Funktionale Programmierung in F# (4)

Domain Driven Design & Property Based Testing

Göran Kirchner[1]

2021-04-09

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

Programm

- Domain Driven Design (DDD)
- Property Based Testing

DDD

- ⇝ 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)

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)

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

Übung 3 (Refactoring)

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.

Übung 4 (Shopping Cart)

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 :)

```
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 "" } ];;
runTests expectoConfig Test1.tests
```

```
[00:51:05 INF] EXPECTO? Running tests... <Expecto>
[00:51:05 INF] EXPECTO! 2 tests run in 00:00:00.0102279 for implementation 1 ?
2 passed, 0 ignored, 0 failed, 0 errored. Success! <Expecto>
```

Evil Developer From Hell :(

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

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

PBT

- ⇝ Property Based Testing

FsCheck

```
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 (1 shrink) (StdGen (74373745, 296875694)):
Original:
2
1
Shrunk:
0
1
```

FsCheck (Generate)

```
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 0 100
test
```

```
val tempGen : Gen<Temp> = Gen <fun:Bind@88>
val test : Temp list =
  [C 81.0; F 196; C 70.0; C 38.0; C 25.0; F 166; C 31.0; C 7.0; F 105; F 73;
   F 50; F 199; C 3.0; C 94.0; C 13.0; C 3.0; C 23.0; F 34; F 192; F 160;
   F 137; C 33.0; C 53.0; F 50; C 63.0; F 32; F 192; F 85; F 53; F 211; C 66.0;
   C 86.0; C 76.0; C 96.0; F 69; F 46; F 195; F 172; F 140; C 25.0; C 15.0;
   C 35.0; C 25.0; F 188; F 156; F 133; F 101; C 65.0; C 55.0; C 75.0; C 65.0;
```

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
```

```
Falsifiable, after 98 tests (1 shrink) (StdGen (76163595, 296875694)):
Original:
83
Shrunk:
81
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]
```

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

Hausaufgabe

- exercism.io (E-Mail bis 16.4)
  - ☐ Accumulate
  - ☐ Space Age
  - ☐ Poker (Programmieraufgabe)