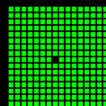
Type Driven Development (TDD) and Idiomatic Data Structures

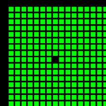
SweNUG November 2016 Meetup

@FooCafé 2016-11-23



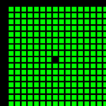
Overview

- About me
- Test vs Type Driven Development (TDD)
- Idiomatic Data Structures
- Show me some code
- Q & A



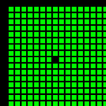
About me (very shortly)

- Ramón Soto Mathiesen
- MSc. Computer Science ***DIKU/Pisa*** and minors in Mathematics ***HCØ***
- ***CompSci*** @ SPISE MISU ApS
  - ***"If I have seen further it is by standing on the shoulders of giants"***
    -- Isaac Newton (Yeah Science, Bitch … Mostly mathematics)
  - ***Elm*** with a bit of ***Haskell*** and a bit of ***F#*** (fast prototyping)
- Elm / Haskell / TypeScript / F# / OCaml / Lisp / C++ / C# / JavaScript
- Blog: http://blog.stermon.com/



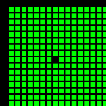
Test Driven Development

- Software approach where you:
  - ***Before*** implementing any piece of code, you ***first*** define ***test cases***
  - These test cases are ***added*** to code as ***stubbed methods*** which will ***initially all fail***
  - Once the code ***stubs*** are ***implemented correctly***, ***all test cases*** will then ***succeed***
  - Test cases ***must*** map 1:1 with ***Use cases***



Test Driven Development

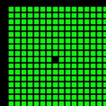
- Normally, this is the kind of approach we use to see ...
- ***Business experts*** produce a lot of ***documents*** combined with ***diagrams*** to specify requirements for the application
- Afterwards, the produced ***documents*** are given to ***developers*** (most of them have no knowledge of the domain) and they will ***implement*** requirements as ***software***



Test Driven Development (Pros/Cons)

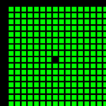
- Pros:
  - We involve Business experts as they are able to express themselves in plain English
- Cons:
  - Produced ***documents*** are not ***made*** for computers but ***for humans***
  - Therefore, once software changes (and it will) due to Software Development Life Cycle, the produced documents will not always be updated (*)

(*) For example, ***Medicinal industry*** always update the documentation as it's a legal requirement, which makes development ***very slow*** ...

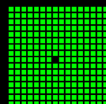


Test Driven Development (Main issue)

- As there is ***no binding*** between ***text*** and ***software***, there might be ***misunderstanding*** on how to implement ***requirements as code***
- Based on the ***complexity*** of some ***organizations*** and ***software providers*** (levels of hierarchy) before you get the ***right answer*** you might have gone through several layers.
  - Do you remember what happened back in ***primary school*** when the ***teacher*** started by ***saying something to one kid*** and it had to go through ***all the kids*** in a ***chain (whispering)***. Did the ***initial message*** sound anything like the ***final***? ***Not really*** right?



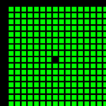
Test Driven Development (WTF did u just say?)



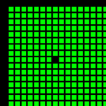
Type (Domain) Driven Development

- So what if you were able to invite ***Business experts*** into our ***Realm*** …
- This is an approach already used by ***Jane-Street***, where ***stockbrokers*** are ***paired*** with ***developers*** to teach each other their domains
  - https://www.janestreet.com/technology/
    - *"It's no secret that we're big believers in functional programming, and use OCaml, a statically typed functional language, as our primary development platform. Jane Street's technology group is small by design, which means we need to maximize the productivity of each person we hire. We believe functional programming helps us do that. But it's not about productivity alone: programming in a rich and expressive language like OCaml is just more fun."*
- They use OCaml

  Note: Actually F# started out as an .NET implementation of that language. Now they are still similar but very different languages

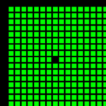


Type Driven Development
(OCaml @ Jane Street ... if you Zoom)



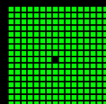
Type Driven Development (Office 365 Saturday Denmark)

- Let’s look into an example I showed at the Office 365 Saturday Denmark (OSDK) talk:
  - It’s based on a ***Travel Agency*** app.
  - I will not tell what the app does, lets ***look firstly*** into an ER (***Entity-relationship model***) digram
  - ***Afterwards*** we will ***look*** into some ***domain modeling***



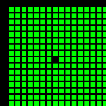
Type Driven Development (Entity–relationship model - ER)

Has
Hotel
0 .. 1
Plane
Has
1
Booking
0 .. 1
Has
Car



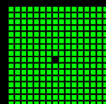
Type Driven Development (Entity–relationship model - ER)

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



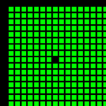
Type Driven Development (Domain)

```
open System

type Booking =
   | Basic of Plane
   | Combo of Combo
   | FullPack of Plane * Hotel * Car
and Plane   = { Outbound: DateTime; Return:    DateTime; Destination: Country }
and Combo   =
   | ``With Hotel`` of Plane * Hotel
   | ``With Car``   of Plane * Car
and Hotel   = { Arrival:  DateTime; Departure: DateTime; Location:    Country }
and Car     = { From:     DateTime; To:        DateTime; Location:    Country }
and Country = { Name:     String;   ``ISO 3166-1``: char * char }
```

Type Driven Development (Domain)

- I 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”
- I 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
  - I can also see that a Plane will require the following information (***still tuples***):
    - Plane => (Outbound date and time, Return date and time, Destination country)



Type Driven Development (Domain)

Ramón Soto Mathiesen
@genTauro42

@simped: "I thought you were doing a technical talk, there is no code in your slides". My job here is done #fsharp

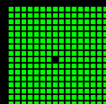
Domain represented as types

```
type Booking =
  | Basic of Plane
  | Combo of Combo
  | FullPack of Plane * Hotel * Car
and Plane   = { Outbound: DateTime; Return:    DateTime; Destination: Country }
and Combo   =
 | ``With Hotel`` of Plane * Hotel
 | ``With Car``   of Plane * Car
and Hotel   = { Arrival:  DateTime; Departure: DateTime; Location:    Country }
and Car     = { From:     DateTime; To:        DateTime; Location:    Country }
and Country = { Name:     String;   ``ISO 3166-1``: char * char }
```

RETWEETS 23 LIKES 37

10:33 AM - 8 Mar 2016

23 37

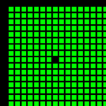


Type Driven Development (OCaml/F#)

- Well we are actually looking into code but it really look like plain English right?
- So we are actually using a bit of mathematics to provide some domain constraints (***Algebraic data types***):
  - ***Product Types***: think of it as the ***tuples*** I mentioned before

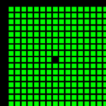
    Note: ***records types*** are also tuples, but they just have labels
  - ***Sum Types***: Think of it as ***disjoint sets***
- So what is great about using mathematics to make constraints? Well we are actually able to Make Illegal States Unrepresentable

  Note: I found an issue with my initial domain, which I fixed for this talk



Type Driven Development (Fixed)

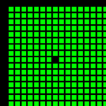
```
open System

type Booking =
   | Basic of Plane
   | Combo of Combo
   | FullPack of Plane * Hotel * Car
and Plane = { Outbound: DateTime; Return:    DateTime; Destination: City }
and Combo =
   | ``With Hotel`` of Plane * Hotel
   | ``With Car``   of Plane * Car
and Hotel = { Arrival:  DateTime; Departure: DateTime; Location:    City }
and Car   = { From:     DateTime; To:        DateTime; Location:    City }
and City  = String
```

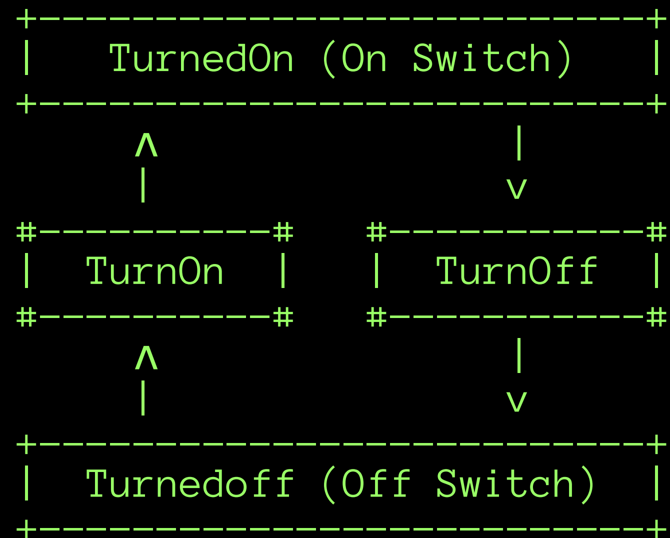
Type Driven Development (Round 2)

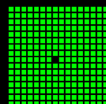
- Be careful with statements like:
  - Yaron Minsky: “Make Illegal States Unrepresentable”
  - Richard Feldman: “Making Impossible States Impossible”

  Note: I’m not dishing neither Yaron or Richard as I’m a huge fan of them both
- As it is not always possible to ensure mathematical correctness by using the ordinary TDD approach …



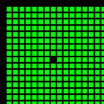
Type Driven Development (Round 2, ASCII Art for the win)





Type Driven Development (Round 2)

```
module WhatYouNormallySee =
    type State = On | Off

    (* Bug due to lack of testing
       Note: ALWAYS use FsCheck, F# implementation of Haskells QuickCheck *)
    let transition = function
        | On  -> On
        | Off -> On

    let transitionFixed = function
        | On  -> Off
        | Off -> On
```

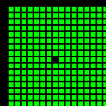
Type Driven Development (Round 2)

- How to handle State transitions in a type safe manner as we are doing with States (***States + State transitions = State machine***)
- Well firstly we will need to introduce the following three simple concepts:
  - **Phantom Types**: Are parametrised types whose parameters do not all appear on the right-hand side of its definition

    Example: type 'a Foo = Bar
  - **Function Types**: Define a function signature as a type

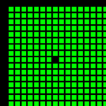
    Example (for the identity function): type 'a Id = 'a -> 'a
  - **Not accessible Sum Type Case Constructors**: By hiding the underlying case constructors for a given sum type, you can ensure that only specific parts of the code can instantiate your type

    Example: type FooBar = private | Foo of int | Bar of float



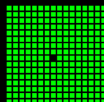
Type Driven Development (Round 2)

```
module Light =
    type 'a Switch = private | State

    and TurnedOn  = On  Switch
    and TurnedOff = Off Switch

    and On  = On
    and Off = Off

    and TurnOn  = TurnedOff -> TurnedOn
    and TurnOff = TurnedOn  -> TurnedOff

    module Switch =
        let private initHelper = State
        let private turnHelper = fun _ -> State

        let initOn  : TurnedOn  = initHelper
        let initOff : TurnedOff = initHelper

        let turnOn  : TurnOn    = turnHelper
        let turnOff : TurnOff   = turnHelper

    module Output =
        (* Expensive call cos of .NET Type Reflection *)
        let state (x:'a Switch)  =
            match typedefof<'a> with
                | t when t = typedefof<On>  -> "on"
                | t when t = typedefof<Off> -> "off"
                | _________________________ -> "invalid type"
```



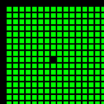
Type Driven Development (Round 2)

```
open Light

let on =
    Switch.initOff
    |> Switch.turnOn

let off =
    on
    |> Switch.turnOff

let error =
    off
    // |> Switch.turnOff
    (* error FS0001: Type mismatch. Expecting a
            TurnedOff -> 'a
        but given a
            TurnOff
        The type 'Off' does not match the type 'On' *)

// on = off
(* error FS0001: Type mismatch. Expecting a
        TurnedOn
    but given a
        TurnedOff
    The type 'On' does not match the type 'Off' *)

on  |> Output.state
off |> Output.state
```

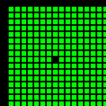
Idiomatic Data Structures (OO data structure in F#)

```
type ResizeArray<'T> = System.Collections.Generic.List<'T>

type EventStore() =
    let eventList =
        new ResizeArray<String * ScoutEvent>()

    member this.Save(name, events) =
        events |> List.iter (fun e -> eventList.Add(name, e))

    member this.Get() =
        eventList
```

Idiomatic Data Structures (OO data structure in F#)

- The ***main issue*** by introducing ***OO data structures in F#*** is that you have to think with a ***different mindset of what is expected***.
- Let me explain by looking into MSDN, where we can see that ***ResizeArray*** is just a type abbreviation for a generic ***.NET*** list



Idiomatic Data Structures (OO data structure in F#)

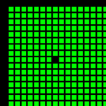
- Let's use the data structure as we normally would:

```
let xs = new ResizeArray<int>()

Array.Parallel.init 1000 (fun i -> xs.Add i) |> ignore
xs |> Seq.reduce(fun x y -> x + y)
```

- We get the following output:

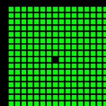
```
> val it : int = 991456
> val it : int = 1490956
> val it : int = 1990456
```

- Which is ***non-deterministic*** as well as an ***incorrect result***



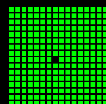
Idiomatic Data Structures (OO data structure in F#)

- So why is this happening? Well if you are used to work with the ***.NET*** platform, you might as well (if you actually read the documentation on ***MSDN***) have seen the following text on the bottom of ***almost every Class definition***, under the ***Thread Safety section***:
  - *"Public static (Shared in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe"*
- The main point here is that ***.NET collections*** are ***not immutable*** and therefore ***don't fit well*** with the ***functional paradigm that F# is mainly built-on***, even though it has support for other paradigms as imperative and OO



Idiomatic Data Structures (Data structures in F# done right)

```
module Immutable =
    type 'a iarray = private | T of 'a array with
        override ia.ToString() =
            ia |> function | T xs -> xs |> sprintf "%A"

    module Array =
        let init n f =
            Array.Parallel.init n f |> T
        let map f (T xs) =
            xs |> Array.Parallel.map f |> T
        let iter f (T xs) =
            xs |> Array.iter f
        let reduce f (T xs) =
            xs |> Array.reduce f
        let fold init f (T xs) =
            xs |> Array.fold f init
        let length (T xs) = xs |> Array.length
        let at i (T xs as ixs) =
            if i < 0 || i >= (length ixs) then
                failwith (sprintf "index: %i is out of boundries." i)
            else
                xs.[i]
        let append (T xs) (T ys) =
            Array.append xs ys |> T

        module Extra =
            let add x (T xs) =
                Array.append xs [| x |] |> T
            let pop (T xs as ixs) = length ixs |> function
                | 0 -> failwith "the array is empty."
                | 1 -> [|         |] |> T
                | n -> xs.[0 .. n-2] |> T
```

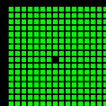
Idiomatic Data Structures (Data structures in F# done right)

- As mentioned previously in these slides, we can use **Not accessible Sum Type Case Constructors** (*) to hide the underlying case constructors for a given sum type, to ensure that only specific parts of the code can instantiate a type:

```
type 'a iarray = private | T of 'a array
```

- Combined with that I'm ***never exposing*** the ***underlying*** and ***mutable array***, therefore, as I ***don't allow any external*** piece of ***code*** to ***instantiate*** my ***type iarray unless*** it's by using the ***init function***, I can therefore ***argue*** that ***my data structure is sound*** to be used ***as an immutable F# data structure*** as the native built-in would be used

(*) Languages like ***Haskell*** and ***Elm*** achieve the same abstraction by not exposing the type(s) from their module(s)

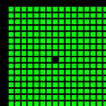


Idiomatic Data Structures (ResizeArray vs iarray)

```
let foobar =
    Array.Parallel.init 1000 id
    |> Array.reduce(fun x y -> x + y)

let foo =
    let xs = new ResizeArray<int>()

    Array.Parallel.init 1000 (fun i -> xs.Add i) |> ignore
    xs |> Seq.reduce(fun x y -> x + y)

let bar =
    let xs = Immutable.Array.init 0 id

    Array.Parallel.init 1000 (fun i -> xs |> Immutable.Array.Extra.add i)
    |> Array.reduce(fun x y -> Immutable.Array.append x y)
    |> Immutable.Array.reduce (fun x y -> x + y)
```

- Producing the following output

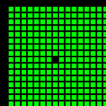
```
> val foobar : int = 499500
> val foo : int = 304641
> val bar : int = 499500
```

What if … we were able to develop ***careful*** but ***fast***? Think about that …

- *“No bug has ever been found in the ‘released for flight’ versions of that code.”*
  -- Henry Spencer (henry@spsystems.net)
- *“Now, a great deal of stuff that goes on in the aerospace industry **should not be emulated by anyone**, and is often self destructive. Most of you have probably read various popular articles about the development process that produces the space shuttle software, and while **some people might think that the world would be better if all software developers were that ‘careful’**, the truth is that we would be decades behind where we are now, with no PC’s and no public internet **if everything was developed at that snail’s pace**.”*
  -- John Carmack (lead dev for Wolfenstein 3D, Doom, Quake among others)

  Source:
  - http://number-none.com/blow/blog/programming/2014/09/26/carmack-on-inlined-code.html

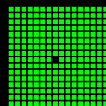


Show me some code … NOT :(

- Remarks:

  “Some of the classes and class elements in the System.Net.WebSockets namespace are supported on Windows 7, Windows Vista SP2, and Windows Server 2008. However, ***the only public implementations of client and server WebSockets are supported on Windows 8 and Windows Server 2012***. The class elements in the System.Net.WebSockets namespace that are supported on Windows 7, Windows Vista SP2, and Windows Server 2008 are abstract class elements. This allows an application developer to inherit and extend these abstract class classes and class elements with an actual implementation of client WebSockets.”

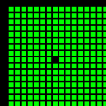
- Source:
  - MSDN ClientWebSocket Class
  - Xamarin ClientWebSocket Class



Summary

- Test vs Type Driven Development (TDD)
  - **M**ake **I**llegal **S**tates **U**nrepresentable (MISU)

    Note: Including State Machines
- Idiomatic Data Structures
  - OO data structures in F# don't really fit well the functional paradigm
- What if ... we were able to develop ***careful*** but ***fast***?
- Show me some code ... NOT :(
- Q & A



Q & A

Any Questions?