Developer Week 2016

EIGENSCHAFTSBASIERENDES TESTEN

MIT F# UND FSCHECK

Carsten König

BEISPIEL SORTIEREN

Funktion

```
sortieren : ('a list -> 'a list)
    when 'a : comparison
```

testen

UNIT-TESTS

EINZELNE FÄLLE

im wesentlichen Arrange Act Assert

```
 1: [<Test>]
 2: member __.``Sortieren der leeren Liste ist die leere Liste``() =
 3:     sortieren [] |> should equal []
 4:
 5: [<Test>]
 6: member __.``Sortieren von [1] ist [1]``() =
 7:     sortieren [1] |> should equal [1]
 8:
 9: [<Test>]
10: member __.``Sortieren von [2;1] ist [1;2]``() =
11:     sortieren [2;1] |> should equal [1;2]
```

NICHT SO SCHÖN...

- Testdaten manuell gewählt
- Viele Code-Wiederholungen
- Spezifikationen nicht direkt klar

TEST-CASES

```
1: [<TestCase([],[])>]
2: [<TestCase([1],[1])>]
3: [<TestCase([2;1],[1;2])>]
4: member __.``Sortiert Eingabe richtig``(eingabe, erwartet) =
5:     sortieren eingabe |> should equal erwartet
```

BESSER

- Weniger Code-Wiederholungen
- Einfach zu erweitern

ABER ...

- manuelle Testdatenerzeugung
- Spezifikationen nicht direkt klar

ZUSAMMENFASSUNG

Mit Unit-Tests werden **bekannte Eingaben** mit **erwarteten Ausgaben** verglichen.

EIGENSCHAFTSBASIERENDE TESTS

IDEE:

- Testdaten automatisch (zufällig) erzeugen
- Viele Testdurchläufe
- Findet hoffentlich Randfälle und Ausnahmen

PROBLEM

Woher kommen die *erwarteten Ausgaben*?

LÖSUNG

nicht *erwartete Ergebnis* für *Eingabe* prüfen

sondern

Eigenschaft der zu testenden Funktion

```
Eingenschaft : Eingabe -> Ausgabe -> Bool
```

BEISPIELE

- jedes Element der sortierten Liste ist nicht-größer als sein Nachfolger
- Anzahl der Elemente bleibt beim Sortieren gleich
- nocheinmal sortieren ändert die Liste nicht mehr

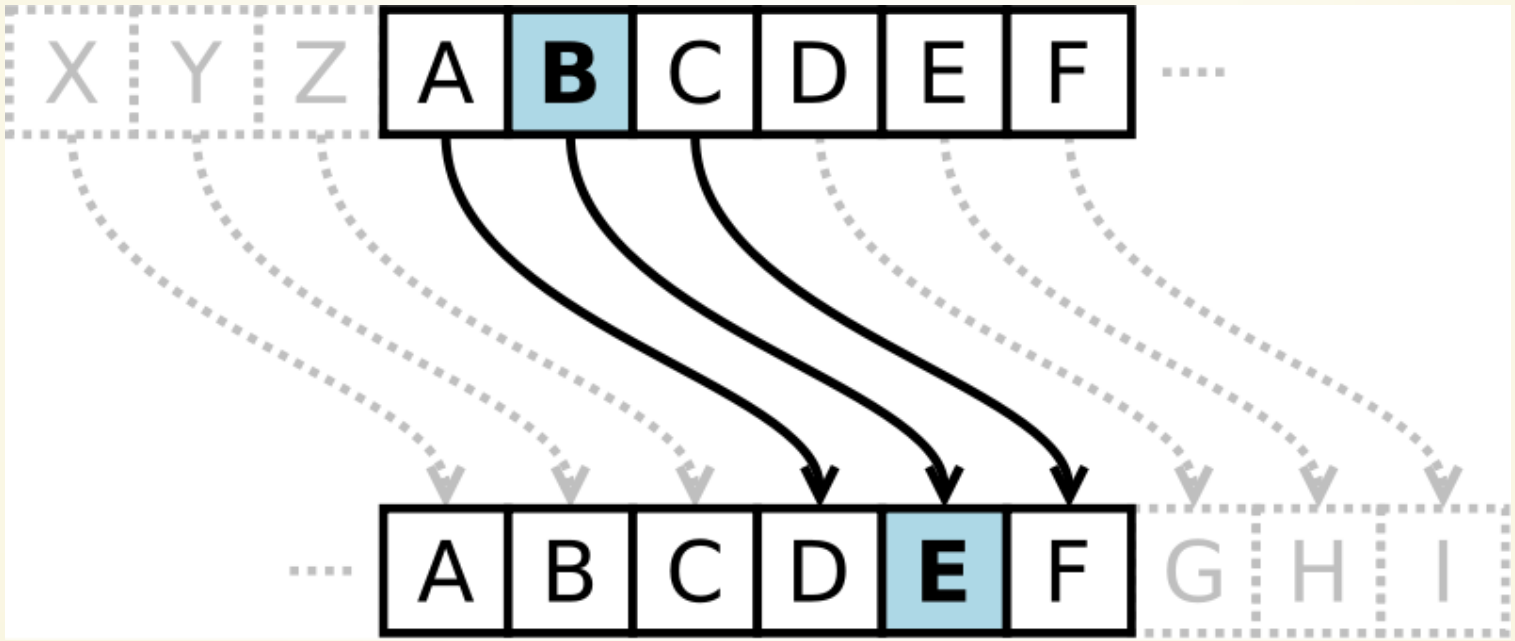
```fsharp
let rec istSortiert =
    function
    | [] | [_] -> true
    | a::b::rest ->
        a <= b && istSortiert (b::rest)


let ``nach Sortieren ist jedes Element kleiner-gleich seinem Nachfolger``
    (zahlen : int list) =
    sortieren zahlen |> istSortiert
```

```
1: let ``Sortieren ändert Länge der Liste nicht``
2:     (zahlen : int list) =
3:     let sortiert = sortieren zahlen
4:     List.length sortiert = List.length zahlen
```

```
1: let ``nochmal sortieren ändert nichts``
2:     (zahlen : int list) =
3:     let einmal = sortieren zahlen
4:     let zweimal = sortieren einmal
5:     einmal = zweimal
```

DEMO
SORTIEREN

DEMO
VERSCHLÜSSELN



GENERATOREN

Instanz von `Arbitrary` schreiben und registrieren

```
1: type MyGenerators =
2:     static member KlartextNachricht() =
3:         { new Arbitrary<Nachricht<Klartext>>() with
4:         ...
5:         }
6:
7: Arb.register<MyGenerators>()
```

GENERATOR

```
1: override __.Generator =
2:     Gen.elements ['A'..'Z']
3:     |> Gen.nonEmptyListOf
4:     |> Gen.map (List.toArray >> String)
5:     |> Gen.map nachricht
```

SHRINKER

```
override __.Shrinker s =
    let s = string s
    if s.Length < 2 then Seq.empty else
    let mid = s.Length / 2
    seq [ s.Substring(0,mid); s.Substring(mid) ]
    |> Seq.map nachricht
```

DEMO

WIE FINDE ICH EIGENSCHAFTEN

EIGENSCHAFTEN/SPEZIFIKATIONEN

```
1: let rec istSortiert =
2:     function
3:     | [] | [_] -> true
4:     | a::b::rest ->
5:         a <= b && istSortiert (b::rest)
6:
7: let ``nach Sortieren ist jedes Element kleiner-gleich seinem Nachfolger``
8:     (zahlen : int list) =
9:     sortieren zahlen |> istSortiert
```

INVARIANZ

```
1: let ``Sortieren ändert Länge der Liste nicht``
2:     (zahlen : int list) =
3:     let sortiert = sortieren zahlen
4:     List.length sortiert = List.length zahlen
```

IDEMPOTENZ

```
1: let ``nochmal sortieren ändert nichts``
2:     (zahlen : int list) =
3:     let einmal = sortieren zahlen
4:     let zweimal = sortieren einmal
5:     einmal = zweimal
```

INVERSE

```
1: let ``verschlüsseln >> entschlüsseln = id``
2:     (key : Schlüssel, msg : Nachricht<Klartext>) =
3:         let ver = Caesar.verschlüsseln key msg
4:         let ent = Caesar.entschlüsseln key ver
5:         msg = ent
```

VERIFIZIEREN

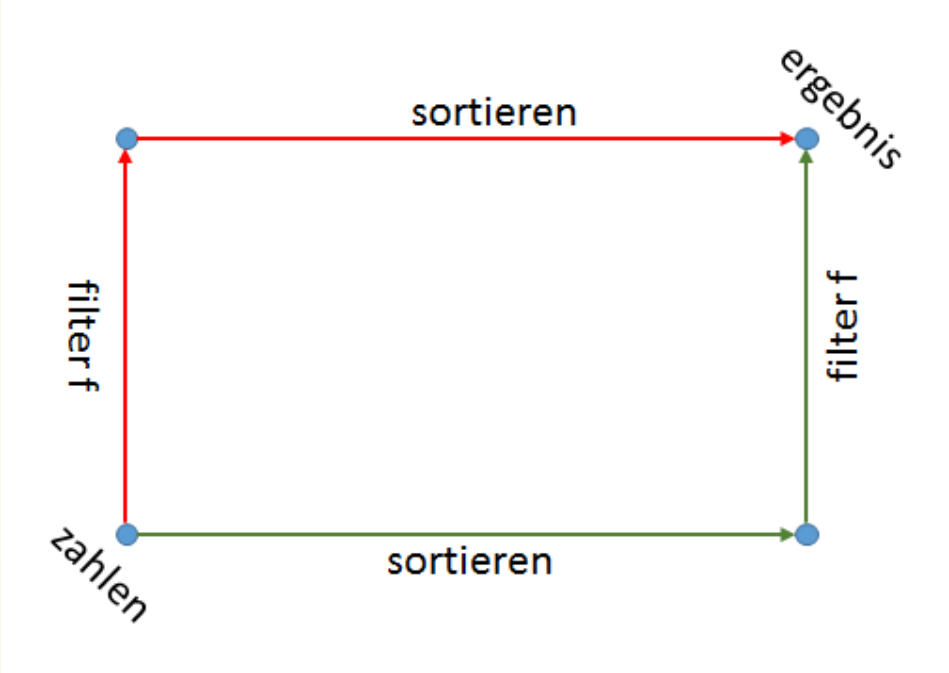
```
1: let ``Primfaktorzerlegung``
2:     (zahl : int) =
3:     let zerlegung = primFaktoren zahl
4:     let produkt = Seq.fold (*) 1 zerlegung
5:     produkt = zahl
```

KOMMUTATIVE DIAGRAMME

```
1: let ``Sort kommutiert mit Filter``
2:     (zahlen : int list)
3:     (f: int -> bool) =
4:     let weg1 = sortieren zahlen |> List.filter f
5:     let weg2 = List.filter f zahlen |> sortieren
6:     weg1 = weg2
```

KOMMUTATIVE DIAGRAMME

```
1: let ``Sort kommutiert mit Filter``
2:       ...
```



REFERENZ IMPLEMENTATION

```
1: let ``Ergibt das Selbe wie die Referenz``
2:     (zahlen : int list)
3:     sortieren zahlen = List.sort zahlen
```

VERBINDUNG MIT TESTFRAMEWORKS/RUNNERN

XUNIT / NUNIT

`Property`-Attribut über Nuget-Package

```
1: [<Property>]
2: let ``nach Sortieren ist jedes Element kleiner-gleich seinem Nachfolger``
3:     (zahlen : int list) =
4:     sortieren zahlen |> istSortiert
```

- FsCheck.XUnit
- FsCheck.NUnit

ÜBER EXCEPTIONS

einfach `Check.QuickThrowOnFailure` benutzen

REFERENZEN

- FsCheck
- Wiki: Analoge Frameworks in anderen Sprachen
- "Paper" zu QuickCheck

VIELEN DANK!

Carsten König

- Twitter: CarstenK_dev
- Email: Carsten@gettingsharper.de
- Web: gettingsharper.de