


# RAILWAY ORIENTED PROGRAMMING

## KOMPLEXE ORCHESTRIERUNG WARTBAR MACHEN

Patrick Drechsler

- [patrick.drechsler@redheads.de](mailto:patrick.drechsler@redheads.de)
-  [@drechsler](https://twitter.com/drechsler)

## PATRICK DRECHSLER

- "gelernter" Biologe
- C# Entwickler bei Redheads Ltd.
- aktuelle Schwerpunkte: DDD, Cloud
- Softwerkskammer

# DISCLAIMER

Ich werde nicht erklären, was eine Monade ist.

*Wenn man verstanden hat, was eine Monade ist, verliert man die Fähigkeit zu erklären, was eine Monade ist.*

(Monaden-Paradoxon)

Audio:

Scott Wlaschin ("Mr. F#") on Monads  
(2min)

How do I work with errors  
in a functional way?

Fremde Begriffe  
lassen wir erstmal weg

# WIR SIND FAUL

## WARTBAREN CODE SCHREIBEN!

(mein zukünftiges ich wirds dir danken)

# WAS IST "ORCHESTRIERUNG"?

## Code

- mit wenig interner Logik
- bei dem viel "zusammenläuft":
  - viele Abhängigkeiten
  - beschreibt den Ablauf einer User Story
  - oft in "Service" Klassen (z.B. RegistrationService)



## Unser Ziel:

```
var customerResult = Validate(createCustomerViewModel);  
var result = customerResult  
    .OnSuccess(c => _customerRepository.Create(c))  
    .OnSuccess(c => _mailConfirmer.SendWelcome(c))  
    .OnBoth(cResultAtEnd => cResultAtEnd.IsSuccess  
        ? new CustomerCreatedViewModel(cResultAtEnd.Value.Id)  
        : CreateErrorResponse(cResultAtEnd.Error));
```

C#

# USER STORY: ANMELDUNG ALS NEUER BENUTZER

Wenn ein neuer Benutzer sich anmeldet,

- werden seine Eingaben validiert
- wird er im System gespeichert
- erhält er eine Bestätigungsmail

```
public CustomerCreatedViewModel RegisterCustomer(SomeVM viewModel)
{
    var customer = Validate(viewModel);
    customer = _customerRepository.Create(customer);
    _mailConfirmer.SendWelcome(customer);

    return new CustomerCreatedViewModel(customer);
}
```

- Cool, wir sind fertig!
- let's go live...

**...NO ERROR HANDLING...**

**WHAT COULD POSSIBLY  
GO WRONG?**

*...potentielle Fallstricke...*

C#

```
// can fail
var customer = Validate(createCustomerViewModel);

// can fail
customer = _customerRepository.Create(customer);

// can fail
_mailConfirmer.SendWelcome(customer);

return new CustomerCreatedViewModel(customer.Id) {Success = ??};
```

# PRO-TIPP

## GEWÜNSCHTES FEHLERVERHALTEN ABKLÄREN

- Nicht einfach drauflos programmieren:
  - Zuerst mit Kunde/Domain-Experten klären!
  - Dann die User Story aktualisieren (oder neue User Story für Fehlerfälle erstellen)

# FEHLERBEHANDLUNG

C#

```
Customer customer;  
try { customer = Validate(createCustomerViewModel); }  
catch (Exception e) { return CreateErrorResponse(e); }  
  
try { customer = _customerRepository.Create(customer); }  
catch (Exception e) { return CreateErrorResponse(e); }  
  
try { _mailConfirmer.SendWelcome(customer); }  
catch (Exception e)  
{  
    // don't fail, but maybe: logging, retry-policy  
}  
  
return new CustomerCreatedViewModel(customer.Id);
```

- Fehlerbehandlung macht einen Großteil des Codes aus
- Ergebnis einer Aktion ist oft Grundlage für weitere Aktion



# FUNKTIONALE PROGRAMMIERUNG

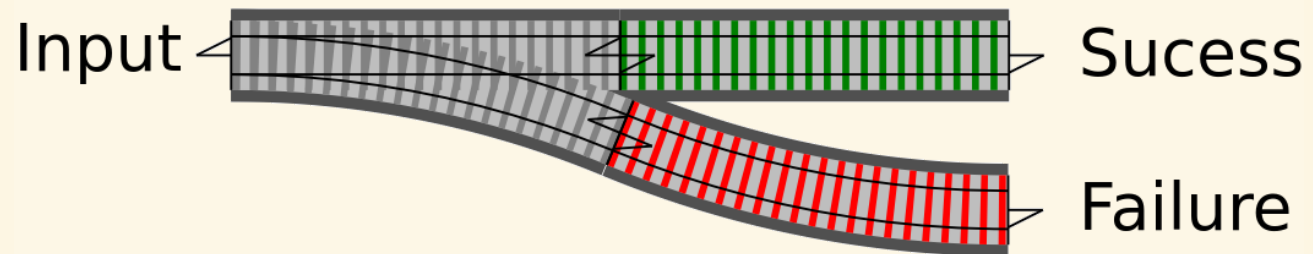
Java und C# haben Methoden

Was unterscheidet Methoden von Funktionen?

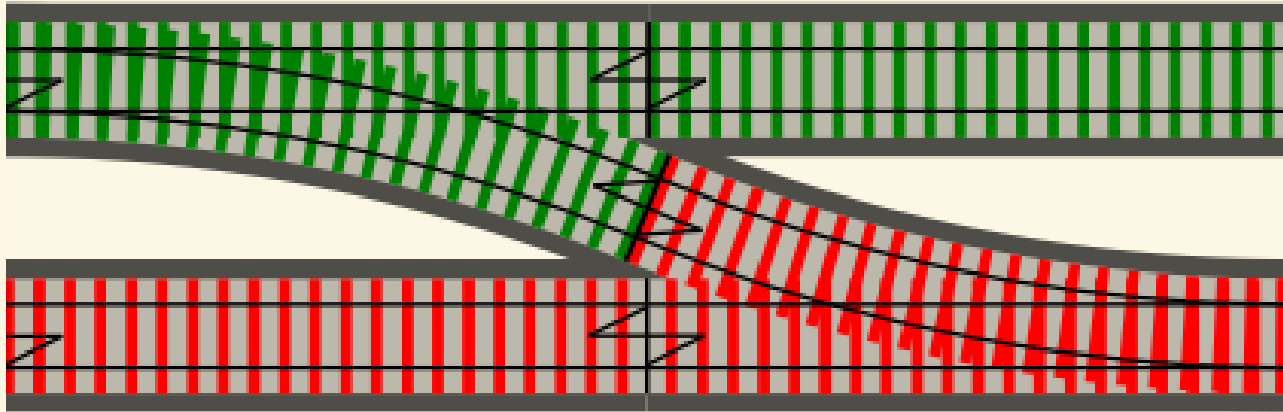
- Pure Functions
  - gleiche Eingabe gibt immer gleiches Ergebnis zurück
  - keine Seiteneffekte
- Higher Order Functions
  - Funktionen können als Eingabe- und Rückgabewert verwendet werden

**WHERE ARE THE TRAINS?**  
**YOU PROMISED RAILWAYS!**

Unsere `if/else` oder `try/catch` sieht bisher so aus:



# Wir brauchen 2 Eingaben



- Eingang 1: Success
- Eingang 2: Failure
- -> beide Infos in ein Objekt kapseln!

Basteln wir uns die Weiche zusammen...

Nennen wir die Weiche **"Result"**...

In C# und in Java gibt es aktuell keine "Result" Klasse



- C#: CSharpFunctionalExtensions (NuGet)
- Java: auch möglich (Link im Abspann)



C#

```
public class Result {  
    public bool Success { get; }  
    public string Error { get; }  
  
    protected Result(bool success, string error) { /* ... */ }  
  
    public static Result Fail(string message) { /* ... */ }  
  
    public static Result<T> Ok<T>(T value) { /* ... */ }  
}
```

C#

```
public class Result<T> : Result {  
    public T Value { get; }  
    public bool IsFailure ⇒ !Success;  
  
    protected internal Result(T value, bool success, string error)  
        : base(success, error) {  
        Value = value;  
    }  
}
```



# ERSTELLEN VON RESULT

```
Result<Customer> ok = Result.Ok(new Customer { /* ... */ })  
Result<Customer> fail = Result.Fail("Ups");
```

C#

```
public Result<Customer> Validate(CustomerWantsToRegisterVM vm) {  
    if (IsValid(vm)) {  
        var customer = MapVm2Domain(vm)  
        return Result.Ok(customer);  
    }  
    else {  
        return Result.Fail("invalid");  
    }  
}
```

C#

# LIVE CODING

## TDD

Wait: what is TDD?

Test Driven Development

Test: Dokumentation, Usage Example

# KOMBINATION VON RESULTS

(via Extension Methods)

- OnSuccess
- OnBoth
- OnFailure

Hinweis: Extension Methods in C# sind wie "traits" (Scala) oder "mixins" (Ruby)

# VERKETTEN VON RESULT

```
static Result<U> OnSuccess(this R<T> result,  
                           Func<T, U> func) { /* ... */ }
```

C#

```
static Result<T> OnFailure<T>(this Result<T> result,  
                              Action<string> action) { /* ... */ }
```

C#

```
static K OnBoth<T, K>(this Result<T> result,  
                      Func<Result<T>, K> func) { /* ... */ }
```

C#

# LIVE CODING

## TDD

(no more slides)

# AUSBLICK FSHARP

```
type Result<'TSuccess,'TFailure> =  
    | Success of 'TSuccess  
    | Failure of 'TFailure  
  
type Request = {name:string; email:string}  
  
let validate1 input =  
    if input.name = "" then Failure "Name must not be blank"  
    else Success input  
  
let validate2 input =  
    if input.name.Length > 50 then Failure "Name must not be longer ... "  
    else Success input  
  
let validate3 input =  
    if input.email = "" then Failure "Email must not be blank"  
    else Success input
```

```
let bind switchFunction twoTrackInput =  
    match twoTrackInput with  
    | Success s → switchFunction s  
    | Failure f → Failure f
```



F#

```
// Option 1
let combinedValidation =
    let validate2' = bind validate2
    let validate3' = bind validate3
    validate1 >> validate2' >> validate3'
```

F#

```
// Option 2
let combinedValidation =
    validate1
    >> bind validate2
    >> bind validate3
```

F#

```
// Option 3
let combinedValidation =
    validate1
    ⇒ validate2
    ⇒ validate3
```




<https://fsharpforfunandprofit.com/posts/recipe-part2/>

# LINKS

- Scott Wlaschin "the original talk"  
<http://fsharpforfunandprofit.com/rop/>
- Stefan Macke "ROP für Java"  
<https://www.heise.de/developer/artikel/Railway-Oriented-Programming-in-Java-3598438.html>
- Vladimir Khorikov "Functional C#: Handling failures..."  
<http://enterprisecraftsmanship.com/2015/03/20/functional-c-handling-failures-input-errors/>
- CSharpFunctionalExtensions  
<https://github.com/vkhorikov/CSharpFunctionalExtensions>

**DANKE!**

# FRAGEN?

-  @drechsler
-   
patrick.drechsler@redheads.de
-  draptik