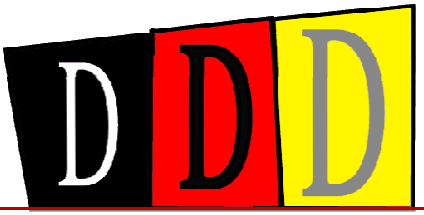
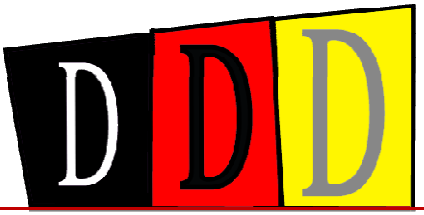


Functional Domain Driven Design

Marco Heimeshoff



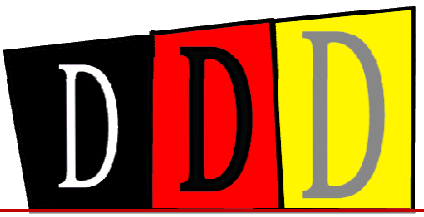
What is **DDD?**

DDD is a cul

Learning

Empathy

Language

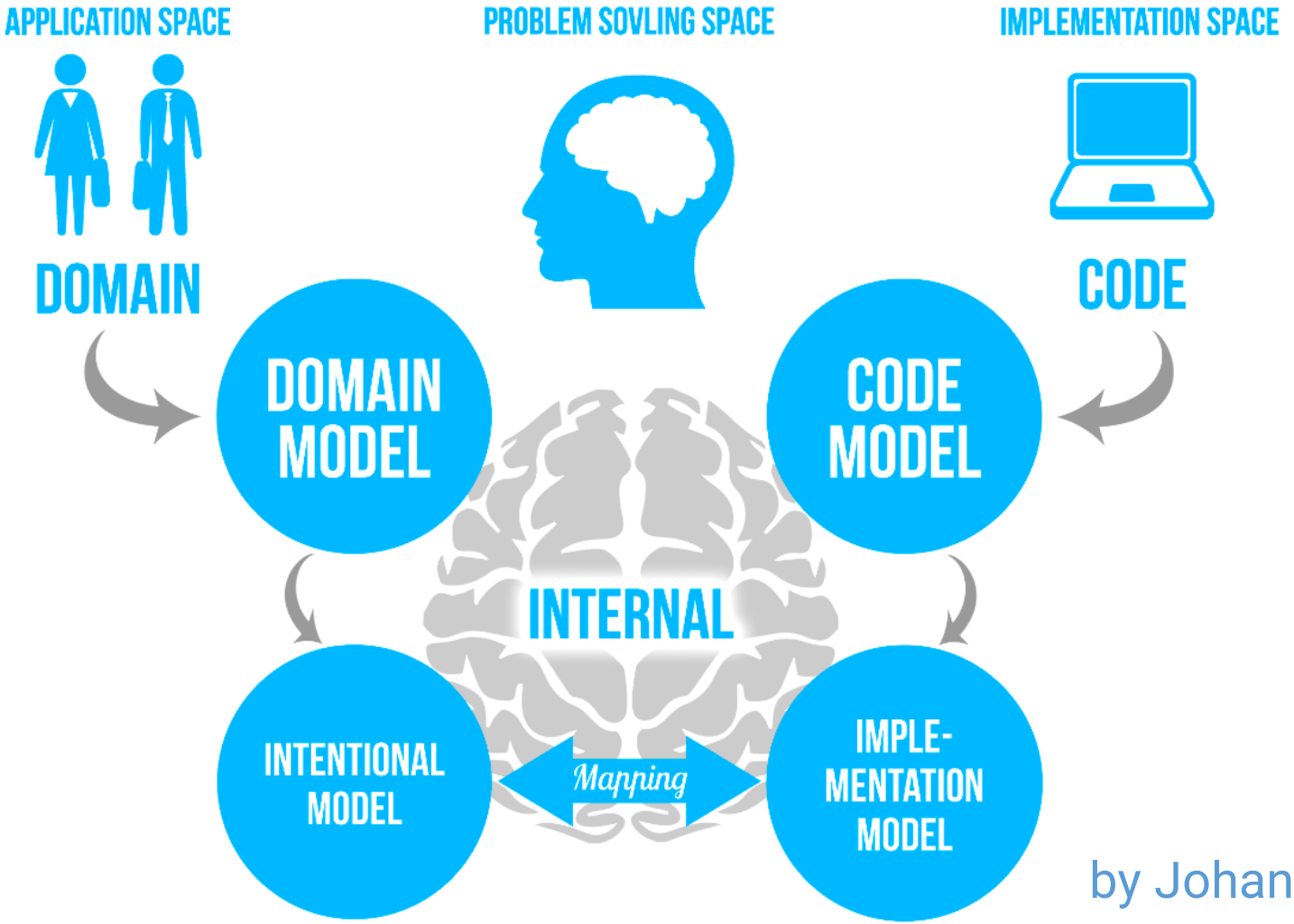


Language matters

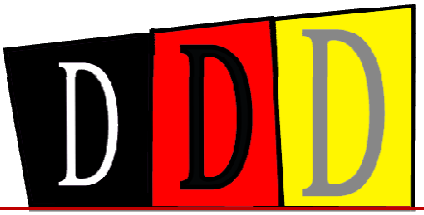
@Heimeshoff

Mapping is hard

@Heimeshoff



by Johannes Hofmeister



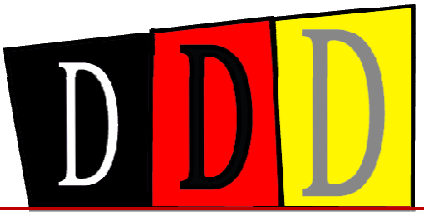
Low Hanging Fruit

Better Semantics

Domain Model

Architecture

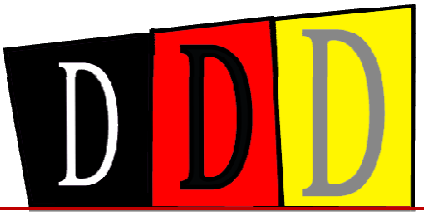
The Fantastic Four



Value objects

“An immutable object, like money or a date range, whose equality isn't based on identity. In general equality is based on all fields equality.”

- Martin Fowler



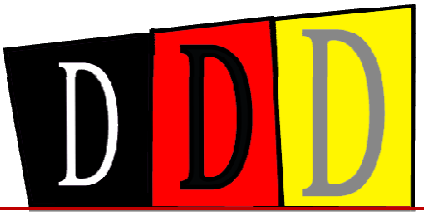
Value objects

Beschreibt und bemisst in der ubiquitären Sprache

Datum, Zeit

Kundenname: Vor-, Mittel-, Nachname

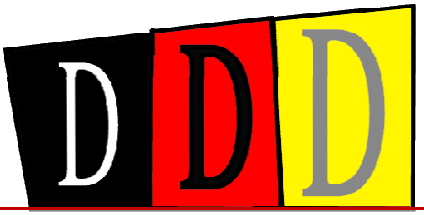
Währung, Farbe, Telefonnummer, Adresse, ...



Value objects in C#

@Heimeshoff

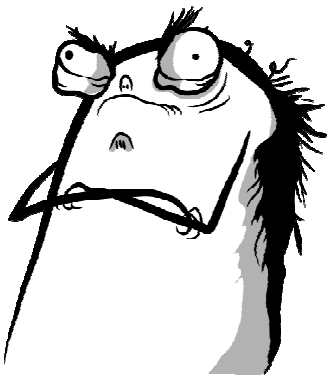
```
public class CompanyProfile
{
   public String BusinessName { get; set; }
   public String TaxCode { get; set; }
   public String VatNumber { get; set; }

   public String AssignedBank { get; set; }
   public Boolean IsBankAuthorized { get; set; }
}
```

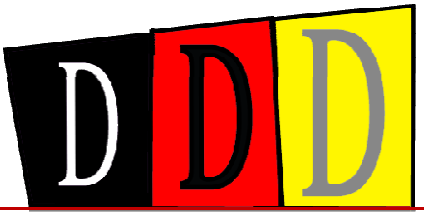


Value objects in C#

@Heimeshoff

```
public class CompanyProfile
{
    public String BusinessName { get; private set; }
    public String TaxCode { get; private set; }
    public String VatNumber { get; private set; }

    public CompanyProfile(String businessName, String taxCode, String vatNumber=null)
    {
        // check if parameters are valid
        BusinessName = businessName;
        TaxCode = taxCode;
        VatNumber = vatNumber;
    }
}
```



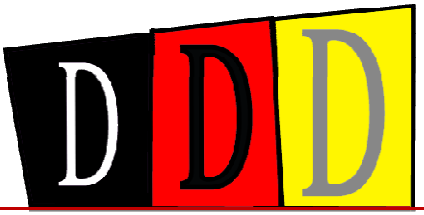


Value objects in C#

@Heimeshoff

```
public class CompanyProfile
{
    public String BusinessName { get; private set; }
    public String TaxCode { get; private set; }
    public String VatNumber { get; private set; }

    public CompanyProfile(String businessName, String taxCode, String vatNumber=null)
    {
        // check if parameters are valid
        BusinessName = businessName;
        TaxCode = taxCode;
        VatNumber = vatNumber;
    }
}
```

```
public override Boolean Equals(Object other)
{
    var target = other as CompanyProfile;
    return target == null ? false :
           target.BusinessName == this.Bu
           && target.TaxCode == this.TaxC
           && target.VatCode == this.VatC
}
```



Value objects in C#

@Heimeshoff

```
public class CompanyProfile
{
    public String BusinessName { get; private set; }
    public String TaxCode { get; private set; }
    public String VatNumber { get; private set; }

    public CompanyProfile(String businessName, String taxCode, String vatNumber=null)
    {
        // check if parameters are valid
        BusinessName = businessName;
        TaxCode = taxCode;
        VatNumber = vatNumber;
    }
}
```

```
public override Int32 GetHashCode()
{
        // ...
}
```

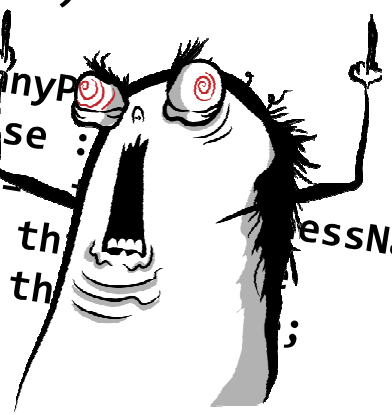
```
public override Boolean Equals(Object other)
{
var target = other as CompanyProfile;
return target == null ? false :
       target.BusinessName == this.BusinessName
       && target.TaxCode == this.TaxCode
       && target.VatCode == this.VatCode;
}
```

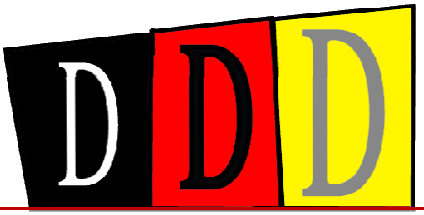
Value objects in C#

@Heimeshoff

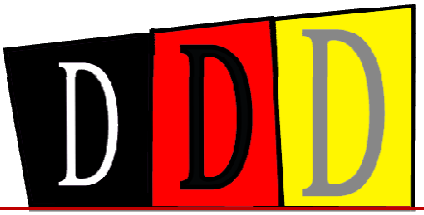
```
public class CompanyProfile
{
    public String BusinessName { get; private set; }
    public String TaxCode { get; private set; }
    public String VatNumber { get; private set; }

    public CompanyProfile(String businessName, String taxCode, String vatNumber=null)
    {
        // check if parameters are valid
        BusinessName = businessName;
        TaxCode = taxCode;
        VatNumber = vatNumber;
    }
}
```

```
public override Int32 GetHashCode()
{
    // ...
}

public override Boolean Equals(Object other)
{
    var target = other as CompanyP
    return target == null ? false :
        target.BusinessName
        && target.TaxCode == th
        && target.VatCode == th
}
```





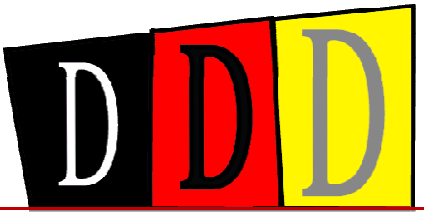
Value types in F#



Value types in F#

@Heimeshoff

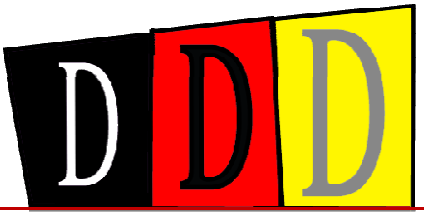
```
type CompanyProfile = {
        BusinessName : string
        Tax_Code     : string
        VatNumber    : string
}
```



Value types in F#

@Heimeshoff

```
type CompanyProfile = {
        BusinessName : string
        Tax_Code     : string
        VatNumber    : string option
}

let profile = {
  BusinessName = “Heimeshoff IT”
  Tax_Code     = “1234567890”
}
```



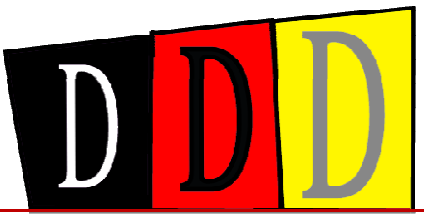
Low Hanging Fruit
Better Semantics
Domain Model
Architecture
The Fantastic Four

DDD

F# is very nice

@Heimeshoff

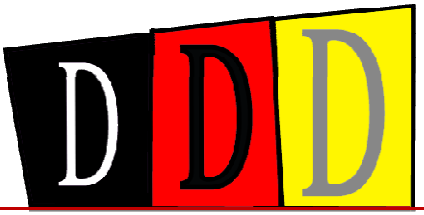
Turning Signal-Noise into Signal-Nice



Option type

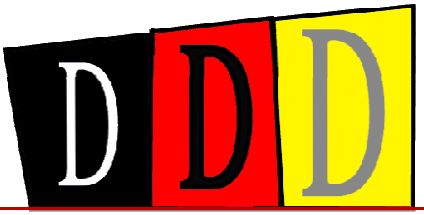
@Heimeshoff

```
let div x y =
    if y = 0 then None
    else Some(x/y)

x:int -> y:int -> int option
```



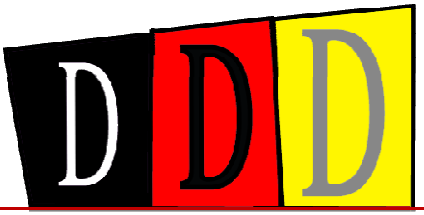
Pattern matching


```
let div x y =
    if y = 0 then None
    else Some(x/y)
```



Pattern matching


```
let div x y =
    if y = 0 then None
    else Some(x/y)

let div x y =
    match y with
        | 0 -> None
        | _ -> Some(x/y)
```



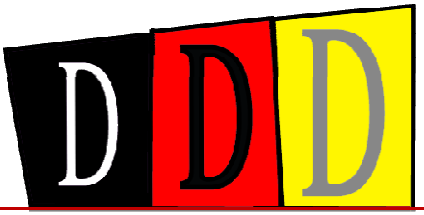
Pattern matching

@Heimeshoff

```
let div x y =
    if y = 0 then None
    else Some(x/y)
```

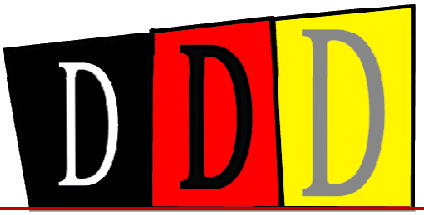
```
let div x y =
    match y with
        | 0 -> None
        | 1 -> Some(x/y)
```

>> Incomplete pattern matches on this expression.



Pattern matching

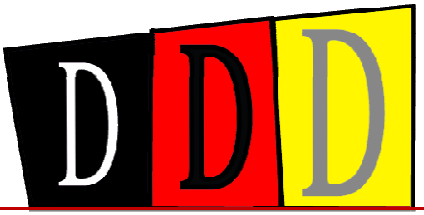

```
let div x y =
    if y = 0 then None
    else Some(x/y)

let div x y =
    match y with
        | 0 -> None
        | 1 -> Some(x)
        | _ -> Some(x/y)
```



Discriminated unions

@Heimeshoff

```
type Automarke =
    | Audi
    | BMW
    | Delorean
```

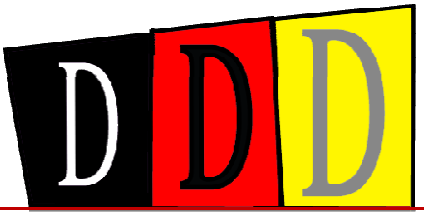


Discriminated unions

@Heimeshoff

```
type Automarke =
    | Audi
    | BMW
    | Delorean
```

```
type Lack =
    | DemonRed
    | DeepBlue
    | BoringGray
```



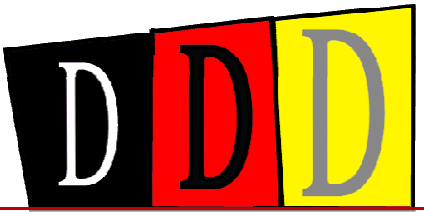
Record

@Heimeshoff

```
type Automarke =
      | Audi
      | BMW
      | Delorean
```

```
type Lack =
      | DemonRed
      | DeepBlue
      | BoringGray
```

```
type Auto = {
   Marke : Automarke
   Farbe : Lack }
```



Immutability

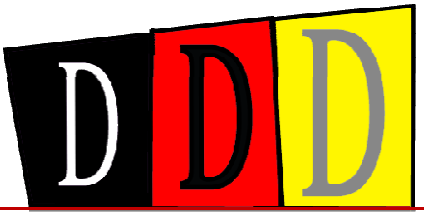
@Heimeshoff

```
type Automarke =
    | Audi
    | BMW
    | Delorean
```

```
type Lack =
    | DemonRed
    | DeepBlue
    | BoringGray
```

```
type Auto = {
   Marke : Automarke
   Farbe : Lack }
```

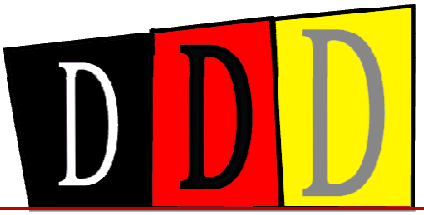
```
let mein_Auto = { Marke = Audi; Farbe = BoringGray }

let pimped = { mein_Auto with Farbe = DemonRed}
```



Type inference


```
let multiply x y = x * y
multiply 2 2
```

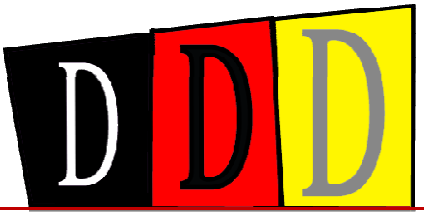
x:int -> y:int -> int



Partial application

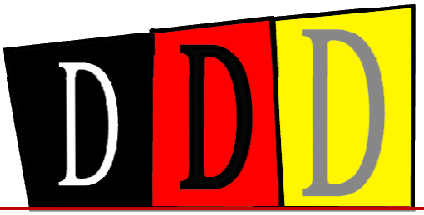
@Heimeshoff

```
let multiply x y = x * y
x:int -> y:int -> int

let umsatzsteuer x = multiply 19
x:int -> int

let steuern = umsatzsteuer 200
(steuern = 38)
```



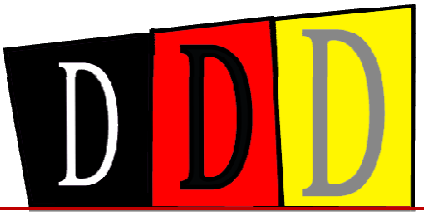
Pipeline operator


```
let versende Waren =
 an_Post(gruppieren(verpacken(sortieren(Waren))))
```



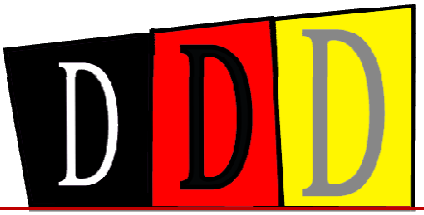
Pipeline operator


```
let versende Waren = Waren
  |> sortieren |> verpacken |> gruppieren |> an_Post…
```



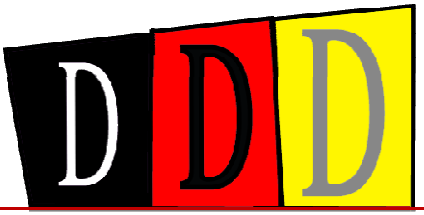
Pipeline operator


```
let versende Waren = Waren
    |> sortieren
    |> verpacken
    |> nach_Produktart_gruppieren
    |> an_PostService_uebergeben
```

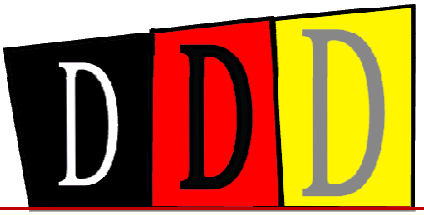


Pipeline operator


```
let versende Waren = Waren
    |> sortieren
    |> nach_Produktart_gruppieren
    |> verpacken
    |> an_PostService_uebergeben
```



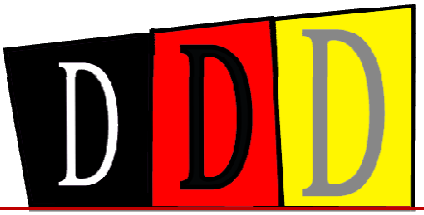
Low Hanging Fruit
Better Semantics
Domain Model
Architecture
The Fantastic Four



Domain model

“By using the model-based language pervasively and not being satisfied until it flows, we approach a model that is complete and comprehensible, made up of simple elements that combine to express complex ideas.”

- Eric Evans



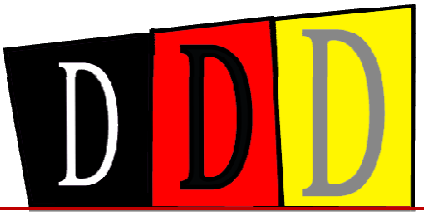
Domain model

@Heimeshoff

```
module CardGame =                                  ← Bounded context
    type Suit = Club | Diamond | Spade | Heart     ← '|' means a choice -- pick one from the list

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King | Ace

    type Card = Suit * Rank                        ← '*' means a pair. Choose one from each type

    type Hand = Card list                          ← list type is built in
    type Deck = Card list

    type Player = {Name:string; Hand:Hand}
    type Game = {Deck:Deck; Players: Player list}

    type Deal = Deck -> (Deck * Card)              ← X -> Y means a function - input of type X - output of type Y

    type PickupCard = (Hand * Card) -> Hand
```

Ubiquitous language

by Scott Wlaschin

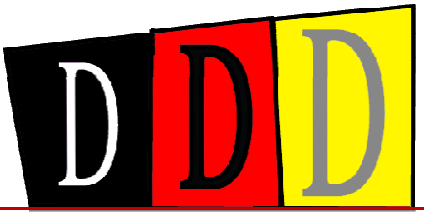


Domain model

1: A company must have a bank to work with

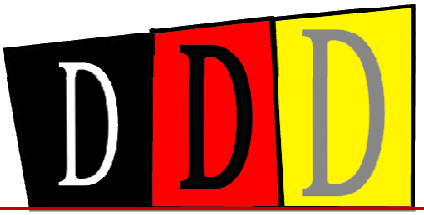
2: A company can be authorized to work with its assigned bank

3: A company can be not authorized to work with its assigned bank



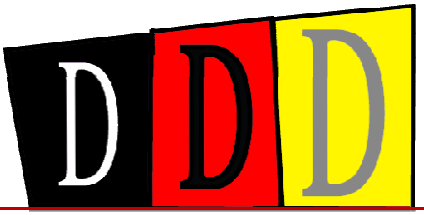
Domain model


```
public class Company
{
   ...

   public String AssignedBank { get; set; }
   public Boolean IsBankAuthorized { get; set; }
}
```



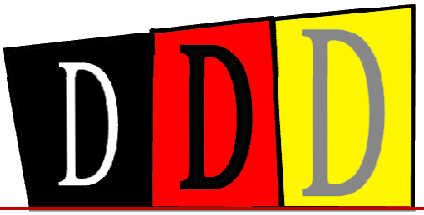
Domain model

Make illegal states unrepresentable!



Domain model

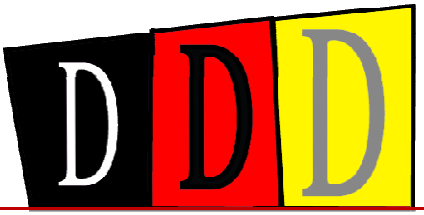

```
type Bank = Bank of string
```



Domain model

@Heimeshoff

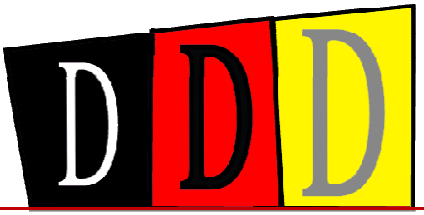
```
type Bank = Bank of string

type UnauthorizedBank = UnauthorizedBank of Bank
type AuthorizedBank   = AuthorizedBank   of Bank
```



Domain model

@Heimeshoff

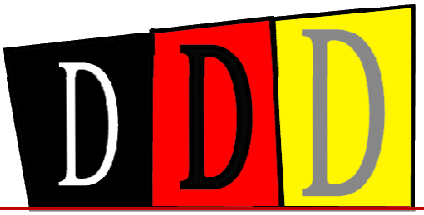
```
type Bank = Bank of string

type UnauthorizedBank = UnauthorizedBank of Bank
type AuthorizedBank   = AuthorizedBank   of Bank


type AssignedBank =
    | Unauthorized of UnauthorizedBank
    | Authorized of AuthorizedBank
```



Domain model

@Heimeshoff

```
type Bank = Bank of string

type UnauthorizedBank = UnauthorizedBank of Bank
type AuthorizedBank   = AuthorizedBank   of Bank


type AssignedBank =
    | Unauthorized of UnauthorizedBank
    | Authorized of AuthorizedBank


 type CompanyProfile = {
    BusinessName : string,
    Tax_Code     : string,
    VatNumber    : string option }
```



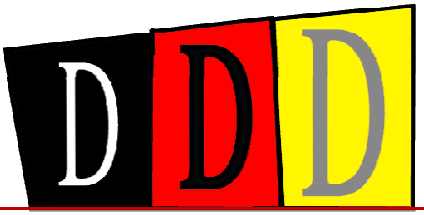
Domain model

@Heimeshoff

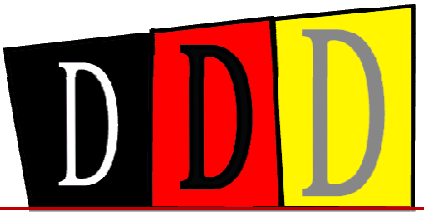
```
type Bank = Bank of string

type UnauthorizedBank = UnauthorizedBank of Bank
type AuthorizedBank   = AuthorizedBank   of Bank


type AssignedBank =
    | Unauthorized of UnauthorizedBank
    | Authorized of AuthorizedBank
```

```
type CompanyProfile = {
    BusinessName : string,
    Tax_Code     : string,
    VatNumber    : string option }
```

```
type Company = {
    Profile : CompanyProfile,
    Bank    : AssignedBank
}
```



Low Hanging Fruit
Better Semantics
Domain Model
Architecture
The Fantastic Four



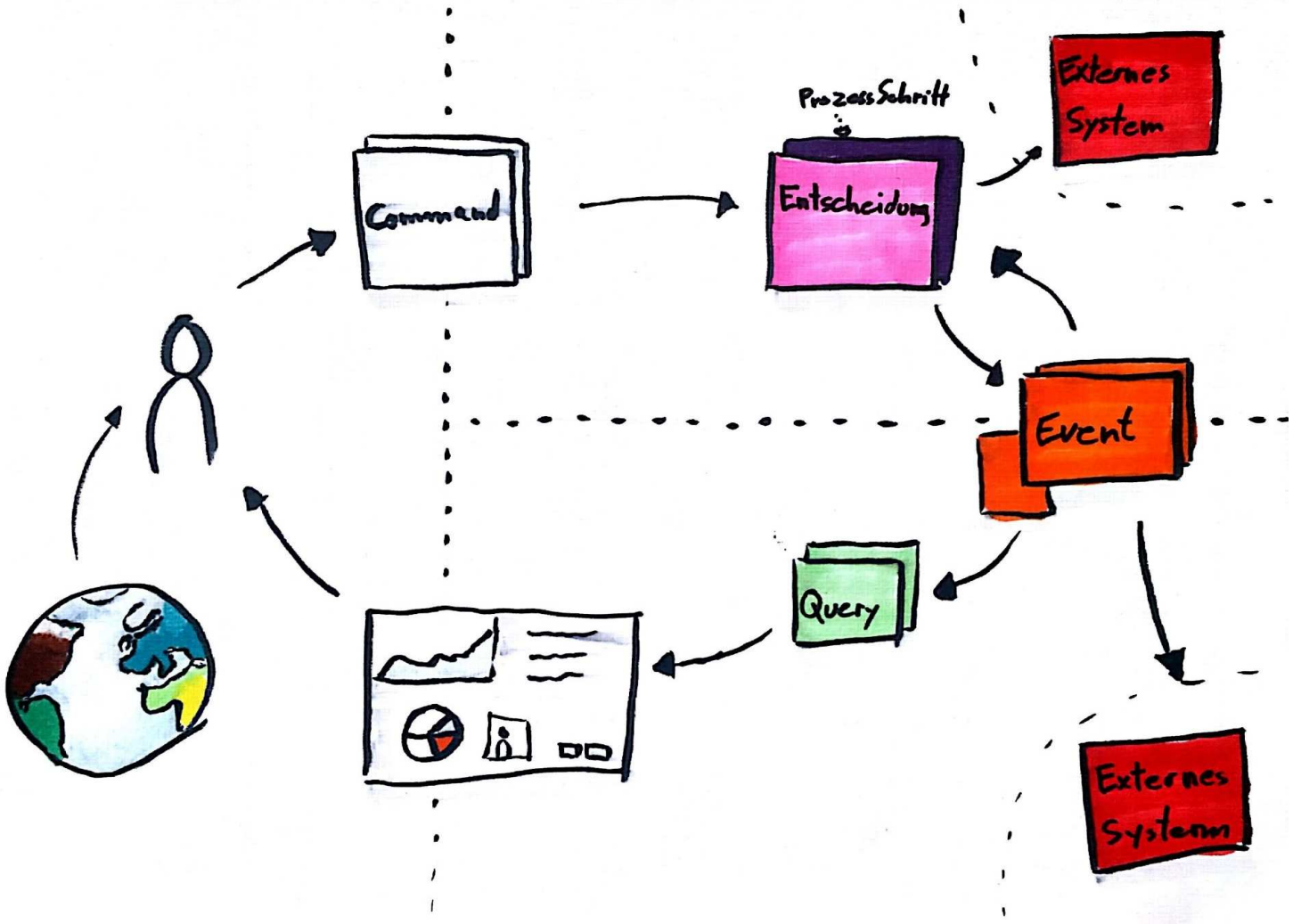
CQRS & Event Sourcing

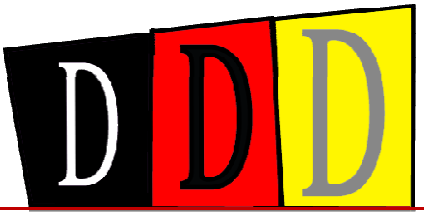
'A single model cannot be appropriate for reporting, searching and transactional behavior.'

- Greg Young

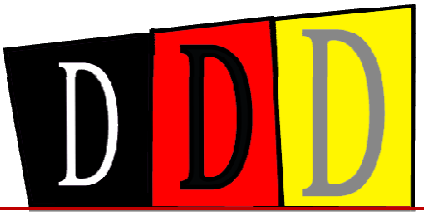
CQRS & Event Sourcing

@Heimeshoff



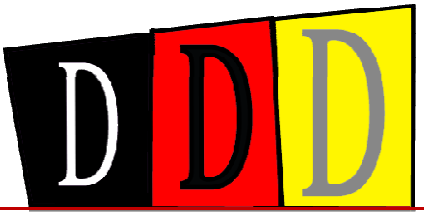


Object oriented approach



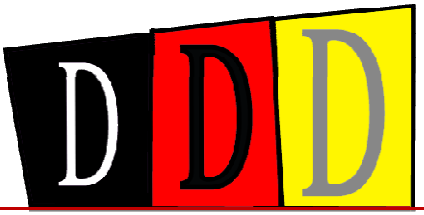
Commands & events


```
public void AddItemToCart(Item item)
{
    // validation
    if (item == null)
        throw new ArgumentNullException();

    // execution
    _items.Add(item.Id);
}
```



Commands & events

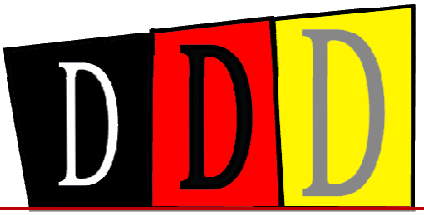

```
public void AddItemToCart(Item item)
{
    if (item == null)
        throw new ArgumentNullException();

    var domainEvent = new ItemAddedToCart
        { CartId = this.Id, ItemId = item.Id };
    Apply(domainEvent)
}

private void Apply(ItemAddedToCart domainEvent)
{
    _items.Add(domainEvent.ItemId);
}
```



Commands & events

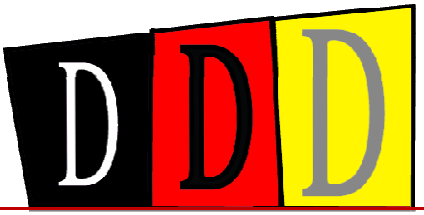
@Heimeshoff

```
public void AddItemToCart(Item item)
{
    if (item == null)
        throw new ArgumentNullException();

    var domainEvent = new ItemAddedToCart
        { CartId = this.Id, ItemId = item.Id };
    Apply(this, domainEvent)
}

private void Apply(Cart target, ItemAddedToCart domainEvent)
{
    target._items.Add(domainEvent.ItemId);
}
```



Commands & events


```
public static Cart Apply(Cart target, CartCreated domainEvent)
{
     return new Cart { Id = domainEvent.CartId, _items = new String[0] };
}

public static Cart Apply(Cart target, ItemAddedToCart domainEvent)
{
     var items = target._items.ToList();
     items.Add(domainEvent.ItemId);

     return new Cart { Id = domainEvent.CartId, _items = items };
}

public static Cart Apply(Cart target, ItemRemovedFromCart domainEvent)
{
     var items = target._items.ToList();
     items.Remove(domainEvent.ItemId);

     return new Cart { Id = domainEvent.CartId, _items = items };
}
```

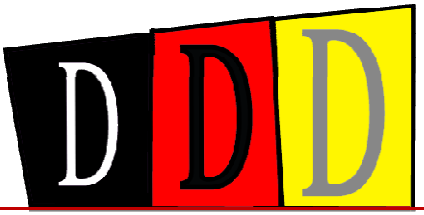


Commands & events

@Heimeshoff

```
var shoppingcart =
```

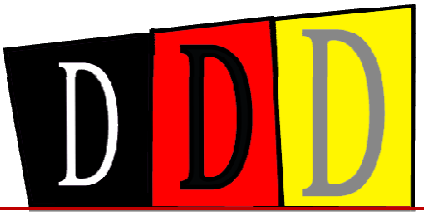
```
Cart.Apply(null, new CartCreated { CartId=1})
```



Commands & events

@Heimeshoff

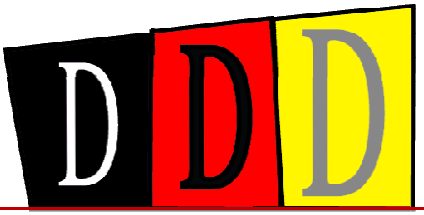
```
var shoppingcart =


            Cart.Apply(
                Cart.Apply(null, new CartCreated { CartId=1}),
                new ItemAddedToCart { CartId = 1, ItemId = "A" }
            )
```



Commands & events

@Heimeshoff

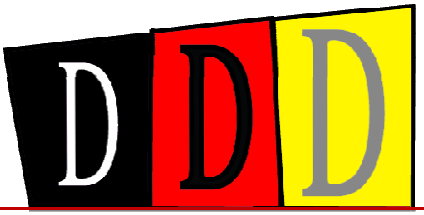
```
var shoppingcart =


        Cart.Apply(
            Cart.Apply(
                Cart.Apply(null, new CartCreated { CartId=1}),
                new ItemAddedToCart { CartId = 1, ItemId = "A" }
            ),
            new ItemAddedToCart { CartId = 1, ItemId = "B" }
        )
```



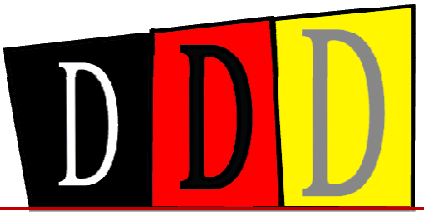
Commands & events

@Heimeshoff

```
var shoppingcart =
    Cart.Apply(
        Cart.Apply(
            Cart.Apply(
                Cart.Apply(null, new CartCreated { CartId=1}),
                new ItemAddedToCart { CartId = 1, ItemId = "A" }
            ),
            new ItemAddedToCart { CartId = 1, ItemId = "B" }
        ),
        new ItemRemovedFromCart { CartId = 1, ItemId = "A" }
    )
```



Functional elegance

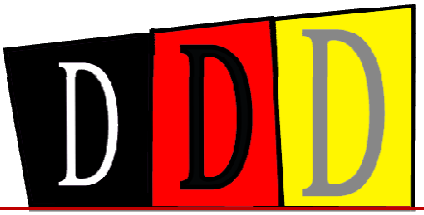


Commands & events

@Heimeshoff

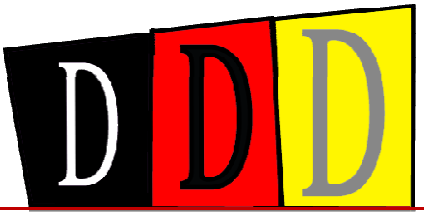
```
type Command =
    | OpenCart of string
    | AddItem of int
    | RemoveItem of int
    | RemoveAllItems
    | Checkout
```

```
type Event =
    | CartOpened of string
    | ItemAdded of int
    | ItemRemoved of int
    | AllItemsRemoved
    | Checkedout
```



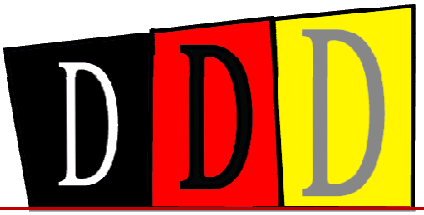
Commands & events


```
type CartState = {
    Name: string;
    Items: List<int>;
    Active: bool;
}
```



Commands & events

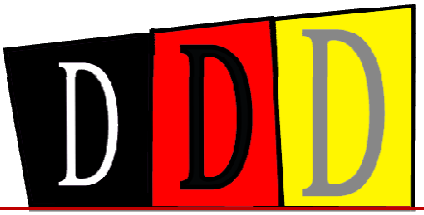

```
type CartState = {
    Name: string;
    Items: List<int>;
    Active: bool;
}


let apply state event =
    match event with
    | CartOpened x -> { Cart.empty with Name = x }
    | ItemAdded x -> { state with Items = List.append state.Items [x] }
    | ItemRemoved x -> { state with Items = List.filter (fun i -> i <> x ) state.Items }
    | Removed _ -> { state with Items = List.empty }
    | Checkedout _ -> { state with Active = false }
```



Commands & events

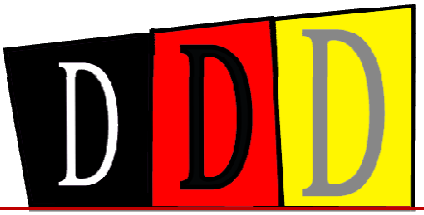
@Heimeshoff

```
var shoppingcart =
    Cart.Apply(
        Cart.Apply(
            Cart.Apply(
                Cart.Apply(null, new CartCreated { CartId=1}),
                new ItemAddedToCart { CartId = 1, ItemId = "A" }
            ),
            new ItemAddedToCart { CartId = 1, ItemId = "B" }
        ),
        new ItemRemovedFromCart { CartId = 1, ItemId = "A" }
    )
```



Commands & events

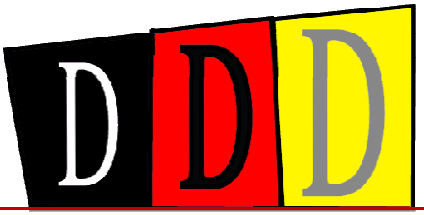

```
var events = new [
    new CartCreated { CartId=1},
    new ItemAddedToCart { CartId = 1, ItemId = "A" },
    new ItemAddedToCart { CartId = 1, ItemId = "B" },
    new ItemRemovedFromCart { CartId = 1, ItemId = "A"}
]

Public Cart State(Cart cart, List<DomainEvents> events)
{
    If (Cart == null) throw new TuMaDieMöhrchenException();
    foreach(var event in events)
    {
        cart.Apply(event);
    }
}
```



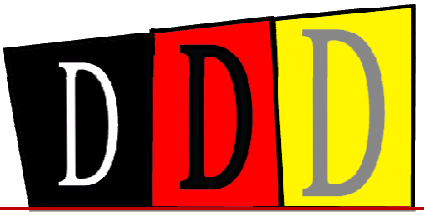
Commands & events

@Heimeshoff

```
let domainEvents = [
    CartOpened("cart1");
    ItemAdded(1);
    ItemAdded(2);
    Removed(1);
]

let state = List.fold apply Cart.empty domainEvents
```



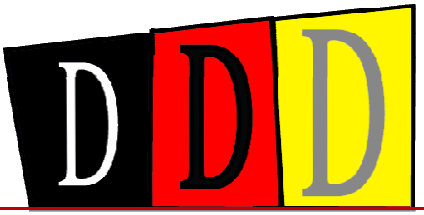
And what about projections?



Projections


```
type CartReadmodel = {
    Name: string;
    Items: List<int>;
    Active: bool;
}
```

```
let apply state event =
    match event with
    | CartOpened x -> { Cart.empty with Name = x }
    | ItemAdded x -> { state with Items = List.append state.Items [x] }
    | ItemRemoved x -> { state with Items = List.filter (fun i -> i <> x ) state.Items }
    | Removed _ -> { state with Items = List.empty }
    | Checkedout _ -> { state with Active = false }
```

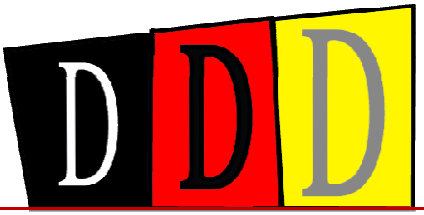


Projections

@Heimeshoff

```
type RemovedItemsReadmodel = {
    Name: string;
    Items: List<int>;
}
```

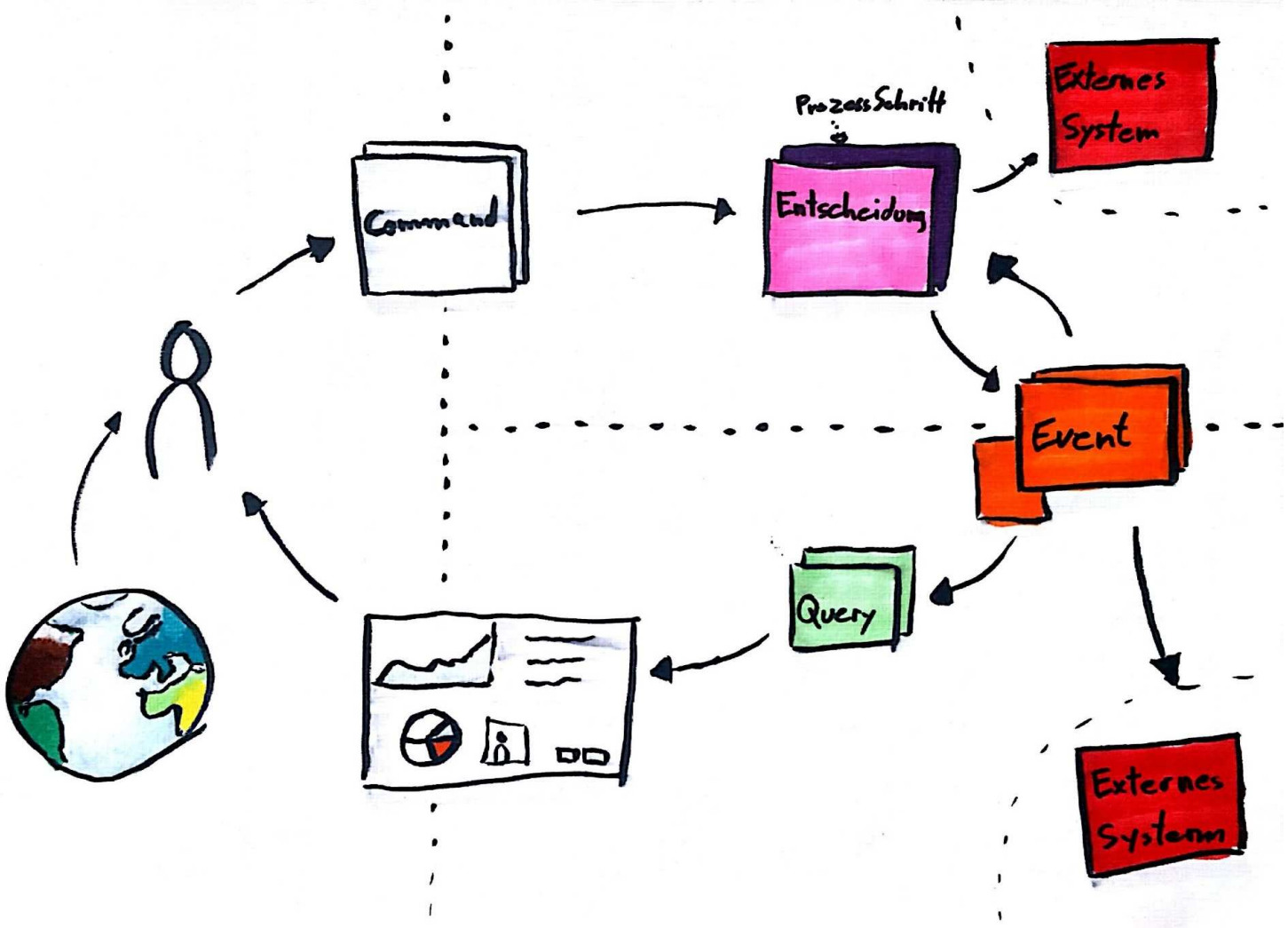
```
let apply state event =
    match event with
    | CartOpened x -> { Cart.empty with Name = x }
    | ItemAdded x -> { state with Items = List.filter (fun i -> i <> x ) state.Items }
    | ItemRemoved x -> { state with Items = List.append state.Items [x] }
    | Removed _ -> { state with Items = List.append state.Items }
    | Checkedout _ -> _
```

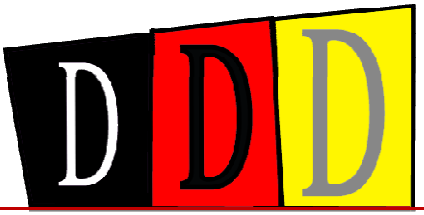


Low Hanging Fruit
Better Semantics
Domain Model
Architecture
The Fantastic Four

The Fantastic Four

@Heimeshoff





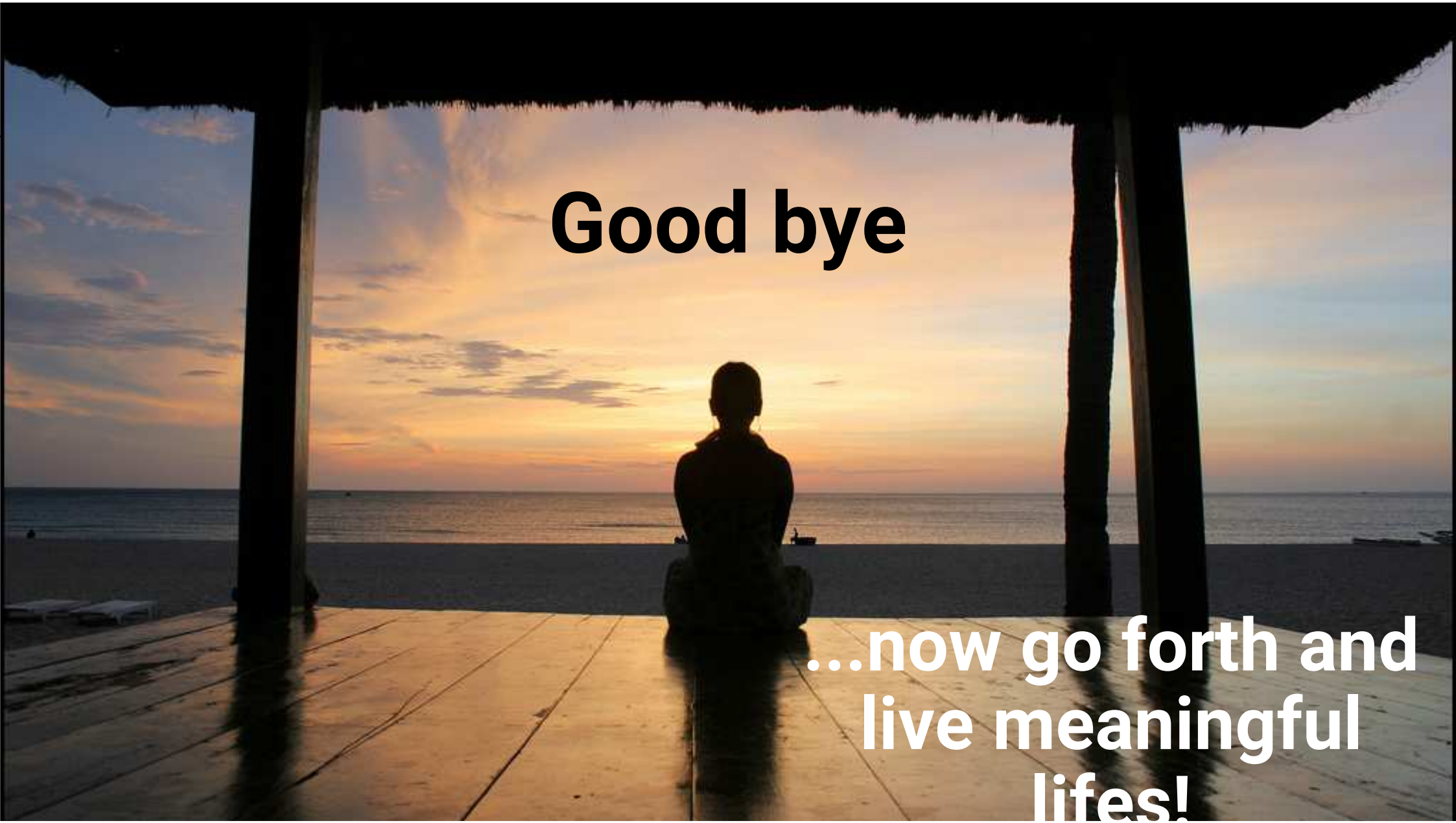
The Fantastic Four


```
type Intention = state -> command

type Geschäftsregel = (history * command) -> events

type Interpretation = event list -> state

type Automation =  event -> command | event
```

FUNCTIONAL ALL THE THINGS!!!!

DDD ALL THE THINGSER!!!!!!!!

FUNCTIONAL ALL THE THINGS!!!!


Good bye
...now go forth and live meaningful lifes!