F# ❤️ DDD

Артём Акуляков

.NET Engineer

OCS

- .NET in startups & enterprise
- dotnetru, krydotnet, dev2dev, dotnet & more
- $f(\lambda)$

F# & DDD

C#

- GoF
- GRASP
- SOLID
- DDD
- CQRS
- Event sourcing
- N-layer arch
- Onion arch
- ...

F#

- ???

Готовить DDD не просто

Но на F# чуть легче

DDD

DDD

Создание модели реального предприятия / процесса в программном коде

- устойчивые абстракции
- самодокументируемый код

DDD

Стратегия

- Ubiquitous Language
- Domain
- Subdomain
- Bounded context
- Anticorruption layer
- ...

Тактика

- Value object
- Entity
- Aggregate
- Domain service
- Domain events
- Repository
- ...

DDD

Стратегия

- **Ubiquitous Language**
- **Domain**
- **Subdomain**
- **Bounded context**
- **Anticorruption layer**
- **...**

Тактика

- Value object
- Entity
- Aggregate
- Domain service
- Domain events
- Repository
- ...

Ubiquitous language / Единый язык

DDD

Стратегия

- Ubiquitous Language
- Domain
- Subdomain
- Bounded context
- Anticorruption layer
- ...

Тактика

- **Value object**
- **Entity**
- **Aggregate**
- **Domain service**
- **Domain events**
- **Repository**
- **...**

Тактика

Инструментарий

ООР

- классы и объекты
- графы объектов
- интерфейсы
- обмен сообщениями / вызов методов

&

- инкапсуляция
- полиморфизм
- наследование

OOP

- классы и объекты
- графы объектов
- интерфейсы
- обмен сообщениями / вызов методов

&

- инкапсуляция
- полиморфизм
- наследование

FP

- функции
- типы
- композиция

Инструментарий F#

Инструментарий

- функции это тоже значения

Инструментарий

- функции это тоже значения
- тип это допустимое множество значений

Инструментарий

```
// Count -> Discount
let calculateDiscount (ordersCount: Count): Discount =...
```

Инструментарий

- функции это тоже значения
- тип это допустимое множество значений
- мы можем:
  - **объединять функции чтобы получать новые функции**

Композиция функций

```
// Order list -> Count
let getFinishedOrdersCount (orders: Order list): Count =...
```

Композиция функций

```
// Count -> Discount
let calculateDiscount (ordersCount: Count): Discount =...
```

Композиция функций

```
// Order list -> Discount
let calculateDiscountByPurchases (orders: Order list): Discount =...
```

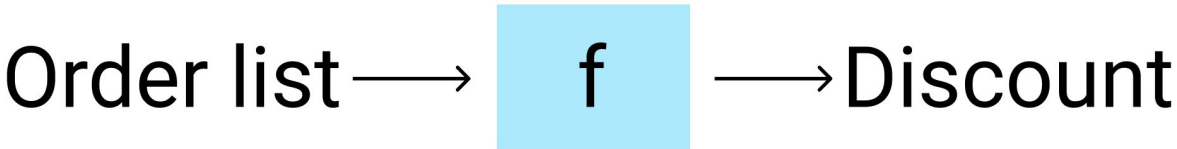
Композиция функций

```
// Order list -> Count
let getFinishedOrdersCount (orders: Order list): Count =...


// Count -> Discount
let calculateDiscount (ordersCount: Count): Discount =...
```

Композиция функций

Композиция функций

```
// Order list -> Discount
let calculateDiscountByPurchasesHistory =
    getFinishedOrdersCount >> calculateDiscount
```



Композиция функций

```
orders
|> getFinishedOrdersCount
|> calculateDiscount
```

Инструментарий

- функции это тоже значения
- тип это допустимое множество значений
- мы можем
  - объединять функции чтобы получать новые функции
  - **каррировать функции**

Каррирование

```
// bool -> Count -> Discount
let calculateDiscount (customerStatus: CustomerStatus)
                      (ordersCount: Count): Discount = ...
```

Compilation error

```
// Order list -> Discount
let calculateDiscountByPurchasesHistory =
    getFinishedOrdersCount >> calculateDiscount
```

Каррирование

```
// bool -> Count -> Discount
let calculateDiscount (customerStatus: CustomerStatus)
                      (ordersCount: Count): Discount = ...
```

Каррирование

```
// bool -> Count -> Discount
let calculateDiscount (customerStatus: CustomerStatus)
                      (ordersCount: Count): Discount = ...


// Count -> Discount & customerStatus = VIP
let calculateDiscountForVip = calculateDiscount CustomerStatus.VIP
```

Каррирование

```
// Order list -> Discount
let calculateDiscountByPurchasesHistory =
    getFinishedOrdersCount >> (calculateDiscount CustomerStatus.VIP)
```

Каррирование

```
// Order list -> Discount
let calculateDiscountByPurchasesHistory =
    getFinishedOrdersCount >> calculateDiscountForVip
```

Инструментарий

- функции это тоже значения
- тип это допустимое множество значений
- мы можем
  - объединять функции чтобы получать новые функции
  - каррировать функции
  - **объединять типы чтобы получать новые типы**

Модель оплаты заказа: сумма в некой валюте и способ оплаты

- наличные
- банковская карта (необходимы реквизиты)
- кредит (необходим номер кредитной линии)
- предоплата

Композиция типов

```
type Payment = {
    Amount: uint32
    Currency: Currency
    Method: PaymentMethod
}
```

Композиция типов

```
type Payment = {
    Amount: uint32
    Currency: Currency
    Method: PaymentMethod
}
```

Оплата **это**

Сумма **и** Валюта **и** Способ оплаты

Композиция типов

```
type PaymentMethod =
    | Cash
    | Card of BankCardRequisites
    | Credit of CreditLine
    | Prepaid
```

Композиция типов

```
type PaymentMethod =
    | Cash
    | Card of BankCardRequisites
    | Credit of CreditLine
    | Prepaid
```

Способ оплаты **это**

- Наличные **или**
- Банковская карта с реквизитами **или**
- Кредит по кредитной линии **или**
- Предоплата

```
public interface IPaymentMethod { ... }

public class Cash : IPaymentMethod { ... }
public class Card : IPaymentMethod { ... }
public class Credit : IPaymentMethod { ... }
public class Prepaid : IPaymentMethod { ... }
```

Тактика

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Какие поля обязательны?

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Ограничения?
Формат?

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

```
let sale = order.Buyer / 100
```

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Связанные поля?

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Еще связанные поля?

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Имеет ли смысл заказ без строк?

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Ограничения?

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

```
let increaseQuantity line addQty =
 {
   Product = line.Quantity
   Quantity = line.Product + addQty
 }
```

Отражает ли тип требования?

Type oriented design

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime

    IsShipped: bool
    ShipmentDate: DateTime
    TrackingNumber: string

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

```
type Option<'T> =
    | Some of 'T
    | None
```

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: string option

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

```
type Order = {
    Number: string
    Buyer: int
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: string option

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: int
    Quantity: int
}
```

Wrap primitive types

```
type ProductId = ProductId of int
type Quantity = Quantity of int
type BuyerId = BuyerId of int
type TrackingNumber = TrackingNumber of string
type OrderNumber = OrderNumber of string
...
```

Compilation error

```
let sale = order.Buyer / 100
```

```
let increaseQuantity line addQty =
 {
   Product = line.Quantity
   Quantity = line.Product + addQty
 }
```

Reuse types

```
let readOrder (con: DbConnection) (orderId: OrderNumber) = ...

type OrderInvoice = {
    Order: OrderNumber
    Total: Money
    ...
}
```

```
let number = OrderNumber ""
```

Make invalid data unrepresentable

```fsharp
type OrderNumber = private OrderNumber of string
   with
       static member create (str: string) =
           if str.Length = 10 &&
              str |> Seq.forall (fun ch -> Char.IsDigit ch || Char.IsLetter ch)
           then Ok(OrderNumber str)
           else Error "Invalid order number..."

       static member unwrap (n: OrderNumber) =
           let (OrderNumber str) = n in str
```

```fsharp
type OrderNumber = private OrderNumber of string
    with
        static member create (str: string) =
            if str.Length = 10 &&
               str |> Seq.forall (fun ch -> Char.IsDigit ch || Char.IsLetter ch)
            then Ok(OrderNumber str)
            else Error "Invalid order number..."

        static member unwrap (n: OrderNumber) =
            let (OrderNumber str) = n in str
```

```
type OrderNumber = private OrderNumber of string
   with
       static member create (str: string) =
           if str.Length = 10 &&
              str |> Seq.forall (fun ch -> Char.IsDigit ch || Char.IsLetter ch)
           then Ok(OrderNumber str)
           else Error "Invalid order number..."

       static member unwrap (n: OrderNumber) =
           let (OrderNumber str) = n in str
```

```
type OrderNumber = private OrderNumber of string
   with
       static member create (str: string) =
           if str.Length = 10 &&
              str |> Seq.forall (fun ch -> Char.IsDigit ch || Char.IsLetter ch)
           then Ok(OrderNumber str)
           else Error "Invalid order number..."

       static member unwrap (n: OrderNumber) =
           let (OrderNumber str) = n in str
```

```
type Order = {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

```
type Order = {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

```
type Order = private {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
with
    static member create ... = ...
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

```
type Order = private {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
with
    static member create ... = ...
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

```
type Order = private {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
with
    static member create ... = ...
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

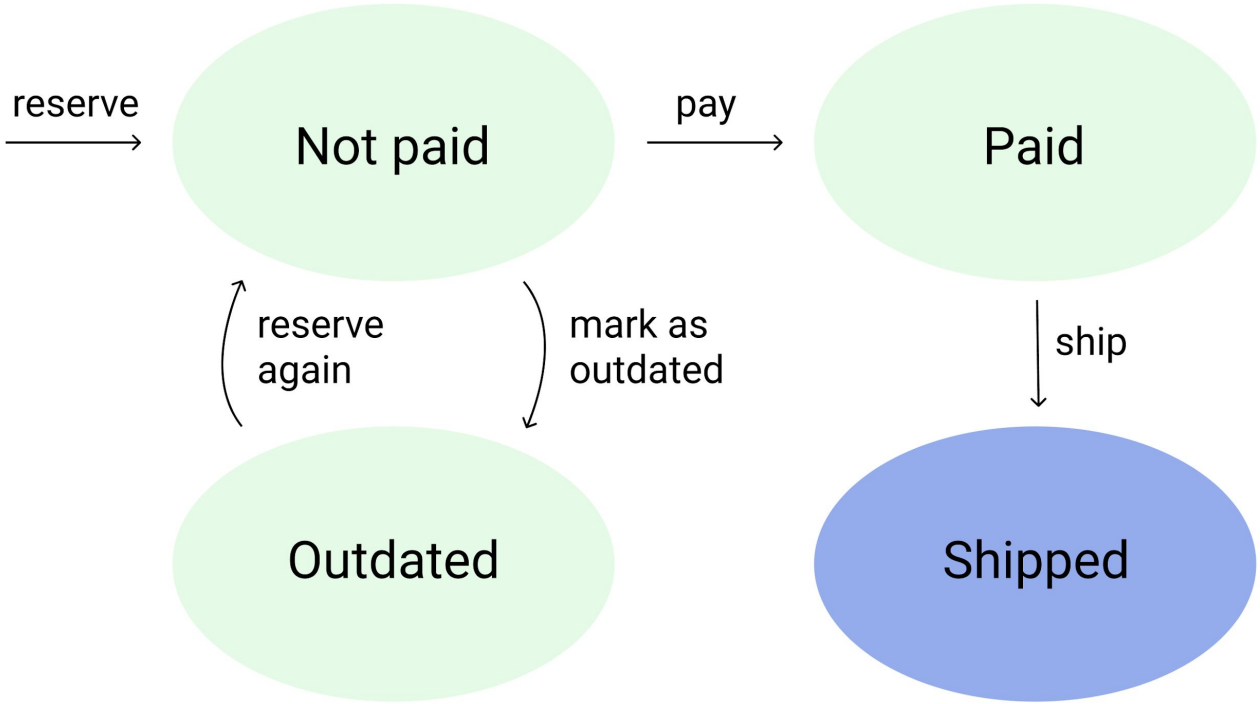
```
type Order = private {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
with
    static member create ... = ...
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

```
type Order = private {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime

    IsPaid: bool
    IsOutdated: bool
    PaymentDate: DateTime option

    IsShipped: bool
    ShipmentDate: DateTime option
    TrackingNumber: TrackingNumber option

    Lines: OrderLine list
}
with
    static member create ... = ...
```

```
type OrderLine = {
    Product: ProductId
    Quantity: Quantity
}
```

```
let payOrder (order: Order) =
    ...
    if not order.IsPaid && not order.IsOutdated then
        failwith "Order..."
    ...
```


reserve
Not paid
pay
Paid
reserve
again
mark as
outdated
ship
Outdated
Shipped

Eliminate implicit states

```
type OrderInformation = {
    Number: OrderNumber
    Buyer: BuyerId
    Created: DateTime
    Lines: OrderLine list
}
```

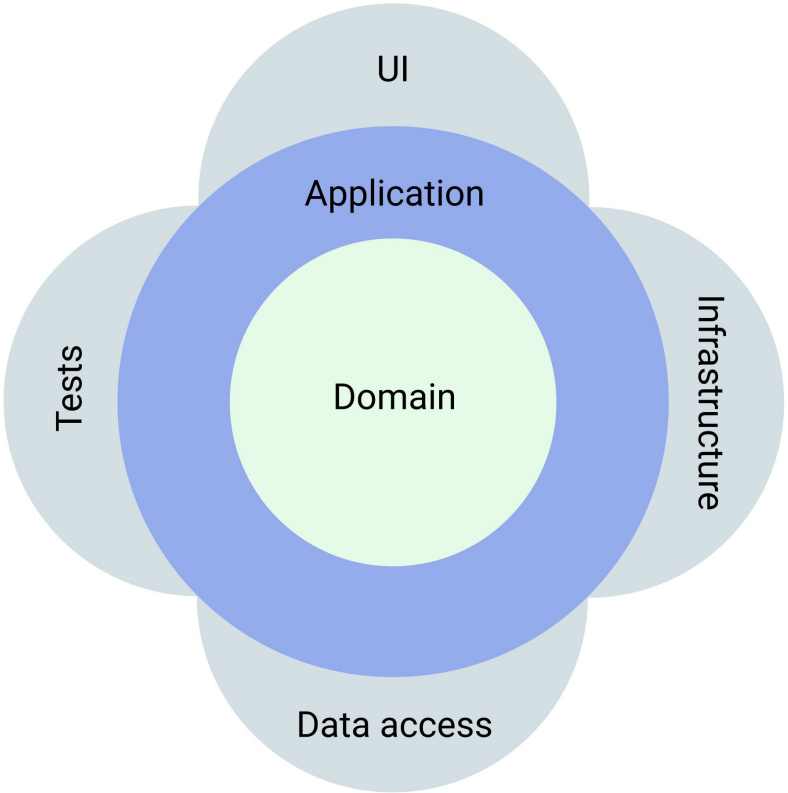
```
type PaidOrder = {
    Order: OrderInformation
    PaymentDate: DateTime
}

type ShippedOrder = {
    Order: OrderInformation
    ShipmentDate: DateTime
    TrackingNumber: TrackingNumber
}

type NotPaidOrder = OrderInformation
type OutdatedOrder = OrderInformation
```

```
type Order =
    | NotPaid of NotPaidOrder
    | Outdated of OutdatedOrder
    | Paid of PaidOrder
    | Shipped of ShippedOrder
```

```
let payOrder (order: NotPaidOrder) : PaidOrder = ...
```

```
let payOrder (order: NotPaidOrder) : PaidOrder = ...
```

Бизнес-логика


UI
Application
Domain
Tests
Infrastructure
Data access


Command
Validation
Security
Domain Logic
Response

```
let processRequest req =
    req
    |> parseRequest
    |> validate
    |> authenticate
    |> doBusinessLogic
    |> (createResponse req)
```

```
let processRequest req =
    req
    |> parseRequest
    |> validate
    |> authenticate
    |> doBusinessLogic
    |> (createResponse req)
```

Управление зависимостями

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder) : PaidOrder =
   db.readBuyerPaymentInformation order.Buyer
   |> calculateInvoice order
   |> processOrderPayment
   |> createPaidOrder order
   |> db.savePaidOrder
```

```
let payOrder (db: PaymentStorage)
            (processOrderPayment: PaymentProcessor)
            (order: NotPaidOrder) : PaidOrder =
    db.readBuyerPaymentInformation order.Buyer
    |> calculateInvoice order
    |> processOrderPayment
    |> createPaidOrder order
    |> db.savePaidOrder
```

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder) : PaidOrder =
    db.readBuyerPaymentInformation order.Buyer
    |> calculateInvoice order
    |> processOrderPayment
    |> createPaidOrder order
    |> db.savePaidOrder
```

Типы функций как контракты

```
type PaymentProcessor = OrderInvoice -> PaymentConfirmation

type PaymentStorage = {
    readBuyerPaymentInformation: BuyerPaymentInformationReader
    savePaidOrder: PaidOrderSaver
}
```

```
type PaymentProcessor = OrderInvoice -> PaymentConfirmation
```

```
type PaymentStorage = {
    readBuyerPaymentInformation: BuyerPaymentInformationReader
    savePaidOrder: PaidOrderSaver
}
```

```
// OrderInvoice -> PaymentConfirmation
let payAsPrepaid (invoice: OrderInvoice): PaymentConfirmation = ...

// OrderInvoice -> PaymentConfirmation
let payViaTransfer (invoice: OrderInvoice): PaymentConfirmation = ...
```

```
// DbConnection -> OrderInvoice -> PaymentConfirmation
let payAsPrepaid (connection: DbConnection)
                 (invoice: OrderInvoice): PaymentConfirmation = ...

// BankClient -> OrderInvoice -> PaymentConfirmation
let payViaTransfer (client: BankClient)
                   (invoice: OrderInvoice): PaymentConfirmation = ...
```

Каррирование для внедрения зависимостей

```
// DbConnection -> OrderInvoice -> PaymentConfirmation
let payAsPrepaid (connection: DbConnection)
                 (invoice: OrderInvoice): PaymentConfirmation = …
```

```
// DbConnection -> OrderInvoice -> PaymentConfirmation
let payAsPrepaid (connection: DbConnection)
                 (invoice: OrderInvoice): PaymentConfirmation = …

// OrderInvoice -> PaymentConfirmation
let paymentProcessor : PaymentProcessor = payAsPrepaid connection
```

```
// DbConnection -> OrderInvoice -> PaymentConfirmation
let payAsPrepaid (connection: DbConnection)
                 (invoice: OrderInvoice): PaymentConfirmation = …

// OrderInvoice -> PaymentConfirmation
let paymentProcessor : PaymentProcessor = payAsPrepaid connection

let paidOrder = payOrder db paymentProcessor order
```

А если что-то пойдет не так?

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder) : PaidOrder =
    ...
    if ... then
       raise(InvalidOperationException("Not enough money"))
    ...
```

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder) : PaidOrder =
    ...
    if ... then
       raise(InvalidOperationException("Not enough money"))
    ...
```

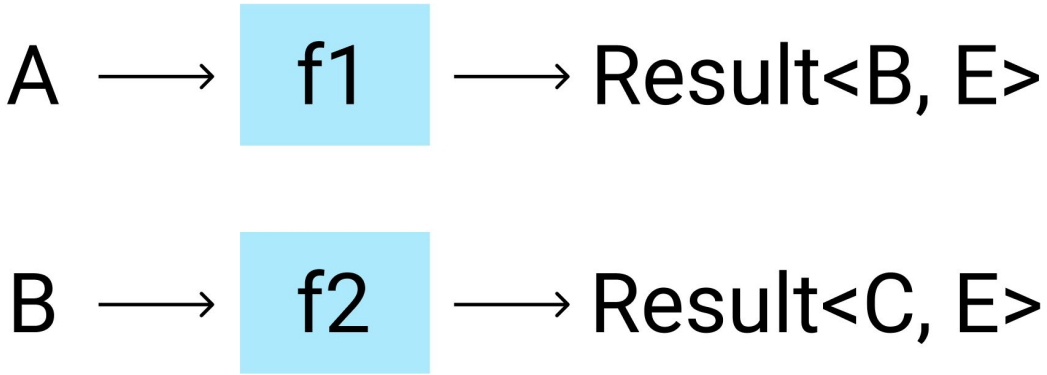
Errors vs Exceptions

```
type Result<'TResult,'TError> =
    | Ok of 'TResult
    | Error of 'TError
```

```
type PaymentOrderError =
    | NoEnoughMoney of ...
    | CreditExceeded of ...
    | ...
```

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder)
                          : Result<PaidOrder, PaymentOrderError> =
    ...
```

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder)
                          : Result<PaidOrder, PaymentOrderError> =
    db.readBuyerPaymentInformation order.Buyer
    |> calculateInvoice order
    |> processOrderPayment
    |> createPaidOrder order
    |> db.savePaidOrder
```

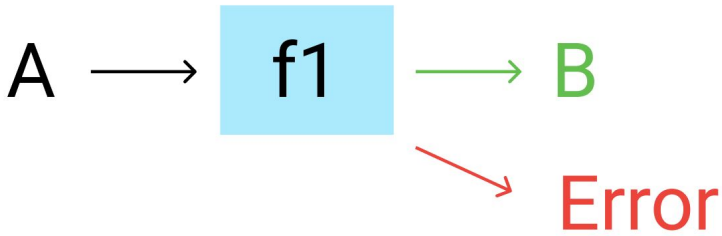

A
f1
Result<B, E>
B
f2
Result<C, E>

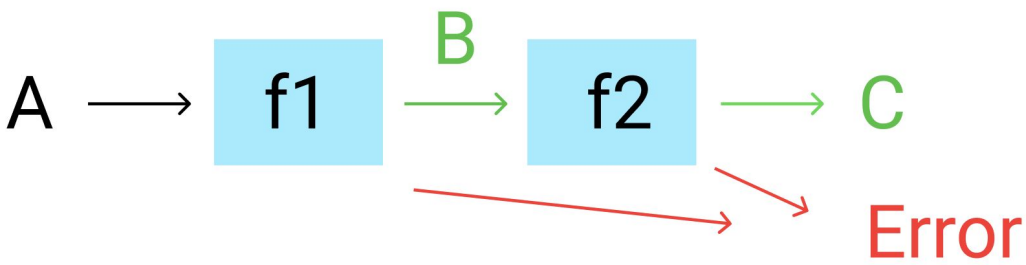
```
let payOrder (db: PaymentStorage)
        (processOrderPayment: PaymentProcessor)
        (order: NotPaidOrder) : Result<PaidOrder, PaymentOrderError> =
   match db.readBuyerPaymentInformation order.Buyer with
   | Ok bpi ->
       match calculateInvoice order bpi with
       | Ok oi ->
           match processOrderPayment oi with
           | Ok pc ->
               match createPaidOrder order pc with
               | Ok po -> db.savePaidOrder po
               | Error e4 -> Error e4
           | Error e3 -> Error e3
       | Error e2 -> Error e2
   | Error e1 -> Error e1
```

```
let payOrder (db: PaymentStorage)
        (processOrderPayment: PaymentProcessor)
        (order: NotPaidOrder) : Result<PaidOrder, PaymentOrderError> =
    match db.readBuyerPaymentInformation order.Buyer with
    | Ok bpi ->
        match calculateInvoice order bpi with
        | Ok oi ->
            match processOrderPayment oi with
            | Ok pc ->
                match createPaidOrder order pc with
                | Ok po -> db.savePaidOrder po
                | Error e4 -> Error e4
            | Error e3 -> Error e3
        | Error e2 -> Error e2
    | Error e1 -> Error e1
```

Railway oriented programming

https://fsharpforfunandprofit.com/rop/


A
f1
B
Error


A
f1
B
f2
C
Error

ROP

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder)
                   : Result<PaidOrder, PaymentOrderError> =
    db.readBuyerPaymentInformation order.Buyer
    |> Result.bind (calculateInvoice order)
    |> Result.bind processOrderPayment
    |> Result.bind (createPaidOrder order)
    |> Result.bind db.savePaidOrder
```

ROP

```
let payOrder (db: PaymentStorage)
             (processOrderPayment: PaymentProcessor)
             (order: NotPaidOrder)
                   : Result<PaidOrder, PaymentOrderError> =
    db.readBuyerPaymentInformation order.Buyer
    >>= (calculateInvoice order)
    >>= processOrderPayment
    >>= (createPaidOrder order)
    >>= db.savePaidOrder
```

Итоги

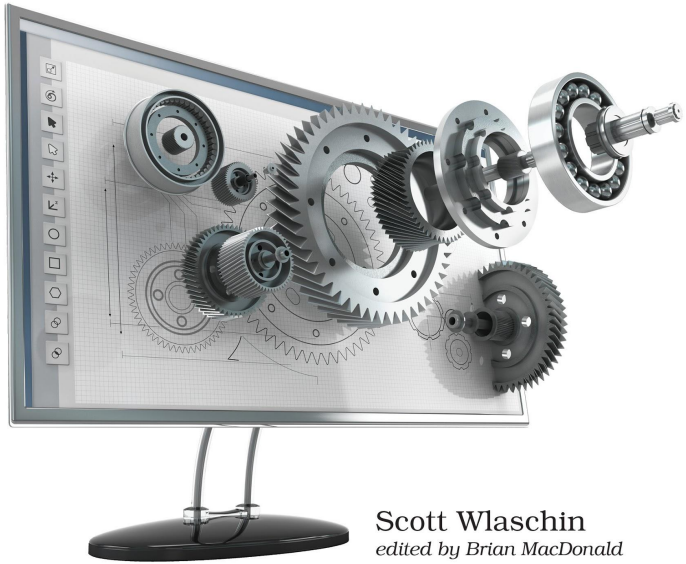
Итоги

- оборачиваем примитивные типы
- исключаем возможность представления неправильных данных
- выделяем состояние явно
- ошибки как часть домена
- используем композицию

The Pragmatic Programmers

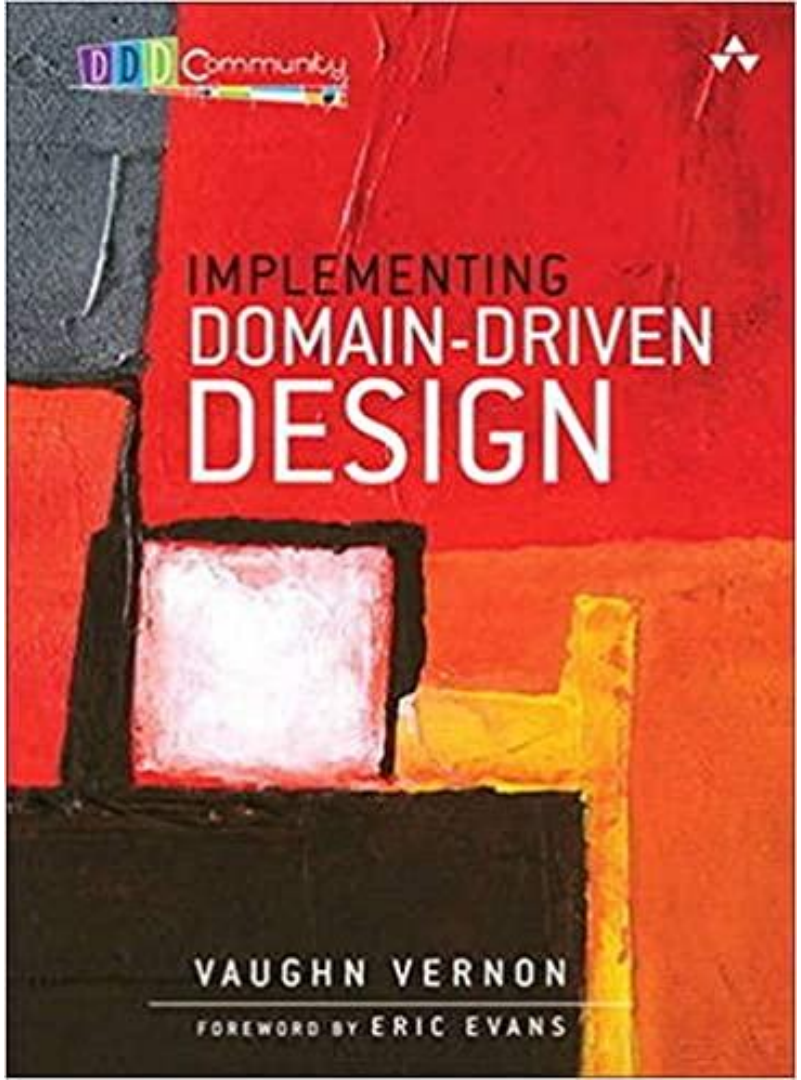
Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#



Scott Wlaschin

edited by Brian MacDonald




DOTNEXT
2019 MOSCOW
Scott Wlaschin
Author of the book "Domain Modeling Made Functional"
The power of composition

DOTNEXT 2018 Moscow

Алексей Мерсон

DotNetRu

Domain-driven design:
рецепт для прагматика

DOTNEXT 2018 Moscow

Максим Аршинов

Хайтек Груп

Быстрорастворимое проектирование

F# ❤️ DDD

Вопросы?