

HoGent

BEDRIJF
EN
ORGANISATIE

Hoofdstuk 3: Model – Unit testen

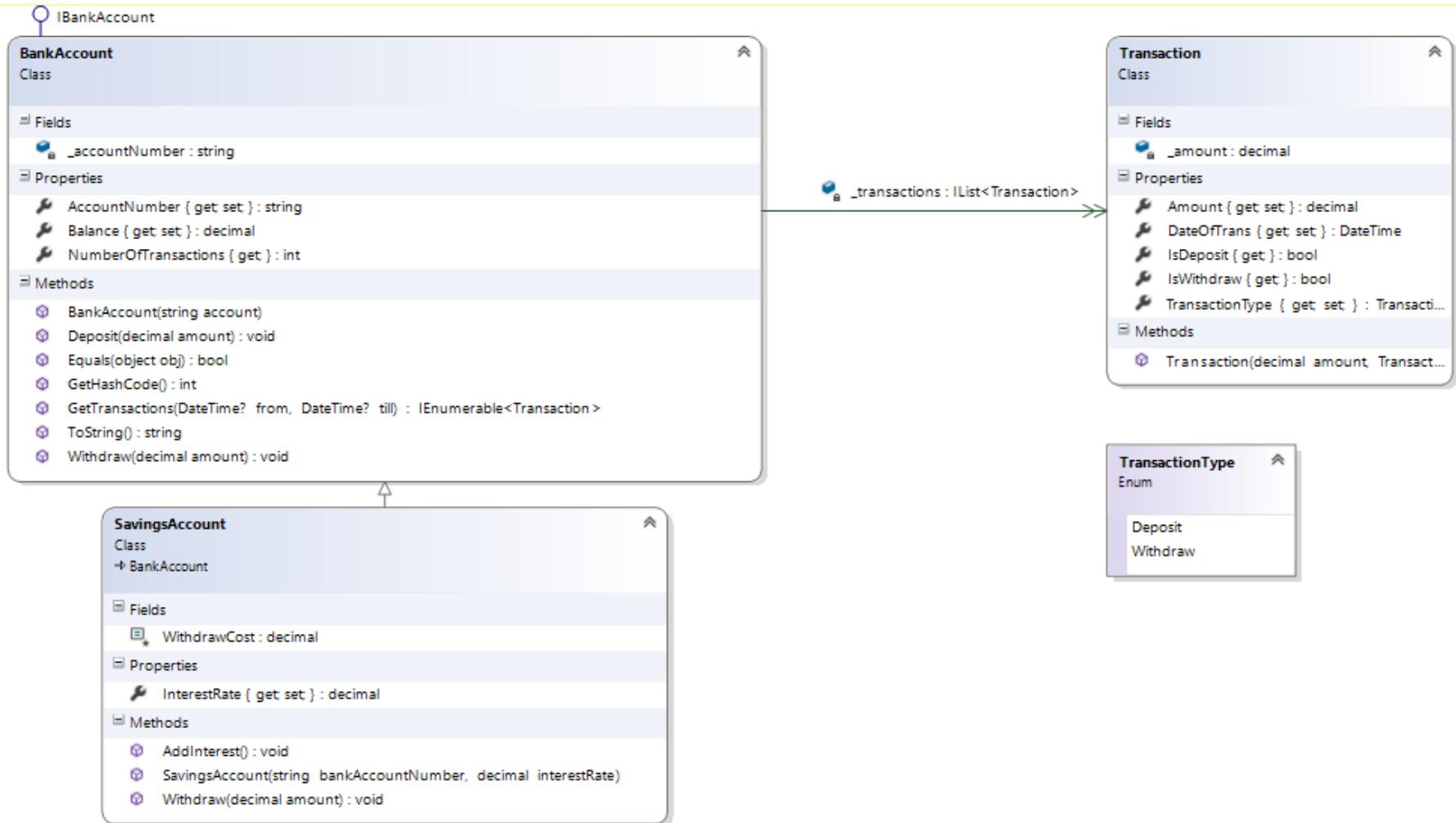
Hoofdstuk 3: Domein – Unit testen

1. Klassen
2. Associaties – collections
3. Overerving
4. Polymorfisme
5. Abstracte klasse
6. Interface
7. Statische members
8. Github
9. Unit Testen

De Banking applicatie

De Banking applicatie

▶ Het ontwerp van de domein laag



De Banking applicatie

► Sprint backlog

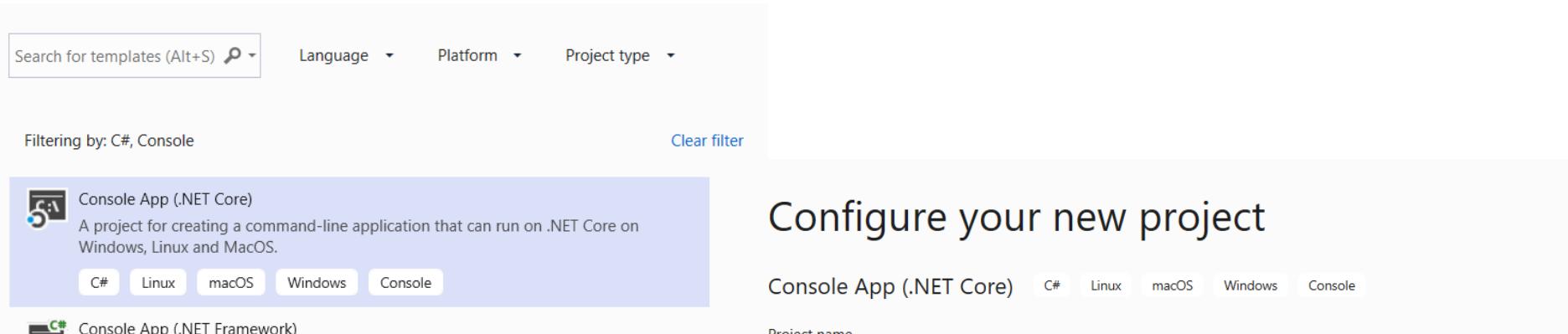
Banking applicatie ☆ Private

Maak domein aan	Unit test domein
Creëer klasse BankAccount	Creëer unit test project Banking.Tests
Creëer klasse Transaction (Associaties)	Unit test BankAccount
Creëer klasse SavingsAccount (Overerving)	Unit test Transaction, BankAccount met Transactions, SavingsAccount
Implementeer overridable methods van klasse Object	Refactor unit test BankAccountTransaction : gebruik van Theory en MemberData
Implementeer interface IBankAccount	Add a card...
Add a card...	

De Banking applicatie

▶ Aanmaken van het Banking project

- Create a new Project > C# (language) en Console (project type)
> Console App(.Net Core)
- Geef naam “Banking” in en kies een locatie. Vink place solution and project in same directory uit



Configure your new project

Console App (.NET Core) C# Linux macOS Windows Console

Project name

Banking

Location

C:\temp

Solution name ⓘ

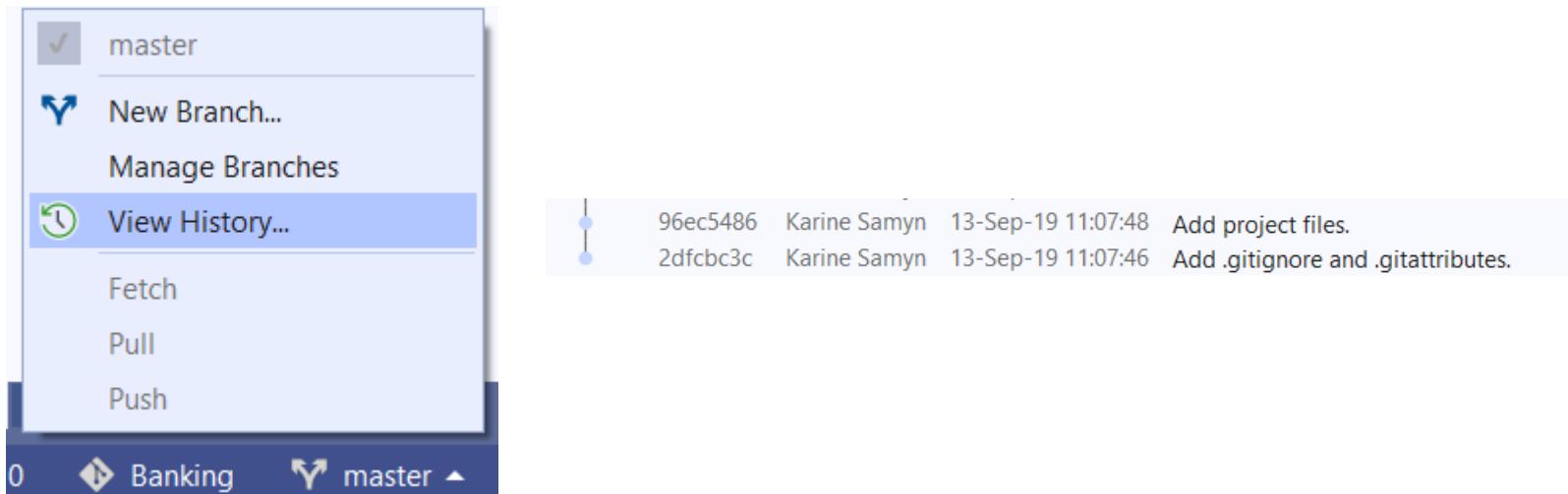
Banking

Place solution and project in the same directory

De Banking applicatie

▶ Aanmaken van het Banking project

- We splitsen de use case op in taken, die we, eens afgerond, committen. We werken in een lokale git repository. In de Solution Explorer > rechtsklik solution Banking > Add solution to source control. (of onderaan )
 - Dit creërt reeds 2 commits. Klik onderaan op master > View History



De Banking applicatie

- ▶ Aanmaken van de domein laag
 - Maak een folder “Models” aan binnen het Banking project. Daarbinnen de folder “Domain”. Dit bevat de domeinklassen. Deze zullen allen behoren tot de namespace Banking.Models.Domain
 - (rechtsklik Banking project > Add > New folder)

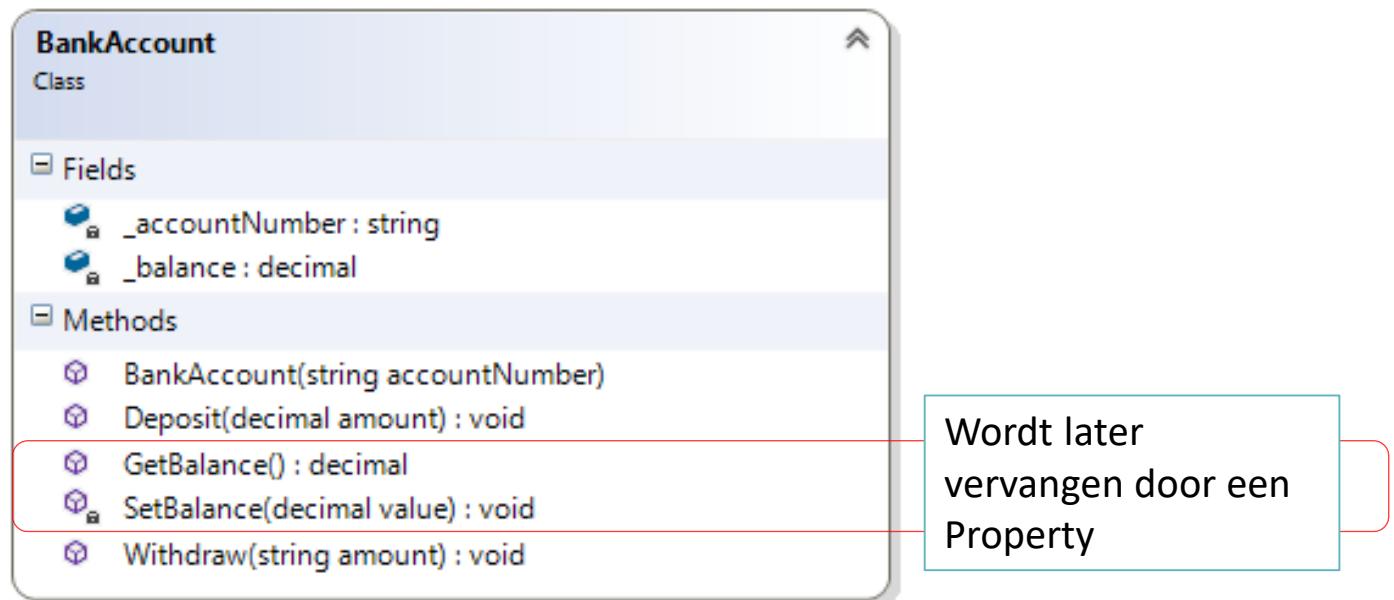
Klassen

1. Klassen

- ▶ Aanmaken van een klasse
- ▶ Members van een klasse
 - Fields
 - Methods
 - Constructor
 - Destructor
 - Properties
 - Region
- ▶ Aanmaken members van een klasse
- ▶ Gebruiken van een klasse
- ▶ Class View/Object Browser

1. Klassen

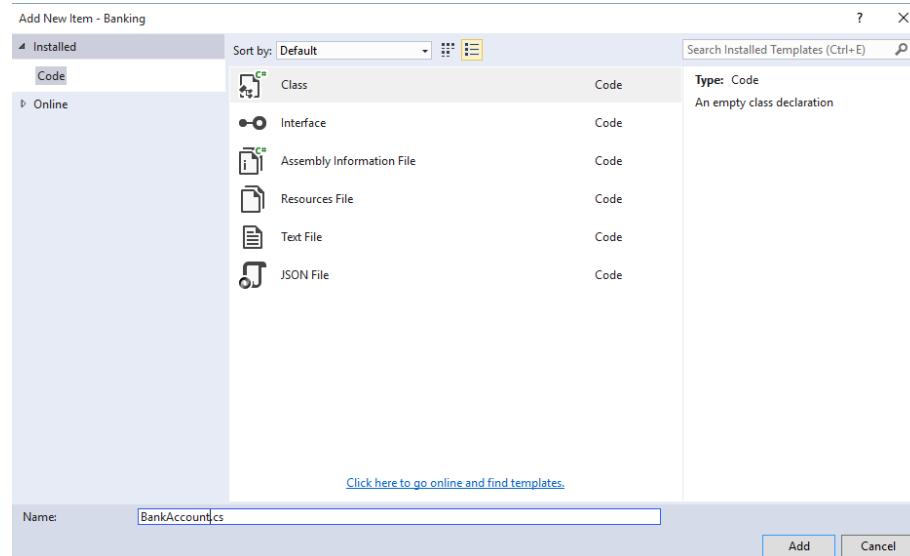
▶ Aanmaken van een domeinklasse BankAccount (eenvoudige versie)



1. Klassen

▶ Aanmaken van een domeinklasse BankAccount

- Rechtsklik op de folder Domain > Add > New Item > Class.
Geef de klasse de naam “BankAccount”
- In de folder Models/Domain wordt bestand BankAccount.cs aangemaakt



- Naming conventions:
<https://github.com/aspnet/Home/wiki/Engineering-guidelines>

1. Klassen

▶ Aanmaken van een domeinklasse

- Dubbelklik BankAccount.cs in Solution Explorer, dit opent de code editor
- De code:
 - using statements: de gebruikte assemblies
 - er staan een aantal niet gebruikte using statements. Ga er over met de muis, het lampje verschijnt en klik “Remove unnecessary usings”. Of run code cleanup.
 - namespace Banking.Models.Domain {}
 - Een namespace is een logische groepering van gerelateerde klassen (packages in Java).
 - Alle klassen in de folder Models/Domain behoren tot deze namespace.

```
namespace Banking.Models.Domain
{
    class BankAccount
    {
    }
}
```

1. Klassen

▶ Aanmaken van een domeinklasse

- class BankAccount {}: de klasse definitie
 - Access modifiers voor een “niet geneste” klasse
 - public
 - ongelimiteerd toegankelijk
 - internal
 - toegankelijk binnen de assembly
 - indien geen access modifier gebruikt wordt is dit de default

een **.NET assembly** komt ongeveer overeen met een **Java .jar** file,

.Net's internal visibility komt ongeveer overeen met package (default) visibility in combinatie met een sealed .jar

Als je unit testen wenst aan te maken voor een klasse, dient de klasse public te zijn. Unit testen behoren tot een andere namespace (zie verder)

1. Klassen

▶ Members van een klasse

- Fields (Attributen)
- Constructor – destructor
- Properties
- Methods
- Events

1. Klassen

▶ Access modifiers voor members

- **public**
 - ongelimiteerd toegankelijk
- **private**
 - enkel toegankelijk binnen de klasse
 - dit is de default
- **internal**
 - enkel toegankelijk binnen de assembly
- **protected**
 - enkel toegankelijk binnen de klasse en binnen klassen die erven van de klasse
- **Protected internal**
 - Letterlijk een combinatie van internal en protected

1. Klassen: Members

1. Fields (Attributen)

[modifier] datatype variableName

- Inkapseling van data
- Kunnen **variabelen** of **constanten** zijn
 - Attributen geven we steeds private access
- Kunnen **static** zijn
 - zijn gekoppeld aan de klasse en niet aan een instantie (object) van de klasse, ze bestaan slechts 1 maal per klasse.
- Namingconventie: **_camelCase**

```
public class BankAccount
{
    private string _accountNumber;
    private decimal _balance;
}
```

1. Klassen: Members

1. Fields (Attributen)

- Constanten
 - gebruik keyword **const**
 - een constant field **moet geïnitialiseerd worden bij declaratie**
 - na initialisatie kan de waarde van een const **nooit meer veranderen**
 - een const is implicit static:
 - je gebruikt geen static bij declaratie
 - je gebruikt de naam van de klasse om de constante op te vragen
 - Namingconventie: **Start met hoofdletter**

dit heeft geen
equivalent in Java...

```
public class BankAccount
{
    private string _accountNumber;
    private decimal _balance;
    public const decimal WithdrawCost = 0.25M;
```

```
public static void Main(string[] args)
{
    Console.WriteLine(BankAccount.WithdrawCost);
    Console.ReadKey();
}
```

1. Klassen: Members

1. Fields (Attributen)

- readonly
 - gebruik keyword **readonly**
 - aan een readonly field kan slechts 1 keer een waarde worden toegekend
 - bij **declaratie** of
 - in **constructor**
 - Hoeft niet in de declaratie <> CONST

equivalent in Java:
final

```
public class BankAccount
{
    private readonly string _accountNumber;
    private decimal _balance;
}
```

1. Klassen: Members

2. Methods

```
[modifier] return_type MethodeName ([parameters]) { ... }
```

- Operaties die een object kan uitvoeren.
- Kunnen al dan niet (void) een waarde retourneren.
- Kunnen static gedeclareerd worden.
- Bevatten parameter lijst: parameters gescheiden door een komma, parameters hebben type en naam, gebruik () indien geen parameters.
- Method overloading: je kan meerdere methodes hebben met dezelfde naam. Ze verschillen in aantal argumenten en/of type van argumenten.

```
public class BankAccount
{
    private readonly string _accountNumber;
    private decimal _balance;

    public void Deposit(decimal amount)
    {
        throw new NotImplementedException();
    }
}
```

Voor Java programmeurs
even wennen:
methodenamen starten
met een **HOOFDLETTER!**

1. Klassen: Members

2. Methodes (vervolg)

- Parameters kunnen **optioneel** zijn (geen method overloading nodig)
 - bij declaratie ken je aan een optionele parameter een defaultwaarde toe
 - voor een optionele parameter hoef je geen waarde mee te geven bij aanroep
 - optionele parameters staan als laatste in de parameterlijst
 - Intellisense gebruikt [] om optionele pars aan te duiden
 - **voorbeeld:**
 - declaratie van een methode met een optionele parameter

```
public void ExampleMethod(int required, int optionalInt = 10)
```

- aanroepen van een methode met een optionele parameter

```
ExampleMethod(5); // optionalInt uses the defaultvalue 10
```

```
ExampleMethod(5, 8); // optionalInt uses the supplied value 8
```

1. Klassen: Members

2. Methodes (vervolg)

- Optionele parameters en named arguments
 - igv een optionele parameterlijst, waar bij aanroep niet alle parameters een waarde hebben
 - **voorbeeld:**
 - declaratie van een methode met een optionele parameterlijst

```
public void ExampleMethod(int required, string optionalstr =  
    "default string", int optionalint = 10)
```

- aanroepen van een methode met sommige optionele parameters

```
ExampleMethod(5, ,3); // geeft een compilatiefout
```

```
ExampleMethod(5, optionalint : 8); // optionalInt gebruikt de  
opgegeven waarde 8, optionalstr de default value “default  
string”
```

1. Klassen: Members

2. Methodes (vervolg)

- Parameters passing kan op 3 manieren gebeuren
 - **Value** parameters: input parameter
 - **Ref** parameters: input/output parameters
 - je moet expliciet **ref** vermelden bij formele en actuele parameter
 - de variabele die je doorgeeft **moet geïnitialiseerd zijn**
 - elke verandering aan de ref-parameter in de aangeroepen methode zal ook doorgevoerd worden op de ref-parameter die werd doorgegeven
 - **Out** parameters: output parameter
 - je moet expliciet **out** vermelden bij formele en actuele parameter
 - de variabele die je doorgeeft hoeft niet geïnitialiseerd zijn
 - de aangeroepen methode **moet een waarde geven aan de out-parameter**

Java kent enkel deze vorm

```
public void Test1(int x) { x += 1; }
public void Test2(ref int x) { x += 1; }
public void Test3(out int x) { x = 10; }
```

```
int i = 0;
Test1(i);      // i heeft nu de waarde 0
Test2(ref i); // i heeft nu de waarde 1
Test3(out i); // i heeft nu de waarde 10
```

1. Klassen: Members

2. Methodes (vervolg)

- Parameters passing kan op 3 manieren gebeuren

Hoezo, in Java heb je enkel value parameters???

In Java zeggen we steeds “*Objects are passed by reference*”, dit betekent echter dat de reference van het doorgegeven object als value parameter wordt doorgegeven...

```
public static void Demonstrate(ref BankAccount bankAccount) {  
    bankAccount = null;  
}  
  
public static void ShowDemo() {  
    BankAccount myAccount = new BankAccount();  
    Console.WriteLine(" myAccount is null: {0}", myAccount == null);  
    Demonstrate(ref myAccount);  
    Console.WriteLine(" myAccount is null: {0}", myAccount == null);  
}
```

```
myAccount is null: False  
myAccount is null: True
```

*de reference naar het object myAccount wordt als ref parameter doorgegeven,
alles wat Demonstrate doet met de formele parameter bankAccount wordt ook op de actuele parameter
myAccount doorgevoerd... (laat je het ref keyword weg, dan wordt 2 maal myAccount is null: False afgeprint)*

1. Klassen: Members

2. Methodes (vervolg)

- Je kan ook een return type opgeven

```
public decimal GetBalance() {  
    return _balance;  
}
```

- return statement
 - Kan om het even waar staan in de code van de methode en kan meerdere malen voorkomen
 - Retourneert de waarde van de methode
 - Uitvoering methode wordt onmiddellijk gestopt (eventueel na uitvoering finally bij exception handling of Dispose bij using), en de controle wordt teruggegeven aan oproepend programma.

1. Klassen: Members

3. Constructor

- Een constructor heeft steeds dezelfde naam als de klasse, en heeft nooit een return type.
- Een klasse hoeft geen constructor te hebben. In dat geval maakt de compiler zelf een default constructor (public naamklasse()) aan.
- Een klasse kan 1 of meerdere constructors hebben. Ze verschillen in aantal argumenten en/of type van argumenten. In dat geval hoeft de klasse geen default constructor te hebben en maakt de compiler ook geen default constructor aan.

```
public BankAccount(string accountNumber)
{
    throw new NotImplementedException();
}

public BankAccount(string accountNumber, decimal balance) : this(accountNumber)
{
    throw new NotImplementedException();
}
```

Handige code snippet: **ctor + tab**
⇒ genereert constructor methode

Klik **Ctrl+K, Ctrl-X** voor een overzicht van alle code snippets...

In Java zet je dit als eerste statement in
de constructor body

1. Klassen: Members

3. Constructor

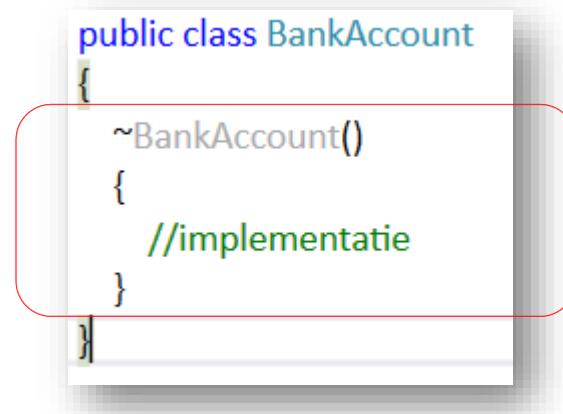
- Ook bij constructors kunnen default parameters opgegeven worden. Geen overloading nodig

```
public BankAccount(string account, decimal balance = 0M) {}
```

1. Klassen: Members

4. Destructor

- Kuist objecten op
- Wordt automatisch uitgevoerd voor de garbage collector een object vrij geeft.
- Heeft geen access modifier – geen parameters en heeft dezelfde naam als de klasse met een tilde voorafgegaan.
- Wordt zelden expliciet geschreven. Je weet ook niet wanneer het wordt uitgevoerd. Beter om IDisposable te gebruiken.



1. Klassen: Members

5. Properties

- combinatie van aspecten van fields en methods
 - voor de gebruiker van een klasse is een property net een field
 - voor diegene die een property implementeert bestaat een property uit 1 of 2 stukjes code die de **getter en/of setter** voorstellen
 - de code voor de getter wordt uitgevoerd wanneer de property wordt gelezen
 - de code voor de setter wordt uitgevoerd als aan de property een waarde wordt toegekend

1. Klassen: Members

5. Properties

```
public class BankAccount
{
    private string _accountNumber;

    public string AccountNumber
    {
        get { return _accountNumber; }

        set { _accountNumber = value; }
    }
}
```

- dit is een C# keyword
- het type van **value** is het type van de property, in dit voorbeeld dus string
- het bevat de waarde die de gebruiker wil toekennen aan de property

- de property **noemt** AccountNumber, de naam van een property start steeds met een hoofdletter!
- het **type** van de property is string

- dit stukje code bij **get** wordt uitgevoerd wanneer de property wordt **gelezen**, bv.
`string accountNumber = myBankAccount.AccountNumber;`

- dit stukje code bij **set** wordt uitgevoerd wanneer aan de property een waarde wordt **toegekend**, bv.
`myBankAccount.AccountNumber = "12-456376-25";`

1. Klassen: Members

5. Properties:

JAVA



C#

```
public class BankAccount {  
    private String accountNumber;  
    public String getAccountNumber() {  
        return accountNumber; }  
    public void setAccountNumber(String value) {  
        accountNumber = value; }  
}
```

```
public class BankAccount {  
    private string _accountNumber;  
    public string AccountNumber  
    {  
        get { return _accountNumber; }  
        set { _accountNumber = value; }  
    }  
}
```

```
BankAccount myBankAccount =  
    new BankAccount("13-455665-13");
```

```
String accountNumber =  
    myBankAccount.getAccountNumber();
```

```
myBankAccount.setAccountNumber("12-456376-25");
```

```
BankAccount myBankAccount =  
    new BankAccount("13-455665-13");
```

```
string accountNumber =  
    myBankAccount.AccountNumber;
```

```
myBankAccount.AccountNumber = "12-456376-25";
```

1. Klassen: Members

5. Properties

- Hoeven niet steeds een get en een set te bevatten
 - **read-only property**: heeft enkel een get.
 - **write-only property**: heeft enkel een set
- get/set nemen per default het access level aan van de property, maar dit kunnen we veranderen
- **voorbeeld**

```
private decimal _balance;
```

```
public decimal Balance
```

```
{
```

```
    get { return _balance; }
```

```
    private set { _balance = value; }
```

```
}
```

public get: de get heeft geen expliciete access modifier en neemt het access level van de property over

private set: buiten deze klasse is het niet toegestaan de balans rechtstreeks te wijzigen

1. Klassen: Members

5. Properties – automatic properties

- properties hoeven niet expliciet gebruik te maken van een field
- er is een verkorte schrijfwijze voor properties
 - de compiler maakt dan achter de schermen gebruik van een field
- **voorbeeld**

```
public decimal Balance { get; set; }
```

er worden geen code blokken gedeclareerd voor get/set,
de compiler houdt nu zelf een private decimal field _balance bij
dit field is niet rechtstreeks beschikbaar voor de programmeur

uitvoering van set wordt achter de schermen vertaald naar:

compiler_generated_field_for_Balance = value

uitvoering van get wordt achter de schermen vertaald naar:

return compiler_generated_field_for_Balance

1. Klassen: Members

5. Properties

- bij automatic properties kan je ook het access level aanpassen
- **voorbeeld**

```
public decimal Balance { get; private set; }
```

- handige code snippet voor automatic property: prop + tab tab
- handige shortcut om voor een field een property te maken
 - selecteer field > rechtsklik > Quick Actions and Refactorings ... > Encapsulate Field

1. Klassen: Members

5. Properties

- Auto-Implemented Property Initializers
- Je kan ook de initiële waarde van een property opgeven. Zo hoef je dit niet in de constructor te doen
- Voorbeeld

```
public decimal Balance { get; private set; } = 0M;
```

- Een **read-only** property heeft enkel een getter. De waarde kan je opgeven via een auto-property initializer of in de constructor

```
public string AccountNumber { get; }

public BankAccount(string accountNumber)
{
    AccountNumber = accountNumber;
}
```

1. Klassen: Members

6. Regions

- Dienen om code te groeperen
 - Een region kan je open- en dichtklappen
- Aanmaken: selecteer een stukje code selecteren > Rechtsklik > Snippet > Surround with > Visual C# > #region of typ de code in.
- Good practice: voorzie in een klasse minstens **4 regions: Fields, Constructors, Methods, Properties**

The screenshot shows a portion of a Visual Studio code editor. A class named 'BankAccount' is defined with the following code:

```
public class BankAccount
{
    #region Fields
    private string _accountNumber;
    #endregion

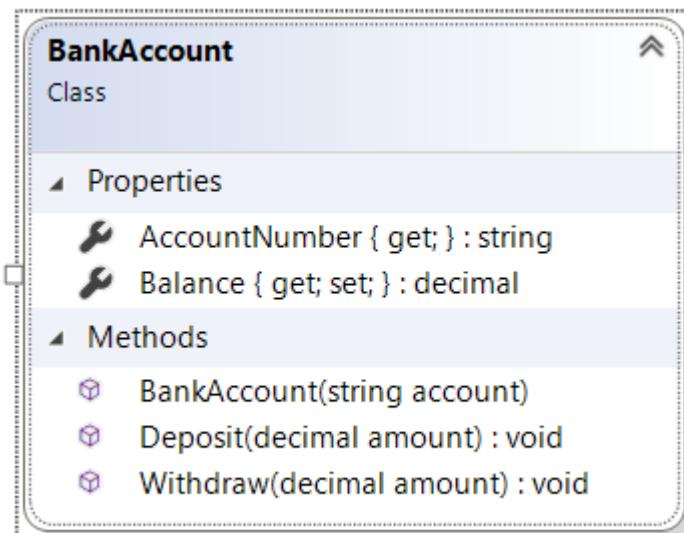
    Properties
    Constructors
    Methods
}
```

The code editor highlights the class name 'BankAccount' and the opening brace '{' in blue. The region 'Fields' is currently expanded, showing the declaration of a private string variable '_accountNumber'. Below the class definition, there are three buttons labeled 'Properties', 'Constructors', and 'Methods', which are part of the region structure. The closing brace '}' at the bottom is also highlighted in blue.

1. Klassen

► Aan de slag nu...

- Implementeer de klasse
- Om een klassendiagram toe te voegen : rechtsklik folder Domain > Add > new Item > Class Diagram.
 - Selecteer BankAccount.cs in de Solution Explorer en drop het in de editor.
 - Aanpassen door Rechtsklik > Add > property,... => de code in de klasse wordt automatisch aangepast



1. Klassen

- ▶ Aan de slag nu...
 - Implementeer de klasse

```
namespace Banking.Models.Domain
{
    class BankAccount
    {
        Fields

        #region Properties
        public decimal Balance { get; private set; }

        public string AccountNumber { get; }
        #endregion

        #region Constructors
        public BankAccount(string account)
        {
            AccountNumber = account;
            Balance = Decimal.Zero;
        }
        #endregion

        Methods
        public void Withdraw(decimal amount)
        {
            Balance -= amount;
        }

        public void Deposit(decimal amount)
        {
            Balance += amount;
        }
        #endregion
    }
}
```

1. Klassen

- ▶ Gebruik van klassen (ga naar Program.cs)
 - Declaratie en instantiatie van een variabele van het type BankAccount

```
public static void Main(string[] args)
{
    BankAccount myAccount;
    myAccount = new BankAccount("123-123123-12");
}
```

- Declaratie en instantiatie kan in 1 statement

```
BankAccount myAccount = new BankAccount("123-123123-12");
```

1. Klassen

▶ Object Initializers

- Waarden toekennen aan properties van een object, tijdens de instantiatie van het object
- Instantiatie en initialisatie zonder object initializer:

```
BankAccount myAccount =  
    new BankAccount("123-123123-12");  
myAccount.Balance = 200M;
```

- analoog maar **met object initializer**:

```
BankAccount myAccount =  
    new BankAccount("123-123123-12") { Balance = 200M };
```

```
class BankAccount  
{  
    #region Properties  
    public decimal Balance { get; set; }  
  
    public string AccountNumber { get; }  
    #endregion  
  
    #region Constructors  
    public BankAccount(string account)  
    {  
        AccountNumber = account;  
        Balance = Decimal.Zero;  
    }  
    #endregion
```

Na de constructor aanroep volgt een sequentie van member initializers: tussen {} en gescheiden door een komma. Voor de default constructor mag je de haakjes bij de aanroep weglaten.

1. Klassen

► Gebruik van klassen

- Uitvoeren van een methode

- void

```
myAccount.Deposit(100.0M);
```

- met return waarde

```
string accountInfo = myAccount.ToString();
```

- Gebruik van properties:

- de compiler bepaalt zelf wanneer get of set wordt uitgevoerd (kan afgeleid worden uit plaats in code)

- Uitvoeren **get** (opvragen inhoud)

```
string balance = myAccount.Balance.ToString();
```

- Uitvoeren **set** (instellen inhoud) (enkel indien setter public)

```
myAccount.Balance = 100;
```

- Beide (uitvoeren **get** en dan **set**)

```
myAccount.Balance += 100;
```

1. Klassen

► Gebruik van klassen

- Ga naar Program.cs en voeg onderstaande code toe
- Run de applicatie

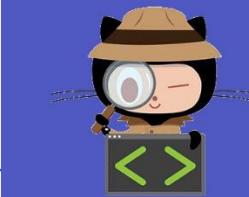
```
public class Program
{
    public static void Main(string[] args)
    {
        BankAccount account = new BankAccount("123-4567890-02");
        Console.WriteLine($"AccountNumber: {account.AccountNumber}");
        Console.WriteLine($"Balance: {account.Balance}");
        account.Deposit(200M);
        Console.WriteLine($"Balance after deposit of 200 euros: {account.Balance}");
        account.Withdraw(100M);
        Console.WriteLine($"Balance after withdraw 100 euros: {account.Balance}");
        Console.ReadKey();
    }
}
```

1. Klassen

▶ Class View/Object Browser

- In menu kies View > Class View/Object Browser
- Class View: Geeft de klasse structuur van je project weer, de namespaces en de klassen in de vorm van een boomstructuur.
Bekijken van types in huidig project
- Object Browser: ook inspecteren van referenced assemblies

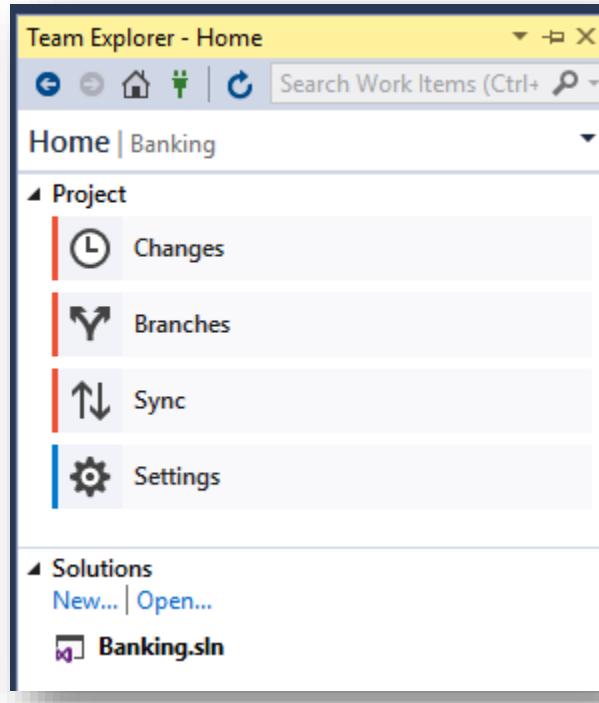
1. Klassen



commit "Add class BankAccount"

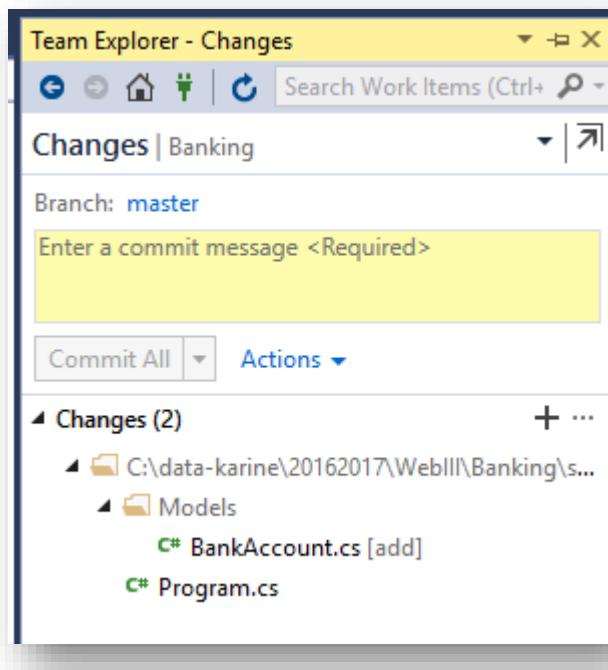
▶ Aanmaken commit in de lokale repository

- BankAccount is aangemaakt
- We gaan dit committen in de lokale repository
- Open Team Explorer (View > Team Explorer)



1. Klassen

- ▶ Aanmaken commit in de lokale repository
 - Klik op Changes



1. Klassen

- ▶ Aanmaken commit in de lokale repository
 - Alvorens je commit, inspecteer je de code (**code review**)
 - Open BankAccount.cs
 - Overloop de code
 - Bekijk nog eens de warnings en messages en pas code cleanup toe

The screenshot shows a code editor with the following code:

```
using System;  
  
namespace Banking.Models.Domain  
{  
    class BankAccount  
    {  
        Properties  
        Constructors  
        Methods  
    }  
}
```

A vertical dashed line highlights the class definition. Two arrows point from the bottom right towards the inspection results at the bottom of the screen.

No issues found

Pag. 46

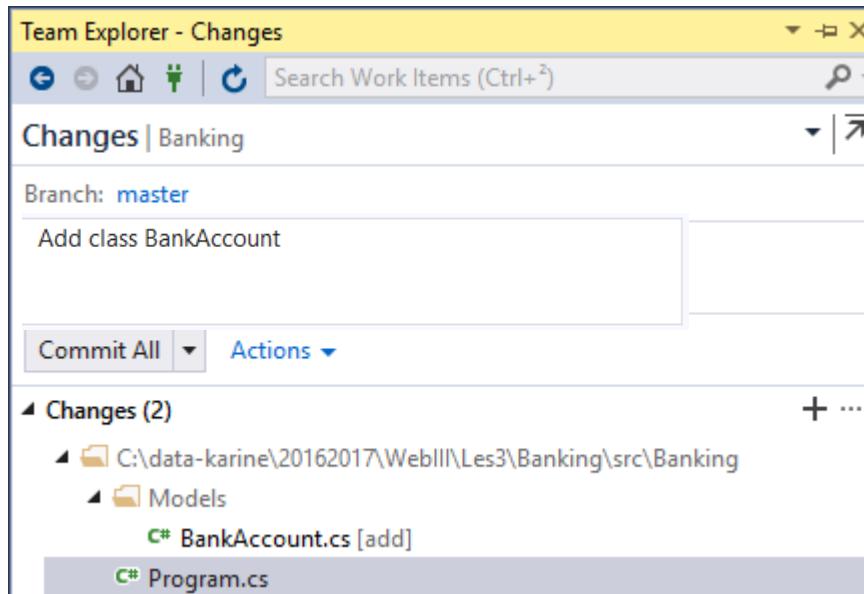
1. Klassen

- ▶ Aanmaken commit in de lokale repository
 - Alvorens je commit, inspecteer de code
 - Open Program.cs
 - Team Explorer toont de verschillen. Overloop deze
 - Code cleanup

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5
6 namespace Banking
7 {
8     public class Program
9     {
10         public static void Main(string[] args)
11         {
12             BankAccount account = new BankAccount();
13             Console.WriteLine($"AccountNumber: {account.AccountNumber}");
14             Console.WriteLine($"Balance: {account.Balance}");
15             account.Deposit(200M);
16             Console.WriteLine($"Balance after deposit");
17             account.Withdraw(100M);
18             Console.WriteLine($"Balance after withdraw");
19         }
20     }
21 }
```

1. Klassen

- ▶ Aanmaken commit in de lokale repository
 - Vul de commit boodschap in en klik Commit All

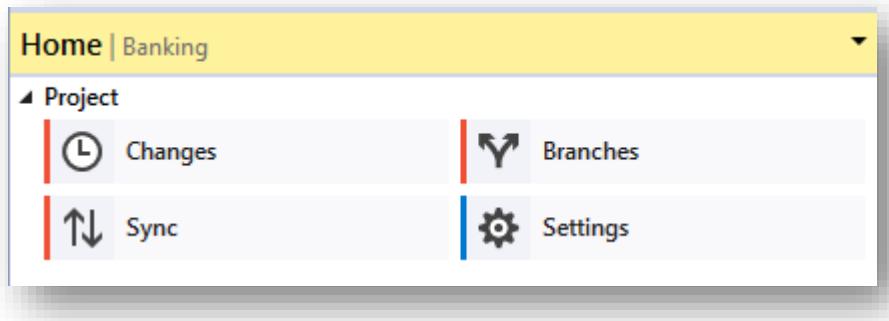


1. Klassen

- ▶ Aanmaken commit in de lokale repository
 - 7 REGELS VOOR EEN GOEDE COMMIT MESSAGE
 - 1. Hou onderwerp en body gescheiden met 1 witregel
 - 2. Beperk de lengte van het onderwerp tot 50 karakters
 - 3. Het onderwerp begint met een hoofdletter
 - 4. Gebruik geen punt op het einde van het onderwerp
 - 5. Gebruik de gebiedende wijs in het onderwerp
 - 6. Beperk de breedte van de body tot 72 karakters
 - 7. Geef in je body een uitleg voor wat en waarom, niet over hoe
 - Meer op: <http://chris.beams.io/posts/git-commit/>

1. Klassen

- ▶ Eventueel pushen commit naar Remote Repository
 - Klik op Home 



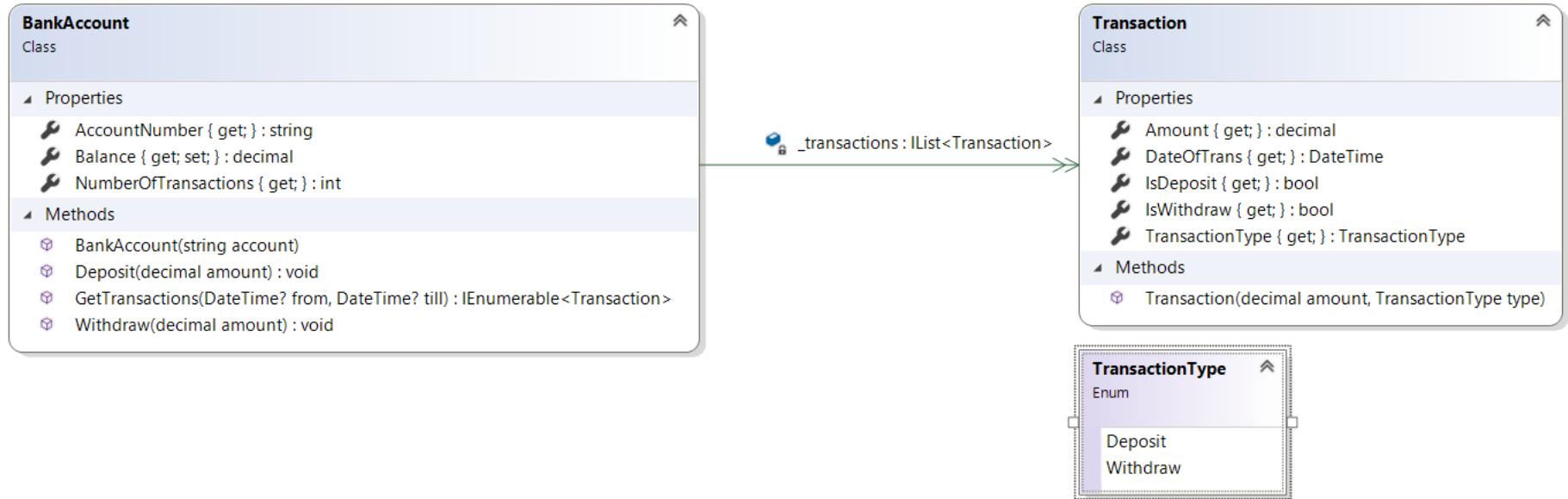
- Kies Sync. Hier kan je publiceren naar een Remote Repository



Associaties en Collections

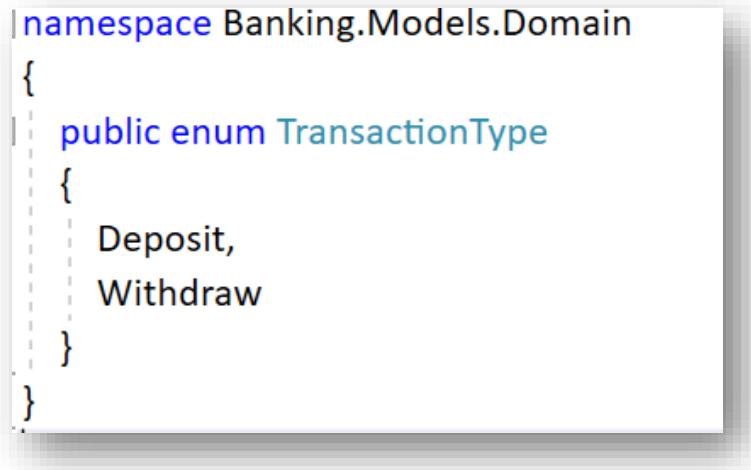
2. Associaties

- ▶ Klasse Transaction : immutable klasse
 - Bijhouden van verrichtingen



2. Associaties

- ▶ Maak de Enumeratie aan
 - Rechtsklik Domain folder > Add > new Item > Class. Naam TransactionType.cs



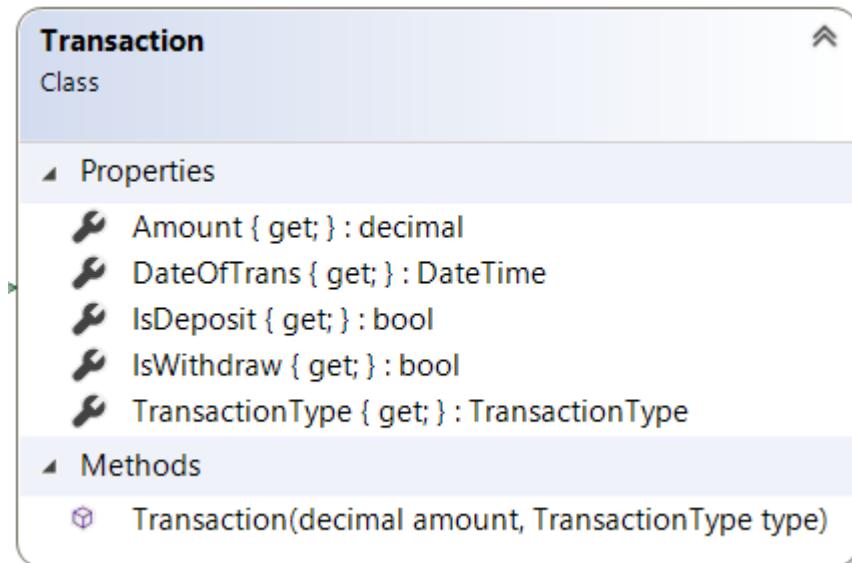
```
namespace Banking.Models.Domain
{
    public enum TransactionType
    {
        Deposit,
        Withdraw
    }
}
```

A screenshot of a code editor window showing a C# code snippet. The code defines an enum named 'TransactionType' within a namespace 'Banking.Models.Domain'. The enum contains two members: 'Deposit' and 'Withdraw'. The code is highlighted with syntax coloring, and the entire block is enclosed in a light gray rectangular box.

- Verwijder "unused" using statements

2. Associaties

- ▶ Maak de klasse Transaction aan
 - Maak zelf de klasse Transaction aan
 - Maak van Amount, DateOfTrans, TransactionType
 - Maak van IsDeposit en IsWithdraw read only properties
 - Implementeer de klasse Transaction



2. Associaties

▶ Implementatie

- Alle props zijn readonly => immutable

```
class Transaction
{
    #region Properties
    public DateTime DateOfTrans { get; }
    public TransactionType TransactionType { get; }
    public decimal Amount { get; }
    #endregion

    #region Constructors
    public Transaction(decimal amount, TransactionType type)
    {
        Amount = amount;
        TransactionType = type;
        DateOfTrans = DateTime.Now;
    }
    #endregion
}
```

2. Associaties

▶ Implementatie (vervolg)

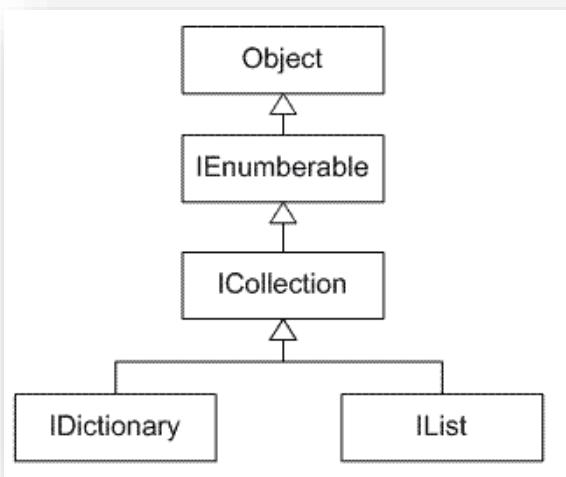
```
#region Methods
public bool IsWithdraw
{
    get { return TransactionType == TransactionType.Withdraw; }
}

public bool IsDeposit
{
    get { return TransactionType == TransactionType.Deposit; }
}
#endregion
```

Als je get selecteert, dan verschijnt het lampje. We kunnen hiervoor gebruik maken van expression bodies. Zie volgend hoofdstuk.

2. Associaties - Collections

- ▶ Om de transactions bij te houden maken we gebruik van **Collections**
- ▶ Namespace: System.Collections.Generic
- ▶ Een generische collection is **strongly typed** (type safe): dit betekent dat het maar 1 type van object kan bevatten
- ▶ Collections worden ook generics genoemd in .Net



- ▶ Meer op:

<http://msdn.microsoft.com/en-us/library/0sbxh9x2.aspx>

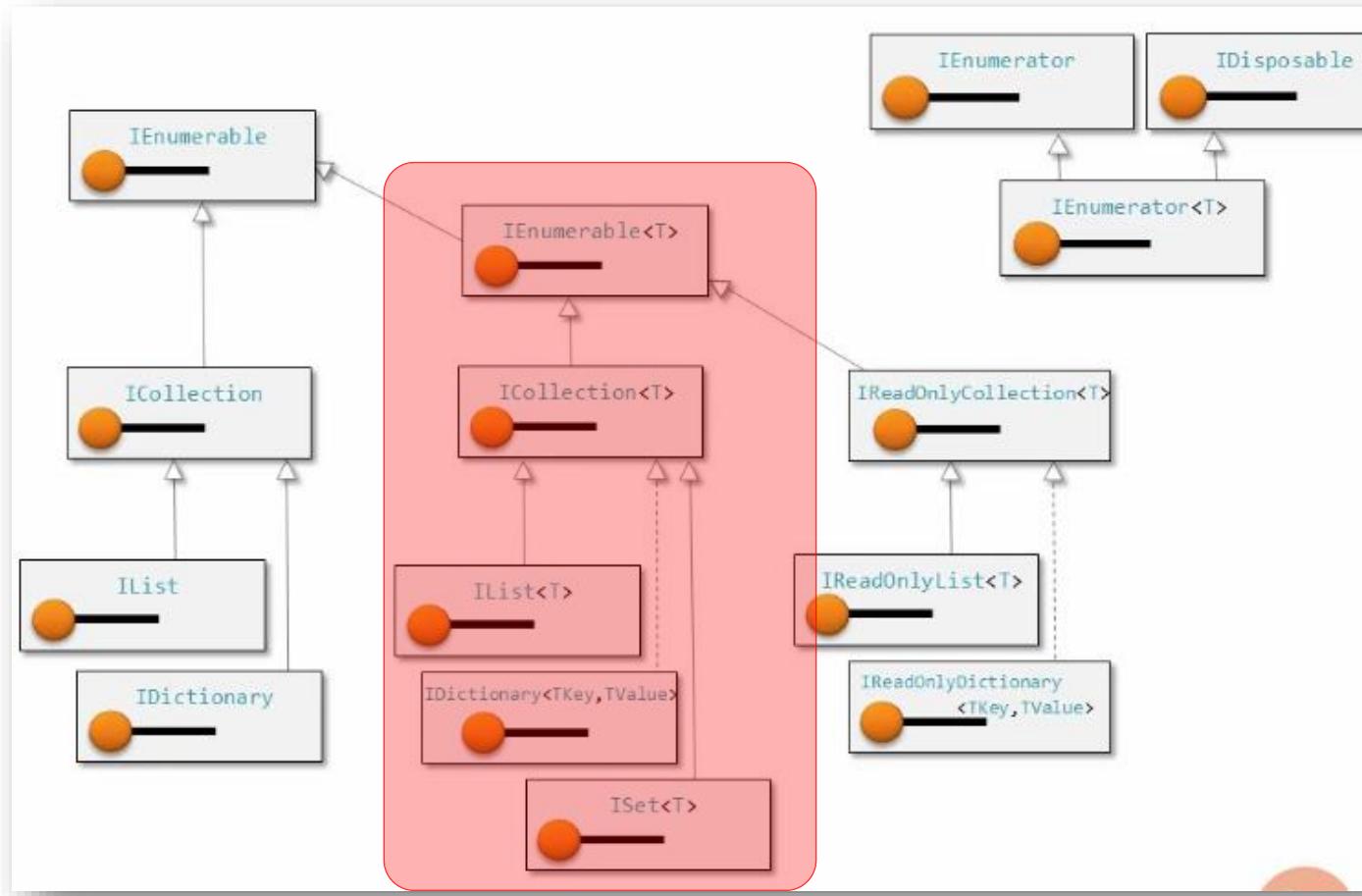
2. Associaties - Collections

▶ Collection Interfaces

- interfaces laten je toe om **loosely coupled, testable code** te schrijven
 - methodes geven liever collecties terug via een interface dan via een concreet type
- we gaan even kijken hoe collecties in C# georganiseerd zijn
 - als je de interfaces begrijpt ga je de collections zelf beter begrijpen en gebruiken

2. Associaties - Collections

▶ Collection Interfaces



dit zijn de core generic interfaces die we behandelen in deze cursus

2. Associaties - Collections



IEnumerable<T>:
"You can iterate my elements"

▶ IEnumerable<T>

- meest belangrijke interface, zegt dat we over de elementen kunnen **itereren**
- biedt een **enumerator** aan om door een collectie te lopen.
 - dit betekent dat je de collectie kunt doorlopen met een **foreach**

IEnumerable Interface

.NET Framework 4.6 and 4.5 | Other Versions ▾

Methods

	Name	Description
≡	GetEnumerator()	Returns an enumerator that iterates through a collection.

slechts 1 methode in deze interface!

2. Associaties - Collections

▶ IEnumerable<T>

```
IEnumerable<string> daysOfWeek = new List<string>
{
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
};

foreach (string day in daysOfWeek)
    Console.WriteLine(day);
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Merk op: List<string> is de concrete implementatie.
Merk op: We maken hier gebruik van een collection initializer.

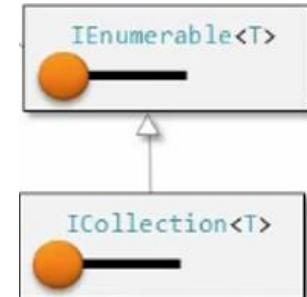
- Merk op: **T is een generic type parameter** die je bij definitie van een collectie moet opgeven.

2. Associaties - Collections



ICollection<T>:

"I know how many elements I have"
"You can modify my contents"



▶ ICollection<T>

- implementeert IEnumerable<T>
- extra properties en methodes laten toe om **de grootte** van de collectie op te vragen en de collectie te **manipuleren**

Properties

	Name	Description
	Count	Gets the number of elements contained in the ICollection<T>.
	IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.

2. Associaties - Collections

▶ ICollection<T>

Methods

	Name	Description
•	Add(T)	Adds an item to the ICollection<T>.
•	Clear()	Removes all items from the ICollection<T>.
•	Contains(T)	Determines whether the ICollection<T> contains a specific value.
•	CopyTo(T[], Int32)	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index.
•	GetEnumerator()	Returns an enumerator that iterates through the collection.(Inherited from IEnumerable<T> .)
•	Remove(T)	Removes the first occurrence of a specific object from the ICollection<T>.

2. Associaties - Collections

▶ ICollection<T>

- voorbeeld: Count, Add, Remove Contains...

```
ICollection<string> daysOfWeek = new List<string> {
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
};
Console.WriteLine("There are {0} days in daysOfWeek", daysOfWeek.Count);
daysOfWeek.Remove("Saturday");
daysOfWeek.Remove("Sunday");
daysOfWeek.Add("Weekend-day");
Console.WriteLine("After manipulating the collection it contains {0} days:", daysOfWeek.Count);
foreach (string day in daysOfWeek)
    Console.WriteLine(day);
Console.WriteLine("daysOfWeek contains Sunday: {0}", daysOfWeek.Contains("Sunday"));
```

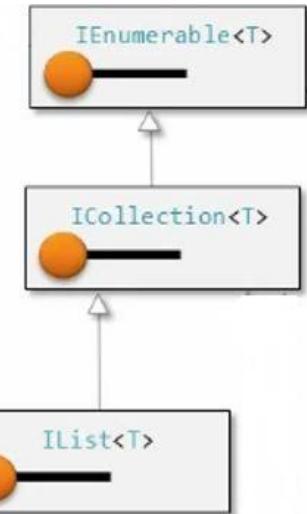
```
There are 7 days in daysOfWeek
After manipulating the collection it contains 6 days:
Monday
Tuesday
Wednesday
Thursday
Friday
Weekend-day
daysOfWeek contains Sunday: False
```

2. Associaties - Collections



IList<T>:

"You can look up my elements with an index"



▶ **IList<T>**

- implementeert **ICollection<T>**
- index gebaseerde toegang tot de elementen van de collectie
 - 0-based indexing
 - gebruik rechte haakjes: **[index]**
- enkele methodes:
 - een element aan de collectie toe te voegen op een specifieke plaats: **Insert**
 - een element uit de collectie weghalen van een specifieke plaats: **RemoveAt**
 - de plaats van een element in de collectie te bepalen: **IndexOf**

2. Associaties - Collections

▶ **IList<T>**

Methods

	Name	Description
≡	Add(Object)	Adds an item to the IList.
≡	Clear()	Removes all items from the IList.
≡	Contains(Object)	Determines whether the IList contains a specific value.
≡	CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array, starting at a particular Array index. (Inherited from ICollection.)
≡	GetEnumerator()	Returns an enumerator that iterates through a collection.(Inherited from IEnumerable.)
≡	IndexOf(Object)	Determines the index of a specific item in the IList.
≡	Insert(Int32, Object)	Inserts an item to the IList at the specified index.
≡	Remove(Object)	Removes the first occurrence of a specific object from the IList.
≡	RemoveAt(Int32)	Removes the IList item at the specified index.

2. Associaties - Collections

▶ **IList<T>**

- **voorbeeld:** gebruik index, Insert, RemoveAt, IndexOf, ...

```
IList<string> daysOfWeek = new List<string> {
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
};
Console.WriteLine("The first day is {0}", daysOfWeek[0]);
daysOfWeek[0] = "Difficult day";
daysOfWeek[4] = "Happy day";
daysOfWeek.RemoveAt(6);
daysOfWeek.RemoveAt(5);
daysOfWeek.Insert(5, "Weekend-day");
foreach (string day in daysOfWeek)
    Console.WriteLine(day);
Console.WriteLine("Tuesday is day {0}", daysOfWeek.IndexOf("Tuesday"));
```

```
The first day is Monday
Difficult day
Tuesday
Wednesday
Thursday
Happy day
Weekend-day
Tuesday is day 1
```

2. Associaties

▶ Klasse BankAccount

- Voorzie in de klasse BankAccount een generische lijst van transacties. Initialisatie kan bij declaratie of in de constructor

```
using System;
using System.Collections.Generic;

namespace Banking.Models.Domain
{
    class BankAccount
    {
        #region Fields
        private readonly IList<Transaction> _transactions = new List<Transaction>();
        #endregion
    }
}
```

```
public BankAccount(string account)
{
    AccountNumber = account;
    Balance = Decimal.Zero;
    _transactions = new List<Transaction>();
}
```

2. Associaties

▶ Klasse BankAccount

- Voeg extra properties en methodes toe

```
public int NumberOfTransactions  
{  
    get { return _transactions.Count; }  
}
```

NumberOfTransactions
is een **read-only**
property

```
public IEnumerable<Transaction> GetTransactions(DateTime? from, DateTime? till)  
{  
    if (from == null && till == null) return _transactions;  
    if (from is null) from = DateTime.MinValue;  
    if (!till.HasValue) till = DateTime.MaxValue;  
  
    IList<Transaction> transList = new List<Transaction>();  
    foreach (Transaction t in _transactions)  
    {  
        if (t.DateOfTrans >= from && t.DateOfTrans <= till)  
            transList.Add(t);  
    }  
    return transList;  
}
```

Nullable types: zie
hoofdstuk 2

2. Associaties

- De methodes Withdraw en Deposit

```
public void Withdraw(decimal amount)
{
    _transactions.Add(new Transaction(amount, TransactionType.Withdraw));
    Balance -= amount;
}

public void Deposit(decimal amount)
{
    _transactions.Add(new Transaction(amount, TransactionType.Deposit));
    Balance += amount;
}
```

2. Associaties

- Pas Program.cs aan (later zien we unit testen)

```
static void Main(string[] args)
{
    BankAccount account = new BankAccount("123-4567890-02");
    Console.WriteLine($"AccountNumber: {account.AccountNumber}");
    Console.WriteLine($"Balance: {account.Balance}");
    account.Deposit(200M, "My first deposit");
    Console.WriteLine($"Balance after deposit of 200 euros: {account.Balance}");
    account.Withdraw(100);
    Console.WriteLine($"Balance after withdraw of 100 euros: {account.Balance}");
    Console.WriteLine($"Number of transactions: {account.NumberOfTransactions}");
    IEnumerable<Transaction> transactions = account.GetTransactions(null, null);
    foreach (Transaction t in transactions)
        Console.WriteLine($"Transaction: {t.DateOfTrans} - {t.Amount} - {t.TransactionType} - {t.Notes ?? "/"});
}
```

2. Associaties

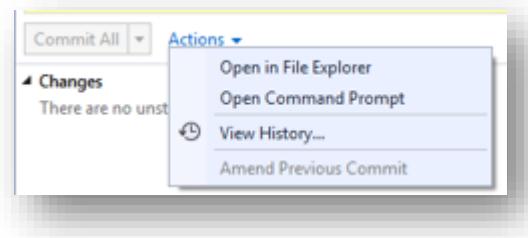


commit “Add class Transaction (Associaties)”

▶ Commit

- Voer code review uit
- Commit boodschap: Add class Transaction

- Klik eens op link Actions > View History. Zo kan je de detail van elke commit bekijken.



Graph	ID	Author	Date	Message
▲ Local History				
	5223499c	Karine Samyn	10-Sep-19 20:41:42	Add class Transaction
	65f5609a	Karine Samyn	10-Sep-19 18:13:00	Add class BankAccount
	5c5d7e4e	Karine Samyn	10-Sep-19 15:04:42	Add project files.
	3b001350	Karine Samyn	10-Sep-19 15:04:39	Add .gitignore and .gitattributes.

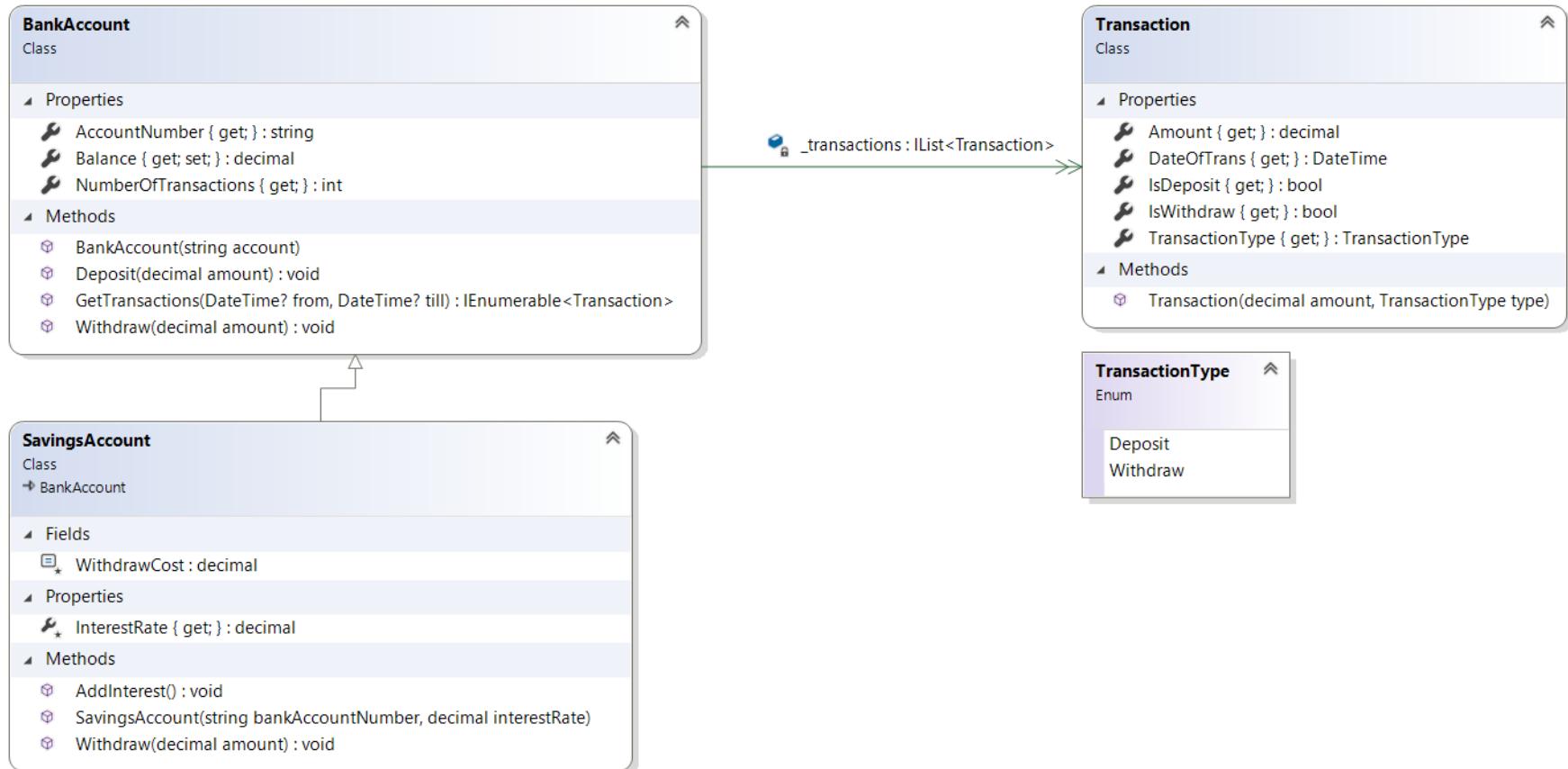
Overerving

3. Overerving

- ▶ Overerving is een mechanisme waarbij software opnieuw wordt gebruikt: nieuwe klassen worden gecreëerd vertrekkende van bestaande klassen
 - De superklasse bevat de gemeenschappelijke attributen, operaties en associaties
 - De subklasse erft alles van de superklasse: attributen, operaties, associaties
 - In een subklasse wordt het gedrag van de superklasse uitgebreid en/of gespecialiseerd
 - Het gedrag van de subklasse kan verder gespecialiseerd worden door methoden van de superklasse te overriden (herdefiniëren) in de subklasse.
 - De subklasse heeft een ‘is een’ relatie met de superklasse

3. Overerving

- Maak klasse SavingsAccount aan.



3. Overerving

▶ Definitie superklasse

```
public class BankAccount
```

- Opmerking: indien van een klasse niet mag worden afgeleid kan je de klasse verzegelen

```
public sealed class BankAccount
```

Het equivalent in Java is een final class

3. Overerving

▶ Definitie subklasse

de: operator wordt in Java **extends**

```
public class SavingsAccount: BankAccount
```

- Bij een subklasse hoort steeds één superklasse. Net zoals Java laat .Net niet meer multiple inheritance toe. Een klasse kan wel meerdere interfaces implementeren
- Access
 - De private members (attributen/methoden) van de superklasse zijn niet toegankelijk vanuit de subklasse.
 - De protected members uit de superklasse, zijn enkel toegankelijk in de subklassen, niet voor de buitenwereld.

In Java laat je met protected access ook toegang vanuit andere klassen binnen dezelfde package toe...

3. Overerving

▶ Constructors

- Constructoren van de superklasse worden niet overgeërfd door de subklassen.
- Als de subklasse geen constructor bevat, maakt de compiler zelf een default constructor aan, die automatisch de constructor van de superklasse oproept.
 - Als de superklasse dan geen default constructor bevat krijg je een compiler fout
- Keyword “**base**”: aanroepen methode/constructor uit superklasse

base komt overeen met Java **super**

- Keyword “**this**”: refereren naar de huidige instantie

3. Overerving

The decimal suffix is M/m since D/d was already taken by double . Although it has been suggested that M stands for money, Peter Golde recalls that M was chosen simply as the next best letter in decimal .

o Voorbeeld

```
class SavingsAccount : BankAccount
{
    #region Fields
    protected const decimal WithdrawCost = 0.25M;
    #endregion

    #region Properties
    public decimal InterestRate { get; }
    #endregion

    #region Constructors
    public SavingsAccount(string bankAccountNumber, decimal interestRate)
        : base(bankAccountNumber)
    {
        InterestRate = interestRate;
    }
    #endregion
```

SavingsAccount erft van de superklasse BankAccount

Enkel SavingsAccount en klassen die erven van SavingAccount hebben toegang tot WithdrawCost

base: aanroepen constructor uit superklasse voor initialisatie members van superklasse

Voorbeeld oproepen constructor van eigen klasse:

```
public SavingsAccount(string bankAccountNumber, decimal interestRate, bool goldMember): this(bankAccountNumber, interestRate)
```

3. Overerving

► Methodes: nieuwe methode toevoegen

```
public class SavingsAccount: BankAccount {  
    protected const decimal WithdrawCost = 0.25M; ← Extra constant field  
  
    public decimal InterestRate { get; } ← Extra property  
  
    public SavingsAccount(string bankAccountNumber, decimal interestRate)  
        : base(bankAccountNumber) {  
            InterestRate = interestRate;  
    }  
  
    public void AddInterest() {  
        Deposit(Balance * InterestRate ); ← Extra methode  
    }  
}
```

3. Overerving

▶ Methodes - overriding

- **Standaard kan je een methode niet overschrijven in een subklasse**
- voorbeeld: de methode Withdraw in BankAccount kan je niet overschrijven in de subklasse SavingsAccount

```
public void Withdraw(decimal amount)
{
    _transactions.Add(new Transaction(amount, TransactionType.Withdraw));
    Balance -= amount;
}
```

in Java zou dit wel kunnen...

3. Overerving

▶ Methodes - overriding

- Indien je een methode wil override-n moet je in de superklasse gebruik maken van het keyword **virtual**
 - Via dynamic binding wordt dan de juiste versie van de methode uitgevoerd

```
public virtual void Withdraw(decimal amount)
{
    _transactions.Add(new Transaction(amount, TransactionType.Withdraw));
    Balance -= amount;
}
```

Maakt het mogelijk de methode Withdraw te override-n in een subklasse van BankAccount

3. Overerving

▶ Methodes - overriding

```
public class BankAccount {  
    public virtual void Withdraw(decimal amount) {  
        ...  
        Balance -= amount;  
    }  
}
```

```
public class SavingsAccount: BankAccount {  
    public override void Withdraw(decimal amount) {  
        base.Withdraw(amount);  
        base.Withdraw(WithdrawCost);  
    }  
}
```

Deze Withdraw override de Withdraw uit bankAccount.

voor Java programmeurs even wennen:
virtual en **override** moeten expliciet
aangegeven worden in de code!

3. Overerving

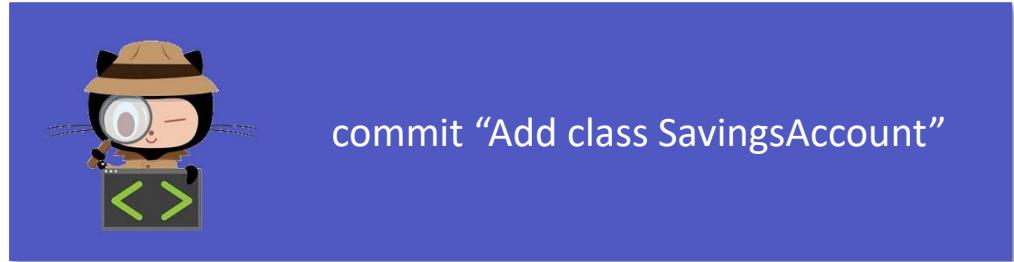
- ▶ voorbeeld: Instantie aanmaken van de subklasse
 - Pas program.cs aan

```
SavingsAccount saving = new SavingsAccount("123-4567891-03", 0.01M);
saving.Deposit(200M);
saving.Withdraw(100M);
saving.AddInterest();
Console.WriteLine($"Balance savingsaccount: {saving.Balance}");
```

Methode uit de subklasse zal worden aangeroepen

3. Overerving

▶ Time to commit



commit "Add class SavingsAccount"

- Inspecteer de code
- Commit

3. Overerving

▶ Klasse Object

- Elke klasse is afgeleid van System.Object
- Deze bevat 3 overridable methodes
 - **ToString()**, die geeft als standaardgedrag de naam van de klasse weer.
 - **Equals()**, standaardgedrag: 2 reference variabelen zijn gelijk als ze wijzen naar hetzelfde object, 2 value type variabelen zijn gelijk als ze dezelfde waarde bevatten
 - **GetHashCode()**, gebruikt in hash-based collections: Dictionary< TKey, TValue > , Hashtable of type afgeleid van DictionaryBase.
- Bevat statische methodes
 - **ReferenceEqual(objA, objB)**: test of 2 variabelen wijzen naar hetzelfde object, of beide null zijn.
 - Object.ReferenceEqual(o1,o2)
 - **Equals(objA, objB)**: checkt op ReferenceEqual, indien niet gelijk retourneert het het resultaat objA.Equals(objB)
 - Object.Equals(o1,o2)

3. Overerving

- ▶ Klasse Object
 - Pas klasse **BankAccount** aan

```
public override string ToString()
{
    return ${AccountNumber} - {Balance};
}

public override bool Equals(object obj)
{
    //BankAccount account = obj as BankAccount;
    //if (account == null) return false;

    // using the is operator with pattern matching:
    if (!(obj is BankAccount account)) return false;
    return AccountNumber == account.AccountNumber;
}

public override int GetHashCode()
{
    return AccountNumber?.GetHashCode() ?? 0;
}
```

3. Overerving

- ▶ Klasse Object
 - Pas Program.cs aan

```
BankAccount savingsAccount = new SavingsAccount("123-4567890-02", 0.05M);
Console.WriteLine($"SavingsAccount : {savingsAccount.ToString()}");
savingsAccount.Deposit(200M);
savingsAccount.Withdraw(100M);
Console.WriteLine($"Balance savingsaccount: {savingsAccount.Balance} ");
Console.ReadKey();
```

- ToString() mag je achterwege laten

```
Console.WriteLine($"SavingsAccount : {savingsAccount}");
```

3. Overerving

▶ Time to commit



commit “Implement overridable methods from Object”

- Inspecteer de code
- Commit

Polymorfisme

4. Polymorfisme

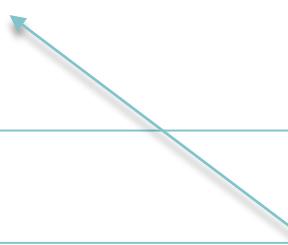
- ▶ Polymorfisme kan optreden als men overerving gebruikt. Zo kan men objecten van een superklasse en 1 of meerder subklassen van die klasse opslaan in een collection die bestaat uit objecten van de superklasse.
- ▶ Er kan ook op een polymorfe manier een object methode aangeroepen worden. Hier wordt dan aan de hand van het type overervende klasse gekozen welke methode er moet worden uitgevoerd. Voorwaarde: de methode moet gedefinieerd zijn in de superklasse.
- ▶ Het type van een object bepalen kan via de **is** operator

```
BankAccount s = new SavingsAccount("13-455665-13", 0.10M);  
if (s is SavingsAccount) {....}
```

4. Polymorfisme

- Voorbeeld

```
BankAccount[] accounts = new BankAccount[3];  
accounts[0] = new BankAccount("13-455665-13");  
accounts[1] = new SavingsAccount("13-455665-13", 0.05M);  
accounts[2] = new SavingsAccount("13-455665-14", 0.03M);  
  
foreach (BankAccount a in accounts)  
{  
    a.Withdraw(10M);  
}
```



afhankelijk van het type wordt de Withdraw methode uit de superklasse BankAccount, of de Withdraw methode uit de subklasse SavingAccount aangeroepen

Abstracte klassen

5. Abstracte klasse

- ▶ Een klasse met 1 of meerdere abstracte methodes (methodes zonder een implementatie) is een abstracte klasse.
 - Een abstracte klasse kan zowel abstracte als normale members bevatten.
- ▶ Van een abstracte klasse kunnen geen instanties worden aangemaakt. Je moet klassen hebben die overerven van deze klasse om ze te kunnen gebruiken.
- ▶ De declaratie van een abstracte klasse bevat het keyword **abstract**.
- ▶ Elke afgeleide klasse van een abstracte klasse moet alle abstracte members van de abstracte klasse implementeren door gebruik te maken van de **override** keyword, tenzij de afgeleide klasse zelf abstract is.

5. Abstracte klasse

▶ Voorbeeld

```
public abstract class BankAccount {  
    public virtual void Withdraw(decimal amount) {...}  
    public abstract string PrintAccount();  
}
```

BankAccount is nu een abstracte klasse en kan niet geïnstantieerd worden

elke concrete subklasse van BankAccount zal PrintAccount override-n

5. Abstracte klasse

▶ Voorbeeld

- Maak de klasse BankAccount abstract

```
public abstract class BankAccount { ... }
```

- Voeg een abstracte methode toe in BankAccount

```
public abstract string Print();
```

- Implementeer de methode in SavingsAccount

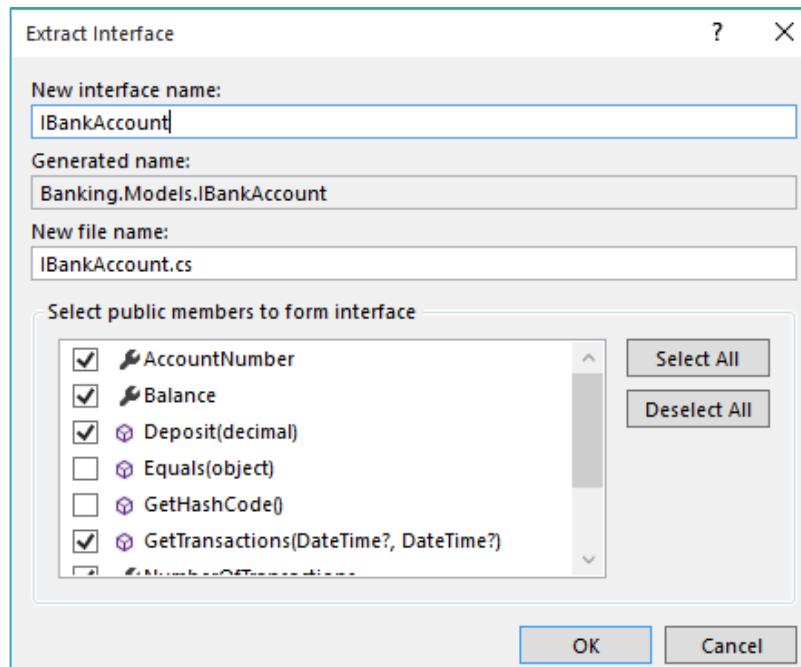
```
public override string Print() {  
    return $"Savingsaccount balance = {Balance}";  
}
```

- Merk op: de code compileert nu niet meer. Waarom?
- Verwijder de toegevoegde code terug

Interfaces

6. Interface

- ▶ Bij wijze van voorbeeld maken we een IBankAccount interface aan.
 - Open de klasse BankAccount in het code venster. Rechtsklik > Quick Actions > Extract interface
 - kan ook via resharper: rechtsklik > Refactor > Extract interface



6. Interface

- ▶ Bij wijze van voorbeeld maken we een IAccount interface aan.
 - The code in the interface

```
interface IBankAccount
{
    string AccountNumber { get; }
    decimal Balance { get; }
    int NumberOfTransactions { get; }

    void Deposit(decimal amount);
    IEnumerable<Transaction> GetTransactions(DateTime? from, DateTime? till);
    void Withdraw(decimal amount);
}
```

merk op: de private setters zouden geen deel uitmaken van de geextraheerde interface, alles in een interface is publiek toegankelijk

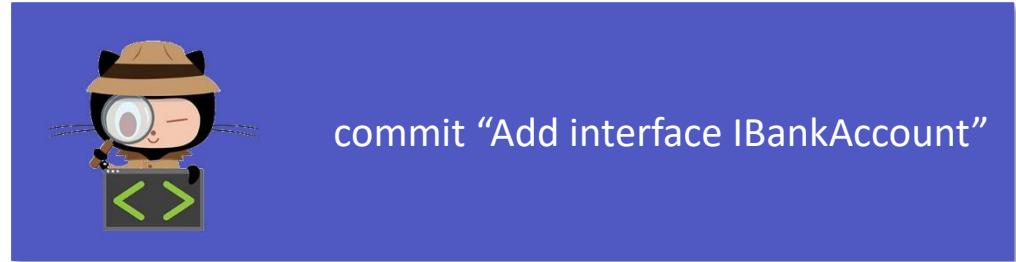
6. Interface

▶ In C# 8.0

- Je kan nu ook members toevoegen aan interfaces met een default implementatie. Zo kunnen API-ontwikkelaars methoden toevoegen aan een interface in latere versies zonder de bron- of binaire compatibiliteit met bestaande implementaties van die interface te verbreken. Bestaande implementaties nemen de standaardimplementatie over.
- Voorbeeld op <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/default-interface-members-versions>

6. Interface

▶ Time to commit



commit “Add interface IBankAccount”

- Inspecteer de code
- Commit

Static members

7. Statische members

- ▶ Informatie eigen aan de klasse, maar niet aan een bepaalde instantie van die klasse

```
double result;  
result = Math.Cos(45);
```

- ▶ Gebruik keyword static.

- Voorbeeld static field nrOfAccounts

```
public class SavingsAccount: BankAccount {  
    public static int nrOfAccounts;
```

C# gebruikt de: operator voor zowel overerving als voor het implementeren van een interface, in Java gebruik je hier **implements**

```
int total = SavingsAccount.nrOfAccounts;
```

Merk op: klassenaam ipv instance naam!

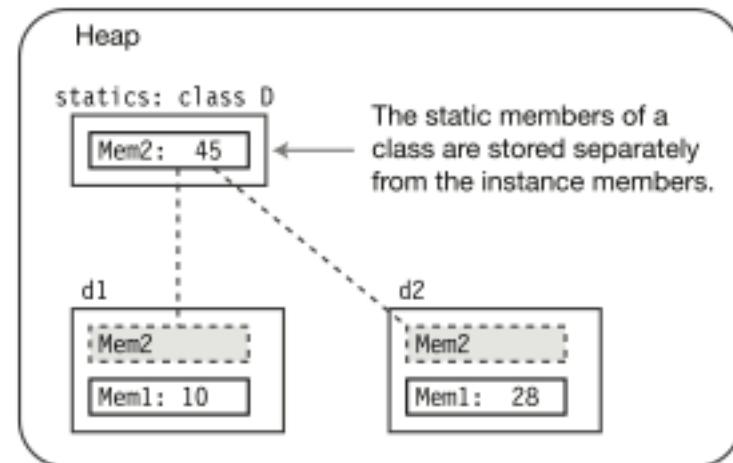
- ▶ Static members zijn altijd toegankelijk, ook al zijn er geen instanties van de klasse aangemaakt.

7. Statische members

- ▶ Static fields worden apart in het geheugen bijgehouden en worden gedeeld door alle instanties van die klasse.

```
class D
{
    int Mem1;
    static int Mem2;
    ...
}

static void Main()
{
    D d1 = new D();
    D d2 = new D();
    ...
}
```



Static field Mem2 is shared by all the instances of class D, whereas each instance has its own copy of instance field Mem1.

7. Statische members

- ▶ Een klasse kan ook static gemaakt worden
 - **voorbeeld** de static class Math

```
double result;  
result = Math.Cos(45);
```

- ▶ Statische klassen
 - hebben enkel static members
 - kunnen niet geïnstantieerd worden
 - zijn sealed (geen overerving mogelijk)

Github

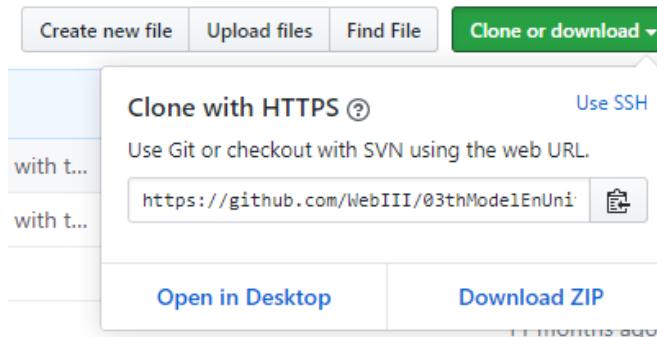
HoGent

Github

- ▶ De code staat op github

<https://github.com/WebIII/03thModelEnUnitTesten>

- Je kan een clone aanmaken vanuit Visual Studio
- Ga naar Team Explorer, klik op het stekker icoon  (Manage connections)
- Klap “Local Git Repositories” open en klik op Clone
- Kopieer de URL vanuit github (Klik daar op Clone or download, en dan op het icoon Copy to clipboard))

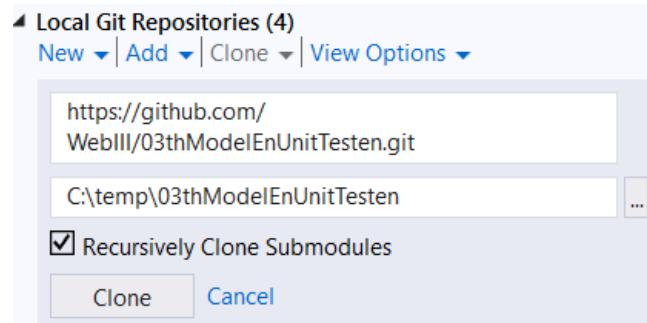


- Meer op

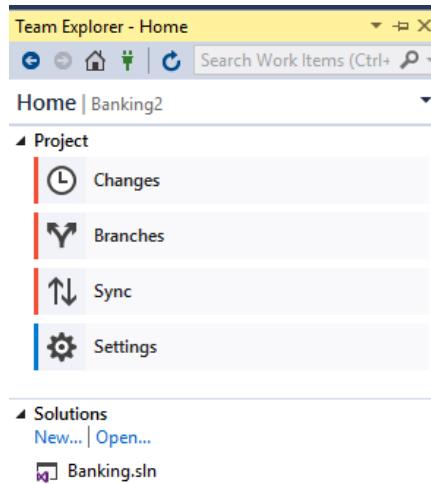
<https://blogs.msdn.microsoft.com/visualstudioalm/2013/02/06/create-connect-and-publish-using-visual-studio-with-git/>

Github

- Paste de URL in VS, geef ook de target folder op

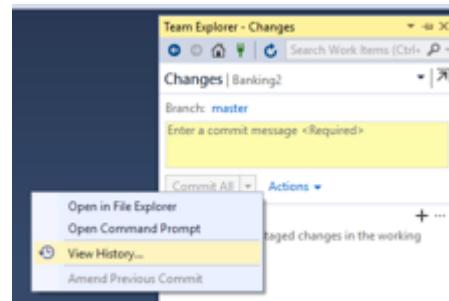


- De repository wordt toegevoegd. Dubbelklikken op de repository toont de solution. Dubbelklik de solution om deze te openen



Github

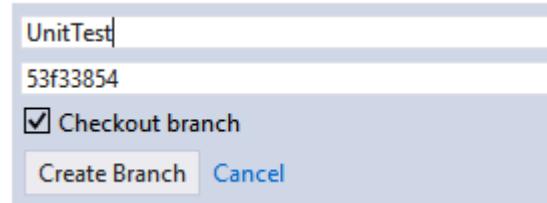
- ▶ Wens je de code van een bepaalde commit te bekijken
 - In Team Explorer > Klik op Changes > Klik op Actions en selecteer **View History**



History - master					C:\Users\ksa607\Ap...ccount.b5b6f7a.cs
Graph	ID	Author	Date	Message	Filter History
Local History					
	7c25818e	Stefaan Samyn	02-Oct-18 12:09:03	Add unit tests for Transaction, SavingsAccount and BankAccount with transactions	master
	f068dbc3	Stefaan Samyn	02-Oct-18 11:50:23	Complete unit tests for BankAccount	
	930510c8	Stefaan Samyn	02-Oct-18 11:37:52	Add some basic unit tests for BankAccount	
	bf046bde	Stefaan Samyn	02-Oct-18 11:28:13	Add unit test project Banking.Tests to solution	
	1b29d4a8	Stefaan Samyn	02-Oct-18 11:24:09	Add interface IBankAccount	
	cf7acb37	Stefaan Samyn	02-Oct-18 11:16:21	Implement overridable methods from Object	
	0a601936	Stefaan Samyn	02-Oct-18 11:01:40	Add class SavingsAccount	
	9607ce0c	Stefaan Samyn	02-Oct-18 10:57:14	Add class Transaction	
	7c437997	Stefaan Samyn	02-Oct-18 10:50:43	Add class BankAccount	
	9198d8a9	Stefaan Samyn	02-Oct-18 10:42:06	Add project files.	
	ac583e3f	Stefaan Samyn	02-Oct-18 10:42:02	Add .gitignore and .gitattributes.	

Github

- ▶ Wens je een bepaalde commit te bekijken
 - Dubbelklik op een commit, toont de changes.
 - Rechtsklik op Commit > New Branch laat toe om een nieuwe branch aan te maken. Zo bekom je de code na deze commit en kan je hierin zelf verder werken.
 - Wens je vanaf commit 1 terug zelf de code in te geven (de stappen op de volgende slides te volgen): rechtsklik deze commit > new branch. Geef naam in (mag geen spaties bevatten) en vink checkout branch aan (zo schakel je onmiddellijk over naar deze branch)



- Nu werk je in deze branch (de code die in de solution zichtbaar is)
- Switchen van branch (bvb terug naar de master): Team Explorer > Branches > dubbelklik op de branch. Je kan alleen switchen van branches als alle wijzigingen gecommit zijn. Je kan ze ook stashen (aan de kant zetten voor later gebruik)
- Meer over branches: <https://msdn.microsoft.com/en-us/library/jj190809.aspx>
- Meer over git : <https://www.git-tower.com/learn/git/ebook/en/command-line/basics/basic-workflow#start>

Unit Testen

***“Extraordinary products
are merely side effects of
good habits.”***



A close-up photograph of a woman with long brown hair tied back, wearing black-rimmed glasses and a small hoop earring. She is leaning over a dark-colored laptop keyboard, her eyes focused intently on the keys. Her hands are resting on the keyboard, and she appears to be inspecting it or performing a detailed examination.

TESTEN

10. Unit testen

- ▶ Test Driven Development
- ▶ Aanmaken van test bibliotheek
- ▶ Aanmaken van unit test
 - Stappenplan
 - De 3 AAA's
 - Klasse Assert
 - Testen op exceptions
- ▶ Aanmaken van unit testen voor domein Banking
- ▶ Aanvullingen
- ▶ Tips
- ▶ Test List

10. Unit testen

Having a suite of automated tests is one of the best ways to ensure a software application does what its authors intended it to do. There are many different kinds of tests for software applications, including integration tests, web tests, load tests, and many others. At the lowest level are unit tests, which test individual software components or methods. Unit tests should only test code within the developer's control, and should not test infrastructure concerns, like databases, file systems, or network resources. Unit tests may be written using Test Driven Development (TDD), or they can be added to existing code to confirm its correctness. In either case, they should be small, well-named, and fast, since ideally you will want to be able to run hundreds of them before pushing your changes into the project's shared code repository.

<https://docs.asp.net/en/latest/testing/unit-testing.html>

10. Unit testen

► Wanneer unit testen schrijven?

Test-First

- Know how the code should look like



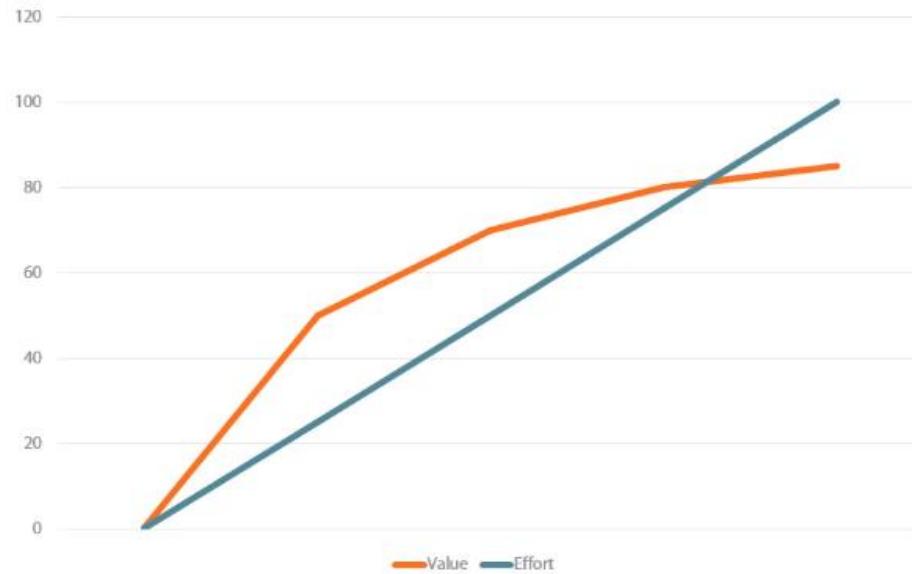
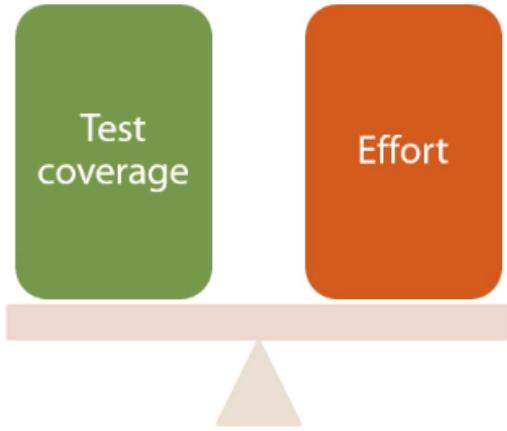
Code-First

- Experimenting with code

<https://app.pluralsight.com/player?course=domain-driven-design-in-practice&author=vladimir-khorikov&name=domain-driven-design-in-practice-m1&clip=8&mode=live>

10. Unit testen

- ▶ Test coverage versus Value distribution



<https://app.pluralsight.com/player?course=domain-driven-design-in-practice&author=vladimir-khorikov&name=domain-driven-design-in-practice-m1&clip=8&mode=live>

10. Unit testen

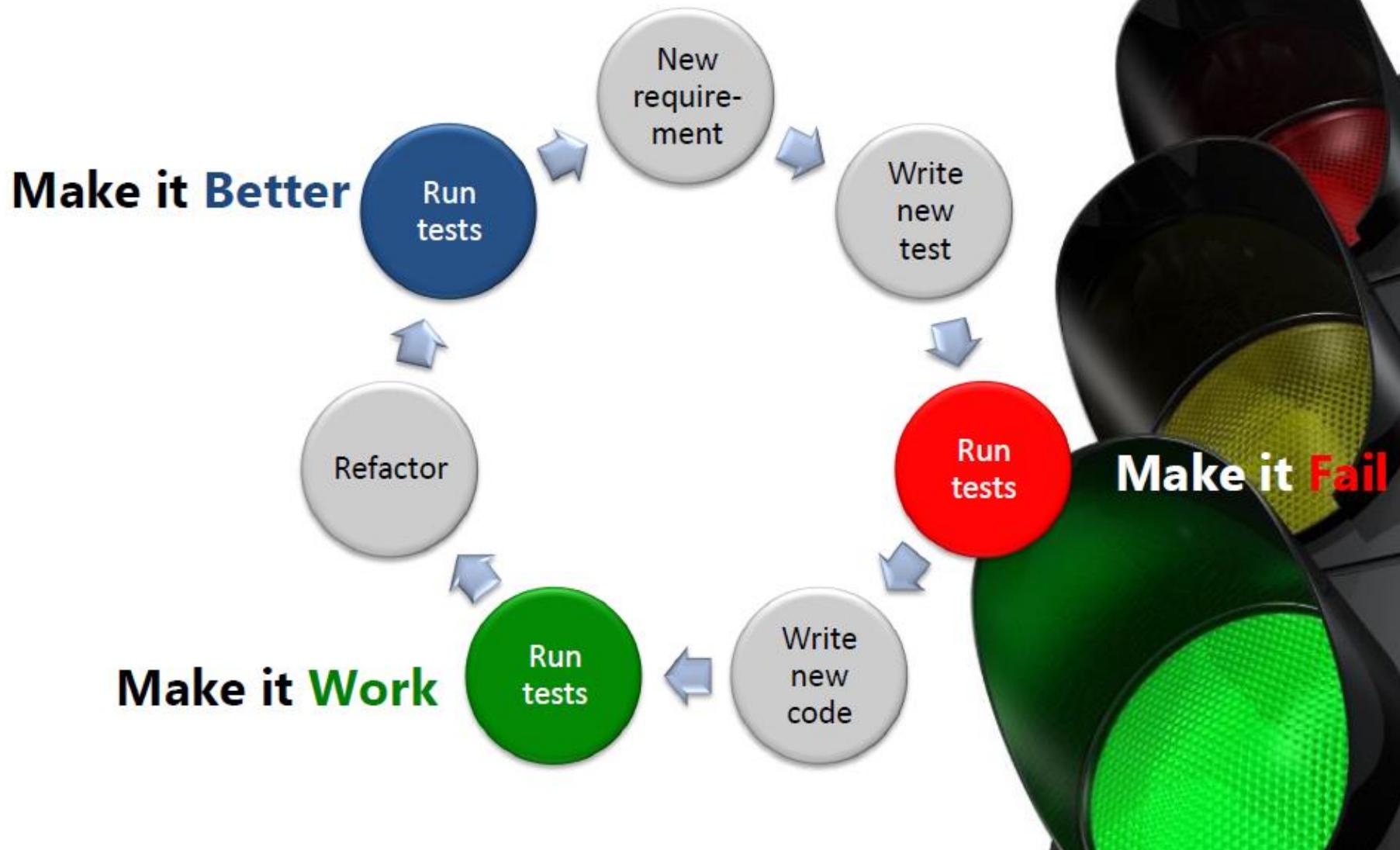
▶ TDD – Motto: **Rood, Groen, Refactor**

- Doe het **Falen**
 - Geen code zonder falende test
- Doe het **Werken**
 - Zo eenvoudig mogelijk
- Maak het **Beter**
 - Refactor

Merk op: in deze cursus houden we ons niet strict aan TDD



TDD cycle



10. Unit testen

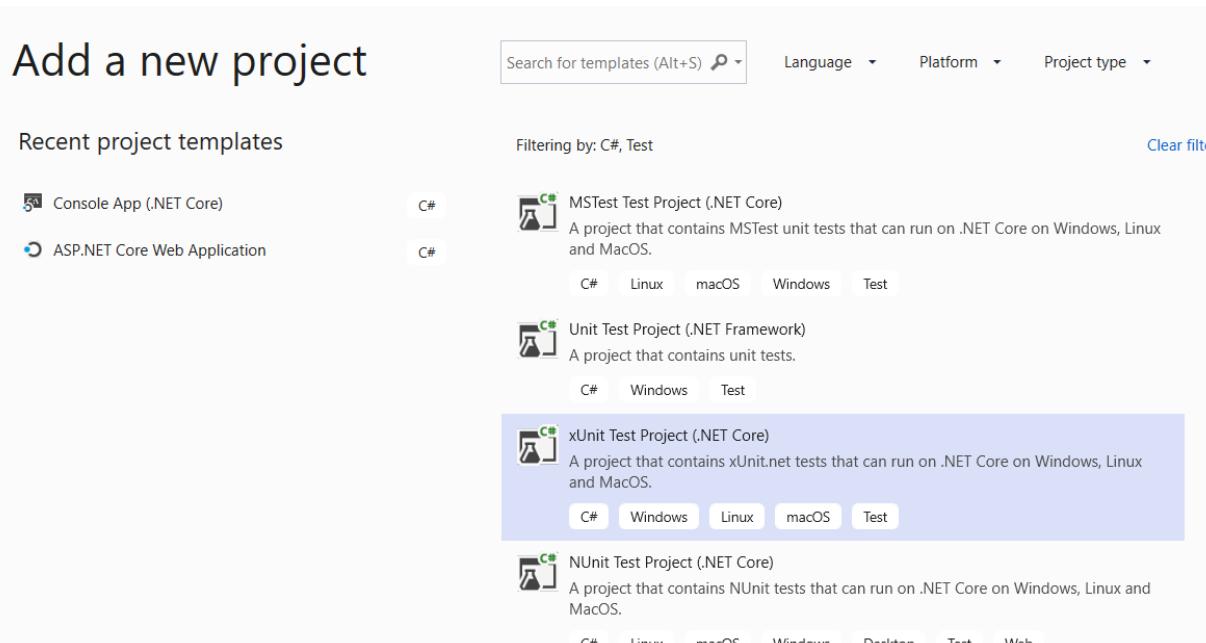
▶ Aanmaken van unit test project

- Een unit test project is een .Net Core **class library** dat refereert
 - naar de **SUT** (System under test, het project dat je test)
 - en een **test runner**. We gebruiken **xUnit**: <http://xunit.github.io/>

10. Unit testen

▶ Aanmaken van unit test project

- Selecteer de solution in Solution explorer.
- Rechtermuisknop > Add > New Project
- Kies als taal C#, en project type Tests
- Selecteer xUnit Test Project (.Net core)
- Geef naam van het project in: Banking.Tests

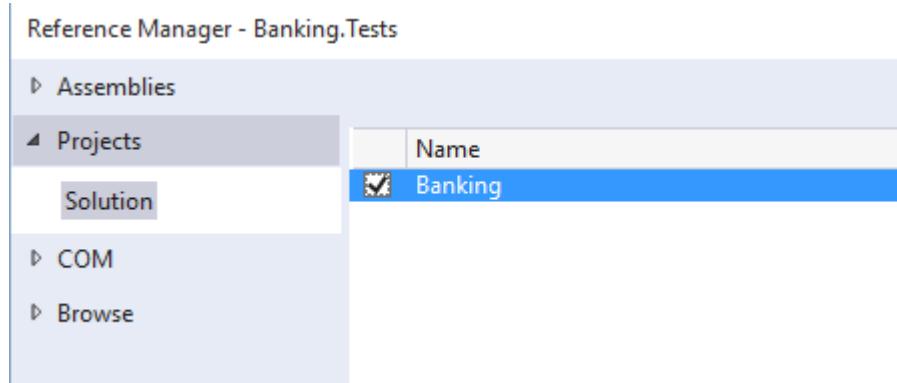


10. Unit testen

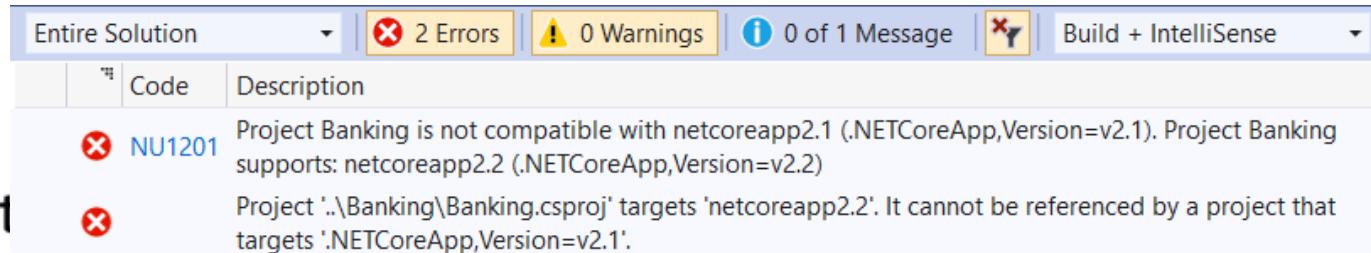
► Aanmaken van unit test project

Voeg een referentie toe naar de SUT, het project “Banking”

- Rechtsklik References in Banking.Tests > Add Reference > onder Projects, selecteer Solution > vink Banking aan
- Dit voegt in project.json Banking toe als dependency



- Dit genereert een fout



10. Unit testen

▶ Aanmaken van unit test project

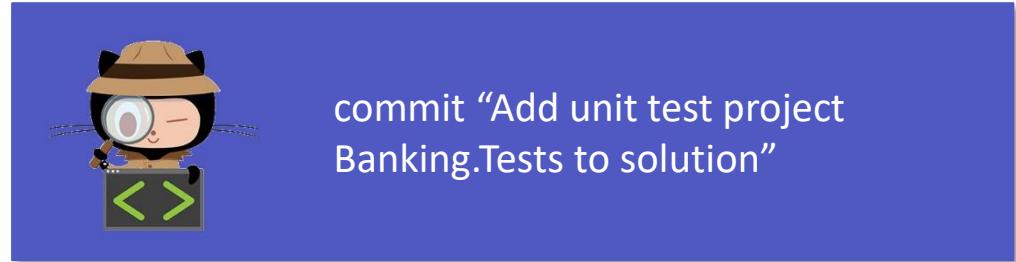
- Open de csproj file. Rechtsklik Banking.Tests project > Edit project file. Pas het target framework aan

```
<PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
```

- Verwijder unittest1.cs

10. Unit testen

► Time to commit



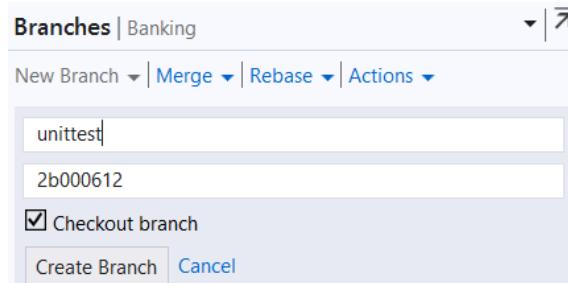
commit “Add unit test project
Banking.Tests to solution”

- Inspecteer de code
- Commit

10. Unit testen

► TIP

- Je zal later bestaande testklassen toevoegen aan het project. Om zeker te zijn dat je geen compilatie fouten krijgt kan je best de repo op github clonen
<https://github.com/WebIII/03thModelEnUnitTesten.git>
- Maak dan een branch aan bij de commit “Add excluded unit tests for later use”
 - Ga naar View History in Team Explorer
 - Rechtsklik deze commit > New Branch



- Nu codeer je verder in deze branch

10. Unit testen

- ▶ Aanmaken van unit testen
 - Stappenplan strikt TDD
 1. Maak een ontwerp van de klasse
=> Methodes throwen NotImplementedException
 2. Maak een testklasse
 3. Schrijf de testen in de testklasse
 4. Run de testen. Testen falen
 5. Pas de code in de klasse aan
 6. Run de testen opnieuw. Testen slagen
 7. Refactor indien nodig
 8. Run testen. Moeten nog steeds slagen
 9. Herhaal 3-9 tot alle gedrag is aangemaakt

10. Unit testen

- ▶ Stap 2: Aanmaken van een testklasse voor BankAccount
 - Maak de folders Models/Domain aan (neem de structuur van gerefereerd project over)
 - Rechtsklik op de folder > Add > New Item > Class en noem deze BankAccountTest
 - Naam testklasse = naam klasse + “Test”
 - Maak gebruik van namespace Xunit
 - Deze klasse bevat testmethodes. Elke testmethode heeft attribuut **[Fact]** of **[Theory]**
 - Facts zijn testen met steeds dezelfde data.
 - Theories zijn data driven unit tests. Dezelfde test definitie voor meerdere test data reeksen.

```
namespace Banking.Tests.Models.Domain
{
    class BankAccountTest { }
}
```

10. Unit testen



The 3A Pattern

- Arrange
- Act
- Assert



10. Unit testen

- ▶ Stap 3: Aanmaken van unit testen
 - 1 unit test is een methode die test of 1 bepaalde methode doet wat ze moet doen gegeven 1 bepaald concreet geval.
 - Zorg voor duidelijke naamgeving.
 - De naam moet aangeven wat getest wordt
 - Conventies:
 - **NaamTeTestenMethode_BeschrijvingGeval_TeVerwachtenResultaat**
 - OF
 - **NaamTeTestenMethodeGivenBeschrijvingGevalShouldTeVerwachtenResultaat**

10. Unit testen

- ▶ Stap 3: Aanmaken van unit testen
 - **AAA:** de normale flow in een unit test
 - **Arrange:** Initialisatie: Maak een object van de te testen klasse, initialiseer variabelen,...
 - **Act:** Roep de te testen methode op
 - **Assert:** Controleer of de methode correct is uitgevoerd.
 - 2 strekkingen
 - 1 assert/test
 - Meerdere asserts/test op voorwaarde dat je 1 type gedrag test

10. Unit testen

- ▶ Stap 3: Aanmaken van unit testen
 - Welke testen aanmaken: wees creatief!!!
 - Baseer je op use case, maar denk verder. Bedenk alternatieven.
 - Test zeker één normaal geval, maar vooral alle mogelijk foute gevallen en grensgevallen (bvb i.g.v. parameters: alle mogelijke inputwaarden voor parameter)
 - Geef betekenisvolle namen aan de test methodes
 - Zie cursus Ontwerpen I en II
 - Schrijf enkel testen voor methodes/properties met gedrag. Automatic props dien je niet te testen.

10. Unit testen

▶ Stap 3: Aanmaken van unit testen

- Aanmaken van unit test voor constructor BankAccount
 - Welk gedrag willen we testen?
 - Balance is 0 voor een nieuwe bankaccount
 - AccountNumber bevat de opgegeven waarde
 - Eerste test: balance=0 voor nieuwe bankaccount
 - Naam methode: NewAccount_BalanceZero
 - Unit test retourneert steeds void
 - Bevat het attribuut [Fact] -> namespace Xunit!
 - Test classes moeten ook public zijn !!!!

```
public class BankAccountTest
{
    [Fact]
    public void NewAccount_BalanceZero()
    {
    }
}
```

10. Unit testen

- ▶ Stap 3: Aanmaken van unit testen
 - Aanmaken van unit test voor constructor BankAccount
 - **Arrange:** Initialiseer de nodige variabelen

```
[Fact]
public void NewAccount_BalanceZero()
{
    //Arrange
    string accountNumber = "123-4567890-02";
}
```

10. Unit testen

▶ Stap 3: Aanmaken van unit testen

- Aanmaken van unit test voor constructor BankAccount
 - **Act:** Voer test effectief uit → roep de te testen methode op

```
[Fact]
public void NewAccount_BalanceZero()
{
    //Arrange
    string accountNumber = "123-4567890-02";
    //Act
    BankAccount account = new BankAccount(accountNumber);
}
```

- Voeg bovenaan de klasse volgende using toe, nodig voor klasse BankAccount
 - using Banking.Models;
 - de foutmelding “BankAccount is unaccessible due to its protection level” => oplossing : pas modifier aan van de class BankAccount. Moet public zijn, daar het gebruikt wordt in een andere assembly

```
public class BankAccount
```

10. Unit testen

- ▶ Stap 3: Aanmaken van unit testen
 - Aanmaken van unit test voor constructor BankAccount
 - **Assert:** vergelijk bekomen resultaat met verwachte resultaat.
 - De klasse Assert
 - Assert.Equal(expected, actual)
 - Test of de waarde van expected gelijk is aan de waarde van actual.
 - Alle primitieve datatypes
 - Vergelijken van objecten gebeurt op reference basis. Dit kan je aanpassen door de **Equals** methode te implementeren
 - Assert.NotEqual(expected, actual): idem maar test op verschillend
 - Assert.(Not)Same(expected, actual): expected en actual wijzen naar hetzelfde object.
 - Assert.True(bool conditie), Assert.False(bool conditie)
 - Test of conditie gelijk is aan true/false

10. Unit testen

- ▶ Stap 3: Aanmaken van unit testen
 - Aanmaken van unit test voor constructor BankAccount
 - De klasse Assert
 - Assert.(Not)Null(actual)
 - Test of actual (niet) gelijk is aan null
 - Assert.Empty(actual)
 - Test of collection leeg is
 - Assert.Contains(item, collection), Assert.DoesNotContain(item, coll)
 - Test of collection item bevat
 - T result = Assert.Is(Not)Type<T>(actual)
 - Test of actual instantie is (exact) van het type T.
 - T result = Assert.IsAssignableFrom<T>
 - Test of actual instantie is van het type T (mag ervan erven)

10. Unit testen

▶ Stap 3: Aanmaken van unit testen

- Aanmaken van unit test voor constructor BankAccount
 - We maken een BankAccount aan. Hiervoor dient de klasse BankAccount public te zijn. Pas aan. Doe dit voor alle klassen in de models folder

```
public class BankAccount : IBankAccount
```

- Assert: vergelijk verwachte resultaat (1ste parameter) met het bekomen resultaat (2de parameter)

```
[Fact]
public void NewAccount_BalanceZero()
{
    //Arrange
    string accountNumber = "123-4567890-02";
    //Act
    BankAccount account = new BankAccount(accountNumber);
    //Assert
    Assert.Equal(0, account.Balance);
}
```

10. Unit testen

▶ Stap 4: Run Test:

- We gebruiken Live Unit Testing.
- We starten dit in Menu > Test > Live Unit Testing > Start
- Alle testen worden uitgevoerd (op dit ogenblik maar één) en slaagt.
- We gaan nu nog testen toevoegen. Van zodra je het bestand 'saved', worden de testen uitgevoerd.
- De test runt, als hij slaagt  . Als faalt 
- Meer informatie over Live unit testing op:
<https://blogs.msdn.microsoft.com/visualstudio/2017/03/09/live-unit-testing-in-visual-studio-2017-enterprise/#integrated>

10. Unit testen

▶ Stap 4: Run Test:

- Je krijgt ook onmiddellijk feedback over de code coverage. Open BankAccount.cs



A line of executable code that is covered by at least one failing test is decorated with a red "X".



A line of executable code that is covered by only passing tests is decorated with a green "√".



A line of executable code that is not covered by any test is decorated it with a blue dash "-".

- Of als je een test aan het wijzigen bent, tot je de testen opnieuw gerund hebt
- Als je op een feedback icon klikt krijg je een overzicht van de testen, en kan je zo de testen opnieuw runnen,... Hover een failed test toont de reden

10. Unit testen

- ▶ Aanmaken van unit test voor constructor BankAccount
 - Oefening: Maak een 2de test aan: Creatie bankaccount, rekeningnummer moet overeenkomstig rekeningnummer zijn
 - De test zal automatisch worden uitgevoerd.

10. Unit testen

► Aanvullingen:

- Beide testmethodes hebben dezelfde arrange/act code
- Oplossing: Voorzie een **SetUp** en **TearDown** methode die runt respectievelijk voor en na de uitvoering van elke test
 - Declareer private variabele die binnen elke testmethode gebruikt kunnen worden in testklasse
 - Setup: maak een parameterloze constructor aan en initialiseer de variabelen
 - VS voert de constructor uit VOOR de uitvoering van iedere testmethode

```
public class BankAccountTest {  
  
    private readonly BankAccount _account;  
    private readonly string _accountNumber;  
  
    public BankAccountTest()  
    {  
        _accountNumber = "123-4567890-02";  
        _account = new BankAccount(_accountNumber);  
    }  
}
```

10. Unit testen

► Aanvullingen:

- Beide testmethodes hebben dezelfde arrange/act code
 - TearDown: Wens je de variabelen op te kuisen na het runnen van elke test:
 - Laat de testklasse erven van **IDisposable**
 - Implementeer de opkuis in de methode **Dispose**
 - VS voert deze methode uit na iedere testmethode-uitvoering

```
public class BankAccountTest : IDisposable
{
    private BankAccount _account;
    private string _accountNumber;

    public BankAccountTest()
    {
        _accountNumber = "123-4567890-02";
        _account = new BankAccount(_accountNumber);
    }
}
```

unit testen

```
public void Dispose()
{
}
```

10. Unit testen

► Aanvullingen:

- Unit test die gebruik maakt van SetUp

```
public class BankAccountTest  {

    private readonly BankAccount _account;
    private readonly string _accountNumber;

    public BankAccountTest()
    {
        _accountNumber = "123-4567890-02";
        _account = new BankAccount(_accountNumber);
    }

    [Fact]
    public void NewAccount_BalanceZero()
    {
        //Assert
        Assert.Equal(0, _account.Balance);
    }

    [Fact]
    public void NewAccount_SetsAccountNumber()
    {
        Assert.Equal(_accountNumber, _account.AccountNumber);
    }
}
```

10. Unit testen

- ▶ Aanmaken test NewAccount_EmptyString_Fails
 - Arrange en Act

```
[Fact]
public void NewAccount_EmptyString_Fails()
{
    _account = new BankAccount(String.Empty);
}
```

- Wat met exceptions?

10. Unit testen

▶ Aanmaken test NewAccount_EmptyString_Fails

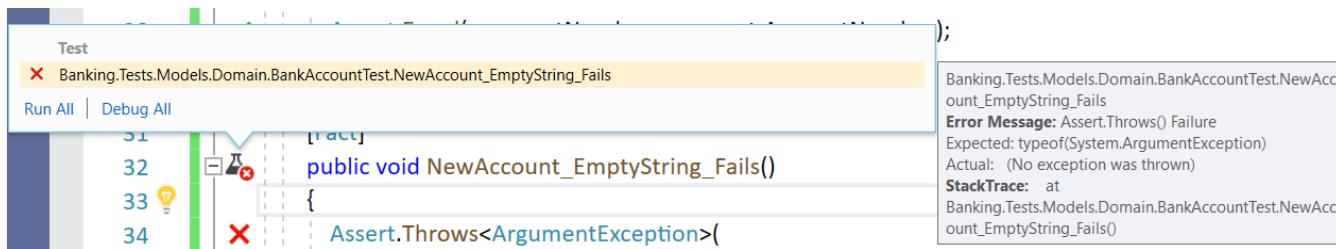
- Testen op Exceptions
 - Als je foutieve parameterwaarden meegeeft aan een test methode moet deze methode een exception throwen.
 - **Assert.Throws<T>(lambda expression) met**
 - **T de exception klasse, controleert of de exception gethrowd wordt**
 - Een lambda expressions. Gebruik de schrijfwijze
 - () => methodeAanroep
 - Lambda's worden uitvoerig behandeld in het volgende hoofdstuk.

```
[Fact]
public void NewAccount_EmptyString_Fails()
{
    Assert.Throws<ArgumentException>(
        () => new BankAccount(String.Empty));
}
```

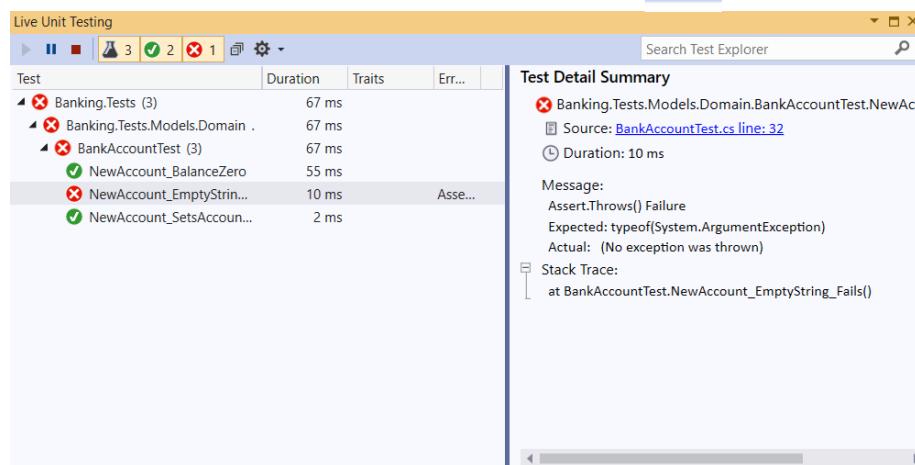
10. Unit testen

▶ Aanmaken test NewAccount_EmptyString_Fails

- De test faalt...



- Of ga naar Test > Live Unit testing Window (versie < 16.3 : Menu Test > Windows > Test Explorer. Klik op icon Open Live Unit Testing tab)



10. Unit testen

▶ Aanmaken test NewAccount_EmptyString_Fails

- Code aanpassen.... Maak van AccountNumber een full property. Selecteer AccountNumber, kies Quick Actions, convert to full property. De code maakt gebruik van lambda's (zie volgend hoofdstuk). Klik terug op Quick Actions en kies « Use block body for property ». Wijzig de naam van het attribuut in _accountNumber.

```
private string _accountNumber;
public string AccountNumber
{
    get
    {
        return _accountNumber;
    }
    private set
    {
        if (value == string.Empty)
            throw new ArgumentException(nameof(AccountNumber), "AccountNumber must have a value");
        _accountNumber = value;
    }
}
```

- ... en de testen slagen

10. Unit testen

► Nog meer testen voor BankAccount...

- Klik in de Solution Explorer op Show All Files icon 
- Rechtsklik op BankAccountTest2.cs > Include in project
- Het **accountnr moet aan bepaalde regels voldoen**, anders wordt een exception gethrowed
- Overzicht van de testen:

Naam test	Rekeningnummer	Gevolg
NewAccount_EmptyString_Fails	string.Empty	ArgumentException
NewAccount_Null_Fails	Null	ArgumentNullException
NewAccount_TooLong_Fails	“133-4567890-0333”	ArgumentException
NewAccount_WrongFormat_Fails	“063-1547563@60 “	ArgumentException
NewAccount_NoDivisionBy97_Fails	“133-4567890-03”	ArgumentException

10. Unit testen

► Nog meer testen voor BankAccount...

- Bekijk de testen. Als ze niet runnen : Test > Live Unit Testing > Stop en dan terug starten
- Enkele nieuwe testen falen

The screenshot shows the 'Live Unit Testing' window in Visual Studio. The top bar displays statistics: 12/13 tests run, 8 passed, 4 failed, and 0 warnings. The main area is divided into two sections: 'Test' and 'Test Detail Summary'.

Test	Duration	Traits	Error Message
Banking.Tests (12)	45 ms		
Banking.Tests.Models.Domain (12)	45 ms		
BankAccountTest (3)	29 ms		
BankAccountTest2 (9)	16 ms		
Deposit_AmountBiggerThanZero_ChangesBalance	1 ms		
Equals_2BankAccountsWithDifferentAccountNumber_ReturnsFalse	1 ms		
Equals_2BankAccountsWithSameAccountNumber_ReturnsTrue	1 ms		
NewAccount_NoDivisionBy97_Fails	8 ms	Assert.Throw	
NewAccount_Null_Fails	1 ms	Assert.Throw	
NewAccount_ToLong_Fails	1 ms	Assert.Throw	
NewAccount_WrongFormat_Fails	1 ms	Assert.Throw	
Withdraw_AmountBiggerThanZero_ChangesBalance (2)	2 ms		

The 'Test Detail Summary' pane on the right shows a detailed view of the failing test 'NewAccount_Null_Fails'. It includes the error message, source code location (BankAccountTest2.cs line 18), duration (1 ms), and stack trace (at BankAccountTest2.NewAccount_Null_Fails()).

- Pas de code aan...

10. Unit testen

► Unit testen voor BankAccount.

- aanpassingen aan de property AccountNumber
- zie help voor Regex, Match

```
private set
{
    if (value == String.Empty)
        throw new ArgumentException("AccountNumber must have a value", nameof(AccountNumber));
    if (value == null)
        throw new ArgumentNullException(nameof(AccountNumber));

    Regex regex = new Regex(@"^(?\d{3})-(?\d{7})-(?\d{2})$");
    Match match = regex.Match(value);
    if (!match.Success)
        throw new ArgumentException("Bankaccount number format is not correct", nameof(AccountNumber));
    int getal = int.Parse(match.Groups["bankcode"].Value + match.Groups["rekeningnr"].Value);
    int checksum = int.Parse(match.Groups["checksum"].Value);
    if (getal % 97 != checksum)
        throw new ArgumentException("97 test of the bankaccount number failed", nameof(AccountNumber));
    _accountNumber = value;
}
```

Deze blok code mag weg want gebruik van de Regex en Match vangt dit op deze manier op

Opm : door @ hoeven we niet \\ te schrijven

(..) laat toe subexpressies te schrijven, die nadien als groep te extraheren zijn

10. Unit testen

► Theory

- Data-driven testen.
- Enkel accountNumber is verschillend in vorige unit testen

InlineData: Alle testgevallen

```
[Theory]
[InlineData("123-4567890-0333")] //too long
[InlineData("123-1547563@60")] //wrong format
[InlineData("123-4567890-03")] //not divisible by 97
public void NewAccount_WrongAccountNumber_Fails(string accountNumber)
{
    Assert.Throws<ArgumentException>(() => new BankAccount(accountNumber));
}
```

1 parameter die de waarde van het testgeval zal bevatten

- ▶ ✓ NewAccount_WrongAccountNumber_Fails (3)
 - ✓ NewAccount_WrongAccountNumber_Fails(accountNumber: "123-1547563@60")
 - ✓ NewAccount_WrongAccountNumber_Fails(accountNumber: "123-4567890-03")
 - ✓ NewAccount_WrongAccountNumber_Fails(accountNumber: "123-4567890-0333")

10. Unit testen

▶ Unit testen voor Deposit/Withdraw die reeds slagen

- Withdraw_AmountBiggerThanZero_ChangesBalance
 - Arrange: nieuwe bankrekening met geldig nummer
 - Act: Deposit 200, Withdraw 100
 - Assert: Balance = 100
 - Maar ook met als de balans onder 0 gaat : bvb 200 storten en dan 300 afhalen
- Deposit_AmountBiggerThanZero_ChangesBalance
 - Arrange: nieuwe bankrekening met geldig nummer
 - Act: Deposit 100
 - Assert: Balance = 100

10. Unit testen

- ▶ Unit testen Deposit/Withdraw falen
 - Maak zelf de testen aan voor
 - Withdraw_NegativeOrZeroAmount_Fails()
 - Deposit_NegativeOrZeroAmount_Fails()
 - run de testen...
 - Pas de code aan en laat ze slagen!

10. Unit testen

▶ Unit testen (TDD) Deposit/Withdraw

- ▶ ✗ Deposit_NegativeOrZeroAmount_Fails (2)
- ▶ ✗ Withdraw_NegativeOrZeroAmount_Fails (2)

oefening: schrijf de gepaste testen

oefening: pas de code aan

- ▶ ✓ Deposit_NegativeOrZeroAmount_Fails (2)
- ✓ Withdraw_NegativeOrZeroAmount_Fails (2)

10. Unit testen

- ▶ We hebben verschillende opties om de validatie te doen
 - In de withdraw en deposit method? Maar dan duplicate code
 - In de Transaction klasse
 - In de constructor
 - Of in een private setter

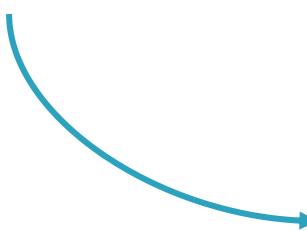
10. Unit testen

▶ Testen van de klasse Transaction

- ▷ ✗ Deposit_NegativeOrZeroAmount_Fails (2)
- ▷ ✗ Withdraw_NegativeOrZeroAmount_Fails (2)



```
#region Constructors
public Transaction(decimal amount, TransactionType type)
{
    if (amount <= 0)
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount must be positive");
    Amount = amount;
    TransactionType = type;
    DateOfTrans = DateTime.Today;
}
```

- 
- ▷ ✓ Deposit_NegativeOrZeroAmount_Fails (2)
 - ✓ Withdraw_NegativeOrZeroAmount_Fails (2)

10. Unit testen

▶ Live unit testing

- In de code kan je per method de testen zien en of ze al dan niet slagen (hover over **v** of **x** voor de methode)

The screenshot shows a code editor interface with a sidebar for tests and the main area for code.

Test sidebar:

- ✓ Banking.Tests.Models.Domain.BankAccountTest2.Withdraw_NegativeOrZeroAmount_Fails(amount: -100)
- ✓ Banking.Tests.Models.Domain.BankAccountTest2.Withdraw_NegativeOrZeroAmount_Fails(amount: 0)
- ✓ Banking.Tests.Models.Domain.BankAccountTest2.Withdraw_AmountBiggerThanZero_ChangesBalance...
- ✓ Banking.Tests.Models.Domain.BankAccountTest2.Withdraw_AmountBiggerThanZero_ChangesBalance...

Run All | Debug All

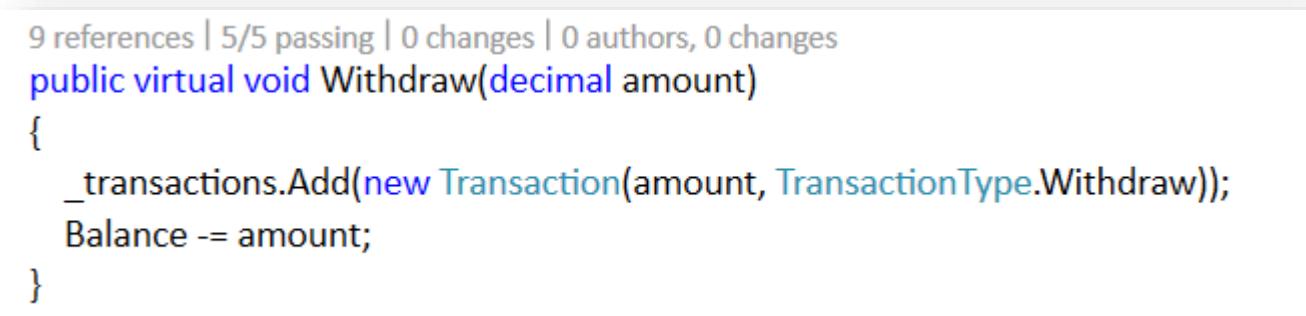
Code area:

```
8 references | 2/4 passing | Karine Samyn, 4 days ago | 1 author, 3 changes
public virtual void Withdraw(decimal amount)
{
    _transactions.Add(new Transaction(amount, TransactionType.Withdraw));
    Balance -= amount;
}
```

10. Unit testen

▶ CodeLens

- In te stellen via Tools > Options > Text Editor > All Languages > Code Lens
- Hiervoor dien je de testen wel te runnen via Test > Windows > Test Explorer en dan “Run all”
- Toont boven elke methode het aantal testen en de aantal testen die slagen (testen moet je gerund hebben)



9 references | 5/5 passing | 0 changes | 0 authors, 0 changes

```
public virtual void Withdraw(decimal amount)
{
    _transactions.Add(new Transaction(amount, TransactionType.Withdraw));
    Balance -= amount;
}
```

10. Unit testen

- ▶ Andere manier om testen te runnen
 - Test > Run all Tests
 - Rechtsklik in een testklasse > Run tests
 - Rechtsklik op een test > Run test
 - Via Test > Test Explorer kan je het resultaat bekijken

Test Explorer			
Test	Duration	Traits	Error Message
✓ Banking.Tests (16)	35 ms		
✓ Banking.Tests.Models.Domain (16)	35 ms		
✓ BankAccountTest (3)	12 ms		
✓ NewAccount_BalanceZero	10 ms		
✓ NewAccount_EmptyString_Fails	1 ms		
✓ NewAccount_SetsAccountNumber	1 ms		
▷ ✓ BankAccountTest2 (13)	23 ms		

10. Unit testen

▶ CodeCoverage

- Test Analyze Code Coverage for All Tests (In versie < 16.3
Menu > Test > Windows > Test explorer. Run eerst alle testen.
Klik dan > Analyze Code Coverage)
 - Toont het % van de code die door de testen getest wordt.

Code Coverage Results				
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
ksa607_NB1100371 2019-09-17 22_41_30.cc	146	57.03%	110	42.97%
banking.dll	134	72.43%	51	27.57%
Banking	50	100.00%	0	0.00%
Banking.Models.Domain	84	62.22%	51	37.78%
BankAccount	61	58.10%	44	41.90%
BankAccount(string)	0	0.00%	5	100.00%
Deposit(decimal)	0	0.00%	6	100.00%
Equals(object)	1	14.29%	6	85.71%
GetHashCode()	6	100.00%	0	0.00%
GetTransactions()	41	100.00%	0	0.00%
ToString()	5	100.00%	0	0.00%
Withdraw(decimal)	5	83.33%	1	16.67%
get_AccountNumber	0	0.00%	2	100.00%
get_Balance()	0	0.00%	1	100.00%
get_NumberOfTransactions	3	100.00%	0	0.00%
set_AccountNumber	0	0.00%	22	100.00%
set_Balance(decimal)	0	0.00%	1	100.00%
SavingsAccount	14	100.00%	0	0.00%
Transaction	9	56.25%	7	43.75%

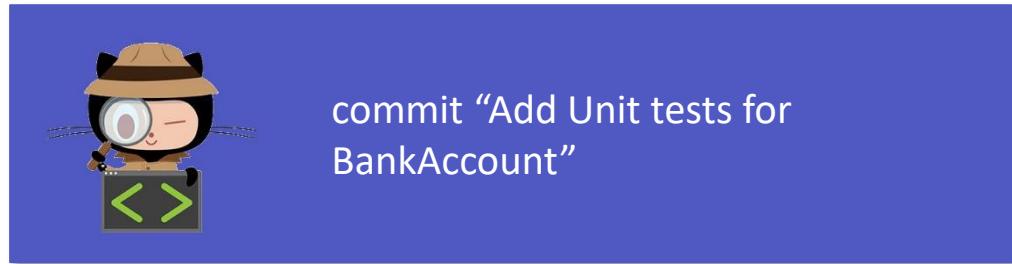
10. Unit testen

▶ Tips

- 1 methode test 1 item. 2 strekkingen
 - Ideaal 1 Assert/test methode
 - OF meerdere asserts/test maar methode test 1 type gedrag
- Nadelen van veel Asserts in 1 methode
 - Als 1 Assert binnen test methode faalt, voert VS de rest methode niet uit
 - De methode wordt moeilijk te lezen
 - De kans dat je in de methode een bug schrijft wordt groter
 - De kans dat je de methode moet debuggen wordt groter
- De werking van 1 test methode mag niet afhangen van de werking van een andere testmethode
- Testmethodes moeten in willekeurige volgorde kunnen uitvoeren.
- Schrijf geen testen voor bestaande libraries die je gebruikt of voor gegenereerde code (get/set)

10. Unit testen

► Time to commit



commit “Add Unit tests for BankAccount”

- Inspecteer de code
- Commit

Changes | Banking

Branch: master

Unit test BankAccount

Commit All Actions ▾

Changes (3)

- ▲ C:\data-karine\20162017\WebIII\Les3\Banking
- ▲ src\Banking\Models
- C# BankAccount.cs
- ▲ test\Banking.Tests\Models
- C# BankAccountTest.cs [add]
- C# BankAccountTestDeel2.cs [add]

10. Unit testen

▶ Testen van de klasse Transaction

- Voeg de klasse **TransactionTest** aan de Models folder (rechtsklik Include in project)
- Run de testen in de klasse TransactionTest
- De testen slagen allemaal

►	✓ TransactionTest (9)	16 ms
	✓ IsDeposit_IfDeposit_ReturnsTrue	3 ms
	✓ IsDeposit_IfWithDraw_ReturnsFalse	1 ms
	✓ IsWithDraw_IfDeposit_ReturnsFalse	1 ms
	✓ IsWithDraw_IfWithDraw_ReturnsTrue	1 ms
▷	✓ NewTransaction_NegativeOrZeroAmount_Fails (2)	2 ms
	✓ NewTransaction_SetsAmount	1 ms
	✓ NewTransaction_SetsDateOfTrans	4 ms
	✓ NewTransaction_SetsTransactionType	3 ms

10. Unit testen

▶ Testen van BankAccount Transaction

- Voeg BankAccountTransactionTest.cs toe aan project
- De testen zijn nog niet volledig geïmplementeerd
 - Deze testen worden aangeduid met de Skip parameter

```
[Fact(Skip="Not yet implemented")]
public void WithDraw_Amount_AddsTransaction()
```



WithDraw_Amount_AddsTransaction

- Implementeer deze testen, verwijder de Skip parameter.
- Bekijk de overige

10. Unit testen

▶ Aanpassen van BankAccountTransactionTest

- Deposit_Amount_AddsTransaction moet nagaan of transactie is toegevoegd.
 - Probleem: I Enumerable laat enkel toe om de collectie te overlopen. Aantal en elementen via index kunnen niet worden opgevraagd
 - vorm de I Enumerable om omdat we toegang moeten krijgen tot het eerste element uit de collectie...

```
[Fact]
public void Deposit_Amount_AddsTransaction()
{
    _bankAccount.Deposit(100);
    Assert.Equal(1, _bankAccount.NumberOfTransactions);
    //Test of de toegevoegde transactie de juiste gegevens bevat
    Transaction t = _bankAccount.GetTransactions(DateTime.Today, DateTime.Today).ToArray()[0];
    Assert.Equal(100, t.Amount);
    Assert.Equal(TransactionType.Deposit, t.TransactionType);
}
```

ToList() kan je ook gebruiken

10. Unit testen

▶ Aanpassen van BankAccountTransactionTest

- GetTransactions_NoParameters_ReturnsAllTransactions()
moet nagaan of alle transacties gereturneerd worden.
 - Probleem: I Enumerable laat enkel toe om de collectie te overlopen. Aantal en elementen via index kunnen niet worden opgevraagd

[Fact]

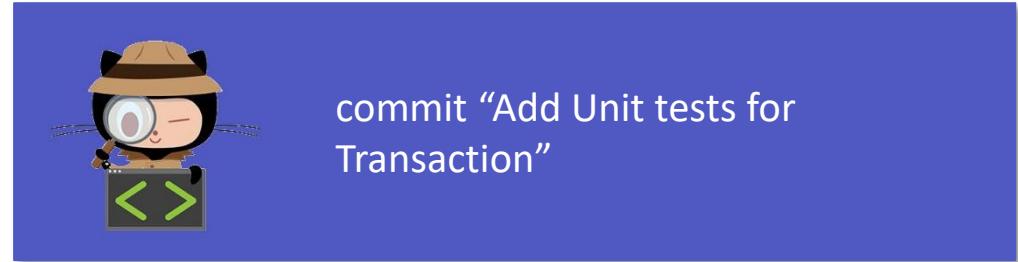
```
public void GetTransactions_NoParameters_ReturnsAllTransactions()
{
    _bankAccount.Deposit(100);
    _bankAccount.Deposit(100);
    Transaction[] t = _bankAccount.GetTransactions(null, null).ToArray();
    Assert.Equal(2, t.Length);
}
```

Of new List<T>() of ToList(), maar dan de Count property gebruiken

```
List<Transaction> t = new List<Transaction>(_bankAccount.GetTransactions(null, null));
Assert.Equal(2, t.Count);
```

10. Unit testen

▶ Time to commit



commit “Add Unit tests for Transaction”

- Inspecteer de code
- Commit

10. Unit testen

▶ SavingsAccount

- Voeg SavingsAccountTest.cs toe aan Models folder Banking.Tests
- run de testen...
- Voeg 2 testen toe
 - Implementeer de test Withdraw_IfBalanceGetsNegative_Fails
 - Je mag niet in het rood gaan op een SavingsAccount. Dit throwt een InvalidOperationException (TDD)
 - Doe de test eerst falen
 - Pas de code aan
 - De test moet slagen

10. Unit testen

▶ SavingsAccount

- Implementeer de test AddInterest_ChangesBalance
 - De test zal onmiddellijk slagen, want de code is reeds geïmplementeerd in SavingsAccount.
 - Het is best practice om een test altijd eerst te laten falen. Plaats daarom de code in SavingsAccount die de test doet slagen eerst in commentaar
 - Dan faalt de test
 - Dan plaats je de code weer uit commentaar
 - Dan moet de test slagen

10. Unit testen

▶ SavingsAccount

- Voeg twee testen toe:

```
[Fact]
```

```
public void Withdraw_IfBalanceGetsNegative_Fails()
{
    Assert.Throws<InvalidOperationException>(() => _savingsAccount.Withdraw(200));
}
```

```
[Fact]
```

```
public void AddInterest_ChangesBalance()
{
    _savingsAccount.AddInterest();
    Assert.Equal(204, _savingsAccount.Balance);
}
```

10. Unit testen

▶ SavingsAccount

- Voeg twee testen toe:

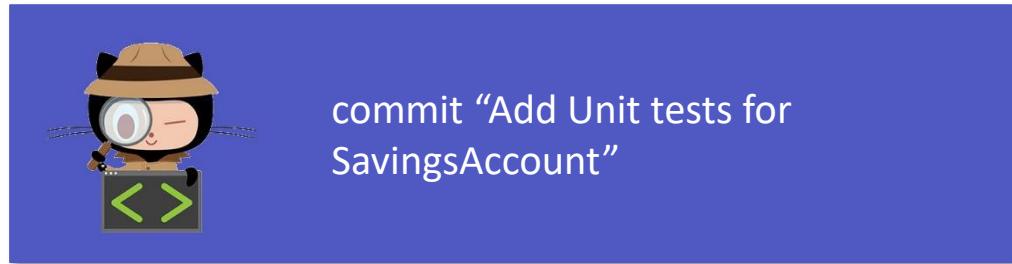
```
public override void Withdraw(decimal amount)
{
    if (amount + WithdrawCost > Balance)
        throw new InvalidOperationException("Balance cannot be negative");
    base.Withdraw(amount);
    base.Withdraw(WithdrawCost);
}
```



- ✓ SavingsAccountsTests (5)
- ✓ AddInterest_ChangesBalance
- ✓ NewSavingsAccount_SetInterestRate
- ✓ Withdraw_Amount_AddsCosts
- ✓ Withdraw_Amount_CausesTwoTransactions
- ✓ Withdraw_IfBalanceGetsNegative_Fails

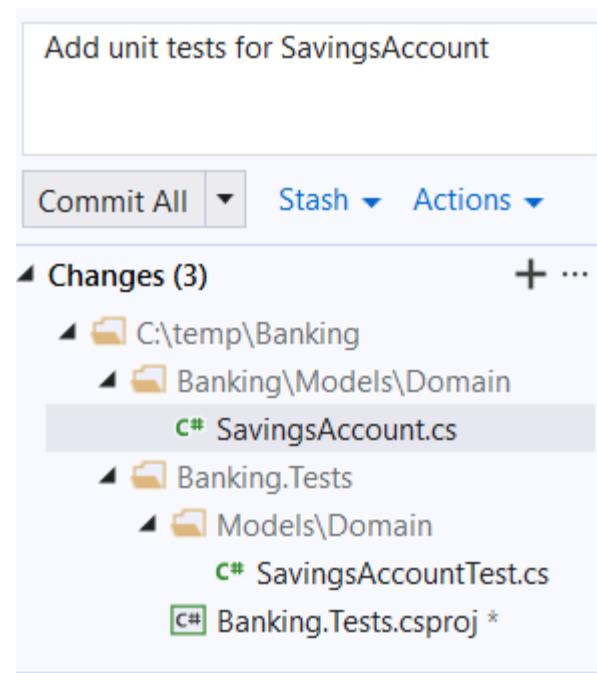
10. Unit testen

► Time to commit



commit “Add Unit tests for SavingsAccount”

- Inspecteer de code
- Commit



10. Unit testen

- ▶ Advanced: use Theory and MemberData
 - Ga terug naar de Master branch. Bekijk de klasse BankAccountTransactionTest.cs
 - Nu wordt gebruik gemaakt van Theories. Maar daar de Inline data geen constante waarden bevat dient een MemberData object te worden aangemaakt
 - Bekijk de code. Meer info op <http://www.martinwilley.com/net/code/test/parametrized.html>
 - commit “Refactor unit test BankAccountTransaction: gebruik van Theory and MemberData”



commit “Refactor unit test
BankAccountTransaction: gebruik van
Theory and MemberData”

10. Unit testen

▶ Refactor de code

- Maak van Balance in BankAccount een berekende property.
 - Overloop alle transacties en bepaal zo het total
 - Alle testen zouden nog steeds moeten slagen.



Appendix

Appendix : Naming Conventions

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChange
Exception class	Pascal	WebException Note Always ends with the suffix Exception .
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable Note Always begins with the prefix I .
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance field	Camel	redValue Note Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	RedValue Note Rarely used. A property is preferable to using a public instance field.

Appendix : Documenteren C# code

- ▶ Start met /// (triple slash) gevolgd door 1 van onderstaande XML elementen
 - Rechtstreeks in code toevoegen
 - Of in Class Diagram > Details View > Summary kolom

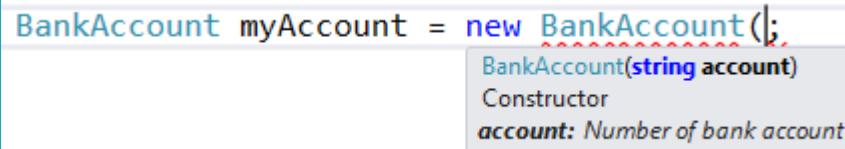
Predefined XML Documentation	
Element	Meaning in Life
<c>	Indicates that the following text should be displayed in a specific “code font”
<code>	Indicates multiple lines should be marked as code
<example>	Mocks up a code example for the item you are describing
<exception>	Documents which exceptions a given class may throw
<list>	Inserts a list or table into the documentation file
<param>	Describes a given parameter
<paramref>	Associates a given XML tag with a specific parameter
<permission>	Documents the security constraints for a given member
<remarks>	Builds a description for a given member
<returns>	Documents the return value of the member
<see>	Cross-references related items in the document
<seealso>	Builds an “also see” section within a description
<summary>	Documents the “executive summary” for a given member
<value>	Documents a given property

Appendix : Documenteren C# code

- Voorbeeld

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="account">Number of bank account</param>
public BankAccount(string account)
```

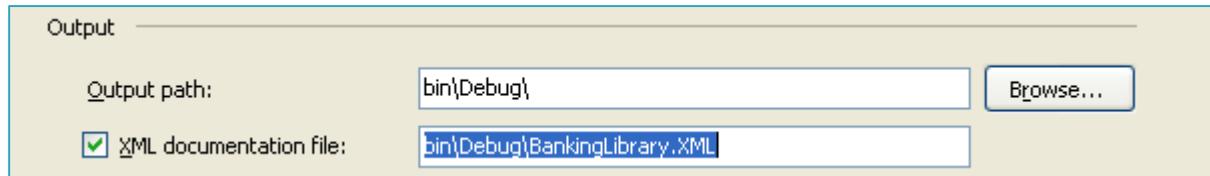
- In VS: documentatie wordt getoond



Appendix : Documenteren C# code

▶ Generatie xml file

- Via command prompt VS
 - Csc /doc:DocBankAccount.xml c:/..../*.cs
- Instellen in VS, wordt bij compilatie automatisch gegenereerd
 - Selecteer project: bvb BankingLibrary. Ga naar de Build tab van zijn Properties. Vul pad en filenaam in



▶ Generatie andere formaten (html,...) vertrekkende van xml file:

- NDoc: <http://sourceforge.net/projects/ndoc>

Appendix: Collections

Many collection classes...

		ReadOnlyCollection<T>
SortedList<TKey, TValue>	LinkedList<T>	
HashSet<T>	Stack<T>	ObservableCollection<T>
Collection<T>	Array	SortedSet<T>
KeyedCollection<TKey, TItem>		Dictionary<TKey, TValue>



Lists

Dictionaries

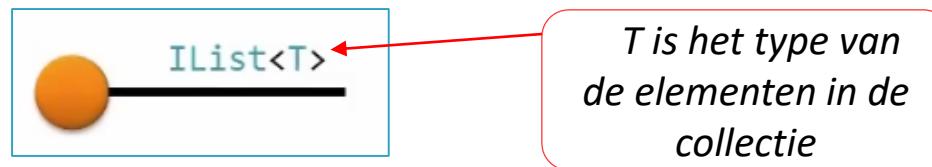
Sets

Appendix: Collections



▶ List collecties

- elementen van een collectie zijn toegankelijk via een **index**
 - zero-based indexing
- het contract voor op **index gebaseerde collecties** ligt vast in **IList<T>**



- List collecties zijn efficient
 - geheugengebruik
 - snelheid om elementen te benaderen

Appendix: Collections

Dictionary<TKey, TValue>

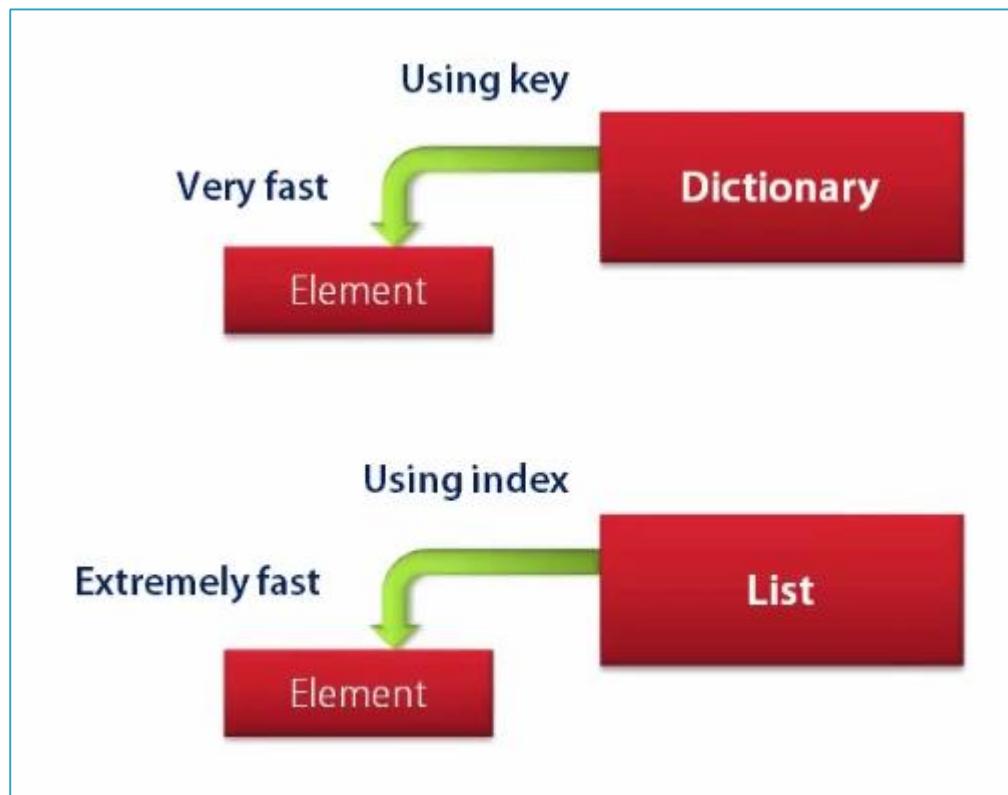
▶ Dictionaries

- elementen in de collectie bevatten een **key-value pair**
- elementen van de collectie zijn toegankelijk via de **key**
- het contract voor dictionaries ligt vast in **IDictionary<TKey, TValue>**



- dictionaries zijn meestal geïmplementeerd als een hash tabel

Appendix: Collections



Appendix: Collections

▶ `IDictionary<Tkey, TValue>`

```
 IDictionary<int, string> klasLijst = new Dictionary<int, string>();  
 klasLijst.Add(1, "Jan Peterson");  
 klasLijst.Add(2, "Steven Spielberg");  
 foreach (int i in klasLijst.Keys)  
     Console.WriteLine(i.ToString() + " : " + klasLijst[i]);  
 foreach (string s in klasLijst.Values)  
     Console.WriteLine(s);  
 Console.WriteLine(klasLijst[1]);  
 Console.WriteLine(klasLijst.Count);  
 Console.ReadLine();
```

- Of verkort met dictionary initializer

```
 IDictionary<int, string> klaslijst = new Dictionary<int, string>  
{  
     [1] = "Jan Peterson",  
     [2] = "Steven Spielberg"
```

```
1 : Jan Peterson  
2 : Steven Spielberg  
Jan Peterson  
Steven Spielberg  
Jan Peterson  
2
```

Appendix: Collections

▶ `IDictionary<Tkey, TValue>`

- Methods

• Add(TKey, TValue)	Adds the specified key and value to the dictionary.
• Clear()	Removes all keys and values from the <code>Dictionary<TKey, TValue></code> .
• ContainsKey(TKey)	Determines whether the <code>Dictionary<TKey, TValue></code> contains the specified key.
• ContainsValue(TValue)	Determines whether the <code>Dictionary<TKey, TValue></code> contains a specific value.
• Equals(Object)	Determines whether the specified object is equal to the current object.(Inherited from Object .)
• Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object .)
• GetEnumerator()	Returns an enumerator that iterates through the <code>Dictionary<TKey, TValue></code> .
• GetHashCode()	Serves as the default hash function. (Inherited from Object .)
• GetObjectData(SerializationInfo, StreamingContext)	Implements the System.Runtime.Serialization.ISerializable interface and returns the data needed to serialize the <code>Dictionary<TKey, TValue></code> instance.
• GetType()	Gets the Type of the current instance.(Inherited from Object .)
• MemberwiseClone()	Creates a shallow copy of the current Object .(Inherited from Object .)
• OnDeserialization(Object)	Implements the System.Runtime.Serialization.ISerializable interface and raises the deserialization event when the deserialization is complete.
• Remove(TKey)	Removes the value with the specified key from the <code>Dictionary<TKey, TValue></code> .
• ToString()	Returns a string that represents the current object.(Inherited from Object .)
• TryGetValue(TKey, TValue)	Gets the value associated with the specified key.

Appendix: Collections

- ▶ Dictionary<Tkey, TValue>
 - Concrete collectie met key/waarde paren
- ▶ SortedDictionary<Tkey, TValue>
 - Collectie key/waarde parent, gesorteerd op de key

Appendix: Collections

HashSet<T>

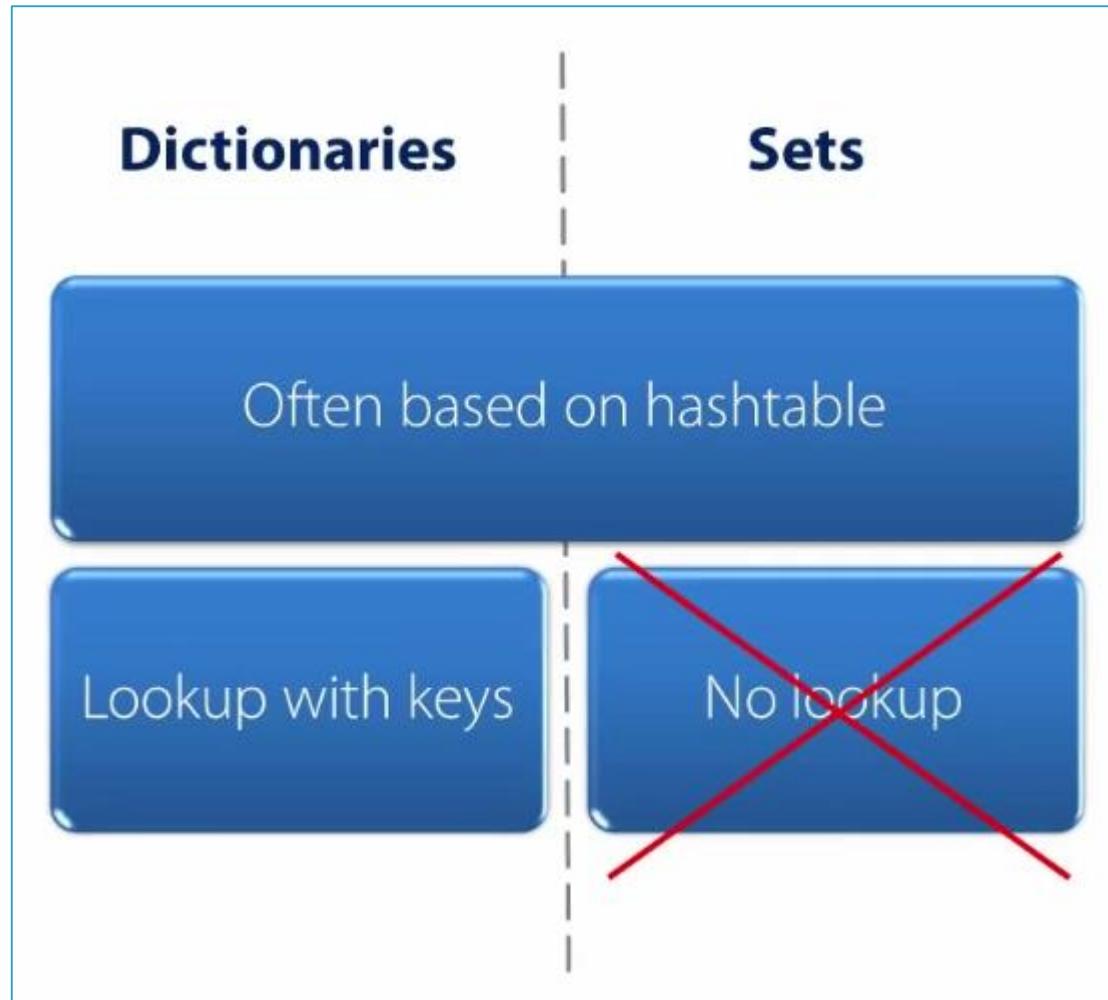
▶ Sets

- focus van deze collecties is niet zozeer op de afzonderlijke elementen maar op al de elementen in de collectie samen
 - er is geen lookup mechanisme om 1 element van een set op te halen
- sets kan je gemakkelijk combineren
 - ~verzamelingen: unie, verschil, ...
- het contract voor sets ligt vast in ISet<T>

ISet<T>

Appendix: Collections

▶ Dictionaries vs Sets



Appendix: Collections

► Raadplegen van collecties

Enumerating

All collections

Looking up items

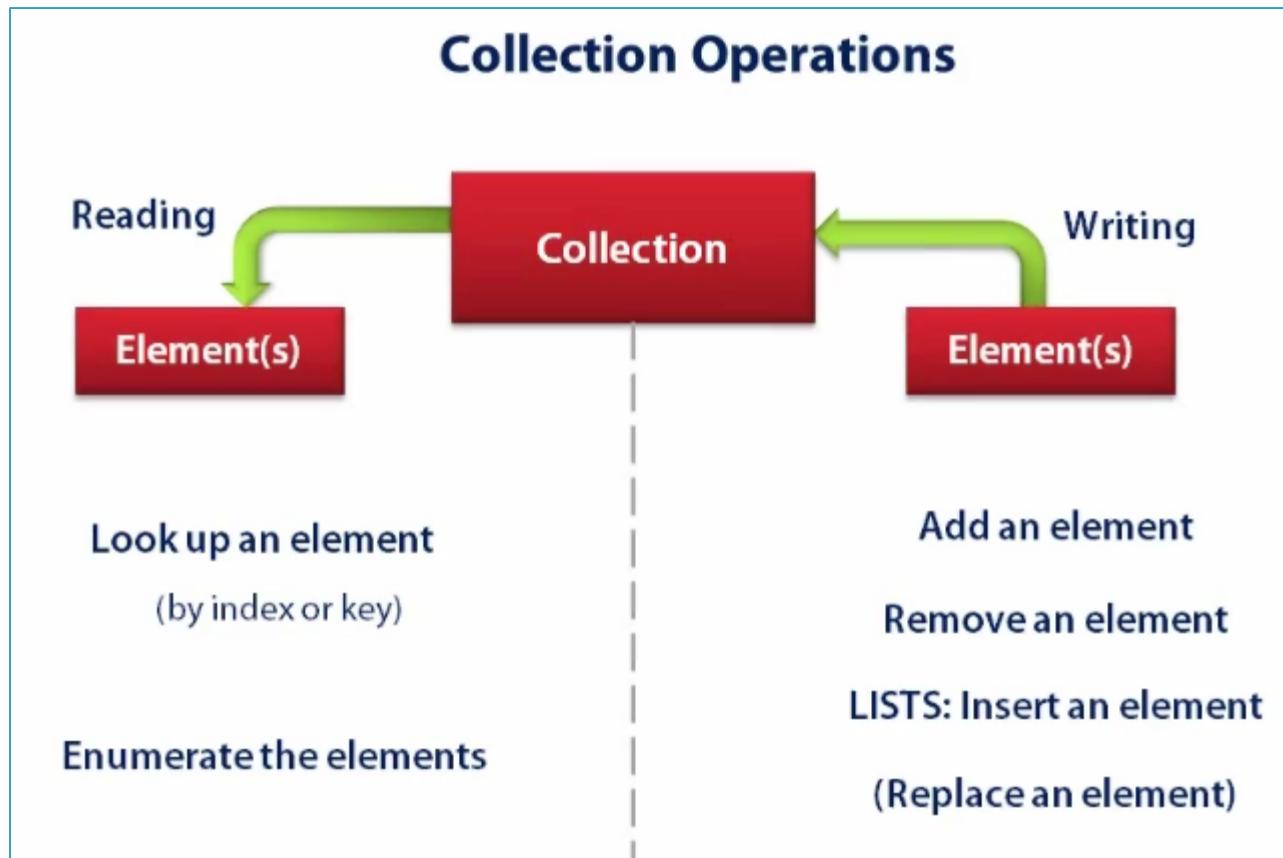
Many collections

NOT: Sets

NOT: Linked lists,
Stacks, Queues

Appendix: Collections

▶ Wijzigen van collecties



Appendix: Collections

▶ Overzicht C# collections

Array	<code>using System;</code>	
Old .NET 1.0 collections	<code>using System.Collections;</code> <code>using System.Collections.Specialized;</code> Obsolete	<i>deze non-generic types (zoals ArrayList) kan je nog tegenkomen (backwards compatibility)</i>
Core generic collections	<code>using System.Collections.Generic;</code> <code>using System.Collections.ObjectModel;</code>	
Concurrent collections	(.NET 4.0 and later only) <code>using System.Collections.Concurrent;</code>	
Immutable collections	(.NET 4.5 only - via NUGET package) <code>using System.Collections.Immutable;</code>	<i>deze collections zal je het meest courant gebruiken</i>

Appendix: Collections

▶ Overzicht C# collections

Array

```
using System;
```

~~Old .NET 1.0 collections~~

```
using System.Collections;
using System.Collections.Obsolete;
```

Obsolete

Core generic collections

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
```

Concurrent collections

```
(.NET 4.0 and later only)
using System.Collections.Concurrent;
```

Immutable collections

```
(.NET 4.5 only - via NUGET package)
using System.Collections.Immutable;
```

+ LINQ Extension methods

dit wordt in een apart hoofdstuk behandeld: zie Hfst 05Linq

Appendix: Collections

▶ ARRAY

- is een reference type
- **declaratie/initialisatie**

```
int[] i;  
  
int[] j = new int[4];
```

- **array initializers**

```
int[] numbers = new int[] { 1, 4, 9, 16, 25 };
```

```
int[] numbers2 = { 1, 4, 9, 16, 25 };
```

Compiler turns this...

```
int eight = 8;  
int[] squares = new int[] {  
    1,  
    2 * 2,  
    eight + 1,  
    int.Parse("16"),  
    (int)Math.Sqrt(625)  
};
```

...into (roughly) this

```
int eight = 8;  
int[] x5 = new int[5];  
x5[0] = 1;  
x5[1] = 2*2;  
x5[2] = eight + 1;  
x5[3] = int.Parse("16");  
x5[4] = (int)Math.Sqrt(625);
```

Appendix: Collections

▶ ARRAY

- Enumereren: **for/foreach**

```
int[] numbers = { 1, 4, 9, 16, 25 };

for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine(numbers[i]);
}

foreach(int i in numbers)
{
    Console.WriteLine(numbers[i]);
}
```

Appendix: Collections

▶ ARRAY

- Enumereren:
 - met foreach kan je de elementen van de array **niet vervangen**
 - als de elementen een reference type zijn kan je wel de **elementen veranderen** (via de reference)

```
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = numbers[i] + 1;
}

foreach(int i in numbers)
{
    i = i + 1;
}
```

(local variable) int i
Cannot assign to 'i' because it is a 'foreach iteration variable'

Appendix: Collections

▶ ARRAY

- Wat kan je doen met array's?
 - zie **documentatie op msdn: Array Class!**
 - enkele properties...
 - Length
 - enkele methods...
 - BinarySearch
 - FindAll
 - Sort
 - Copy/CopyTo
 - IndexOf
 - ...

Appendix: Collections

- ▶ nog meer collections:
 - Queue<T>
 - first-in, first-out collectie van objecten
 - Stack<T>
 - Represents a variable size last-in-first-out (LIFO) collection of instances of the same arbitrary type.
 - LinkedList<T>
 - Represents a doubly linked list.
 - SortedSet<T>
 - Represents a collection of objects that is maintained in sorted order.

Appendix: Collections

▶ yield return

- Voor het bouwen van een `IEnumerable`

```
private IEnumerable<int> ComputeAges()
{
    yield return 21;
    yield return 22;
    for (int i = 23; i < 32; i++)
        yield return i;
}
```

```
foreach(int age in ComputeAges())
    Console.WriteLine(age.ToString());
```

De eerste iteratie in `foreach` loop zorgt voor de uitvoering van de `ComputeAges` t.e.m. het eerste `yield return` statement. Deze iteratie retourneert 21, en de huidige locatie in de `ComputeAges` methode wordt behouden.

Bij de volgende iteratie in `foreach` gaat de uitvoering in de iteratie methode `ComputeAges` verder en wordt 22 geretourneerd,... tot einde iteratie methode bereikt

Appendix: Generieke klassen

- ▶ Generieke klassen kapselen operaties, die niet specifiek voor een bepaald gegevenstype zijn, in.
- ▶ Het meest voorkomende gebruik voor generieke klassen zijn collecties zoals linked lists, hash tables, stacks, queues, trees, enzoverder.
- ▶ Bewerkingen zoals het toevoegen en verwijderen van objecten uit de collectie worden uitgevoerd op dezelfde manier, ongeacht het type gegevens dat wordt opgeslagen.

Appendix: Generieke klassen

```
public class MagischeHoed<T>
{
    private IList<T> dingen = new List<T>();
    public void Add(T ding)
    {
        dingen.Add(ding);
    }
}
```

```
Konijn roger = new Konijn("Roger");
MagischeHoed<Konijn> hoed = new MagischeHoed<Konijn>();
hoed.Add(roger);
```

Appendix: Generieke klassen

- ▶ Je kan constraint toevoegen aan T
 - Je kan eisen dat T een class of struct is
 - Je kan eisen dat T een public default constructor heeft
 - Nodig als je ergens in een methode de default constructor aanroeft, anders krijg je compilatiefout
 - Je kan eisen dat T erft van een interface of van base class

```
public class MagischeHoed<T> where T : class, new()
```

```
public class MagischeHoed<TDier> where TDier : IDier
```

Appendix: Oefening

- ▶ Tutorial : [https://docs.microsoft.com/en-us/dotnet/csharp/tutorials\(nullable-reference-types\)](https://docs.microsoft.com/en-us/dotnet/csharp/tutorials(nullable-reference-types))

Referenties

- ▶ Unit Testing with Visual Studio: Freeman, A. (2014). *Pro ASP.NET MVC 5* (p. 784). Apress, hoofdstuk 6, p137 – p145.
- ▶ Pluralsight: cursus C# Fundamentals with C# 5.0 van Scott Allen
- ▶ Pluralsight: cursus C# Collections Fundamentals van Simon Robinson

Documentatie en Tutorials

- ▶ Tutorial:
 - <https://www.microsoftvirtualacademy.com/>
 - <http://www.csharp-station.com/Tutorial.aspx>
- ▶ C# Programming Guide:
 - <http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx>
- ▶ C# Reference:
 - <http://msdn.microsoft.com/en-us/library/618ayhy7.aspx>