

VALUE OBJECTS ON STEROIDS

Patrick Drechsler

#Dodnedder



27.09.2018



@drechsler



daptik

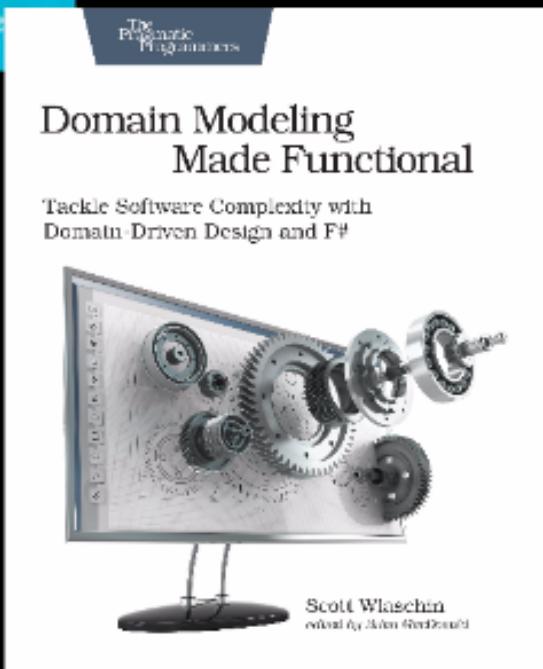
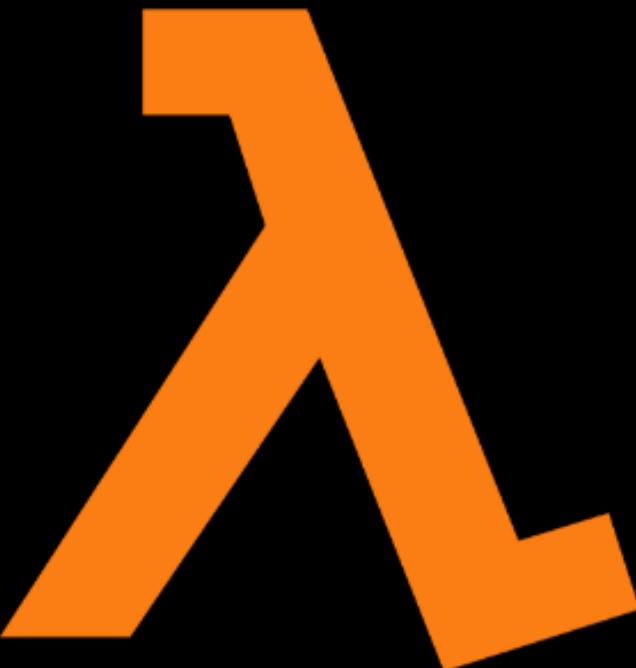
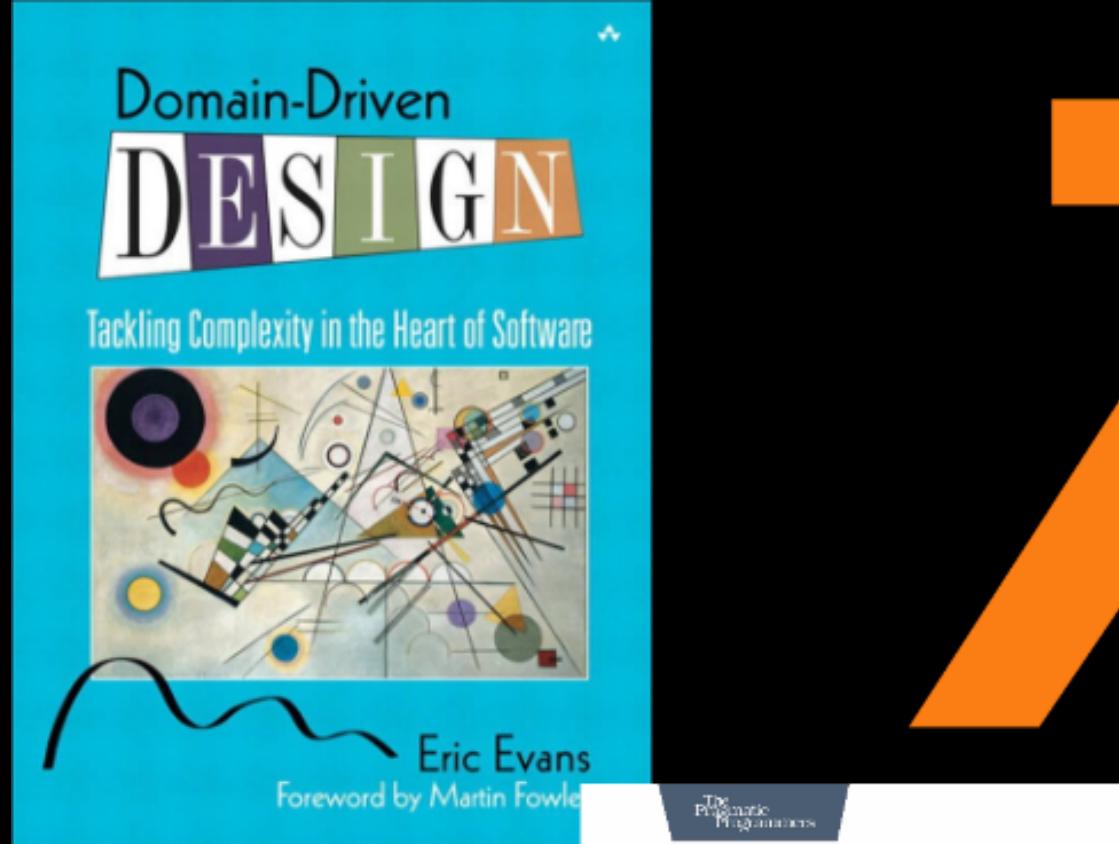
Patrick Drechsler

- C# Entwickler bei Redheads Ltd.
- aktuelle Schwerpunkte: DDD, FP, Cloud
- Softwerkskammer

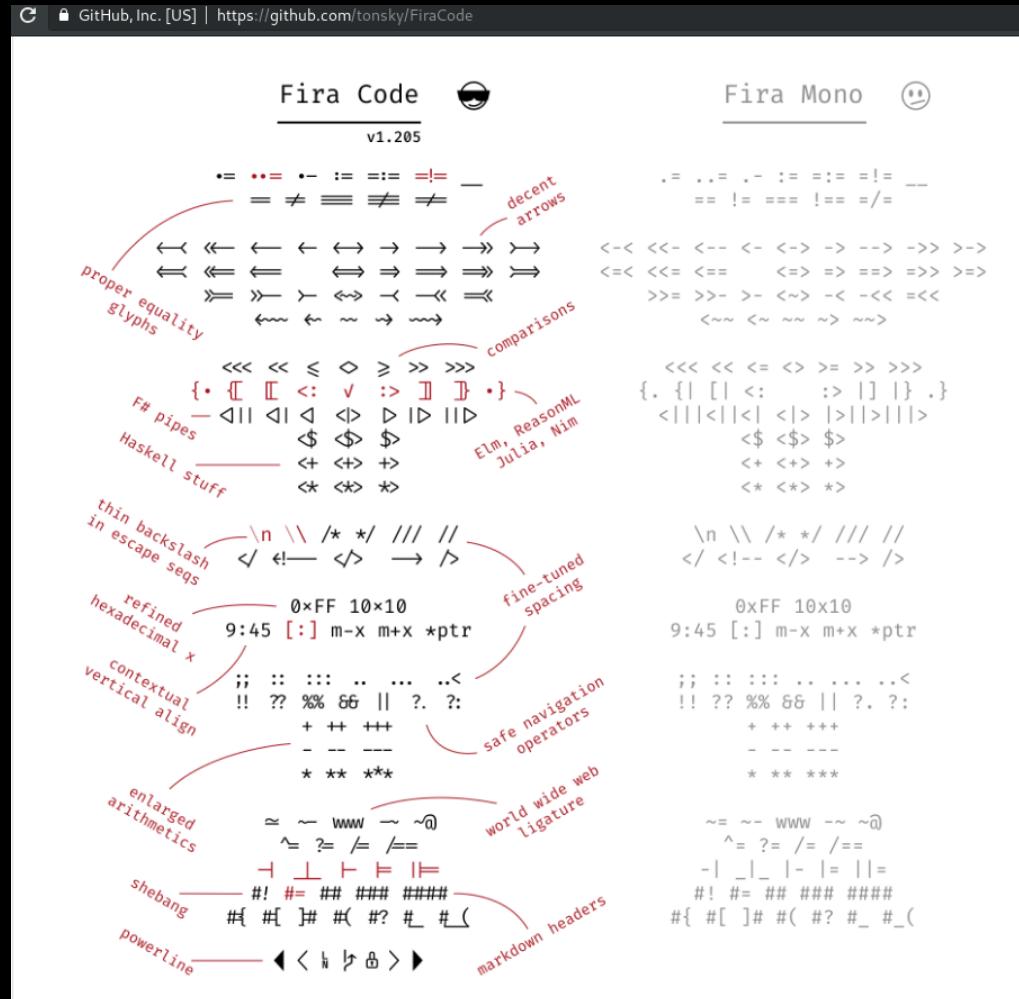
AGENDA

- Warum ?
- Wie ?
- Fallstricke
- On
Steroids





FiraCode



```
public class Konto
{
    public int Kontostand { get; private set; } = 0;

    public void Einzahlen(int geld)
    {
        Kontostand += geld;
    }
}
```

[Fact]

```
public void Kontostand_ist_nach_Einzahlung_groesser_als_davor()
{
    var konto = new Konto();
    var before = konto.Kontostand;
    konto.Einzahlen(-10); // <----- AUTSCH!
    konto.Kontostand.Should().BeGreaterThan(before);
}
```

- Problem: **Signatur lügt!** (Geld ist kein Integer)
- Willkommen in der Welt der 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>

```
public class Konto
{
    public Geld Kontostand { get; private set; } = new Geld(0);

    public void Einzahlen(Geld geld)
    {
        Kontostand = new Geld(Kontostand.Betrag + geld.Betrag);
    }
}
```

Geld Klasse existiert noch nicht...

```
public class Geld
{
    // Betrag kann nur ueber Konstruktor veraendert werden
    public int Betrag { get; }                                // ← 1

    public Geld(int betrag)                                  // ← 2
    {
        if (!IsValid(betrag))                               // ← 3
        {
            throw new InvalidGeldException(betrag.ToString());
        }

        this.Betrag = betrag;                                // ← 2
    }

    private bool IsValid(int betrag) => betrag ≥ 0; // ← 3'
}
```

- Es kann nur gültige Geld-Objekte geben
- Wert kann nicht verändert werden

IMMUTABILITY

- einfacher zu Erstellen & Testen
- keine Seiteneffekte
- keine Null References

```
[Fact]
public void Geld_schmeisst_wenn_Betrag_zu_gross()
{
    var max = Int32.MaxValue;

    Action action = () => new Geld(max + 1);

    action.Should().Throw<InvalidGeldException>();
}
```

- kein ungültiges Geld Objekt!
- vernünftige Exception!
- (bessere Implementierung folgt)

Vorschau: mehr Verhalten in die Geld-Klasse packen

```
public class Geld
{
    public int Betrag { get; }

    // ...

    public Geld Add(Geld geld)
    {
        try
        {
            return new Geld(this.Betrag + geld.Betrag);
        }
        catch
        {
            throw new InvalidGeldException(
                $"Ups. Can't add {geld.Betrag} to {Betrag}!");
        }
    }
}
```

```
[Fact]
public void Geld_laesst_sich_addieren()
{
    var geld1 = new Geld(1);
    var geld2 = new Geld(10);

    var geld3 = geld1.Add(geld2); // geld1, geld2 werden nicht verändert
    geld3.Betrag.Should().Be(11);
}
```

Geld ist mehr als nur **Betrag**



≠



≠



Fügen wir **Währung** zur Klasse **Geld** hinzu...

```
public class Geld
{
    public int Betrag { get; }
    public Waehrung Waehrung { get; } // ----- NEU

    public Geld(int betrag, Waehrung waehrung)
    {
        if (!IsValid(betrag, waehrung))
        {
            throw new InvalidGeldException();
        }
        this.Betrag = betrag;
        this.Waehrung = waehrung; // ----- NEU
    }

    private bool IsValid(int betrag, Waehrung waehrung)
        ⇒ betrag ≥ 0 &&
           waehrung ≠ Waehrung.Undefined;
}
```

```
[Fact]
public void Ein_Euro_ist_gleich_Ein_Euro()
{
    var geld1 = new Geld(1, Waehrung.EUR);
    var geld2 = new Geld(1, Waehrung.EUR);

    geld1.Should().BeEqualTo(geld2); // ← Fails! :-(

}
```

AUTSCH!

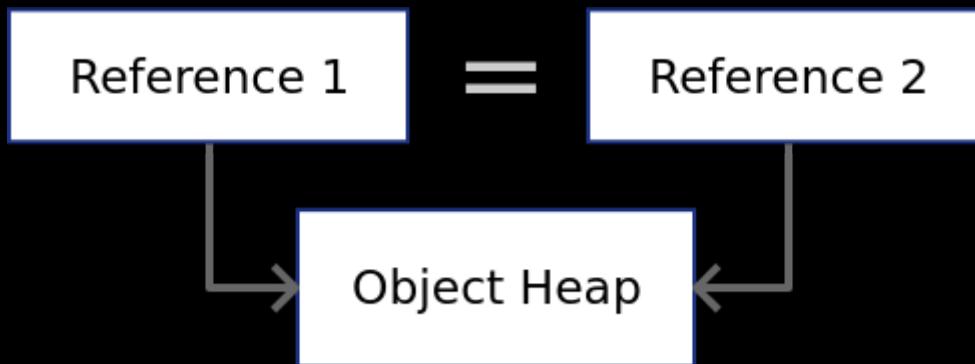
Da müssen wir was machen

EXKURS: VERGLEICHBARKEIT

"Equal"



EQUALITY BY REFERENCE



"Geld gleich Geld" 👎

EQUALITY BY IDENTIFIER

Object 1

- Id: 1
- Name: foo

=

Object 2

- Id: 1
- Name: bar

"Geld gleich Geld" 

EQUALITY BY STRUCTURE

Object 1

- Street: foo
- Zipcode: foo

=

Object 2

- Street: foo
- Zipcode: foo

"Geld gleich Geld" 

```
public class Geld
{
    // ...
    public override bool Equals(Geld other)
    {
        return
            other.Betrag == this.Betrag &&
            other.Waehrung == this.Waehrung;
    }

    public override int GetHashCode()
    {
        // ...
    }
}
```

```
[Fact]
public void Ein_Euro_ist_gleich_Ein_Euro()
{
    var geld1 = new Geld(1, Waehrung.EUR);
    var geld2 = new Geld(1, Waehrung.EUR);
    geld1.Should().BeEqual(geld2); // ← green :-)
}
```

"GELD" IST JETZT STABIL

- "Geld" ist **nicht** im nachhinein
änderbar
- "Geld" ist **vergleichbar**

und ganz nebenbei haben wir zu Fuß ein

Value Object

erstellt

VALUE OBJECT

- "**Expressiveness**" Methodensignaturen lügen nicht
- "**Immutability**"
- "**Equality by structure**"
- "**Encapsulation**" Logik ist da wo sie hingehört

DDD JARGON

- **Entity** Objekt mit Lebenszyklus (Identität)
 - z.B. Kunde
- **Value Object** Unveränderliches Objekt
 - z.B. Geld, Adresse, Email, ...

Entscheidung ist immer kontextabhängig!

IMPLEMENTING YOUR OWN VALUE OBJECT

```
public abstract class ValueObject<T> where T : ValueObject<T>
{
    public override bool Equals(object other) { // ← 1
        return Equals(other as T);
    }

    protected abstract IEnumerable
        GetAttributesToIncludeInEqualityCheck(); // ← 2

    public bool Equals(T other) { // ← 3
        if (other == null) return false;

        return GetAttributesToIncludeInEqualityCheck()
            .SequenceEqual(
                other.GetAttributesToIncludeInEqualityCheck());
    }

    public override int GetHashCode() { // ← 4
        var hash = 17;
        foreach (var obj in GetAttributesToIncludeInEqualityCheck())
            hash = hash * 31 + (obj == null ? 0 : obj.GetHashCode());

        return hash;
    }
}
```



```
public class Email : ValueObject<Email> // ← provide structural equality
{
    public string Value { get; }

    public Email(string input)
    {
        if (!IsValid(input)) {
            throw new InvalidEmailException(input)
        }
        Value = input;
    }

    // TODO: pimp validation :-)
    private bool IsValid(string input) => true;

    // provide structural equality
    protected override IEnumerable<object>
        GetAttributesToIncludeInEqualityCheck()
    {
        return new List {Value};
    }
}
```

SUGAR: OPERATOR OVERLOADING

```
public abstract class ValueObject<T> where T : ValueObject<T>
{
    public static bool operator ==(ValueObject<T> left,
                                    ValueObject<T> right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(ValueObject<T> left,
                                    ValueObject<T> right)
    {
        return !(left == right);
    }
}
```

MORE SUGAR: IMPLICIT OPERATOR

```
var email = new Email("foo@bar.baz");
string s = email.Value;           // ← ".Value" nervt auf Dauer ...
```

```
public class Email : ValueObject<Email>
{
    public string Value { get; }

    // Output: string
    public static implicit operator string(Email mail)
    {
        return mail.Value.ToString();
    }
}
```

```
var email = new Email("foo@bar.baz");
string s = email;                // ← einfacher
```

OPTIONALE VALUE OBJECTS

kleiner FP Plug

```
public class BahnKunde
{
    // Optional
    public BonusPunkte BonusPunkte { get; } = new BonusPunkte(null);
}
```

```
public class BonusPunkte : ValueObject<BonusPunkte>
{
    public int Punkte { get; } = 0;

    // ...
    private bool IsValid(int? punkte)
    {
        return punkte == null
            ? true
            : punkte > 0;
    }
}
```

C# kennt keine optionale Datentypen

- Wer kennt C# noch bevor LINQ?

```
public class BonusPunkte : ValueObject<BonusPunkte> {  
    public int Punkte { get; } = 0;  
  
    private bool IsValid(int? punkte) {  
        return punkte = null ? true : punkte > 0;  
    }  
}
```

- Vorteil: Signaturen bleiben "ehrlich"
- Nachteil: lästige NullChecks
 - Abhilfe: **Maybe** oder **Option** Datentyp verwenden
 - noch kein C# Sprachfeature, aber Bibliotheken vorhanden

```
public Maybe<BonusPunkte> Punkte { get; }
```

ERWEITERBAR...

```
public class CompanyEmail
{
    public CompanyEmail(Email mail) // ← "Email" ist ein Value Object
    {
        if (!IsValid(mail))
        {
            throw new InvalidCompanyEmailException(mail.Value);
        }

        Value = mail;
    }

    public Email Value { get; }

    private bool IsValid(Email mail) ⇒
        mail.Value.EndsWith("companynname.com");
}
```

"Composition over Inheritance"

FAQ

- Ist das nicht schlecht für die Performance?
 - Ja, aber...
- Wann sollte man statt eines Basistyps ein VO einsetzen?
 - Sobald Geschäftslogik im Spiel ist (z.B. Validierung)
- Funktionieren VOs auch mit meinem OR-Mapper?
 - Ja

FALLSTRICKE

- ORM
- Collections

OR-MAPPER

Entity Framework (EF) und NHibernate können mit VO's umgehen.

Bsp.: EF mit **ComplexType**

| | | Email VO | Address VO | | |
|------------|------|-------------|----------------|--------------|-----------------|
| CustomerId | Name | Email_Value | Address_Street | Address_City | Address_ZipCode |
| 1 | Foo | foo@test.de | Hauptstr. 1 | Berlin | 12345 |
| 2 | Bar | bar@test.de | Hauptstr. 2 | Berlin | 12345 |
| 3 | Baz | baz@test.de | Hauptstr. 3 | Berlin | 12345 |

Tip: Wenn möglich die Domänenlogik von der ORM Logik entkoppeln

COLLECTIONS

- Können im ORM Umfeld problematisch sein
- Wenn doch:
 - Umdenken oder
 - Serialisieren

ON STEROIDS

Dan Bergh Johnsson: The Power of Value - Power Use
of Value Objects in Domain Driven Design

<https://vimeo.com/13549100>

NICHT NUR FÜR ENTITÄTEN NÜTZLICH



```
public class TodoRepository
{
    private IEnumerable<Todo> Todos { get; }

    public IEnumerable<Todo> GetTodosBetween(DateTime from, DateTime to)
    {
        if (from <= to)
        {
            throw new InvalidDateRangeException();
        }

        return Todos.Where(x =>
            x.ErstelltAm >= from && x.ErstelltAm <= to);
    }
}
```

Fachliches Konzept **Zeitspanne** in Objekt kapseln:

```
public class Zeitspanne
{
    public DateTime Von { get; }
    public DateTime Bis { get; }

    public Zeitspanne(DateTime von, DateTime bis)
    {
        if (!IsValid(von, bis))
        {
            throw new InvalidDateRangeException();
        }

        Von = von;
        Bis = bis;
    }

    private bool IsValid(DateTime von, DateTime bis) => von < bis;
}
```

noch **mehr Fachlichkeit** ins Objekt packen

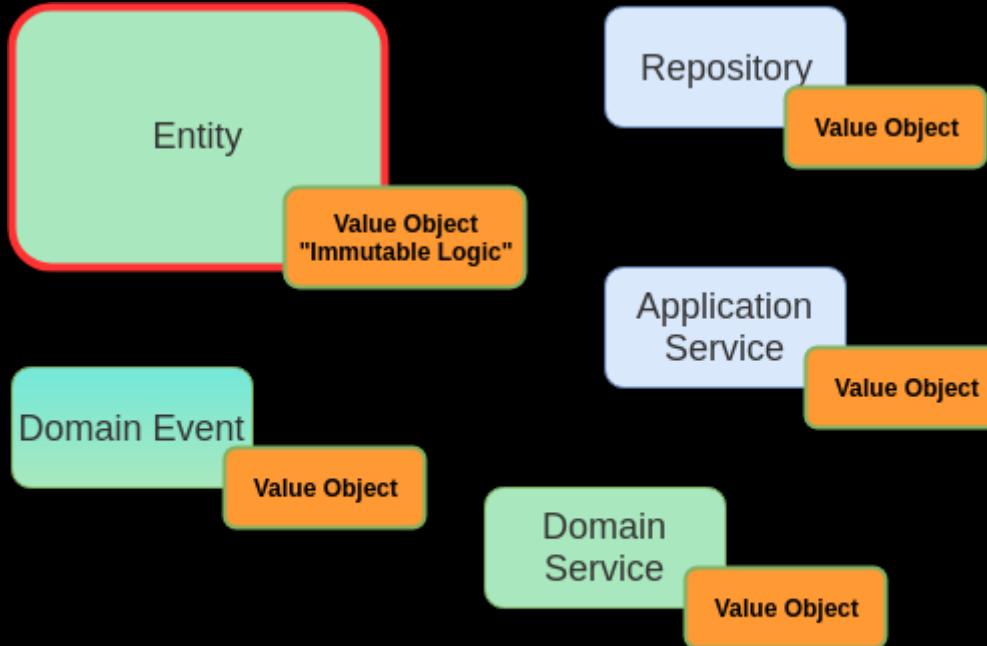
```
public class Zeitspanne
{
    public bool Umfasst(DateTime d) => d >= Von && d <= Bis;
    // ...
}
```

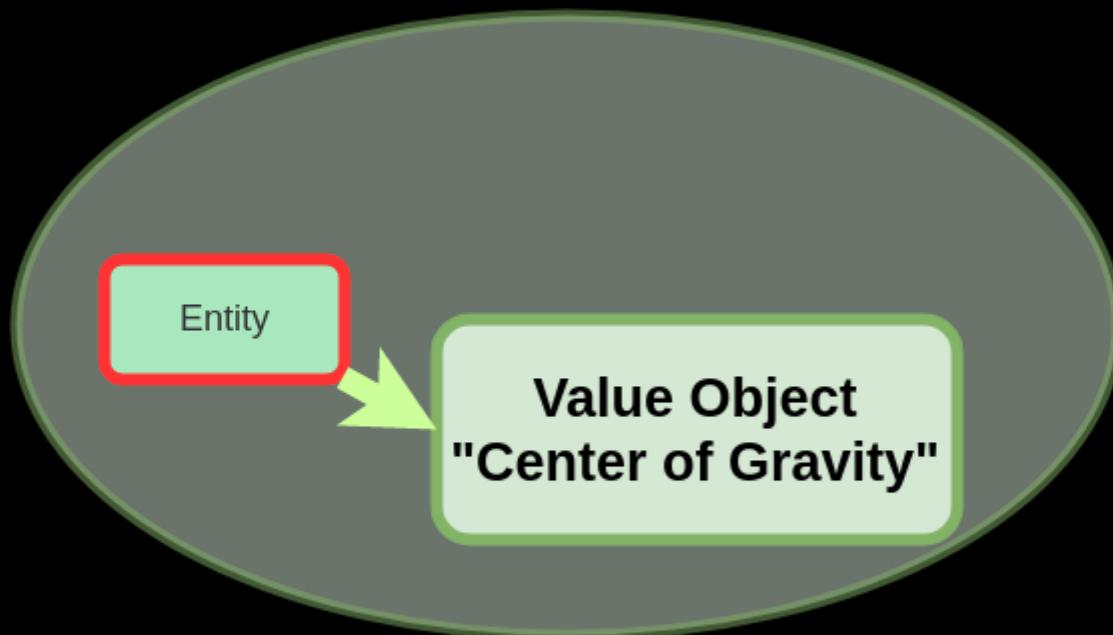
```
public class TodoRepository
{
    public IEnumerable<Todo> GetTodosBetween(DateTime from, DateTime to)
    {
        if (from <= to) {
            throw new InvalidDateRangeException();
        }

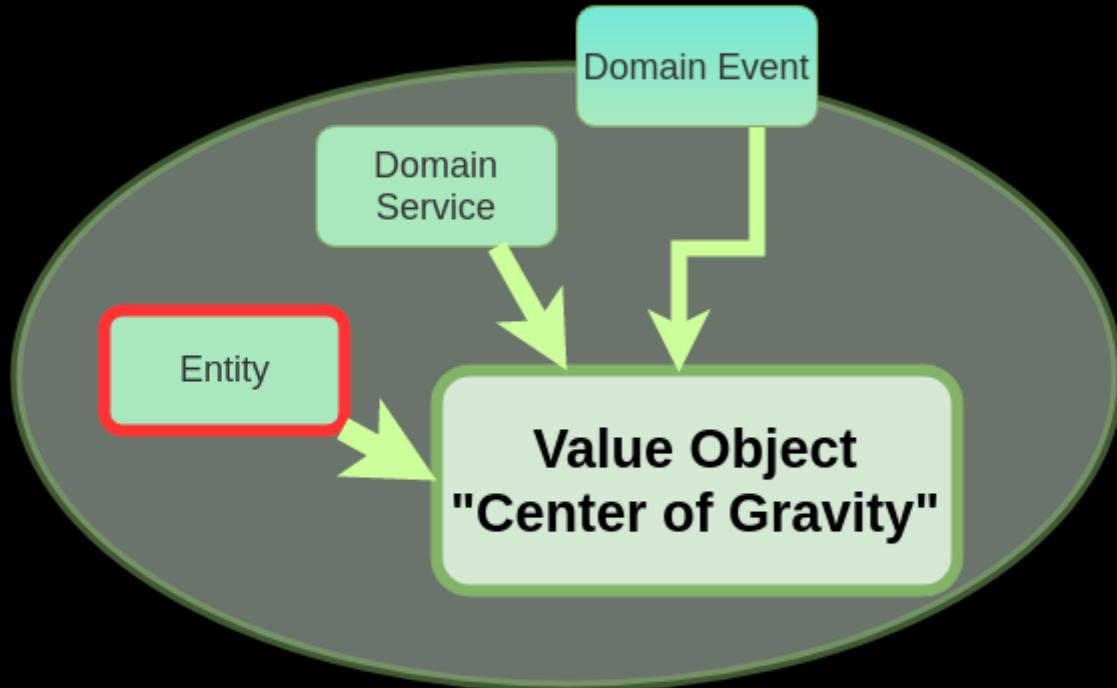
        return Todos.Where(x =>
            x.ErstelltAm >= from && x.ErstelltAm <= to);
    }
}
```

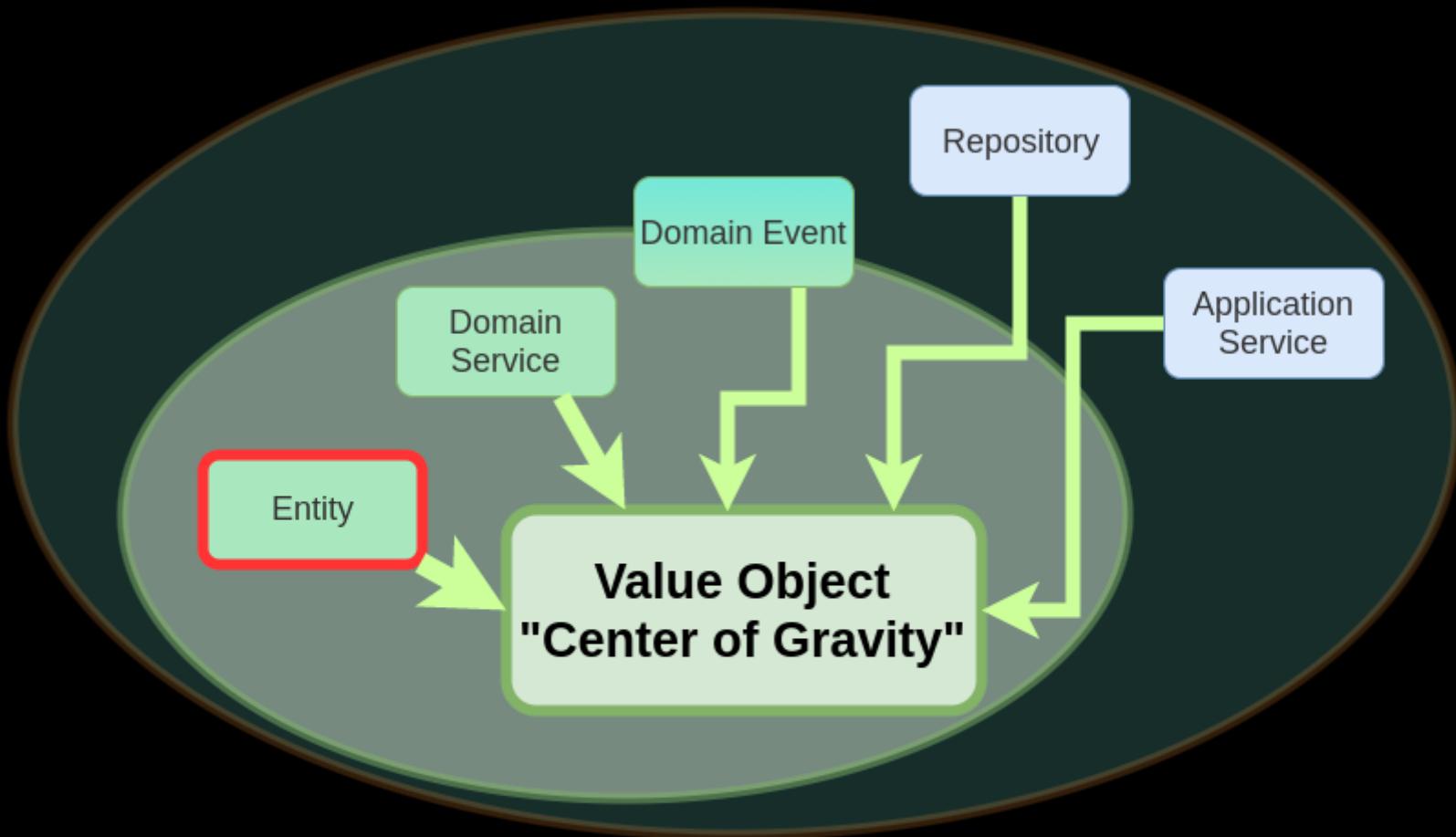
VS

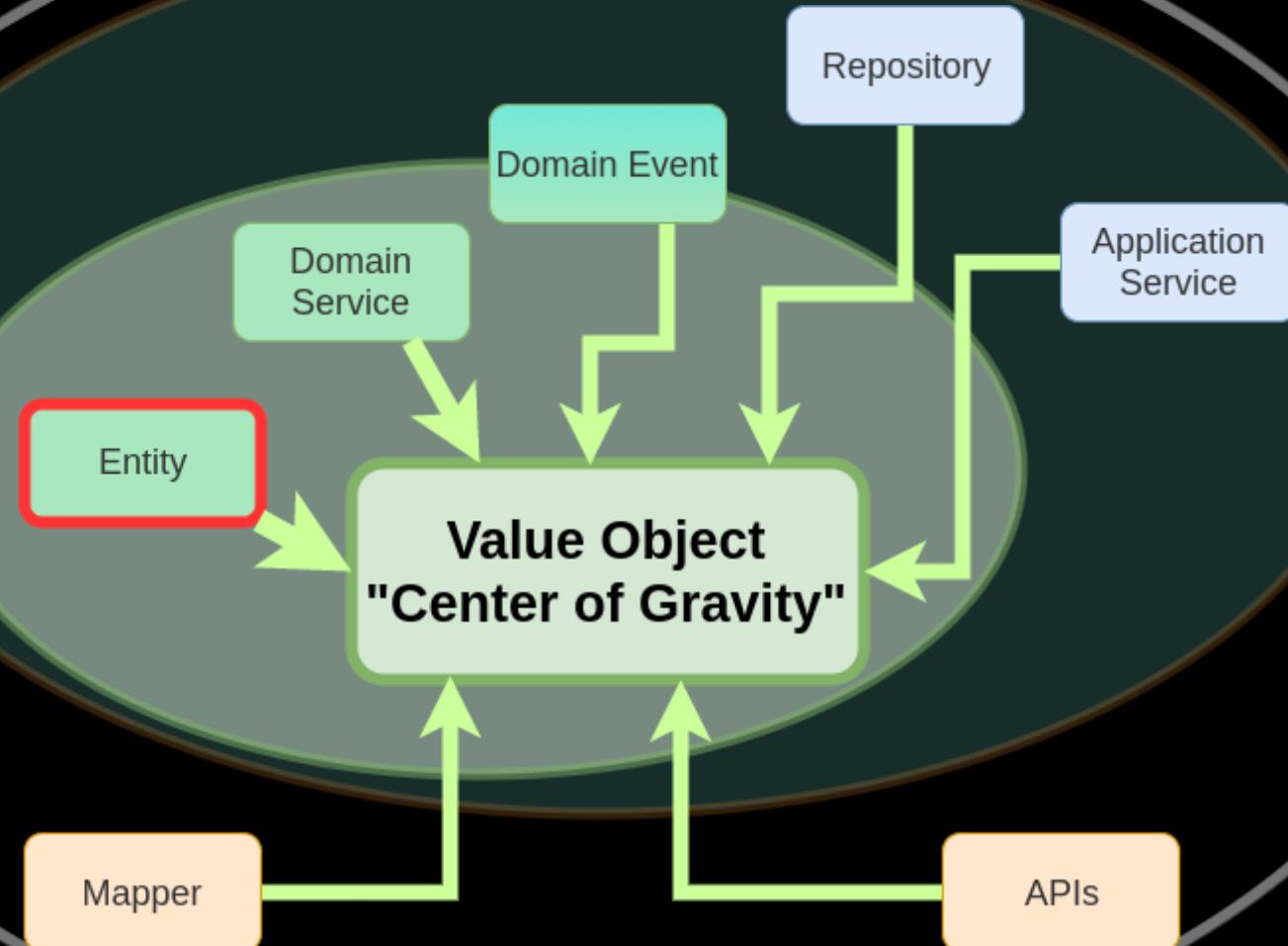
```
public class TodoRepository
{
    public IEnumerable<Todo> GetTodosInnerhalbVon(Zeitspanne zeitspanne)
        => Todos.Where(x => zeitspanne.Umfasst(x.ErstelltAm));
}
```











DEMOS

Money, DateRange, Mail

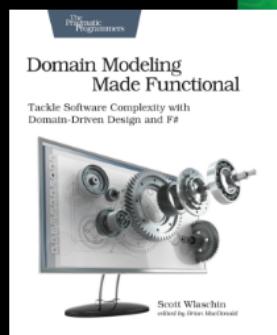
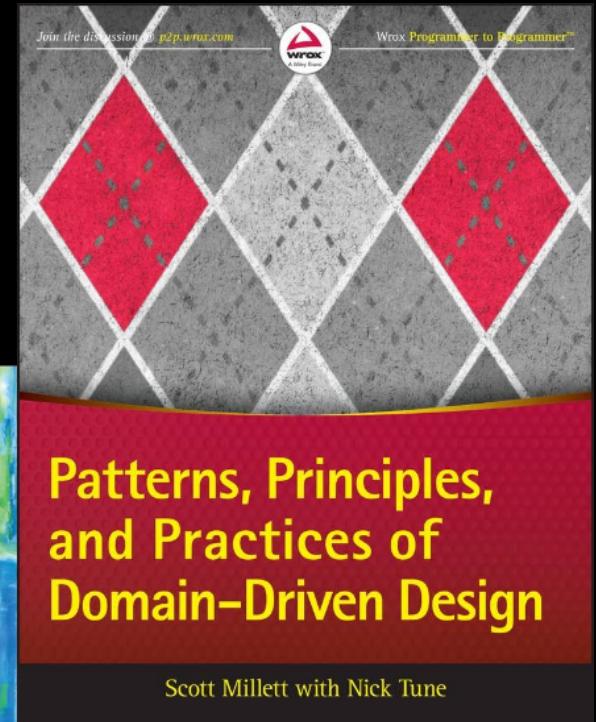
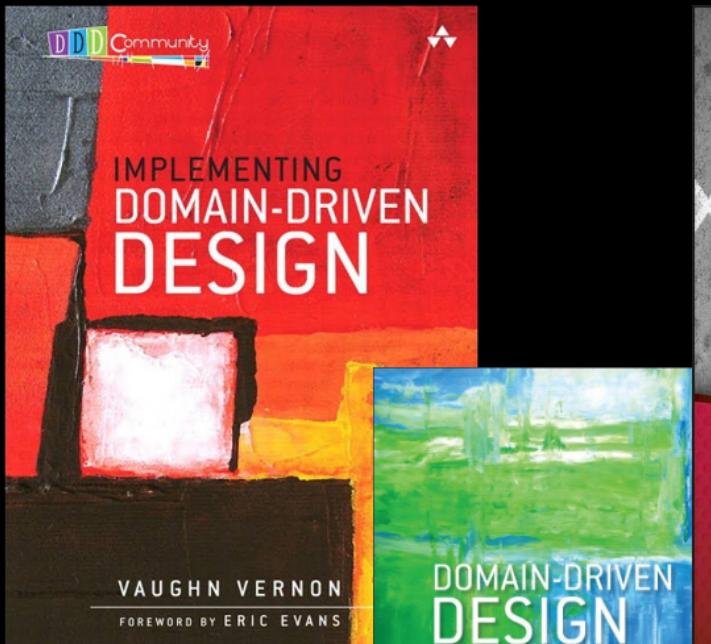
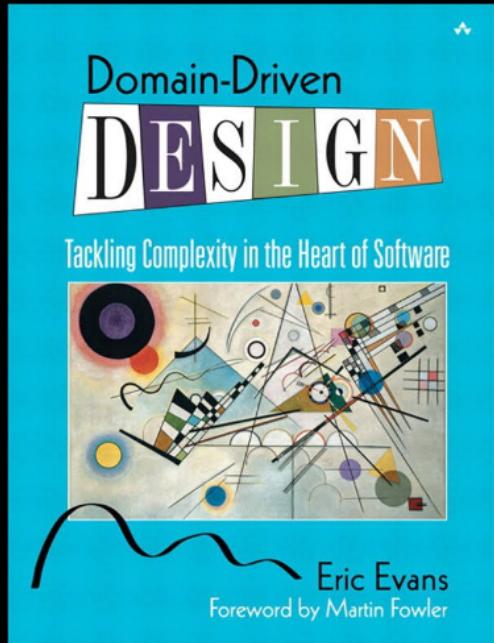
MERKZETTEL

- Immutability
 - keinen parameterlosen Konstruktor
 - keine "setter"
 - Methoden dürfen nie den Zustand ändern
- Vergleichbarkeit
 - Equals / Hashcode überschreiben

VALUE OBJECT

- **Expressiveness** "ehrliche" Methodensignaturen
- **Immutability**
- **Equality by structure**
- **Encapsulation** Logik ist da wo sie hingehört

DOMAIN DRIVEN DESIGN



Weniger Boilerplate mit F# "Record Types"

```
type Geld = {  
    betrag: int  
    waehrung: Waehrung  
}
```

- Out of the box:
 - ...Immutability
 - ...Structural equality

ZUSAMMENFASSUNG

- Value Object:
 - immer dann, wenn Basistyp und Businesslogik aufeinandertreffen
- Vorteil: Kleine Einheit (immutable)
 - → **verständlich**
 - → **weniger denken**
 - → **einfach testbar**

DANKE!

-  @drechsler
 -  github.com/draftik
 - 
- patrick.drechsler@redheads.de