

# Building Rich Domain Models with DDD and TDD

Ivan Paulovich

Betsson Dev'talk #3

Stockholm – September 12<sup>th</sup>, 2018



<https://paulovich.net>



[ivan.paulovich@betssongroup.com](mailto:ivan.paulovich@betssongroup.com)

@ivanpaulovich





Ivan Paulovich

Developer

**betsson** group



30+ Microsoft Certifications

[paulovich.net](http://paulovich.net)

@ivanpaulovich 

# Betsson Wallet Team

- Seniors Developers
  - Agile Team
  - Business Oriented
  - .NET – SQL Server – Angular
- 
- Stockholm Office
  - We are hiring!



# How to shoot yourself in the foot:

1. Design your application starting from the data model.
2. Create your domain model by reverse engineering.
3. Pretend that you're doing TDD and start testing your domain classes.
  - Particularly getters and setters.
4. Now start testing the logic with Integration Tests and get stuck by test data and related issues.
5. Declare that TDD provides no benefit and only slows you down.
6. Comment tests in your Continuous Integration proccess.
7. Keep on whining.

*Alberto Brandolini*

# Domain-Driven Design

Tiny Domain Objects

Frequent Rewriting

Exploratory Coding

Quick Feedback

Self Explanatory Coding

# Test-Driven Development

Focus on Unit Tests

Frequent Short Cycles

Freedom to Change



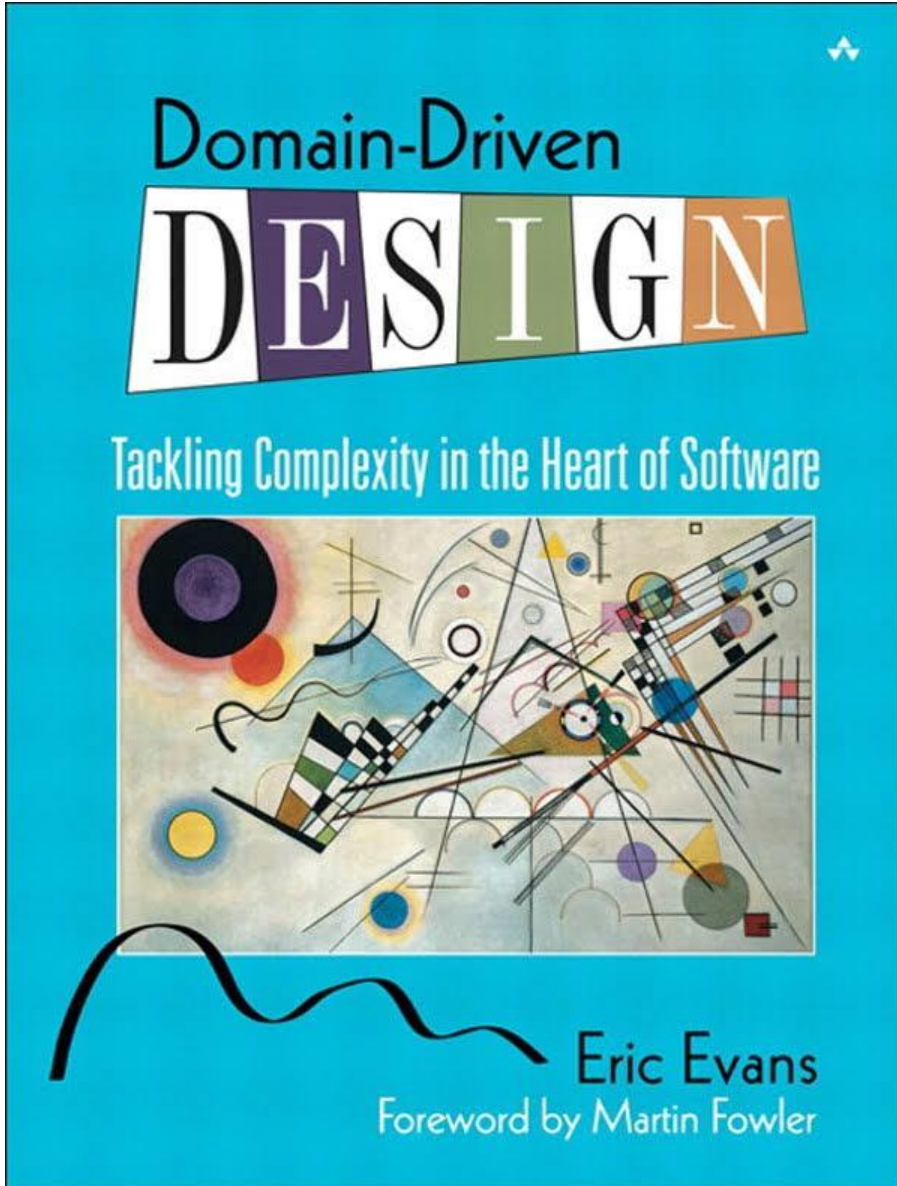
# A Customer Entity with Primitive Obsession...

```
public class Customer : IEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Personnummer { get; set; }
    public string Email { get; set; }
    public string MobilePhoneNumber { get; set; }
}
```

# Leads to Services Like..

```
public class RegisterCustomerUseCase
{
    public RegisterOutput Execute(
        string firstName,
        string lastName,
        string personnummer,
        string email,
        string mobilePhoneNumber)
    { ... }
}
```

- Needs to verify for required parameters
- Needs to verify for Data Format
- Needs to verify for Data Range
- Services are Big and Fat
- Easy to confuse one parameter with the another.




- Not a technology.
- Not a methodology.
- Set of principles and patterns for focusing the design effort where it matters most.



# An Customer Entity Using Value Objects..

```
public class Customer : IEntity
{
    public int Id { get; set; }
    public FirstName FirstName { get; set; }
    public LastName LastName { get; set; }
    public Personnummer Personnummer { get; set; }
    public Email Email { get; set; }
    public MobilePhoneNumber MobilePhoneNumber { get; set; }
}
```



# Business Rules Enforced Through Value Objects

```
public class RegisterCustomerUseCase
{
    public RegisterOutput Execute(
        FirstName firstName,
        LastName lastName,
        Personnummer personnummer,
        Email email,
        MobilePhoneNumber mobilePhoneNumber)
    { ... }
}
```

- The simple existence of a Value Object means that it is valid.
- No need to verify for the data format on every method.
- **Services are thinner when using Value Objects.**

DDD express the Model with  
Value Objects, Entities and Services.

Some Entities act as root of Aggregates.

# An Example with Some Use Cases

- A customer can register a new account using its personal details.
- Allow a customer to deposit funds into an existing account.
- Allow to withdraw from an existing account.
- Do not allow to withdraw more than the existing funds.

## Customer 5557-8

### Account Number 4444-6 (Day-to-Day)

Date	Description	Debit (SEK)	Credit (SEK)	Balance (SEK)
01-08-2018	Initial Balance			50,000
03-08-2018	Withdrawn	10,000		40,000
07-08-2018	Withdrawn	5,000		35,000
17-09-2018	Deposited		7,000	42,000

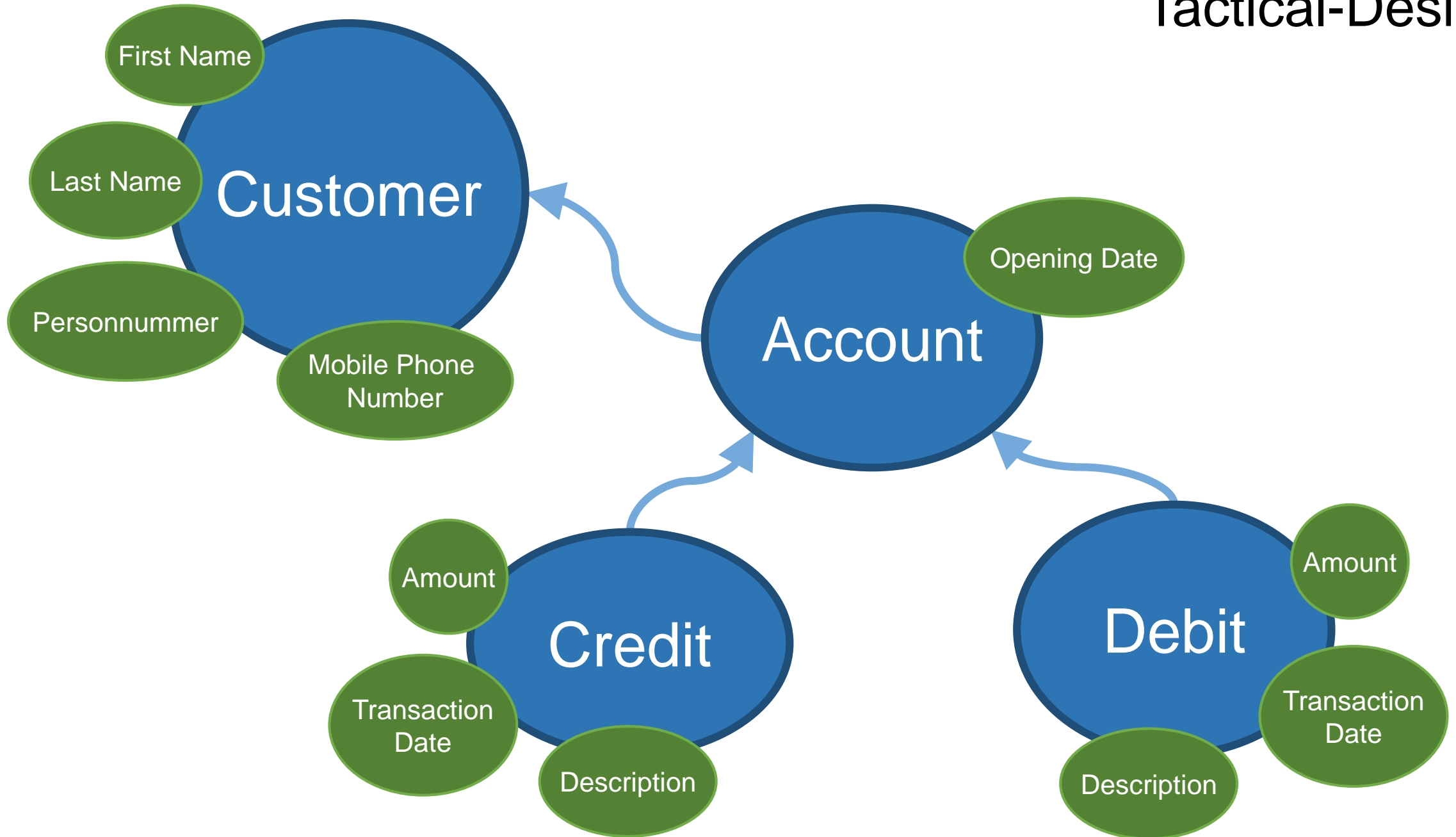
### Account Number 7777-0 (Savings)

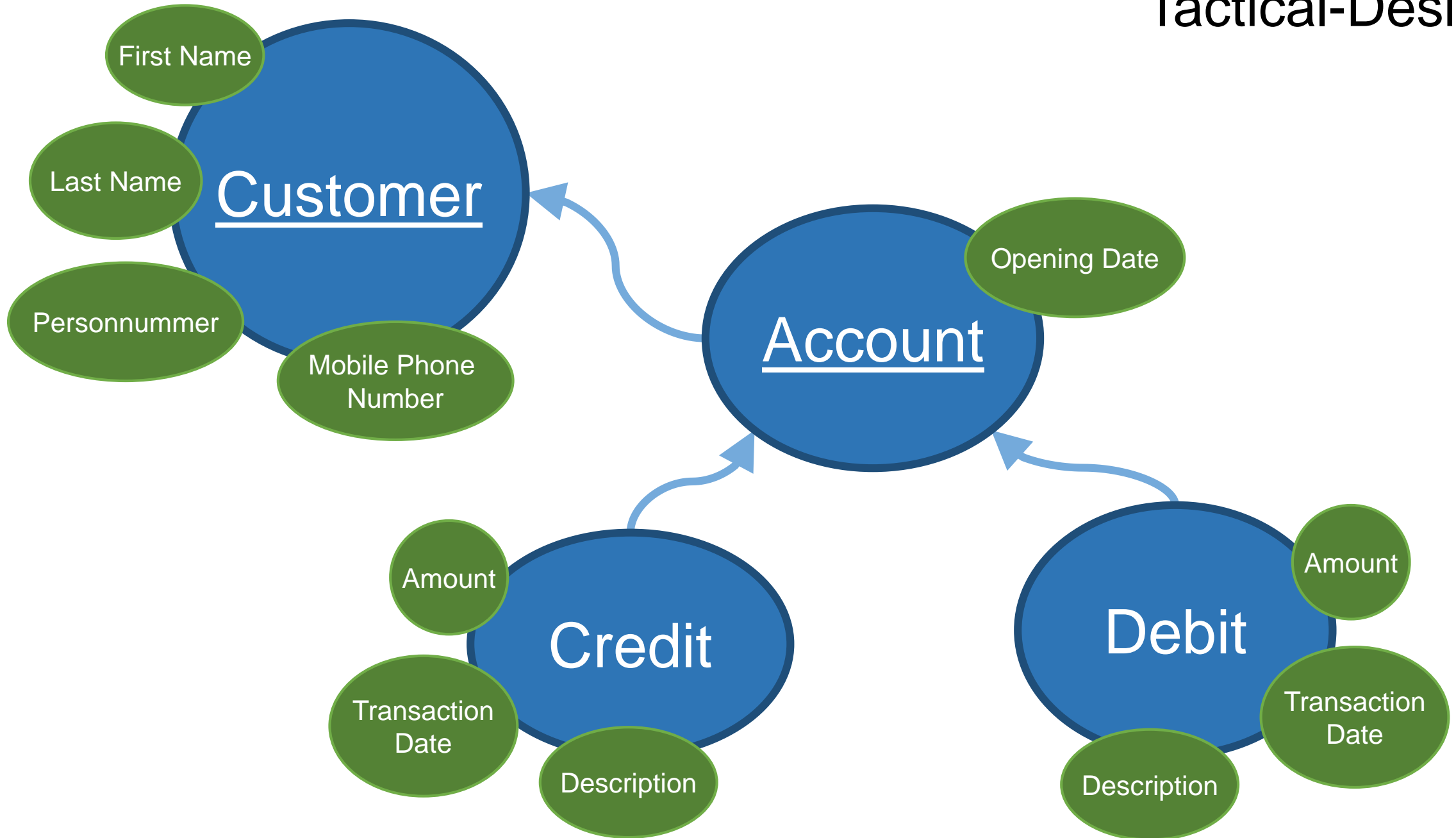
Date	Description	Debit (SEK)	Credit (SEK)	Balance (SEK)
01-09-2018	Initial Balance			10,000



# Some Nouns and Verbs are Useful

- A **customer** can **register** a new account using its personal details.
- Allow a **customer** to **deposit** funds into an existing account.
- Allow to **withdraw** from an existing **account**.
- Do not allow to **withdraw** more than the existing funds.





## Personal Expenses Bounded Context

Customer

First Name

Last Name

Personnummer

Mobile Phone  
Number

Account

Opening Date

Credit

Amount

Transaction  
Date

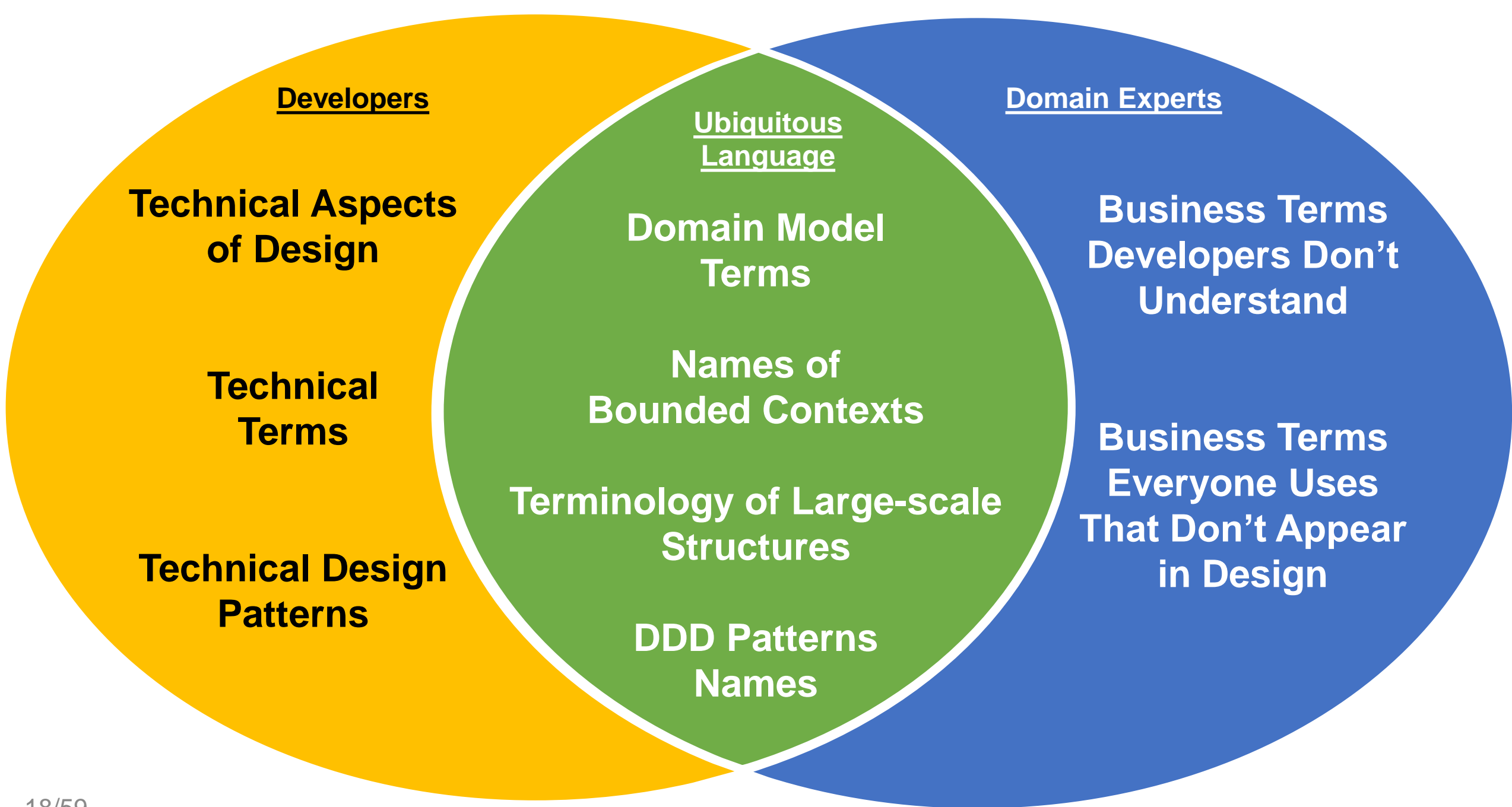
Description

Debit

Amount

Transaction  
Date

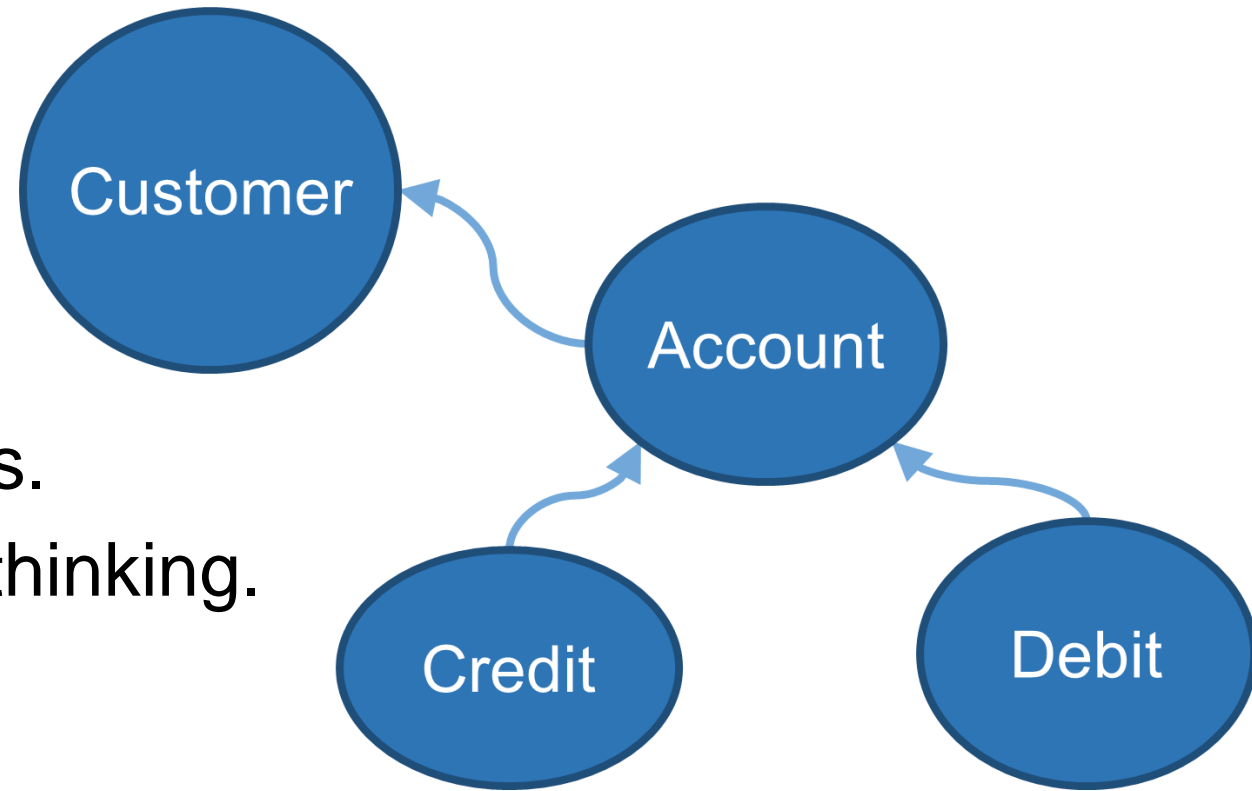
Description





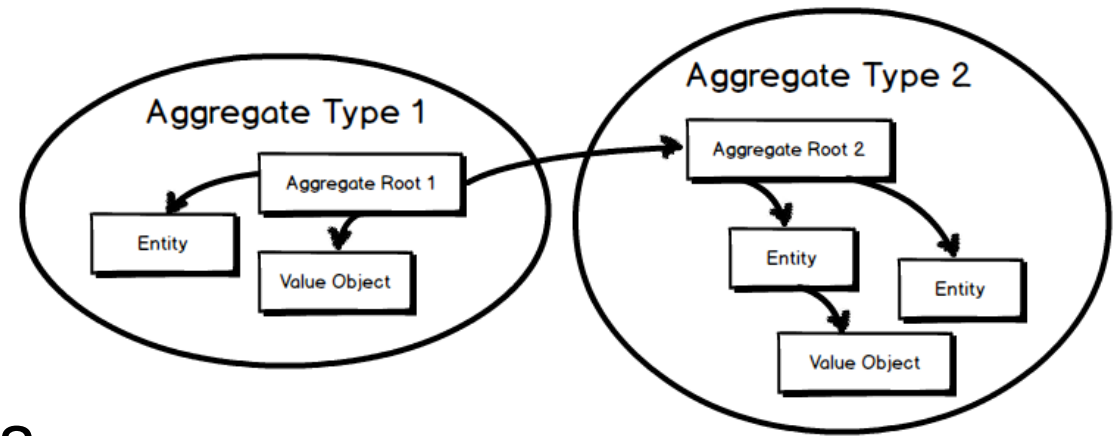
# Entities

- Have a unique identity.
- Are mutable or not.
- Refer others entities by their IDs.
- Don't get trapped by datastore thinking.



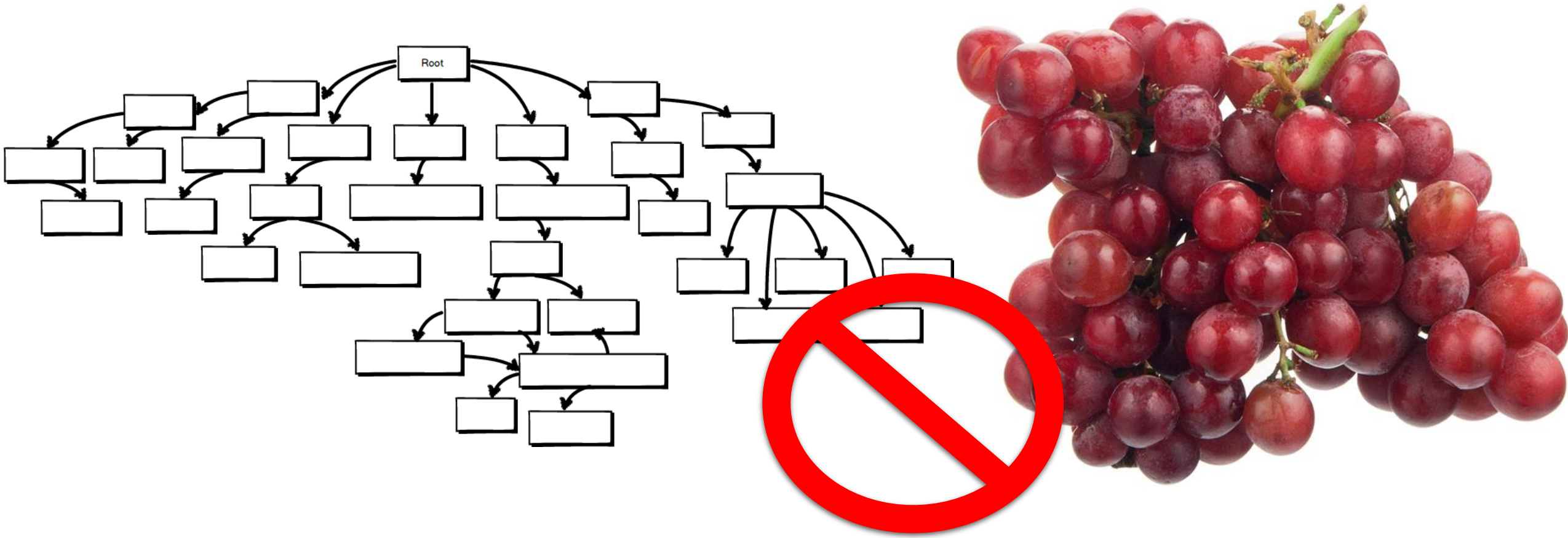
# Aggregate Roots (Are Entities)

- Refer other aggregates by identity only.
- Scope of consistency inside the aggregate boundaries and update other aggregates through eventual consistency.
- Aggregates **are small**.



- Aggregates implement behaviors.
- One Aggregate Root for every Entity is a Code Smell.

# An Aggregate Root is not your Entire Model



# An Aggregate Root



# Account Aggregate Root

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Transactions = new TransactionCollection();
    }

    public void Deposit(Amount amount) { ... }
    public void Withdraw(Amount amount) { ... }
    public void Close() { ... }
    public Amount GetCurrentBalance() { ... }
    public ITransaction GetLastTransaction() { ... }

    private Account() { }

    public static Account LoadFromDetails(Guid id, Guid customerId, TransactionCollection transactions) { ... }
}
```



# Account Aggregate Root

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Transactions = new TransactionCollection();
    }

    public void Deposit(Amount amount) { ... }
    public void Withdraw(Amount amount) { ... }
    public void Close() { ... }
    public Amount GetCurrentBalance() { ... }
    public ITransaction GetLastTransaction() { ... }

    private Account() { }
}
```

It is an Entity

```
24/59 public static Account LoadFromDetails(Guid id, Guid customerId, TransactionCollection transactions) { ... }
}
```

# Account Aggregate Root

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Transactions = new TransactionCollection();
    }

    public void Deposit(Amount amount) { ... }
    public void Withdraw(Amount amount) { ... }
    public void Close() { ... }
    public Amount GetCurrentBalance() { ... }
    public ITransaction GetLastTransaction() { ... }

    private Account() { }
}
```

It is an Entity

Only mandatory fields are  
required in the constructor

```
25/59 public static Account LoadFromDetails(Guid id, Guid customerId, TransactionCollection transactions) { ... }
}
```

# Account Aggregate Root

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Transactions = new TransactionCollection();
    }

    public void Deposit(Amount amount) { ... }
    public void Withdraw(Amount amount) { ... }
    public void Close() { ... }
    public Amount GetCurrentBalance() { ... }
    public ITransaction GetLastTransaction() { ... }

    private Account() { }
}
```

It is an Entity

Only mandatory fields are required in the constructor

Implements behaviors which maintain the state consistent.

```
26/59 public static Account LoadFromDetails(Guid id, Guid customerId, TransactionCollection transactions) { ... }
}
```

# Account Aggregate Root

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Transactions = new TransactionCollection();
    }

    public void Deposit(Amount amount) { ... }
    public void Withdraw(Amount amount) { ... }
    public void Close() { ... }
    public Amount GetCurrentBalance() { ... }
    public ITransaction GetLastTransaction() { ... }

    private Account() { }

    public static Account LoadFromDetails(Guid id, Guid customerId, TransactionCollection transactions) { ... }
}
```

It is an Entity

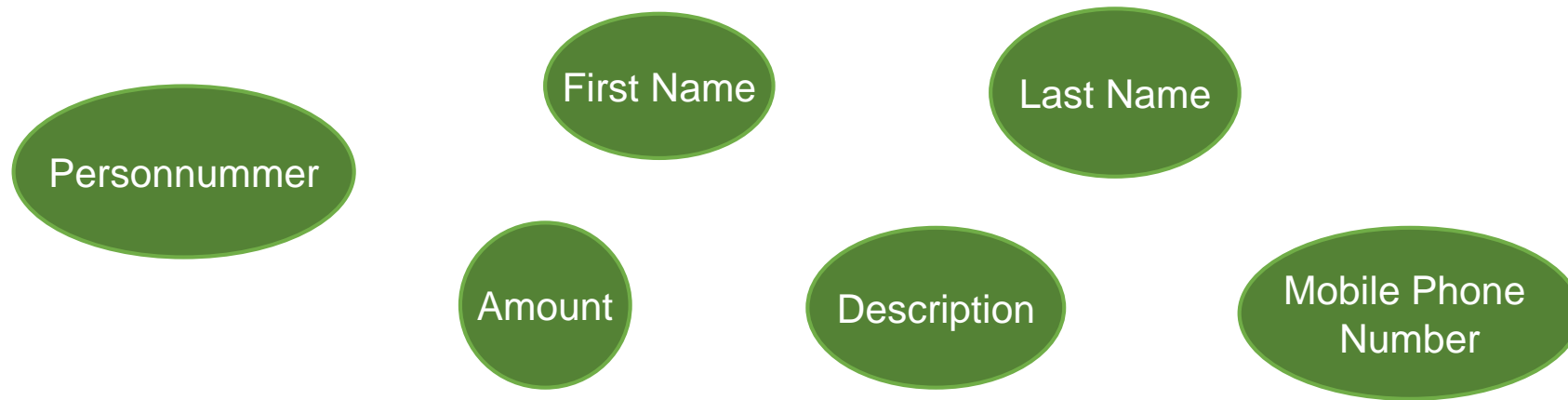
Only mandatory fields are required in the constructor

Implements behaviors which maintain the state consistent.

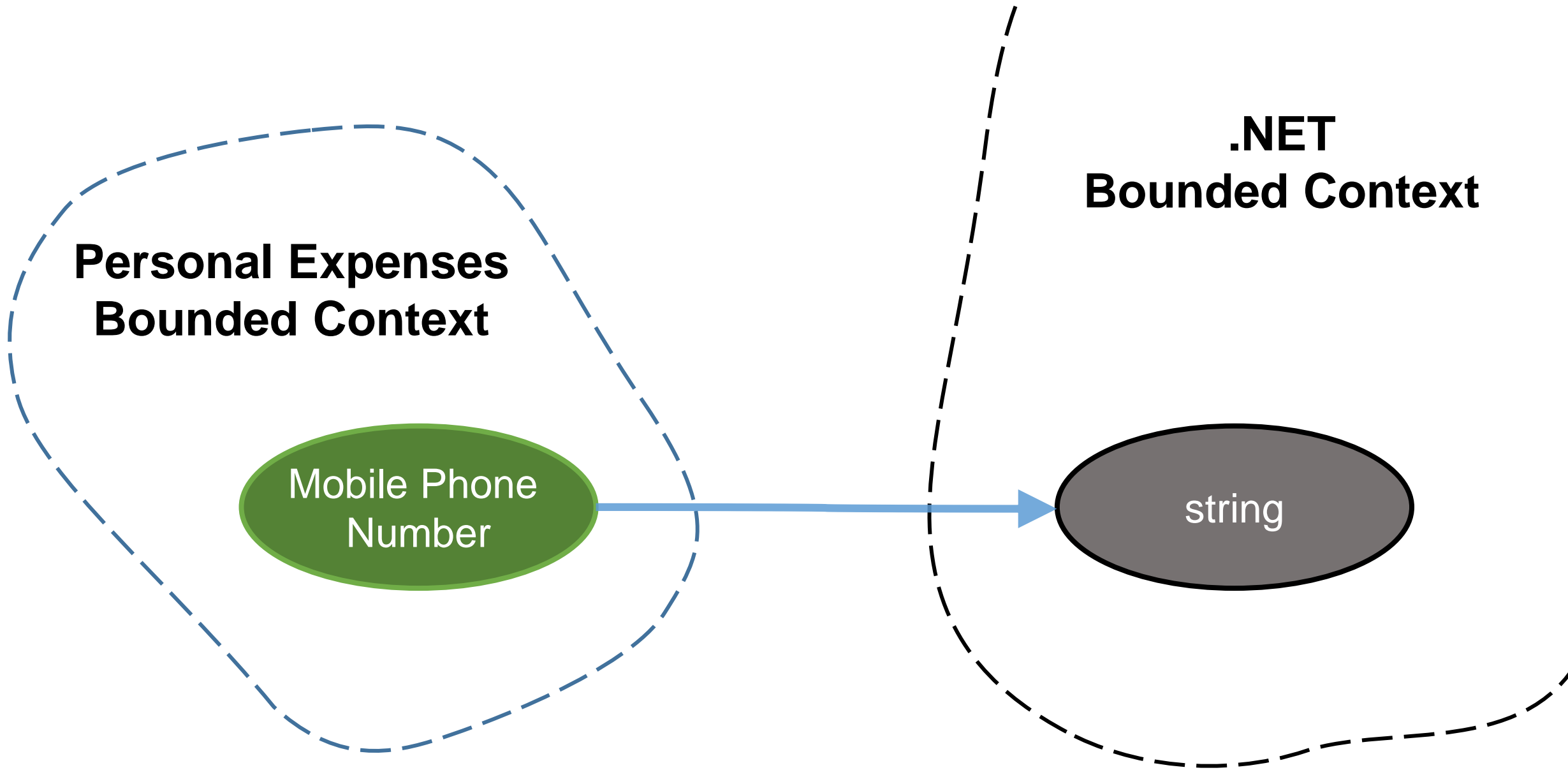
Factory method to restore state.

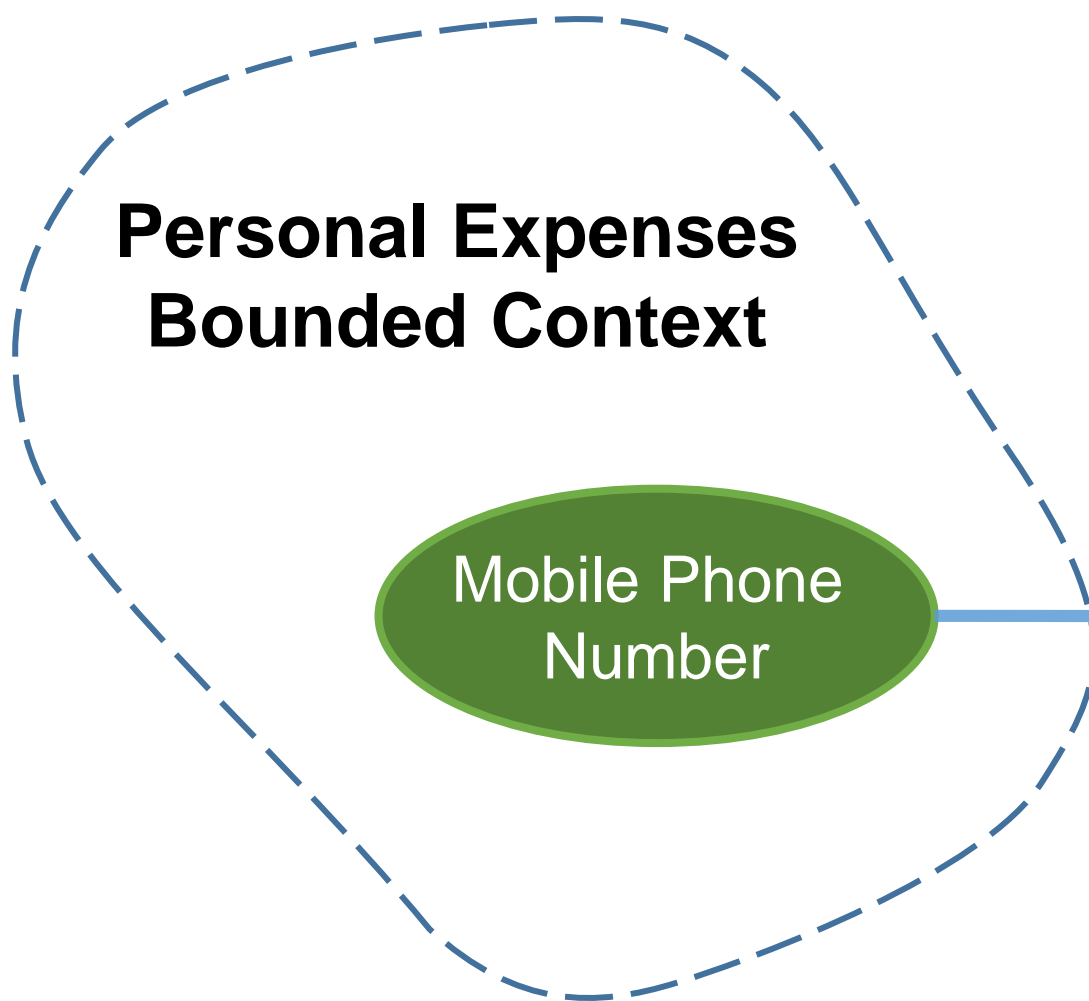
# Value Objects

- Immutable.
- Have no explicit identity.
- Unique by the comparison of the attributes.
- Used to describe, measure or quantify an Entity.









# .NET Bounded Context

Clone	IndexOfAny	StartsWith
Compare	Insert	Substring
CompareOrdinal	Intern	ToCharArray
CompareTo	IsInterned	ToLower
Concat	IsNormalized	ToLowerInvariant
Contains	IsNullOrEmpty	ToString
Copy	IsNullOrWhiteSpace	ToUpper
CopyTo	Join	ToUpperInvariant
Create	LastIndexOf	Trim
EndsWith	LastIndexOfAny	TrimEnd
Equals	Normalize	TrimStart
Format	PadLeft	
GetEnumerator	PadRight	
GetHashCode	Remove	
GetTypeCode	Replace	
IndexOf	Split	

## Personal Expenses Bounded Context

Mobile Phone  
Number

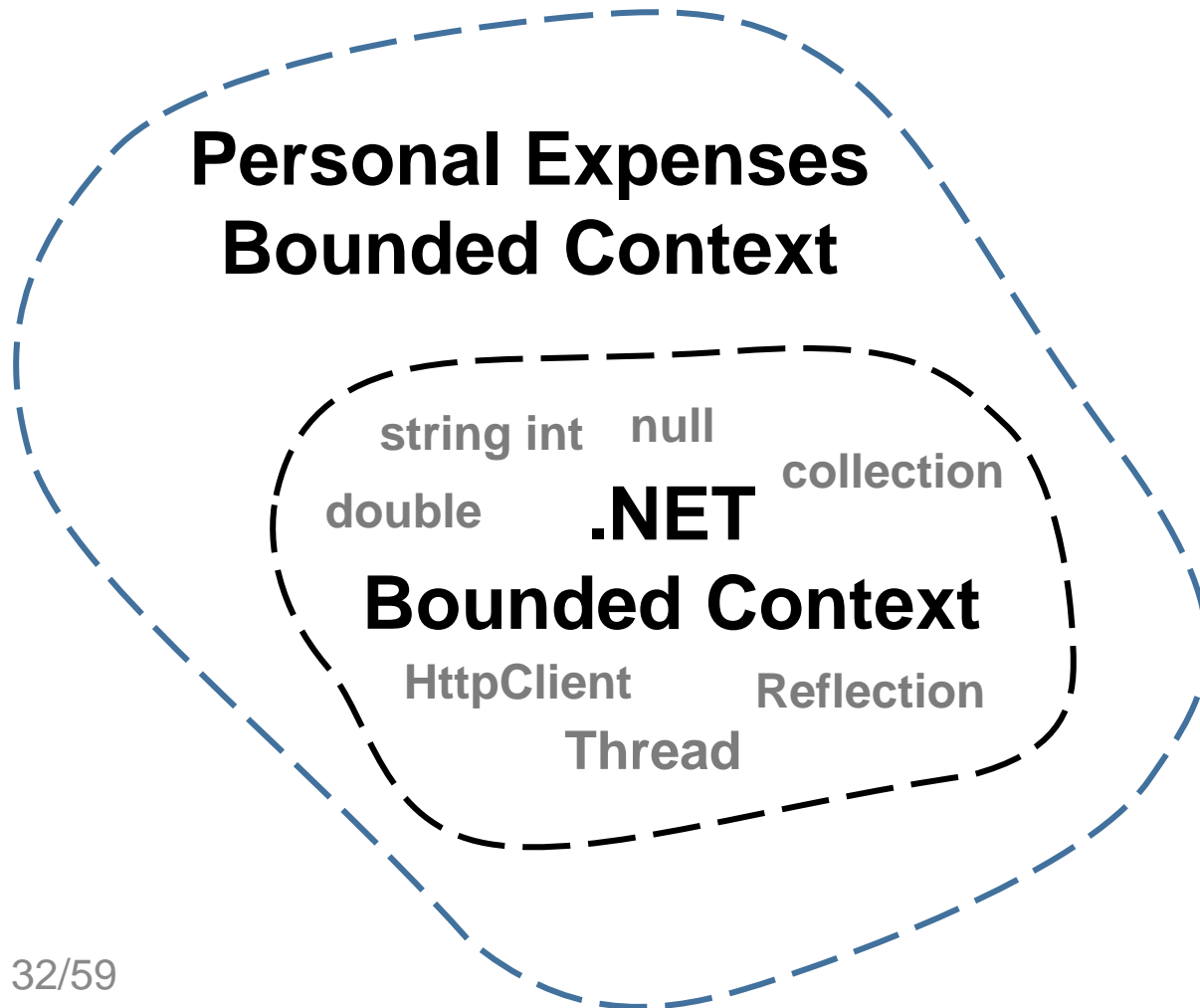
Create  
GetAreaCode  
GetLastFourDigits  
ToString

## .NET Bounded Context

Clone	IndexOfAny	StartsWith
Compare	Insert	Substring
CompareOrdinal	Intern	ToCharArray
CompareTo	IsInterned	ToLower
Concat	IsNormalized	ToLowerInvariant
Contains	IsNullOrEmpty	ToString
Copy	IsNullOrWhiteSpace	ToUpper
CopyTo	Join	ToUpperInvariant
Create	LastIndexOf	Trim
EndsWith	LastIndexOfAny	TrimEnd
Equals	Normalize	TrimStart
Format	PadLeft	
GetEnumerator	PadRight	
GetHashCode	Remove	
GetTypeCode	Replace	
IndexOf	Split	

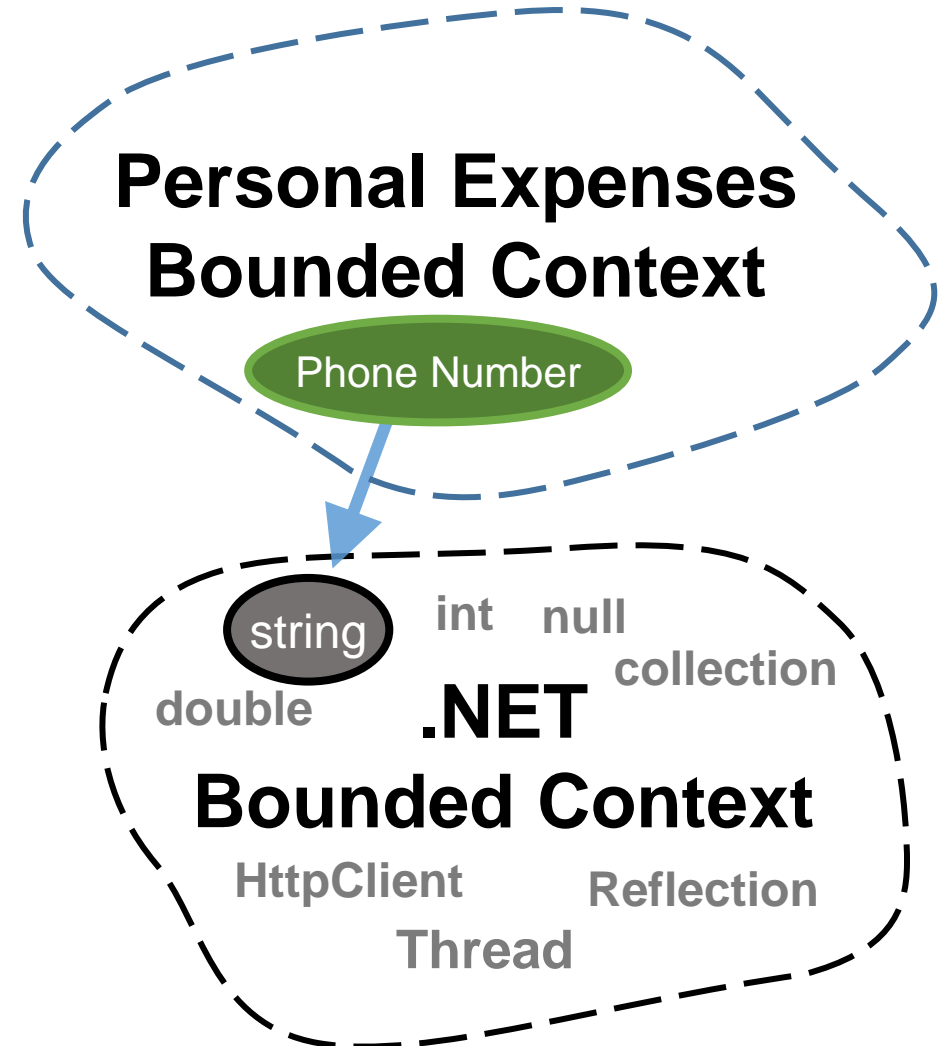
## Without Value Objects

We bring the .NET Framework Complexity into our Bounded Context.



## With Value Objects

We only pay for the complexity we really use



# Personnummer Value Object

```
public sealed class Personnummer
{
    private string _text;
    const string RegexForValidation = @"^\d{6,8}[-|(\s)]{0,1}\d{4}$";

    public Personnummer(string text)
    {
        if (string.IsNullOrEmpty(text))
            throw new SSNShouldNotBeEmptyException("The 'Personnummer' field is required");

        Regex regex = new Regex(RegexForValidation);
        Match match = regex.Match(text);

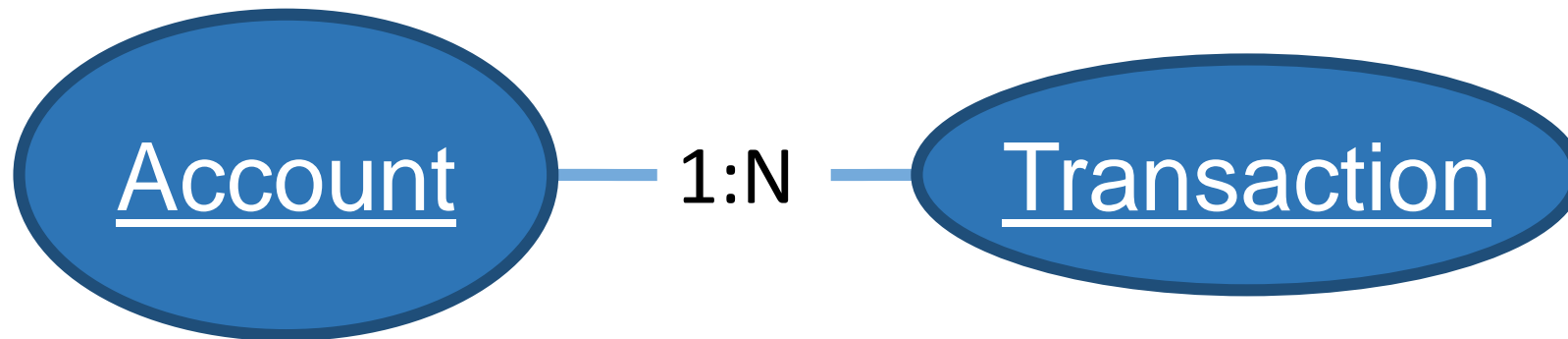
        if (!match.Success)
            throw new InvalidSSNException("Invalid Personnummer format. Use YYMMDDNNNN.");

        _text = text;
    }
}
```

33/59

# First-Class Collections

- Each collection should be wrapped in its own class<sup>1</sup>.
- Classes that contains collections do not contains any other variable.
- Behaviors have a home.
- When necessary return immutable collection copies.



# First-Class TransactionCollection

```
public sealed class TransactionCollection
{
    private readonly IList<ITransaction> _transactions;

    public TransactionCollection()
    {
        _transactions = new List<ITransaction>();
    }

    public void Add(ITransaction transaction) { ... }
    public void Add(IEnumerable<ITransaction> transactions) { ... }
    public Amount GetBalance() { ... }

    public IReadOnlyCollection<ITransaction> ToReadOnlyCollection() { ... }
    public ITransaction CopyOfLastTransaction() { ... }
}
```

# First-Class TransactionCollection

```
public sealed class TransactionCollection
{
    private readonly IList<ITransaction> _transactions;

    public TransactionCollection()
    {
        _transactions = new List<ITransaction>();
    }

    public void Add(ITransaction transaction) { ... }
    public void Add(IEnumerable<ITransaction> transactions) { ... }
    public Amount GetBalance() { ... }

    public IReadOnlyCollection<ITransaction> ToReadOnlyCollection() { ... }
    public ITransaction CopyOfLastTransaction() { ... }
}
```

**Copy collections and mutable objects  
when passing them between objects.<sup>1</sup>**



# How to Use the TransactionCollection Class

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId) { ... }

    public void Withdraw(Amount amount)
    {
        Amount balance = Transactions.GetBalance();

        if (balance < amount)
            throw new InsufficientFundsException(
                $"The account {Id} does not have enough funds to withdraw {amount}. Current Balance {balance}.");

        Debit debit = new Debit(Id, amount);
        Transactions.Add(debit);
    }

    public void Deposit(Amount amount) { ... }
}
```

# How to Use the TransactionCollection Class

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId) { ... }

    public void Withdraw(Amount amount)
    {
        Amount balance = Transactions.GetBalance();

        if (balance < amount)
            throw new InsufficientFundsException(
                $"The account {Id} does not have enough funds to withdraw {amount}. Current Balance {balance}.");

        Debit debit = new Debit(Id, amount);
        Transactions.Add(debit);
    }

    public void Deposit(Amount amount) { ... }
}
```

The GetBalance() implementation belongs to the TransactionCollection class.

# How to Use the TransactionCollection Class

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId) { ... }

    public void Withdraw(Amount amount)
    {
        Amount balance = Transactions.GetBalance();

        if (balance < amount)
            throw new InsufficientFundsException(
                $"The account {Id} does not have enough funds to withdraw {amount}. Current Balance {balance}.");

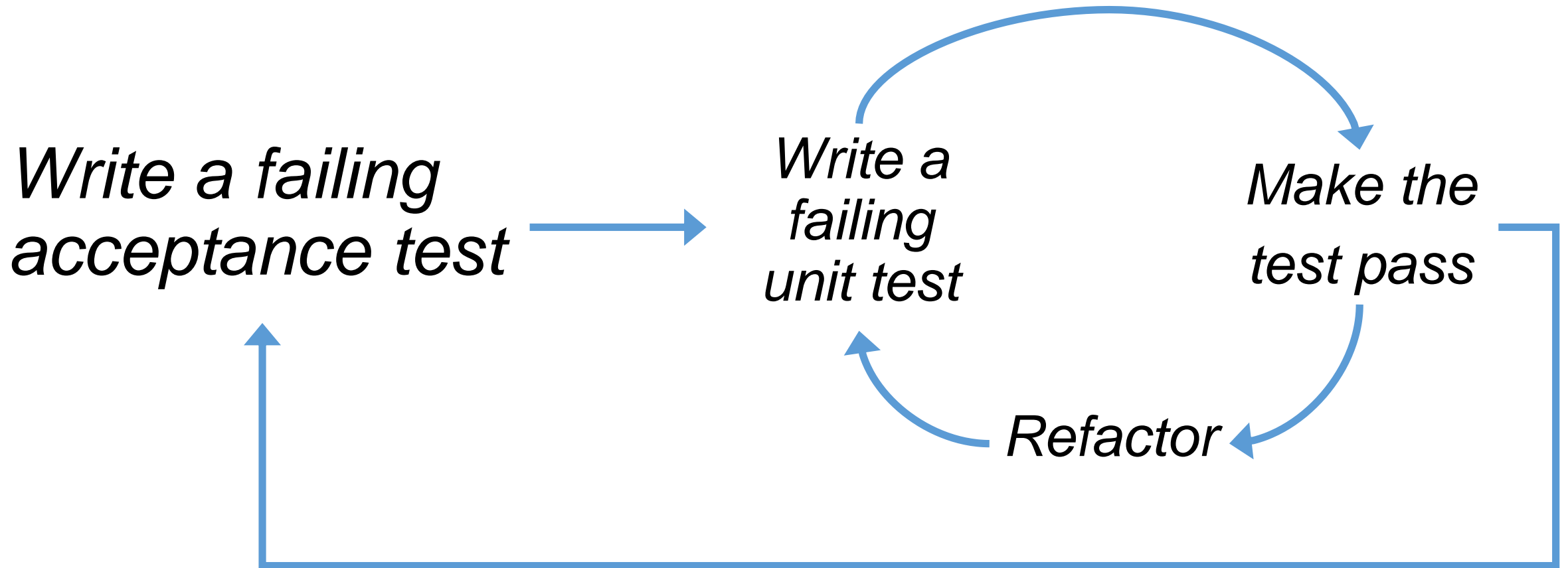
        Debit debit = new Debit(Id, amount);
        Transactions.Add(debit);
    }

    public void Deposit(Amount amount) { ... }
}
```

**Composite simpler than  
the sum of its parts**

**The GetBalance() implementation belongs to  
the TransactionCollection class.**

# Inner and outer feedback loops in TDD



[Fact]

```
public void Deposit_Should_Change_Balance_When_Account_Is_New()
```

```
{
```

```
    //
```

```
    // Arrange
```

```
    Guid expectedCustomerId = Guid.Parse("ac608347-74ac-4607-abc2-7b95cdc8a122");
```

```
    Amount expectedAmount = new Amount(400m);
```

```
    //
```

```
    // Act
```

```
    Account sut = new Account(expectedCustomerId);
```

```
    sut.Deposit(expectedAmount);
```

```
    Amount balance = sut.GetCurrentBalance();
```

```
    //
```

```
    // Assert
```

```
    Assert.Equal(expectedCustomerId, sut.CustomerId);
```

```
    Assert.Equal(expectedAmount, balance);
```

```
    Assert.Single(sut.Transactions.ReadOnlyCollection());
```

```
}
```

**Write a failing  
acceptance test**

Write a  
failing  
unit test

Make the  
test pass

Refactor

[Fact]

```
public void Deposit_Should_Change_Balance_Equivalent_Amount()
```

```
{
```

```
    //
```

```
    // Arrange
```

```
    Guid expectedCustomerId = Guid.Parse("ac608347-74ac-4607-abc2-7b95cdc8a122");
```

```
    Amount expectedAmount = new Amount(400m);
```

```
    //
```

```
    // Act
```

```
    Account sut = new Account(expectedCustomerId);
```

```
    sut.Deposit(expectedAmount);
```

```
    Amount balance = sut.GetCurrentBalance();
```

```
    //
```

```
    // Assert
```

```
    Assert.Equal(expectedAmount, balance);
```

```
}
```

*Write a failing  
acceptance test*

***Write a  
failing  
unit test***

*Make the  
test pass*

*Refactor*

```
public sealed class Account : IEntity, IAggregateRoot
{
    public Account(Guid customerId) { }

    private Amount balance;

    public void Deposit(Amount amount) {
        balance = amount;
    }

    public Amount GetCurrentBalance() {
        return balance;
    }
}
```

*Write a failing  
acceptance test*

*Write a  
failing  
unit test*

***Make the  
test pass***

*Refactor*

[Fact]

```
public void Deposit_Should_Add_Single_Transaction()
```

```
{
```

```
    //
```

```
    // Arrange
```

```
    Guid expectedCustomerId = Guid.Parse("ac608347-74ac-4607-abc2-7b95cdc8a122");
```

```
    Amount expectedAmount = new Amount(400m);
```

```
    //
```

```
    // Act
```

```
    Account sut = new Account(expectedCustomerId);
```

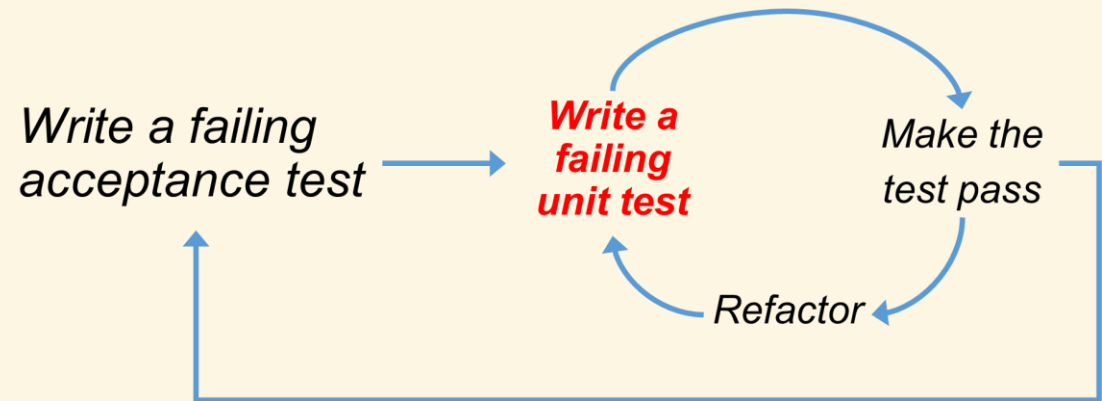
```
    sut.Deposit(expectedAmount);
```

```
    //
```

```
    // Assert
```

```
    Assert.Single(sut.Transactions.ReadOnlyCollection());
```

```
}
```

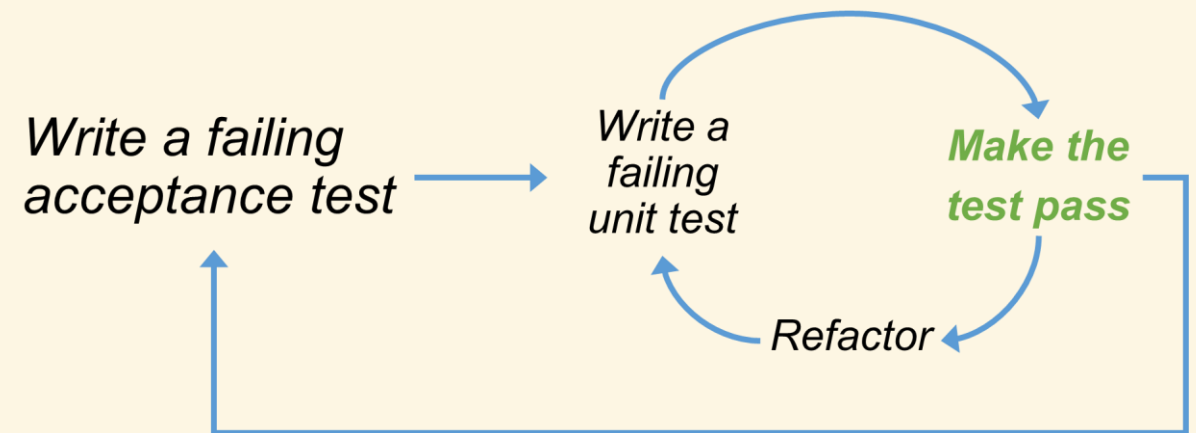




```
public sealed class Account : IEntity, IAggregateRoot
{
    public Account(Guid customerId) { }

    public void Deposit(Amount amount) {
        Credit credit = new Credit(Id, amount);
        Transactions.Add(credit);
    }

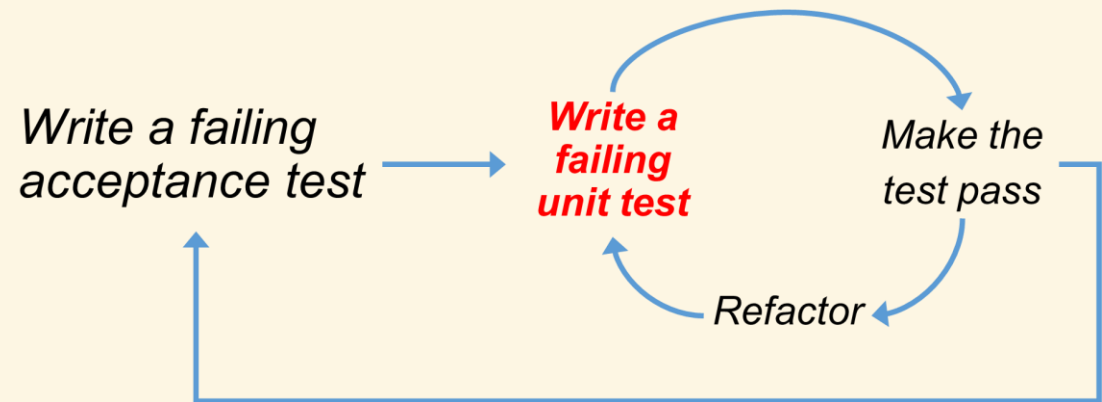
    public Amount GetCurrentBalance() {
        Amount balance = Transactions.GetBalance();
        return balance;
    }
}
```



```
[Fact]
public void NewAccount_Should_Return_The_Correct_CustomerId()
{
    //
    // Arrange
    Guid expectedCustomerId = Guid.Parse("ac608347-74ac-4607-abc2-7b95cdc8a122");
    Amount expectedAmount = new Amount(400m);

    //
    // Act
    Account sut = new Account(expectedCustomerId);

    //
    // Assert
    Assert.Equal(expectedCustomerId, sut.CustomerId);
}
```



```
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    public Account(Guid customerId) {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Transactions = new TransactionCollection();
    }

    public void Deposit(Amount amount) {
        Credit credit = new Credit(Id, amount);
        Transactions.Add(credit);
    }

    public Amount GetCurrentBalance() {
        Amount balance = Transactions.GetBalance();
        return balance;
    }
}
```

*Write a failing  
acceptance test*

*Write a  
failing  
unit test*

*Make the  
test pass*

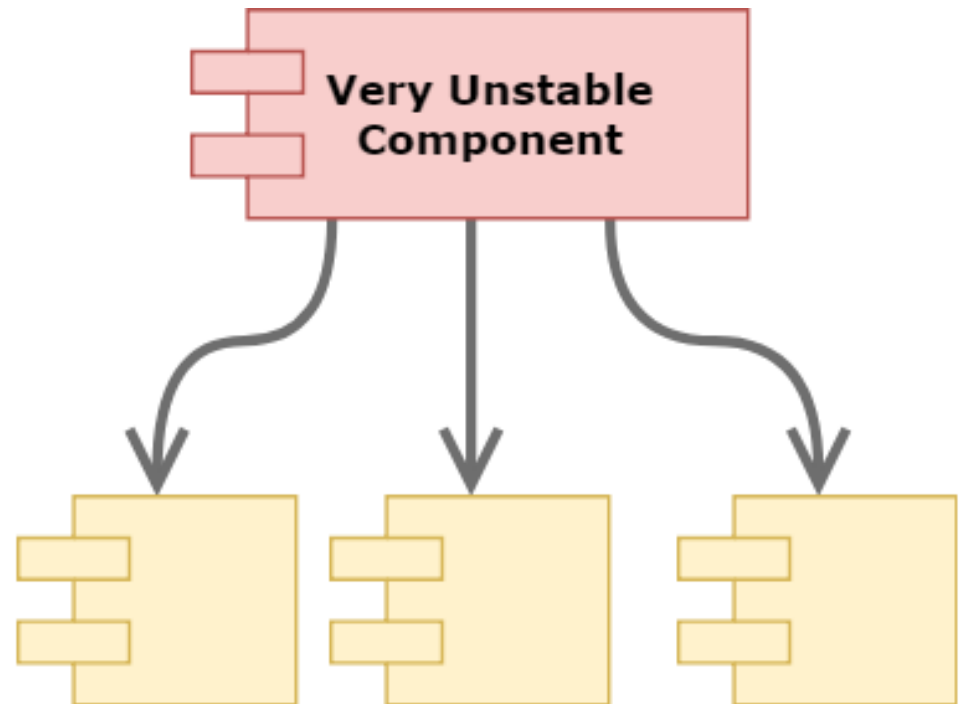
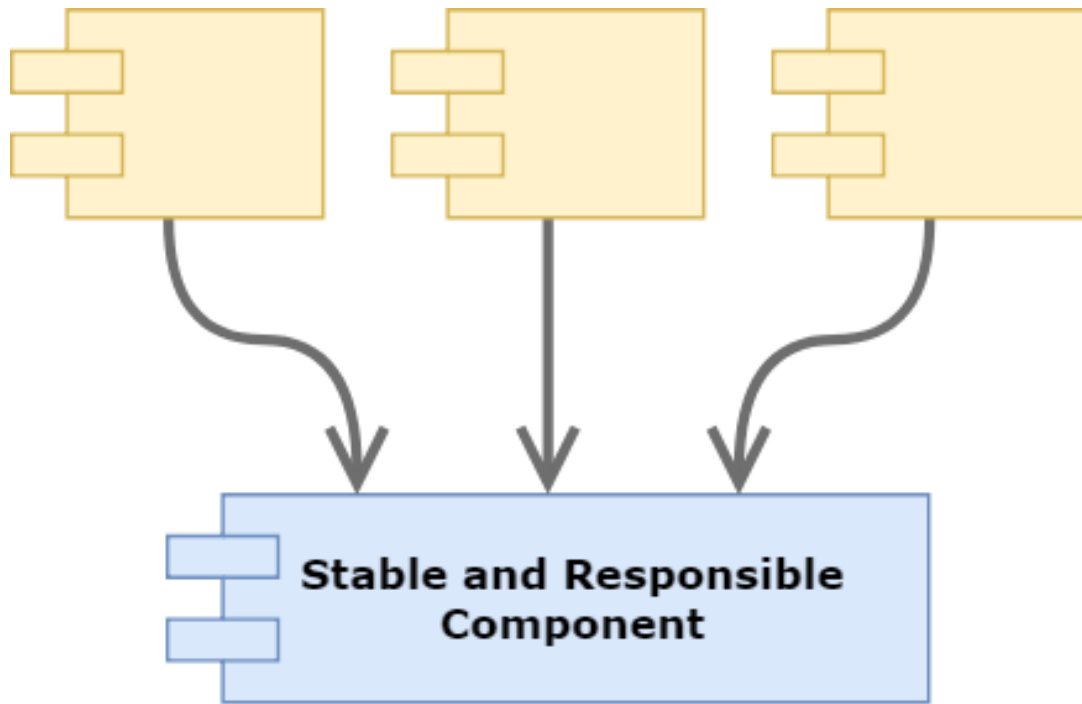
*Refactor*

# Opinionated DDD/TDD

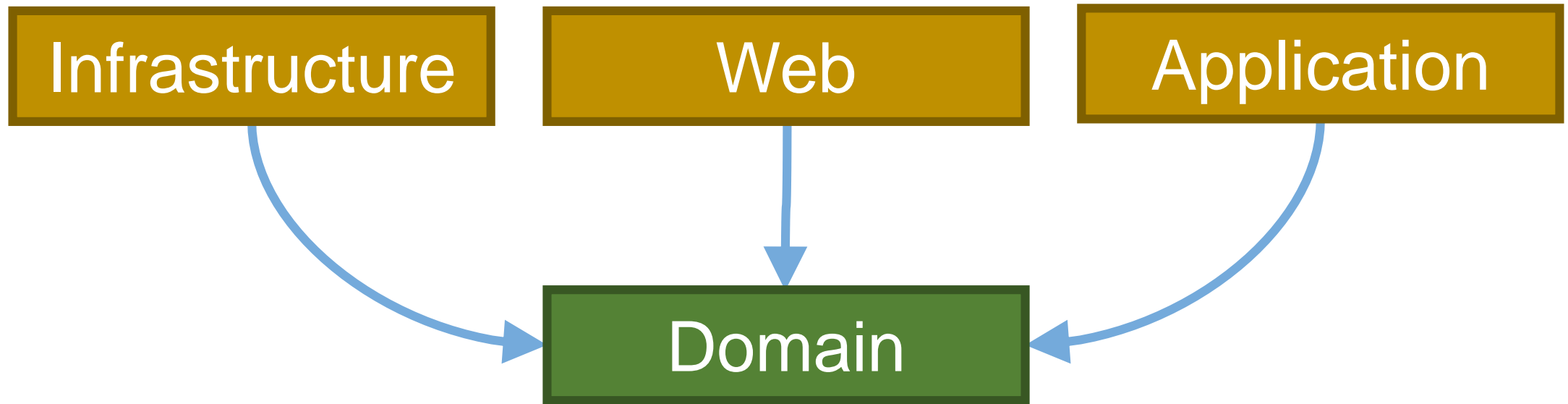
- Sometimes I implement too much of the Domain Model. Then return to covering it with unit tests.
  - By knowing the DDD patterns I underestimate the TDD value then I'm slapped in the face.
- My goal is to maintain a high test coverage on the Domain Model.
- If testing is hard. It is an architectural issue!

*I won't reverse engineer my  
data model to create a  
domain model.*

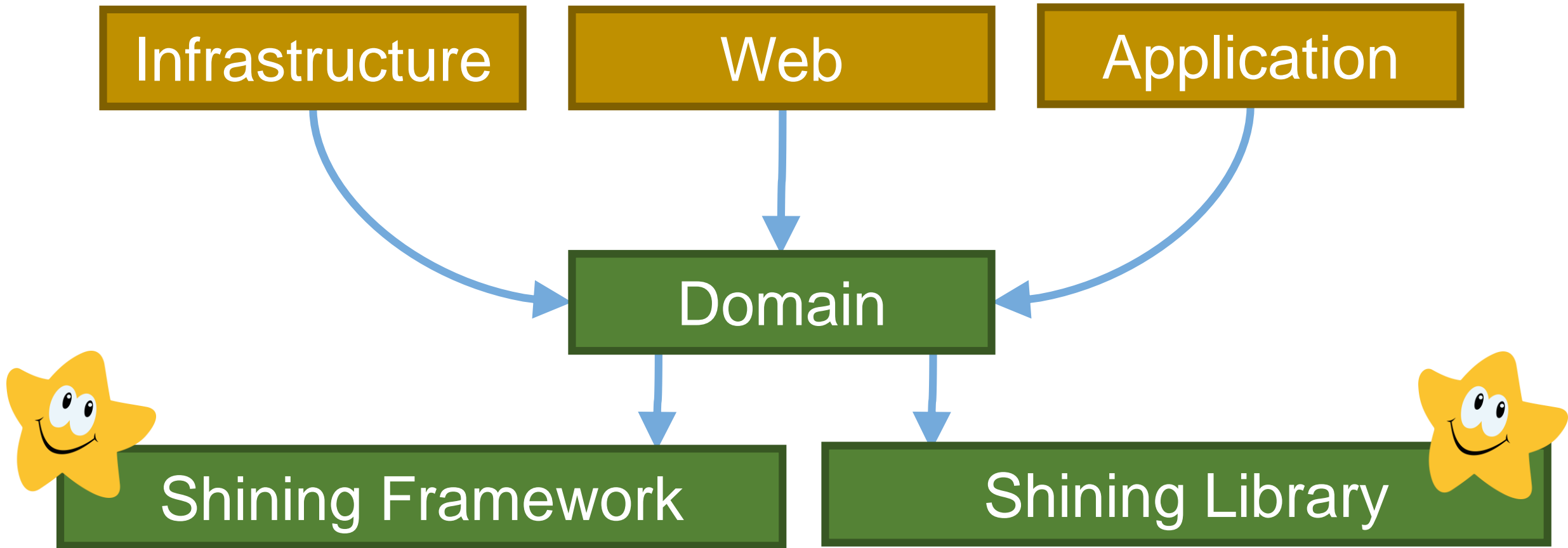
# The Stable Dependencies Principle<sup>1</sup>



# The Stable Dependencies Principle

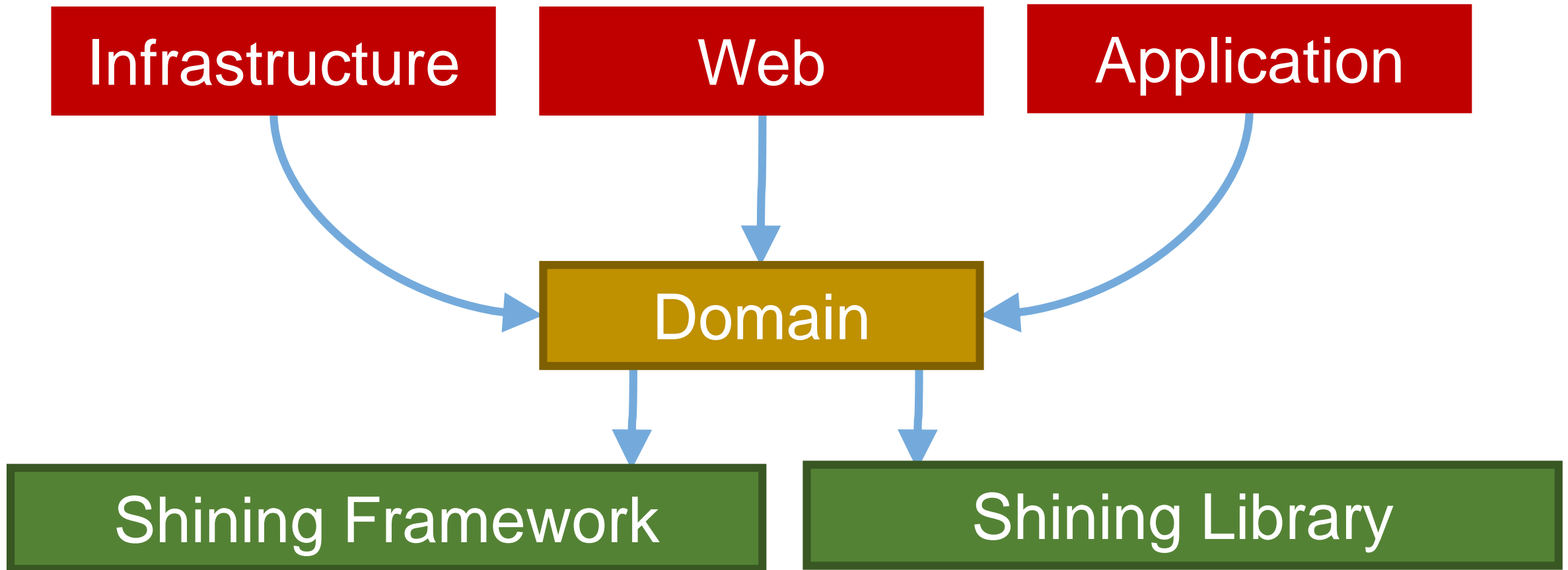


# The Stable Dependencies Principle

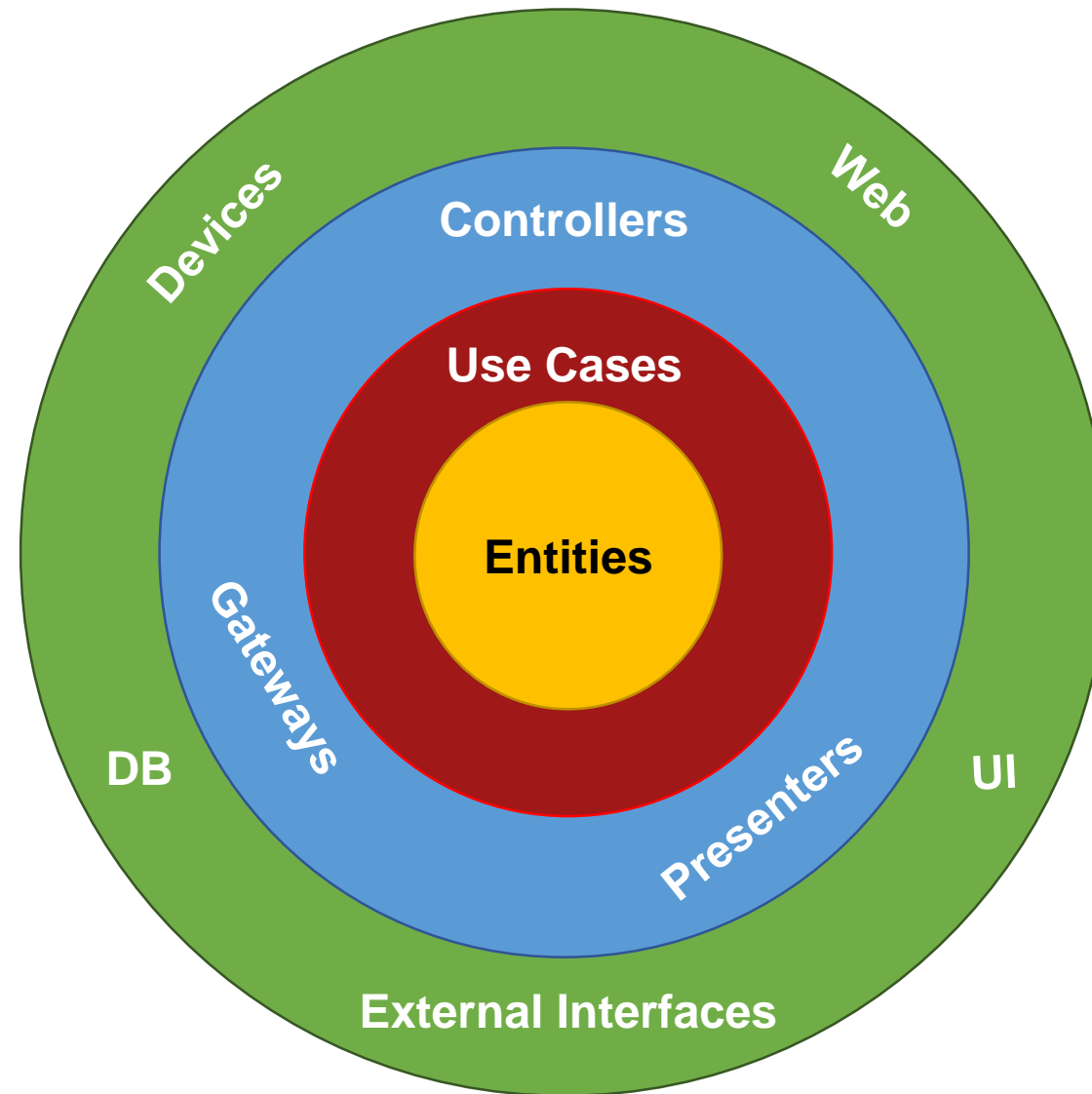




# The Stable Dependencies Principle



# Isolate the Domain with a Layered Architecture



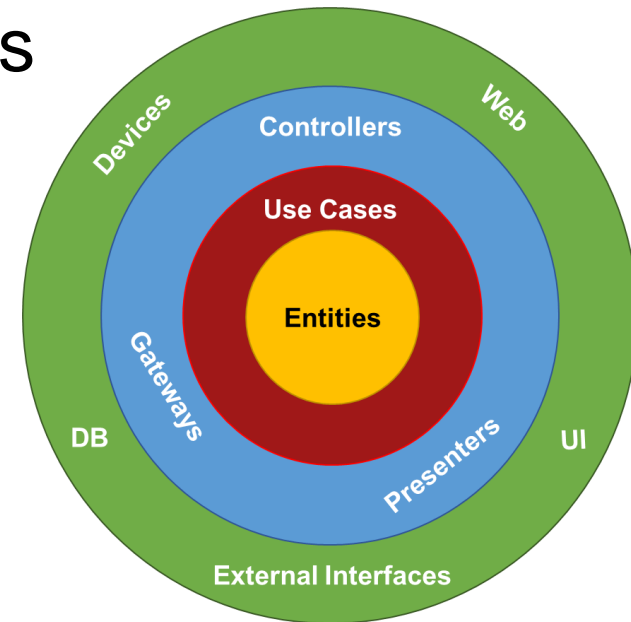
# Testing Strategies

Outside In

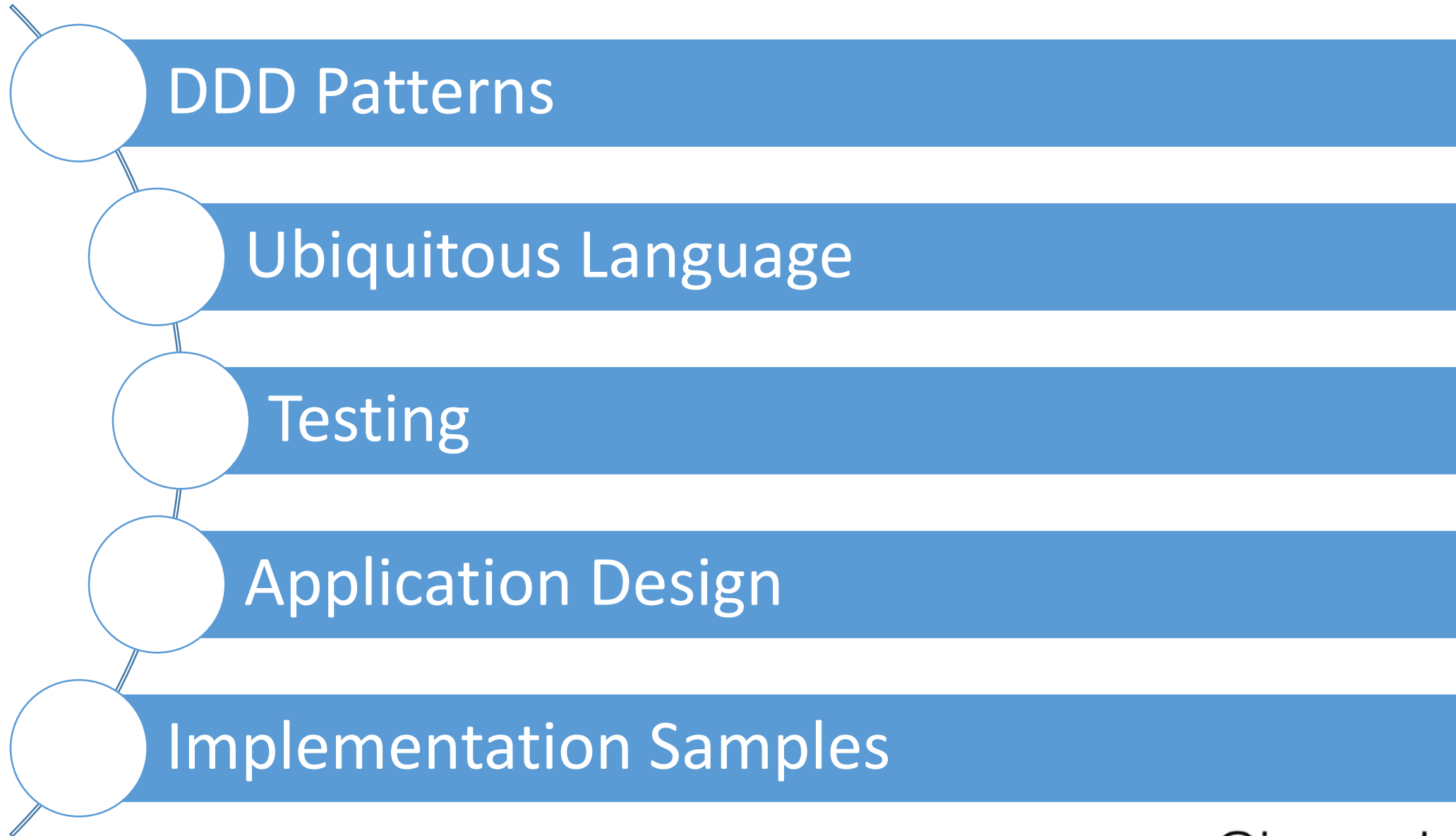
Controllers > Use Cases > Aggregates > Value Objects

Inside Out

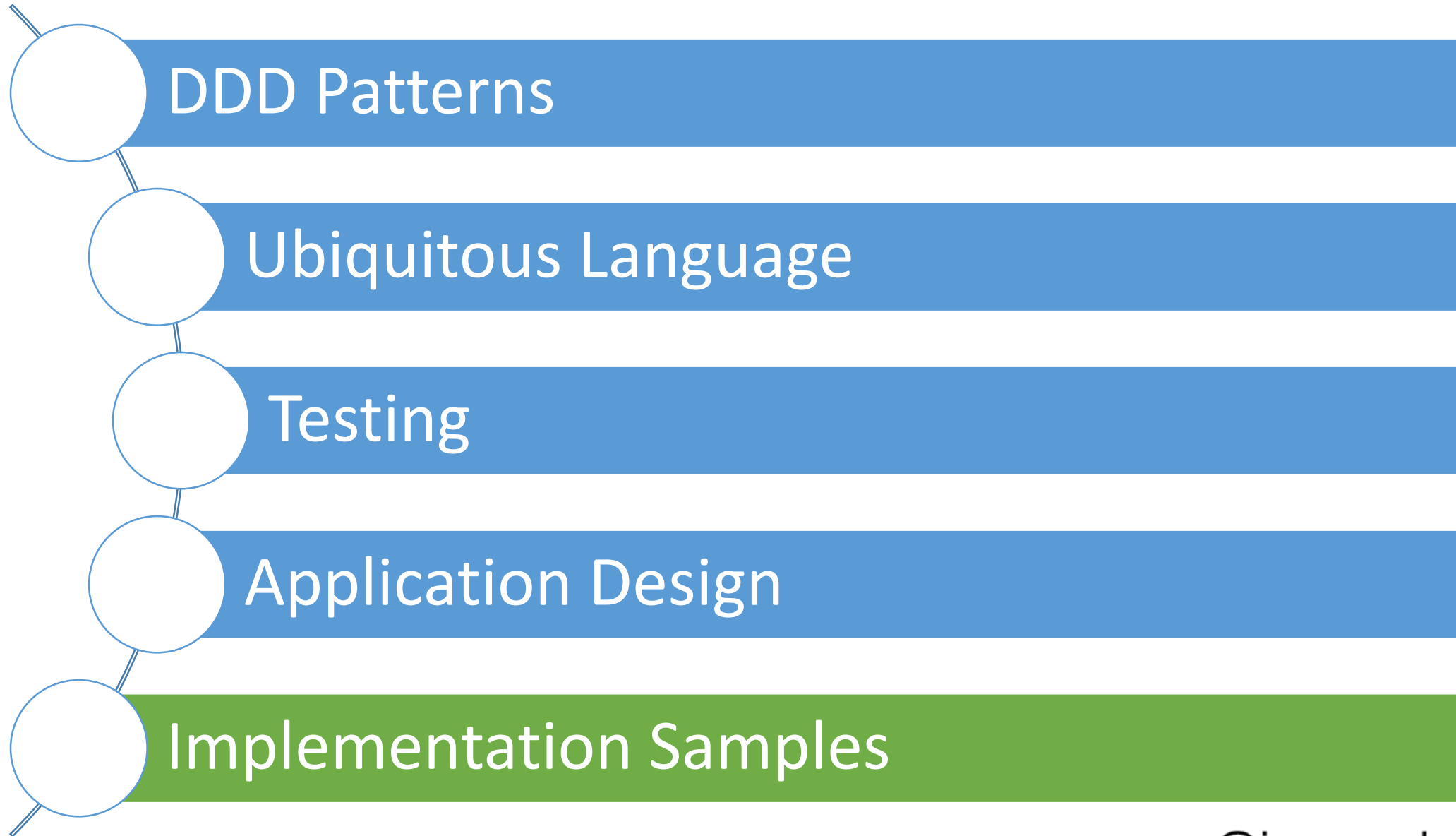
1. Aggregates > Value Objects
2. Use Cases
3. Controllers



# Quick Review

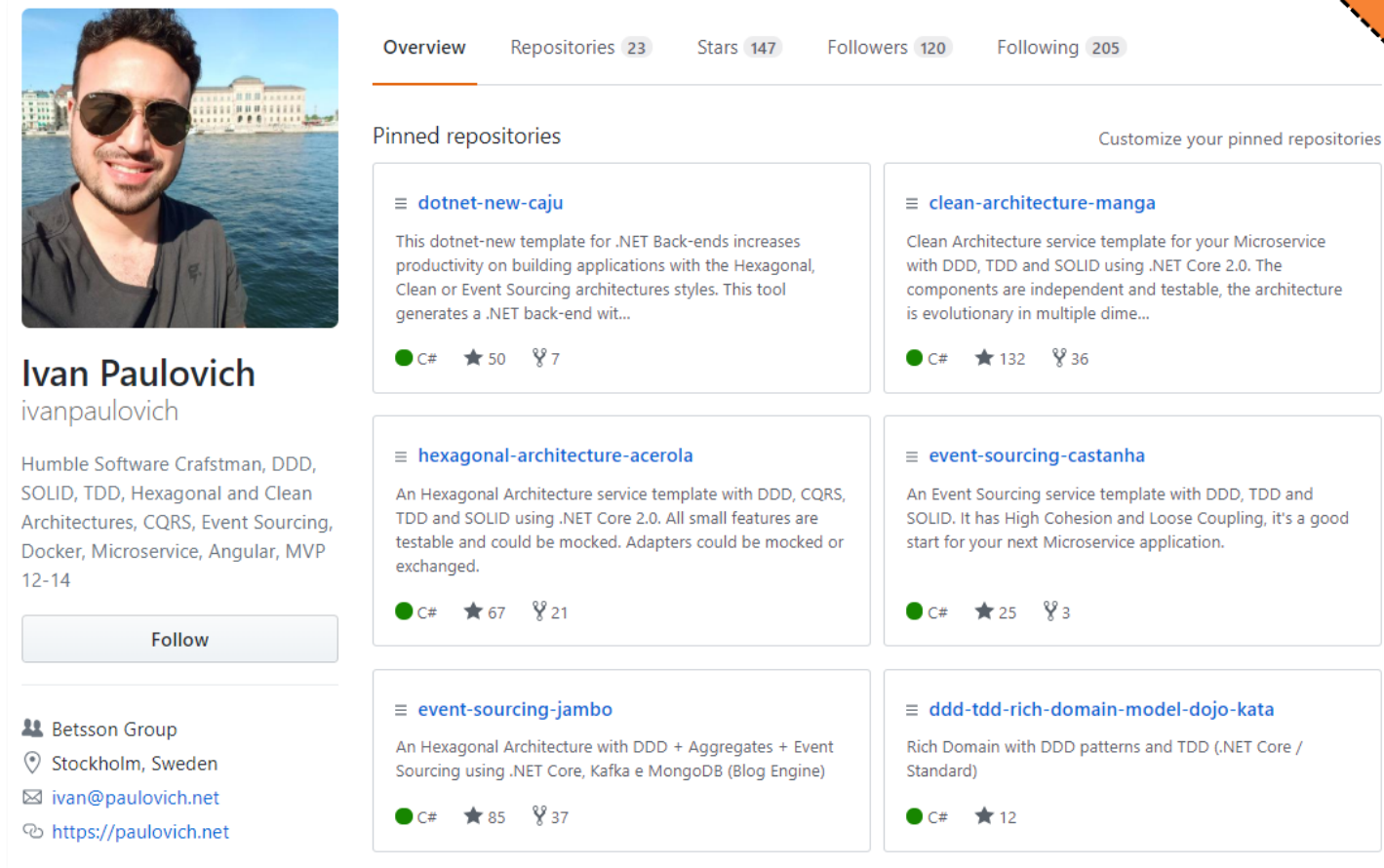


# Quick Review



# Implementation Samples

- Clean Architecture
- Hexagonal Architecture
- Event Sourcing
- DDD
- TDD
- Microservices



The screenshot shows the GitHub profile of Ivan Paulovich. The profile includes a profile picture of a man with sunglasses, a bio stating he is a 'Humble Software Craftman' with expertise in various architectural patterns, and a list of pinned repositories. The repositories are categorized by language (C#) and include star and fork counts.

**Ivan Paulovich**  
ivanpaulovich

Humble Software Craftman, DDD, SOLID, TDD, Hexagonal and Clean Architectures, CQRS, Event Sourcing, Docker, Microservice, Angular, MVP 12-14

[Follow](#)

**Pinned repositories**

- dotnet-new-caju**  
This dotnet-new template for .NET Back-ends increases productivity on building applications with the Hexagonal, Clean or Event Sourcing architectures styles. This tool generates a .NET back-end wit...  
C# ★ 50 🍴 7
- clean-architecture-manga**  
Clean Architecture service template for your Microservice with DDD, TDD and SOLID using .NET Core 2.0. The components are independent and testable, the architecture is evolutionary in multiple dime...  
C# ★ 132 🍴 36
- hexagonal-architecture-acerola**  
An Hexagonal Architecture service template with DDD, CQRS, TDD and SOLID using .NET Core 2.0. All small features are testable and could be mocked. Adapters could be mocked or exchanged.  
C# ★ 67 🍴 21
- event-sourcing-castanha**  
An Event Sourcing service template with DDD, TDD and SOLID. It has High Cohesion and Loose Coupling, it's a good start for your next Microservice application.  
C# ★ 25 🍴 3
- event-sourcing-jambo**  
An Hexagonal Architecture with DDD + Aggregates + Event Sourcing using .NET Core, Kafka e MongoDB (Blog Engine)  
C# ★ 85 🍴 37
- ddd-tdd-rich-domain-model-dojo-kata**  
Rich Domain with DDD patterns and TDD (.NET Core / Standard)  
C# ★ 12

Fork me on GitHub

# Resources

- Domain-driven Design, Eric J. Evans, 2003
- The ThoughtWorks Anthology: Essays on Software Technology and Innovation (Pragmatic Programmers), 2008
- Clean Architecture, Robert C. Martin, 2017
- Growing Object-Oriented Software, Guided by Tests, 1st Edition, 2009
- Secure by Design, Dan Bergh Johnsson, Daniel Deogun, Daniel Sawano, 2018
- Domain-Driven Design Quickly, 2007
- Effective Aggregate Design, Vaughn Vernon, 2011