goto; oslo × DFDS

Life After Business Objects

Confessions of an OOP veteran

Vagif Abilov

#GOTOoslo

This talk isn’t about
a war for the one and only
best programming paradigm

We will focus on what may lead pragmatic developers ("pragmatists in pain" *) to the paradigm shift

* Eric Sink "Why your F# evangelism isn't working"
https://ericsink.com/entries/fsharp_chasm.html

Our product


Til nrk.no
NRK TV
Se TV direkte
Kategorier
TV-guide
Søk
Populært
Nytt
Side om side 6:10
Drømmeturen 4:6
Lilyhammer 1:8
TEMA: Andre verdenskrig
Girls 7:12
Aktuelt
Meny

Let’s begin with basics: Modeling a point

Dmitry Ivanov (JetBrains)

Immutable Collections in .NET

```
class Point {

int X { get; set; }
int Y { get; set; }

Point(int x, int y) { X = x; Y = y }

void IncreaseX (int xOffset) { X += xOffset; }
void IncreaseY (int yOffset) { Y += yOffset; }

}
```

```
class Point {

int X { get; set; }
int Y { get; set; }

Point(int x, int y) { X = x; Y = y }

void IncreaseX (int xOffset) { X += xOffset; }
void IncreaseY (int yOffset) { Y += yOffset; }

}
```

```
class Point {

int X { get; set; }
int Y { get; set; }

Point(int x, int y) { X = x; Y = y }

void IncreaseX (int xOffset) { X += xOffset; }
void IncreaseY (int yOffset) { Y += yOffset; }

int GetHashCode() {…}
bool Equals(object other) {…}
}
```

```
class Point {

readonly int X;
readonly int Y;

Point(int x, int y) { X = x; Y = y }

Point IncreaseX (int xOffset) => new Point(x + xOffset, y);
Point IncreaseY (int yOffset) => new Point(x, y + yOffset);

int GetHashCode() {…}
bool Equals(object other) {…}
}
```

Data structures in F#

```
type Point = {
    X : int
    Y : int
}
```

Data structures in F#

```
type Point = {
    X : int
    Y : int
}

let p = { X = 1; Y = 2 }
let q = { p with X = p.X+1 }
```

Consequences of
~~design mistake~~
insufficient experience

Principle difference in initial sets of defaults between OOP and FP

Object Oriented Programming

Empowers
through
variety of choices

Functional Programming

Prevents
unconscious
mistakes

Functional Programming

Path
to
concurrency

Amdahl's law in action

If you have 10 processors
but only 40% of your code can be parallelized,
you will achieve performance gain of 1.56

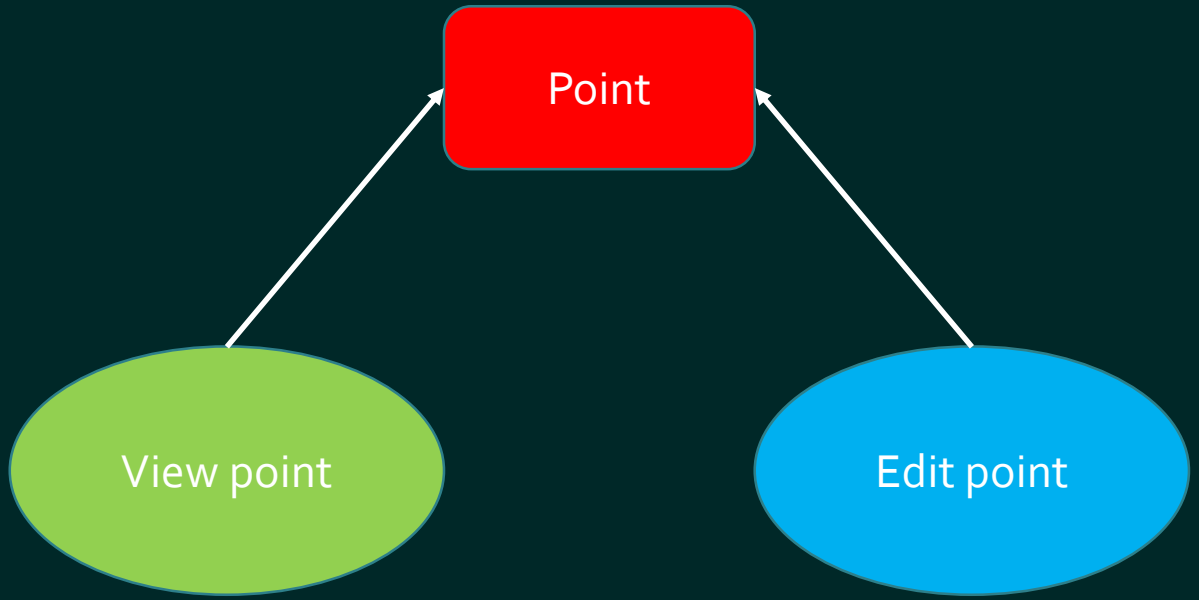
Time to have a closer look at business objects

```
class Point {

readonly int X;
readonly int Y;

Point(int x, int y) { X = x; Y = y }

Point IncreaseX (int xOffset) => new Point(x + xOffset, y);
Point IncreaseY (int yOffset) => new Point(x, y + yOffset);

int GetHashCode() {…}
bool Equals(object other) {…}
}
```

```
class Point {

public readonly int X;
public readonly int Y;

public Point(int x, int y) { X = x; Y = y }

public Point IncreaseX (int xOffset) => …;
public Point IncreaseY (int yOffset) => …;

public int GetHashCode() {…}
public bool Equals(object other) {…}
}
```

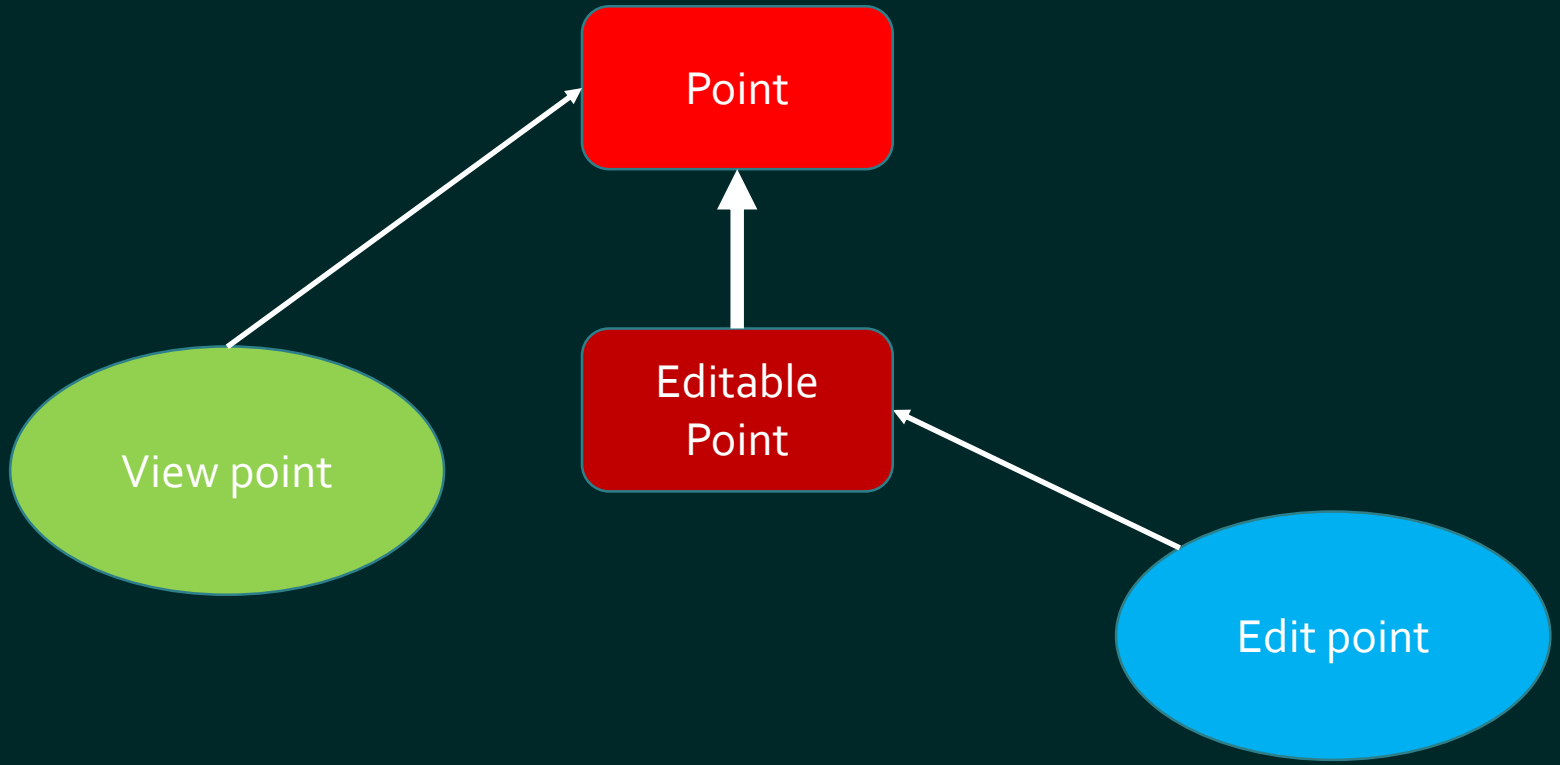
Why public?

```
class Point {

public readonly int X;
public readonly int Y;

public Point(int x, int y) { X = x; Y = y }

public Point IncreaseX (int xOffset) => …;
public Point IncreaseY (int yOffset) => …;

public int GetHashCode() {…}
public bool Equals(object other) {…}
}
```


Point
View point
Edit point

Inheritance?

```
class Point {
public readonly int X;
public readonly int Y;
…
}

class EditablePoint : Point {
public Point IncreaseX (int xOffset) => …;
public Point IncreaseY (int yOffset) => …;
}
```


Point
Editable Point
View point
Edit point

Alternative

Move methods that change the state
to a separate class
a.k.a. PointManager

Alternative

Move methods that change the state
to a separate class
a.k.a. PointManager

This is essentially abandoning Point as business object

F# modules as business logic scopes

```
type Point = {
    X : int
    Y : int
}

module Point =
    let increaseX v p = { p with X = p.X+v }
    let increaseY v p = { p with Y = p.Y+v }
```

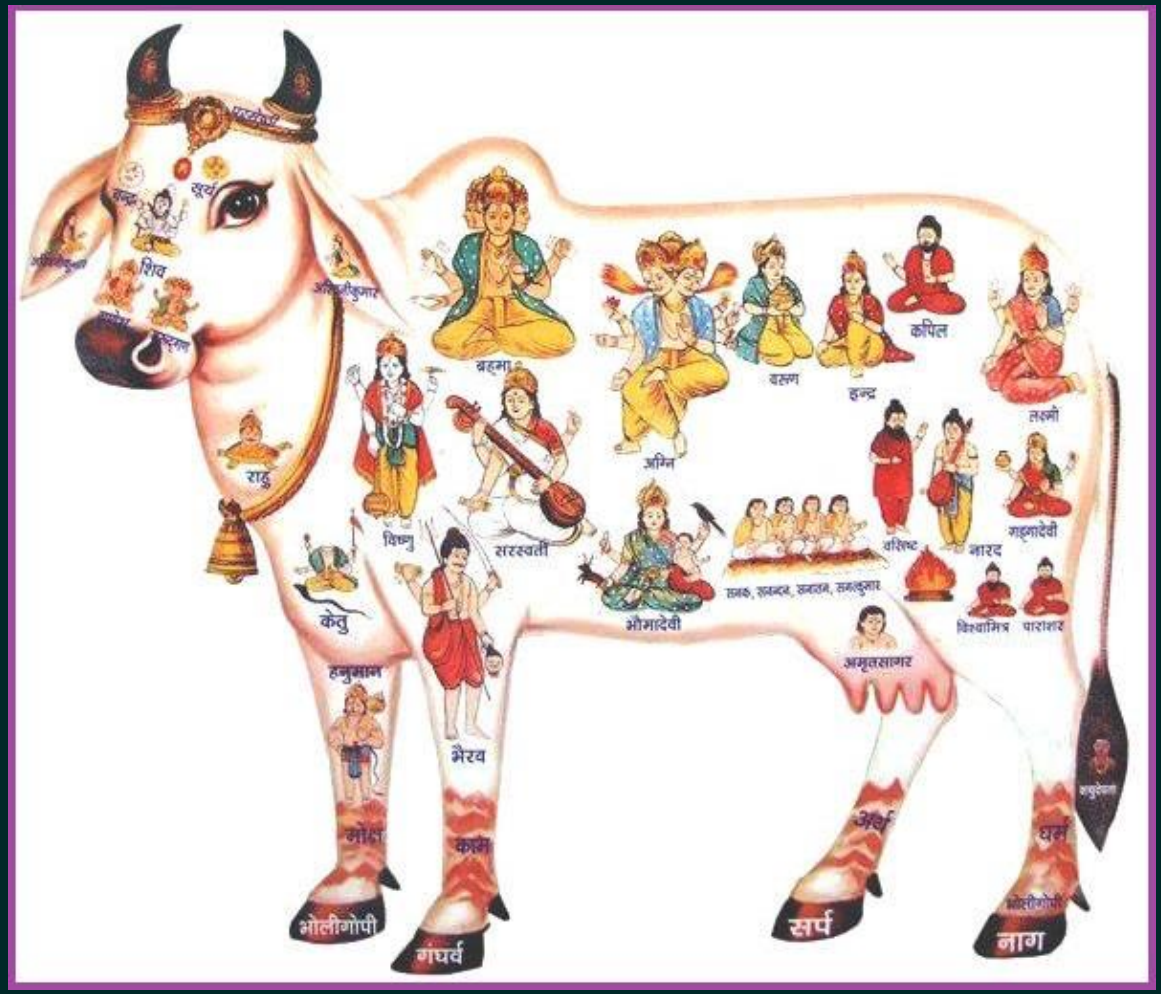
F# modules as business logic scopes

```fsharp
type Point = {
    X : int
    Y : int
}

module Point =
    let increaseX v p = { p with X = p.X+v }
    let increaseY v p = { p with Y = p.Y+v }

let v = { X = 5; Y = 6 }
let z = p |> Point.increaseX 1
```

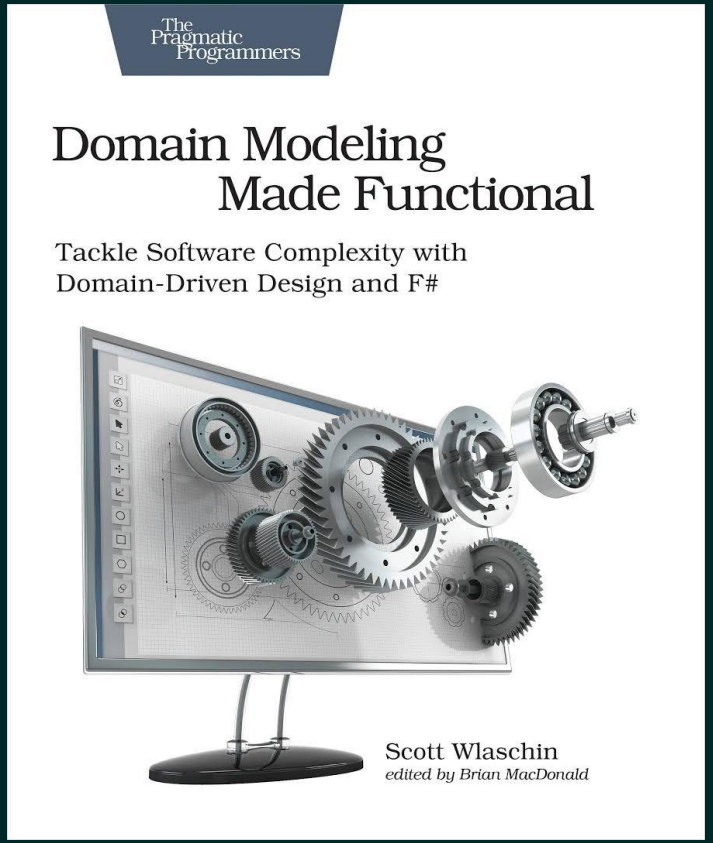
Controlling business logic visibility via modules

```
type Point = {…}

module PointUpdate =
    let increaseX v p = { p with X = p.X+v }
    let increaseY v p = { p with Y = p.Y+v }

open PointUpdate

let v = { X = 5; Y = 6 }
let z = p |> increaseX 1
```

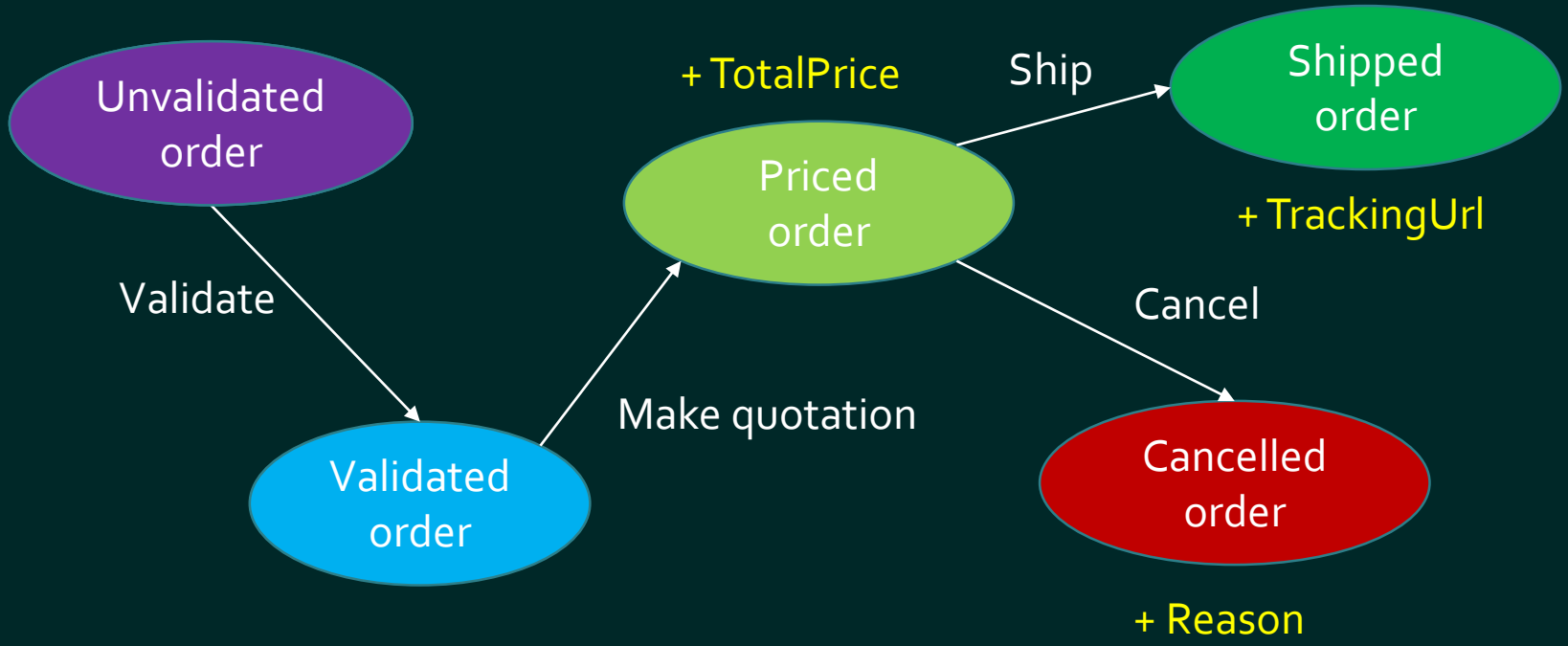
Business objects


शिव
ब्रह्मा
विष्णु
सरस्वती
अग्नि
वरुण
इन्द्र
कपिल
लक्ष्मी
राहु
केतु
भौमादेवी
सनक, सनन्दन, सनातन, सनत्कुमार
वशिष्ठ
नारद
गङ्गादेवी
विश्वामित्र
पाराशर
अमृतसागर
हनुमान
भैरव
मोक्ष
काम
अर्थ
धर्म
भोलीगोपी
गंधर्व
सर्प
नाग

Scott Wlaschin «Domain Modeling Made Functional»



Order processing



```
class Order {
…
decimal TotalPrice { get; }
Uri TrackingUrl { get; }
string CancellationReason { get; }


bool IsValidated { get; }
bool IsShipped { get; }
bool IsCancelled { get; }
}
```

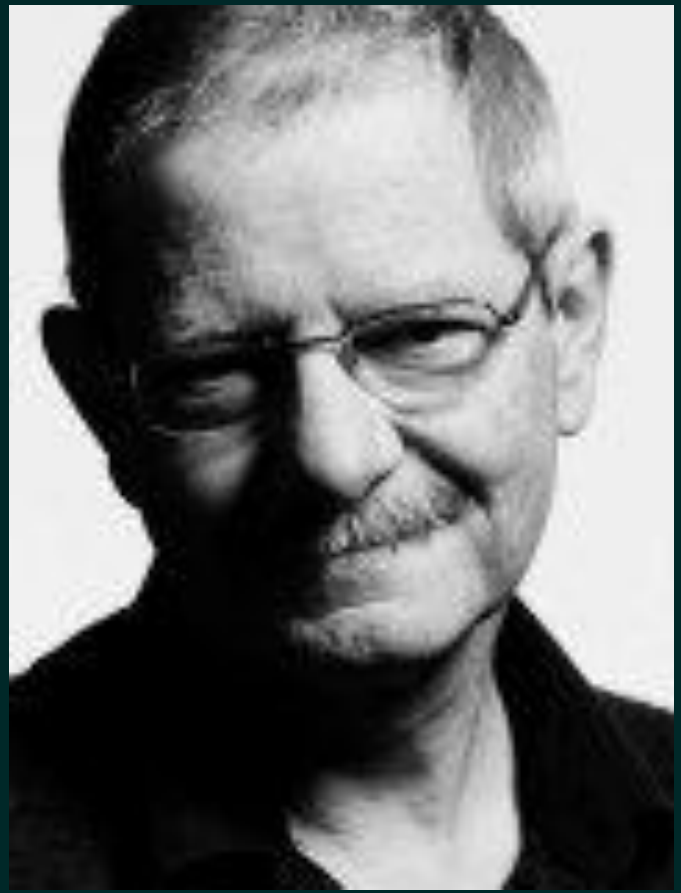
```
class Order {
…
decimal TotalPrice { get; }
Uri TrackingUrl { get; }
string CancellationReason { get; }

bool IsValidated { get; }
bool IsShipped { get; }
bool IsCancelled { get; }

void Validate();
void Ship();
void Cancel();
}
```

```
class Order {
…
decimal TotalPrice { get; }
Uri TrackingUrl { get; }
string CancellationReason { get; }

bool IsValidated { get; }
bool IsShipped { get; }
bool IsCancelled { get; }
}

class OrderManager {
void Validate(Order order);
void Ship(Order order);
void Cancel(Order order);
}
```

```
class Order {
…
decimal TotalPrice { get; }
Uri TrackingUrl { get; }
string CancellationReason { get; }

bool IsValidated { get; }
bool IsShipped { get; }
bool IsCancelled { get; }
}

class OrderManager {
void Validate(Order order);
void Ship(Order order);
void Cancel(Order order);
}
```

Pure data

Pure business nothing personal



Joe Armstrong on OOP

Since functions and data structures are completely different types of animal it is fundamentally incorrect to lock them up in the same cage

```
class UnvalidatedOrder { … }

class ValidatedOrder { … }

class PricedOrder {
… decimal TotalPrice { get; }
}

class ShippedOrder {
… Uri TrackingUrl { get; }
}

class CancelledOrder {
… string Reason { get; }
}
```

```
class OrderValidator {
  ValidatedOrder
    ValidateOrder(…)
}

class QuotationMaker {
  PricedOrder
    MakeQuotation(…)
}

class OrderDispatcher {
  ShippedOrder
    ShipOrder(…)
}
```

Domain modeling in F#

```
type OrderDetails = string list

type UnvalidatedOrder = {
    Details : OrderDetails
}

type ValidatedOrder = {
    Details : OrderDetails
    ValidationTime : DateTimeOffset
}
```

Domain modeling in F#

```
type PricedOrder = {
    Details : OrderDetails
    TotalPrice : decimal
}

type ShippedOrder = {
    Details : OrderDetails
    Uri : TrackingUrl
}

type CancelledOrder = {
    Details : OrderDetails
    Reason : string
}
```

Domain modeling in F#

```
module OrderProcessing =

    let validateOrder (order : UnvalidatedOrder) =
        { Details = order.Details
          ValidationTime = DateTimeOffset.Now }

    let priceOrder totalPrice (order : ValidatedOrder) =
        { Details = order.Details
          TotalPrice = totalPrice }

    let shipOrder trackingUrl (order : PricedOrder) =
        { Details = order.Details
          TrackingUrl = trackingUrl }
```

Domain modeling in F#

```
open OrderProcessing

let order =
    { Details = ["book"] }
    |> validateOrder
    |> priceOrder 9.90m
    |> shipOrder (Uri "http://www.orders.com/40395874")
```

Algebraic data types in F#

```
type ExpiryDate = {
    Year : int
    Month : int
}

type CardNumber = CardNumber of string

type PaymentCard = {
    CardNumber : CardNumber
    ExpiryDate : ExpiryDate
}

type BankAccount = BankAccount of string
```

Algebraic data types in F#

```
type FundingSource =
    | PaymentCard of PaymentCard
    | BankAccount of BankAccount

let isSourceValid source =
    let now = DateTime.Now
    match source with
    | PaymentCard x ->
        x.ExpiryDate >= { Month = now.Month
                          Year = now.Year }
    | BankAccount _ -> true
```

Active patterns in F#

```fsharp
let (|Even|Odd|) n =
    if n % 2 = 0 then Even
    else Odd

let printNumberKind n =
    match n with
    | Even -> "Even"
    | Odd -> "Odd"
```

Nulls should be avoided
not just by replacing them with options,
but avoiding options wherever possible

Yaron Minsky

Make illegal state unrepresentable

https://blog.janestreet.com/effective-ml-revisited/

Optional values are fine
at domain boundaries
but corrupt its business logic

Why do we need to pass optional values?

- To cover multiple scenarios in a single handler
  - Should the handler be split into several?
- To forward it to a next handler in the business logic chain
  - Should the data that is unused in the current handler be hidden from it?

Maybe Not

Rich Hickey

Rich Hickey – Maybe Not

- Maybe/Either are not type system's 'or/union' type
  - Rather, evidence of *lack* of first-class union type
- Either is a ~~malarkey~~ misnomer
  - Not associative/commutative/composable/symmetric

https://www.youtube.com/watch?v=YR5WdGrpoug

Rich Hickey – Sets vs Slots

Could we make it in C#?

Absolutely!

But…

What main advantage did we gain with F#?

Shortened the cycle
from specification
to production

Impact of F# on feature development cycle

1. Algebraic types help to better express functional requirements

Impact of F# on feature development cycle

1. Algebraic types help to better express functional requirements
2. Small immutable records are efficient to represent data structures for each stage of the business process

Impact of F# on feature development cycle

1. Algebraic types help to better express functional requirements
2. Small immutable records are efficient to represent data structures for each stage of the business process
3. Elimination of nulls and (mostly) options keeps business logic compact and straightforward

Impact of F# on feature development cycle

1. Algebraic types help to better express functional requirements
2. Small immutable records are efficient to represent data structures for each stage of the business process
3. Elimination of nulls and (mostly) options keeps business logic compact and straightforward
4. Use of modules expose right business logic for each scope – opposed to class public methods visible to every class observer

Impact of F# on feature development cycle

1. Algebraic types help to better express functional requirements
2. Small immutable records are efficient to represent data structures for each stage of the business process
3. Elimination of nulls and (mostly) options keeps business logic compact and straightforward
4. Use of modules expose right business logic for each scope – opposed to class public methods visible to every class observer

Thank you!

Vagif Abilov
Consultant in Miles
Norway - Russia

Github: object
Twitter: @ooobject
vagif.abilov@mail.com