VAKMANSCHAP OF STIJLQUEESTE?

In het eerste deel van deze rubriek heb ik betoogd dat 'software beter maken' hard nodig is. Door aan ons imago te werken en door ons 'ingenieurschap' uit te buiten. Maar ook door betere software te maken. In deze aflevering onderzoeken we wánneer software beter is. En waarom we wellicht een stukje persoonlijke vrijheid moeten inleveren voor betere code.

Wat is beter?

We beginnen met een eenvoudig experiment: Neem twee verschillende, werkende programma's van beperkte omvang. Het maakt niet uit welke programma's het zijn, wie ze gemaakt heeft, of wat ze doen; als ze maar geschreven zijn in een taal die je goed beheerst. Mijn eerste vraag is eenvoudig: Welk van deze twee is het beste? Laten we de betere van die twee 'B' noemen, zodat 'A' minder goed is. Dan kunnen we verder gaan met het experiment. Ik wil je vragen om de code van 'A' te verbeteren zodanig dat het beter is dan 'B'. Dat verbeterde programma noem je 'C'. Nu komt het lastigste deel van het experiment: je hebt nu twee programma's die exact hetzelfde doen, maar waarvan de ene beter is dan de andere. Mijn vraag is: "Waarom is het beter"?

Testen baat niet

Het kan niet zo zijn dat een programma beter is, louter omdat het andere functionaliteit biedt. Want 'C' is gebaseerd op 'A' en afgezien van de code is er niets veranderd. Natuurlijk kan je een programma verbeteren door extra functionaliteit te bieden, maar dat is eigenlijk iets anders. Als twee programma hetzelfde doen, kun je dus niet testen welke het beste is.

De keuze van variabele- en functienamen, de structuur van het programma en gebruikte taalconstructies hebben weinig effect op de werking van het programma., maar zijn wel bepalend voor de code-kwaliteit. Duidelijke namen zorgen voor leesbare code; terwijl diepgeneste programmapaden de werking van het programma verhullen. Bovendien wordt het programma slecht testbaar.
Is het daarom niet raar dat software 'goed'gekeurd wordt als de functionaliteit werkt? Dat zegt toch niets over de code-kwaliteit? Hoewel dat de dagelijkse praktijk is, is een test is niet bedoeld om een uitspraak te doen over de 'goedheid' van code. Daarvoor moeten we de code bekijken; naar de lay-out, naar de hoeveelheid commentaar, etc. Dat zijn de zaken die de leesbaarheid en de begrijpbaarheid bepalen! Soms wordt dit aspect laagdunkend 'stijl' genoemd. Vaak ziet men dit als de persoonlijke vrijheid van elke software ingenieur. Alsof het niet belangrijk zou zijn.

Een interessante, nieuwe manier om heel snel veel code te bekijken maakt gebruik van 'fingerprints' of 'signatures'. Met een paar regels awk of perl kun je code scannen op bekende constructies. Zoals alternatieve paden (if-then-else), commentaarblokken, functies en lussen. Maar ook eenvoudige zaken als een enkelvoudig statement (scan op ';'). Elk keer als zo'n constructie gevonden wordt, wordt één letter geprint; bijvoorbeeld een '{' bij het begin van een conditioneel codeblok (if/then, else, lus), een '}' bij het einde daarvan en een punt voor een elk statement. Door elke functie te beginnen op een nieuwe regel (print '\n':-) krijg je een fingerprint waarin heel eenvoudig te zien is hoeveel functies aanwezig zijn en hoe lang elke functies is. Dat geeft een snelle indruk van de kwaliteit van de code!

Zo zijn extreem lange functies snel te vinden. Ook over-complexe functies, met te veel geneste 'if-then-else' of lussen zijn makkelijk te herkennen. Gewoon haakjes tellen. Een eenvoudige vuistregel is dat meer dan 3 à 4 open haakjes aandacht vragen.

In een experiment ben ik nog verder gegaan. Door de fingerprint om te zetten in html, opnieuw te scannen en bekende of gevaarlijke sequenties in te kleuren met css, kon ik controleren of alle Makefile van 'EQSL' een gelijke opbouw hebben. En zo het gebruik van EQSL verder vereenvoudigen.

Stijl is vakmanschap

Software wordt keer op keer veranderd. Omdat requirements veranderen, om bugs op te lossen, of omdat we 'oude code' hergebruiken. Eén kleine, lokale verandering kan een groot, globaal effect hebben. Maar is elke verandering een bugfix? Een kleine verandering kan ook negatief uitpakken; iedereen kent wel een voorbeeld van een 'verfoutering'.

"Zet twee programmeurs bij elkaar en ze zijn het oneens", zou een volkswijsheid kunnen zijn. Zeker als je ze vraagt naar de correcte plaats van regelovergangen, haakjes of het benodigd aantal spaties. Terwijl dat details zijn die de compiler niet ziet, die de C-taal niet voorschrijft en nauwelijks aan bod komen in opleidingen. Waarom vinden programmeurs stijl belangrijk genoeg om eindeloos over te discussiëren?

Een verstandig gebruik van haakjes, correct inspringen en al die andere zaken die geen direct effect hebben op de betekenis van de code zijn belangrijk omdat ze de codekwaliteit, de 'goedheid' van de code, bepalen. Ze maken code leesbaar en begrijpbaar. En zo heeft de stijl van de code ook indirect effect op de kwaliteit van het product. Eén slordig ingesprongen regel heeft nu geen effect. Maar het kan iemand op het verkeerde been zetten; een verkeerde verandering is dan snel gemaakt.

Daarom is stijl belangrijk. Voor de werking van de eerste versie maakt de stijl niet uit. Maar daarna zal de code veranderen. Dan kunnen fouten gemaakt worden. Die volgende programmeur zou de code verkeerd kunnen lezen, te veel afgaand wat de stijl suggereert. Zo heeft de stijl van deze versie, effect op de werking van de volgende versies!

Is mijn stijl echt beter?

Het is dus belangrijk om een goede stijl te gebruiken. Maar hoe doen we dat? Er bestaan veel bekende stijlen, zoals 'K&R', 'GNU', en 'Stroustrup' en talloze variaties. Er is zelfs een Linux-stijl! Elke software ingenieur heeft zijn eigen voorkeur. Maar waar is die voorkeur op gebaseerd? En is die voorkeur belangrijker dan de noodzaak om software beter te maken?

Zo ben ik een voorstander van braces op dezelfde regel als het voorafgaande statement. Waarom? Omdat het een programma korter maakt! En korte programma's zijn overzichtelijker, beter te begrijpen en dus beter. Bovendien blijkt dat een programmeur typisch één schermvol code per dag schrijft. (Ja, zo weinig!) Dit is een gemiddelde en als je de tijd voor debuggen, testen en ontwerpen meeneemt. Bovendien onafhankelijk van de schermgrootte. Een groter scherm maakt programmeurs effectiever! Een groter scherm is daarom een goedkope manier om al gauw 10% efficiënter te zijn. Alleen een stijl die bondige, maar leesbare programma's oplevert, zoals mijn keuze voor de plaats van braces, is nog goedkoper.

Maar als iedereen zijn eigen stijl gebruikt dan zal de code niet beter worden. Het mengen van stijlen maakt het nooit beter. We zullen één stijl moeten kiezen. Daar geef ik mijn voorkeur graag voor op. Of je moet mij kunnen overtuigen dat jouw stijl betere code oplevert.

Terug naar het experiment. Je hebt drie programma's van oplopende 'goedheid': A, B en C. Ik geloof best dat je nog een 'D' kunt maken door B te verbeteren met jouw stijl. Maar hoe zit het met 'E' en 'F'? En is nog betere code mogelijk? Of moeten we dan eerst en betere stijl aanleren?

ALbert Mietus

Redundantie

Elke moderne editor kan automatisch inspringen. Dat is niet alleen makkelijk, het zorgt ook voor betere code. Het geeft namelijk feedback. Want als de editor een tab 'vergeet' dan betekent dat meestal dat er ergens een haakje mist. Met consequent inspringen bouw je redundantie in, waar je editor gebruik van maakt om aan te geven dat de formele en informele 'flow' niet overeen komen.
Hiervoor moet er een formele afspraak zijn welk stijl gebruikt wordt; zodat die "geprogrammeerd" kan worden in de editor. Kies daarom ook een editor die stijlen ondersteund. Of een stijl die ondersteund wordt door je editor.

Python

De discussies over witruimte zat? Kijk dan eens naar de programmeertaal 'python'! In die moderne, populaire taal worden codeblokken niet aangegeven met door haakjes ('{'...'}' zoals in C) of met BEGIN/END (pascal), maar met inspringen! Na een if-statement wordt ingesprongen. En elke regel die (evenveel) is ingesprongen hoort bij het 'then' deel!

Behalve om stijldiscussies te vermijden, is 'het (beeld)scherm efficiënter gebruiken' een belangrijk argument voor Guido van Rossum - de (Nederlandse) bedenker van deze taal. Het helpt de taal compacter te maken. Python programma's zijn typisch veel korter dan hun C-tegenhanger.

Voor meer info: http://www.python.org/

Testen of meten?

Laat er geen misverstand zijn; testen is belangrijk. Maar te vaak wordt te beperkt getest. De meeste testen meten de functionaliteit van het product. Andere metingen zijn vaak proces-gerelateerd; zoals de 'voortgang' en een 'bug-count'. Maar er zijn ook andere metingen: De bekendste is een 'code-inspectie', een dure en daarom weinig populaire kwaliteitsmeting.
Een heel eenvoudige meting is een 'line-change-count'. Bij het inchecken van code wordt die vaak gemeld. Daarmee kan ik snel zien of een kleine verbetering inderdaad slechts een kleine verandering was. Geen heel sterke meting, maar wel een heel goedkope; dus waarom zou je hem niet gebruiken? Ik heb er al een paar keer een blunder mee kunnen terugdraaien.

Iets als de 'cyclomatische complexiteit' is veel complexer. Maar met de juiste tools goed te meten. Het is een meting die aangeeft hoe diep de code genest is. Want elk 'if-statement' maak het aantal mogelijke paden groter; het nesten van 'if's' verdubbelt het aantal zelfs. Daardoor kan het onmogelijk worden om code volledig te testen. Voor meer info zie: http://albert.mietus.nl/read.IT/ToolView/metre.pdf

Zo zijn er talloze metingen mogelijk. Ze kunnen (functioneel) testen nooit vervangen, maar wel aanvullen en effectiever maken!