goto;

GOTO Copenhagen 2021

Vagif Abilov

#GOTOcph

Life After Business Objects
Confessions of an OOP veteran

Vagif Abilov
Consultant in Miles
Oslo, Norway

Work with F# and C#

@ooobject
vagif.abilov@mail.com

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
Aktuelt
Populært
Nytt
Meny

Let’s begin with basics: Modeling a point

Dmitry Ivanov (former 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

Half-lives of software related entities

- Developers 3.1 years
- Applications 6.2 years
- Lines of code 13 years

Robert Smallshire
"Predictive Models of Development Teams and the Systems They Build"
https://www.youtube.com/watch?v=_ohjcq6LAHw

For the sake of the next developer

Keep code succinct
and intentions clear

Functional Programming

Path
to
concurrency

Locks do not compose

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

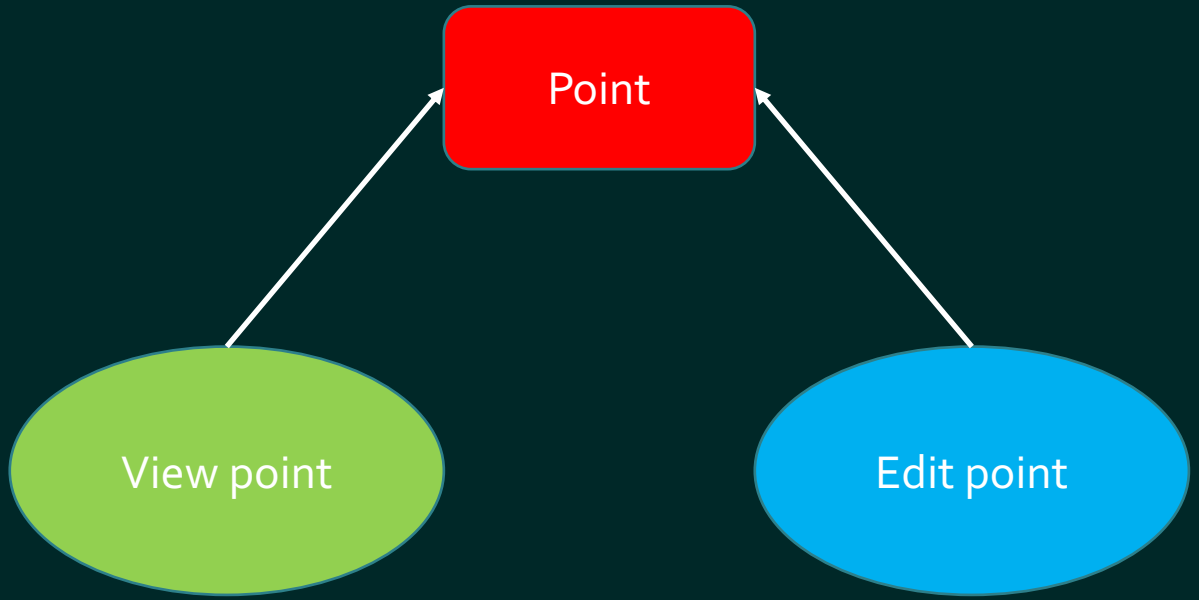
Time to have a closer look at business objects

```
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) {…}
}
```

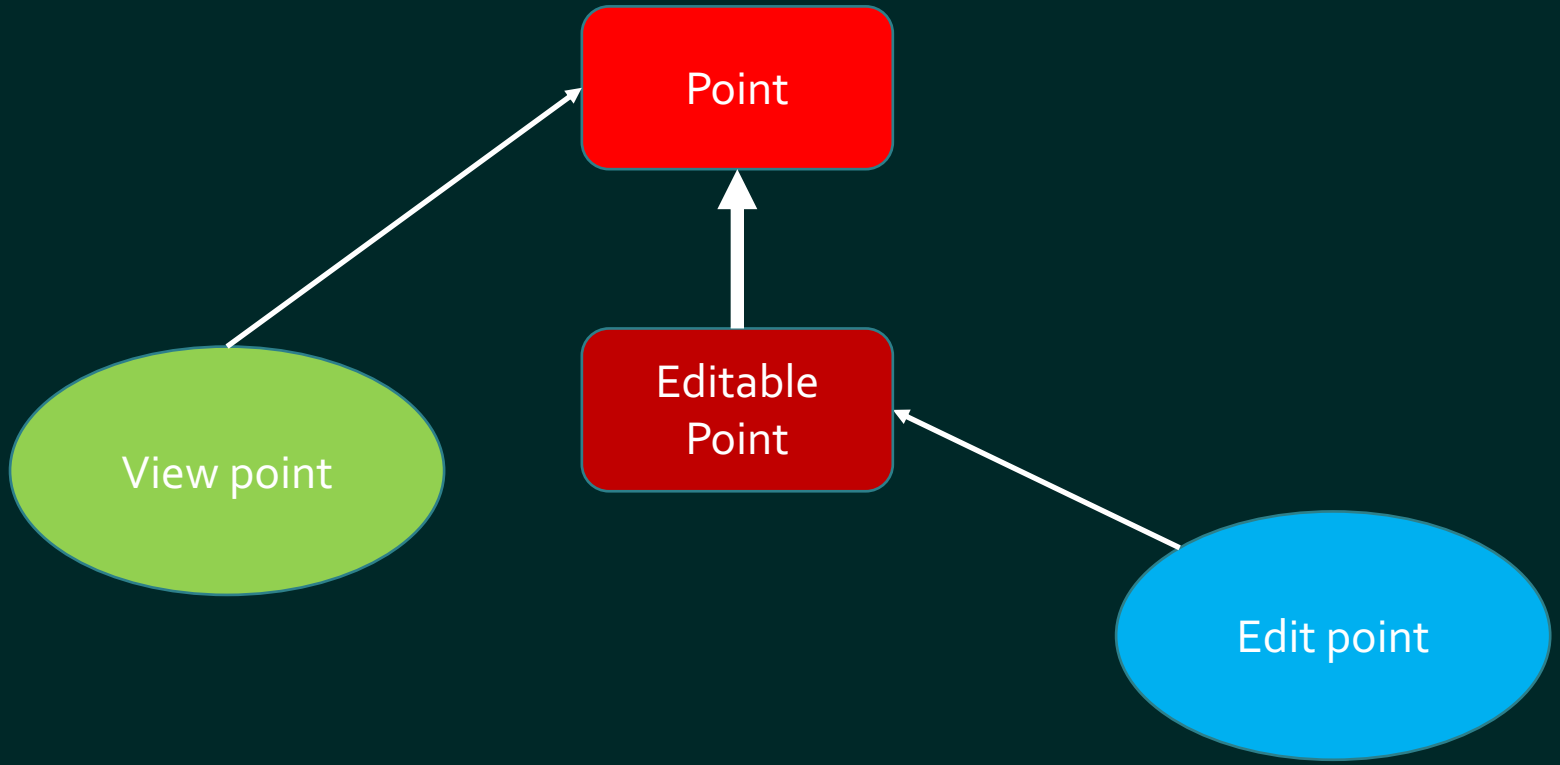
Why public?

```
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

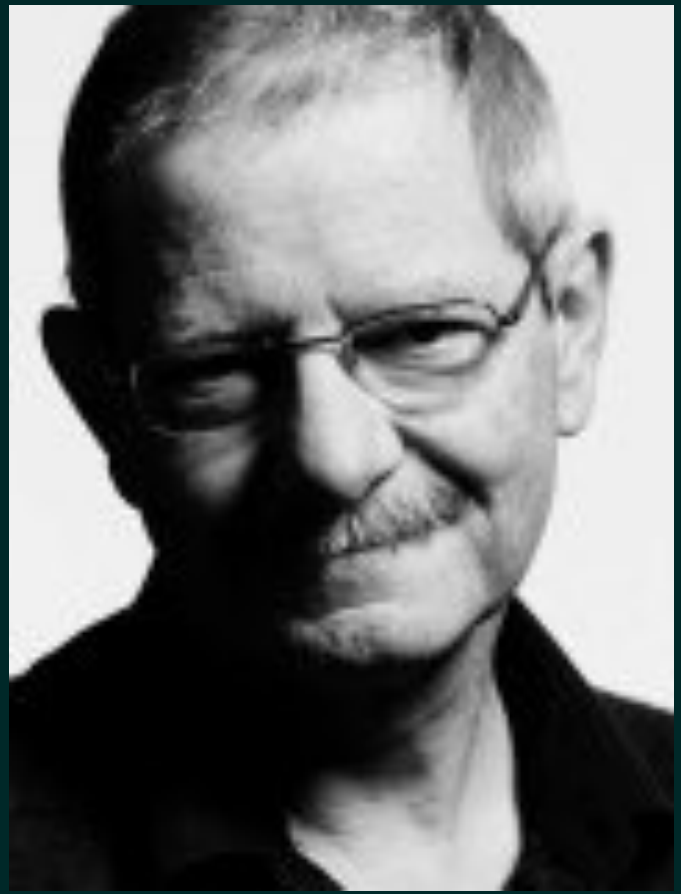
```
type Point = {
    X : int
    Y : int
}

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

```
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

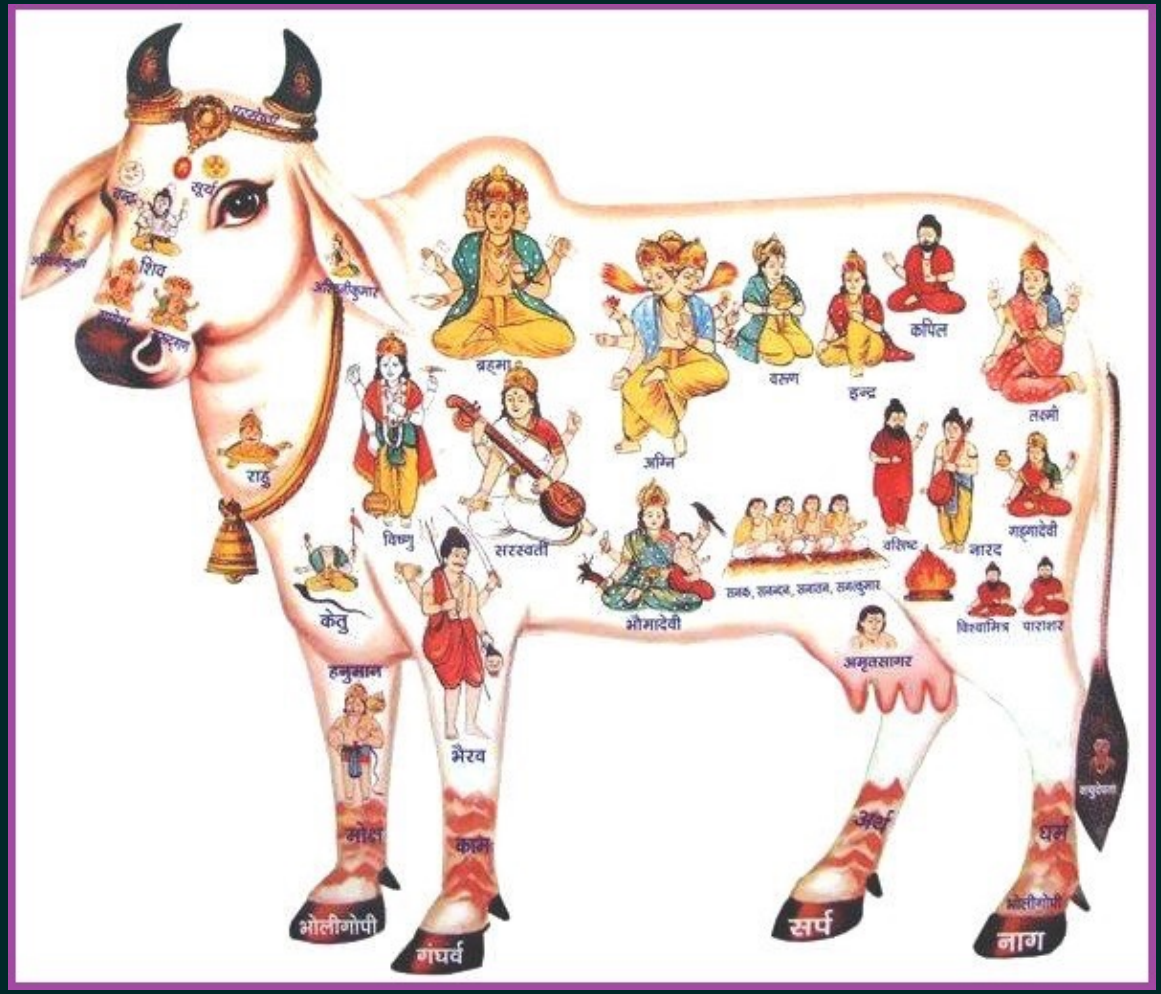
```
type Point = {…}

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
```



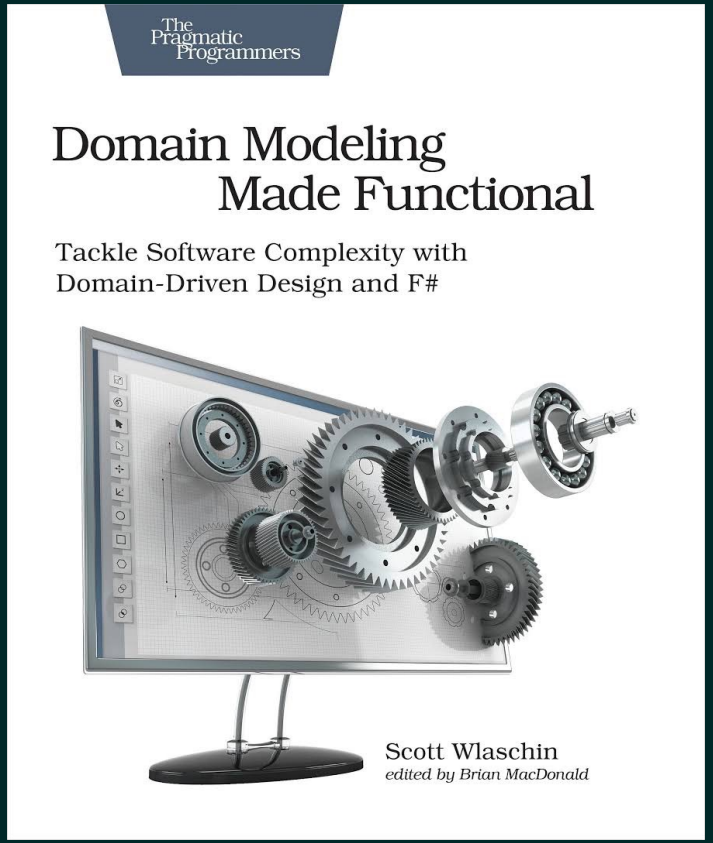
Joe Armstrong on OOP

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

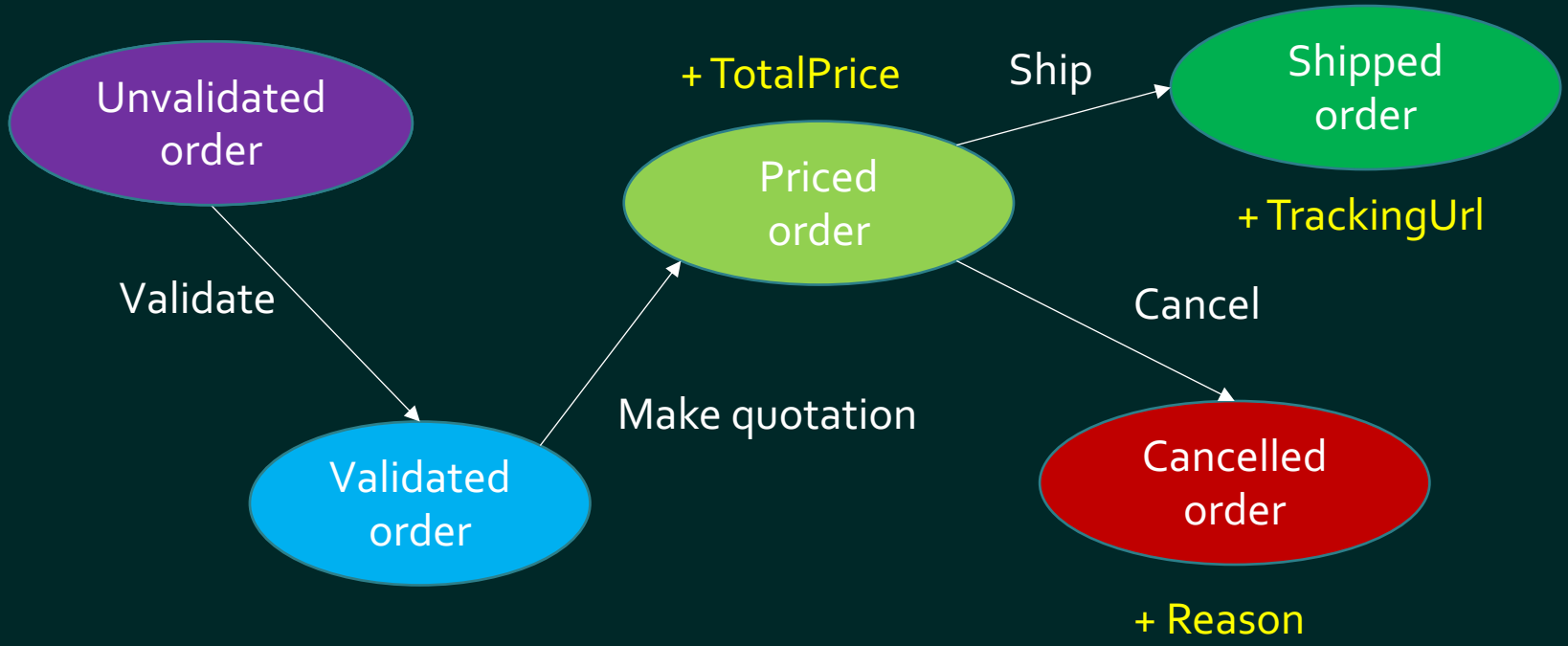
Business objects


शिव
ब्रह्मा
विष्णु
सरस्वती
राहु
केतु
हनुमान
भैरव
भौमादेवी
अग्नि
वरुण
इन्द्र
कपिल
तरणी
गरुडादेवी
नारद
वशिष्ट
सनक, सनन्दन, सनातन, सनत्कुमार
विश्वामित्र
पाराशर
अमृतसागर
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

```
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
```

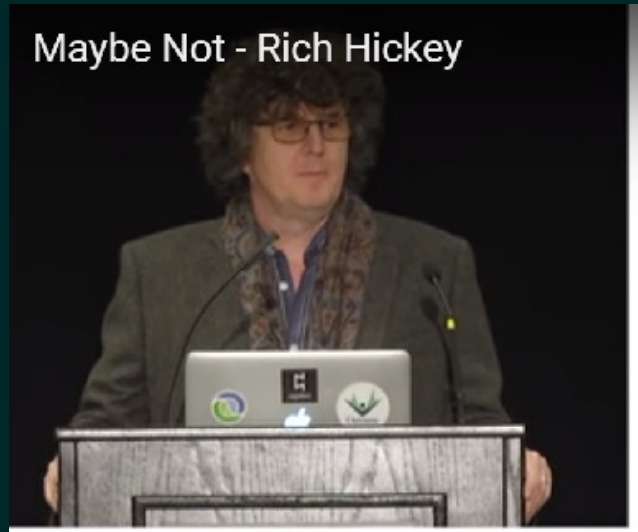
Nulls should be avoided
not just by replacing them with options,
but avoiding options wherever possible

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



Maybe Not

Rich Hickey

Can’t we adopt FP style in C#/Java?

OO languages become multiparadigm

- Java
- Kotlin
- C#
- C++

Phil Nash

OO Considered Harmful

Cppcon 2018

https://www.youtube.com/watch?v=pH-q2m5sb04

OO Considered Harmful - Phil Nash - CppCon 2020

Cppcon The C++ Conference September 13-18

Phil Nash

OO Considered Harmful

Best of ~~both~~ all worlds?

Low level: Prefer immutable value types

- Persistent data structures
- Monadic operations
- Builders
- Functionally composable algos (e.g ranges)

ansatz

51:17 / 57:38

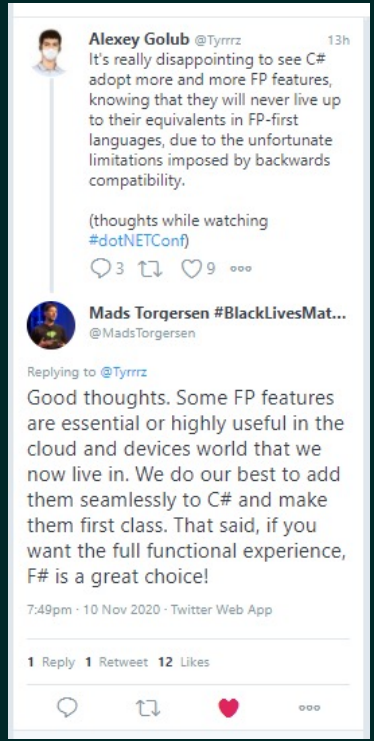
Can’t we adopt FP style in C#/Java?

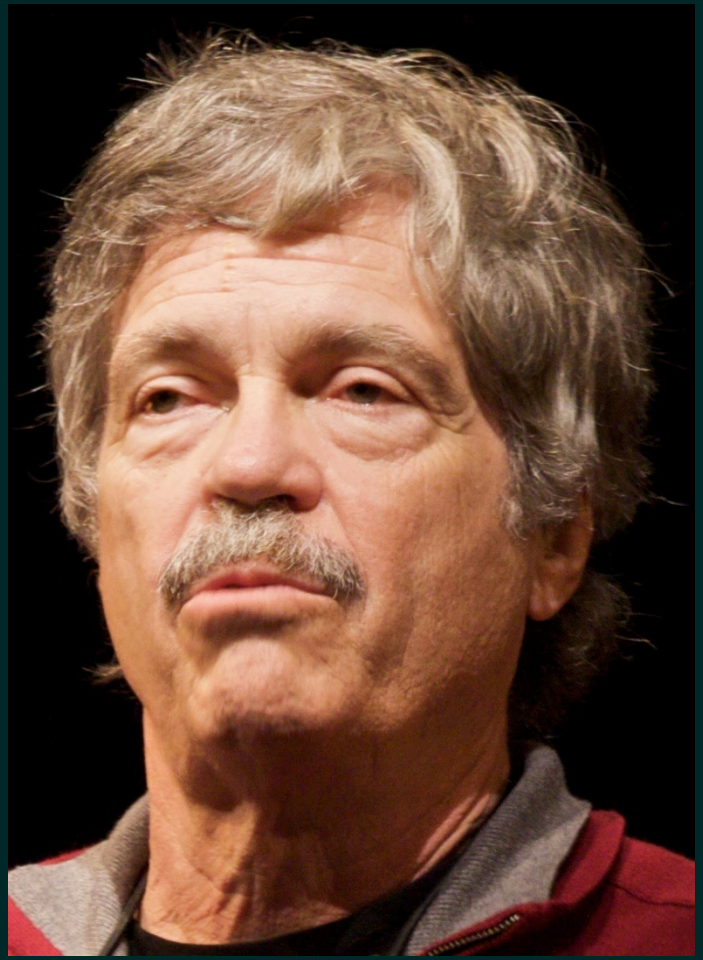
Absolutely!

But…

Can’t we adopt FP style in C#/Java?

Absolutely!
But…





Alan Kay

I invented the term
object-oriented,
and I can tell you
I did not have C++ in mind

Alan Kay on objects

"I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages."

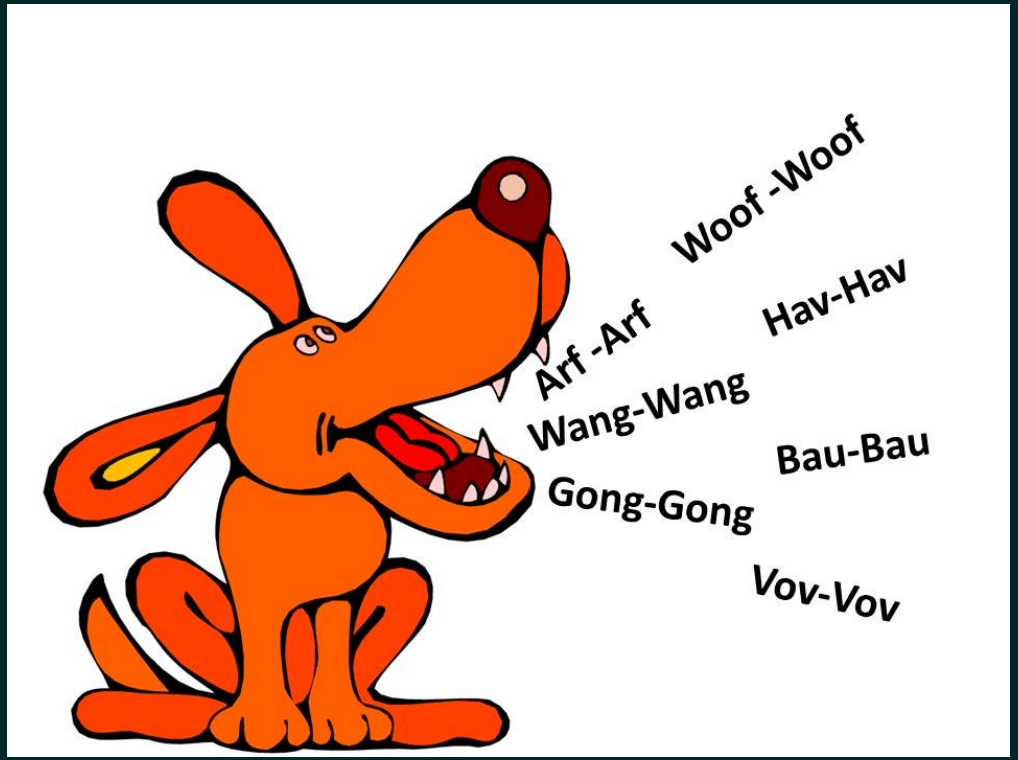
"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

Classic OOP vs real world

```
class Dog : Mammal {
…
string Name { get; }

void Bark(string phrase) {
…
}
```

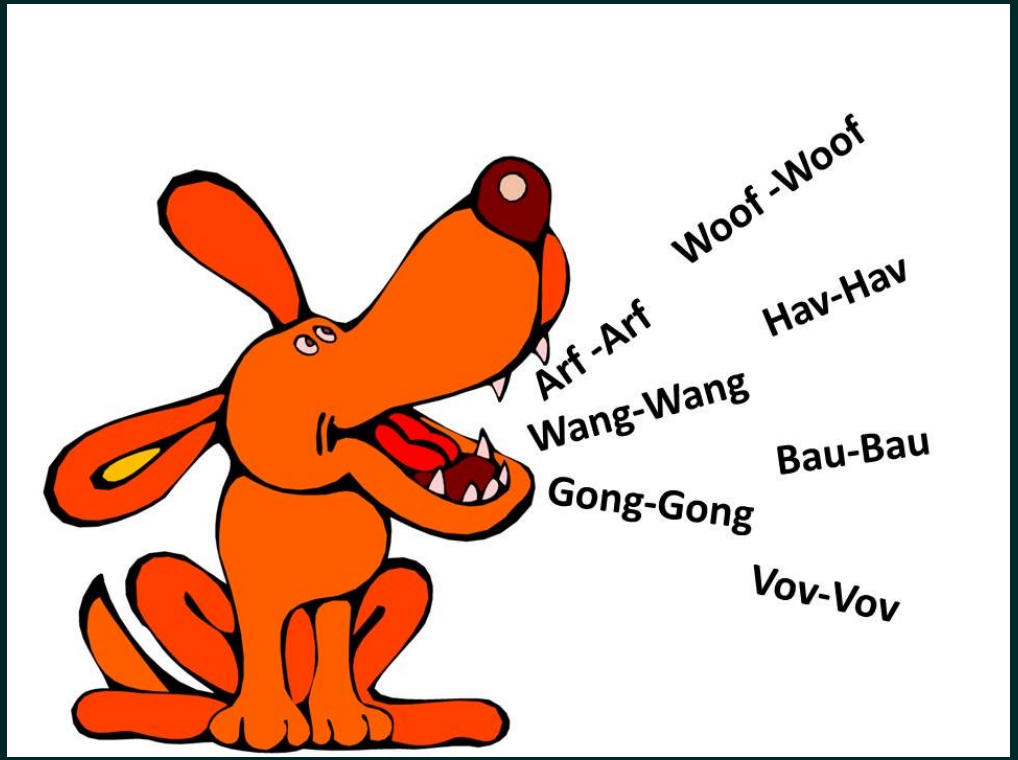
```
var myDog = new Dog();
myDog.Bark(«Woof-Woof");
```



Classic OOP vs real world

```
class Dog : Mammal {
…
string Name { get; }

void Bark(string phrase) {
…
}
```

```
var myDog = new Dog();
myDog.Bark("Hav-Hav");
```



Actor model as OO done right

We use actor model (Akka.NET)
to implement stateful objects

Impact of FP on feature development cycle

1. Algebraic types help to better express functional requirements

Impact of FP on feature development cycle

1. Algebraic types help to better express functional requirements
2. Elimination of nulls and (mostly) options keeps business logic compact and straightforward

Impact of FP on feature development cycle

1. Algebraic types help to better express functional requirements
2. Elimination of nulls and (mostly) options keeps business logic compact and straightforward
3. Use of modules to expose right business logic for each processing scope

Impact of FP on feature development cycle

1. Algebraic types help to better express functional requirements
2. Elimination of nulls and (mostly) options keeps business logic compact and straightforward
3. Use of modules to expose right business logic for each processing scope
4. Communication between objects are based on the actor model

Impact of FP on feature development cycle

1. Algebraic types help to better express functional requirements
2. Elimination of nulls and (mostly) options keeps business logic compact and straightforward
3. Use of modules to expose right business logic for each processing scope
4. Communication between objects are based on the actor model

What main advantage did we gain with FP?

Shortened the cycle
from specification
to production

Thank you!

Vagif Abilov
Consultant in Miles
Norway - Russia

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

goto;

Don’t forget to
vote for this session
in the **GOTO Guide app**