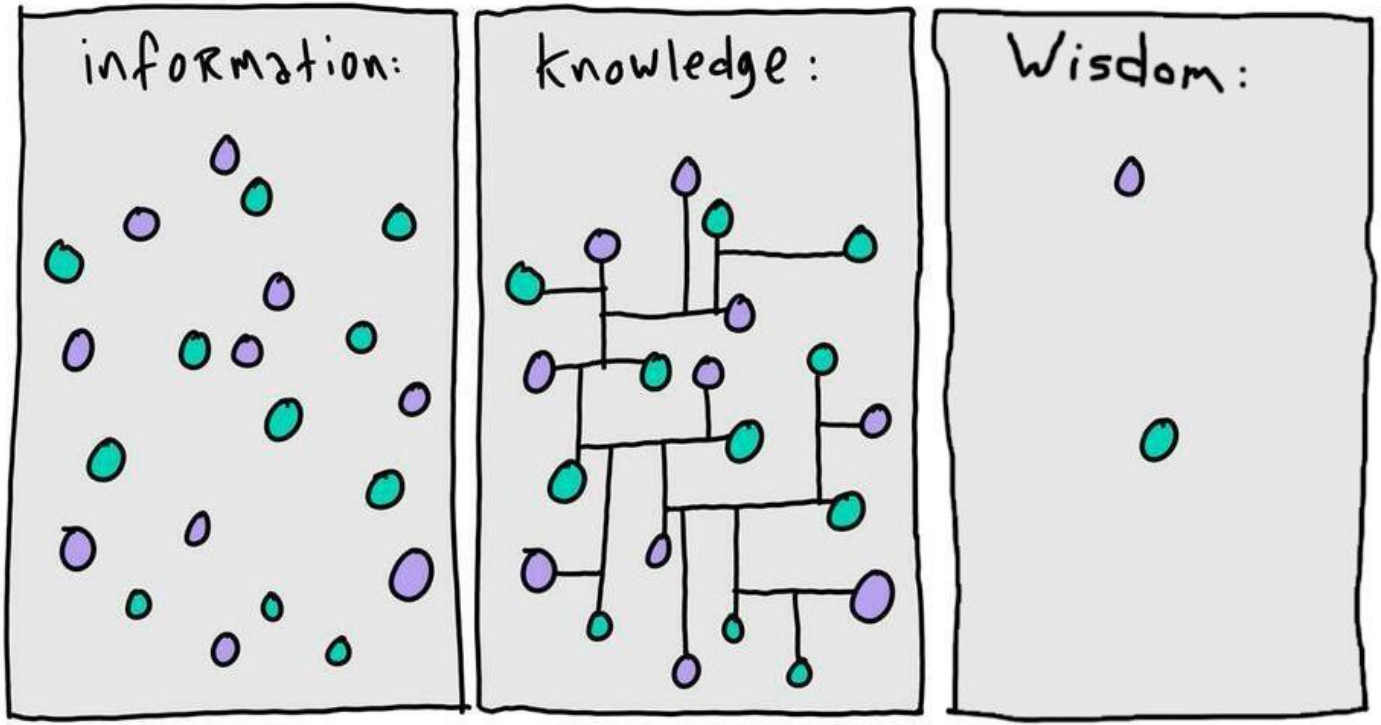
Web-разработка на C# и платформе Microsoft .NET

SOLID и шаблоны проектирования

Принципы – не правила



Симптомы плохого дизайна или его отсутствия

Жесткость - трудно изменить

Неподвижность - трудно повторно использовать

Хрупкость - легко сломать при изменении

Вязкость – легче воткнуть костыль, чем исправить должным образом

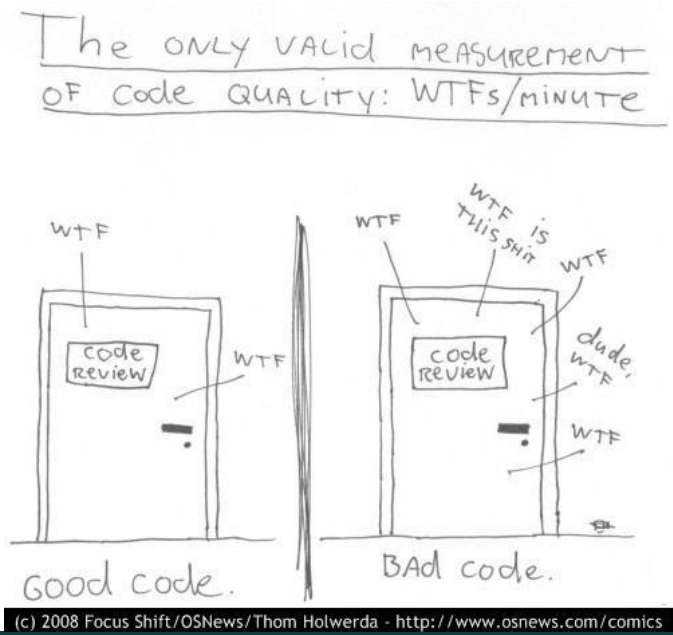
Сложность - ненужная сложность

Повторение - повторяющийся код

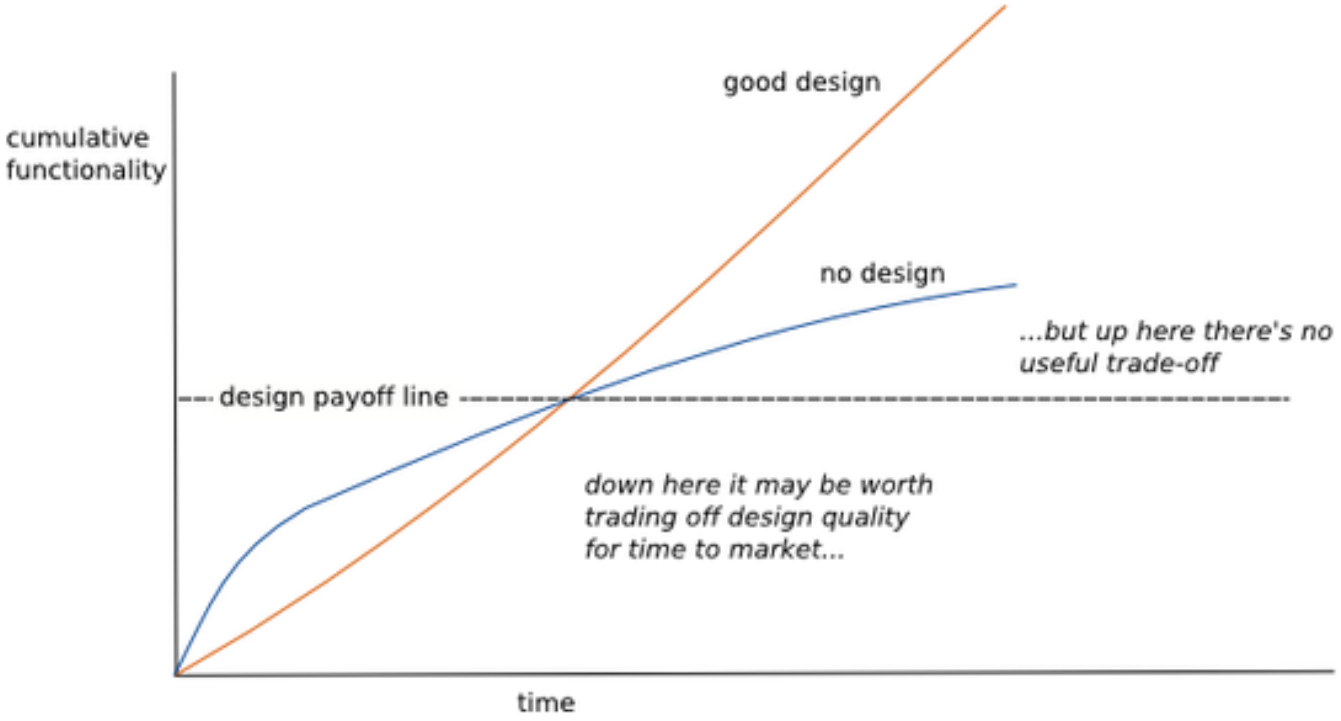
Непрозрачность - трудно понять

Последствия плохого дизайна

- Большой шанс провала продукта
- Трудно поддерживать и развивать
- Дорого поддерживать и развивать
- С таким кодом совсем не весело работать
- Технический долг



Компромисс дизайн - время



Признаки хорошего дизайна

- **Слабое связывание** - изменение в одной части не требует внесения изменений в другую
- **Абстрактность = Гибкость**
  - Чем меньше вы знаете о том, как это работает, тем лучше
  - Программа написана на уровне абстракции, а не конкретной реализации
- Следует принципам **SOLID**

Общие принципы дизайна

DRY –Don’t Repeat Yourself

Если что-то надо скопировать, то настало время для рефакторинга

YAGNI –You Ain’t Gonna Need It

Не делайте, если считаете, что это понадобится в будущем

KISS –Keep It Simple Stupid

Не усложняйте

Принципы SOLID

S	• SRP – Single Responsibility Principle
O	• OCP – Open/Closed Principle
L	• LSP – Liskov Substitution Principle
I	• ISP – Interface Segregation Principle
D	• DIP – Dependency Inversion Principle

Принцип единственной ответственности

S

- SRP – Single Responsibility Principle

«Каждый модуль должен обладать только одним предназначением»
или
«У каждого модуля должна быть только одна причина для изменения»

Нарушение принципа единой ответственности

```
public class Transaction {
        public void SaveToDatabase(){…}
        public string GenerateHtml() {…}
        public string Format() {…}
        public void GenerateReport() {…}
        public void NotifyCustomer() {…}
        public bool Validate() {…}
}
```

Принцип открытости/закрытости

O

- OCP – Open/Closed Principle

«Сущности должны быть открыты для расширения, но закрыты для модификации»

Нарушение OCP

```
Public double TotalPrice(IList<IProduct> products)
{
        var total=0.0;
        foreach (var product in products)
        {
                if(product is Carburetor)
                        total += (0.95*product.Price);
                else if(product is Battery)
                        total+=(0.8*product.Price);
                else
                        total+=product.Price;
        }
        return total;
}
```

Добавляем сущность

```
public class PricePolicy: IPricePolicy
{
        private readonly double factor;

        public PricePolicy(double factor)
        {
                this.factor=factor;
        }

        public double Price(double price)
        {
                return price*factor;
        }
}
```

Делаем лучше

```
public class Product: IProduct
{
        private double price;
        private readonly IPricePolicy pricePolicy;

        public Product(double price, IPricePolicy pricePolicy)
        {
                this.price=price;
                this.pricePolicy=pricePolicy;
        }
        public double Price
        {
                get { return pricePolicy.Price(price); }
        }
}
```

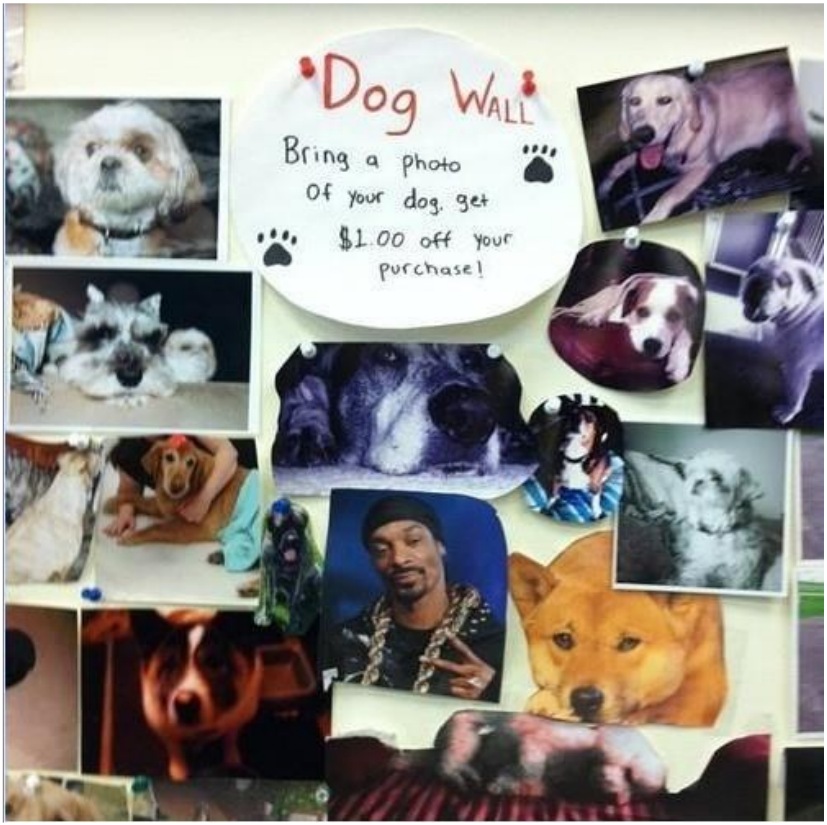
Принцип подстановки Барбары Лисков

L

- LSP – Liskov Substitution Principle

«Объекты классов должны легко заменяться на экземпляры любых своих потомков без нарушения функциональности»

Кольцо – это круг, но прямоугольник – не линия



Принцип разделения интерфейсов

I

- ISP – Interface Segregation Principle

«Объекты – пользователи не должны зависеть от методов, которые они не используют»

Пример ICP

```
public interface IEmployee
{
        void SubmitTimeSheet(TimeSheet timeSheet);
}

public interface IManager: IEmployee
{
        void ApproveTimeSheet(TimeSheet timeSheet);
        InternetUsage GetInternetUsage(int employeeId);
}
```

Принцип инверсии зависимостей

D

- DIP – 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».

Инверсия зависимостей

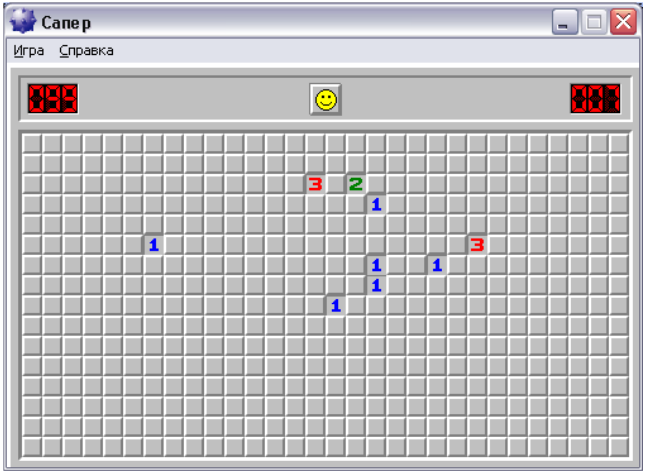
Что такое зависимость

- Если вы ссылаетесь на что-то, вы **зависите** от этого.
- Все объекты, создаваемые внутри класса – его зависимости.
- Если то, от чего вы зависите, меняется, вы тоже должны измениться.
- Зависимости покажут, как изменения распространяются через систему.

Спагетти код (Spaghetti code)

- **Спагетти-код** — слабо структурированная и плохо спроектированная система, запутанная и очень сложная для понимания.
- множество примеров анти-паттерна программирования **копи-пастом**.
- **огромные**, по размеру кода, **методы**.
- подобный код в будущем не сможет разобрать даже его автор.

Антипаттерн «тайное знание»




```
public void Draw(Graphics graphics)
{
    Rectangle drawRect = aquaRect;
    drawRect.Inflate(-1, -1);

    graphics.DrawRectangle(Pens.Black, drawRect);

    if (fishPoints == null) // Вызов метода Init обязателен!
        throw new ApplicationException("Call Init method before
Draw");
```

Антипаттерн «божественный объект»

- «Мне нужен такой-то функционал. — Используй MegaCoreObject! — И ещё, мне нужен и… — Я же сказал, используй MegaCoreObject!»
- Божественный объект —это объект, который хранит в себе слишком много, или делает слишком много.
- Основная часть функциональности программы кодируется **в одном объекте**.

Антипаттерн «слепая вера»

Проверяйте входные аргументы на корректность

```
public abstract class BaseObject
{
    protected ICore core;
    protected Random rand;

    public BaseObject(Random rand, int initX, int initY, ICore core)
    {
        if (core == null)
            throw new ArgumentNullException("core");

        this.rand = rand;
        this.X = initX;
        this.Y = initY;
        this.core = core;
    }
```

«Что за 42?» или Магические числа (Magic numbers) + (Hard code)

Магическое число — константа, использованная в коде для чего либо, само число не несёт никакого смысла без соответствующего комментария.
Жёсткое кодирование — внедрение различных данных об окружении в реализацию. Например — различные пути к файлам, имена процессов, устройств и так далее.

Антипаттерн «неожиданное поведение»

Поведение объекта должно быть логичным не только для автора класса

```
class Car
{
    private static int carCount = 0;

    public static int CarCount { get { return carCount; } }

    public Car()
    {
        carCount++;
    }
}
```

Если кто-то не понимает логику вашего кода, возможно, что дурак не он, а вы.

Для чего нужны паттерны

- Повторное использование удачных решений типовых задач.
- Упрощение сопровождения кода.
- Улучшение документированности кода.
- Дополнительная информация об архитектуре приложения.
- Унификация терминологии, упрощение обсуждений архитектуры приложений.

Когда паттерны – зло?

- Это же паттерн! Он не может быть неправильным!
- Давайте напихаем их побольше!
- Зачем разбираться в архитектуре? На все есть готовые шаблоны!

Как правильно работать с паттернами

- В каких ситуациях применяется паттерн?
- Что дает использование паттерна?
- Что будет, если не использовать данный паттерн?
- Какие есть альтернативы?

Вопросы эти задать себе должен ты, юный падаван!

Типы паттернов

- Порождающие
  - Factory
- Структурные
  - Adapter
  - Facade
- Поведенческие
  - Strategy

Что почитать?

Head First Design Patterns

By Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra

Паттерны проектирования на платформе .NET

By Сергей Тепляков

СПАСИБО ЗА ВНИМАНИЕ