Простая архитектура: разработка и тестирование приложений по Маслову

Никита Маслов, работяга

Обо мне

- Работаю программистом в системно значимом банке
- Более девяти лет коммерческой разработки в области автоматизации бизнес-процессов
- Успел поработать как на небольших, так и на крупных проектах

Viewer discretion is advised

Доклад будет об опыте автора. Автор не претендует на звание эксперта

Часть I
О чем поговорим

Агенда

1. Архитектура приложений
2. Стратегические и тактические паттерны
3. Обработка ошибок
4. Тестирование

Мои мысли о разработке enterprise-приложений

- Код приложения - это не цель, а просто инструмент для заработка денег

- Цель кода - минимизировать издержки бизнеса на автоматизацию процессов

Мой главный принцип в разработке

KISS – keep it simple, stupid!

Делать явно

Избегать неявного, рефлексии и библиотек, основанных на ней

Не делать абстракций над абстракциями

```
interface IRepository<TEntity>
{
    IQueryable<TEntity> Query();
    Task Create(TEntity entity);
    Task Update(TEntity entity);
    Task Delete(TEntity entity);
}
```

```
interface IUnitOfWork
{
    Task SaveChanges();
}
```

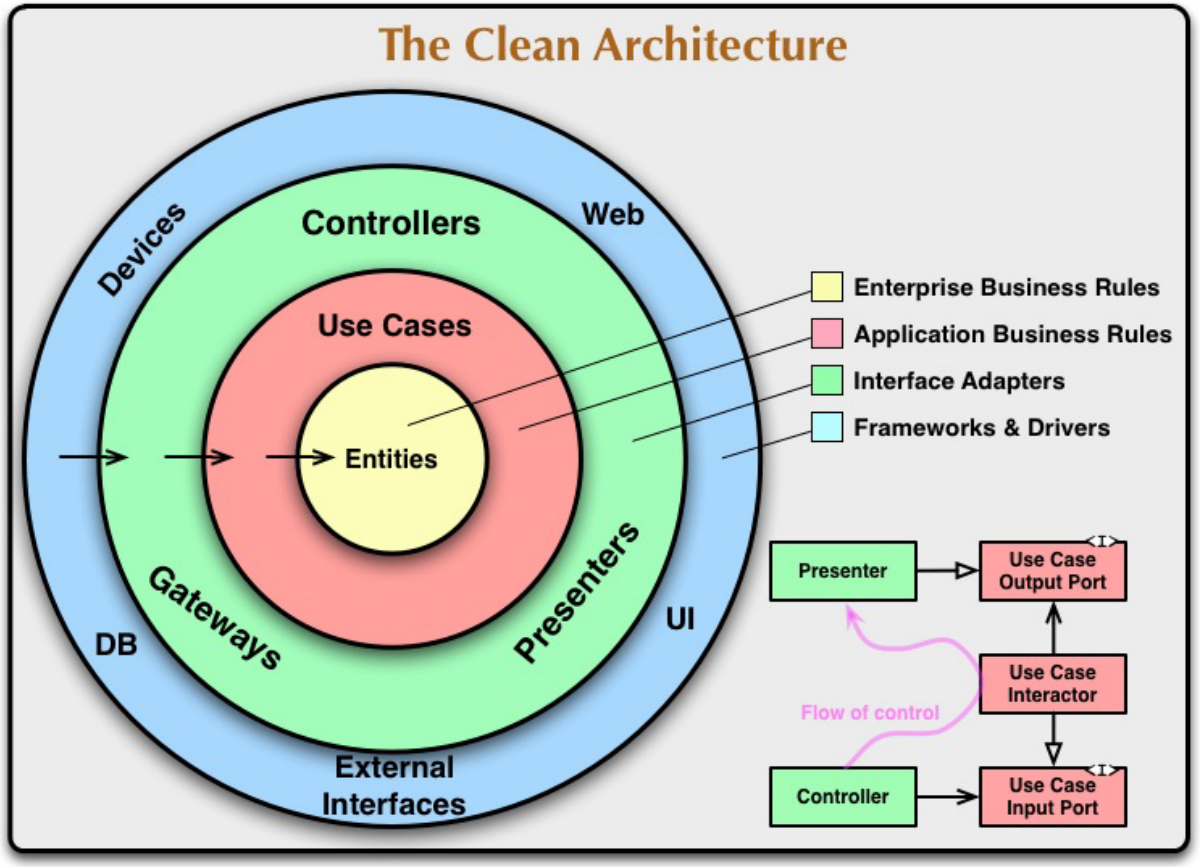
Не делать преждевременных обобщений

Явно обрабатывать ошибки

~~`throw new UserNotFoundException();`~~

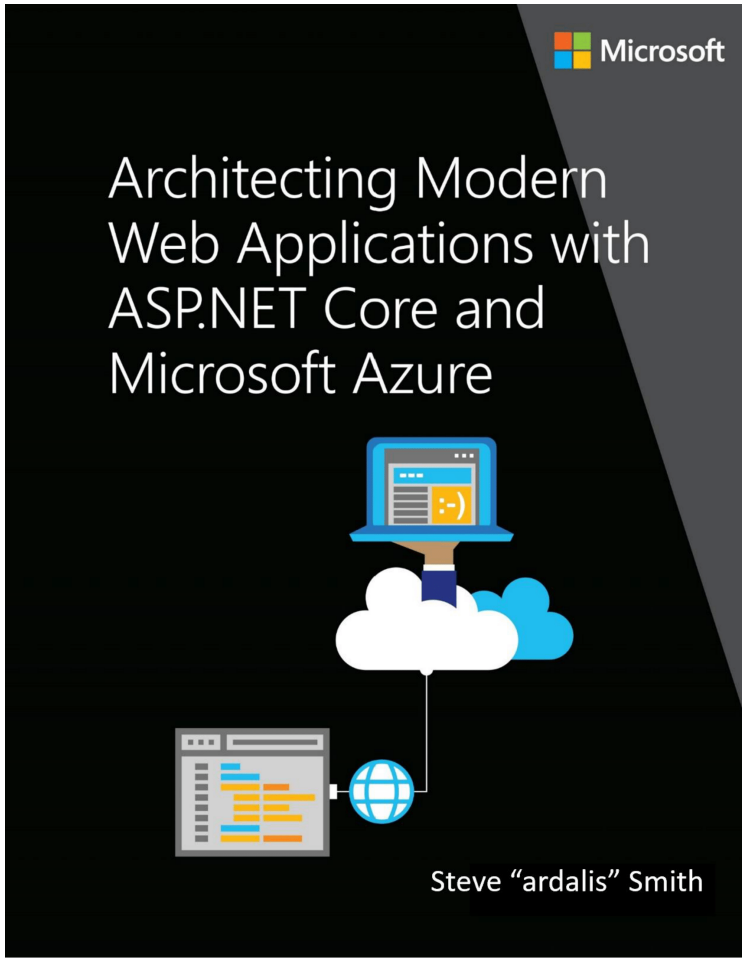
Часть II
Об архитектуре

Чистая архитектура



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

eShopOnWeb



https://github.com/dotnet-architecture/eShopOnWeb

Чем не устраивает чистая архитектура?

Entities = сущности для ORM

```
namespace Microsoft.eShopWeb.ApplicationCore.Entities.OrderAggregate;

public class Order : BaseEntity, IAggregateRoot
{
    #pragma warning disable CS8618 // Required by Entity Framework
    private Order() {}
```

https://github.com/dotnet-architecture/eShopOnWeb/blob/fc8cbc2b83c0d45e4be50b5998acc11de4ee30d1/src/ApplicationCore/Entities/OrderAggregate/Order.cs#L8-L8

```
namespace Microsoft.eShopWeb.Infrastructure.Data;

public class CatalogContext : DbContext
{
    public DbSet<ApplicationCore.Entities.OrderAggregate.Order> Orders { get; set; }
}
```

https://github.com/dotnet-architecture/eShopOnWeb/blob/fc8cbc2b83c0d45e4be50b5998acc11de4ee30d1/src/Infrastructure/Data/CatalogContext.cs

Примеры в eShopOnWeb очень простые

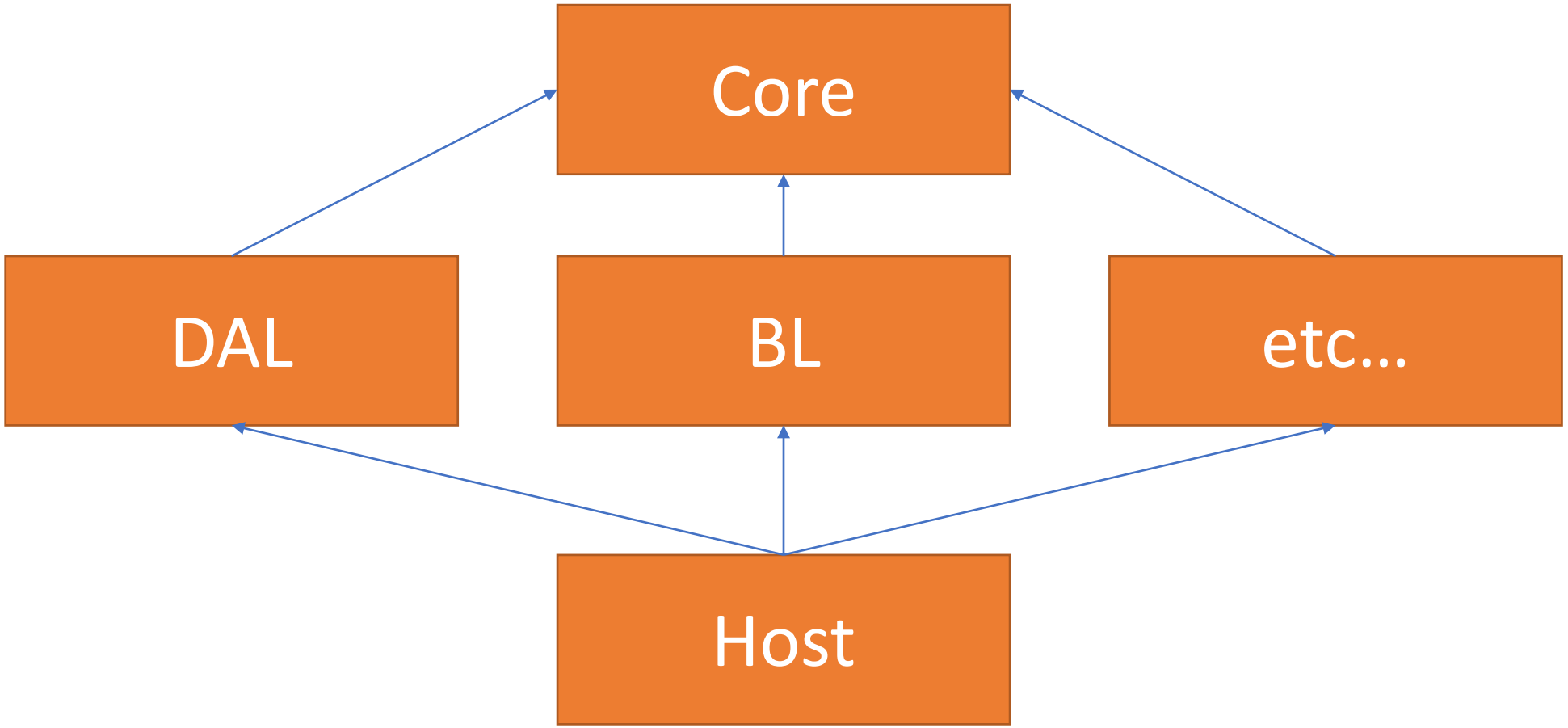
```
public decimal Total()
{
    var total = 0m;
    foreach (var item in _orderItems)
    {
        total += item.UnitPrice * item.Units;
    }
    return total;
}
```

https://github.com/dotnet-architecture/eShopOnWeb/blob/fc8cbc2b83c0d45e4be50b5998acc11de4ee30d1/src/ApplicationCore/Entities/OrderAggregate/Order.cs

Мой демо-пример

- Приложение с иерархичным списком пользователей
- Каждый пользователь может хранить файлы в S3

«Простая» архитектура



Базис простой архитектуры

- Ядро приложения: сборка (C# assembly), включает в себя:
  - Неизменяемые доменные модели
  - Декларация сервисов
- БД – это часть бизнес-логики
- Минимально необходимые абстракции

Модели БД не равны доменным моделям

```
internal class User
{
    public int Id { get; set; }
    public string Name { get; set; } = null!;
    public string Email { get; set; } = null!;
    public int? ParentId { get; set; }
    public DateTimeOffset CreationDate { get; set; }
    public User? Parent { get; set; }
    public List<User> Children { get; set; } = null!;
    public List<UserFile> Files { get; set; } = null!;
}
```

```
public sealed record UserModel(UserId Id,
    string Name,
    int FilesCount,
    UserId? ParentId,
    string? ParentName);

public sealed record UserWithChildrenModel(UserId Id,
    string Name,
    IReadOnlyList<UserWithChildrenModel> Children);
```

Сервисы

```
public interface IUsersService
{
    Task<CreateUserResult> CreateUser(CreateUserModel model);

    Task<UserModel?> GetUser(UserId id);

    Task<IReadOnlyList<UserModel>> GetUsers(GetAllUsersFilter? filter);

    Task<IReadOnlyList<UserWithChildrenModel>> GetUserTree();
}
```

БД – часть доменной модели

Дилемма: при использовании достаточно мощной БД, либо БД протекает в бизнес-логику, либо бизнес-логика протекает в БД.

Database and Always-Valid Domain Model

https://enterprisecraftsmanship.com/posts/database-always-valid-domain-model/

Трилемма Domain Driven Design

Нельзя получить три свойства системы одновременно:

- Полнота доменной модели (доменная модель содержит всю логику приложения)
- Чистота доменной модели (доменная модель не использует внешние зависимости)
- Производительность

https://enterprisecraftsmanship.com/posts/domain-model-purity-completeness/

Абстракции над абстракциями

1. IRepository
2. IUnitOfWork
3. ISpecification

EF DbContext

```
/// <summary>
///     A DbContext instance represents a session with the database and can be used to query and save instances of your entities. DbContext is a combination of the Unit Of Work and Repository patterns.
/// </summary>
public class DbContext
```

https://github.com/dotnet/efcore/blob/bdd9846218b002005321efed1cf5195cae12f1f2/src/EFCore/DbContext.cs#L15

Спецификация

- https://github.com/ardalis/Specification - абстракция над EF

- https://github.com/axelheer/nein-linq - позволяет писать свои спецификации в вызовах EF

```
internal static class UserExtensions
{
    [InjectLambda]
    public static bool FilterByName(this User user, string? filter)
    {
        throw new NotImplementedException();
    }

    private static Expression<Func<User, string?, bool>> FilterByName()
    {
        return (user, filter) =>
            string.IsNullOrEmpty(filter) || user.Name.Contains(filter);
    }
}
```

```
var users = await _context
            .Users
            .ToInjectable()
            .Where(u => u.FilterByName(filter.Name) &&
                        u.FilterByCreationDate(filter.CreatedAfter))
            .Select(u => new UserModel(new UserId(u.Id),
                u.Name,
                u.Files.Count,
                u.Parent == null ? null : new UserId(u.Parent!.Id),
                u.Parent == null ? null : u.Parent.Name))
            .ToArrayAsync(cancellationToken);
```

```
SELECT u.id, u.name, (
    SELECT count(*)::int
    FROM user_files AS u1
    WHERE u.id = u1.user_id), u0.id IS NULL, u0.id, u0.name
FROM users AS u
LEFT JOIN users AS u0 ON u.parent_id = u0.id
WHERE (@__filter_Name_0 = '' OR
       strpos(u.name, @__filter_Name_0) > 0)
      AND u.creation_date >= @__filter_CreatedAfter_1
```

БД – часть доменной модели:

- Простор для возможных оптимизаций
- Использование всех возможностей БД (ограничения, джоины, …)

Обновление почты

Проверить уникальность email пользователя можно unique-ограничением на базе

Своя проверка – нужно писать, легко ошибиться

ORM не идеальны

linq2db открывает транзакцию с уровнем repeatable read, если не может получить результат запроса за один поход в базу

```
var query = db.Master.Select(x => new
{
    x.Id1,
    Details = x.Details.Select(d => d.DetailValue)
}).FirstOrDefault(x => x.Id1 == 3);
```

https://github.com/linq2db/linq2db/issues/4053

Защитные интерфейсы

Установка границ через защитные интерфейсы: для каждой внешней зависимости свой слой

```
namespace Amazon.S3;

public interface IAmazonS3 : IDisposable, ICoreAmazonS3, IAmazonService

{

    Task<GetObjectResponse> GetObjectAsync(GetObjectRequest request,
CancellationToken cancellationToken);

}
```

```
namespace ArchitectureDemo.Services;

public interface IS3Service

{

    Task<Stream> GetFile(string fileName, CancellationToken
cancellationToken);

}
```

Разбивка по сборкам

- Разбивать на сборки исходя из зависимостей
- Группировка по фичам, а не по ролям

Демо

Часть III
Принципы простой архитектуры

Делать проще!

- Минимально необходимые абстракции
- Бизнес-требования должны быть изоморфны коду
- Не обобщать заранее

Бизнес-требования должны быть изоморфны коду

- Не делать «на будущее»
- Если в ТЗ одинаковые сущности разделены, то и в коде их надо разделить

Не обобщать заранее

- Нужно как минимум два примера!
- Не использовать if в обобщенном коде
- Каждый if ухудшает цикломатичность

Явное всегда лучше неявного

- Не придумывать своих конвенций (регистрация по имени, интерфейсу, …)
- Не использовать ambient-контексты (например, с транзакциями, авторизацией пользователя)

Меньше рефлексии

```
enum Color
{
    Red,
    Green,
    Blue
}
```

Решение из интернета

```
enum Color
{
    [Display(Name = "Красный")]
    Red,

    [Display(Name = "Зеленый")]
    Green,

    [Display(Name = "Синий")]
    Blue
}
```

```
public static class Extensions
{
    public static TAttribute GetAttribute<TAttribute>(this Enum
enumValue)
            where TAttribute : Attribute
    {
        return enumValue.GetType()
                        .GetMember(enumValue.ToString())
                        .First()
                        .GetCustomAttribute<TAttribute>();
    }
}

var colorDisplayName = Color.Green.GetAttribute<DisplayAttribute>();
```

https://stackoverflow.com/a/25109103

Решение без рефлексии

```
static string GetDisplayName(this Color color)
{
    return color switch
    {
        Color.Red => "Красный",
        Color.Green => "Зеленый",
        Color.Blue => "Синий",
        _ => throw new ArgumentOutOfRangeException()
    };
}

var colorDisplayName = Color.Green.GetDisplayName();
```

~~AutoMapper~~

- Ошибки из compile-time переходят в runtime
- Ломается навигация по коду
- Медленнее, чем статический маппинг

https://cezarypiatek.github.io/post/why-i-dont-use-automapper/

https://habr.com/ru/articles/705296/

~~MediatR~~

- Сложная навигация через request-response
- Компилятор не видит использование обработчиков

http://arialdomartini.github.io/mediatr

https://habr.com/ru/post/686278/

Исключения

Библиотеки от Microsoft (ASP.NET, Entity Framework)

Демо

Часть IV
Обработка ошибок (продолжаем бороться за явное)

Проблемы исключений

- В C# не отражены в контракте метода
- Это goto с контекстом

Решение из функциональных языков

Размеченные определения (Discriminated Union)

```
type User = { Id : int; Name : string }

type CreateUserResult =
   | User
   | EmailAlreadyRegistered
   | ParentNotFound
```

Виды результатов

Как мы привыкли думать:

- Успешный результат
- Ошибки

```
class CreateUserResult
EmailAlreadyRegisteredException
ParentNotFoundException
```

Как на самом деле:

- Ожидаемый (успешный) результат
- Ожидаемые ошибки
- Неожидаемые ошибки

```
type CreateUserResult =
    | User
    | EmailAlreadyRegistered
    | ParentNotFound

SqlException
```

Виды результатов более широко

Мы не можем за потребителя решить, что для него ошибка, а что нет

Поэтому правильная классификация результатов:

- Ожидаемые результаты
- Неожиданная ошибка

```
type CreateUserResult =
    | User
    | EmailAlreadyRegistered
    | ParentNotFound
```

gRPC

```
service Users {
  rpc Create(CreateRequest) returns (CreateResponse);
}

message CreateRequest {
  string name = 1;
  string Email = 2;
  google.protobuf.Int32Value parent_id = 3;
}
```

```
message CreateResponse {
    message EmailAlreadyRegistered {}
    message ParentNotFound {}

    oneof result {
      int32 user_id = 1;
      EmailAlreadyRegistered email_already_registered = 2;
      ParentNotFound parent_not_found = 3;
    }
}
```

Чем это хорошо

- В сигнатуре метода явно описана ошибка
- Нужно явно обработать все части размеченного определения
- Добавление нового варианта – ошибка времени компиляции

```
match x with
| User -> printfn "User case"
| EmailAlreadyRegistered -> printfn "EmailAlreadyRegistered case"
| ParentNotFound -> printfn "point"
```

Недостатки

- *Зараженность* методов (как с async-await)
- Повышается вложенность
- До сих пор нет нативной поддержи в языке
- Коллеги не понимают, зачем это нужно

Текущее состояние в C#

В языке:

https://github.com/dotnet/csharplang/discussions/7010

Библиотеки от сообщества:

https://github.com/mcintyre321/OneOf

https://github.com/domn1995/dunet

```
public abstract class CreateUserResult
{
    private CreateUserResult() { }

    public sealed class Created : CreateUserResult
    {
        public UserId Id { get; }
    }

    public sealed class EmailAlreadyRegistered : CreateUserResult { }

    public sealed class ParentNotFound : CreateUserResult { }
}
```

```
CreateUserResult result = ...;
result switch
{
    CreateUserResult.Created created => ...,
    CreateUserResult.EmailAlreadyRegistered => ...,
    CreateUserResult.ParentNotFound => ...,
    _ => throw new SwitchExpressionException()
};
```

Демо

Часть V
Тестирование

Проблемы с тестированием

Если тесты есть, то часто они бесполезные:

- проходят на окружении, которое непонятно как разворачивается
- проверяют правильное использование сторонних классов вместо бизнес-логики
  - например, проверяют получение данных (IQuerable, HttpClient) вместо операций с этими данными
  - мокают, не учитывая особенности окружения (InMemoryDb вместо настоящей базы)

Интеграционные тесты

- Реальное окружение (БД, очереди, …) в докере
- Запуск приложения в докере на CI/CD
- Полный прогон ASP.NET пайплайна

- Не замена юнит-тестам!

Плюсы

- Описание окружения в репозитории
- Можно легко разрабатывать по TDD
- Тест-кейсы для быстрого воспроизведения бага
- Примеры использования API

Минусы

- Тесты долго выполняются, их сложнее параллелить
- Такие тесты сложнее и дольше писать

Демо

Часть VI
Заключение

О чем сегодня поговорили

- Архитектура приложений
- Стратегические и тактические паттерны
- Обработка ошибок
- Тестирование

Выводы

Делать проще и явно!

Контакты

- https://t.me/mister_m0j0
- https://github.com/m0j0/architecture-demo