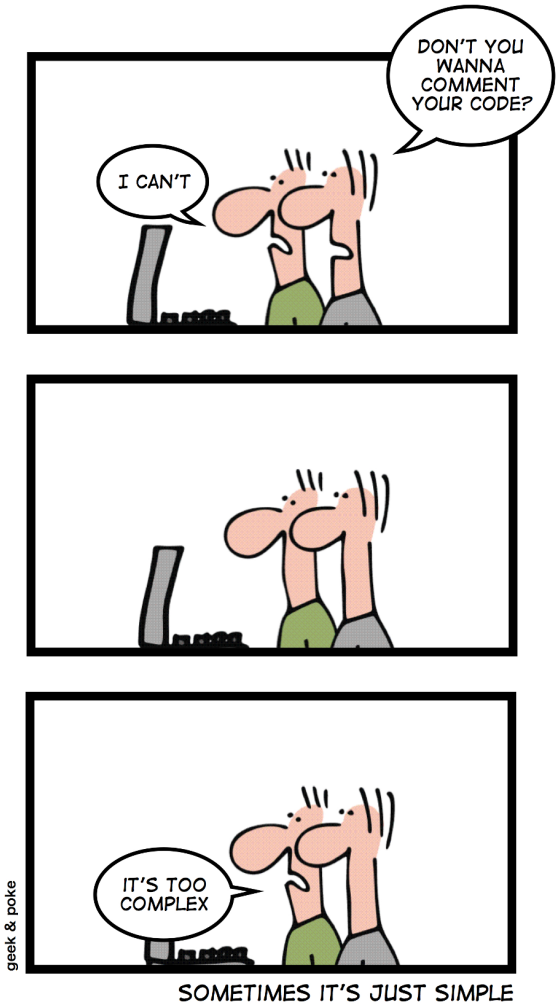
Universiteit van Amsterdam
Inleiding Programmeren

José Lagerberg - Robin de Vries - Tim van Deurzen - Stephen Swatman

Styleguide



Inhoudsopgave

1 Inleiding

Bij het schrijven van je eerste programma maakt het nog niet zoveel uit hoe de code eruit ziet, zolang het maar werkt en duidelijk is hoe het werkt. Zo'n eerste programma is meestal ook niet zo groot en zal, negen van de tien keer, een *'hello world'*[1] programma zijn. Echter, als je grotere en meer complexe programma's gaat schrijven, wordt het ook belangrijk om te kijken naar hoe de code van dat programma eruit ziet. Nette code heeft namelijk enkele zeer belangrijke voordelen:

- De code is makkelijker te lezen.
- Fouten zijn makkelijker te vinden.
- De werking van het programma is sneller duidelijk voor programmeurs die met de code moeten of willen werken.

Er zijn vele verschillende richtlijnen te vinden op het internet, *minstens* één per programmeertaal, en er zijn zeer verhitte discussies over welke het beste is. Teams zijn uit elkaar gegaan door onenigheid over de stijl van de code. Daarom is het belangrijk dat je al vroeg een keer in aanraking komt met een styleguide. Zo weet je hoe je zelf de programmacode in orde kunt maken.

Het woord zegt het al, het is een 'guide', een gids. Het gaat hier niet om strakke regels die op straffe des doods gevolgd moeten worden. Het zijn richtlijnen, die je kunt aanhouden of negeren. Dit artikel geeft een aantal van zulke richtlijnen die handig zijn om aan te houden bij het maken van de opdrachten voor het vak inleiding programmeren. Wij raden aan deze richtlijnen goed te lezen en er altijd voor te zorgen dat de code er keurig netjes uitziet bij het inleveren van de opdrachten.

[1] Simpel programma dat slechts de woorden 'hello world' op het scherm print.

Het is bij het inleveren van code altijd belangrijk dat de code er netjes uit ziet, dat de naamgeving zinnig is, dat het makkelijk is te begrijpen hoe het programma werkt, dat er voldoende commentaar is, dat de structuur van de code logisch is en dat de stijl consistent is.

2 Nette code

Wanneer is programmacode netjes? Wat is de beste stijl voor je code? Dat zijn vragen waar vele verschillende antwoorden op gegeven kunnen worden. Het is voornamelijk een kwestie van smaak. En zoals we allemaal weten: over smaak valt *niet* te twisten.

2.1 Indentatie

Laten we beginnen met de indentatie: het gebruik van tabs of spaties om de code visueel te structureren. Waarom zou je de code willen indenteren? De belangrijkste reden om je code (goed) te indenteren is dat het de structuur van je programma duidelijker maakt. Kijk bijvoorbeeld eens naar de volgende voorbeelden:

Voorbeeldcode 1: Verschillende manieren om code te indenteren.

(a) Ongeïndenteerde code.

```
int counter() {
if(a==b || b==c) a++;
b=c; } else {
a=b; c--;
}
```

(b) Geïndenteerde code.

```
int counter() {
    if (a==b || b==c) {
        a++;
        b=c;
    } else {
        a=b;
        c--;
    }
}
```

Het tweede voorbeeld geeft een beter beeld van de functie, omdat de code visueel de structuur weergeeft: je ziet meteen wat onder het `if` gedeelte valt en wat niet. Wat je gebruikt om de code te indenteren, tabs of spaties, is jouw eigen keuze. Het belangrijkste is dat je tabs en spaties nooit door elkaar heen gebruikt. Je maakt een keuze en daar houd je je aan. Omdat de grootte van één tab een verschilt per editor, kan de code onleesbaar worden bij een mix van spaties en tabs.

Indien je hebt gekozen voor het gebruiken van spaties, blijft de vraag over *hoeveel* spaties je moet gebruiken. Meestal gebruik je twee, drie, vier of acht spaties. Welke de beste is, mag je zelf bepalen.

Vuistregel: *Correct geïndenteerde code is beter leesbaar dan niet (correct) geïndenteerde code.*

2.2 Accolades en haakjes

Bij veel programmeertalen is het noodzakelijk om gebruik te maken van accolades (ook wel krulhaakjes genoemd) om stukken code te groeperen. Dit is voornamelijk het geval bij conditionele statements (if-else, switch) en loops (do-while, for). Vergelijk de volgende stukken code om de verschillende mogelijkheden te zien voor het gebruik van accolades.

Voorbeeldcode 2: Verschillende manieren om accolades te plaatsen in `if-else` constructies.

(a) Accolade en conditie op dezelfde regel.

```
if (conditie) {
    ...
} else {
    ...
}
```

(b) Alleen sluitende accolade op een eigen regel.

```
if (conditie) {
    ...
}
else {
    ...
}
```

(c) Zowel openende als sluitende accolades op een eigen regel.

```
if (conditie)
{
    ...
}
else
{
    ...
}
```

Geen van deze voorbeelden is het beste. Je kunt zelfs combinaties maken van de bovenstaande voorbeelden. Het kan zijn dat je de code graag kort houdt. In dat geval kun je ervoor kiezen om accolades op dezelfde regel te zetten als de conditie. Als je de code graag visueel groepeert, dan wil je waarschijnlijk de accolades op een eigen regel zetten. Bij sommige programmeertalen (Python) mag je de accolades zelf helemaal weglaten. De voorbeelden hier zijn allemaal voor een if-statement, maar deze richtlijnen kun je ook toepassen bij methoden, functies en andere statements (switch, for, while).

2.3 Declaratie van variabelen

Binnen een programma worden vaak veel variabelen gebruikt. De meeste programmeertalen eisen dat je variabelen eerst declareert voordat je ze gebruikt. Bij programma's in Java, wordt er niet afgedwongen dat je de variabelen op één locatie declareert maar als je later met een programmeertaal als C gaat werken, dan wordt daar wel op gelet. Het is dan ook een goede gewoonte om variabelen bovenaan de klassen en methoden te declareren. Zo krijg je gemakkelijk een overzicht van welke variabelen je tot je beschikking hebt in een bepaalde methode.

Vuistregel: *Het is een goede gewoonte om variabelen bovenaan de klasse of methode te declareren.*

Bij het declareren van variabelen,initialiseer je de variabelen meestal ook. Je geeft ze dus gelijk een waarde. Sommige mensen vinden het fijn om deze waardes netjes onder elkaar in dezelfde kolom te zetten. Je kunt variabelen van hetzelfde type op dezelfde regel zetten, maar maak hierbij wel onderscheid tussen variabelen die wel en niet geïnitialiseerd worden.

Wanneer je variabelen declareert, is het handig om variabelen van hetzelfde type bij elkaar te zetten. Wanneer twee of meer variabelen gerelateerd zijn, ook al zijn van verschillende types, is het beter ze te groeperen per relatie.

Voorbeeldcode 3: Verschillende manieren om variabelen te declareren.

(a) Ongestructureerde declaratie en initialisatie van variabelen.

```
int a = 0;
long e = 5;
int b = 1, c;
int d;
char f = 'a';
```

(b) Gestructureerde declaratie en initialisatie van variabelen.

```
int  a = 0, b = 1;
int  c, d;
long e = 5;
char f = 'a';
```

(c) Variabelen gegroepeerd op onderlinge relatie.

```
int vos_aantal = 5;
Kleur vos_kleur = ROOD;

int lynx_aantal = 5;
Kleur lynx_kleur = BRUIN;
```

2.4 Regelafbreking

Vroeger programmeerde men op schermen die in de breedte 80 karakters konden weergeven. In die tijd was het dan ook erg handig om regels in je programmacode onder de 80 karakters te houden. We hebben tegenwoordig weliswaar computerschermen waar je veel meer tekst op kwijt kan, toch zweren veel programmeurs nog steeds bij de 80 karakter-regel. Redenen daarvoor zijn dat mensen optimaal kunnen lezen als de regels rond de 70 karakters zijn en omdat lange regels ook inhoudelijk snel te ingewikkeld worden. Langere programma's krijgen ook een plezante rechthoekige vorm.

Bij dit vak zullen we de limiet van 80 karakters niet als regel gehanteerd, maar wordt wel gehandeld in de geest van deze regel. Probeer je er dus aan te houden en ga zeker niet over de 120 tekens per regel heen.

Toch kan het gebeuren dat een regel langer wordt dan 120 tekens. Wat doe je in dat geval? Je moet de regel op een logische plaats afbreken en op de volgende regel verdergaan. Het is dan handig om op de nieuwe regel, de code extra te indenteren. Probeer daarbij alles verticaal uit te lijnen zodat het er netjes uit ziet. De regels hiervoor staan niet vast en je kunt aan deze verticale uitlijning je eigen invulling geven.

Voorbeeldcode 4: Verschillende manieren om lange regels af te breken.

(a) Onjuiste afbreking van lange regels.

```
if (zeer_lang_conditie_statement && nog_een_lang_conditie_statement
    || kort_conditie_statement || nog_een_statement &&
    belangrijk_statement || laatste_conditie) {
    ...
}
```

(b) Juiste afbreking van lange regels.

```
if (zeer_lang_conditie_statement
    && nog_een_lang_conditie_statement
    || kort_conditie_statement
    || nog_een_statement
    && belangrijk_statement
    || laatste_conditie) {
    ...
}
```

2.5 Witruimte

Als je een boek leest, is dat niet een onafgebroken lap tekst. De tekst zal in plaats daarvan onderverdeeld zijn in verschillende alinea's met witruimte ertussen. Ook verschillende woorden zijn van elkaar gescheiden met spaties. Kun je je voorstellen dat je een boek zou moeten lezen zonder spaties én zonder alinea's? Het gebruik van lege ruimte kan verrassend veel structuur aanbrengen in allerlei soorten informatie en programmacode is daar geen uitzondering op. Het is dan ook belangrijk dat je goed om leert gaan met witruimte en dat je het kunt gebruiken om structuur aan je code te geven – zowel horizontaal als verticaal.

2.5.1 Horizontaal

Als je gaat programmeren kan het zijn dat je ingewikkelde wiskundige formules en expressies gaat opschrijven. Als je alle operatoren aan elkaar plakt zonder witregel, ontstaat er een enorm blok wiskunde

waar niemand iets van begrijpt. Zorg er daarom voor, dat je rond je operatoren spaties neerzet zodat de code als het ware wat luchtiger wordt. De enige uitzondering op deze regel is de `!` operator, de logische negatie. Hierop volgt geen spatie.

Vuistregel: *Door spaties om operatoren heen te zetten, maak je expressies makkelijker om te lezen.*

Voorbeeldcode 5: Verschillende manieren om expressies te noteren.

(a) Zonder spaties kan een expressie onleesbaar zijn.

```
resultaat = (12*(12+n)-14*5);
```

(b) Met spaties kan een expressie leesbaarder worden.

```
resultaat = (12 * (12 + n) - 14 * 5);
```

Wees ook bedacht op het gebruik van spaties bij methodedefinities, het aanroepen van methoden, loops en conditionele statements. Wees hierbij consistent met spaties voor en na haakjes. Wanneer je meerdere argumenten meegeeft aan een methode zet je altijd een spatie na de komma, net zoals in een tekst.

Vuistregel: *Bij het definiëren of meegeven van argumenten aan methoden volgt na iedere komma een spatie.*

2.5.2 Verticaal

Om stukken code die logisch bij elkaar passen te structureren kun je na ieder blok code een of twee regels open laten. Op deze manier wordt het visueel duidelijk dat bepaalde regels code bij elkaar horen. Gebruik lege regels om structuur aan te brengen, maar niet te vaak. Teveel lege regels maakt de code weer minder leesbaar. Goede plekken om een witregel in te voegen zijn bijvoorbeeld na een aantal declaraties en zowel voor als na for-loops en andere ingewikkelde constructies.

Vuistregel: *Met één of twee lege regels tussen blokken code, kun je duidelijk maken wat logisch bij elkaar hoort.*

Voorbeeldcode 6: Het gebruik van witruimte kan code leesbaarder maken.

(a) Zonder witruimte is het een warboel.

```
int faculteit(int getal) {
    int waarde = 1;
    for (int i = getal; i > 0; i--) {
        waarde *= i;
    }
    return waarde;
}
```

(b) Met witruimte zie je meteen de structuur.

```
int faculteit(int getal) {
    int waarde = 1;

    for (int i = getal; i > 0; i--) {
        waarde *= i;
    }

    return waarde;
}
```

---

3 Structuur

Dat programmacode complex kan zijn, is een gegeven; de Linux kernel bestaat uit meer dan vijftien miljoen regels code! Toch wordt die kernel zeer regelmatig aangepast en verbeterd door een heel aantal mensen, dus de complexiteit van de code is nog te overzien. Dat is mogelijk doordat de complexiteit van de code in de Linux kernel goed in bedwang gehouden wordt aan de hand van allerlei regels. We zullen een aantal belangrijke manieren bespreken om structuur aan te brengen in je programmacode als geheel

om zo de complexiteit onder de duim te houden. Veel van deze onderwerpen zijn iets geavanceerder dan simpele stijlregels en het kan daarom zijn dat je pas gedurende het vak echt gaat begrijpen wat de bedoeling is. Tot dan kun je het merendeel van dit hoofdstuk overslaan.

3.1 Code lengte

Je komt het overal tegen in het leven: complexe zaken die opgebroken zijn in kleine stukjes. Misschien is je favoriete televisieserie bijvoorbeeld opgedeeld in meerdere afleveringen. Dat maakt het niet alleen makkelijker om te produceren maar ook makkelijker om te kijken. Dit geldt ook voor programmeercode. Functies van minder dan 15 regels zijn vaak makkelijk te begrijpen en daarmee makkelijker om te schrijven en lezen. Door jezelf te dwingen kleine eenheden code te schrijven maak je je programma meestal modulairder en verhelp je veel van de hieronder genoemde problemen.

Vuistregel: *Houd functies in je programma onder de 15 regels en hanteer 30 regels als harde limiet.*

3.2 Modulariteit en interfaces

Als je later gaat samenwerken aan code kan het heel goed zijn dat er aan je gevraagd wordt om code te schrijven die één bepaalde taak uitvoert. Het is voor iedereen in die situatie makkelijk als je dat op een modulaire manier doet, dat wil zeggen op zo'n manier dat het makkelijk is om de code in een ander programma te zetten en ook om de code weer te verwijderen. Je kunt dat doen door je code te schrijven als (korte) methoden. Die kunnen dan in hun geheel toegevoegd worden aan de code zonder dat er veel knip-en-plakwerk aan te pas hoeft te komen. Een bijkomend voordeel is dat je je code op deze manier ook tussentijds kunt testen.

Een stapje verder zijn interfaces in Java. Dit zijn als het ware contracten die je aangaat waarin je belooft dat de klasse die je gaat programmeren bepaalde methoden met een bepaalde naam en een bepaald type zal hebben. Dit is een uitstekende manier om modulair te programmeren, omdat diegene die jou vraagt code te schrijven al van tevoren weet welke methoden hij of zij gaat krijgen en daarom kan inspelen. Het is dan ook van cruciaal belang dat je interfaces die je aangeleverd krijgt *niet* aanpast. Dit kan ervoor zorgen dat je code bij het nakijken niet meer compileert!

Vuistregel: *Breek code altijd op in logische eenheden en houd je altijd aan geleverde interfaces.*

3.3 Cyclomatische complexiteit

De cyclomatische complexiteit of McCabe complexiteit van een eenheid code is grof gezegd het aantal verschillende paden dat je kunt nemen door die eenheid. De cyclomatische complexiteit wordt voornamelijk verhoogd door het gebruik van if-else statements en kan zeer sterk toenemen als je gebruik gaat maken van conditionele statements binnen conditionele statements (*nesting*). Je vergroot de complexiteit van de eenheid code dan exponentieel.

We geven geen strikte waarden waaraan je je moet houden, maar probeer het aantal paden door een methode te beperken tot 10. Je hoeft dit niet steeds te tellen maar wees vooral voorzichtig met het gebruik van geneste conditionele statements. Je zult zelf ook merken dat dit het erg moeilijk maakt om zelf een idee te krijgen van hoe de uitvoer van de code gaat verlopen. Het verminderen van het aantal manieren waarop een methode doorlopen kan worden maakt het ook makkelijker om de code te testen.

Vuistregel: *Pas op met geneste conditionele statements.*

3.4 Separation of concerns en coupling

Als je bij een hotel komt om een kamer te huren, hoef je meestal niet zelf op zoek te gaan naar een beschikbare kamer; dat doet de receptie voor je. Het enige wat jij hoeft te doen is aangeven wat je wilt, en de tegenpartij doet het noodzakelijke achtergrond werk, die is daar immers voor aangenomen.

Wederom kunnen we dit concept goed inzetten bij het programmeren. In een objectgeoriënteerde taal zoals Java ga je veel werken met klassen. Die klassen zullen ook met elkaar interacteren. Het kan zijn dat je dan de neiging krijgt om variabelen in de ene klasse te veranderen vanuit de andere. Dit is echter verwarrend en niet modulair en DRY (zie verderop).

Zorg er daarom altijd voor dat je vanuit de ene klasse zo min mogelijk direct interacteert met een andere klasse. Dit noemen we ook wel *loose coupling*. De klassen zijn als het ware amper gekoppeld en daarom makkelijk te ontkoppelen – een eigenschap van een modulair ontwerp. Gebruik daarom altijd methodes van de andere klasse om ermee te interacteren. Ook als je een simpele verandering wilt maken aan een bepaalde waarde in een klasse gebruik je een setter omdat het kan zijn dat er bepaalde actie ondernomen moet worden wanneer de waarde veranderd wordt.

Vuistregel: *Gebruik in een klasse altijd zo min mogelijk details van een andere klasse. Laat een klasse zoveel mogelijk zichzelf beheren.*

Om jezelf te dwingen om klassen los te koppelen, gebruik je access modifiers zoals `public`, de standaard modifier, `protected` of `private`. Hoe strikter hoe beter. Maak dus niet alles publiek.

Vuistregel: *Gebruik altijd de meest strikte access modifier mogelijk.*

3.5 DRY

Als je trek hebt in spaghetti en je een recept opzoekt op internet, staat de tomatensaus meestal kant en klaar bij de ingrediënten. Dat is maar goed ook, want anders zou elk recept waar je tomatensaus voor nodig hebt, zelf een heel recept voor tomatensaus moeten bevatten! Dat zou het schrijven en lezen van recepten veel moeilijker maken en wat nou als het recept voor tomatensaus plotseling veranderd moet worden? Dan moet je elk recept aanpassen!

We kunnen dit concept wederom ook toepassen op het programmeren. Als je op meerdere plaatsen in je programma hetzelfde moet doen, is het niet handig om elke keer dezelfde code te schrijven (of te kopiëren)! Dat duurt langer, is moeilijker te lezen en ook moeilijker om achteraf aan te passen als er iets toch niet blijkt te kloppen.

Vuistregel: *Herhaal nooit blokken van meer dan twee regels code tenzij het echt niet anders kan.*

4 Naamgeving

Als er iets belangrijk (en moeilijk) is bij het programmeren, dan is het wel naamgeving. Je werkt vaak met veel verschillende variabelen, met ieder een ander doel. Het is fijn als je meteen van een variabele kunt zien wat het doel ervan is. Ook functies en methodes moeten een goede naam krijgen. Een goede richtlijn is om functies, variabelen en methodes een naam te geven die omschrijvend is. Een functie voor het berekenen van de som van twee getallen noem je '*som*' en niet '*bereken*'. Veel mensen proberen tijd te besparen door hele korte namen te gebruiken voor hun variabelen, maar dat maakt de code veel minder leesbaar. Kijk maar eens naar het volgende voorbeeld.

Vuistregel: *Duidelijke, leesbare namen maken de code veel begrijpelijker.*

Voorbeeldcode 7: Verschillende manieren om variabelen namen te geven.

(a) Onduidelijke variabele namen die je alleen kunt begrijpen als er commentaar bij staat.

```
int t; /* telefoonnummer */
int f; /* fax */
int m; /* mobiel */
```

(b) Variabele namen waaraan je direct kunt zien wat ze betekenen.

```
int tel_nr;
int fax_nr;
int mobiel_nr;
```

Zoals je ziet heb je bij het tweede voorbeeld geen commentaar meer nodig om de inhoud van de variabelen te omschrijven. Het is dus handig om omschrijvende namen te gebruiken voor variabelen. Hetzelfde geldt voor methodes en functies:

Voorbeeldcode 8: Verschillende manieren om methoden namen te geven.

(a) Methode met onduidelijke naam.

```
public static int bereken(int x) {
    return (x == 1) ? x : x * bereken(x - 1);
}
```

(b) Methode met duidelijke naam.

```
public static int faculteit_recursief(int x) {
    return (x == 1) ? x : x * faculteit_recursief(x - 1);
}
```

Uitzonderingen op deze regel zijn waardes die je gebruikt in een for-loop. Vaak hebben deze waarden geen doel naast dat ze tellen en noemen we ze i, j of k, afhankelijk van welke variabelen nog niet gebruikt zijn.

Vuistregel: *Gebruik namen die de bedoeling van de variabele, functie of methode duidelijk maken.*

Vuistregel: *Duidelijkheid is altijd belangrijker dan humor!*

5 Commentaar

Hoe netjes je code ook is, hoe duidelijk de namen van je variabelen ook zijn, er zullen altijd onduidelijke regels code in je programma zitten. Om die stukken code, die niet direct te begrijpen zijn, begrijpbaar te maken, gebruik je commentaar. Commentaar is een stuk code dat niet wordt uitgevoerd. Het vult de code aan door (kort) in menselijke taal uit te leggen wat de bedoeling is. Een goede richtlijn om aan te houden is om boven iedere methode, functie en klasse een stuk commentaar te zetten. In dat commentaar wordt de bedoeling beschreven van het volgende stuk code. Op die manier kan een specifieke functie snel worden gevonden.

Het is belangrijk om ervoor te zorgen dat commentaar de code echt aanvult. Het is, bijvoorbeeld, niet nuttig om voor de variabele `tel_nr` een stuk commentaar te schrijven om uit te leggen dat hier een telefoonnummer in wordt bewaard of dat het een integer is.

Vuistregel: *Zet boven ieder bestand, methode, functie en klasse een omschrijven commentaar.*

Vuistregel: *Gebruik commentaar om je code te verduidelijken.*

5.1 Verschillende soorten commentaar

In iedere programmeertaal moet je op een andere manier commentaar geven. Er zijn echter wel een aantal standaard methodes. Het geven van een enkele regel commentaar doe je in Java op een van de volgende manieren. Zet commentaar altijd op zijn eigen regel en nooit op dezelfde regel achter code. Dit ziet er zeer slordig uit.

Voorbeeldcode 9: Verschillende manieren om commentaar te geven.

(a) Enkele regel commentaar op de aanbevolen manier.

```
/* bereken het percentage mussen. */
percentage_mussen = (aantal_mussen / aantal_vogels) * 100;
```

(b) Enkele regel commentaar op de niet-aanbevolen manier.

```
// bereken het percentage mussen.
percentage_mussen = (aantal_mussen / aantal_vogels) * 100;
```

Bij inleiding programmeren wordt de voorkeur gegeven aan de eerste methode. Die methode staat namelijk toe dat een enkele regel gemakkelijk kan worden uitgebreid naar meerdere regels. Deze methode is vanzelfsprekend ook het handigst als je meer dan een regel commentaar wil toevoegen.

We raden aan om commentaar met '//' alleen te gebruiken om een regel tijdelijk 'uit' te zetten. Als je probeert een fout te vinden in je code, of als je het programma wil testen zonder een bepaalde functie, kun je die regels tijdelijk weg-commenten die je niet nodig hebt. Dit is de aanpak die vaak wordt gebruikt bij het programmeren. Je kunt bij het debuggen dan steeds een paar regels terugzetten totdat het weer fout gaat. Je weet dat waar de fout zit.

```
int aantal_vogels, aantal_mussen, percentage_mussen;

/* Bereken het percentage mussen. Deze waarde wordt later
 * gebruikt om af te leiden wat het percentage duiven is.
 */
percentage_mussen = (aantal_mussen / aantal_vogels) * 100;
```

Voorbeeldcode 10: Meerdere regels commentaar.

Verschillende vormen van commentaar door elkaar maakt de code erg onoverzichtelijk. Er volgen nu twee voorbeelden van code die op een onduidelijke en op een duidelijke wijze zijn voorzien van commentaar.

```java
public static String filterInput(String input)
{
    String schoneInput = ""; // String waar de opgeschoonde input in komt.
    int schoneTekens = 0; /* Om bij te houden hoeveel er moest
                             worden opgeschoond, tellen we hoeveel tekens
                             er worden veranderd.*/
    char huidigTeken; // Het huidige teken van de input.

    // Itereer over alle tekens in de input en kijk of een teken wel
    // of niet in het resultaat mag komen.
    for(int i = 0; i < input.length(); i++) {
        huidigTeken = input[i]; // Pak het huidige teken.

        /* Controleer of het huidige teken in het normale alfabet
        zit.*/
        if(huidigTeken >= 'a' && huidigTeken <= 'z') {
            schoneInput[schoneTekens++] = huidigTeken; /* Voeg het
            huidige teken aan het resultaat toe */
        }
    }

    return schoneInput;
}
```

Voorbeeldcode 11: Onduidelijk commentaar.

```java
public static string filterinput(string input)
{
    String schoneInput = "";
    int schoneTekens = 0;
    char huidigTeken;

    /* Itereer over alle tekens in de input en kijk of een teken wel
     * of niet in het resultaat mag komen.
     */
    for(int i = 0; i < input.length(); i++) {
        huidigTeken = input[i];

        if(huidigTeken >= 'a' && huidigTeken <= 'z') {
            schoneInput[schoneTekens++] = huidigTeken;
        }
    }

    return schoneInput;
}
```

Voorbeeldcode 12: Duidelijk commentaar

Vuistregel: *Commentaar moet de code aanvullen en niet triviaal op te maken zijn uit de code.*

Vuistregel: *Commentaar moet kort en bondig zijn, maar wel duidelijk.*

Vuistregel: *Gebruik bij voorkeur altijd de commentaarstijl voor meerdere regels, zelf al schrijf je er maar één.*

Vuistregel: *Zorg ervoor dat het commentaar in correcte taal is geschreven en geen spellingfouten bevat.*

6 Consistentie

Het belangrijkste,bij het schrijven van nette code is *consistentie*. Dat wil zeggen dat je steeds dezelfde stijl gebruikt. Welke stijl je ook aanhoudt, hoe groot je programma ook is, als je consistent bent is je code beter en prettiger leesbaar. Je kunt zelfs een eigen manier voor het inrichten van de code bedenken zolang je vervolgens bij alle programmacode dezelfde stijl gebruikt.

Vuistregel: *Welke programmeerstijl je ook gebruikt, consistent zijn is het belangrijkste.*

7 Conclusie

Ondertussen is duidelijk dat er zeer veel verschillende mogelijkheden zijn om programmacode netjes te maken. De richtlijnen en opties die hier zijn uiteengezet, zijn slecht een handvol van alle mogelijkheden. Neem vooral een keer de tijd om onderzoek te doen naar de stijl die jou het meeste aanspreekt. Het zal zeer zeker een waardevolle besteding van je tijd zijn, want nette code leidt tot programmacode die beter te begrijpen is. En nette code leidt weer tot betere programma's.

Als je meer wilt lezen over de verschillende code stijlen die je kunt gebruiken, bekijk dan de volgende websites:

- `http://en.wikipedia.org/wiki/indent_style`
- `http://java.sun.com/docs/codeconv/html/codeconvtoc.doc.html`
- `http://en.wikipedia.org/wiki/coding_conventions`
- `http://en.wikipedia.org/wiki/programming_style`

Succes met programmeren!