

Object Oriented Design

Syllabus



Academiejaar 2022-2023

1.0

Versie

Wim Vanden Broeck

Auteur(s)

HISTORIEK

Datum	Versie	Omschrijving	Auteur
09/04/2022	1.0		Wim Vanden Broeck

VOORAF

Dit document maakt gebruik van de volgende stijlen en conventies.



Bijkomende informatie.



Idee, voorbeeld of tip.



Waarschuwing, belangrijke info.



Verdieping.



Vraag aan de lezer.



Doelstelling.



Oefening, taak of opdracht voor de lezer.

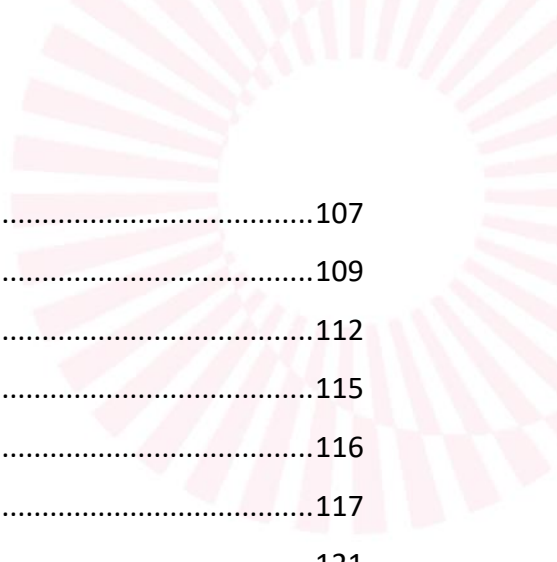


Bijkomend materiaal of referentie.

INHOUD

1 INLEIDING.....	7
1.1 CURSUS.....	7
1.1.1 Positie van deze cursus in je opleiding	7
1.1.2 Positie van deze cursus in je rol als ontwikkelaar.....	7
2 OBJECT ORIENTED DESIGN	8
2.1 HERHALING OO-PROGRAMMING.....	8
2.1.1 Class	8
2.1.2 Inheritance	10
2.1.3 Interface.....	10
2.1.4 Oefening.....	11
2.2 REFACTOREN	12
2.3 DRY	12
2.4 KISS.....	15
2.5 SOLID	17
2.5.1 Single Responsibility Principle (SRP)	17
2.5.2 Open/Closed Principle	17
2.5.3 Liskov's Substitution Principle (LSP)	18
2.5.4 Interface Segregation Principle (ISP)	19
2.5.5 Dependency Inversion Principle (DIP)	21
2.6 CONCLUSIE.....	24
3 DESIGN PATTERNS	25
3.1 SOORTEN DESIGN PATTERNS	25
3.1.1 Creational Design Pattern	25
3.1.2 Structural Design Pattern.....	26
3.1.3 Behavioral Design Pattern	26
3.2 ARCHITECTURAL PATTERNS	26
3.3 SINGLETON	27
3.4 INVERSION OF CONTROL (IOC)	30

3.4.1 Dependency Injection via constructor	31
3.4.2 Dependency Injection via Property	32
3.4.3 Dependency Injection via Method Injection	33
3.4.4 Dependency Injection via interfacing	34
3.4.5 IoC Container	35
3.5 ABSTRACT FACTORY	40
4 MODEL-VIEW-CONTROLLER	45
4.1 MODEL	45
4.2 VIEW	46
4.3 CONTROLLER	46
4.4 PRAKTISCH: MVC CONSOLE APP	47
4.5 PRAKTISCH: MVC IN ASP.NET CORE	53
4.6 VIEWMODEL	63
5 UNIT TESTING	66
5.1 INLEIDING	FOUT! BLADWIJZER NIET GEDEFINIEERD.
5.2 ONZE EERSTE UNIT TESTEN	67
5.3 TEST DRIVEN DEVELOPMENT	75
5.3.1 Praktisch.....	76
5.4 MOCKING	80
5.5 CODE COVERAGE	86
6 ASYNCHROON PROGRAMMEREN.....	87
6.1 INLEIDING	FOUT! BLADWIJZER NIET GEDEFINIEERD.
6.2 PRAKTISCH: VAN SYNC NAAR ASYNC.....	87
6.3 ASYNC-AWAIT PATTERN	96
6.4 ASYNCHROON TAKEN ANNULEREN	97
7 ROBUUST PROGRAMMEREN.....	101
7.1 VALIDATIES	101
7.1.1 Standaard validaties.....	101
7.1.2 Custom validaties.....	104



7.1.3 Validaties over meerdere velden.....	107
7.2 EXCEPTIONS	109
7.2.1 Environments	112
7.3 LOGGING	115
7.3.1 Loglevel	116
7.3.2 Praktisch voorbeeld	117
7.3.3 Logging naar file	121

1 Inleiding

Deze cursus zal je begeleiden in je eerste stappen als ontwerper van code. Dit hoofdstuk zal je alle nodige achtergrond leveren in verband met de cursus, technologie en tools die we zullen gebruiken tijdens het verdere verloop ervan.

Heb je opmerkingen of verbeteringen voor deze cursus, gelieve dan een e-mail te sturen met je bevindingen naar wim.vandenbroeck@ucll.be.

1.1 Cursus

Het hoofddoel van de cursus “Object Oriented Design” is studenten de technieken aanleren om hun code beter onderhoudbaar te maken en ontwerpprincipes aan te leren die de code robuuster maken. De cursus levert een ondersteunende rol in je ontwikkeling als programmeur. Door termen zoals abstractie en design pattern nader te bekijken, zal je leren hoe je je code moet opbouwen en onderhouden.

1.1.1 Positie van deze cursus in je opleiding

Object Oriented Design (OOD) heeft als doel de programmeur te ondersteunen in ontwerpbeslissingen. De student heeft kennis nodig van C#, alsook van Object Oriented programmeren. Dit wordt gezien in C# basis en gevorderd.

OOD bouwt verder op (abstracte) classes en interfaces, en combineert deze elementen tot nieuwe herbruikbare gehelen. Het laat de student proeven van de volledige kracht van OO programmeren, en de voor- en nadelen van ontwerpkeuzes.

OOD bereidt de student voor op Geïntegreerd project 3 en Geïntegreerd project 4.

1.1.2 Positie van deze cursus in je rol als ontwikkelaar

OOD geeft de student de nodige tools om een logisch model te vertalen naar een technisch ontwerp. Het gebruikt hiervoor standaarden in de sector, die de student dan leert implementeren. Dit laat de student toe om sneller code te schrijven en om zich sneller in te werken in bestaande code.

2 Object Oriented Design

In de opleiding Graduaat Programmeren leer je als student hoe je software maakt. Dit kan je doen in verschillende talen (C#, Java, Python, ...) en voor verschillende omgevingen (desktop, web, ...). Software verschilt van hardware in dat het makkelijker aan te passen is. Een programma aanpassen en opnieuw compileren is makkelijker dan een hardware aanpassing te doen, zeker in het verleden wanneer hardware duurder en fysiek groter was.

Als ontwikkelaar moeten we ervoor zorgen dat de software die we schrijven correct werkt, maar ook dat de code zelf onderhoudbaar is. Hiermee bedoelen we dat:

- Code moet leesbaar en begrijpbaar zijn voor andere ontwikkelaars
- Code moet makkelijk uit te breiden zijn met nieuwe functionaliteit
- Bestaande functionaliteit moet makkelijk aan te passen zijn

In OOD gaan we verschillende technieken zien om code op lange termijn onderhoudbaar te maken. Dit doen we door de levensloop van een programma te bekijken, principes mee te geven waaraan een ontwikkelaar zich kan vasthouden en technische designs die een ontwikkelaar kan gebruiken om robuustere code te schrijven.


Om te starten met de cursus OOD moeten we eerst enkele basisbegrippen bekijken. Dit zijn onze fundamenteen waarop we praktijkgerichte voorbeelden kunnen opbouwen.

2.1 Herhaling OO-programming

Om te beginnen met Object Oriented Design, moeten we natuurlijk nog weten wat objecten, classes en interfaces zijn. In dit hoofdstuk herhalen we (beknopt) deze termen.

2.1.1 Class

Een class beschrijft de blauwdruk van een object. Een class kan properties en functies bevatten. We gaan classes aanmaken voor herbruikbaarheid van code en het centraliseren van logica. Stel, we willen persoonsgegevens opvragen voor verschillende gebruikers. Zonder classes zou dit zoiets worden:



```
var person1Name = Console.ReadLine();
var person1FirstName = Console.ReadLine();
var person1Age = DateOnly.Parse(Console.ReadLine());

var person2Name = Console.ReadLine();
var person2FirstName = Console.ReadLine();
var person2Age = DateOnly.Parse(Console.ReadLine());

var person3Name = Console.ReadLine();
var person3FirstName = Console.ReadLine();
var person3Age = DateOnly.Parse(Console.ReadLine());
```

Als we deze gegevens in een herbruikbare lijst willen steken, wordt dit direct zeer complex. Om personen te verwijderen of aan te passen moeten we ook zelf redelijk wat code schrijven, waardoor het onderhoud van deze code redelijk zwaar wordt.

Als we dit zouden herschrijven naar een class, zou dit eerder iets als volgt worden:

```
List<Person> persons = new List<Person>();
while (true)
{
    persons.Add(new Person
    {
        Name = Console.ReadLine(),
        Firstname = Console.ReadLine(),
        Birthdate = DateOnly.Parse(Console.ReadLine())
    });
}
```

De Person class zou dan kunnen zijn:

```
public class Person
{
    public string Name { get; set; }
    public string Firstname { get; set; }
    public DateOnly Birthdate { get; set; }
}
```

Door deze logica te centraliseren onder een class, kunnen we er ook complexere bewerkingen mee doen.

2.1.2 Inheritance

Classes kunnen overerven van elkaar. Dit zorgt ervoor dat we gedragingen en eigenschappen kunnen automatisch gebruiken in een nieuwe class. Deze nieuwe class kan dan ook zijn eigen gedragingen en eigenschappen toevoegen.

Het grote voordeel hiervan is het gebruik van **polymorfisme**. Dit wil zeggen dat een parent class geïmplementeerd kan worden als één van zijn child classes.

```
Mammal myDog = new Dog();
```

In dit geval is er een parent class Mammal. De Dog class **erft** hiervan over. Zo is het mogelijk om een Dog object aan te maken in een definitie van een Mammal.

In versimpelde termen kunnen we zeggen dat overerving gebruikt wordt voor een *is-relatie*.

2.1.3 Interface

De grootste beperking van overerving in C# en Java, is dat we slechts van één parent class kunnen overerven. Soms willen we echter gedragingen en eigenschappen gebruiken die niet echt tot de overervingsboom behoren. Via interfaces kunnen we groeperen op gemeenschappelijk gedragingen en eigenschappen, zonder dat we rekening moeten houden met een overervingsboom. We zeggen dan dat een class een interface **implementeert**.

In versimpelde termen gaat het bij interfaces over een *kan-relatie*.

2.1.4 Oefening



Zie **Oefeningenboek** – 2.1 Herhaling programmeren

2.2 Refactoren

Refactoren is de naam dat we geven aan bestaande code te verbeteren. We gaan functioneel niets wijzigen aan de code, dus het eindresultaat blijft hetzelfde, maar we gaan wel onze code leesbaarder of makkelijker in gebruik maken. Het makkelijkste voorbeeld is het hernoemen van een variabele. Stel, je hebt ergens in het programma staan:

```
string test = "Wim";
```

We zouden deze lijn code dan kunnen refactoren naar:

```
var username = "Wim";
```

Functioneel doet dit niets, wij blijven met een variabele van het type string zitten, die als waarde Wim heeft. Qua leesbaarheid zijn we er wel op vooruit gegaan, omdat we gebruik maken van **var** in plaats van **string** en omdat onze variabele een omschrijvende naam heeft gekregen.

Veel van de technieken die we in OOD gaan leren, gaan we toepassen tijdens het refactoren van bestaande code. De technieken zorgen ervoor dat onze code steeds verbeterd, waardoor **code rot** vermeden wordt. Code rot, of spaghetticode, is code dat na vele aanpassingen eigenlijk niet meer intuïtief is. Hierdoor wordt nieuwe functionaliteit toevoegen complexer dan het eigenlijk zou mogen zijn.

2.3 DRY

Don't Repeat Yourself. We willen onze code zo min mogelijk zichzelf laten herhalen. Als we code schrijven, zullen we zo veel mogelijk proberen onszelf niet te herhalen. We zullen de volledige kracht van OO gebruiken om onze code te groeperen, om zo de herbruikbaarheid van onze code te maximaliseren.

We nemen als voorbeeld één van de eerste oefeningen van C# basis:

Opdracht 1

Maak een programma dat een naamkaart zal maken op basis van de ingegeven waarden. Het programma zal de naam, adresgegevens en design karakter aan de gebruiker vragen om daarna het volgende naamkaartje als output voorzien:



```
C:\Program Files\dotnet\dotnet.exe
Gelieve een design karakter te kiezen (vb *, #, //...)
#
Gelieve je coördinaten in te geven:
Voornaam: John
Familienaam: Doe
Adres: Geldenaaksebaan 335
Postcode: 3001
Plaats: Heverlee

###
#      Voornaam      : John
#      Familienaam   : Doe
#      Adres         : Geldenaaksebaan 335
#      Postcode      : 3001
#      Plaats        : Heverlee
####
```

Een mogelijke oplossing zou er zo kunnen uitzien:

```
Console.WriteLine("Gelieve een design karakter te kiezen (vb. *, #, //, ... );");
string designKarakter = Console.ReadLine();

Console.WriteLine("Gelieve je coördinaten in te geven:");
Console.Write("Voornaam: ");
string voornaam = Console.ReadLine();
Console.Write("Familienaam: ");
string familienaam = Console.ReadLine();
Console.Write("Adres: ");
string adres = Console.ReadLine();
Console.Write("Postcode: ");
string postcode = Console.ReadLine();
Console.Write("Plaats: ");
string plaats = Console.ReadLine();
```

```

Console.WriteLine(designKarakter + designKarakter + designKarakter);
Console.WriteLine(designKarakter + "\tVoornaam\t: " + voornaam);
Console.WriteLine(designKarakter + "\tFamiliennaam\t: " + familiennaam);
Console.WriteLine(designKarakter + "\tAdres \t: " + adres);
Console.WriteLine(designKarakter + "\tPostcode\t: " + postcode);
Console.WriteLine(designKarakter + "\tPlaats \t: " + plaats);
Console.WriteLine(designKarakter + designKarakter + designKarakter);

```

We zien echter dat we redelijk wat code logica herhalen. De input kunnen we moeilijk veralgemenen, gezien we daar vast zitten met onze variabelen. Onze output kunnen we wel centraliseren door een nieuwe methode `PrettyPrint` toe te voegen, die 3 parameters combineert:

- Het designcharacter
- De naam van het te printen veld (bv. Voornaam)
- De waarde van het te printen veld (bv. de voornaam variabele)

Ook voor de start- en eindmarkering zouden we een methode kunnen maken, die we dan 2x aanroepen.

Als we nu de inspringen wensen te vergroten of verkleinen, of een extra design karakter toe te voegen, moeten we dit slecht op één plek doen. Als we onze separator wensen aan te passen, zit dit ook centraal op één locatie. Onze code is op eerste zicht complexer dan onze eerste poging, maar flexibeler om aan te passen.

Deze twee methodes zouden er dan als volgt kunnen uitzien:

```

static void PrettyPrint(string designCharacter, string name, string
value)
{
    Console.WriteLine($"{designCharacter}\t{name.PadRight(15)}:
\t{value}");
}

static void PrintSeparator(string designCharacter)
{
    Console.WriteLine(designCharacter + designCharacter +

```

```
        designCharacter);  
    }
```

2.4 KISS

Keep It Simple Stupid. Een nogal directe afkorting, die de ontwikkelaar er probeert aan te herinneren dat de simpelste oplossing vaak de beste is. Als ontwikkelaar heb je soms de reflex om zaken te overdenken, of dingen complexer te zien als ze werkelijk zijn.


Het duidt ook op de manier van communiceren. Als ontwikkelaar moet je soms met eindgebruikers in dialoog gaan. Zij hebben vaak geen kennis van specifieke technologische oplossingen. Het is dus belangrijk om je voorgestelde oplossing duidelijk te presenteren, zonder jezelf te verliezen in technisch jargon.

KISS betekent echter niet dat korter altijd beter is. We nemen volgende lijn code als voorbeeld:

```
string result = isStarted ? isFinished ? hasError ? "Error detected" :  
"Run completed without errors" : "Not finished yet" : "Not started yet";  
Console.WriteLine(result);
```

Deze code maakt gebruik van de verkorte notatie van het if-statement. Deze notatie is uitstekend voor simpele vergelijkingen, maar dreigt complex te worden bij meerdere controles. Soms is het dan beter om toch de if-statements uit te schrijven, zodat de code eenvoudig te lezen is:

```
if (isStarted)  
{  
    if (isFinished)  
    {  
        result = hasError ? "Error detected" : "Run completed without  
errors";  
    }  
    else  
    {  
        result = "Not finished yet";  
    }  
}
```



```
else
{
    result = "Not started yet";
}
```

Stel dat je later aanpassingen moet doen aan de logica (bijvoorbeeld, je gaat als `isStarted` `true` is, ook rekening houden met de startdatum), dan zal het eerste voorbeeld code veel meer wijzigingen vereisen dan het tweede voorbeeld. Simpel is niet altijd korter.

Veel hangt natuurlijk ook af van het doelpubliek. Senior ontwikkelaars gaan bepaalde technieken die juniors complex vinden juist zien als iets makkelijk, zij gaan dus complexe code maken. Denk er steeds aan dat er ook mensen na jouw komen voor aan je code te werken.

2.5 SOLID

SOLID is een acroniem die de eerste 5 principes van object georiënteerd design beschrijven. Je kan zeggen dat dit de spelregels zijn om goede object georiënteerde code te schrijven. De regels zijn voor het eerst beschreven door Robert C. Martin (ook bekend als Uncle Bob) in zijn paper *Design Principles and Design Patterns* uit 2000.

2.5.1 Single Responsibility Principle (SRP)

Elke class heeft één verantwoordelijkheid. De class mag enkel rekening houden met dat ene afgezonderde deel van de logica. Deze logica mag nergens anders herhaald worden dan in de ene class.

Meer gedetailleerd, elke methode binnen die class heeft zijn eigen verantwoordelijkheid. Je gaat geen twee methodes schrijven voor bijvoorbeeld dezelfde validatie af te toetsen.



Zie **Oefeningenboek** - 2.2 Single Responsibility Principle (SRP)

2.5.2 Open/Closed Principle

Code moet *open* zijn voor uitbreiding, maar *gesloten* zijn voor aanpassingen. Dit wil zeggen dat wanneer je een toevoeging moet doen aan hoe je programma werkt, je dit idealiteit gewoon kan toevoegen in de code, zonder eerst bestaande code te moeten herschrijven.

Natuurlijk, we kunnen niet zomaar elke aanpassing flexibel voorzien. Soms gaan aanpassingen ervoor zorgen dat je structuur niet meer klopt. Hiervoor zullen we dan grondig moeten refactoren om ook deze wijzingen te implementeren.

Het Open/Closed principle is ook een gevaarlijk principe, in de zin dat het een ontwikkelaar kan aanmoedigen om de code nodeloos complex te maken.



Zie **Oefeningenboek** - 2.3 Open/Closed Principle

2.5.3 Liskov's Substitution Principle (LSP)

Het Liskov's Substitution Principle stelt dat we een superclass object referentie vervangen door een object van één van zijn child classes, het programma niet mag breken. Dit is één van de basisregels van *polymorfisme*. We moeten ervoor zorgen wanneer we parent class vervangen gaan vervangen door een child class, onze code correct blijft werken.

Stel, we hebben volgende class:

```
internal class Zoo
{
    public void NurseMammal(Mammal mammal)
    {
        var babyMammal = mammal.GivesBirth();
        if(babyMammal != null)
        {
            //code to nurse the baby mammal
        }
    }
}
```

Deze functie zou dus voor een Mammal object moeten werken, maar ook voor child classes van Mammal. Als we echter een Platypus class toevoegen krijgen we problemen. Een platypus is een zoogdier, maar legt eieren. Een ei is geen Mammal, dus de logica breekt hier. We weten niet hoe we een ei moeten omzetten naar babyMammal. Een Platypus laten overerven van Mammal zou onze GivesBirth methode doen breken, en we zouden er mee het LSP overtreden. In plaats van overerving zouden we dan beter werken met een implementatie via interfaces.

LSP dwingt dus af dat de logica tussen overervende classes correct zit.

2.5.4 Interface Segregation Principle (ISP)

ISP stelt dat objecten die een interface implementeren, niet gedwongen mogen worden om methodes te implementeren die ze niet gebruiken.

Stel, we hebben een interface die een service beschrijft dat toelaat om verschillende dranken te bestellen:

```
internal interface IDrinkService
{
    void OrderWine(int quantity);
    void OrderSoda(int quantity);
    void OrderWater(int quantity);
    void OrderBeer(int quantity);
}
```

Als we echter een service toevoegen voor een wijn bar (waar natuurlijk ook water te verkrijgen is), moeten we 2 methodes implementeren die we niet gebruiken:

```
internal class WineService : IDrinkService
{
    public void OrderBeer(int quantity)
    {
        throw new NotImplementedException();
    }

    public void OrderSoda(int quantity)
    {
        throw new NotImplementedException();
    }

    public void OrderWater(int quantity)
    {
        //code to order water
    }

    public void OrderWine(int quantity)
    {
        //code to order wine
    }
}
```

```
}  
}
```

Deze manier van werken zorgt voor extra methodes in een class die we niet gaan gebruiken. Dit leidt tot onduidelijke code, die complexer is dan wat er eigenlijk gebruikt wordt. Interfaces moeten ervoor zorgen dat enkel de nodige methodes geïmplementeerd moeten worden. Om dit op te lossen kunnen we verschillende interfaces aanmaken, en deze dan apart implementeren:

```
internal class WineServiceGood : IWaterService, IWineService  
{  
    public void OrderWater(int quantity)  
    {  
        //code to order water  
    }  
  
    public void OrderWine(int quantity)  
    {  
        //code to order wine  
    }  
}
```

We zorgen er dus voor dat onze interfaces enkel het noodzakelijk bevatten, waardoor we bij implementatie geen “overschot” aan methodes hebben.

We moeten er wel op letten dat we dit niet tot het extreme gaan doortrekken. Dit zou betekenen dat we altijd slechts één methode per interface gaan definiëren. In praktijk zijn bepaalde methodes wel gelinkt, dus deze mogen dan wel in dezelfde interface.



Zie **Oefeningenboek** - 2.4 Interface Segregation Principle (ISP)

2.5.5 Dependency Inversion Principle (DIP)

DIP stelt dat een class idealiter niet op een implementatie steunt, maar op de abstractie ervan. Deze abstractie kan zowel komen van overerving van classes, of van implementatie van interfaces. Van een abstract iets kan er dus geen object gemaakt worden.

Stel je voor dat we een portaal maken, waarin gebruikers verschillende games kunnen starten.

```
internal class GamePortal
{
    public Tetris Tetris { get; set; }
    public PacMan PacMan { get; set; }
}
```

Deze portal gebruikt de twee classes die onze games zelf beschrijven:

```
public class Tetris
{
    public void StartTetris() { }
}

public class PacMan
{
    public void StartPacMan() { }
}
```

De games hebben een start functie. In onze Program.cs kunnen dan bijvoorbeeld zulke code schrijven:

```
Console.WriteLine("Pick a game:");
Console.WriteLine("1. Tetris");
Console.WriteLine("2. Pac-Man");
var gameId = Console.ReadLine();
GamePortalBad gamePortal;
switch (gameId)
{
    case "1":
        gamePortal = new GamePortal();
        gamePortal.Tetris = new Tetris();
}
```

```

        gamePortal.Tetris.StartTetris();
        break;
    case "2":
        gamePortal = new GamePortal();
        gamePortal.PacMan = new PacMan();
        gamePortal.PacMan.StartPacMan();
        break;
    default:
        break;
}

```

Als we een nieuwe game wensen toe te voegen, moeten we eerst een nieuwe class aanmaken die deze game beschrijft, maar is er ook een aanpassing nodig aan het portaal. Via interfacing en dependency injection kunnen we dit vermijden, waardoor we geen aanpassing meer nodig hebben in onze GamePortal.

De eerste stap is een interface aan te maken die het gedrag van de games beschrijft:

```

internal interface IGame
{
    void StartGame();
}

```

Onze game classes zullen dan deze interface *implementeren*. De StartGame methode vervangt de bestaande specifieke start methode:

```

public class Tetris : IGame
{
    public void StartGame()
    {
        // start Tetris
    }

    //public void StartTetris() { }
}

public class PacMan : IGame

```

```

{
    public void StartGame()
    {
        //starts PacMan
    }

    //public void StartPacMan() { }
}

```

Ons nieuw portaal kan er dan zo uitzien:

```

internal class GamePortal
{
    public IGame Game { get; set; }
}

```

In Program.cs kunnen we dan:

```

GamePortal portal = new GamePortal();
switch (gameId)
{
    case "1":
        portal.Game = new Tetris();
        break;
    case "2":
        portal.Game = new PacMan();
        break;
    default:
        Console.WriteLine("Game not found");
        break;
}
portal.Game.StartGame();

```

We hebben eigenlijk het juiste type game *geïnjecteerd* in onze GamePortal, door gebruik te maken van een interface. Als we nu een nieuw game aanmaken, dat ook overerft van IGame, moeten we de code van GamePortal niet aanpassen. Dit zal er ook voor zorgen dat GamePortal *open* is voor uitbreiding, maar *gesloten* is voor aanpassingen.

We gaan hier dieper op in tijdens de les over dit design pattern.



Zie **Oefeningenboek** - 2.5 Dependency Inversion Principle (DIP)

2.6 Conclusie

Door de 5 principes van OOD te hanteren, kunnen we onze code beschermen tegen code die per aanpassing moeilijker wordt om te onderhouden. Vaak is dit niet van de eerste versie van onze code, maar volgt dit na uitbreiding van functionaliteit en door genoeg te refactoreren.

De bedoeling is dat de verschillende classes die we maken zo los mogelijk van elkaar werken. Zo kunnen aanpassingen of uitbreidingen gebeuren, zonder dat we steeds bestaande code moeten aanpassen. Dit zorgt voor robuustere code, en veiliger (minder kans op bugs) programmeren.

3 Design Patterns

Design Patterns zijn blauwdrukken van hoe je code schrijft. Waar de vorige hoofdstukken vooral algemene richtlijnen zijn, dienen design patterns expliciet om bepaalde problemen gestructureerd op te lossen.

Waar een klasse de blauwdruk is van een object, kan je een design pattern zien als een blauwdruk van een verzameling van classes, die op een specifieke manier samenwerking om een specifiek probleem op te lossen. Deze blauwdruk optimaliseert de performantie van de code, alsook de kwaliteit. Design patterns zijn gegroeid doorheen de tijd, en hebben hierdoor hun minpunten kunnen verbeteren. Door gebruik te maken van design patterns gaan wij eigenlijk gebruik maken van historisch bewezen ideeën.

Design patterns combineren theorie met een uitwerking in praktijk. Als programmeurs weten welke verschillende classes er nodig zijn om tot een resultaat te komen, is het ook makkelijker om samen te werken. Voor bijvoorbeeld het Model-View-Controller design pattern (gezien bij C# gevorderd) weten we dat er 3 grote onderdelen zijn. Je zou dus het werk kunnen opsplitsen, en toch nog goed kunnen samenwerken, omdat iedereen weet wat de verwachtingen zijn.

We kunnen design patterns opdelen in 3 grote groepen:

- Creational
- Structural
- Behavioral

3.1 Soorten Design Patterns

3.1.1 Creational Design Pattern

Worden gebruikt om classes te instantiëren. Creational design patterns gaan ons dus helpen met objecten te maken. Deze objecten kunnen dan volgens een geordende, herbruikbare en flexibele manier aangemaakt worden. Voorbeelden van creational design patterns zijn:

- Builder
- Abstract Factory
- Singleton

3.1.2 Structural Design Pattern

Structural design patterns zijn patterns die objecten gaan samenstellen op een of andere manier. Ze gaan met andere woorden specifieke relaties tussen objecten gaan definiëren. Zo gaat het Adapter design pattern ervoor zorgen dat classes die geen gemeenschappelijke interface hebben, toch kunnen samenwerken.

Voorbeelden:

- Adapter
- Composite

3.1.3 Behavioral Design Pattern

Dit zijn patterns die de communicatie tussen objecten beschrijven. Ze zorgen ervoor dat twee objecten op een transparante manier met elkaar kunnen communiceren. Voorbeelden hiervan zijn:

- Command
- Observer
- Mediator

3.2 Architectural patterns

We weten nu dat design patterns gebruikt worden om verschillende classes makkelijk en flexibel te laten samenwerken. Een hoger niveau zou zijn om verschillende design patterns met elkaar te laten samenwerken. Dit kan een architectural pattern zijn. De scope is groter dan bij de design patterns, en kan gebruikt worden om de architectuur van een applicatie te beschrijven. Voorbeelden van deze complexere beschrijvingen zijn:

- Layered
- Broker
- Model View Controller (MVC)
- Peer-to-Peer
- Client-Server

Deze patterns zijn soms ingebouwd in software frameworks (zie ASP.NET MVC)

3.3 Singleton

Het eerste design pattern dat we gaan bekijken in detail is een creational design pattern genaamd Singleton. Het gaat ons dus helpen bij het aanmaken van een instantie van een object. Wat maakt Singleton zo speciaal?

- Er kan maar één instantie van de klasse zijn
- Er is geen overerving mogelijk van deze klasse
- Properties en methodes worden rechtstreeks op de klasse uitgevoerd, i.p.v. het object



De volledige code van onderstaand voorbeeld vind je terug onder de solution **UcIII.OOD.DesignPatterns**, project **Singleton**

Om een class te maken waarvan geen object kan worden aangemaakt en die niet overerfbaar is, hebben we volgende code nodig

```
sealed class OnlyOneDatabase
{
    private OnlyOneDatabase()
    {
    }
}
```

- Sealed: zorgt ervoor dat de class niet overgeërfd kan worden
- Private constructor: zorgt ervoor dat de class niet aangemaakt kan worden

Hoe zorgen we er dan voor dat we één instantie krijgen? We laten de class zelf een instantie aanmaken. Dit doen we door een datamember van hetzelfde type als de class zelf toe te voegen. We schrijven hierna dan een property die enkel een Get doet:

```
private static OnlyOneDatabase instance = null;
public static OnlyOneDatabase Instance
{
    get
    {
        if (instance is null)
```

```
    {  
        instance = new OnlyOneDatabase();  
    }  
    return instance;  
}  
}
```

De eerste keer dat onze instance datamember wordt aangesproken, gaat die ook aangemaakt worden. Vanaf dan werken we altijd met hetzelfde object. Properties die we toevoegen zullen dan slechts in één object gedigitaliseerd kunnen worden:

Bijvoorbeeld, we voegen volgende property toe aan onze OnlyOneDatabase class:

```
public int TestValue { get; set; }
```

Als we hierna dan in Program.cs volgende code schrijven:

```
var instance = OnlyOneDatabase.Instance;  
instance.TestValue = 5;  
  
var instance2 = OnlyOneDatabase.Instance;  
Console.WriteLine(instance2.TestValue);
```

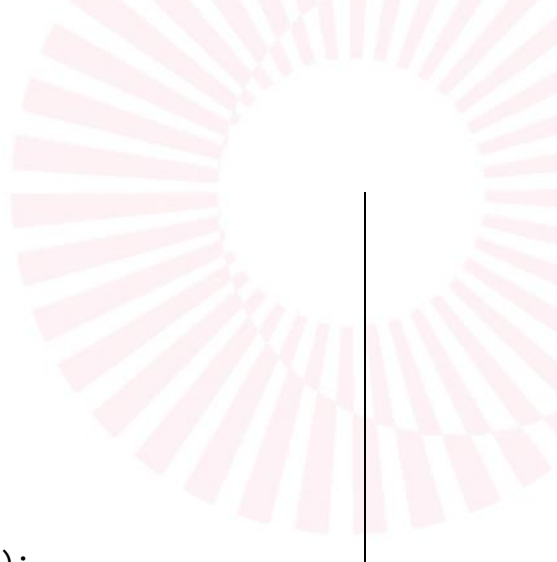
Zullen we zien dat de waarde van de TestValue voor variabele instance als instance2 dezelfde is, namelijk 5.

Bij deze code is er echter een probleem bij multi-threading. Als we meerdere threads hebben, dan zouden we op elke thread een instantie gelijktijdig kunnen aanmaken. Dit is natuurlijk niet de bedoeling. Het pattern is dus uitgebreid om met deze complexiteit om te gaan. We zullen hiervoor het lock keyword moeten gebruiken.

Het lock keyword lockt een object en laat weer los wanneer de scope verlaten is. Als de lock niet mogelijk is (omdat het object al gelocked is), dan zal de uitvoering van de volgende statements niet gebeuren. Het grote nadeel hierbij is dat de lock check telkens gebeurt wanneer we een instantie aanvragen, en dit is nadelig voor de performantie.

De nieuwe code met het lock keyword ziet er dan zo uit:

```
private static readonly object padlock = new object();  
  
public static OnlyOneDatabase Instance
```



```

{
    get
    {
        lock (padlock)
        {
            if (instance is null)
            {
                instance = new OnlyOneDatabase();
            }
            return instance;
        }
    }
}

```

We maken dus een nieuwe datamember die zal functioneren als lock object. Via het lock keyword gaan we dan kijken of we een lock kunnen zetten op dit object. Als dit lukt, zal de volgende code uitgevoerd worden (de initialisatie van onze OnlyOneDatabase class). Als dit niet lukt, gebeurt de initialisatie niet.

Om te vermijden dat we de lock bij elke aanspreken van de instantie moeten controleren, kunnen we een extra controle toevoegen:

```

public static OnlyOneDatabase Instance
{
    get
    {
        if (instance is null)
        {
            lock (padlock)
            {
                if (instance is null)
                {
                    instance = new OnlyOneDatabase();
                }
            }
        }
        return instance;
    }
}

```

```
}  
}
```

We gaan dus de lock enkel uitvoeren als ons instance object nog niet geïnitieerd is. Dit is dan ook de finale versie van ons design pattern.

3.4 Inversion of Control (IoC)

Inversion of Control is een overkoepelende benaming voor design patterns die de controle bij de uitvoerende class gaan leggen. Een client class gaat gebruik maken van de service class. De service class gaat echter geen controle hebben over hoe de client class deze service gebruikt.

We gaan hiervoor twee design patterns in detail bekijken:

- Dependency Injection
- Abstract factory

3.4.1 Dependency Injection (DI)


De implementatie van Dependency Injection bestaat minstens uit drie classes:

- De **client** class
Dit is de class die gebruikt gaat maken van de functionaliteit van een andere class
- De **service** class
Dit is de class die de functionaliteit gaat aanbieden die een andere class nodig heeft
- De **injector** class
Dit is de class die verantwoordelijk is voor de injection van de service class in de client class. Het is deze class dus dat de koppeling gaat leggen tussen de client en de service class

Omdat Dependency Injection een van de meest gebruikte implementatie is van IoC worden deze twee vaak als synoniemen gebruikt. We gaan namelijk de controle niet langer bij de client class leggen, maar bij de aanroepende code. De aanroepende code wordt dan verantwoordelijk voor de service class die de client class zal gebruiken.

Een voorbeeld zou het volgende kunnen zijn:

```
public class Spaceship  
{
```



```
public Spaceship() { }

public void Fly()
{
    FuelEngine engine = new FuelEngine(1000);
    engine.Throttle(950);
}
}
```

Onze Spaceship class gebruikt een FuelEngine object, die ook de afstand meekrijgt dat we gaan afleggen. Spaceship is dus onze client class. FuelEngine onze service class. Op dit moment is er nog geen injector class. Aanpassingen aan de constructor van FuelEngine zullen ervoor zorgen dat we in deze werkwijze ook de Spaceship class moeten aanpassen. We noemen in dit geval Spaceship en WarpEngine *tightly coupled*.

Om de twee classes *loosly coupled* te maken, kunnen we het FuelEngine object meegeven aan de class Spaceship (= injecteren). Dit kunnen op 3 manieren: via de **constructor**, via een **property** of via de **methode**.

3.4.1.1 Dependency Injection via Constructor

```
public class Spaceship
{
    private FuelEngine _fuelEngine;

    public Spaceship(FuelEngine fuelEngine)
    {
        _fuelEngine = fuelEngine;
    }

    public void Fly()
    {
        _fuelEngine.Throttle(950);
    }
}
```

We geven het FuelEngine object mee aan de constructor van ons Spaceship en slagen dit op in een data member van de Spaceship class. In de methode Fly gaan we dan ons data member gebruiken om onze Throttle methode aan te roepen. In de code waar we onze Spaceship dan gaan declareren, moeten we dan ook eerst een FuelEngine object aanmaken. Dit is dan de injector class (bv. Program.cs):

```
FuelEngine engine = new FuelEngine(1000);  
Spaceship ship2 = new Spaceship(engine);
```

3.4.1.2 Dependency Injection via Property

In plaats van de constructor te gebruiken, kunnen we een property aanmaken in onze Spaceship class. Deze property kunnen dan opnieuw laten opvullen vanuit de aanroepende code. Ons Spaceship zal er dan zo uitzien:

```
public class Spaceship  
{  
    public FuelEngine FuelEngine { get; set; }  
  
    public void Fly()  
    {  
        FuelEngine.Throttle(950);  
    }  
}
```

Als de property echter niet opgevuld is op het moment dat we de Fly methode aanroepen, zal deze crashen. We zullen er dus voor moeten zorgen dat dit opgevangen is door een controle te doen op null:

```
public void Fly()  
{  
    if(instance is null)  
    {  
        FuelEngine.Throttle(950);  
    }  
}
```


Door gebruik te maken van de property, kunnen we het FuelEngine object later vanuit de aanroepende code nog veranderen. Dit kunnen we niet als we de service class injecteren via de constructor injectie.

```
ship.FuelEngine = new FuelEngine(1200);  
ship.Fly();
```

3.4.1.3 Dependency Injection via Method Injection

De laatste manier is ons object meegeven aan de methode zelf. Dit ziet er dan als volgt uit:

```
public class Spaceship  
{  
    public void Fly(FuelEngine fuelEngine)  
    {  
        fuelEngine.Throttle(950);  
    }  
}
```

Dit betekent dat we bij elke call van de Fly methode, een FuelEngine object zullen moeten meegeven:

```
Spaceship ship = new Spaceship();  
ship.Fly(new FuelEngine(1000));  
ship.Fly(new FuelEngine(1200));
```

Natuurlijk kunnen we een FuelEngine object éénmaal aanmaken in een lokale variabele en dit zo steeds meegeven aan onze Fly methode:

```
FuelEngine reusableEngine = new FuelEngine(800);  
ship.Fly(reusableEngine);  
ship.Fly(reusableEngine);
```

3.4.1.4 Dependency Injection via Interfacing

Technologie staat echter niet stil, en de ontwerpers van ons Spaceship hebben een nieuwe engine uitgevonden: een WarpEngine. Als we een WarpEngine in plaats van een FuelEngine wensen te gebruiken, zien we dat onze code terug *tightly coupled* wordt: we kunnen namelijk onze soort engine niet veranderen zonder aanpassingen te doen in onze Spaceship class.

We zullen dus een manier moeten vinden om ook het concrete **type** te kunnen **injecteren**.

Gelukkig kennen we hiervoor al een oplossing: *polymorfisme*. We kunnen een *interface* aanmaken die het engine gedrag gaat beschrijven, en dan het juiste type van de engine via Dependency Injection (**constructor**, **property** of **method**) mee te geven.

```
internal interface IEngine
{
    public void Throttle(int speed);
}
```

Onze engines gaan dan deze interface *implementeren*:

```
public class FuelEngine : IEngine
public class WarpEngine : IEngine
```

As we dan ons voorbeeld voor Dependency Injection via de constructor aanpassen, ziet deze er zo uit:

```
public class Spaceship
{
    private IEngine _engine;

    public Spaceship(IEngine engine) => _engine = engine;

    public void Fly()
    {
        _engine.Throttle(950);
    }
}
```

We krijgen onze engine dus niet meer als een vast type binnen, maar als een contract (interfacing). Dit betekent dat we aan onze constructor elk object kunnen meegeven dat de IEngine interface implementeert:

```
var fuelEngine = new FuelEngine(1000);  
var warpEngine = new WarpEngine();  
  
var fuelShip = new Spaceship(fuelEngine);  
var warpShip = new Spaceship(warpEngine);
```

We hebben onze code terug *loosly coupled* gemaakt. De class Spaceship is open voor uitbreiding, maar gesloten voor aanpassingen. Als we een nieuwe engine wensen toe te voegen, volstaat het om deze te laten implementeren van de IEngine interface, waardoor er aan de Spaceship class geen code wijzigingen gedaan moeten worden.



Volledige code vind je terug onder solution **Ucll.OOD.DI-Spaceship**, project **DI_Spaceship**



Onder de **Media Gallery** is er een video **OOD - Dependency Injection – Spaceship** die dit voorbeeld doorloopt

3.4.2 IoC Container

Een Inversion of Control Container is een framework waardoor we Dependency Injection makkelijk kunnen implementeren. Het framework heeft 3 verantwoordelijkheden:

- Creatie van objecten
- Beheer life time van objecten
- Injectie van objecten in klassen

De IoC container biedt hiervoor 3 diensten aan:

- Register
- Resolve

- Dispose

De .NET community heeft verschillende frameworks tot hun dienst:

- StructureMap
- AutoFac
- CastleWindsor
- Unity
- Microsoft DependencyInjection library

3.4.2.1 Praktische uitwerking



Volledige code vind je terug onder solution **Uccl.OOD.DesignPatterns**, project **DependencyInjection**



Onder de **Media Gallery** is er een video **OOD – IoC Container** die dit voorbeeld doorloopt

We gaan een console applicatie maken die een databank gebruikt. In deze databank gaan we films bewaren. We hebben dus 3 classes:

- Movie class met enkele properties
- MovieRepository class, wat onze databank zal zijn
- MovieController class, die acties toelaat en met de databank communiceert

Gezien MovieController gebruik gaat maken van MovieRepository, zullen we moeten zorgen dat deze *loosly coupled* zijn. Dit kunnen we doen door een **interface** te maken die het gedrag van MovieRepository beschrijft, en deze via **constructor dependency injection** toe te voegen aan onze MovieController class:

```
private IMovieRepository _movieRepository;

public MovieController(IMovieRepository movieRepository)
{
    _movieRepository = movieRepository;
}
```

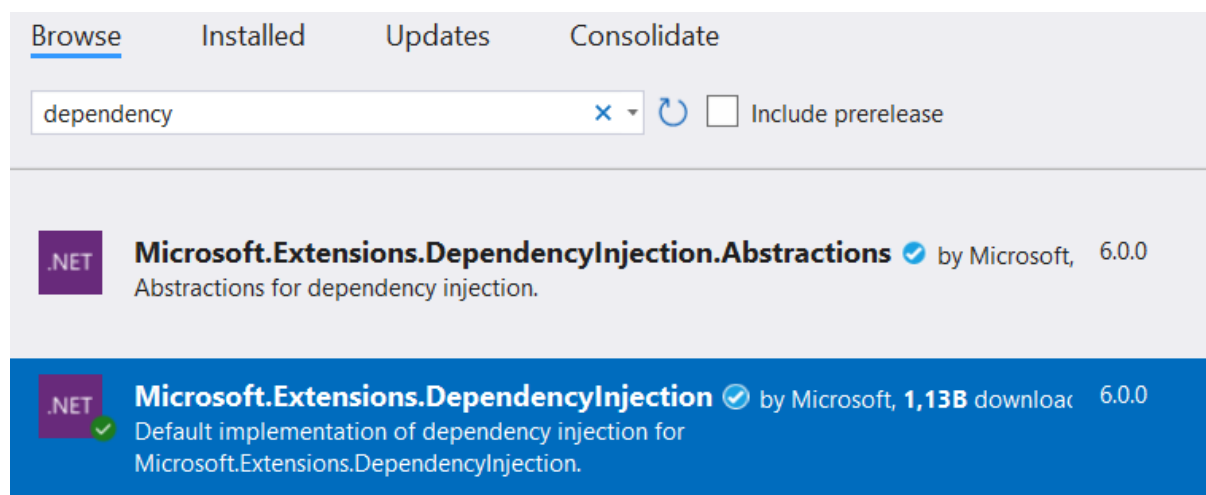
```
}
```

Bij het aanroepen van onze code ziet het er dan zo uit:

```
var movieRepository = new MovieRepository();  
var movieController = new MovieController(movieRepository);  
Console.WriteLine(movieController.GetMovie(1));
```

We wensen echter nu gebruik te maken van de **Microsoft DependencyInjection** library. Dit zodat we de injectie van MovieRepository kunnen centraliseren. Wanneer we websites gaan maken in ASP.NET Core MVC gaan we dezelfde werkwijze zien.

Onze eerste stap is de **NuGet package** installeren van de Microsoft DependencyInjection:



```
PM> Install-Package Microsoft.Extensions.DependencyInjection
```

De aanpassingen die we gaan doen, gaan in Program.cs gebeuren. Hiervoor moeten we eerst Program.cs herschrijven naar zijn lange notatie:

```
class Program  
{  
    static void Main(string[] args)  
    {  
        var movieRepository = new MovieRepository();  
        var movieController = new MovieController(movieRepository);  
        Console.WriteLine(movieController.GetMovie(1));  
    }  
}
```

```
}
```

We willen dus twee zaken doen: we willen een service `MovieController` registreren, die gebruikt zal maken van de service `MovieRepository`. Hiervoor moeten we dus deze twee services gaan registreren in onze IoC container:

```
private static void RegisterServices()
{
    var services = new ServiceCollection();
    services.AddSingleton<IMovieRepository, MovieRepository>();
    services.AddSingleton<MovieController>();
    _serviceProvider = services.BuildServiceProvider(true);
}
```

We gaan beginnen met een `ServiceCollection` object aan te maken en hierin onze twee services toe te voegen. We gaan dit als Singleton toevoegen, wat betekent dat er maar één instantie van deze services gemaakt zal worden. Onze `MovieController` verwacht een `IMovieRepository` object. We kunnen echter van interfaces geen object maken, dus moeten we tegen onze service zeggen welke *implementatie* we wensen mee te geven aan deze service. In dit geval is dit de `MovieRepository`.

Wanneer we beide services toegevoegd hebben kunnen we hiervan een serviceprovider maken via de methode `BuildServiceProvider`.

Wanneer we onze services niet meer nodig hebben kunnen we deze disposes. Dit wil zeggen dat we eventuele referenties naar onze objecten gaan laten opkuisen. Dit doen we via onderstaande methode:

```
private static void DisposeServices()
{
    if (_serviceProvider == null)
    {
        return;
    }
    if (_serviceProvider is IDisposable)
```

```
    {  
        ((IDisposable)_serviceProvider).Dispose();  
    }  
}
```

Onze nieuwe main wordt dan:

```
static void Main(string[] args)  
{  
    RegisterServices();  
    var movieController =  
        _serviceProvider.GetRequiredService<MovieController>();  
    Console.WriteLine(movieController.GetMovie(1));  
    DisposeServices();  
}
```

We vragen aan onze serviceprovider om ons de MovieController service te geven. Onze serviceprovider bevat de nodige services om hiervan een object te maken, gezien we de IMovieRepository erin hebben geregistreerd. We hebben de serviceprovider ook gezegd dat wanneer hij een IMovieRepository tegenkomt, hij hier een MovieRepository object van moeten maken.

Het resultaat zal dus een MovieController object zijn, dat via dependency injection een MovieRepository heeft meegekregen.

Stel, we gaan een nieuwe versie van onze MovieRepository aanmaken, MovieRepository2, dan moeten dit slechts op één locatie in de code aanpassen, namelijk bij de service registratie:

```
services.AddSingleton<IMovieRepository, MovieRepository2>();
```

Hierna zijn al de verwijzingen naar IMovieRepository die via injection aan services toegewezen worden, omgezet naar een instantie van MovieRepository2.

We kunnen services op 3 manieren registreren:

- AddSingleton - zelfde object bij elke request (applicatie niveau)
- AddScoped - één object per request
- AddTransient - maakt een nieuw object aan voor elke request



Onder de solution **DependencyInjectionSample** vind je geheel informatief een uitgewerkt voorbeeld dat deze drie manieren van registreren aantoont.

3.4.3 Abstract Factory

Ook Abstract Factory is een creational design pattern. Het gaat ons, net zoals Dependency Injection, helpen om objecten te creëren die *loosly coupled* met elkaar zijn.

Stel, we krijgen de opdracht om een applicatie te maken die verschillende look-and-feels ondersteunt. Onze schermen, scrollbars, knoppen, ... zullen er dus anders uitzien afhankelijk van welk thema we selecteren. We zouden dit kunnen oplossen door, telkens wanneer we een visuele component tonen, hier een switch rond te schrijven. Zo kunnen we dan telkens de juiste component laden. Dit zal er echter voor zorgen dat onze code zeer complex wordt. We willen de logica om de juiste component te kiezen dus centraliseren.

Een Abstract Factory gaat dus een *familie* van gerelateerde classes initialiseren, op de manier dat de eindgebruiker tijdens runtime kan kiezen.

We gaan in een console app dit zeer basis uitwerken. We hebben 2 componenten: een titel en een opsomming. We hebben ook twee layouts, de ster-layout en de dollar-layout. Een voorbeeld van een ster-layout voor een titel kan dan de volgende class zijn:

```
public class StarTitle
{
    public void PrintTitle(string title)
    {
        Console.WriteLine($"*** {title} ***");
    }
}
```


Het equivalent in de dollar-layout is dan:

```
public class DollarTitle
{
    public void PrintTitle(string title)
    {
        Console.WriteLine($"$$$ {title} $$$");
    }
}
```

Gezien we niet met hun concrete implementatie zullen werken, zullen we eerst een abstract class aanmaken die boven onze concrete implementatie zit:

```
public interface ITitleElement
{
    public void PrintTitle(string title);
}

public class StarTitle : ITitleElement
{
    public void PrintTitle(string title)
    {
        Console.WriteLine($"*** {title} ***");
    }
}
```

Nu we onze componenten hebben, kunnen we een factory gaan maken. De factory zal verantwoordelijk zijn om objecten te maken. Voor elke familie (star, dollar) hebben we één factory nodig. Deze zal dan methodes hebben die een concrete implementatie teruggeven:

```
public class StarFactory
{
    public StarTitle CreateTitle()
    {
        return new StarTitle();
    }
}
```

```
public StarList CreateList()
{
    return new StarList();
}
```

Omdat er van een factory slechts één instantie van mag zijn, maken we van onze factories ook Singletons.

In onze Program.cs kunnen we nu dan zoiets doen:

```
void ShowTitle()
{
    Console.WriteLine("Enter the title:");
    Console.Write(">");
    var title = Console.ReadLine();
    TitleElement startTitle = StarFactory.Instance.CreateTitle();
    startTitle.PrintTitle(title);
}
```

We willen echter ook kunnen wisselen van factory tijdens de runtime. Hiervoor zullen we een nieuwe abstracte class aanmaken, waarvan onze concrete factories zullen overerven. Onze twee factories zullen dus niet langer hun concrete implementatie teruggeven, maar we gaan werken met de abstracte componenten:

```
public interface IAbstractFactory
{
    public ITitleElement CreateTitle();
    public IListElement CreateList();
}
```

In onze StarFactory worden de methodes dan:

```
public override ITitleElement CreateTitle()
{
    return new StarTitle();
}
```

```
public override IListElement CreateList()
{
    return new StarList();
}
```

Dankzij **polymorfisme** kunnen we dus een concrete implementatie doorgeven als zijn abstracte vorm. Hierna kunnen we dan in onze Program.cs een AbstractFactory definiëren, dat we initiëren aan de hand van gebruikersinput. Voor elementen aan te maken, gebruiken we dan de call van deze abstract class:

```
void ShowTitle(IAbstractFactory factory)
{
    Console.WriteLine("Enter the title:");
    Console.Write(">");
    var title = Console.ReadLine();
    ITitleElement titleElement = factory.CreateTitle();
    titleElement.PrintTitle(title);
    Console.WriteLine("Press key to continue");
    Console.Read();
}
```

De Abstract Factory bestaat dus uit 5 onderdelen:

- **AbstractFactory** – interface die beschrijft welke abstract producten we kunnen aanmaken
- **ConcreteFactory** - implementatie van de AbstractFactory
- **AbstractProduct** – interface voor een soort product
- **ConcreteProduct** – implementatie van AbstractProduct voor object dat bij een ConcreteFactory hoort
- **Client** – gebruikt enkel de AbstractProduct en AbstractFactory classes



Het volledig uitgewerkt voorbeeld vindt u terug onder solution **Ucll.OOD.Factory**



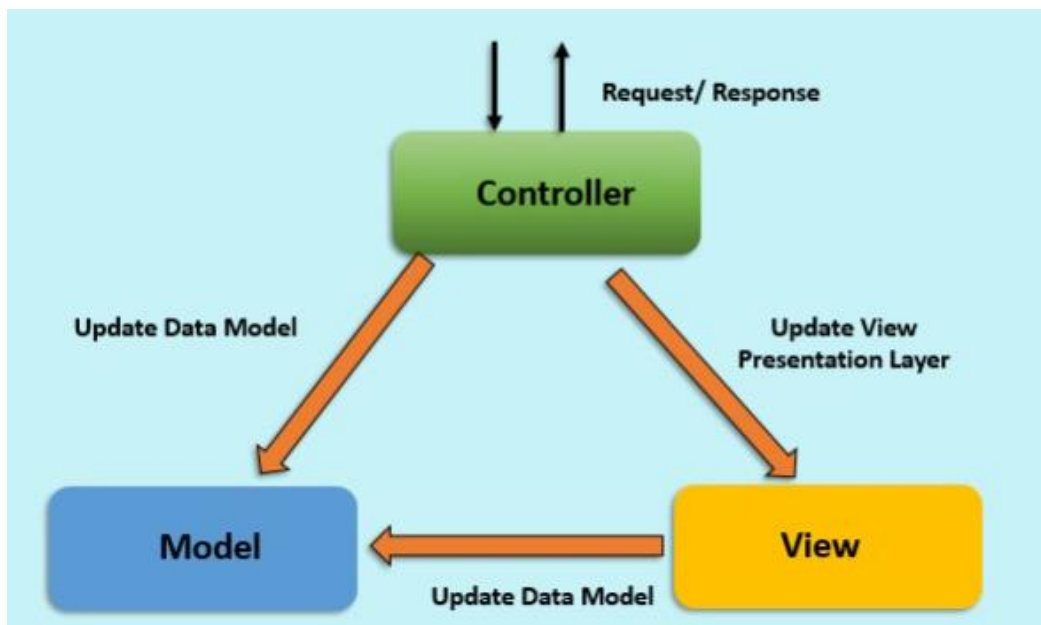
Onder de **Media Gallery** is er een video **OOD – Abstract Factory** die dit voorbeeld doorloopt

4 Model-View-Controller

Model-View-Controller is een architectural pattern. Zoals elke pattern is de manier van werken code onafhankelijk. Je vindt MVC dus terug in bijvoorbeeld Smalltalk, Java Spring, Ruby on Rails en andere OO-talen.

Je kan MVC zien als een pattern om CRUD (create, read, update, delete) acties te visualiseren. Door SRP toe te passen gaan we ervoor zorgen dat elke class zijn eigen verantwoordelijkheid heeft. Hierdoor gaan we een Separation of Concern verkrijgen: elke class weet enkel wat zijn eigen verantwoordelijkheid is, en weet niets af van de andere classes. Binnen MVC zijn er drie verantwoordelijkheden:

- Het model
- De view
- De controller



4.1 Model

Het model beschrijft via classes de datastructuur. Welke classes willen we weergeven, en welke properties hebben deze classes?

Aan welke requirements moeten de waarden die de properties krijgen voldoen? Moeten ze een bepaalde lengte hebben, mogen ze null zijn?

4.2 View

Een view is de weergave waarin ons model getoond wordt. Dit kan console output zijn, een Windows GUI scherm, een html pagina, ...

In een ASP.NET Core MVC is dit een Razor view (.cshtml). Een Razor view combineert de mogelijkheden van een front-end pagina (html, css en scripting talen zoals JavaScript) met de kracht van C#. We zullen dus C# code kunnen schrijven in onze front-end.

Elke Razor view wordt weergave in een algemene container. De standaard container die voor ons wordt aangemaakt is de `_Layout.cshtml` pagina. Eigen gemaakte views (maar ook views zoals `Index.cshtml`) worden in deze view ingeladen via volgende code:

```
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>
</div>
```

We kunnen in een view definiëren welk model we in deze view gaan gebruiken:

```
@model Movie
```

In de Razor view zelf kunnen we dit model (en zijn properties) aanspreken via:

```
@Model.Title
```

Model met een kleine letter: definitie van het model

Model met een grote letter: instantie van het model dat we kunnen aanspreken

4.3 Controller

De controller is de lijm tussen het model en de view. Het is ook in de controller dat requests van user gaan binnenkomen. Denk hierbij aan navigeren naar een pagina, een formulier submitten, Al deze acties lopen steeds via de controller.

In .NET heeft een controller steeds de suffix *Controller*, bv.:

- HomeController

- PeopleController
- CategoriesController

Wanneer je dus een nieuwe controller aanmaakt moet deze eindigen op de *Controller* suffix. Een Controller bestaat uit Actions. Dit zijn de eigenlijke acties die aangeroepen kunnen worden (bv. vanuit een View). Aan deze actions kunnen ook parameters meegegeven worden. In ASP.NET Core geeft een action een IActionResult object terug, dit kan bijvoorbeeld zijn:

- ViewResult
- PartialViewResult
- JsonResult

Een action is een HTTP call. HTTP staat voor HyperText Transfer Protocol en is het protocol voor client-server communicatie (web). De communicatie loopt via een request (aanvraag vanuit de client naar de server) en een response (het antwoord op deze aanvraag, vanuit de server naar client). De controller zal dus een request (de action en zijn parameters) binnenkrijgen, en een antwoord (het IActionResult object) versturen.

Http is een stateless protocol, er is geen relatie tussen opeenvolgende request-response en er is dus geen blijvende client-server connectie. De client start de aanvragen en moet ervoor zorgen dat de request voldoende informatie bevat om deze succesvol door de server te laten verwerken.

4.4 Praktisch: MVC Console App

Voordat we het MVC pattern in ASP.NET gaan implementeren, gaan we dit eens in een standaard console app aanmaken, zodat de basis hiervan duidelijker is.

Usecase:

We wensen een console applicatie waarin we gebruikers kunnen registreren en uitlezen. Een gebruiker bestaat uit een loginnaam en een wachtwoord.

Gezien we via het MVC pattern gaan werken heeft onze console app 3 mappen nodig:

- Controllers
- Views
- Models

4.4.1 Opzetten data

Eerst en vooral moeten een systeem hebben dat onze users kan bewaren. Gezien we geen databank hebben, zullen we kiezen voor een Singleton class, die we gaan gebruiken als persistentie laag. We gaan in deze Singleton class ook een property toevoegen voor onze users te bewaren. We voegen ook twee methodes toe om users op te halen en toe te voegen:

```
public void AddUser(User user)
{
    user.Id = users.Count == 0 ? 1 : users.Max(u => u.Id) + 1;
    users.Add(user);
}

public User? GetUser(int id)
{
    foreach (var user in users)
    {
        if (user.Id == id)
        {
            return user;
        }
    }
    return null;
}
```

We gaan nu het model toe te voegen. We doen dit door een class toe te voegen in de Models folder, genaamd User. Deze class heeft de properties die in de use case beschreven staan en een nieuw property Id (type int).

4.4.2 Opzetten Controller

Onze tweede stap bestaat uit onze UserController aan te maken. Op deze controller gaan we dan twee *actions* definiëren:

- Create
- Details (= weergeven van één gebruiker)

Beide geven een view terug. Omdat wij nu nog geen concrete view hebben, gaan we eerst een interface maken, zodat we een contract hebben voor deze view:

```
public interface IView
{
    void RenderView();
}
```

In de RenderView methode gaan we straks dan onze gegevens naar de console schrijven. Onze Controller ziet er dan als volgt uit:

```
public class UserController
{
    public IView Create()
    {
    }

    public IView Details()
    {
    }
}
```

4.4.3 Opzetten Views

De volgende stap is dan twee views maken. Voor de code structuur maken we onder de map Views eerst een map *Users*, waaronder al onze user gerelateerd views komen. Dit maakt het duidelijker moesten we straks ook andere entiteiten willen gaan modelleren. We maken twee classes: Create en Details, die beide IView implementeren.

In onze Create class gaan we dan een User aanmaken op basis van input van een eindgebruiker:

```
public void RenderView()
{
    User user = new User();
    Console.Write("Enter login: ");
    user.Login = Console.ReadLine();
}
```

```
Console.WriteLine("Enter password: ");
user.Password = Console.ReadLine();
}
```

We zien al direct dat we hierbij een issue gaan hebben. De aangemaakte user kan niet in de view zelf opgeslagen worden (Single Responsibility Principle, een view mag enkel gebruikt worden om zaken weer te geven).

We zullen er dus voor moeten zorgen dat deze User teruggegeven wordt. Onze RenderView moet dus aangepast worden zodat deze een object teruggeeft. In plaats van User, gaan we hier een IModel van maken, zodat we de interface kunnen hergebruiken bij andere entiteiten. Deze interface gaat enkel een property ID hebben (int):

```
public interface IModel
{
    public int Id { get; set; }
}
```

We passen onze RenderView in zowel de interface als de implementatie aan, zodat deze een IModel teruggeeft. In de implementatie geeft dit dan de aangemaakte user terug. Dit kan natuurlijk enkel als onze User class de IModel interface implementeert.

Een Create actie bestaat echter uit twee stappen:

1. Tonen van het toevoegen scherm
2. Verwerking van het toevoegen scherm

Gezien onze controller deze twee bewerkingen doet, hebben een tweede Create action nodig op onze Controller, die dan een object doorkrijgt als parameter:

```
public IView Create(object toCreate)
{
    var user = (User)toCreate;
    UserRepository.Instance.AddUser(user);
    return new Create(user);
}
```

We willen nu een melding tonen dat onze user correct is toegevoegd. Hiervoor gaan we de Create view herschrijven:

```

public class Create : IView
{
    private readonly User _user = null;
    public Create() { }

    public Create(User user)
    {
        _user = user;
    }
    public IModel RenderView()
    {
        if (_user is null)
        {
            User user = new User();
            Console.Write("Enter login: ");
            user.Login = Console.ReadLine();

            Console.Write("Enter password: ");
            user.Password = Console.ReadLine();

            return user;
        }
        else
        {
            Console.WriteLine($"User {_user.Login} added");
            return _user;
        }
    }
}

```

Door volgende code te schrijven in Program.cs kunnen we onze vooruitgang al even testen:

```

UserController userController = new UserController();
var createView = userController.Create();
var model = createView.RenderView();
createView = userController.Create(model);
createView.RenderView();

```

4.4.4 Afwerking

Voor de details toe te voegen, moeten we een manier hebben om te weten welke user we willen weergeven. We hebben hiervoor het Id. We geven dus aan onze controller de Id van de User mee. Deze zal dan in onze repository de bijhorende user ophalen:

```
public IView Details(int id)
{
    var user = UserRepository.Instance.GetUser(id);
    return new Details(user);
}
```

We kunnen ook een List action toevoegen, dat dan alle gebruikers weergeeft:

```
public IView List()
{
    return new List(UserRepository.Instance.GetUsers());
}
```

Deze kunnen we dan in Program.cs gebruiken om de gebruiker te kiezen:

```
var listView = userController.List();
listView.RenderView();

Console.WriteLine("Pick a user:");
var userId = int.Parse(Console.ReadLine());
var detailView = userController.Details(userId);
detailView.RenderView();
```

Als we meerdere controllers willen toevoegen, is het best dat we hiervoor ook een interface voorzien. Zo kunnen we at runtime kiezen met welke controller we willen werken. Als laatste stap maken we dus een IController interface, met de methodes Create, Details en List:

```
public interface IController
{
    public IView Create();
    public IView Create(object model);
    public IView Details(int id);
}
```

```
public IView List();  
}
```



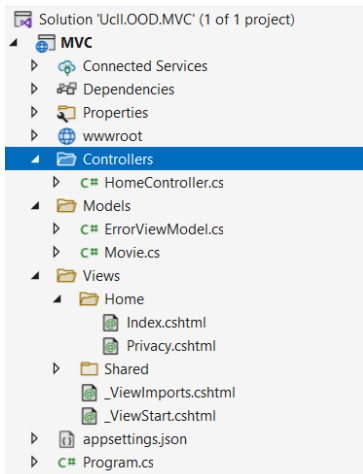
Het volledig uitgewerkt voorbeeld vindt u terug onder solution **Ucll.OOD.Console_MVC**



Onder de **Media Gallery** is er een video **OOD – Console Model-View-Controller** die dit voorbeeld doorloopt

4.5 Praktisch: MVC in ASP.NET Core

Het MVC pattern wordt standaard toegevoegd bij het aanmaken van een nieuw ASP.NET Core MVC project:



Ook in deze opzet zullen de views verantwoordelijk zijn voor de weergave, de models verantwoordelijk zijn voor de data en de controllers de lijm tussen de twee zijn. ASP.NET Core MVC voorziet ook dependency injection. Als we bijvoorbeeld onze HomeController openen, zien we:

```
private readonly ILogger<HomeController> _logger;  
  
public HomeController(ILogger<HomeController> logger)  
{  
    _logger = logger;  
}
```

We zien dus dat onze `_logger` via dependency injection wordt toegevoegd aan onze controller. Bij zelf aangemaakte controllers moeten we dit nog injecteren. De injectie zelf wordt in dit geval gedaan via een IoC Container binnen `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

De dependency injection voor logging zit gedefinieerd in de `CreateBuilder` actie. Er gaat een standaard implementatie voorzien worden, die wij dan ook in onze eigen controllers kunnen gebruiken. Voor meer informatie, zie <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder?view=aspnetcore-6.0>.

We zullen later zien hoe we zelf services kunnen registreren, maar deze werking volgt dezelfde logica als wat we bij IoC Container gezien hebben.

4.5.1 Opzetten Controller

We maken eerst een `UserController` aan. Als we rechtermuis klik > Add doen op de `Controllers` folder, krijgen we de optie om een controller toe te voegen. We krijgen als we dit selecteren dan 3 mogelijkheden. We kiezen voor `MVC Controller – Empty` en noemen deze `UserController`. Net zoals in onze `Console app`, gaan we hier ook verschillende acties toevoegen:

- Create
- List (dit gaat onze Index worden)
- Details

Zoals in onze `Console app` gaan we ook een user repository moeten maken (via Singleton). Dit is eigenlijk de exact zelf code dat we kunnen hergebruiken. Onder de maps `Models` maken we een class `UserRepository` en een class `User`. Het enige verschil is dat deze `User` class geen interface implementeert.

We willen nu onze UserRepository via dependency injection kunnen gebruiken in onze controller. We zullen deze service dus moeten registreren in onze IoC container:

```
var builder = WebApplication.CreateBuilder(args);

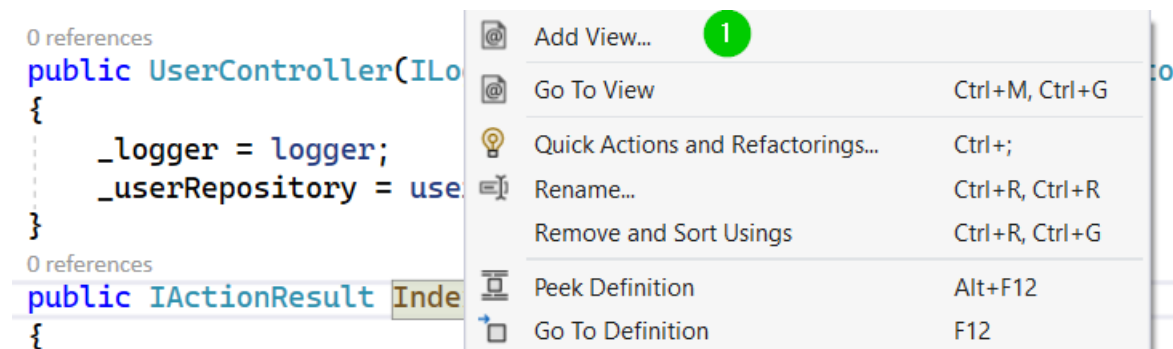
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton<UserRepository>();
var app = builder.Build();
```

Hierna kunnen we onze Index action aanpassen naar volgende code:

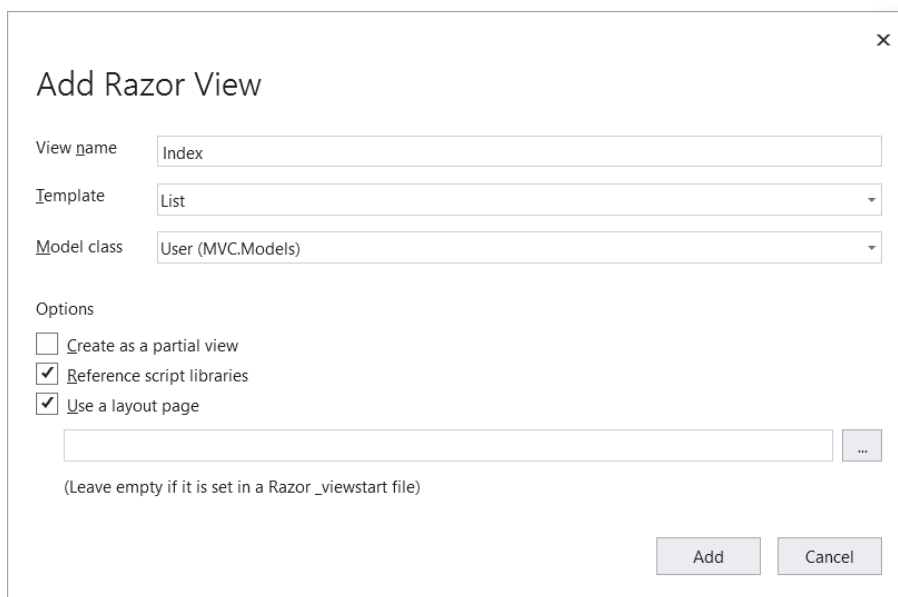
```
public IActionResult Index()
{
    var users = _userRepository.GetUsers();
    return View(users);
}
```

4.5.2 Opzetten Views

Als we dit al willen testen dat dit werkt, zullen we ook een view moeten aanmaken. Door rechtermuis klik te doen op onze Action-naam, kunnen we voor nieuwe view kiezen (1):



Kies hierna voor Razor view en kies volgende settings:



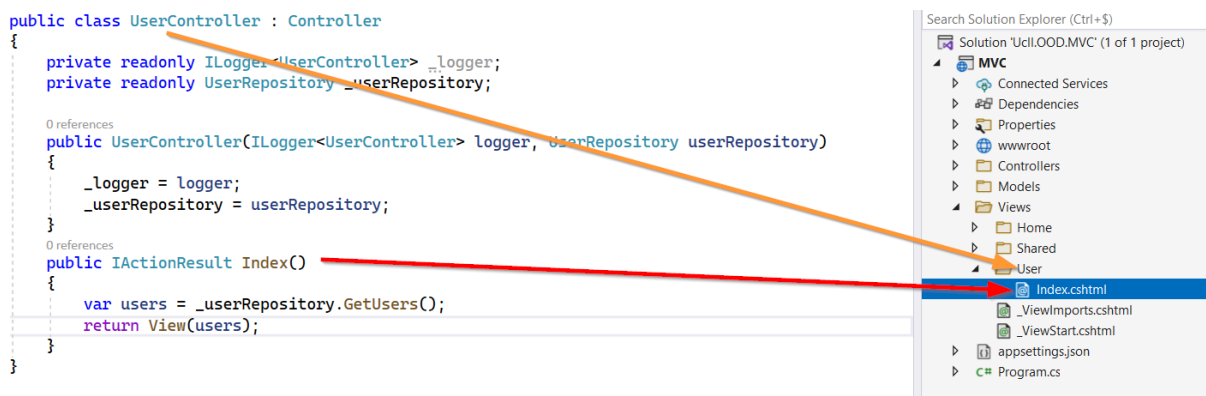
Dit wil zeggen:

We maken een View aan met als naam Index. We gaan deze view voor ons laten genereren met een template van een list (omdat we direct al de users op het eerste scherm wensen te zien) en als model class User (omdat het over een List<User> zal gaan). Visual Studio zal dan een view scaffolden (= code generatie) die we dan kunnen gebruiken.

Als we onze applicatie dan runnen, en we navigeren naar onze Controller (bv. <https://localhost:7046/user>) dan zien we een lege lijst. De link tussen de Controller en de View is dan deze regel code:

```
return View(users);
```

Gezien we hier niet expliciet zeggen welke view we wensen te nemen, gaat de compiler er van uit dat het de view met de naam van onze action is (in dit geval: Index). Waar zoekt hij die view? In de map User (= naam van de controller) onder de map views:



Er zit dus een relatie tussen de naam van onze Controller en de naam van de actie, en onze view.

Onze volgende stap is dan de Create action maken. Als we in net aangemaakte Index.cshtml openen zien we volgende lijn:

```
<p>
    <a asp-action="Create">Create New</a>
</p>
```

Dit is een link naar een action op een controller. Gezien we nog steeds in de map User zitten, gaat het dus over de UserController. We kunnen ook de controller expliciet meegeven, maar dit gaan we enkel doen als we naar een andere controller willen navigeren:

```
<p>
    <a asp-action="Create" asp-controller="User">Create New</a>
</p>
```

Om de Create action toe te voegen aan onze UserController, hebben we onderstaande code nodig:

```
public IActionResult Create()
{
    return View();
}
```

Net zoals bij Index kunnen hier via rechtermuisklik een view van maken. Kies weer Razor view, maar neem ditmaal Create als template. Het model blijft User. Na het aanmaken zien we dat er in onze User folder een nieuw view is toegevoegd:



Als we deze view openen, zien we dat hier een form is in aangemaakt. Deze form kan dan ingevuld worden door een eindgebruiker. Als we de form echter onderzoeken, zien we dat deze ook naar de Create action verwijst:

```
<form asp-action="Create">
```

4.5.3 Get en Post

We zien dus dat voor de Create action twee werkwijzen nodig hebben:

- Action voor het invulformulier te tonen
- Action voor het verwerken van het ingevulde formulier

Dit onderscheid kunnen we maken door het type van de verwerking.

Een action is eigenlijk een http-call. HTTP staat voor **H**yper**T**ext **T**ransfer **P**rotocol en is het protocol voor client-server communicatie (web). De communicatie loopt via een **request** (aanvraag vanuit de client naar de server) en een **response** (het antwoord op deze aanvraag, vanuit de server naar client). De controller zal dus een request (de action en zijn parameters) binnenkrijgen, en een antwoord (het IActionResult object) versturen.

Http is een *stateless protocol*, er is geen relatie tussen opeenvolgende request-response en er is dus geen blijvende client-server connectie. De client start de aanvragen en moet ervoor zorgen dat de request voldoende informatie bevat om deze succesvol door de server te laten verwerken. Elke HTTP actie kan met parameters aangeroepen worden. In je browser kan je requests inspecteren via F12 (Firefox, Chrome, Edge)

We gaan twee http acties in detail bekijken: **Get** en **Post**.

4.5.3.1 GET

Een GET request haalt data op. Het is dus een zuivere lees (read) operatie. Een GET kan slechts één object teruggeven, maar dit object kan wel een lijst zijn. Parameters komen binnen in een GET request via een Query string.

Dit is het deel van de url dat begint achter het vraagteken (?). Het gaat om name-value pairs, geschieden door een &:

- /test/testget.php?name1=value1&name2=value2
- /Test/TestGet/value1?name2=value2&name3=value3

name1, name2, name3 en name4 zijn de namen van de parameters.

value1, value2, value3 en value4 zijn hun respectievelijke waardes.

GET is in .NET het standaard type van de action. Elke Action binnen een Controller waarbij geen omschrijving wordt meegegeven zal dus via de GET methode werken:

```
public IActionResult Index()
{
    return View();
}

0 references
public IActionResult Privacy()
{
    return View();
}
```

Zowel de action Index als Privacy zullen dus HTTP GET acties zijn. In de browser inspector ziet bijvoorbeeld de call naar de Privacy action er als volgt uit:

Headers		Cookies	Request	Response	Timings
Filter Headers					
▶ GET https://localhost:7138/Home/Privacy					
Status	200 OK ?				
Version	HTTP/2				
Transferred	2,84 kB (2,72 kB size)				
Referrer Policy	strict-origin-when-cross-origin				

De status 200 OK duidt aan dat de request gelukt is.

We gaan een GET typisch gebruiken als we data wensen te tonen op het scherm.

4.5.3.2 POST

Wordt gebruikt om een update request naar de server te sturen. Data wordt niet via de query string meegegeven, maar via de request body. Dit betekent het niet zichtbaar zal zijn voor de gebruiker wat er meegestuurd wordt (tenzij ze de requests kunnen capteren).

Om een POST te versturen moeten we werken met een form. Deze form wordt vaak opgevuld met bestaande data. Om deze data op te halen, zullen we dus eerst een GET moeten doen. We krijgen in onze controller dus twee Actions voor één pagina:

```
[HttpGet]
public IActionResult Person()
{
    return View(new Person { Firstname = "Peter", Name = "Selie" });
}
```

```

    }
    [HttpPost]
    public IActionResult Person(Person person)
    {
        //logic to update the person
        return View(person);
    }

```

De eerste action zal de Person view opvullen met de geselecteerde persoon (het model). Het gaat hier over een HttpGet Action. De twee Action krijgt een Person object binnen. Dit is dus onze HttpPost Action.

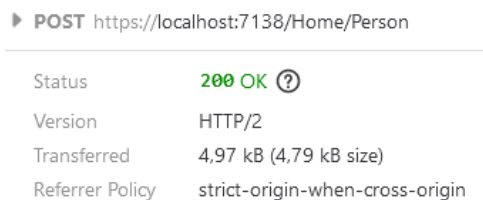
4.5.4 Afwerking

Hoe komt dat we de ene keer GET ontvangen van de client en de andere keer POST?

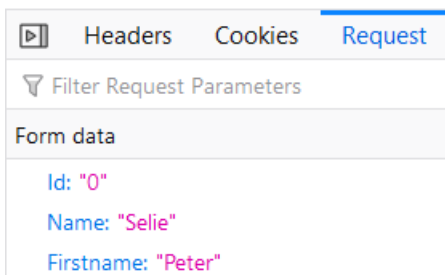
Als we gewoon navigeren naar de juiste endpoint (/Home/Person) zullen we een GET doen.

Als we echter een submit van een form doen naar deze endpoint, gebeurt er achter de schermen een POST call.

Als we dit bekijken via onze browser inspector zien we dit:



Op het tabblad request zien we dan de meegeleverde form data:



We zullen dus een extra action moeten toevoegen dat een User ontvangt, die naast onze action bestaat om de view te tonen:

```

public IActionResult Create()
{

```

```

        return View();
    }

    [HttpPost]
    public IActionResult Create(User user)
    {
        return View();
    }

```

In deze nieuw action kunnen we de user opslaan, en opnieuw de Index tonen:

```

[HttpPost]
public IActionResult Create(User user)
{
    _userRepository.AddUser(user);
    return RedirectToAction("Index");
}

```

We doen dit door de Index action binnen dezelfde controller op te roepen. Deze bevat reeds de logica om de lijst van users te tonen. De list view ondersteund ook enkele actions die we niet gaan uitwerken. We gaan dus onze Index view aanpassen om deze te verwijderen. De lijnen


```

@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
@Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })

```

mag je dus verwijderen.

Het gaat hier over ActionLinks. Dit zijn links naar actions binnen de controller. Door de definitie te bekijken, zien we wat de verschillende parameters doen:

 (extension) Microsoft.AspNetCore.Html.IHtmlContent IHtmlHelper.ActionLink(string linkText, string actionName, TModel routeValues) (+ 6 overloads)
Returns an anchor (<a>) element that contains a URL path to the specified action.

We laten de Details ActionLink staan, en maken hiervoor een nieuwe action op onze UserController. Via Add View kunnen hier dan een view voor aanmaken met als template Details en als model User. Gezien onze controller echter het model moet aanleveren aan de view, zullen we de user moeten ophalen. De code in commentaar geeft ons al een hint hoe we dit kunnen doen:

```

@Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ })

```

We gaan de Id van de User aanleveren aan de action. De action zal dan op basis van deze Id de user opzoeken in de repository en deze meegeven aan de Details view.

```
@Html.ActionLink("Details", "Details", new { id=item.Id })
```

In de controller vangen we deze parameter dan op:

```
public IActionResult Details(int id)
{
    return View();
}
```

De naamgeving is hier zeer belangrijk. Als we in de view een anoniem object aanmaken met de naam id, dan moet ook de parameter in de controller de naam id krijgen. Omdat we data gaan ophalen, zal dit dus een GET operatie zijn:

```
public IActionResult Details(int id)
{
    var user = _userRepository.GetUser(id);
    return View(user);
}
```



De volledige code van deze praktische uitwerking vind je terug onder de solution **Ucll.OOD.MVC**



Onder de **Media Gallery** is er een video **OOD –MVC in ASP.NET Core** die dit voorbeeld doorloopt

4.6 ViewModel

Wanneer we een view wensen te tonen met verschillende gecombineerde gegevens moeten gebruik maken van een ViewModel. Dit is een zelf gemaakt model, die verschillende modellen kan combineren en hierop aparte logica kan toepassen.



Dit voorbeeld gebruikt de code van de solution **Ucll.OOD.ViewModel**

In deze solution vinden we twee models:

- Order – een bestelling op naam van een klant
- OrderDetail – de verschillende items die in deze bestelling zitten

Stel, we wensen nu een overzicht van de klantgegevens, met de nettoprijs per order detail. We willen dit ook tonen in een view. We gaan dus een nieuwe class moeten aanmaken die deze gegevens combineert. Naast een action op de controller en een view, zullen we dus ook een specifiek model moeten aanmaken. Dit model mapt niet rechtstreeks op een databank object, en noemen we dus een ViewModel.

We lossen dit op door een nieuwe folder ViewModels te maken, en hieronder een nieuwe class NetPriceViewModel te creëren:

```
public class NetPriceViewModel
{
    public int Id { get; set; }
    public string CustomerName { get; set; }
    public decimal Quantity { get; set; }
    public decimal Price { get; set; }
    public string Item { get; set; }
    public decimal LineValue
    {
        get
        {
            return Quantity * Price;
        }
    }
}
```

```
}  
}
```

Dit model combineert dan de waardes van zowel het Order model als het OrderDetail model. In onze Controller kunnen we dan de nodige gegevens samenvoegen tot één model:

```
public IActionResult NetPriceOrderDetail()  
{  
    var modellist = new List<NetPriceViewModel>();  
    var orders = _orderRepository.GetOrders();  
    foreach (var order in orders)  
    {  
  
        var details =  
_orderRepository.GetDetailOrdersForOrder(order.Id);  
        foreach (var detail in details)  
        {  
            var model = new NetPriceViewModel  
            {  
                Id = order.Id,  
                CustomerName = order.CustomerName,  
                Item = detail.ItemCode,  
                Price = detail.UnitPrice,  
                Quantity = detail.Quantity  
            };  
            modellist.Add(model);  
        }  
  
    }  
  
    return View(modellist);  
}
```

Een ViewModel is dus niets meer dan een class die een combinatie tussen 2 of meerdere models bepaald. Het zorgt ervoor dat de view zelf geen kennis moet hebben van vanwaar de

gegevens komen, de view is enkel verantwoordelijk om de gegevens die binnen komen te tonen.



5 Unit Testing

In de eerste les hebben we het concept refactoring besproken. Tijdens de development lifecycle van een programma komt het vaak voor dat er iets moet aangepast worden aan een methode, zonder dat we de functionaliteit ervan willen wijzigen. We willen bijvoorbeeld een design pattern implementeren zodat we de code makkelijk kunnen uitbreiden, maar dit mag bestaande code niet *breken*.

Om dit praktisch voor elkaar te krijgen moeten we dan, na elke aanpassing, onze methode(s) volledig testen. Wanneer we een complexe methode hebben met veel mogelijke paden, moeten we dus elk pad opnieuw testen. Dit is zeer tijdsintensief. Door dit manueel te doen is er ook een grote kans op fouten. We kunnen een specifieke test vergeten, we kunnen het resultaat foutief interpreteren, We moeten dit dus automatiseren.

Binnen .NET hebben we de keuze uit verschillende frameworks om deze testen te automatiseren. Deze verschillende frameworks doen wel grotendeels hetzelfde. Via Unit Testing gaan we *één* enkele methode testen. Door zo unit tests te schrijven voor elke methode binnen ons programma, kunnen we er zeker van zijn dat onze werking na aanpassingen nog steeds correct is.

Stel we hebben volgende class:

```
public class ComplexCalculator
{
    public int CalculateTotal(int start, bool extraComplexity)
    {
        if (extraComplexity)
        {
            if (start < 0)
                return -1;
            else
                return start + 25;
        }
        else
        {
```

```

        if (start < 0)
            return 0;
        else
            return start + 20;
    }
}

```

Als we dit manueel zouden moeten testen, moeten we volgende controles doen:

- Start < 0 && extraComplexity == true
- Start = 0 && extraComplexity == true
- Start > 0 extraComplexity == true
- Start < 0 && extraComplexity == false
- Start = 0 && extraComplexity == false
- Start > 0 extraComplexity == false

We zouden dus een regel kunnen maken dat zegt:

Als de waarde van *start* -5 is en *extraComplexity* is true, dan verwacht ik een resultaat met waarde -1.

Zulke regels definiëren, levert ons redelijk wat voordelen op:

- Code kan geautomatiseerd getest worden
- Er is extra documentatie
- Men wordt gedwongen na te denken over de edge cases

5.1 Onze eerste Unit Testen

We gaan via een praktisch voorbeeld aantonen hoe we Unit Testen kunnen schrijven en wat hun voordeel op lange termijn is.



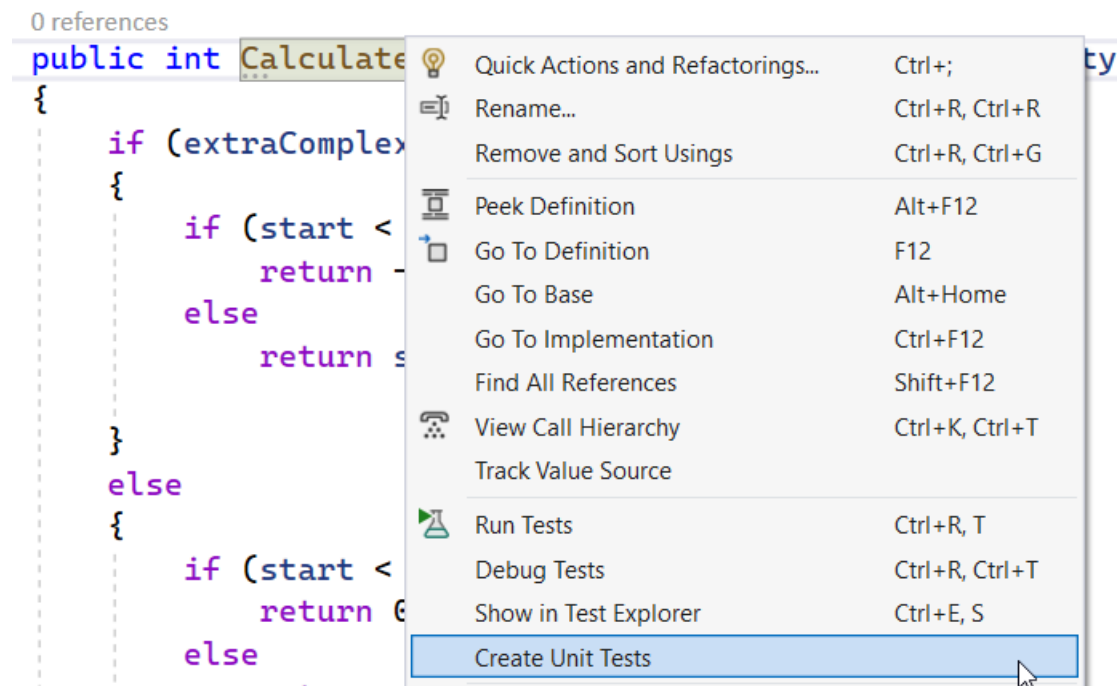
Dit voorbeeld gebruikt de code van de solution **Ucll.OOD.UnitTesting**

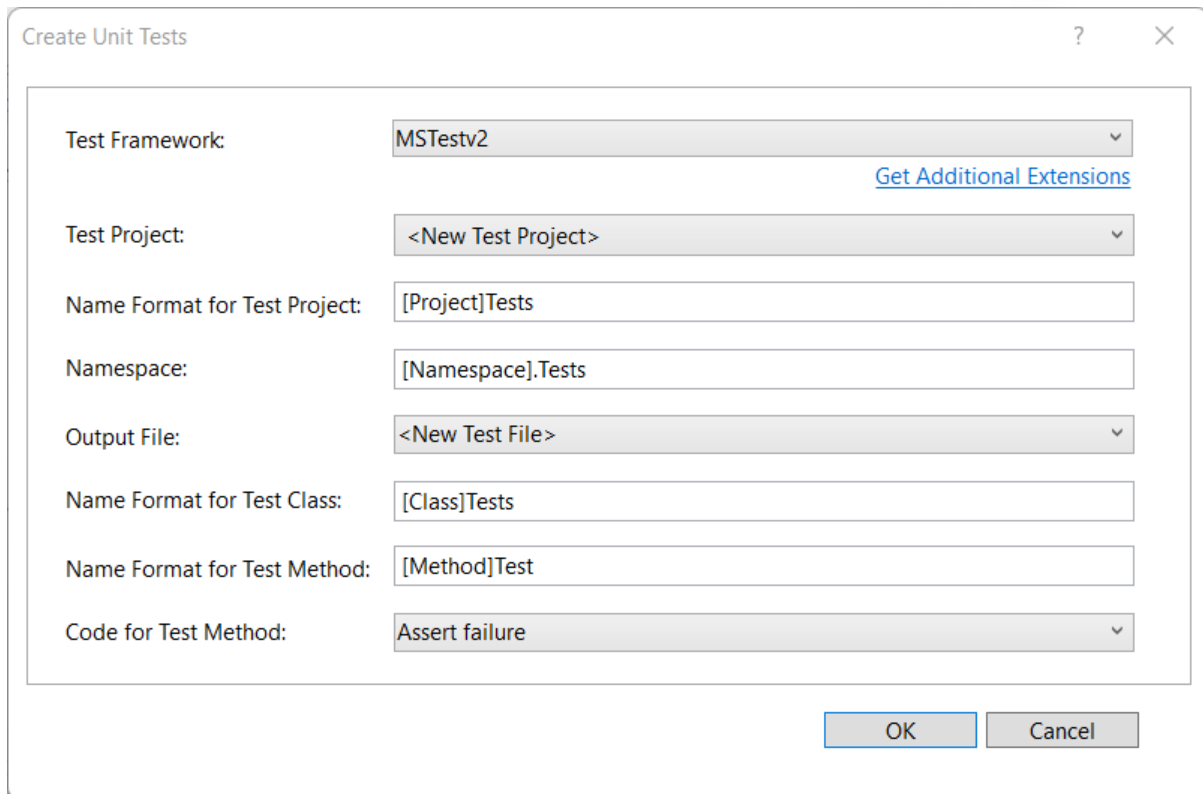


Onder de **Media Gallery** is er een video **OOD – Eerste UnitTests** die dit voorbeeld doorloopt

5.1.1 Testen aanmaken

Visual Studio maakt Unit Testen aanmaken zeer makkelijk. Via rechtermuisklik op je methode, kan je een unit test toevoegen:





Create Unit Tests

Test Framework: MSTestv2 [Get Additional Extensions](#)

Test Project: <New Test Project>

Name Format for Test Project: [Project]Tests

Namespace: [Namespace].Tests

Output File: <New Test File>

Name Format for Test Class: [Class]Tests

Name Format for Test Method: [Method]Test

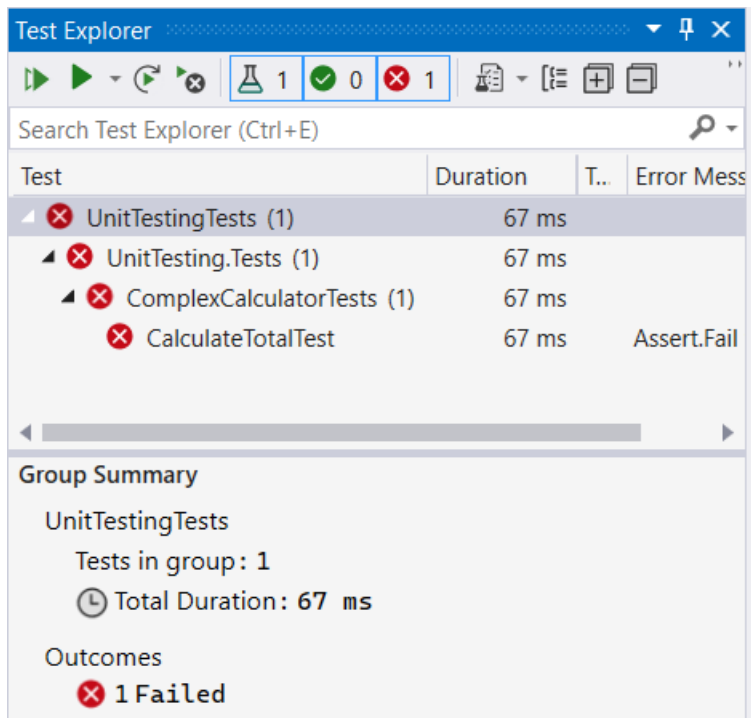
Code for Test Method: Assert failure

OK Cancel

We duwen hierop gewoon OK. We zien dat er 4 dingen gebeuren:

- Er wordt een project toegevoegd aan onze solution waarin we al onze testen verzamelen
- Er wordt een class aangemaakt in dit nieuw project, waarin de testen van onze geselecteerde class verzameld worden
- Er wordt binnen deze class een methode aangemaakt om de methode die we geselecteerd hebben te testen
- Het resultaat van deze test is negatief (`Assert.Fail();`)

Als we rechts klikken op dit nieuwe project, zien we de optie *Run Tests*. De *Test Explorer* (View > Test Explorer) wordt zichtbaar en onze testen worden uitgevoerd. We krijgen het resultaat te zien in een overzicht scherm:



De methode dat aangemaakt is, is echter niet degene dat we willen testen. We gaan deze hernoemen naar **CalculateTotalStarNegativeComplexTrueTest**, ons eerste scenario dat we daarstraks besproken hebben. We verwachten hierbij een uitkomst van -1.

Onze test wordt dan:

```
[TestMethod()]
public void CalculateTotalStartNegativeComplexTrueTest()
{
    var tester = new ComplexCalculator();
    Assert.AreEqual(-1, tester.CalculateTotal(-5, true));
}
```

Via `Assert.AreEqual` kunnen we controleren of onze verwachte waarde (-1) gelijk is aan de waarde die de methode teruggeeft (`tester.CalculateTotal(-5, true)`).

We kunnen op een gelijkaardige manier onze andere scenario's ook implementeren. Opgelet, bij de nieuwe methodes moeten ook de annotations (het `[TestMethod()]` gedeelte) toegevoegd worden:

```
[TestMethod()]
public void CalculateTotalStartZeroComplexTrueTest()
{
```

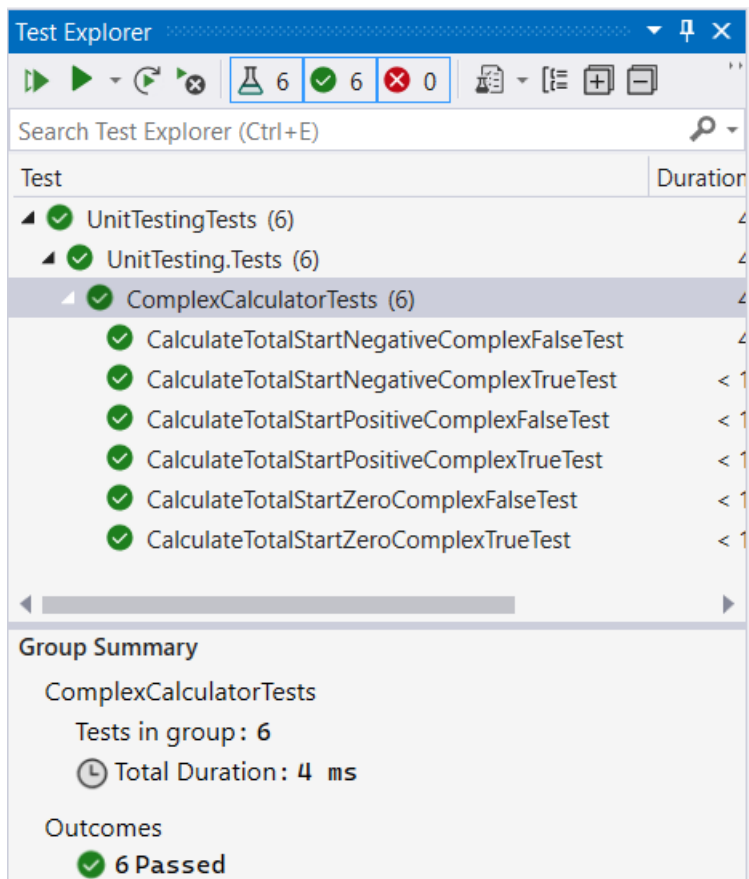
```

    var tester = new ComplexCalculator();
    Assert.AreEqual(25, tester.CalculateTotal(0, true));
}

```

Zo weet de compiler dat dit methodes zijn dat gebruikt worden als tests.

We kunnen dan zo onze 6 scenario's toevoegen:



5.1.2 Optimimalisaties

We zien echter dat we één regel code steeds herhalen voor onze testen:

```

var tester = new ComplexCalculator();

```

Dit is natuurlijk in strijd met het **DRY principe**. Dit is makkelijk op te lossen, gezien ComplexCalculatorTests een class is. We kunnen dus een private variabele aanmaken die onze initiatie doet van onze ComplexCalculator object:

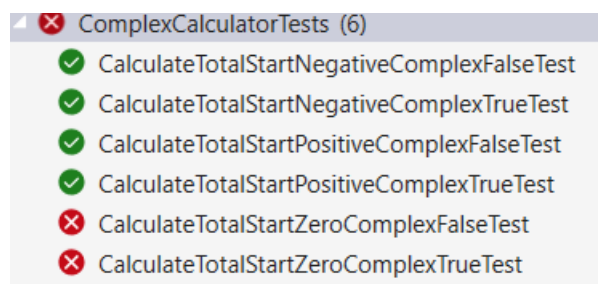
```

private ComplexCalculator _calculator;

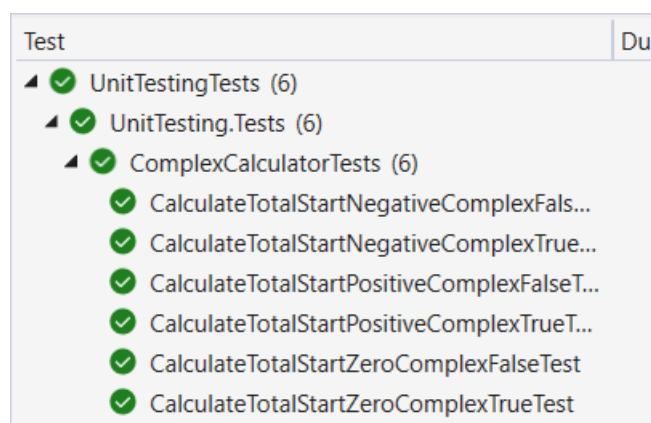
```

```
public ComplexCalculatorTests()
{
    _calculator = new ComplexCalculator();
}
```

Na een tijdje wordt er gevraagd om de CalculateTotal methode aan te passen, zodat als de waarde van start 0 is, we dit gaan afhandelen alsof het negatief is. Door onze testen hierna opnieuw te laten lopen, zien we dat er twee testen falen:



Dit is normaal, gezien hun werking functioneel veranderd is. Als een van de andere testen ook gefaald had, dan was dit het resultaat van een bug die geïntroduceerd was bij de aanpassing. Door de testen zouden we dit dan direct opmerken. We moeten dus ook onze testen aanpassen, zodat de nieuwe berekening correct gecontroleerd wordt. Hierna kunnen we de testen opnieuw laten draaien:



5.1.3 Complexere objecten

Wat als we echter geen ingebouwd type wensen te controleren, maar eigen objecten?

We maken een nieuwe class aan:

```
public class CalculatorResult
```



```
{  
    public string Description { get; set; }  
    public int Result { get; set; }  
}
```

In onze ComplexCalculator maken we dan een nieuwe methode aan:

```
public CalculatorResult CalculateCalculatorResult(int start, bool  
                                                    type)  
{  
    if (type)  
    {  
        return new CalculatorResult { Description = "My Result",  
                                       Result = start + 5 };  
    }  
    return new CalculatorResult { Description = "Another Result",  
                                   Result = start - 5 };  
}
```

We kunnen deze methode dan ook unit testen:

```
[TestMethod()]
public void CalculateCalculatorResultTestTypeTrue()
{
    var correct = new CalculatorResult
    {
        Description = "My Result",
        Result = 10
    };
    Assert.AreEqual(correct,
        calculator.CalculateCalculatorResult(5, true));
}
```

Als we echter onze testen laten lopen, zien we dat onze nieuwe unit test faalt. Dit komt omdat de compiler niet weet wanneer een object gelijk is. We zullen dus de **Equals methode** moeten implementeren op onze CalculatorResult class:

```
public override bool Equals(object? obj)
{
    var caculatorResult = obj as CalculatorResult;
    if (caculatorResult is null) return false;

    return caculatorResult.Description == Description &&
        caculatorResult.Result == Result;
}
```

Hierna zal onze test wel groen kleuren. We moeten er dus op letten dat wanneer we objecten vergelijken, het voor de compiler steeds duidelijk is wanneer objecten gelijk zijn aan elkaar.

Tot zo ver zijn we de happy paths aan het testen. Een happy path is de flow in de code waarin de code normaal uitgevoerd wordt. Stel dat onze methode op een bepaald punt echter een exception zal smijten, hoe testen we dit?

We voegen in onze CalculateCalculatorResult methode een controle toe, die een exception gaat gooien:

```
if (start < 0)
    throw new ArgumentOutOfRangeException("Start can't be below zero");
```

We gaan dus geen CalculateResult krijgen wanneer we de start parameter doorgeven met een waarde kleine dan 0. Via onze normale Assert functies kunnen we dit niet aftoetsen. We zullen dus de ThrowsException methode moeten gebruiken:

```
[TestMethod()]
public void CalculateCalculatorResultTestException()
{
    Assert.ThrowsException<ArgumentOutOfRangeException>(() =>
        _calculator.CalculateCalculatorResult(-5, true));
}
```

We zien in deze werking enkele grote verschillen met de Assert.AreEqual werking.

Tussen de <> geven mee welk type van exception we verwachten. Als parameter geven we niet langer een waarde mee, maar een delegate. Een delegate is een manier om een functie door te geven als parameter. In dit geval geven we een anonieme functie door (= functie zonder naam) waarvan de uitvoering gelijkgesteld wordt aan de uitvoering van `_calculator.CalculateCalculatorResult(-5, true)`.

5.2 Test Driven Development

Unit testing worden vooral na de feiten geschreven. De ontwikkelaar gaat eerst de code schrijven en hierna deze code beveiligen door unit testen errond te schrijven. Test Driven Development (TDD) is een programmeerstijl waarbij we echter eerst de testen gaan schrijven, en hierna pas de eigenlijke code.

Dit verplicht ons om vanaf het eerste moment een duidelijk beeld te hebben wat we willen en waar we heen gaan, en zorgt ervoor dat al onze code onder unit testing komt. Het nadeel is dat deze manier van werking trager is in het begin, en meer moeite van de ontwikkelaar vraagt.

5.2.1 Praktisch voorbeeld

Stel, we hebben volgende story.

Een gebruiker wil een validatie voor wachtwoorden. Een wachtwoord moet bestaan uit minstens 8 karakters, en moet een hoofdletter en een cijfer bevatten. De validatie is ofwel juist, ofwel fout.

We zouden dit als volgt kunnen oppakken. Nadat we een console app gemaakt hebben, maken we direct ook een unit test project in dezelfde solution. Gezien we dit nu manueel doen, kiezen we voor een MSTest Test Project. We maken hierin een class PasswordValidationTests:

```
[TestClass]
public class PasswordValidationTests
{
    [TestMethod]
    public void PasswordValidationMinimumTest()
    {
        PasswordValidator validator = new PasswordValidator();
        Assert.AreEqual(true, validator.Validate("Abcdef12"));
    }
}
```

Bij het schrijven van deze test bestaat onze PasswordValidator class nog niet. We kunnen onze test dus niet runnen. We gaan dus deze class aanmaken in ons hoofdproject. We kunnen deze test doen slagen door onderstaande code te implementeren:

```
public class PasswordValidator
{
    public bool Validate(string password)
    {
        return true;
    }
}
```

We zien echter al één probleem. Voor de validatie te doen moeten we steeds PasswordValidator initialiseren. Dit is in se niet nodig. We herschrijven onze test dus als volgt:

```
[TestMethod]
public void PasswordValidationMinimumTest()
{
    Assert.AreEqual(true, PasswordValidator.Validate("Abcdef12"));
}
```

En passen hierna de code aan (static maken van de class):

```
public static class PasswordValidator
{
    public static bool Validate(string password)
    {
        return true;
    }
}
```

Natuurlijk geeft deze test een fout beeld terug. Eigenlijk werkt onze code niet zoals functioneel gevraagd is. We introduceren dus een tweede test:

```
[TestMethod]
public void PasswordValidationTooShortTest()
{
    Assert.AreEqual(false, PasswordValidator.Validate("Abcd12"));
}
```

Als we deze nu runnen, zal deze test falen. We zullen onze code dus moeten aanpassen om ook deze test te doen slagen.

```
public static bool Validate(string password)
{
    if (password.Length < 8) return false;
    return true;
}
```

We hadden echter ook de noodzaak dat het paswoord een numerieke waarde zou bevatten. We schrijven hiervoor dus een nieuwe test:

```
[TestMethod]
public void PasswordValidationNoDigitTest()
{
    Assert.AreEqual(false, PasswordValidator.Validate("Abcdefgh"));
}
```

We zorgen ervoor dat de lengte van het paswoord wel correct is, zodat we niet in het vaarwater komen van onze eerdere test. Zoals verwacht faalt deze test ook. We passen wederom onze code aan:

```
public static bool Validate(string password)
{
    if (password.Length < 8) return false;
    if (password.Any(char.IsDigit) is false) return false;
    return true;
}
```

We gebruiken LINQ om ons paswoord te valideren. Via de Any functie kunnen meegeven of een char in de string een digit is. Als dit niet zo is, geven we false terug.

Een laatste vereiste was dat we een hoofdletter moest hebben in onze string. We schrijven dus een nieuwe test:

```
[TestMethod]
public void PasswordValidationNoUppercaseTest()
{
    Assert.AreEqual(false, PasswordValidator.Validate("abcdefgh"));
}
```

En passen nog eens onze code aan:

```
public static bool Validate(string password)
{
    if (password.Length < 8) return false;
    if (password.Any(char.IsDigit) is false) return false;
    if (password.Any(char.IsUpper) is false) return false;
    return true;
}
```

We merken ook op dat onze bestaande testen blijven werken. We weten dus dat onze ontwikkeling de juiste kant aan het uitgaan is. We kunnen nu enkele edge cases bekijken. Wat met een string dat alleen maar uit nummers bestaat?

```
[TestMethod]
public void PasswordValidationOnlyDigitsTest()
{
    Assert.AreEqual(false, PasswordValidator.Validate("12345678"));
}
```

Of een test met enkel hoofdletters:

```
[TestMethod]
public void PasswordValidationOnlyUppercaseTest()
{
    Assert.AreEqual(false, PasswordValidator.Validate("ABCDEFGH"));
}
```

Beide testen geven inderdaad false terug. Een andere edge case zou een paswoord zijn met enkel hoofdletters en nummers:

```
[TestMethod]
public void PasswordValidationOnlyUppercaseAndDigitsTest()
{
    Assert.AreEqual(false, PasswordValidator.Validate("ABCD1234"));
}
```

Deze test faalt! We zullen dus onze code terug moeten aanpassen:

```
public static bool Validate(string password)
{
    if (password.Length < 8) return false;
    if (password.Any(char.IsDigit) is false) return false;
    if (password.Any(char.IsUpper) is false) return false;
    if (password.Any(char.IsLower) is false) return false;

    return true;
}
```

Dit is een zeer eenvoudig voorbeeld van hoe we TDD kunnen doen. We schrijven eerst onze testen, waarna we de code maken. We zorgen er steeds voor dat we de meest eenvoudige code schrijven om de test te doen slagen. Per verfijning (refactoring) wordt onze code dan complexer. Doordat we testen hebben, zien we ook de edge cases en de verwachte resultaten voor deze edge cases. Onze unit testing zijn eigenlijk de acceptatiecriteria van onze code. We kunnen zelfs onze code refactoren naar een andere vorm:

```
public static bool Validate(string password)
{
    return password.Length >= 8 &&
        password.Any(char.IsDigit) &&
        password.Any(char.IsUpper) &&
        password.Any(char.IsLower);
}
```

We zien door onze testen opnieuw te runnen, dat deze allemaal groen blijven. We kunnen er dus zeker van zijn dat de bestaande functionaliteit niet in het gedrang komt.



Dit voorbeeld gebruikt de code van de solution **Ucll.OOD.TDD**.

5.3 Mocking

Tot zo ver hebben we ons beperkt om unit testen te schrijven voor methodes die geen afhankelijkheden hebben. Deze methodes krijgen parameters binnen, doen hun ding en geven een resultaat terug. Denk even terug aan de ViewModel les. Hierin nam een methode (een action op onze controller) een parameter binnen, maar ging op basis van deze parameter gegevens ophalen uit een databank. Deze gegevens werden dan getransformeerd naar onze ViewModel.

Als we unit testen, kunnen we geen calls naar databanken of filesystemen doen. We willen enkel de functionaliteit van onze methode testen, niet van zijn afhankelijkheden. We zullen deze afhankelijkheden dus moeten nabootsen. Dit doen noemen we mocking.



Om dit te demonstreren is een **ASP.NET Core MVC** project dat je kan vinden onder **Uccl.OOD.Mocking**

Dit project werkt met een SQLite databank. In plaats van Entity Framework, wordt er *Dapper* gebruikt. *Dapper* is ook een ORM (object-relational mapping) tool.

Dit wil zeggen dat, net zoals Entity Framework, Dapper ervoor gaat zorgen dat we records uit tabellen van de databank gaan kunnen gebruiken als objecten in onze code. Het project heeft twee repositories die via dependency injection toegevoegd worden (via de IoC container onder Program.cs).

We zien dat in de HomeController een action Details bestaat. Deze action gaat twee modellen combineren naar een CustomerDetailModel. Hierbij zijn enkele basis berekeningen nodig. We willen er echter zeker van zijn dat deze berekeningen kloppen en dat dit ViewModel dus wel degelijk de juiste resultaten zal tonen. Hiervoor zullen we onze databank dus moeten **mocken**.

5.3.1 Moq Framework

We hebben dit makkelijker gemaakt door de logica te separeren van de controller. We hebben een CustomerOrderService class gemaakt, die gaat fungeren als tussenlaag tussen onze data en de controllers. Dit maakt het zo dat we deze service class kunnen unit testen, zonder dat we rekening moeten houden met de functionaliteit van de controllers. We hebben dus een methode die een concreet resultaat teruggeeft (class CustomerDetailModel, methode GetCustomerDetailModel) die we kunnen unit testen.

We maken een test class CustomerOrderServiceTests aan, die deze methode gaat testen:

```
[TestClass()]
public class CustomerOrderServiceTests
{
    public CustomerOrderServiceTests() { }

    [TestMethod()]
    public void GetCustomerDetailModelTest()
    {
        Assert.Fail();
    }
}
```

```
}
```

Als we een `CustomerOrderService` object wensen aan te maken, zien we echter dat de constructor twee andere classes verwacht:

- `ICustomerRepository`
- `IOrderRepository`

Dit zijn twee repositories met connecties naar de databank. Deze connectie wensen we niet te testen, dus we zullen deze twee objecten moeten nabootsen. Gelukkig gaat het hier over interfaces, dus kunnen we zelf een mock object aanmaken, die deze interfaces volgen. Via het **Moq framework** (te installeren via NuGet, zoeken op Moq) kunnen we dit als volgt implementeren:

```
private readonly Mock<IOrderRepository> _orderRepositoryMock = new
Mock<IOrderRepository>();

private readonly Mock<ICustomerRepository> _customerRepositoryMock = new
Mock<ICustomerRepository>();
```

We zeggen hier dat we een object gaan aanmaken dat de logica nabootst van de twee interfaces die we mocken. We gaan dus objecten krijgen waarvan de verschillende methodes die gedefinieerd zijn op onze interface kunnen aanspreken. Hierna kunnen we onze service class dan aanmaken met deze twee objecten:

```
_customerOrderService = new
CustomerOrderService(_customerRepositoryMock.Object,
_orderRepositoryMock.Object);
```

Deze objecten implementeren de nodige interfaces, dus dit lukt perfect.

Onze mock objecten hebben echter geen echte implementatie achter de nagebootste methodes zit, zullen we deze ook moeten definiëren.

De methode `GetCustomerDetailModel` gebruikt twee methodes van de repositories:

- `_customerRepository.GetCustomerById(customerId)`
- `_orderRepository.ForCustomer(customerId);`

We focussen eerst op de GetCustomerById. De eerste stap is een object aanmaken dat zal dienen als het resultaat van deze functie:

```
var customer = new Customer
{
    Id = 1,
    Name = "Wim Vanden Broeck",
    Email = "wim.vandenbroeck@uc11.be"
};
```

Hierna zullen we dan de methode GetCustomerById nabootsen, waarbij we zelf het resultaat meegeven:

```
_customerRepositoryMock.Setup(x =>
    x.GetCustomerById(1)).Returns(customer);
```

Hier zeggen we dus: voor onze CustomerRepository die we nabootsen, zullen we voor de methode GetCustomerById het object customer teruggeven.

Voor de ForCustomer methode is de werking identiek. We maken eerst het object dat we wensen terug te geven:

```
var orders = new List<Order>
{
    new Order { Id = 1, ProductName = "Laptop", Price = 1250,
        Quantity = 1, CustomerId = 1 },
    new Order { Id = 1, ProductName = "Keyboard", Price = 33.33,
        Quantity = 2, CustomerId = 1 }
};
```

En hierna definiëren we dat de specifieke methode dit object teruggeeft:

```
_orderRepositoryMock.Setup(x => x.ForCustomer(1)).Returns(orders);
```

Door dan onze te testen methode uit te voeren:

```
var model = _customerOrderService.GetCustomerDetailModel(1);
```

gaan we achterliggend de gemockte objecten aanspreken. We weten dus welke objecten we gaan terugkrijgen in de aangesproken methode. We kunnen dus een bepaalde property van ons model vergelijken met wat we verwachten:

```
Assert.AreEqual(1316.66, model.TotalPrice);
```

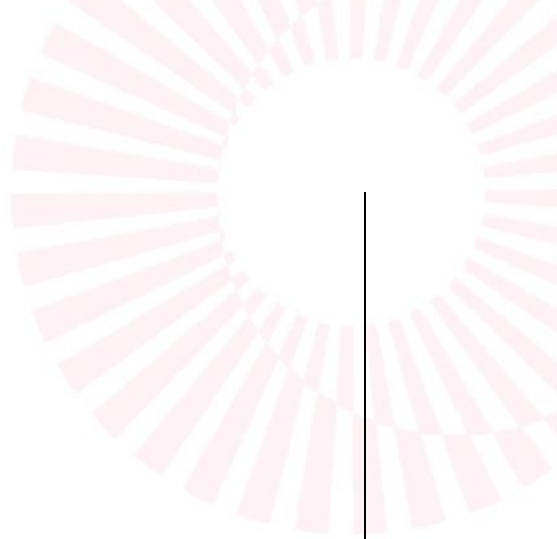
De ganse class ziet er dan als volgt uit:

```
[TestClass()]
public class CustomerOrderServiceTests
{
    private readonly CustomerOrderService _customerOrderService;
    private readonly Mock<IOrderRepository> _orderRepositoryMock =
        new Mock<IOrderRepository>();
    private readonly Mock<ICustomerRepository> _customerRepositoryMock =
        new Mock<ICustomerRepository>();

    public CustomerOrderServiceTests()
    {
        _customerOrderService = new
            CustomerOrderService(_customerRepositoryMock.Object,
                _orderRepositoryMock.Object);
    }

    [TestMethod()]
    public void GetCustomerDetailModelTotalPriceTest()
    {

```



```

//1. Create the object that we will return
var customer = new Customer
{
    Id = 1,
    Name = "Wim Vanden Broeck",
    Email = "wim.vandenbroeck@uc11.be"
};

//2. Mock the method
_customerRepositoryMock.Setup(x =>
    x.GetCustomerById(1)).Returns(customer);

//We will also mock this method

//1. Create the object that we will return
var orders = new List<Order>
{
    new Order { Id = 1, ProductName = "Laptop",
        Price = 1250, Quantity = 1, CustomerId = 1 },
    new Order { Id = 1, ProductName = "Keyboard",
        Price = 33.33, Quantity = 2, CustomerId = 1}
};

//2. Mock the method
_orderRepositoryMock.Setup(x =>
    x.ForCustomer(1)).Returns(orders);
var model = _customerOrderService.GetCustomerDetailModel(1);
Assert.AreEqual(1316.66, model.TotalPrice);
}
}

```

Zo kunnen we dus code testen die leunt op externe factoren waar we geen macht over hebben.

5.4 Code coverage

Code coverage is een term om aan te duiden voor hoeveel van de code er unit tests bestaan. Het wordt uitgedrukt in een percentage. Voor Visual Studio is deze optie enkel beschikbaar in de (betalende) Enterprise Edition.

Het grootste voordeel van code coverage is dat het aantoont welke methodes er *niet* ge-unit test zijn. Dit kan ons helpen eventuele blinde vlakken te identificeren.

Vaak wordt er een bepaald percentage gehanteerd om te bewijzen dat de code voldoende automatische testen heeft. Dit percentage wordt door het team zelf bepaald en zit tussen de 80% en 90%.

Code coverage werkt op basis van de paden in je code. Stel, je heb een if-structuur in je code, dan heb je eigenlijk 2 paden: één voor wanneer we *true* moeten evalueren en één voor het *false* pad. Code coverage zal dus nagaan of er voor elk pad een test bestaat.

Code coverage gaat enkele dingen niet doen:

- Kan niet controleren of de testen juist zijn
 - Testen kunnen zelf fouten bevatten
 - Testen kunnen geschreven zijn op foutieve analyse
- Kan niet controleren of alle testen gedaan zijn voor een bepaald pad
 - Zeker bij complexe return types zoals objecten gaat code coverage enkel zien dat er een object moet teruggegeven worden. Met het veelvoud aan mogelijkheden wordt geen rekening gehouden

6 Asynchroon programmeren

De code die we tot nu toe in onze console applicaties geschreven hebben kunnen we zien als synchroon programmeren. We voeren een statement uit, wachten tot deze voltooid is en voeren dan de volgende lijn code uit. In onze GUI betekent dit bijvoorbeeld dat je niets kan intypen terwijl er een berekening bezig is. Deze werking heet **synchroon programmeren**. Met de komst van het web is deze werking echter achterhaald. We willen natuurlijk niet dat onze website helemaal vastloopt terwijl er een berekening loopt of terwijl we wachten op data vanuit de databank.

We gaan dus **asynchroon** willen werken: terwijl het programma bezig is met de data op te halen, willen we gewoon kunnen scrollen en eventueel op andere links duwen.

Asynchroon werken maakt gebruik van de processerkracht. Er gaan statements uitgevoerd worden op de verschillende cores, waardoor deze **tegelijkertijd** uitgevoerd kunnen worden. Hoe sterker je pc dus, hoe meer voordeel er gehaald kan worden uit asynchroon werken.

Asynchroon programmeren kan echter niet in elke situatie gebruikt worden. Bij sommige code is de volgorde natuurlijk belangrijk. We gaan dus selectief moeten zijn in de use-cases waarin we asynchroon programmeren gebruiken.

Een laatste puntje dat we graag meegeven is de naamconventie. Asynchrone methodes krijgen de suffix Async. Dit passen we overal toe, behalve bij events als we GUI applicaties maken.

6.1 Praktisch: van Sync naar Async

We gaan voor de verandering eens een applicatie maken met een GUI. Hiervoor gebruiken we een Windows Form App project. In deze applicatie gaan we enkele sites downloaden (een actie die relatief veel tijd kost), zodat we kunnen kijken wat het effect is van zulke zware operaties, en hoe we deze kunnen verminderen.



De code van het volgende voorbeeld vind je onder solution **Uccl.OOD.ASYNC**



Onder de **Media Gallery** is er een video **OOD – Sync naar Async** die dit voorbeeld doorloopt

6.1.1 Opzetten project

In ons Windows Form App maken we een class DownloadService. In deze class maken we een synchrone methode DownloadWebsite aan. We gebruiken hiervoor de verouderde class WebClient, en negeren de foutboodschappen dat dit geen ideale oplossing is:

```
public string DownloadWebsite(string url)
{
    //we use an obsolete class to demonstrate sync downloading
    var client = new WebClient();
    return client.DownloadString(url).Length.ToString();
}
```

We gaan dus een webpagina downloaden, en de lengte van de gedownloade html pagina teruggeven. We doen dit dan voor een paar sites, zodat we daadwerkelijk wat tijd nodig hebben:

```
private void btnSync_Click(object sender, EventArgs e)
{
    var watch = new Stopwatch();
    watch.Start();

    StringBuilder sb = new StringBuilder();
    foreach (var website in _websites)
    {
        sb.AppendLine($"{website}: {
            DownloadService.DownloadWebsite(website)}");
    }
    watch.Stop();
    sb.AppendLine($"Elapsed time: {watch.ElapsedMilliseconds}");
    tbOutput.Text = sb.ToString();
}
```



```
}
```

Wanneer we dit testen zien we volgend resultaat:

```
http://www.stackoverflow.com: 175224
http://www.google.com: 49759
http://www.wikipedia.com: 73114
http://www.youtube.com: 651598
https://intranet.ucll.be/: 24576
http://www.euri.com: 268222
Elapsed time: 4478
```

Als we dit een paar keer uitvoeren zien we dat de tijd wisselt tussen de 2000 en 5000ms. Er is echter nog een belangrijke zaak die opvalt. Tijdens het uitvoeren van onze synchrone methode kunnen we niet interacteren met de GUI. We kunnen ons scherm niet verplaatsen, vergroten of verkleinen. Pas nadat onze methode uitgevoerd is, kunnen we terug andere acties doen. Onze eerste stap is dus de GUI vrijgeven, terwijl onze methode verder blijft lopen.


6.1.2 GUI vrijgeven

We kunnen onze download methode laten uitvoeren als een Task. Dit ziet er als volgt uit:

```
await Task.Run(() => _downloadService.DownloadWebsite(website))
```

De Task.Run is een statische methode op de Task class. Deze neemt een delegate als parameter aan. Een delegate kunnen zien als een functie die doorgegeven wordt als parameter. Het return type van deze Task.Run is het resultaat van de doorgegeven methode, gewrapt in een Task object. In dit geval zal dit dus Task<string> worden.

Task<string> kan je lezen als een **belofte**. Er wordt beloofd dat er een string object opgeleverd zal worden, nadat de methode succesvol uitgevoerd is. Deze belofte wordt pas voldaan wanneer we het nieuwe **await** keyword gebruiken. Dit zal ervoor zorgen dat de belofte Task<string> omgezet wordt naar een bruikbaar string object. We wachten dus tot de belofte voltooid is, en als dit succesvol gedaan is, zullen we een string object terugkrijgen. We zien dat we echter nog steeds een error krijgen:

 (awaitable) `class System.Threading.Tasks.Task`
Represents an asynchronous operation.

CS4033: The 'await' operator can only be used within an async method. Consider marking this method with the 'async' modifier and changing its return type to 'Task'.

[Show potential fixes](#) (Alt+Enter or Ctrl+;)

We moeten onze methode nog markeren als async. Dit duidt aan dat onze methode asynchroon loopt. Ons event zal er dus als volgt uitzien:

```
private async void BtnSync2_Click(object sender, EventArgs e)
{
    var watch = new Stopwatch();
    watch.Start();
    tbOutput.Text = "";
    var sb = new StringBuilder();
    foreach (var website in _websites)
    {
        sb.AppendLine($"{website}: {await Task.Run(() =>
            DownloadService.DownloadWebsite(website))}");
    }
    watch.Stop();
    sb.AppendLine($"Elapsed time: {watch.ElapsedMilliseconds}");
    tbOutput.Text = sb.ToString();
}
```

We hebben dus de volgende keywords geleerd:

- **Async:** duidt aan dat de methode asynchroon loopt
- **Await:** gaat de belofte van een asynchrone methode laten inlossen. Zal dus wachten op de eigenlijke uitvoering
- **Task<string>:** een belofte van een string object. Wanneer we dit awaiten, zullen we een string object ontvangen. Het type string in deze syntax kan natuurlijk vervangen worden door elk type dat er bestaat/zelf gemaakt is.

Wat gebeurt er moesten we het await keyword vergeten? Stel dat we dan volgende code hebben, in plaats van de code met await:

```
sb.AppendLine($"{website}: {Task.Run(() =>
DownloadService.DownloadWebsite(website))}");
```

```
http://www.stackoverflow.com: System.Threading.Tasks.Task`1[System.String]
http://www.google.com: System.Threading.Tasks.Task`1[System.String]
http://www.wikipedia.com: System.Threading.Tasks.Task`1[System.String]
http://www.youtube.com: System.Threading.Tasks.Task`1[System.String]
https://intranet.ucll.be/: System.Threading.Tasks.Task`1[System.String]
http://www.euri.com: System.Threading.Tasks.Task`1[System.String]
Elapsed time: 22
```

We zien dat we niet langer onze resultaten te zien krijgen, maar we krijgen de ToString versie te zien van een Task<string> object. Gezien we niet gaan awaiten, krijgen we dus enkel de **belofte** van een string object en niet het eigenlijke string object dat we wensen.

We zien ook dat de totale duurtijd verminderd is tot 22ms. Gezien we op geen enkele download wachten, zal onze methode gewoon eindigen wanneer we al de downloads gelanceerd hebben. We wachten dus niet op het resultaat.

Door deze manier te implementeren, zien we dat we nu wél onder GUI kunnen manipuleren. We gaan dus asynchroon werken in de zin dat de DownloadWebsite methode in de achtergrond loopt, terwijl we andere acties kunnen doen in onze GUI.

6.1.3 Parallele acties

We merken echter wel op dat we geen tijds winst maken met deze werking. Dit is omdat we nog steeds elke website één voor één downloaden (we wachten via het await keyword op elke download). Als we dus echt verbeteringen willen doorvoeren, zullen we moeten de verschillende downloads tegelijkertijd starten, en deze in parallel en asynchroon van elkaar laten uitvoeren.

We maken een nieuwe methode aan in onze DownloadService:

```
public static Task<string> DownloadWebsiteAsync(string url)
{
    var client = new HttpClient();
    return client.GetStringAsync(url);
}
```



```
}
```

We maken gebruik van de HttpClient, wat je kan zien als de opvolger van WebClient. HttpClient heeft enkel asynchrone methodes om gegevens te downloaden. We kunnen deze nieuwe DownloadWebsite methode dan gebruiken in ons Program.cs:

```
private async void BtnAsync_Click(object sender, EventArgs e)
{
    var watch = new Stopwatch();
    watch.Start();
    tbOutput.Text = "";
    var sb = new StringBuilder();
    foreach (var website in _websites)
    {
        string websitePage = await
            DownloadService.DownloadWebsiteAsync(website);
        sb.AppendLine($"{website}: {websitePage.Length}");
    }
    watch.Stop();
    sb.AppendLine($"Elapsed time: {watch.ElapsedMilliseconds}");
    tbOutput.Text = sb.ToString();
}
```

Als we dit testen, zien we dat de duurtijd ongeveer even lang is als onze BtnSync2 werking. We gebruiken dus nog geen parallelle calls. De verklaring is eenvoudig, we passen weer ons await keyword toe op elke download. We zullen de sites dus nog steeds één voor één downloaden. We kunnen echter alle tasks tegelijkertijd starten, om dan op ze allemaal te wachten. Dit doen we als volgt:

```
List<Task<string>> tasks = new();
```

We maken een list aan, die onze Task<string> zal bijhouden.

```
foreach (var website in _websites)
{
    tasks.Add(DownloadService.DownloadWebsiteAsync(website));
}
```

We voegen elke download toe aan deze lijst. Onze DownloadWebsiteAsync methode geeft een Task<string> terug, dus we kunnen dit probleemloos doen. Hierna kunnen we op al deze tasks wachten:

```
var results = await Task.WhenAll(tasks);
```

Deze statische methode op Task gaat een lijst teruggeven van objecten teruggeven wanneer al zijn beloften voldaan zijn. We kunnen dan over deze lijst itereren om de lengtes weer te geven van de webpagina's:

```
foreach (var item in results)
{
    sb.AppendLine($"{item.Length}");
}
```

6.1.4 ViewModel toevoegen

We zien echter één groot probleem. We hebben in de laatste lus geen kennis meer van bij welke website de lengte hoort. We zullen dus nog een aanpassing moeten doen om dit functioneel te krijgen. We gaan een ViewModel aanmaken om onze resultaten in weer te geven.

```
public class WebsiteModel
{
    public string WebsiteName { get; set; }
    public int PageLength { get; set; }
}
```

Onze download methode wordt dan:

```
public async static Task<WebsiteModel> DownloadWebsiteAsync(string url)
{
    var client = new HttpClient();
    var page = await client.GetStringAsync(url);
    return new WebsiteModel
    {
        WebsiteName = url,
        PageLength = page.Length
    };
}
```

```
}
```

Het await keyword zal er dus voor zorgen dat we ons ViewModel enkel gaan teruggeven als de task van het downloaden van de webpagina voltooid is. We zien dat het returntype van de methode Task<WebsiteModel> is. We zien echter ook dat we een WebsiteModel returnen. De reden dat dit gewrapped wordt in een task is de aanwezigheid van het async keyword. Elke methode dat aangeduid is async zal een Task teruggeven.

Onze code in Program.cs wordt dan:

```
private async void BtnAsync_Click(object sender, EventArgs e)
{
    var watch = new Stopwatch();
    watch.Start();
    tbOutput.Text = "";
    List<Task<WebsiteModel>> tasks = new();
    var sb = new StringBuilder();
    foreach (var website in _websites)
    {
        tasks.Add(DownloadService.DownloadWebsiteAsync(website));
    }

    var results = await Task.WhenAll(tasks);
    foreach (var item in results)
    {
        sb.AppendLine($"{item.WebsiteName} - {item.PageLength}");
    }
    watch.Stop();
    sb.AppendLine($"Elapsed time: {watch.ElapsedMilliseconds}");
    tbOutput.Text = sb.ToString();
}
```

Deze code zal tussen de 1000 en 2500ms lopen, een winst van rond de 1000ms. Dit is een volledig asynchrone methode, we starten de aanvragen en zullen de resultaten pas weergeven wanneer ze succesvol gedownload zijn.

6.2 Async-await pattern

Wat we in de vorige praktische oefening gezien hebben, heet het **async/await pattern**. Dit pattern gaat 3 zaken achter de schermen voor ons als ontwikkelaar afhandelen:

- Methodes geven geen object terug, maar een **belofte** van een object
- Object wordt pas **geïnitieerd** wanneer de belofte **voldaan** wordt
- **Nieuwe syntax** om dit zo transparant mogelijk aan te duiden

Deze nieuwe syntax bestaat dus uit onderstaande elementen:

- **Async**: nieuw **keyword** om aan te duiden dat de methode asynchroon uitgevoerd zal worden. Functies die async zijn zullen steeds een belofte (= Task) teruggeven. Als ontwikkelaar kan je wel gewoon je returntype teruggeven.

Bijvoorbeeld:

```
public async Task<string> GetTextFromClient(Uri uri)
{
    var client = new HttpClient();
    string data = await client.GetStringAsync(uri);
    return data;
}
```

De variabele data is van het type string. We moeten dit niet expliciet als Task<string> teruggeven. De aanwezigheid van het async keyword in de handtekening van de methode zal dit voor ons doen.

- Task<string>: **belofte** van een string object
In deze syntax is string natuurlijk slechts een voorbeeld van een type. Er kan van elk type (voor gedefinieerd of zelfgemaakte) een Task gemaakt worden.

- **Await:** gaat de belofte **voldoen**, er wordt gewacht tot de asynchrone functie klaar is. Een asynchrone methode die een Task<object> teruggeeft waarbij het await keyword gebruikt wordt, zal dus het eigenlijke object retourneren.

Stel dat we een asynchrone methode hebben die niets teruggeeft (void), dan moeten we dit als volgt declareren:

```
public async Task RunBackgroundCopy(Uri source, Uri destination)
{
    //implementation
}
```

In plaats van void zullen deze dus markeren als Task. De methode belooft nog altijd iets uit te voeren, zelfs als krijgen hiervan geen feedback terug.

6.3 Asynchrone taken annuleren

Asynchrone taken worden gelanceerd in de achtergrond. We krijgen pas resultaat als deze taken voltooid zijn en ze een daadwerkelijk resultaat zullen opleveren. Wat gebeurt er echter als we deze taken willen annuleren?

Stel, we starten een download die even zal duren, maar we wensen deze te annuleren. Hoe pakken we dit het best aan?

Binnen het async-await pattern kunnen we gebruik maken van **CancellationTokens**. Dit is een class die binnen de System.Threading namespace beschikbaar is.

De theorie is als volgt:

- Er wordt een CancellationTokenSource object aangemaakt, dit object is verantwoordelijk voor het aanmaken van verschillende tokens.
- Deze token worden meegegeven aan asynchrone methodes.
- Wanneer we een cancel oproepen op onze CancellationTokenSource, zullen de tokens in de asynchrone methodes een exception throwen.

We gaan het voorbeeld van onze praktische uitwerking aanpassen, zodat we deze werking kunnen demonstreren.

We gaan onze download code naar een aparte methode refactoren:

```
private async void BtnAsync_Click(object sender, EventArgs e)
{
    await DownloadAllWebsitesAsync();
}
```

Dit laat ons toe om de methode aan te passen, zonder ons zorgen te maken over het aanroepende event. Onze volgende stap is dan variabele aan te maken voor onze CancellationTokensource. Dit is het object dat gaat bepalen of onze taken geannuleerd zullen worden al dan niet:

```
private readonly CancellationTokensource _tokenSource = new();
```

We passen dan onze download methode aan, zodat we het token van onze token source meegeven:

```
await DownloadAllWebsitesAsync(_tokenSource.Token);
```

DownloadAllWebsitesAsync geeft dan op zijn beurt dit token door aan de eigenlijke async methode:

```
tasks.Add(DownloadService.DownloadWebsiteAsync(website,
cancellationTokens));
```

In deze methode kunnen we dit token dan gaan gebruiken. We hebben twee manieren:

- Een exception gooien wanneer er een annulering aangevraagd wordt
- De code enkel uitvoeren wanneer er **geen** annulering aangevraagd wordt

De code voor de eerste manier ziet er als volgt uit:

```
public async static Task<WebsiteModel> DownloadWebsiteAsync(string url,
CancellationTokens cancellationTokens)
{
    var client = new HttpClient();
```

```
var page = await client.GetStringAsync(url);
cancellationToken.ThrowIfCancellationRequested();

return new WebsiteModel
{
    WebsiteName = url,
    PageLength = page.Length
};
}
```

Nadat we de `await` aangeroepen hebben, voegen we een extra regel toe. Wanneer het token vanuit zijn `TokenSource` het signaal krijgt dat er een annulering aanvraag is, zal deze een exception smijten. De locatie van dit statement is zeer belangrijk. Als we dit bijvoorbeeld voor de `await` zouden plaatsen, dan zou de methode nooit geannuleerd kunnen worden, omdat er quasi geen tijd zit een `HttpClient` aanmaken en de `GetStringAsync` aanvraag te starten.

De tweede manier kan je als volgt implementeren:

```
public async static Task<WebsiteModel> DownloadWebsiteAsync2(string url,
CancellationTokn cancellationToken)
{
    if (!cancellationToken.IsCancellationRequested)
    {
        var client = new HttpClient();
        var page = await client.GetStringAsync(url);

        return new WebsiteModel
        {
            WebsiteName = url,
            PageLength = page.Length
        };
    }
    return null;
}
```

Het nadeel hier is dat je een null object moet retourneren, terwijl eigenlijk de ganse task geannuleerd is. In praktijk kiezen we vaak voor de eerste oplossing wanneer er data teruggestuurd wordt, en voor de tweede als het over een methode gaat zonder returntype.

Om onze annulering dan uit te laten voeren door de gebruiken kunnen we een nieuwe knop toevoegen. De code achter deze knop zit er dan zo uit:

```
private void BtnCancel_Click(object sender, EventArgs e)
{
    _tokenSource.Cancel();
}
```

De TokenSource gaat een cancel request doorgeven naar zijn token, waarop die ofwel de exception gaat gooien, ofwel zijn IsCancellationRequested property op true zal zetten.

7 Robuust programmeren

In de vorige hoofdstukken hebben we geleerd hoe je je code op een mooie manier kan opbouwen, zodat deze makkelijk te onderhouden en uit te breiden is. We hebben ook gezien hoe je je code kan testen, zodat we weten dat onze code doet wat hij moet doen.

In dit laatste hoofdstuk gaan we kijken wat er gebeurt als onze code dan toch faalt. Hoe gaan we om met validaties, exceptions en hoe zorgen we ervoor dat als er iets fout gaat, we weten wat er fout gegaan is.

7.1 Validaties

We bekijken als eerste validaties. Dit zijn controles die we kunnen toevoegen op onze modellen, zodat we de ingegeven data kunnen aftoetsen. Deze validaties kunnen gaan van ingebouwde controles, tot eigen custom validaties.

7.1.1 Standaard validaties

Doordat we met MVC werken zijn validaties makkelijk te voegen. We kunnen dit namelijk rechtstreeks op onze modellen toevoegen, met behulp van data annotations. Data annotations zijn stukken code, die extra informatie geven over een methode/getter. We hebben deze notatie al gebruikt toen we actions van een controller aanduiden als post of get.



We gebruiken de solution **Uccl.OOD.Validations** om dit aan te tonen.

Dit project is een ASP.NET Core MVC project, met één model: Book. Het heeft hiervan een repository (zonder achterliggende databank). We gaan de create en edit actions gebruiken om onze validaties voor te doen.

Ons model is vrij eenvoudig:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
```

```
public string ISBN13 { get; set; }
public string ISBN10 { get; set; }
public string Author { get; set; }
public int Releaseyear { get; set; }
}
```

We zien, als we een create doen zonder velden in te vullen, dat we direct validatie fouten krijgen:

Create

Book

Name

The Name field is required.

ISBN13

The ISBN13 field is required.

ISBN10

The ISBN10 field is required.

Author

The Author field is required.

Releaseyear

The Releaseyear field is required.

Create

Hoe komt dit? Er is impliciet een Required attribuut toegevoegd, doordat de velden van ons model niet nullable zijn. Als we ons model nullable maken:

```

public class Book
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public string? ISBN13 { get; set; }
    public string? ISBN10 { get; set; }
    public string? Author { get; set; }
    public int? Releaseyear { get; set; }
}

```

Zien we dat deze validatie verdwijnt. .NET heeft er dus voor gezorgd dat onze properties impliciet gemarkeerd werden met het [Required] attribuut. Als we dit terug manueel toevoegen, zullen we terug dezelfde foutboodschappen krijgen:

```

public class Book
{
    public int Id { get; set; }
    [Required]
    public string? Name { get; set; }
    [Required]
    public string? ISBN13 { get; set; }
    [Required]
    public string? ISBN10 { get; set; }
    [Required]
    public string? Author { get; set; }
    [Required]
    public int? Releaseyear { get; set; }
}

```

We kunnen dus gebruik maken van ingebouwde data annotations om ons model te valideren. .NET biedt ons enkele standaarden aan die we kunnen gebruiken. Deze zijn terug te vinden onder <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-6.0>.

We kunnen bijvoorbeeld de StringLength annotation gebruiken om een eerste controle te doen van de ISBN-waardes:

```
[Required]
[StringLength(13)]
public string? ISBN13 { get; set; }
[Required]
[StringLength(10)]
```

Het interessante is wanneer we nu een bestaand boek editen. We kunnen bijvoorbeeld de naam weghalen, en dan krijgen we volgende fout:

Name

The Name field is required.

We zien dus dat we enkele basisprincipes van OOD zoals DRY en SRP hanteren, door de modelvalidatie op één plek te houden. We moeten niet in elke view deze controles opnieuw doen, maar we hebben ze gecentraliseerd op één locatie.

7.1.2 Custom validaties

We zouden onze controles verder kunnen verfijnen. Zo wensen we bijvoorbeeld dat het ingegeven jaar ligt tussen 1800 en het jaar van de huidige datum. Hiervoor zullen we dus een eigen validatie moeten aanmaken.

Gezien we dit mogelijks gaan hergebruiken voor andere modellen, gaan we hier ook een attribuut van maken. Dit doen we door een class te maken die overerft van `ValidationAttribute`. Elke class die hiervan overerft geeft de suffix "Attribute". Dit doen we zodat het direct duidelijk is dat we deze class als attribute kunnen gebruiken.

```
public class YearAttribute : ValidationAttribute
{
}
}
```

Natuurlijk moeten we nog een eigenlijke validatie toevoegen. `ValidationAttribute` heeft hiervoor twee methodes die we kunnen overriden:

- IsValid
- FormatErrorMessage

Via de IsValid methode zullen we kunnen aangeven of een object dat we binnenkrijgen een geldige waarde is voor dit attribute. Via FormatErrorMessage kunnen we dan een standaard foutboodschap aanleveren als deze waarde niet geldig is. Een uitgewerkte class zou er dan zo kunnen uitzien:

```
public class YearAttribute : ValidationAttribute
{
    public override bool IsValid(object? value)
    {
        var year = value as int?;
        if (year == null) return false;

        return year >= 1800 && year <= DateTime.Now.Year;
    }

    public override string FormatErrorMessage(string name)
    {
        return $"{name} should be between 1800 and {DateTime.Now.Year}";
    }
}
```

We decoreren ons ReleaseYear property dan met dit nieuw attribute:

```
[Required]
[Year]
public int? Releaseyear { get; set; }
```

En we zien onze nieuwe foutboodschap wanneer we een foute waarde ingeven:

Releaseyear

Releaseyear should be between 1800 and 2022

Create

Als we echter terug naar de hoofdpagina gaan, zien we dat deze record toch is toegevoegd:

Index

[Create New](#)

Id	Name	ISBN13	ISBN10	Author	Releaseyear	
1	The Clean Coder	978-0137081073	0137081073	Robert Martin	2011	Edit
2	The Mythical Man-Month	978-0201835953	9780201835953	Frederick Brooks Jr.	1995	Edit
0	test	test	test	test	1000	Edit

Dit is natuurlijk niet de bedoeling. We zullen dus nog een extra controle moeten toevoegen aan onze controller, zodat foutieve modellen niet worden toegevoegd aan onze lijst. Dit kunnen we redelijk eenvoudig. MVC levert een functie hiervoor aan, namelijk `ModelState.IsValid`. Dit is een ingebouwde property, die het aangeleverde model controleert voor ons. Als we deze dus in onze controller toevoegen, kunnen we hierdoor het onderscheid maken tussen correcte en foutieve informatie:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Book book)
{
    if (ModelState.IsValid)
    {
        BookRepository.Instance.AddBook(book);
        return RedirectToAction("Index");
    }
    return View();
}
```

Als we hierna terug onze foute waardes wensen toe te voegen, zien we dat we tegengehouden worden door onze nieuwe controle. De reden trouwens dat we deze foutmelding zien, is de volgende lijn code in onze view:

```
<span asp-validation-for="Releaseyear" class="text-danger"></span>
```

Onze validatie zal zijn `FormatErrorMessage` waarde in dit span element weergeven.

7.1.3 Validaties over meerdere velden

Tussen een ISBN10 en een ISBN13 code zit een relatie. Voor het simpel te maken gaan we er van uit dat we om een ISBN13 code aan te maken we aan onze ISBN10 code een prefix 978 moeten toevoegen (opmerking: in werkelijk is dit een complexere materie, met afwijkingen per land, we vereenvoudigen dit louter als voorbeeld). We willen dus controleren dat als iemand een ISBN10 en een ISBN13 code ingeeft dat deze conversie mogelijk is.

De werking hiervoor is grotendeels hetzelfde als bij een single validation. We gaan weer een attribute aanmaken dat overerft van ValidationAttribute. In plaats van de IsValid methode die een bool teruggeeft te overriden, gaan we nu de ValidationResult versie overriden. Deze ziet er als volgt uit:

```
public class ISBNAttribute : ValidationAttribute
{
    protected override ValidationResult? IsValid(
        object? value,
        ValidationContext validationContext)
    {
        return base.IsValid(value, validationContext);
    }
}
```

Het voordeel van deze methode is dat we de ValidationContext ook meekrijgen. In deze context zit namelijk de rest van het model ook. We kunnen dit dus gebruiken om andere velden samen te controleren:

```
l override ValidationResult? IsValid(object? value, ValidationContext validationContext)
{
    validationContext {System.ComponentModel.DataAnnotations.ValidationContext}
    DisplayName View "ISBN13"
    Items Count = 0
    MemberName View "ISBN13"
    ObjectInstance {Ucll.OOD.Validations.Models.Book}
    Author View "Book's author" Book FullName = "Ucll.OOD.Validations.Models.Book"
    ISBN10 View "0123456789"
    ISBN13 View "1234567891230"
    Id 0
    Name View "Title of the book"
    Releaseyear 1995
}
```

We zouden dit dus als volgt kunnen invullen:

```

public class ISBNAttribute : ValidationAttribute
{
    protected override ValidationResult? IsValid(object? value,
ValidationContext validationContext)
    {
        var book = validationContext.ObjectInstance as Book;
        if (book == null) return null;

        if(book.ISBN13.Equals($"978{book.ISBN10}"),
StringComparison.InvariantCultureIgnoreCase))
        {
            return ValidationResult.Success;
        }

        return new ValidationResult("ISBN10 and ISBN13 do not match");
    }
}

```

Als we geen validatie errors hebben, geven we een ValidationResult.Success. In het andere geval geven we een foutboodschap:

ISBN13

ISBN10 and ISBN13 do not match

ISBN10

ISBN10 and ISBN13 do not match

Als de velden wel correct ingevuld zijn, krijgen we ons nieuwe book te zien in de overzicht lijst:

Id	Name	ISBN13	ISBN10	Author	Releaseyear	
1	The Clean Coder	9780137081073	0137081073	Robert Martin	2011	Edit
2	The Mythical Man-Month	9780201835953	0201835953	Frederick Brooks Jr.	1995	Edit
0	Title of the book	9780123456789	0123456789	Book's author	1995	Edit

7.2 Exceptions

Exceptions zijn fouten die ons programma smijt wanneer er onverwachte zaken gebeuren. Denk hierbij aan een databank die niet beschikbaar is, een fileserver die (tijdelijk) down is, of andere uitzonderlijke fouten. Exceptions geven vaak heel veel informatie over wat er precies gebeurd is. In ASP.NET Core MVC kunnen we makkelijk een exception creëren, door de Privacy view te deleten. Als we dan in onze applicatie naar deze view willen navigeren, krijgen we volgende fout:

An unhandled exception occurred while processing the request.

```
InvalidOperationException: The view 'Privacy' was not found. The following locations were searched:
/Views/Home/Privacy.cshtml
/Views/Shared/Privacy.cshtml
Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful(IEnumerable<string> originalLocations)
```

Stack Query Cookies Headers Routing

```
InvalidOperationException: The view 'Privacy' was not found. The following locations were searched: /Views/Home/Privacy.cshtml /Views/Shared/Privacy.cshtml
Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful(IEnumerable<string> originalLocations)
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewResultExecutor.ExecuteAsync(ActionContext context, ViewResult result)
Microsoft.AspNetCore.Mvc.ViewResult.ExecuteResultAsync(ActionContext context)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResultFilterAsync>g__Awaited|30_0<TFilter, TFilterAsync>(ResourceInvoker invoker, Task lastTask, State next, Scope scope, object state, bool isCompleted)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResultExecutedContextSealed context)
```

Het detail van fouten dat we hier zien noemen de stacktrace. We krijgen informatie van wat er precies misloopt en waar deze fout opgetreden is. Zeer handig voor ons als programmeur, maar een eindgebruiker is hier natuurlijk niets mee. We gaan dus een onderscheid moeten maken in hoe we exceptions weergeven. We hebben de functionele fout, die getoond wordt aan de gebruiker en de technische fout, die de programmeur nodig heeft om het euvel te corrigeren.

We gaan werken met een eigen API om te kijken wat er gebeurt als deze down gaat (gezien we dit makkelijk zelf kunnen controleren).



De code dat we hier demonstreren vindt u onder onder solution **Ucll.OOD.Exceptions**. Deze solution bevat twee projecten:

- **Ucll.OOD.Exceptions** – een ASP.NET Core Web App MVC project
- **MyExampleAPI** - ASP.NET Core Web API project

Ons MVC-project gaat een service hebben die de API aanspreekt. Wanneer onze API bereikbaar is, krijgen we volgende resultaat:

Index

[Create New](#)

Date	TemperatureC	TemperatureF	Summary
30/05/2022 16:56:53	-13	9	Sweltering
31/05/2022 16:56:53	39	102	Hot
01/06/2022 16:56:53	22	71	Bracing
02/06/2022 16:56:53	52	125	Cool
03/06/2022 16:56:53	12	53	Sweltering

(Het WeatherForecast voorbeeld is voorbeeld code die automatisch aangemaakt wordt bij het aanmaken van een API-project)

Als dit echter niet het geval is, krijgen we een exception:

An unhandled exception occurred while processing the request.

SocketException: No connection could be made because the target machine actively refused it.

System.Net.Sockets.Socket+AwaitableSocketAsyncEventArgs.ThrowException(SocketError error, CancellationToken cancellationToken)

HttpRequestException: No connection could be made because the target machine actively refused it. (localhost:7067)

System.Net.Http.HttpConnectionPool.ConnectToTcpHostAsync(string host, int port, HttpRequestMessage initialRequest, bool async, CancellationToken cancellationToken)

[Stack](#) [Query](#) [Cookies](#) [Headers](#) [Routing](#)

SocketException: No connection could be made because the target machine actively refused it.

System.Net.Sockets.Socket+AwaitableSocketAsyncEventArgs.ThrowException(SocketError error, CancellationToken cancellationToken)

System.Net.Sockets.Socket+AwaitableSocketAsyncEventArgs.System.Threading.Tasks.Sources.IValueTaskSource.GetResult(short token)

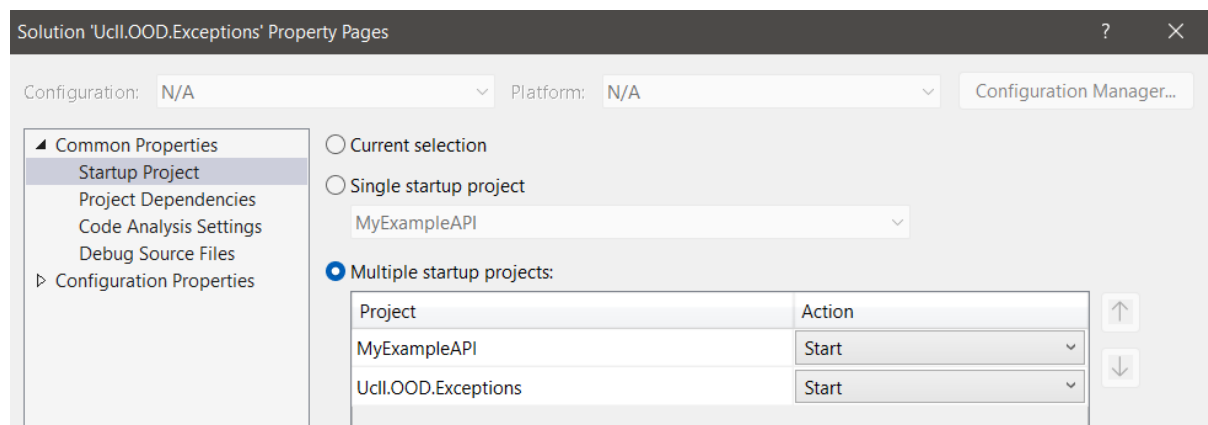
System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable+ConfiguredValueTaskAwaiter.GetResult()

System.Net.Sockets.Socket.<ConnectAsync>g__WaitForConnectWithCancellation|277_0(AwaitableSocketAsyncEventArgs saea, ValueTask connectTask, CancellationToken cancellationToken)

System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable+ConfiguredValueTaskAwaiter.GetResult()

System.Net.Http.HttpConnectionPool.ConnectToTcpHostAsync(string host, int port, HttpRequestMessage initialRequest, bool async, CancellationToken cancellationToken)

We kunnen beide projecten tegelijkertijd opstarten via de solution properties:



Als we dan in de Program.cs van ons api-project een Thread.Sleep toevoegen, kunnen wij zien hoe het programma reageert op een tijdelijke uitval. Doordat er een 10 seconden verschil is, zien we eerst het gedrag wanneer onze api niet beschikbaar is. Zodra dat ons api-project opgestart is, kunnen we onze index van het andere project vernieuwen en zien we de correcte resultaten.

We bespreken het feit dat de api er soms heel tijdelijk uitligt met business. Voor hen maakt het niet uit dat de site er langer over doet om de resultaten te laden. We zullen dus enkele keren proberen om de resultaten op te halen, en pas wanneer dit na 20 seconden niet lukt, zullen we een foutmelding geven. We gaan dus onze ApiService aanpassen om deze functionaliteit toe te voegen:

```
public async Task<IEnumerable<WeatherForecast>>
GetWeatherForecastAsync()
{
    HttpClient client = new HttpClient();
    HttpResponseMessage response;

    while (true)
    {
        try
        {
            response = await
client.GetAsync("https://localhost:7067/WeatherForecast");
            break; // success!
        }
        catch
        {
            if (tries == 20)
            {
                throw;
            }
            Thread.Sleep(1000);
        }
    }
}
```

```
    }

    response.EnsureSuccessStatusCode();
    return await
response.Content.ReadFromJsonAsync<IEnumerable<WeatherForecast>>();
}
```

We gaan dus lopen zo lang we een exception krijgen. Als we echter 20x een exception hebben gekregen (wat overeenkomt met ~20 seconden wachten) gooien we alsnog een exception. Als we nu onze twee projecten runnen, zien we dat ons hoofdproject er langer over doet om de pagina te laden, maar er uiteindelijk wel in slaagt om de resultaten te tonen. We wachten dus totdat onze api beschikbaar is, en we tonen in de tussentijd geen fouten. Als de api nog steeds niet beschikbaar is na de wachttijd, dan krijgen we de gekende fout pagina, met al zijn details.

7.2.1 Environments

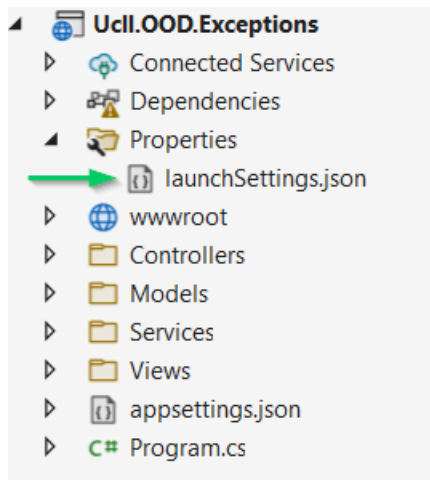
We willen echter een andere afhandeling per environments. Met environment bedoelen we de omgeving waarin onze applicatie draait. In development wensen we ander gedrag te zien dan in productie. In development wensen we de volledige details te zien van onze fouten. In productie lossen we dit best subtieler op. We willen namelijk te veel details geven aan mogelijke hackers of mensen met foute bedoelingen.

Je kan dit vergelijken met een login scherm. Als je ofwel een foute username of een fout password ingeeft, krijg je altijd de melding “username or password foutief”. Dit is om geen overbodige informatie aan te leveren aan hackers. Als ze bijvoorbeeld weten dat de user bestaat, kunnen ze verder proberen met deze user binnen je applicatie te raken.

Voor exceptions geldt hetzelfde. Exceptions doen je applicatie crashen, waardoor hackers informatie kunnen verzamelen met betrekking tot de werking van de applicatie.

We gaan dus onze applicatie zo instellen dat we andere details zien in de verschillende environments.

In een ASP.NET Core MVC Web App zien we deze verschillende environments in de launchSettings.json. Deze is terug te vinden onder de properties map in de solution:



Hierin hebben we twee profiles:

```
"profiles": {
  "Ucll.OOD.Exceptions": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "applicationUrl": "https://localhost:7007;http://localhost:5007",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
  "IIS Express": {
    "commandName": "IISExpress",
    "launchBrowser": true,
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  }
}
```

Het is de lijn

```
"ASPNETCORE_ENVIRONMENT": "Development"
```

dat de eigenlijke enviroment bepaald. Standaard staat dit op Development. Als we onze Ucll.OOD.Exceptions profiel veranderen naar Production, zien we een andere error.

Als dit veld de waarde Development heeft, zien we:

An unhandled exception occurred while processing the request.

SocketException: No connection could be made because the target machine actively refused it.

System.Net.Sockets.Socket+AwaitableSocketAsyncEventArgs.ThrowException(SocketError error, CancellationToken cancellationToken)

HttpRequestException: No connection could be made because the target machine actively refused it. (localhost:7067)

System.Net.Http.HttpConnectionPool.ConnectToTcpHostAsync(string host, int port, HttpRequestMessage initialRequest, bool async, CancellationToken cancellationToken)

[Stack](#) [Query](#) [Cookies](#) [Headers](#) [Routing](#)

SocketException: No connection could be made because the target machine actively refused it.

System.Net.Sockets.Socket+AwaitableSocketAsyncEventArgs.ThrowException(SocketError error, CancellationToken cancellationToken)

System.Net.Sockets.Socket+AwaitableSocketAsyncEventArgs.System.Threading.Tasks.Sources.IValueTaskSource.GetResult(short token)

System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable+ConfiguredValueTaskAwaiter.GetResult()

System.Net.Sockets.Socket.<ConnectAsync>g__WaitForConnectWithCancellation|277_0(AwaitableSocketAsyncEventArgs saea, ValueTask connectTask, CancellationToken cancellationToken)

System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable+ConfiguredValueTaskAwaiter.GetResult()

System.Net.Http.HttpConnectionPool.ConnectToTcpHostAsync(string host, int port, HttpRequestMessage initialRequest, bool async, CancellationToken cancellationToken)

[Show raw exception details](#)

HttpRequestException: No connection could be made because the target machine actively refused it. (localhost:7067)

System.Net.Http.HttpConnectionPool.ConnectToTcpHostAsync(string host, int port, HttpRequestMessage initialRequest, bool async, CancellationToken cancellationToken)

System.Threading.Tasks.ValueTask<TResult>.get_Result()

System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>+ConfiguredValueTaskAwaiter.GetResult()

System.Net.Http.HttpConnectionPool.ConnectAsync(HttpRequestMessage request, bool async, CancellationToken cancellationToken)

Als dit veld de waarde Production heeft, zien we:

[UcIl.OOD.Exceptions](#) [Home](#) [Privacy](#)

Error.

An error occurred while processing your request.

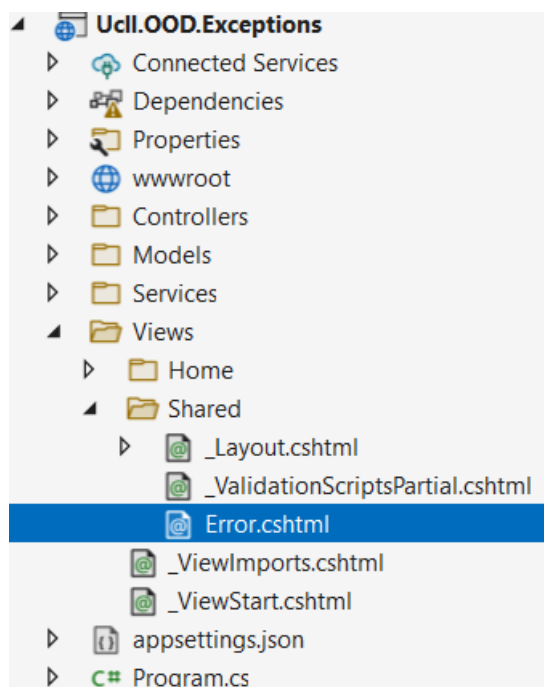
Request ID: 00-33494a349c20bd96a5370439c72ceacd-38eff0232c5a1805-00

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The **Development** environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

Deze laatste error is de View die gedefinieerd is onder de Shared views:



We kunnen deze dus aanpassen naar onze eigen wensen. Hoe komt het dan dat deze view geladen wordt? In Program.cs zit volgende code:

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
```

Als onze environment dus verschilt van Development, zullen we bij exceptions naar de Home/Error view geredirect worden. Zo kunnen we dus een standaardfout pagina maken, die aan onze gebruikers uitlegt wat er gebeurd is. We kunnen deze view dan aanpassen zodat deze mooier is naar de eindgebruikers. De eigenlijke fout zullen we dan verbergen, en via logging opslaan voor onderzoek door de ontwikkelaar.

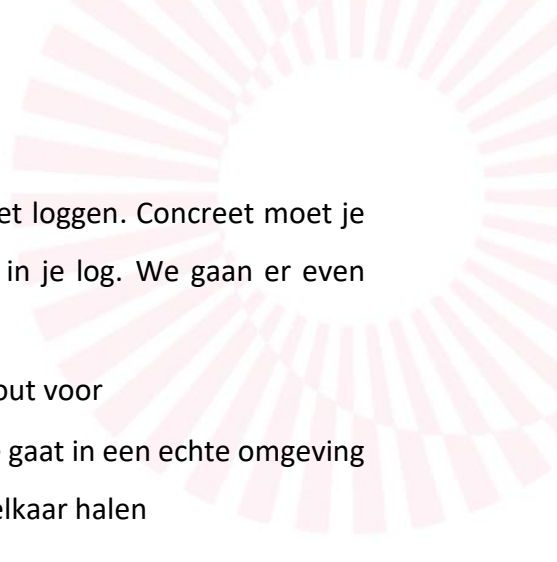
7.3 Logging

Logging dient ervoor om ons programma op lange termijn onderhoudbaar te houden. Als er zich exceptions voordoen, is het belangrijk dat we loggen, zodat we de oorzaak van deze problemen kunnen onderzoeken. Stel bijvoorbeeld dat één van onze gebruikte webservices vaak onbereikbaar is, dan zouden we dit via logging kunnen zien. We zouden dan bijvoorbeeld kunnen opmerken dat deze download steeds rond een bepaal tijdstip valt, en deze informatie kunnen we dan gebruiken om verder onderzoek te leveren.

Logs worden vaak weggeschreven naar tekstbestanden. Dit betekent dat hier natuurlijk ook limieten aan zitten. We willen niet dat we logbestanden krijgen van ettelijke gigabytes, want dan zijn deze bestanden niet echt meer leesbaar. We zullen dus een systeem moeten hebben dat onze logfiles zelf opsplijst wanneer deze te groot worden.

Gezien logs gelezen moeten worden (door ofwel ontwikkelaars, ofwel computers), zullen je logs steeds hetzelfde formaat nodig hebben. Een loglijn kan er dan bijvoorbeeld zo uitzien:

```
2022-06-01 12:54:02/MyApplication/Error/This is my error
message/Stacktrace.../
```



We zien hierin enkele terugkerende elementen die je zeker moet loggen. Concreet moet je ervoor zorgen dat de wie, wat en wanneer zeker duidelijk zijn in je log. We gaan er even doorheen:

- 2022-06-01 12:54:02: datum en tijd, wanneer kwam de fout voor
- Naam van de applicatie: welke applicatie geeft de fout. Je gaat in een echte omgeving meerdere applicaties hebben, dus je wilt deze niet door elkaar halen
- Loglevel: komen we zo dadelijk op terug
- Leesbare foutboodschap: een duidelijke omschrijving van de fout
- Stacktrace van de fout: via de stacktrace kunnen we te weten komen op welke locatie in de code de fout zich voordeed

7.3.1 Loglevel

Het logleven dient ervoor om te kijken hoe belangrijk de loglijn. Als we deze sorteren van onbelangrijk naar kritisch, zijn de levels als volgt:

- Trace
- Debug
- Info
- Warn
- Error
- Fatal

Het **Trace** level wordt enkel gebruikt in development. Het zijn loglijnen die je kunnen helpen met de ontwikkeling van het programma. Je kan bijvoorbeeld tussenstatussen loggen op dit Trace niveau. Trace wordt standaard genegeerd als men code deployed in release modus.

Debug is een gelijkaardig niveau, dat best ook vermeden wordt als je je code in productie zet. Het is een niveau om te gebruiken wanneer we deployen naar onze test omgeving, zodat we ook daar bepaalde zaken kunnen in opvolgen.

Info dient dan voor informatieve loglijnen aan te duiden. Je kan bijvoorbeeld wensen te weten dat iemand is ingelogd, of andere informatieve zaken wensen te monitoreren.

Vanaf **Warn** krijgen we dan met fouten te maken. Warn is een level dat gebruikt wordt om aan te duiden dat er iets mis aan het lopen is, maar dat de algemene werking van het programma niet in gedrang komt. Concreet ga je dit niveau niet vaak tegenkomen in het echt, omdat de meeste fouten toch een level hoger liggen.

Error is het loglevel dat we gebruiken om standaard exceptions in te noteren. Dit is het standaard level om fouten in te noteren die de werking van ons programma in gedrang brengen.

Fatal is dan het hoogste level. Wanneer er een fatal error optreedt (en we deze dus loggen) zal er niets van ons programma meer werken.

Vaak gaan we logs wegschrijven naar bestanden. Om deze bestanden niet nodeloos te laten aangroeien met ongebruikte informatie, zullen we selecties maken welke loglevels we daadwerkelijk te zien krijgen. In een productie omgeving gaan we bijvoorbeeld zelden logslevel zien onder Warn en al zeker geen loglevels onder Information. Dit zou onze logs nodeloos vergroten.

Sommige frameworks hanteren andere namen voor deze verschillende niveaus. In grote lijnen zijn dit echter de verschillende niveaus die je beschikbaar hebt om naar te loggen.

7.3.2 Praktisch voorbeeld



De code dat we hier demonstreren vindt u onder onder solution **Ucll.OOD.Logging**

Als we een nieuw ASP.NET CORE MVC project aanmaken, kunnen het gedrag van logging wat nader onderzoeken. Na het opstarten krijgen we twee dingen te zien:

- Een webpagina waarin onze applicatie getoond wordt
- Een console scherm waarin we onze log zien

Het is het console scherm waar we onze aandacht gaan op vestigen:

```
C:\Users\wivan\UC Leuven-Limburg\Graduaat Programmeren - Object Oriented Design\Lessen\Les 6 - Robuust programmeren\Ucll.O...
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: https://localhost:7267
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: http://localhost:5267
info: Microsoft.Hosting.Lifetime[0]
  Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
  Content root path: C:\Users\wivan\UC Leuven-Limburg\Graduaat Programmeren - Object Oriented Design\Lessen\Les 6 - Robuust programmeren\Ucll.OOD.Logging\Ucll.OOD.Logging\
```

We zien enkele loglijnen met als level *info*.

Als we in onze controller een exception throwen, zien we dit ook in onze log:

```
public HomeController(ILogger<HomeController> logger)
{
    throw new Exception("Exception was thrown");
    _logger = logger;
}
```

```
C:\Users\wivan\UC Leuven-Limburg\Graduaat Programmeren - Object Oriented Design\Lessen\Les 6 - Robuust programmeren\Ucll.O...
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: https://localhost:7267
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: http://localhost:5267
info: Microsoft.Hosting.Lifetime[0]
  Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
  Content root path: C:\Users\wivan\UC Leuven-Limburg\Graduaat Programmeren - Object Oriented Design\Lessen\Les 6 - Robuust programmeren\Ucll.OOD.Logging\Ucll.OOD.Logging\
[err]: Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware[1]
  An unhandled exception has occurred while executing the request.
  System.Exception: Exception was thrown
    at Ucll.OOD.Logging.Controllers.HomeController..ctor(ILogger<1> logger) in C:\Users\wivan\UC Leuven-Limburg\Graduaat Programmeren - Object Oriented Design\Lessen\Les 6 - Robuust programmeren\Ucll.OOD.Logging\Ucll.OOD.Logging\Controllers\HomeController.cs:line 13
    at lambda_method2(Closure, IServiceProvider, Object[])
    at Microsoft.AspNetCore.Mvc.Controllers.ControllerActivatorProvider.<>c__DisplayClass7_0.<CreateActivator>b__0(ControllerContext controllerContext)
    at Microsoft.AspNetCore.Mvc.Controllers.ControllerFactoryProvider.<>c__DisplayClass6_0.<CreateControllerFactory>b__0(CreateControllerContext controllerContext)
    at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)
    at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()
    --- End of stack trace from previous location ---
    at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResourceFilter>b__Awaited25_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope, Object state, Boolean isCompleted)
    at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResourceExecutedContextSealed context)
    at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Next(State& next, Scope& scope, Object& state, Boolean isCompleted)
```

We kunnen ook echter zelf controleren wat we loggen, en op welk niveau we dit doen. We krijgen namelijk het object dat verantwoordelijk is voor de logging binnen via Dependency Injection:

```
private readonly ILogger<HomeController> _logger;

public HomeController(ILogger<HomeController> logger)
{
    _logger = logger;
}
```

Dit object kunnen we dan gebruiken om loglijnen toe te voegen:

```
public IActionResult Index()
{
    _logger.LogTrace("Logging a Trace level logline");
    _logger.LogDebug("Logging a Debug level logline");
    _logger.LogInformation("Logging an Information level logline");
    _logger.LogWarning("Logging a Warn level logline");
    _logger.LogError("Logging an Error level logline");
    _logger.LogCritical("Logging a Critical level logline");
    return View();
}
```

Als we dit uitvoeren, zijn we volgende loglines verschijnen in onze console:

```
info: Ucll.00D.Logging.Controllers.HomeController[0]
      Logging an Information level logline
warn: Ucll.00D.Logging.Controllers.HomeController[0]
      Logging a Warn level logline
fail: Ucll.00D.Logging.Controllers.HomeController[0]
      Logging an Error level logline
crit: Ucll.00D.Logging.Controllers.HomeController[0]
      Logging a Critical level logline
```

We merken op dat de Trace en Debug levels niet zichtbaar zijn. Hoe komt dit?

De zichtbare levels worden bepaald door de appsettings.json file. Hierin kunnen we per environment instellen welke level van logs we wensen bij te houden:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```


Als we appsettings.Development.json aanpassen, zodat ons default loglevel *Trace* is, dan zien we wel alle loglijnen:

```
dbug: Microsoft.Extensions.Hosting.Internal.Host[2]
      Hosting started
trce: Ucll.OOD.Logging.Controllers.HomeController[0]
      Logging a Trace level logline
dbug: Ucll.OOD.Logging.Controllers.HomeController[0]
      Logging a Debug level logline
info: Ucll.OOD.Logging.Controllers.HomeController[0]
      Logging an Information level logline
warn: Ucll.OOD.Logging.Controllers.HomeController[0]
      Logging a Warn level logline
fail: Ucll.OOD.Logging.Controllers.HomeController[0]
      Logging an Error level logline
crit: Ucll.OOD.Logging.Controllers.HomeController[0]
      Logging a Critical level logline
```

De dependency injection zelf gebeurt in de lijn

```
var builder = WebApplication.CreateBuilder(args);
```

Dit statement gaat standaard een ILogger object toelaten om geïnjecteerd te worden in de controllers. We kunnen dit zelf nog wat aanpassen. In Program.cs kunnen we expliciet de logger gaan bepalen, waarna deze dan wordt geïnjecteerd:

```
builder.Logging.AddSimpleConsole(options =>
{
    options.SingleLine = true;
    options.ColorBehavior = LoggerColorBehavior.Enabled;
    options.TimestampFormat = "yyyy-MM-dd hh:mm:ss";
});
```

Onze log zal er dan als volgt uitzien:

```
2022-06-22 09:38:30dbug: Microsoft.Extensions.Hosting.Internal.Host[2] Hosting started
2022-06-22 09:38:31trce: Ucll.OOD.Logging.Controllers.HomeController[0] Logging a Trace level logline
2022-06-22 09:38:31dbug: Ucll.OOD.Logging.Controllers.HomeController[0] Logging a Debug level logline
2022-06-22 09:38:31info: Ucll.OOD.Logging.Controllers.HomeController[0] Logging an Information level logline
2022-06-22 09:38:31warn: Ucll.OOD.Logging.Controllers.HomeController[0] Logging a Warn level logline
2022-06-22 09:38:31fail: Ucll.OOD.Logging.Controllers.HomeController[0] Logging an Error level logline
2022-06-22 09:38:31crit: Ucll.OOD.Logging.Controllers.HomeController[0] Logging a Critical level logline
```


Het enige nadeel dat deze ingebouwde logger heeft, is dat we niet kunnen wegschrijven naar bestanden. We kunnen dus geen logfiles gaan bijhouden. Hiervoor hebben we een extra package nodig.

7.3.3 Logging naar file

Waarom wensen we naar een bestand te loggen? Zodat we controles kunnen doen wanneer er iets fout loopt. Als onze log niet opgeslagen wordt, kunnen we deze nooit gebruiken.

Er zijn ook tools zoals Kibana die logfiles kunnen innemen, waarna er makkelijk gefilterd en gezocht kan worden naar een bepaalde fout op een bepaald tijdstip.

Gezien onze standaard logger geen schrijven naar files ondersteund, zullen we een package moeten installeren. Van de vele verschillende mogelijkheden kiezen we voor Serilog. We moeten hiervoor twee packages downloaden vanuit Nuget:

- Serilog.AspNetCore
- Serilog.Extensions.Logging.File

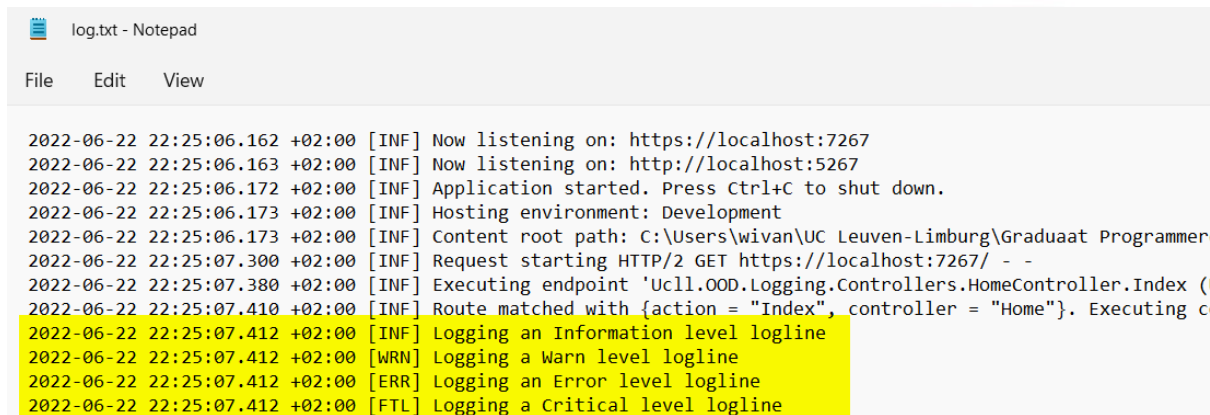
Hierna moeten we onze Program.cs aanpassen, zodat we de juiste logger gebruiken:

```
using Serilog;

var builder = WebApplication.CreateBuilder(args);
builder.Host.UseSerilog((hostbuilder, config) =>
{
    config.WriteTo.File(@"C:\logs\log.txt");
});
```

We zeggen tegen de Host van onze builder dat we Serilog gebruiken in plaats van de standaard implementatie. We configureren deze Serilog zodat we de log wegschrijven naar een bestand genaamd log.txt.

Hierna zien we een map logs verschijnen onder onze C schijf. We zien dat Serilog standaard een pak meer logt dan we via de console hadden. We zien ook onze logs levels vanaf level Information verschijnen:

A screenshot of a Notepad window titled 'log.txt - Notepad'. The window has a menu bar with 'File', 'Edit', and 'View'. The text content shows a series of log entries with timestamps, ISO 8601 times, and log levels. The last four lines are highlighted in yellow.

```
log.txt - Notepad
File Edit View

2022-06-22 22:25:06.162 +02:00 [INF] Now listening on: https://localhost:7267
2022-06-22 22:25:06.163 +02:00 [INF] Now listening on: http://localhost:5267
2022-06-22 22:25:06.172 +02:00 [INF] Application started. Press Ctrl+C to shut down.
2022-06-22 22:25:06.173 +02:00 [INF] Hosting environment: Development
2022-06-22 22:25:06.173 +02:00 [INF] Content root path: C:\Users\wivan\UC Leuven-Limburg\Graduaat Programmeren
2022-06-22 22:25:07.300 +02:00 [INF] Request starting HTTP/2 GET https://localhost:7267/ - -
2022-06-22 22:25:07.380 +02:00 [INF] Executing endpoint 'Uc11.OOD.Logging.Controllers.HomeController.Index (/
2022-06-22 22:25:07.410 +02:00 [INF] Route matched with {action = "Index", controller = "Home"}. Executing c
2022-06-22 22:25:07.412 +02:00 [INF] Logging an Information level logline
2022-06-22 22:25:07.412 +02:00 [WRN] Logging a Warn level logline
2022-06-22 22:25:07.412 +02:00 [ERR] Logging an Error level logline
2022-06-22 22:25:07.412 +02:00 [FTL] Logging a Critical level logline
```

Dankzij dependency injection hebben we niets moeten wijzigen aan de werking van onze controllers. Zij gebruiken nog steeds de ILogger als abstractie. Zowel de default logger als onze nieuwe Serilog houden zich aan dit contract, waardoor van logger implementatie wisselen zeer simpel wordt.

Natuurlijk kunnen we dit ook doen via config, zodat we weer op verschillende environments andere settings kunnen hebben.

Hiervoor passen we wederom de Program.cs aan om aan Serilog te zeggen dat we zijn configuratie uit de appsettings halen:

```
builder.Host.UseSerilog((hostbuilder, config) =>
{
    config.ReadFrom.Configuration(hostbuilder.Configuration);
});
```

Nu moeten we nog de appsettings aanpassen. We nemen onze appsettings.Development.json. Deze laten we loggen op het laagste niveau, Trace. Serilog heeft hiervoor het level *Verbose* nodig:

```
{
  "Serilog": {
    "Using": [ "Serilog.Sinks.File" ],
    "MinimumLevel": {
      "Default": "Verbose",
      "Override": {

```



```
    },
    "WriteTo": [
      {
        "Name": "File",
        "Args": {
          "name": "log",
          "path": "C:\\logs\\log.txt"
        }
      }
    ],
    "Enrich": [ "FromLogContext", "WithMachineName" ],
    "Properties": {
    }
  }
}
```

Hierna krijgen we het gewenste resultaat voor onze development log:

```
2022-06-22 23:02:36.675 +02:00 [VRB] Logging a Trace level logline
2022-06-22 23:02:36.675 +02:00 [DBG] Logging a Debug level logline
2022-06-22 23:02:36.675 +02:00 [INF] Logging an Information level logline
2022-06-22 23:02:36.676 +02:00 [WRN] Logging a Warn level logline
2022-06-22 23:02:36.676 +02:00 [ERR] Logging an Error level logline
2022-06-22 23:02:36.676 +02:00 [FTL] Logging a Critical level logline
```