Goeie code schrijven

5.1 Inleiding

In dit hoofdstuk geef ik een aantal goeie programmeer gewoonten die de meeste programmeurs zouden beaamen als Goeie Principes, maar die toch door zo weinigen in de praktijk worden toegepast, ofwel uit onwetendheid, ofwel uit luiheid. Het zijn vaak heel triviale, voor de hand liggende dingen, maar ze kunnen wel je leven als programmeur een pak eenvoudiger maken, en de code die je produceert een pak beter.

Goeie code is code die leesbaar, onderhoudbaar en uitbreidbaar is, in die volgorde. Leesbare code is code waar je snel je weg in vindt, waar snel duidelijk is wat iets doet en wat er gebeurt. Code die niet leesbaar is, is moeilijk onderhoudbaar. Onderhoudbare code is code waar je makkelijk wijzigingen in kunt aanbrengen zonder dat de helft moet herschreven worden. Code die op zich al moeilijk onderhoudbaar is, is al zeker niet makkelijk uit te breiden. Uitbreidbare code is betrouwbare code waarop je kunt bouwen, die je makkelijk kunt integreren in een groter geheel.

De meeste tips en practices die volgen leren de meesten onder ons wellicht op de harde manier, wanneer je om 3u 's ochtends nog achter een bug zit te zoeken en je jezelf afvraagt wie die idioot is die deze rommel heeft geschreven, en je tot eigen scha en schande moet concluderen: ik.

5.2 Leesbaarheid

- **Gebruik zinvolle namen**

  Het basisidee is eenvoudig: schrijf code zo dat wanneer iemand ze leest die geen idee heeft van wat die code zou moeten doen, die persoon zo snel als mogelijk

begrijpt wat er gaande is. Eén manier om dit te bereiken zijn de namen die je aan je variabelen en methoden geeft. Kies altijd duidelijke en omschrijvende namen zodat meteen duidelijk is waarvoor die variabele of methode gebruikt wordt.

- Vermijd éénletterige namen zoals `a`, `b` en `c`, tenzij voor tellers in loops, waar je net wel `i`, `j` en `k` gebruikt omdat iedereen dat meteen herkent als een teller. Tellervariabelen zijn meestal niet de plaats om origineel te doen.
- Vermijd samentrekkingen zoals `krkts` of `numLetArb`, op het moment zelf misschien een goed idee maar achteraf of voor een buitenstaander absoluut een nachtmerrie.
- Vermijd namen die te abstract zijn, zoals `handleStuff` of `filterDeelB`.
- Maak namen niet te lang zoals `maxAliensOnScreenAtTheSameTime`. Je moet ze ook nog kunnen intypen.

Maar soms is het verschil tussen een goeie en een slechte naam natuurlijk minder duidelijk. In de code van het Hoger Lager spel had ik oorspronkelijk een variabele `aantalBeurten`. Maar over welk aantal gaat het? Het aantal gespeelde beurten, of het aantal resterende beurten?

Comment like a smart person

De bedoeling van commentaar schrijven bij code is om het achteraf makkelijker te maken om die code te begrijpen. Maar je hebt goeie en slechte commentaar, commentaar schrijven is een competentie waar je met tijd beter in zult worden.

Schrijf niet te veel. Commentaar moet de leesbaarheid verbeteren, maar té veel commentaar haalt de leesbaarheid weer onderuit. Veel beginnende programmeurs hebben de neiging om in overdrive te gaan als het op commentaar aankomt, en schrijven letterlijk overal commentaar bij, ook bij heel triviale dingen zoals:

```
// Now we increase numberAliensOnScreen by one
numberAliensOnScreen = numberAliensOnScreen + 1;
```

Als code op zich al duidelijk is, dan moet er geen commentaar bij. Spaar je ook weer een regel uit.

Schrijf ook niet te weinig. Als code te lang doorgaat zonder commentaar om de zaken wat onder te verdelen, dan is de kans dat het je achteraf veel tijd gaat

kosten om te achterhalen wat daar allemaal staat vrij groot. In methoden kan je meestal een aantal functionele blokjes onderscheiden, gebruik commentaar om dat onderscheid te verduidelijken.

- **Methodenamen moeten de lading dekken**

  Een voorbeeldje uit het leven gegrepen:

```
public void vraagInput() {
  System.out.print("IN>");
  String foolingave = Input.readString();
  foolingave.trim();
  ingave = foolingave.toUpperCase();
  krktrs = ingave.length();

  if (ingave.equals("INFO")){
    toonInfo();
  }
  else if(ingave.equals("STOP")){
    System.exit(0);
  }
  else{
    naamgevingKarakters();
    bepaalCijferstelsel();
  }
}
```

  Een student noemt z'n methode `vraagInput`, maar wanneer je die methode bekijkt doet die naast het vragen achter input - wat je toch zou verwachten van een methode `vraagInput` - ook nog verwerk input en bepaal op basis van die input wat er vervolgens moet gebeuren. Dit soort code zet je een goed eind op weg op het pad naar ondoorgrondelijke code. Let trouwens op de geweldige variabele `foolingave` (Huh?) en de methode `naamgevingKarakters` (Ben benieuwd wat die zou doen?).

- **Gebruik constanten**

  Gebruik nooit letterlijke waarden in je code, tenzij voor de 0 waarop de teller start in je `for` lus. Volgend stukje code illustreert mooi hoe belangrijk constanten kunnen zijn om de leesbaarheid van code te verbeteren:

```
public double[][] aanmakenRijenKeuze2(...) {
  // Instantiatie van de 2-dim. rij voor type1
  double [][]rijKeuze2 =
    new double[aantalMaanden+1][5];
```

```
    // De elementen opvullen
    for (int i = 0; i < aantalMaanden+1; i++) {
      // Maand
      rijKeuze2[i][0] = i;
      // Intrest
      rijKeuze2[i][3] = (i<1 ? 0 :
        rijKeuze2[i-1][1]*(maandRentevoet/100) );
      // Maandelijkse kapitaalsaflossing
      rijKeuze2[i][2] = (i<1 ? 0 :
        rijKeuze2[i-1][4] - rijKeuze2[i][3]);
      // Resterend saldo
      rijKeuze2[i][1] = (i<1 ? teLenenBedrag :
        rijKeuze2[i-1][1]-rijKeuze2[i][2] );
      // Totale aflossing
      totaalRijKeuze2[4] += (i<1 ? 0 :
        rijKeuze2[i][4]);
      // Totale intrest
      totaalRijKeuze2[3] += rijKeuze2[i][3];
      // Totale kapitaalsaflossing
      totaalRijKeuze2[2] = totaalRijKeuze2[4] -
        totaalRijKeuze2[3];
    }
    return rijKeuze2;
}
```

Er schort vanalles aan dit stukje code, rechtstreeks uit een project Java Basisconcepten geplukt trouwens. Ten eerste is er de naam van de methode: los van de context (en eigenlijk zelf al ken je de context) heb je geen idee wat een methode `aanmakenRijenKeuze2` zou kunnen doen. Die naam zegt je helemaal niets, en is dus slecht gekozen. Ten tweede zijn er constante waarden 0, 1, 2, etc. waarmee de array `rijKeuze2` wordt geïndexeerd. Waarom hier geen constanten voor gebruiken, dan is het meteen duidelijk op welke posities in de array welke waarden worden bewaard. En te slotte de commentaar: als code helder is geschreven wordt commentaar voor een groot stuk overbodig. Door constanten te gebruiken spreekt de code voor zich en kunnen we de commentaar, zonder afbreuk te doen aan de leesbaarheid, weglaten:

```
for (int i = 0; i < aantalMaanden+1; i++) {
  rijKeuze2[i][MAAND] = ...;
  rijKeuze2[i][INTREST] = ...;
  ...
}
```

En nu nog een passende naam voor de array zelf (`rijKeuze2`?) en we kunnen beginnen...

- **Probeer niet té slim te zijn**

  Je kan programma code zó nifty maken dat je er de precedentieregels voor operatoren moet bij nemen of obscure hoofdstukken achteraan je Java boek moet raadplegen om te begrijpen wat er staat. Hou het syntactisch simpel. Code zoals volgt (welliswaar geen Java maar C) kom je liever niet tegen:

```
main(v,c)char**c;{
  for(v[c++]="Hello, world!\n)"; (!!c)[*c]&&(v↘
    →--||--c&&execlp(*c,*c,c[!!c]+!!c,!c)); **c↘
    →=!c)write(!!*c,*c,!!**c);
}
```

- **Volg stijlregels**

  Elke programmeertaal heeft zo z'n stijlregels, helaas verschillen die nogal van taal tot taal. Maar dat neemt niet weg dat je die regels best kunt volgen. Stijlregeles zorgen er voor dat je je sneller thuis voelt in andermans code, er is immers weinig code die voor z'n volledige levensduur door één en dezelfde persoon wordt onderhouden. Stijlregels verhogen de leesbaarhied van code enorm. Programmeerstijl is geen originaliteitswedstrijd. De stijlregels voor Java kan je o.a. op `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html` raadplegen.

5.3 Onderhoudbaarheid

- **Schrijf minder code**

  Hoe minder code je hebt, hoe minder code je moet onderhouden (*complexity is the no. 1 enemy of maintainability*). Er zijn natuurlijk grenzen aan wat je doet om code beknopter te maken, maar algemeen gesproken is minder code om hetzelfde te doen beter. Het vergt vaak wat meer ervaring om goeie beknopte code te schrijven die nog steeds goed leesbaar is. Gebruik codebibliotheken waar mogelijk (bv. `Arrays.sort` gebruiken i.p.v. het zelf te schrijven). Wanneer methoden of klassen te lang dreigen te worden moet je de reflex hebben en jezelf de vraag stellen of je bepaalde code niet kunt opsplitsen. En vooral: leer van goeie voorbeelden.

- **Herschrijf indien nodig**

  Heel veel code ontstaat als volgt: je denk even na over het probleem, je zet de grote lijnen uit van hoe je het gaat aanpakken, en je werkt de code uit. Dan

komt vaak het moment waarop blijkt dat er nog zaken ontbreken, en begin je voort te borduren op wat je hebt. Maar op een bepaald moment kan je op het punt komen dat je code meer op gefoefel begint te kijken, dat je code van ver niet meer lijkt op de code die je had geschreven als je je design van in het begin goed had gedaan. Ook al is het op korte termijn veel werk, herschrijf je code vanaf nul voor je verder gaat, je zult het jezelf niet beklagen.

5.4 Uitbreidbaarheid

- **Wantrouw parameters**

  Een noodzakelijke voorwaarde voor uitbreidbaarheid van code is dat ze betrouwbaar is. De bouwstenen waarmee je een applicatie bouwt moeten solide zijn, anders krijg je gegarandeerd gedonder. En betrouwbaarheid begint met de parameters van je methoden. Je hebt geen idee van waar de data komt die je binnen krijgt, vertrouw er dan ook niet op dat het bruikbare waarden zijn. Bij parameters van een complex type, zijn het geen `null` referenties? Zijn parameters van het type `Object` instanties van de klasse die je verwacht? Liggen getallen binnen de toelaatbare grenzen? *Never trust user input!*

- **Gebruik exception handling**

  Gebruik exception handling waar nodig, bv. wanneer je een string naar een getal converteert of wanneer je uit een tekstbestand leest. Raadpleeg steeds de documentatie om te weten welke methoden mogelijks excepties opgooien.

- **Gebruik access specifiers**

  Wat tot de interne keuken van je klassen behoort maak je `private`, zo eenvoudig is het. Denk bv. aan de `load` en `save` methoden uit de klasse `Highscore`.

- **Let op met strings**

  Veel van je code zal zich in de logica laag van je code bevinden. Volgende methode is daar totaal onbruikbaar:

```
public void controleerStelsel(int stelsel) {
  if (stelsel == 0) {
    // doe ites
  }
  else if (stelsel == 1) {
    // doe iets anders
  }
```

```
    else {
      System.out.println("Fout!");
    }
  }
```

Het is niet aan deze methode om te bepalen wat de output moet zijn i.g.v. een fout, het is aan de code die de gebruikersinteractie verzorgt om te bepalen wat de reactie moet zijn, en in welke taal. Bovendien is output gegenereerd met `System.out.println` in een Swing applicatie gewoon niet zichtbaar.

Niet dat volgende constructie veel beter is:

```
public String controleerStelsel(int stelsel) {
  ...
  else {
    return "Fout!";
  }
}
```

Voor foutcodes gebruik je geen strings maar getallen, en van die getallen maak je constanten.

- **Documenteer je code**

  Als je je code ook maar een beetje kans wil geven op een lange levensduur, voorzie documentatie. Niets zo ergerlijk als ongetwijfeld geweldige code op internet vinden die niet gedocumenteerd is. Want op den duur ben je langer bezig met die code uit te vlooien dan ze gewoon zelf te schrijven.

Tot slot nog een leuk artikel: *How to Write Unmaintainable Code.*
`http://www.freevbcode.com/ShowCode.Asp?ID=2547`