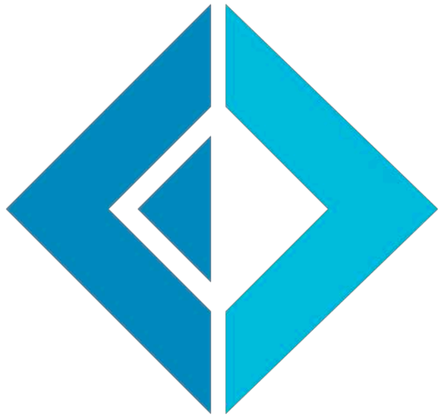
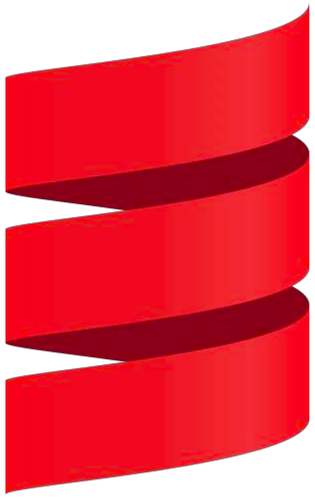
Revolut

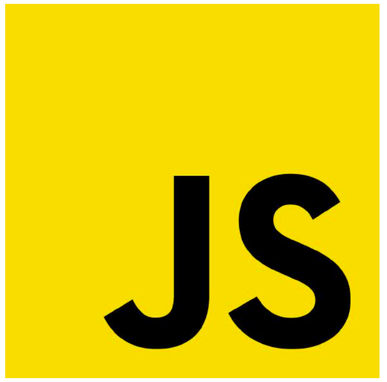
Почему ваша архитектура - функциональная

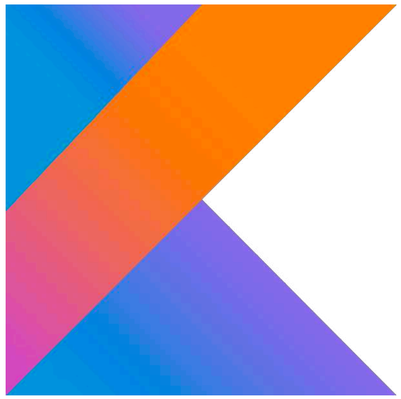
(и как с этим жить)

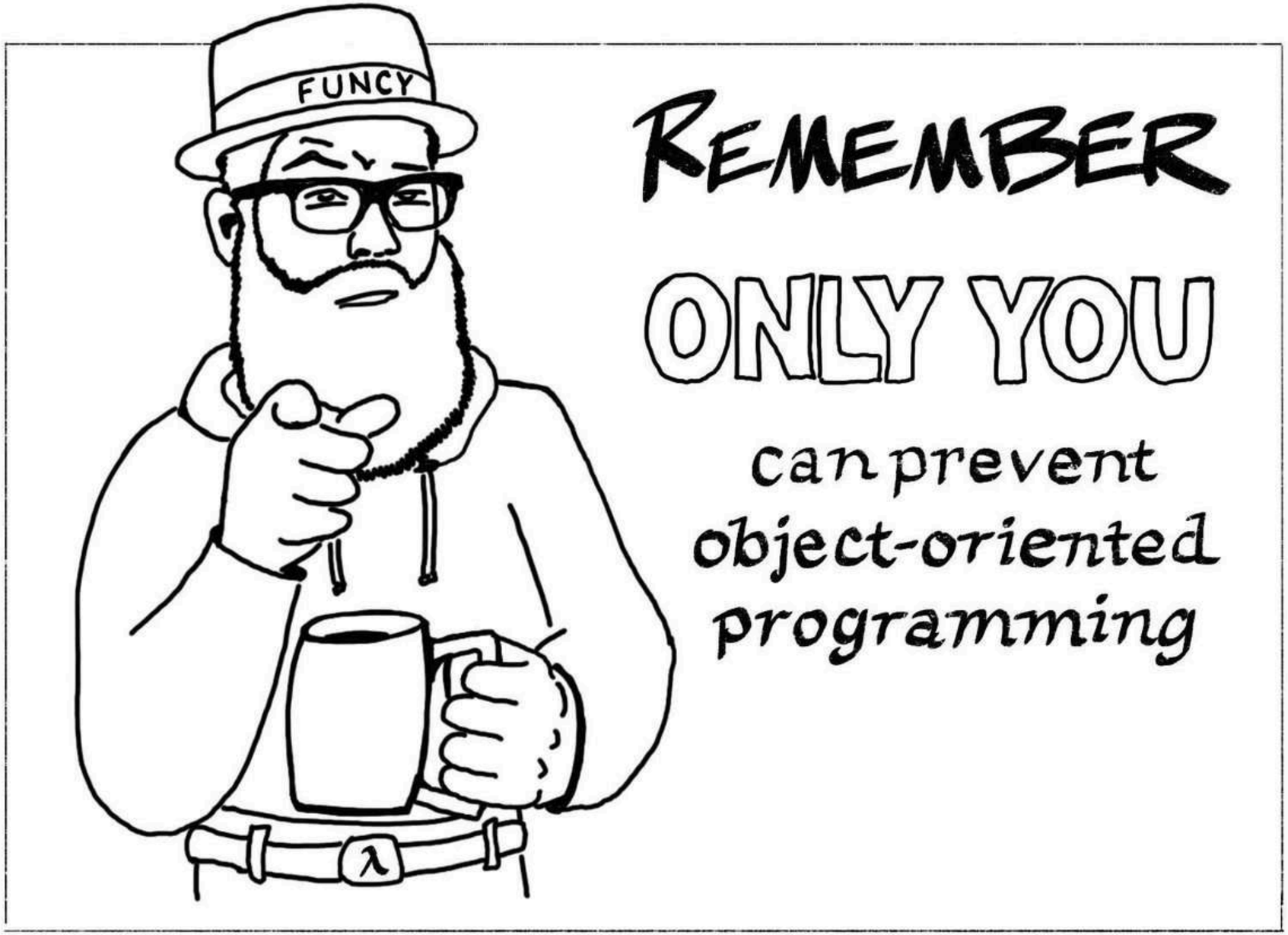
Здравствуйте, меня зовут Роман
и я люблю функциональное программирование




JS

php



FUNCY
λ
REMEMBER
ONLY YOU
can prevent
object-oriented
programming
@Reverent Geek

На самом деле нет

ООП неплохо взаимодействует с функциональным подходом

О чем мы сегодня?

- Что такое функциональное программирование и почему оно важно прямо сейчас

О чем мы сегодня?

- Что такое функциональное программирование и почему оно важно прямо сейчас
- Какие идеи возникли вокруг функционального подхода и как применить их в каждодневной разработке

О чем мы сегодня?

- Что такое функциональное программирование и почему оно важно прямо сейчас
- Какие идеи возникли вокруг функционального подхода и как применить их в каждодневной разработке
- Как эти идеи используются в современной архитектуре

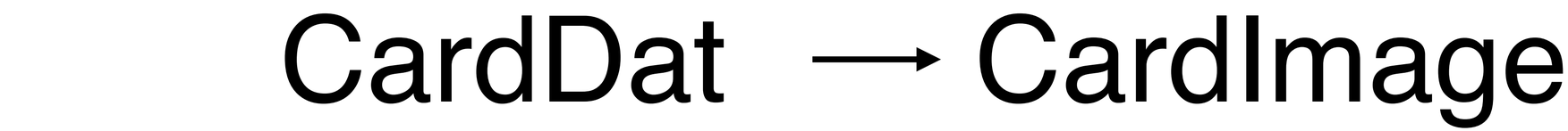
Часть первая

Почему вдруг функционального программирования стало так много
(и что это вообще такое)

Что такое функциональное программирование?

Функциональное программирование — парадигма, в которой ход выполнения программы описывается переходами данных между различными состояниями

ID → CardData


CardDat
CardImage

CardImage⟶ Response


Request → ID
ID → CardData
CardData → CardImage
CardImage → Response

Request → Response

Чем характерна чистая функция?

- Не изменяет состояние

Чем характерна чистая функция?

- Не изменяет состояние
- Не производит побочных эффектов

Чем характерна чистая функция?

- Не изменяет состояние
- Не производит побочных эффектов
- Выдает одинаковый результат «на выходе» при одинаковых входных параметрах

И за это мы получаем

- Параллельность «из коробки»

И за это мы получаем

- Параллельность «из коробки»
- Простота отладки и тестирования

И за это мы получаем

- Параллельность «из коробки»
- Простота отладки и тестирования
- Отказоустойчивость

И за это мы получаем

- Параллельность «из коробки»
- Простота отладки и тестирования
- Отказоустойчивость
- Развертывание без перезапуска

В какой-то момент все это стало очень важным


intel
Core™ i7


aws

Конкурентность — это сложно.
И дорого.


You don't need to control shared state
If you don't have shared state

Итого, функциональное программирование помогает нам решать современные проблемы масштабирования

Итого, функциональное программирование помогает нам решать современные проблемы масштабирования

Часть вторая

Интересные идеи из функционального мира (и способы их использовать вне этого мира)

Функция как объект

```
users.Where(x => x.RegistrationDate.Year <= 2018)
```

```
Func<User, bool> filter =
    x => x.RegistrationDate.Year <= 2018
```

```
interface ICombiner
{
    int Combine(int a, int b);
}

class Multiplier : ICombiner
{
    public int Combine(int a, int b)
    {
        return a * b;
    }
}

class Adder : ICombiner
{
    public int Combine(int a, int b)
    {
        return a + b;
    }
}
```

А что, если представить высокоуровневый код как чистую функцию?

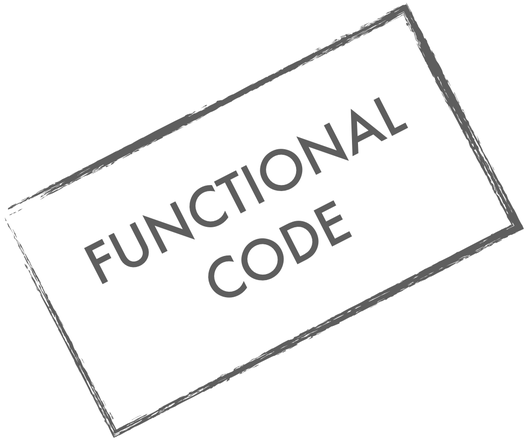
Request → Response


Request → ID
ID → CardData
CardData → CardImage
CardImage → Response

```
ID id = GetId(request);
CardData card = FindCard(id);
CardImage image = BuildCardImage(card);
Response response = ToResponse(image);
```

```
var response = GetId(request)
    .Bind(FindCard)
    .Bind(BuildCardImage)
    .Bind(ToResponse);
```

```
var response = GetId(request)
    .Bind(FindCard)
    .Bind(BuildCardImage)
    .Bind(ToResponse);
```



jbogard / MediatR

Watch 237 | Star 3,298 | Fork 474

Code | Issues 5 | Pull requests 2 | Projects 0 | Wiki | Insights

Dismiss

Join GitHub today

GitHub is home to over 36 million developers working together to host and review code, manage projects, and build software together.

Sign up

Simple, unambitious mediator implementation in .NET

342 commits | 4 branches | 17 releases | 36 contributors | Apache-2.0

Branch: master | New pull request | Find File | Clone or download

jbogard Merge pull request #368 from dariogriffo/PostProcessorCancellationToken ... Latest commit 8e0c349 on 26 Feb

.vscode	Add editor settings for VSCode and update Structuremap	3 years ago
assets/logo	Added various logos	4 years ago
samples	Add CancellationToken to IRequestPostProcessor	3 months ago
src/MediatR	Add CancellationToken to IRequestPostProcessor	3 months ago
test/MediatR.Tests	Add CancellationToken to IRequestPostProcessor	3 months ago
.editorconfig	Add editor settings for VSCode and update Structuremap	3 years ago
.gitattributes	Create .gitattributes	5 years ago
.gitignore	added: D...	a year ago
Build.ps1	Updating packages and marking for release	a year ago

github.com/jbogard/MediatR

А как насчет логов, авторизации и прочей валидации?


Request
ValidatedRequ

```
public ValidatedRequest Validate(Request request)
{
    if (request.IsInvalid)
    {

    }
    return new ValidatedRequest(request);
}
```


WARNING
F# INSIDE

```
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
    | Failure of 'TFailure
```

```fsharp
let bind<'T,'TError> result next =
    match result with
    | Success(value) -> next value
    | Failure(error) -> Failure error
```

Плохие новости — на C# это выглядит неоднозначно :(

vkhorikov 19 сентября 2015 в 08:19

Functional C#: работа с ошибками

Программирование, .NET, C#, ООП, Функциональное программирование

В этой части мы рассмотрим как иметь дело со сбоями и ошибками ввода в функциональном стиле.

- Functional C#: Immutability
- Functional C#: Primitive obsession
- Functional C#: Non-nullable reference types
- **Functional C#: работа с ошибками**

Работа с ошибками в C#: стандартный подход

Концепция валидации и обработки ошибок хорошо отработана, но код, необходимый для этого, может быть весьма неуклюжим в таких языках как C#. Эта статья написана под впечатлением от Railway Oriented Programming — идеи, представленной Скотом Влашиным (Scott Wlaschin) в его презентации на NDC Oslo.

Хорошие новости — можно просто кидаться исключениями :)

```
public ValidatedRequest Validate(Request request)
{
    if (request.IsInvalid)
    {
        throw new InvalidRequestException(request);
    }
    return new ValidatedRequest(request);
}
```

```
public long ValidateAmount(long amount)
{
    if (amount < 0 || amount > MAX_AMOUNT)
        throw new InvalidAmountException(amount);
    return amount;
}
```

```
public class InvalidAmountException : ValidationException
{
    public InvalidAmountException(long amount) : base(
        "Invalid operation amount",
        new Dictionary<string, object> {{"amount", amount}}
    ) { }
}
```

```
public class ValidationException : BusinessException
{
    public ValidationException(string errorMessage,
        Dictionary<string, object> payload) :base(
        HttpStatusCode.BadRequest,
        errorMessage,
        payload) { }
}
```

```
public abstract class BusinessException : Exception
{
    public readonly HttpStatusCode HttpErrorCode;
    public readonly string ErrorMessage;
    public readonly Dictionary<string, object> Payload;

    protected BusinessException(HttpStatusCode httpErrorCode,
        string errorMessage, Dictionary<string, object> payload)
    {
        HttpErrorCode = httpErrorCode;
        ErrorMessage = errorMessage;
        Payload = payload;
    }
}
```

```
var response = request
    .Bind(Authorize)
    .Bind(Validate)
    .Bind(GetId)
    .Bind(FindCard)
    .Bind(BuildCardImage)
    .Bind(ToResponse);
```

```
var response = request
    .Bind(Authorize)
    .Bind(Validate)
    .Bind(GetId)
    .Bind(FindCard)
    .Bind(Log)
    .Bind(BuildCardImage)
    .Bind(ToResponse);
```

Как все-таки взаимодействовать с внешним миром?

Немного выводов

- Всевозможные валидации и авторизации можно сделать частью пайплайна

Немного выводов

- Всевозможные валидации и авторизации можно сделать частью пайплайна
- Самый простой способ обработки ошибок — правильные бизнес исключения (но есть варианты)

Немного выводов

- Всевозможные валидации и авторизации можно сделать частью пайплайна
- Самый простой способ обработки ошибок — правильные бизнес исключения (но есть варианты)
- Для получения всех преимуществ функционального пайплайна нужно избавиться от побочных эффектов

Немного выводов

- Всевозможные валидации и авторизации можно сделать частью пайплайна
- Самый простой способ обработки ошибок — правильные бизнес исключения (но есть варианты)
- Для получения всех преимуществ функционального пайплайна нужно избавиться от побочных эффектов
- Но можно получать не все

Полнота функций

```
public CardData FindCard(Guid id)
{
    if (cards.ContainsKey(id))
    {
        return cards[id];
    }
    else
    {
        return null;
    }
}
```

```
public Optional<CardData> FindCard(Guid id)
{
    if (cards.ContainsKey(id))
    {
        return Optional.Some(cards[id]);
    }
    else
    {
        return Optional.None();
    }
}
```

```
public CardData? FindCard(Guid id)
{
    if (cards.ContainsKey(id))
    {
        return cards[id];
    }
    else
    {
        return null;
    }
}
```

```
public CardData? FindCard(Guid id)
{
    if (cards.ContainsKey(id))
    {
        return cards[id];
    }
    else
    {
        return null;
    }
}
```

На самом деле, стало лучше!

- Сигнатура функции более точна

На самом деле, стало лучше!

- Сигнатура функции более точна
- К нашим услугам все фичи, связанные с nullability

```
CardData card = FindCard(pan)
                ?? throw new CardNotFoundException(pan);
```

На самом деле, стало лучше!

- Сигнатура функции более точна
- К нашим услугам все фичи, связанные с nullability
- Теперь получить null reference чуть сложнее

Nullable contexts

- enable
- disable
- safeonly
- warnings
- safeonlywarnings

docs.microsoft.com/en-us/dotnet/csharp/nullable-references


Не хотите немного монад?

Забавный факт, Optional — это монада

Забавный факт, Optional — это монада

Result, кстати, тоже

Монады нужны для связывания цепочки преобразований

```fsharp
let bind<'T,'TError> result next =
    match result with
    | Success(value) -> next value
    | Failure(error) -> Failure error
```

Optional (или Nullable) — это промежуточное состояние

```
public class User
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public DateTime Birthdate { get; set; }
    public string? Email { get; set; }
}
```

```
class User
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public DateTime Birthdate { get; set; }
}

class ValidatedUser
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Surname { get; set; }
    public DateTime Birthdate { get; set; }
}
```

Модели вместо Optional полей

- Уменьшается количество проверок

Модели вместо Optional полей

- Уменьшается количество проверок
- Типы в коде становятся более «описательными»

Модели вместо Optional полей

- Уменьшается количество проверок
- Типы в коде становятся более «описательными»
- Сложнее передать неверное значение

Модели вместо Optional полей

- Уменьшается количество проверок
- Типы в коде становятся более «описательными»
- Сложнее передать неверное значение
- Приходится писать больше моделей

```
public CardData? FindCard(Guid id)
```

```
public CardData? FindCard(PAN pan)
```

Обертки вместо примитивов

- Мы точнее описываем назначение функции

```
public CardData? FindCard(PAN pan)

public CardData? FindCard(CardToken id)
```

Обертки вместо примитивов

- Мы точнее описываем назначение функции
- Передать неверное значение намного сложнее

Обертки вместо примитивов

- Мы точнее описываем назначение функции
- Передать неверное значение намного сложнее
- В создание типа можно добавить валидацию

```
public long ValidateAmount(long amount)
{
    if (amount < 0 || amount > MAX_AMOUNT)
        throw new InvalidAmountException(amount);
    return amount;
}
```

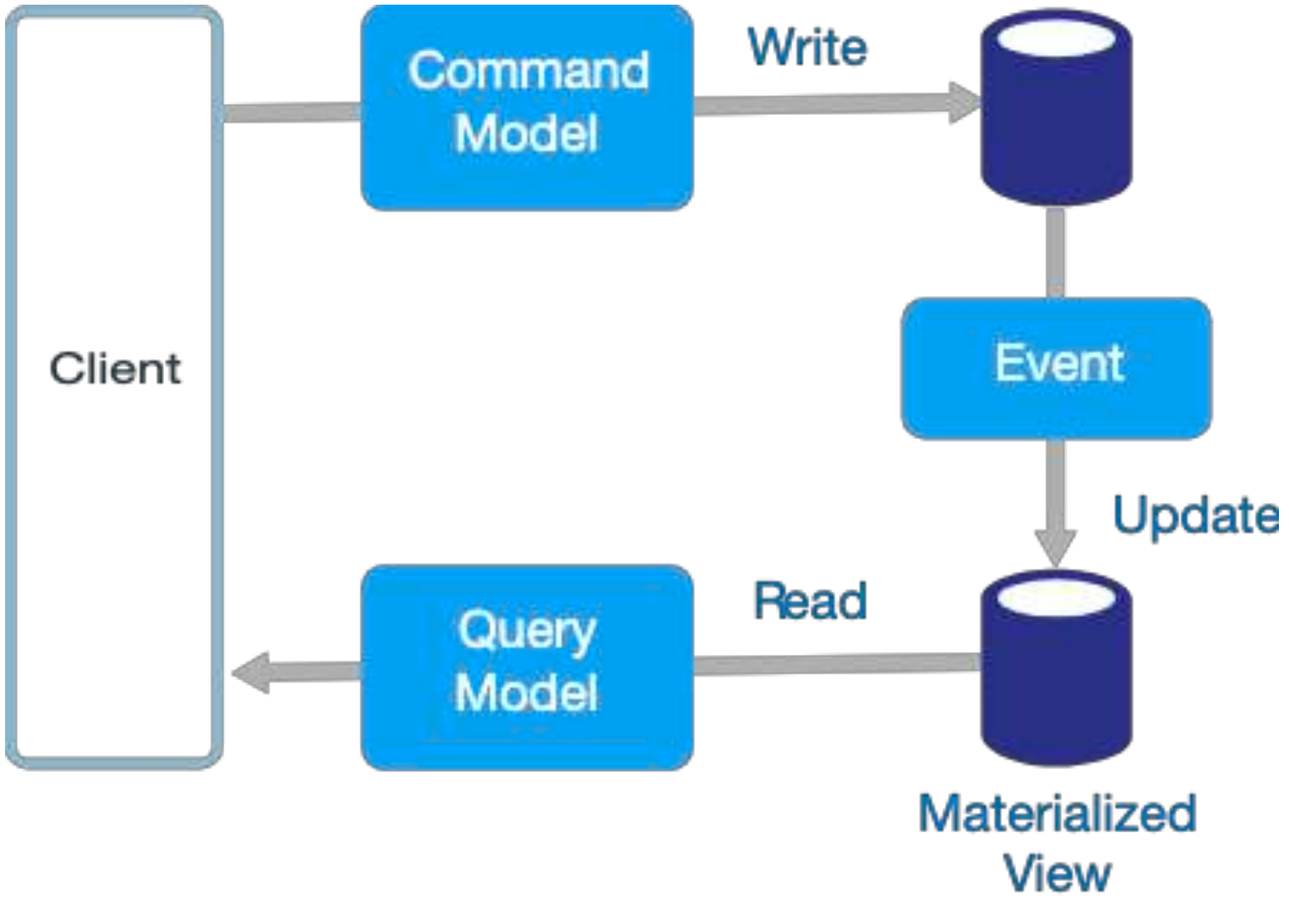
```
public class Amount
{
    private const long MAX_AMOUNT = 100;
    public readonly long Value;

    public Amount(long value)
    {
        if (value < 0 || value > MAX_AMOUNT)
            throw new InvalidAmountException(value);
        Value = value;
    }
}
```

А еще наш код становится ближе к иммутабельности

Часть третья

Как функциональное программирование используется в архитектуре
(и что мы можем сделать не так)

CQRS


Client
Command Model
Write
Event
Update
Query Model
Read
Materialized View

CQRS

- Код с побочными эффектами отделен от кода без них

CQRS

- Код с побочными эффектами отделен от кода без них
- Чтение легко масштабируется

CQRS

- Код с побочными эффектами отделен от кода без них
- Чтение легко масштабируется
- Мы можем частично использовать преимущества функционального пайплайна

DOTNEXT 2018 Moscow

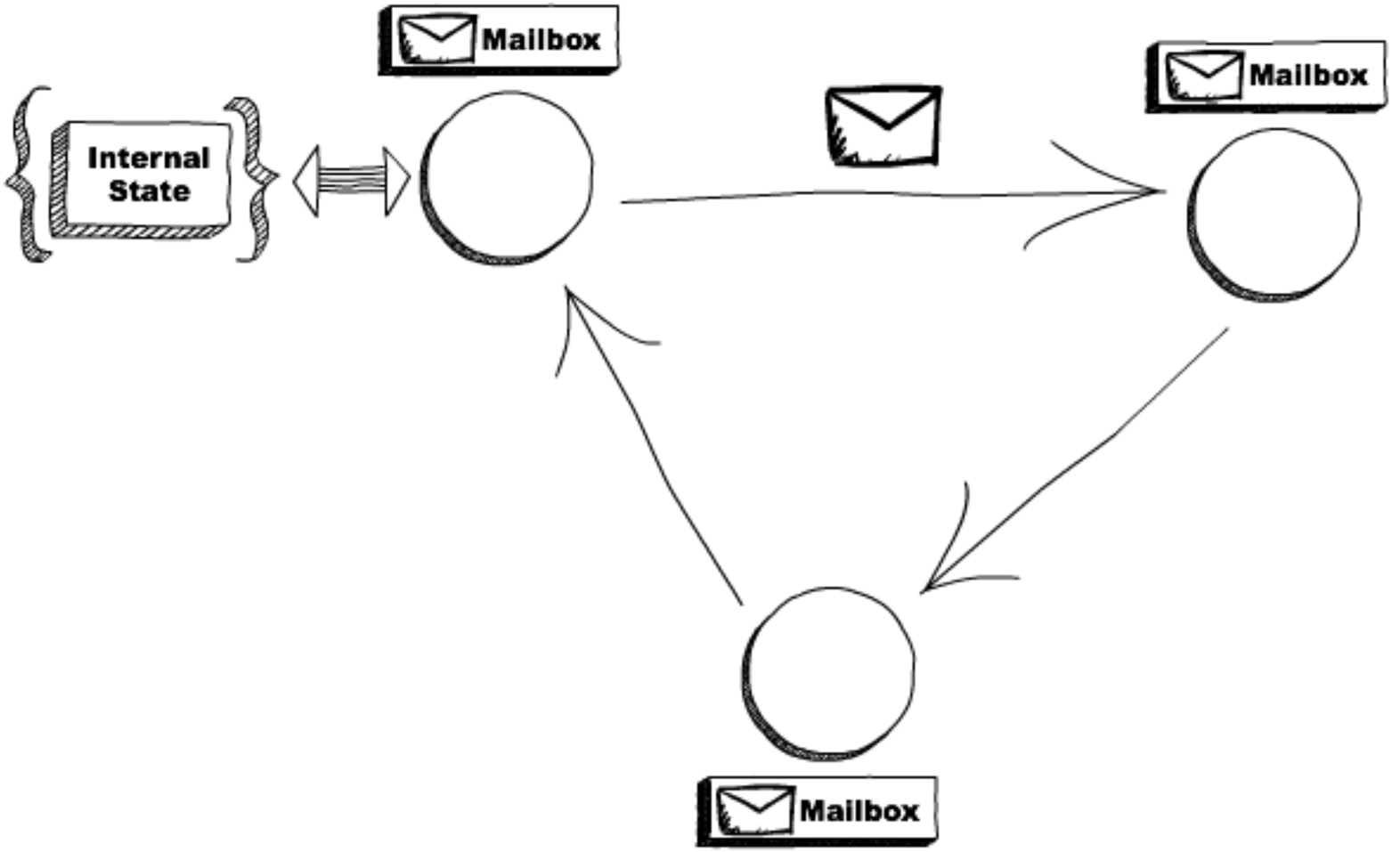
Максим Аршинов

Хайтек Груп

Быстрорастворимое проектирование

youtube.com/watch?v=qJPwSvDLmQE

Actor Model


Mailbox
Internal State
Mailbox
Mailbox

Actor Model

- Актор может служить имплементацией «чистого» пайплайна

Actor Model

- Актор может служить имплементацией «чистого» пайплайна
- Модель позволяет легко изолировать «грязную» логику

Actor Model

- Актор может служить имплементацией «чистого» пайплайна
- Модель позволяет легко изолировать «грязную» логику

DOTNEXT

Moscow 2016

Вагиф Абилов

Miles

Моя жизнь с актерами: опыт внедрения модели актеров на F#

youtube.com/watch?v=wRxO5ky7S8g

DDD

DDD

- Все объекты и их состояния должны соответствовать единому языку

DDD

- Все объекты и их состояния должны соответствовать единому языку
- Состояния должны быть изолированы

DDD

- Все объекты и их состояния должны соответствовать единому языку
- Состояния должны быть изолированы
- Мы описываем переходы между этими состояниями

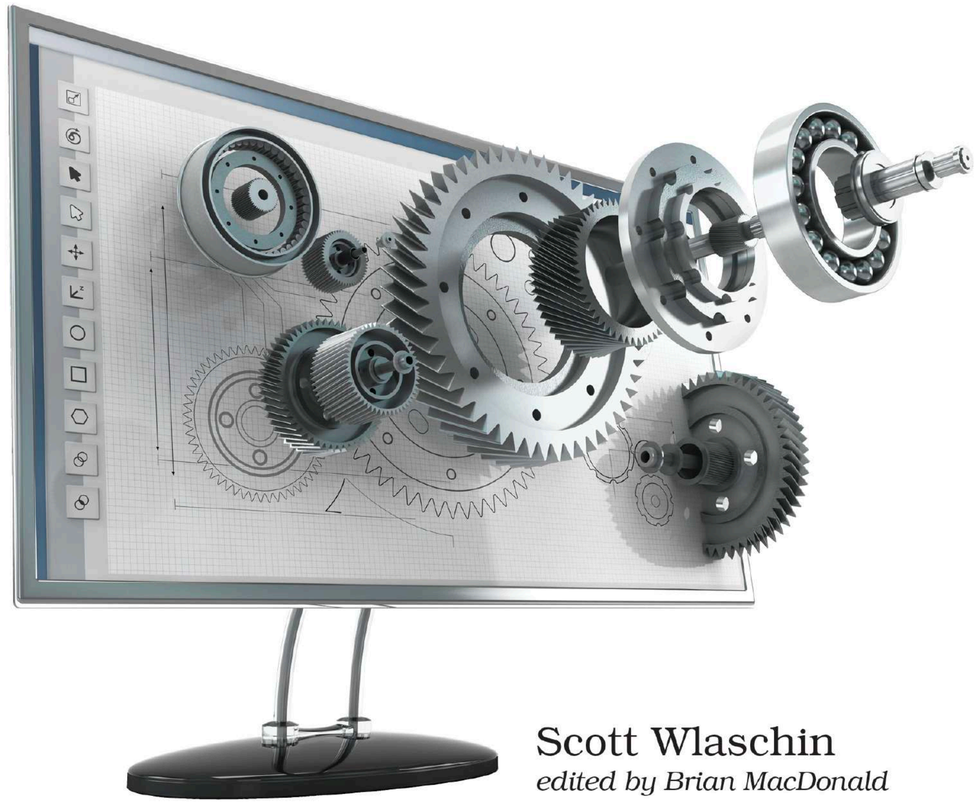
DDD

- Все объекты и их состояния должны соответствовать единому языку
- Состояния должны быть изолированы
- Мы описываем переходы между этими состояниями
- ФП способствуем двум важнейшим целям — избавить бизнес-логику от сторонних зависимостей и корректно описать состояния

The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#



Scott Wlaschin
edited by Brian MacDonald

Проблема

Смешивание «чистого» и «грязного» кода

И за это мы получаем

- Параллельность «из коробки»
- Простота отладки и тестирования
- Отказоустойчивость
- Развертывание без перезапуска

Проблема

Многократная обработка ошибок

Проблема

Использование промежуточного состояния в качестве поля

Проблема

Одержимость примитивами

Итого

- Чтобы воспользоваться отдельными преимуществами функционального подхода, нет нужды писать все функционально

Итого

- Чтобы воспользоваться отдельными преимуществами функционального подхода, нет нужды писать все функционально
- Если изолировать те места, где ваш код «чистый» — можно очень сильно упростить себе масштабирование и многое другое

Итого

- Чтобы воспользоваться отдельными преимуществами функционального подхода, нет нужды писать все функционально
- Если изолировать те места, где ваш код «чистый» — можно очень сильно упростить себе масштабирование и многое другое
- А отдельные идеи из функционального программирования неплохо применяются вне зависимости от стиля

Спасибо!

 nevoroman@gmail.com

 nevoroman