

WIR SIND FAUL

(dein zukünftiges Ich wirds dir danken)

7

WAS IST "ORCHESTRIERUNG"?

Code

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

8

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

9

Unser Ziel:

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

C#

10

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

    return new CustomerCreatedViewModel(customer);
}
```

C#

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

11

...NO ERROR HANDLING...

WHAT COULD POSSIBLY GO WRONG?

12.1

...potentielle Fallstricke...

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

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

// can fail
_mailConfirmer.SendWelcome(customer);

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

C#

13

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)

14

```
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);
```

C#

15

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

16

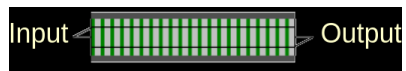
FUNKTIONALE PROGRAMMIERUNG

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

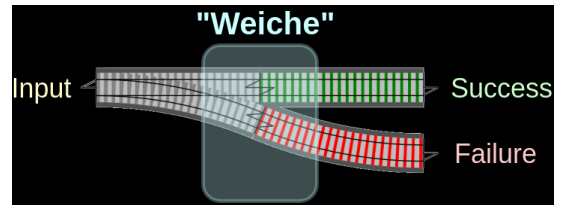
17



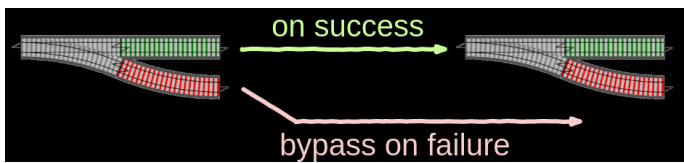
18



19



20

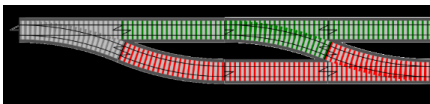


21



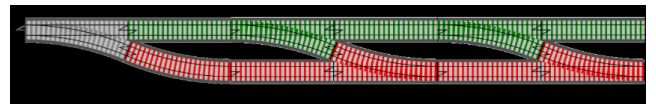
22

einfach: F2 kann Failure empfangen:



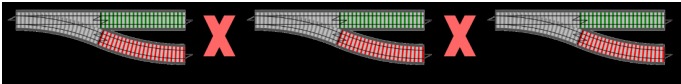
23

dann kann man weiterarbeiten:



24

Ups: F2, F3 können keinen Fehler entgegennehmen:



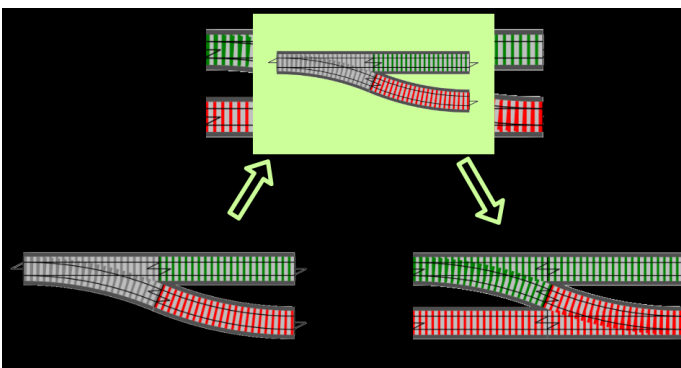
25

wir brauchen eine Funktion, die Fehler entgegennimmt:

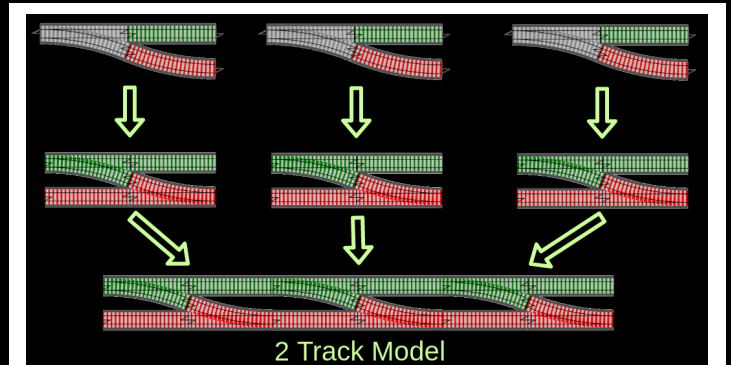


26

Umwandeln von 1-Track Input in 2-Track Input mit einem "Adapter Block"

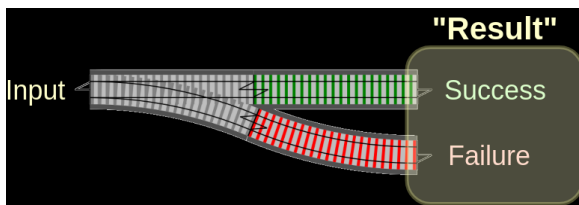


27



28

"Result" kapselt Success und Failure



29

Basteln wir uns ein Result...

30

- "Result" ist kein Sprachfeature von C# / Java ☹️
- C#
 - CSharpFunctionalExtensions
 - LaYumba.Functional
 - language-ext
- Java: auch möglich (Link im Abspann)
- F#: Sprachfeature



31

"RESULT ZU FUSS"...

```
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) { /* ... */ }
}
```

```
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;
    }
}
```

32

ERSTELLEN VON RESULT

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

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

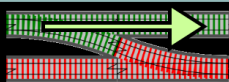
33

LIVE CODING

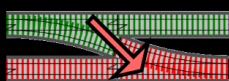
(Result Klasse zu Fuß)

34

Creating a Result
(Customer -> Result<Customer>)

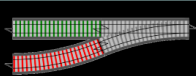


"OnSuccess"



"OnFailure"

Convert Result<T> back to T
"OnBoth"



35

KOMBINATION VON RESULTS

(via Extension Methods)

- OnSuccess
- OnBoth
- OnFailure

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

36

VERKETTEN VON RESULT

```
static Result<U> OnSuccess<T>(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#

37

LIVE CODING?

38

Ausblick: **F#**...

39.1

Result ist mittlerweile ein Sprachfeature von F#, kann aber auch einfach selbst implementiert werden:

```
// discriminated union  
type Result<'TSuccess,'TFailure> =  
    | Success of 'TSuccess  
    | Failure of 'TFailure
```

F#

39.2

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

F#

... kombiniert zwei 2-Track Funktionen ...

(entspricht OnSuccess, OnFailure, OnBoth)

39.3

Anwendungsbeispiele

```
type Request = {name:string; email:string} // ← Record type  
  
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
```

F#

39.4

```
// 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
```

F#

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

39 . 5

Haben wir unser Ziel erreicht?

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

C#

40

- lesbarer & wartbarer Code
- kompakte Fehlerbehandlung
- **Fehlerbehandlung wird Bestandteil der Domäne!**




...nebenbei haben wir Sinn und Zweck der "Either-Monade" verstanden... 😊

41

- 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/>
- C# Bibliotheken
 - CSharpFunctionalExtensions
<https://github.com/vkhorikov/CSharpFunctionalExtensions>
 - LaYumba.Functional <https://github.com/la-yumba/functional-csharp-code>
 - language-ext <https://github.com/louthy/language-exts>

42

DANKE!

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

43