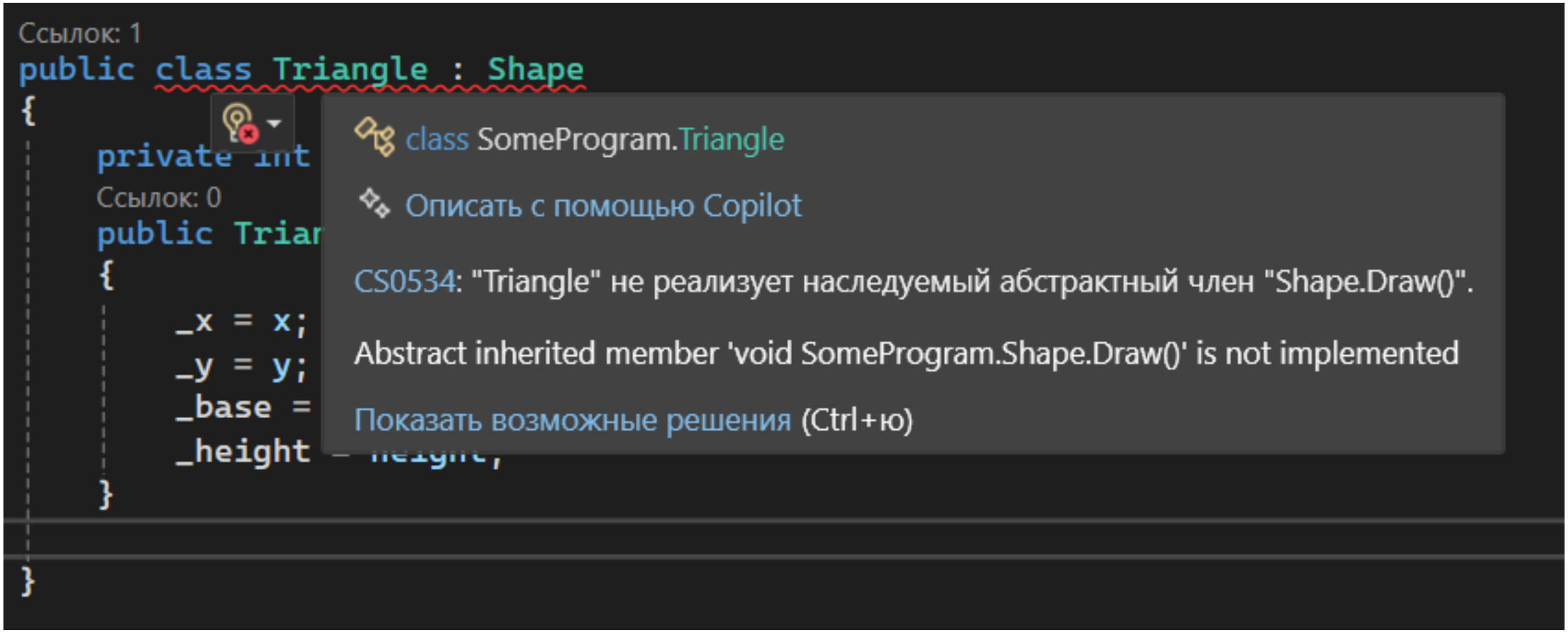
Полиморфизм

```
Ссылок: 3
public class Circle : Shape
{
    private int _radius;

    Ссылок: 1
    public Circle(int x, int y, int radius)
    {
        _x = x; // Поле из базового класса!
        _y = y; // И это тоже!
        _radius = radius;
    }
    // Переопределяем метод Draw для рисования круга.
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую круг в {_x},{_y} радиусом {_radius}");
}
```

Полиморфизм

```
Ссылок: 3
public class Rectangle : Shape
{
    private int _width, _height;

    Ссылок: 1
    public Rectangle(int x, int y, int width, int height)
    {
        _x = x;
        _y = y;
        _width = width;
        _height = height;
    }
    // Переопределяем метод Draw для рисования прямоугольника.
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую прямоугольник в {_x},{_y} размером {_width}x{_height}");
}
```

Полиморфизм



Полиморфизм

```
Ссылок: 1
public class Triangle : Shape
{
    private int _base, _height;
    Ссылок: 0
    public Triangle(int x, int y, int bas, int height)
    {
        _x = x;
        _y = y;
        _base = bas;
        _height = height;
    }
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую треугольник в {_x},{_y} основанием {_base} и высотой {_height}");
}
```

Полиморфизм

```
// ПОСМОТРИТЕ НА ЭТО. ЭТО КОД, КОТОРЫЙ НЕ СТЫДНО ПОКАЗАТЬ ДРУЗЬЯМ
Ссылок: 0
public class GraphicEditor
{
    Ссылок: 0
    public void DrawAllShapes(List<Shape> shapes)
    {
        foreach (var shape in shapes)
        {
            shape.Draw(); // Полиморфный вызов метода Draw в 1 строку
        }
    }
}
```

Полиморфизм

```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Circle(0,0, radius: 15);
    Shape Rectangle = new Rectangle(0,0, width: 10, height: 15);

    Shape shape = new Shape(); // Откуда ты это сказал???

    List<Shape> shapes = new List<Shape>();

    shapes.Add(Circle);
    shapes.Add(Rectangle);
}
```

Абстракция

Проблема №2. «А что такое *просто фигура*?»

- Проблема: Сейчас можно создать объект new Shape(). А как его рисовать? У него нет реализации метода Draw().
- Решение: Делаем класс Shape и его метод Draw() абстрактными.
- Суть: Мы создали контракт. Теперь нельзя создать бессмысленную 'просто фигуру'. Хочешь быть фигурой? Будь добр, реализуй метод Draw().

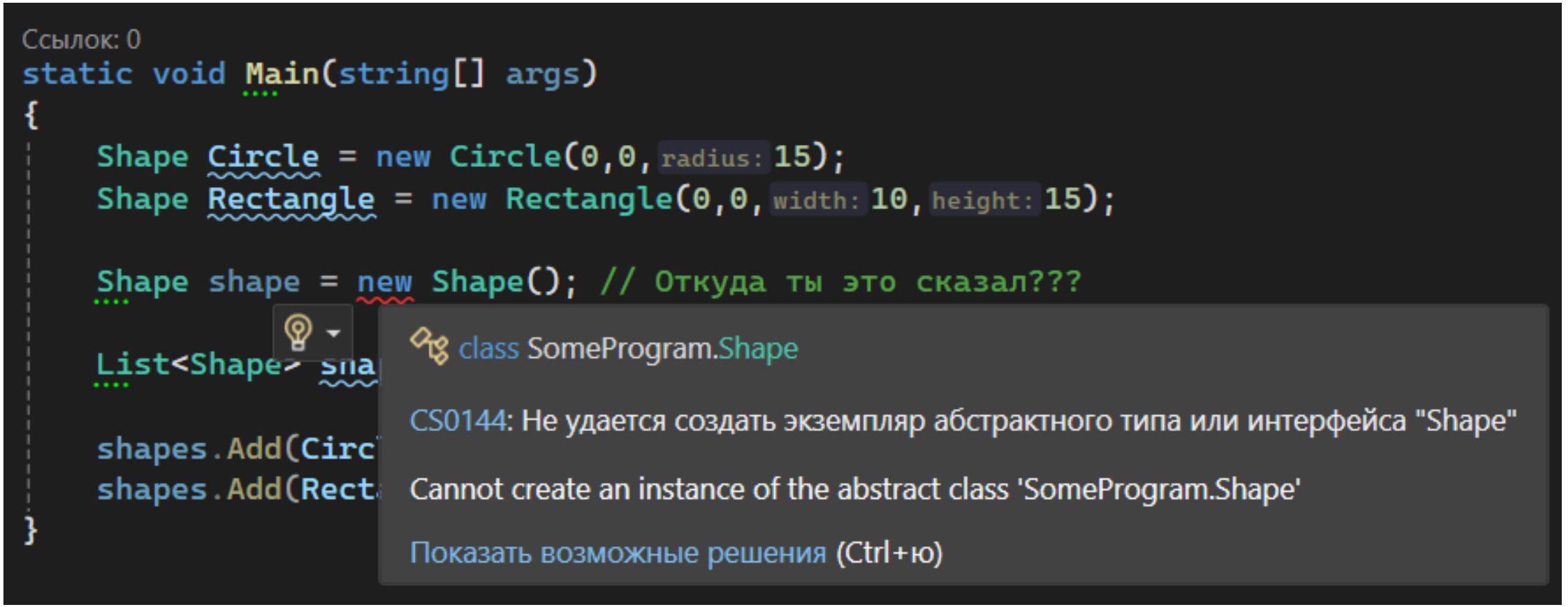
Абстракция

```
Ссылок: 7
public abstract class Shape
{
    protected int _x, _y;

    // Контракт: любой, кто назовётся Фигурой, ОБЯЗАН иметь метод Draw.
    Ссылок: 3
    public abstract void Draw();

}
```

```
Ссылок: 3
public abstract void Draw()
{}
```

Abstract method cannot declare a body

Абстракция


Ссылок: 0
static void Main(string[] args)
{
Shape Circle = new Circle(0,0, radius: 15);
Shape Rectangle = new Rectangle(0,0, width: 10, height: 15);
Shape shape = new Shape(); // Откуда ты это сказал???
List<Shape>
shapes.Add(Circ
shapes.Add(Rect
}
class SomeProgram.Shape
CS0144: Не удается создать экземпляр абстрактного типа или интерфейса "Shape"
Cannot create an instance of the abstract class 'SomeProgram.Shape'
Показать возможные решения (Ctrl+ю)

S.R.P. - Принцип единственной ответственности

- Проблема: Нам говорят: Теперь фигуры надо сохранять в файл. Возможная первая идея - засунуть метод SaveToFile() прямо в класс Shape.
- Почему это плохо: Класс Shape и его дети отвечают за геометрию и отрисовку. Сохранение - это совершенно другая задача.
- Решение: Создаём отдельный класс ShapePersistence, который занимается только сохранением и загрузкой.
- *Суть: "Не будь швейцарским ножом-мутантом. Делай что-то одно, но делай это хорошо".*

S.R.P. - Принцип единственной ответственности

```
// НОВЫЙ КЛАСС! Его единственная работа - сохранять фигуры.
Ссылок: 0
public class ShapePersistence
{
    Ссылок: 0
    public void SaveToFile(string filePath, List<Shape> shapes)
    {
        Console.WriteLine($"Сохраняю {shapes.Count} фигур в файл {filePath}...");
    }
}
```

O.C.P. — Принцип открытости/закрытости

- Проблема: Нам говорят: "Добавьте треугольник".
- Как мы это делаем: Мы просто создаём новый класс Triangle : Shape. ВСЁ. Мы не трогаем ни строчки в существующем коде отрисовки (foreach... shape.Draw()).
- *Суть: "Наш код готов к появлению нового, но защищён от изменений в старом. Мы добавляем функционал, не рискуя сломать то, что уже работает".*

O.C.P. — Принцип открытости/закрытости

```
Ссылок: 1
public class Triangle : Shape
{
    private int _base, _height;
    Ссылок: 0
    public Triangle(int x, int y, int bas, int height)
    {
        _x = x;
        _y = y;
        _base = bas;
        _height = height;
    }
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую треугольник в {_x},{_y} основанием {_base} и высотой {_height}");
}
```

O.C.P. — Принцип открытости/закрытости

```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Circle(0,0, radius: 15);
    Shape Rectangle = new Rectangle(0,0, width: 10, height: 15);
    Shape Triangle = new Triangle(0, 0, bas: 10, height: 15);

    List<Shape> shapes = new List<Shape>();

    shapes.Add(Circle);
    shapes.Add(Rectangle);
    shapes.Add(Triangle);

    GraphicEditor editor = new GraphicEditor();
    editor.DrawAllShapes(shapes);
}
```

```
Консоль отладки Microsoft Visual Studio
Рисую круг в 0,0 радиусом 15
Рисую прямоугольник в 0,0 размером 10x15
Рисую треугольник в 0,0 основанием 10 и высотой 15
```

O.C.P. — Принцип открытости/закрытости

```
// ПОСМОТРИТЕ НА ЭТО. ЭТО КОД, КОТОРЫЙ НЕ СТЫДНО ПОКАЗАТЬ ДРУЗЬЯМ
Ссылок: 2
public class GraphicEditor
{
    Ссылок: 1
    public void DrawAllShapes(List<Shape> shapes)
    {
        foreach (var shape in shapes)
        {
            shape.Draw(); // Полиморфный вызов метода Draw в 1 строку
        }
    }
}
```

L.S.P. — Принцип подстановки Лисков.

- Проблема (классика): Давайте сделаем Square наследником Rectangle.
- Почему это плохо. У Rectangle есть Width и Height. Если мы меняем square.Width, то должна меняться и Height, что нарушает поведение родителя. square - это не совсем rectangle.

L.S.P. — Принцип подстановки Лисков.

```
Ссылок: 4
public class Rectangle : Shape
{
    private int _width, _height;
    Ссылок: 3
    public virtual int Width { get => _width; set => _width = value; }
    Ссылок: 3
    public virtual int Height { get => _height; set => _height = value; }

    Ссылок: 2
    public Rectangle(int x, int y, int width, int height)
    {
        _x = x;
        _y = y;
        _width = width;
        _height = height;
    }
    // Переопределяем метод Draw для рисования прямоугольника.
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую прямоугольник в {_x},{_y} размером {_width}x{_height}");
}
```

L.S.P. — Принцип подстановки Лисков.

```
// "Квадрат - это прямоугольник", ага, щас.
Ссылок: 1
public class Square : Rectangle
{
    Ссылок: 0
    public Square(int x, int y, int width, int height) : base(x, y, width, height)
    {
    }

    Ссылок: 3
    public override int Width { set { base.Width = base.Height = value; } }
    Ссылок: 3
    public override int Height { set { base.Width = base.Height = value; } }
}
```

L.S.P. — Принцип подстановки Лисков.

```
Ссылок: 0
public class Tester
{
    // Безобидный метод, который сломается
    Ссылок: 0
    public void TestRectangleBehavior(Rectangle r)
    {
        r.Width = 5;
        r.Height = 10;
        // ЛЮБОЙ ЗДРАВОМЫСЛЯЩИЙ ЧЕЛОВЕК ОЖИДАЕТ, ЧТО ПЛОЩАДЬ БУДЕТ 50.
        if (r.Width * r.Height != 50)
        {
            // НО ЕСЛИ ПРИШЕЛ КВАДРАТ, ЗДЕСЬ БУДЕТ 100! (10*10)
            throw new Exception("Что у вас здесь происходит?! Контракт родителя нарушен!");
        }
    }
}
```

L.S.P. — Принцип подстановки Лисков.

```
// Квадрат - это тоже просто фигура. ОН НЕ НАСЛЕДНИК ПРЯМОУГОЛЬНИКА.
// Они братья, а не отец и сын.
Ссылок: 1
public class Square : Shape
{
    Ссылок: 2
    public int Side { get; set; }

    Ссылок: 0
    public Square(int x, int y, int side)
    {
        _x = x;
        _y = y;
        Side = side;
    }
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую квадрат в {_x},{_y} со стороной {Side}");
}
// Теперь ты физически не сможешь передать Square в метод, который ждёт Rectangle.
// Проблема решена на уровне проектирования, а не ночных дебагов.
```

I.S.P. - Принцип разделения интерфейса

- Проблема: Нам говорят: "Некоторые фигуры должны быть интерактивными, например, кнопка". Мы добавляем в Shape методы OnClick(), OnHover().
- Почему это снова мимо: Теперь у обычного круга или квадрата болтаются бессмысленные пустые методы OnClick.
- Решение: Создаём тонкие интерфейсы: IDrawable, IClickable.
- class Button : IDrawable, IClickable { ... }
- class Circle : IDrawable { ... }
- *Суть: "Лучше много маленьких и конкретных контрактов, чем один жирный и бестолковый".*

I.S.P. - Принцип разделения интерфейса

```
// Мы захламляем базовый класс функционалом, который нужен не всем.

Ссылок: 11
public abstract class Shape
{
    protected int _x, _y;

    Ссылок: 5
    public abstract void Draw();
    Ссылок: 0
    public virtual void OnClick() { /* Пустая реализация по умолчанию */ }

}
```

I.S.P. - Принцип разделения интерфейса

```
// Обычный круг теперь тоже "кликабельный", хотя ему это не нужно.
Ссылок: 2
public class Circle : Shape
{
    private int _radius;

    Ссылок: 1
    public Circle(int x, int y, int radius)...
    Ссылок: 2
    public override void Draw() => Console.WriteLine($"Рисую круг в {_x},{_y} радиусом {_radius}");
    // Этот метод здесь - мусор.
    Ссылок: 1
    public override void OnClick() { }
}
```

I.S.P. - Принцип разделения интерфейса

```
// А вот кнопке это нужно.
Ссылок: 0
public class Button : Shape
{
    Ссылок: 2
    public override void Draw() { Console.WriteLine("Рисую кнопку"); }
    Ссылок: 1
    public override void OnClick() { Console.WriteLine("Кнопка нажата!"); }
}
```

I.S.P. - Принцип разделения интерфейса

```
// Маленький контракт ТОЛЬКО для рисования.
Ссылок: 1
public interface IDrawable { void Draw(); }
// Маленький контракт ТОЛЬКО для кликов.
Ссылок: 0
public interface IClickable { void OnClick(); }

Ссылок: 12
public abstract class Shape : IDrawable
{
    protected int _x, _y;

    Ссылок: 7
    public abstract void Draw();

}
```

I.S.P. - Принцип разделения интерфейса

```
// Круг - просто рисуемый. И ничего лишнего.
Ссылок: 2
public class Circle : Shape
{
    private int _radius;

    Ссылок: 1
    public Circle(int x, int y, int radius)...
    Ссылок: 3
    public override void Draw() => Console.WriteLine($"Рисую круг в {_x},{_y} радиусом {_radius}");
}
```

I.S.P. - Принцип разделения интерфейса

```
// А вот Кнопка — и рисуемая, и кликабельная!
Ссылок: 0
public class Button : Shape, IClickable
{
    Ссылок: 3
    public override void Draw() { Console.WriteLine("Рисую кнопку"); }
    Ссылок: 1
    public void OnClick() { Console.WriteLine("Кнопка нажата!"); }
}
```

D.I.P. - Принцип инверсии зависимостей.

- Проблема: Наш главный класс GraphicEditor создаёт экземпляры фигур прямо внутри себя (new Circle(), new Rectangle()). Он жестко зависит от конкретных классов. Завтра придет Triangle - придется лезть в код GraphicEditor.
- Решение: GraphicEditor должен зависеть от абстракции (Shape), а не от конкретики (Circle). Конкретные фигуры должны "внедряться" в него извне (Dependency Injection).
- Суть: "Модули верхнего уровня не должны зависеть от модулей нижнего. И те, и другие должны зависеть от абстракций. Говоря по-русски: твой главный модуль не должен знать, с какими конкретными пешками он работает, ему достаточно знать, что это 'фигуры'».

D.I.P. - Принцип инверсии зависимостей.

```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Circle(0, 0, radius: 15);
    Shape Rectangle = new Rectangle(0, 0, width: 10, height: 15);
    Shape Triangle = new Triangle(0, 0, bas: 10, height: 15);

    List<Shape> shapes = new List<Shape>();

    shapes.Add(Circle);
    shapes.Add(Rectangle);
    shapes.Add(Triangle);

    // Редактор САМ СОЗДАЕТ своего помощника. Он прибит к нему гвоздями.
    ShapePersistence persistence = new ShapePersistence();
    persistence.SaveToFile("myShapes.txt", shapes);
    // Как протестировать этот редактор без реального файла? Никак.
    // Как поменять способ сохранения на базу данных? Лезть и переписывать код редактора.
}
```

D.I.P. - Принцип инверсии зависимостей.

```
// НОВЫЙ КЛАСС! Его единственная работа – сохранять фигуры.
Ссылок: 2
public class ShapePersistence
{
    Ссылок: 1
    public void SaveToFile(string filePath, List<Shape> shapes)
    {
        Console.WriteLine($"Сохраняю {shapes.Count} фигур в файл {filePath}...");
    }
}
```

D.I.P. - Принцип инверсии зависимостей.

```
// D - Создаём АБСТРАКЦИЮ, контракт. "Мне нужен кто-то, кто умеет сохранять".
Ссылок: 2
public interface IShapePersistence
{
    Ссылок: 2
    void Save(string destination, List<Shape> shapes);
}

// Конкретная реализация контракта для файлов.
Ссылок: 0
public class FilePersistence : IShapePersistence
{
    Ссылок: 1
    public void Save(string filePath, List<Shape> shapes) { Console.WriteLine($"Сохранил в файл {filePath}"); }
}

// Ещё одна реализация. Например, для базы данных.
Ссылок: 0
public class DatabasePersistence : IShapePersistence
{
    Ссылок: 1
    public void Save(string connectionString, List<Shape> shapes) { Console.WriteLine($"Сохранил в файл {connectionString}"); }
}
```

D.I.P. - Принцип инверсии зависимостей.

```
// ПОСМОТРИТЕ НА ЭТО. ЭТО КОД, КОТОРЫЙ НЕ СТЫДНО ПОКАЗАТЬ ДРУЗЬЯМ
Ссылок: 0
public class GraphicEditor
{
    Ссылок: 0
    public void DrawAllShapes(List<Shape> shapes)
    {
        foreach (var shape in shapes)
        {
            shape.Draw(); // Полиморфный вызов метода Draw в 1 строку
        }
    }
}
```

D.I.P. - Принцип инверсии зависимостей.

```
// Редактор больше не знает о файлах или базах. Он знает только о КОНТРАКТЕ.
Ссылок: 3
public class GraphicEditor
{
    private readonly IShapePersistence _persistence;

    // Зависимость ВНЕДРЯЕТСЯ СНАРУЖИ. Он не создаёт её сам.
    Ссылок: 2
    public GraphicEditor(IShapePersistence persistence)
    {
        _persistence = persistence;
    }

    Ссылок: 2
    public void SaveAllShapes(string destination, List<Shape> shapes)
    {
        _persistence.Save(destination, shapes);
    }
}
```

D.I.P. - Принцип инверсии зависимостей.

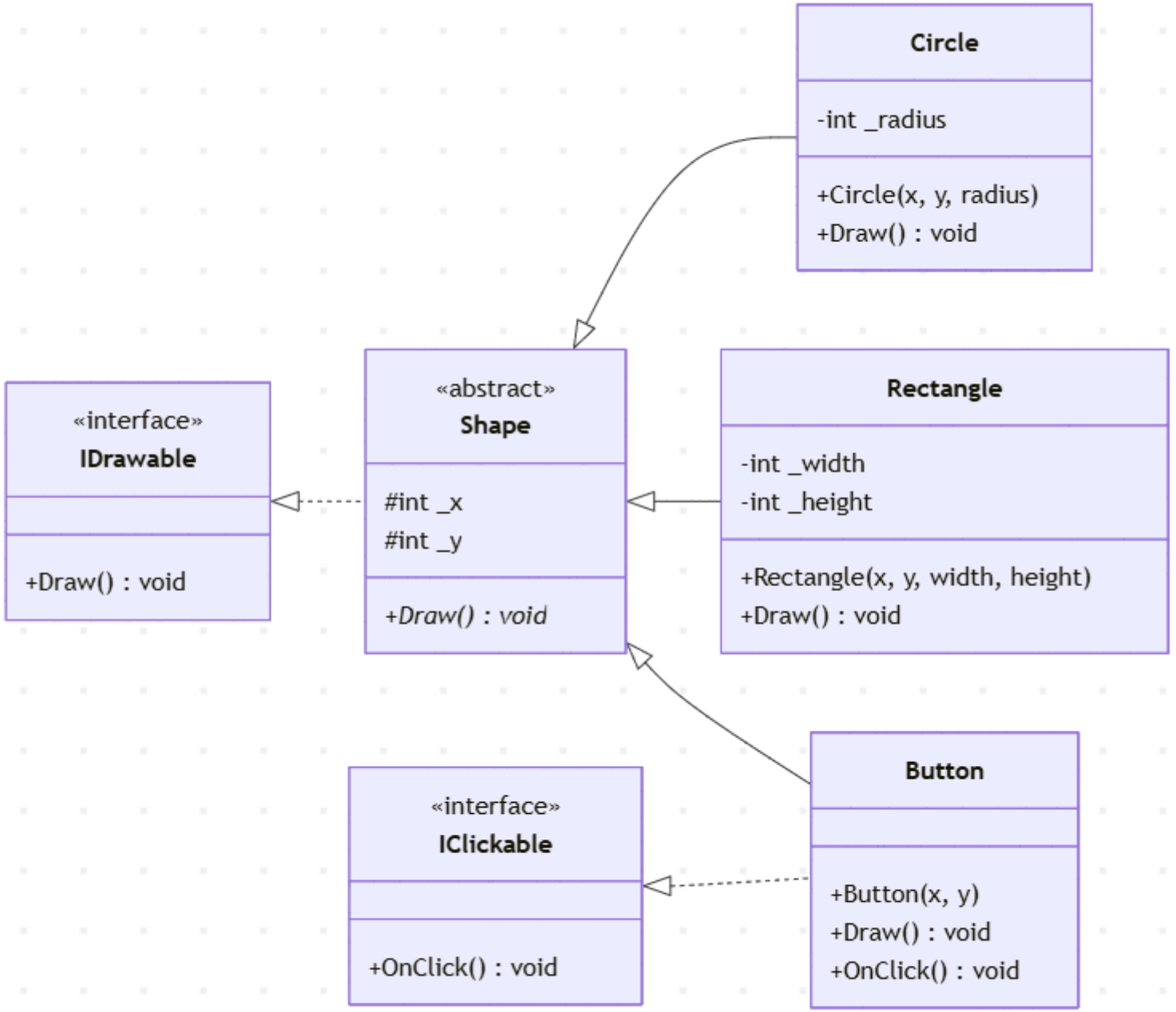
```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Circle(0, 0, radius: 15);
    Shape Rectangle = new Rectangle(0, 0, width: 10, height: 15);
    Shape Triangle = new Triangle(0, 0, bas: 10, height: 15);

    List<Shape> shapes = new List<Shape>();

    shapes.Add(Circle);
    shapes.Add(Rectangle);
    shapes.Add(Triangle);

    // Использование:
    // Хотим сохранять в файл?
    var fileEditor = new GraphicEditor(new FilePersistence());
    fileEditor.SaveAllShapes(destination: "myfile.txt", shapes);
    // Хотим сохранять в базу?
    var dbEditor = new GraphicEditor(new DatabasePersistence());
    dbEditor.SaveAllShapes(destination: "Server=.;Database=Shapes;Trusted_Connection=True;", shapes);
    // Код самого редактора НЕ ИЗМЕНИЛСЯ. Он стал управляемым извне.
}
```

D.I.P. - Принцип инверсии зависимостей.

```
Ссылок: 0
static void Main(string[] args)
{
    Shape Circle = new Circle(0, 0, radius: 15);
    Shape Rectangle = new Rectangle(0, 0, width: 10, height: 15);
    Shape Triangle = new Triangle(0, 0, bas: 10, height: 15);

    List<Shape> shapes = new List<Shape>();

    shapes.Add(Circle);
    shapes.Add(Rectangle);
    shapes.Add(Triangle);

    // Использование:
    // Хотим сохранять в файл?
    var editor = new GraphicEditor(new FilePersistence());
    editor.SaveAllShapes(destination: "myfile.txt", shapes);
    // Хотим сохранять в базу?
    editor = new GraphicEditor(new DatabasePersistence());
    editor.SaveAllShapes(destination: "Server=.;Database=Shapes;Trusted_Connection=True;", shapes);
    // Код самого редактора НЕ ИЗМЕНИЛСЯ. Он стал управляемым извне.
}
```

```
Консоль отладки Microsoft Visual Studio
Сохранил в файл myfile.txt
Сохранил в файл Server=.;Database=Shapes;Trusted_Connection=True;

C:\Users\geniu\source\repos\TestApplicationForLection\SomeProgram\b
.exe (процесс 2936) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включи
аметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке
Нажмите любую клавишу, чтобы закрыть это окно:
```

Итог


Circle
-int _radius
+Circle(x, y, radius)
+Draw() : void
«interface»
IDrawable
+Draw() : void
«abstract»
Shape
#int _x
#int _y
+Draw() : void
Rectangle
-int _width
-int _height
+Rectangle(x, y, width, height)
+Draw() : void
«interface»
IClickable
+OnClick() : void
Button
+Button(x, y)
+Draw() : void
+OnClick() : void