FUNKTIONALE PROGRAMMIERUNG

FÜR OO ENTWICKLER

Patrick Drechsler

Softwerkskammer Nürnberg 1.2.2018

WAS IST FUNKTIONALE PROGRAMMIERUNG?

- Sprachunabhängig
- **Nur ein Paradigma!**
  - andere Paradigmen:
    - Prozedural
    - Objektorientiert
    - Logisch

Unit Currying Higher Order

Functions Event Sourcing/CQRS Applicatives Monad

filter/map/reduce bind side effects

purity honest functions Functor

Immutability category

theory Monoid tuples discriminated unions elevated

types Typed FP Either Option arrow notation

railway oriented programming Lambda

Composition

FP KONZEPTE

IMMUTABILITY

- Lambdas: Sprachfeatures verwenden (LINQ, Streaming API)
- Value Objects ("fight primitive obsession")

Immutability ist eine häufige Voraussetzung für viele weiteren FP Konzepte.

das ist ok:

Javascript

```
let list = [1, 2, 3, 4, 5];
for (let i = 0; i < list.length; i++) {
    list[i] = list[i] + 1;
}
console.log(list)
```

...aber das einfacher:

Javascript

```
let list = [1, 2, 3, 4, 5];
let result = list.map(x -> x + 1); // oder eine "addOne" Funktion nehmen
console.log(list)
```

ok...

C#

```csharp
public Risk CheckRisk(int age) // <- primitive obsession
{
    if (age <= 0) { /* error handling */ }
    else if (age > 120) { /* error handling */ }
    else if (age < 20) { return Risk.Low }
    else if (age < 40) { return Risk.Medium }
    else { return Risk.High }
}
```

...weniger "Krach":

C#

```csharp
// "Age": immutable value object
public Risk CheckRisk(Age age)
{
    if (age < 20) { return Risk.Low }
    else if (age < 40) { return Risk.Medium }
    else { return Risk.High }
}
```

MEHR RECHTE FÜR FUNKTIONEN!

- **Expressions** statt Statements
- **Higher Order Functions**: Methoden können auch Funktionen zurückgeben
  - → Currying/Applicative Functions

EXPRESSIONS

C#

```
// statement
public int AddOne(int i)
{
    return i + 1;
}
```

C#

```
// expression
public int AddOne(int i) => i + 1;
```

HIGHER ORDER FUNCTIONS

C#

```
// int -> (int -> bool)
Func<int, bool> IsDivisibleBy(int divisor) => num => num % divisor == 0;
// (int -> bool)
var isDivisibleByFive = IsDivisibleBy(5);

isDivisibleByFive(10); // TRUE
```

F#

```
// int -> (int -> bool)
let isDivisibleBy divisor = (fun num -> num % divisor = 0)
// (int -> bool)
let isDivisibleByFive = isDivisibleBy 5

10 |> isDivisibleByFive // TRUE
```

COMPOSITION

- Funktionen miteinander kombinieren (Alternative zu Ableitung in OO)
  - z.B. Method Chaining (LINQ)
  - → kann IoC ersetzen

COMPOSITION

C#

```
Func<int, bool> isLargerThanFive = x => x > 5;
Func<int, bool> isSmallerThenTen = x => x < 10;

Func<int, bool> isBetweenFiveAndTen = x =>
    isLargerThanFive(x) && isSmallerThenTen(x);

isBetweenFiveAndTen(7); // TRUE
```

COMPOSITION

C#

```csharp
static string Abbreviate(string s) => s.SubString(0, 2).ToLower();

static string AbbreviateName(Person p)
    => Abbreviate(p.FirstName) + Abbreviate(p.LastName);

static string AppendDomain(string localPart)
    => $"{localPart}@company.com";

// composition
Func<Person, string> emailFor = p => AppendDomain(AbbreviateName(p));

var joe = new Person("Joe", "Smith")
emailFor(joe).Should().Be("josm@company.com");
```

C#

```csharp
// method chaining (using C# Extensions)
static string AbbreviateName(this Person p)
    => Abbreviate(p.FirstName) + Abbreviate(p.LastName);

static string AppendDomain(this string localPart)
    => $"{localPart}@company.com";

joe.AbbreviateName().AppendDomain().Should().Be("josm@company.com");
```

COMPOSITION

F#

```
let add1 x = x + 1
let times2 x = x * 2

let add1Times2 x = times2(add1 x) // ok...

let add1Times2 = add1 >> times2   // ">>": composition operator
```

F#

```
open System
type Person = { FirstName: string; LastName: string }
let p = {FirstName = "Joe"; LastName = "Smith"}
let abbreviate (s: string) = s.[0..1].ToLower()
let abbreviateName p = abbreviate(p.FirstName) + abbreviate(p.LastName)
let appendDomain (s: string) = s + "@company.com"

let emailFor = abbreviateName >> appendDomain

p |> emailFor // josm@company.com
```

SAFETY THROUGH TYPES

- Stärkeres Typsystem kann Entwicklung erleichtern
  - Discriminated Union
  - Wrapper wie Option, Either, etc

TYPSYSTEM

C#

```
public Option<Customer> GetCustomer(int id) { /* ... */ }

public string Greet(int id)
    => GetCustomer(id).Match(
            None: () => "Sorry, who?",
            Some: (customer) => $"Hello, {customer.Name}");
```

TYPSYSTEM MIT BUSINESS-LOGIK

F#

```
open System
type AccountStatus = // discriminated union
    Requested | Active | Frozen | Dormant | Closed

type CurrencyCode = string // "type alias"

type Transaction = { // record type
    Amount: decimal
    Description: string
    Date: DateTime
}

type AccountState = {
    Status: AccountStatus
    Currency: CurrencyCode
    AllowedOverdraft: decimal
    TransactionHistory: Transaction list
}

type AccountState with
member this.WithStatus(status) = { this with Status = status }
member this.Add(transaction) =
    { this with TransactionHistory =
        transaction :: this.TransactionHistory }
```

ZUSAMMENFASSUNG

- Immutability
- Expressions
- HOF
- Composition
- Typsystem

Vorschläge?

- welche FP Konzepte sind für OO Programmierer interessant?
- **in welcher Reihenfolge sollte diese Konzepte vorgestellt werden?**
- Konzepte: immutability, lambdas (filter/map/reduce), applicatives, HOF, option, typed FP
- **was wird immer falsch gemacht bei der Einführung in FP?**
- was sind die einfachen, was die schwierigen Konzepte von FP?
- **welche Konzepte beißen sich (OO vs FP)?**
- **Erfahrungen aus der Praxis**
- Unterschiede beim Testing (FP Leute machen gerne REPL plus Property Based Testing)

DANKE!

Kontaktinfos:

- 🐦 @drechsler
- ✉ socialcoding@pdrechsler.de

RESOURCES

- Videos
  - One kata, 3 languages
  - Functional Principles for Object-Oriented Development
  - What Every Hipster Should Know About Functional Programming
  - Don't fear the Monad
- Blog
  - Less is more: language features
  - Partial Application in C#
- Books
  - Functional Programming in C#. *Enrico Buonanno*
  - Domain modeling made Functional. *Scott Wlaschin*