# BACHELORARBEIT

zur Erlangung des akademischen Grades
„Bachelor of Science in Engineering“
im Studiengang Informatik

# Design Patterns in funktionalen Sprachen

Ausgeführt von: Constantin Matheis

Personenkennzeichen: 1510257020

BegutachterIn: DI Harald Steinlechner, BSc.

Wien, den 27. Februar 2018

FH University of Applied Sciences
TECHNIKUM WIEN

# Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Wien, 27. Februar 2018 Unterschrift

# Kurzfassung

Design Patterns haben in der objektorientierten Programmierung große Aufmerksamkeit erhalten, weniger dafür in der funktionalen. Aus diesem Grund beantwortet diese Arbeit die Frage, ob und wie sich Design Patterns in funktionaler Programmierung äußern. Dabei wurden drei Muster ausfindig gemacht (Lazy Evaluation, Memoisation, Railway Oriented Programming), die unter diesen Begriff fallen. Anschließend wurde überprüft, ob Design Patterns aus der objektorientierten Programmierung auch in der funktionalen angewendet werden können. Dafür wurden beispielhaft drei objektorientierte Patterns ausgewählt (Strategy, Command, Composite) und in funktionalen Code übersetzt. Dadurch wurde festgestellt, dass die Patterns Strategy und Command durch Higher-Order Functions abgelöst werden, und Composite nur noch in Grundzügen vorhanden bleibt.

**Schlagworte:** Entwurfsmuster, Funktionale Programmierung

# Abstract

While design patterns have become established in object-oriented programming, they are not particularly known for their application in functional programming. This work, therefore, sets out to answer the question whether design patterns are present in functional programming and how they manifest themselves. As a result, three patterns were found (Lazy Evaluation, Memoization, Railway Oriented Programming) that fall under this concept. Furthermore, it was examined whether object-oriented design patterns can be applied in functional programming. This is exemplified by three design patterns (Strategy, Command, Composite), which were translated to functional code. It was found that the Strategy and Command patterns were completely replaced by higher order functions and that only the basic structure of the Composite pattern remained.

**Keywords:** Design Pattern, Functional programming

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Definition Design Pattern

Design Patterns sind allgemeine Lösungen für immer wiederkehrende Probleme in der Softwareentwicklung. Sie stellen dabei keine Anweisungen dar, die direkt in Code übersetzt werden können, sondern sind vielmehr Schablonen oder Templates, die in unterschiedlichen Situationen angewendet werden [25]. Was nun wirklich als Design Pattern klassifiziert wird und was nicht, ist Interpretationssache. Gamma u. a. [8] nennen in ihrem Standardwerk zu objektorientierten Design Patterns[1] jene Strukturen, die definieren, wie Objekte und Klassen miteinander kommunizieren, um ein allgemeines Design-Problem zu lösen. Daneben existieren noch andere Auffassungen von Patterns, wie zum Beispiel Architektur-Patterns, die einen weiter gefassten Scope haben [27, p. 19], Idiome, die low-level Patterns beschreiben und programmiersprachenspezifisch sind [6, p. 14], Integrations-Patterns, die einzelne Komponenten miteinander verbinden [11], u.v.m. In dieser Arbeit wird der Fokus auf Patterns gesetzt, die einen ähnlichen Scope haben, wie die der GoF[2], d.h. es geht um kleine, zusammenhängende Elemente, die programmiersprachenunabhängig sind und daher nicht mehr unter den Begriff Idiom fallen.

## 1.2 Anwendbarkeit des Konzepts in funktionaler Programmierung

Design Patterns haben besonders in der objektorientierten Programmierung (OOP) Aufmerksamkeit erhalten, zum Teil wegen dem Werk *Design Patterns: Elements of Reusable Object-Oriented Software* [8], das allerdings auch kritische Stimmen hervorgerufen hat. Paul Graham [9], zum Beispiel, sieht das Auftauchen von Patterns in seinem Code als problematisch an und meint, es sei der Beweis für unzureichend mächtige Abstraktionen. Peter Norvig [17] kritisiert die Fähigkeiten von OOP generell und meint, 16 der 23 GoF-Patterns seien entweder gar nicht mehr sichtbar oder zumindest simpler in dynamischen Sprachen[3]. In der funktionalen Programmierung (FP) gibt es bis dato zwar kein Standardwerk zu Design Patterns, jedoch lassen sich zahlreiche, sich wiederholende Strukturen erkennen, die als Design Patterns klassifiert werden

[1] *Design Patterns: Elements of Reusable Object-Oriented Software* [8].

[2] Gang of Four, Bezeichnung für die vier Autoren von *Design Patterns: Elements of Reusable Object-Oriented Software* [8].

[3] Eine dynamische Sprache ist eine Programmiersprache, in der Operationen zur Runtime gemacht werden können, die sonst zur Compiletime gemacht werden [15].

können [40], und in dieser Arbeit mithilfe von F# wiedergegeben werden. Die Ziele der Arbeit sind, diese Strukturen im ersten Teil (Kapitel 3) zu identifizieren und zu beschreiben, und im zweiten Teil (Kapitel 4) zu analysieren, inwiefern OOP Patterns in FP anwendbar sind.

# 2 Begriffserklärungen

## 2.1 Referentielle Transparenz

Referentielle Transparenz bezeichnet die Eigenschaft eines Ausdrucks, mit seinem Resultat ausgetauscht werden zu können, ohne dabei das Verhalten des Programms zu verändern [4, p. 637].

## 2.2 Seiteneffekte

Modifiziert eine Funktion Variablen, Datenstrukturen, kommuniziert mit der Außenwelt oder wirft eine Exception, dann wird von Seiteneffekten gesprochen. In der funktionalen Programmierung sollten Seiteneffekte vermieden werden, da es sonst zu unvorhersehbaren Resultaten kommen kann [18].

## 2.3 Reine Funktionen

Reine Funktionen sind *referentiell transparent* und haben keine *Seiteneffekte*. Solche Funktionen berechnen ihr Resultat ausschließlich aus den überreichten Argumenten, verändern nicht den State des Objektes, das die Funktion enthält und führen auch keine Input/Output Operationen durch. Beispiele dafür sind mathematische Funktionen, wie die Multiplikation oder Addition. Das Gegenteil davon sind Funktionen, die unrein genannt werden, weil sie zum Beispiel in einer Datenbank lesen oder schreiben [4, p. 637-638].

## 2.4 Mutable/Immutable

Objekte werden mutable genannt, wenn diese nach ihrer Erzeugung verändert werden können. Immutable Objekte können im Gegensatz dazu nicht mehr verändert werden, wodurch sie sich zum Beispiel sehr gut für Applikationen mit mehreren Threads eignen [1].

## 2.5 Function Composition

Mittels Function Composition kann aus mehreren Funktionen eine einzelne, neue Funktion erstellt werden. Eine Funktion, die von Typ **T1** nach Typ **T2** geht, und eine weitere, die von **T2** nach **T3** geht, können zu einer einzelnen Funktion zusammengefügt werden, die von **T1** nach **T3** geht. Dabei wird in F# der Function Composition Operator >> verwendet [35, 32]. Ein simples Beispiel dazu:

```
1 let add1 x = x + 1
2 let times2 x = x * 2
3 let add1Times2 = add1 >> times2
4 // Verwendung
5 add1Times2 3
```

Quellcode 1: Function Composition, Quelle: [35]

## 2.6 First-Class/Higher-Order Functions

Unterstützt eine Programmiersprache First-Class Functions, dann werden Funktionen als Wert wie jeder andere behandelt und können als Argument einer anderen Funktion übergeben werden. Dadurch haben Funktionen auch einen Typen und können an jeder Stelle verwendet werden, an der auch ein Integer oder String, zum Beispiel, stehen könnte [19, p. 41].
Higher-Order Functions sind Funktionen, die entweder eine Funktion als Parameter nehmen, oder eine Funktion als Rückgabewert haben [19, p. 41].

# 3 FP Design Patterns

In diesem Kapitel werden drei Muster beschrieben (Lazy Evaluation, Memoisation und Railway Oriented Programming), die in FP Code wahrnehmbar sind und daher als Design Patterns klassifiziert werden können. Diese Liste soll keine vollständige Sammlung aller Patterns darstellen, sondern exemplarisch zur Argumentation für die Existenz von Patterns in FP herangezogen werden.

## 3.1 Lazy Evaluation

### 3.1.1 Motivation

In Sprachen, die Seiteneffekte erlauben (wie F#), gibt es eine fix vorgegebene Auswertungsreihenfolge. Wäre das nicht der Fall, könnte nicht vorhergesagt werden, in welcher Reihenfolge die Seiteneffekte auftreten würden [19, p. 301]. Hat eine Funktion einen Ausdruck als Parameter, wird dieser sofort ausgewertet, noch bevor die Funktion aufgerufen wird, ungeachtet dessen, ob das Ergebnis des Ausdrucks auch tatsächlich benötigt wird. Diese Auswertungsstrategie wird Eager Evaluation genannt [36].

```
1 let test b t f = if b then t else f
2 test true (printfn "true") (printfn "false")
3
4 // Output
5 true
6 false
```

Quellcode 2: Eager Evaluation, Quelle: modifiziert übernommen aus [36]

Obwohl die Funktion `test` nicht in den “else” Branch kommt, wird “false” ausgegeben, weil `(printfn "false")` sofort ausgewertet wird, bevor `test` aufgerufen wird.
Der Vorteil ist, dass Eager Evaluation leicht nachvollziehbar ist, jedoch zu dem Preis, dass unter Umständen die Performance darunter leidet. Das Gegenstück zu Eager Evaluation ist Lazy Evaluation, wobei die Auswertungsreihenfolge von Ausdrücken nicht vorgegeben ist und diese erst dann ausgewertet werden, wenn sie tatsächlich benötigt werden. Die funktionale Programmiersprache Haskell, zum Beispiel, verfolgt diese Strategie, was möglich ist, weil sie keine Seiteneffekte zulässt. Der Aufruf von `test` aus Quellcode 2 würde in Haskell daher nur “true” ausgeben [36]. Aber auch in Programmiersprachen, die unrein sind, können die Vorteile von Lazy Evaluation genutzt werden. Neben Lazy Evaluation ist noch Lazy Initialization zu erwähnen, womit die Instanzierung von Objekten bis zum ersten Gebrauch aufgeschoben wird [24]. Eine Anwendung davon findet sich beim Singleton Pattern, siehe [26].

### 3.1.2 Implementierung

#### OOP

Lazy Evaluation ist ein Konzept, das nicht nur in FP, sondern auch in OOP Verwendung findet. In diesem Abschnitt wird die Funktionsweise davon anhand von C# erklärt. Seit C# 4.0 ist die Klasse `Lazy<T>` verfügbar, die eine Art Wrapper für ein Objekt von Typ `Func<T>` definiert [19, p. 306-307]. Sie repräsentiert eine Berechnung, die erst dann ausgewertet wird, wenn das Ergebnis gebraucht wird. Zur Demonstration ist nachfolgend eine vereinfachte Definition dieser Klasse zu sehen, die das grundlegende Prinzip veranschaulicht:

```
1 public class Lazy<T> {
2     readonly Func<T> func;
3     bool evaluated = false;
4     T value;
5
6     public Lazy(Func<T> func) {
7         this.func = func;
8     }
9
10    public T Value {
11        get {
12            if (!evaluated) {
13                value = func();
14                evaluated = true;
15            }
16            return value;
17        }
18    }
19 }
20
21 public class Lazy {
22    public static Lazy<T> Create<T>(Func < T > func) {
23        return new Lazy<T>(func);
24    }
25 }
```

Quellcode 3: Lazy<T> vereinfacht, Quelle: modifiziert übernommen aus [19, p. 306-307]

Der Konstruktor nimmt eine Funktion entgegen, die keine Argumente hat, und speichert diese im `readonly` Feld `func`. Es wird das Flag `evaluated` verwendet, um bestimmen zu können, ob `func` schon einmal ausgeführt wurde. Im Getter `Value` wird zuerst überprüft, ob `func` schon einmal ausgewertet wurde. Ist das der Fall, wird nur noch der zwischengespeicherte Wert zurückgegeben. Ansonsten wird `func` aufgerufen und der Rückgabewert im Feld `value` gespeichert [19, p. 307]. Um die Typinferenz von C# besser nutzen zu können, wird in den Zeilen 21-25 noch eine nicht-generische Klasse mit der statischen Methode `Create<T>` definiert, die nicht Teil der .NET Standard-Bibliothek ist [2]. Nun zu der Verwendung:

```
1 Func<int, bool> foo = (x) => {
2     Console.WriteLine("foo({0})", x);
3     return x <= 10;
4 };
5
6 var lazy = Lazy.Create(() => foo(10));
7 Console.WriteLine(lazy.Value); // Gibt 'foo(10)' und 'True' aus
8 Console.WriteLine(lazy.Value); // Gibt nur 'True' aus
```

Quellcode 4: Lazy<T> Verwendung, Quelle: modifiziert übernommen aus [19, p. 307]

Beim Erstellen von `lazy` wird die Funktion `foo` in einer Lambda übergeben, wobei diese noch nicht aufgerufen wird. Der erste Aufruf von `Value` ruft `foo` auf und jeder weitere Aufruf gibt nur mehr das zwischengespeicherte Ergebnis zurück.

### FP

In F# gibt es dafür ein Feature namens lazy values, das ähnlich angewendet wird, wie die Klasse `Lazy<T>` in C# [19, p. 304]. Da auch beide Sprachen beim Kompilieren in die gleiche Common Intermediate Language (CIL)[1] übersetzt werden, funktioniert Lazy Evaluation in F# und C# im Hintergrund genau gleich [14, p. 153]. Wird F# Code, der lazy values verwendet, kompiliert und anschließend wieder in C# Code dekompiliert[2], ist auch zu sehen, dass die Klasse `Lazy<T>` verwendet wird.
Wird ein Ausdruck mit dem Keyword `lazy` markiert, wird er nicht sofort ausgewertet, sondern in ein lazy value gewrappt [19, p. 304]. Dieses Feature wird mit folgender Syntax verwendet:

```
let identifier = lazy ( expression )
```

Quellcode 5: lazy value Syntax, Quelle: [31]

Quellcode 4 sieht in F# in der Interactive-Shell folgendermaßen aus:

```
> let foo(n) =
    printfn "foo(%d)" n
    n <= 10;;
val foo : int -> bool

> let n = lazy foo(10);;
val n : Lazy<bool> = Value is not created.

> n.Value;;
foo(10)
val it : bool = true

> n.Value;;
val it : bool = true
```

Quellcode 6: lazy value Verwendung (F# Interactive), Quelle: [19, p. 304]

[1] Plattformunabhängige, objektorientierte, Assemblersprache, in die alle high-level .NET Sprachen übersetzt werden [23].
[2] mit einem Tool, wie z.B.: ILSpy: https://github.com/icsharpcode/ILSpy.

Im Output in Zeile 7 ist zu sehen, dass `foo` zu diesem Zeitpunkt noch nicht aufgerufen wurde und `n` von Typ `Lazy<bool>` ist. So wie bei der C# Variante wird mittels `Value` der Wert des Ausdrucks berechnet, der im lazy value steckt, was in diesem Fall wieder die Funktion `foo` ist. Auch hier ist zu sehen, dass nur beim ersten Aufruf von `Value` die Funktion tatsächlich ausgeführt wird. Daraus lässt sich auch ableiten, dass `Lazy<T>` mutable sein muss [19, p. 304-305].

## 3.2 Memoisation

### 3.2.1 Motivation

Im letzten Kapitel wurde erläutert, wie Lazy Evaluation es möglich macht, Ausdrücke erst bei Bedarf auszuwerten und deren Resultate danach zu cachen, wobei allerdings immer nur ein einzelner Wert zwischengespeichert wird. Im Gegensatz dazu macht Memoisation (v. lat. memorandum: das zu Erinnernde [3]) es möglich, viele, bereits berechnete Resultate teurer Funktionen samt dazugehörigen Argumenten zwischenzuspeichern und bei Bedarf wieder abzurufen [5]. Memoisation ist eine Form von Dynamic Programming [7, p. 387], bei dem ein Problem in Sub-Probleme zerlegt wird, welche nur ein einziges Mal berechnet werden, und deren Resultate danach zwischengespeichert werden [7, p. 359]. Für eine ausführliche Erklärung von Dynamic Programming und dessen Ausprägungen siehe [7, p. 359]. Memoisation kann nur dann angewendet werden, wenn mit *reinen Funktionen* gearbeitet wird [5], welche sich durch *referentielle Transparenz* und die Abwesenheit von *Seiteneffekten* auszeichnen [4, p. 637]. Das liegt daran, dass beim Memoisieren kein State, sondern nur die Argumente der Funktion und deren Resultat gecachet werden [5].

### 3.2.2 Implementierung

```
1 let memoize f =
2     let cache = System.Collections.Generic.Dictionary()
3     fun x ->
4         match cache.TryGetValue(x) with
5         | true,res -> printfn "returned memoized";res
6         | _ -> let res = f x
7                cache.Add(x,res)
8                printfn "memoized, then returned"
9                res
```

Quellcode 7: Memoisation Implementierung, Quelle: modifiziert übernommen aus [5]

Der F# Compiler leitet dabei folgenden Typen ab:

```
memoize : f:('a -> 'b) -> ('a -> 'b) when 'a : equality
```

Quellcode 8: Typsignatur von memoize

Da F# Funktionen als Werte handhabt, wird `memoize` schon bei seiner Erzeugung ausgewertet, wobei einmalig die Dictionary-Instanz `cache` erzeugt wird. Das bedeutet, dass Aufrufe von `memoize` nicht neue `cache`-Instanzen erzeugen. `memoize` nimmt als Argument eine Funktion `f`, die memoisiert werden soll, und gibt wieder eine Funktion zurück, die auf die anfangs erzeugte `cache`-Instanz zugreift. Bei einem Aufruf der zurückgegebenen Funktion wird zuerst überprüft, ob das Argument von `f` als Schlüssel im Dictionary existiert. Wenn ja, wird der dazugehörige Wert direkt zurückgegeben. Andernfalls wird `f x` aufgerufen und der Rückgabewert danach in `cache` abgespeichert.

### 3.2.3 Verwendung

```
let rec fib_memo = memoize (fun n ->
    if n < 1 then 1 else
    (fib_memo (n - 1) + fib_memo(n - 2)))
```

Quellcode 9: Verwendung von Memoisation, Quelle: modifiziert übernommen aus [20]

Die Funktion `memoize` wird hier anhand der rekursiven Berechnung der Fibonacci-Zahlen getestet. `fib_memo 4` liefert folgenden Output:

```
memoized, then returned
memoized, then returned
memoized, then returned
returned memoized
memoized, then returned
returned memoized
memoized, then returned
returned memoized
memoized, then returned
```

Quellcode 10: Konsolen-Output

Bereits berechnete Resultate werden zwischengespeichert und beim nächsten Aufruf mit dem gleichen Argument nicht nochmals berechnet, sondern direkt aus `cache` geholt. Dabei wird `returned memoized` in den Output geschrieben.

## 3.3 Railway Oriented Programming

### 3.3.1 Motivation

Beim Erstellen eines Programms müssen oft eine Reihe von Funktionen hintereinander ausgeführt werden, wobei jede nur dann aufgerufen werden soll, wenn die vorhergehende fehlerfrei ausgeführt wurde. Abbildung 1 repräsentiert einen Workflow mit möglichen Fehlern an jeder Stelle [38]. Die Implementierung davon ist in Quellcode 11 angeführt.

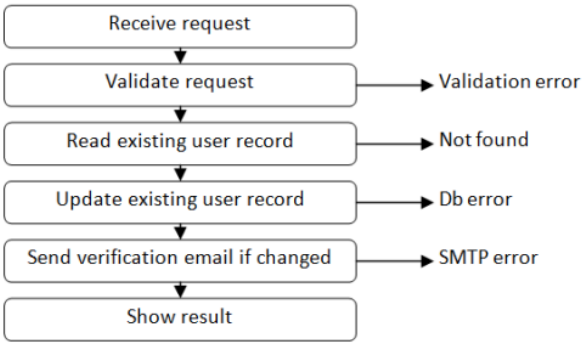


Abbildung 1: Beispiel eines Workflows, Quelle: [38]

```
let example request =
    match validate request with
    | None -> None
    | Some(a) ->
        match readUserRecord a with
        | None -> None
        | Some(b) ->
            match updateUserRecord b with
            | None -> None
            | Some(c) ->
                match sendVerificationMail c with
                | None -> None
                | Some(d) ->
                    showResult d
```

Quellcode 11: Workflow Implementierung

In Quellcode 11 ist die sogenannte Pyramid of Doom zu erkennen (siehe Abbildung 2 für eine Illustration der Pyramiden-Analogie), die bei Code auftritt, der viele verschachtelte Einrückungen enthält. Dieses Muster ist oft bei der Behandlung von Fehlern oder Callbacks zu beobachten, und macht den Code unübersichtlich [10, 40]. Um die Pyramid of Doom zu vermeiden, wird in der funktionalen Programmierung Railway Oriented Programming eingesetzt. Dadurch lassen

sich diese verschachtelten Konstrukte in linearen Code übersetzen[39].

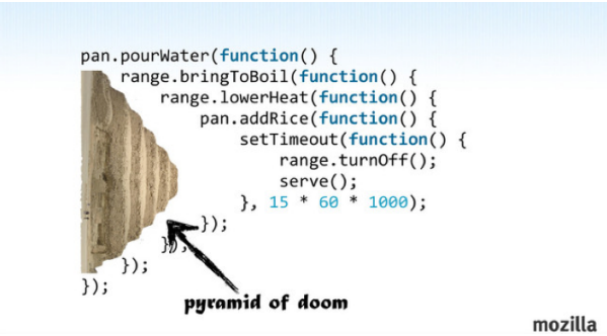


Abbildung 2: Pyramid of Doom, Quelle: [22]

### 3.3.2 Implementierung

#### Design eines einzelnen Schritts

In Quellcode 11 wurde der `Option` Typ verwendet, um Success oder Failure einer Funktion auszudrücken. F# bietet noch einen anderen Typen an, der dafür besser geeignet ist und im weiteren Verlauf verwendet wird: `Result<'T,'TError>` [16]. Dieser ist wie folgt definiert:

```
1 [<StructuralEquality; StructuralComparison>]
2 [<CompiledName("FSharpResult`2")>]
3 [<Struct>]
4 type Result<'T,'TError> =
5 /// Represents an OK or a Successful result. The code succeeded
    with a value of 'T.
6 | Ok of ResultValue:'T
7 /// Represents an Error or a Failure. The code failed with a value
    of 'TError representing what went wrong.
8 | Error of ErrorValue:'TError
```

Quellcode 12: Result<'T,'TError>, Quelle: [33]

Dieser Typ ist `Option` sehr ähnlich. Der entscheidende Unterschied liegt im Fall von `Error`, bei dem noch Informationen über die Gründe des Scheiterns mitgegeben werden können, was bei `None` nicht möglich war. Die einzelnen Schritte des Workflows werden mit Funktionen realisiert, die alle `Result` zurückgeben, anhand dessen entschieden werden soll, ob die nächste Funktion aufgerufen wird [39]. Eine Visualisierung der Implementierung vom Schritt Validate request aus Abbildung 1 ist in Abbildung 3 zu sehen. Der Funktion wird ein Input gegeben, der validiert werden soll. Tritt dabei ein Fehler auf, wird `Error` zurückgegeben. Es macht Sinn, dabei einen String mit einer textuellen Beschreibung des Problems mitzuliefern. Wird die Funktion hingegen

ohne Fehler ausgeführt, gibt sie `Ok` mit den resultierenden Daten zurück. Das dient dann im weiteren Verlauf wiederum als Input für die nächste Funktion [39]. Eine Beispielimplementierung von Validate request ist in Quellcode 13 zu sehen.


Abbildung 3: Validate Visualisierung, Quelle: modifiziert übernommen aus [39]

```
type Request = {name:string; email:string}

// validate : input:Request -> Result<Request,string>
let validate input =
    if input.name = "" then Error "Name must not be blank"
    else if input.email = "" then Error "Email must not be blank"
    else Ok input
```

Quellcode 13: Validate request Implementierung, Quelle: modifiziert übernommen aus [39]

Die Typsignaturen der anderen Funktionen des Workflows sehen ähnlich aus. Alle nehmen irgendeinen Input entgegen und geben `Result` zurück [39].

### Railway Analogie

Nun müssen die Funktionen, die die einzelnen Schritte des Workflows repräsentieren, so zusammengefügt werden, dass der `Ok` Output einer Funktion als Input der nächsten fungiert, jedoch die folgenden Funktionen übersprungen werden, sollte ein Fehler aufgetreten sein. Eine Analogie, die hierfür verwendet wird, sind die Schienen einer Eisenbahnstrecke und deren Weichen. Davon kommt auch der Name Railway Oriented Programming (Railway = Eisenbahn). Eine Funktion, wie `validate` aus Quellcode 13, kann auch folgendermaßen mittels dieser Analogie dargestellt werden [39]:

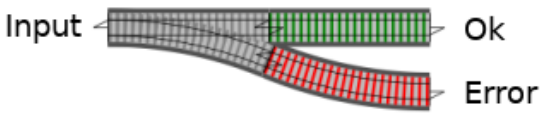


Abbildung 4: Eisenbahnschienen-Analogie einer Funktion, Quelle: modifiziert übernommen aus [39]

Die Verkettung vieler solcher Funktionen ist in Abbildung 5 dargestellt. Der obere Pfad wird ausgeführt, wenn es zu keinen Fehlern kommt. Sollte in einer Funktion etwas schieflaufen, kommt der Programmfluss über die nächste Weiche auf den unteren Pfad und umgeht somit die nächsten Funktionen im Workflow [39, 38].

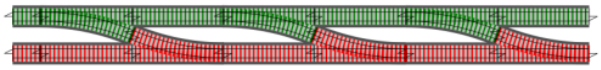

Abbildung 5: Verkettung der Eisenbahnschienen, Quelle: [39]

### Umsetzung der Analogie

Um die einzelnen Funktionen miteinander zu verbinden, so wie es mittels *Function Composition* in funktionalen Sprachen gängig ist, wird zusätzlich noch eine Art Adapter-Funktion benötigt. Diese wird üblicherweise `bind` genannt und ist für den Typen `Result` bereits implementiert [39, 34].

```
let bind binder result =
    match result with
    | Error e -> Error e
    | Ok x -> binder x
```

Quellcode 14: bind, Quelle: modifiziert übernommen aus [34]

`bind` nimmt zwei Argumente entgegen: eine Funktion `binder` und das Resultat `result` der vorigen Funktion. `binder` ist die Funktion, die aufgerufen werden soll, falls es zu keinem `Error` in der vorigen Funktion gekommen ist. In diesem Fall wird das Resultat `x` aus `result` geholt und der Workflow wird durch den Aufruf `binder x` fortgeführt. Sollte es jedoch zu einem Fehler gekommen sein, stoppt die Abarbeitung an dieser Stelle und es wird lediglich `Error e` zurückgegeben. Eine Implementierung des Workflows aus Abbildung 1 sieht unter Verwendung von *Function Composition* mittels `bind` nun so aus:

```
let testRailway =
    Result.bind validate
    >> Result.bind readUserRecord
    >> Result.bind updateUserRecord
    >> Result.bind sendVerificationMail
    >> Result.bind showResult
```

Quellcode 15: Workflow mit Function Composition und bind

Hier kommt es immer noch zu einer Verletzung des DRY-Prinzips (Don't Repeat Yourself): `Result.bind` muss bei jedem Schritt explizit aufgerufen werden. Um das zu umgehen, kann der Aufruf in einem neu definierten Operator versteckt werden:

```
let (>>=) result binder =
    bind binder result
```

Quellcode 16: Infix Operator »=, Quelle: modifiziert übernommen aus [39]

Hier ist die finale Implementierung des Workflows mithilfe des eigens definierten Operators:

```
let testRailway request =
    request |> validate
    >>= readUserRecord
    >>= updateUserRecord
    >>= sendVerificationMail
    >>= showResult
```

Quellcode 17: Workflow mit »= Operator

**Fazit**

Der Unterschied zwischen der ersten (Quellcode 11) und der finalen Implementierung (Quellcode 17) des Workflows äußert sich in Komplexität und Länge. Letztere weist keinerlei Verschachtelung auf (dadurch auch nicht mehr die Pyramid of Doom) und hat daher ein komplett lineares Codebild. Außerdem war es möglich, dieselbe Funktionalität in weniger als der Hälfte der Zeilen auszudrücken.

### 3.3.3 Computation Expressions

F# bietet ein Feature namens Computation Expressions, das dazu verwendet wird, Berechnungen auszudrücken, die aneinandergereiht und kombiniert werden können [29] (das Äquivalent dazu ist in Haskell die do-Notation [12]). Damit lässt sich auch Railway Oriented Programming umsetzen und somit kann es als Alternative zu der Implementierung aus Quellcode 17 verwendet werden. Um dieses Feature verwenden zu können, wird zuerst die sogenannte Builder-Klasse mit speziellen Methoden implementiert:

```
type ExampleWorkflowBuilder() =

    member this.Bind(m, f) = Result.bind f m

    member this.Return(x) = Ok x

let workflow = new ExampleWorkflowBuilder()
```

Quellcode 18: Computation Expression Builder, Quelle: modifiziert übernommen aus [37]

Die Methode `Bind` im Computation Expression Builder hat den gleichen Zweck wie die Methode `Result.bind` (siehe Quellcode 14), weshalb diese hier auch wieder verwendet wird. Die Methode `Return` wird benötigt, um das Ergebnis des gesamten Workflows wieder in den benötigten Typen zu wrappen, der in diesem Fall `Result<'T,'TError>` ist. Nun zu der Verwendung:

```
let testRailway request = workflow {
    let! a = validate request
    let! b = readUserRecord a
    let! c = updateUserRecord b
    let! d = sendVerificationMail c
    return d
}
```

Quellcode 19: Workflow mit Computation Expression Builder

Die Methode `Bind` wird also ausgeführt, wenn das Keyword `let!` verwendet wird, und die Methode `Return` bei `return`. Ein Vorteil ist, dass hier die Zwischenergebnisse von jedem Schritt in Variablen (`a, b, c` und `d`) gespeichert werden, auf die im gesamten `workflow { ... }`-Block zugegriffen werden kann.
Computation Expressions beschränken sich aber nicht nur auf Error Handling und die Verwendung von `let!` und `return`. Es gibt eine Vielzahl von anderen Methoden, die in der Builder-Klasse definiert werden können [29]. Weitere Anwendungen von Computation Expressions sind unter anderem:

- Asynchrone Workflows, siehe [28]
- Vortäuschen eines mutable States, siehe [41]
- Logging, siehe [21]

# 4 OOP Design Patterns in FP

In diesem Kapitel werden drei exemplarische OOP Design Patterns von Gamma u. a. [8] (Strategy, Command und Composite) auf ihre Anwendbarkeit in FP untersucht.

## 4.1 Strategy Pattern

### 4.1.1 Motivation

Das Strategy Pattern kommt aus der OOP-Welt. Gamma u. a. [8] definieren, dass es eine Gruppe von Algorithmen (Strategies) bestimmt, die abgekapselt und austauschbar sind. Diese Algorithmen können je nach Bedarf, unabhängig vom Verwender, innerhalb der Gruppe ausgetauscht werden. Laut Gamma u. a. [8] wird es angewendet, wenn

- viele Klassen sich nur in ihrem Verhalten unterscheiden.
- verschiedene Varianten eines Algorithmus benötigt werden (zum Beispiel unterschiedliche Sortieralgorithmen).
- ein Algorithmus Datenstrukturen verwendet, die der Verwender nicht zu sehen bekommen sollte.
- eine Klasse unterschiedliches Verhalten durch bedingte Verzweigungen aufweist, welche in eigene Strategies ausgelagert werden können.

### 4.1.2 Implementierung

#### OOP

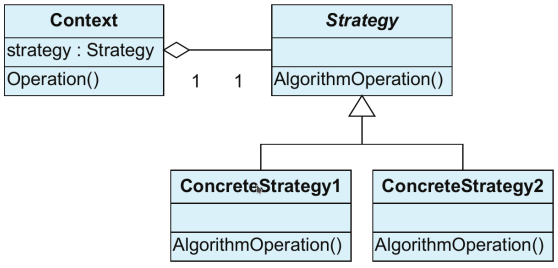


Abbildung 6: Klassendiagramm des Strategy Patterns, Quelle: [19, p. 212]

In Abbildung 6 ist die Struktur des Strategy Patterns in Form von einem Klassendiagramm dargestellt. `Strategy` ist ein Interface mit einer Methode, die eine Operation repräsentiert. Die beiden konkreten Strategien implementieren diese Operation jeweils anders und die `Context` Klasse kann zwischen den beiden Implementierungen wählen [19, p. 212]. Eine beispielhafte Implementierung ist in Quellcode 20 zu sehen. Dabei wird der `Context` Klasse die konkrete `strategy` als Konstruktorargument übergeben.

```
1 public Context
2 {
3    public Context(Strategy strategy) { ... }
4
5    int Operation(int x) {
6       return _strategy.AlgorithmicOperation(x);
7    }
8 }
```

Quellcode 20: Strategy Pattern Implementierung, Quelle: modifiziert übernommen aus [40]

#### FP

Das Interface `Strategy` der OOP Implementierung definiert nur eine einzige Methode, was nahe legt, dass es in FP durch eine Funktion ersetzt werden kann. Die beiden konkreten Strategien werden durch einfache Funktionen ersetzt, die als Parameter einer *Higher-Order Function* übergeben werden [19, p. 212]. Quellcode 21 verdeutlicht dies.

```
1 // strategy ist eine Funktion, also eine konkrete Strategie
2 let operation strategy x =
3     strategy x
```

Quellcode 21: Stragey Pattern in FP, Quelle: modifiziert übernommen aus [40]

Im Vergleich zur OOP Variante fällt das Interface weg und es müssen auch keine eigenen Klassen für die einzelnen Strategien erstellt werden, wodurch das Pattern in Sprachen mit *First-Class Functions* nicht mehr vorhanden ist [19, p. 212].

## 4.2 Command Pattern

### 4.2.1 Motivation

Beim Command Pattern in OOP werden Operationen in Command Objekte gekapselt, um diese zu einem späteren Zeitpunkt auszuführen, mehrere aneinanderzureihen oder sie wieder Rückgängig machen zu können [8, p. 263].

### 4.2.2 Implementierung

#### OOP

Abbildung 7 zeigt den Aufbau des Patterns in OOP. `Invoker` hat ein oder mehrere `Command` Objekte, die ihren `Receiver` kennen. Diese Objekte können durch ihr gemeinsames `Command` Interface in einer Liste gespeichert werden.

#### FP

In FP benötigt nicht jede Operation ein eigenes Wrapper-Objekt, das von einem gemeinsamen Interface ableitet, sondern kann durch eine Funktion dargestellt werden. In Quellcode 22 ist ein Beispiel von Tomas Petricek [19] zu sehen, in dem überprüft wird, ob ein Kredit für einen Klienten infrage kommt. Die Logik dafür ist in einer Liste von Funktionen enthalten, welche als Commands im Sinne des Command Patterns gesehen werden können.

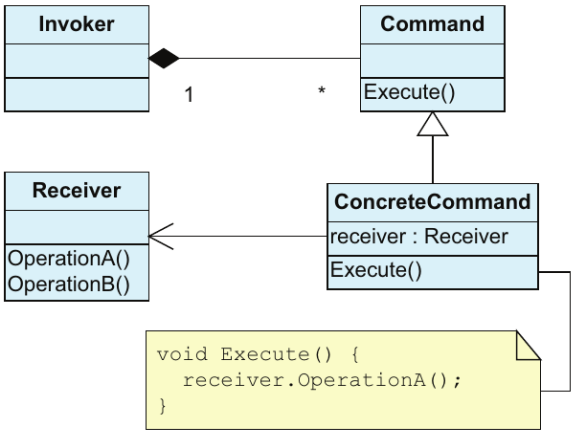


Abbildung 7: Klassendiagramm des Command Patterns, Quelle: [19, p. 213]

```
type Client =
    { Name : string; Income : int; YearsInJob : int
    UsesCreditCard : bool; CriminalRecord : bool }

let tests =
    [
        (fun cl -> cl.CriminalRecord = true);
        (fun cl -> cl.Income < 30000);
        (fun cl -> cl.UsesCreditCard = false);
        (fun cl -> cl.YearsInJob < 2)
    ]

let testClient(client) =
    let issues = tests |> List.filter (fun f -> f (client))
    let suitable = issues.Length <= 1
    suitable
```

Quellcode 22: Command Pattern in FP, Quelle: modifiziert übernommen aus [19, p. 210]

Die einzelnen Commands werden mithilfe der Lambda-Schreibweise in `tests` gespeichert und dann zu einem späteren Zeitpunkt aufgerufen. Genauso wie beim Strategy Pattern, wird hier das Pattern wieder durch Funktionen ersetzt.

## 4.3 Composite Pattern

### 4.3.1 Motivation

Dieses Pattern hat den Zweck, Objekte in einer Baumstruktur zusammenzufassen, um eine Teil-Ganzes-Beziehung darzustellen. Dadurch können individuelle Objekte und Kompositionen von Objekten einheitlich gehandhabt werden [8, p. 183].

### 4.3.2 Implementierung

#### OOP

In Abbildung 8 ist der Aufbau des Patterns aufgeschlüsselt. `AbstractComponent` ist eine abstrakte Klasse, von der `ConcreteComponent` und `CompositeComponent` ableiten. `CompositeComponent` enthält eine Liste von `AbstractComponents`. Die Methode `Operation()` von `CompositeComponent` iteriert über diese Liste und ruft dabei wieder die jeweiligen `Operation()` Methoden auf. Das Programm arbeitet nur mit Objekten vom Typ `AbstractComponent`, wodurch es nicht den Unterschied zwischen einzelnen und zusammengesetzten Objekten verstehen muss [19, p. 200].

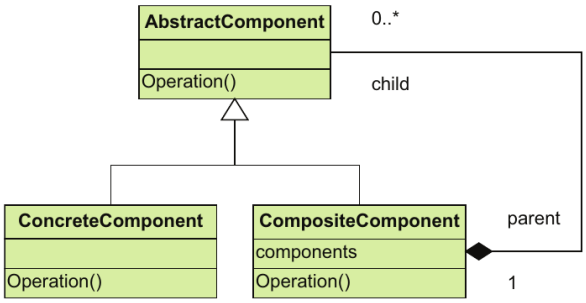


Abbildung 8: Klassendiagramm des Composite Patterns, Quelle: [19, p. 200]

#### FP

Wie in Quellcode 23 zu sehen ist, kann dieselbe Struktur in F# mittels Discriminated Unions (siehe [30]; in Haskell Algebraic Data Types, siehe [13]) dargestellt werden. Hier wird genauso rekursiv auf `AbstractComponent` verwiesen. Anders ist, dass keine `Operation()` im Datentyp selbst definiert wird, sondern unabhängig davon implementiert wird. Der Fokus in FP liegt auf der Fähigkeit, zu bestehenden Typen neue Funktionalitäten definieren zu können, weshalb hier die Komposition sozusagen “public” ist [19, p. 201].

```
1 type AbstractComponent =
2     | CompositeComponent of list<AbstractComponent>
```

```
3     | ConcreteComponent of int
```

Quellcode 23: Composite Pattern in FP, Quelle: modifiziert übernommen aus [19, p. 200]

In diesem Fall bleibt die Struktur des Patterns in FP also erhalten. Die Implementierung vereinfacht sich allerdings, da keine Klassen mit Ableitungshierarchie erstellt werden müssen.

# 5 Fazit

Der Begriff Design Pattern kann unterschiedlich weit gefasst werden. In der Interpretation nach Gamma u. a. [8] ruft er bei Softwareentwicklern differenzierte Reaktionen hervor, besonders im Bezug auf die funktionale Programmierung. In dieser Arbeit wurde deswegen analysiert, ob Muster in FP auftreten, die als Design Patterns klassifiziert werden können, und wie sich diese äußern.
Dafür wurden im ersten Teil beispielhaft drei Muster gezeigt, die in FP auftreten können. Lazy Evaluation, Memoisation und Railway Oriented Programming sind gängige Strukturen, die immer wieder zu beobachten sind, und daher sehr wohl unter den Begriff Design Pattern fallen können.
Der zweite Teil befasste sich mit der Frage, ob die klassischen GoF Patterns auch in FP anwendbar sind. Dafür wurden drei Patterns (Strategy, Command und Composite) exemplarisch ausgewählt und es wurde ermittelt, wie diese in FP aussehen würden. Strategy und Command Pattern wurden dabei beide durch *First-Class* und *Higher-Order Functions* abgelöst, wodurch das OOP Pattern in seiner ursprünglichen Form nicht mehr sichtbar war. Beim Composite Pattern konnte zumindest die generelle Struktur in FP beibehalten werden, wenn auch die Implementierung keine komplexe Klassenhierarchie mehr erforderte.
Design Patterns existieren daher auch in funktionalen Programmiersprachen, allerdings handelt es sich dabei um andere, als die GoF Patterns aus der objektorientierten Welt.

# Literaturverzeichnis


[1] ABILOV, V.: *Why care about functional programming? Part 1: Immutability*, 2013. [Online] Verfügbar unter: <https://www.miles.no/blogg/tema/teknisk/why-care-about-functional-programming-part-1-immutability> [Zugang am 28.12.2017].

[2] ABRAHAMSSON, J.: *A neat little type inference trick with C#*, 2011. [Online] Verfügbar unter: <http://joelabrahamsson.com/a-neat-little-type-inference-trick-with-c/> [Zugang am 13.01.2018].

[3] ACADEMIC: *Memoization*, 2017. [Online] Verfügbar unter: <http://deacademic.com/dic.nsf/dewiki/942644> [Zugang am 28.12.2017].

[4] ALEXANDER, A.: *Scala Cookbook*. O'Reilly Media, Sebastopol, 1 Aufl., 2013.

[5] BELITSKI, G.: *F# 4.0 Design Patterns*. Packt Publishing, Birmingham, 1 Aufl., 2016.

[6] BUSCHMANN, F. U. A.: *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, 1 Aufl., 1996.

[7] CORMAN, T. H. U. A.: *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 3 Aufl., 2009.

[8] GAMMA, E. U. A.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall, Upper Saddle River, 1 Aufl., 1994.

[9] GRAHAM, P.: *Revenge of the Nerds*, 2002. [Online] Verfügbar unter: <http://www.paulgraham.com/icad.html> [Zugang am 10.01.2018].

[10] HERMAN, D.: *Why coroutines won't work on the web*, 2011. [Online] Verfügbar unter: <http://calculist.org/blog/2011/12/14/why-coroutines-wont-work-on-the-web/> [Zugang am 30.12.2017].

[11] HOHPE, G.; WOOLF, B.: *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, 1 Aufl., 2003.

[12] LIPOVAČA, M.: *A Fistful of Monad*, o. J. [Online] Verfügbar unter: <http://learnyouahaskell.com/a-fistful-of-monads> [Zugang am 10.02.2018].

[13] LIPOVAČA, M.: *Making Our Own Types and Typeclasses*, o. J. [Online] Verfügbar unter: <http://learnyouahaskell.com/making-our-own-types-and-typeclasses> [Zugang am 12.02.2018].

[14] MASOOD, A.: *Learning F# Functional Data Structures and Algorithms*. Packt Publishing, Birmingham, 1 Aufl., 2015.

[15] MDN WEB DOCS: *Dynamic programming language*, o. J. [Online] Verfügbar unter: <https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language> [Zugang am 12.02.2018].

[16] MICROSOFT: *Results*, 2017. [Online] Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/results> [Zugang am 07.01.2018].

[17] NORVIG, P.: *Design Patterns in Dynamic Programming*, 1996. [Online] Verfügbar unter: <http://norvig.com/design-patterns/design-patterns.pdf> [Zugang am 10.01.2018].

[18] PANADERO, C.: *Functional Programming: Side Effects*, 2017. [Online] Verfügbar unter: <https://dzone.com/articles/side-effects-1> [Zugang am 28.12.2017].

[19] PETRICEK, T.: *Real-World Functional Ptrogramming: With Examples in F# and C#*. Manning, o. O., 1 Aufl., 2009.

[20] PETRICEK, T.: *Memoization for dynamic programming*, 2011. [Online] Verfügbar unter: <http://www.fssnip.net/8P/title/Memoization-for-dynamic-programming> [Zugang am 28.12.2017].

[21] PODWYSOCKI, M.: *A Kick in the Monads – Writer Edition*, 2010. [Online] Verfügbar unter: <http://codebetter.com/matthewpodwysocki/2010/02/02/a-kick-in-the-monads-writer-edition/> [Zugang am 10.02.2018].

[22] PRANAV, R.: *Promises in Sails JS - Remove Callback Hell*, 2014. [Online] Verfügbar unter: <http://maangalabs.com/blog/2014/08/23/promises-in-sails-js-remove-callback-hell/> [Zugang am 30.12.2017].

[23] RISHIKESH_SINGH: *Understanding Common Intermediate Language (CIL)*, 2013. [Online] Verfügbar unter: <https://www.codeproject.com/Articles/362076/Understanding-Common-Intermediate-Language-CIL> [Zugang am 09.02.2018].

[24] SANULLA, M.: *Lazy Initialization, Singleton Pattern and Double Checked locking*, 2012. [Online] Verfügbar unter: <https://javabeat.net/lazy-initialisation-singleton-and-double-check-locking/> [Zugang am 10.02.2018].

[25] SHVETS, A.: *Design Patterns*, o. J. [Online] Verfügbar unter: <https://sourcemaking.com/design_patterns> [Zugang am 10.01.2018].

[26] SKEET, J.: *Implementing the Singleton Pattern in C#*, o. J. [Online] Verfügbar unter: <http://csharpindepth.com/Articles/General/Singleton.aspx> [Zugang am 10.02.2018].

[27] TEAM, M. P. . P.: *Microsoft Application Architecture Guide*. Microsoft Press, o. O., 2 Aufl., 2009.

[28] THE F# SOFTWARE FOUNDATION: *Asynchronous Workflows*, 2016. [Online] Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/asynchronous-workflows> [Zugang am 10.02.2018].

[29] THE F# SOFTWARE FOUNDATION: *Computation Expressions*, 2016. [Online] Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions> [Zugang am 14.01.2018].

[30] THE F# SOFTWARE FOUNDATION: *Discriminated Unions*, 2016. [Online] Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions> [Zugang am 12.02.2018].

[31] THE F# SOFTWARE FOUNDATION: *Lazy Computations*, 2016. [Online] Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lazy-computations> [Zugang am 14.01.2018].

[32] THE F# SOFTWARE FOUNDATION: *Symbol and Operator Reference*, 2016. [Online] Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/symbol-and-operator-reference/index> [Zugang am 08.01.2018].

[33] THE F# SOFTWARE FOUNDATION: *fsharp*, 2017. [Online] Verfügbar unter: <https://github.com/fsharp/fsharp/blob/43289af17fad4869600fde51a5921246aaa86dd6/src/fsharp/FSharp.Core/prim-types.fsi> [Zugang am 07.01.2018].

[34] THE F# SOFTWARE FOUNDATION: *fsharp*, 2017. [Online] Verfügbar unter: <https://github.com/fsharp/fsharp/blob/master/src/fsharp/FSharp.Core/result.fs> [Zugang am 08.01.2018].

[35] WLASCHIN, S.: *Function associativity and composition*, 2012. [Online] Verfügbar unter: <https://fsharpforfunandprofit.com/posts/function-composition/> [Zugang am 08.01.2018].

[36] WLASCHIN, S.: *Overview of F# expressions*, 2012. [Online] Verfügbar unter: <https://fsharpforfunandprofit.com/posts/understanding-fsharp-expressions/> [Zugang am 13.01.2018].

[37] WLASCHIN, S.: *Computation expressions and wrapper types*, 2013. [Online] Verfügbar unter: <https://fsharpforfunandprofit.com/posts/computation-expressions-wrapper-types/> [Zugang am 14.01.2018].

[38] WLASCHIN, S.: *How to design and code a complete program*, 2013. [Online] Verfügbar unter: <https://fsharpforfunandprofit.com/posts/recipe-part1/> [Zugang am 29.12.2017].

[39] WLASCHIN, S.: *Railway oriented programming*, 2013. [Online] Verfügbar unter: <https://fsharpforfunandprofit.com/posts/recipe-part2/> [Zugang am 29.12.2017].

[40] WLASCHIN, S.: *Functional Programming Patterns (NDC London 2014)*, 2014. [Online] Verfügbar unter: <https://www.slideshare.net/ScottWlaschin/fp-patterns-ndc-london2014> [Zugang am 29.12.2017].

[41] WLASCHIN, S.: *The "Handling State" series*, 2015. [Online] Verfügbar unter: <https://fsharpforfunandprofit.com/series/handling-state.html> [Zugang am 10.02.2018].

# Abbildungsverzeichnis

# Quellcodeverzeichnis