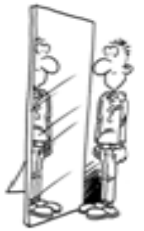


Programmer solutions

Jef Daels



Inhoud

1	Random topics	7
1.1	Extention methods	7
1.1.1	Gebruik van Extention methods	8
1.1.1.1	LINQ en extention methods	8
1.1.1.2	Animaties en extention methods	8
1.1.2	Extention method surprises	8
1.2	Transformations	8
1.3	Awaitable methodes	9
1.4	(XML) serialisatie in .NET	9
1.5	Var keyword	10
1.6	TryParse methodes	10
1.7	Dynamic keyword	10
1.7.1	lock en volatile keywords	10
1.7.2	WPF toolkit	10
1.7.3	Application settings in WPF	10
1.7.4	conditional[debug]	10
1.7.5	custom build	10
1.7.6	sys tools	10
2	Delegates	11
2.1	Delegates: duiding	11
2.1.1	Relevante namespace	11
2.2	Delegates: technische intro	12
2.2.1	Demo-code en toelichting	12
2.2.2	Covariance en contravariance	13
2.2.2.1	Covariance	13
2.2.2.2	Contravariance	13
2.3	Delegates en events	13
2.3.1	Definitie van een event	14
2.3.2	Abonneren (to subscribe) op een event	14
2.3.2.1	Event subscription: verschil met VB.NET	15
2.3.3	Afvuren van een event	15
2.3.3.1	Afvuren van een event: verschil met VB.NET	15
2.3.4	Meerdere clients voor hetzelfde event	16

2.4	Event Guidelines!	16
2.5	Omtrent Events:	17
2.5.1	Static events	17
2.5.2	Delegates of Interfaces: .NET of JAVA events	18
2.5.3	Routed events	18
2.6	Delegates en multithreading	19
2.6.1	Multithreading: wat bedoelt u?	19
2.6.2	Multithreading: iets voor ons?	19
2.6.3	Asynchrone processing en delegates	20
2.6.3.1	Asynchroon, geen AsyncCallback	20
2.6.3.2	Asynchroon, met AsyncCallback, zonder UI	21
2.6.3.3	Asynchroon, met AsyncCallback, met UI	22
2.6.4	Multithreading met de Thread klasse	22
2.6.5	Event based Asynchronous Pattern (EAP)	23
2.6.6	Asynchroon programmeren made easy	23
2.6.7	CancellationToken	24
2.6.8	Meer info?	24
2.7	Code plumbing, aka 'the strategy pattern'	24
2.7.1	Codeplumbing: een XMLViewer	25
2.7.2	Codeplumbing pitfalls	26
2.8	Delegates: faits divers	27
2.8.1	Action en Func	27
2.8.1.1	Action	27
2.8.1.2	Func	27
2.8.2	Meer dan (begin)-invoke	27
2.8.3	Timers	27
2.8.4	Asynchrone eventverwerking	28
2.8.5	Anonymous methods (.NET 2.0)	28
2.8.6	Lambda expressions (.NET 3.0)	28
2.9	Delegates: niet voor elk programma een geschikte oplossing	29
2.10	Delegates: opgaves	29
3	Reflection	31
3.1	Reflection: duiding	31
3.1.1	Relevante namespaces	31
3.2	Reflection: attributen voorbeeld	32
3.3	Reflection in .NET: meer dan attributen	33
3.3.1	Reflection in .NET: start	33
3.3.2	Hoe een instantie maken van een klasse in een assembly?	34
3.3.3	.NET en attributes	35
3.3.3.1	.NET en attribute informatie opvragen	35
3.3.3.2	Hoe zelf Attribute klassen maken in .NET?	35
3.4	Een eigen plugin- infrastructuur!	36
3.4.1	Een eigen plugin- infrastructuur: vereisten	36

3.4.2	Een eigen plugin- infrastructuur: mogelijke oplossing	36
3.4.3	Een eigen plugin- infrastructuur: implementatie	37
3.4.4	Eigen plugin infrastructuur: demo	38
3.5	Namespaces met interessante attributen	38
3.6	Reflection: opgaves	38
4	Runtime compilatie	39
4.1	Runtime compilatie: duiding	39
4.1.1	Relevante namespaces	39
4.2	.NET compilaties starten	40
4.2.1	Command line compilatie	40
4.2.1.1	Command line compilatie: C#	40
4.2.1.2	Command line compilatie: VB.NET	41
4.2.1.3	Een extraatje omtrent MSBuild	42
4.2.2	Runtime compilatie: System.CodeDom.Compiler	43
4.3	CodeDom	44
4.4	System.Reflection.emit	44
4.5	Het Roslyn project	44
4.6	Assembly code inspecteren	45
4.6.1	ILDasm	45
4.6.2	ILSpy	46
4.6.3	Reflector	47
4.6.4	Obfuscatie	48
4.6.4.1	Confuser	48
4.7	Opgaves	49
4.7.1	Een tekenprogramma	49
4.7.2	ILSpy	49
4.7.3	Obfuscatie	49
5	Backtracking	51
5.1	Backtracking: duiding	51
5.1.1	Relevante namespaces	51
5.1.2	Backtracking: principe	51
5.2	N-Queens probleem	52
5.2.1	Probleemstelling, synchrone oplossing	52
5.2.2	Asynchroon: BeginInvoke/4	52
5.2.3	Asynchroon: async/await	53
5.3	Sudoku's oplossen	53
5.3.1	Sudoku: spelregels	53
5.3.2	Software vereisten	54
5.3.3	Sudoku oplos algoritme	54
5.3.3.1	Constraint satisfaction solver	55
5.4	Sudoku routed events introductie	55
5.5	Async methodes: TaskCompletionSource	56
5.5.1	Task- klassen	57

5.5.2	TaskCompletionSource- klassen	57
5.6	Het MiniMax algoritme (geen examen stof)	59
5.6.1	MiniMax benadering voor vier op een rij	59
5.6.1.1	BordWaarde	59
5.6.1.2	BesteZet	60
5.6.1.3	MiniMax methode	61
5.6.2	Minimax optimalisaties	61
5.6.2.1	Alfa-Beta pruning	61
5.6.2.2	Multi- threading	62
5.7	Backtracking opgaves	63
6	Dependency properties	65
6.1	Dependency properties: duiding	65
6.1.1	Relevante namespaces	65
6.2	F1URL: een help infrastructuur	66
6.2.1	F1URL gebruik in XAML	66
6.2.2	F1URL definitie	67
6.2.3	F1URL infrastructuur	68
6.3	Dependency properties: technisch	69
6.4	Dependency property XAML binding	70
6.5	Dependency property syntax	70
6.6	Dependency properties en animaties	71
6.6.1	DoubleAnimation	71
6.6.2	ColorAnimation	72
6.6.3	Transformatie animaties	73
6.6.4	Animaties en XAML	74
6.6.5	C# of XAML animaties?	74
6.7	DependencyProperties in XAML: TypeConverters	75
6.7.1	IValueConverter interface	75
6.7.2	Default dataconverters	76
6.7.2.1	TryParse	76
6.8	Dependency properties en OnRender	77
6.9	Veel voorkomende fouten	77
6.10	Extra lectuur	77
6.11	Dependency property: opgaves	78

Hoofdstuk 1

Random topics

1.1 Extention methods

Extention methods bieden de mogelijkheid om, op syntactische wijze, methodes toe te voegen aan klassen: u kan met behulp van de `.-` notatie deze methodes oproepen. Dit is enkel *syntactic sugar*, omdat de klasse definitie zelf niet wordt uitgebreid (hiervoor zou u overerving nodig hebben).

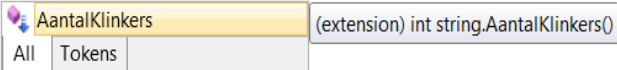
Het definiëren van een extention method gebeurt door middel van een static routine, zoals hieronder met een voorbeeld beschreven:

```
public static int AantalKlinkers(this string tekst)
{
    int iAantal = 0;
    foreach (char c in tekst)
        if (IsKlinker(c)) iAantal++;
    return iAantal;
}
```

Figuur 1.1: Extention method definition

Bij het coderen van een extention method zal de intellisense tonen dat het een extention method betreft:

```
public MainWindow()
{
    InitializeComponent();
    Console.WriteLine("aantal klinkers in " + this.Title + ": "
        + this.Title.AantalKlinkers());
    int i = this.Title.aantal
}
}
```



Figuur 1.2: Extention method gebruik

1.1.1 Gebruik van Extention methods

1.1.1.1 LINQ en extention methods

Extention methods werden geïntroduceerd tesamen met LINQ, wiens implementatie grotendeels gebaseerd is op Extention methods (zie de intellisense bij het oproepen van een LINQ-methode)

1.1.1.2 Animaties en extention methods

Designers zullen animaties beschrijven door middel van XAML code (bijvoorbeeld bij een STATE-wijziging bij het gebruik van de VisualStateManager). Eenzelfde animatie zal op verschillende plaatsen gedupliceerd worden. Indien we de animatie als extention method implementeren kan ze eenvoudig (via code) herhaaldelijk worden toegepast:

```
spNumbers11.FadeTo((speler1Waits) ? 0.15 : 1.0, 500);  
spNumbers12.FadeTo((speler1Waits) ? 0.15 : 1.0, 500);  
spNumbers21.FadeTo(!speler1Waits) ? 0.15 : 1.0, 500);  
spNumbers22.FadeTo(!speler1Waits) ? 0.15 : 1.0, 500);
```

Figuur 1.3: Extention method gebruik: fadeto

Opmerking: de gecodeerde animatie is gemakkelijk in een behavior te plaatsen zodat ze ook vanuit XAML snel kan worden geactiveerd.

1.1.2 Extention method surprises

Extention methodes gedefinieerd op een klasse zijn ook van toepassing op instanties van een ervende klasse.

1.2 Transformations

Een transformatie kan opgebouwd zijn uit verschillende elementen: een draaiing (rotatie) een schaling (scaling), een verplaatsing (translation) en kanteling (skewing). Indien u een tranformatie op een object toepast dient u rekening te houden met de reeds bestaande transformaties: indien u een object wenst te herschalen mag u de eventuele draaiing niet verwijderen. Een CompositeTransform object zal in veel gevallen uitermate geschikt zijn om met één object al uw tranformaties te definiëren en animeren. Dit vermijdt dat u voor een animatie alle TransformGroup onderdelen moet nakijken om de gepaste tranformatie te vinden.

1.3 Awaitable methodes

.NET 4.5 introduceert *awaitable* methodes. Deze methodes zijn herkenbaar aan de suffix *Async*. Hun uitvoering zal (normaal gezien :) door het keyword *await* worden voorafgegaan. In dat geval zal de rest van de code pas worden uitgevoerd nadat de (asynchrone) oproep van de Async-methode is afgelopen (dit wordt meer in detail toegelicht in het hoofdstuk omtrent delegates en het hoofdstuk omtrent backtracking).

```
public static async Task XMLSerialise(Bord brd)
{
    StorageFolder storageFolder = ApplicationData.Current.LocalFolder;
    StorageFile sf = await storageFolder.CreateFileAsync("numberlegions
Stream s = await sf.OpenStreamForWriteAsync();
XmlSerializer serializer = new XmlSerializer(typeof(Bord));
serializer.Serialize(s, brd);
s.Dispose();
}
```

Figuur 1.4: Await demo

Belangrijk hierbij op te merken zijn volgende elementen:

- indien u een methode *await*, dan kan dit enkel in een methode die als *async* gedefinieerd werd;
- indien een methode als *async* gedefinieerd wordt is ze op haar beurt *awaitable*.
 - een *async* methode zal, indien opgeroepen zonder *await*, asynchroon worden uitgevoerd: de oproepende methode wacht niet op voltooiing vooraleer verder te gaan met zijn logica:

```
async void brdNumbers_SpelerToggled(object se
{
    _isToetsVerwerkingBusy = false;
    pSpeler1ScoreA.Text = "" + brdNumbers.Sco
    pSpeler2ScoreA.Text = "" + brdNumbers.Sco
    pSpeler1ScoreB.Text = "" + brdNumbers.Sco
    pSpeler2ScoreB.Text = "" + brdNumbers.Sco
    await Bord.XMLSerialise(brdNumbers);
    await ZetVolgendeZet(e);
}
```

Figuur 1.5: Await demo async methode

1.4 (XML) serialisatie in .NET

aanbrengen In een context van application life cycle

1.5 Var keyword

als je type kent: gebruik het! var is vooral interessant om het resultaat type van een linq query niet zelf te moeten tikken.

1.6 TryParse methodes

boolean result, twee parameters, tweede parameter is ref, lijkt me zeer toepasbaar bij dataconverters in XAML

1.7 Dynamic keyword

1.7.1 lock en volatile keywords

1.7.2 WPF toolkit

1.7.3 Application settings in WPF

Best way to bind WPF properties to ApplicationSettings in C#?

1.7.4 conditional[debug]

1.7.5 custom build

1.7.6 sys tools

Hoofdstuk 2

Delegates

2.1 Delegates: duiding

In een volledig object georiënteerde programmeeromgeving is het mogelijk om code- onderdelen als objecten te aanzien. Dit betekent dat deze onderdelen (routines, classes, assemblies, types, ..) instanties zijn van klassen die dergelijke code elementen beschrijven.

Routine- objecten (het onderwerp van dit hoofdstuk) zijn instanties van een *Delegate-type*. Deze Delegate- types zijn klassen die erven van de *Delegate-* class. Deze laatste is zelf geen Delegate-type (er zijn geen routines die instanties zijn van de basis Delegate-class).

In het hoofdstuk omtrent reflection zullen we ook andere code onderdelen als objecten verwerken. Routines zullen daar ook als instanties van (Reflection-) classes terugkeren. In een reflection context zullen deze instanties vooral beschrijvende (meta-) informatie bevatten.

Delegates spelen in .NET een belangrijke rol bij het implementeren van events, multithreading en codeplumbing. Ook de implementatie van WCF en (attached) dependency properties maken gebruik van delegates. Delegates zijn dan ook zonder twijfel een belangrijke werkstuk voor de betere programmeur!

2.1.1 Relevante namespace

- *System*

2.2 Delegates: technische intro

2.2.1 Demo-code en toelichting

De meest voor de hand liggende actie die we op een routine willen toepassen, is deze uitvoeren. Hierbij is het van belang te weten welke argumenten moeten voorzien worden en van welke type het eventuele resultaat een instantie zal zijn. Een delegate- type definitie zal zich hiertoe beperken: welke zijn de argumenttypes, en welk is het eventuele resultaat type? De naam van de routine en de parameters, en de geïmplementeerde logica, zijn volledig irrelevant voor de delegate type definitie. Een voorbeeld delegate- type definitie:

```
public delegate void DoeIetsMetTekstDelegate(string sTekst);
```

Figuur 2.1: Delegate definitie

Merk hierbij volgende punten op:

- het keyword *delegate* heeft aan dat we een delegate-type definiëren (een speciale klasse, zonder body);
- het return type wordt genoteerd voor de naam van het delegate type. In het voorbeeld is dit return type void (we verwachten geen resultaat);
- de argumentnamen zijn niet van belang (en moeten zeker niet overeenstemmen met de argumentnamen gebruikt in de routines die instanties van deze klasse zullen zijn). De argumenttypes zijn *wel* belangrijk!

```
public delegate void doeIetsMetTekst(String tekst);

public partial class MainWindow : Window
{
    private void btnDemoUCLC_Click(object sender, RoutedEventArgs e)
    {
        doeIetsMetTekst routine = new doeIetsMetTekst(toonUC); //maak een routinereferentie en koppel ze
        routine.Invoke("hAllo"); //start de methode gekoppeld aan deze routinereferentie, en voorzie de oproep van de nodige parameters
        routine = new doeIetsMetTekst(toonLC);
        routine.Invoke("hAllo");
    }
    private void toonUC(string tekst) { MessageBox.Show(tekst.ToUpper()); }
    private void toonLC(string tekst) { MessageBox.Show(tekst.ToLower()); }
}
```

Figuur 2.2: Delegate demo: uppercase en lowercase

Bovenstaande demo-code zal u wellicht niet overtuigen van het belang van delegates: het aanroepen van een methode wordt gewoon complexer gemaakt. Verder in dit hoofdstuk behandelen we een aantal (niet exhaustief (=uitputtend)) situaties waarbij het gebruik van delegates een elegante oplossing biedt.

Een delegate- definitie is niet hetzelfde als een signatuur definitie: de routinenaam maakt onderdeel uit van de signatuur (en niet van het delegate- type), en het returntype is geen onderdeel van de signatuur, maar wel van belang in de delegate definitie.

2.2.2 Covariance en contravariance

Wat indien een routine hoofding niet volledig overeenstemt met de delegate definitie? Een routine kan op twee manieren afwijken van een delegate type definitie, en toch nog als instantie van dit delegate type worden aanzien: covariance en contravariance. Beide worden beschreven in Covariance and Contravariance in Delegates (C# Programming Guide). Een combinatie van beide is natuurlijk ook toegelaten.

2.2.2.1 Covariance

Indien het returntype van een routine een subclasse is van het return type van het delegate type, dan is deze routine toch een instantie van dit delegate type. Uiteraard levert dit geen problemen op, gezien elk resultaat kan gecast worden naar het return type van het delegate type.

2.2.2.2 Contravariance

Indien een argument van de routine minder gespecialiseerd is dan het overeenkomstige argument in het delegate type is er uiteraard ook geen probleem: elke actuele parameter moet voldoen aan het delegate (formeel) parameter type, en voldoet dus zeker ook aan het algemenere routine parameter type.

2.3 Delegates en events

Events leveren een mechanisme waarbij een server-object (levert diensten) initiatief neemt om code door een client-object te laten uitvoeren. In .NET wordt dit gerealiseerd door middel van delegates (dit in tegenstelling tot JAVA waar men gebruikt maakt van interfaces (denk bijvoorbeeld aan de *ActionListener* interface)).

Het client of server zijn van een object ten opzichte van elkaar hangt af van de te leveren dienst: indien we een Button op een Window plaatsen dan is deze Button client van het Window omwille van de visuele ondersteuning (het Window levert een dienst), en is het Window evenzeer client van de Button omdat de Button onder andere een click- dienst aanlevert. Het is dus zeker niet zo dat een client/server relatie gelezen moet worden als een klein/groot- relatie.

2.3.1 Definitie van een event

De definitie van een event voorziet een delegate- typed field van het keyword *event*:

```
public delegate void EventVoorbeeldNietVolgensGuidelines(string tekst); //delegate voor het event
public class DemoTekstVak : TextBox
{
    public event EventVoorbeeldNietVolgensGuidelines TekstIsVijfLang; //eventdefinitie
    public DemoTekstVak()
    {
        ..
    }
}
```

Figuur 2.3: Event definitie (zonder guidelines)

De delegate in de eventdefinitie beschrijft de client-routines die aan dit event gekoppeld kunnen worden. Bemerkt dat de code in dit voorbeeld *niet* voldoet aan de guidelines omtrent eventhandling.

Bovenstaande code is onderdeel van een gespecialiseerde TextBox klasse die voorzien wordt van een TekstIsVijfLang event. Het is de bedoeling dit event af te vuren indien de tekst in het tekstvak exact 5 tekens lang wordt.

2.3.2 Abonneren (to subscribe) op een event

De syntax nodig om een client te abonneren op een event is verschillend met deze van VB.NET (waarbij een *handles* clause achteraan een routine werd toegevoegd). In *C#* wordt een routine gekoppeld aan een event door middel van de *+=* operator:

```
public MainWindow()
{
    InitializeComponent();
    txt5Tekens.TekstIsVijfLang += new EventVoorbeeldNietVolgensGuidelines(txt5Tekens_TekstIsVijfLang);
}
void txt5Tekens_TekstIsVijfLang(string tekst)
{
    Console.WriteLine(tekst + " is 5 tekens lang");
}
```

Figuur 2.4: Event abonnering

Visual Studio ondersteunt het abonneren op volgende wijze:

- indien u na de *+=* operator de *TAB* toets indrukt wordt automatisch een correcte right-hand expressie gegenereerd;
- indien u hierna nog eens *TAB* indrukt zal ook een correcte eventhandler routine gegenereerd worden, rekening houdend met de event definiërende delegate.
- indien u dubbelklikt op een control wordt er een *+=* expressie aangemaakt in de generated code en wordt een passende eventhandler in de *.cs*- fiel geplaatst;

2.3.2.1 Event subscription: verschil met VB.NET

- *Handles-* clause: Hiervoor bestaat geen *C#*- equivalent: *C#* kent enkel de *+=*- operator om routines aan events te koppelen;
- *AddHandler*: De VB.NET *handles* clause is niet in staat om routines *at run-time* aan events te koppelen. Hierdoor is deze werkwijze ongeschikt om events van bijvoorbeeld dynamisch toegevoegde controls op te vangen. VB.NET kent de *AddHandler* instructie om *at run-time* routines aan events te koppelen. Indien u de documentatie erop naleest merkt u dat VB.NET de *AddressOf*- operator gebruikt in combinatie met de *AddHandler* instructie.

2.3.3 Afvuren van een event

De klasse die het event definieert kan dit event ook uitvoeren. Het is belangrijk om te testen of er clients zijn voor dit event (indien er geen clients zijn is het field *null* en zou het afvuren een uitvoeringsfout veroorzaken):

```
public delegate void EventVoorbeeldNietVolgensGuidelines(string tekst); //delegate voor het event
public class DemoTekstVak : TextBox
{
    public event EventVoorbeeldNietVolgensGuidelines TekstIsVijfLang; //eventdefinitie
    public DemoTekstVak()
    {
        this.TextChanged += new TextChangedEventHandler(DemoTekstVak_TextChanged);
    }

    void DemoTekstVak_TextChanged(object sender, TextChangedEventArgs e)
    {
        if (this.Text.Length == 5)
            if (TekstIsVijfLang != null) //minstens 1 client voor dit event nodig, anders null
                TekstIsVijfLang(this.Text); //starten van het event (voor alle clients)
    }
}
```

Figuur 2.5: Event afvuren (zonder guidelines)

Bemerkt dat in bovenstaande code, die het afvuren van een event illustreert, er ook een event abonnering aanwezig is: het object abonneert zichzelf (in de constructor) op zijn eigen *TextChanged*- event (door middel van de *+=*- operator) om na te kijken of de tekst na wijziging exact vijf tekens lang is.

2.3.3.1 Afvuren van een event: verschil met VB.NET

- *RaiseEvent* is de instructie om in VB.NET een event af te vuren. Het is *niet* nodig om te controleren of er clients zijn voor dit event.

2.3.4 Meerdere clients voor hetzelfde event

Het is perfect mogelijk om meerdere event-handlers te koppelen aan eenzelfde event. Bij het afvuren van het event zullen de gekoppelde routines in volgorde, één voor één uitgevoerd worden.

2.4 Event Guidelines!

Hoewel er technisch geen beperkingen worden opgelegd aan de event definiërende delegates worden door Microsoft volgende guidelines beschreven:

- het return type is *void*;
- de delegate heeft exact twee argumenten:
 - Object sender: het object waardoor het event wordt gestart;
 - EventArgs e: EventArgs mag vervangen worden door een klasse die overerft van EventArgs indien u extra informatie aan het event wenst toe te voegen (voorbeelden hiervan zijn de mouselistener events);
- het afvuren van het event gebeurt in een overschrijfbare (*protected virtual*) methode. Dit laat ervende klassen toe om het event gedrag aan te passen (zie DemoButton in het demo project). De methodenaam is *onEvent* waarbij Event natuurlijk wordt vervangen door de naam van het event.
- gebruik *System.EventHandler<T>* in plaats van zelf aangemaakte delegate definities.

Deze guidelines *moeten* in deze cursus worden toegepast opdat uw event-implementaties als correct zouden worden aanzien.

Als een eerste oefening kan men de event implementatie, zoals hierboven beschreven, hercoderen zodat ze voldoet aan de guidelines.

Wie niet (voldoende) bekend is met *generics* (*System.EventHandler<T>*) wordt verondersteld dit op te zoeken. Indien zelfstudie onvoldoende duidelijkheid schept kan u vragen stellen in de komende lessen.

2.5 Omtrent Events:

2.5.1 Static events

Meestal associëren we een event met een instantie. Om te abonneren op een event van een instantie hebben we een referentie naar die instantie nodig. Dit is niet altijd mogelijk: veronderstel dat client code wenst te reageren op het aanmaken van een nieuw object van een klasse: het is niet mogelijk in te tekenen op het *PersoonCreated* event van een nog onbestaand object. .NET voorziet de mogelijkheid om static events te definiëren: de client abonneert dan op een (static) event van een klasse:

```
public class clsPersoon
{
    public static event EventHandler<PersoonCreatedEventArgs> PersoonCreated;
    public string FNaam { get; set; } //property definitie
    public string VNaam { get; set; } //property definitie
    public clsPersoon() : this("", "")
    {
    }
    public clsPersoon(string fnaam, string vnaam)
    {
        FNaam = fnaam;
        VNaam = vnaam;
        if (PersoonCreated != null)
            PersoonCreated(null, new PersoonCreatedEventArgs(this));
    }
    public override string ToString()
    {
        return this.FNaam + ", " + this.VNaam;
    }
}

public class PersoonCreatedEventArgs : EventArgs
{
    private clsPersoon _persoon;
    public PersoonCreatedEventArgs(clsPersoon persoon) { _persoon = persoon; }
    public clsPersoon persoon { get { return _persoon; } }
}
```

Figuur 2.6: Static event definitie (volgens de guidelines)

```
public MainWindow()
{
    InitializeComponent();
    clsPersoon.PersoonCreated += new EventHandler<PersoonCreatedEventArgs>(clsPersoon_PersoonCreated);
}

void clsPersoon_PersoonCreated(object sender, PersoonCreatedEventArgs e)
{
    Console.WriteLine(e.persoon.ToString());
}
```

Figuur 2.7: Static event client

2.5.2 Delegates of Interfaces: .NET of JAVA events

In .NET gebruikt men delegates om eventhandling te realiseren. In JAVA definieert men interfaces om eventhandling mogelijk te maken: een instantie van een interface implementerende klasse kan dan als eventhandler aan een object toegekend worden. Welke is te verkiezen?

Bij manuele codering (wie doet dat nog?) moet in JAVA elk interface element voorzien worden, ook al bent u enkel geïnteresseerd in een specifieke routine. In .NET moet u enkel die events die u interesseren van code voorzien. Beide oplossingen zijn voldoende eenvoudig om een intuïtief gebruik mogelijk te maken (zeker indien de IDE u ook nog wat helpt). Beide benaderingen zijn dan ook sterk gelijkwaardig: ik verkies er geeneen boven de andere.

2.5.3 Routed events

Voorgaande eventhandling is reeds van bij de start van .NET aanwezig in het framework. Recentelijk (WPF, .NET 3.5) werden ook *routed events* geïntroduceerd. Deze events zijn WPF specifiek en worden geïntroduceerd om volgende problemen op te lossen:

- veronderstel een Button, met daarop een tekening (Image);
- de gebruiker klikt op de tekening: wellicht wil hij de button inklikken? Misschien ook niet? Hoe wordt de Button op de hoogte gesteld van het klikken op de Image? Wat indien er geen Image is?

Een routed event is een event (anders dan voorgaande) om binnen een geneste lijst van controls een gebeurtenis door te geven. We spreken van *tunneling events* indien de container het event doorgeeft naar zijn child, en van *bubbling events* indien het child het event doorgeeft aan zijn container. Een parent (bijvoorbeeld Window) is in staat om te subscriben (addHandler) op het routed event (bijvoorbeeld *TextChanged*- event van een bij abonnering niet gekend child (zie ook Sudoku routed events).

Indien een WPF- gebeurtenis zich voordoet zal het doorgeven van events in twee stappen gebeuren:

- eerst wordt het event van de containers naar de children doorgegeven: *tunneling*. Deze events worden prefixed met *preview*;
- daarna volgt een bubbling fase waarbij de gewone event naam wordt gebruikt;
- indien een control meent het event correct af te handelen kan de handled property van het tweede event-argument op *true* worden geplaatst, wat het doorgeven van die specifieke fase (grotendeels) stopt.

Interessant om weten:

- routed events bestaan enkel in een WPF-omgeving. Ze kunnen dan ook niet gebruikt worden om eventhandling buiten de WPF omgeving (uw visualisatie) te implementeren.
- snoop is een leuke tool om WPF-constructies te onderzoeken. Ook het verloop van routed events kan met deze tool worden nageplozen.

2.6 Delegates en multithreading

2.6.1 Multithreading: wat bedoelt u?

Bij het uitvoeren van een programma worden instructies één voor één door de CPU verwerkt. Een thread is een onderdeel van een proces (programma) dat in staat is om opeenvolgende instructies uit te voeren. Een multithreaded programma heeft meerdere onafhankelijk van elkaar uitvoerende threads. Wanneer het programma wordt uitgevoerd door een multithreaded operating system worden verschillende zaken op hetzelfde moment uitgevoerd.

2.6.2 Multithreading: iets voor ons?

Multithreaded programmeren is *ongelooflijk* delicaat: indien gelijktijdige threads eenzelfde variabele wensen aan te passen krijgen we de problemen die we kennen vanuit de cursus databases: *lost update*, *inconsistent read* en *ghost data* om er slechts enkele te noemen. Het schrijven van correcte multithreaded toepassingen is dan ook geen *walk in the park/piece of cake*.

Betekent dit dat we geen multithreading kunnen programmeren? Natuurlijk niet: het ophalen van een grote lijst gegevens, het verwerken van een lijst gegevens die niet gewijzigd worden: allemaal ideale kandidaten om multithreaded uitgewerkt te worden. Indien er zich concurrency (gelijktijdigheid) problemen kunnen voordoen zullen we locks gebruiken. Bij het database programmeren worden locks door het RDBMS geplaatst. In *C#* (en VB.NET) kunnen we dit zelf programmeren met behulp van de *lock*- instructie.

Wanneer we spreken over multithreaded toepassingen hebben we het veelal ook over:

- parallel processing: parallelle (gelijktijdige) uitvoeringen van meerdere threads impliceren multithreading;
- asynchrone verwerking (in tegenstelling tot synchrone verwerking bij single threaded toepassingen): hierbij wordt een routine gestart en wachten we niet op het einde van deze routine vooraleer we verder gaan. Asynchrone verwerking wordt in .NET gerealiseerd met behulp van delegates. Een voor de hand liggend probleem dat moet worden opgelost: hoe weet mijn programma dat een asynchrone uitvoering afgelopen is (om bijvoorbeeld een *printen beëindigd* boodschap te kunnen tonen)?

Multithreaded applicaties zijn ook op een single- core CPU belangrijk: zo kan men het printen van een document op de achtergrond laten uitvoeren, terwijl de user-responsiveness niet daalt. Met de introductie van multi-core CPU's wordt het ontwikkelen van multi-threaded toepassingen nog interessanter: een single-threaded toepassing zal op een quad core nooit meer dan vijventwintig procent van de beschikbare rekenkracht kunnen gebruiken. Dit maakt het multi-threaded zijn van uw toepassing belangrijk indien u het volle potentieel van de machine wenst te gebruiken.

2.6.3 Asynchrone processing en delegates

2.6.3.1 Asynchroon, geen AsyncCallback

Bij een synchrone verwerking zal een routine de volledige uitvoering van een instructie (eventueel een deelprobleem) afwachten vooraleer de volgende instructie wordt gestart. Zo zal de uitvoering van `btnDemoUCLC_click` in Delegate demo: uppercase en lowercase na het tonen van de eerste messagebox wachten tot de gebruiker op *ok* klikt vooraleer de tweede messagebox wordt getoond.

Wanneer we de code aanpassen door de *invoke/1*- methods van de delegates te vervangen door *BeginInvoke/3* dan worden de delegate routines asynchroon ten opzichte van de oproepende methode uitgevoerd:

```
private void btnAsyncDemo_Click(object sender, RoutedEventArgs e)
{
    doeIetsMetTekst routine = new doeIetsMetTekst(toonUC); //maak een routinereferentie en koppel ze
    routine.BeginInvoke("hAllo", null, null); //start de delegate asynchroon|
    routine = new doeIetsMetTekst(toonLC);
    routine.BeginInvoke("hAllo", null, null);
}
```

Figuur 2.8: Asynchrone verwerking demo

Op te merken bij de nieuwe situatie:

- deze uitvoering resulteert in drie threads: de thread die het klik- event afhandelt, en een tread voor elk van de asynchrone oproepen;
- *BeginInvoke/3* kent drie parameters in plaats van één: de twee laatste parameters van een asynchrone delegate oproep bevatten een callback routine (de routine die wordt uitgevoerd als de asynchroon gestartte routine eindigt) en een status object als laatste argument (beide komen in het volgende punt aan bod). Het is de bedoeling om in dit status object informatie, verzameld gedurende de asynchrone uitvoering (bijvoorbeeld het goed of slecht aflopen) terug te geven aan de oproeper. Dit laatste argument is (later) terug te vinden als state property van de *IAAsyncResult*- parameter van de *AsyncCallback* routine.
- de callback methode wordt via een *AsyncCallBack*- delegate aan de asynchrone oproep meegegeven (de callback methode is die routine die zal worden uitgevoerd wanneer de asynchrone oproep afgelopen is). Het is belangrijk op te merken dat de callback routine wordt uitgevoerd op de asynchrone thread!

2.6.3.2 Asynchroon, met AsyncCallback, zonder UI

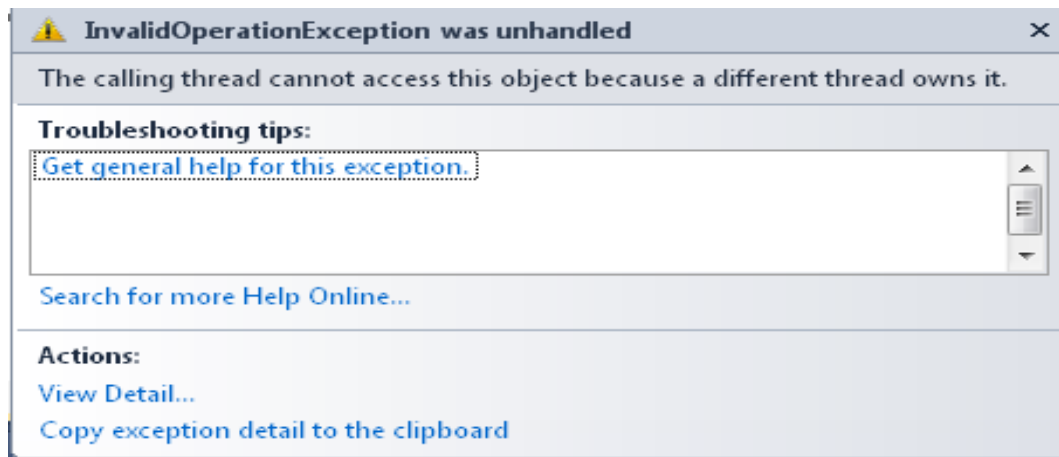
```
private void btnAsyncDemo_Click(object sender, RoutedEventArgs e)
{
    doeIetsMetTekst routine = new doeIetsMetTekst(toonUC); //maak een routinereferentie en koppel ze
    routine.BeginInvoke("hAllo", new AsyncCallback(asyncAfgelopen), this); //start de delegate asynchroon
    routine = new doeIetsMetTekst(toonLC);
    routine.BeginInvoke("hAllo", null, null);
}

private void asyncAfgelopen(IAsyncResult iar)
{
    toonResultaat(iar);
}

private void toonResultaat(IAsyncResult iar)
{
    object state = iar.AsyncState; //bevat laatste argument van de oproep
    this.Title = "afgelopen";
}
```

Figuur 2.9: AsyncCallback demo

In bovenstaande figuur zal de eerste asynchrone uitvoering bij het aflopen de routine *asyncAfgelopen* uitvoeren (op de nieuwe thread). In het *IAsyncResult* argument vinden we informatie omtrent de uitgevoerde routine. De callback routine wordt uitgevoerd op de nieuw gestarte thread. Bovenstaande stukje code loopt fout omdat we op de nieuwe thread de UI wensen aan te passen:



Figuur 2.10: UI-thread problem

In een Windows-omgeving is het enkel mogelijk om de UI-elementen aan te passen op de thread die deze UI-elementen aanmaakte. Indien we in de UI feedback omtrent een afgelopen asynchrone methode wensen te zien, zullen we in staat moeten zijn om naar de UI-thread terug te keren. U raadt het al: dit gebeurt met behulp van delegates.

2.6.3.3 Asynchroon, met AsyncCallback, met UI

```
private void btnAsyncDemo_Click(object sender, RoutedEventArgs e)
{
    doeIetsMetTekst routine = new doeIetsMetTekst(toonUC); //maak een routinereferentie en koppel ze
    routine.BeginInvoke("hAllo", new AsyncCallback(asyncAfgelopen), this); //start de delegate asynchroon
    routine = new doeIetsMetTekst(toonLC);
    routine.BeginInvoke("hAllo", null, null);
}
private void asyncAfgelopen(IAsyncResult iar)
{
    Dispatcher.BeginInvoke(new AsyncCallback(toonResultaat), iar); //Terug naar de ui thread
}
private void toonResultaat(IAsyncResult iar)
{
    object state = iar.AsyncState; //bevat laatste argument van de oproep
    this.Title = "afgelopen";
}
```

Figuur 2.11: Dispatcher demo

Het terugkeren naar de UI-thread gebeurt afhankelijk van de UI-technologie:

- WPF: elke visueel element heeft een *Dispatcher* property. Deze kan een thread starten op zijn UI-thread door middel van de *BeginInvoke/2* methode. Hoewel deze methode dezelfde naam draagt als deze die de delegate routines asynchroon start is het een andere methode (geen twee extra argumenten bijvoorbeeld);
- WinForms: elke control heeft een *BeginInvoke* methode. Deze start een delegate op de thread die de control maakte (de UI-thread).

Het is interessant op te merken dat een *MessageBox.show* oproep zichzelf terug op de gepaste thread plaatst.

2.6.4 Multithreading met de Thread klasse

Een alternatief om multithreaded toepassingen te maken kan gebruik maken van de *Thread*-klasse. Dit valt buiten de scope van deze les.

2.6.5 Event based Asynchronous Pattern (EAP)

Het Event based Asynchronous Pattern plaatst het asynchroon ontwikkelen in de alomgekende event- context:

- zowel de langlopende taak (op de andere thread) als de callback worden gekoppeld aan events (zie het BackgroundWorker voorbeeld hieronder);
- op basis van de uitleg omtrent events zou het duidelijk moeten zijn dat dit een wrapper-oplossing rond de asynchroon uit te voeren delegate en zijn callback is.
- het pattern voorziet ook de mogelijkheid om een langdurende bewerking te onderbreken en om voortgangs events af te vuren. U vindt dit terug in de documentatie :).

De BackgroundWorker class is een mooi voorbeeld van deze benadering:

- het DoWork- event: ik kan het niet beter uitleggen dan wat u in MSDN vindt. Bemerkt de relatie met RunWorkerAsync
- het RunWorkerCompleted- event wordt ook netjes uitgelegd op MSDN.

2.6.6 Asynchroon programmeren made easy

Voorgaande code, waarbij het nodig was om *Callback delegates* en *DispatcherThreads* te begrijpen maakte het niet eenvoudig om multithreaded toepassingen op te zetten. In .NET 4.5 werd gepoogd om dit (syntactisch) een flink stuk te vereenvoudigen met de introductie van de *await*- en *async* keywords:

```
public async Task XMLSerialize()
{
    //dit| wordt enkel gebruikt als er nog geen bord geserialiseerd was (su
    StorageFolder storageFolder = ApplicationData.Current.LocalFolder; //a
    StorageFile sf = await storageFolder.CreateFileAsync("numberlegionspl
    Stream s = await sf.OpenStreamForWriteAsync();
    XmlSerializer serializer = new XmlSerializer(typeof(SplashPage));

    serializer.Serialize(s, this);
    s.Dispose();
}
```

Figuur 2.12: Await- keyword

- het *await*- keyword impliceert volgende zaken:
 - de oproep rechts van het *await*- keyword wordt asynchroon uitgevoerd;
 - de rest van de methode wordt slechts uitgevoerd nadat de asynchrone oproep eindigt. De compiler zal hiertoe de rest van de methode in een delegate wrappen en als callback met de asynchrone oproep meegeven;

- het *async*- keyword in een routine hoofding heeft aan dat deze routine *awaited* kan opgeroepen worden. Meer nog, indien een routine een *awaited*- oproep bevat (typisch een methode oproep die lang kan duren) moet deze routine verplicht *async* gedefinieerd worden. Dit maakt het mogelijk (niet noodzakelijk) om deze routine zelf ook terug *awaited* op te roepen;
 - indien u een *async* methode *niet* awaited oproept dan wordt ze synchroon uitgevoerd (en blokkeert ze dus de oproepende thread).
 - *async* methodes moeten een return type *Task* (indien geen resultaat- waarde) of *Task<resultaattype>* indien een resultaat wordt terug gegeven.
- bij het ontwikkelen van *Modern Style* apps (Microsoft tablet toepassingen) bent u verplicht deze keywords te gebruiken: het manipuleren van files, het raadplegen van webservices: al deze functionaliteit is enkel nog in een asynchrone versie beschikbaar;
- wat u momenteel nog ontbreekt is de kennis om zelf een *asyn* methode te ontwikkelen (zonder gebruik te maken van een andere *asyn* methode die u *await*). Dit zal aan bod komen in het hoofdstuk omtrent backtracking waar we onze artificiële intelligentie asynchroon zullen oproepen.
 - voor wie hierop niet kan wachten: we zullen gebruik maken van de *TaskCompletionSource* klasse.

2.6.7 CancellationToken

De *CancellationToken* structure wordt gebruikt in *GeolocationCS* sample. Dit token dat bij het starten van een *Task* wordt meegegeven geeft de caller de mogelijkheid om de *Task* af te breken. Hiertoe is het nodig dat de gestarte *Task* regelmatig dit token raadpleegt en indien nodig zijn bewerkingen afbreekt.

2.6.8 Meer info?

Over multithreading zijn boeken, wellicht ganse bibliotheken, geschreven. De laatste jaren is er hernieuwde aandacht voor deze topic omwille van de multi core processoren (de programmeer talen worden aangepast om parallelle verwerking gemakkelijker te ondersteunen). Een eventueel startpunt om zich in deze materie te verdiepen kan je vinden op *Asynchronous Programming Overview*. Deze link is wellicht niet beter dan uw eigen favoriete instap link.

2.7 Code plumbing, aka 'the strategy pattern'

Deze laatste topic behandelt het 'at run time' vervangen van code in *één welbepaald* object. Dit is helemaal anders dan overerving: bij overerving zullen we code at compile time aanpassen voor *elke* instantie van de ervende klasse.

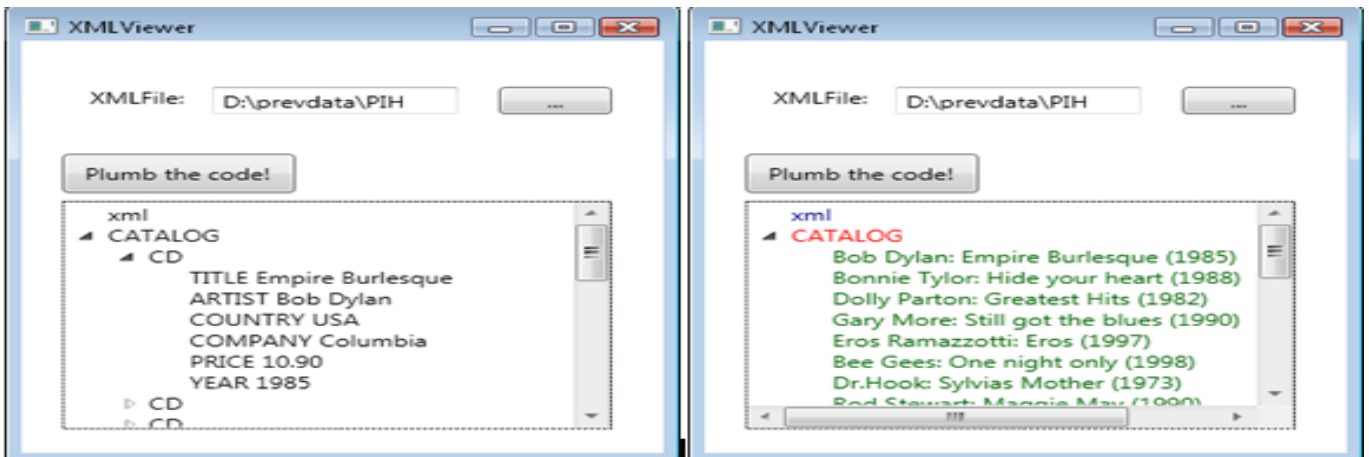
2.7.1 Codeplumbing: een XMLViewer

Veronderstel dat u betrokken bent bij de ontwikkeling van een algemene XML-viewer. Deze is in staat om elke mogelijke XML-file in een treeview te visualiseren. Omdat u op voorhand niet alle XML-tags van uw potentiële klanten kent kan u in uw XMLviewer geen tag- afhankelijke manipulaties coderen: u weet niet op voorhand hoe een XML-file met CD-gegevens vertoont moet worden. Het is onze doelstelling een XMLViewer control te ontwikkelen waarin de gebruiker bepaalde routines kan vervangen door zijn eigen implementaties. Wij voorzien default implementaties die een basis functionaliteit aanbieden zodat de XML-viewer op zijn minst een standaard gedrag vertoont. We gaan hiervoor als volgt te werk:

- maak een XMLViewer usercontrol die erft van TreeView. Deze overloopt bij het verwerken van een XML-file recursief de verschillende tags en toont deze. Wellicht is het interessant om ook een eigen TreeViewItem klasse te maken. Houd rekening met volgende elementen:
 - plaats de berekening van de getoonde tekst voor elke XML-node in een aparte routine (met de XML-node als argument);
 - plaats de berekening van de kleur voor elke XML-node in een aparte routine (met de XML-node als argument);
 - plaats de test die bepaalt of subnodes moeten verwerkt worden in een aparte routine (met de XML-node als argument);

U merkt dat elke functionaliteit die we later voor de gebruiker aanpasbaar willen stellen in aparte routines terechtkomen. Deze routines implementeren het default gedrag (en mogen enkel gebruik maken van de parameters, niet van de module variabelen);

- voorzie voor elke default routine een delegate- class waar deze toe behoort;
- definieer voor elke default routine een field van het overeenkomstige delegate type, en koppel dit field in de constructor aan zijn default implementatie;
- vervang elke oproep van de default-routines door een *invoke*- aanroep van het field. Dit start de methode in het field waarin default onze eigen routines te vinden zijn.
- voorzie *setters* voor elk delegate- field in de klasse. Dit geeft gebruikers van onze XMLViewer de kans om onze default methodes door zijn eigen routines te vervangen.



Figuur 2.13: XMLViewer en code plumbing

De linker versie toont het default gedrag van de XMLViewer. De rechter versie toont dezelfde informatie in dezelfde uitvoering nadat de XMLViewer control werd voorzien van aangepaste visualisatie routines.

De screenshot toont een WPF- programma. De techniek op zich is uiteraard ook bruikbaar in een WinForms context.

2.7.2 Codeplumbing pitfalls

Wanneer we codeplumbing toepassen verdienen volgende punten uw aandacht:

- vermijd delegate properties indien u usercontrols maakt. Een property wordt door Visual Studio in het property window getoond (door de property te serialiseren). In het geval van een delegate zal dit in Visual Studio voor problemen zorgen (herhalende foutboodschappen, ook na het afsluiten en heropenen van het project in Visual Studio).
- laat uw codeplumbing routines *enkel* gebruik maken van de routine parameters. De delegate wordt immers uitgevoerd waar hij werd gedefinieerd. De module niveau variabelen zijn dus soms wel (wanneer de default routines worden uitgevoerd), soms niet (wanneer de nieuwe variant wordt uitgevoerd) deze van het object dat de delegate oproept.

2.8 Delegates: faits divers

2.8.1 Action en Func

Omdat in veel situaties gelijkaardige delegate definities nodig zijn werden in .NET een aantal delegate definities voorzien. Het is aan te raden deze indien mogelijk te gebruiken, eerder dan zelf nieuwe delegate klassen te definiëren;

2.8.1.1 Action

De klasse Action is een delegate klass voor routines zonder argumenten en zonder resultaat. Action< T > is een (generische) klasse voor routines met exact één argument (gelijk welk type) en zonder resultaat.

2.8.1.2 Func

De generische klasse Func< outTResult > wordt gebruikt voor functies zonder argumenten (en gelijk welk resultaat type), en Func< inT, outTResult > wordt gebruikt voor functies met exact één argument (willekeurig type) en een willekeurig resultaat type.

2.8.2 Meer dan (begin)-invoke

Delegates ondersteunen meer dan enkel een (begin)invoke routine. Zo bestaat ook de mogelijkheid om verschillende delegates (van hetzelfde delegate type) te combineren (geen co- of contravariance mogelijk) en betaan er ook multicast delegates (die achter de schermen gebruikt worden om meerdere clients toe te laten op hetzelfde event te abonneren)

2.8.3 Timers

.NET ondersteunt verschillende Timer- classes (zie ook Comparing the Timer Classes in the .NET Framework Class Library). Indien u deze tekst doorneemt zal u begrijpen dat niet elke Timer 'tikt' op de UI-thread, wat betekent dat niet elke Timer geschikt is om UI-wijzigingen rechtstreeks aan te sturen (zie ook Asynchroon, met AsyncCallback, zonder UI).

Bemerk trouwens dat de constructor van de *System.Threading.Timer* klasse onder andere een delegate object verwacht. De twee andere timer-objecten werken met events, wat gezien Delegates en events eigenlijk niet echt een verschil uitmaakt.

2.8.4 Asynchrone eventverwerking

Wie de voorgaande topics goed begrepen heeft zou in staat moeten zijn om een eigen, asynchrone eventimplementatie op te zetten: een uitstekende oefening voor thuis.

2.8.5 Anonymous methods (.NET 2.0)

Bij het maken van een delegate instantie werd in voorgaande tekst telkens gebruikt gemaakt van een methode met gepaste signatuur. Indien men deze extra methode definitie wenst te vermijden (bijvoorbeeld bij het definiëren van MVVMCommands) kan men gebruik maken van *anonieme methodes*. Deze bestaan uit een (geschikte) formele parameterlijst definitie en een body:

```
btnAnonymousMethod.Click +=  
    delegate(object o, RoutedEventArgs e)  
        {MessageBox.Show("test1"); MessageBox.Show("test2");}  
    ;
```

Figuur 2.14: Anonymous method voorbeeld

2.8.6 Lambda expressions (.NET 3.0)

Lambda expressions kunnen onder andere gebruikt worden om delegate objecten te creëren. Ze bestaan uit twee stukken gescheiden door middel van de lambda operator `=>`. Links bevindt zich de input (het equivalent van de formele parameterlijst van een functiehoofding), rechts bevindt zich de body van de delegate. Het resultaat van de lambda expressie is (in dit geval) een delegate object:

```
btnAnonymousMethod.Click +=  
    (object o, RoutedEventArgs e) =>  
        {MessageBox.Show("test3"); MessageBox.Show("test4");}  
    ;
```

Figuur 2.15: Lambda expressie voorbeeld

Lambda expressions worden veel gebruikt in een LINQ-context, waar het injecteren van bijvoorbeeld selectielogica in een foreach-lus frequent voorkomt.

2.9 Delegates: niet voor elk programma een geschikte oplossing

- niet elk programma heeft nood aan delegates;
- (algemener:) niet elke 3nMCT-techniek is noodzakelijk om een goed programma te maken;
- (advies:) indien uw programma geen nood heeft aan delegates, gebruik ze dan niet: er zijn wellicht meer programma's te ontwikkelen zonder delegate- nood dan met.

2.10 Delegates: opgaves

- events: maak een eigen TextBox klasse met een extra event (guidelines!): dit event wordt afgevuurd indien het tekstvak leeg wordt. Demonstreer uw oplossing.
- multithreading: maak een window waarin verschillende tellers asynchroon kunnen oplopen. De teller wordt gerealiseerd door een control die erft van TextBox. Om dit te realiseren zal u de prioriteit van de visualisatie op background priority moeten uitvoeren. De demo toont de volgende elementen:
 - singlethreaded is er maar één activiteit actief;
 - multithreaded zijn er verschillende acties terzelfdertijd mogelijk: het tellen in twee textboxes en het verplaatsen van het window.
- maak een XMLViewer waarin de klant eigen routines kan inpluggen om specifieke xml's anders te visualiseren (zie ook XMLViewer). Demonstreer uw oplossing.

Jef Dael's

Hoofdstuk 3

Reflection

3.1 Reflection: duiding

Reflection is technologie waardoor (meta-) informatie omtrent programma-elementen programmatorisch toegankelijk wordt voor eigen programma's. Deze programma-elementen bevinden zich veelal in *assemblies* die niet tot het uitvoerende programma zelf behoren (externe modules). Op basis van de bekomen informatie zal ons eigen programma wellicht extra functionaliteit aanbieden. In dit hoofdstuk zullen we reflection gebruiken om een *plugin* infrastructuur op te zetten.

Hoewel reflection, net als delegates, de mogelijkheid biedt om routines te starten zou het verkeerd zijn om delegates als onderdeel van reflection te interpreteren (of omgekeerd). Een belangrijk verschil is mijns insziens de op voorhand gekende interface elementen van een delegate- instantie (*invoke*, ..) terwijl reflection net de mogelijkheid biedt om met at-compile-time ongekende klassen te werken.

Het belang van reflection kan moeilijk overschat worden. Visual Studio illustreert mooi de mogelijkheden van reflection: intellisense, het property-window, de compiler .. : veel van de programmeur ondersteunende features maken gebruik van reflection, en in het bijzonder van attributen. Ook het hoofdstuk omtrent Aspect Oriented Programming steunt sterk op reflection.

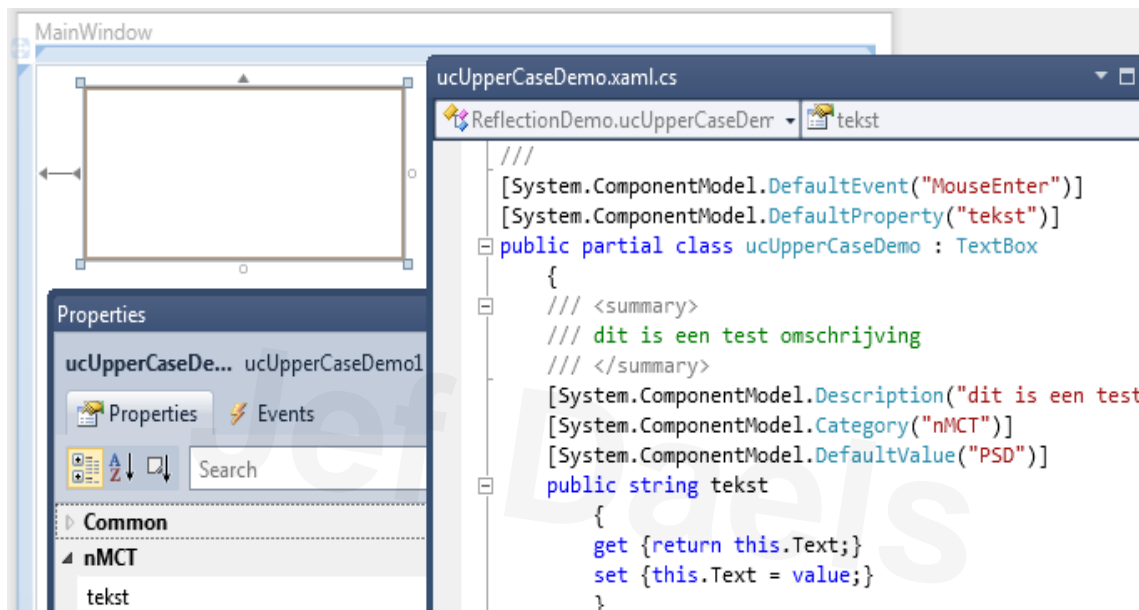
Tot slot zal de in dit hoofdstuk opgedane kennis toelaten om object oriëntatie als een syntactische constructie bovenop procedureel programmeren te zien: een inzicht dat hopelijk de blik verruimt.

3.1.1 Relevante namespaces

- *System.ComponentModel*
- *System.Reflection*

3.2 Reflection: attributen voorbeeld

Een programmeur kan code elementen voorzien van attributen om extra informatie aan het code element te koppelen. Deze attributen kunnen door andere programma's gelezen worden om een gepaste verwerking van het element mogelijk te maken. Deze verwerking staat los van de werking van het code element zelf die wellicht niet gewijzigd wordt. De namespace *System.ComponentModel* bevat attributen die door Visual Studio gebruikt worden. Een aantal van deze zijn enkel van toepassing in een WinForms context, en niet in WPF:



Figuur 3.1: ComponentModel attributen (C#)

Voor bovenstaande voorbeeld kunnen we volgende elementen vermelden:

- *DefaultEvent* en *DefaultProperty* worden enkel toegepast op klasse (usercontrol) niveau. Ze worden door Visual Studio gebruikt om in de designer volgende acties te ondernemen:
 - *DefaultProperty*: deze property wordt in het property window automatisch geselecteerd wanneer een control van deze klasse in de (Visual Studio-) designer wordt geselecteerd;
 - *DefaultEvent*: dit event wordt automatisch geselecteerd in het property window (event tab) wanneer een control van deze klasse in de (Visual Studio) designer wordt geselecteerd. Dubbel klikken op deze control zal code genereren om dit event op te vangen.
- *Category* is een attribuut waarmee de category (gebruikt in het propertywindow) van een property wordt opgegeven. Deze wordt door Visual Studio gebruikt (zie figuur);
- *Description* en *DefaultValue* zijn attributen die enkel in een Winforms Visual Studio designer omgeving gebruikt worden (de WPF designer in Visual Studio negeert deze attributen).

3.3 Reflection in .NET: meer dan attributen

3.3.1 Reflection in .NET: start

Maak een nieuwe solution met hierin een WPF-project. Maak een tweede(!) WPF project met volgende basis functionaliteit:

- voorzie een klasse *clsPersoon* met een aantal features;
- voorzie een knop op het default window om een nieuwe persoon aan te maken.

Compileer het nieuwe project en programmeer onderstaande code in het eerste project (zonder een referentie tussen beide projecten te leggen):

```
private void verwerkDemoUI()
{
    Microsoft.Win32.OpenFileDialog ofd = new Microsoft.Win32.OpenFileDialog();
    ofd.Filter = ".NET assemblies|*.exe;*.dll";
    ofd.InitialDirectory = System.IO.Path.GetDirectoryName(
        System.Reflection.Assembly.GetExecutingAssembly().CodeBase);

    if (ofd.ShowDialog().Value)
        verwerkDemoUI(ofd.FileName);
}

private void verwerkDemoUI(string sAssembly)
{
    Assembly ass = Assembly.LoadFile(sAssembly);
    foreach (Type t in ass.GetTypes())
        Console.WriteLine(t.Name);
}
```

Figuur 3.2: Assembly reflection (C#)

Indien we bij uitvoering van deze code en het tonen van de *OpenFileDialog* het compilatie resultaat van het tweede project selecteren, zal het uitvoerende programma in de console de naam van elk type in de gekozen assembly tonen: *MainWindow*, *App*, *Resources*, *Settings* en *clsPersoon*. Eenvoudig nazicht in Visual Studio leert dat dit klopt.

Bovenstaande code is een reflection startpunt: vertrekkend vanuit een assembly wordt informatie omtrent de code in de assembly opgevraagd (bijvoorbeeld de namen van de klassen). De Type-klasse ondersteunt flink wat getters: zo kunnen de attributen, constructoren, methodes, properties, events, ... worden opgevraagd. Elk van deze code elementen is terug ondervraagbaar. Het wordt sterk aangeraden om een aantal getters zelf te verwerken, ook al vind u hiervan geen afbeeldingen in deze tekst terug.

3.3.2 Hoe een instantie maken van een klasse in een assembly?

Normaal gezien zullen we, voor een gekende klasse `clsX`, volgende *C#*-syntax gebruiken:

```
clsX x = new clsX (..);
```

Figuur 3.3: Strongly typed instantiatie

In bovenstaande code wordt de gekende klasse `clsX` op twee plaatsen gebruikt: als type van de referentie en als type waarvan de constructor wordt opgeroepen. Indien we een type bekomen via reflection is bovenstaande onmogelijk: gezien het nodige type at compile time niet gekend is (`clsX` wordt gevonden in een at run-time geladen assembly) is bovenstaande lijn code onmogelijk (in *C#*), en moeten we twee problemen oplossen:

- het referentie type zullen we vervangen door *System.Object*. Omdat elke klasse (al dan niet indirect) erft van *System.Object* is elk instantie van een klasse te koppelen aan een referentie van dit root- type (polymorfisme door overerving). Natuurlijk kunnen we, indien we extra kennis omtrent de klassen hebben, eventueel een meer gespecialiseerd type gebruiken (zie het voorbeeld hieronder);
- het oproepen van de constructor van een klasse zal gecodeerd worden via de *Activator*-klasse:

```
private void verwerkIndienWindow(Type t)
{
    if (t.Name.Equals("MainWindow")) //dergelijke test vermijden => attributes
    {
        Window wdw = (Window) Activator.CreateInstance(t);
        wdw.Show();
    }
}
```

Figuur 3.4: *Activator.CreateInstance* (*C#*)

- indien de constructor argumenten vereist kunnen deze aan de *CreateInstance*- oproep worden meegegeven (u bent *niet* beperkt tot parameterloze constructoren);
- omdat we weten dat een klasse met deze naam (beslissingen op basis van een klasse naam zijn te vermijden!) een *Window* is gebruiken we een *Window* referentie waarvan de *Show*-method at compile time gekend is.

Een iets gecompliceerder voorbeeld illustreert het starten van een methode vertrekkend vanuit zijn object referentie:

```
private void verwerkIndienPersoon(Type t)
{
    if (t.Name.Equals("clsPersoon")) //dergelijke test vermijden => attributen
    {
        object p = Activator.CreateInstance(t);
        t.InvokeMember("FNaam", BindingFlags.SetProperty, null, p, new object[] { "Familie" });
        t.InvokeMember("VNaam", BindingFlags.SetProperty, null, p, new object[] { "lid" });

        string sPersoon = (string) t.InvokeMember("ToString", BindingFlags.InvokeMethod, null, p, new object[] { });
        Console.WriteLine(sPersoon);
    }
}
```

Figuur 3.5: Activator.InvokeMember (C#)

- de Type- klasse ondersteunt een *InvokeMember* methode waarmee een methode kan gestart worden. Het object dat de methode uitvoert is een argument van deze oproep, de nodige argumenten worden als een array van objecten voorzien;
- deze manier van werken is case sensitief!
- deze voorbeeld code is weinig representatief: de kans is klein dat de klasse- naam op voorhand gekend is, net zomin als men mag verwachten de methodes op voorhand te kennen. Indien de methodes op voorhand gekend zijn is het wellicht aangewezen om met een interface te werken. Op die manier is een verwerking als deze van het window (zie vorige voorbeeld) mogelijk.

3.3.3 .NET en attributes

3.3.3.1 .NET en attribute informatie opvragen

De klasse *Type* ondersteunt een *GetCustomAttributes* methode waarmee de (custom-) attributen van een klasse kunnen opgevraagd worden. Ook andere code elementen hebben een *GetCustomAttributes* property. Op basis van de extra informatie gevonden in een attribuut kan een gebruikers programma gestuurd worden. Verschillende attribute klassen ondersteunen natuurlijk verschillende properties. Een voorbeeld attribuut zullen we uitwerken in onze eigen plugin infrastructuur implementatie.

3.3.3.2 Hoe zelf Attribute klassen maken in .NET?

Een goede beschrijving kan u vinden op Writing Custom Attributes. Als opmerking hierbij zou ik formuleren dat ik verwacht dat de meeste properties *read-only* gedefinieerd zijn (en niet *read-write* zoals in de voorbeeldcode).

Er wordt zeker verwacht dat u in staat bent om attributen te definiëren die enkel op classes of enkel op methodes toegepast kunnen worden.

3.4 Een eigen plugin- infrastructuur!

3.4.1 Een eigen plugin- infrastructuur: vereisten

Ik wens volgende situatie te realiseren:

- we moeten de mogelijkheid voorzien om in een toepassing extra logica uit een op voorhand niet gekende *assembly* te laden;
- de extra logica wordt aangeboden door middel van plugins. Een plugin- klasse erft van *Window*, en alle relevante extra's worden ontsloten via deze klassen. Er kunnen *Window* klassen in de *assembly* worden gevonden die geen plugins zijn (maar bijvoorbeeld gestart kunnen worden door een plugin in dezelfde *assembly*);
- het is de bedoeling om bij het laden van een *assembly* alle gevonden plugins via het menu te presenteren. De menu-tekst voor een plugin moet ook in de *assembly* gevonden worden;
- zie demo.

3.4.2 Een eigen plugin- infrastructuur: mogelijke oplossing

Voorgaande vereisten kunnen op verschillende wijzen gerealiseerd worden. De hier geformuleerde oplossing is attribute- gebaseerd:

- het onderscheid maken (differentiëren) tussen klassen die al dan niet als plugin moeten aanzien worden kan door middel van een attribuut. Dit attribuut ondersteunt twee properties: het al dan niet plugin zijn (*isPlugin*, bool), en de omschrijving van de eventuele plugin (*description*, String). Dit resulteert in een plugin- klasse met twee fields, twee (read only) properties en een constructor met twee argumenten;
 - indien een klasse dit attribuut niet heeft is de klasse geen plugin (default).
- indien een klasse het plugin attribuut heeft, en *isPlugin* is true, dan is dit een plugin klasse waarvoor we een extra menu item zullen voorzien (in de software die de plugin-assembly verwerkt). Indien op dit menu item wordt geklikt wordt een instantie van de plugin klasse aangemaakt en getoond
 - we veronderstellen (leggen op) dat het plugin attribuut enkel is toegepast geworden op *Window*- klassen;
 - het is *niet* de bedoeling om in het client- window een boekhouding tussen menuitems en plugin- klassen bij te houden: het menu-item zelf moet bij het klikken alle nodige bewerkingen onafhankelijk kunnen afhandelen.
 - * dit laatste punt is belangrijk en lijkt soms in de totaal oefening verloren te gaan.

3.4.3 Een eigen plugin- infrastructuur: implementatie

Zowel de plugin- assembly als het client- programma (het programma dat de plugins wil laden) moeten toegang hebben tot dezelfde plugin- attribuut klasse. Omdat er geen referentie tussen de plugin- en de client- assembly mag liggen (anders is de plugin assembly at compile time gekend en is de ganse plugin- opzet overbodig) en beide assemblies toch dezelfde attribuutklasse moeten kennen zijn we verplicht het plugin- attribuut in een apart project te plaatsen (waar zowel het plugin- project als het client- project naar refereren). We hebben dus minimum drie projecten! Omtrent die drie projecten kunnen we volgende vragen stellen:

- welk project type is elk project?
- welke referenties liggen tussen deze projecten?
- waarom hebben we eigenlijk drie projecten nodig? Indien het plugin- en client project beide dezelfde bron-file voor de attribuut klasse bevatten, is dit niet voldoende?
- formuleer een oplossing met maar twee projecten (perfect mogelijk trouwens, minder goed, naam gebaseerd).

Maak in dit derde project een attribuut klasse met volgende eigenschappen:

- volg de naamgeving guideline!
- voorzie de mogelijkheid om het plugin zijn en een plugin- omschrijving te bewaren, toe te kennen via de constructor en op te vragen via read only properties;
- maak het onmogelijk om dit attribuut toe te passen op andere code elementen dan klasse definities.

Maak in het tweede project (het plugin- project dat plugins aanlevert) een window aan en koppel wat logica aan het klikken op een knop. Plaats het plugin- attribuut op de MainWindow klasse en niet op de klasse clsPersoon.

Het clientproject dat de eventuele plugins zal laden gaat als volgt tewerk:

- bij het opstarten van een window dat de plugins ter beschikking stelt (de constructor van dit window is een goede plaats) lezen we alle assembly files gevonden op een vooraf afgesproken plaats (in mijn eigen oplossing is dit de plugins subdirectory van de toepassing)
 - System.IO is uw favoriete namespace voor dergelijke zaken;
 - gebruik een filter om enkel de assembly files op te vragen;
 - in een echte toepassing zou de plugin-directory configureerbaar moeten zijn.
- de types in elke *dll*- assembly worden overlopen. Indien ze voorzien zijn van een plugin attribuut (en `isPlugin` is *true*) voorzien we een extra menu item in het (main-)menu van dit window. Dit nieuwe menu-item is een instantie van een eigen menuitem klasse:

- ze erft van de klasse *MenuItem*;
- de omschrijving getoond in het menu is de description gevonden in het plugin attribuut;
- ze implementeert een eventhandler zodat bij het klikken op een menu-item van deze klasse een instantie van het geassocieerde plugin-type wordt gemaakt en getoond;
- bepaal zelf de nodige constructor argumenten.

3.4.4 Eigen plugin infrastructuur: demo

Gedemonstreerde elementen:

- het verwijderen van plugin assemblies verwijdert de plugins uit het menu;
- indien *isPlugin false* terug geeft wordt de eventuele plugin niet getoond;
- het verwijderen van het plugin attribuut zal (na hercompilatie) de plugin niet meer tonen in het menu;
- het plaatsen van het plugin- attribuut op een niet window klasse resulteert uiteraard in een foute werking. Een goed programma zal hierop niet crashen!
- er wordt in het window met de menu-items geen koppeling tussen de menu items en het plugin type voorzien: de volledige afhandeling om een plugin window te starten dient in de eigen plugin menu item-klasse geprogrammeerd te zijn.

3.5 Namespaces met interessante attributen

- System.Diagnostics: vooral debug en performance attributes;
 - ConditionalAttribute: zeer interessant wanneer u in uw code het onderscheid wensts te maken tussen Debug- en Release- uitvoeringen (klik nu op de link :)).
- System.ComponentModel: vooral component en control gerichte klassen (zie msdn);
 - onder andere tal van convertoren (niet echt IDE stuff);
 - licensing attributen (!);

3.6 Reflection: opgaves

- ontwerp een plugin infrastructuur (analoog met de demo);
- pas het fractalen programma aan zodat het tonen van de drietallige boom en Sierpinski als plugin in een programma wordt opgenomen.

Hoofdstuk 4

Runtime compilatie

4.1 Runtime compilatie: duiding

De meeste softwareontwikkelingen bestaan uit twee stappen: er wordt code geschreven (of gegenereerd) (stap één), waarna deze gecompileerd wordt tot een uitvoerbaar programma (stap twee). Eventueel zal een toepassing verschillende projecten bevatten die elk gecodeerd en gecompileerd worden. Dit beperkt uiteraard de mogelijkheden van de toepassing: wat niet op voorhand werd geprogrammeerd (eventueel in een PlugIn) kan niet uitgevoerd worden. Indien we een tekenprogramma wensen te ontwikkelen, waarbij de gebruiker het functievoorschrift intikt hebben we een probleem (we wensen niet zelf een wiskunde parser te schrijven). Ook indien we een toepassing maken waarbij de beveiliging van sommige onderdelen door de klant wordt bepaald kunnen we niet altijd alle mogelijkheden op voorhand voorzien. Sommige database toepassingen zullen een property list in een tabel bewaren (bv.: de talen waarin een produkt gekend moet zijn, zie ook de vertalingen demo): ook hier is het niet mogelijk om op voorhand alle properties in een klasse te compileren. In dergelijke situaties is het interessant om, gedurende de uitvoering van het programma, een extra stukje code te genereren, compileren en uitvoeren: *runtime compilatie* (ook wel on the fly compilatie geheten), zodat databinding zijn rol kan spelen.

4.1.1 Relevante namespaces

- *System.CodeDom.Compiler*

4.2 .NET compilaties starten

Zowel de *C#* als de VB.NET compilers maken deel uit van de CLR. Afhankelijk van de gegenereerde code (*C#* of VB.NET) zullen we dan ook de juiste compiler moeten oproepen. We zullen de code en het compilatieresultaat (de assembly) in memory houden zodat er geen files overblijven nadat ons programma eindigt.

4.2.1 Command line compilatie

Wellicht compileerde u tot nu toe altijd met behulp van Visual Studio. Het achterliggende compilatie proces (in welke volgorde compileren, welke compilatie parameters zijn nodig) wordt volledig verborgen (een voordeel). Om het compilatie proces duidelijker te maken zullen we in een eerste stap een volledig (console-) programma editeren, compileren en uitvoeren zonder gebruik te maken van Visual Studio.

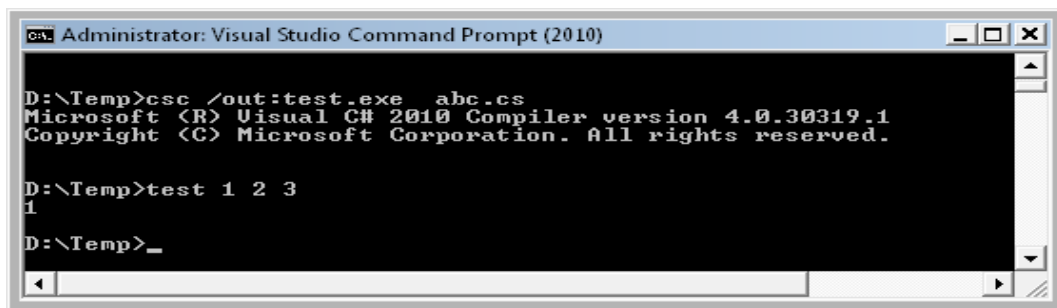
4.2.1.1 Command line compilatie: *C#*

- maak een testfile *abc.cs* met een ascii- tekstverwerker (uw code files mogen enkel programma code bevatten, een tekstverwerker als Word is dus niet geschikt) en codeer volgende elementen (*C#* is case sensitive!):
 - voorzie een gepaste Main- methode (zie ook programmeren II: elk programma start door het uitvoeren van een static Main methode; of lees *Main()* and *Command Line Arguments* (*C# Programming Guide*)). Uiteraard moet u deze methode in een klasse coderen. Anders dan in JAVA moet de klasse naam niet identiek zijn aan de filenaam;
 - * de klasse *String* zal enkel gekend zijn indien u ze prefixt met de namespace *System* of wanneer u een overeenkomstig *using* statement codeert;
 - schrijf het eerste runtime argument weg in de Console;

```
using System;  
public class Test  
{  
    public static void Main(String[] args)  
    {  
        Console.WriteLine(args[0]);  
    }  
}
```

Figuur 4.1: Commandline compilatie: *C#* broncode

- start een Visual Studio command prompt (
 - win8: tik op het startscherf *tools* en kies de *native tools command prompt*;
 - win7: Start.All Programs.Microsoft Visual Studio xxx.Visual Studio Tools.Visual Studio Command prompt(xxx);
 - `csc.exe` is de *C#* compiler. U dient deze op te roepen met de juiste set parameters (`csc /?` toont u alle mogelijke opties met een summiere toelichting). Voor ons eenvoudige programma is de `/out:` parameter eigenlijk overbodig (indien geen out- parameter wordt opgegeven heeft de resulterende assembly dezelfde naam als de bron-file);
 - met de target- parameter (niet in het voorbeeld) kan u aangeven of u een .exe (default) of .dll assembly wenst te bekomen;



Figuur 4.2: C#commandline compilatie en uitvoering

- indien ons programma gebruik zou maken van niet standaard bibliotheken moeten deze bij compilatie uiteraard ook worden meegegeven. Voor een gewoon WinForms of WPF programma wordt de parameterlijst zo groot (ongeveer elke project property komt terug als compiler optie), en is de compilatie volgorde zo belangrijk dat we dit zeker niet manueel wensen uit te voeren. Gedurende het programmeren zal Visual Studio dit voor ons doen. Indien u het proces wenst te automatiseren (om een groot project elke nacht volledig te hercompilieren) kan u zich best wat verdiepen in MSBuild.

4.2.1.2 Command line compilatie: VB.NET

We volgen gelijkaardige stappen, maar dan voor VB.NET:

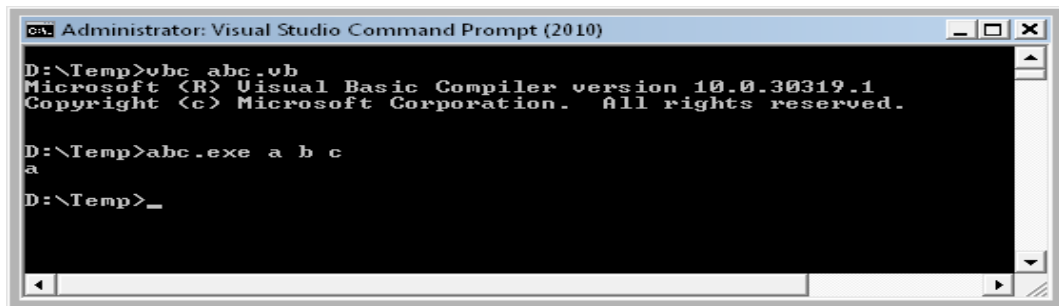
- VB.NET code:

```
imports System

public class test
    public shared sub main(args as String())
        Console.WriteLine(args(0))
    end sub
end class
```

Figuur 4.3: Commandline compilatie: VB.NET broncode

- VB.NET compilatie en uitvoering:



```
C:\> Administrator: Visual Studio Command Prompt (2010)

D:\Temp>vbnc abc.vb
Microsoft (R) Visual Basic Compiler version 10.0.30319.1
Copyright (c) Microsoft Corporation. All rights reserved.

D:\Temp>abc.exe a b c
a
D:\Temp>_
```

Figuur 4.4: VB.NET commandline compilatie voorbeeld

Het coderen en compileren is voor beide programmeertalen sterk gelijkaardig.

4.2.1.3 Een extraatje omtrent MSBuild

Het compileer proces van MSBuild gebeurt door middel van informatie gevonden in tekst files (zie ook MSBuild .Targets Files). Indien u hierin onderlegt bent kan u deze aanpassen (neen, u bent het wellicht niet). Aanpassen van deze files is de manier waarop PostSharp er in slaagt om het compilatie proces aan te passen zonder dat u als programmeur dient tussen te komen.

4.2.2 Runtime compilatie: System.CodeDom.Compiler

Het is natuurlijk niet de bedoeling om voor normale situaties een eigen compilatie proces uit te werken. Het is wel onze bedoeling om, vanuit een programma, programmacode te genereren en compileren (tot een .dll of .exe), en de bekomen assembly te verwerken zoals reeds werd gezien in reflection.

De .NET compilers zijn beschikbaar via de *System.CodeDom*- namespace:

System.CodeDom.Compiler Namespace

The **System.CodeDom.Compiler** namespace contains types for managing the generation and compilation of source code in supported programming languages. Code generators can each produce source code in a particular programming language based on the structure of Code Document Object Model (CodeDOM) source code models consisting of elements provided by the [System.CodeDom](#) namespace.

Classes

Class	Description
CodeCompiler	Provides an example implementation of the ICodeCompiler interface.
CodeDomProvider	Provides a base class for CodeDomProvider implementations. This class is abstract.
CodeGenerator	Provides an example implementation of the ICodeGenerator interface. This class is abstract.
CodeGeneratorOptions	Represents a set of options used by a code generator.
CodeParser	Provides an empty implementation of the ICodeParser interface.
CompilerError	Represents a compiler error or warning.
CompilerErrorCollection	Represents a collection of CompilerError objects.
CompilerInfo	Represents the configuration settings of a language provider. This class cannot be inherited.
CompilerParameters	Represents the parameters used to invoke a compiler.
CompilerResults	Represents the results of compilation that are returned from a compiler.
Executor	Provides command execution functions for invoking compilers. This class cannot be inherited.
GeneratedCodeAttribute	Identifies code generated by a tool. This class cannot be inherited.
IndentedTextWriter	Provides a text writer that can indent new lines by a tab string token.
TempFileCollection	Represents a collection of temporary files.

Figuur 4.5: C#System.CodeDom.Compiler namespace

Bemerk dat er geen aparte klassen per programmeertaal zijn:

- de klasse *CodeDomProvider* geeft toegang tot de .NET- compilers. De static methode *CreateProvider* heeft een argument waarmee wordt aangegeven voor welke programmeertaal we een compiler wensen te bekomen;
 - indien u twijfelt omtrent de taal-parametertekst (c# of csharp? VB of VB.NET? case sensitive?) kan u deze bekomen door de static methode *GetLanguageFromExtension* op te roepen (de file extentie moet u dan uiteraard wel kennen);
- *CompilerParameters* is handiger dan de commandline benadering: de verschillende parameters zijn niet als een lange tekst te formuleren maar zijn properties op een *CompilerParameters* object;
 - het toevoegen van een assembly kan gebeuren door middel van zijn naam: *System.Data.dll*
 - een assembly waarin een type XYZ zich bevindt kan worden teruggevonden door *typeof(XYZ).Assembly.Location*

- *CompilerResults* bevat het compilatie resultaat. Dit bevat zowel de compilatie feedback (error, warning, ..) als de (eventueel) resulterende assembly;

4.3 CodeDom

De eerder aan bod gekomen oplossing verwacht broncode die gecompileerd kan worden: deze is natuurlijk programmeertaal afhankelijk. In een andere benadering kunnen we een programmeer boom maken waarin programmeer- entiteiten worden gehangen (types, methodes, variabelen, commando's, ..) waarvoor achteraf programma code (in de programmeertaal naar keuze) wordt gegenereerd:

```
CodeCompileUnit ccunit = new CodeCompileUnit();
CodeNamespace cns = new CodeNamespace("taalprog");
ccunit.Namespaces.Add(cns);

CodeTypeDeclaration klas = new CodeTypeDeclaration("klasnaam");
cns.Types.Add(klas);
klas.Attributes = MemberAttributes.Public;
```

Figuur 4.6: CodeDOM

We spreken in deze context over een abstract syntax tree.

4.4 System.Reflection.emit

Voor de volledigheid vermelden we ook de System.Reflection.Emit Namespace. Deze laat toe om bij uitvoering *IL*- code genereren (normaal gezien het resultaat van een compilatie). Dit valt jammer genoeg buiten het bereik van deze cursus en mijn kennis.

4.5 Het Roslyn project

Een interessante ontwikkeling in het compilatie gebeuren van .NET is de ontwikkeling van het Roslyn project (momenteel nog in CTP (community technical preview)). Hierbij worden tussentijdse resultaten (de abstract syntax tree, koppeling tussen namen en objecten, ..) van het compilatie proces ontsloten naar ontwikkelaars toe. Dit biedt gigantisch veel mogelijkheden naar geautomatiseerde code processing (zeker naar rapportering toe, wellicht ook naar het automatisch aanpassen van code toe). Voor deze cursus wordt dit op termijn wellicht een belangrijk onderdeel.

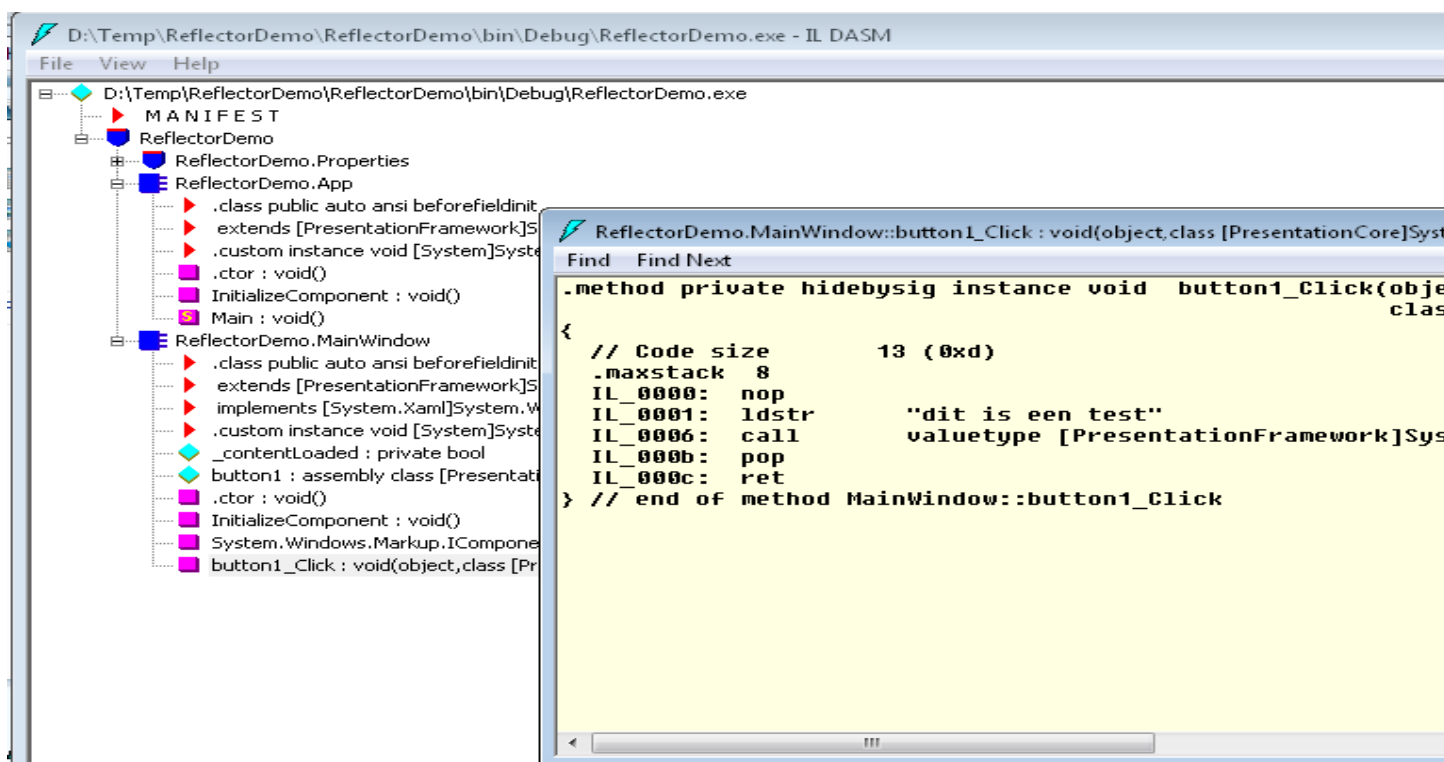
4.6 Assembly code inspecteren

Het resultaat van een .NET compilatie is (indien gelukt) een assembly. Deze bestaat uit *IL* (Intermediate language) code: de machinetaal van .NET. Deze assemblies kunnen door middel van disassemblers gemakkelijk gedecompileerd worden. Dit laat derden toe om uw broncode grotendeels te reproduceren wat meestal niet wenselijk is. Het gebruik van obfuscators kan de leesbaarheid van de gedecompileerde broncode sterk omlaag halen.

Deze problematiek is niet .NET specifiek: ook byte code (het java equivalent van IL- code) kan gedecompileerd worden. Ook voor C(++) output zijn er disassemblers te vinden. In de .NET context zal u merken dat u een assembly kan decompileren naar uw programmeertaal naar keus, wat ook de oorspronkelijke programmeertaal was.

4.6.1 ILDasm

ILDasm is de *IL* disassembler meegeleverd met *Visual Studio*. Deze kan gebruikt worden om *IL*- code te tonen in een textfile formaat. Op deze manier is het mogelijk om bijvoorbeeld de aanwezigheid van attributen na te kijken. Hoewel *ILDasm* zeker zijn plaats heeft (zie ook *Ildasm.exe* (MSIL Disassembler)) is het zeker niet de meest geschikte tool om de brontekst van een programma na te kijken:



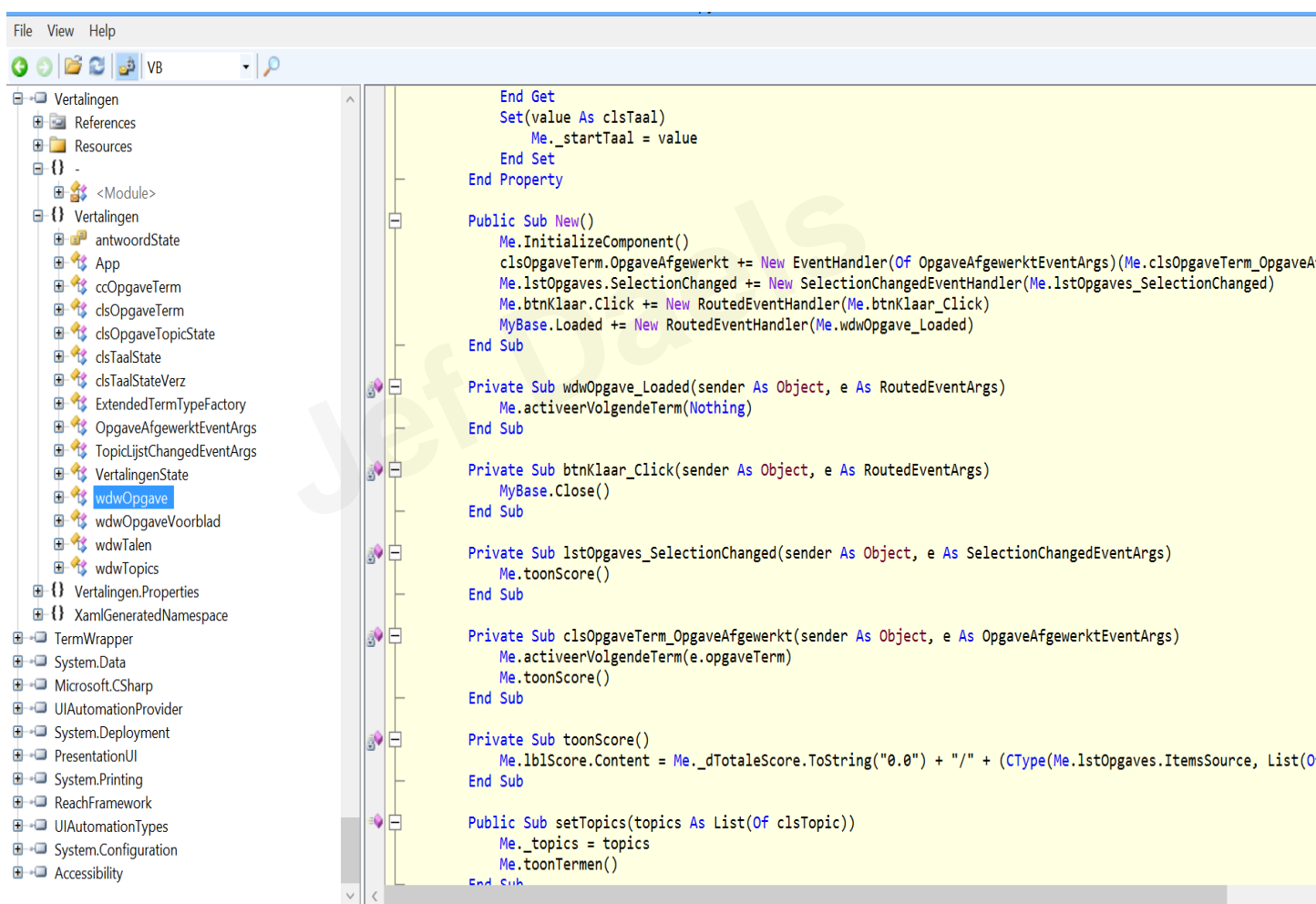
Figuur 4.7: ILDasm output

4.6.2 ILSpy

ILSpy is het gratis antwoord waarmee werd gestart nadat de voorheen gratis Reflector software betalend werd. Voor zover ik beide oppervlakkig gebruikt heb vind ik ze sterk gelijkwaardig: beide laten toe om een door ons gecompileerd programma om te vormen naar een .NET-taal naar keuze.

Bij het downloaden van ILSpy kan u kiezen tussen de broncode en de gecompileerde bestanden. Ikzelf verkies meestal de broncode, maar het staat u uiteraard vrij om de binaries te installeren.

Gelieve na het installeren *ILSpy* eens op te starten en een programma te decompileren:

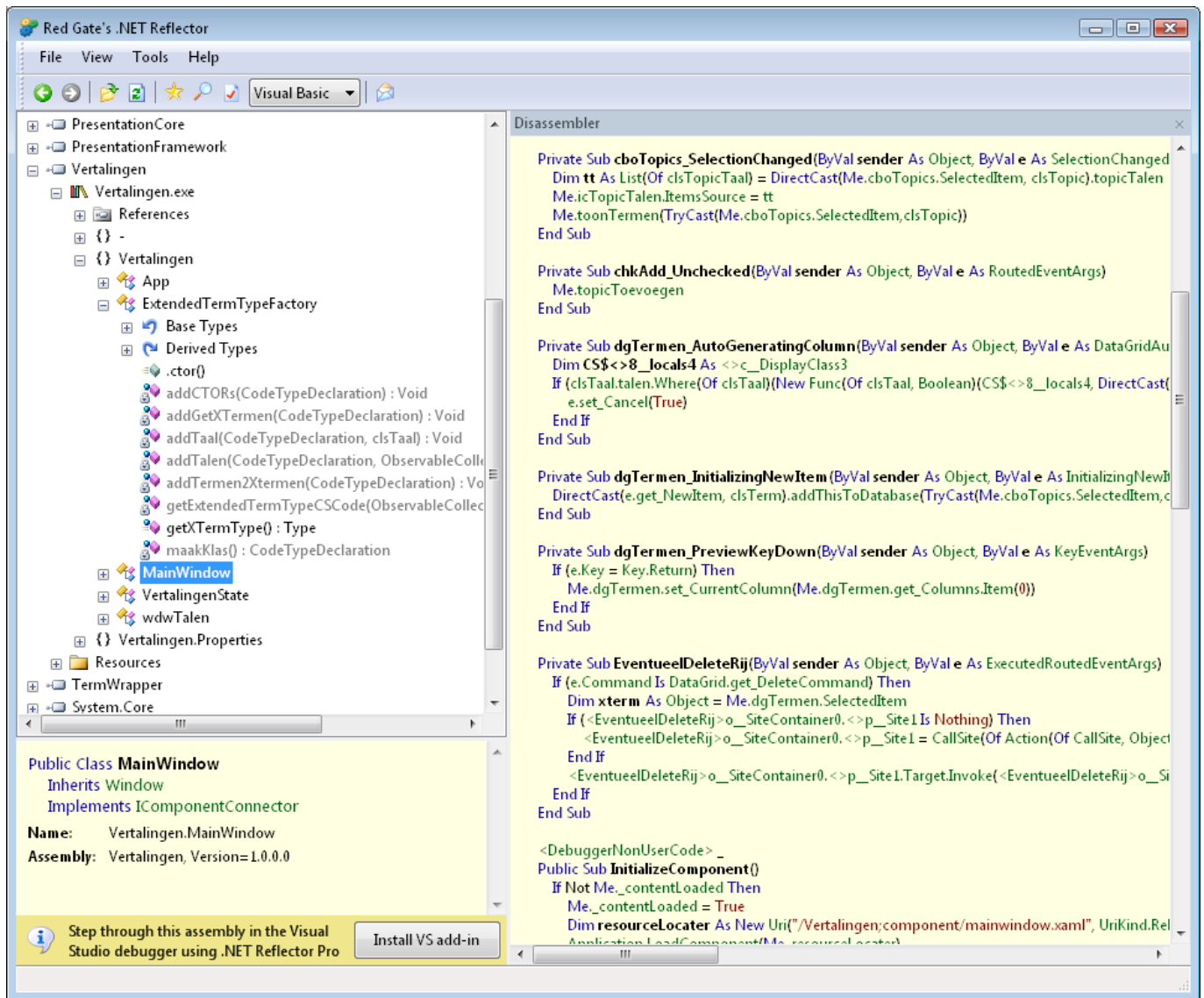


Figuur 4.8: ILSpy output

Bemerkt dat in bovenstaande figuur VB.NET- code werd gegenereerd terwijl de oorspronkelijke programmatekst in *C#* werd geschreven.

4.6.3 Reflector

Deze (niet langer gratis) tool is (niet langer) wat u zoekt! Hiermee bent u in staat om een assembly te decompileren in *de programmeertaal naar keuz*. Indien u de nodige plugins installeert kan u een assembly volledig decompileren tot een project (sommige namen zullen wel niet compileerbaar zijn, maar dat is voor kwaadwillige derden slechts een kleine beperking):

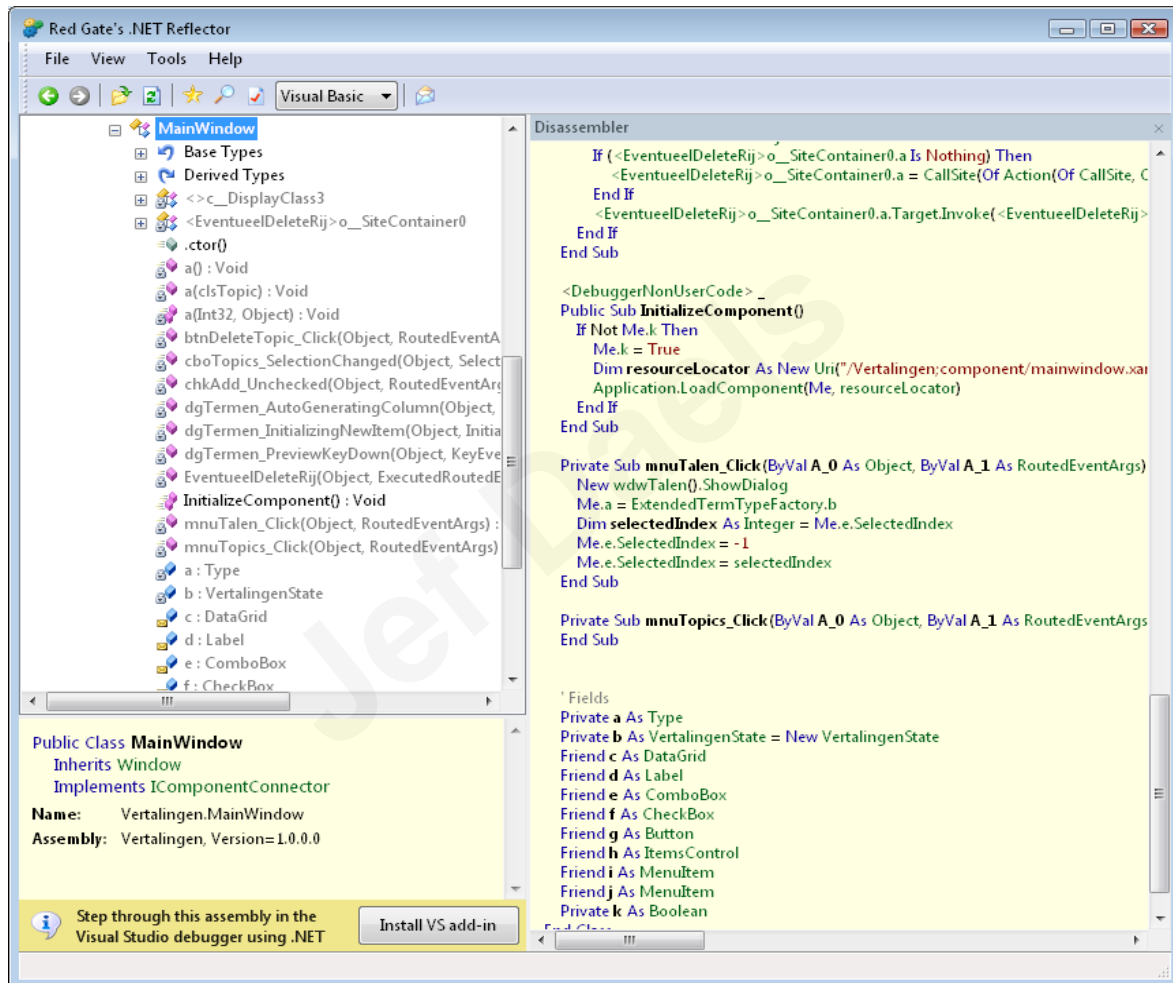


Figuur 4.9: Reflector output

Bemerk dat in bovenstaande figuur VB.NET- code werd gegenereerd terwijl de oorspronkelijke programmatekst in C# werd geschreven. In de programma code zijn de groene tekst elementen hyperlinks (niet beschikbaar in ILSpy): hiermee kan u navigeren naar de definitie van dit element (ongeveer zoals F12 in Visual Studio).

4.6.4 Obfuscatie

Obfuscatie is een techniek om een assembly aan te maken die ook na decompilatie quasi onleesbaar is. De (gratis) versie die geïnstalleerd wordt met Visual Studio (onder tools) vertrekt van een compilatie resultaat en zal alle hierin gevonden namen (klassen, methodes, parameters, ..) *scramblen*, wat het geheel voor een complex programma quasi onleesbaar maakt. Het is uiteraard niet wenselijk om de broncode van het project zelf te scramblen, omdat u in dat geval geen aanpassingen meer zou kunnen doorvoeren.



Figuur 4.10: Obfuscated output via Reflector

Bemerk in bovenstaande figuur het hernoemd zijn van de controls. Veel routine namen zijn niet hernoemd geworden, wat spijtig is. Wellicht zal de betalende versie hier een betere obfuscatie mogelijk maken.

4.6.4.1 Confuser

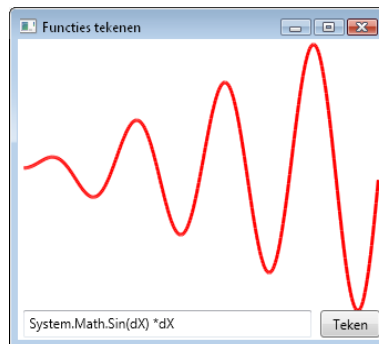
Een betere obfuscator vindt u op Confuser. Het resultaat van deze obfuscator kan niet geopend worden in ILDASM en is totaal onleesbaar wanneer u bijvoorbeeld ILSpy gebruikt.

4.7 Opgaves

4.7.1 Een tekenprogramma

Het te realiseren programma moet aan volgende vereisten voldoen:

- de gebruiker kan zelf een functievoorschrift intikken (reden waarom we on-the-fly compilatie nodig hebben: we zullen dit functievoorschrift compileren zodat we het kunnen uitvoeren);
- het tekenen gebeurt in een usercontrol, waarbinnen de tekening geschaald wordt getoond: we rekken (of krimpen) de te tekenen figuur zodat de bovenkant van de figuur de bovenkant van de usercontrol raakt (idem voor de onderkant);



Figuur 4.11: Tekenend van een functie

4.7.2 ILSpy

Op het examen wordt verwacht dat u volgende manipulaties vlot kan uitvoeren:

- installeer ILSpy en pas deze toe op een assembly naar keus;
- demonstreer dat u kan decompileren naar verschillende programmeer talen.

4.7.3 Obfuscatie

Op het examen wordt verwacht dat u volgende manipulaties vlot kan uitvoeren:

- gebruik *Confuser* om een assembly naar keus te obfuscaten;
- gebruik de obfuscator (gevonden onder tools) om een assembly naar keus te obfuscaten;
- gebruik ILSpy om het obfuscated zijn van uw nieuwe assembly te demonstreren.

Jef Daele

Hoofdstuk 5

Backtracking

5.1 Backtracking: duiding

De eerste computers werden oorspronkelijk gebruikt om oplossingen te bieden voor problemen die mensen niet (snel genoeg) konden oplossen. Dit hoofdstuk behandelt het gebruik van een computer om problemen op te lossen waarvoor we zelf, zonder computer, geen antwoord kunnen formuleren. De gebruikte techniek heet *backtracking*: hij is niet op alle problemen toepasbaar, is soms hopeloos inefficiënt maar is een haalbare kaart om artificiële intelligentie te introduceren. In de laatste sectie zullen we backtracking gebruiken om AI in een spelletjes context te introduceren.

5.1.1 Relevante namespaces

5.1.2 Backtracking: principe

Backtracking is een benadering waarbij men, door het overlopen van alle mogelijke oplossingen, zoekt naar een beste (eerste, ..) oplossing. De implementatie van het algoritme is meestal recursief, met hierin volgende stappen:

- *stopconditie*: indien de oplossing werd gevonden geven we true terug;
- anders worden alle mogelijkheden voor deze stap in een lus overlopen:
 - voer één van de stap mogelijkheden uit;
 - indien de recursieve oproep voor de volgende stap slaagt hebben we een oplossing gevonden: we geven true terug;
 - indien de recursieve oproep faalt maken we de zopas gezette stap ongedaan en vervangen deze door één van de andere mogelijkheden, en herhalen de recursieve stap;
- indien geen enkele van de mogelijkheden tot een oplossing leidt moeten we concluderen dat de eerder gezette stappen het vinden van een oplossing blokkeren, en keren we terug naar de oproeper (return false), die dan op zijn beurt een volgende mogelijkheid kan onderzoeken.

5.2 N-Queens probleem

5.2.1 Probleemstelling, synchrone oplossing

Een gemakkelijk uit te leggen backtrack probleem is het N-Queens probleem: hoe plaats ik N koninginnen op een schaakbord zodat deze elkaar niet slaan. Twee koninginnen slaan elkaar indien ze:

- op dezelfde kolom staan;
- op dezelfde rij staan;
- uit de eerste koningin een diagonaal vertrekt waarop de tweede koningin zich bevindt.

Een geïllustreerde probleemstelling vind u op Eight queens puzzle. De toelichting van het algoritme gebeurt tijdens de les.



Figuur 5.1: NQueens oplossen

5.2.2 Asynchroon: BeginInvoke/4

Indien we grotere aantallen (bijvoorbeeld 30) koninginnen dienen te plaatsen neemt de berekening flink wat tijd in beslag. Om de interface hiermee niet te blokkeren zullen we de berekening van de oplossing op een andere thread laten uitvoeren. Hiertoe gaan we als volgt te werk:

- maak een delegate-class voor de zoekroutine;
- start een asynchrone oproep voor deze zoekroutine, en voorzie een callback waarmee het berekende resultaat in de UI kan getoond worden (u zal hoogst waarschijnlijk een state object nodig hebben);
- laat de asynchrone thread de berekende oplossing in de UI tonen (let op de UI-thread pitfall!).

Hint: om te vermijden dat gedurende de asynchrone zoekoperatie een volgende zoekoperatie gestart zou worden (er kan immers geklikt worden in de UI terwijl de zoekoperatie loopt) is het best om de UI-elementen die de zoekopdracht starten of specificeren disabled worden geplaatst: het is immers niet wenselijk de zoek- informatie te wijzigen terwijl de vorige zoekopdracht nog loopt (de getoonde resultaten zouden niet langer overeenstemmen met de zichtbare criteria) net zomin als we een volgende zoekopdracht willen starten terwijl de vorige nog niet ten einde was. Als oefening kan eventueel een cancel- button voorzien worden.

5.2.3 Asynchroon: async/await

Hiervoor verwijs ik naar Await/Async verder in deze tekst. Deze is ook bruikbaar voor de Sudoku- toepassing die straks aan bod komt.

5.3 Sudoku's oplossen

Het oplossen van Sudoku's is niet altijd eenvoudig. Een computerprogramma kan u behulpzaam zijn, maar ontnemt u wellicht het liefhebbers plezier.

5.3.1 Sudoku: spelregels

Het invullen van een 9x9 sudoku gebeurt op volgende wijze:

- elk hokje van de matrix moet worden ingevuld met een geheel getal tussen 1 en 9 (grenzen inbegrepen);
- in een rij mogen geen dubbels voorkomen;
- in een kolom mogen geen dubbels voorkomen;
- in elk van de negen 3x3 hokjes (naast elkaar, niet overlappend) mogen er geen dubbels voorkomen;
- een opgave toont een 9x9 rooster met een aantal ingevulde velden:

		5	2	3	6		8	9
						2		
2								
3			4					
			8	9		3	2	7
	9		3			4	5	
		1	6	8				
					5			
		3	7					

Figuur 5.2: Sudoku spelregels: een opgave

- de oplossing is gevonden wanneer de speler het rooster correct aanvult:

1	4	5	2	3	6	7	8	9
6	3	7	9	1	8	2	4	5
2	8	9	5	7	4	1	3	6
3	1	2	4	5	7	6	9	8
4	5	6	8	9	1	3	2	7
7	9	8	3	6	2	4	5	1
5	2	1	6	8	3	9	7	4
9	7	4	1	2	5	8	6	3
8	6	3	7	4	9	5	1	2

Figuur 5.3: Sudoku spelregels: een oplossing

5.3.2 Software vereisten

- UI vereisten:
 - enkel numerieke input, beperkt tot één positie is mogelijk. Het is niet de bedoeling feedback te geven indien de gebruiker hiervan afwijkt, het is de bedoeling dat hij er niet kan van afwijken (zie ook TXTINT);
 - er is een intuïtieve navigatie op basis van de pijltjes mogelijk;
 - er is kleurverschil tussen opgave en oplossing;
 - bij ingave van een foutief getal wordt dit (discreet) gekleurd.
 - de gebruiker kan de opgave niet wijzigen gedurende het oplossen
- Sudoku vereisten:
 - we kunnen opgaves ingeven en bewaren (op file), en deze later terug inlezen;
 - een opgave kan door het programma correct worden opgelost (indien er een oplossing bestaat).

5.3.3 Sudoku oplos algoritme

De recursieve oproep *zoekOplossing(Cell cell, int[,]bord)* kent twee parameters:

- *cell* is de cell die we nu wensen in te vullen. Deze stuurt de stopconditie: indien de cel die we wensen in te vullen buiten het bord valt zijn alle bord cellen ingevuld en hebben we een oplossing;
 - *Cell* is een eigen klasse (we houden rij- en kolom- waardes bij (Integers). De klasse Point werd niet gebruikt omdat deze doubles bevat);
- *bord* bevat de bordsituatie zoals ze nu gekend is. Deze situatie hebben we nodig om het correct zijn van een nieuwe zet te beoordelen. Omdat in het bord de oplossing die we zoeken verder wordt opgebouwd bij elke recursieve stap spreken we van een *accumulerende parameter*.

- het return type is *boolean*: *true* indien een oplossing wordt gevonden, *false* anders.

Het algoritme gaat als volgt:

- indien *cell* buiten het bord valt zijn alle velden ingevuld en hebben we een oplossing in *bord*;
- overloop elke mogelijke waarde voor *cell* en voer volgende stappen uit:
 - vul de waarde in op de nieuwe positie;
 - * indien dit een Sudoku probleem veroorzaakt gaan we over naar de volgende waarde;
 - * indien dit geen probleem veroorzaakt voeren we een recursieve oproep uit voor de volgende cel. Indien deze oproep *true* teruggeeft vonden we een oplossing (in *bord*) en returnen we zelf ook *true*. In het andere geval (de recursieve oproep levert *false*) was het onmogelijk om de rest van het bord in te vullen op een correcte manier;
 - verwijder de zopas gezette waarde en probeer een volgende mogelijkheid;
- indien na het overlopen van alle mogelijkheden nog geen oplossing werd gevonden moeten we *false* terug geven: het was niet mogelijk de Sudoku verder op te lossen: we returnen *false* waardoor de oproeper zijn laatste poging ongedaan maakt en een andere poging zal ondernemen.

Heb speciaal aandacht voor volgende elementen:

- een zet die niet voldoet moet terug van het bord genomen worden;
- indien een oplossing wordt gevonden mag de *bord* parameter (die deze oplossing bevat) niet meer gewijzigd worden.

Dit algoritme wordt ook aan *bord* op basis van voorbeelden ontwikkeld. Ik heb niet de ambitie backtracking volledig schriftelijke toe te lichten.

5.3.3.1 Constraint satisfaction solver

Voorgaande is niet de enige manier om Sudoku's op te lossen. Een interessante alternatieve benadering wordt beschreven in *Sudoku using Microsoft solver foundation*. Het onderliggende algoritme zou opnieuw backtracking kunnen zijn.

5.4 Sudoku routed events introductie

Het is de bedoeling om pijltjes navigatie op het Sudoku bord toe te laten: we vangen het *PreviewKeyDownEvent* op en voeren de eventuele gepaste navigatie uit. Elke *TextBox* op het Sudoku-bord (er zijn er 81) kan de bron van dergelijk event zijn: in een klassieke benadering moeten minstens 81 eventhandlers gekoppeld worden. Indien men beslist andere controls dan *TextBoxen* te gebruiken moet men events van deze andere controls opvangen. Om in dergelijke situaties een elegante oplossing te bieden werden *routed events* geïntroduceerd:

- een WPF-control kan zich abonneren op een *routed event* gedefinieerd op een andere klasse (zoals een DependencyObject een DependencyProperty gedefinieerd op een andere klasse kan hebben):

```
public ccSudoku9x9()
{
    this.Background = new SolidColorBrush(Colors.Transparent);
    initSudoku9x9();
    zetContextMenu();

    this.AddHandler(TextBox.TextChangedEvent, new TextChangedEventHandler( text_changed));
    this.AddHandler(TextBox.PreviewKeyDownEvent, new KeyEventHandler(keyDown));
}
```

Figuur 5.4: Sudoku routed events demo

- de visual tree wordt eerst van container naar child elementen doorlopen (de *preview-events*): de *tunneling* fase;
- daarna wordt de visual tree van child naar container element doorlopen (de gewone event namen): de *bubbling* fase;
- indien een eventhandler (zowel tunneling als bubbling) het event afhandelt kan de verdere eventhandling afgebroken worden (handled=true op het argument e);
 - dit afbreken is op zijn beurt terug te omzeilen;
 - door het toepassen van templates en styles kan de visual tree behoorlijk complex worden. Toch wordt hij volledig doorlopen.
- hoewel *ccSudoku9x9* noch een *TextChangedEvent* noch een *PreviewKeyDownEvent* kent kan hij toch subscriben op deze routed events. In zoverre een element in de visual tree deze events afvuurt zal de *ccSudoku9x9* control aan de bubbling- en tunneling fases deelnemen;
 - gebruik *e.OriginalSource* om te achterhalen welke control het event veroorzaakte.

5.5 Async methodes: TaskCompletionSource

Indien opfrissing omtrent await/async nodig verwijs ik naar de Random topics: awaitable methodes.

Zoals in het hoofdstuk omtrent delegates werd aangekaart kan met behulp van de klasse TaskCompletionSource een awaitable methode ontwikkeld worden.

Het .NET framework 4.5 bevat veel awaitable methodes, herkenbaar aan de suffix Async (zie ook Random topics: awaitable methodes). Om dergelijke methode asynchroon uit te voeren, en de rest van de routine definitie in een callback methode voor die oproep te plaatsen volstaat het om await voor de methode oproep te plaatsen. Meer nog, de rest code volgend op de awaited

oproep wordt automatisch terug op de huidige thread geplaatst (meestal de UI-thread).

Het gebruik van een awaitable methode is gemakkelijk, het opzetten van een awaitable methode is dit minder. De volgende tekst beschrijft hoe een awaitable methode kan aangemaakt worden.

5.5.1 Task- klassen

Een awaitable methode heeft *altijd* een Task- object terug. Indien het return type van een synchrone oproep *void* is, is het return type van de asynchrone versie *Task*. Indien het return type van de synchrone uitvoering *T* was geweest zullen we *Task< T >* als resultaat type gebruiken van de asynchrone oproep.

Indien de oproep wordt *awaited* mag het resultaat van de oproep (te vinden in de *Result*-paramter van de Task) als resultaat van de await gebruikt worden (de compiler lost dit op):

```
Boolean gevonden = await ZoekOplossingAsync(oplossing,0,mogelijkeKolommen);
if (gevonden)
    ToonOplossing(oplossing);
```

Figuur 5.5: Awaiting een oproep

Zowel de NQueens- als Sudoku oplosser geeft als resultaat een Boolean terug. Het resultaat type van de Async- versie zal dan ook *Task< Boolean >* zijn.

5.5.2 TaskCompletionSource- klassen

De *await*- infrastructuur dient geïnformeerd te worden omtrent het verloop van de asynchrone uitvoering: voltooiing, onderbreking, voortgang en dergelijke meer zijn belangrijk. Omdat deze informatie niet enkel via een callback kan gecommuniceerd worden werd een nieuwe generische klasse geïntroduceerd: *TaskCompletionSource*:

- de informatie omtrent de uitvoering van een asynchrone taak wordt verzameld in een TaskCompletionSource- object;
- dergelijk object wordt door ons programma als field gedefinieerd zodat het kan gemanipuleerd worden door onze routines;

```
private void ZoekOplossingCompleted(IAsyncResult ar)
{
    Cel[] cellen = ar.AsyncState as Cel[];
    if (cellen[cellen.Length - 1] != null) //er was e
        _tcsOplossing.SetResult(true);
    else
        _tcsOplossing.SetResult(false);
}
```

Figuur 5.6: TaskCompletionSource: SetResult

- de start van onze asynchrone methode initialiseert dit field, start het asynchrone proces (met klassieke callback) en geeft de Task- property van dit object als resultaat terug;
- het is nu de bedoeling dat onze asynchrone methode (of zijn callback) na uitvoering dit object als completed of cancelled gaat instellen. De compiler heeft code gegenereerd die hierop inpikt om dan de logica die wachtte verder uit te voeren.

Wellicht kan een codevoorbeeld dit verduidelijken:

```
private delegate Boolean ZoekOplossingDelegate(Cel[] vorigeZetten, int
private TaskCompletionSource<Boolean> _tcsOplossing;
private Task<Boolean> ZoekOplossingAsync(Cel[] vorigeZetten, int huidi
{
    _tcsOplossing = new TaskCompletionSource<Boolean>();

    ZoekOplossingDelegate del = new ZoekOplossingDelegate(ZoekOplossin
    del.BeginInvoke(vorigeZetten, huidigeRij, mogelijkeKolommen, ZoekOplo

    return _tcsOplossing.Task;
}
```

Figuur 5.7: TaskCompletionSource demo

- ZoekOplossingDelegate is de delegate klasse om het zoeken asynchroon te programmeren;
- tcsOplossing is het resultaat van de awaitable methode. Ze wordt als field gedefinieerd zodat mijn logica deze kan aansturen;
- ZoekOplossingAsync is de awaitable methode (herkenbaar aan het return-type en de suffix van de methode). Deze methode zal intern een asynchrone oproep starten en geeft onmiddellijk een Task-object terug. De compiler genereert code die in de oproeper de rest van de logica uitvoert nadat deze task voltooid wordt. Het voltooien van de Task gebeurt door het resultaat in te stellen op het *TaskCompletionSource*- object waar de Task onderdeel van uitmaakt;

5.6 Het MiniMax algoritme (geen examen stof)

Backtracking wordt onder andere gebruikt als Artificiële Intelligentie algoritme bij het ontwikkelen van (turn-based) games. Het *MiniMax* algoritme is hiervan een voorbeeld. De Engelstalige wikipedia-entry voor MiniMax vind ik inhoudelijk sterker dan de Nederlandstalige.

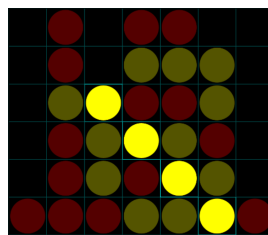
5.6.1 MiniMax benadering voor vier op een rij

Het MiniMax algoritme realiseert volgende doelstelling: indien de speler aan zet niet wint met zijn volgende zet, dan minimaliseert hij de kansen van de andere speler. De andere speler zal bij het overlopen van zijn eigen mogelijkheden, indien hij niet direct wint, veronderstellen dat zijn tegenstander (speler 1) geen fouten maakt, en diens score maximaliseren, vandaar de naam van het algoritme. Omdat we bordsituaties met elkaar vergelijken (om een beste of slechtste situatie te kunnen kiezen) zullen we een BordWaarde- functie nodig hebben die de waarde van een bordsituatie weergeeft.

5.6.1.1 BordWaarde

- de parameters zijn die waardes nodig om een bordsituatie te beschrijven;
- het resultaat is een numerieke waarde, zodat vergelijkingen mogelijk worden:
 - +1: indien de speler aan zet uitspeelt;
 - -1: indien de tegenspeler uitspeelt;
 - 0: anders.
- de BordWaarde- functie zal intens gebruikt worden: elke mogelijke stap zal geëvalueerd worden door deze functie. Het is dan ook belangrijk om deze functie zo efficient mogelijk te maken.

Het is contra- intuïtief om de bord- waarde te beperken tot 3 waardes: winst, verlies of onbeslist. Immers, wanneer we zelf vier op een rij spelen streven we bijvoorbeeld naar een situatie waar twee aansluitende rijen in een kolom tot winst leiden (zie kolom 3 in onderstaande figuur). In onze beoordeling van een bord is dit duidelijk een zeer goede situatie. MiniMax gewijs zal de AI, indien hij drie zetten vooruit *denkt* (rekent) ook merken dat in elke mogelijke invulling van een dergelijke situatie we een winnende zet kunnen realiseren, wat van dergelijke bord- situatie een winnende situatie maakt (+1). In onderstaande figuur werd de situatie in kolom 3 hierdoor als winnend herkend (bij de vorige zet) omdat de AI minstens drie zetten vooruit denkt:



Figuur 5.8: 4 op een rij: bordwaarde

5.6.1.2 BesteZet

De functie `BesteZet` berekent voor een gegeven bord situatie de best volgende zet voor een speler. In een vier op een rij context is het resultaat van de oproep een kolom nummer (int).

- het resultaat van de oproep is het kolomnr waarin de speler zijn fiche moet plaatsen. De parameters van de oproep worden bepaald door de nood van de hieronder geformuleerde logica;
- één van de argumenten van de oproep, *depth* geeft aan hoeveel stappen de computer vooruit mag denken. Hoe meer stappen de computer vooruit denkt, hoe langer de berekening zal duren.
- de speler aan zet (de maximizing speler) overloopt elke mogelijke kolom en voert volgende zaken uit:
 - plaats een fiche in deze kolom;
 - indien vier op een rij werd gevonden hebben we een winnende kolom: *return kolomnr*
 - indien geen directe winst wordt de waarde van deze kolom bepaalt als `MiniMax(maximizingSpeler, minimizingSpeler, depth -1, ..)`.
 - indien de MiniMax- waarde kleiner is dan het huidige minimum wordt deze zet het nieuwe minimum (bewaar zowel de kolom als het minimax resultaat);
 - verwijder deze fiche terug
- de kolom met de kleinste MiniMax waarde (de slechtst mogelijke zet voor de tegenstander) wordt als resultaat teruggegeven.

Om wat random zetten aan het algoritme toe te voegen kunnen we volgende zaken aanpassen:

- we houden niet enkel de eerste beste zet bij, maar alle zetten die de huidige beste waarde als resultaat hebben;
- bij het teruggeven van de beste kolomwaarde wordt een random kolomnummer uit de lijst gekozen.

5.6.1.3 MiniMax methode

De Engelstalige wikipedia-entry voor MiniMax beschrijft het MiniMax algoritme als volgt:

```
function minimax(node, depth, maximizingPlayer)
  if depth = 0 or node is a terminal node
    return the heuristic value of node
  if maximizingPlayer
    bestValue := -∞
    for each child of node
      val := minimax(child, depth - 1, FALSE)
      bestValue := max(bestValue, val);
    return bestValue
  else
    bestValue := +∞
    for each child of node
      val := minimax(child, depth - 1, TRUE)
      bestValue := min(bestValue, val);
    return bestValue
```

Figuur 5.9: Minimax algoritme

- een *terminal node* is een zet die het spel afrondt: één van beide spelers realiseert vier op een rij. We veronderstellen dat enkel de speler die nu een fiche plaatst kan uitspelen, en dat zijn huidige zet deel uitmaakt van de oplossing (dat wil zeggen: we veronderstellen dat er zijn geen andere vier op een rij oplossingen zijn).
- u moet weten wie de maximizing speler is opdat u weet of de speler aan zet de mogelijkheden van zijn tegenstander minimaliseert of maximaliseert;
- die MiniMax algoritme overloopt *alle* mogelijke zetten (tot eindspelsituaties) voor beide spelers, wat in veel spelsituaties een gigantisch aantal mogelijkheden betekent. Het beperken van die set mogelijkheden, zonder de beste oplossing te verliezen is dan ook zeer wenselijk.

5.6.2 Minimax optimalisaties

5.6.2.1 Alfa-Beta pruning

Indien u het MiniMax algoritme correct kon implementeren (geen onderdeel van de cursus), dan zal u wellicht geïnteresseerd zijn in Alfa-Beta pruning voor de MiniMax zoek boom. De uitleg en het bijhorende voorbeeld onderaan de webpagina kan ik niet verbeteren.

Een nadeel van maximaal pruning is het verliezen van de alternatieve beste zetten. Dit kan als gevolg hebben dat de AI zijn zetten nogal éézijdig in het speelveld plaatst: indien de kolommen van links naar rechts worden doorgelopen zullen de eerste beste oplossingen zich altijd links ten opzichte van hun alternatieven bevinden. Omdat enkel de eerste beste oplossing wordt weerhouden wordt het speelveld vooral links ingevuld.

5.6.2.2 Multi- threading

Op een multi core toestel zal het single- threaded berekenen van de beste zet geen optimaal gebruik maken van de beschikbare rekenkracht. Het op een aparte thread plaatsen van de BesteZet berekening maakt deze berekening zelf niet multi threaded. Het multithreaded maken van vier op een rij lijkt voor de hand liggend, maar moet ik zelf nog eens uitwerken. Bij het multithreaded maken moet rekening worden gehouden met volgende elementen:

- de verschillende threads mogen elkaars situatie niet beïnvloeden. Elke variabele die gewijzigd wordt moet dan ook een copy van de oorspronkelijke waarde zijn.
 - veronderstel dat elke kolom-invulling in een aparte thread wordt gestart. Voor de eerste zet van de eerste speler worden dan onmiddellijk de acht eerste kolommen ingevuld. Indien ze op een gemeenschappelijk veld worden geplaatst heb ik onmiddellijk *acht* op een rij gerealiseerd. Bovenstaande redenering is van toepassing op *elke* variabele die door een thread zal worden aangepast;
 - bij het multithreaded maken van de oplossing worden verschillende deelzoekbomen parallel uitgevoerd. Dit conflicteert met de winst bekomen door de alfa- beta pruning waarbij een nieuwe deelboom pas wordt verwerkt nadat de beste oplossing uit de vorige deelboom werd berekend.
 - * wellicht is het alfa-beta prunen van een multithreaded oplossing wel realiseerbaar, maar het is maar de vraag hoe groot de winst nog zal zijn. Indien de multithreaded oplossing pas even snel is als de pruned singled threaded oplossing hebben we gewoon resources onnodig verbruikt. Hier is nog ruimte voor wat onderzoek :).

5.7 Backtracking opgaves

- ontwikkel een N-queens oplosser;
 - Extended WPF Toolkit bevat tal van interessante WPF-controls, onder andere verschillende up/down controls. Wellicht kan u een aantal hiervan gebruiken bij het opzetten van uw oplossing;
 - voorzie een synchrone uitvoering, en demonstreer het blokkeren van de interface;
 - voorzie een asynchrone uitvoering (met cancel- mogelijkheid) en demonstreer de responsiveness van de interface (bijvoorbeeld door de cancel uitvoering);
 - voorzie een tweede asynchrone uitvoering die gebruikt maakt van het async/await mechanisme (zie cursustekst).
- ontwikkel een Sudoku oplosser.
 - voorzie een synchrone uitvoering. Het resultaat wordt zo snel bekomen dat een asynchrone uitvoering weinig zin heeft.

Jef Dael's

Hoofdstuk 6

Dependency properties

6.1 Dependency properties: duiding

In een klassieke object georiënteerde omgeving definieert en implementeert een klasse de eigenschappen van haar instanties. In .NET wordt hiertoe een property- syntax gebruikt (we spreken hier van een CLR-property), in JAVA gebruikt men getter- en setter- routines. Zo zal de Text- property van de klasse TextBox in de TextBox-klasse zelf worden geïmplementeerd.

WPF introduceert de extra mogelijkheid om een property, gedefinieerd in klasse A, te koppelen aan instanties van klasse B. Dit vereenvoudigt sterk het opzetten van nieuwe infrastructuren: indien een Grid-klasse nood heeft aan het toekennen van *row* en *column* waarden aan subcontrols, dan kan de *Grid* klasse deze properties zelf introduceren. Deze dependency properties spelen ook een cruciale rol in WPF's databinding en animatie.

In praktijk kan u het dependency property systeem zien als een dictionary waarin voor objecten extra waarden worden bijgehouden. De infrastructuur laat toe om bij het instellen van de waarden custom code uit te voeren.

Attached dependency properties zijn XAML-georiënteerde dependency properties: om in XAML bruikbaar te zijn moet een dependency property een CLR wrapper hebben (die at run-time trouwens gebypassed wordt ..).

6.1.1 Relevante namespaces

- *System.Windows*
- *System.ComponentModel*
- *System.Windows.Media*
- *System.Windows.Data*

6.2 F1URL: een help infrastructuur

We zullen een F1-gebaseerd help systeem opzetten voor de UI-elementen in onze toepassing(en). Hiertoe wensen we de mogelijkheid te hebben om voor een willekeurige control een URL met de gewenste help bij te houden. Wanneer de gebruiker F1 klikt zal de webpagina van de control waarboven de cursor zich bevindt getoond worden. Indien de controls genest zijn zullen de eventuele verschillende pagina's getoond worden. Momenteel wordt in het .NET framework niet de mogelijkheid voorzien om declaratief een URL te koppelen aan een control (u kan natuurlijk zelf een dictionary systeem opzetten).

6.2.1 F1URL gebruik in XAML

```
<Window x:Class="URLDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"
        xmlns:src="clr-namespace:URLDemo"
        x:Name="window"
        src:F1URL.URL="abc"
    >
    <src:F1URL x:Name="grid">
        <Button Height="175" Name="btnOuter"
                HorizontalAlignment="Left" Margin="128,69,0,0" VerticalAlignment="Top" Width="262"
                src:F1URL.URL="www.google.com" >
            <Button Content="Button" Height="67" HorizontalAlignment="Left"
                    x:Name="btnInner" VerticalAlignment="Top" Width="121"
                    src:F1URL.URL="http://www.howest.be" />
        </Button>
    </src:F1URL>
</Window>
```

Figuur 6.1: Attached dependency property in XAML

In bovenstaande XAML-code zijn volgende elementen van belang:

- de namespace *src* hebben we zelf toegevoegd (gebruik de intellisense!);
- *src:F1URL* is een klasse die erft van *Grid*. De controls waarop de F1URL benadering moet werken zullen binnen dergelijke grid geplaatst worden (de *F1URL*- control zal het *KeyUp* event opvangen voor alle geneste subcontrols);
- de (attached) dependency property *F1URL.URL* wordt ingesteld voor het window en twee (geneste) buttons. De geneste buttons introduceren onmiddellijk een extra complexiteit: indien we F1 drukken als de inner button geselecteerd is, wensen we dan de help van de inner of van de outer button?

Wanneer we de Button- klasse erop nazien merken we dat er geen URL-property gekend is. Hoe werd het dan mogelijk om deze property toch in te stellen?

6.2.2 F1URL definitie

```
public class F1URL : System.Windows.Controls.Grid
{
    public static readonly DependencyProperty URLProperty =
        DependencyProperty.RegisterAttached("URL", typeof(string), typeof(F1URL), new UIPropertyMetadata(null));

    ///[System.Windows.AttachedPropertyBrowsableForType(typeof(UIElement))]
    [System.ComponentModel.Browsable(true)]
    public static string GetURL(UIElement target)
    {
        return (string)target.GetValue(F1URL.URLProperty);
    }

    public static void SetURL(UIElement target, string value)
    {
        target.SetValue(F1URL.URLProperty, value);
    }
}
```

Figuur 6.2: Attached dependency property definitie (C#)

Bemerk in bovenstaande code volgende zaken:

- een dependency property is een *static* field van het type *DependencyProperty*. De naam van het field eindigt met het woord *Property*;
- een nieuwe dependency property instantie wordt aangemaakt door middel van een *register(Attached)*- oproep met volgende argumenten:
 - de naam in de registratie is identiek aan de fieldnaam, met weglating van de suffix *Property*;
 - het property type definieert het type van de property (een tekst- property heeft hiervoor als parameter *typeof(string)*);
 - het ownertype definieert de klasse die als prefix voor de property zal gebruikt worden: *F1URL.URL* (*typeof(F1URL)* is hierin het owner type);
 - extra informatie omtrent het verwerken van de property waardes.
- we voorzien de dependency property van (static) Get- en Set- wrapper routines (de hoofdletters *G* en *S* zijn belangrijk!). De Get- en Set- namen zijn (op de drie-letter prefix na) identiek aan de naam gebruikt in de register methode. Zonder deze wrappers is de dependency property niet gekend in XAML.
 - merkwaardig genoeg zal XAML de wrappers volledig bypassen, zoals beschreven in XAML Loading and Dependency Properties;
- indien we de Get- wrapper voorzien van een *Browsable*- attribuut zal deze dependency property ook getoond worden in de property list in Blend (in Visual Studio 2010 wordt de property getoond indien hij in de XAML-code aanwezig is);
- de snippet *propdp* is standaard in Visual Studio aanwezig om de dependency property definiërende code te genereren(*propa* voor attached dependency properties).

6.2.3 F1URL infrastructuur

We definiëren de dependency property *F1URL.URL* om een help infrastructuur op te zetten. Indien een gebruiker F1 duwt wensen we een help-pagina te tonen. Om het key-up event van een willekeurige control op te vangen gaan we als volgt te werk:

- we maken een eigen Grid-klasse versie waarbinnen alle controls die F1URL-ondersteuning nodig hebben geplaatst zullen worden;
 - de hier voorgestelde oplossing werkt enkel voor controls binnen onze eigen Grid-klasse. Deze beperking is overbodig wanneer we echte behaviors zullen ontwikkelen;
 - de huidige opzet is geschikt om bubbling (routed) events toe te lichten;
- de container control zal het bubbling *keyup*-event opvangen en actie ondernemen om de nodige help-topics te tonen:

```
public class F1URL : System.Windows.Controls.Grid
{
    [dep_prop URL]
    public F1URL() : base()
    {
        this.KeyUp += new System.Windows.Input.KeyEventHandler(URLF1_KeyUp);
    }

    void URLF1_KeyUp(object sender, System.Windows.Input.KeyEventArgs e)
    {
        if (e.Key == System.Windows.Input.Key.F1)
            verwerkF1(e.OriginalSource as FrameworkElement);
    }

    private void verwerkF1(FrameworkElement fwe)
    {
        if (fwe == null) return;
        string naam = fwe.Name;
        string url = (string) fwe.GetValue(F1URL.URLProperty);
        if (url != null)
            toonURL(url, naam);
        verwerkF1(fwe.Parent as FrameworkElement);
    }

    private void toonURL(string URL, string naam)
    {
        try
        {
            System.Diagnostics.Process.Start(URL);
        }
        catch (Exception ex)
        {
        }
    }
}
```

Figuur 6.3: F1 infrastructuur (C#)

- de naam van het frameworkelement (een control parent klasse) is aanwezig omwille van debug-redenen;
- de *Process*-klasse start automatisch uw favoriete browser (bemerkt dat *Process* ook het geschikte programma zal starten indien het argument geen URL zou zijn);

- de infrastructuur verwerkt alle geneste controls.

Uiteraard is deze eenvoudige benadering niet helemaal naar wens: zo willen we help kunnen opvragen omtrent een knop zonder(!) dat we deze dienen aan te klikken: het aanklikken start namelijk het proces waarvoor u de help wenst op te vragen, dit is zeker niet wenselijk. De infrastructuur moet rekening houden met de positie van de cursor en niet met de geselecteerde control. De lesdemo illustreert dit en maakt hiertoe gebruik van hit- testing om de control onder de cursor op te zoeken.

6.3 Dependency properties: technisch

- een dependency property is een static `DependencyProperty`- field in een klasse;
- een dependency property heeft een *owner type*. Dit is niet noodzakelijk de klasse waarbinnen de property wordt gedefinieerd. Een dependency property kan trouwens meerdere owner types hebben (we maken dan gebruik van de `addOwner` methode van de `DependencyProperty` klasse).
- een attached dependency property is een dependency property bruikbaar via XAML. Opdat XAML de property zou kennen moeten de naamgevingsregels gevolgd worden!
- bij het registreren van een dependencyproperty kan optioneel extra informatie worden meegegeven:
 - een default waarde;
 - verschillende routines (allen in de vorm van delegates):
 - * `propertyChangedCallback`: wordt uitgevoerd wanneer de property waarde wijzigt (niet noodzakelijk bij elke toekenning!);
 - * `coerceValueCallback`: wordt gebruikt om een eventuele nieuwe waarde om te vormen tot een andere waarde;
 - * `validateValueCallback`: valideert de geldigheid van de nieuwe waarde voor deze klasse (niet in combinatie met de andere state elementen van het target object waaraan de value wordt toegekend). Het boolse resultaat wordt onmiddellijk gebruikt in de XAML-editor om de eventuele incorrecte waarde aan te geven. Deze routine wordt uitgevoerd *na* de `coerce`- routine;
- het definiëren van een dependency property is niet voldoende om deze property voor een object te kunnen instellen: dependencyproperties zijn enkel instelbaar op instanties van klassen die erven van `DependencyObject`!

Het is een aanrader om als delegate oefening eens een uitgewerkte URL- attached dependency property aan te maken met volgende elementen:

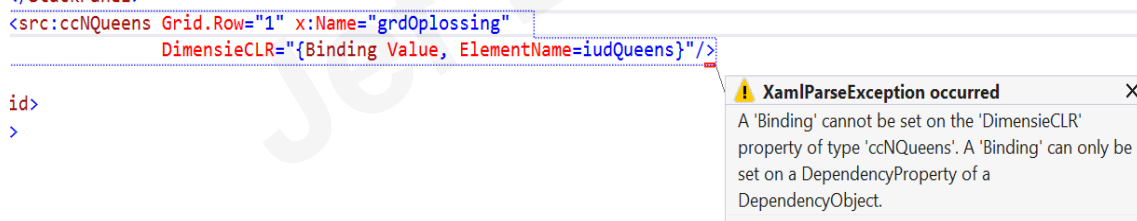
- een URL is slechts geldig indien hij start met *http://* (hoofdletterongevoelig);
- voor elke TextBox control zal de url *http://www.howest.be* worden omgevormd tot *http://www.nmct.be*.

Stel in XAML-code een aantal URL's in om uw implementatie te testen. Voorzie in uw testen volgende situaties:

- voorzie een URL-waarde die niet start met de gevraagde prefix, en observeer het resultaat in XAML en bij het opstarten;
- voorzie een textbox van een om te vormen URL;
- pas de coerce routine aan zodat de nieuwe waarde niet voldoet aan de validate routine en observeer het resultaat.

6.4 Dependency property XAML binding

Via XAML-code kunnen we enkel dependency properties *binden*. Indien we proberen een binding op te zetten voor een gewone (CLR) property, dan krijgen we bij uitvoering volgende foutmelding:



Figuur 6.4: CLR binding error

6.5 Dependency property syntax

Het instellen van een dependency property kan in C# op twee manieren gecodeerd worden:

- *target.SetValue(ownertype.depprop, value)*: het object gebruikt de geërfde methode *SetValue* om een dependencyproperty waarde te bewaren;
- *ownertype.Setdepprop(target,value)*: de static setter routine wordt gebruikt;

```
anWidth.SetValue(Storyboard.TargetNameProperty, btnInner.Name);
anWidth.SetValue(Storyboard.TargetPropertyProperty, new PropertyPath( Button.WidthProperty));
```

Figuur 6.5: Dependency property syntax 1 (C#)

```
Storyboard.SetTargetName(anWidth, btnInner.Name);
Storyboard.SetTargetProperty(anWidth, new PropertyPath(Button.WidthProperty));
```

Figuur 6.6: Dependency property syntax 2 (C#)

Er bestaan uiteraard gelijkaardige notaties voor het opvragen van de dependency property. Beide notaties komen aan bod gedurende het bespreken van de animaties.

6.6 Dependency properties en animaties

WPF ondersteunt een animatie infrastructuur (niet de topic hier) gebaseerd op dependency properties: het wijzigen van een dependency property met impact op de visualisatie zal deze visualisatie onmiddellijk wijzigen. Daarbovenop komen een aantal (animatie-) klassen die gebruikt kunnen worden om waarden van start- waarde naar een eind- waarde te laten evolueren gedurende een tijdsinterval (met easing). In combinatie met Blend maakt dit het vlot opzetten van tijdlijn animaties en storyboards mogelijk.

In dit cursusdeel zullen we de animaties implementeren door middel van C#- code. Door Blend gegenereerde code is XAML code. Het is *niet* de bedoeling om een impliciete voorkeur te laten blijken.

6.6.1 DoubleAnimation

De breedte van een control is een double waarde. De DoubleAnimation klasse voorziet de mogelijkheid om een double waarde geleidelijk te evolueren tussen een start- en eindwaarde. Indien men dan de breedte- eigenschap van een control koppelt aan deze double- evolutie bekomt men een animatie:

```
private void animeerBreedte()
{
    DoubleAnimation anWidth = new DoubleAnimation(btnInner.Width, btnInner.Width * 1.1,
        new Duration(new TimeSpan(0, 0, 2)));
    anWidth.SetValue(Storyboard.TargetNameProperty, btnInner.Name);
    anWidth.SetValue(Storyboard.TargetPropertyProperty, new PropertyPath(Button.WidthProperty));
    anWidth.AutoReverse = true;
    anWidth.RepeatBehavior = RepeatBehavior.Forever;

    Storyboard sb = new Storyboard();
    sb.Children.Add(anWidth);
    sb.Begin(this);
}
```

Figuur 6.7: Breedte animatie (C#)

Op basis van bovenstaande code formuleren we volgende opmerkingen:

- de `DoubleAnimation` instantie zelf weet niet op welke eigenschap van welk object ze zal worden toegepast;
- het koppelen van een `DoubleAnimation` aan een eigenschap van een control gebeurt door middel van twee dependency properties, gedefinieerd in de `Storyboard` klasse:
 - `Storyboard.TargetNameProperty` wordt gebruikt om de naam te bepalen van het element (de control) waarvan een eigenschap geanimeerd zal worden;
 - `Storyboard.TargetPropertyProperty` wordt gebruikt om de dependencyproperty die geanimeerd zal worden te identificeren op basis van zijn `PropertyPath`.
- animaties worden (hier) als onderdeel van een `Storyboard` uitgevoerd;
- het starten van een `Storyboard` heeft nood aan een visueel element waarbinnen de te animeren controls worden gezocht.
 - in *WinRT*-toepassingen is het niet nodig de *Begin*- methode van een *Storyboard* van parameters te voorzien.

6.6.2 ColorAnimation

Het animeren van de achtergrondkleur van een control gebeurt op gelijkaardige wijze maar is verschillend op volgende punten:

- de achtergrond van een control is een `Brush` (en geen `Color`!). Het animeren van de achtergrondkleur gebeurt dan ook door de kleur van een `(SolidColor)Brush` te animeren (we hebben een `ColorAnimation`- class, en geen `SolidColorBrushAnimation` class);
- het `PropertyPath`, nodig om een achtergrondkleur te animeren bestaat uit twee stukken: het eerste stuk kiest de `Background` property, het tweede stuk kiest daarvan de `SolidColorBrush.Color` property. De code zal enkel werken indien de background een `SolidColorBrush` waarde bevat. Indien dit niet het geval is (de achtergrond is niet ingesteld, of is een ander soort `Brush`), dan zal de achtergrond uiteraard niet geanimeerd worden (de gezochte eigenschap is niet aanwezig). Dit genereert trouwens *geen* uitvoeringsfout.

```
private void animeerBackground()
{
    ColorAnimation anKleur = new ColorAnimation(Colors.Red, new Duration(new TimeSpan(0,0,2)));
    Storyboard.SetTargetName(anKleur, btnInner.Name);
    Storyboard.SetTargetProperty(anKleur, new PropertyPath("(Panel.Background).(SolidColorBrush.Color)"));
    anKleur.AutoReverse = true;
    anKleur.RepeatBehavior = RepeatBehavior.Forever;
    Storyboard sb = new Storyboard();
    sb.Children.Add(anKleur);
    sb.Begin(this);
}
```

Figuur 6.8: Background animatie

6.6.3 Transformatie animaties

Het transformeren van een figuur bestaat veelal uit drie onderdelen: een schaling, een rotatie en een skewing. Voor elk van deze onderdelen bestaat er een transformatie klasse. Verschillende transformaties kunnen worden gebundeld tot een transformatiegroep:

```
private void transformeer()
{
    ScaleTransform scale = new ScaleTransform(0.75, 1.1);
    RotateTransform rot = new RotateTransform(90);
    SkewTransform skew = new SkewTransform(30, 30);
    TransformGroup tg = new TransformGroup();
    tg.Children.Add(scale);
    tg.Children.Add(rot);
    tg.Children.Add(skew);

    btnOuter.RenderTransform = tg;
}
```

Figuur 6.9: Transformatie van een Button (C#)

- in veel gevallen is het gemakkelijker om een CompositeTransform te gebruiken in plaats van een transformatiegroep.
- bovenstaande code is nog geen animatie: er wordt een eenmalige transformatie toegepast op een control die daarna onveranderd blijft;

Een animatie zou er in bestaan om bijvoorbeeld de schaling en/of de draaihoek te animeren. Dit zijn properties van de zopas gedefinieerde objecten rot en scale. Deze objecten zijn niet bij naam gekend door de animatie infrastructuur (in de vorige animaties kon het te animeren object (een control) via zijn naam geïdentificeerd worden). Om properties van deze nieuwe objecten toch te kunnen animeren zullen we een naam voor deze objecten registreren. Na registratie kunnen ze via hun naam in een TargetNameProperty context gebruikt worden:

- in een *WinRT* toepassing kunnen we een object animeren zonder zijn naam te kennen door middel van de *TargetProperty*- dependency property van de StoryBoard klasse.

```

private void transformeer()
{
    ScaleTransform scale = new ScaleTransform(0.75, 1.1);
    RotateTransform rot = new RotateTransform(90);
    SkewTransform skew = new SkewTransform(30, 30);
    TransformGroup tg = new TransformGroup();
    tg.Children.Add(scale);
    tg.Children.Add(rot);
    tg.Children.Add(skew);

    btnOuter.RenderTransform = tg;

    this.RegisterName("myrot", rot);
    this.RegisterName("myscale", scale);

    DoubleAnimation anAngle = new DoubleAnimation(rot.Angle, rot.Angle + 360, new Duration(new TimeSpan(0,0,4)));
    anAngle.SetValue(Storyboard.TargetNameProperty, "myrot");
    anAngle.SetValue(Storyboard.TargetPropertyProperty, new PropertyPath(RotateTransform.AngleProperty));
    anAngle.RepeatBehavior = RepeatBehavior.Forever;

    Storyboard sb = new Storyboard();
    sb.Children.Add(anAngle);
    sb.Begin(this);
}

```

Figuur 6.10: Transformatie animatie (C#)

6.6.4 Animaties en XAML

In veel gevallen is het gemakkelijker om animaties aan te maken met behulp van een design tool (*Blend*) dan via programmeer code. De door Blend gegenereerde code is XAML-code die buiten het bestek van dit hoofdstuk valt.

6.6.5 C# of XAML animaties?

Het animeren via code wordt naar mijn aanvoelen minder aangemoedigd dan het animeren via XAML (met behulp van tools zoals *Blend*). Toch meen ik dat een aantal situaties te herkennen zijn waarbij C#-animatie een elegante oplossing kan bieden:

- animatie van dynamisch toegevoegde controls;
- animaties waarvoor de toe te passen waarden afhankelijk zijn van de run-time situatie en niet at-design time gekend zijn. Voorbeelden hiervan zijn:
 - het zweven van een fiche naar een speelveld op een spelbord;
 - het proportioneel verschuiven van elementen binnen een control, die niet altijd even groot moet zijn;

- een aantal basis- animaties (zoals bv. opacity) heb ik als extention method gedefinieerd op de control klasse. Het lijkt me dan ook weinig zinvol om deze animatie in XAML te definiëren.

Bovenstaande situaties zijn implementeerbaar via XAML, maar behoeven dan meestal andere extra *C#*-code elementen zoals *TypeConverter* of extra properties.

6.7 DependencyProperties in XAML: TypeConverters

Indien we in XAML een dependency property instellen moeten we ons beperken tot tekst. Dit is bruikbaar voor tekstwaarden, maar niet indien het resultaat type een niet-tekst type is. Om dit toch mogelijk te maken worden typeconvertoren gebruikt. Deze zijn in staat om op basis van een string een instantie van het property type te construeren (een vorm van deserialisatie). WPF bevat reeds een aantal typeconvertoren voor frequent voorkomende conversies (getallen, kleuren, ..) maar in sommige gevallen is het toch nodig om ook zelf een typeconverter te implementeren.

6.7.1 IValueConverter interface

Deze interface bevat een *Convert* en *ConvertBack* methode (in onderstaand voorbeeld is enkel de *Convert* van belang).

Het *ValueConversion* attribuut wordt gebruikt om het systeem te informeren omtrent de geïmplementeerde typeconversies. Onderstaande code codeert een converter die tekst omvormt tot een *LinearGradientBrush*. Er wordt verondersteld dat de twee kleurnamen in het Engels, en hoofdlettergevoelig geschreven worden:

```
[System.Windows.Data.ValueConversion(typeof(string),typeof(System.Windows.Media.LinearGradientBrush))]
public class LinearGradientConverter : System.Windows.Data.IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        string tekst = value as string;
        if (tekst == null) return null;

        string[] kleuren = tekst.Split(new char[]{' '});
        BrushConverter bc = new BrushConverter();
        Color kleur1 = ((SolidColorBrush) bc.ConvertFrom(kleuren[0])).Color;
        Color kleur2 = ((SolidColorBrush) bc.ConvertFrom(kleuren[1])).Color;
        return new LinearGradientBrush(kleur1, kleur2, 45);
    }

    public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Figuur 6.11: ValueConverter definitie (*C#*)

In XAML kunnen we dan volgende code noteren:

```
<Window.Resources>
  <src:LinearGradientConverter x:Key="lgc"></src:LinearGradientConverter>
  <s:String x:Key="redgreen">Red,Green</s:String>
</Window.Resources>
<src:F1URL x:Name="grid">
  <Button Height="129" Name="btnOuter"
    HorizontalAlignment="Left" Margin="128,69,0,0" VerticalAlignment="Top" Width="262"
    src:F1URL.URL="http://www.google.com"
    Background="{Binding Source={StaticResource ResourceKey=redgreen}, Converter={StaticResource ResourceKey=lgc}}" >
  </Button>
```

Figuur 6.12: ValueConverter XAML (C#)

Hier wordt de achtergrond van een Button (een Brush dus) ingesteld door middel van een tekst (Red,Green) en een typeconverter. Het resultaat is een LinearGradientBrush.

Jammer in de XAML notatie is dat de tekst-waarde momenteel als static resource gedefinieerd wordt (misschien kan dit anders?)

6.7.2 Default dataconverters

Veelal lijkt de conversie van tekst in XAML naar een overeenkomstig object spontaan te gebeuren, zonder expliciete vermelding van de nodige type converter. Om dit te realiseren wordt een klasse voorzien van een default typeconverter (door middel van het `TypeConverterAttribute` attribuut). De typeconverter klasse wordt jammer genoeg anders opgebouwd dan de *ValueConverters* die eerder aan bod kwamen. Om een klasse als impliciete typeconverter te kunnen gebruiken gaan we als volgt te werk:

- de klasse die de conversie zal uitvoeren *moet* erven van de klasse *TypeConverter* (de interface *IValueConverter* is hier onbelangrijk);
- het is belangrijk om minstens twee methodes te overriden:
 - *CanConvertFrom* die default *false* terug geeft die aangeeft of het gevraagde brontype wel geschikt is om te converteren (in XAML is het brontype *String*);
 - *ConvertFrom* is de methode die de effectieve conversie uitvoert.

Wanneer een klasse een default typeconverter kent die tekst kan omvormen tot een instantie van de klasse, dan is het niet langer nodig om in de XAML-code een expliciete converter te voorzien.

6.7.2.1 TryParse

Wanneer ik een conversie van tekst naar een type X moet coderen zal ik het type X voorzien van een *TryParse* methode:

- het resultaat type is Boolean: true indien de conversie lukt, false anders;

- de eerste parameter bevat de tekstwaarde die geconverteerd moet worden;
- de tweede parameter (een referentie parameter) bevat het resultaat na de conversie (ongewijzigd indien de conversie niet lukt).

Het gebruik van *TryParse* is in .NET reeds op vele plaatsen terug te vinden (Integer, Double, DateTime, IPAddress, ..). Het implementeren van dergelijke methode is vooral een conventie, en zeker geen technische vereiste.

6.8 Dependency properties en OnRender

Indien een dependencyproperty belangrijk is voor het visuele aspect van de dependency objecten waarop de property wordt toegepast dan kan dit in de meta informatie worden opgenomen (bij het registreren van de dependency property). Indien dit het geval is zal het wijzigen van de dependencywaarde de *OnRender*- methode van het object waarvoor de dependencyproperty wijzigt triggeren. Dit op zijn beurt heeft als gevolg dat u het hertekenen niet moet oproepen via de propertychanged callback methode. Indien u dit wenst uit te pluizen (helemaal niet moeilijk) kan u zoeken op *FrameworkPropertyMetadata.AffectsRender* en *FrameworkPropertyMetadata.AffectsMeasure*.

6.9 Veel voorkomende fouten

- niet volgen van de naamgevings conventies (ook hoofdletters!);
- *null* als default waarde van een value type;

6.10 Extra lectuur

- Wie een interessant voorbeeld met vele dependency properties wenst na te lezen verwijst ik graag door naar *Render Text On A Path With WPF*
- het aanmaken van een eigen dependency property wordt beschreven in *Custom Dependency Properties*
- een goede technische beschrijving omtrent attached dependency properties

6.11 Dependency property: opgaves

- animeer via code een control:
 - zijn afmetingen (breedte en grootte);
 - zijn achtergrondkleur;
 - roteer de control;
- maak een extention method waarmee u de opacity van een control (elke control) kan animeren: `FadeInOut(double opacity)`;
- implementeer de *F1URL* infrastructuur en demonstreer dat deze werkt.
- voorzie de NQueens toepassing van een *Dimensie*- dependency properties, en bind deze aan de `IntegerUpDown` control zodat de code-behind leeg wordt;

```
<src:ccNQueens Grid.Row="1" x:Name="grdOplossing"
                src:ccNQueens.Dimensie="{Binding Value, ElementName=iudQueens}" />
```

Figuur 6.13: XAML binding dependency property

Figuren

1.1	Extention method definition	7
1.2	Extention method gebruik	7
1.3	Extention method gebruik: fadeto	8
1.4	Await demo	9
1.5	Await demo async methode	9
2.1	Delegate definitie	12
2.2	Delegate demo: uppercase en lowercase	12
2.3	Event definitie (zonder guidelines)	14
2.4	Event abonnering	14
2.5	Event afvuren (zonder guidelines)	15
2.6	Static event definitie (volgens de guidelines)	17
2.7	Static event client	17
2.8	Asynchrone verwerking demo	20
2.9	AsyncCallback demo	21
2.10	UI-thread problem	21
2.11	Dispatcher demo	22
2.12	Await- keyword	23
2.13	XMLViewer en code plumbing	26
2.14	Anonymous method voorbeeld	28
2.15	Lambda expressie voorbeeld	28
3.1	ComponentModel attributen (C#)	32
3.2	Assembly reflection (C#)	33
3.3	Strongly typed instantiatie	34
3.4	Activator.CreateInstance (C#)	34
3.5	Activator.InvokeMember (C#)	35
4.1	Commandline compilatie: C#broncode	40
4.2	C#commandline compilatie en uitvoering	41
4.3	Commandline compilatie: VB.NET broncode	41
4.4	VB.NET commandline compilatie voorbeeld	42
4.5	C#System.CodeDom.Compiler namespace	43
4.6	CodeDOM	44
4.7	ILDasm output	45

4.8	ILSpy output	46
4.9	Reflector output	47
4.10	Obfuscated output via Reflector	48
4.11	Tekenen van een functie	49
5.1	NQueens oplossen	52
5.2	Sudoku spelregels: een opgave	53
5.3	Sudoku spelregels: een oplossing	54
5.4	Sudoku routed events demo	56
5.5	Awaiting een oproep	57
5.6	TaskCompletionSource: SetResult	57
5.7	TaskCompletionSource demo	58
5.8	4 op een rij: bordwaarde	59
5.9	Minimax algoritme	61
6.1	Attached dependency property in XAML	66
6.2	Attached dependency property definitie (C#)	67
6.3	F1 infrastructuur (C#)	68
6.4	CLR binding error	70
6.5	Dependency property syntax 1 (C#)	70
6.6	Dependency property syntax 2 (C#)	71
6.7	Breedte animatie (C#)	71
6.8	Background animatie	72
6.9	Transformatie van een Button (C#)	73
6.10	Transformatie animatie (C#)	74
6.11	ValueConverter definitie (C#)	75
6.12	ValueConverter XAML (C#)	76
6.13	XAML binding dependency property	78

Index

- Action, 27
- Activator, 34
 - CreateInstance, 34
- addOwner, 69
- AddressOf, 15
- AffectsMeasure, 77
- AffectsRender, 77
- Alfa Beta pruning, 61
- Animaties, 71
- Animation
 - ColorAnimation, 72
 - DoubleAnimation, 71
 - Transformation, 73
- Anonymous method, 28
- Assembly, 31
- Async, 23
- async, 9
- AsyncCallback, 20
- Asynchrone verwerking, 20
 - UI-thread, 22
- Attached dependency property, 65
- Attribute
 - Category, 32
 - DefaultEvent, 32
 - DefaultProperty, 32
 - DefaultValue, 32
 - Description, 32
- Await, 23
- await, 9
- backtracking, 51
- BeginInvoke, 20
- BesteZet, 60
- Blend, 74
- BordWaarde, 59
- Brush, 72
- bubbling, 18
- CanConvertFrom, 76
- Category, 32
- CodeDomProvider, 43
- Color, 72
- ColorAnimation, 72
- CompilerParameters, 43
- CompilerResults, 44
- contravariance, 13
- Convert, 75
- ConvertBack, 75
- ConvertFrom, 76
- covariance, 13
- CreateInstance, 34
- CreateProvider, 43
- DefaultEvent, 32
- DefaultProperty, 32
- DefaultValue, 32
- Delegate, 11, 31
 - Asynchrone verwerking, 20
 - BeginInvoke, 20
 - codeplumbing, 24
 - events, 13
 - multithreading, 19
- Dependency properties, 65
- DependencyProperty, 67
 - addOwner, 69
 - RegisterAttached, 67
- Description, 32
- deserialisatie, 75
- Dispatcher, 22
- DoubleAnimation, 71
- EAP, 23
- Event based Asynchronous Pattern, 23
- EventArgs, 16
- Events
 - +=, 14

- AddHandler, 15
- AddressOf, 15
- event keyword, 14
- EventArgs, 16
- guidelines, 16
- handles, 14
- onEvent, 16
- preview, 18
- RaiseEvent, 15
- routed events, 18
- static events, 17
- tunneling event, 18
- Events:bubbling event, 18
- F1URL, 66
- Func, 27
- Generics, 16
- GetCustomAttributes, 35
- GetLanguageFromExtension, 43
- Ghost data, 19
- IAsyncResult, 20
- IL, 44, 45
- ILDasm, 45
- inconsistent read, 19
- instructies
 - lock, 19
- intermediate language, 45
- InvokeMember, 35
- IValueConverter, 75, 76
 - Convert, 75
 - ConvertBack, 75
- Lambda expression, 28
- LinearGradientBrush, 75
- LINQ, 28
- Lost update, 19
- Main, 40
- MenuItem, 38
- MinMax, 59
 - BesteZet, 60
 - BordWaarde, 59
 - MinMax, 61
 - Pruning, 61
- Multithreading, 19
- N-Queen, 52
- Namespace
 - System, 11
 - System.CodeDom.Compiler, 39
 - System.ComponentModel, 31, 32, 65
 - System.Reflection, 31
 - System.Windows, 65
 - System.Windows.Data, 65
 - System.Windows.Media, 65
- Obfuscation, 48
- OnRender, 77
- OpenFileDialog, 33
- patterns
 - strategy pattern, 24
- plugin, 31
- preview, 18
- PreviewKeyDownEvent, 55
- Process, 68
- PropertyPath, 72
- Pruning, 61
- Reflection, 31
- Reflector, 47
- runtime compilatie, 39
- signatuur, 13
- snippet
 - propa, 67
 - prodp, 67
- Storyboard, 72
 - TargetNameProperty, 72
 - TargetPropertyProperty, 72
- String, 40
- System, 11, 40
- System.CodeDom.Compiler, 39, 43
- System.ComponentModel, 31, 32, 65
- System.Reflection, 31
- System.Windows, 65
- System.Windows.Data, 65
- System.Windows.Media, 65
- Task, 57

Transformation, 73
TryParse, 76
tunneling, 18
Type.GetCustomAttributes, 35
Type.InvokeMember, 35
TypeConverter, 75, 76

UI-thread, 22
using, 40

ValueConversion, 75

Window, 36

XMLViewer, 25