Funktionale Programmierung in F# (4)

Domain Driven Design & Property Based Testing

Göran Kirchner[1]

2022-04-29

[1] e_kirchnerg@doz.hwr-berlin.de

Programm

- Hausaufgaben (4/7)
  - ☒ Queen Attack
  - ☒ Raindrops
  - ☒ Gigaseconds
  - ☒ Bank Account
- Domain Driven Design (DDD)
- Property Based Testing

Queen Attack

```
open System
let create (row, col) = row >= 0 && row < 8 && col >= 0 && col < 8
let canAttack (queen1: int * int) (queen2: int * int) =
    let (r1, c1) = queen1
    let (r2, c2) = queen2
    Math.Abs(r1 - r2) = Math.Abs(c1 - c2) || r1 = r2 || c1 = c2
let whiteQueen1, blackQueen1 = (2, 2), (1, 1)
let test1 = canAttack blackQueen1 whiteQueen1
let whiteQueen2, blackQueen2 = (2, 4), (6, 6)
let test2 = canAttack blackQueen2 whiteQueen2
[test1; test2]
```

```
val create: row: int * col: int -> bool
val canAttack: int * int -> int * int -> bool
val whiteQueen1: int * int = (2, 2)
val blackQueen1: int * int = (1, 1)
val test1: bool = true
val whiteQueen2: int * int = (2, 4)
val blackQueen2: int * int = (6, 6)
val test2: bool = false
```

Raindrops

```
let rules =
    [ 3, "Pling"
      5, "Plang"
      7, "Plong" ]
let convert (number: int): string =
    let divBy n d = n % d = 0
    rules
    |> List.filter (fst >> divBy number)
    |> List.map snd
    |> String.concat ""
    |> function
       | "" -> string number
       | s -> s
let test = convert 105
test
```

```
val rules: (int * string) list = [(3, "Pling"); (5, "Plang"); (7, "Plong")]
val convert: number: int -> string
val test: string = "PlingPlangPlong"
val it: string = "PlingPlangPlong"
```

Gigasecond

```
let add (beginDate : System.DateTime) = beginDate.AddSeconds 1e9
let test = add (DateTime(2015, 1, 24, 22, 0, 0)) = (DateTime(2046, 10,
↪  2, 23, 46, 40))
test
```

```
val add: beginDate: DateTime -> DateTime
val test: bool = true
val it: bool = true
```

Bank Account (1)

```
type OpenAccount =
    { mutable Balance: decimal }
type Account =
    | Closed
    | Open of OpenAccount
let mkBankAccount() = Closed
let openAccount account =
    match account with
    | Closed -> Open { Balance = 0.0m }
    | Open _ -> failwith "Account is already open"
```

```
type OpenAccount =
  { mutable Balance: decimal }
type Account =
  | Closed
  | Open of OpenAccount
val mkBankAccount: unit -> Account
val openAccount: account: Account -> Account
```

Bank Account (2)

```
let closeAccount account =
    match account with
    | Open _ -> Closed
    | Closed -> failwith "Account is already closed"
let getBalance account =
    match account with
    | Open openAccount -> Some openAccount.Balance
    | Closed -> None
let updateBalance change account =
    match account with
    | Open openAccount ->
        lock (openAccount) (fun _ ->
            openAccount.Balance <- openAccount.Balance + change
            Open openAccount)
    | Closed -> failwith "Account is closed"
```

```
val closeAccount: account: Account -> Account
val getBalance: account: Account -> decimal option
val updateBalance: change: decimal -> account: Account -> Account
```

Bank Account (3)

```fsharp
let account = mkBankAccount() |> openAccount
let updateAccountAsync =
    async { account |> updateBalance 1.0m |> ignore }
let ``updated from multiple threads`` =
    updateAccountAsync
    |> List.replicate 1000
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore
let test1 = getBalance account = (Some 1000.0m)
```

```
val account: Account = Open { Balance = 1000.0M }
val updateAccountAsync: Async<unit>
```

Pause

You’re bound to be unhappy if you optimize everything.

– Donald Knuth

DDD

$\rightsquigarrow$ Domain Driven Design

Prinzipien

- Verwende die Sprache der Domäne (ubiquitous Language)
- Values und Entities
- der Code ist das Design (kein UML nötig)
- Design mit (algebraischen) Typen
    - Option statt Null
    - DU statt Vererbung
- illegale Konstellationen sollten nicht repräsentierbar sein!

Pause

Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set?

– Edsger Dijkstra

DDD Übung 1 (Contacts – ex 2)

A Contact has

- a personal name
- an optional email address
- an optional postal address
- Rule: a contact must have an email or a postal address

A Personal Name consists of a first name, middle initial, last name

- Rule: the first name and last name are required
- Rule: the middle initial is optional
- Rule: the first name and last name must not be more than 50 chars
- Rule: the middle initial is exactly 1 char, if present

A postal address consists of a four address fields plus a country

- Rule: An Email Address can be verified or unverified

DDD Übung 2 (Payments – ex 3)

The payment taking system should accept:

- Cash
- Credit cards
- Cheques
- Paypal
- Bitcoin

A payment consists of a:

- payment
- non-negative amount

After designing the types, create functions that will:

- print a payment method
- print a payment, including the amount
- create a new payment from an amount and method

DDD Übung 3 (Refactoring – ex 4)

Much C# code has implicit states that you can recognize by fields called "IsSomething", or nullable date.
This is a sign that states transitions are present but not being modelled properly.

DDD Übung 4 (Shopping Cart – fsm ex 3)

Create types that model an e-commerce shopping cart.

- Rule: ”You can’t remove an item from an empty cart”!
- Rule: ”You can’t change a paid cart”!
- Rule: ”You can’t pay for a cart twice”!

States are:

- Empty
- ActiveCartData
- PaidCartData

Pause

About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.

– Edsger Dijkstra

Example Based Tests :)

```
open System
#r "../src/4/02_PBT/lib/Expecto.dll"
open Expecto
let expectoConfig = {defaultConfig with colour =
↪  Expecto.Logging.ColourLevel.Colour0}
#load "../src/4/02_PBT/A1_Add_Implementations.fsx"
open A1_Add_Implementations
```

```
module Test1 =
    open Implementation1
    let tests = testList "implementation 1" [
        test "add 1 3 = 4" {
            let actual = add 1 3
            let expected = 4
            Expect.equal expected actual "" }
        test "add 2 2 = 4" {
            let actual = add 2 2
            let expected = 4
            Expect.equal expected actual "" } ];;
```

Evil Developer From Hell :(

```
module Implementation1 =
    let add x y =
        4
```

```
module Implementation1 =
  val add: x: 'a -> y: 'b -> int
```

PBT

$\rightsquigarrow$ Property Based Testing

FsCheck

```
open System
#r "../src/4/02_PBT/lib/FsCheck.dll"
```

```
let add1 x y = x + y
let add2 x y = x - y
let commutativeProperty f x y =
   let result1 = f x y
   let result2 = f y x
   result1 = result2;;
FsCheck.Check.Quick (commutativeProperty add1)
FsCheck.Check.Quick (commutativeProperty add2)
```

```
Ok, passed 100 tests.
Falsifiable, after 1 test (0 shrinks) (StdGen (501447179, 297030377)):
Original:
0
1
val it: unit = ()
```

FsCheck (Generate)

```fsharp
type Temp = F of int | C of float;;
let fGen =
    FsCheck.Gen.choose(32,212)
    |> FsCheck.Gen.map (fun i -> F i);;
let cGen =
    FsCheck.Gen.choose(0,100)
    |> FsCheck.Gen.map (fun i -> C (float i));;
let tempGen =
    FsCheck.Gen.oneof [fGen; cGen]

let test = tempGen |> FsCheck.Gen.sample 0 20
test
```

```
val tempGen: FsCheck.Gen<Temp> = Gen <fun:Bind@88>
val test: Temp list =
  [C 86.0; C 5.0; F 53; F 211; F 179; F 156; F 124; F 101; C 62.0; F 136;
   C 75.0; F 32; C 25.0; F 158; C 60.0; F 199; C 89.0; F 124; F 34; C 71.0]
val it: Temp list =
  [C 86.0; C 5.0; F 53; F 211; F 179; F 156; F 124; F 101; C 62.0; F 136;
   C 75.0; F 32; C 25.0; F 158; C 60.0; F 199; C 89.0; F 124; F 34; C 71.0]
```

FsCheck (Shrink)

```
open FsCheck
let smallerThan81Property x = x < 81
FsCheck.Check.Quick smallerThan81Property

let test1 = FsCheck.Arb.shrink 100 |> Seq.toList
let test2 = FsCheck.Arb.shrink 88 |> Seq.toList
test2
```

```
Ok, passed 100 tests.
val smallerThan81Property: x: int -> bool
val test1: int list = [0; 50; 75; 88; 94; 97; 99]
val test2: int list = [0; 44; 66; 77; 83; 86; 87]
val it: int list = [0; 44; 66; 77; 83; 86; 87]
```

Auswahl der Eigenschaften

- Unterschiedlicher Weg, gleiches Ziel (Map(f)(Option(x))=Option(f x))
- Hin und Her (z.B. Reverse einer Liste)
- Invarianten (z.B. Länge einer Liste bei Sortierung)
- Idempotenz (noch einmal bringt nichts mehr)
- Divide et Impera! (teile und herrsche)
- Hard to prove, easy to verify (Primzahlzerlegung)
- Test-Orakel (z.B. einfach aber langsam)

Zusammenfassung

- funktionales Domain Modeling (DDD)
- eigenschaftsbasiertes Testen (Property Based Testing)

Links

- Domain Driven Design
- Domain Modeling Made Functional
- FsCheck
- An introduction to property-based testing
- Choosing properties for property-based testing