CHAPTER 1

So you wanna be a Software Engineer?

Alexander

Stel je hebt de smaak van het *programmeren* te pakken hebt gekregen en wilt meer over *Software Engineering* te weten komen. In dit *part* enkele onderwerpen die eerder nog niet (uitgebreid) aan bod zijn gekomen maar ook belangrijk zijn.

CHAPTER 2

Achtergronden bij Software Engineering: Inleiding

(Dit stuk tekst komt voornamelijk af van Alexander)

Enkele belangrijke concepten

- Accessibility: internal en public voor classes, public, protected, private

voor functies.

- Inheritance, met daarbij abstracte classes abstracte en virtuele functies,

override van functies, aanroepen van base constructor

- Interfacing - waarom kennen we dat concept en waarom zou je het gebruiken
- Exception handling - wanneer is iets een fout en wanneer acceptabel gedrag

met een onverwacht resultaat.

- Events

CHAPTER 3

Basisconcepten van programmeren

Bij OOP (Object Oriented Programming) worden een aantal basisconcepten gebruikt die voor elke taal hetzelfde zijn en slechts afwijken in de manier waarop je ze opschtijft. De basisconcepten zijn op te delen in grofweg drie categorien:

Statements Een commando wat onvoorwaardelijk wordt uitgevoerd. Denk hierbij bijvoorbeeld aan het toekennen van een variabele of het aanroepen van een functie.

Keuzestructuren Een (aantal) statement(s) wat alleen uitgevoerd wordt als er aan een bepaalde voorwaarde wordt voldaan.

Herhalingsstructuren Een (aantal) statement(s) wat meermaals uitgevoerd wordt, mogelijk ook onder bepaalde condities.

3.1 Statements

Een statement is het meest basis concept bij het programmeren. Met een statement wordt een commando in het programma uitgevoerd. Hierbij kun je denken aan het declareren van een variabele, het toekennen van een waarde aan een variabele en het aanroepen van een methode. In code zien statements er als volgt uit:

```
int i = 4;
int j = 5;
int uitkomst = BerekenProduct(i, j);
```

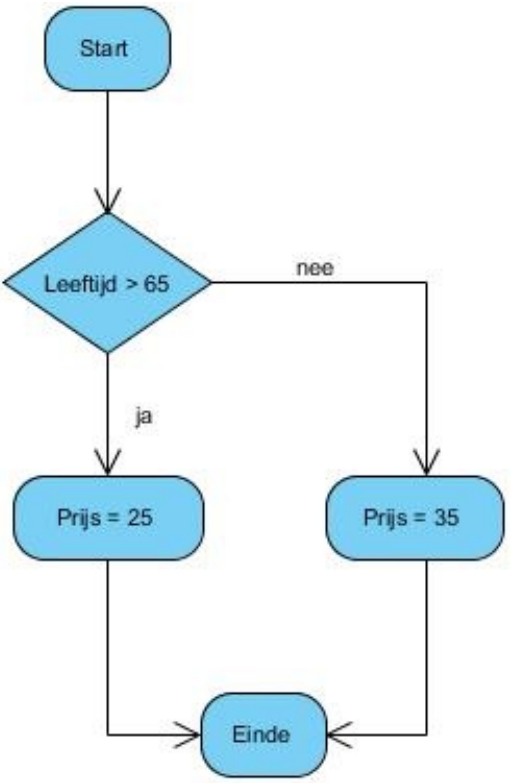

Figure 3.1 keuzestructuur

3.2 Keuzestructuren (if-statements)

Zoals de naam al aangeeft, gaat het bij keuzestructuren om het maken van een keuze of een bepaald stuk code wel of niet uitgevoerd wordt, afhankelijk van een bepaalde conditie. Meestal bestaat een keuzestructuur uit 2 delen: De code die uitgevoerd wordt als de conditie waar is en de code die uitgevoerd wordt als de conditie niet waar is. Dit wordt schematisch weergegeven in 3.1

In code ziet dit er dan als volgt uit:

```
if (Leeftijd > 65)
{
   prijs = 25;
}
else
{
   prijs = 35;
```

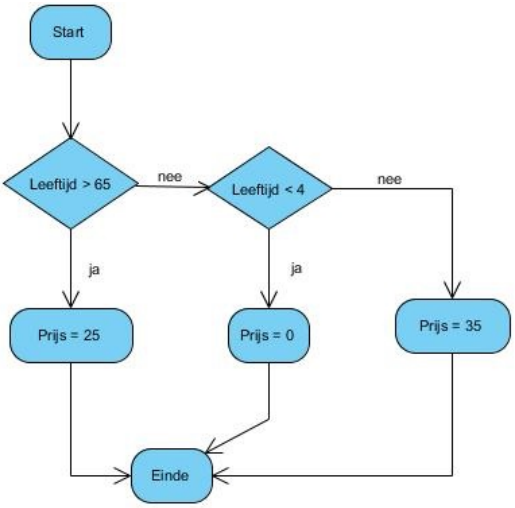

Figure 3.2 keuzestructuur met extra condities

```
}
```

Als uitbreiding op keuzestructuren zijn er nog twee varianten.

De eerste variant is de keuzestructuur, waarin er alleen code uitgevoerd wordt als de conditie waar is. Dit betekent dat in 3.1

als de conditie niet waar is, er geen actie meer uitgevoerd wordt en er dus meteen naar 'Einde' gegaan wordt. In de code betekent dit dat de 'else' constructie weggelaten kan worden.

De tweede variant is wat complexer en bevat meerdere keuzes die uitgevoerd worden. Dit betekent dat er in figuur 2.1 een extra keuze bijkomt aan de kant van de 'nee' en daarin wederom een keuze gemaakt wordt. Dit ziet er schematisch dan als volgt uit:

3.2

Zoals in het schema te zien is, kan elke aanpassing van de prijs in dit voorbeeld, maar onder 1 conditie uitgevoerd worden: De prijs wordt 25 als de leeftijd boven de 65 is, de prijs wordt 0 als de leeftijd NIET boven de 65 is, maar wel onder de 4 en in alle andere gevallen wordt de prijs 35. In de code komt er dan een 'else if' bij:

```
if (Leeftijd > 65)
{
   prijs = 25;
}
else if (Leeftijd < 4)
```

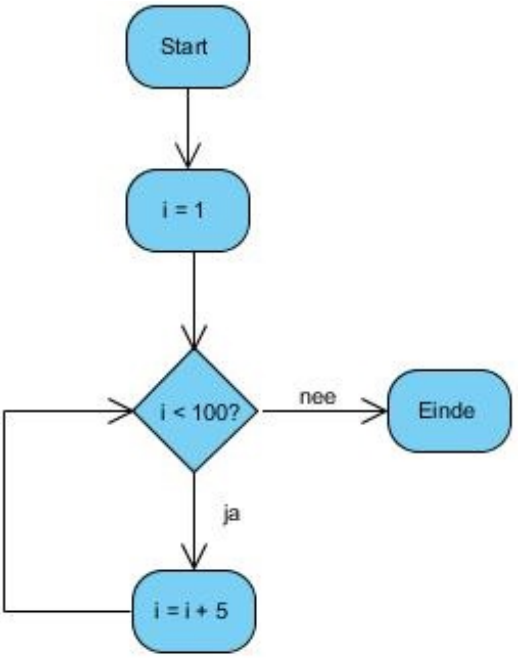

Figure 3.3 herhalingsstructuur

```
{
   prijs = 0;
}
else
{
   prijs = 35;
}
```

3.3 Herhalingsstructuren (loops)

Als bepaalde statements meerdere keren achter elkaar uitgevoerd moeten worden, dan worden daar herhalingsstructuren (lussen of loops in het Engels) voor gebruikt. Conceptueel zijn er drie varianten van de herhalingsstructuur: conditie-gebaseerd, aantal-gebaseerd en lijst-gebaseerd.

Afhankelijk van op welke manier de code meermalen uitgevoerd moet worden, kies je 1 van onderstaande methoden.

Conditie-gebaseerde structuur (while)

Het komt vaak voor dat code uitgevoerd moet worden zolang er aan een bepaalde conditie voldaan is. Stel dat je bijvoorbeeld een getal moet ophogen met 5 tot dit getal boven de 100 komt, dan gebruik je hiervoor een conditie-gebaseerde structuur. Zie 3.3

In code ziet dit er als volgt uit:

```
int i = 1;
while (i < 100)
{
   i = i + 5;
}
```

Als de conditie de eerste keer al niet waar is, dan wordt de code binnen de while niet uitgevoerd. Een alternatief van deze variant is de do-while constructie. Bij de do-while wordt de code uitgevoerd, totdat de conditie niet meer waar is en wordt de code dus altijd ten minste 1x uitgevoerd. De code hiervoor ziet er als volgt uit:

```
int i = 1;
do
{
   i = i + 5;
} while (i < 100)
```

Aantal-gebaseerde structuur (for)

Een andere structuur die vaak voorkomt is het voor een vast aantal keer uitvoeren van dezelfde code. Denk hierbij aan het op het scherm tonen van de tafel van 10. Vooraf is duidelijk dat het tonen van de waarde precies 10 maal plaats moet vinden en dit aantal zal niet veranderen. Dit kan opgelost worden met de conditie-gebaseerde structuur, maar in (bijna) alle programmeertalen is hier een speciale constructie voor: de for-herhalingsstructuur. In code ziet deze er als volgt uit:

```
for (int i = 0; i < 10; i++)
{
   Console.WriteLine(i * 10);
}
```

De eerste regel bestaat uit 3 delen:

- int i = 0: De startwaarde van de teller
- i < 10: De conditie die vertelt wat de eindwaarde van de teller is, ofwel: zolang deze conditie niet waar is, wordt de code uitgevoerd.
- i++: De verhoging van de teller.

Bovenstaande code zou ook als een while-lus geschreven kunnen worden, door de startwaarde boven de while te zetten, en de verhoging als laatste statement binnen de lus op te nemen. Je krijgt dan qua structuur een zelfde schema als die in figuur 2.3.

Lijst-gebaseerde structuur (foreach)

Als je op alle elementen van een lijst een actie wilt uitvoeren, dan kun je hiervoor een for-lus gebruiken waarbij de conditie voor de eindwaarde het aantal elementen uit de lijst bevat. De meeste talen bieden hier ook een specifieke structuur voor aan: de foreach-lus. In code ziet dit er als volgt uit:

```
List<Persoon> personen = HaalAllePersonenOp();
foreach (Persoon persoon in personen)
{
   Console.WriteLine(persoon.Naam);
}
```

Je ziet dat je in deze structuur niet expliciet hoeft aan te geven hoeveel iteraties uitgevoerd gaan worden, en je hebt direct de instantie van de Persoon beschikbaar.

CHAPTER 4

Variabelen

4.1 Scope

Elke variabele heeft een scope. De scope van een variabele geeft aan wanneer de variabele bestaat. Neem bijvoorbeeld de volgende code:

```
int i = 1;
if (i == 1)
{
   int j = i;
   j = j * 2;
}
i = i + 1;
```

Vanaf het moment dat de variabele i aangemaakt wordt, bestaat deze en blijft deze in dit hele stuk code bestaan en kan dus binnen en na de if-statement gebruikt worden. Voor de variabele j is dit anders. De variabele j wordt binnen de scope van het if-statement gedefinieerd, waardoor deze variabele dus alleen binnen het if-statement gebruikt kan worden. Na de afsluitende accolade van het if-statement kan de variabele j niet meer gebruikt worden. De compiler herkent dit en zal een foutmelding geven als je een variabele buiten zijn scope probeert te gebruiken.

Scoping kennen we op een aantal niveau's:

- Een variabele die gedeclareerd wordt binnen een structuur (herhalings- of keuzestructuur) kan alleen binnen die structuur gebruikt worden.
- Een variabele die gedeclareerd wordt binnen een methode kan alleen binnen die methode gebruikt worden.
- Een variabele die gedeclareerd wordt in een klasse (bijvoorbeeld een

property of private field) kan gebruikt worden zolang de instantie van een klasse bestaat.

- Een variabele die static in een klasse gedeclareerd wordt kan altijd gebruikt worden.

4.2 Typen

In programmeren hebben we het over het type van een variabele als we bedoelen wat de variabele voor data en functionaliteit kan bevatten. Denk hierbij aan een `int` (integer) voor gehele getallen, string voor tekst en File voor een bestand. Ook kun je zelf nieuwe typen toevoegen door een 'class' te definieren. Bij object-georienteerde talen onderscheiden we twee soorten typen:

Primitieve typen: Zoals de naam al aangeeft, zijn primitieve typen de meest basis variant van variabelen. Hierbij kun je denken aan int en decimal om getallen op te slaan, bool om true of false op te slaan en char om een letter in op te slaan. Dit is dan ook het enige wat je met deze typen doet.

Object typen: Object typen (of ook wel eens complexe typen genoemd) zijn typen die gebruikt worden om meer complex gedrag te modelleren. In deze typen kunnen naast waarden, ook gedrag gecodeerd worden. In OO-talen worden deze typen gemaakt door een nieuwe Class te definieren en deze van een implementatie te voorzien.

Uitzondering: String: Strings zijn binnen OO-talen een vreemde eend in de bijt. Eigenlijk is een string een object (namelijk: een collectie letters van het type char in een specifieke volgorde). Echter: in de praktijk is dit meestal helemaal niet handig. Daarom is de string in een aantal OO-talen een hybride vorm. In C# bijvoorbeeld, wordt de string toegepast als primitief type als je strings opslaat, maar als object als je string-functionaliteiten gebruikt zoals Split, Replace en Substring.

4.3 Value typen versus Reference typen

We hebben in de vorige sectie gekeken naar primitieve typen versus object typen. Het verschil tussen deze typen leidt tot een belangrijk verschil in OOprogrammeren tussen deze typen: Alle primitieve typen zijn value typen en alle object typen zijn reference typen. De String valt in dit geval onder de primitieve typen.

Hoe werkt een value type?

Wanneer je in code een primitief type declareert (bijvoorbeeld: `int leeftijd = 4`), dan wordt er een stukje geheugen gereserveerd waarin deze waarde

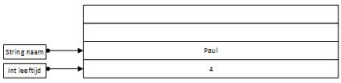

Figure 4.1 Geheugen layout bij uitvoeren code

Figure 4.2 Geheugen layout bij uitvoeren code

wordt opgeslagen. De variabele leeftijd verwijst dan naar dit stukje geheugen, waarin de integer waarde 4 is opgeslagen. Dit heeft een aantal gevolgen:

- In dit stukje geheugen mag alleen maar een integer staan. Hier kun je dus geen string, komma-getallen of objecten aan toekennen.
- Dit stukje geheugen is alleen beschikbaar voor de variabele leeftijd. Er is geen andere variabele die naar hetzelfde stuk geheugen verwijst en een aanpassing van de waarde kan dus alleen via een aanpassing van de variabele leeftijd.
- Wanneer de variabele leeftijd niet meer bestaat, wordt het geheugen weer vrijgegeven.

Stel je hebt de volgende code:

```
int leeftijd = 4;
string naam = "Paul";
```

Dan ziet het geheugen er op dat moment uit als in figuur 4.1.

Een ander aspect van value types is het toekennen van de ene variabele aan de andere (bijvoorbeeld int oudeLeeftijd = leeftijd). Als dit uitgevoerd wordt, dan wordt er in het geheugen een kopie gemaakt van de waarde in leeftijd en deze wordt toegekend aan oudeLeeftijd. Dit betekent dus, dat als je daarna de waarde van leeftijd aanpast, de waarde van oudeLeeftijd NIET mee verandert. Stel dus dat je de volgende code uitvoert:

```
int leeftijd = 4;
int oudeLeeftijd = leeftijd;
leeftijd = 10;
```

Dan ziet het geheugen er na het uitvoeren van de code uit als in figuur 4.2

Hoe werkt een reference type?

Als je een reference type declareert dan doe je dit met het keyword *new*. Op dat moment worden er niet één, maar twee stukjes geheugen gereserveerd:

Figure 4.3 Geheugen layout bij uitvoeren code

Figure 4.4 Geheugen layout bij uitvoeren code

In één stuk geheugen wordt de inhoud van de variabele opgeslagen en in het andere stuk geheugen wordt een referentie naar deze inhoud opgeslagen. Stel dus dat je de volgende code uitvoert:

```
Auto opel = new Auto("OpelZafira");
```

dan ziet het geheugen er uit als in 4.3

Zoals je ziet zijn er 2 stukjes geheugen gereserveerd, in verschillende delen van het geheugen.

Ook het toekennen van de ene variabele aan de andere werkt bij reference typen anders. Wanneer je een variabele toekent aan een andere, wordt er één extra stukje geheugen gereserveerd, waarin een referentie naar de al bestaande inhoud van wordt opgeslagen. Stel dus dat je de volgende code hebt:

```
Auto opel = new Auto("OpelZafira");
Auto nogEenOpel = opel;
```

Dan ziet het geheugen er uit als in figuur 4.4

Gevolg hiervan is dat als je de naam van de auto aanpast via de variabele *opel* (`opel.Naam = 'Opel Corsa'`), deze ook meteen aangepast is als je de naam ophaalt via *nogEenOpel*.

Note Dit geldt ook bij parameters in functies.

Tips bij typen variabelen

Een vuistregel om te weten of iets een value of reference type is, is te kijken naar de declaratie van de variabele: –Wordt de variabele toegekend door er direct een waarde aan te koppelen (`int i = 4`), dan is het een value type. –Wordt de variabele toegekend door het gebruik van new (`Auto auto = new Auto()`), dan is het een reference type.

CHAPTER 5

OO-technieken

5.1 Inheritance: Hoe werkt dit in het geheugen

In de theorie van object georienteerd programmeren is inheritance (afgeleide klassen) en interfacing aan bod gekomen (Zie hiervoor de relevante modules in Canvas of zoek op internet). In dit hoofdstuk komt aan bod hoe inheritance in het geheugen werkt en wordt daarmee een achtergrond gegeven over inheritance als concept.

Een voorbeeld van een *class diagram* dat gebruik maakt van *inheritance* is te zien in figuur 5.1

Als je dit klassendiagram geïmplementeerd hebt, dan kun je al deze klassen gebruiken om objecten mee te maken. Stel dat je de volgende code schrijft:

```
Motorboot yamaha = new Motorboot();
```

Er wordt dan een stuk geheugen gereserveerd waarin deze Motorboot wordt opgeslagen (op de manier zoals beschreven in hoofdstuk 3.3). In het stuk geheugen waar het object opgeslagen is, zitten alle gegevens van Motorboot

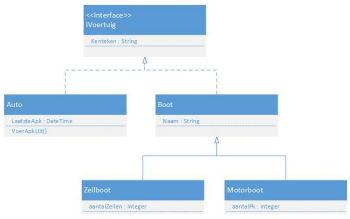

Figure 5.1 Class diagram

(aantalPk), alle gegevens uit Boot (naam) en alles uit IVoertuig (Kenteken). Het type van de variabele geeft aan bij welke van deze gegevens je mag. In het geval van Motorboot betekent dit dat je bij alle gegevens mag.

Stel dat je nu de volgende code hebt:

```
Motorboot yamaha = new Motorboot();
Boot boot = yamaha;
IVoertuig voertuig = yamaha;
```

Er is nu een stuk geheugen gereserveerd waarin de Motorboot is opgeslagen. De variabele yamaha bevat een referentie naar dit stuk geheugen. De regel code waarin de boot variabele wordt aangemaakt, bevat een referentie naar hetzelfde stuk geheugen als de motorboot, maar als je de variabele boot gebruikt kun je niet meer bij het aantalPk. Hetzelfde geldt voor de variabele voertuig: je kunt nu alleen nog bij het Kenteken, ondanks dat de verwijzing naar het stuk geheugen is waar alle data staat.

Beperkingen

In het voorbeeld van de Motorboot werd een motorboot aangemaakt, waarna deze als Boot en IVoertuig gebruikt werd. Andersom kan dit niet. De volgende code zal dus niet compilen:

```
Boot boot = new Boot();
Motorboot yamaha = boot;
```

De reden hiervan is dat het geheugen wat gereserveerd is wel de gegevens van Boot en IVoertuig bevat, maar niet van Motorboot. Als je dat geheugen dan wilt benaderen als Motorboot zou er een stukje geheugen moeten zijn waarin het aantalPk opgeslagen is, maar dit is er niet. Wat ook niet kan is het volgende:

```
Boot boot = new Boot();
Auto auto = boot;
```

Hier geldt ongeveer hetzelfde: In het geheugen is wel IVoertuig en Boot beschikbaar, maar niet Auto. Hierdoor is het niet mogelijk het geheugen te benaderen alsof het een auto is. 13

5.2 Software Architecture

Als je een hondehok bouwt kun je wat planken nemen en een hamer met spijkers en dan direct beginnen met knutselen. Een vakman zal eerst een schets maken hoe het er uit moet komen te zien. Gaan we een huis (of groter) bouwen dan komt daar meer ont-werp bij kijken en spreken we van architectuur.

Ook bij het bouwen van software is het al snel verstandig om van te voren een *schets* of ander ontwerp te maken. Het *probleem* is doorgaans *opgelost* voordat de *code* ingeklopt wordt!

Om van te voren een oplossing(srichting) te kunnen bedenken is ervaring nodig, het valt buiten de scope van dit materiaal. Een geïnteresseerde verwijzen we graag naar de volgende site: http://aosabook.org/en/index.html