

A photograph of a dark road curving away from the viewer into a vast landscape. The sky is filled with large, billowing clouds illuminated by a bright, setting sun, casting a warm orange glow over the scene. In the distance, dark silhouettes of hills or mountains are visible.

Funktionale Programmierung

@mobilgroma

@drechsler

Redheads Ltd.

 Herbstcampus



PATRICK DRECHSLER

- Software Entwickler / Architekt
- Beruflich: C#
- Interessen:
 - Software Crafting
 - Domain-Driven-Design
 - Funktionale Programmierung



MARTIN GROTZ

- Software Entwickler
- Beruflich: Angular mit TypeScript
- Interessen:
 - Funktionale Programmierung
 - Software-Architektur

LERNZIELE

- Grundlagen der funktionalen Programmierung
- Funktionale Programmierung mit C#
- Grundlagen von F#

FP 101

- **Functions as First Class Citizens**
- Immutability
- 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

IMPERATIV...

Wie mache ich etwas

```
var people = new List<Person>
{
    new Person { Age = 20, Income = 1000 },
    new Person { Age = 26, Income = 1100 },
    new Person { Age = 35, Income = 1300 }
};

var incomes = new List<int>();
foreach (var person in people)
{
    if (person.Age > 25)
    {
        incomes.Add(person.Income);
    }
}

var avg = incomes.Sum() / incomes.Count;
```

versus...

DEKLARATIV

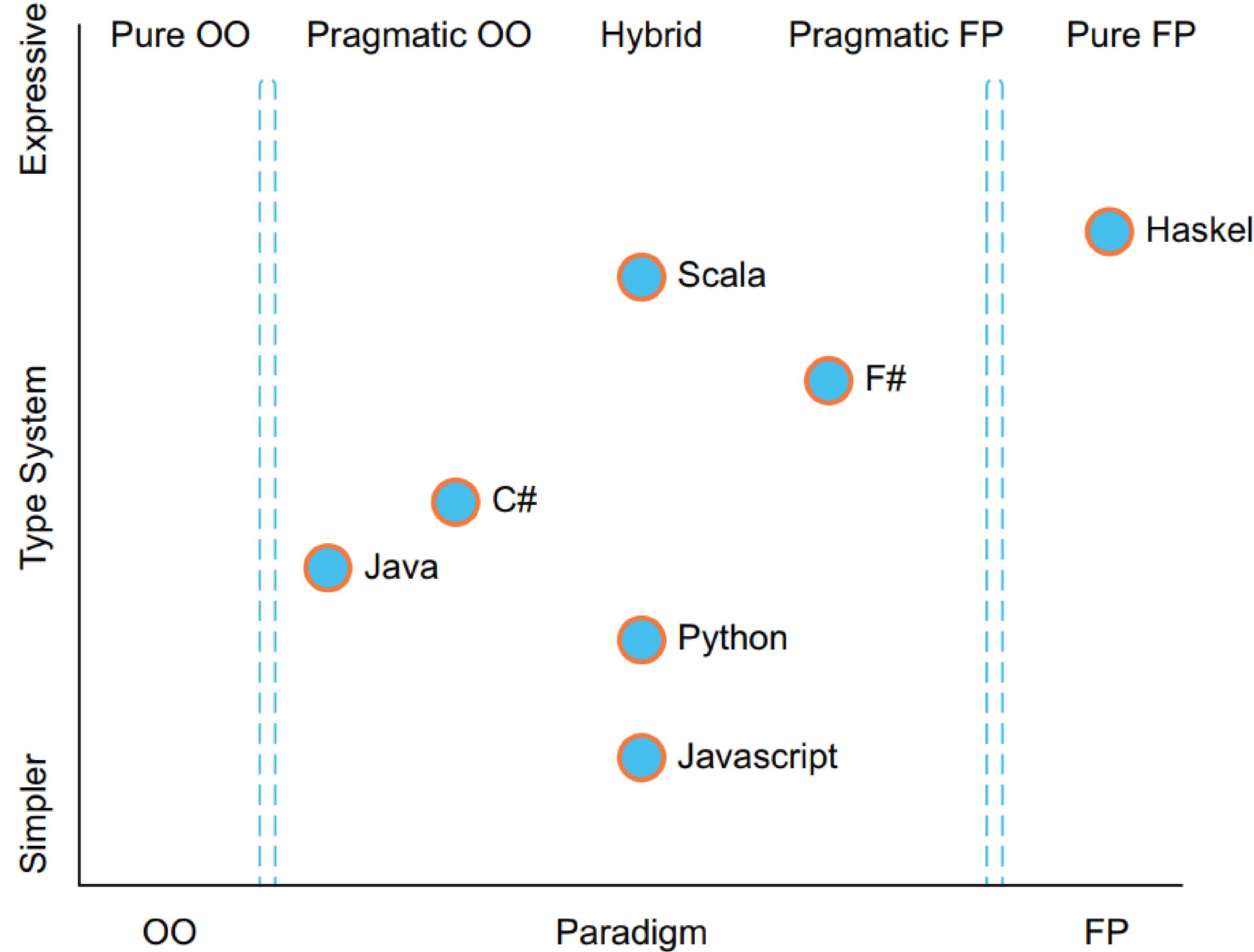
Was will ich erreichen?

Bsp: Filter / Map / Reduce

```
var people = new List<Person> {
    new Person { Age = 20, Income = 1000 },
    new Person { Age = 26, Income = 1100 },
    new Person { Age = 35, Income = 1300 }
}

var averageIncomeAbove25 = people
    .Where(p => p.Age > 25) // "Filter"
    .Select(p => p.Income) // "Map"
    .Average(); // "Reduce"
```

- aussagekräftiger
- weniger fehleranfällig



Schränken uns diese FP Paradigmen ein?

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

KLEINE FUNKTIONEN ZU GRÖSSEREN VERBINDEM

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

VORHANDENSEIN EINES
WERTS

ODER: NULL MUSS WEG.

```
// Enthält die Signatur die ganze Wahrheit?  
public string Stringify<T>(T data)  
{  
    return null;  
}
```

```
// Sind Magic Values eine gute Idee?  
public int Intify(string s)  
{  
    int result = -1;  
    int.TryParse(s, out result);  
    return result;  
}
```

```
public class Data
{
    public string Name;
}

public class Do
{
    public Data CreateData() => null;

    public string CreateAndUseData()
    {
        var data = CreateData();
        // kein null-Check → ist dem Compiler egal
        return data.Name;
    }
}
```

OPTION

```
// Pseudocode  
type Option<T> = Some<T> | None
```

- entweder ein Wert ist da - dann ist er in "Some" eingepackt
- oder es ist kein Wert da, dann gibt es ein leeres "None"
- alternative Bezeichnungen: Optional, Maybe

MIT OPTION

```
public Option<int> IntifyOption(string s)
{
    int result = -1;
    bool success = int.TryParse(s, out result);
    return success ? Some(result) : None;
}
```

WIE KOMME ICH AN EINEN EINGEPACKTEN WERT RAN?

*Pattern matching allows you to match
a value against some patterns to
select a branch of the code.*

```
public string Stringify<T>(Option<T> data)
{
    return data.Match(
        None: () => "",
        Some: (existingData) => existingData.ToString()
    );
}
```

VORTEILE

- Explizite Semantik: Wert ist da - oder eben nicht
- Auch für Nicht-Programmierer verständlich(er): "optional" vs. "nullable"
- Die Signatur von Match erzwingt eine Behandlung beider Fälle - nie wieder vergessene Null-Checks!
- Achtung: In C# bleibt das Problem, dass "Option" auch ein Objekt ist - und daher selbst null sein kann

LINQ - FÜR LISTEN (IENUMERABLE IN C#)

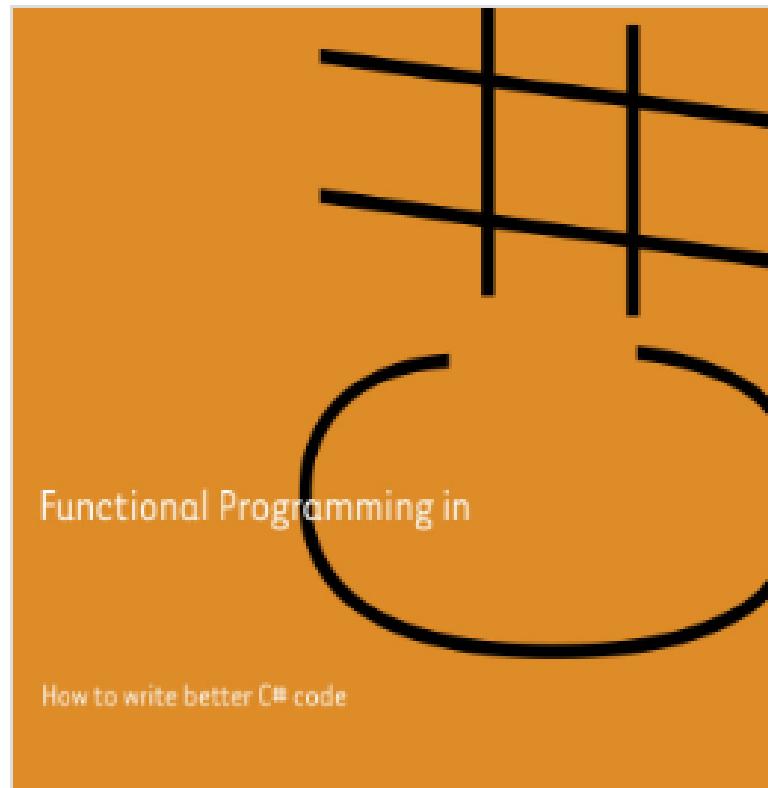
Allg.: Funktionen, die auf eine Liste angewendet werden

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 I Enumerable) anhand der Funktionen, die sie bereitstellen

LAYUMBA



Functional Programming in C# ❤

How to write better C# code

Enrico Buonanno

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

“

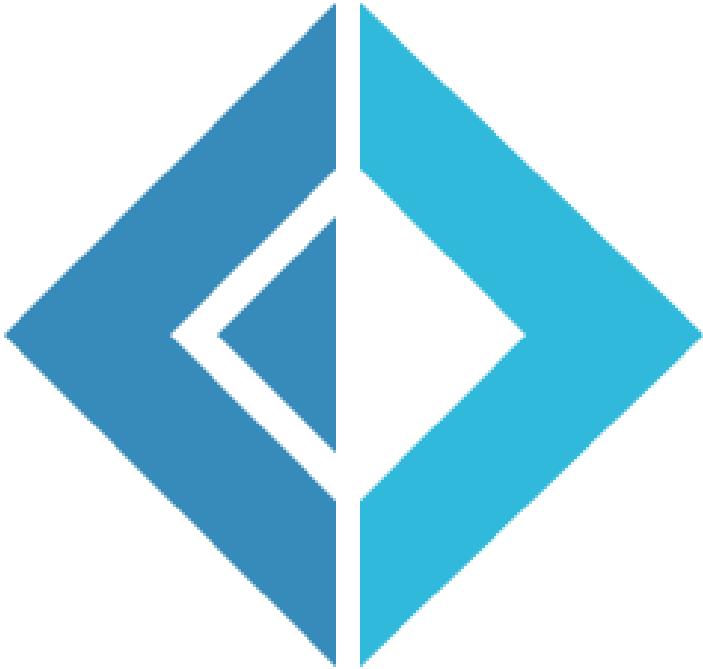
Functional programming can make your head
explode. This book stitches it back together.

Daniel Marbach, Particular Software

- NuGet Paket
- kann nicht alles
- Fokus: Didaktik (Ähnlichkeit mit F#, Haskell)
- "einfache" Variante von [language-ext](#)

EINFÜHRUNG IN

F#



F#

- Ursprünglich: Microsoft Forschungsprojekt
- Heute: Community-driven
- inspiriert von OCaml
- Multi-Paradigma
- Fokus auf funktionale Programmierung

F#

- erzwingt keine puren Funktionen, sondern erlaubt Seiteneffekte
- Statisch typisiert
- voll integriert ins .NET Ökosystem
- C# / VB.net Interop

BESONDERHEITEN

- Significant whitespace
- Reihenfolge der Definitionen in Datei wichtig
- Reihenfolge der Dateien im Projekt wichtig

IMMUTABILITY ALS DEFAULT

```
// Achtung: = ist hier keine Zuweisung, sondern heißt  
// "linke und rechte Seite sind gleich und bleiben es auch immer"  
let x = 3  
let add a b = a + b  
let m = if 3 > 0 then 7 else 42
```

```
// Mutability nur auf Wunsch - normalerweise unnötig  
let mutable y = 3  
y ← 42
```

TYP-INFERENZ

```
// Typen werden automatisch abgeleitet sofern möglich  
let double a = a * 2 // int → int
```

```
// Explizite Angaben möglich  
let doubleExplicit (a: int) : int = a * 2
```

CURRYING

Currying ist die Umwandlung einer Funktion mit mehreren Argumenten in eine Funktion mit einem Argument, die wiederum eine Funktion zurückgibt mit dem Rest der Argumente.

```
// int → int → int → int
// eigentlich: int → (int → (int → int))
let addThree a b c = a + b + c
```

PARTIAL APPLICATION

- Eine Funktion mit mehreren Parametern bekommt nur einen Teil ihrer Argumente übergeben - der Rest bleibt offen und kann später ausgefüllt werden

```
// Partial Application
let add a b = a + b // int → (int → (int))
let add2 = add 2 // (int → (int))
let six = add2 4 // (int)
let ten = add2 8 // (int)
```

PIPE-OPERATOR

```
// der letzte Parameter kann mit dem Ergebnis  
// der vorherigen Expression ausgefüllt werden  
let double a = a * 2  
4 ▷ double // ergibt 8  
4 ▷ double ▷ double // ergibt 16
```

DISCRIMINATED UNIONS

```
// Discriminated Unions ("Tagged Union", "Sum Type", "Choice Type")
type Vehicle = | Bike | Car | Bus

// Pattern Matching zur Behandlung der verschiedenen Fälle
let vehicle = Bike
match vehicle with
| Bike → "Ima ridin my bike"
| Car → "Driving along in my automobile"
| Bus → "SPEED"
```

DISCRIMINATED UNIONS MIT WERTEN

```
// auch mit unterschiedlichen(!) Daten an jedem Fall möglich
type Shape =
    | Circle of float
    | Rectangle of float * float
let c = Circle 42.42
match c with
| Circle radius → radius * radius * System.Math.PI
| Rectangle(width, height) → width * height
```

RECORD TYPES

```
// Record Type
type ShoppingCart = {
    products: Product list
    total: float
    createdAt: System.DateTime
}

// Typ muss nur angegeben werden wenn er nicht eindeutig ist
let shoppingCart = {
    products = []
    total = 42.42
    createdAt = System.DateTime.Now
}
```

RECORD TYPES

- Immutable by default
- Unmöglich einen ungültigen Record zu erzeugen
- Structural Equality

STRUCTURAL EQUALITY

```
// Structural Equality
type Thing = {content: string; id: int}
let thing1 = {content = "abc"; id = 15}
let thing2 = {content = "abc"; id = 15}
let equal = (thing1 = thing2) // true
```

- Record Types mit Structural Equality sind ideal, um sehr kompakt "Value Objects" ausdrücken zu können

STRUCTURAL EQUALITY VS. DDD AGGREGATES

- Möchte man die Standard-Equality nicht, ist es best practice, Equality und Comparison zu verbieten
- dann muss explizit auf eine Eigenschaft verglichen werden (z.B. die Id)

```
[<NoEquality; NoComparison>]
type NonEquatableNonComparable = {
    Id: int
}
```

VALUE OBJECTS

VALUE OBJECTS

Warum?

- Methoden sollten nicht lügen!
 - Null: NullPointerException, Null-Checks
 - Antipattern: Primitive Obsession

BEISPIELE

```
// :-(  
void Einzahlen(int wert, SomeEnum waehrung) { /* ... */ }  
  
// ;-)  
void Einzahlen(Geld geld) { /* ... */ }
```

```
class Kunde {  
    int Alter { get; set; } // :-(  
  
    // ist `i` das aktuelle Alter oder das Geburtsjahr??  
    bool IstVolljaehrig(int i) { /* ... */ }  
}  
  
class Kunde {  
    Alter Alter { get; set; } // ;-)  
  
    bool IstVolljaehrig(Alter alter) { /* ... */ }  
    bool IstVolljaehrig(Geburtsjahr geburtsjahr) { /* ... */ }  
}
```

← → ⌂ https://en.wikipedia.org/wiki/Value_object

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Value object

From Wikipedia, the free encyclopedia

In computer science, a **value object** is a small **object** that represents a *simple entity* whose equality is not based on identity: i.e. two value objects are *equal* when they *have the same value*, not necessarily being the *same object*.^{[1][2]}

Examples of value objects are objects representing an amount of money or a date range.

Being small, one can have multiple copies of the same value object that represent the same *entity*: it is often simpler to create a new object rather than rely on a single instance and use references to it.^[2]

Value objects should be **immutable**:^[3] this is required for the implicit contract that two value objects created *equal*, should remain equal. It is also useful for value objects to be immutable, as client code cannot put the value object in an invalid state or introduce buggy behaviour after instantiation.^[4]

Value objects are among the **building blocks** of DDD.

VALUE OBJECTS

- nur gültige Objekte erlaubt
- immutable
- equality by structure

NUR GÜLTIGE OBJEKTE

Es muss bei der Erstellung gewährleistet sein, dass
das Objekt gültig ist.

NUR GÜLTIGE OBJEKTE

Optionen:

- Konstruktor mit allen Parametern
- statische Hilfsmethode & privater Konstruktor

```
class Geld
{
    int Betrag { get; }
    Waehrung Waehrung { get; }

    Geld(int betrag, Waehrung waehrung) {
        if (!IsValid(betrag, Waehrung))
            throw new InvalidGeldException();

        Betrag = betrag;
        Waehrung = waehrung;
    }

    bool IsValid(int betrag, Waehrung waehrung)
        => betrag > 0 && waehrung != Waehrung.Undefined;
}
```

IMMUTABILITY

Damit ein C# Objekt unveränderlich wird, muss gewährleistet sein, dass es auch **nach Erstellung nicht verändert wird.**

- interne Werte dürfen ausschließlich vom Konstruktor verändert werden
- kein public oder private setter
- kein parameterloser Konstruktor

EQUALITY BY STRUCTURE

Zwei Objekte sind gleich, wenn sie die gleichen
Werte haben.

EXKURS: VERGLEICHBARKEIT

- Equality by reference
- Equality by id
- Equality by structure

EQUALITY BY STRUCTURE

Zwei Objekte sind gleich, wenn sie die gleichen Werte haben.

- Equals und GetHashCode überschreiben

```
override bool Equals(Geld other)
    => other.Betrag == this.Betrag &&
        other.Waehrung == this.Waehrung;

override int GetHashCode() { /* ... */ }
```

FP-KONZEPTE

PROBLEM: WERT IN CONTAINER, FUNKTION KANN NICHTS DAMIT ANFANGEN

```
// 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
```

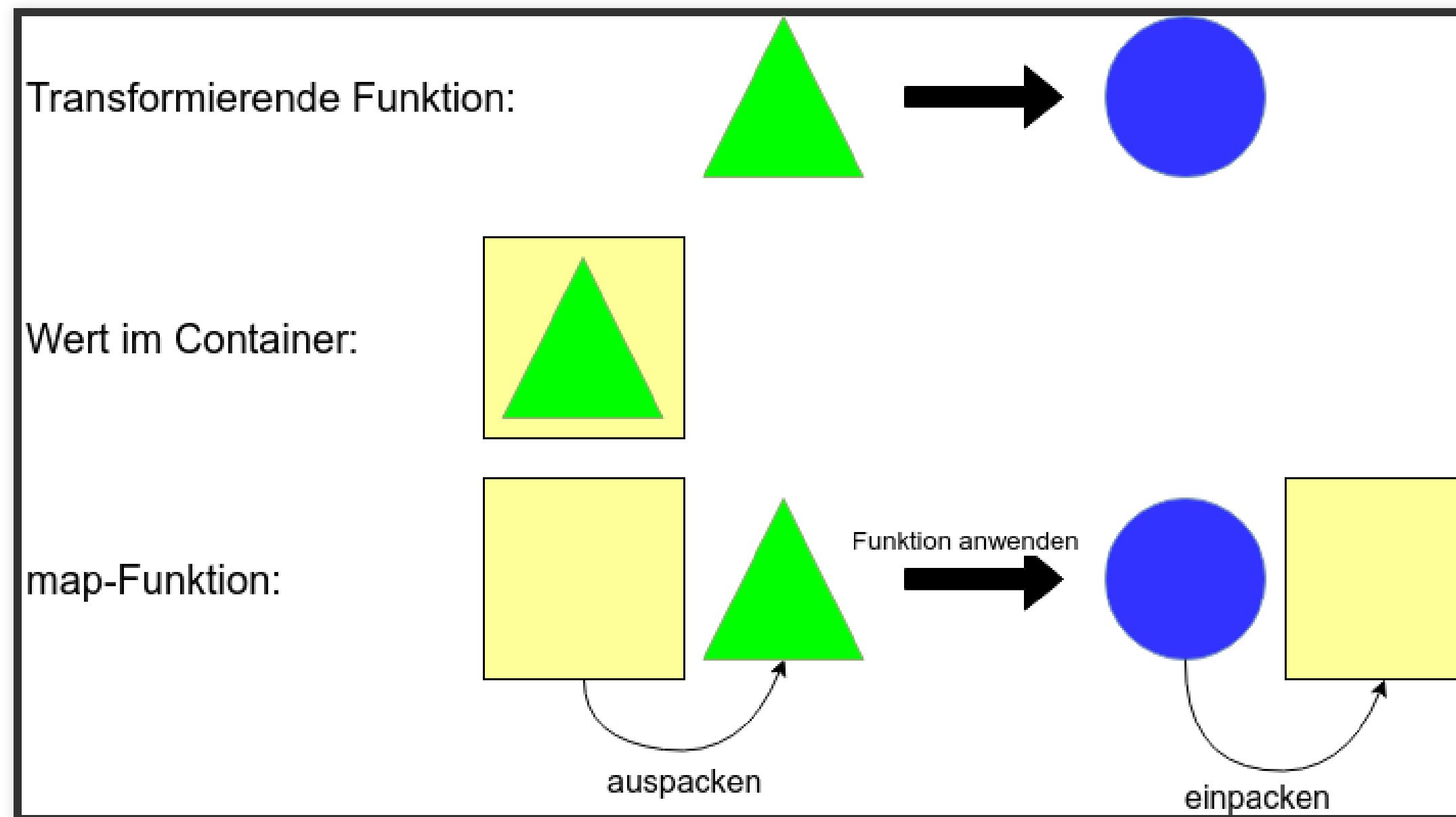
```
// C#
using LaYumba.Functional;
using static LaYumba.Functional.F;

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);
    }
}
```

FUNKTOR ('MAPPABLE')



FUNKTOR ("MAPPABLE")

- Container mit "map" Funktion (die bestimmten Regeln folgt): "Mappable"
- Bezeichnung in der FP-Welt: **Funktor**
- $\text{map}: (a \rightarrow b) \rightarrow F\ a \rightarrow F\ b$
- 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
```

PROBLEM: VERKETTUNG EINGEPACKTER WERTE

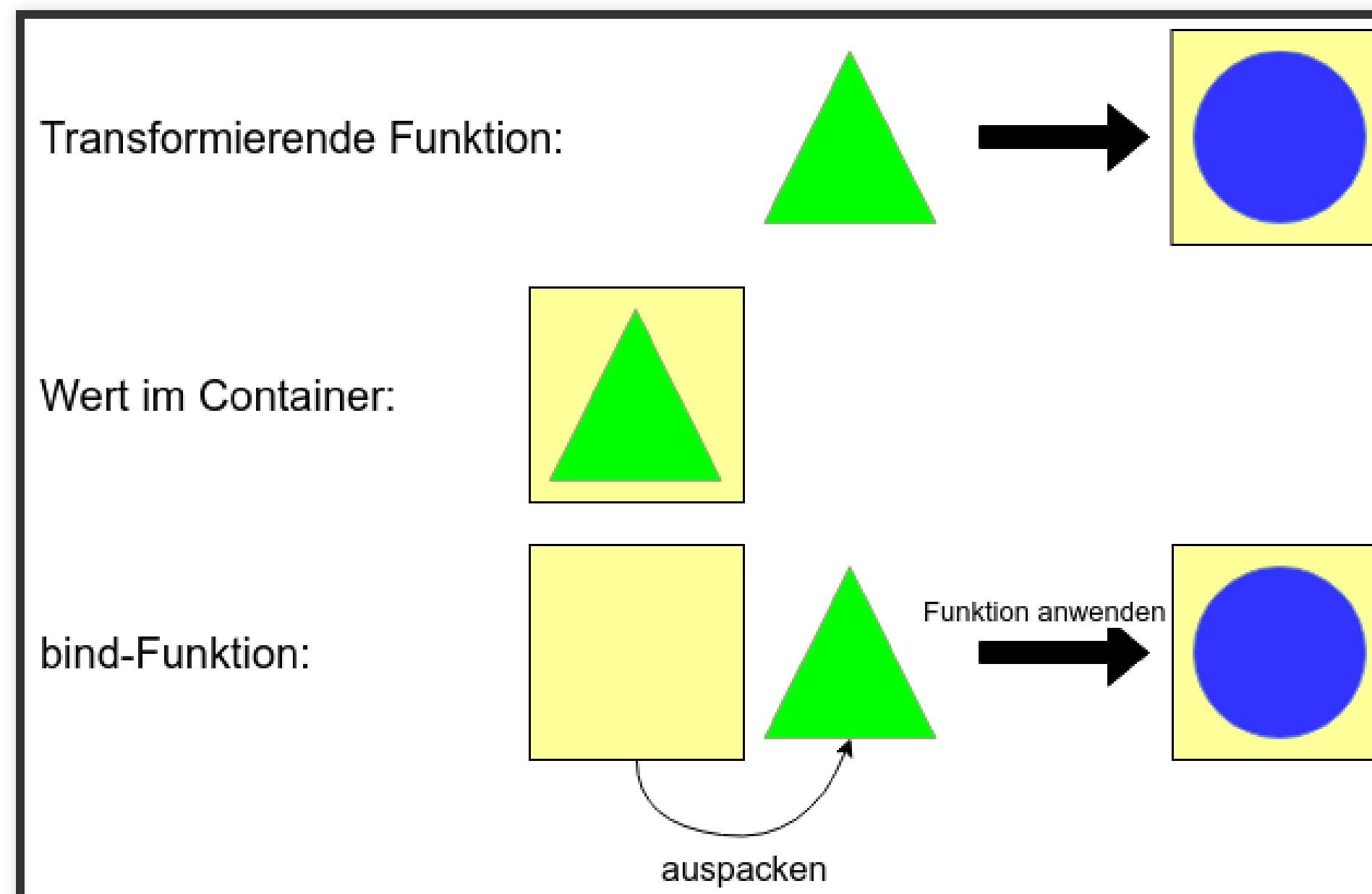
```
let storeInDatabase path content =
    try
        System.IO.File.WriteAllText(path, content)
        Some content
    with
        ex → None

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" bekommen
    let nonEmptyUpper = Option.map toUpper stored
```

MONADE ("CHAINABLE")



MONADE ("CHAINABLE")

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

VERKETTUNG

```
let storeInDatabase path content =
    try
        System.IO.File.WriteAllText(path, content)
        Some content
    with
        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
```

RAILWAY ORIENTED PROGRAMMING

Funktionale Programmierung wird oft als das
"Zusammenstöpseln" von Funktionen dargestellt...

Beispiel:

f1: Eingabe string, Ausgabe int
f2: Eingabe int, Ausgabe bool

FP: Komposition von f1 und f2
f3: Eingabe string, Ausgabe bool

// FP Syntax
f1: string → int
f2: int → bool
f3: string → bool

```
// Klassisch =====
int F1(string s) => int.TryParse(s, out var i) ? i : 0;
bool F2(int i) => i > 0;

// "verschachtelter" Aufruf
F2(F1("1")) // → true
F2(F1("0")) // → false

// "composition"
bool F3(string s) => F2(F1(s));
```

```
// Method Chaining =====
// mit C# extension methods
static int F1(this string s) => int.TryParse(s, out var i) ? i : 0;
static bool F2(this int i) => i > 0;

// Lesbarer (erst F1, dann F2)
"1".F1().F2() // →true
"0".F1().F2() // →false

// Lesbarer (erst F1, dann F2)
bool F3(string s) => s.F1().F2();
```

Problem: Keine standardisierte Strategie für
Fehlerbehandlung

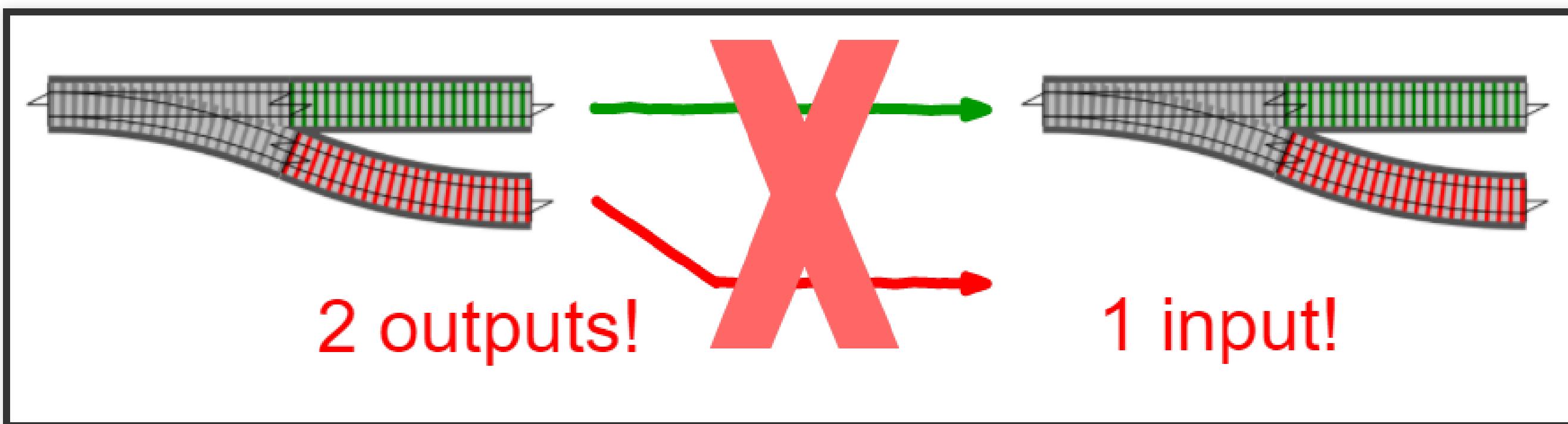
- Wenn wir davon ausgehen, dass Funktionen auch einen Fehlerfall haben, benötigen wir einen **neuen Datentyp**, der das abbilden kann

RESULT/EITHER

- kann entweder
 - das Ergebnis beinhalten, oder
 - einen Fehlerfall

- In Railway-Sprech bedeutet dass, dass man "2-gleisig" fährt:
- Jede **Funktion** bekommt eine Eingabe, und
 - hat "im Bauch" eine Weiche, die entscheidet ob
 - auf das Fehlertgleis oder
 - auf das Erfolgsgleis umgeschaltet wird.
- Die Wrapperklasse mit der **Funktion** ist das Entscheidende!

- In anderen Worten: die Funktionen haben aktuell 1 Eingabe (1 Gleis), und 2 Ausgaben (2 Gleise)



- Man benötigt also einen Mechanismus, der eine 2-gleisige Eingabe so umwandelt, dass eine Funktion, die eine 1-gleisige Eingabe erwartet, damit umgehen kann

WAS MUSS DIESER MECHANISMUS KÖNNEN?

- wenn die Eingabe fehlerhaft ist, muss die Funktion nichts tun, und kann den Fehler weiterreichen
- wenn die Eingabe nicht fehlerhaft ist, wird der Wert an die Funktion gegeben

```
bind: (string → Result int) → Result string → Result int
```

```
bind: (a → M b) → M a → M b
```

- FP-Jargon: eine Wrapper-Klasse, die bind bereitstellt, wird **Monade** genannt (sehr stark vereinfacht!).

- Either besteht aus 2 Teilen
 - Left
 - Right ("richtig" ...)
- Result besteht aus 2 Teilen
 - Failure
 - Success

- Option hat Some(T) und None

```
Option<string> IsValidOpt(string s) =>
    string.IsNullOrEmpty(s)
        ? None
        : Some(s);
```

- Either/Result ist ähnlich zu Option
- None wird durch Failure/Left ersetzt (frei wählbar, z.B. selbst definierter Error Typ).

```
Either<string, string> IsValidEither(string s)
    => string.IsNullOrEmpty(s)
        ? (Either<string, string>) Left("ups")
        : Right(s);
```

WEITERE KONZEPTE DER FUNKTIONALEN PROGRAMMIERUNG

PROBLEM: FUNKTION MIT MEHREREN EINGEPACKTEN PARAMETERN

```
let add a b = a + b

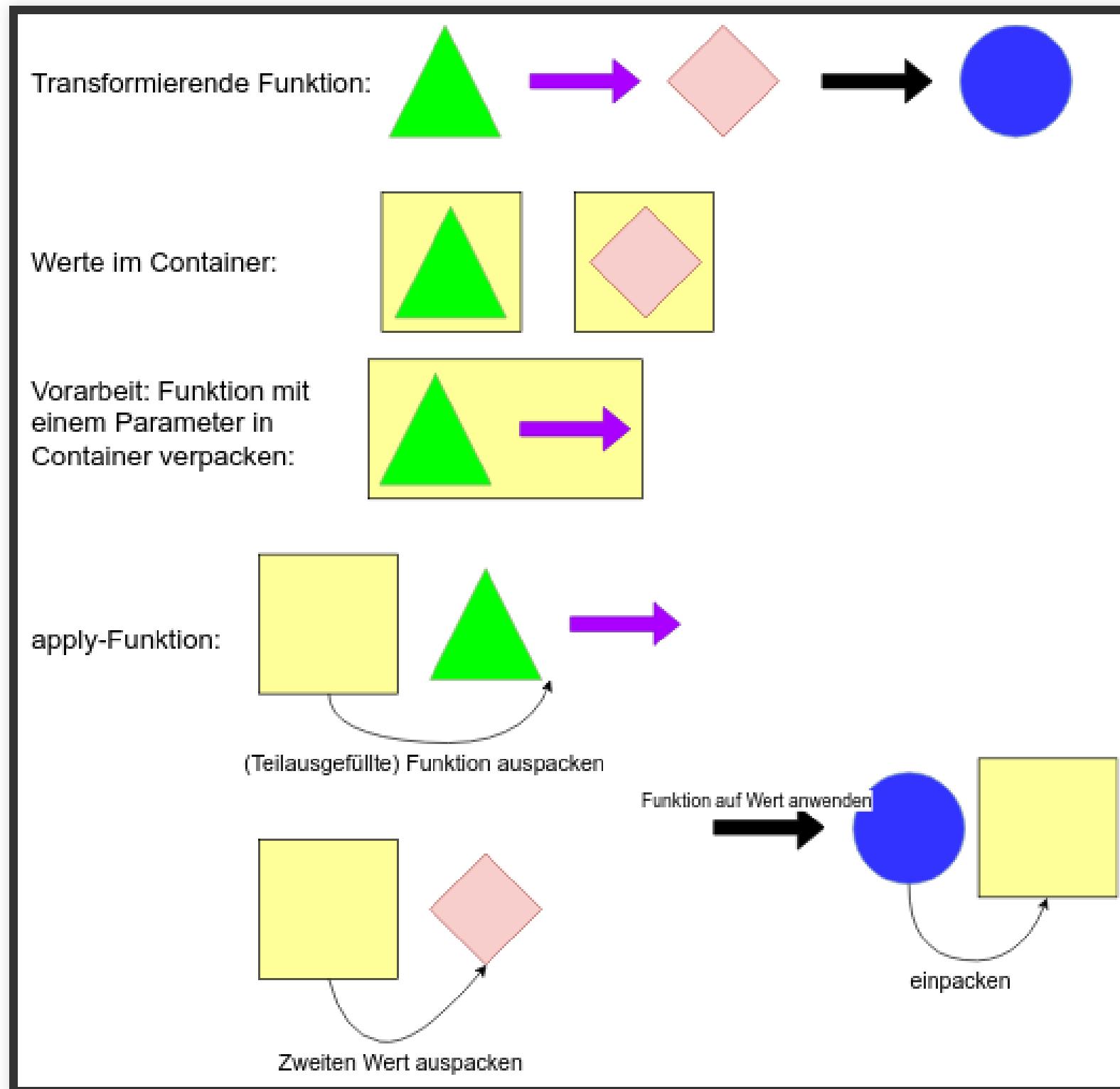
let onlyPositive i =
    if i > 0 then
        Some i
    else
        None

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

```
// F#
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
```

FUNKTION MIT MEHREREN PARAMETERN

```
// C#
Func<int, int, int, int> sum = (a, b, c) => a + b + c;

Func<int, Validation<int>> onlyPositive = i
    => i > 0
        ? Valid(i)
        : Error($"Number {i} is not positive");

Validation<int> AddNumbers(int a, int b, int c) {
    return Valid(sum)                                // returns int → int → int → int
        .Apply(onlyPositive(a))                      // returns int → int → int
        .Apply(onlyPositive(b))                      // returns int → int
        .Apply(onlyPositive(c));                     // returns int

AddNumbers(1, 2, 3);    // → Valid(6)
AddNumbers(-1, -2, -3); // → [
                        // Error("Number -1 is not positive"),
                        // Error("Number -2 is not positive"),
                        // Error("Number -3 is not positive")
                        // ]
```

INTERESSANTE VERANSTALTUNGEN

BUSCONF

<https://www.bus-conf.org/>

The Ultimate FP Unconference

1.Aug - 3.Aug 2019, Rückersbach (de)



LAMBDA LOUNGE NÜRNBERG

<https://www.meetup.com/de-DE/Lambda-Lounge-Funktionale-Programmierung-Nurnberg/>



Foto bearbeiten

Lambda Lounge - Functional Programming Nürnberg

Nürnberg, Deutschland

89 Mitglieder · Öffentliche Gruppe

Organisiert von Martin Grotz

Teilen: [f](#) [t](#) [in](#) [e](#)

Über uns [Events](#) [Mitglieder](#) [Fotos](#) [Diskussionen](#) [Gruppe verwalten](#) [Event erstellen](#)

[Liste](#) [Kalender](#)

SOFTWERKS KAMMER

<https://www.softwerkskammer.org/groups/nuernberg>

Softwerkskammer

Software Craftsmanship Communities in
Deutschland, Österreich und der Schweiz.

Über uns

Die Softwerkskammer hat sich 2011 gegründet, um den Austausch
Interessierter zum Thema Software Craftsmanship zu vereinfachen.

Hierüber organisieren die lokalen Communities (die
Softwerkskammern) ihre Treffen und andere Events wie CodeRetreats
oder OpenSpaces oder tauschen sich einfach nur via Mailingliste aus.

Was wir in den nächsten Wochen veranstalten, findest Du unter
Aktivitäten.

Wenn...

F# IN BESTEHENDES PROJEKT INTEGRIEREN

SCRIPTING ZUR AUTOMATION

- statisch typisierte Skripte
- .fsx Files
- kann C# mittlerweile auch

TESTS

- FsUnit für lesbarere Tests

FsUnit

FsChe

What is FsUnit?

FsUnit is a set of libraries that makes unit-testing with F# more enjoyable. It adds a special syntax to your favorite .NET testing framework. FsUnit currently supports NUnit, xUnit, MbUnit, and MsTest (VS11 only).

The goals of FsUnit are:

- to make unit-testing feel more at home in F#, i.e., more functional.
- to leverage existing test frameworks while at the same time adapting them to the F# language in new ways.

- FsCheck für Property Based Testing

FsCheck

FsCheck is a tool for testing .NET programs automatically. The programmer provides a specification of the program, in the form of properties which functions, methods or objects should satisfy, and FsCheck then tests that the properties hold in a large number of randomly generated cases. While writing the properties, you are actually writing a testable specification of your program. Specifications are expressed in F#, C# or VB, using combinators defined in the FsCheck library. FsCheck provides combinators to define properties, observe the distribution of test data, and define test data generators. When a property fails, FsCheck automatically displays a minimal counter example.



- Unquote für besseren "Callstack" bei fehlschlagenden Tests



- für Mocks: Object Expressions für direkte Interface-Implementierung, Foq, mountebank mit F# Binding

The screenshot shows the homepage of the mountebank project. At the top left is a cartoon illustration of a man in a top hat and green coat holding a bottle. To his right is the text "mountebank - over the wire test doubles". Below the header is a navigation bar with links for "home", "imposters", "logs", "config", and a feed icon. A search bar is also present. On the far right of the header is an orange "Fork me on GitHub" button. The main content area features a sidebar on the left with a "the apothecary" heading and a list of links: "getting started", "mental model", "client libraries", "install options", "command line", "security", "faqs", and "support". The main content area has a large "Welcome, friend" heading and a paragraph explaining what mountebank is: "mountebank is the first open source tool to provide cross-platform, multi-protocol test doubles over the wire. Simply point your application under test to mountebank instead of the real dependency, and test like you would with traditional stubs and mocks." It also states that "mountebank is the most capable open source service virtualization tool in existence, and will cure what ails you, guaranteed."

- Browser-Fernsteuerung mit canopy

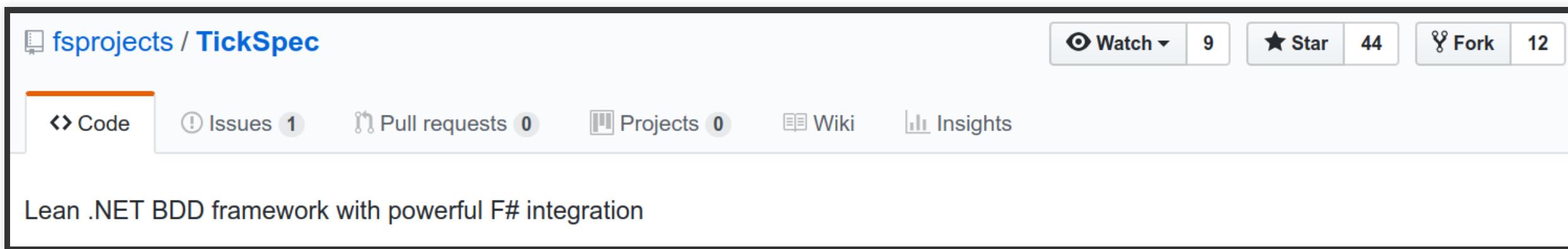
canopy - f#rictionless web testing

The canopy library can be [Installed from NuGet](#):

```
PM> Install-Package canopy
```



- BDD-Tests mit TickSpec



BUILDS

WEITERE NUTZUNGSMÖGLICHKEITEN

- Mobile Apps mit Fabulous



The image shows the official website for Fabulous. It features a dark teal header with the word "Fabulous" in large white letters and "F# Functional App Development" below it. A "View on GitHub" button is located in the center of the header. The main content area is white and contains the text "F# Functional App Development, using Xamarin.Forms" in green, followed by the tagline "Never write a ViewModel class again! Conquer the world with clean dynamic UIs!" in smaller gray text.

Fabulous

F# Functional App Development

View on GitHub

F# Functional App Development, using
Xamarin.Forms

Never write a ViewModel class again! Conquer the world with clean dynamic UIs!

- Full-Stack-Webanwendungen mit dem SAFE-Stack



DANKE

RESSOURCEN

- FP
 - Blog: [Mark Seeman](#)
 - Book: [Domain Modeling Made Functional](#)
- C#
 - Book: [Functional Programming in C#](#)
 - Blog: [Vladimir Khorikov](#)
- F#
 - [F# Homepage](#)
 - [F# for Fun and Profit](#)
 - [Railway Oriented Programming](#)

KONTAKT

PATRICK DRECHSLER

- patrick.drechsler@redheads.de
- Github: <https://github.com/draptik>
- Twitter: @drechsler
- Blog: <http://draptik.github.io>

MARTIN GROTZ

- martin.grotz@redheads.de
- Github: <https://github.com/groma84>
- Twitter: @mobilgroma
- Blog: <https://elm.findest-ich-super.de>

BILDQUELLEN

- [road-3478977_1920.jpg](#)
- [mob-programming-setup.png](TODO: Maaret Pyhäjärvi and Llewellyn Falco; The Mob Programming Guide Book)
- [wikipedia-value-objects.png](#) Screenshot 2019-03-30
- [fp-languages-overview.png](Get Programming with F#, Isaac Abraham, fig. 1, page 4)
- [addressbook1.png](#)

FEEDBACK

