S.O.L.I.D.

Принципы на практике


ТРОЛЬ

План

- История S.O.L.I.D.
- S - ***The Single Responsibility Principle***
- Абстракция
- D - ***The Dependency Inversion Principle***
- L - ***The Liskov Substitution Principle***
- O - ***The Open Closed Principle***
- I - ***The Interface Segregation Principle***
- Конфликты S.O.L.I.D. с другими подходами проектирования
- Вывод

История S.O.L.I.D.

- Роберт Мартин собрал принципы в 2002г.
- Книга *Agile Software Development: Principles, Patterns, and Practices* (Быстрая разработка программ. Принципы, примеры, практика)

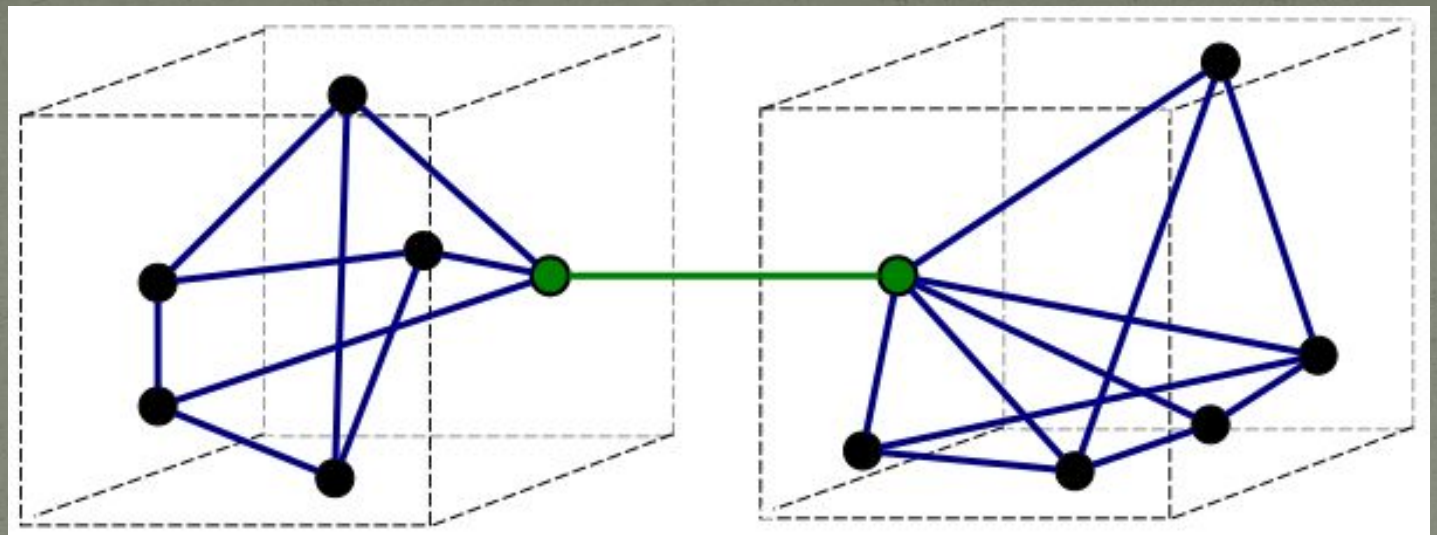
S - The Single Responsibility Principle

- *A class should have only one reason to change.*

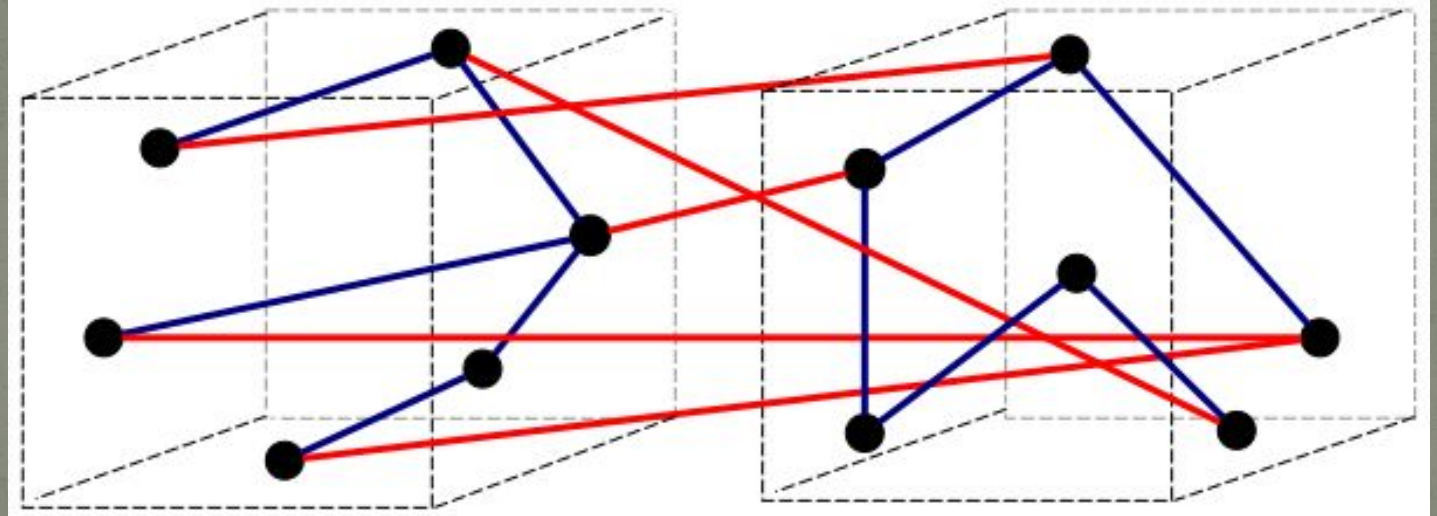
  Robert C. Martin
- Не должно быть больше одной причины для изменения класса
- Связность - мера силы взаимосвязанности элементов внутри модуля.
- Связность характеризует то, насколько хорошо все методы класса или все фрагменты метода соответствуют главной цели

S - The Single Responsibility Principle

- Связность != Связанность (Зацепление)
- Связанность - способ и степень взаимозависимости между программными модулями
- Хорошо спроектированная система = сильная связность + слабая связанность.



a) Слабое зацепление, сильная связность



b) Сильное зацепление, слабая связность

Абстракция

- Абстракция — выделение общих характеристик множества объектов, достаточных для решения рассматриваемой задачи.
- Идея абстракции - представление множества объектов минимальным набором полей и методов для решения задачи.
- Типы абстракций: Абстрактный класс, Interface (Контракт)

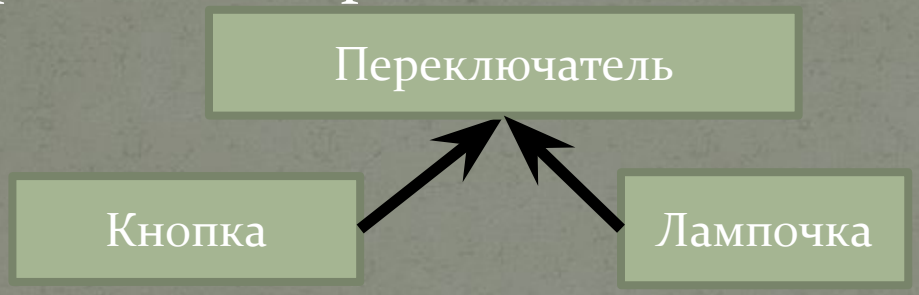
D - *The Dependency Inversion Principle*

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.
- Зависимость == Знание.

D - *The Dependency Inversion Principle*

- Реализация без инверсии

- Недостатки: кнопка – абстракция, но зависит от лампочки, и может зажигать только лампочку.
- Внедрение инверсии зависимости



- Кнопка и лампочка зависят от абстракции, кнопка может управлять любым переключателем.

D - *The Dependency Inversion Principle*

- Самая сильная зависимость – знания как создавать объекты
- Все зависимости должны приходить снаружи
- Composition root (Корень композиции) – входная точка в модуль. В корне композиции создаются/регистрируются компоненты модуля.
- Resolution root (Корень разрешения) – корневой объект модуля.

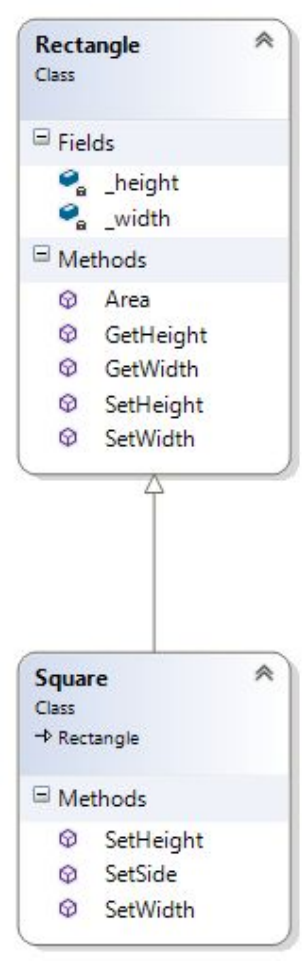
L - *The Liskov Substitution Principle*

- Наследующий класс должен дополнять, а не изменять базовый.
- Взаимозаменяемость подтипов позволяет расширять функциональные возможности модуля, основанного на базовом типе, не изменяя его.

L - *The Liskov Substitution Principle*

- Проектирование по контракту (design by contract)
- Контракт – ожидаемое поведение
- У каждого класса и у каждого метода есть контракт
- Контракт определяется ограничениями на входные и выходные условия
- Расширение (наследование) исходного класса может заменять оригинальное входное условие только равным ему или более «слабым», выходное условие заменяется равным или более “сильным”
- **X «слабее» У, если X не выполняет все ограничения У**

L - *The Liskov Substitution Principle*




```
class Rectangle
{
    float _width;
    float _height;

    public virtual void SetWidth(float width) {...}

    public virtual void SetHeight(float height) {...}

    public float GetWidth() {...}

    public float GetHeight() {...}

    public float Area() {...}
}

class Square : Rectangle
{
    public override void SetHeight(float height)
    {
        SetSide(height);
    }

    public override void SetWidth(float width)
    {
        SetSide(width);
    }

    public void SetSide(float side)
    {
        base.SetWidth(side);
        base.SetHeight(side);
    }
}
```

L - *The Liskov Substitution Principle*

- Проблема подстановки

```
public void SetRectangleSides(Rectangle target)
{
    target.SetHeight(5);
    target.SetWidth(4);

    target.Area() ???
}
```

- Square ослабляет пост условие – изменение высоты изменяет ширину

O - *The Open Closed Principle*

- *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*

Bertrand Meyer

- Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения

O - *The Open Closed Principle*

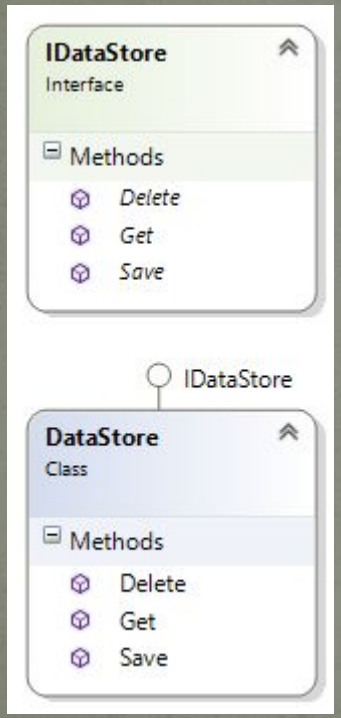
- Идея принципа открытости/закрытости – выделение “запечатанных” абстракций в часто изменяемых частях программы.
- Открытость (Расширение) – добавление новых реализаций абстракции.
- Закрытость (Без модификаций) – расширение не влияет на абстракцию и зависящие модули.
- **Расширяемая система должна оставаться неизменной.**

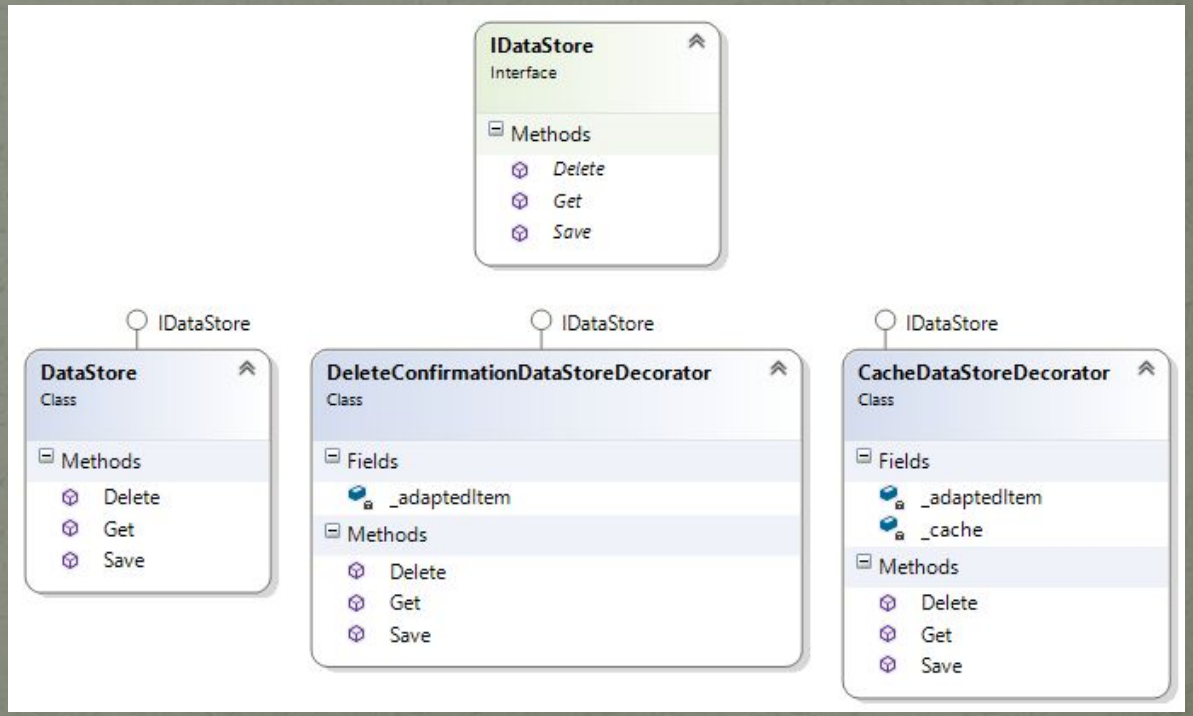
I - *The Interface Segregation Principle*

- *Клиенты не должны зависеть от методов, которые они не используют*
- Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, клиенты должны знать только о методах, которые необходимы им в работе.
- При изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

I - *The Interface Segregation Principle*

- “Тучные” интерфейсы затрудняют расширение
- Шаблон «Декоратор»





I - *The Interface Segregation Principle*

- Каждый «Декоратор» расширяет только один метод, хотя знает о всех !

```
public void Delete(DataObject entity)
 {
     if (Console.ReadKey().Key == ConsoleKey.Y)
     {
         _adaptedItem.Delete(entity);
     }
 }
```

```
public DataObject Get(int id)
{
    if (!_cache.ContainsKey(id))
    {
        _cache[id] = _adaptedItem.Get(id);
    }

    return _cache[id];
}
```

Конфликты S.O.L.I.D. с другими подходами проектирования

- Singleton pattern – нарушает инверсию зависимости
- Decorator (Пример из ***Liskov Substitution***) – декорирование метода Delete нарушает контракт метода
- DDD – МФУ с точки зрения DDD объект предметной области, с точки зрения ***Interface Segregation*** – антишаблон

Вывод

- Слепое следование каким либо принципам может расходиться со здравым смыслом
- S.O.L.I.D. – мощный инструмент проектирования
- S.O.L.I.D. – позволяет создавать гибкие программные модули

- Изменения неизбежны! Будьте гибкими.
- Прочитайте книги:
- *Agile Software Development: Principles, Patterns, and Practices*
- *Adaptive Code via C#: Agile coding with design patterns and SOLID principles*