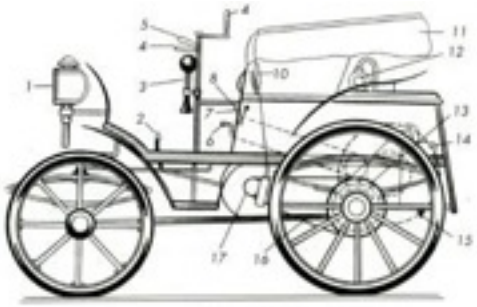
Принципы проектирования

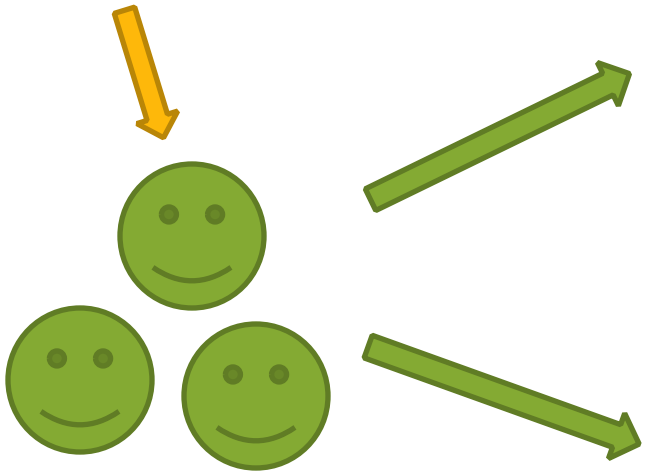
План

- разные подходы к внутреннему качеству приложения
- чем вреден некачественный код
- нарастание энтропии и сложности
- технический долг
- шаблоны и принципы проектирования
- описание некоторых принципов проектирования

Процесс разработки

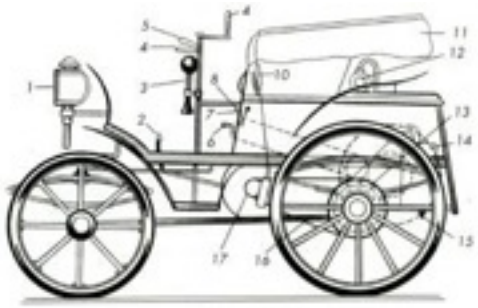
Клиент



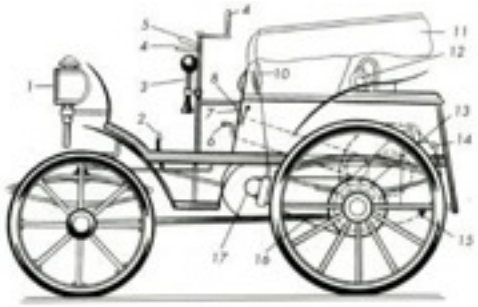


Разработчики

С точки зрения клиента:



=

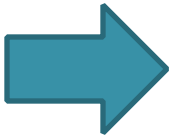


=

?

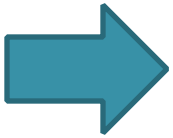
Момент истины

- расширение приложения
- исправление ошибок
- написание тестов



- больше времени
- больше ошибок
- иногда очень сложно

- плохое внутреннее качество



- плохое внешнее качество

Нарастание энтропии

- разные люди пишут код
- растет размер приложения
- растет связность между модулями, классами

Технический долг

Технические долги включают ту работу в проекте, которую мы решаем не делать в данный момент, но которая будет мешать развитию проекта в дальнейшем, если не будет выполнена

- черновая разработка
- отсутствие проектирования

- увеличение времени на изменения
- рост количества ошибок

- рефакторинг
- улучшение качества дизайна и кода

Тело кредита

Вывод: технические долги == финансовые долги

Свойства качественного кода

- расширяемость, гибкость (extensibility, agility)
- сопровождаемость (maintainability)
- простота (simplicity)
- читабельность, понятность (readability, clarity)
- тестируемость (testability)

Теперь понятно, что качественный код - это не просто какой-то абстрактный «красивый» код, а код, которые обладает полезными внутренними свойствами.

Как писать качественный код

Чем руководствоваться?

- здравый смысл, опыт
- паттерны проектирования
- принципы проектирования
- правила рефакторинга
- модульные тесты

Когда нужно рефакторить?

- анти-паттерны проектирования
- code smell
- костыли
- большие временные затраты на изменения

Практики:

- парное программирование
- code review
- рефакторинг
- модульные тесты и TDD/BDD

Иерархия. Принципы и паттерны

Общие принципы проектирования

SoC: Separation of Concerns

Разделение системы на отдельные части (модули, звенья, слои, классы), которые будут как можно меньше связаны между собой.

Достигается за счет:

- разделение на звенья (tiers)
- разделение на слои (layers)
- модульность
- разделение на классы
- инкапсуляция

DRY: Don't Repeat Yourself

Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы

Свойства:

- снижает затраты на поддержку/развитие/изменение.
- относится не только к дублированию кода, но и к дублированию других абстракций системы

Достигается за счет:

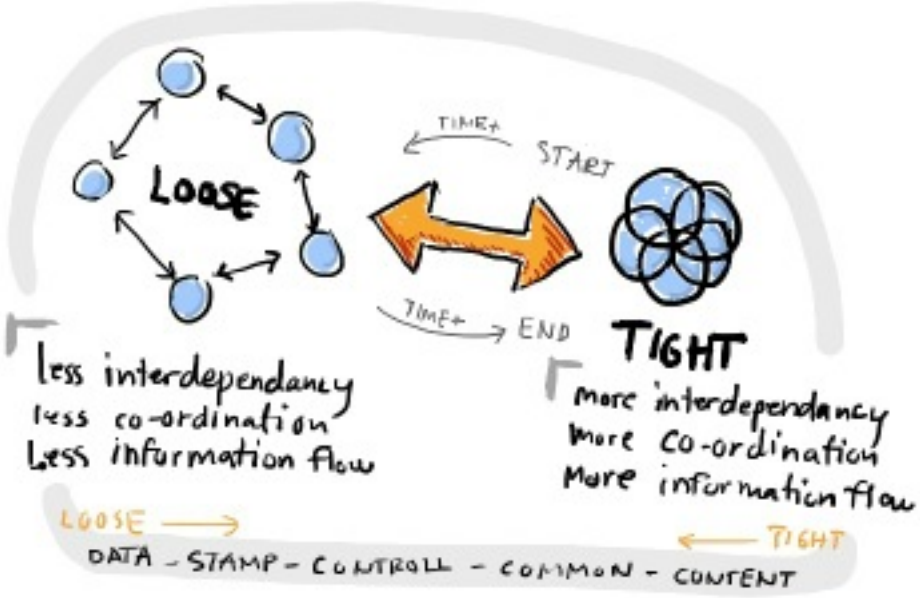
- отсутствие copy-paste
- повторное использование кода
- кодогенерация, AOP

Low Coupling

*Coupling (*связанность*)* – мера, определяющая, насколько жестко один элемент связан с другими элементами, или каким количеством данных о других элементах он обладает.

Достигается за счет:

- однонаправленные связи
- зависимость от интерфейсов
- don‘t talk to the strangers



High Cohesion

*Cohesion (*сцепленность*)* – мера, определяющая связанность и сфокусированность обязанностей/ответственности элемента.

Достигается за счет:

- сужение обязанностей элемента
- разделение ответственности между несколькими элементами
- группирование похожей ответственности в одном элементе

KISS: Keep it simple, stupid!

Простота системы является основной целью и ценностью.

Связаны:

- Бритва Оккама: «Не следует плодить новые сущности без самой крайней на то необходимости» или «Объяснение, требующее наименьших допущений, с большей вероятностью является правильным»
- Эйнштейн: «Все должно быть предельно просто, но не проще»

Достигается за счет:

- прагматичный подход к проектировани
- чувство меры, опыт

YAGNI: You Ain't Gonna Need It

Не нужно добавлять функциональность пока в ней нет непосредственной нужды.

Вытекает:

- предварительная оптимизация вредна

Достигается за счет:

- «ленивый» подход к проектированию

Минус: новый функционал может занимать много времени

Важно быть прагматиком и учитывать будущие требования!

Don’t make me think

Код должен легко читаться и восприниматься с минимумом усилий, если код вызывает затруднения чтобы его понять, то вероятно его стоит упростить

Write Code for the Maintainer

Практически любой код, который вы пишете, предстоит поддерживать в будущем вам или кому-то другому. В будущем, когда вы вернётесь к коду, обнаружите, что большая его часть совершенно вам незнакома, так что старайтесь писать как будто для другого.
“Пишите код так, как будто сопровождать его будет склонный к насилию психопат, который знает, где вы живете.” (Стив Макконнелл «Совершенный код»)

Hide Implementation Details

Скрытие деталей реализации позволяет вносить изменения в код компонента с минимальными затрагиванием других модулей которые используют этот компонент

Law of Demeter

Компоненты кода должны взаимодействовать только с их непосредственными связями (например, классы от которых они унаследованы, объекты, которые они содержат, объекты, переданные с помощью аргументов и т.д.)

Avoid Premature Optimization

Даже не думайте об оптимизации, если ваш код работает, но медленней, чем вы хотите. Только потом можно начать задумываться об оптимизации, и только основываясь на полученном опыте.
Мы должны забыть про небольшие улучшения эффективности, скажем, около 97% времени: преждевременная оптимизация — корень всех бед. © Дональд Кнут

Code Reuse is Good

Не очень содержательный, но тоже хороший принцип как и все другие. Повторное использование кода повышает надежность и уменьшает время разработки

Принципы объектно-ориентированного проектирования


SOLID
Software Development is not a Jenga game

SRP: Single Responsibility Principle

SRP: Single Responsibility Principle

1. Не должно быть больше одной причины для изменения класса.

2. Каждый класс должен быть сфокусирован на своей области ответственности.

Цель:

- упростить внесение изменений
- защититься от побочных эффектов при изменениях
- Separation of Concerns на уровне классов

Достигается за счет:

- правильное проектирование
- использование паттернов проектирования

SRP: неправильный вариант

```
public class Account
{
    public string Number;
    public decimal CurrentBalance;
    public void Deposit(decimal amount) { ... }
    public void Withdraw(decimal amount) { ... }
    public void Transfer(decimal amount, Account recipient) { ... }
    public TaxTable CalculateTaxes(int year) { ... }
    public void GetByNumber(string number) { ... }
    public void Save() { ... }
}
```

SRP: правильный вариант

```
public class Account
{
    public string Number;
    public decimal CurrentBalance;
    public void Deposit(decimal amount) { ... }
    public void Withdraw(decimal amount) { ... }
    public void Transfer(decimal amount, Account recipient) {
... }
}

public class AccountRepository
{
    public Account GetByNumber(string number) { ... }
    public void Save(Account account) { ... }
}

public class TaxCalculator
{
    public TaxTable CalculateTaxes(int year) { ... }
}
```

OCP: Open-Closed Principle

OCP: Open-Closed Principle

Программные сущности *(модули, классы, методы)* должны быть открыты для расширения, но закрыты для изменения.

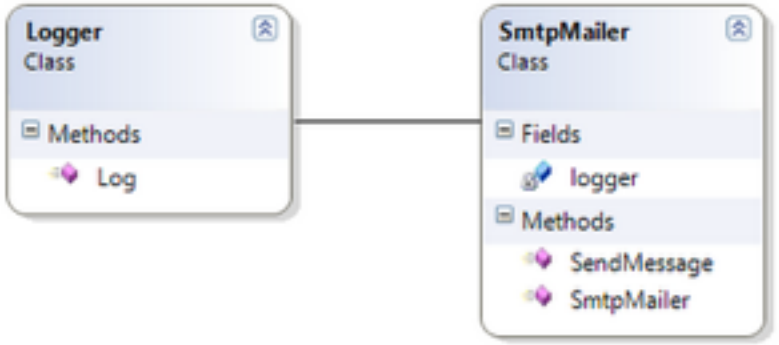
Цель:

- добиться гибкости системы
- избежать сильных переработок дизайна при изменениях

Достигается за счет:

- правильное наследование
- инкапсуляция

OCP: неправильный вариант

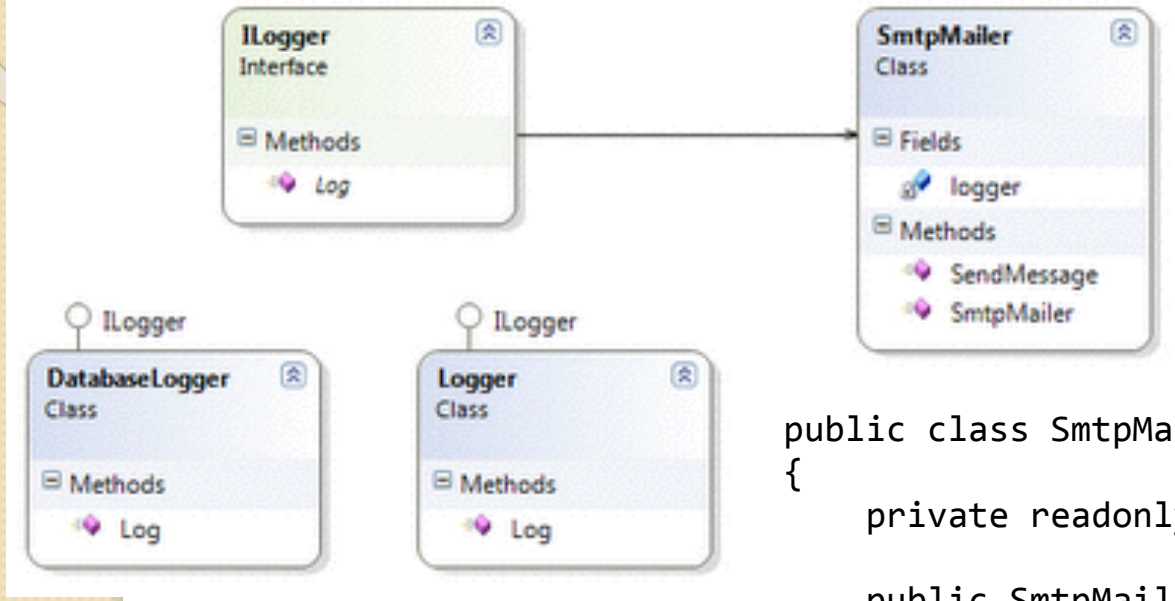



```
public class SmtpMailer
{
    private readonly Logger logger;

    public SmtpMailer()
    {
        logger = new Logger();
    }

    public void SendMessage(string message)
    {
         // отправить сообщение
         logger.Log(message);
    }
}
```

Изменение: нужно писать лог в базу данных

```
public class DatabaseLogger
{
    public void Log(string logText)
    {
        // сохранить лог в базе данных
    }
}

public class SmtpMailer
{
    private readonly DatabaseLogger logger;

    public SmtpMailer()
    {
        logger = new DatabaseLogger();
    }

    public void SendMessage(string message)
    {
         // отправить сообщение
         logger.Log(message);
    }
}
```

OCP: правильный вариант




```
public class SmtpMailer
{
    private readonly ILogger logger;

    public SmtpMailer(ILogger logger)
    {
        this.logger = logger;
    }

    public void SendMessage(string message)
    {
        // отправить сообщение
        logger.Log(message);
    }
}
```

LSP: Liskov Substitution Principle

LSP: Liskov Substitution Principle

1. Подтипы должны быть заменяемыми их исходными типами.
2. Наследники должны соблюдать контракт предка

Цель:

- избежать побочных эффектов и ошибок в существующем коде, работающем с базовыми классами, при добавлении наследников
- строить правильные иерархии наследования

Достигается за счет:

- правильное наследование классов

LSP: неправильный вариант

```
class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
    public int CalculateArea() { return Width * Height; }
}

class Square : Rectangle
{
    public override int Height
    {
        get { return base.Height; }
        set { base.Height = value; base.Width = value; }
    }
    public override int Width
    {
        get { return base.Width; }
        set { base.Width = value; base.Height = value; }
    }
}
class Program
{
    static void Main()
    {
        Rectangle r = new Square();
        r.Width = 3;r.Height = 2;
        Assert.AreEqual(6, r.CalculateArea());
    }}
```

LSP: правильный вариант

```
class Rectangle : IFigure
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int CalculateArea() {
        return Width * Height;
    }
}
class Square : IFigure
{
    public int Side { get; set; }
    public int CalculateArea() {
        return Side * Side;
    }
}

class Program {
    static void Main() {
        Rectangle r = new Square(); // fail on compilation
        r.Width = 3;
        r.Height = 2;
        Assert.AreEqual(6, r.CalculateArea());
    }
}
```

ISP: Interface Segregation Principle

ISP: Interface Segregation Principle

1. Клиент не должен вынужденно зависеть от элементов интерфейса, которые он не использует.

2. Зависимость между классами должна быть ограничена как можно более «узким» интерфейсом.

Цель:

- ограничить знание одного класса о другом
- уменьшить зависимость между классами
- уменьшить количество методов для реализации при наследовании

Достигается за счет:

- фокусирование интерфейсов на своей ответственности
- наследование от нескольких интерфейсов, а не от одного

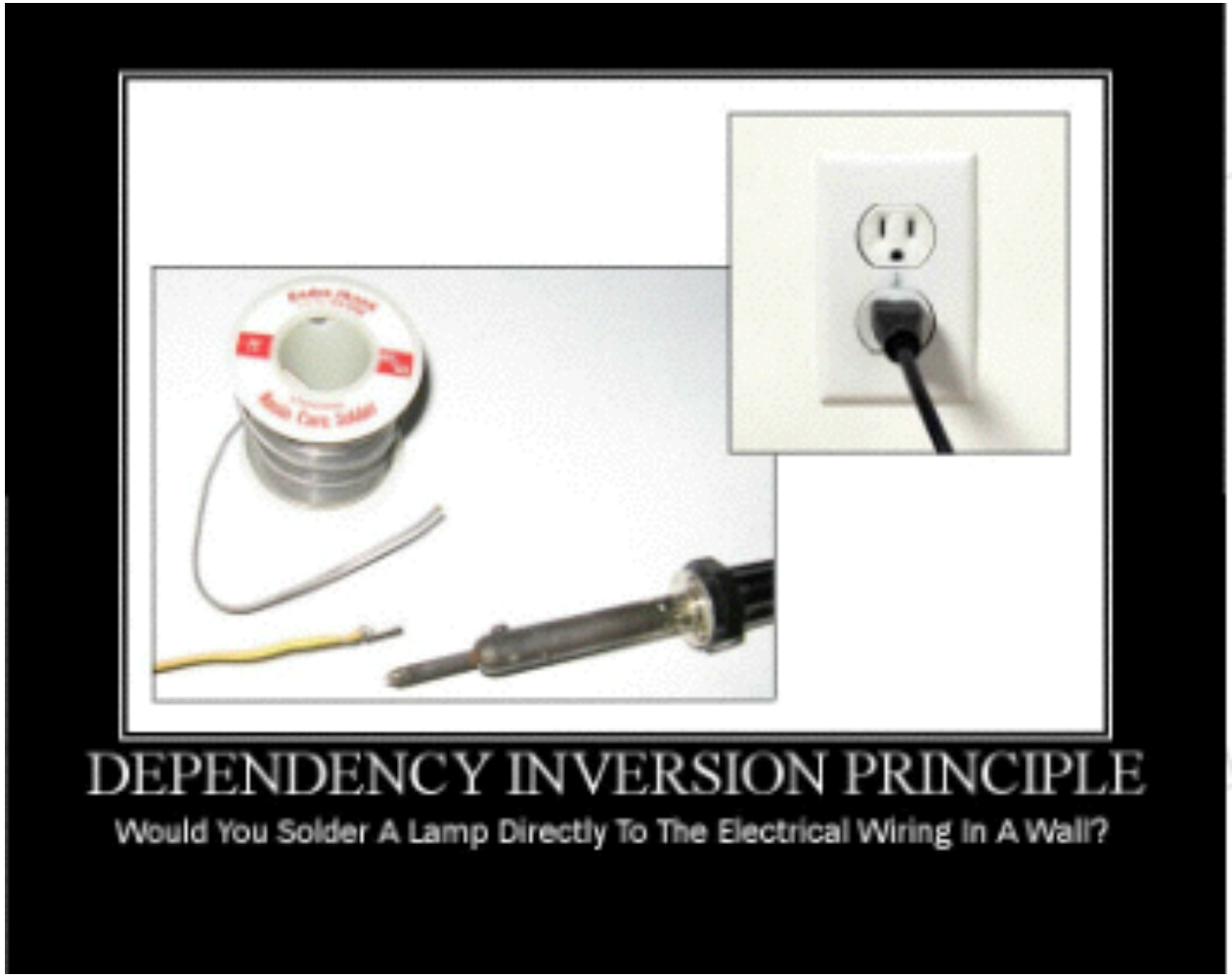
ISP: неправильный вариант

```
public interface IBird
{
    void Eat();
    void Sing();
    void Fly();
}

public class Parrot : IBird
{
    // здорово
}

public class Pigeon : IBird
{
    // ну, я не очень хорошо пою, но ладно
}

public class Penguin : IBird
{
    // хм... а я вообще птица?
}
```

ISP: правильный вариант

```
public interface ICommonBird
{
    void Eat();
}
public interface ISingingBird
{
    void Sing();
}
public interface IFlyingBird
{
    void Fly();
}
public class Parrot : ICommonBird, ISingingBird, IFlyingBird
{
    // хм, ничего не изменилось
}
public class Pigeon : ICommonBird, IFlyingBird
{
    // о, так лучше, я могу не петь
}
public class Penguin : ICommonBird
{
    // так намного лучше! хотя я еще и плавать могу ☺
}
```

IoC: Inversion of Control
DIP: Dependency Inversion Principle



IoC: Inversion of Control
DIP: Dependency Inversion Principle

1. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Они должны зависеть от абстракции.

2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Цель:

- уменьшить связанность (coupling)

Связан:

- Hollywood principle: *Don't call us, we'll call you*

Достигается за счет:

- паттерны Dependency Injection, Service Locator
- зависимость от абстракций
- IoC frameworks

IoC, DIP: неправильный вариант

```
class Developer
{
    public void WriteApplication()
    {
        IApplication helloWorld = new HelloWorldApp();
        WriteCode(helloWorld);
    }

    private void WriteCode() { ... };
}

public interface IApplication
{
    string GetCode();
}

public class HelloWorldApp : IApplication
{
    // реализация
}

public class VeryBigApp : IApplication
{
    // реализация
}
```

IoC, DIP: правильный вариант

```
class Developer
{
    public void WriteApplication(IApplication app)
    {
        WriteCode(app);
    }

    private void WriteCode() { ... };
}

public static void Main()
{
    Developer dev = new Developer();
    IApplication app = new VeryBigApp(); // теперь мы умеем писать что угодно
    dev.WriteApplication(app);
}
```

Это шаблон Dependency Injection (Method Injection), но можно использовать и другие шаблоны (например, Service Locator), а также IoC Frameworks