# Funktionale Programmierung

@mobilgroma
@drechsler
Redheads Ltd.

# PATRICK DRECHSLER

- Software Entwickler / Architekt
- Beruflich: C# (und TS/JS)
- Interessen:
  - Software Crafting
  - Domain-Driven-Design
  - Funktionale Programmierung
- ...

@drechsler draptik

red·head·s
SOFTWARE SOLUTIONS SUPPORT

# MARTIN GROTZ

- Software Entwickler
- Interessen:
  - Software Crafting
  - Domain-Driven-Design
  - Funktionale Programmierung
- ...

red·head·s
SOFTWARE SOLUTIONS SUPPORT

# VORSTELLUNGSRUNDE

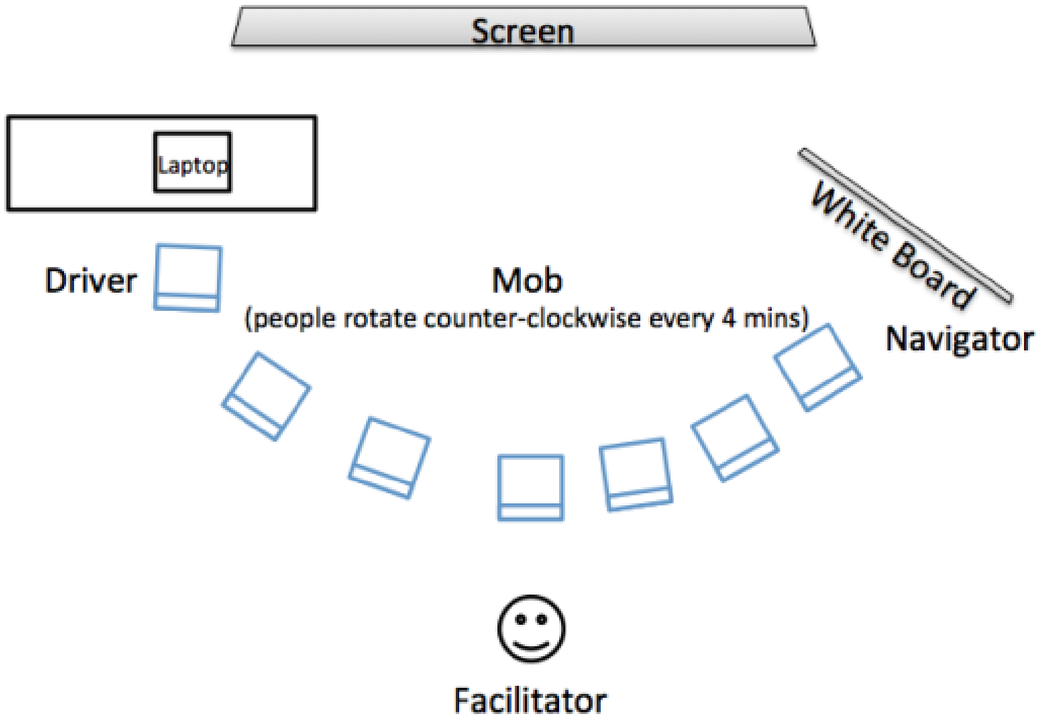
# &

# ERWARTUNGEN

# MOB PROGRAMMING

- wir lernen gemeinsam
- Pair Programming in der Gruppe

# Mob Programming Setup



- Driver: Sitzt an der Tastatur (darf nicht denken)
- Navigator: Sagt dem Driver, was zu tun ist
- Mob: Unterstuetzt den Navigator
- Regelmaessiger Wechsel (3-5min)

# "ASSISTED" MOB PROGRAMMING

- Facilitator unterstuetzt den Navigator

# FP 101

- Immutability
- Functions as First Class Citizens
- Pure Functions (see Immutability)

That's it!

# IMMUTABILITY IN C#

```
public class Customer
{
  public string Name { get; set; } // set → mutable :-(
}
```

VS

```
public class Customer
{
  public Customer(string name)
  {
    Name = name;
  }

  public string Name { get; } // ← immutable
}
```

# Syntax matters!

## Classic C#

```
int Add(int a, int b)
{
  return a + b;
}
```

## Expression body

```
int Add(int a, int b) ⇒ a + b;
```

# Syntax matters!

## Classic C#

```
int Add(int a, int b)
{
  Console.WriteLine("bla"); // ← side effect!
  return a + b;
}
```

## Expression body: side effects are less likely

```
int Add(int a, int b) ⇒ a + b;
```

# 1ST CLASS FUNCTIONS IN C#

```
// Func as parameter
public string Greet(Func<string, string> greeterFunction, string name)
{
  return greeterFunction(name);
}
```

```
Func<string, string> formatGreeting = (name) ⇒ $"Hello, {name}";
var greetingMessage = Greet(formatGreeting, "dodnedder");
// → greetingMessage: "Hello, dodnedder"
```

# PURE FUNCTIONS IN C#

- haben niemals Seiteneffekte!
- sollten immer nach `static` umwandelbar sein

```
// method signature lies!
int Add(int a, int b)
{
  Console.WriteLine("foo"); // ← side effect!
  return a + b;
}
```

```
int Add(int a, int b)
{
  Console.WriteLine("foo");
  return a + b;
}
```

```
int Add(int a, int b)
{
  return a + b;
}
```

# Expression body syntax:

```
int Add(int a, int b) ⇒ a + b;
```

Und was hat es mit

Filter

Map

Reduce

auf sich?

Schränken uns diese FP Paradigmen ein?

Wie kann man mit diesem "Purismus" Software schreiben, die etwas tut?

# LET'S CODE

(almost)

One more thing: Die Fachlichkeit

(laesstiges Detail...)

# KONTAKT-APP

# KONTAKT-APP

## Einträge

- Hinzufügen
- Auflisten
- Löschen

- Eintrag
  - Vorname
  - Nachname
  - (Kontaktmethode)
  - optionale Felder
    - Geburtstag
    - Twitter-Link

# PERSISTENZ

- Eintrag wird serialisiert und in Datei abgelegt
- Eine Datei für alle Einträge
- In der Praxis problematisch: Gleichzeitiger Zugriff auf die gleiche Datei

Und warum hat C# andere Verben verwendet?

- Als C# LINQ eingefuehrt hat, war FP nicht hipp
- SQL ist funktional

-> Syntax

# LINQ - FUER LISTEN (IENUMERABLE IN C#)

Allg.: Funktionen, die auf eine Liste angewendet werden

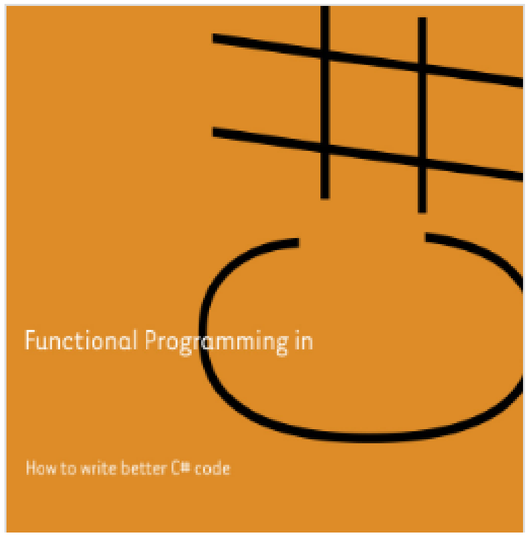
Bsp:

- Option ist eigentlich nur eine Liste mit 2 Werten (Some und None)
- Result -> Liste mit 2 Werten (Left und Right)
- etc.

In FP unterscheidet man die Wrapper-Klassen (zB IEnumerable) anhand der Funktionen, die sie bereitstellen

LAYUMBA

# LAYUMBA




## Functional Programming in C# ♡

How to write better C# code

Enrico Buonanno

August 2017 · ISBN 9781617293955 · 408 pages · printed in black & white

> Functional programming can make your head explode. This book stitches it back together.
>
> Daniel Marbach, Particular Software

# LAYUMBA

"einfache" Variante von language-ext

# LAYUMBA

- NuGet Paket
- kann nicht alles
- Fokus: Didaktik (Aehnlichkeit mit F#, Haskell)

# LAYUMBA

alle C# Beispiele haben..

```
using LaYumba.Functional;             // ← lib
using static LaYumba.Functional.F;    // ← extensions
```

# FP-KONZEPTE

- Gängige Vorgehensweise: Kleine Funktionen werden zu immer größeren Funktionalitäten zusammengesteckt
- Problem: Nicht alle Funktionen passen gut zusammen

# WERT IN CONTAINER,
# FUNKTION PASST NICHT

# F#

```
module X

let toUpper (s : string) = s.ToUpper()

let stringToOption s =
    if String.IsNullOrWhiteSpace s then
        None
    else
        Some s

let nonEmptyStringToUpper s =
    let nonEmpty = stringToOption s
    // passt nicht: "string" erwartet, aber "string option" bekommen
    let nonEmptyUpper = toUpper nonEmpty
```

# C#

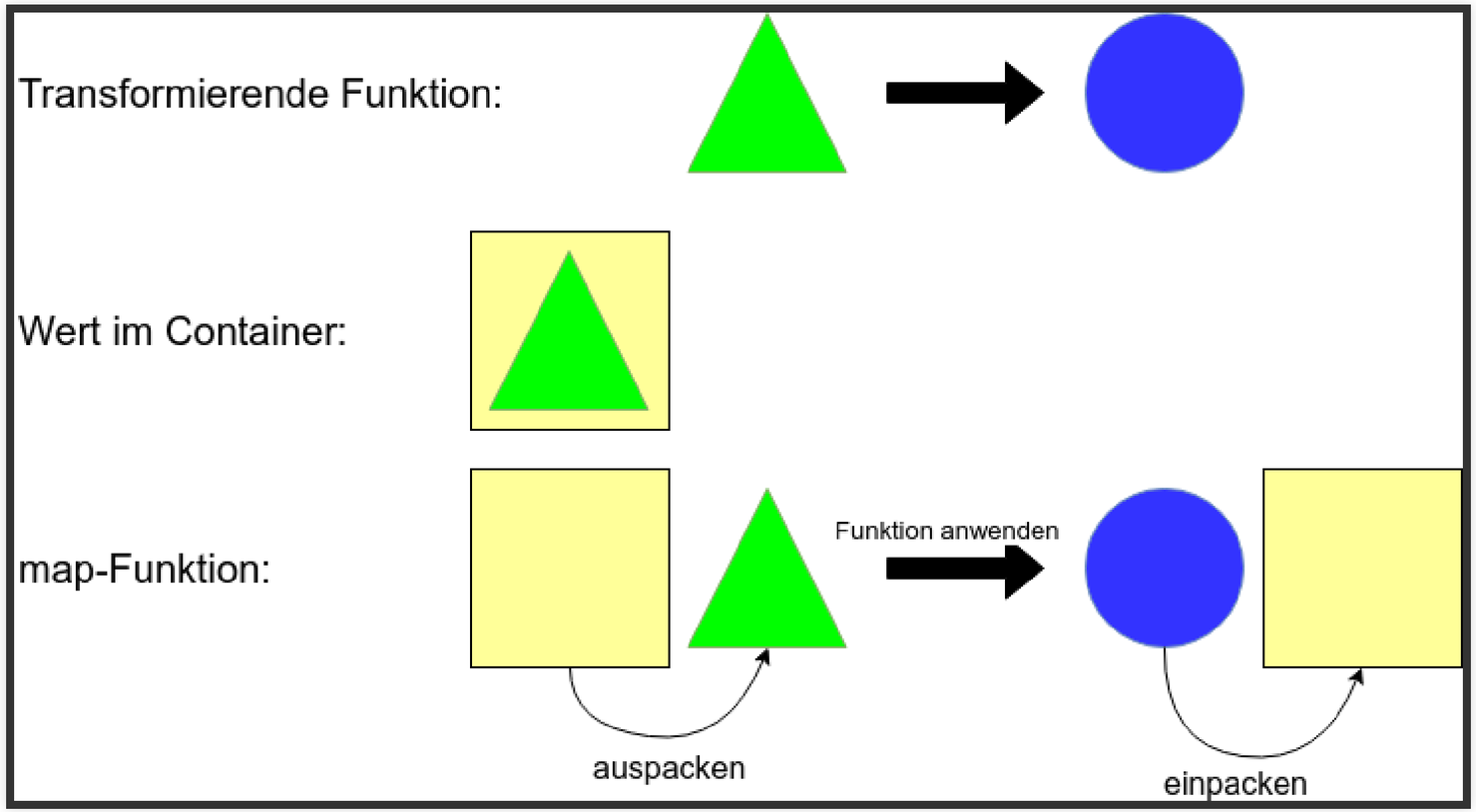
```
using LaYumba.Functional;
using static LaYumba.Functional.F;

static class X
{
  string ToUpper(string s) ⇒ s.ToUpper();

  Option<string> StringToOption(string s)
    ⇒ string.IsNullOrEmpty(s) ? None : Some(s)

  NonEmptyStringToUpper(string s)
  {
    var nonEmpty = StringToOption(s);
    // passt nicht: "string" erwartet, aber "string option" bekommen
    return ToUpper(s);
  }
}
```

# MAPPABLE


Transformierende Funktion:
Wert im Container:
map-Funktion:
Funktion anwenden
auspacken
einpacken

# MAPPABLE

- Container mit "map" Funktion (die bestimmten Regeln folgt): Mappable
- Bezeichnung in der FP-Welt: **Funktor**
- ```
  map: (a → b) → F a → F b
  ```
- Andere Bezeichnungen für "map": fmap (z.B. in Haskell), Select (LINQ), <$>, <!>
  ```

# WERT IN CONTAINER, FUNKTION PASST NICHT

```
let toUpper (s : string) = s.ToUpper()

let stringToOption s =
    if String.IsNullOrWhiteSpace s then
        None
    else
        Some s

let nonEmptyStringToUpper s =
    let nonEmpty = stringToOption s
    let nonEmptyUpper = Option.map toUpper nonEmpty
```

# VERKETTUNG

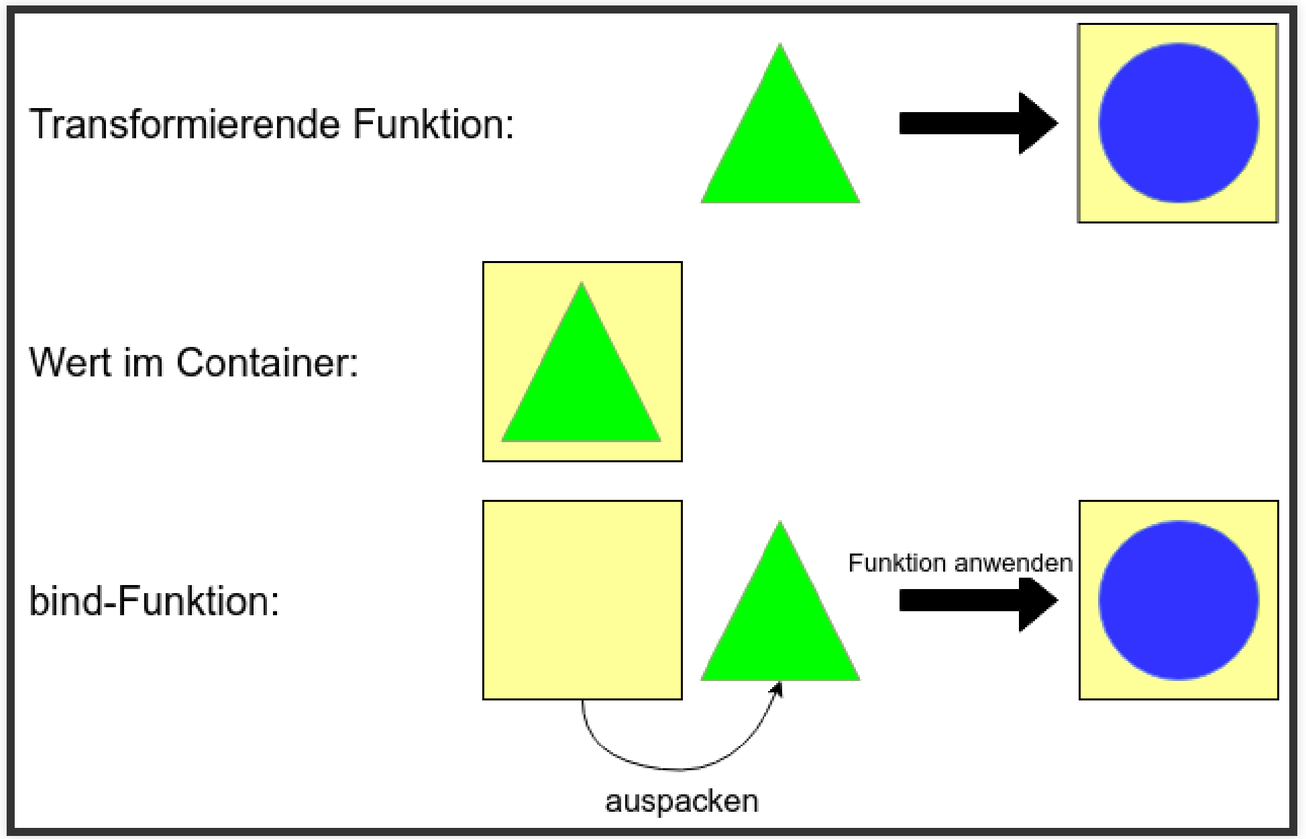
```
let storeInDatabase path content =
  try
    System.IO.File.WriteAllText(path, content)
    Some content
  with
  | :? Exception as ex → None

let stringToOption s =
    if String.IsNullOrWhiteSpace s then
        None
    else
        Some s

let toUpper (s : string) = s.ToUpper()

let nonEmptyStringStoreInPersistenceAndToUpper path content =
    let nonEmpty = stringToOption content
    // passt nicht: "string" erwartet, aber "string option" bekommen
    let stored = storeInDatabase path nonEmpty
    // passt nicht: "string option" erwartet, aber "string option option"
    let nonEmptyUpper = Option.map toUpper stored
```

# CHAINABLE


Transformierende Funktion:
Wert im Container:
bind-Funktion:
Funktion anwenden
auspacken

# CHAINABLE

- Container mit "bind" Funktion (die bestimmten Regeln folgt): Chainable
- Bezeichnung in der FP-Welt: **Monade**
- ```
  bind: (a → M b) → M a → M b
  ```
- Andere Bezeichnungen für "bind": flatMap, SelectMany (LINQ), >>=
  ```

# VERKETTUNG

```
let storeInDatabase path content =
  try
    System.IO.File.WriteAllText(path, content)
    Some content
  with
  | :? Exception as ex → None

let stringToOption s =
    if String.IsNullOrWhiteSpace s then
        None
    else
        Some s

let toUpper (s : string) = s.ToUpper()

let nonEmptyStringStoreInPersistenceAndToUpper path content =
    let nonEmpty = stringToOption content
    let stored = Option.bind (storeInDatabase path) nonEmpty
    let nonEmptyUpper = Option.map toUpper stored
```

# FUNKTION MIT MEHREREN PARAMETERN

```
let add a b = a + b

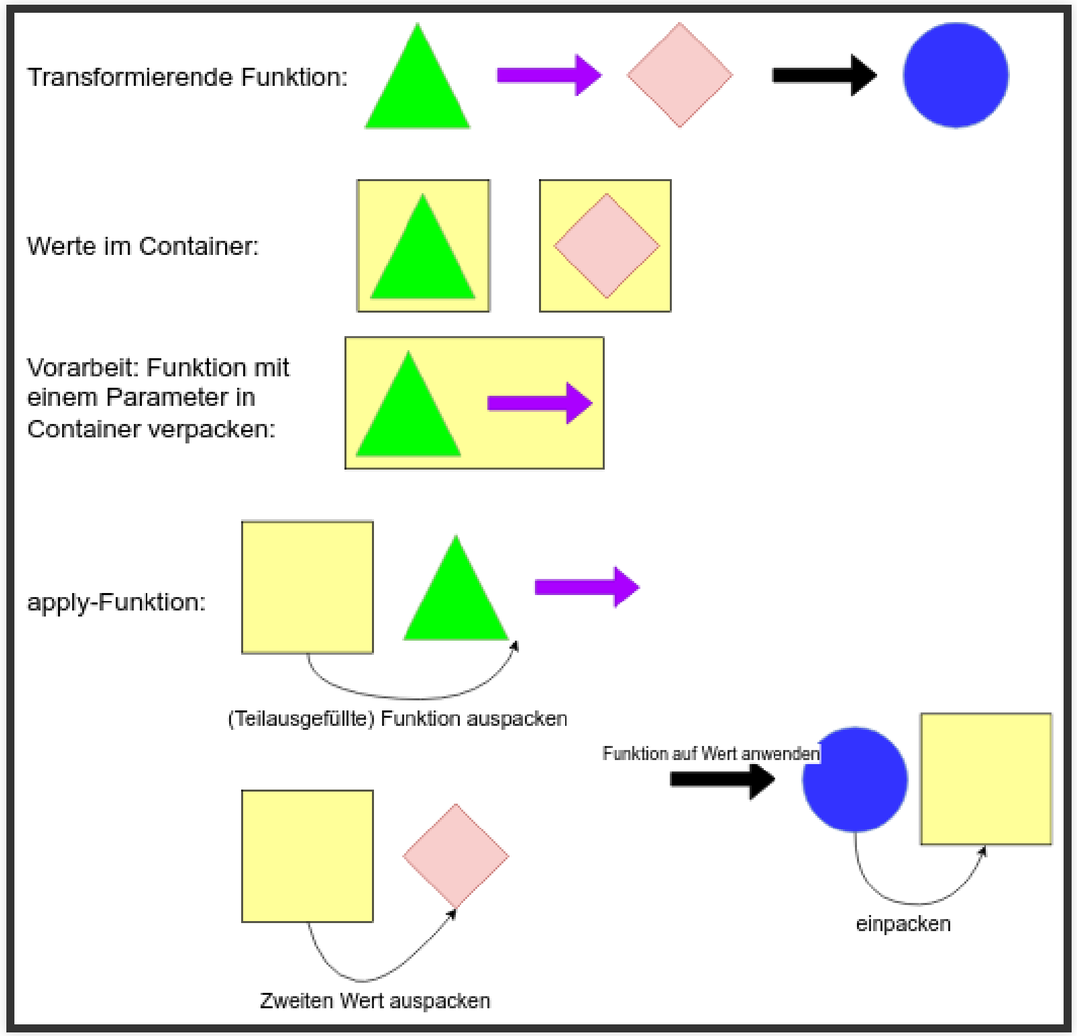
let onlyPositive i =
  if i > 0 then
    Some i
  else
    None

let addTwoNumbers a b =
  let positiveA = onlyPositive a
  let positiveB = onlyPositive b
  // passt nicht, 2x int erwartet, aber 2x int option übergeben
  let sum = add positiveA positiveB

  // für zwei in F# bereits vordefiniert:
  let sum = Option.map2 add positiveA positiveB

  // aber was, wenn man mehr Parameter hat?
```

# APPLICATIVE



# APPLICATIVE

- Container mit "apply" Funktion (die bestimmten Regeln folgt): Applicative
- Bezeichnung in der FP-Welt: **Applicative Functor**
- ```
  apply: AF (a → b) → AF a → AF b
  ```
- Andere Bezeichnungen für "apply": ap, <*>
  ```

# FUNKTION MIT MEHREREN PARAMETERN

```fsharp
let sum a b c = a + b + c

let onlyPositive i =
    if i > 0 then Some i
    else None

let addNumbers a b c =
    let positiveA = onlyPositive a
    let positiveB = onlyPositive b
    let positiveC = onlyPositive c

    // sum ist vom Typ: (int → int → int → int)
    // jede Zeile füllt ein Argument mehr aus
    // (Partial Application dank Currying)
    let (sum' : (int → int → int) option) = Option.map sum positiveA
    let (sum'' : (int → int) option) = Option.apply sum' positiveB
    let (sum''' : (int) option) = Option.apply sum'' positiveC
```