Het gebruik van Visual Basic for Applications in Microsoft Excel

Theo Peek

Januari 2003

Inhoud

1 Inleiding

1.1 Voorkennis

Deze handleiding is geschreven voor derdejaars studenten Bedrijfswiskunde en informatica. Zij hebben de volgende vakken reeds gehad:

- Inleiding Programmeren I en II (IP);
- Datastructuren (DS);
- Spreadsheetcursus (SC).

Al tijdens IP wordt object georiënteerd programmeren geïntroduceerd. Dit wordt tijdens DS intensief gebruikt. Daarnaast worden tijdens SC de basisbeginselen van Excel behandeld (o.a. Solver, Pivottable, Dialog Boxes en macro's). Voor het lezen en gebruiken van deze handleiding verwacht ik dan ook, dat de lezer bekend is met object georiënteerd programmeren in Java en het basisgebruik van Excel.

Andere geïnteresseerden zijn van harte welkom deze handleiding ook te lezen en gebruiken. Ervaring met Excel is onontbeerlijk, kennis van Java en object georiënteerd programmeren is handig.

1.2 Doelstelling

Na het doorwerken van deze handleiding, ken je:

- De structuur van VBA;

Na het doorwerken van deze handleiding, kun je:

- Macro's opnemen en uitvoeren;
- Macro's programmeren in VBA;
- Omgaan met de Visual Basic Editor in Excel;
- Een Graphical User Interface maken in VBA;
- Excel laten samenwerken met andere programma's zoals Matlab.

1.3 Hulpmiddelen en software

Voor deze handleiding heb je een geïnstalleerde Engelse versie van Excel 2000 nodig op een PC met een Engelse versie van Windows 2000 Professional. Excel is meestal op te starten via **Start** > **Programs** > **Microsoft Excel** of via **Start** > **Programs** > **Microsoft Office** > **Microsoft Excel**. In Hoofdstuk 5 laten we Excel met andere programma's samenwerken. Om Excel met Matlab te laten samenwerken is uiteraard een installatie van Matlab nodig. Om een DOS-commando uit te voeren met VBA moet de gebruiker rechten hebben om de Command Prompt te openen. Aan deze voorwaarden was voldaan op 16 september 2002 in de computerzalen op de VU.

1.4 Opbouw en gebruik van deze handleiding

Deze handleiding is verdeeld in 6 hoofdstukken. In het eerste hoofdstuk worden de basisbeginselen van VBA behandeld, gevolgd door iets geavanceerdere technieken en achtergronden in Hoofdstuk 3. In Hoofdstuk 4 geven we de beginselen van het maken van een grafische omgeving in Excel. Hoofdstuk 5 geeft een manier om Excel samen te laten werken met andere programma's en Hoofdstuk 6 geeft een manier om betere prestaties te bereiken door het cachen van ranges in arrays.

Je leest de handleiding in principe van begin tot eind. Ik heb de volgende notatie gebruikt:

Hoera!	Normale tekst
Visual Basic Editor (VBE)	Een nieuw begrip of een nieuwe term
`Function`	VBA-code
Start	Invoer, ofwel via het toetsenbord, ofwel aanklikken

`VBA-code` kom je tussen normale tekst tegen, maar ook apart:

```
Function VBAcode
       Exit Function
End Function
```

In de tekst daaronder wordt uitgelegd, wat bovenstaande code doet. Ik raad je aan zowel een digitale als een papieren versie te gebruiken. Uit de digitale versie kun je dan stukken VBA-code kopiëren naar de Visual Basic Editor, om zelf te kijken wat de code doet. Ik raad je daarnaast aan, om veel zelf uit te proberen. Als er in de tekst staat: hier

gebruiken we niet `dit`, maar `dat`, omdat `dat` beter is, dan begrijp je pas waarom `dat` beter is, als je zelf ook even `dit` uitprobeert.

Soms passen we een eerder geprogrammeerde procedure aan. De nieuwe code verschijnt dan **vet**:

```
Function VBAcode
        DoEvents
        Exit Function
End Function
```

Een enkele keer is een functie zo lang dat ik hem in delen beschrijf:

```
Function VBAcode
        DoEvents
        ...
```

En het tweede deel:

```
        ...
        Exit Function
End Function
```

In de digitale versie kun je ook links gebruiken. Klik bijvoorbeeld eens op Appendix A: Literatuur. Ook zijn de plaatjes in kleur in de digitale versie.

De VBA-Help is bijzonder handig. Je start hem door in de Visual Basic Editor **F1** of **Help > Microsoft Visual Basic Help** te kiezen. In de Help kun je beschrijvingen van alle standaard functies vinden.

2 Over macro's in Excel

2.1 Een eenvoudige macro

Tijdens de Spreadsheetcursus heb je kennisgemaakt met het gebruik van macro's in Excel. Om er weer een beetje in te komen, beginnen we met een bijzonder eenvoudige macro.

- Zet tien willekeurige waarden in de cellen A1:A10, bijvoorbeeld de getallen 1, ..., 10;
- Start de ***Macro recorder*** met **Tools > Macro > Record New Macro**, geef de macro de naam **ErgSimpel** en bevestig met **OK**;
- In beeld verschijnt de ***Stop Recording Toolbar***, zie Figuur 2.1;

Figuur 2.1: Stop Recording Toolbar.

- Selecteer de cellen A1:A10;
- ***Copy*** deze (**Ctrl-C**);
- Selecteer cel C1;
- ***Paste*** (**Ctrl-V**);
- De selectie is nu C1:C10. Kies ***Format Cells*** (**Ctrl-1**) en kies op het tabblad ***Font*** een blauwe kleur uit voor de letters;
- Kies het tabblad ***Patterns*** en kies een gele kleur uit als celachtergrond;
- Klik **OK**;
- Selecteer cel A1;
- Stop de macro recorder door op de Stopknop op de Stop Recording Toolbar te klikken;
- Start de ***Visual Basic Editor (VBE)*** (**ALT-F11**).

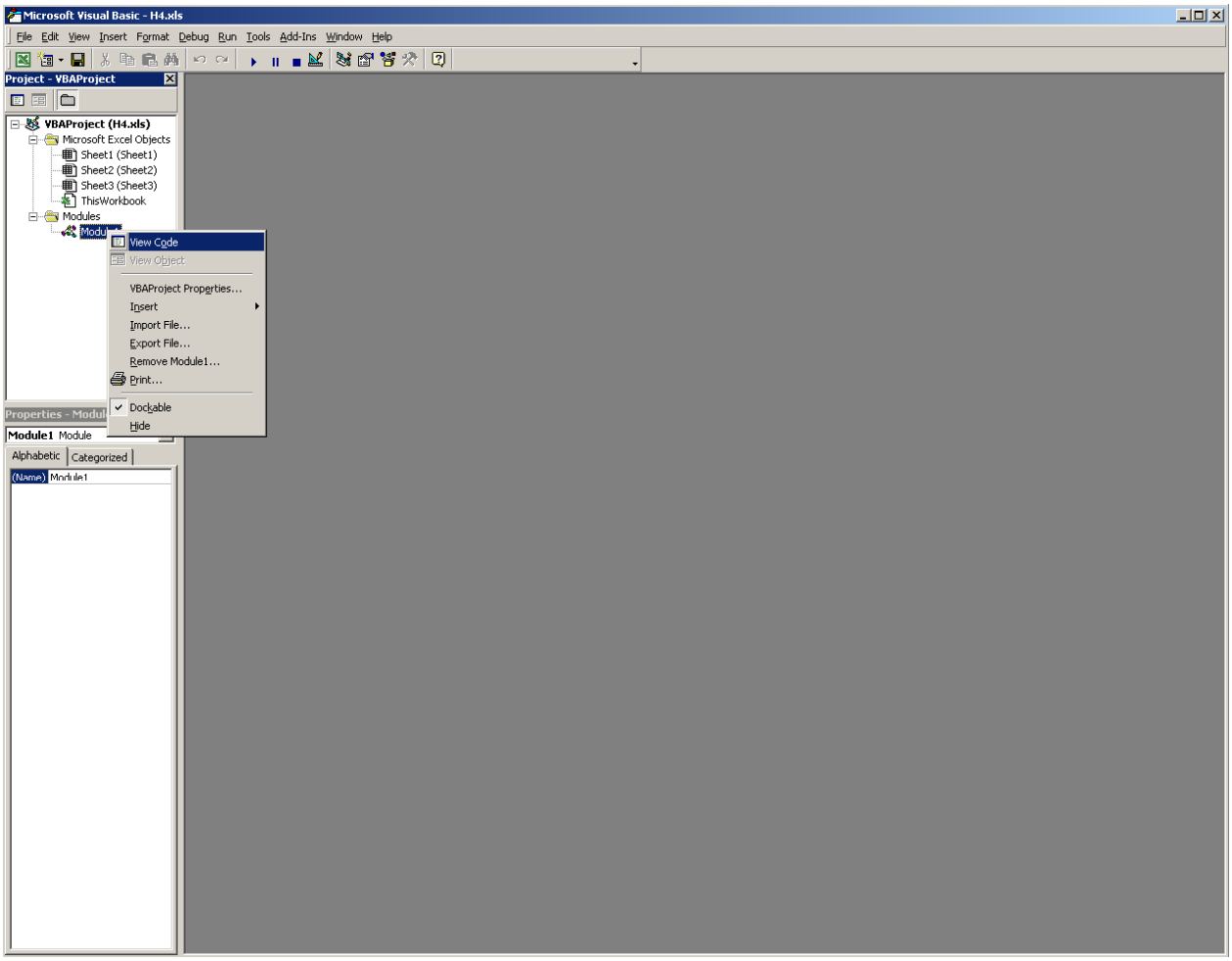
Je krijgt dan een window als in Figuur 2.2. Klik rechts op **Module1** in het Project Window en kies **View Code**. Het ***Code Window*** wordt geopend en je ziet de code zoals hieronder:

```
Sub ErgSimpel()
'
' ErgSimpel Macro
' Macro recorded 14-10-2002 by studenttest
'

'
    Range("A1:A10").Select
    Selection.Copy
    Range("C1").Select
    ActiveSheet.Paste
    Application.CutCopyMode = False
    With Selection.Font
        .Name = "Arial"
        .FontStyle = "Regular"
        .Size = 10
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ColorIndex = 32
    End With
    With Selection.Interior
        .ColorIndex = 6
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
    End With
    Range("A1").Select
End Sub
```

Deze code is niet erg ingewikkeld. Houd er wel rekening mee, dat de code overbodigheden kan bevatten, doordat hij gegenereerd in plaats van geprogrammeerd is. Merk als eerste op dat commentaar met een `'` begint. `Application.CutCopyMode = False` wil zeggen dat je de gekopieerde cellen niet nog een keer kunt plakken. Onder `With Selection.Font` staan de dingen die je op het tabblad Font veranderd hebt. Eigenlijk heb je alleen de laatste regel daarvan veranderd: `.ColorIndex = 32`. De waarde van de ColorIndex kan verschillen, omdat die afhankelijk is van de kleur die je hebt gekozen. Onder `With Selection.Interior` staan de veranderingen van het tabblad Patterns.

Ook daar is `.ColorIndex = 6` het enige dat je zelf hebt veranderd. Als laatste zie je het selecteren van cel A1 weer terug: `Range("A1").Select`.



Figuur 2.2: De Visual Basic Editor (VBE).

Nu je geheugen weer opgefrist is, is het tijd om te gaan programmeren in VBA.

2.2 Een eenvoudige functie

De rij van Fibonacci is een rij getallen, waarbij het volgende getal steeds de som is van de twee voorgaande getallen. De eerste twaalf getallen zijn 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144. De rij van Fibonacci komt in de natuur op veel plaatsen voor. Het bekendste voorbeeld is het feit dat zonnebloempitten zodanig staan ingeplant dat ze twee stelsels spiralen lijken te vormen. Die stelsels bevatten meestal 34 en 55 spiralen, maar 55 en 89, of 89 en 144 komen ook voor. Zoals je ziet zijn het steeds opeenvolgende Fibonaccigetallen. De rij van Fibonacci heeft veel te maken met de inverse van de Gulden Snede $\Phi = 2/(-1+\sqrt{5}) \approx 1{,}618$. Als F_n het n-de element van de rij van Fibonacci is, dan geldt:

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \Phi.$$

Voor meer informatie over de Gulden Snede en de rij van Fibonacci, zie [2]. We gaan nu een functie schrijven, die het n-de element van de rij van Fibonacci berekent. Links zie je code, die je zou gebruiken als je de functie in Java moest schrijven. Rechts zie je de VBA-code.

```
int fib (int n) {
   int result;
    if (n == 1 || n == 2) {
        result = 1;
    } else {
        result = fib(n - 1) + fib(n - 2);
    }
    return result;
}
```

```
Function fib(n As Integer) As Integer

    If n = 1 Or n = 2 Then
        fib = 1
    Else
        fib = fib(n - 1) + fib(n - 2)
    End If

End Function
```

In beide gevallen is de functie `fib` recursief geprogrammeerd, met als stopconditie dat de eerste twee getallen van de rij bekend zijn. In de eerste regel staat de functie-declaratie. Daar waar in Java het type vóór de variabele staat, staat in VBA het type achter de variabele (`int n` tegenover `n As Integer`). Ook het return-type van de functie staat in VBA achter de functienaam. In Java moet de integer `result` handmatig gedeclareerd worden, in VBA bestaat er direct al een integer `fib`. Deze integer wordt automatisch aan het einde van de functie geretourneerd, waar dat in Java handmatig moet gebeuren. Let tot slot nog op het volgende: in de regel `fib = fib(n - 1) + fib(n - 2)` is de eerste `fib` de variabele, de twee andere `fib`'s zijn functie-aanroepen.

Vraag: de bovenstaande functie voor het uitrekenen van Fibonacci-getallen is niet bepaald robuust en efficiënt. Geef voor beide een argument.

Open de file "Fibonacci.xls" en start vervolgens de Visual Basic Editor. Je ziet in het ***Project Window*** wat er allemaal in dit ***workbook*** staat: drie ***worksheets*** en een workbook (die zitten standaard in elke nieuwe Excel-file) als ***Microsoft Excel Objects*** en één ***module***: Module1. Klik rechts op **Module1** en kies **View Code**. Daar zie je de code zoals hier rechtsboven. De functie kun je op twee manieren gebruiken. De eerste manier is via het ***Immediate Window***. Dat open je via het menu **View** of met **Ctrl-G**. Hierin kun je commando's geven als **?1+2**. De functie `fib` roep je aan door het commando **?fib(7)** te geven. Dan krijg je het zevende Fibonacci-getal 13. Het Immediate Window is ideaal om functies te testen en debug-informatie naartoe te sturen. De andere manier om de functie aan te roepen is op een worksheet. In een willekeurige cel typ je dan **=fib(7)**.

We gaan nu deze functie verbeteren, door de manier van inputverwerking robuuster te maken. Het eerste probleem zit in het argument `n`. Omdat `n` een integer is, kunnen we niet controleren of het meegegeven argument ook een integer is. Je kunt de functie prima aanroepen met een double, bijvoorbeeld 2.5. VBA rondt een double af naar een integer, maar hoe dat gebeurt, is niet te voorspellen. Het kan voorkomen dat 2.5 naar 2 wordt afgerond en 5.5 naar 6. Dit is vreemd gedrag van VBA en we moeten zorgen dat de situatie niet voorkomt. Daarom schrijven we hierna een verbeterde functie.

Opdracht: Schrijf een kleine functie om te controleren hoe VBA doubles afrondt.

Hieronder staat de functie `fib2`.

```
Function fib2(n As Variant) As Integer
    If (checkInput(n)) Then
        If (n = 1 Or n = 2) Then
            fib2 = 1
        Else
            fib2 = fib(n - 1) + fib2(n - 2)
        End If
    Else
        fib2 = -1
    End If
End Function


Function checkInput(n As Variant) As Boolean
    If (Not IsNumeric(n)) Then
        checkInput = False
    ElseIf (CInt(n) <> Abs(n)) Then
        checkInput = False
    Else
        checkInput = True
    End If
End Function
```

`fib2` heeft als argument een ***Variant***. Een Variant is een verzameltype voor alle objecten en types van VBA. Zie Hoofstuk 3.1 voor een overzicht van de typehiërarchie van VBA. Deze Variant geeft `fib2` door aan de functie `checkInput`. `CheckInput` retourneert false als `n` geen positieve integer is. In de eerste regel van CheckInput wordt gekeken of `n` wel een getal is. Via `ElseIf` (is één woord) wordt gekeken of `n` een positieve integer is. De expressie `CInt(n) <> Abs(n)` zou in Java `int(n) != Math.abs(n)` zijn. Ga na hoe deze expressie werkt. Merk ook op dat we het ook in één expressie (zonder `ElseIf`) hadden kunnen testen.

Als `checkInput` ***true*** retourneert, gaat de functie `fib2` verder en voert hetzelfde algoritme uit als de oorspronkelijke functie `fib`. Anders wordt ***-1*** als antwoord gegeven.

We hebben er nu voor gezorgd dat de input gecheckt wordt, maar een antwoord ***–1*** geven is natuurlijk niet netjes. We zouden eigenlijk een soort ***error*** moeten genereren. Voor de volgende verbetering van onze functie gaan we dan ook gebruik maken van een ***error handler***. Dit zullen we doen in paragraaf 2.5.3.

2.3 Het verschil tussen macro's en functies

Macro's en functies zijn beide ***procedures***. Een macro is een procedure die geen waarde retourneert, vergelijkbaar met een void-method in Java. Een functie retourneert wel een waarde, bijvoorbeeld een Integer of een String.

In paragraaf 2.1 hebben we een macro opgenomen. In de code kun je zien dat deze gedeclareerd is met `Sub`. Je kunt macro's aanroepen in Excel via **Tools > Macro > Macros** of **Alt-F8**. In het Immediate Window roep je macro's aan door de naam in te typen, bijvoorbeeld **ergsimpel** (VBA is niet case-sensitive).

In paragraaf 2.2 hebben we een functie geprogrammeerd. Deze declareer je met `Function` in plaats van `Sub`. Je kunt een functie in cellen op een worksheet aanroepen met bijvoorbeeld **=fib(A1)**. De functie gebruikt de waarde in de cel als argument. In het Immediate Window gebruik je **?fib(3)**.

Een ander verschil tussen macro's en functies zit in het gebruik van argumenten. Stel dat we de volgende macro hebben:

```
Sub makro(s As String)
    Debug.Print "Ja, " & s
End Sub
```

Dan roepen we deze in het Immediate Window aan door `makro ("zeker!")` of `makro "zeker!"`. De eerste aanroep heet een ***functie-aanroep***, de tweede een ***macro-aanroep***. Hebben we meer dan één argument, zoals in de volgende macro:

```
Sub makro(s As String, n as Integer)
    Debug.Print "Ja, " & s
End Sub
```

dan werkt de functie-aanroep `makro("zeker!", 2)` niet meer. We moeten dan de macro-aanroep `makro "zeker!" , 2` gebruiken.

Functies retourneren een waarde; wanneer we deze waarde in een expressie gebruiken, moeten we de functie aanroepen met een functie-aanroep. Deze functie retourneert heel eenvoudig de meegegeven waarde:

```
Function funktie(s As String)
    funktie = s
End Function
```

Deze kunnen we met een functie-aanroep gebruiken in een Debug.Print statement: `Debug.Print funktie("Ja, zeker!")`. Als we niets met de geretourneerde waarde doen, dan moeten we de functie aanroepen alsof we een macro aanroepen. Dat wil zeggen: met één argument mogen we kiezen, met meer dan één argument moeten we de macro-aanroep gebruiken. Bovenstaande functie kunnen we dus met `funktie("Ja, zeker!")` of met `funktie "Ja, zeker!"` aanroepen. Onderstaande functie:

```
Function funktie(s As String, n As Integer)
    funktie = s
End Function
```

moeten we met `funktie "Ja, zeker!", 2` aanroepen.

Macro's en functies kun je ook via een ***Call*** aanroepen. In dat geval moet je altijd haakjes om de argumenten zetten:

```
Call makro ("zeker!", 2)
```

Als je een functie met Call aanroept, kunnen we niets met de geretourneerde waarde doen.

2.4 Control structures

2.4.1 While

De standaard manier voor een While-loop in VBA is de volgende:

```
Dim check As Boolean
While (check = True)
        '...
Wend
```

Een variabele-declaratie begint altijd met `Dim`. Het type (boolean in dit geval) staat achter de variabele. De haakjes zijn optioneel. Een meer geavanceerde manier van looping is via Do-while:

```
Dim check As Boolean
While (check = True)
        '...
Wend
```

Of, equivalent:

```
Dim check As Boolean
Do While check = True
        '...
Loop
```

Of zelfs:

```
Do Until check = False
        '...
Loop
```

En

```
Dim check As Boolean
Do
        '...
Loop Until check = False
```

Het voordeel van de Do-Loop is het `Exit Do` statement, vergelijkbaar met `break` in Java:

```
Do Until check = False
        '...
        If Not tweedeCheck()
                Exit Do
        End If
Loop    '...
```

2.4.2 For

Het For-statement wordt als volgt gebruikt:

```
For i = 1 To 9 'Step 2
        '...
        'Exit For
Next i
```

De stapgrootte kan aangepast (bijvoorbeeld vergroot tot 2) worden door in de eerste regel `Step 2` toe te voegen, zoals nu in commentaar staat. `Exit For` werkt net zo als `Exit Do` in paragraaf 2.4.1.

2.5 Arrays

2.5.1 Arrays in VBA

VBA kent ***fixed arrays*** en ***dynamic arrays***. Fixed arrays hebben een vaste grootte die niet veranderd kan worden. Dynamic arrays kun je na hun declaratie nog vergroten of verkleinen. We spelen eerst wat met een fixed array, zie de functie `fixedArray()`.

```
Function fixedArray()
    Dim rij(1 To 4) As Integer
    For i = 1 To 4 'indextelling begint bij 1
        rij(i) = i
        Debug.Print rij(i); " ";
    Next i
End Function
```

Merk als eerste op, dat we de indextelling bij 1 laten beginnen en niet bij 0, zoals in Java. Dat stellen we in met het argument `1 To 4`. We hadden ook `0 To 3` kunnen geven. De puntkomma's in het `Debug.Print`-statement geven aan dat je op dezelfde regel blijft. Het meegeven van het argument `1 To 4` in de declaratie van `rij` maakt het array fixed.

Als we geen argument meegeven, declareren we een dynamic array, zoals in de volgende functie: `dynamicArray()`.

```
Function dynamicArray()
    Dim rij() As Integer
    ReDim rij(1 To 4)
```

```
        For i = 1 To 4
            rij(i) = i
            Debug.Print rij(i); " ";
        Next i
        Debug.Print
        ReDim Preserve rij(1 To 5)
        For i = 1 To 5
            Debug.Print rij(i); " ";
        Next i
        Debug.Print
        ReDim Preserve rij(1 To 3)
        For i = 1 To 3
            Debug.Print rij(i); " ";
        Next i
        Debug.Print
        ReDim rij(1 To 3)
        For i = 1 To 3
            Debug.Print rij(i); " ";
        Next i
    End Function
```

Met `ReDim` geven we het array een grootte. Om naar de volgende regel te gaan met printen geven we een `Debug.Print`-statement zonder `;`. Vervolgens maken we het array 5 elementen groot. `Preserve` geeft aan dat de huidige waarden bewaard moeten blijven. Het vijfde element heeft geen waarde en krijgt standaard de waarde 0. Dan maken we het array 3 elementen groot, met behoud van de waarden en vervolgens zonder behoud van waarden. De output van de functie fixedArray() (als we die aanroepen in het Immediate Window) is de volgende:

```
1   2   3   4
```

En van de functie dynamicArray():

```
1   2   3   4
1   2   3   4   0
1   2   3
0   0   0
```

Je hebt inmiddels voldoende informatie om de output van beide functies te verklaren (toch?).

2.5.2 Arrays en array formulas in Excel

In de vorige paragraaf heb je gezien hoe je arrays in VBA gebruikt. Je kunt ze echter ook op een Excel worksheet gebruiken. Stel, we hebben een eenvoudige kruidenierswinkel, gespecialiseerd in boterhambeleg. We verkopen pindakaas, chocopasta en kaas. Tabel 2.1 bevat de gegevens van de maand september.

Tabel 2.1: Verkoopcijfers over de maand september.

Product	Afzet	Prijs
Pindakaas	100	€ 1,69
Chocopasta	200	€ 1,39
Kaas	20	€ 4,45

We willen de omzet over de maand september berekenen. We zetten deze gegevens op een worksheet. Daarbij kiezen we voor kolom C een ***currency*** format (op tabblad Number van Format Cells). We zouden een extra kolom kunnen maken, met daarin de omzet van een bepaald product, en daaronder een sommatie van de productomzetten om de totale omzet te berekenen. Zie Tabel 2.2:

Tabel 2.2: Omzetcijfers over de maand september.

Product	Afzet	Prijs	Productomzet
Pindakaas	100	€ 1,69	€ 169,00
Chocopasta	200	€ 1,39	€ 278,00
Kaas	20	€ 4,45	€ 89,00
		Omzet:	€ 536,00

Met een ***array formula*** is zo'n extra kolom niet nodig. We zetten in plaats daarvan de volgende formule in cel D5: **=SUM(B2:B4*C2:C4)**. Om aan te geven dat het een array formula is, geven we geen **Enter**, maar **Ctrl-Shift-Enter**. B2:B4 en C2:C4 worden als arrays (vectoren) gezien en SUM geeft hun inproduct. In cel D5 zie je accolades {} om de functie heen staan, om aan te geven dat het een array formula is.

B2:B4*C2:C4 als array formula levert eigenlijk een array met drie elementen als resultaat. SUM telt die drie op om tot de totale omzet te komen. Je kunt het array met drie elementen ook laten zien. Dan genereer je eigenlijk de kolom Productomzet. Selecteer daartoe drie boven elkaar gelegen cellen, klik op de formulebalk en typ **=B2:B4*C2:C4**. Bevestig uiteraard met **Ctrl-Shift-Enter**.

Overigens: omdat deze berekening nogal eens voorkomt heeft Excel een functie SUMPRODUCT. Deze neemt arrays als argumenten. Je kunt dus ook in cel D5 **=SUMPRODUCT(B2:B4;C2:C4)** (bevestig met **Enter**, want het is geen array formula) zetten.

2.5.3 Nogmaals Fibonacci

In paragraaf 2.2 lieten we onze functie `fib2` achter zonder goede error handling. We gaan nu een verbeterde versie `fib3` maken met behulp van onze kennis van arrays.

```
Function fib3(n)
    On Error GoTo catch
    If Not IsNumeric(n) Then
        tmp = CVErr(xlErrNA)
    ElseIf CInt(n) <> Abs(n) Then
        tmp = CVErr(xlErrNA)
    Else
        tmp1 = 1
        tmp2 = 0
        For i = 0 To n
            tmp = tmp1
            tmp1 = tmp1 + tmp2
            tmp2 = tmp
        Next i
    End If
ret:
    fib3 = Array(tmp, typeName(tmp))
    Exit Function
catch:
    tmp = CVErr(xlErrNA)
    Resume ret
End Function
```

Met `On Error GoTo Catch` geven we aan dat we een error handler gebruiken. Als we een niet-numeriek argument `n` gekregen hebben, maken we met `CVErr(xlErrNA)` een error aan met foutnummer `xlErrNA`. We zetten die error in de variabele `tmp`. Ook als we een negatief of gebroken getal krijgen, creëren we die error. Indien we een correcte invoer gekregen hebben (positieve integer), gaan we het *n*-de Fibonaccigetal uitrekenen. Dit doen we niet meer recursief, maar iteratief.

Vraag: Leg uit, waarom deze methode een stuk efficiënter is.

Merk op dat we nergens een variabele declareren. Een ongedeclareerde variabele krijgt het type van de eerste waarde die hij krijgt. Zo krijgt `tmp` het type Error als het argument geen positieve integer is, en het type Integer als begonnen wordt met het uitrekenen van een Fibonaccigetal. Het spreekt voor zich dat het gestructureerder is een variabele wel te declareren, ook gezien de veelvoorkomende en weinigzeggende VBA-error "Type-mismatch" zonder regelnummer erbij. `ret:` en `catch:` zijn labels, waar je met `GoTo` en `Resume` naartoe kunt gaan. Als je zonder `GoTo` of `Resume` bij een label terechtkomt, wordt het label genegeerd. `Resume` heeft dezelfde functie als `GoTo`, maar `Resume` gebruik je in een error handler. De functie `TypeName` geeft het type van een variabele. De waarde die deze functie retourneert, is een array met twee elementen (zie onder `ret:`). Het eerste element is ofwel het antwoord of een errormessage. Het tweede element is het type van het antwoord, ofwel Integer, ofwel Error). Omdat deze functie een array als antwoord geeft, kun je hem niet in het Immediate Window aanroepen, maar alleen op een worksheet. Dit doe je door twee naast elkaar gelegen cellen te selecteren, **=fib3(3)** te typen en te bevestigen met **Ctrl-Shift-Enter**. In de linkercel zie je dan het antwoord (#N/A of het *n*-de Fibonaccigetal), in de rechtercel staat het type (Error of Integer). Zie **Tabel 2.3**.

Tabel 2.3: Resultaat van de functie fib3.

n	*n*-de Fibonaccigetal	Type
-2	#N/A	Error
-1	#N/A	Error
0	1	Integer
1	1	Integer
2	2	Integer
3	3	Integer
4	5	Integer

5	8	Integer
6	13	Integer
7	21	Integer
8	34	Integer
9	55	Integer
10	89	Integer

2.5.4 Hilbertmatrix

Een matrix is een tweedimensionaal array. Een Hilbertmatrix is een matrix waarin element (*i*,*j*) de waarde 1/(*i*+*j*-1) heeft. We gaan een functie schrijven die de Hilbertmatrix berekent.

```
Function HilbertMatrix()
    numcols = Application.Caller.Columns.Count
    numrows = Application.Caller.Rows.Count
    Dim M() As Double
    ReDim M(1 To numrows, 1 To numcols)
    For i = 1 To numrows
        For j = 1 To numcols
            M(i, j) = CDbl(1) / CDbl(i + j - 1)
        Next j
    Next i
    HilbertMatrix = M
End Function
```

We selecteren een aantal cellen, typen **=HilbertMatrix()** en bevestigen met **Ctrl-Shift-Enter** (want het is een array formula). De functie moet er zelf achterkomen, hoeveel cellen er geselecteerd zijn. Daarom gebruikt de functie in de eerste regels `Application.Caller.Columns.Count` en `Application.Caller.Rows.Count`. De functie is verder eenvoudig te begrijpen.

2.6 Nog meer VBA

2.6.1 Celnotatie

De standaardmanier om naar cellen op een worksheet te verwijzen is met de zogenaamde ***A1-notatie***. Deze notatie is ***absoluut***, het geeft namelijk precies aan welke cel je bedoelt. De ***R1C1-notatie*** is ***relatief***.

Typ in cel B3 de formule: **=A1+B2+C2**. Nu gaan we de notatie veranderen. Ga naar het tabblad **General** onder **Tools** > **Options** en vink **R1C1 reference style** aan. Bevestig met **OK** en je ziet in B3 de formule **=R[-2]C[-1]+R[-1]C+R[-1]C[1]** verschijnen. R[-2] betekent twee rijen terug, C[-1] betekent één kolom terug. C is een afkorting van C[0] en betekent dus dezelfde kolom. Je moet deze notatie begrijpen om enkele VBA-functies goed te kunnen gebruiken.

2.6.2 Range

Een ***range*** is de belangrijkste datastructuur van Excel. Een range kan de volgende vormen aannemen:

- Een enkele cel;
- Een blok cellen (een ***area***);
- Meer dan één blok cellen.

Zie de code van de functie `RangeInfo()`.

```
Sub RangeInfo()
    If TypeName(Selection) <> "Range" Then
        MsgBox "Selection is geen Range"
        Exit Sub
    End If
    Dim msg As String
    Dim blok As Range
    Dim cel As Range
    msg = "Selection bevat " & Selection.Count & " cel(len):"
    For Each cel In Selection
        msg = msg & " """ & cel.Address & """"
    Next cel
    MsgBox msg
    msg = "Selection bevat " & Selection.Areas.Count & " area(s):"
    For Each blok In Selection.Areas
        msg = msg & " """ & blok.Address & """"
    Next blok
    MsgBox msg
End Sub
```

Het idee is, net als bij de Hilbertmatrix in paragraaf 2.5.4, om een aantal cellen te selecteren en vervolgens `RangeInfo()` te starten. Je krijgt dan informatie over de selectie. We bespreken eerst de VBA-code. Als eerste testen we, of de selectie wel een range is. De selectie zou namelijk ook bijvoorbeeld een grafiek kunnen zijn. Als de selectie inderdaad een range blijkt, maken we een String met tekst, het aantal cellen in de selectie (`Selection.Count`) en de adressen van die cellen (`cel.Address`). De gemaakte String wordt vervolgens getoond in een ***message box***. De volgende message box bevat een String met informatie over de areas: het aantal en de adressen.

Ga na hoe deze functie werkt door cellen en areas te selecteren en vervolgens de macro te starten. Je selecteert meer dan één cel of area door **Ctrl** ingedrukt te houden als je de tweede cel of area selecteert.

De volgende functie is de macro `CellInfo()`. Deze geeft informatie over een enkele cel. Daartoe checken we eerst, of de selectie wel een enkele cel is. Vervolgens maken we een String met de informatie. Merk op dat het `&`-teken bij Strings dezelfde functie heeft als het `+`-teken in Java, namelijk het ***concateneren*** van twee Strings. `Chr(13)` is een regeleinde. De rest van de macrocode is eenvoudig te begrijpen. Je kunt de functie testen op verschillende cellen. Let er op, dat je ook een cel kiest, die een formule bevat.

2.6.3 De Cells property

Aan de macro CelInfo() kun je al een beetje de structuur van Excel aflezen.

```
Sub CelInfo()
    If TypeName(Selection) <> "Range" Or Selection.Count <> 1 Then
        MsgBox "Selection is geen cel"
        Exit Sub
    End If
    msg = "Enige gegevens over de geselecteerde cel:"
    msg = msg & Chr(13) & "Address: " & Selection.Address
    msg = msg & Chr(13) & "CurrentRegion: " & _
                          Selection.CurrentRegion.Address
    msg = msg & Chr(13) & "Value: " & Selection.Value
    If Selection.HasFormula Then
    msg = msg & Chr(13) & "Formula: " & Selection.Formula
    msg = msg & Chr(13) & "FormulaR1C1: " & Selection.FormulaR1C1
    End If
    msg = msg & Chr(13) & "Worksheet: " & Selection.Worksheet.Name
    msg = msg & Chr(13) & "Workbook: " & _
                          Selection.Worksheet.Parent.Name
    MsgBox msg
End Sub
```

Je hebt een Workbook, dat meerdere Worksheets bevat. Waarschijnlijk wist je dit al, bovendien komen we er in Hoofdstuk 3 uitgebreider op terug. Met deze structuur kun je met VBA alle cellen in alle worksheets van alle geopende workbooks aanspreken. Applications, worksheets en ranges hebben allemaal een ***Cells property***. Een ***property*** is officieel een eigenschap van een object, maar is het best te vergelijken met een method van een object. Zo staat Application.Workbooks(“Fibonacci.xls”).Sheets(“Sheet1”).Cells(1,1) voor cel A1 op worksheet “Sheet1” in Workbook “Fibonacci”. De argumenten van de Cells property werken relatief, net als de R1C1-notatie, maar niet exact hetzelfde! Dat kun je zien aan de expressie Worksheets(“Sheet1”).Range(“B5”).Cells(0,0), die voor cel A4 staat. Waar je op basis van de R1C1-notatie zou verwachten dat (0, 0) voor B5 staat, staat (1, 1) voor B5 en (0, 0) voor A4. Een overzicht:

Tabel 2.4: Een overzicht van de Cells property.

Object	Object.Cells() staat voor
Application	Een range met alle cellen op de actieve worksheet
Worksheets(“Sheet1”)	Een range met alle cellen op de worksheet Sheet1
Range(“A1:B2”)	Een range met alle cellen in de range A1:B2

Let wel, dat de expressie Worksheets(“Sheet1”).Range(“B5”).Cells(0,0) voor cel A4 staat, die *geen* onderdeel is van de range B5.

2.6.4 Selection

In VBA kun je met ***Selection*** werken en dat is handig. Een probleem daarbij is, dat de huidige Selection op je Worksheet verloren gaat. In het volgende voorbeeld bewaren we eerst de huidige Selection, dan gebruiken we Selection om een aantal cellen de waarde 4 te geven en vervolgens zetten we Selection weer op terug. In commentaar

staat een manier om hetzelfde resultaat in één regel te bewerkstelligen, zonder iets met Selection te doen. Het gegeven voorbeeld is vooral nuttig om te laten zien hoe je Selection kan gebruiken, zonder de huidige Selection verloren te laten gaan.

```
Sub SelectionTest()
    'Worksheets(2).Range("A1:E7").Value = 4
    Set SaveActiveSheet = ActiveSheet
    Sheets(2).Select
        Set SaveSelection = Selection
        Set SaveActiveCell = ActiveCell
            Range("A1:E7").Select
            Selection.Value = 4
        SaveSelection.Select
        SaveActiveCell.Activate
            'Gebruik Activate voor een cel
            'i.p.v. Select
     SaveActiveSheet.Select
End Sub
```

Met het ***Set-statement*** ken je een ***object reference*** toe, te vergelijken met een ***reference*** in Java. Met `Set SaveActiveSheet = ActiveSheet` maak je een object aan, de reference ernaartoe noem je `SaveActiveSheet` en in het object stop je de huidige actieve worksheet (`ActiveSheet`). Het bewaren van de Selection en de actieve cel gaat analoog. Om de cellen A1:E7 van worksheet 2 de waarde 4 te geven, selecteren we eerst worksheet 2 en vervolgens de range A1:E7. Deze selectie geven we de waarde 4. We sluiten af door onze bewaarde objecten te selecteren respectievelijk te activeren.

2.6.5 Automatic Calculation

Als je bezig bent met VBA of Excel, wordt telkens automatisch het worksheet doorgerekend. Mocht er in een VBA-procedure een fout optreden, die niet afgevangen wordt door een error handler, dan stopt VBA ermee. Zie hieronder een voorbeeld van zo'n situatie:

```
Sub test()
    Range("a1").Value = 1
    Range("a1").Formula = "=myfn(a1)"
    Range("a1").Clear
    MsgBox "Hello"
End Sub

Function myfn(a)
    myfn = 1 / a
End Function
```

Hier geef je cel A1 eerst de waarde 1. Vervolgens zet je in cel A2 de formule **=myfn(A1)**, die op dat moment 1 / 1 = 1 oplevert. Dan maak je A1 leeg, het werkblad wordt doorgerekend en `myfn` genereert een fout, want 1 / 0 heeft geen antwoord. VBA stopt direct, en er verschijnt geen MessageBox.

Om dit te voorkomen, kan het handig zijn om worksheets handmatig in plaats van automatisch door te rekenen. Dit kun je in Excel instellen door **Tools > Options...** te kiezen en op het tabblad Calculation **Manual** te kiezen. Met **F9** kun je dan handmatig doorrekenen.

Je kunt er ook voor kiezen om in je VBA-procedures de automatic calculation uit te schakelen. Dit is typisch een geval waarin het gemakkelijk is om erachter te komen, hoe je dat in VBA doet; namelijk door de Macro recorder te starten, de instelling veranderen zoals hiervoor beschreven, de Macro recorder te stoppen en de opgenomen code te bekijken.

Desondanks vertel ik je hoe het moet. Met `Application.Calculation = xlCalculationManual` schakel je handmatig berekenen in, met `Application.Calculation = xlCalculationAutomatic` schakel je automatisch berekenen weer in. Je kunt er zelfs voor kiezen om eerst de huidige status op te slaan met `SaveCalculation = Application.Calculation`, vervolgens je worksheet-manipulatie te doen en ten slotte de status weer terug te zetten met `Application.Calculation = SaveCalculation`.

2.6.6 Constanten

In paragraaf 2.2 heb je gezien hoe je een variabele declareert (met `Dim`). Je kunt in VBA echter ook constanten declareren. Dat lijkt erg op het declareren van variabelen, zie:

```
Const Name As String = "Theo"
```

Met andere woorden, vervang `Dim` door `Const` en geef de constante meteen een waarde (dit is uiteraard verplicht).

2.6.7 Data types

VBA geeft je de mogelijkheid om zelf data types te definiëren. Je zou een zogenaamd ***user-defined data type*** kunnen zien als een zeer eenvoudig object. Voordat je een user-defined data type kunt gebruiken, moet je het definiëren. Dit moet bovenaan in een module gebeuren, in het zogenaamde Declarations-gedeelte:

```
Type persoon
    naam As String
    leeftijd As Integer
    geboortedatum As Date
End Type
```

Vervolgens gaan we dit type gebruiken in een macro:

```
Sub persoonlijk()
    Dim p As persoon
    p.naam = "Theo Peek"
    p.leeftijd = 24
    p.geboortedatum = #10/17/1978#

    Debug.Print p.naam; " "; p.leeftijd; " "; p.geboortedatum
End Sub
```

Dit spreekt allemaal voor zich, behalve misschien de datum: `#10/17/1978#`. Dit is nu eenmaal de manier waarop een datum in VBA gebruikt wordt. Maar als je `#17-10-78#` typt, maakt de VBE er automatisch `#10/17/1978#` van.

2.6.8 Het aanpassen van de menustructuur van Excel

Laten we eens de volgende macro toevoegen aan het Tools menu in Excel:

```
Sub action()
    MsgBox "Hello world!"
End Sub
```

Dat doen we met de macro `setup`:

```
Sub setup()
    Set c = Application.CommandBars("Tools").FindControl( _
Tag:="Hello World") 'Tag wordt uitgelegd
    Select Case TypeName(c)
      Case "CommandBarButton"
      Case "Nothing"
          Set c = Application.CommandBars("Tools").Controls.Add( _
            Type:=msoControlButton, Before:=3)
            c.Caption = "Say Hello"
            c.Style = msoButtonIconAndCaption
            c.OnAction = "action" 'de naam van de macro die uitgevoerd moet worden
            c.Tag = "Hello World"
      Case Else
        Debug.Print "dit zou niet moeten kunnen"
    End Select
End Sub
```

In de eerste regel zie je aan het einde een underscore (_) staan. Deze dient om de compiler te vertellen, dat het statement op de volgende regel doorgaat. Voor FindControl zijn vijf argumenten gedefinieerd, namelijk Type, Id, Tag, Visible en Recursive. Om aan te geven dat we in dit geval alleen het argument Tag willen gebruiken, roepen we `FindControl` aan met `(Tag := “Hello World”)`. Het statement zoekt of het menu-item met Tag “Hello World” al in het menu staat. Een tag is een soort identifier voor het menu-item. In de regel `c.Tag = "Hello World"` geven we het menu-item de tag “Hello World”. Deze tag heeft overigens niets te maken met de tekst in de message box (“Hello World!”).

Als in de eerste regel al een menu-item met tag “Hello World” bestaat, gaat de reference `c` ernaar wijzen en krijgt `c` dus het type “CommandBarButton”. Als het item nog niet in het menu staat, wijst `c` naar niets.

Het `Select Case` statement is te vergelijken met het switch-statement van Java. Als het type van `c` “CommandBarButton” is, dan gebeurt er niets (want dan staat hij er al). Als het type van `c` “Nothing” is, dan wordt er een nieuw menu-item gemaakt. `Before:=3` geeft aan dat het nieuwe item op plaats drie in het menu moet komen. In `c.Caption` komt de naam van het menu-item, in `c.OnAction` komt de naam van de uit te voeren macro. Als laatste maken we een `Tag` aan, om het menu-item te kunnen identificeren, zoals we in de eerste regel van de macro `setup` doen.

Mochten we later dit menu-item weg willen halen, dan gebruiken we de macro `destroy`:

```
Sub destroy()
    Set c = Application.CommandBars("Tools").FindControl( _
```

```
Tag:="Hello World")
    Select Case TypeName(c)
      Case "CommandBarButton"
        c.Delete
      Case "Nothing"
      Case Else
        Debug.Print "dit zou niet moeten kunnen"
    End Select
End Sub
```

Eerst kijken we weer of het menu-item wel bestaat, zo ja, dan halen we het weg (`c.Delete`), zo nee, dan doen we niets.

2.7 Opgave: driehoek van Pascal

De driehoek van Pascal is een oneindige driehoek alle binomiaalcoëfficiënten. Ze zijn zodanig geordend, dat elk element van de driehoek de som is van het getal er rechts- en linksboven. Aan de rand heb je alleen maar enen. De eerste tien rijen van de driehoek van Pascal vind je in Figuur 2.3.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
  1   7   21  35  35  21  7   1
1   8   28  56  70  56  28  8   1
1   9   36  84  126 126 84  36  9   1
```

Figuur 2.3: De eerste tien regels van de driehoek van Pascal.

De opdracht is het schrijven van een macro die op de active worksheet de eerst *n* regels van de driehoek van Pascal afbeeldt. De macro moet aan de volgende eisen voldoen:

1) Het aantal regels *n* moet aan de gebruiker worden gevraagd;
2) Op de eerste regel mag een waarde worden gezet; op alle andere regels mogen alleen maar formules voorkomen;
3) De macro moet in ieder geval werken voor $n \leq 10$; het is echter niet moeilijk hem uit te breiden tot $n \leq 128$;
4) De getallen moeten mooi onder elkaar komen te staan, zoals in Figuur 2.3. Dit betekent ook dat de kolommen zo smal moeten zijn, dat alle getallen tegelijk op het scherm komen.

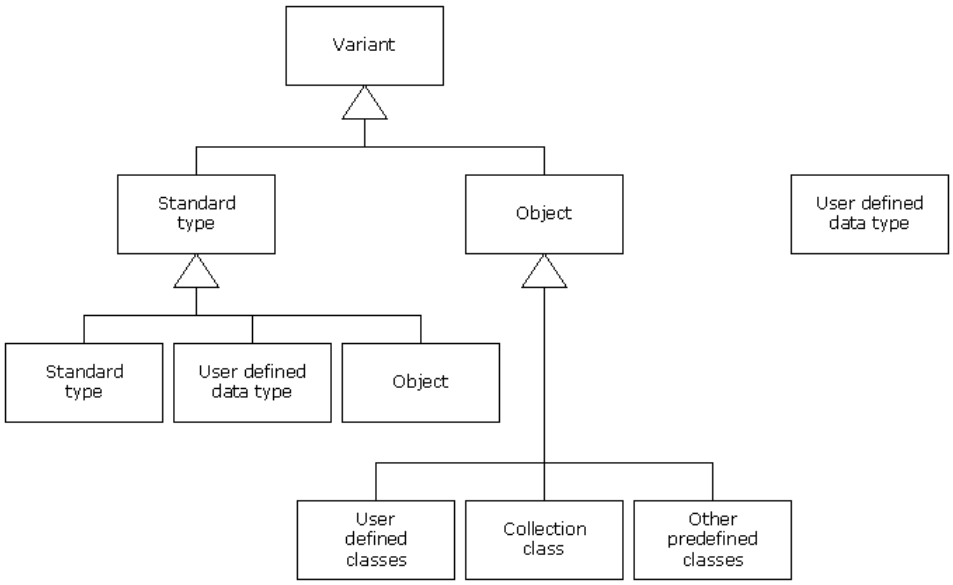
Aanwijzingen:

- Om aan de vierde eis te voldoen, kun je gebruik maken van de Merge and Center functie, die op de knoppenbalk van Excel zit;
- Om uit te vinden hoe je bijvoorbeeld Merge and Center in VBA programmeert, kun je gebruik maken van de Record Macro functie;
- Begin met het zoeken van een manier om de driehoek goed weer te geven; vul daarna pas de formules in.
- Een oplossing is te vinden in Appendix B.1.

3 Structuur van VBA en de VBE

3.1 Type system

Figuur 3.1 toont de hiërarchie van types in VBA.



Figuur 3.1: De type hiërarchie van VBA.

Het belangrijkste van dit overzicht, is dat Variant de generalisatie van alle types en objecten is, behalve van zelfgemaakte types. In paragraaf 2.5.3 heb je gezien, dat je niet per se een variabele hoeft te declareren. Een variabele krijgt namelijk altijd het type Variant, tenzij anders aangegeven. Dat dit niet ideaal is, bewijst het volgende voorbeeld:

```
Sub TimeLoops()
    Debug.Print "For-Next met Integer: ";
    aanvang = Timer
    For i = 1 To 1000
        Dim CountFaster As Integer
        For CountFaster = 0 To 32766
        Next CountFaster
    Next i
    Debug.Print Tab(25); Timer - aanvang
       'Tab(25) zorgt ervoor, dat de tijden precies onder elkaar komen

    Debug.Print "For-Next met Long: ";
    aanvang = Timer
    For i = 1 To 1000
        Dim CountMedium As Long
        For CountMedium = 0 To 32766
        Next CountMedium
    Next i
    Debug.Print Tab(25); Timer - aanvang

    Debug.Print "For-Next met Variant: ";
    aanvang = Timer
    For i = 1 To 1000
        Dim CountSlower As Variant
        For CountSlower = 0 To 32766
        Next CountSlower
    Next i
    Debug.Print Tab(25); Timer - aanvang
End Sub
```

`Timer` geeft de huidige tijd van het systeem. Die tijd zetten we in `aanvang`. Vervolgens declareren we 1.000 keer een Integer, die we elke keer van 0 tot 32766 laten lopen. Datzelfde doen we met een Long en met een Variant. Deze macro heb ik op een Pentium II-350 en een Pentium III-1000 gedraaid. De resultaten staan in **Tabel 3.1**.

Tabel 3.1: Resultaten van de macro TimeLoops().

	Pentium II-350	Pentium III-1000
Integer	1,921875	0,703125
Long	2,0625	0,8007813
Variant	3,007813	1,210938

Op beide computers is te zien, dat de tijden voor berekeningen met Integer en Long elkaar niet veel ontlopen. Bij gebruik van een Variant duurt de het uitvoeren van de macro 50% langer. Dat komt, doordat bij de berekeningen met Variants het volgende gebeurt:

- Zoek het juiste type voor de variabele (Integer);
- Converteer de variabele naar dat type;
- Tel er één bij op;
- Converteer de variabele naar Variant.

Je kunt je voorstellen, dat dit veel extra tijd kost. Een ander nadeel van het gebruik van Variants, is dat ze veel geheugen in beslag nemen. Een Integer is 2 bytes groot, een Long 4 bytes en een Variant maar liefst 16 bytes. In dit voorbeeld maakt dat niet zoveel uit, maar bij het gebruik van een groot aantal variabelen (bijvoorbeeld in een array), zul je dat zeker gaan merken.

Als je het nut van het declareren van variabelen onderkent, kun je dit als verplicht instellen. Vink in de VBE (Visual Basic Editor) onder **Tools > Options** op tabblad Editor "Require Variable Declaration" aan. Dan is het in alle modules verplicht om variabelen te declareren. Wil je het slechts voor een enkele module verplicht stellen, begin de module dan met de regel `Option Explicit`.

3.2 Objecten en properties

Veel dingen die je in Excel gebruikt, zijn ***objecten***. Workbooks, worksheets, ranges en grafieken bijvoorbeeld. Je kunt net als in Java ook zelf objecten maken. Objecten kunnen in VBA ***properties*** en ***methods*** hebben. Properties kun je zien als zelfstandige naamwoorden die het object beschrijven. Methods zijn werkwoorden die het object zelf onderneemt. Een object Auto zou bijvoorbeeld properties Gewicht, Snelheid en Lengte kunnen hebben en methods Accelereer en Rem.

Voor het object Auto maken we een class, waarvan je objecten kunt maken. Kies in de VBE **Insert > Class Module**. Deze willen we de naam "Auto" geven. Dat doen we in het ***Properties Window***, dat standaard geopend staat (zie Figuur 2.2). Mocht het niet geopend zijn, kun je het via **F4** of **View > Properties Window** alsnog openen. Op het Categorized tabblad staat bij (Name) "Class1". Dit kun je veranderen in "Auto". Vervolgens schrijven we de class:

```
Private weight As Integer 'in kg
Private speed As Integer 'in km/h
Private length As Single 'in m

Public Property Let gewicht(g As Integer)
    weight = g
End Property

Public Property Let lengte(l As Single)
    length = l
End Property

Public Property Let snelheid(s As Integer)
    speed = s
End Property

Public Property Get snelheid() As Integer
    snelheid = speed
End Property

Public Property Get lengte() As Single
    lengte = length
End Property

Public Property Get gewicht() As Integer
    gewicht = weight
End Property

Public Sub accelereren(kmh As Integer)
    speed = speed + kmh
End Sub

Public Sub remmen(kmh As Integer)
    speed = speed - kmh
End Sub
```

We beginnen met het declareren van drie ***private*** variabelen: `weight`, `speed` en `length`. De namen zijn in het Engels om de properties in het Nederlands te kunnen houden zonder dat er naamgevingconflicten optreden. `Private` heeft hier dezelfde betekenis als in Java, zie ook paragraaf 3.4. `Length` declareren we als `Single`, een gebroken getal dat minder ruimte inneemt dan een Double.

Vervolgens definiëren we de properties. Met het ***Property Let*** statement geef je een property een waarde (`gewicht`, `lengte` en `snelheid`). Met het ***Property Get*** statement haal je de waarde van de property op. Met het ***Property Set*** statement, dat we hier niet gebruiken, kun je een ***reference*** naar het object maken.

Als laatste schrijven we de twee methods: `accelereren` en `remmen`. Om de class Auto te kunnen gebruiken, schrijven we nog een macro `use()`:

```
Sub use()
    Dim a As New Auto
    a.lengte = 4.3
    a.gewicht() = 900
    a.snelheid() = 100
    Debug.Print a.lengte; Tab(10); a.gewicht; Tab(20); a.snelheid
    a.accelereren (20)
    Debug.Print a.snelheid
    a.remmen (140)
    Debug.Print a.snelheid()
End Sub
```

Hierin maken we eerst een object aan met reference `a`. De regel `Dim a As Auto` zou alleen een reference van het type Auto maken; met het keyword `New` wordt direct een object gemaakt, waarin alle variabelen de waarde 0 krijgen. Met `a.lengte = 4.3` gebruiken we eigenlijk de `Property Let` procedure uit de class Auto. In de volgende regel zie je, dat je zelf kunt kiezen of je haakjes gebruikt of niet. Dan printen we de properties. `Tab(10)` is voornamelijk handig als je meerdere regels onder elkaar wilt printen, dan kun je gegevens precies onder elkaar zetten. Merk op dat we met `a.remmen` een negatieve snelheid bereiken. Dit kan gebeuren, doordat we nergens op goede input testen.

In het voorbeeld maken we geen gebruik van het Property Set statement. Dat statement hebben we nodig als we een object in een object gebruiken. Daartoe maken we een object Locatie, dat de plaats aangeeft waar de auto staat, als extra property van Auto. De code daarvan staat hieronder:

```
Private town, street As String
Private number As Integer

Property Let plaats(p As String)
    town = p
End Property

Property Let straat(s As String)
    street = s
End Property

Property Let huisnummer(n As Integer)
    number = n
End Property

Property Get plaats() As String
    plaats = town
End Property

Property Get straat() As String
    straat = street
End Property

Property Get nummer() As Integer
    nummer = number
End Property

Function toString()
    toString = street & " " & number & ", " & town
End Function
```

De Property Let en Property Get statements zijn duidelijk als je het vorige voorbeeld begrepen hebt. Verder is er nog een method toString, die alle gegevens van de Locatie in één String zet. Ons object Auto wordt nu (nieuwe code is **vet**):

```
Private weight As Integer 'in kg
Private speed As Integer 'in km/h
Private length As Single 'in m
Private loc As locatie 'object
```

```
        Public Property Let gewicht(g As Integer)
            weight = g
        End Property

        Public Property Let lengte(l As Single)
            length = l
        End Property

        Public Property Let snelheid(s As Integer)
            speed = s
        End Property

        Public Property Set locatie(l As locatie)
            Set loc = l
        End Property

        Public Property Get snelheid() As Integer
            snelheid = speed
        End Property

        Public Property Get lengte() As Single
            lengte = length
        End Property

        Public Property Get gewicht() As Integer
            gewicht = weight
        End Property

        Public Property Get locatie() As locatie
            Set locatie = loc
        End Property

        Public Sub remmen(kmh As Integer)
            speed = speed - kmh
        End Sub

        Public Property Set Auto(a As Object)
            Set CurrentAuto = a
        End Property
```

In de variabele `loc` gaan we de Locatie bewaren. In `Public Property Set locatie(l As locatie)` is `l` de reference naar de `Locatie` waar de auto staat. Omdat we een object reference gebruiken in plaats van een type, kennen we deze plaats niet met `loc = l` aan `loc` toe, maar met `Set loc = l`.

Uiteraard schrijven we ook een `Property Get Locatie()`, die de locatie retourneert. Dit alles gaan we gebruiken met de nieuwe macro use():

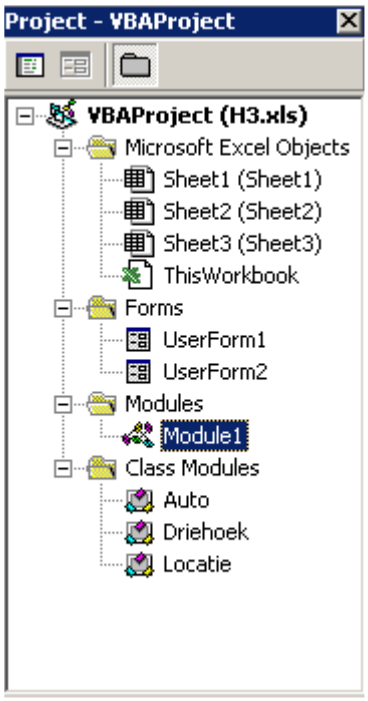
```
Sub use()
    Dim a As New Auto
    Dim l As New locatie
    l.plaats = "Amsterdam"
    l.straat = "De Boelelaan"
    l.nummer = 1081
    a.lengte = 4.3
    a.gewicht() = 900
    a.snelheid() = 100
    Set a.locatie = l
    Debug.Print a.lengte; Tab(10); a.gewicht; Tab(20); a.snelheid
    Debug.Print a.locatie.toString()
End Sub
```

In de regel `Set a.locatie = l` kennen we de eerder gedefinieerde Locatie `l` toe aan Auto `a`. In de laatste regel gebruiken we de `toString` method van Locatie.

3.3 VBE

We hebben inmiddels een aantal windows binnen de VBE gezien: het Project Window (paragraaf 2.1), het Code Window (paragraaf 2.1), het Immediate Window (paragraaf 2.2) en het Properties Window (paragraaf 3.2). In het Project Window (**Fout! Verwijzingsbron niet gevonden.**) zie je de volgende onderverdeling:

VBA Project
- Microsoft Excel Objects
- Forms
- Modules
- Class Modules



Figuur 3.2: Project Window

Onder Microsoft Excel Objects staan de standaard objecten van Excel, zoals workbook en worksheets. Onder Forms vind je Formulieren terug, die we in Hoofdstuk 4 uitgebreid zullen behandelen. Onder Modules staan procedures (macro's en functies), die je voor het overzicht kunt onderverdelen in verschillende modules. Merk op dat je in het Code Window bovenin snel kunt springen naar een bepaalde procedure. Ook hier komen we in Hoofdstuk 4 nog uitgebreider op terug. De classes van objecten zoals beschreven in paragraaf 3.2, staan onder Class Modules.

Je kunt ten alle tijde nieuwe forms, modules en class modules toevoegen, door met rechts in het Project Window te klikken en **Insert** te kiezen. Je kunt ook het menu **Insert** gebruiken.

Het laatste onderdeel, de ***Object Browser***, zullen we in paragraaf 3.5 behandelen.

3.4 Scope

De ***scope*** van een variabele, is het gebied, waarin je een variabele of procedure kunt gebruiken. Deze is afhankelijk van de manier waarop een variabele of procedure is gedeclareerd.

3.4.1 Scope van variabelen

VBA kent drie scope-levels voor variabelen:
- Procedure-level;
- Module-level;
- Project-level.

Een variabele die op procedure-level is gedeclareerd, kan alleen binnen die procedure gebruikt worden. Een voorbeeld hiervan is de variabele a uit de macro use in paragraaf 3.2. Deze variabele "bestaat niet" in een andere procedure.

Een variabele die op module-level is gedeclareerd, kan door alle procedures uit de module gebruikt worden. Om een variabele op module-level te declareren, plaats je de declaratie in het Declarations-gedeelte van een module, helemaal bovenaan. Een voorbeeld:

```
Dim b As Integer

Sub gebruikVariabele()
    b = 10
    Debug.Print b
End Sub
```

Een variabele op project-level tenslotte, kun je in het hele project gebruiken. Deze declareer je ook in het Declarations-gedeelte van een module:

```
Public c As Integer
```

Merk op, dat er nu geen `Dim` meer staat, maar `public`. De scope en declaratie van constanten is gelijk aan die van normale variabelen, op de declaratie van een project-level constante na:

```
Public Const d As Integer = 7
```

Hier staat nog wel het keyword `Const`.

3.4.2 Scope van procedures

Voor procedures kent VBA slechts twee scope-levels (wat logisch is): module-level en project-level. Een procedure is standaard op project-level gedeclareerd. Om een procedure op module-level te declareren, plaats je het keyword `Private` voor de procedure:

```
Private Sub interneBerekening()
```

3.4.3 Static

Een procedure-level variabele houdt op te bestaan, als het einde van de procedure bereikt is. Door een variabele static te declareren, blijft deze wél bestaan na het einde van een procedure. De volgende code laat zien hoe dat werkt:

```
Sub eersteSub()
    Static staticVar As Integer
    Dim nonStaticVar As Integer
    If staticVar = 0 Then
        staticVar = 1
        nonStaticVar = 1
    End If
    Debug.Print staticVar; " "; nonStaticVar
End Sub

Sub tweedeSub()
    eersteSub
    eersteSub
End Sub
```

Door tweedeSub aan te roepen, krijg je de output:

```
1   1
1   0
```

Dat komt zo: `staticVar` en `nonStaticVar` krijgen bij declaratie automatisch de waarde 0. Dan is "`staticVar = 0`" true, dus krijgen `staticVar` en `nonStaticVar` de waarde 1. Er wordt dan ook `1 1` geprint. Aan het einde van de procedure wordt `nonStaticVar` weggegooid, maar `staticVar` niet. Dan is "`staticVar = 0`" dus false, krijgt `nonStaticVar` niet de waarde 1, maar houdt de waarde 0 en wordt er `1 0` geprint.

Het gebruik van static variabelen is overigens niet aan te bevelen. Er zijn maar weinig situaties waarin het nuttig is en vaak zorgt het alleen voor verwarring. De situatie hierboven is daar een goed voorbeeld van. Daarnaast heeft een static variabele een eigen stukje geheugenruimte nodig, wat niet bevorderlijk is voor de prestaties van je programma.

3.5 References

Public variabelen en procedures kun je binnen een VBA-project overal aanroepen. Je kunt ze ook in een ander VBA-project aanroepen, maar dan moet je wel een ***reference*** maken. We gaan dat stap voor stap doen, om te kijken hoe dat in zijn werk gaat:

- Maak twee Excelfiles aan: Book1.xls en Book2.xls;
- Start de VBE;
- Selecteer in het Project Window "VBA Project (Book1.xls)". In het Properties Window zie je bij (Name) "VBAProject" staan. Verander dit in "Project1";
- Doe hetzelfde voor het andere project, noem dat echter "Project2";
- Zet de volgende macro in Module1 van Project1:

```
Public Sub pubProc()
    Debug.Print "Hello World"
End Sub
```

- Zet de volgende macro in Module1 van Project2:

```
Sub roepHemAan()
    Project1.Module1.pubProc
    Project1.pubProc
    pubProc
End Sub
```

- Als Project2 nog niet is geselecteerd, selecteer het dan en kies **Tools > References...**. Vink Project1 aan. Er verschijnt onder Project2 een map References, met daarin een reference naar Book1.xls;
- Typ in het Immediate Window **roepHemAan**.

De output is driemaal `Hello World`. De volledige naam van de procedure `pubProc` is `Project1.Module1.pubProc`, maar als `pubProc` binnen `Project1` een unieke naam is, kun je volstaan met `Project1.pubProc`. Als `pubProc` binnen `Project1` én `Project2` uniek is, kun je zelfs met `pubProc` volstaan.

Nu je kennis hebt gemaakt met scopes en references, is het tijd voor de ***Object Browser***. Deze kun je in de VBE openen met **View > Object Browser** of **F2**. Hierin zie je links alle objecten uit het geopende project, plus alle objecten waarnaar een reference bestaat. Standaard zijn dat er al aardig wat. Als je er één aanklikt, zie je rechts alle variabelen en procedures uit dat object. Stel dat je het project met de Auto en Locatie uit paragraaf 3.2 geopend hebt, dan kun je Auto vinden en aanklikken. Rechts zie je dan weight, gewicht, remmen etc. staan.

Als je de ***Solver*** in VBA wilt gebruiken, zul je er ook een reference naar moeten maken. Dat is met name van belang als je met Record Macro een macro opneemt, die de Solver gebruikt.

3.6 Add-Ins

Als je een functie uit een ander workbook alleen op je worksheet wilt gebruiken, hoef je geen reference te maken. Het openen van het andere workbook is voldoende. Plaats bijvoorbeeld de functie `fib` uit paragraaf 2.2 in een nieuw workbook: Book3.xls. Declareer de functie wel ***public***. Maak nu, zonder Book3.xls te sluiten, een nieuw workbook aan. Typ in een willekeurige cel van het nieuwe workbook: **=Book3.xls!fib(3)**.

Zoals gezegd is het openen van het workbook voldoende om een functie daaruit in een ander workbook te gebruiken. Je kunt ook een Add-In van het workbook maken, zodat je dat niet apart hoeft te openen. We maken een Add-In van Book3.xls, dat we net gemaakt hebben. Die we gaan laden in een ander workbook.

- Verander de naam van het project in Book3.xls in Project3, op dezelfde manier als in de vorige paragraaf;
- Compileer het project, door in de VBE **Debug > Compile Project3** te kiezen;
- Sluit de VBE en kies in Excel **File > Save As...**. Kies onder "Save as type" **Microsoft Excel Add-In (*.xla)**. Kies vervolgens de goede directory (standaard wordt de AddIn-directory van Excel gekozen) en sla het bestand daarin als **Book3.xla** op;
- Sluit nu Book3.xls;
- Maak een nieuw workbook aan en sla het op als Book4.xls;
- Kies **File > Open**. Open nu de Add-In die we net gemaakt hebben (Book3.xla);
- Typ nu in een willekeurige cel **=fib(3)** en je ziet dat het werkt.

De Add-In die we net geopend hebben, wordt niet automatisch geopend. Daar kunnen we wel voor zorgen. Kies daartoe **Tools > Add-Ins**. Klik op **Browse...** en open de Add-In. Je krijgt de vraag, of de Add-In naar de library gekopieerd moet worden. Als je op de VU werkt, heb je daarvoor de rechten niet, dus kies je **No**. Book3 staat nu aangevinkt in de lijst. Klik **OK**. Vanaf dat moment wordt elke keer dat je Excel start de Add-In geladen. Door het vinkje voor de Add-In in de lijst van Add-Ins (die je krijgt als je **Tools > Add-Ins** kiest) weg te halen, maak je dat weer ongedaan. Overigens blijft het op de VU slechts werken totdat je uitlogt.

3.7 Collection

Een ***collection*** is de VBA-variant van een genummerde lijst van gelabelde objecten. Een voorbeeld zie je in Tabel 3.2. We gebruiken onze in paragraaf 3.2 geprogrammeerde Auto's als objecten.

Tabel 3.2: Voorbeeld van een Collection.

Collection: een file met auto's (op de A2)		
Nummer	Label	Object

1	“Opel”	Een auto van het merk Opel
2	“Ford”	Een auto van het merk Ford
3	“Toyota”	Een auto van het merk Toyota

We gaan nu deze Collection in VBA programmeren.

```
Sub useCollection()
    Dim file As New Collection ‘op de A2 file != bestand
    Dim a As New Auto
    Dim b As New Auto
    Dim c As New Auto
    a.lengte() = 4.5
    a.snelheid() = 10
    a.gewicht() = 900
    b.lengte() = 5
    b.snelheid() = 9
    b.gewicht() = 1200
    c.lengte() = 4
    c.snelheid() = 8
    c.gewicht() = 1000

    Debug.Print "File is nu: " & file.Count & " auto's"
    file.Add Item:=a, key:="Opel"
    Dim d As Auto
    Set d = file.Item(1)
    Debug.Print "d: " & d.toString()

    file.Add b, "Toyota"

    file.Add c, "Ford", 2
    Debug.Print "1: " & file(1).toString
    Debug.Print "2: " & file(2).toString
    Debug.Print "3: " & file(3).toString

    Debug.Print "File is nu: " & file.Count & " auto's"
    file.Remove (2)
    Debug.Print "File is nu: " & file.Count & " auto's"
    file.Remove ("Toyota")
    Debug.Print "File is nu: " & file.Count & " auto's"
End Sub
```

Eerst maken we een nieuwe Collection aan met `Dim file As New Collection`. Met file bedoel ik in dit verband een rij auto’s, die langzaam achter elkaar aan rijden op de A2. Dan maken we drie objecten die in de Collection gaan komen, de auto’s `a`, `b` en `c`. Om ze te kunnen herkennen geven we ze verschillende properties. De snelheden zijn laag, ze staan immers in de file.

In de regel `Debug.Print "File is nu: " & file.Count & " auto's"` maken we gebruik van de property Count van Collection: deze retourneert het aantal objecten in de Collection.

Met de method `Add` voegen we een object aan de Collection toe. Add kent vier argumenten: Item, Key, Before en After. In Item zet je het object dat je toe wilt voegen, in Key een string met het label. Before en After gebruik je als je een object niet achteraan wilt toevoegen, maar ergens tussenin.

Met de method `Item` kun je een object uit de Collection bekijken. Eerst maken we een vierde Auto aan met `Dim d As Auto`. d laten we vervolgens naar het eerste item van de Collection wijzen met `Set d = file.Item(1)`. Je zou ook `Set d = file.Item(“Opel”)` kunnen gebruiken (daar dient dat label dus voor).

We voegen nog een Toyota toe, vervolgens een Ford, maar die komt vóór de Toyota. Dit kun je aan de uitvoer van de volgende drie regels controleren.

Als laatste gaan we wat objecten uit de Collection verwijderen. `Remove` kun je aanroepen met de index als argument, maar ook met het label.

De output van de macro useCollection() is:

```
File is nu: 0 auto's
d: Gewicht: 900, snelheid: 10, lengte: 4,5.
1: Gewicht: 900, snelheid: 10, lengte: 4,5.
2: Gewicht: 1000, snelheid: 8, lengte: 4.
3: Gewicht: 1200, snelheid: 9, lengte: 5.
File is nu: 3 auto's
File is nu: 2 auto's
File is nu: 1 auto's
```

4 Het maken van een GUI

Een ***Graphical User Interface (GUI)*** kun je gebruiken om gemakkelijk gegevens in te voeren in je VBA-programma of spreadsheet. Daar heb je al een begin mee gemaakt in de cursus Spreadsheets. In Excel kun je via **View > Toolbars > Forms** en **View > Toolbars > Control Toolbox** toolbars laten verschijnen, waarmee je op een worksheet een GUI kunt maken. Een betere manier om een GUI te maken, is in de VBE. Daar heb je meer mogelijkheden en bovendien kun je die manier ook gebruiken in andere Officeapplicaties zoals Word en PowerPoint. We gaan dan ook de laatste manier (VBA) gebruiken.

In dit hoofdstuk gaan we eerst eenvoudige ***dialog boxes*** bekijken, daarna wat complexere ***User Forms*** en tot slot maken we een template voor een kleine ***wizard***. VBA biedt veel mogelijkheden voor een GUI. Het is ondoenlijk om alles in deze handleiding te behandelen. Daarom geldt in het bijzonder voor dit hoofdstuk, dat je veel aan het gebruik van de Help hebt.

4.1 Standaard dialog boxes

Een ***dialog box*** is een eenvoudige ***user form***. De meest eenvoudige heb je al gezien in paragraaf 2.6: de message box (`MsgBox`). In het volgende voorbeeld gebruiken we naast een message box ook ***input boxes***:

```
Function InputAdres() As String
    Dim naam As String
    Dim adres As String
    Dim plaats As String
    MsgBox "Voer nu uw naam en adres in", , "Adres invoeren"
    Do
        naam = InputBox("Naam", "Adres invoeren", naam)
        If naam = "" Then Exit Do
        adres = InputBox("Straat en huisnummer", "Adres invoeren", adres)
        If adres = "" Then Exit Do
        plaats = InputBox("Postcode en woonplaats", "Adres invoeren", plaats)
        If plaats = "" Then Exit Do
        volledigAdres = naam & Chr(13) & adres & Chr(13) & plaats
        msg = "Is het volgende adres correct?" & Chr(13) & volledigAdres
        buttonstyle = vbYesNoCancel + vbDefaultButton1 + vbQuestion
        Select Case MsgBox(msg, buttonstyle, "Adres invoeren")
            Case vbYes
                InputAdres = volledigAdres
                Exit Function
            Case vbNo
            Case vbCancel
                Exit Do
        End Select
    Loop
    InputAdres = ""
End Function
```

Eerst declareren we drie strings voor de informatie die de gebruiker in gaat voeren. De functie `MsgBox` heeft vijf argumenten; we gebruiken de eerste en de derde. Het eerste argument is een string met daarin de tekst die in de dialog box verschijnt. Het tweede argument is een getal, dat het type dialog box specificeert. De default waarde is `vbOKOnly = 0` waarmee een dialog box met alleen een OK-button gespecificeerd wordt. Zie voor een overzicht van alle types dialog boxes de VB-help onder “MsgBox Function”. Het derde argument is de titel van de dialog box, die in de blauwe titelbalk verschijnt. Het vierde en vijfde argument worden voor context-specifieke Help gebruikt. Merk op dat `MsgBox "Voer nu uw naam en adres in", , "Adres invoeren"` equivalent is met `MsgBox prompt:="Voer nu uw naam en adres in", Title:="Adres invoeren"`. Zie Figuur 4.1 voor een afbeelding van deze message box.

Figuur 4.1: Message Box (OK).

Figuur 4.2: Input Box.

Vervolgens gebruiken we een Do-Loop, voor het geval we een foutje maken tijdens het invoeren. De functie `InputBox` retourneert een string met daarin de tekst die in het tekstvak van de dialog box is ingevuld. De string zetten we in `naam`. `InputBox` heeft zeven argumenten, helaas in een andere volgorde dan `MsgBox`. Het eerste argument is de tekst in de dialog box, het tweede is de titel en het derde is de defaultwaarde van het tekstvak. Als defaultwaarde geven we de huidige inhoud van naam mee. Als we dan een foutje maken tijdens het invoeren en we gaan nog een

keer door de loop, dan staat onze vorige invoer al in het tekstvak. Dat is handig met verbeteren. Zoals gezegd retourneert `InputBox` de tekst uit het tekstvak, maar alleen als er **OK** geklikt wordt. Als er **Cancel** geklikt wordt, wordt de lege string geretourneerd. Daarom checken we vervolgens of `naam = ""`, zo ja, dan stoppen we de loop en retourneren we een lege string. Zie **Figuur 4.2** voor een afbeelding van deze input box.

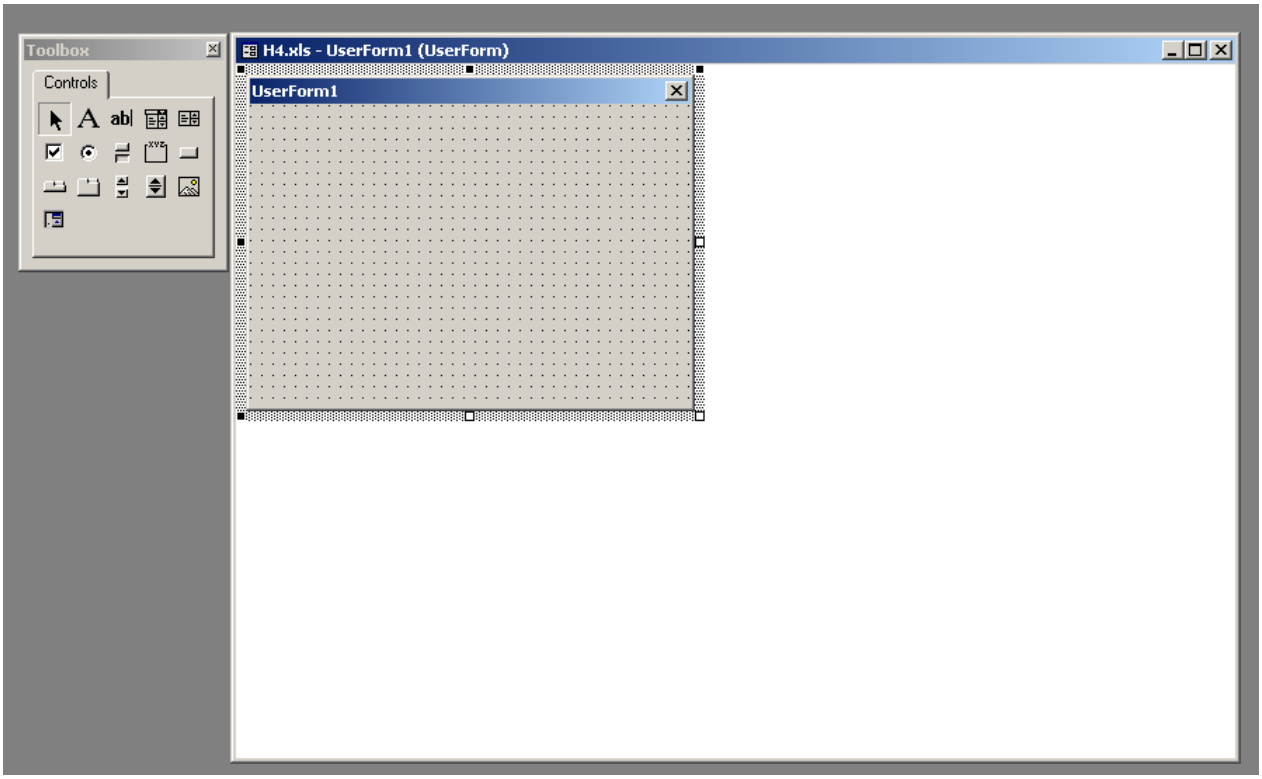
Wat we voor de naam gedaan hebben, doen we ook voor het adres en de plaats. Dan zetten we alle gegevens, gescheiden door regeleindes (`Chr(13)`), in één string `volledigAdres`. Deze string wordt afgebeeld in een message box, met als tweede argument de som van de constanten `vbYesNoCancel`, `vbDefaultButton1` en `vbQuestion`. `vbYesNoCancel` geeft aan, dat we een dialog box met drie buttons willen: Yes, No en Cancel. `vbDefaultButton1` geeft aan dat de eerste van die buttons de standaardbutton is. Als je niet klikt, maar **Enter** geeft, wordt die button gekozen. `vbQuestion` geeft aan, dat er een vraagtekentje in de dialog box moet staan, om het mooi en duidelijk te maken. De waarden van deze constanten zijn zodanig gekozen, dat je voor alle mogelijke combinaties een unieke waarde hebt.

Figuur 4.3: Message Box (Yes, No, Cancel).

De functie MsgBox retourneert een waarde, in dit geval `vbYes`, `vbNo` of `vbCancel`. Als de gebruiker “Yes” heeft geklikt, laten we de functie het ingevoerde adres retourneren. Als de gebruiker “No” heeft geklikt, gaan we nog een keer door de loop, zodat de gebruiker zijn eerder ingetypte gegevens nog kan verbeteren. In het geval van “Cancel” stoppen we de loop en retourneren we een lege string. Zie Figuur 4.3 voor een afbeelding van deze message box.

4.2 User Forms (I)

Iets ingewikkelder windows krijg je met ***User Forms***. Kies **Insert > UserForm**. Je krijgt dan een ontwerpwindow en een Toolbox te zien, ongeveer zoals in **Figuur 4.4**. Mocht deze niet verschijnen, of heb je hem per ongeluk weggeklikt, dan kun je hem terughalen via **View > Toolbox**.

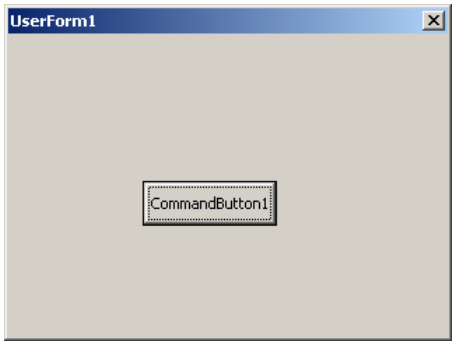


Figuur 4.4: Het ontwerpen van een UserForm.

Het werken hiermee lijkt erg sterk op het werken met een tekenprogramma als bijvoorbeeld Microsoft Paint. Door met de muis over de knoppen te bewegen, zie je in tooltips. Klik in de Toolbox de ***CommandButton*** aan en klik vervolgens in het ontwerpwindow. Je krijgt dan een button van standaardgrootte. Deze kun je eenvoudig vergroten of verkleinen. Als je een actie aan de button wilt toekennen, klik je rechts op de button en kies je **View Code**. Voeg de regel `Me.Hide` toe, zodat je krijgt:

```
Private Sub CommandButton1_Click()
    Me.Hide
End Sub
```

Je start de UserForm door **Run > Run** te kiezen. Je krijgt je window te zien, zoals in Figuur 4.5.



Figuur 4.5: UserForm in actie.

Door op de button te klikken, verdwijnt het window (verrassing!). De button kun je op vele manieren aanpassen door er met rechts op te klikken en **Properties** te kiezen. Dan verschijnen in het window linksonder de properties van de button. Je kunt bijvoorbeeld bij "(Name)" de naam veranderen waarmee je aan de button gaat refereren. Deze naam is nu "CommandButton1" en dat laten we zo. Bij "Caption" kun je de tekst op de button veranderen.

Maak nu twee ***OptionButtons*** aan. Klik de bovenste met rechts aan en kies **View Code**. Voeg de regel `CommandButton1.Caption = "Eén"` toe, zodat je krijgt:

```
Private Sub OptionButton1_Click()
    CommandButton1.Caption = "Eén"
End Sub
```

Voeg op dezelfde manier de regel `CommandButton1.Caption = "Twee"` aan de andere OptionButton toe. Als je de User Form start, zie je het effect van deze regels. Wat nog niet zo netjes is, is dat de initiële waarde "CommandButton1" is. Daarom zetten we OptionButton1 standaard aan. Daartoe klik je rechts op een lege plaats in de User Form en kies je View Code. De volgende code verschijnt:

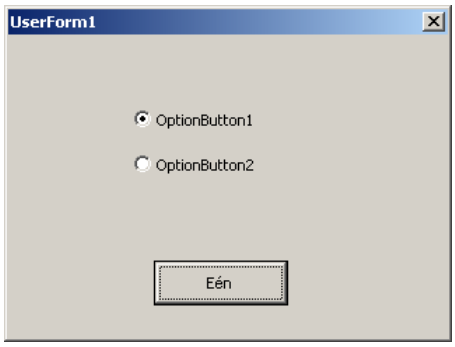
```
Private Sub UserForm_Click()

End Sub
```

Dit verander je in:

```
Private Sub UserForm_Initialize()
        OptionButton1.Value = True
End Sub
```

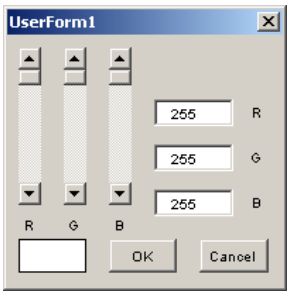
Het uiteindelijke resultaat is te zien in Figuur 4.6.



Figuur 4.6: De uiteindelijke UserForm.

4.3 User Forms (II)

We gaan nu een UserForm maken, die iets ingewikkelder is. Het uiteindelijke resultaat wordt een window als in Figuur 4.7.



Figuur 4.7: RGB-editor.

Kleuren worden vaak digitaal opgeslagen als RGB-waarden. Dit zijn drie getallen (0-255) die de hoeveelheid Rood, Groen en Blauw in de kleur aangeven. Het vlak linksonder geeft de kleur weer, die de RGB-waarden definiëren. Om het geheel wat gebruiksvriendelijker te maken, hebben we ook drie schuifbalken om de RGB-waarden in te stellen.

We beginnen met het aanmaken van een UserForm door **Insert > UserForm** te kiezen. Daarin maken we twee CommandButtons aan: “OK” en “Cancel” (hint: caption). Vervolgens gebruiken we een ***Image*** voor het kleurvakje. De schuifbalken maken we met ***ScrollBars*** en de RGB-waarden voeren we in in ***TextBoxes***. De letters R, G en B, die je in Figuur 4.7 ziet, maken we met ***Labels***. Nu hebben we een UserForm als in Figuur 4.7, maar hij doet nog niets.

Klik nu in de Project Explorer rechts op de betreffende UserForm. Kies **View Code**. Het code window dat dan verschijnt is in principe leeg, maar misschien staan er wat lege procedures. Eerst declareren we drie private variabelen in het Declarations-gedeelte om de RGB-waarden in op te slaan:

```
Private Red As Integer
Private Green As Integer
Private Blue As Integer
```

Dan gaan we de UserForm initialiseren door in een leeg stuk ervan dubbel te klikken en de volgende code toe te voegen:

```
Private Sub UserForm_Initialize()
    'We willen standaard de kleur wit
    Red = 255
    Green = 255
    Blue = 255

    'OK is de default button
    CommandButton1.SetFocus

    'Standaard Tag
    Me.Tag = vbAbort

    'ScrollBars initialiseren
    ScrollBar1.Min = 0
    ScrollBar2.Min = 0
    ScrollBar3.Min = 0
    ScrollBar1.Max = 255
    ScrollBar2.Max = 255
    ScrollBar3.Max = 255
End Sub
```

We zetten de Tag van de UserForm standaard op vbAbort. Dan heeft de Tag automatisch de goede waarde als we de UserForm met het kruisje rechtsboven afsluiten. We geven de UserForm een andere Tag op het moment dat er op **OK** of **Cancel** geklikt wordt. Klik daartoe rechts de OK-button, kies **View Code** en voeg de volgende code toe:

```
Private Sub CommandButton1_Click()
    Me.Tag = vbOK
    Me.Hide
```

```
End Sub
```

Doe vrijwel hetzelfde voor de Cancel-button:

```
Private Sub CommandButton2_Click()
    Me.Tag = vbCancel
    Me.Hide
End Sub
```

Als je de UserForm nu start, zie je nog geen witte kleur in het vlak. Dat komt, doordat er nog nergens iets met de variabelen `Red`, `Green` en `Blue` gedaan wordt. Bovendien moeten de juiste waarden in de TextBoxes gezet worden en de schuifbalken op de juiste plaats. Daarom maken we van de kleuren properties:

```
Private Property Let Rood(R As Integer)
    Red = R
    TextBox1.Value = Red
    ScrollBar1.Value = Red
    Image1.BackColor = Kleur 'deze property moeten we nog schrijven
End Property
```

Je ziet dat de juiste roodwaarde in de TextBox wordt gezet en dat de ScrollBar in de juiste positie geplaatst wordt. Merk op dat hierbij de volgorde waarin de TextBoxes en ScrollBars aangemaakt zijn belangrijk is (als je met de rode begonnen bent, krijgen die volgnummer 1). De property Kleur moeten we nog schrijven, dat doen we nu:

```
Public Property Get Kleur() As Long
    Kleur = RGB(Red, Green, Blue)
End Property
```

De functie RGB is al in VBA gedefinieerd en retourneert een unieke Longwaarde voor alle kleuren die je met RGB-waarden kunt maken. We maken nu de procedures `Property Let` voor de andere kleuren:

```
Private Property Let Groen(G As Integer)
    Green = G
    TextBox2.Value = Green
    ScrollBar2.Value = Green
    Image1.BackColor = Kleur 'deze property moeten we nog schrijven
End Property

Private Property Let Blauw(B As Integer)
    Blue = B
    TextBox3.Value = Blue
    ScrollBar3.Value = Blue
    Image1.BackColor = Kleur 'deze property moeten we nog schrijven
End Property
```

Nu zijn we er nog steeds niet, want tijdens de initialisatie moeten deze properties gebruikt worden, anders gebeurt er nog steeds niet. Daarom herschrijven we `UserForm_Initialize()`:

```
Private Sub UserForm_Initialize()
    'We willen standaard de kleur wit
    Rood = 255
    Groen = 255
    Blauw = 255
    'OK is de default button
    CommandButton1.SetFocus
    Me.Tag = vbAbort
End Sub
```

Nu zien we dat het vlakje standaard de kleur wit krijgt. Nu gaan we programmeren, wat er gebeurt, als we met een ScrollBar schuiven. Schrijf daartoe de volgende macro's:

```
Private Sub ScrollBar1_Change()
    Rood = ScrollBar1.Value
End Sub

Private Sub ScrollBar1_Scroll()
    Rood = ScrollBar1.Value
End Sub
```

Dit moet uiteraard ook gebeuren voor de andere ScrollBars:

```
Private Sub ScrollBar2_Change()
    Groen = ScrollBar2.Value
End Sub

Private Sub ScrollBar2_Scroll()
    Groen = ScrollBar2.Value
```

```
End Sub

Private Sub ScrollBar3_Change()
    Blauw = ScrollBar3.Value
End Sub

Private Sub ScrollBar3_Scroll()
    Blauw = ScrollBar3.Value
End Sub
```

Dan hebben we nog de TextBoxes. We willen dat de waarde geëvalueerd wordt op het moment dat we de TextBox verlaten. Daarom schrijven we:

```
Private Sub Textbox1_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    Dim waarde As Integer
    waarde = val(TextBox1.Value)
    If (waarde < 0) Then
        waarde = 0
    ElseIf (waarde > 255) Then
        waarde = 255
    End If
    Rood = waarde
End Sub
```

Het argument van een Exit-procedure is nu eenmaal `ByVal Cancel As MSForms.ReturnBoolean`. Zie voor meer informatie hierover de VBA-help. De waarde van een TextBox is een string, die we met de functie `val` omzetten naar een integer. We kopiëren deze macro voor de andere TextBoxes:

```
Private Sub Textbox2_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    Dim waarde As Integer
    waarde = val(TextBox2.Value)
    If (waarde < 0) Then
        waarde = 0
    ElseIf (waarde > 255) Then
        waarde = 255
    End If
    Groen = waarde
End Sub

Private Sub Textbox3_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    Dim waarde As Integer
    waarde = val(TextBox3.Value)
    If (waarde < 0) Then
        waarde = 0
    ElseIf (waarde > 255) Then
        waarde = 255
    End If
    Blauw = waarde
End Sub
```

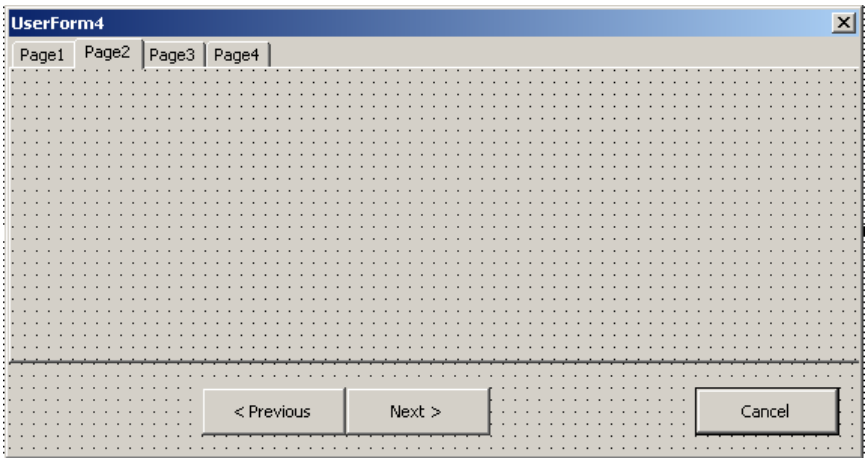
We hebben nu een volledig werkende RGB-editor.

4.4 Een wizard

In Windows is het gebruikelijk om ingewikkelde procedures voor een gebruiker eenvoudiger te maken door middel van een ***wizard***. We gaan nu het skelet van een wizard in VBA maken door middel van een ***MultiPage***: een window met een aantal tabbladen. Op elk tabblad komt een stap van de wizard.

Maak eerst een UserForm aan (**Insert > UserForm**). Maak het window iets breder en voeg over het gehele window een MultiPage in, maar laat aan de onderkant een stuk vrij voor de navigatiebuttons. We gaan een vier-stappen wizard maken, dus klikken we rechts op één van de tabs en kiezen **New Page**. Dit herhalen we totdat we vier tabbladen hebben.

In een wizard zie je normaal gesproken geen tabs. Later zullen we de tabs verbergen, maar tijdens het bouwen is het erg handig om ze nog te laten staan. Voeg nog drie CommandButtons toe, zodat je een window krijgt dat op Figuur 4.8 lijkt.



Figuur 4.8: Het begin van een wizard.

We gaan de puntjes op de i zetten. Klik rechts op de UserForm en kies **View Code**. Voeg de volgende code toe:

```
Private Sub UserForm_Initialize()
    MultiPage1.Value = 0 'de telling van de pages begint bij 0
    CommandButton1.Enabled = False
    If (MultiPage1.Pages.Count = 1) Then
        CommandButton2.Caption = "Finish"
    End If
    SetCaption
End Sub

Private Sub SetCaption()
     Caption = "Wizard step " & MultiPage1.Value + 1 & " of " & MultiPage1.Pages.Count
End Sub
```

Ik zal in het vervolg de tabbladen met ***pagina's*** aanduiden. De eerste regel specificeert dat we op pagina 1 beginnen. Dan kunnen we natuurlijk niet terug, dus zetten we de Previous-button uit. Mocht onze wizard slechts één pagina bevatten, dan wordt de Next-button een Finish-button. De titel veranderen we in een aparte macro: `SetCaption()`, omdat we de titel bij elke stap van de wizard willen veranderen. Dan voeren we nog wat Cancel-button code in:

```
Private Sub CommandButton3_Click()
    Me.Tag = vbCancel
    Unload Me
End Sub
```

Merk op dat we nu niet Me.Hide gebruiken om de UserForm te stoppen, maar Unload Me. Bij Me.Hide wordt het window alleen verborgen, bij Unload Me wordt het window helemaal uit het geheugen gewist. Als we hier Me.Hide zouden gebruiken en we zouden in stap 3 op Cancel klikken, dan zou de volgende wizard op pagina 3 beginnen. Nu start de wizard altijd op pagina 1. Dan gaan we de Next-button programmeren. Voeg de volgende code toe:

```
Private Sub CommandButton2_Click()
    If ((MultiPage1.Value + 1) = MultiPage1.Pages.Count) Then
        Me.Tag = vbOK
        Unload Me
    Else
        MultiPage1.Value = MultiPage1.Value + 1
        If (MultiPage1.Value + 1 = MultiPage1.Pages.Count) Then
            CommandButton2.Caption = "Finish"
        End If
        CommandButton1.Enabled = True
        SetCaption
    End If
End Sub
```

Eerst checken we of we op de laatste pagina zijn. Is dat het geval, dan verlaten we de wizard. Als we niet op de laatste pagina zijn, gaan we naar de volgende pagina. Het kan dan zijn, dat we op de laatste pagina aangekomen zijn. In dat geval, moet de Next-button een Finish-button worden. Als we Next hebben geklikt, weten we zeker dat we niet op de eerste pagina zijn, dus de Previous-button moet werken. Als laatste passen we de titel aan.

De enige button die we nu nog moeten programmeren is de Previous-button:

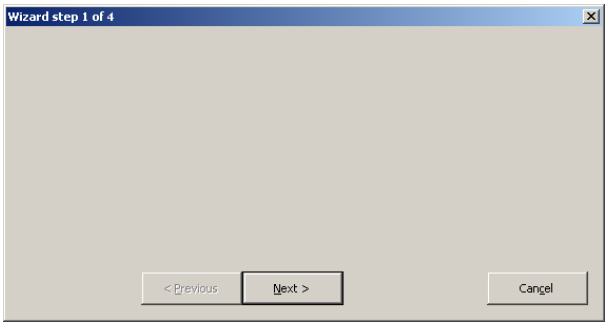
```
Private Sub CommandButton1_Click()
    MultiPage1.Value = MultiPage1.Value - 1
    If (MultiPage1.Value = 0) Then
        CommandButton1.Enabled = False
```

```
        End If
        CommandButton2.Caption = "Next >"
        SetCaption
    End Sub
```

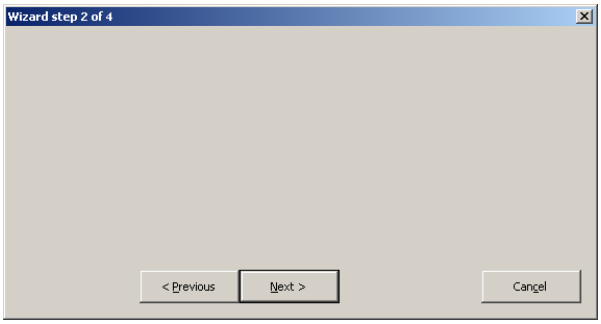
Deze code is nu niet meer moeilijk te begrijpen. Als we op de eerste pagina aankomen, mag de Previous-button niet meer werken. Verder kunnen we niet meer op de laatste pagina zijn, dus op de Next-button moet “Next >” staan (en geen Finish).

We kunnen de wizard nog iets verfraaien door de button ook met het toetsenbord bereikbaar te maken. Dat doe je door in het properties window een CommandButton te kiezen en bij Accelerator een letter in te vullen. Daarnaast kun je de Next/Finish-button als standaard instellen bij de property Default.

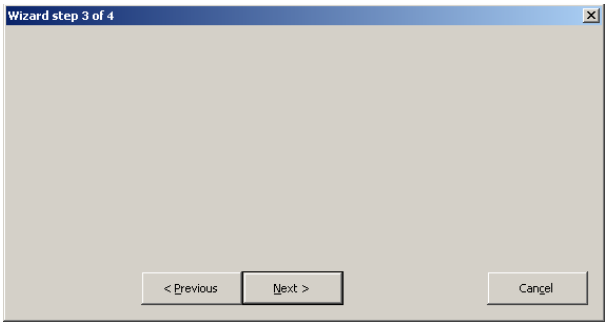
We zijn nu klaar met bouwen, dus het is tijd geworden om de tabs te verwijderen. In het properties window kiezen we daartoe eerst MultiPage1 en vervolgens zetten we Style op “2 - fmTabStyleNone”. De wizard is nu klaar, zie Figuur 4.9 tot en met Figuur 4.12.



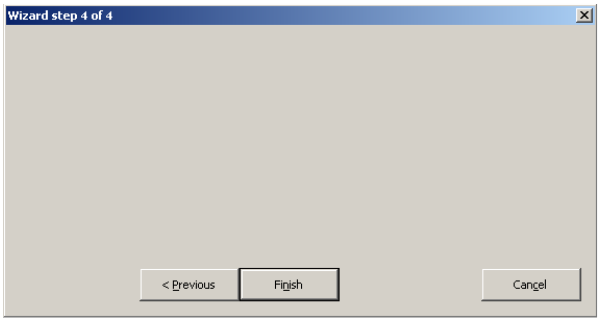
Figuur 4.9: Wizard, stap 1.



Figuur 4.10: Wizard, stap 2.



Figuur 4.11: Wizard, stap 3.



Figuur 4.12: Wizard, stap 4.

5 Samenwerking met andere programma's

Bij Matlab kun je een Excel-koppeling kopen, die voor normaal gebruik $ 9.400 kost (alleen de koppeling, dus zonder Matlab zelf). Er is wel een studentenversie, maar waarom zou je hem niet zelf programmeren? In dit hoofdstuk gaan dat doen, maar eerst is het tijd voor een eenvoudiger voorbeeld, namelijk het uitvoeren van een DOS-commando.

5.1 Een DOS-commando

We gaan in VBA een DOS-commando geven en de output van dat commando inlezen. Je kunt met een DOS-commando bijvoorbeeld een Java-programma starten, maar in dit geval zullen we het eenvoudige "dir" gebruiken, dat de inhoud van de directory weergeeft. Zie hieronder het begin van de functie `System`:

```
Function System(cmdline As String)
    tempdir = Environ("TEMP") & "\"
    If Dir(tempdir & "tmp.bat") <> "" Then
        Debug.Print tempdir & "tmp.bat already exists"
        System = CVErr(xlErrNA)
        Exit Function
    End If
    If Dir(tempdir & "tmp.out") <> "" Then
        Debug.Print tempdir & "tmp.out already exists"
        System = CVErr(xlErrNA)
        Exit Function
    End If
    If Dir(tempdir & "tmp.run") <> "" Then
        Debug.Print tempdir & "tmp.run already exists"
        System = CVErr(xlErrNA)
        Exit Function
    End If
    ...
```

Het commando dat we uit willen voeren geven we als String mee aan System. Windows gebruikt een ***omgevingsvariabele*** TEMP om aan te geven dat programma's daar hun tijdelijke files neer moeten zetten. Dat doen wij ook netjes. In de variabele `tempdir` zetten we de TEMP-directory gevolgd door een backslash. Dan kijken we of er al files bestaan met de namen die wij willen gebruiken(tmp.bat, tmp.out en tmp.run). Als dat zo is, dan geven we een foutmelding en stoppen we de functie.

Vervolgens schrijven we een hulp-batch-file, waarin onder andere ons commando zit:

```
    ...
    freeFileNumber = FreeFile
    Open tempdir & "tmp.bat" For Output As freeFileNumber
    Print #freeFileNumber, "echo > " & tempdir & "tmp.run"
    Print #freeFileNumber, cmdline & " > " & tempdir & "tmp.out"
    Print #freeFileNumber, "del " & tempdir & "tmp.run"
    Close freeFileNumber
    ...
```

In `freeFileNumber` zetten we het nummer dat het `Open`-statement nodig heeft. Het nummer representeert een plaats waar de file komt te staan. Deze plaats moet vrij zijn, voordat we daar een file kunnen maken. Omdat we naar de file gaan schrijven, specificeren we `For Output`. Dan schrijven we enige Strings in de file en sluiten de file met `Close freeFileNumber`. Deze reeks commando's creëert de volgende batch-file:

```
echo > C:\Temp\tmp.run
dir > C:\Temp\tmp.out
del C:\Temp\tmp.run
```

Merk op dat in dit geval C:\Temp de TEMP-directory is (dat kan op een andere computer dus anders zijn) en dat we als commando "dir" hebben gekozen. De eerste regel maakt een lege file aan met de naam tmp.run. Deze dient als ***lock-file***. Hierop komen we later terug. De tweede regel voert ons commando uit en stuurt de output ervan naar C:\temp\tmp.out. In de derde regel wordt de lock-file verwijderd (met het DOS-commando *del*, dat synoniem is met *erase*), wat het teken is, dat ons commando klaar is.

Nu is het tijd om ons commando uit te voeren:

```
    ...
    Shell tempdir & "tmp.bat", vbHide
    While Dir(tempdir & "tmp.out") = ""
        DoEvents
    Wend
    While Dir(tempdir & "tmp.run") <> ""
        DoEvents
    Wend
```

```
    ...
```

Shell neemt als argumenten het commando en een variabele `vbHide`, die ervoor zorgt dat er geen MSDOS-Promptwindow geopend wordt. Vervolgens wachten we tot de file tmp.out bestaat. `Dir(tempdir & "tmp.out")` retourneert namelijk de volledige padnaam van tmp.out als deze file bestaat. Zolang er een lege String wordt geretourneerd, bestaat de file dus nog niet. DoEvents is een macro die niets doet en bedoeld is voor situaties waarin gewacht moet worden. Zodra de file bestaat (dat is waarschijnlijk al heel snel), wordt gekeken of de lock-file bestaat. Deze blijft namelijk bestaan, zolang ons commando nog bezig is, zie de batch-file. Zodra de lock-file verwijderd is, gaat de functie verder:

```
    ...
    tmpout = FreeFile
    Open tempdir & "tmp.out" For Input As tmpout
    result = ""
    Do Until EOF(tmpout)
        Line Input #tmpout, textline
        result = result & Chr(13) & textline
    Loop
    Close tmpout

    Kill tempdir & "tmp.bat"
    Kill tempdir & "tmp.out"
    system = result
End Function
```

We gaan de outputfile tmp.out inlezen en daarom openen we hem `For Input`. Met `Line Input` lezen we steeds een regel in, net zolang, tot de end-of-file bereikt is.

Vervolgens ruimen we de door ons aangemaakte bestanden op en retourneren we wat we ingelezen hebben.

Hieronder volgt de gehele functie:

```
Function system(cmdline As String)
    tempdir = Environ("TEMP") & "\"
    If Dir(tempdir & "tmp.bat") <> "" Then
        Debug.Print tempdir & "tmp.bat already exists"
        system = CVErr(xlErrNA)
        Exit Function
    End If
    If Dir(tempdir & "tmp.out") <> "" Then
        Debug.Print tempdir & "tmp.out already exists"
        system = CVErr(xlErrNA)
        Exit Function
    End If
    If Dir(tempdir & "tmp.run") <> "" Then
        Debug.Print tempdir & "tmp.run already exists"
        system = CVErr(xlErrNA)
        Exit Function
    End If

    freeFileNumber = FreeFile
    Open tempdir & "tmp.bat" For Output As freeFileNumber
    Print #freeFileNumber, "echo > " & tempdir & "tmp.run"
    Print #freeFileNumber, cmdline & " > " & tempdir & "tmp.out"
    Print #freeFileNumber, "del " & tempdir & "tmp.run"
    Close freeFileNumber

    Shell tempdir & "tmp.bat", vbHide
    While Dir(tempdir & "tmp.out") = ""
        DoEvents
    Wend

    While Dir(tempdir & "tmp.run") <> ""
        DoEvents
    Wend

    tmpout = FreeFile
    Open tempdir & "tmp.out" For Input As tmpout
    result = ""
    Do Until EOF(tmpout)
        Line Input #tmpout, textline
        result = result & Chr(13) & textline
    Loop
    Close tmpout

    Kill tempdir & "tmp.bat"
    Kill tempdir & "tmp.out"
    system = result
End Function
```

5.2 Automation

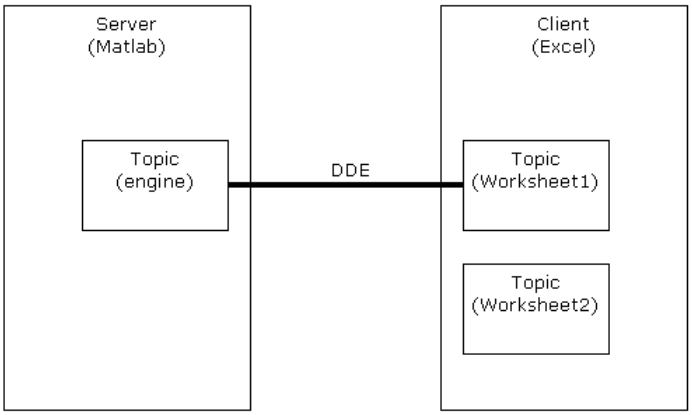
Als je programma's wilt laten samenwerken, heb je vaak te maken met ***Automation***. Automation is een technologie, waarbij je een ***controller application*** en een ***object application*** hebt. Vanuit de controller application kun je de object application manipuleren. Sommige programma's ondersteunen helemaal geen Automation, sommige alleen als controller of object application en andere ondersteunen Automation volledig. De naam Automation is overigens door Microsoft bedacht.

Excel ondersteunt Automation volledig. Vaak wordt er gewerkt met een ***master*** en een ***slave*** object van Excel. Je maakt een nieuw object van Excel met één van de volgende twee opdrachtregels:

```
Set slave = New Excel.Application
Set slave = CreateObject("Excel.Application")
```

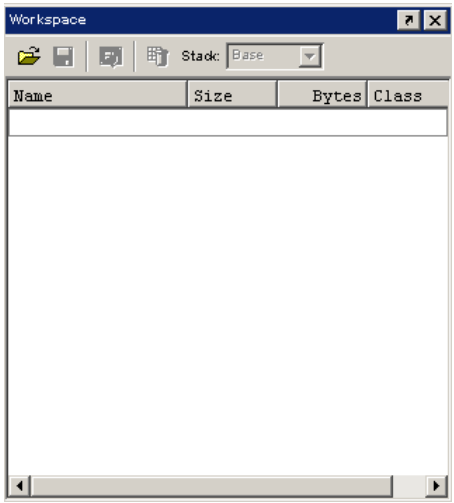
5.3 DDE-verbinding met Matlab

Dynamic Data Exchange (DDE) is een oudere techniek dan Automation, maar uitermate geschikt om Excel met Matlab te laten communiceren. ***ActiveX***, bekend van internet, is de opvolger van DDE. DDE werkt op de volgende manier (Figuur 5.1).



Figuur 5.1: DDE

Er is een ***server*** (Matlab) en een ***client*** (Excel). Een ***topic*** van de client maakt verbinding met een topic van de server. In Excel is elke geopende worksheet een topic; Matlab heeft slechts één topic, de ***workspace***, waarin alle variabelen staan, zie Figuur 5.2. Deze topic wordt aangeduid met "engine".



Figuur 5.2: De Workspace van Matlab

De client maakt verbinding met de server, voordat er communicatie plaats kan vinden. Na de communicatie verbreekt de client de verbinding weer. Excel heeft vijf ingebouwde DDE-methods:

- DDEInitiate om verbinding te maken;
- DDEPoke om gegevens te versturen;
- DDERequest om gegevens op te vragen;
- DDEExecute om opdrachten uit te voeren;
- DDETerminate om de verbinding te verbreken.

Het nadeel van deze methods is dat ze niet te gebruiken zijn bij de herberekening van een werkblad. Om dat probleem op te lossen, laten we deze functies uitvoeren door *een ander object* van Excel. Het ***master***-object laat het ***slave***-object een DDE-verbinding met Matlab maken. Je maakt een nieuw Excel-object met één van de volgende twee opdrachtregels:

```
Set slave = New Excel.Application
Set slave = CreateObject("Excel.Application")
```

We gaan nu een Add-In maken, die een DDE-verbinding met Matlab regelt.

Maak in een nieuw project een Class Module SlaveExcel aan met de volgende code erin:

```
Private slave As Excel.Application
Private Sub Class_Initialize()
    Set slave = New Excel.Application
    slave.Workbooks.Add
End Sub
Public Property Get Application() As Excel.Application
    Set Application = slave
End Property
Public Property Get sheet() As Excel.Worksheet
    Set sheet = slave.ActiveSheet
End Property
Private Sub Class_Terminate()
    Dim wb As Excel.Workbook
    For Each wb In slave.Workbooks
        wb.Close False
    Next
    slave.Quit
End Sub
```

We gebruiken een object van deze Class Module in een aparte module, die we Locals noemen:

```
Option Private Module
Public slave As New SlaveExcel
```

De regel `Option Private Module` zorgt ervoor, dat niets uit die module buiten het project zichtbaar is. Met andere woorden: `slave` is alleen binnen het project te gebruiken, maar wel in alle modules vanwege de `public` declaratie. Het voordeel hiervan is, dat als we het Matlab-koppelingsproject als Add-In gaan gebruiken, de variabele `slave` alleen door de Add-In gebruikt kan worden.

In de volgende module MatlabInterface komen de Matlab-DDE-functies:

```
Public Function MatlabDDERequest(Item As String)
    channel = slave.Application.DDEInitiate("matlab", "engine")
    MatlabDDERequest = slave.Application.DDERequest(channel, Item)
    slave.Application.DDETerminate channel
End Function
```

Je ziet wat er gebeurt: eerst laten we ons slave-object een verbinding met "engine" van "matlab" maken. "matlab" is hier de naam van het Windowsproces waarin Matlab draait. Matlab moet op dat moment wel reeds opgestart zijn. Vervolgens wordt uit Matlab de String `Item` opgevraagd. In `Item` (het argument van de functie) moet dan de naam van een variabele staan. Deze functie retourneert uiteraard de waarde van `Item`.

De volgende functie is de `Poke`-functie:

```
Public Function MatlabDDEPoke(Item, Data, Optional UseSlave = True)
    If UseSlave Then
        Set Data2 = slave.sheet.Range(Data.Address)
        For i = 1 To Data.Rows.Count
            For j = 1 To Data.Columns.Count
                Data2.Cells(i, j).Value = Data.Cells(i, j).Value
            Next j
        Next i
        Set app = slave.Application
    Else
        Set Data2 = Data
```

```
            Set app = Application
        End If
        channel = app.DDEInitiate("matlab", "engine")
        app.DDEPoke channel, Item, Data2
        app.DDETerminate channel
    End Function
```

Deze functie maakt standaard gebruik van ons slave-object, maar we kunnen er ook voor kiezen om de data direct te verzenden, door UseSlave = False mee te geven. Verder komt in `Item` de naam van de variabele en in `Data` de waarde(n). We beginnen (als UseSlave = True) met het maken van een kopie van de meegegeven data in de slave-application. Dan maken we verbinding en stoppen de waarden van de kopie in een nieuwe variabele met als naam de String `Item`. Tot slot verbreken we de verbinding.

De laatste functie is de Execute-functie:

```
Public Function MatlabDDEExecute(cmd As String)
    channel = slave.Application.DDEInitiate("matlab", "engine")
    slave.Application.DDEExecute channel, cmd
    slave.Application.DDETerminate channel
End Function
```

Deze functie gebruikt heel eenvoudig de VBA-functie `DDEExecute`. De DDE-functies gaan we gebruiken in een Command Bar. Daarom maken we een module MatlabCommandBar aan en daarin zetten we de volgende declaratie en functies:

```
Option Private Module
Sub setup()
    ClearCommandBar "Matlab"
    With Application.CommandBars("Matlab")
        With .Controls
            With .Add(msoControlComboBox)
                .Caption = "Variable"
                .TooltipText = "Variabele in Matlab"
                .OnAction = "InputMatlabVariable"
            End With
            With .Add(msoControlButton)
                .Caption = "&Put"
                .Style = msoButtonCaption
                .TooltipText = "Vervang variabele in Matlab door Selection"
                .OnAction = "PutMatlabVariable"
            End With
            With .Add(msoControlButton)
                .Caption = "&Get"
                .Style = msoButtonCaption
                .TooltipText = "Vervang Selection door variabele in Matlab"
                .OnAction = "GetMatlabVariable"
            End With
        End With
    End With
End Sub
```

`Option Private Module` hebben we eerder in deze paragraaf al gezien, de macro `setup()` maakt de CommandBar aan. Eerst wordt de functie `ClearCommandBar` aangeroepen. Deze functie, die zorgt dat er niet al een MatlabCommandBar bestaat, gaan we zo programmeren. Vervolgens voegen we een ControlComboBox toe, waarin de naam van de variabele gaat komen. Daarnaast komen twee buttons: Put en Get, met voor de hand liggende functies. Dan nu de functie ClearCommandBar:

```
Sub ClearCommandBar(naam As String)
    On Error GoTo nieuw
    Set bar = Application.CommandBars(naam)
    For Each c In bar.Controls
        c.Delete
    Next c
    Exit Sub
nieuw:
    With Application.CommandBars.Add(naam)
        .Visible = True
        .Position = msoBarTop
    End With
    Resume
End Sub
```

Deze functie haalt eerst elke CommandBar met de naam `naam` weg en voegt er daarna één toe.

De ControlComboBox en de buttons uit de macro `setup` maken gebruik van de volgende drie macro's:

```
Sub InputMatlabVariable()
    With Application.CommandBars("Matlab").Controls("Variable")
        If .ListIndex = 0 Then
```

```
            .AddItem .Text
        End If
    End With
End Sub
Sub PutMatlabVariable()
    With Application.CommandBars("Matlab").Controls("Variable")
        If .Text = "" Then
            MsgBox "Geef eerst de naam van een variabele in Matlab op!"
            Exit Sub
        End If
        MatlabDDEPoke .Text, Selection
    End With
End Sub
Sub GetMatlabVariable()
    With Application.CommandBars("Matlab").Controls("Variable")
        If .Text = "" Then
            MsgBox "Geef eerst de naam van een variabele in Matlab op!"
            Exit Sub
        End If
        Selection.Value = MatlabDDERequest(.Text)
    End With
End Sub
```

De werking van deze macro's zal duidelijk worden als we ze gebruiken. Tot slot programmeren we nog een macro destroy, die de CommandBar weghaalt als we de Add-In sluiten.

```
Sub destroy()
    On Error Resume Next
    Application.CommandBars("Matlab").Delete
End Sub
```

We kunnen nu variabelen ophalen en versturen van en naar Matlab. De volgende drie functies maken het mogelijk om Matlabfuncties vanuit VBA uit te voeren. Allereerst de functie MLeval:

```
Public Function MLeval(exp As String, ParamArray args())
    For i = 0 To UBound(args)
        MatlabDDEPoke "ExcelTemp" & i, args(i)
    Next i
    MatlabDDEExecute ParseCmd(exp)
    MLeval = MatlabDDERequest("ans")
End Function
```

Eerst stopt Mleval alle meegegeven argumenten in de vector ExcelTemp. Vervolgens wordt het commando `exp` uitgevoerd. De functie retourneert de uitkomst (`ans`). De functie Mlexec doet precies hetzelfde, maar retourneert niet de uitkomst:

```
Public Function MLexec(cmd As String, ParamArray args())
    For i = 0 To UBound(args)
        MatlabDDEPoke "ExcelTemp" & i, args(i)
    Next i
    MatlabDDEExecute ParseCmd(cmd)
    MLexec = "OK"
End Function
```

Beide functies maken gebruik van de functie ParseCmd:

```
Private Function ParseCmd(cmd As String) As String
    ParseCmd = ""
    src = cmd
    While src <> ""
        eerste = InStr(1, src, "%")
        tweede = InStr(eerste + 1, src, "%")
        Select Case tweede - eerste
        Case 0
            ParseCmd = ParseCmd & src
            src = ""
        Case 1
            ParseCmd = ParseCmd & Left(src, eerste)
            src = Mid(src, tweede + 1)
        Case Else
            ParseCmd = ParseCmd & Left(src, eerste - 1) & "ExcelTemp"
            ParseCmd = ParseCmd & Mid(src, eerste + 1, tweede - eerste - 1)
            src = Mid(src, tweede + 1)
        End Select
    Wend
End Function
```

Hoe deze functie precies werkt, laat ik aan de lezer over om uit te zoeken met behulp van de Visual Basic Help. De functie retourneert een geldig Matlabcommando. Op de manier van Paragraaf 3.6 maken we van dit alles een Add-In, die we Matlab.xla noemen. Gebruik als naam voor het project MatlabInterface.

We kunnen de functies als volgt testen:

- Start Matlab op;
- Maak in het Command Window een variabele aan: **matlab_variabele = [1;2;3]**; In de workspace zie je die variabele dan ook verschijnen;
- Start Excel op en open de Add-In Matlab.xla. De CommandBar verschijnt;
- Typ in de ControlComboBox **matlab_variabele** en druk op **Enter**;
- Selecteer cel A4 en klik op **Get**;
- Selecteer de area B2:B4 en klik op **Get**;
- Selecteer de area C1:C4 en klik op **Get**;
- Typ in cel D1 **456**, klik in de ControlComboBox, typ **Excelvar01** en geef **Enter**;
- Klik op **Put**;
- Typ in cel D2 **9**, selecteer D1:D2, klik in de ControlComboBox, typ **Excelvar02** en geef **Enter**;
- Klik op **Put**;
- Selecteer A1:D1, klik in de ControlComboBox, typ **Excelvar03** en geef **Enter**;
- Klik op **Put**;
- Selecteer A1:D4, klik in de ControlComboBox, typ **Excelvar04** en geef **Enter**;
- Klik op **Put**;
- Bekijk in Matlab de variabelen Excelvar01, Excelvar02, Excelvar03 en Excelvar04.

Je begrijpt nu hoe de CommandBar werkt.

We gaan nu bewijzen dat Matlab inderdaad nauwkeuriger werkt dan VBA. Maak met behulp van de functie HilbertMatrix uit Paragraaf 2.5.4 de 9x9 Hilbertmatrix in de cellen B3:K12. We gaan nu de inverse van deze matrix door Matlab laten berekenen, door de cellen B14:K23 te selecteren, **=mleval("inv(%0%)";B3:K12)** te typen en te bevestigen met **Ctrl-Shift-Enter**.

Excel kan zelf de inverse ook berekenen. Selecteer daartoe de cellen B25:K34, typ **=minverse(B3:K12)** en bevestig met **Ctrl-Shift-Enter**.

De Hilbertmatrix met zichzelf vermenigvuldigd zou de eenheidsmatrix op moeten leveren. Door de numerieke benadering zullen computers echter een afrondingsfout maken. De afrondingsfout van Matlab is echter veel kleiner dan die van Excel, zoals we nu zullen aantonen.

Selecteer de cellen M14:V23 en typ **=mmult(B14:K23;B3:K12)** en bevestig met **Ctrl-Shift-Enter**.
Selecteer de cellen M25:V34 en typ **=mmult(B25:K34;B3:K12)** en bevestig met **Ctrl-Shift-Enter**.

Om de verschillen met de eenheidsmatrix te kunnen berekenen, plaats je die in de cellen M3:V12. In de cellen X14:AG23 bereken je het verschil tussen de matrices in M3:V12 en M14:V23. In de cellen X25:AG34 bereken je het verschil tussen de matrices in M3:V12 en M25:V34. De kwadratische som van de fouten bereken je met **=sumsq(X14:AG23)** en **=sumsq(X25:AG34)**. Je ziet dat de kwadratische som van de fouten in Matlab in de orde van 10^{-8} ligt, terwijl hij bij Excel maar liefst in de orde van 10^{-5} ligt.

6 Geavanceerd VBA-gebruik

In dit hoofdstuk gaan we VBA op een geavanceerde manier gebruiken, namelijk om een ***cache*** te programmeren.

6.1 Cache

Een cache wordt gebruikt om lees- en schrijfoperaties sneller te laten verlopen. Een computer heeft langzaam geheugen (een ***harde schijf***) en snel geheugen (***werkgeheugen***). Een harde schijf is langzamer, maar ook veel groter dan het werkgeheugen. De gegevens die de processor nodig heeft worden eerst van de harde schijf gelezen en in het werkgeheugen geplaatst, zodat de processor er snel bij kan. Dat is al een vorm van caching.

Iets soortgelijks doet zich voor in VBA. We hebben in Hoofdstuk 2 ranges en arrays besproken. Wat we toen niet duidelijk hebben gemaakt, is dat lezen en schrijven van en naar arrays veel sneller gaat dan lezen en schrijven van en naar ranges. Bij het declareren van arrays wordt namelijk een stukje werkgeheugen gereserveerd, wat ervoor zorgt dat de processor precies weet waar elk element van dat array staat. Dit is voor ranges niet mogelijk. Die moeten (vaak op de harde schijf) opgezocht worden, hetgeen langer duurt. Door nu bepaalde ranges in arrays te bewaren, heb je ook een vorm van caching.

Bij caching heb je dus altijd langzaam geheugen en snel geheugen, waarbij data uit het langzame geheugen in het snelle geheugen bewaard worden om ze sneller te kunnen lezen en schrijven. Er zijn twee vormen van caching: ***write-through*** en ***write-back***. Zoals de namen al aangeven, verschillen deze vormen in de schrijfoperaties; de leesoperaties zijn gelijk.

Bij elke leesoperatie wordt eerst gekeken of de gevraagde data in het snelle geheugen (de cache) staan. Zo ja, dan worden de data uit de cache gelezen. Zo nee, dan worden de data uit het langzame geheugen gelezen en in de cache gezet.

Bij write-through worden data naar zowel het snelle als het langzame geheugen geschreven. Bij write-back worden data alleen naar de cache geschreven. Later wordt dan de gehele cache in het langzame geheugen geschreven. Vaak gebeurt dat vlak voor een cache ***flush***, een operatie waarbij de hele cache geleegd wordt.

6.2 Het cachen van ranges in arrays

We gaan nu een cache in VBA schrijven. Deze cache bewaart enkele ranges in snel geheugen. We maken daarom een class module, zodat we voor iedere range die we willen cachen een instantie van deze class kunnen maken.

```
Private present() As Boolean
Private cache() As Variant
Private SourceRange As Range
Private numrows As Integer
Private numcols As Integer

Public Property Set source(r As Range)
    Set SourceRange = r
    numcols = r.Columns.Count
    numrows = r.Rows.Count
    ReDim present(1 To numrows, 1 To numcols)
    ReDim cache(1 To numrows, 1 To numcols)
End Property
Public Property Get source() As Range
    Set source = SourceRange
End Property
Public Property Get data(r As Integer, c As Integer) As Variant
    If present(r, c) Then
        data = cache(r, c)
    Else
        data = SourceRange.Cells(r, c).Value
        cache(r, c) = data
        present(r, c) = True
    End If
End Property
Public Property Let data(r As Integer, c As Integer, v As Variant)
    If present(r, c) Then
        If cache(r, c) = v Then
            Exit Property
        End If
    Else
        present(r, c) = True
    End If
    cache(r, c) = v
    SourceRange.Cells(r, c).Value = v
End Property
Public Sub flush()
    ReDim present(1 To numrows, 1 To numcols)
```

```
End Sub
```

In het cache-array gaan we de data opslaan. Deze data kunnen alle vormen van celinhoud zijn, dus daar maken we Variants. Dit array is tweedimensionaal, zodat het eigenlijk een stuk worksheet representeert. We maken een boolean-array present van gelijke grootte om aan te geven of de cel wel (true) of niet (false) in de cache staat.

In de Property Set source wordt een range meegegeven, die in SourceRange wordt gezet. De dimensies numcols en numrows worden goed gezet en er worden arrays present en cache van goede grootte aangemaakt. Door ReDim worden alle waarden in het boolean-array present automatisch false. De Property Get source retourneert de SourceRange.

De Property Get data is een leesoperatie. Zoals in de vorige paragraaf beschreven, wordt eerst gekeken of de cel met coördinaten r en c in de cache staat. Zo ja, dan wordt de waarde uit de cache geretourneerd. Zo nee, dan wordt de waarde uit de SourceRange gelezen, in de cache gezet en geretourneerd.

De Property Let data is een schrijfoperatie. Eerst wordt gekeken of de data in de cache staat en of de te schrijven waarde toevallig dezelfde als in de cache. In dat geval hoeven we niets te doen. Anders wordt present(r, c) true. Vervolgens wordt de waarde in de cache en meteen in de SourceRange geschreven, het is tenslotte een write-through cache.

De laatste procedure is een macro flush, die de cache leegmaakt. Dit gebeurt door met ReDim alle waarden in het present-array op false te zetten.

6.3 Benchmark

Allemaal leuk en aardig, zo'n cache, maar werkt het eigenlijk wel en wat levert zoiets nou op? Daartoe schrijven we een module ***benchmark***, een test die een tijdmeting doet.

```
Dim TestRange As Range
Dim numrows As Integer
Dim numcols As Integer
Dim cache As New RangeCache
Sub test()
    Set TestRange = Range("A1:J10")
    numrows = TestRange.Rows.Count
    numcols = TestRange.Columns.Count
    Set cache.source = TestRange

    Debug.Print "cache benchmarks"
    readtest
    inctest
End Sub
...
```

We hebben natuurlijk een range nodig om te cachen, die zetten we in TestRange. Verder maken we nog de integers numrows en numcols aan en een nieuw object van de class RangeCache. De macro test selecteert een range en start twee andere macro's: readtest en inctest.

```
...
Sub ResetRange()
    For Each cel In TestRange
        cel.Value = 0
    Next cel
    cache.flush
End Sub
Sub readtest()
    Dim r As Integer: Dim c As Integer
    ResetRange
    Start = Timer()
    For h = 1 To 1000
        For r = 1 To numrows
            For c = 1 To numcols
                tmp = TestRange.Cells(r, c).Value
            Next c
        Next r
    Next h
    Debug.Print "Read 1000 times direct:"; Timer() - Start
    ResetRange
    Start = Timer()
    For h = 1 To 1000
        For r = 1 To numrows
            For c = 1 To numcols
                tmp = cache.data(r, c)
            Next c
        Next r
    Next h
    Debug.Print "Read 1000 times cached:"; Timer() - Start
End Sub
...
```

De macro `ResetRange`, die alle celwaarden in de `TestRange` op nul zet en de cache flusht, is handig, want die hebben we een aantal keer nodig. De macro `readtest` leest eerst duizend keer alle waarden uit de range en vervolgens duizend keer alle waarden uit de cache. De code is duidelijk.

```
...
Sub inctest()
    ResetRange
    Start = Timer()
    For i = 1 To 100
        IncDirect 1
    Next i
    Debug.Print "inc 100 x 1, direct:"; Timer() - Start
    ResetRange
    Start = Timer()
    IncDirect 100
    For i = 2 To 100
        IncDirect 0
    Next i
    Debug.Print "inc 1 x 100, 99 x 0, direct:"; Timer() – Start

    ResetRange
    Start = Timer()
    For i = 1 To 100
        IncCache 1
    Next i
    Debug.Print "inc 100 x 1, cached:"; Timer() - Start
    ResetRange
    Start = Timer()
    IncCache 100
    For i = 2 To 100
        IncCache 0
    Next i
    Debug.Print "inc 1 x 100, 99 x 0, cached:"; Timer() - Start
End Sub

Sub IncDirect(inc As Integer)
    Dim r As Integer: Dim c As Integer
    For r = 1 To numrows
        For c = 1 To numcols
            With TestRange.Cells(r, c)
                .Value = .Value + inc
            End With
        Next c
    Next r
End Sub
Sub IncCache(inc As Integer)
    Dim r As Integer: Dim c As Integer
    For r = 1 To numrows
        For c = 1 To numcols
            cache.data(r, c) = cache.data(r, c) + inc
        Next c
    Next r
End Sub
```

De macro `inctest` verhoogt honderd keer de waarde van alle cellen van `TestRange` met 1. Vervolgens verhoogt hij 99 keer de waarde van alle cellen van `TestRange` met 0. Daarna doet hij precies hetzelfde, maar dan via de cache. Als we de test runnen, komen we tot de volgende resultaten:

Tabel 6.1: Resultaten van de cache benchmark

	Pentium II-350	Pentium III-1000
Read 1000 times direct	7,0195	2,3906
Read 1000 times cached	0,2617	0,1328
inc 100 x 1, direct	9,1133	4,1992
inc 1 x 100, 99 x 0, direct	9,0742	4,2852
inc 100 x 1, cached	8,9844	4,3359
inc 1 x 100, 99 x 0, cached	0,1484	0,0820

Deze tabel laat duidelijk zien waar onze cache wel en niet goed in is. De leestest veel sneller. Dat komt doordat via de cache slechts eenmaal uit de range gelezen hoeft te worden, tegen duizendmaal direct. Bij de `inctest` zijn de verschillen tussen direct en via de cache miniem. Dat komt doordat het een write-through cache is, die alles ook meteen in de range zet. Bij het met nul verhogen zien we wel een groot verschil, dat is te danken aan ons slimmigheidje in de code van de `Property Let data`: als de te schrijven waarde dezelfde is, doen we niets.

Appendix A: Literatuur

[1] Studiegids Wiskunde en Informatica 2002/2003
[2] http://www.phys.tue.nl/TULO/info/guldensnede/
[3] http://java.sun.com/products/jfc/tsc/articles/component_gallery/index.html
[4] Eric Wells & Steve Harshbarger: Microsoft Excel 97 Developer's Handbook, Micro Modeling Associates

Appendix B: Oplossingen

B.1 Fibonacci

Vraag: de bovenstaande functie voor het uitrekenen van Fibonacci-getallen is niet bepaald robuust en efficiënt. Geef voor beide een argument.

Antwoord: De methode is niet robuust, omdat het argument niet wordt gecheckt. Wat zou er gebeuren met een negatief argument? De methode is niet efficiënt, omdat hetzelfde getal tweemaal uitgerekend wordt. Daarom is het beter om een array met Fibonaccigetallen te maken.

Vraag: Leg uit, waarom deze methode een stuk efficiënter is.

Antwoord: Zie antwoord op de eerste vraag.

B.2 Driehoek van Pascal

De volgende VBA-code genereert een driehoek van Pascal:

```
Sub Pascal()
    Const MaxAantalRegels As Integer = 128
    Dim AantalRegels As Integer
    Do Until (False)
        invoer = InputBox("Aantal regels", "Driehoek van Pascal", 10)
        If invoer = "" Then
            Exit Sub
        Else
            AantalRegels = CInt(invoer)
        End If

        If (AantalRegels < 1 Or AantalRegels > MaxAantalRegels) Then
            MsgBox "Vul minimaal 1 en maximaal " & MaxAantalRegels & " in!"
        Else
            Exit Do
        End If
    Loop

    With Cells
        .ClearContents
        .ColumnWidth = AantalRegels * 3 / 20
        .NumberFormat = "0"
    End With

    With Range(Cells(1, AantalRegels), Cells(1, AantalRegels + 1))
        .Merge
        .HorizontalAlignment = xlCenter
        .Value = 1
        .Select
    End With
    For regel = 2 To AantalRegels
        For j = 0 To regel - 1
            With Range(Cells(regel, AantalRegels - regel + 1 + 2 * j), Cells(regel, AantalRegels -
regel + 2 + 2 * j))
                .Merge
                .HorizontalAlignment = xlCenter
                .FormulaR1C1 = "=R[-1]C[-1] + R[-1]C[1]"
            End With
        Next j
    Next regel
End Sub
```