

betsson group

Building Rich Domain Models with DDD and TDD

Ivan Paulovich

Betsson Dev'talk #3
Stockholm – September 12th, 2018

 <https://paulovich.net>  ivan.paulovich@betssongroup.com  @ivanpaulovich



Photo by Rafael Ruiz on Unsplash

Ivan Paulovich

Developer

betsson group



30+ Microsoft Certifications

paulovich.net

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 process.
7. Keep on whining.

Alberto Brandolini

04/59

Domain-Driven Design

Tiny Domain Objects

Frequent Rewriting

Exploratory Programming

Quick Feedback

Self Explanatory Coding

05/59

Test-Driven Development

Focus on Unit Tests

Frequent Short Cycles

Confidence 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; }
}
```

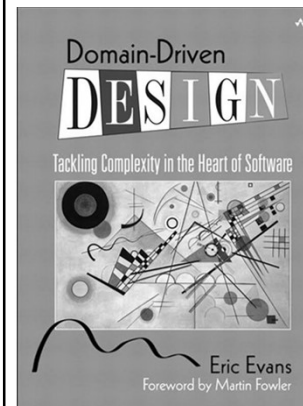
06/59

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, Data Format and Data Range.
- Services are Big and Fat.
- Easy to confuse one parameter with the another.

07/59



08/59

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

A 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; }
}
```

09/59

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 parameters values on every method.
- **Services are thinner and smaller when using Value Objects.**

10/59

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

Some Entities act as root of Aggregates.

11/59

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 current balance.

12/59

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

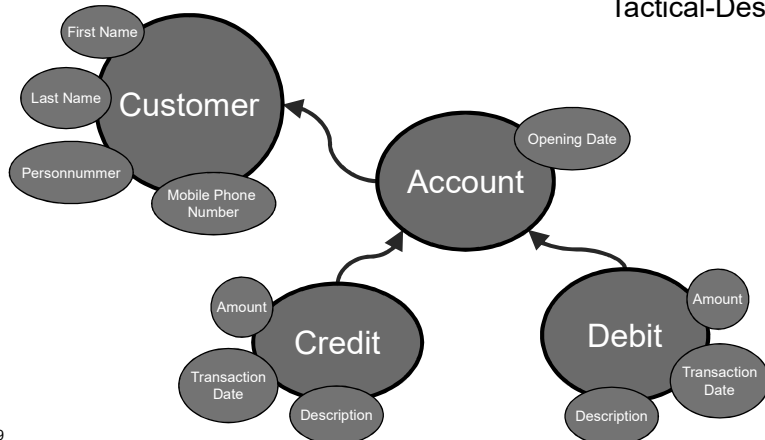
13/59

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 current balance.

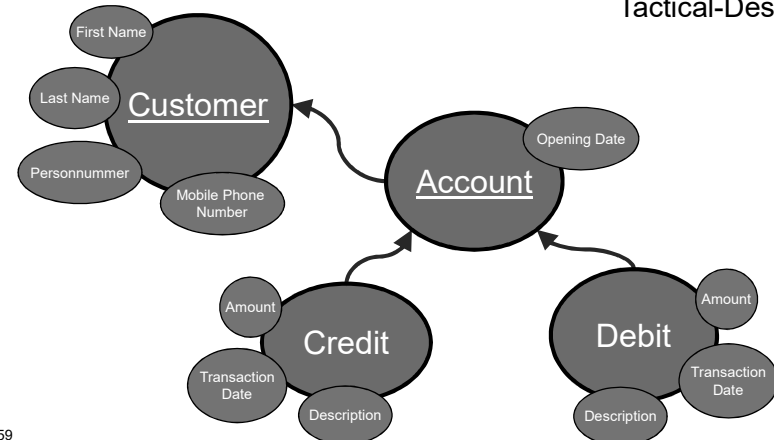
14/59

Tactical-Design

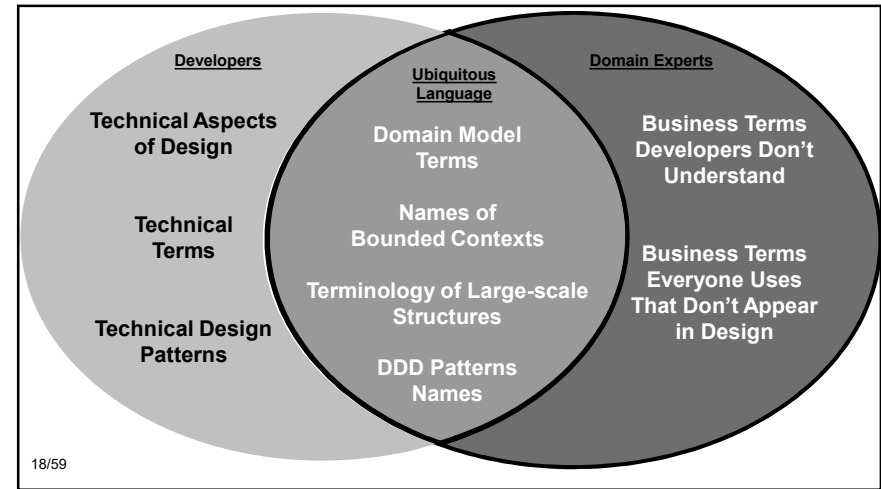
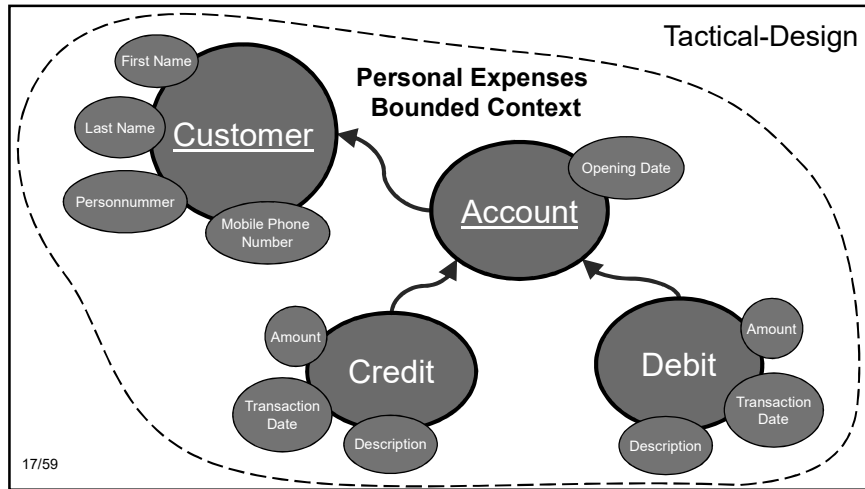


15/59

Tactical-Design

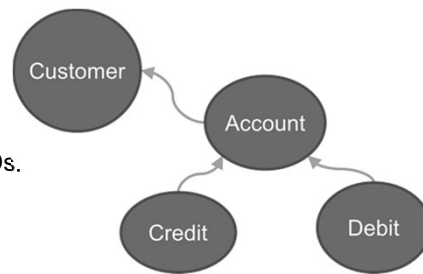


16/59



Entities

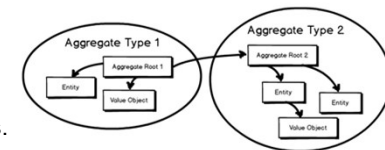
- Have a unique identity.
- Are mutable or not.
- Refer others entities by their IDs.



19/59

Aggregate Roots (Are Entities)

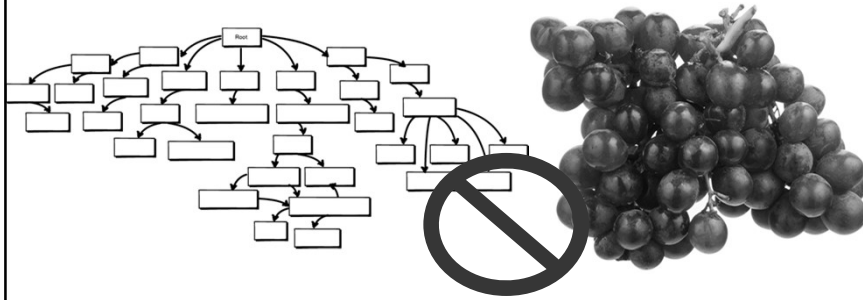
- Refer other aggregates by identity only.
- Scope of consistency inside the aggregate boundaries.
- Eventual consistency between aggregates.
- Aggregates **are small**.



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

20/59

An Aggregate Root is not your Entire Model



21/59

An Aggregate Root



22/59

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) { ... }
}
```

03/59

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) { ... }
}
```

24/59

It is an Entity

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; }
}
```

It is an Entity

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

Only mandatory fields are required in the constructor

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

```
private Account() { }
```

```
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; }
}
```

It is an Entity

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

Only mandatory fields are required in the constructor

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

Implements behaviors which maintain the state consistent.

```
private Account() { }
```

```
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; }
}
```

It is an Entity

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

Only mandatory fields are required in the constructor

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

Implements behaviors which maintain the state consistent.

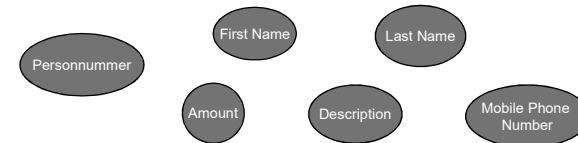
```
private Account() { }
```

Factory method to restore state.

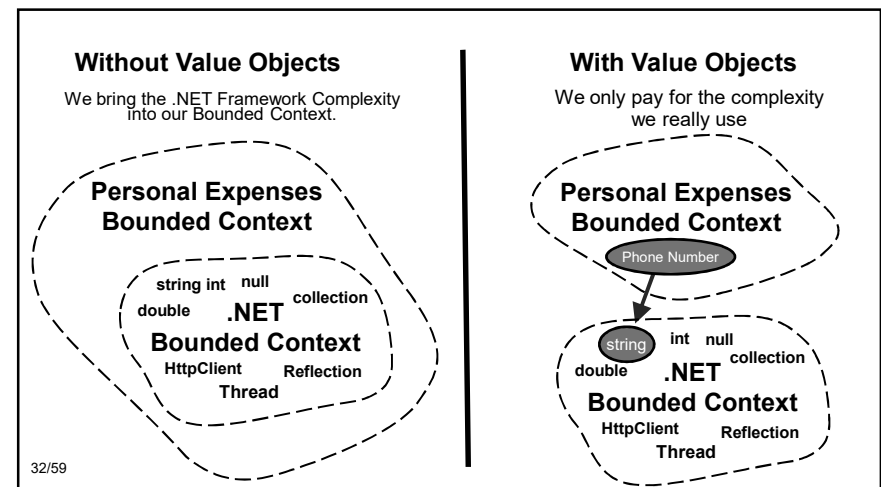
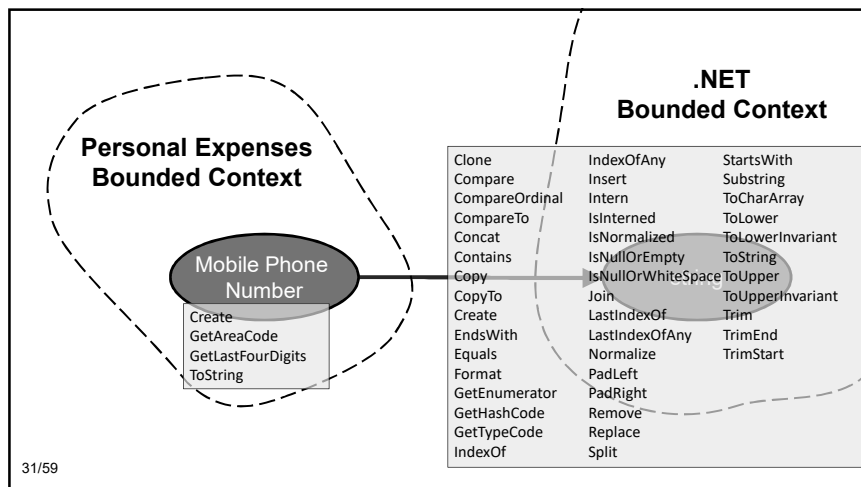
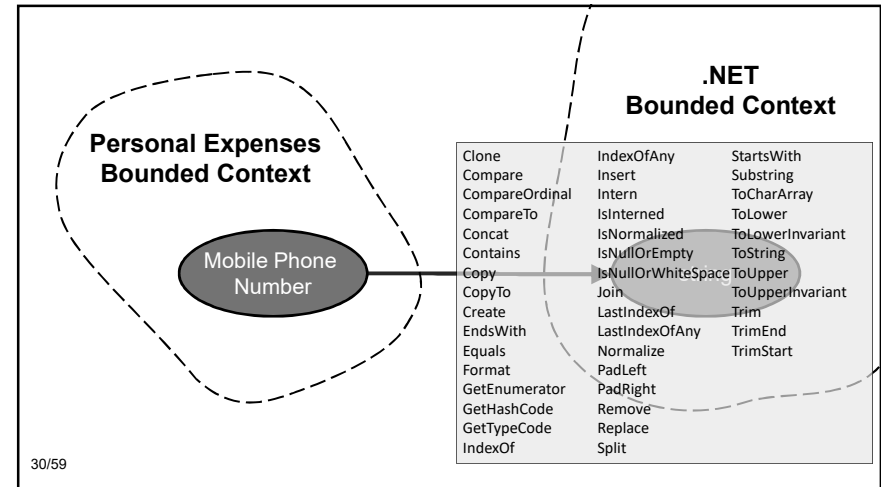
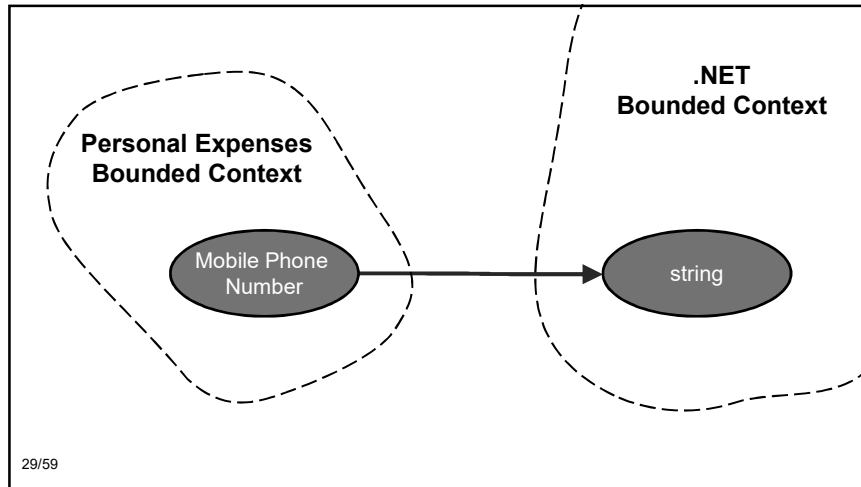
```
27/59 public static Account LoadFromDetails(Guid id, Guid customerId, TransactionCollection transactions) { ... }
}
```

Value Objects

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



28/59



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);

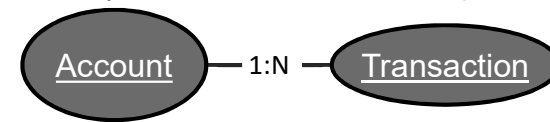
        if (!match.Success)
            throw new InvalidSSNException("Invalid Personnummer format. Use YYYYDDNNNN.");

        _text = text;
    }
}
```

33/59

First-Class Collections

- Each collection should be wrapped in its own class¹.
- Classes that contains collections do not contains any other variable.
- Behaviors have a home.
- When necessary return immutable collection copies.



34/59

¹The ThoughtWorks Anthology: Essays on Software Technology and Innovation (Pragmatic Programmers), 2008

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() { ... }
}
```

35/59

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() { ... }
}
```

36/59

Copy collections and mutable objects
when passing them between objects.¹

¹Growing Object-Oriented Software Guided by Tests, 2010

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) { ... }
}
```

37/59

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) { ... }
}
```

38/59

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);
    }

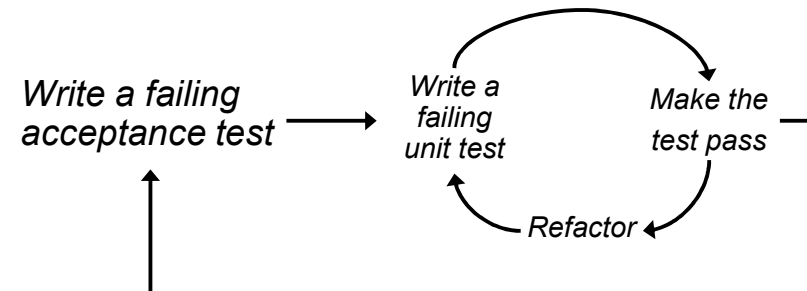
    public void Deposit(Amount amount) { ... }
}
```

39/59

Composite simpler than the sum of its parts

The GetBalance() implementation belongs to the TransactionCollection class.

Inner and outer feedback loops in TDD



40/59

```
[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.ToReadOnlyCollection());
}
```

41/59

Write a failing
acceptance testWrite a failing
unit testMake 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);
}
```

42/59

Write a failing
acceptance testWrite a failing
unit testMake 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;
    }
}
```

43/59

Write a failing
acceptance testWrite a failing
unit testMake 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.ToReadOnlyCollection());
}
```

44/59

Write a failing
acceptance testWrite a failing
unit testMake the
test pass

Refactor

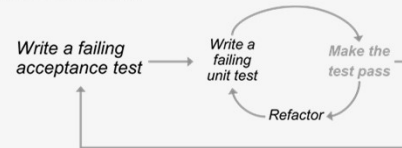
```

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;
    }
}

```



45/59

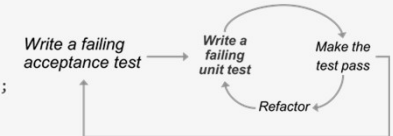
```

[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);
}

```



46/59

```

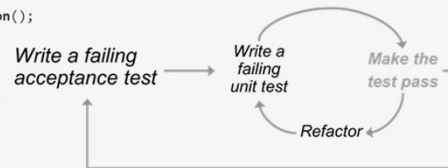
public sealed class Account : IEntity, IAggregateRoot
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public TransactionCollection Transactions { get; private set; }

    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;
    }
}

```



47/59

Opinionated DDD/TDD

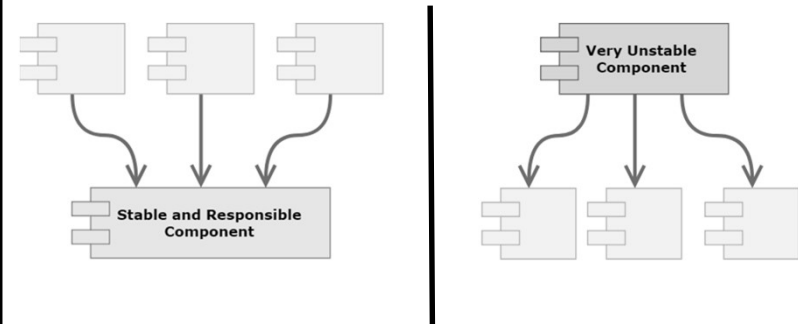
- Sometimes I implement too much of the Domain Model. Then return covering it with unit tests.
 - By knowing the DDD patterns I underestimate the TDD value then I'm slapped in the face.
- My goal is to maintain a high test coverage on the Domain Model.
- If testing is hard. It is an architectural issue!

48/59

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

49/59

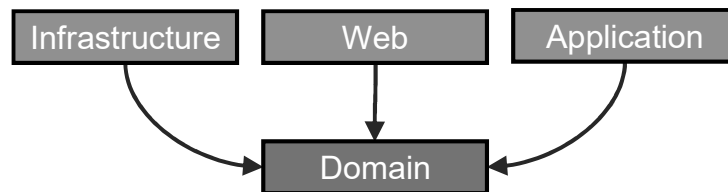
The Stable Dependencies Principle¹



50/59

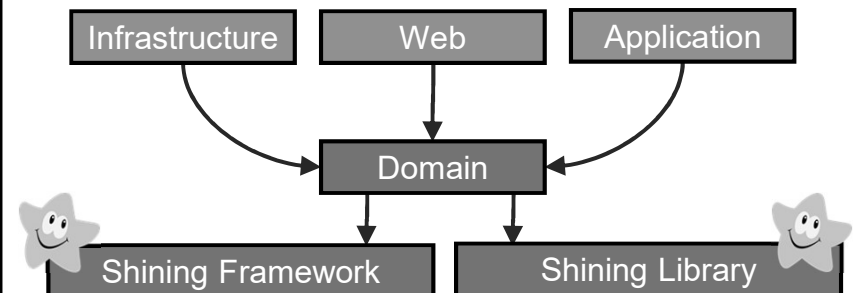
¹Clean Architecture, Robert C. Martin, 2017

The Stable Dependencies Principle



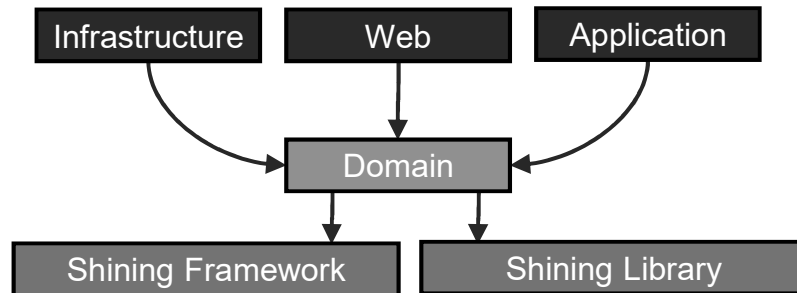
51/59

The Stable Dependencies Principle



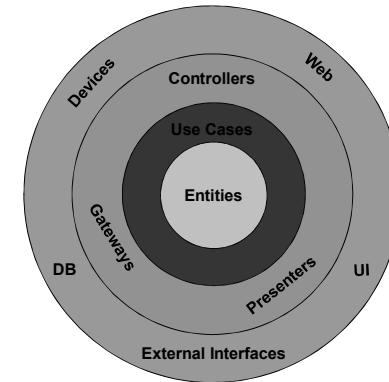
52/59

The Stable Dependencies Principle



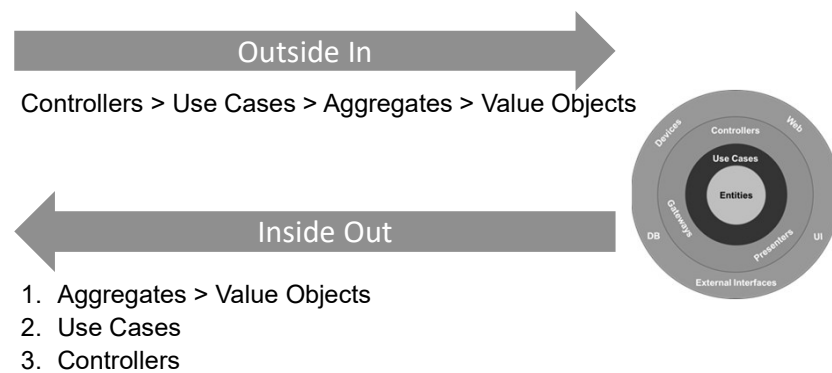
53/59

Isolate the Domain with a Layered Architecture



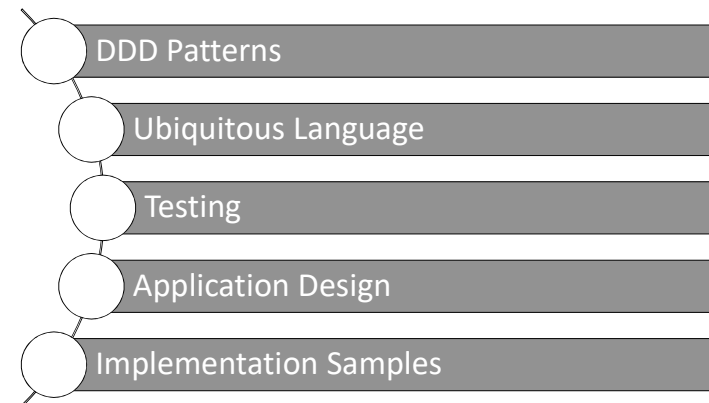
54/59

Testing Strategies



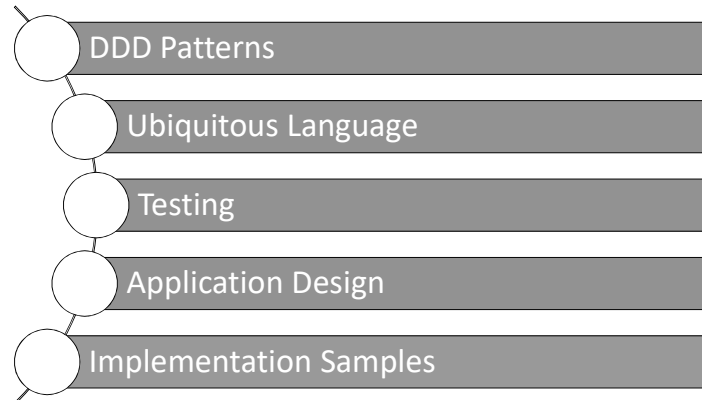
55/59

Quick Review



56/59

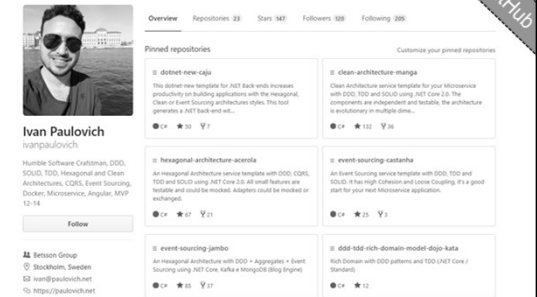
Quick Review



57/59

Implementation Samples

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



58/59

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

59/59