Railway Oriented Programming

Patrick Drechsler
@drechsler
Redheads Ltd.

NUG Gelsenkirchen
2019-05-07

@drechsler
github.com/draptik
draptik.github.io
redheads.de

red·head·s
SOFTWARE SOLUTIONS SUPPORT

PATRICK DRECHSLER

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

https://pixabay.com/users/Larisa-K-1107275/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=1635118

MEIN ZIEL

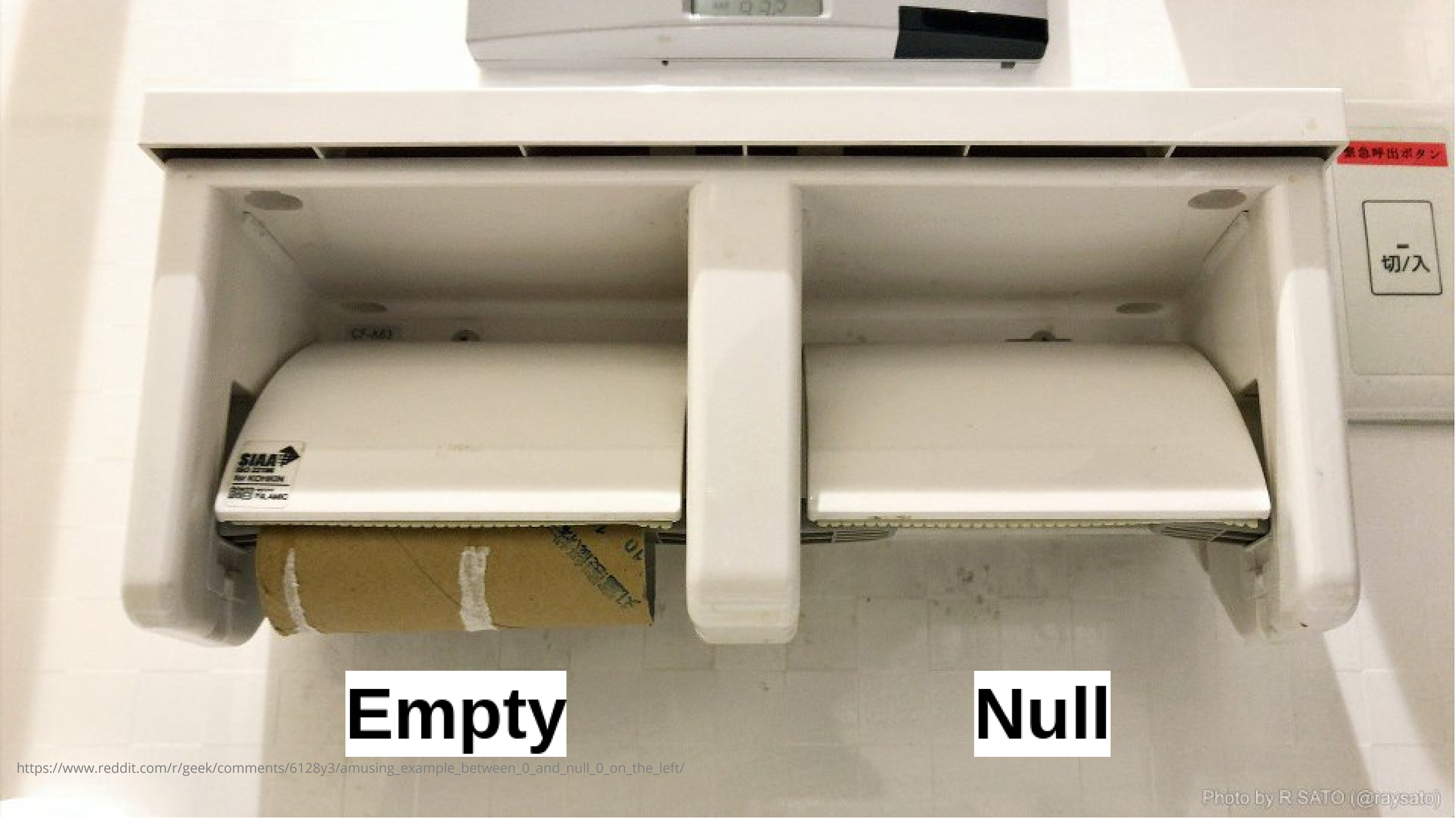
https://pixabay.com/users/Fotoworkshop4You-2995268/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=1555348

NULL IS A PAIN

```
public void Foo(string s)
{
    if (string.IsNullOrWhiteSpace(s))
    {
        // ...
    }
}
```

```
public void Foo(List<string> someStringCollection)
{
    if (someStringCollection ≠ null && someStringCollection.Any())
    {
        // ...
    }
}
```

```
public void Foo(Customer customer)
{
    if (customer?.Address?.ZipCode ≠ null)
    {
        // ..
    }
    else
    {
        // ..
    }
}
```


緊急呼出ボタン
切/入
Empty
Null
https://www.reddit.com/r/geek/comments/6128y3/amusing_example_between_0_and_null_0_on_the_left/
Photo by R SATO (@raysato)

ANTIPATTERN: PRIMITIVE OBSESSION

> *Like most other [code] smells, primitive obsessions are born in moments of weakness.* ***"Just a field for storing some data!"*** *the programmer said. Creating a primitive field is so much easier than* ***making a whole new class****, right?*

https://sourcemaking.com/refactoring/smells/primitive-obsession

Tony Hoare

From Wikipedia, the free encyclopedia

For the bicycle racer, see Tony Hoar.

Sir Charles Antony Richard Hoare FRS FREng[3] (born 11 January 1934),[4] is a British computer scientist. He developed the sorting algorithm quicksort in 1959/1960.[5] He also developed Hoare logic for verifying program correctness, and the formal language communicating sequential processes (CSP) to specify the interactions of concurrent processes (including the dining philosophers problem) and the inspiration for the occam programming language.[6][7][8][9][10][11]

Contents [hide]

Sir Tony Hoare

FRS FREng

Tony Hoare in 2011

Born	Charles Antony Richard Hoare 11 January 1934 (age 85) Colombo, British Ceylon
Residence	Cambridge

Tony Hoare

From Wikipedia, the free encyclopedia

For the bicycle racer, see Tony Hoar.

Sir Charles Antony Richard Hoare FRS FREng[3] (born 11 January 1934),[4] is a British computer scientist. He developed the sorting algorithm quicksort in 1959/1960.[5] He also developed Hoare logic for verifying program correctness, and the formal language communicating sequential processes (CSP) to specify the interactions of concurrent processes (including the dining philosophers problem) and the inspiration for the occam programming language.[6][7][8][9][10][11]

Contents [hide]

Sir Tony Hoare

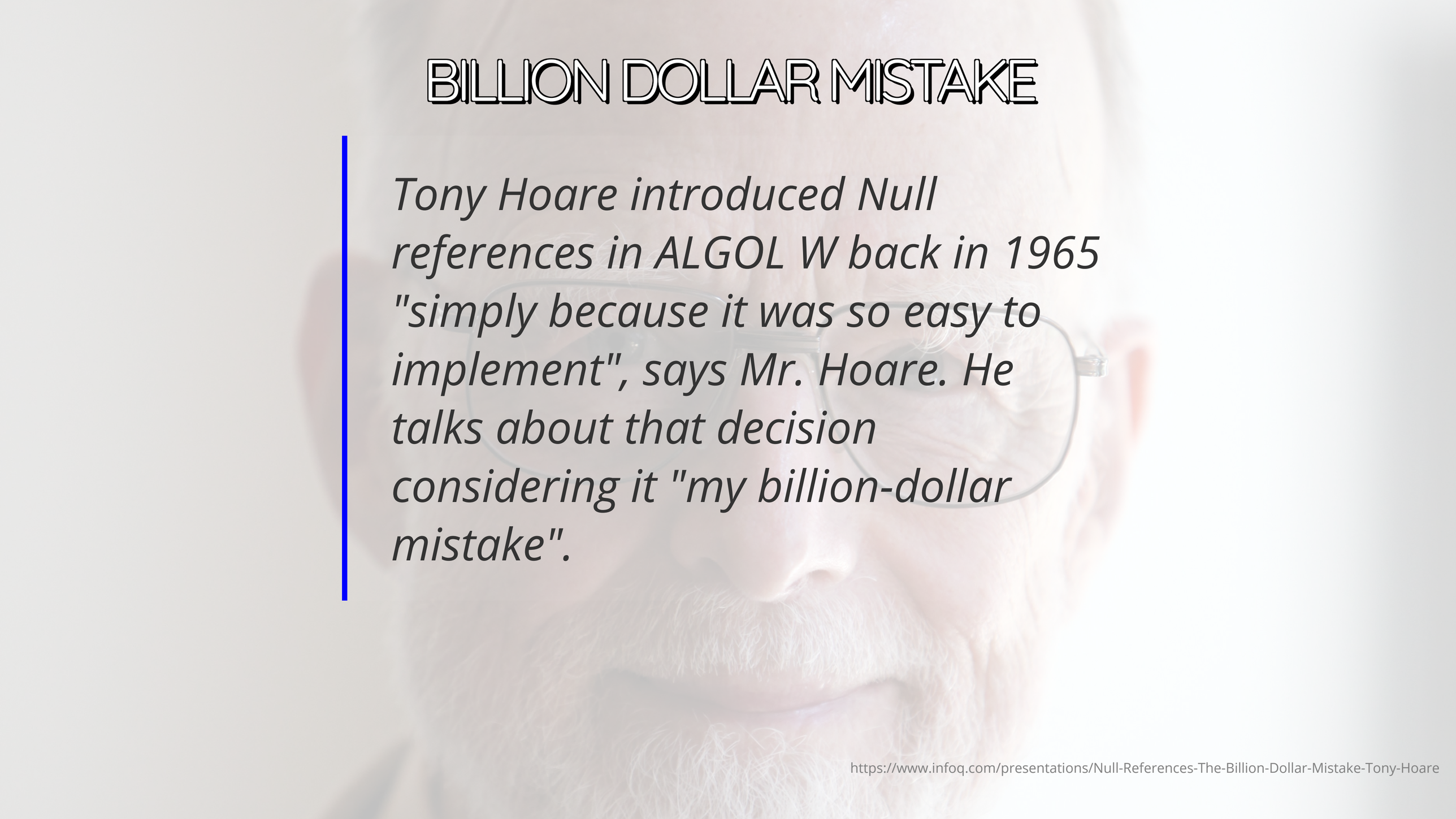
FRS FREng

Tony Hoare in 2011

Born	Charles Antony Richard Hoare 11 January 1934 (age 85) Colombo, British Ceylon
Residence	Cambridge

BILLION DOLLAR MISTAKE

Tony Hoare introduced Null references in ALGOL W back in 1965 "simply because it was so easy to implement", says Mr. Hoare. He talks about that decision considering it "my billion-dollar mistake".



https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

everybody lies
and everybody
pays a price.

ERRORS ARE A PAIN

ARITHMETIC ERRORS

```
int Add(int a, int b) ⇒ a + b;

var i1 = int.MaxValue;
var i2 = 1;

var result = Add(i1, i2);
```

INFRASTRUCTURE ERRORS

```
Customer GetById(Guid id)
{
    return db.GetById(id);
}
```

- Offensichtliche Lüge!

IMHO

- using Exception-Handling as control flow is bad practice
  - (unless you are performance optimizing)

Anticipate Errors!

- Fehler im Workflow sind normal (nicht die Ausnahme)
- Fehler müssen auf Business-Ebene behandelt werden

everybody lies
and everybody
pays a price.

BAD MODELS ARE A PAIN

...hier sind wir (als Software Entwickler) gefragt...


```
class Customer
{
    string FirstName { get; set; }
    string LastName { get; set; }
    string Email { get; set; }
}
```

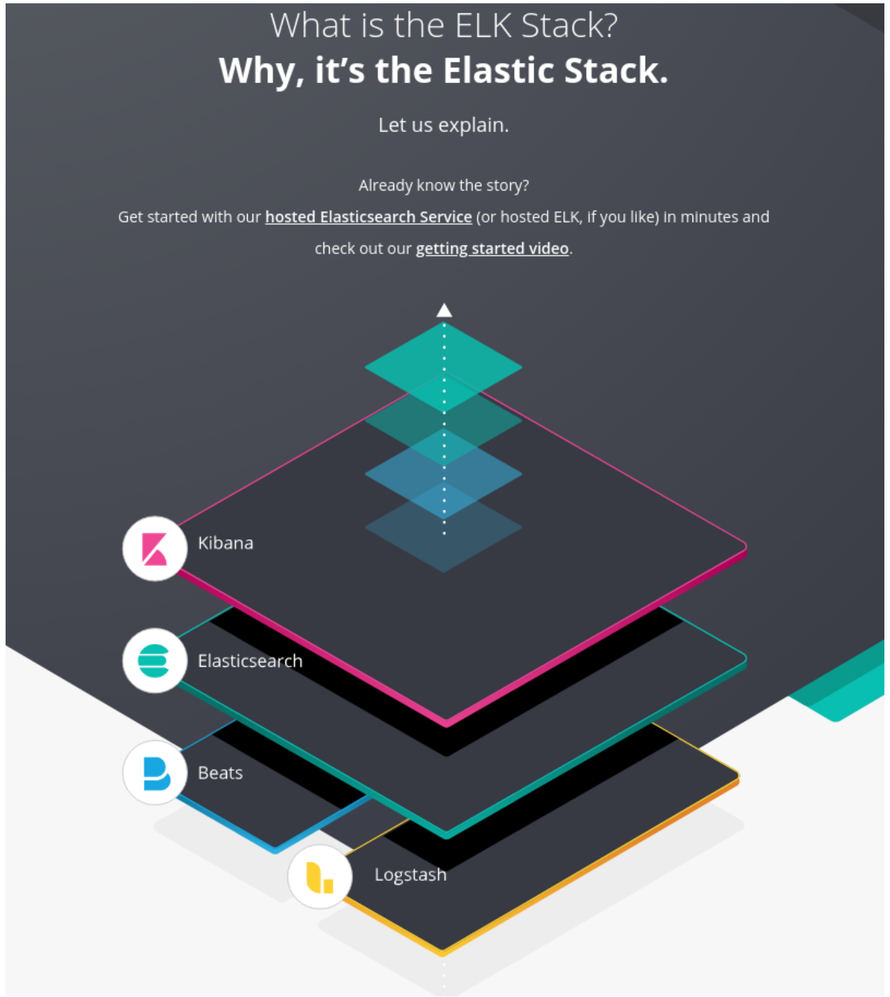
VS

```
class Customer
{
    Name FirstName { get; set; }
    Name LastName { get; set; }
    Email Email { get; set; }
}

class Name { /* ... */ }
class Email { /* ... */ }
```

LOG LIKE A PRO

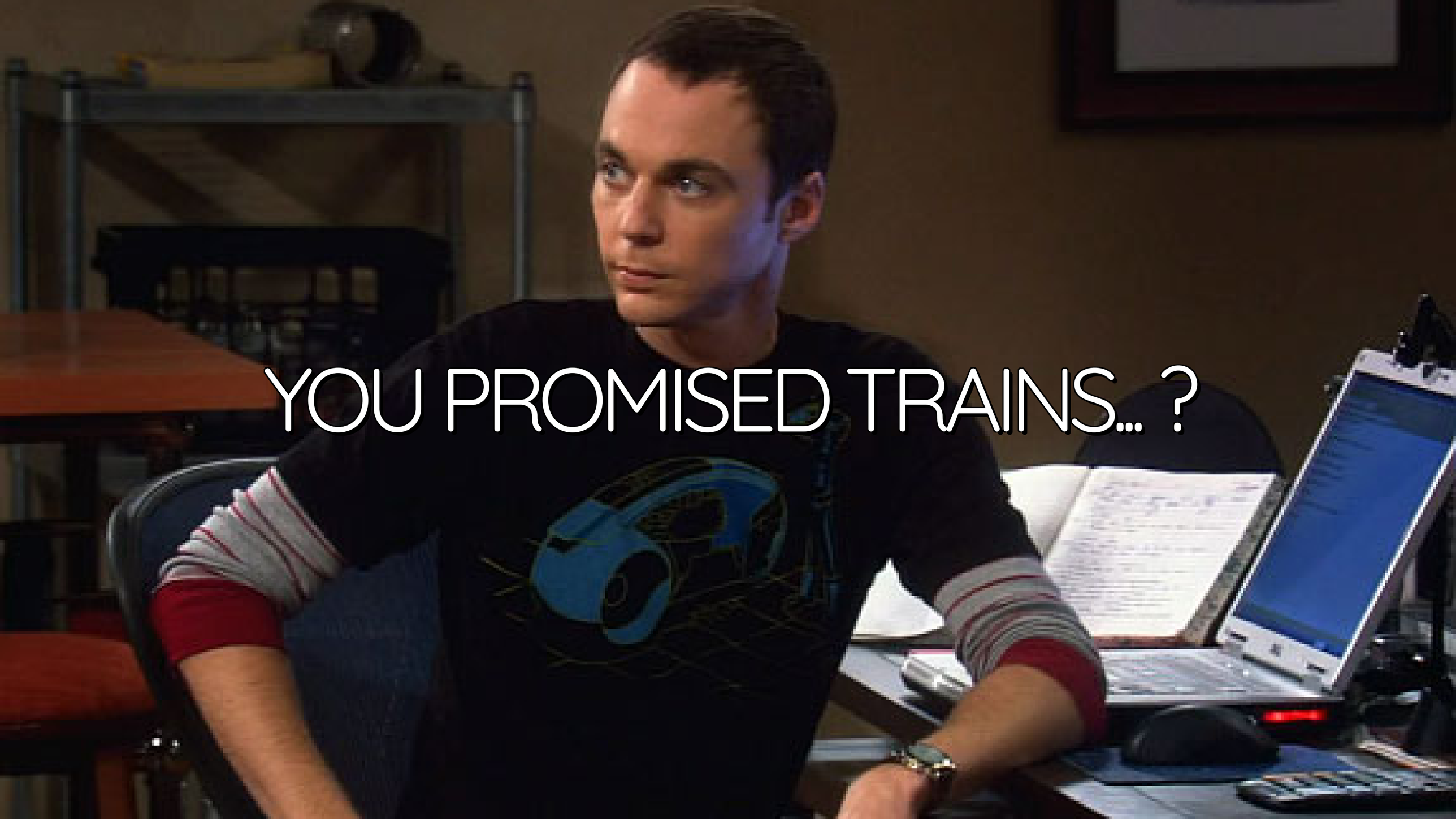
ELK

- Elastic
- Logstash
- Kibana

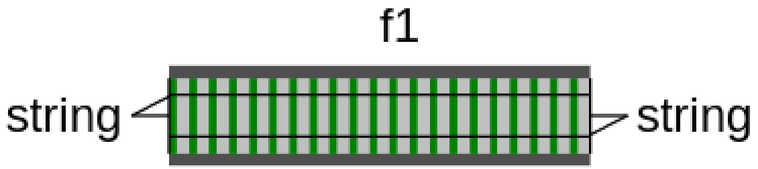


OPS

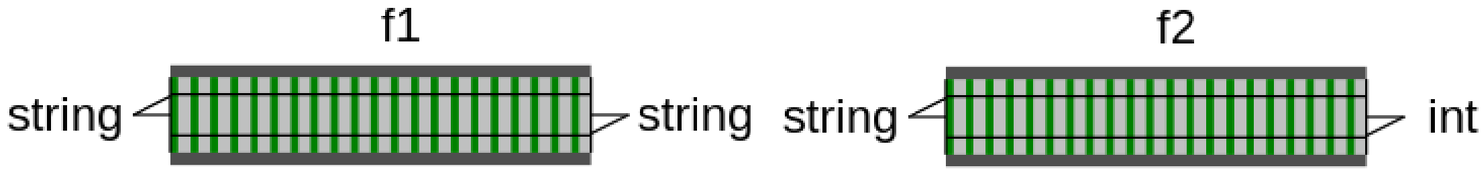
...will thank you


YOU PROMISED TRAINS... ?

SINGLE TRACK MODEL

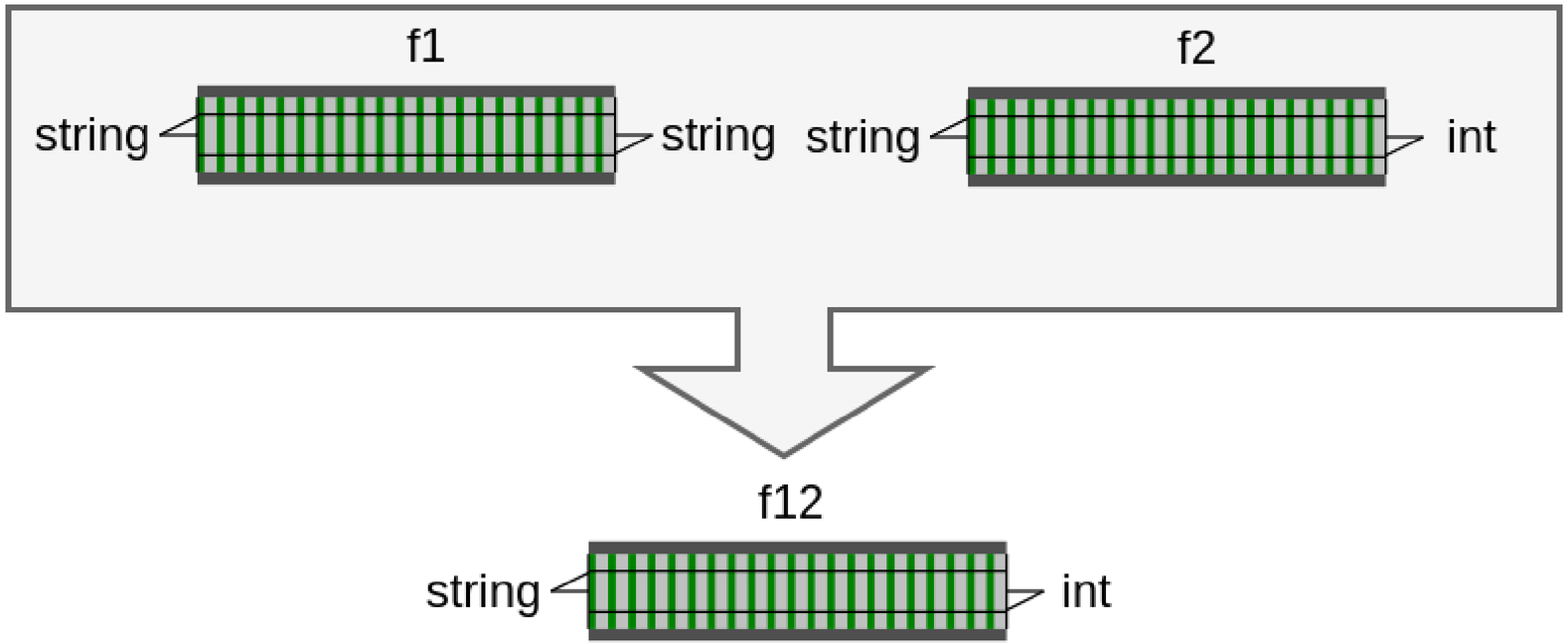



```
string f1(string input) ⇒ input.ToUpper();
var result = f1("abc");      // → "ABC"
```

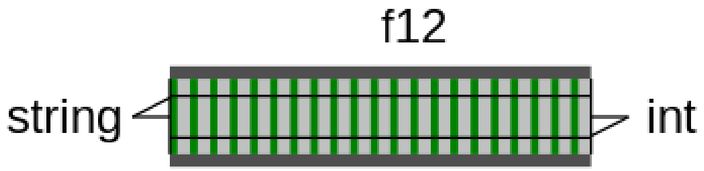



```
string f1(string input) ⇒ input.ToUpper();
int    f2(string input) ⇒ input.Length;

var r1 = f1("abc");
var r2 = f2(r1);   // → 3
```

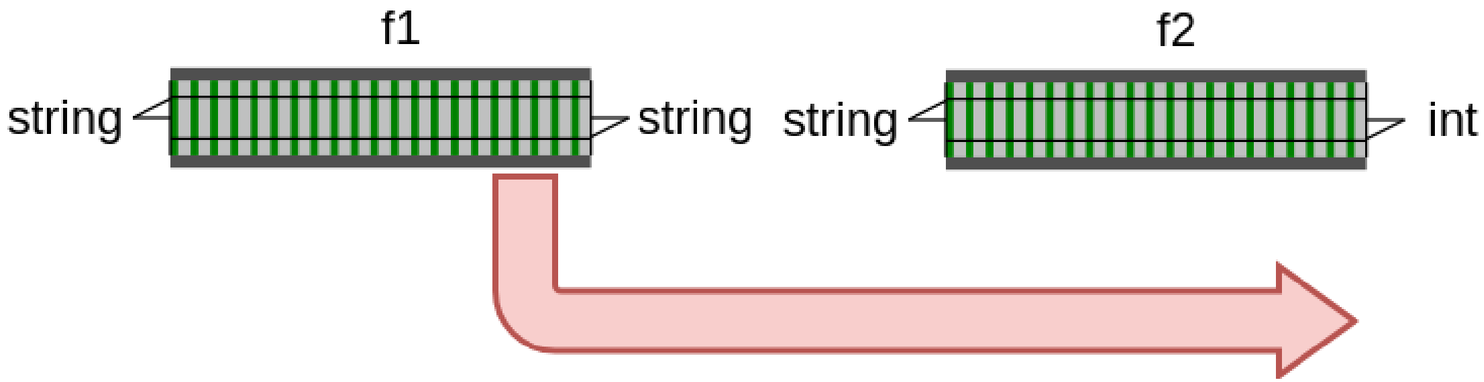
Composition



Composition



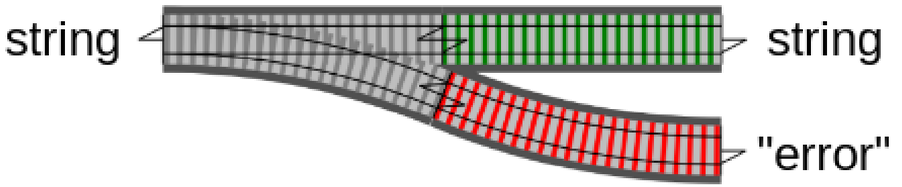

```
string f1 (string input) ⇒ input.ToUpper();
int    f2 (string input) ⇒ input.Length;

int    f12(string input) ⇒ f2(f1(input)); // "Composition"

var result = f12("abc");      // → 3
```

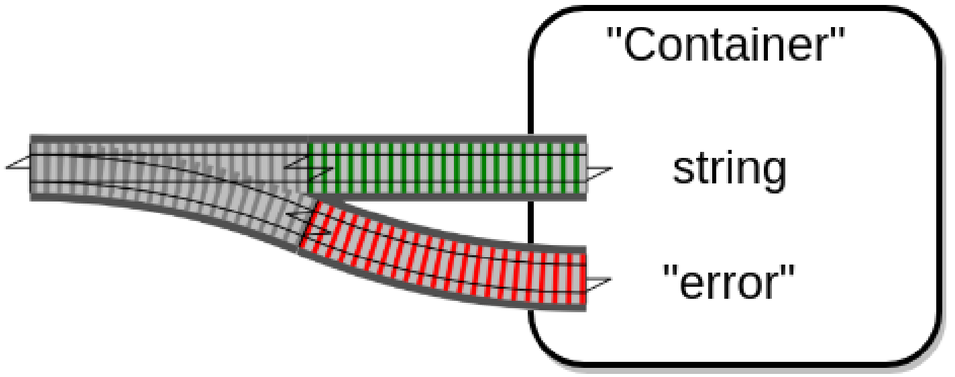


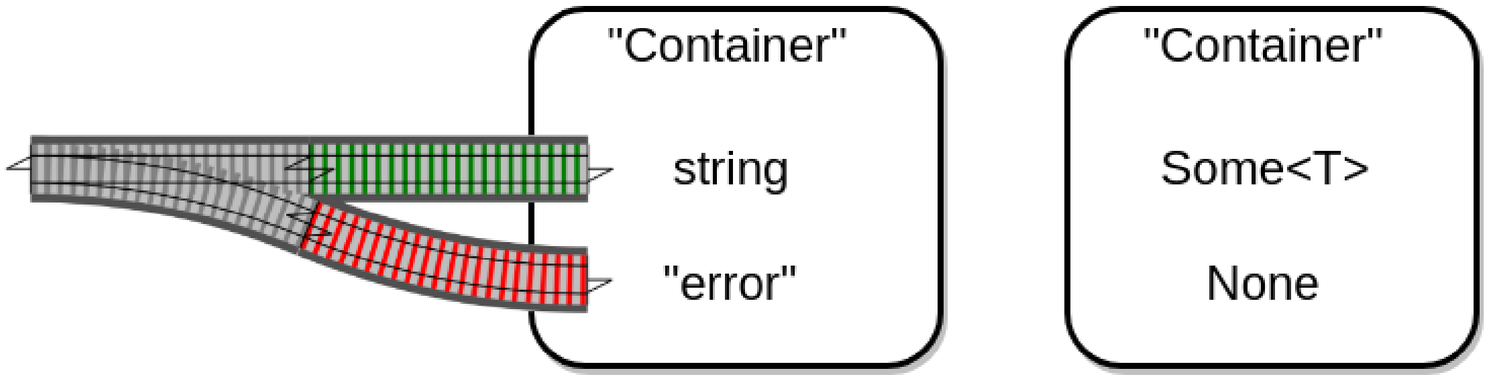
- `f1` kann fehlschlagen
- Komposition von `f1` und `f2` nicht möglich

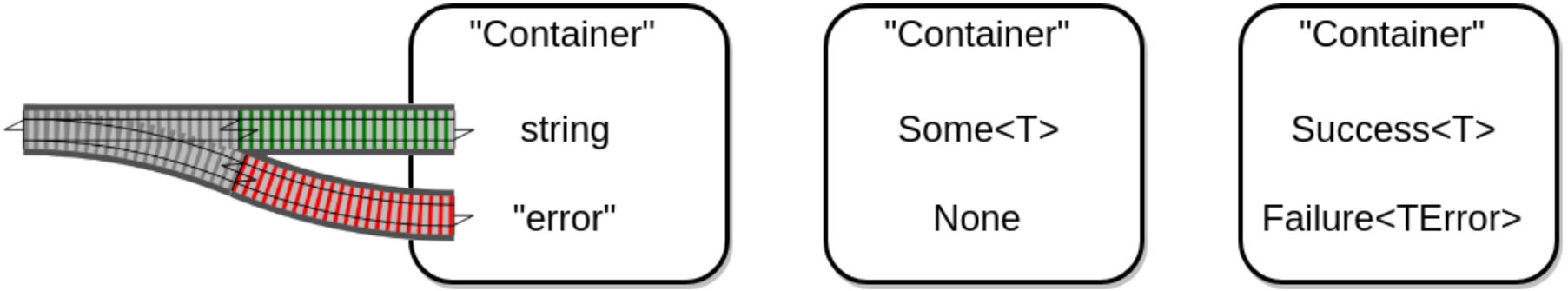
```
try {
    var resultF1 = f1(s);
    var resultF2 = f2(resultF1);
} catch {
    // ...
}
```

SINGLE TRACK WITH SWITCH


string
string
"error"


"Container"
string
"error"


"Container"
string
"error"
"Container"
Some<T>
None


"Container"
string
"error"
"Container"
Some<T>
None
"Container"
Success<T>
Failure<TError>

nennen wir `Container` besser `Result`

Funktionen, die fehlschlagen können, sollten `Result` zurückgeben

wie könnte eine `Result` Klasse aussehen?

```
// Pseudo code (!)
class Result<TSuccess, TError>
{
    TSuccess Success { get; }
    TError Failure { get; }

    private Result(TSuccess success, TError error) {
        Success = success;
        Failure = error;
    }

    static Success(TSuccess success) ⇒ Result(success, null);

    static Failure(TError error) ⇒ Result(null, error);

    bool IsSuccess ⇒ Success ≠ null;
    bool IsFailure ⇒ !IsSuccess;
}
```

- `Result` muss eindeutigen Zustand haben...

Wie kann man Funktionen verketten?

- `f1` gibt `Result` zurück
- `f2` kann `Result` nicht entgegennehmen!

`Result` muss **Single-Input Funktion** entgegennehmen

Single-Input Funktion mit `Result` Rückgabewert hat folgende Signatur:

```
Result<int, Error> f2(string input) { /* .. */ }

// Function signature
Func<string, Result<int, Error>> func2

// Allgemein
Func<TSuccess1, Result<TSuccess2, TError>> func2
```

Das Lesen so einer Signatur benötigt anfangs etwas Übung

FP ARROW NOTATION

```
f1 :: string → string

f2 :: string → int

f12 :: (string → string) → int

f12' :: string → int
```

```
Func<string, string> f1

Func<string, int> f2

Func<Func<string, string>, int> f12

Func<string, int> f12'
```

COMBINING RESULTS

```
static class ResultExtensions
{
    static Result<TSuccess2, TError> OnSuccess   //⟵ FP-Jargon: "Bind"
        (this Result<TSuccess, TError> result,
        Func<TSuccess, Result<TSuccess2, TError>> func)
    {
        if (result.IsSuccess)
            return func(result.Success) // auspacken (!) und func geben

        return result.Failure;
    }
})
```

Test stuff...

Code with default syntax highlighting in C#

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

// expression body syntax
public void Add(int a, int b) ⇒ a + b;
```

```
public async Task<int> AddAsync(int a, int b)
{
    var result = Task.FromResult(a + b);
    return result;
}
```

Code with additional syntax highlighting (i.e. focus on some parts of the code) requires pure HTML code

- no markdown
- escape everything

Example 1: custom styling on certain elements

```
// expression body syntax
public void Add(int a, int b) ⇒ a + b;
```

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

Example 2: styling per line using fragments

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