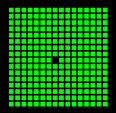
Domain Driven Design (DDD) with Algebraic Data Types (ADT)

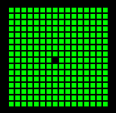
(Mr. Math, SPISE MISU ApS)

2020-11-01 @ foss-north 2020 take II (virtual edition)

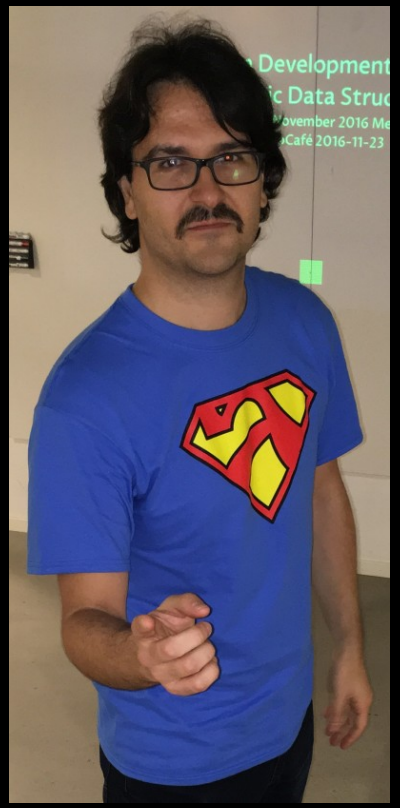


Overview

- About me (very shortly)
- Domain Driven Design + Algebraic Data Types
  - Background
  - Programming paradigms
  - Demo (live coding)



About me (very shortly)

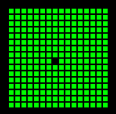


- **Mr.** Ramón Soto **Math**iesen (***Spaniard*** + ***Dane***)
- MSc. Computer Science and minors in Mathematics
- ***CompSci*** @ SPISE MISU ApS
  - Trying to solve EU GDPR with a scientific approach (https://uniprocess.org)
    - Permissive copyleft license (LGPL-3.0)
  - Mostly with ***Haskell*** and to a lesser extend ~~***Elm***~~ (***PureScript***)
- Blog: http://blog.stermon.com/ (slides under /talks/)
- Member of the Free Software Foundation (FSF) since ***November 2007***
- Founder of Meetup F#unctional Copenhageners EST. ***November 2013***
- PureScript / Elm / Haskell / TypeScript / F# / OCaml / Lisp / C++ / C# / JavaScript

Matching of expectations

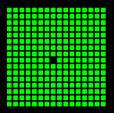
- In this talk, we will see how it's possible to add constraints, at ***compile-time***, to our code implementations so they comply with with a specified domain
- There will be **shown code**, but it ***shouldn't be necessary*** to know **how to code** as it (hopefully or at least the part where ADT are used) will remind of plain **English** :)

  Note: I would love questions, but please save them to the end of the talk (Demo + Q&A), as time is limited and I don't want to overlap other speakers



Background

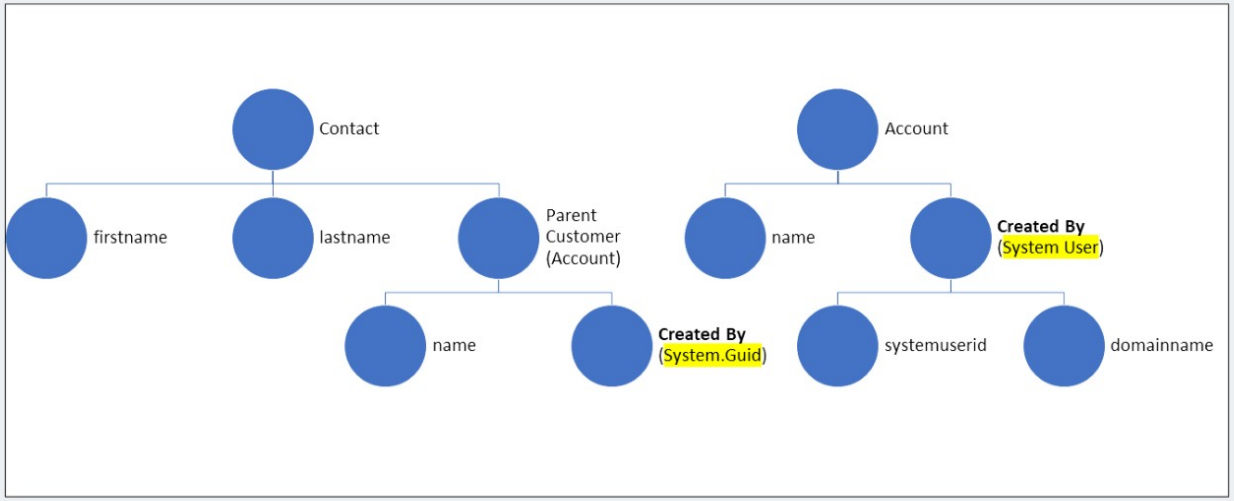
- When I'm not working with uniprocesses, I tend to do some .NET (core) freelance gigs. What I miss the most, when using the language C#, is that it lacks of ADT. See slide 36 from one of my recent talks:
  - CrmWebApiUtil + LINQ Provider = Cloud (Docker)



Background

CRM WebAPI Util > DFDS.CEM.Dynamics.CRM.WebAPI.Util (bin)

CRM WebAPI Util > WebAPI Sum types (1/3)



36

DFDS

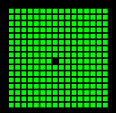
Background

- As I mention in the abstract of this talk, the .NET design team just added product types to C#, which is really good, but they still miss the most important one, which is sum types (*)
- We will use a tool, that has support for the following programming paradigms, in order to not get lost in translation (my opinions might be biased, bare with me):
  - Imperative: Used in languages like C
  - Object Oriented (OO): Used in languages like Java, C++ or C#
  - Functional Programming (FP): Used in languages like OCaml or Haskell

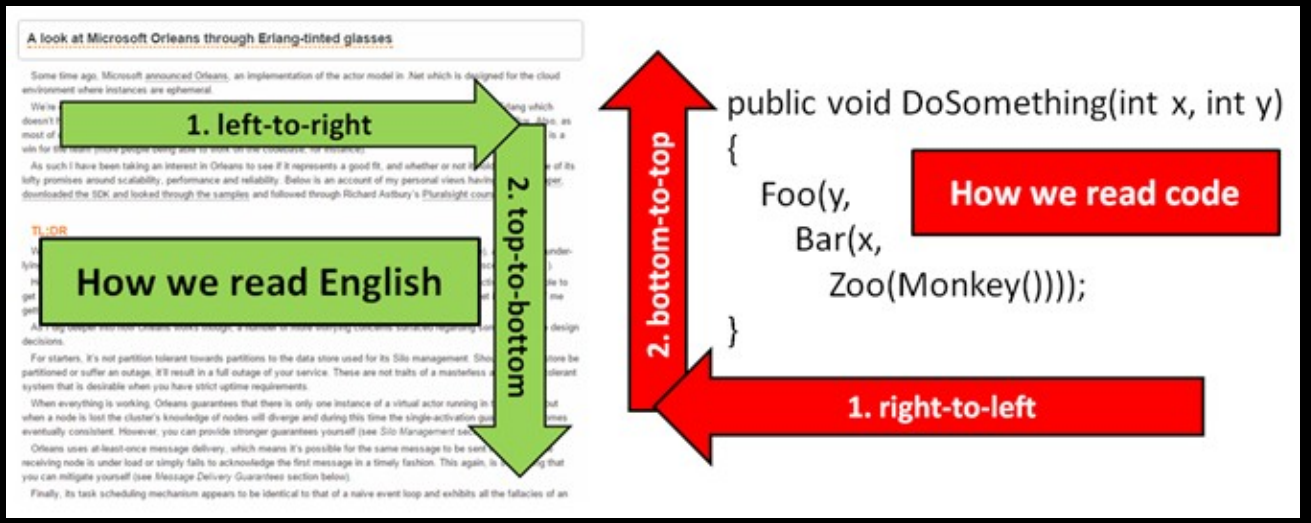
(*) - Later on we will see how we can mimic product types with sum types

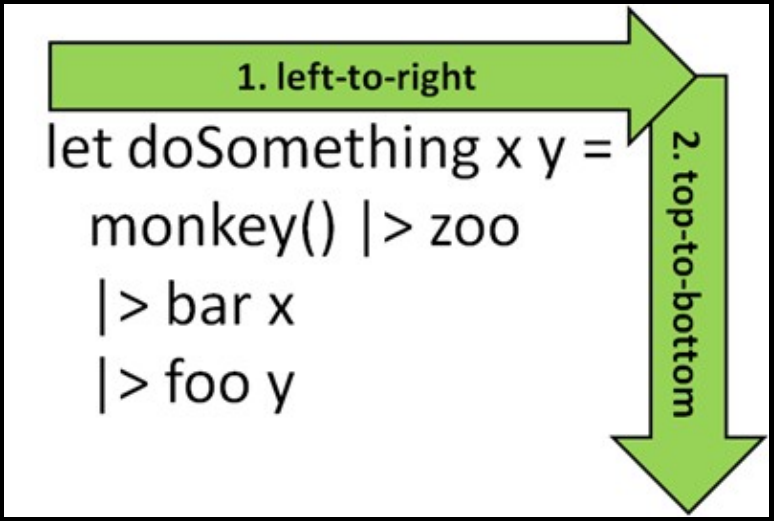
The tool (F#)
Definition and features

- Wikipedia: is a functional-first, general purpose, strongly typed, multi-paradigm programming language that encompasses functional, imperative, and object-oriented programming methods.
- It has the following features, that will help us understand code from both snippets in the slides and from the demo:
  - Simple and intuitive (readable)
  - Functions as first-class citizens
  - Strongly type-safe
  - With a built-in REPL



The tool (F#)
Simple and intuitive (readable)





- Forced indentation, just like Python, in combination with |> operator, makes it easy to read again & again

The tool (F#)
Functions as first-class citizens

- Higher-order functions.
  - Passing functions as arguments is just like gluing functions together, composing them to bigger ones:

```
[0 .. 9] |> List.map (fun x -> x + x)
```

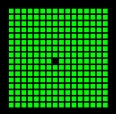
The tool (F#)
Strongly type-safe

- Computer says no :

```
(* error FS0001: The type 'string' does not match the type 'int' *)
let result = 42 + "42"
```

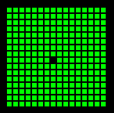
The tool (F#)
With a built-in REPL

- Read, Evaluate, Print and Loop (REPL):
  - Possible to evaluate functions, modules and types directly from the IDE to F# interactive (interpreted code)
  - This makes it easy to reason about creating smaller pieces of logic and composing them to greater blocks
  - F# script files (.FSX) are also interpreted, which means that files are type checked before executing a single line



The tool (F#)
Imperative code example

```
File Edit Options Buffers Tools F# Help
15    [<RequireQualifiedAccess>]
16    module Imperative =
17
18      type Point2D =
19        struct
20          val mutable x: int
21          val mutable y: int
22        end
23
24      type Point3D =
25        struct
26          val mutable x: int
27          val mutable y: int
28          val mutable z: int
29        end
30
31      let point2D (x, y) =
32        let mutable p = Point2D()
33        p.x <- x
34        p.y <- y
35        p
36
37      let point3D (x, y, z) =
38        let mutable p = Point3D()
39        p.x <- x
40        p.y <- y
41        p.y <- z
42        p
43
44      let scale2D (factor, point : Point2D) =
45        let mutable p = point
46        p.x <- p.x * factor
47        p.y <- p.y * factor
48        p
49
50      let scale3D (factor, point : Point3D) =
51        let mutable p = point
52        p.x <- p.x * factor
53        p.y <- p.y * factor
54        p.z <- p.z * factor
55        p
56
```

The tool (F#)
OO code example

```
File Edit Options Buffers Tools F# Help
 57    [<RequireQualifiedAccess>]
 58    module OO =
 59
 60      [<AbstractClass>]
 61      type Point(x : int, y : int) =
 62        let mutable internalX = x
 63        let mutable internalY = y
 64
 65        member this.X with get() = internalX and set x = internalX <- x
 66        member this.Y with get() = internalY and set y = internalY <- y
 67
 68        abstract member Scale : int -> Point
 69        default this.Scale(factor) =
 70          this.X <- this.X * factor
 71          this.Y <- this.Y * factor
 72          this
 73
 74      type Point2D(x : int, y : int) =
 75        inherit Point(x, y)
 76
 77      type Point3D(x : int, y : int, z : int) =
 78        inherit Point(x, y)
 79
 80        let mutable internalZ = z
 81
 82        member this.Z with get() = internalZ and set z = internalZ <- z
 83
 84        override this.Scale(factor) =
 85          this.X <- this.X * factor
 86          this.Y <- this.Y * factor
 87          this.Z <- this.Z * factor
 88          (* The conversion from Point3D to Point is a compile-time safe upcast *)
 89          this :> Point
```

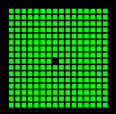
The tool (F#)
FP code example

```
File Edit Options Buffers Tools Complete In/Out Signals Help
 91      [<RequireQualifiedAccess>]
 92      module FP =
 93
 94        type point =
 95          | Point2D of x:int * y:int
 96          | Point3D of x:int * y:int * z:int
 97
 98        let point2D x y   = Point2D (x, y)
 99        let point3D x y z = Point3D (x, y, z)
100
101        let scale factor = function
102          | Point2D (x,y)   -> Point2D (x * factor, y * factor)
103          | Point3D (x,y,z) -> Point3D (x * factor, y * factor, z * factor)
```

Programming paradigms
Imperative code

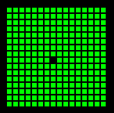
- Pros
  - Data-structs are represented as value types and not reference types (fast)
- Cons
  - Instantiated with default values for each field
  - All fields are accessible and therefore can be mutated, not ideal to work with concurrent software (*)
  - Lack of polymorphism so a lot of repetitive code

(*) - As in F#, they have made value types immutable by default so you will have to mark the binding value as mutable



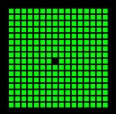
Programming paradigms
Imperative code

```
File Edit Options Buffers Tools F# Help
  15 |   [<RequireQualifiedAccess>]
  16 |   module Imperative =
  17 |
  18 |     type Point2D =
  19 |       struct
  20 |         val mutable x: int
  21 |         val mutable y: int
  22 |       end
  23 |
  24 |     type Point3D =
  25 |       struct
  26 |         val mutable x: int
  27 |         val mutable y: int
  28 |         val mutable z: int
  29 |       end
  30 |
  31 |     let point2D (x, y) =
  32 |       let mutable p = Point2D()
  33 |       p.x <- x
  34 |       p.y <- y
  35 |       p
  36 |
  37 |     let point3D (x, y, z) =
  38 |       let mutable p = Point3D()
  39 |       p.x <- x
  40 |       p.y <- y
  41 |       p.y <- z
  42 |       p
  43 |
  44 |     let scale2D (factor, point : Point2D) =
  45 |       let mutable p = point
  46 |       p.x <- p.x * factor
  47 |       p.y <- p.y * factor
  48 |       p
  49 |
  50 |     let scale3D (factor, point : Point3D) =
  51 |       let mutable p = point
  52 |       p.x <- p.x * factor
  53 |       p.y <- p.y * factor
  54 |       p.z <- p.z * factor
  55 |       p
  56 |
```

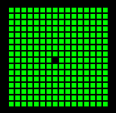
Programming paradigms
OO code

- Pros
  - Re-usability of code by inheritance
  - Encapsulation of internal state (***get*** and ***set*** methods)
  - Enforce logic
  - Specify logic to be overridden
- Cons
  - Internal state, not ideal to work with concurrent software
  - Logic is bound to the data (lack of separation)
  - Pattern matching on: `char`, `string`, `bool`, integrals (`int`, `long`) and `enum` types (C# 6.0 and earlier)
    - From C# 7.0 on any non-null type (with the `is` casting pattern)
  - Upcasting conversion can be statically type checked at compile-time
  - Downcasting conversion can ***not*** be type checked at compile-time



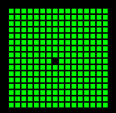
Programming paradigms
OO code

```
File Edit Options Buffers Tools F# Help
  57    [<RequireQualifiedAccess>]
  58    module OO =
  59
  60      [<AbstractClass>]
  61      type Point(x : int, y : int) =
  62        let mutable internalX = x
  63        let mutable internalY = y
  64
  65        member this.X with get() = internalX and set x = internalX <- x
  66        member this.Y with get() = internalY and set y = internalY <- y
  67
  68        abstract member Scale : int -> Point
  69        default this.Scale(factor) =
  70          this.X <- this.X * factor
  71          this.Y <- this.Y * factor
  72          this
  73
  74      type Point2D(x : int, y : int) =
  75        inherit Point(x, y)
  76
  77      type Point3D(x : int, y : int, z : int) =
  78        inherit Point(x, y)
  79
  80        let mutable internalZ = z
  81
  82        member this.Z with get() = internalZ and set z = internalZ <- z
  83
  84        override this.Scale(factor) =
  85          this.X <- this.X * factor
  86          this.Y <- this.Y * factor
  87          this.Z <- this.Z * factor
  88          (* The conversion from Point3D to Point is a compile-time safe upcast *)
  89          this :> Point
```

Programming paradigms
FP code

- Pros
  - Simple and concise
  - ADT ensure that we don't need to do any casting at all
  - ADT types provide constructors (uncurried in OCaml/F#/Haskell but the later also have support for curried)
  - Exhaustive pattern-matching on all types
  - Data is immutable by default, therefore ideal to work with concurrent software
- Cons
  - All fields are accessible and therefore can be changed, lack of encapsulation
  - Immutability can allocate a lot of memory and therefore make software slow



Programming paradigms
FP code

```
File Edit Options Buffers Tools Complete In/Out Signals Help
 91      [<RequireQualifiedAccess>]
 92      module FP =
 93
 94        type point =
 95          | Point2D of x:int * y:int
 96          | Point3D of x:int * y:int * z:int
 97
 98        let point2D x y   = Point2D (x, y)
 99        let point3D x y z = Point3D (x, y, z)
100
101        let scale factor = function
102          | Point2D (x,y)   -> Point2D (x * factor, y * factor)
103          | Point3D (x,y,z) -> Point3D (x * factor, y * factor, z * factor)
```

ADT + DDD

ADT + DDD
Algebraic data types

- **Product types**: think of it as tuples (pairs, triples, ...): `(42uy,'C')`
  - Record types are just tuples with labels:

```
{ age: 42uy; initial: 'C' }
```

- **Sum types (also know as Union types)**: think of it as disjoint sets (have no element in common). The element ***must*** be in one of the assigned disjoint sets:
  - A person is either a child or an adult:

```
type person = Child | Adult
```

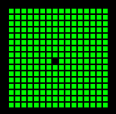
  - Temperature is measured Celsius or Fahrenheit:

```
type temperature = Celsius of float | Fahrenheit of int
```

Note: Record types are equivalent to single case Sum types, with named fields

```
type rt =     { age : byte ; initial : char }

type st = SC of age : byte * initial : char
```

ADT + DDD
Algebraic data types

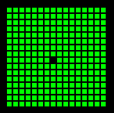
- ADT allows you to pattern match on all branches:

```
type suit = Clubs | Diamonds | Hearts | Spades
type card = { rank : byte ; suit : suit }
let isAce : card -> bool = function
  | { rank = 1uy } -> true
  | _______________ -> false

type person = Child | Adult
let assertAge : byte -> person -> bool =
  fun age ->
    function
      | Child -> age <  18uy
      | Adult -> age >= 18uy
```

Note: For exhaustive pattern-matching in F#, use the following compiler flag:

--warnaserror:25



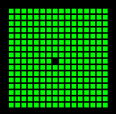
ADT + DDD
Algebraic data types

- With ADT you will be able to compose simpler types together in order to create more complex data structures:

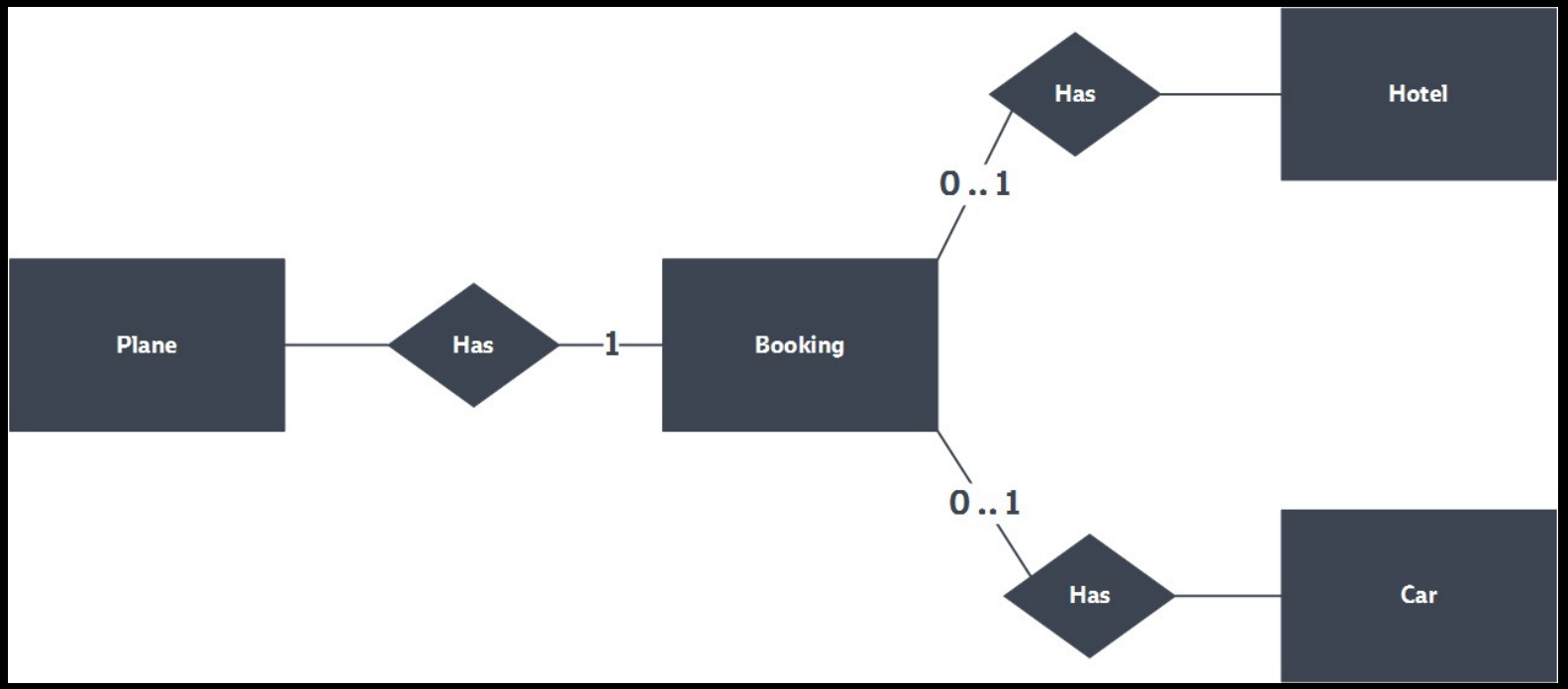
```
type product =         byte *                  char  (* = Cartesian product)
type record  = {age : byte ; initial : char}  (; = Cartesian product)
type sum     = Foo of byte | Bar of char      (| = Union)
```

- This is ideal for domain modeling (TDD/DDD) as it allows you to use these mathematically constraints to:
  “Make Illegal States Unrepresentable” -- Yaron Minsky

  Note: If you can’t represent invalid data, you don’t need to test for it



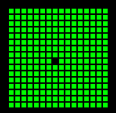
ADT + DDD
Domain Driven Design



Domain modeled with an ER-diagram

ADT + DDD
Domain Driven Design

- It’s intuitive to see that we aren’t able to make a booking unless a plane is specified (***mandatory***)
- Also, we can see that we might book a hotel or rent a car, but they are not required (***optional***)
- I don’t think we can get any other information out from this diagram unless we also read some text
- Which products are they offering?



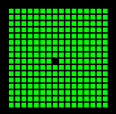
ADT + DDD
Domain Driven Design

```
File Edit Options Buffers Tools Complete In/Out Signals Help
 11 | module SpiseMisu =
 12 |
 13 |   module FossNorth =
 14 |
 15 |     type booking =
 16 |       | Basic    of plane
 17 |       | Combo    of combo
 18 |       | Fullpack of plane * hotel * car
 19 |     and plane =
 20 |       { Outbound    : date
 21 |         Return      : date
 22 |         Destination : city
 23 |       }
 24 |     and combo =
 25 |       | ``With hotel`` of plane * hotel
 26 |       | ``With car``   of plane * car
 27 |     and hotel =
 28 |       { Arrival   : date
 29 |         Departure : date
 30 |         Location  : city
 31 |       }
 32 |     and car =
 33 |       { From     : date
 34 |         To       : date
 35 |         Location : city
 36 |       }
 37 |     and city = string
 38 |     and date = System.DateTime
```

Domain modeled with F# type definitions

ADT + DDD
Domain Driven Design

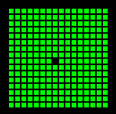
- We can easily see the 3 product which are offered
  - Basic, Combo and Fullpack
- Combo products can be of two types
  - “With hotel” and “With car”



ADT + DDD
Domain Driven Design

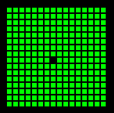
- We can see some constraints:
  - A Booking can either be Basic, Combo or Fullpack (disjoint union)
  - With each of these products requirements (tuples):
    - Basic → (plane) single
    - Combo → (plane, hotel) pair or (plane, car) pair
    - Fullpack → (plane, hotel, car) triple
  - We can also see that a plane will require the following information (still a tuple):
    - plane → (Outbound date and time, Return date and time, Destination country)

Note: With this approach, the domain design and implementation are still separated, even though, both will be represented as code



ADT + DDD
Demo

- Lets implement the domain of a Book, usable for a Bookstore or a Library:
  - Types: Audio, electronic and physical
  - Formats:
    - AAC, MP3, M4B and WAV (audio books)
    - EPUB, MOBI and PDF (electronic books)
    - Hardcover and Paperback (printed books)
  - Info:
    - Mandatory: title, authors, publisher, language, isbn10 and isbn13
    - Optional: pages
  - Rating: 1 to 5 stars



ADT + DDD
Demo

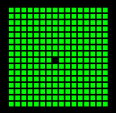
Note: Encapsulation can be achieved by limiting the exposure by using modules and private constructors. The functional way implies, in some cases, to keep sum type constructors private to modules:

```
module Review =
  type rating = private Rate of byte
  let  rate x =
    if x >= 0uy && x <= 5uy
    then Some (Rate x)
    else None
Review.Rate 42uy
(* error FS1093: The union cases or fields of the type 'rating' are not
   accessible from this code location *)
Review.rate 42uy
(* > val it : Review.rating option = None *)
```

Summary

- All programming paradigms have their pros/cons but personally, since I'm an advocate of strongly-typed code, I think it is easier (and sound) to model and design business logic into applications in a more safe, secure and robust manner by using ADT. Hopefully, in a near future, the C# design team will add support for sum types
- Therefore I hope I have convinced you that ADT are ideal for domain modeling (TDD/DDD) as it allows you to use mathematically constraints to: "Make Illegal States Unrepresentable" -- Yaron Minsky

  Reminder: If you can't represent invalid data, you don't need to test it



Q&A

Any questions?