Object Georiënteerd Programmeren Basis

Wouter Brinksma, NHL Stenden Hogeschool

Augustus 2018

Inhoudsopgave

Voorwoord

Software Engineering is een van de meest veelzijdige beroepen ter wereld. Als Software Engineer ben je verantwoordelijk voor projecten uiteenlopend van een kleine website voor een lokale winkel, tot een groot bevoorradingssysteem voor een internationaal bedrijf. Om al deze verschillende dingen te kunnen maken, gebruiken Software Engineers universele technieken die kunnen worden ingezet voor het ontwerpen, bouwen en onderhouden van software. Deze reader zal je de basis leren van die technieken.

Een Persoonlijke Anekdote

Ik zal mij eerst even voorstellen. Mijn naam is Wouter Brinksma, docent aan de School of ICT, een afdeling van NHL Stenden University of Applied Sciences. Wanneer je kijkt naar Software Engineering, ligt mijn interesse erg bij de elementaire ideeën achter dit proces. De manier waarop we onze software bouwen stoelt op ontwerpprincipes die zijn gevormd door jaren aan ervaring en het harde werk van vele mensen. Die kennis is voor mij een van de meest belangrijke onderdelen wanneer ik studenten lesgeef over het maken van software. Het is ook precies de reden waarom ik deze tekst ben gaan schrijven.

Tijdens mijn eigen studietijd ben ik achter het verschil gekomen tussen iets weten en iets begrijpen. Mijn keuze voor dit prachtige vakgebied was namelijk niet altijd zo zeker als nu. Een tijd lang dacht ik dat ik advocaat of iets dergelijks zou moeten worden. (Uiteindelijk praat ik nu ook voor menigtes, al is het wel met een ietwat andere lading) Een goede vriend heeft mij echter het Software Engineering vakgebied laten zien, en toen wist ik dat ik hierin verder wilde. "Maar dan zullen mijn mede-studenten wel veel meer kennis hebben op dit gebied omdat ze al langer weten dat ze dit leuk vinden", zo dacht ik. Om die reden heb ik in de vakantie voor het begin van mijn studie veel kennis opgezocht op het internet en video's gekeken op YouTube. Echter, niets bleek minder waar. Mijn mede-studenten begonnen ook bijna allemaal net met programmeren, en ik had nu zelfs een kleine voorsprong.

Het enige dat bij mij miste, wat bij iedereen ontbreekt die net begint, is het grote plaatje kunnen zien en hoe alles met elkaar samenwerkt. Ik kon wel programmeren, maar hoe je nou met al die stukjes code een groot programma goed in elkaar zet was voor mij een raadsel. Omdat programmeren ook over talen gaat past hier een mooie anekdote bij.

Richard Feynman, een groot natuurkundige en geweldige leraar, heeft hier ooit iets over gezegd. Als voorbeeld nam hij een vogel. Wanneer je de naam voor een vogel in het Engels, Frans en Nederlands kent, kun je zeggen dat je veel over de vogel weet. Maar, zoals Feynman opmerkte, kun je nog zoveel namen voor iets kennen, dat betekent niet dat je er iets over weet. Je weet alleen hoe je iets moet noemen. Dit noemde hij "weten versus begrijpen". Het is precies zo met code. Wanneer je weet hoe je een stukje code schrijft dat een `integer` aanmaakt, betekent dat niet dat je weet wat een `integer` is, laat staan wat een variabele aanmaken is. Ik heb dit boek geschreven omdat ik dat verschil zelf ervaren heb. In deze reader probeer ik je niet alleen de namen van iets te leren, maar ook het begrip erachter te laten zien. Wanneer je dit boek uit hebt, zul je niet alleen de basisprincipes achter Software Engineering begrijpen, maar ze ook kunnen toepassen.

Opbouw van het Boek

Dit boek is opgebouwd uit een aantal hoofdstukken. Het eerste hoofdstuk behandeld algemene theorie en concepten die steeds terug zullen komen terwijl je het boek leest. De rest van de hoofdstukken verduidelijken deze concepten door per hoofdstuk een praktisch probleem te behandelen, en op te lossen met object georiënteerde technieken. Doordat elk probleem weer anders is dan de vorige kom je automatisch nieuwe aspecten tegen van object georiënteerd programmeren. Deze zullen behandeld worden, en na alle hoofdstukken te hebben doorgenomen heb je alle relevante basiskennis een keer gezien en er mee gewerkt. Wanneer je dit boek zelfstandig volgt kun je per week een hoofdstuk doornemen en de oefenvragen maken. De code die in het boek voorkomt kun je op GitHub vinden. Volg deze ULR om de code te verkrijgen: [NOG NIET BESCHIKBAAR]

Het hebben van een computer is in theorie geen vereiste bij het gebruik van dit boek, maar wordt wel aangeraden.Het kan handig zijn om de voorbeelden die in deze reader worden gegeven uit te weken. De voorbeelden zijn zo gemaakt dat ze platform-onafhankelijk werken. Dit betekent dat je ze op elk systeem kunt uitvoeren, mits je de juiste configuratie hebt. Er is geen voorkennis vereist voor het gebruik van dit boek, maar enige wiskundige kennis wordt wel gewaardeerd.

Methoden

We gaan de programmeertaal *C#* gebruiken in deze reader. C# (uitgesproken als C-sharp) is in 2000 geïntroduceerd door Microsoft, 5 jaar nadat de andere grote object georiënteerde programmeertaal Java het licht zag. C# is begonnen als de taal "Cool", een korte versie van "C-like Object Oriented Language". Deze naam is echter veranderd in C#, omwille van trademarks. Er bestaat tegenwoordig nog steeds een taal genaamd "Cool", maar dit is een taal die wordt gebruikt om te leren hoe je een compiler maakt. C# is een gestandaardiseerde taal, waarvoor het ECMA verantwoordelijk is. De taal heeft meerdere implementaties, en Microsoft heeft zelfs een cross-platform versie uitgebracht genaamd *.NET Core*.

Om te garanderen dat je deze leergang met elk besturingssysteem kunt volgen, gaan we gebruikmaken van .NET Core. Op het moment van schrijven heeft .NET Core nog geen ondersteuning voor een cross-platform GUI (Graphical User Interface), maar dit is in ons geval geen probleem. .NET Core werkt met de *terminal*, een programma waarin we tekstuele commando's kunnen geven aan de computer. Vroeger was deze terminal de enige vorm van interactie met de computer. Besturingssystemen met een bureaublad waren in die tijd een grote vernieuwing, zeker toen programma's GUI's begonnen te krijgen. In dit boek focussen we ons echter op de code van onze programma's, en dus zal een tekstuele interface volstaan.

Ik wens je veel succes met dit boek, en hoop dat het je helpt bij het behalen van je studie.

Hoofdstuk 1

Programmeertalen

Na het ontstaan van programmeertalen zoals C++ en Java is object georiënteerd programmeren (OOP) de dominante vorm van programmeren geworden tijdens de jaren 90 van de vorige eeuw. Dit is echter niet altijd zo geweest. Voordat OOP het licht zag, waren manieren van programmeren zoals procedureel programmeren de norm. In dit hoofdstuk wordt OOP kort uitgelegd, en ook een beeld geven van programmeren als geheel.

1.1 Algemeen

Kort gezegd is een programmeertaal een verzameling symbolen en opdrachten waarmee de programmeur commando's kan geven aan de computer. Er zijn verschillende niveaus van programmeertalen ontstaan door de jaren heen, welke vaak worden gecategoriseerd als hogere of lagere programmeertalen. Een lagere programmeertaal wil niet zeggen dat deze makkelijker of juist minder goed is, maar het betekent dat deze taal een sterkere relatie heeft met de hardware van de computer dan hogere talen. De laagste programmeertaal die bestaat is dan ook machinecode. Dit is een set aan instructies welke direct door de CPU van de machine (computer) worden uitgevoerd. Numerieke machinecode is geschreven in bits, en is daarom erg moeilijk met de hand te schrijven en ook foutgevoelig. Een niveau hoger zit de assembly language. Deze taal gebruikt simpele, door mensen te lezen commando's in plaats van bits. Assembly language is echter nog steeds erg verbonden met de machinecode van een bepaalde CPU (of beter gezegd, CPU-*architectuur*), wat het een lastige vorm van programmeren maakt. Een voorbeeld van een stukje machinecode staat hieronder beschreven.[4]

Wanneer we op het machine-niveau opereren, hebben we het niet langer over variabelen die we een waarde toekennen. Op het machine-niveau

spreken we van het toekennen van waarden aan registers. Dit zijn kleine hardwarematige opslagplaatsen voor data, bijvoorbeeld in de CPU. Wanneer een CPU een berekening moet doen met twee waarden, bijvoorbeeld 1 + 2, dan worden die waarden eerst in de juiste registers van de CPU geplaatst zodat deze daarmee kan werken. Een verplaatsing van een waarde naar een register ziet er in machinecode als volgt uit.

```
10110000 01100001
```

In dit stukje voorbeeldcode zitten drie dingen verstopt. Het eerste gedeelte, 10110000, is een instructie. Deze instructie bestaat uit twee delen. Het eerste deel, 10110, is de instructie zelf. Deze vertelt de CPU dat een waarde naar een register moet worden geschreven. De laatste drie bits, 000, identificeren het juiste register, welke in dit geval AL heet. Het laatste gedeelte van de machinecode, 01100001, is de daadwerkelijke waarde die in het register wordt opgeslagen. Het veranderen van dit stukje machinecode naar hexadecimale notatie maakt het iets inzichtelijker.

```
B0 61
```

Hier staat B0 voor de volledige instructie om de waarde 61 (97 in decimaal) in register AL te stoppen. Een programmeur zou B0 kunnen herkennen en weten wat het is. Om het nog leesbaarder te maken kunnen we het veranderen in assembly language.

```
MOV AL, 61h
```

Deze code gebruikt het keyword MOV. Dit staat voor de instructie om een waarde te verplaatsen naar een register. Het register wordt nu simpelweg aangegeven met AL, de daadwerkelijke naam. Daarachter noteren we de waarde, 61h, wat staat voor hexadecimaal 61 (wat decimaal 97 is). Deze talen zijn een tijd lang veelgebruikt geweest, en staan in sterk contrast met de talen die we vandaag de dag gebruiken. Echter, deze talen zijn nog steeds aanwezig. Enerzijds kunnen we er nog steeds handmatig in programmeren, en anderzijds veranderen programma's genaamd *compilers* onze hogere code in deze lagere vorm van code om het uitvoerbaar te maken door computers. Dit proces heet dan ook *compileren.*[4]

Na deze twee talen gezien te hebben, wordt het moeilijker om een opeenvolgende lijst van talen te produceren. De taal die de brug vormt tussen lagere en hogere talen is de taal genaamd C. Deze taal bestaat al sinds 1972,

en wordt vandaag de dag nog steeds volop gebruikt. C wordt gebruikt bij het maken van besturingssystemen en embedded systems, maar vormt ook de basis van hogere talen zoals Python, Perl en PHP. Dit komt doordat de programma's die deze code uitvoeren zijn geschreven in C. Het gebruik van C kan worden verklaard doordat programma's in C erg snel zijn, en ook op bijna alle hardware uitvoerbaar zijn. Ook kunnen programmeurs, ondanks dat dit een hogere taal is, nog steeds gemakkelijk bij hardware adressen komen en staat C dichterbij assembly language dan andere talen zoals Python.

1.2 Programmeerparadigma's

Programmeertalen kunnen worden geclassificeerd met programmeerparadigma's. Een programmeerparadigma is een bepaald denkpatroon of concept waar een programmeertaal aan voldoet. Er zijn verschillende programmeerparadigma's, waarvan we een aantal hieronder gaan behandelen. De taal die we hierboven besproken hebben, C, valt onder het *imperatieve* programmeerparadigma.

1.2.1 Imperatief Programmeren

Dit is een van de meest belangrijke stijlen van programmeren. Imperatief programmeren kan het best worden omschreven als een vorm van programmeren waarbij commando's worden gebruikt welke de staat van het programma veranderen. We kunnen het beste een voorbeeld gebruiken om uit te leggen wat dit betekent. Programma's hebben variabelen die informatie opslaan over dat programma. Deze variabelen kunnen we *declareren*, wat betekent dat we ruimte vrijmaken in het geheugen voor die informatie. We geven een variabele een naam zodat we de informatie via die naam weer terug kunnen vinden. We noemen al die informatie (de waarden van alle variabelen) als geheel de staat van het programma. Wanneer code van het programma veranderingen aanbrengt in de waarde van die variabelen, noemen we dit het veranderen van de staat van het programma.

Door deze definitie vallen alle talen die eerder hierboven genoemd zijn onder het imperatieve paradigma, zelfs machinecode. Machinecode is zelfs een van de beste manieren om imperatief programmeren uit te leggen, omdat machinecode alleen maar data verplaatst en daar operaties op doet. Het verandert dus constant de staat van het programma.

1.2.2 Procedureel Programmeren

Een stijl van programmeren die nauw verbonden is met imperatief programmeren is *procedureel* programmeren. Procedureel programmeren is een vorm van imperatief programmeren, waarbij het programma is opgebouwd uit een of meer procedures (ook wel subroutines of functies genoemd). Dit betekent dat machinecode geen procedurele taal is, omdat het geen functies kan definiëren. C is echter wel een procedurele taal, omdat de mogelijkheid tot het definiëren van functies hierbij wel bestaat. Een stukje C code is hieronder bijgevoegd om dit te illustreren.

```
#include <stdio.h>

int main()
{
  printf("Hello, World!");
  return 0;
}
```

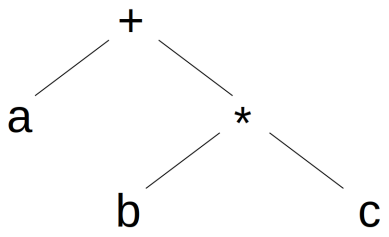
De functie die hier getoond wordt heet *main*. De functie print de tekst *Hello, World!* op het scherm. Je kunt op deze manier meerdere functies schrijven, waarmee je gebruikmaakt van een van de belangrijkste voordelen van procedurele talen, namelijk: *herbruikbaarheid*. Een functie schrijf je als programmeur één keer, en kan oneindig vaak hergebruikt worden. Dit scheelt je heel wat tijd, en is ook nog eens veel minder foutgevoelig dan alles met de hand kopiëren.

Een imperatieve/procedurele taal werkt door commando's sequentieel uit te voeren en ze te groeperen in verschillende functies. Als programmeur wil je ook controle houden over de volgorde waarop deze commando's worden uitgevoerd. Deze vorm van controle is op twee manieren aanwezig: *expressie-level* en *statement-level* uitvoeringscontrole.

Expressie-level Control

Expressie-level controle wordt in praktijk gebracht zodra een expressie in je code wordt geëvalueerd. Dit gebeurt bijvoorbeeld wanneer een wiskunde expressie wordt uitgevoerd zoals a + b * c. Bepaalde *operatoren* (+, -, *, /, ...) hebben voorrang op anderen. Door die voorrangsregels moet eerst b * c worden uitgevoerd, waarna a bij het resultaat van dat product wordt opgeteld. Dit betekent dat vermenigvuldiging voorrang heeft ten opzichte van optellen. Computers modelleren deze vorm van voorrang met een *expressieboom*. Een

expressieboom geeft de volgorde weer waarop een expressie moet worden uitgevoerd. De expressieboom van de hierboven genoemde voorbeeldexpressie is gegeven.



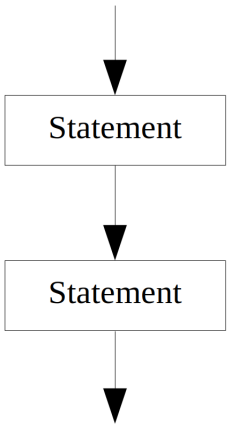
Figuur 1.1: Expressieboom van a + b * c

Een computer lost deze boom op door onderaan te beginnen, en vanaf daar omhoog te werken. Eerst evalueert het b * c. Het resultaat van die vermenigvuldiging komt nu op de plaats te staan van de *. Ditzelfde proces geschiedt nu ook voor de optelling, en het resultaat van de expressie blijft over. Dit voorbeeld is een simpele, en illustreert het concept. Wanneer we echter een volledige compiler bouwen, komen we in aanraking met grotere problemen zoals het feit dat het minus symbool (-) verschillende voorrangsregels kent onder verschillende omstandigheden. Normaal gesproken wordt het gebruikt voor het verminderen van een waarde met een andere waarde zoals in a - b. Het komt echter ook voor als een aanduiding dat een getal negatief is, zoals bij -a. Dit staat in feite voor -1 * a, wat een andere voorrang kent dan a - b.

Statement-level Control

De tweede vorm van uitvoeringscontrole is statement-level controle, wat veel meer omvat dan zijn voorganger van hierboven. Normaal gesproken volgt een programmeertaal de voor ons *natuurlijke* volgorde van uitvoeren: van links naar rechts en van boven naar beneden. Let wel, deze vorm verschilt echter per cultuur. In andere culturen dan de westerse leest men van rechts naar links, wat betekent dat daar de natuurlijke volgorde anders is. Programmeertalen zijn gemaakt in de westerse culturen, wat voor ons betekent dat ze allemaal die vorm van uitvoeren volgen. Deze natuurlijke volgorde is echter niet altijd afdoende. Soms willen we kunnen kiezen welk stukje code we onder bepaalde omstandigheden wel uitvoeren en welke niet, of willen we een of meer operaties een aantal keer herhalen. Deze concepten kunnen we het best illustreren met een aantal *flowcharts*. Flowcharts zijn diagrammen

waarmee we algoritmen, workflows en processen mee kunnen modelleren. De natuurlijke volgorde van het uitvoeren van een programma ziet er dan als volgt uit.

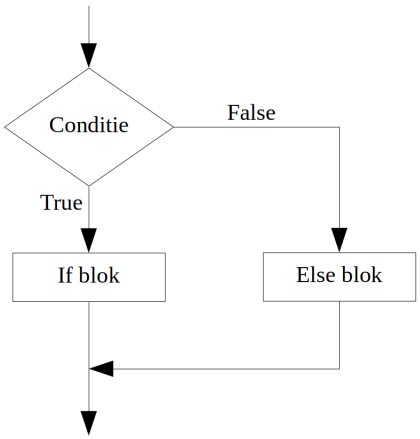


Figuur 1.2: Flowchart van natuurlijke uitvoer

De pijlen tonen de volgorde van het uitvoeren van het programma. De rechthoeken staan normaal gesproken voor processen, welke in deze context staan voor regels code die worden uitgevoerd. Deze flowchart laat de natuurlijke orde van uitvoeren zien, maar soms willen we meer controle. Dit kunnen we doen door zogenaamde control-statements te gebruiken.

Control-statements zijn onderdelen van een programmeertaal waarmee we kunnen aangeven wat een programma, gezien bepaalde condities, moet doen. Er zijn drie vormen van control-statements, namelijk: *if-statement*, *loops* (for, while en do-while) en het switch-statement. If- en switch-statements kunnen worden gegroepeerd als statements die een beslissing maken. Ze laten de programmeur een of meer condities specificeren welke moeten worden geëvalueerd. Dit betekent dat ze de programmeur iets laten testen dat waar of niet waar kan zijn. Aan de hand van dat resultaat zal het programma dan iets uitvoeren of juist niet. Beslissingen kunnen ook in flowcharts gemodelleerd worden. Ze worden aangegeven met een diamant- of ruitvorm. Hieronder staat de flowchart van het if-statement.

Deze statements hebben allemaal één kernonderdeel, namelijk het conditie element (welke wordt aangegeven met de ruitvorm). Dit specificeert welke conditie geëvalueerd moet worden, en volgt aan de hand van het resultaat de pijl gelabeld met *True* of met *False*. True en False zijn zogenaamde *booleaanse waarden*. True betekent dat iets waar is, en wordt aangegeven met een 1. False betekent niet waar, en wordt aangegeven met een 0. Een if-statement test of de resulterende waarde van de conditie een 1 of een 0 is. (Simpel



Figuur 1.3: Flowchart van if-statement

gezegd, de precieze technische invulling van het if-statement verschilt per taal en vaak kan dit statement ook meer dingen testen, zoals het aanwezig zijn van een object in een variabele.) Hieronder staat een kort voorbeeld van een if-statement.

```
int a = 3;

if (a > 3)
{
  Console.WriteLine("a is greater than 3");
}
```

EXTRA
Er zijn verschillende manieren om een conditie te maken. We vergelijken vaak waarden met elkaar, en dat doe je met de volgende operatoren:
`== (is gelijk aan)`
`!= (is niet gelijk aan)`
`> (groter dan)`
`< (kleiner dan)`
`>= (groter of gelijk aan)`
`<= (kleiner of gelijk aan)`

Het switch-statement is een variant van dit concept, waarbij de conditie is opgesplitst in twee delen. Een switch statement neemt een beginwaarde,

en test vervolgens een of meerdere *cases*. Een case is een mogelijke waarde die de beginwaarde kan hebben. Wanneer je bijvoorbeeld een tekstwaarde controleert, en je test of deze één van een bepaald aantal vaststaande waarden bevat, dan is het raadzaam om een switch-statement te gebruiken. Dit komt doordat een switch-statement bij dit soort vraagstukken het meest efficiënt is. Hieronder vind je een voorbeeld van een switch-statement.

```
string country = "Netherlands";
string language = "";

switch(country) {
  case "Netherlands":
  language = "Dutch";
  break;

  case "Germany": case "Austria": case "Switzerland":
  language = "German";
  break;

  case "England": case "USA":
  language = "English";
  break;

  default:
  language = "No language found";
  break;
}

Console.WriteLine("Language = " + language);
```

Net als if-statements voeren loops ook code uit, en hebben ook een conditie. Echter, nadat de code in het statement is uitgevoerd gaat het programma terug naar de conditie en herhaalt deze opnieuw totdat de uitkomst van de conditie false is, en de loop stopt. Oplettende lezers zullen opmerken dat er iets mist uit deze omschrijving, en dat klopt. In feite staat hierboven een *while-loop* beschreven. Dit is de meest simpele loop die we kennen (uitgezonderd het gebruik van Goto), en ziet er als volgt uit.

```
int i = 0;
```

```
while (i < 10) {
  Console.WriteLine("Looped " + (i + 1) + "times!");
  i++;
}
```

Wat in de uitleg miste is het feit dat er iets in het programma moet veranderen om de conditie uiteindelijk op false te laten uitkomen. Wanneer dit niet gebeurt zal de conditie altijd hetzelfde teruggeven, en komt het programma in een *infinite loop*. Dit is wanneer het programma onafgebroken dezelfde handeling herhaald, en zeer waarschijnlijk constant 100% van de CPU in beslag neemt. Bij een while-loop ligt de verantwoordelijkheid van het voorkomen van een infinite loop dus bij de programmeur zelf. Dit betekent dat je zelf moet nadenken over hoe je het verloop van de loop vormgeeft, bijvoorbeeld met een variabele die het aantal keren dat de loop is uitgevoerd bijhoudt.

Om dit makkelijker te maken kunnen we een *for-loop* gebruiken. Een for-loop is in feite hetzelfde als een while-loop, maar heeft in zijn definitie niet alleen een conditie, maar ook een variabele om het aantal keren te tellen dat de loop is uitgevoerd, en een stukje code dat wordt uitgevoerd wanneer de loop aan het einde is en weer van vooraf aan begint. Hieronder staat een voorbeeld.

```
for (int i = 0; i < 10; i++) {
  Console.WriteLine("Looped " + (i + 1) + "times!");
}
```

In feite doen beide loops hetzelfde, maar de manier van opschrijven is anders en worden for-loops op andere momenten gebruik dan while-loops. Je gebruikt doorgaans een for-loop wanneer bekend is hoe vaak de loop moet doorgaan. Wanneer je bijvoorbeeld door alle elementen van een lijst heengaat, weet je hoe lang de lijst is. Dat is bekend wanneer de loop start. Een while-loop gebruik je vaak wanneer niet bekend is hoe vaak de loop uitgevoerd moet worden. Dit gebruik je bijvoorbeeld wanneer je informatie vanaf een netwerk leest, en daardoor niet duidelijk is wanneer deze binnenkomt en hoeveel informatie het is.

Als laatste behandelen we de *do-while-loop*. Deze loop is wezenlijk anders dan een while- of for-loop omdat het principe erachter anders is. Normaal gesproken wordt altijd eerst de conditie gecheckt voordat de code van de loop wordt uitgevoerd, ook de eerste keer. Bij een do-while-loop is dit anders. De eerste keer dat de code bij deze loop komt en gaat uitvoeren wordt altijd

de code binnen de loop uitgevoerd. Daarna wordt de conditie gecheckt en gaat de loop verder of niet. Een do-while-loop garandeert dus dat de code binnen in de loop minimaal één keer wordt uitgevoerd. Hieronder staat een voorbeeld.

```
int i = 0;

do {
  Console.WriteLine("This message should only appear
     once");
  i++;
} while (i < 1)
```

Door gebruik te maken van deze systemen heeft de programmeur controle over een programma geschreven in een imperatieve programmeertaal. Er zijn naast imperatieve talen echter nog meer verschillende programmeertalen die andere paradigma's volgen. Twee andere soorten talen die je mogelijk tegen kunt komen zijn *functionele talen* en *declaratieve talen*. Beide zullen hieronder kort behandeld worden.

1.2.3 Functioneel Programmeren

Functionele programmeertalen zijn het tegenovergestelde van imperatieve programmeertalen omdat ze juist het veranderen van de staat van het programma proberen te vermijden. In plaats daarvan zijn ze ontworpen om alleen te werken met functies die waarden teruggeven na uitvoer. Dit concept heeft zijn oorsprong in lambda calculus, een wiskundig systeem dat is ontwikkeld in de jaren 30 van de vorige eeuw. Een aantal bekende functionele programmeertalen zijn Haskell, Common Lisp en Clojure. Een klein Haskell voorbeeld is hieronder getoond. Het berekent de eerste 10 Fibonacci getallen.

```
fibonacci_aux = \n first second->
  if n == 0 then "" else
  show first ++ "\n" ++ fibanocci_aux (n - 1) second
     (first + second)
fibonacci = \n-> fibonacci_aux n 0 1
main = putStr (fibonacci 10)
```

In dit voorbeeld kunnen we een veelgebruikte techniek bij functionele

talen zien. De `main` functie, het beginpunt van het programma, roept de `fibonacci` functie aan met het getal 10. Dit staat voor het aantal opeenvolgende Fibonacci getallen dat de functie terug moet geven. Dan gebeurt er iets bijzonders. De `fibonacci` functie roept de functie `fibonacci_aux` aan. Deze functie neemt niet één parameter zoals zijn voorganger, maar neemt er wel drie. Dit wordt gedaan omdat je geen variabelen kunt declareren in een functionele programmeertaal zoals je dat kunt in een imperatieve taal. Vaak zijn variabelen in een functionele taal *read only*, of moet je specifiek bepaalde opties veranderen om ze te kunnen gebruiken. De parameters van functies vervangen in feite de noodzaak voor gedeclareerde variabelen (de staat van een programma op de imperatieve manier). De `fibonacci_aux` functie heeft de volgende parameters: *n* voor het totaal aantal keren dat de functie nog uitgevoerd moet worden, *first* voor het eerste getal en *second* voor het tweede getal die gebruikt worden om het volgende Fibonacci getal te berekenen. Fibonacci getallen worden berekend door de twee vorige getallen in de reeks bij elkaar op te tellen. De reeks begint door 0 en 1 bij elkaar op te tellen, wat resulteert in 1. Daarna tel je 1 en 1 bij elkaar op wat resulteert in 2. Dan 2 en 1, wat resulteert in 3. Dan 3 en 2, wat resulteert in 5 en ga zo maar door. Uiteindelijk komt daar de volgende reeks uit: *0, 1, 2, 3, 5, 8, 13, 21, 32, etc.*. De aanroep van de `fibonacci_aux` functie zou er de eerste keer als volgt uitzien:

```
fibonacci_aux 10 0 1
```

De functie controleert eerst of het getal voor n gelijk is aan 0. Dit is de stopconditie, en is noodzakelijk omdat het een *recursieve* functie is. Recursieve functies werken doordat ze zichzelf aanroepen. Wanneer ze zonder limiet zichzelf aanroepen gaat dit proces tot in de eeuwigheid door, en zijn we in feite beland in een infinite-loop. Het recursieve gedeelte van de `fibonacci_aux` functie staat hieronder weergegeven.

```
fibonacci_aux (n - 1) second (first + second)
```

Hier roept de functie zichzelf aan met nieuwe parameters. N wordt met 1 in mindering gebracht, en het laatste getal (second) wordt meegegeven samen met het nieuwe getal (first + second). Wanneer we geen stopconditie gebruiken kan dit tot in de eeuwigheid doorgaan. Daarom is het volgende stukje code meegegeven.

```
if n == 0 then
```

Dit is de stopconditie. Het zegt in feite: "Als het aantal Fibonacci getallen dat ik nog moet berekenen 0 is, dan stop ik en geef ik niks () terug."Is dit niet het geval, dan laat de functie het eerste getal zien, en roept zichzelf voor het volgende getal aan. Dit resulteert in de Fibonacci reeks. Om dit verder te illustreren zijn een aantal van deze opeenvolgende recursieve aanroepen hieronder weergegeven.

```
fibonacci_aux 8 1 2
fibonacci_aux 7 2 3
fibonacci_aux 6 3 5
fibonacci_aux 5 5 8
fibonacci_aux 4 8 13
```

Zoals je kunt zien beginnen de Fibonacci getallen tevoorschijn te komen. De reeks die deze functie produceert werkt door het getal *first* (de tweede parameter) te tonen op het scherm. Dan krijg je de volgende reeks: *1, 2, 3, 5, 8*. Dit is het begin van de Fibonacci reeks. Het programma kan dit produceren zonder gebruik te maken van gedeclareerde variabelen. Dit is de magie van functionele programmeertalen, en maakt het makkelijker om te voorspellen wat een programma zal doen, omdat het geen zogenaamde *side effects* heeft (veranderingen elders in het programma). Functionele programmeertalen hebben ook overeenkomsten met een andere soort programmeertaal, namelijk declaratieve programmeertalen.

1.2.4 Declaratief Programmeren

Declaratieve programmeertalen zijn een interessante soort. In plaats van aan te geven *hoe* een programma iets moet doen, geven declaratieve talen aan *wat* het programma moet doen. Om dit verschil aan te geven kunnen we het best een aantal voorbeelden bekijken. Declaratieve werken vaak met informatie die als verschillende relaties worden opgeslagen in het geheugen. De programmeertaal *Prolog* is een bekend voorbeeld dat ook zo werkt. Prolog wordt vaak gebruikt voor toepassingen in kunstmatige intelligentie. In Prolog kun je namelijk *feiten* als volgt declareren.

```
cat(tom).
```

Deze regel code declareert dat Tom een kat is. Met dit feit opgeslagen in het geheugen van Prolog kunnen we nu vragen gaan stellen. Bekijk het volgende stukje code en de resultaten.

```
?- cat(tom).
Yes

?- cat(X).
X = tom
```

Het eerste stukje code vraagt: *is Tom een kat?*. Het resultaat is *Yes*, wat klopt omdat we eerder aan hebben gegeven dat Tom inderdaad een kat is. We kunnen ook Prolog vragen om alle katten te tonen. Dat doen we door in plaats van een specifieke waarde, een variabele te geven. Prolog zal dan alle mogelijkheden voor die variabele laten zien. Het tweede stukje code laat dit gebruik zien. Op deze manier kun je complexe structuren bouwen en kan een programma geschreven in Prolog als het ware *redeneren* over de feiten die hem worden verschaft.

1.3 Object Georiënteerd Programmeren

Het laatste en, voor ons, meest belangrijke programmeerparadigma dat we zullen behandelen is het object georiënteerde paradigma. Dit is tevens de focus van dit boek en het zal verder worden uitgelegd in de volgende paragrafen en hoofdstukken. Object georiënteerde programmeertalen worden veelgebruikt om programma's te maken voor een groot scala aan afnemers, hetzij de zakelijke wereld, industrie, de overheid en meer. Het doel van object georiënteerde programmeertalen is om het schrijven van software makkelijker te maken door het dichter bij de *echte wereld* te brengen. Hiermee bedoelen we dat je object georiënteerde software kunt vormgeven naar gelang het probleem dat je aan het oplossen bent. Dit komt doordat object georiënteerde talen, zoals de naam al doet vermoeden, focussen op het gebruik van *objecten*. Objecten bestaan in onze echte wereld, bijvoorbeeld de stoel waar je nu mogelijk op zit of het boek dat je nu leest. Om software schrijven makkelijk te maken bieden object georiënteerde talen ook een mogelijkheid om deze objecten te maken.

Objecten in de software wereld bestaan uit twee dingen, namelijk *data* en *gedrag*. Deze twee worden in vaktermen respectievelijk attributen en methoden van het object genoemd. Ze bestaan om een object vorm te kunnen geven. Stel dat je in software een persoon-object wilt maken. Een persoon in de echte wereld heeft bijvoorbeeld een naam en kan zich voortbewegen (bijvoorbeeld door te lopen). In software kunnen we het object dan een attribuut geven voor de naam van het persoon, wat in dit geval een tekstwaarde is. Ook kunnen we functionaliteit toevoegen om te kunnen voortbewegen. We

hebben dan het volgende object.

```
Persoon
+ naam :  tekst
- beweeg :  functie
```

Wat hierboven staat is een (zeer) versimpelde weergave van het persoon object in *UML*. UML staat voor *Unified Modelling Language*, en is een veelgebruikte manier om software schematisch weer te geven in verschillende modellen. Het juiste gebruik van UML zal straks aan de orde komen. In de beschrijving van persoon staat een interessant iets dat nog niet eerder is genoemd, dat is het gebruik van de + en - symbolen. Deze symbolen staan voor de zichtbaarheid van de onderdelen voor de *buitenwereld*. Hiermee wordt bedoeld: *"Kunnen andere objecten dit onderdeel veranderen of aanspreken?"*. Een - staat voor het onzichtbaar zijn voor de buitenwereld, een + staat voor het zichtbaar zijn. Dit betekent dat een ander object wel bij de naam van een persoon kan, maar niet de functie om te kunnen voortbewegen kan aanroepen. Dit is in theorie logisch, omdat ieder persoon zelf gaat over of hij of zij wil gaan bewegen. Let wel, dit voorbeeld is sterk versimpeld, en het is zelfs een slecht idee om andere object onbeperkt toegang tot een variabele te geven. Dit onderwerp zal later verder aan de orde komen. Het concept wat hier wordt getoond heet *information hiding*. Dit concept wordt toegepast om software robuuster te maken. Eerder hebben we gezien dat *side effects* voorkomen in software doordat code soms veel extra waarden binnen een programma veranderd. Wanneer code toegang heeft tot *alle* waarden binnen een programma kan elk onderdeel van het programma het gehele programma veranderen. Wanneer we dit toelaten is het einde zoek, omdat de kans erg groot is dat verschillende stukken code met dezelfde informatie binnen het programma aan de haal gaan, en zo fouten creëren. Om dit te voorkomen plaatsen we restricties op bepaalde onderdelen van het programma, zodat alleen bepaalde onderdelen van het programma daar bij kunnen komen. Zo hebben we een beter overzicht van wat onze software doet. Het geheel wat we hierboven beschreven hebben heet *data abstraction*. Data abstraction betekent dat programmeurs zelf objecten kunnen definiëren, deze kunnen opbouwen uit verschillende onderdelen (attributen en methoden) en kunnen aangeven welke wel en niet zichtbaar zijn voor andere objecten.

Object georiënteerde talen hebben nog enkele andere onderdelen wat ze tot object georiënteerde talen maakt. Deze zullen hieronder kort worden besproken en later in dit boek verder worden uitgelegd. Het eerste onderdeel is de mogelijkheid tot wat we in vaktaal noemen *inheritance*. In het Nederlands heet dit *overerving*. Dit heeft betrekking op objecten, en betekent dat

objecten de definitie over kunnen nemen van andere objecten. Hierboven hebben we een persoon object gezien. Stel dat we in software een *werknemer* object willen definiëren. Een werknemer is ook een persoon, en het is handig voor ons om die functionaliteit direct over te kunnen nemen. In plaats van de code voor persoon te kopiëren, wat foutgevoelig is, kunnen we overerving gebruiken om dezelfde functionaliteit van de class persoon ook in het object werknemer te kunnen gebruiken. Dit voorbeeld kunnen we gebruiken om het volgende concept uit te leggen, namelijk *polymorfisme*. We nemen aan dat in onze wereld bepaalde soorten van objecten allemaal dezelfde eigenschappen hebben. Personen hebben allemaal een naam en kunnen zich allemaal voortbewegen. Echter, de manier waarop bepaalde personen zich voortbewegen kan verschillen. Polyformisme, ook wel veelvormigheid genoemd, gaat over het feit dat alle soorten personen zich kunnen voortbewegen (werknemers, sporters, zeelui), maar dat ze dit allemaal op hun eigen specifieke manier doen. In ons eerdergenoemde voorbeeld bestaat er voor elk type persoon een soort object. Objecten voor werknemers, sporters en zeelui, en nog veel meer. Elk van deze objecten is een persoon, en kan zich dus voortbewegen. Maar de code die daarvoor wordt gebruikt verschilt per type persoon. Het feit dat we per type object andere code voor dezelfde methode kunnen gebruiken noemen we polymorfisme.

Als laatste behandelen we nog kort hoe objecten worden opgebouwd in code. Je definieert objecten in een *klasse* (in het Engels een *class*). Een klasse is een stuk code waarin staat gespecificeerd hoe het object heet en welke attributen en methoden het heeft. Een voorbeeld hiervan staat hieronder.

```
class Person {
  public string name;

  private void move() {
    //Code here...
  }
}
```

Eerder hebben we gezien dat bij procedurele programmeertalen een van de grote voordelen *herbruikbaarheid* is. Dit is ook een voordeel bij het gebruik van klassen. Binnen het object georiënteerde domein wordt dit *modulariteit* genoemd. Modulariteit betekent dat software wordt opgesplitst in onderdelen (modulen) die hergebruikt kunnen worden, binnen de eigen software maar zelfs ook binnen andere software.

1.3.1 4 Kernprincipes van object georiënteerde talen

Uit de stof hierboven kunnen we een viertal kernprincipes van object georiënteerde talen destilleren, namelijk:

1. **Abstraction**
   Het kunnen omzetten van het probleem (bijvoorbeeld uit de echte wereld) naar de software wereld.
   Kernwoorden: Data abstraction

2. **Seperation**
   Het kunnen kiezen welke onderdelen van een object wel en niet zichtbaar zijn voor de buitenwereld.
   Kernwoorden: Information hiding

3. **Composition**
   Het kunnen opbouwen van objecten uit andere objecten
   Kernwoorden: Modulariteit

4. **Generalization**
   Het kunnen aangeven van relaties tussen verschillende soorten objecten
   Kernwoorden: Overerving, Polymorfisme

Deze kernprincipes helpen ons object georiënteerd programmeren beter te begrijpen door te doorgronden wat deze principes betekenen. Ze zullen dan ook worden behandeld in de volgende hoofdstukken.

1.4 Van code tot programma

Wanneer we een programma schrijven in een bepaalde programmeertaal, kan de computer deze code nog niet uitvoeren. Dit komt doordat een computer is opgebouwd vanuit een binaire gedachte, wat inhoud dat computers werken met bits. (1 & 0) Dit is in de paragrafen hierboven reeds behandeld, maar behoeft toch iets meer uitleg. Wanneer we de verschillende programmeerparadigma's van eerder in dit hoofdstuk in volgorde van verschijnen bekijken, zien we een trend van een steeds hoger niveau. Dit wordt mogelijk gemaakt omdat nieuwe programmeertalen door kunnen bouwen op hun voorgangers. Machine code wordt versimpeld door commando's in assembly language te beschrijven. Met deze assembly language kan dan weer de taal C mogelijk worden gemaakt, enzovoorts. Dit komt doordat programmeurs makkelijker konden programmeren in assembly language, en zodoende complexere programma's konden schrijven. Een van die programma's is een *compiler*.

1.4.1 Compileren

Een compiler word gebruikt in een proces dat compileren heet. Compileren is het proces om code, geschreven in een bepaalde programmeertaal, uitvoerbaar te maken door een computer. Eerder hebben we gezien dat een computer gemaakt is om programma's in machinecode uit te kunnen voeren. Een compiler genereert dan ook vaak dat soort code. Een compiler brengt de complexere structuren van een C programma terug naar assembly language, welke wordt veranderd in machinecode om uitgevoerd te kunnen worden. Dit proces werkt via allerlei slimme technieken om de compiler de code te kunnen laten lezen. Ook deze techniek heeft een grote ontwikkeling doorgemaakt. Om dit te kunnen doen neemt de compiler de code, en verdeelt deze in verschillende *tokens*. Dit zijn als het ware labeltjes die de compiler op onderdelen van code plakt. Wanneer de compiler het woordje `for` tegenkomt krijgt dat een speciaal label voor het *for keyword*, en wanneer een getal zoals `3.14` wordt gezien krijgt dat ook zijn eigen label. Zo verdeelt de compiler de enorme verzameling aan code in een verzameling gelabelde onderdelen. Daarna probeert de compiler te achterhalen hoe die verschillende onderdelen in elkaar passen. Dit werkt doordat een programmeertaal vol zit met taalregels. Wanneer we een variabele een waarde toewijzen doen we dit meestal als volgt: `getalVar = 14+3;`. De compiler verdeelt labels over de tekst, wat resulteert in het volgende: `variabele(getalVar) isTeken(=) expressie(14+3)`. De compiler kijkt dan in zijn regels om uit te vogelen wat hiermee bedoelt wordt. De compiler heeft een regel waarin staat dat wanneer een variabele gevolgd wordt door een = teken met daarachter ten slotte een waarde, dit betekent dat een waarde moet worden toegekend aan een variabele. Zodra dit bekend is zal de compiler dit stuk code vertalen naar assembly language, of een andere doeltaal. Deze vertaalde code wijst dan en waarde toe aan de variabele. Dit is een versimpeld voorbeeld van de werkelijkheid, omdat compilers zeer complexe programma's zijn met complexe regels. Taalregels worden samen de *syntax* van een taal genoemd, en compilers controleren aan de hand van die syntax de code.

Software-gesimuleerde talen

Wanneer een programmeertaal een simpele syntax met relatief weinig regels heeft , indiceert dit meestal dat de taal een *software-gesimuleerde* programmeertaal is. Voorbeelden hiervan zijn Python en Javascript. Deze talen worden uitgevoerd door een *Interpreter*. Een interpreter is een programma dat lijkt op een compiler, maar het programma dat wordt ingelezen direct uitvoert in plaats van het eerst om te zetten naar machinecode. Dit heeft

voordelen, zoals het snel kunnen aanpassen van het programma en het feit dat het overal uitgevoerd kan worden mits de interpreter aanwezig is. Het heeft echter ook nadelen, zoals het feit dat dit soort programma's langzamer uitvoeren dan programma's die zijn gecompileerd naar machinecode, wat goed het verschil tussen interpreters en compilers illustreert. Een interpreter zet tijdens het uitvoeren van het programma de code om in instructies voor de machine waarop deze wordt uitgevoerd. Een compiler zet juist *voordat* het programma wordt uitgevoerd de code om in machinecode, en daarna wordt pas het programma uitgevoerd. De stappen die de compiler al heeft gedaan voordat het programma wordt uitgevoerd, moeten nog tijdens uitvoeren door de interpreter worden behandeld. Dit kost extra tijd en is daardoor langzamer.

Gecompileerde talen

Een programmeertaal met een groot aantal (vrij strenge) regels indiceert meestal een gecompileerde taal. Deze talen zijn zo specifiek bedacht omdat ze een relatie hebben met de hardware van een computer. Ze zijn zo ontworpen dat ze goed kunnen worden omgezet naar instructies van de hardware van de computer. Dit proces heet *vertalen*, en het programma dat vertaalt wordt, de *source code*, wordt vertaald naar zogenaamde *object code*. Deze uitvoerbare object code wordt opgeslagen in *object bestanden*. Wanneer je programmeert in C zijn deze object bestanden letterlijk bestanden met de `.o` extensie. In de C# implementatie van Microsoft is dit echter anders, omdat het .NET framework gebruikmaakt van *Just-In-Time Compilatie*. (JIT-Compilatie) Dit model is een variant op de beide modellen die we eerder hebben gezien.

Compileren in het .NET Framework

Zoals hierboven genoemd maakt het .NET framework gebruik van JIT-Compilatie. JIT-Compilatie is een speciale vorm van compilatie, omdat het zoals de naam zegt *precies op tijd gebeurt*. Dit is het beste uit te leggen door te kijken naar de structuur van het .NET framework. Wanneer je een programma schrijft voor .NET kan dat in een aantal talen gebeuren, waar C# er eentje van is. Microsoft ondersteund daarnaast ook andere talen zoals F# en VB.NET. Deze talen zijn allemaal uitvoerbaar in het .NET framework. Een ander stukje van de puzzel is dat wanneer je een .NET programma wilt uitvoeren, je altijd eerst .NET moet installeren op je computer. Dit alles wijst erop dat er meer achter .NET schuilgaat dan het simpel vertalen naar een machinetaal. Dit is inderdaad het geval, omdat .NET gebruikmaakt van een zogenaamde *virtuele machine*. Dit is een programma dat als het ware een

computer simuleert. Dit is een krachtig concept, omdat Microsoft deze virtuele machine heeft gemaakt voor alle grote besturingssystemen, in de vorm van .NET Core. .NET Core heeft een onderdeel genaamd *CoreCLR*, en de gewone .NET op Windows heeft de *CLR*. CLR staat voor *Common Language Runtime*, en is de virtuele machine waar we eerder over spraken. Deze machine kan geen C# of een van de andere .NET talen lezen, maar wel de zogenaamde *CIL*, de *Common Intermediate Language*. Dit is een tussentaal, welke lijkt op assembly langauge. De CLR zorgt ervoor dat middels JIT-Compilatie de CIL code wordt omgezet naar machinecode van de huidige machine waarop de CLR draait. Dit betekent dat een programma geschreven in C# wordt gecompileerd naar CIL code. Die code is uitvoerbaar in de CLR, welke aanwezig moet zijn op de computer. Wanneer dit zo is, begint de CLR met het uitvoeren van het programma. De CLR zorgt middels de JIT-Compiler ervoor dat de CIL code wordt omgezet naar machinecode die de computer kan uitvoeren.

Wanneer je een IDE (Integrated Development Environment) gebruikt zul je niet veel van de compiler en deze andere onderdelen zien. Deze ontwikkelomgevingen nemen veel van dat werk uit handen, maar het is goed om te weten dat deze programma's in de achtergrond ervoor zorgen dat je code uitvoerbaar is op de computer.

1.5 Samenvatting

We hebben gelezen over wat programmeertalen zijn, welke verschillende niveaus van programmeertalen bestaan, hoe ze gecategoriseerd kunnen worden en hoe object georiënteerde talen globaal werken en welke principes ze volgen. Ook hebben we kort gekeken naar wat compilers zijn en wat ze doen, en hoe dit werkt binnen het .NET framework. In de volgende hoofdstukken zal verder in worden gegaan op hoe we kunnen programmeren in de object georiënteerde taal C#, aan de hand van de hierboven genoemde kernprincipes.

Bibliografie

[1] Stanley Selkow George T. Heineman, Gary Pollice. *Algorithms in a nutshell*. O'Reilly, 2009.

[2] Dietrich Birngruber Albrecht Wöß Hanspeter Mössenböck, Wolfgang Beer. *.NET Application Development with C#, ADO.NET, ASP.NET and Web Services*. Addison Wesley, 2004.

[3] Dennis Kafura. *Object-Oriented Software Design & Construction with Java*. Prentice Hall, 2000.

[4] Wikipedia. Assembly language. `https://en.wikipedia.org/wiki/Assembly_language`, 2018. [Online; Gezien 10-Augustus-2018].

[5] Dorian P. Yeager. *Object-Oriented Programming Languages and Event-Driven Programming*. Mercury Learning and Information, 2014.

[]