DE COMPUTER ANALYSEERT AUTO

In onze zoektocht om software beter te maken hebben we vorige keer betoogd dat stijl belangrijk is. Stijl gaat immers over vakmanschap; stijl maakt code leesbaar en helpt bij het onderhoud van programmatuur. Maar wat als dat niet gebeurd is, wat als de code 'onleesbaar' is? Dit keer gaan we daarom een stap verder. We gaan we kijken hoe we code sneller kunnen lezen. Bijvoorbeeld door de computer te gebruiken.

Software engineering wordt vaak als creatief beroep gezien. Het is immers de bedoeling dat we iets nieuws maken. Maar, maken we altijd iets nieuws? Ik denk het niet. Zoals we vorige keer gezien hebben veranderen we code vaker dan dat we iets nieuws maken! Naast de creatieve discipline om code te maken is ook 'software analyse' – ***het betere code-browsen*** – een wezenlijk onderdeel van ons vak.

Natuurlijk, in de ideale wereld is er een actueel ontwerpdocument, zijn er uitsluitend ervaren ontwikkelaars, werkt de code altijd en in één keer, en snappen we de code direct. Maar wat te doen als we gewone stervelingen zijn? Het lezen van andermans code is lastig: er is altijd te veel code om te lezen en er is altijd haast! Ooit was het modern om alle code te printen en die listings te archiveren. Zo was er altijd een basis om te bestuderen. Maar die tijd is voorbij.
Een beetje project bestaat tegenwoordig uit duizenden pagina's code. 'Gewoon doorlezen' kan niet meer; het moet sneller en slimmer. Toch wordt nauwelijks onderwezen hoe we code moeten lezen; laat staan dat we weten hoe we code 'sneller' kunnen lezen. Als we dat weten kunnen we die kennis bovendien gebruiken bij het schrijven van de code, zodat die weer het 'snelste' te lezen is.

Er zijn zo minder regels om te lezen. Toch zullen we nog steeds, als we niet alles kunnen onthouden, veel moeten scrollen en vaak van file moeten wisselen. Kunnen we dat niet automatiseren? Immers, een computer kan veel sneller zoeken dan wij. En heeft een veel beter geheugen. Bovendien is elke programmeurtaal per definitie begrijpbaar voor een computer. Zou het niet mogelijk zijn om de computer te vragen om de code te analyseren?
Dit idee is al oud. Zoals ik ooit besproken heb ik de 'cscope'-ToolView is het eenvoudig om snel door veel code te navigeren. Daarmee kunnen we kris-kras door honderden files springen. Om een thread-flow te volgen, om uit te vinden waar een functie aangeroepen wordt, of om snel een data-type-definities op te zoeken.

Dit maakt het lezen aan eenvoudiger. Maar het blijft behelpen. Tekst

Een goed begin

Al is er geen kant en klare oplossing, er zijn diverse tools die gecombineerd erg behulpzaam zijn bij een goede analyse. Diverse tools zijn al een keer besproken in een ToolView (zie http://albert.mietus.nl/read.IT). Hier een opsomming van handige combinaties:

- cflow & graphviz & scripting (Zie illustratie)
  Om snel een plaatje te krijgen van alle functie aanroepen Als nadeel kan opgemerkt worden dat het plaatje vaak erg groot wordt; printen op (meerdere) A3tjes helpt. Bovendien herkent cflow geen (indirecte) functie-pointer-aanroepen.
- ncsl & scripts
  'ncsl' telt Non-Commented-Source-Lines, commentaarregels, combinaties daarvan en blanco regels. Helaas alleen per file, niet per functie. Met wat extra scripting is dat wel te doen; bijvoorbeeld vanuit Emacs. Ook het output formaat is niet optimaal. De source is beschikbaar, maar mag officieel niet aangepast worden.
- (x)tags & tags2stat.xslt
  Alternatief om regels code te tellen: telt wel per functie maar telt minder nauwkeurig.
- 'gcc –E ' & grep & wc
  Een derde manier om regels te tellen: telt regels zoals de compiler het ziet. Geeft soms verrassende verschillen!
- gcc-xml & XSLT & Wrapper-code
  Legt een schil om bestaande (gecompileerde!) code heen om elke aanroep te loggen. Geeft dus een dynamische (runtime) analyse. Erg krachtig, maar wat lastig in gebruik.
- 'gcc –E' & signature-awk-script
  Berekent de fingerprint van een functie. Erg krachtig qua concept, maar het script is nog niet zo uitgewerkt als zou moeten.

Het nadeel van al deze onderdelen is dat een framewerk ontbreekt. De resultaten zijn dus nog niet (automatisch) combineerbaar. Ik ben ooit begonnen met een framework (in xml formaat), maar dat is nog niet af.

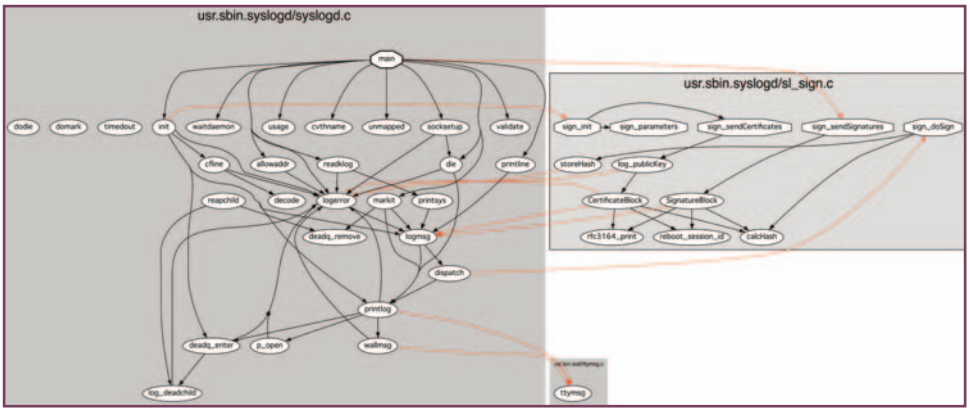
N.B. Als je zelf ook 'gouden combinaties hebt' liefst al aan elkaar gescript, laat het me weten. Ik wil ze graag verzamelen en distribueren.

Analyseren of beoordelen?

Naast de analyse tools zoals bedoeld in dit artikel zijn er ook andere soorten tools die (C) code kunnen analyseren. Denk aan (het commerciële tool) "QAC" of "SPlint" (OpenSource). Dat zijn prima tools om de kwaliteit van code te verbeteren. Ze beoordelen code, aan de hand van opgegeven of ingebouwde criteria. Een bekend (meestal) onjuist codefragment als "if (a=0) ..." zal gesignaleerd worden; evenals vele andere.
Op die manier kunnen ze helpen om code, die gecreëerd wordt beter te maken. Ook beter leesbaar. Maar uitsluitend tijdens de 'synthese-fase'.

Op het moment dat bestaande code veranderd moet worden, zijn deze tools minder nuttig. Eventueel kunnen ze nog van pas komen om 'buggy code' te vinden. Wel helpen ze bestaande code inzichtelijk te maken. Dan willen we namelijk niet weten 'hoe goed' de code is maar 'hoe deze werkt'. Een oordeel is dan niet gevraagd; we hebben tools nodig die ons helpen de code te analyseren.

Dit (vrijwel) geheel automatisch gegenereerde diagram laat alle functies, in drie files, van syslogd zien. Een deel van de code is oude, bestaande code. Een andere file is een moderne uitbreiding. Kun je het verschil zien?



is nu eenmaal niet de meest toegankelijke vorm van informatie. Grafische omgevingen hebben hun nut al bewezen aan de synthese kant. Zelfs als die IDE's als Visual Studio en Eclipse slechts gebruikt worden als luxe editor helpt het al om de productiviteit te verhogen. Ik denk dat we ook grafische tools nodig hebben voor effectieve software-analyse. Al zijn ze er nauwelijks.

Toch zijn er mogelijkheden als we bereid zijn om bestaande tools zelf aan elkaar te scripten.
Een goed begin is: "Doxygen". Deze bekende documentatiegenerator kan ook gebruikt worden om code te analyseren. Het kan bijna alle populaire talen lezen en relaties tussen (geïnclude) files tekenen. Bovendien geeft het 'klikbare' opsommingen van alle functies, hun parameters en andere belangrijke zaken. Dat zijn zaken die zeker bruikbaar zijn, voor de beginnende software-analist.

Voor een diepgaande analyse volstaat doxygen niet, al wordt het steeds krachtiger. Het ultieme tool laat niet alleen relaties zien, maar geeft veel meer relevante informatie. Eenvoudige statische gegevens als de omvang van alle functies, liefst uitgesplitst naar code, commentaar en lege regels, zijn onontbeerlijk. Maar ook de 'fingerprint' van een functie geeft veel informatie. Want zijn alternatieve paden en lussen niet belangrijke zaken om een programma te begrijpen?
Bij gebrek aan beter, grep ik soms op de reguliere expressie '; *;' om de mainloop te vinden.... Dat moet toch automatisch kunnen?

Inmiddels heb ik een behoorlijke set van handigheidjes die samen een aardig beeld geven. In de kadertekst laat ik een aantal van die combinatie zien. Het lukt nog niet om automatisch een totaalbeeld te krijgen; elke keer blijft het handwerk, dat slechts deels geautomatiseerd is. Wat er ontbreekt ,is een framework om alle informatie in op te staan en te combineren. En de tijd om alle onderdelen te integreren. Het ultieme tool is er dus nog niet.

Toch blijkt de aanpak te werken. Het computerondersteund snellezen van code geeft meer inzicht en geeft sneller inzicht.
Ooit schrijf ik nog dat ultieme tool. Het tool dat al die delen integreert. Het tool dat mij heel snel vertelt hoe software werk. Ooit is het af! De ultieme test is het tool loslaten op zijn eigen code. Zal het zichzelf herkennen?

ALbert Mietus