generic//de
the clean code company.

Mit Ideen der Funktionalen Programmierung

CleanCode erweitern

26.06.17

Über mich

Remy Loy

Kontakt

remy.loy@generic.de

+49 721 619096 47

blog.generic.de

Ziele von Clean Code Development

Lesbarkeit

Nachvollziehbarkeit

Testbarkeit

Wiederverwend-
barkeit

Evolvierbarkeit

Status quo

20 + 23 = 43

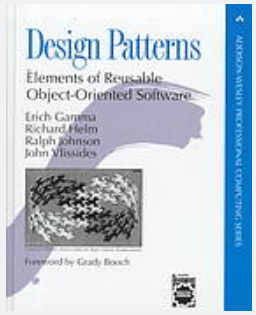
Werte


Evolvier-
barkeit
Korrektheit
Produktions-
effizienz
Kont. Ver-
besserung

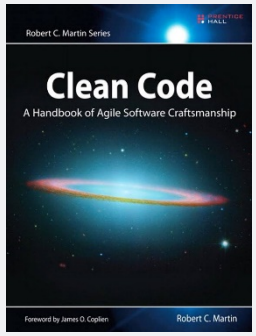
Funktionale Programmierung kann CleanCode bereichern

Woher kommen die Prinzipien?

Design Patterns: Elements of Reusable Object-Oriented Software



Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code



Programmiermodelle

Objekt-orientierte Programmierung (OOP)

Imperative Programmierung

Logische Programmierung

Funktionale Programmierung (FP)

CleanCode ist sprachübergreifend

Syntax, Monoide, Monaden, Morphismen

Ideen != Prinzipien

Kein null

NullReferenceException

```
public string Kapitälchen(string s)
{
    return char.ToUpper(s[0]) + s.Substring(1);
}
```

Option<T> (F#)

```fsharp
let kapitälchen (os : Option<string>) : Option<string> =
    match os with
    | None -> None
    | Some s -> Char.ToUpper s.[0] + s.Substring(1) |> Some
```

```fsharp
let kapitälchen : (Option<string>) -> Option<string> =
    Option.map (fun s -> Char.ToUpper s.[0] + s.Substring(1))
```

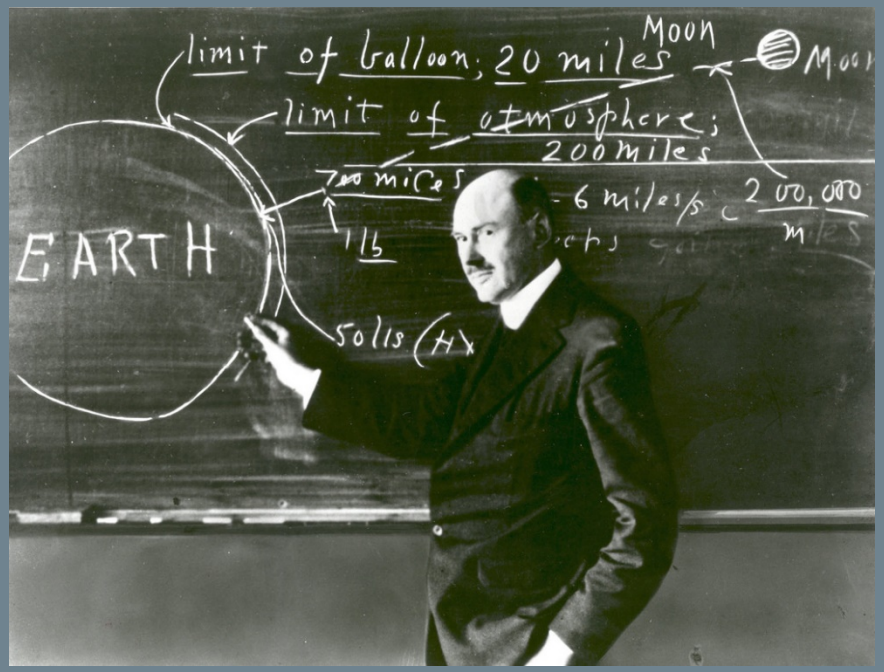
IOptionalString (C#)

```
public interface IOptionalString {}

public class None : IOptionalString {}

public class Some : IOptionalString
{
    public Some(string s)...

    public string S { get; }
}
```

IOptional<T> (C#)

```
public class Empty {}

public interface IOptional<T> {}

public class None : IOptional<Empty> {}

public class Some<T> : IOptional<T>
{
    public Some(T value)
    {
        Value = value;
    }

    public T Value { get; }
}
```

Kurzer, technischer Einschub: Catamorphism



Catamorphism (C# 7)

```
public T Cata<T>(
    IOptionalString os,
    Func<None, T> onNone,
    Func<Some, T> onSomeString)
{
    switch (os)
    {
        case None n:
            return onNone(n);
        case Some s:
            return onSomeString(s);
        default:
            throw new Exception("Unknown type");
    }
}
```

Ideen

- ✓ Mache die Optionalität von Daten explizit
- ✓ Mache dir Gedanken, welche Art von Polymorphie du brauchst
  - Über Verhalten
  - Über Daten
- ✓ Gut für…

Unveränderlichkeit

Klassen mit veränderlichen Eigenschaften

```
internal class MaterialManager
{
    public List<Material> Materials { get; set; }
}

internal class Material
{
    public string Label { get; set; }

    public int Units { get; set; }
}
```

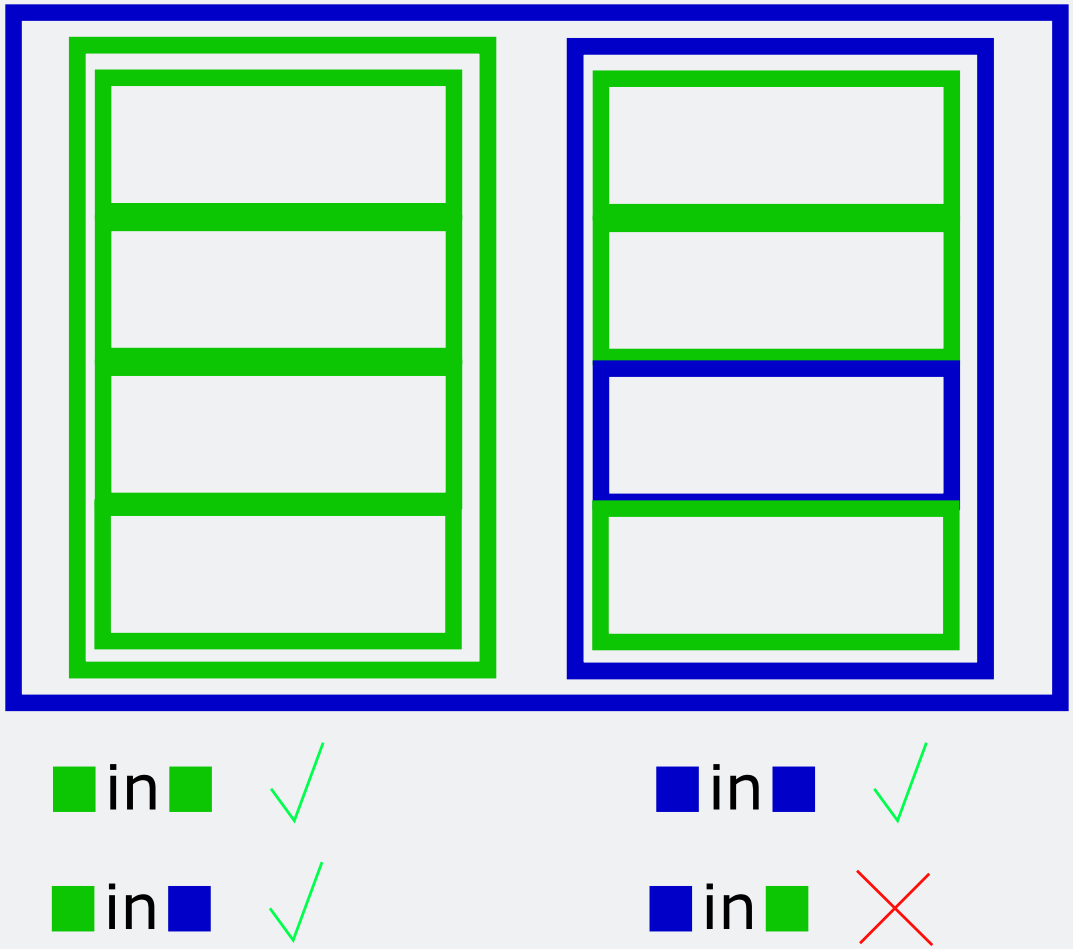
Unveränderliche Klassen

```
internal class MaterialManager
{
    public MaterialManager(IReadOnlyCollection<Material> materials)...

    public IReadOnlyCollection<Material> Materials { get; }
}

internal class Material
{
    public Material(string label, int units)...

    public string Label { get; }

    public int Units { get; }
}
```

Copy-on-write

```
internal class Material
{
    public Material(string label, int units)...

    public string Label { get; }

    public int Units { get; }

    public Material SetUnits(int units)
    {
        return new Material(Label, units);
    }
}
```

Hierarchie der Unveränderlichkeit



Ideen

- ✓ Mache deine Daten unveränderlich
  - Oder: Behandle deine Daten als unveränderlich
- ✓ Voraussetzung für weitere Ideen

- ✓ Gut für…

Reinheit

Seiteneffekte

Seiteneffekte

```
public int Add(int a, int b)
{
    _calculations++;
    Debug.WriteLine($"{_calculations} calculations");
    var sum = a + b;
    return sum < _max ? sum : _max;
}
```

Reinheit

Ohne Seiteneffekte

```
public static int Add(int a, int b, int max)
{
    var sum = a + b;
    return sum < max ? sum : max;
}
```

Ideen

- ✓ Mach dir Gedanken über die Seiteneffekte deiner Methoden
- ✓ Trenne Methoden mit Seiteneffekten von Methoden ohne Seiteneffekte
- ✓ Markiere Methoden ohne Seiteneffekte

- ✓ Gut für…

Typen-zentrisches Design

Modelliere dein Problem in Daten

FizzBuzz

```
type FizzBuzz = FizzBuzz | Fizz | Buzz
type MaybeFizzBuzz = Choice<FizzBuzz, int>

type TryMakeFizzBuzz = int -> MaybeFizzBuzz
type TryNext = MaybeFizzBuzz -> TryMakeFizzBuzz -> MaybeFizzBuzz
type ToString = MaybeFizzBuzz -> string
```

Modelliere dein Problem in Daten

Tankbuch

```
type PricePerUnit = float<euro/ liter>
type Fuel = Biodiesel | Diesel | LPG | Gasoline
type RefuelingKind =
  | FillTank of PricePerUnit
  | FillCanister of PricePerUnit
  | UseCanister
type Refueling =
  { Counter : float<km>
    Fuel : Fuel
    Amount : float<liter>
    Kind : RefuelingKind }
type Journey =
  { Start : DateTime
    Driver : string
    Distance : float<km>
    Refueling : list<Refueling> }
type JourneyLog = list<Journey>
type ReportExpenses = JourneyLog -> DateTime * TimeSpan -> Fuel -> float<euro>
type AveragePrice = JourneyLog -> Fuel -> float<euro / km>
```

Reduziere den Wertebereich

Vorname
NVARCHAR(50
)

```
public void SaveToDatabase(string vorname)
{
    if (vorname.Length > 50)
    {
        vorname = vorname.Substring(0, 50);
    }

    DB.Store(vorname);
}
```

Reduziere den Wertebereich

Vorname
NVARCHAR(50
)

```
type String50 = String50 of string

let create (s: string) : Option<String50> =
    if s.Length <= 50 then Some (String50 s)
    else None
```

```
let saveToDatabase (vorname : String50) =
    DB.store vorname
```

Ungültige Zustände vermeiden

```
public class Connection
{
    public bool IsConnected { get; }
    public string ClientAddress { get; }
    public DateTime? OpenedAt { get; }
    public DateTime? DisconnectedAt { get; }
    public int WrittenBytes { get; }
    ...
}
```

Ungültige Zustände vermeiden (F#)

```
type Connection =
    | Connecting
    | Connected of clientAddress: string * openedAt: DateTime * writtenBytes: int
    | Disconnected of disconnectedAt: DateTime
```

Ungültige Zustände vermeiden (C#)

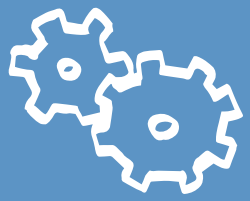
```
public interface IConnection {}

public class Connecting : IConnection {}

public class Connected : IConnection
{
    public string ClientAddress { get; }
    public DateTime OpenedAt { get; }
    public int WrittenBytes { get; }
    ...
}

public class Disconnected : IConnection
{
    public DateTime DisconnectedAt { get; }
    ...
}
```

Ideen

- ✓ Dein Model bestimmt dein Lösungsansatz
- ✓ Begrenze den erlaubten Wertebereich
- ✓ Mach ungültige Zustände nicht repräsentierbar im Model

- ✓ Gut für…

Produktions-effizienz

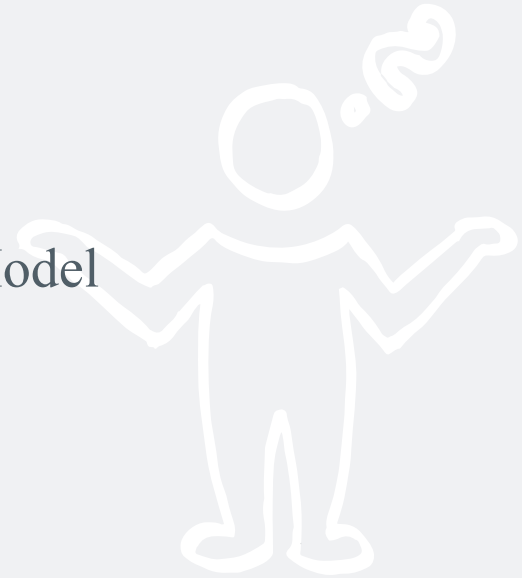


Korrektheit

Zusammenfassung

Gesammelte Ideen

- ✓ Mache die Optionalität von Daten explizit
- ✓ Mache dir Gedanken, welche Art von Polymorphie du brauchst
- ✓ Behandle deine Daten unveränderlich
- ✓ Trenne reine und unreine Methoden
- ✓ Dein Model bestimmt dein Lösungsansatz
- ✓ Begrenze den erlaubten Wertebereich
- ✓ Mach ungültige Zustände nicht repräsentierbar im Model



Schlussworte

„Wir brauchen nicht alles Bewährte über Bord zu werfen. Aber Erneuerung tut Not, schon um das Bewährte für die Zukunft zu sichern.“

Roman Herzog


Sitzt
Ihnen ihr
BAD CODE
im Nacken,
dann kontaktieren Sie uns...
www.generic.de
remy.loy@generic.de
blog.generic.de