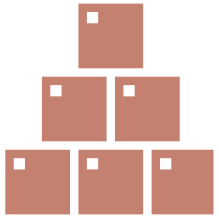
Проектирование программных компонентов

Часть 1. Программные структуры **среднего уровня**

Принципы SOLID

План

Уровни проектирования



Понятие программного компонента

Принципы SOLID

Уровни проектирования

Программный компонент

- **Программный компонент** — программная часть системы компонент программного обеспечения — [Л.Г.Суменко. Англо русский словарь по информационным технологиям. М.: ГП ЦНИИС, 2003.]
- **Программная компонента** – это *единица* программного обеспечения, исполняемая на одном компьютере в пределах одного процесса, и предоставляющая некоторый набор сервисов, которые используются через ее внешний *интерфейс* другими компонентами, как выполняющимися на этом же компьютере, так и на удаленных компьютерах.
  Ряд *компонент* пользовательского интерфейса предоставляют свой сервис конечному пользователю.

Программный компонент

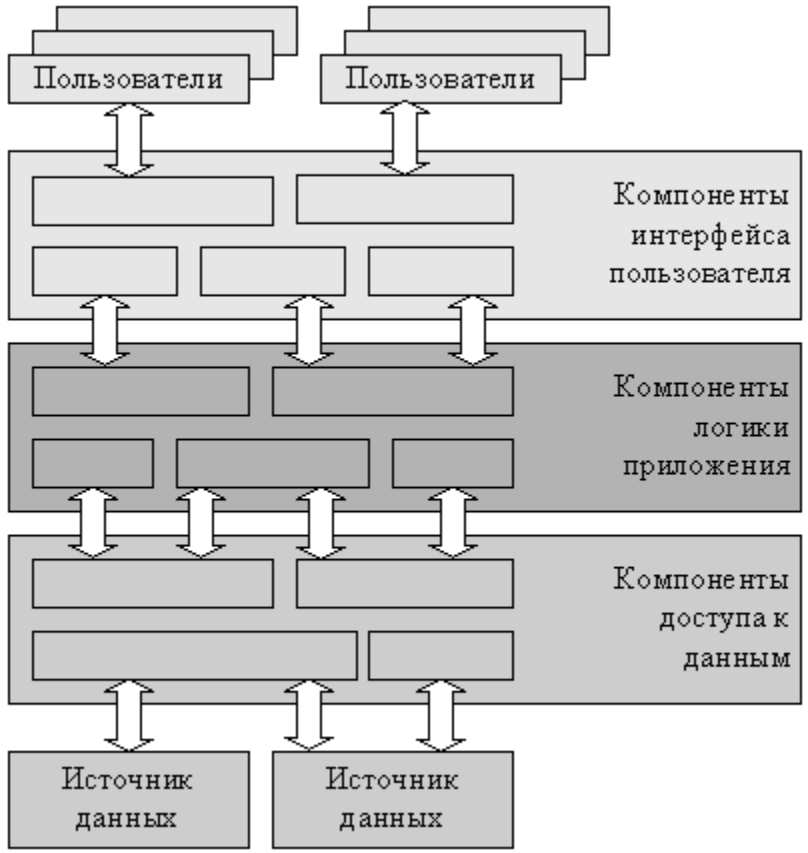
Программный компонент - Это единица развертывания (.jar,.gem,.dll).

Для чего?

- Независимое развертывание
- Независимая разработка

Однако в реальности – все несколько иначе…

Программный компонент



Из чего состоит программный компонент?

Из *хорошо* спроектированных программных структур среднего уровня.

Как создать *хорошую* программную структуру среднего уровня?

SOLID

- **SRP**: Single Responsibility Principle

Принцип единственной ответственности

- **OCP**: Open-Closed Principle

Принцип открытости/закрытости

- **LSP**: Liskov Substitution Principle

Принцип подстановки Барбары Лисков

- **ISP**: Interface Segregation Principle

Принцип разделения интерфейсов

- **DIP**: Dependency Inversion Principle

Принцип инверсии зависимости

- Роберт Мартин
- Майкл Физерс

SRP. Принцип единственной ответственности

- Модуль должен иметь одну и только одну причину для изменения
- Модуль должен отвечать за одного и только одного актора

Актор – группа, состоящая из одного или более лиц, желающих некоторого изменения.

Модуль - связный набор функций и структур данных.

SRP. Принцип единственной ответственности

Несоблюдение:

- Проблема модификации общих частей
- Проблема слияния изменений (git merge)

SRP. Принцип единственной ответственности

Универсальный подход

- Уровень функций и методов – «функция должна делать что-то одно и только одно»
- Уровень компонентов – принцип согласованного изменения (CCP)
- Архитектурный уровень – принцип оси изменения (Axis of Change)

OCP. Принцип открытости/закрытости

- Программные сущности должны быть открыты для расширения и закрыты для изменения

- Цель: легкая расширяемость и безопасность от влияния изменений

OCP. Принцип открытости/закрытости

Упорядочивание в иерархию, защищающую компоненты уровнем выше от изменения в компонентах уровнем ниже

Чем выше политики – тем выше защита

LSP. Принцип подстановки Барбары Лисков

- Если для каждого объекта o1 типа S существует такой объект o2 типа T, что для всех программ P, определенных в терминах T, поведение P не изменяется при подстановке o1 вместо o2, то S является подтипом T.

- Простое нарушение совместимости может вызвать **загрязнение** архитектуры системы **значительным количеством дополнительных механизмов**.

LSP. Принцип подстановки Барбары Лисков

Проблема квадрат-прямоугольник

(как выстроить наследование между квадратом и прямоугольником?)

ISP. Принцип разделения интерфейсов

- Зависимости, несущие лишний груз ненужных и неиспользуемых особенностей, могут стать причиной неожиданных проблем.

DIP. Принцип инверсии зависимости

Для максимальной гибкости:

Зависимости должны быть направлены на абстракции, а не конкретные реализации

DIP. Принцип инверсии зависимости

Набор правил:

1) Не ссылайтесь на изменчивые конкретные классы, ссылайтесь на абстрактные интерфейсы
2) Не наследуйте изменчивые конкретные классы
3) Не переопределяйте конкретные функции, делайте функции абстрактными и добавляйте несколько реализаций.
4) Не ссылайтесь на имена конкретных и изменчивых сущностей

Некоторые практические вопросы Инверсия управления

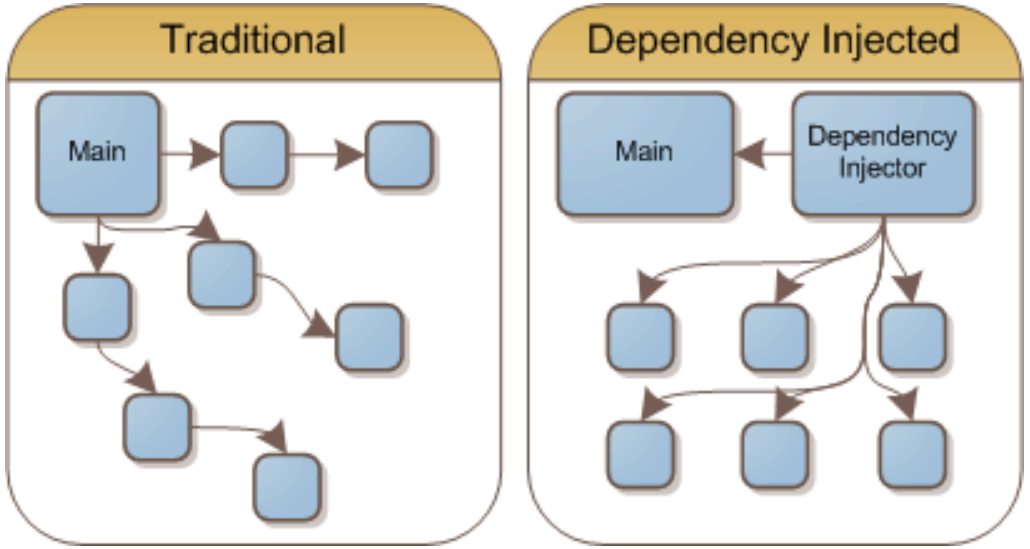
IoC - архитектурное решение интеграции, упрощающее расширение возможностей системы, при котором поток управления программы контролируется фреймворком

Критика:

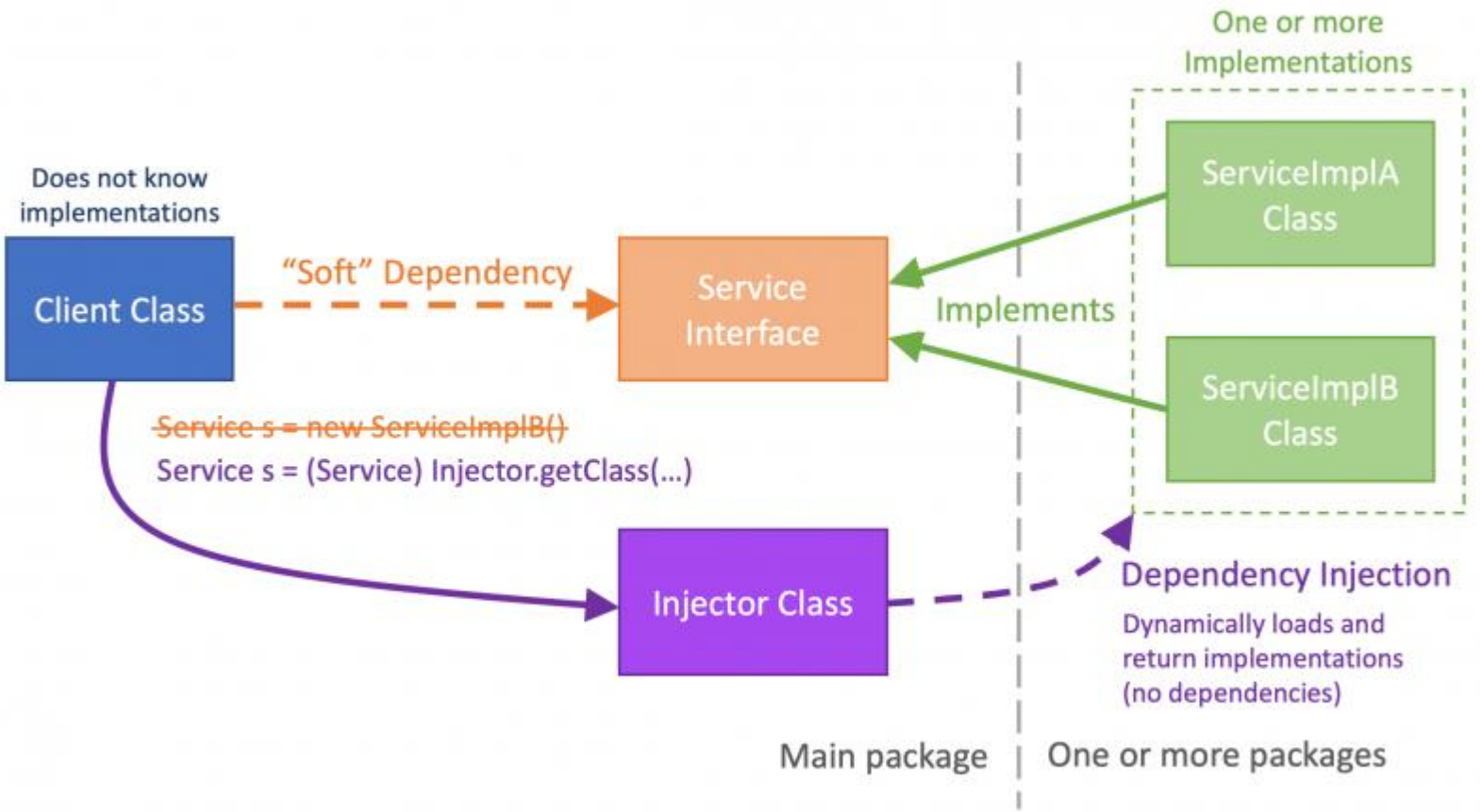
- логика взаимодействия программы разбросана
- поток управления задан неявно

Dependency Injection

- Внедрение зависимости (*англ. Dependency injection, DI*) — процесс предоставления внешней зависимости программному компоненту.
- В соответствии с SRP объект отдаёт заботу о построении требуемых ему зависимостей внешнему общему механизму



Dependency Injection



DI. Пример. Зависимость

```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```

```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```

DI. Пример. Регистрация зависимости

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

DI. Пример. Подключение зависимости

```
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```