De software-evolutieparadox

Intreerede

In verkorte vorm uitgesproken
op 23 februari 2005
ter gelegenheid van de aanvaarding van het ambt van
hoogleraar software engineering
aan de faculteit van
Electrotechniek, Wiskunde en Informatica
van de Technische Universiteit Delft

door

Arie van Deursen

Afbeelding op de voorzijde: detail van een reproductie van een muurmozaiek uit het mausoleum van Galla Placidia (386–452) te Ravenna.

Mijnheer de Rector Magnificus,
Leden van het College van Bestuur,
Collegae hoogleraren en andere leden van de universitaire gemeenschap,
Zeer gewaardeerde toehoorders,
Dames en heren,

1 Enkele vragen vooraf

Welke technologie is nodig om goede software te maken? Wat is goede software? Hoe maken we software? Kan het beter? Kan het goedkoper? En hoe belangrijk is het maken van software eigenlijk?

Vanmiddag ga ik met u proberen dit soort vragen over het maken van software ("software engineering") te beantwoorden. Daarbij wil ik u iets vertellen over het soort onderzoek dat ik met mijn groep doe, waarom ik dat doe en ga ik zelfs proberen u mee te nemen naar enkele recente onderzoeksresultaten. Bovendien wil ik u een beeld schetsen van wat ik samen met mijn collega's de komende jaren denk te gaan doen op het terrein van de software engineering.

Maar voordat ik daarmee begin wil ik u een aantal vragen meegeven die u zelf kunt proberen te beantwoorden, desnoods door een antwoord te gokken. Nu zitten er veel verschillende mensen in de zaal, van computer-leek tot super-informatica-expert, van scholier tot professor, en daarom heb ik verschillende soorten vragen, met hopelijk voor elk wat wils.

Allereerst drie vragen voor iedereen in de zaal.

1. Met hoeveel software-systemen bent u vandaag in aanraking gekomen? Probeer de dag af te lopen en vraag bij elke activiteit of er wellicht software voor nodig was.[1]

2. Met hoeveel nieuwe versie of updates van de software-systemen waar u mee werkt heeft u direct te maken gehad?

3. Hoeveel wijzigingen zou u zelf willen voorstellen om deze systemen prettiger in het gebruik of in ander opzicht beter te maken?

De antwoorden op deze vragen leveren voor iedereen drie getallen op, die u bijvoorbeeld tijdens de receptie kunt vergelijken met die van uw buurman of buurvrouw.

[1] Zie ook het voorwoord van Campbell-Kelly's geschiedenis van de software-industrie (Campbell-Kelly 2003).

Daarnaast heb ik vier vragen voor de verschillende soorten informatici in de zaal.

1. Voor software engineers uit de praktijk:

   Wat zijn de belangrijkste technische belemmeringen voor het in korte tijd doorvoeren van wijzigingen in uw software-systemen?

2. Voor studenten:

   Welke kennis denk je nodig te hebben zodat je daadwerkelijk kunt bijdragen aan het maken en wijzigen van succesvolle software-systemen?

3. Voor docenten:

   Hoe moet het universitaire informatica-onderwijs in het algemeen en het software engineering onderwijs in het bijzonder worden ingericht om recht te doen aan het belang van aanpassingen aan software-systemen?

4. Voor onderzoekers:

   Hoe kan een software engineering onderzoeker de waarde van zijn voorgestelde nieuwe technieken en theorieën objectief vaststellen? Met andere woorden: hoe evalueer je de geldigheid van software engineering onderzoeksresultaten?

Genoeg stof tot nadenken. Ik zou er dus goed aan doen nu een poosje mijn mond te houden, en vervolgens voldoende tijd per vraag uit te trekken om de verschillende antwoorden uit de zaal te bespreken.

Helaas gaat dat niet lukken in de 30 minuten die ik heb. In plaats daarvan ga ik u het verhaal vertellen van mijn onderzoek en nodig ik u uit daarnaar te luisteren met de zojuist gestelde vragen in uw achterhoofd.

2 Software-evolutie

Onze beschaving draait op software. Althans, dat stelt Bjarne Stroustrup,[2] de ontwerper van de programmeertaal C++. Hoe langer u nadenkt over uw antwoord op mijn eerste vraag — met hoeveel software-systemen u vandaag te maken heeft gehad — hoe meer u het met Stroustrup eens moet zijn. Een beetje CV-installatie, thermostaat, wasmachine, droger, afwasmachine, radio, televisie, of telefoon zit vol met software. En als u gewoon iets contant betaalt

[2] "*Our civilization runs on software*", geciteerd door onder meer Booch (Booch 2004).

in de winkel dan rekent de kassa uit wat u moet betalen, wat uw wisselgeld is, en hoeveel geld er in kas zit, waarna dat geld weer terugkomt in het boekhoudprogramma van de winkelier, in de software die het betalingsverkeer regelt, en in de bancaire software om rente uit te rekenen, afschriften te versturen, enz.

Al deze software moet gemaakt worden en dat gebeurt door software engineers. Wereldwijd zijn er daar momenteel zo'n 15 miljoen van (Booch 2004). Zij werken niet alleen bij typische informatica-bedrijven, maar in vrijwel alle sectoren. Zo heeft autofabrikant Ford 11.000 informatici in dienst. Er wordt geen product meer gemaakt of er komen software engineers aan te pas.

Software moet echter niet alleen gemaakt, maar vooral ook veranderd kunnen worden. Als u zelf gebruik maakt van software zoals Word, Windows XP of Linux, zal het u niet ontgaan zijn dat er met de regelmaat van de klok nieuwe versies verschijnen. Hiervoor zijn een aantal redenen aan te wijzen.

- Een eerste is dat de oude versie fouten zal bevatten, die verbeterd zouden kunnen zijn in de nieuwe versie.
- Een tweede reden is dat de wereld waarin de software draait verandert, bijvoorbeeld door standaardisering, technologische innovaties zoals mobiele telefonie, veranderende wet- of regelgeving, strengere veiligheidseisen als reactie op terrorismedreiging, enz.
- Een verdere reden is dat de eerdere versies van een software-product vaak bewust eenvoudig worden gehouden, bijvoorbeeld om snel een product op de markt te kunnen zetten, of omdat men zich moeilijk een voorstelling kan maken van de manier waarop het product precies gebruikt zal gaan worden.

In al deze gevallen moet de software aangepast worden. Het verschijnsel dat programma's steeds maar aan verandering onderhevig zijn wordt *software evolutie* genoemd. Het verschijnsel trok in de jaren zeventig de aandacht van Belady en Lehman, twee onderzoekers van IBM die werkten aan een groot (5000 mensjaar) en legendarisch project, de ontwikkeling van het besturingssysteem voor de IBM 360 mainframes (Belady en Lehman 1976).

Op basis van dit project formuleerden zij een aantal karakteristieken van programmatuur die zij formuleerden als *wetten van de software-evolutie* (zie ook Figuur 1). Hun belangrijkste observatie was dat een software-systeem dat werkelijk gebruikt wordt een voortdurend proces van verandering ondergaat, totdat het goedkoper wordt om het systeem weg te gooien en overnieuw te beginnen. Dit noemden zij de *wet van de voortdurende verandering*, the law of continuing change en werd hun eerste software-evolutie wet.

I *Law of continuing change*

A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.

II *Law of increasing entropy*

The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

Figuur 1: De eerste twee software-evolutie wetten van Belady en Lehman.

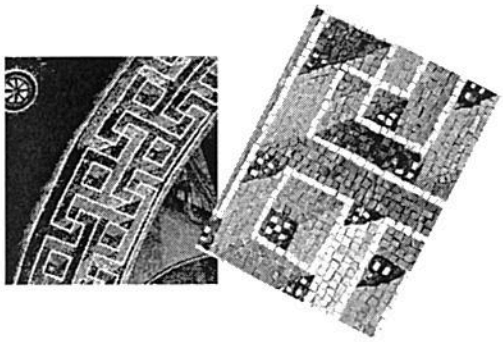
3 Software-entropie

Het wijzigen van software heeft echter veelal onbedoelde gevolgen: Een tweede observatie van Belady en Lehman was dat de *entropie* (d.w.z. de mate van wanorde of ongestructureerdheid) van een software-systeem in de loop der tijd toeneemt, tenzij specifiek werk wordt verricht om de structuur van het systeem te verbeteren. Dit noemden zij de *wet van de toenemende entropie* — the law of increasing entropy.[3]

Laten we om dit te begrijpen programmeren eens vergelijken met het maken van een mozaiek, zoals het voorbeeld uit Ravenna dat u ziet in Figuur 2. Een programmeur schrijft regels code, net zoals een mozaiekkunstenaar rijen steentjes legt. De code als geheel heeft een structuur, evenals het mozaiek, dat in dit geval is opgebouwd volgens het Griekse-sleutelmotief waarin witte lijnen en schuine vlakken die een 3D-effect teweeg brengen te onderscheiden zijn. De code kan aangepast worden, wat overeenkomt met het verplaatsen, toevoegen, of weghalen van mozaieksteentjes.

Dit veranderen van de steentjes in een mozaiek kan eenvoudig de structuur verstoren en bijvoorbeeld een lijn doorbreken of een gekleurd vlak vervuilen. Datzelfde geldt voor programma's: de structuur bestaat bijvoorbeeld uit lagen, modules en tijdsafhankelijkheden, die maar al te gauw verstoord kunnen worden wanneer regels code worden toegevoegd, verwijderd, of gewijzigd.

[3]Belady en Lehman hebben ook statistiche groeimodellen opgesteld voor software en de geldigheid van hun wetten is nog steeds onderwerp van onderzoek — zie bijvoorbeeld Godfrey's artikel over evolutie in open source systemen (Godfrey en Tu 2000). De door ons gebruikte eerste twee wetten van Belady en Lehman worden algemeen geaccepteerd.



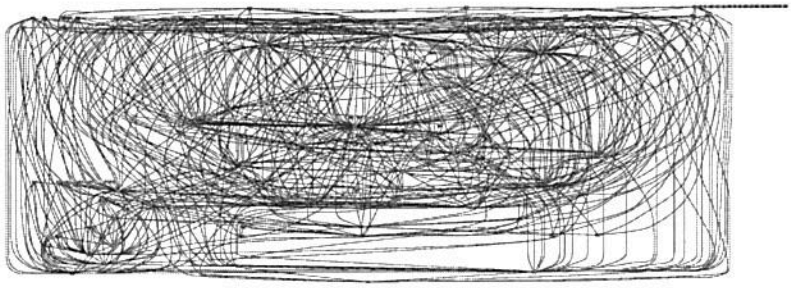
Figuur 2: Mozaiek uit Ravenna met Griekse-sleutelstructuur.

De kans dat dit gebeurt is des te groter omdat software, gemeten in regels door programmeurs geschreven code, zeer omvangrijk kan zijn en veelal jaarlijks groeit. Bovendien worden de aanpassingen door diverse programmeurs tegelijkertijd gedaan. Zo werken er bij ASML, producent van machines om chips mee te maken, 400 ontwikkelaars aan de besturingssoftware van deze machines die bij elkaar zo'n 12,5 miljoen regels C code vormt, en is ABN Amro eigenaar van 50 miljoen regels Cobol code. Vergelijken we elke regel code met een rijtje van 40 steentjes van 1×1 cm dan komt dit overeen met een jaarlijks groeiend mozaiek ter grootte van (momenteel) 10 voetbalvelden[4] waar 400 mensen continu veranderingen in aanbrengen.

Nu gaan software engineers natuurlijk niet zo maar regels code wijzigen en zullen ze er binnen de hun gegeven tijd alles aan doen om de structuur van hun programma goed te houden. Echter, elke wijziging wordt onder tijdsdruk uitgevoerd. De snelste manier om de wijziging door te voeren hoeft zeker niet de beste manier te zijn om de structuur in stand te houden.

Dit leidt dan gaandeweg tot een discrepantie tussen de inherente of vereiste complexiteit van een systeem (een auto of televisie is nu eenmaal gecompliceerd) en de interne complexiteit, d.w.z. de ingewikkeldheid van de gekozen oplossing. De meeste software-systemen zijn van binnen overmatig complex, iets wat bijvoorbeeld geïllustreerd kan worden door te kijken naar de afhankelijkheden tussen de verschillende systeemonderdelen, zoals in Figuur 3 gedaan

[4]Zie `http://nl.wikipedia.org/wiki/Voetbalveld` voor een discussie over het gebruik van voetbalvelden als oppervlaktemaat.



Figuur 3: Systeemstructuur in de praktijk.

voor de software van een groot Nederlands bedrijf.[5] Het netto effect van dit soort plaatjes is dan dat bij een eenvoudige buitenkant (een enkele knop uit Figuur 4) maar al te vaak een binnenkant hoort die veel ingewikkelder is dan noodzakelijk.

4 De software-evolutieparadox

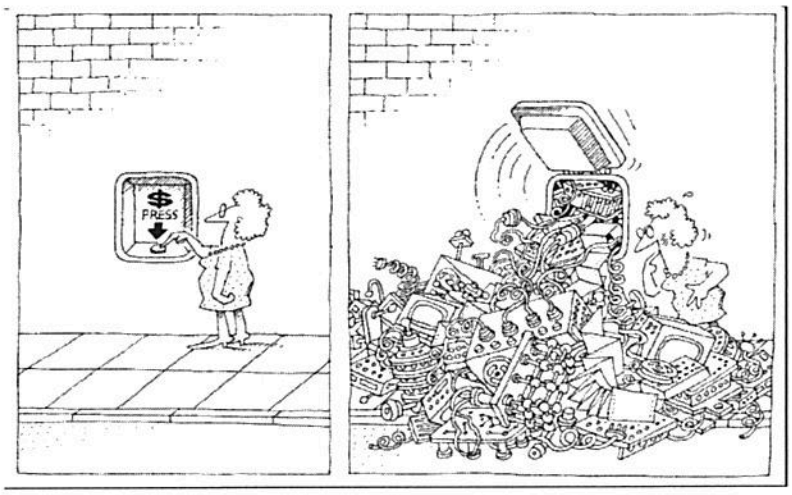
Nu zou een slechte interne structuur niet zo erg zijn wanneer u, als eindgebruiker, er niets van zou merken. Die interne structuur wordt echter belangrijk op het moment dat er wijzigingen aangebracht moeten worden en helpt dan bijvoorbeeld om de gevolgen van wijzigingen adequaat te voorspellen of om te garanderen dat bestaande, goed werkende functionaliteit ongemoeid blijft. Hoe slechter de structuur, hoe foutgevoeliger en tijdrovender, en dus hoe duurder het aanbrengen van wijzigingen is. Structuur is essentieel voor evolutie.

Maar zo belanden we wel in een ongewenste situatie: Evolutie moet maar leidt tot structuurerosie, waardoor evolutie moeilijker en uiteindelijk ondoenlijk wordt. Met andere woorden: *evolutie belemmert verdere evolutie* — en dit noemen we de software-evolutieparadox.[6]

Omdat vrijwel alle software-systemen wereldwijd hieronder lijden, vormt het doorbreken van deze paradox een flinke prikkel voor veel software enginee-

[5] Figuur 3 is afgeleid door de analyse-gereedschappen van de Software Improvement Group BV te Diemen. Zie `www.sig.nl`.

[6] In de literatuur is de software-evolutieparadox geen gangbaar begrip: het dichtst in de buurt komt het artikel van Tourwé en anderen dat handelt over aspect-georiënteerde softwareontwikkeling (Tourwé *et al.* 2003).



Figuur 4: Eenvoudige buitenkant, onverwachte binnenkant.

ring onderzoek. Daarbij valt op dat het meeste onderzoek zich richt op *preventie* en het voorkomen van erosie: als je nu maar netjes gestructureerd, object-georënteerd, testgestuurd, of volgens methode XYZ ontwikkelt, dan krijg je heus wel een goed aanpasbaar systeem. Ook het software engineering onderwijs is veelal volgens die lijn opgezet: we leren de student hoe hij of zij netjes moet programmeren teneinde problemen te voorkomen.

Maar alleen werken aan pogingen entropie te voorkomen is, hoe nuttig ook, onvoldoende: uiteindelijk zal de paradox zich manifesteren. Daarom is het cruciaal ook onderzoek te doen naar methoden en technieken die ons helpen om te gaan met overgeëvolueerde systemen.

Dit kan allereerst bestaan uit *structuurherstel* – we maken een pas op de plaats en proberen rigoureus alle slechte stukken code uit het systeem te verwijderen om zo de entropie te verminderen. Proberen we dit in grote stappen dan noemen we dit *renovatie* (Van Deursen *et al.* 1999); doen we het in kleine stapjes dan spreken we van *refactoring* (Fowler 1999). Helaas zijn de grote stappen bijzonder tijdrovend, risicovol en duur, en leveren de kleine stappen maar beperkte winst op.

Het alternatief is het ontwikkelen van methoden en technieken waarmee

we ondanks de toegenomen entropie toch wijzigingen kunnen blijven maken zonder dat dit direct leidt tot meer fouten of stijgende kosten. Het belangrijkste effect van toenemende entropie is dat het voor software engineers steeds moeilijker wordt de werking van een programma te doorgronden. Omdat dit *begrijpen* van programma's minstens de helft van de wijzigingsinspanning bedraagt (Corbi 1989), is het essentieel onderzoek te doen naar het inzichtelijk maken van bestaande programmatuur. Dit resulteert in gereedschap voor *software-exploratie*, dat de software engineer ondersteunt bij het op verschillende niveaus verkennen en doorgronden van een software-systeem (Van Deursen en Kuipers 1999, Moonen 2002). Dergelijk gereedschap is doorgaans gebaseerd op *reverse engineering* technieken, waarmee abstracte modellen uit bijvoorbeeld code gedestilleerd kunnen worden (Chikofsky en Cross 1990, Van Deursen en Stroulia 2005). Dit kan bijvoorbeeld door structuurrestanten te identificeren en aan te geven waar de afwijkingen zitten, door essentiële afhankelijkheden te detecteren en te visualiseren, of door het afleiden van architectuurinformatie uit broncode (Van Deursen *et al.* 2004).

Onderzoek naar preventie, structuurherstel, en software-exploratie roept een reeks aan fundamentele vragen in uiteenlopende disciplines op. Daarbij kunt u bijvoorbeeld denken aan:

Psychologie: Hoe begrijpt een programmeur zijn of haar programma? Hoe kunnen we hem of haar daarbij helpen?

Programmatuurtechnologie: Wat voor type structuren zijn belangrijk bij het doorvoeren van wijzigingen? Hoe kunnen we garanderen dat deze niet verstoord worden tijdens het maken van wijzigingen?

Programmacorrectheid: Hoe kunnen we garanderen dat programma's correct blijven werken als we de interne structuur aanpassen?

Organisatie: Hoe moeten we software-ontwikkelteams organiseren zodat ze evolutie verwelkomen in plaats van proberen tegen te houden?

Veldonderzoek: Kunnen we de invloed van evolutie op de structuur meten? Kunnen we middels modellen evolutie voorspellen?

Het zoeken naar antwoorden op deze vragen zal de komende jaren een centrale rol spelen in mijn onderzoek. Wat ik de rest van deze rede wil doen is dit concreet maken aan de hand van enkele recente onderzoeksresultaten (Bruntink *et al.* 2004, Bruntink *et al.* 2005, Van Deursen *et al.* 2005, Marin *et al.* 2004). Dit betreft onderzoek dat zich afspeelt op het terrein van

de programmatuurtechnologie, en wel in het bijzonder op dat van het nieuwe *aspect-georiënteerde programmeren*. Ik zal derhalve eerst kort uitleggen wat aspect-georiënteerde software-ontwikkeling inhoudt.

5 Aspect-georiënteerde software-ontwikkeling

Een belangrijk middel om evolutie mogelijk te maken is een modulaire manier van software-ontwikkeling. Een systeem wordt opgesplitst in componenten met elk een eigen taak. Het wijzigen of uitbreiden van zo'n taak is dan relatief eenvoudig, want beperkt tot slechts een enkele module. Althans, dat is de theorie.

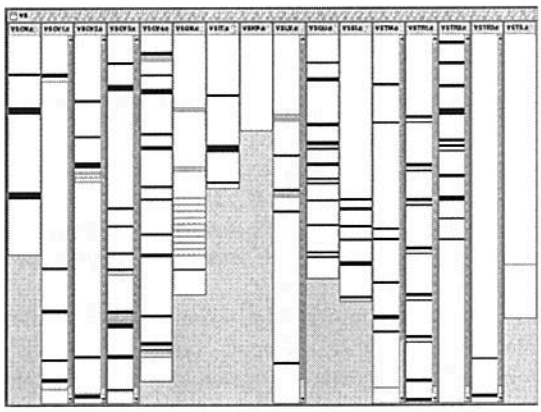
Helaas is de praktijk weerbarstiger. Sommige taken of functies vallen niet in een enkele module te isoleren, maar zijn inherent "alomtegenwoordig". Een bekend voorbeeld is *logging*, het bijhouden van de status van een programma in de loop der tijd. Dit kan niet door één enkele component gedaan worden, maar moet door elke component gebeuren. Andere voorbeelden zijn uniforme foutafhandeling, efficiënt geheugenbeheer, of beveiliging tegen bewust misbruik van het systeem — een op dit moment zeer actueel thema.

Dergelijke alomtegenwoordige functies (in het Engels *crosscutting* genoemd, omdat zij de modularisering *doorsnijden*) hebben twee voor evolutie vervelende gevolgen. Allereerst zijn zij *verspreid* (in het Engels *scattered*) geïmplementeerd en daarom zijn ze *zelf* moeilijk aan te passen. Wie een andere logging-strategie wil, loopt kans dat hij alle componenten moet wijzigen. Ten tweede zijn zij *verweven* (in het Engels *tangled*) met andere functies: wie een component wil aanpassen moet toch ook iets weten van logging, daar immers elke component aan logging doet.

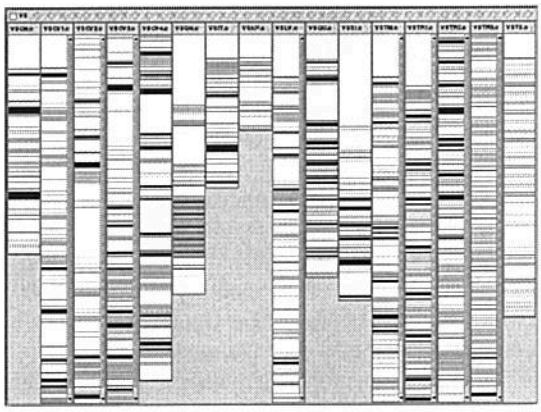
Laten we ter illustratie kijken naar een plaatje dat is gegenereerd uit code afkomstig van ASML (Bruntink *et al.* 2004). Figuur 5 toont de verspreiding van de code voor een specifieke verantwoordelijkheid, namelijk het controleren van inkomende parameters. Elke kolom is een module, en elke blauwe lijn is een regel code die gewijd aan parametercontroles. Duidelijk is te zien hoe de code voor parametercontroles verspreid is over alle modules.

Figuur 6 laat voor hetzelfde systeem meerdere doorsnijdende verantwoordelijkheden zien, zoals logging, tracing, en foutafhandeling, die elk een eigen kleur streep hebben. Nu is goed te zien hoe verweven de code is en hoe elke module (kolom) zich bezig dient te houden met een reeks van verschillende verantwoordelijkheden.

Een dergelijk veelkleurig plaatje is ongewenst en duidt op een moeilijk



Figuur 5: Spreiding van parameter controle code over modules.



Figuur 6: Code voor diverse doorsnijdende verantwoordelijkheden.

aanpasbaar systeem. Helaas zijn dit soort plaatjes nauwelijks te voorkomen wanneer gebruik gemaakt wordt van traditionele programmeertalen zoals C of Java. Aspect-georiënteerde software-ontwikkeling vormt een veelbelovende onderzoeksrichting om dergelijke veelkleurige plaatjes en de bijbehorende problemen van verspreiding en verwevenheid te voorkomen (Elrad *et al.* 2001). Centraal hierin staat het *aspect*, een nieuwe modulariseringsconstructie waarmee bestaande modules op specifieke plekken uitgebreid kunnen worden.

Wanneer we traditionele software-ontwikkeling vergelijken met het schrijven van een kookboek met een enkel recept van een paar duizend bladzijden, dan kunnen we aspect-georiënteerde software-ontwikkeling, wat oneerbiedig, vergelijken met het uitgeven van een addendum op dat kookboek. Zo'n addendum is een "aspect", dat ingrijpt op diverse plekken in het kookboek. Een aspect-georiënteerde uitbreiding zou kunnen bestaan uit regels zoals:

- *Zet elke keer als u uien snijdt het keukenraam open.*

of

- *Houdt telkens wanneer u een ingrediënt toevoegt aan uw gerecht bij in uw schrift hoeveel calorieën het bevat.*

In de terminologie van het aspect-georiënteerd programmeren is de plek waarop ingegrepen wordt een *joinpoint* en wordt de extra actie die daar ondernomen moet worden *advies* genoemd. Een *aspectenwever* kan gebruikt worden om het advies in de code te injecteren, of, in termen van het kookboek, om de regels uit het addendum direct op de relevante plekken in het boek te integreren en zo te komen tot een nieuw, dikker boek zonder addendum. Hierbij kan een enkele regel (bijvoorbeeld over het raam open zetten) naar diverse plekken in het kookboek gekopieerd worden (namelijk overal waar uien gesneden worden).

Aspecten kunnen helpen om onze veelkleurige plaatjes te ordenen. Zonder aspecten waren we gedwongen code voor een functie zoals parametercontroles (de blauwe strepen) in elke component op te nemen. Met aspecten schrijven we een nieuwe component (een extra kolom met ons addendum, het aspect) waarin we aangeven uit welke acties een parametercontrole zou moeten bestaan (ons advies, het openzetten van uw raam) en op welke plekken in de code dit moet gebeuren (de joinpoints, telkens als u uien snijdt). Op deze manier kunnen we elke kleur strepen in een aspect onder brengen en krijgen we een keurig systeem waarin elke kolom een eigen kleur heeft. Met andere woorden: de verpreiding en verwevenheid van alomtegenwoordige functies is verholpen en het systeem is eenvoudiger aan te passen.

Aspect-georiënteerd programmeren roept tal van interessante onderzoeksvragen op. Wordt evolutie daadwerkelijk gemakkelijker door aspect-georiënteerd programmeren? Is het inderdaad gemakkelijker een verbeterde versie van mijn kookboek uit te brengen? Wat voor functionaliteit laat zich goed beschrijven middels aspecten? Op welke plekken in een programma moeten we kunnen ingrijpen, of, met andere woorden, wat is het joinpoint-model? Dienen dergelijke joinpoints lexicaal, syntactisch, of semantisch gedefinieerd te worden? Moeten we het soort advies dat gegeven kan worden beperken? Kan aspect-georiënteerd programmeren helpen om bestaande systemen te verbeteren of werkt het alleen voor nieuw gemaakte systemen?

6 Aspect-oriëntatie en de evolutie-paradox

Nu is aspect-georiënteerd programmeren begonnen als een techniek om entropie als gevolg van evolutie te voorkomen. We hebben echter vastgesteld dat preventie alleen niet genoeg is, maar dat er juist behoefte is aan technieken om overgeëvolueerde systemen te doorgronden of opnieuw te structureren. Hoe kunnen aspecten hier een rol in spelen?

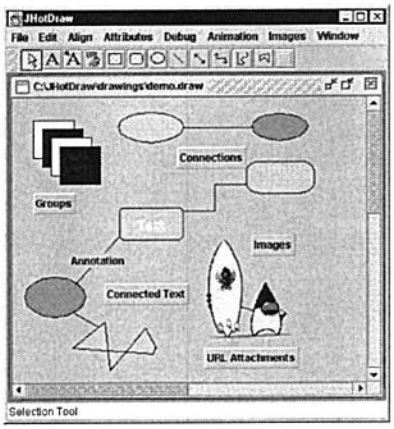
6.1 Experimentele data

Om de mogelijkheden van aspect-georiënteerd programmeren voor bestaande systemen te onderzoeken hebben we allereerst representatieve systemen nodig — deze vormen immers de experimentele data voor ons onderzoek. Hiertoe maken we onder meer gebruik van *open source* systemen waarvan de broncode vrijelijk beschikbaar is op het Internet. Zo kunnen andere onderzoekers onze experimenten herhalen en onze resultaten in vol detail bekritiseren. Eén van de systemen waar we ons momenteel op richten is JHOTDRAW[7] (zie Figuur 7), een tekenpakket bestaande uit zo'n 40.000 regels Java code.

Omdat open source ontwikkeling echter in veel opzichten anders is dan commerciële software-ontwikkeling met zijn doorgaans hardere deadlines en financiële randvoorwaarden, werken we ook met bedrijven samen. Op het gebied van aspect-georiënteerd programmeren doen we dat vanuit het Amsterdamse Centrum voor Wiskunde en Informatica met ASML,[8] producent van

[7] `www.jhotdraw.org`

[8] We doen dit binnen het Ideals project, een samenwerking tussen het Centrum voor Wiskunde en Informatica CWI, ASML, de universiteiten van Eindhoven en Twente en het Embedded Systems Institute te Eindhoven. Doel van dit project is het reduceren van de op dit moment jaarlijks stijgende software-ontwikkelkosten en het vergroten van de betrouwbaarheid van de ASML software.



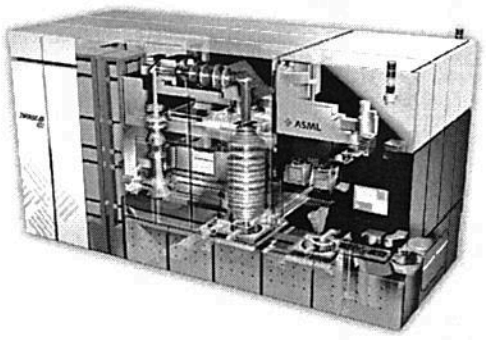
Figuur 7: Het JHOTDRAW tekenpakket.

chipmachines zoals getoond in Figuur 8.

6.2 Software-exploratie

De vraag die we ons nu stellen is hoe de aspect-georiënteerde theorie ons kan helpen bij het doorgronden van deze (of beter, dit soort) systemen. Om dit na te gaan proberen we een bestaand systeem te begrijpen in termen van alomtegenwoordige functionaliteit en proberen we de bijbehorende problemen van *verspreiding* en *verwevenheid* inzichtelijk te maken.

Onze eerste aanpak bestaat uit het inzetten van een techniek waarmee volledig automatisch gedupliceerde stukken code gevonden kunnen worden – *clone-detectie* genoemd. Het idee is dat een doorsnijdende verantwoordelijkheid zeer waarschijnlijk zal leiden tot veel kleine, sterk op elkaar lijkende stukjes code die overal verspreid in het systeem voor komen. Het "terugvinden" van deze vele kleine stukjes code maakt het mogelijk de verspreide implementatie toch in zijn geheel te begrijpen. Er bestaan verschillende clone-detectietechnieken, die variëren in het soort informatie dat gebruikt wordt om clones in te zoeken, zoals lexicale (woorden), syntactische (ontleedbomen) of semantische (program dependency graphs) informatie. Momenteel onderzoeken we welke van de diverse clone-detectietechnieken het meest geschikt is



Figuur 8: Waferstepper zoals ontwikkeld door ASML.

voor dit doel (Bruntink *et al.* 2004). Toepassing op de ASML code wijst er op dat voor bepaalde functies, zoals geheugenbeheer en parametercontroles, zo'n 80% van de bijbehorende code voldoende precies gereconstrueerd kan worden (Bruntink *et al.* 2004).

Daarnaast proberen we verweven en verspreide functionaliteit te vinden door een *metriek* te ontwikkelen die de mate uitdrukt waarin een bepaalde verantwoordelijkheid verspreid is geïmplementeerd (Marin *et al.* 2004). Momenteel bekijken we of routines (Java methoden) die vanaf veel verschillende plekken worden aangeroepen wellicht beter als *advies* gezien zouden kunnen worden. Het aantal verschillende plekken (de *fan-in* van de methode) vormt in dit geval de gezochte metriek en kan bijvoorbeeld gebruikt worden als waarschuwingssignaal voor verspreide functionaliteit.

Toepassing op de JHOTDRAW code wijst uit dat bijvoorbeeld *undo*-functionaliteit bij uitstek doorsnijdend is: dit is goed te begrijpen want bij het tekenen moet vrijwel elke gebruikersactie ongedaan gemaakt kunnen worden. Daarnaast illustreert het dat ontwerppatronen of (*design patterns* (Gamma *et al.* 1994) veelal doorsnijdend zijn, in overeenstemming met het werk van Hannemann en Kiczales die aspect-georiënteerde oplossingen voor de meest bekende ontwerppatronen voorstellen (Hannemann en Kiczales 2002).

```
if(queue == (CC_queue *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Input parameter %s error (NULL)",
          "CC_queue_empty", "queue"));
}
```

Figuur 9: Voorbeeld van een controle op parameter `CC_queue`.

6.3 Herstructurering

De ontwikkelde technieken, clone-detectie en fan-in analyse, stellen ons in staat automatisch functionaliteit te vinden die geschikt lijkt om met behulp van een aspect geïmplementeerd te worden. De volgende vraag is hoe dit het beste kan en wat de bijbehorende voor- en nadelen zijn.

Onze aanpak berust op de ontwikkeling van domein-specifieke aspect-georiënteerde talen. In het bijzonder hebben we gekeken naar de eerder genoemde parametercontroles met hun verspreide blauwe lijntjes uit Figuur 5. Wanneer de diverse software-componenten van de waferstepper informatie met elkaar uitwisselen, moet die informatie aan bepaalde eisen voldoen. Een voorbeeld hiervan is dat bepaalde parameters nooit de onbekende waarde `NULL` mogen bevatten. Momenteel worden dergelijke controles geprogrammeerd volgens een idioom zoals getoond in Figuur 9. Ons onderzoek heeft geleid tot een compactere weergave die automatisch uit bestaande code kan worden afgeleid, en die bovenal aanmerkelijk betrouwbaarder is doordat er geen controles vergeten worden (Bruntink *et al.* 2005). Op dit moment bekijken we of dezelfde ideeën toegepast kunnen worden op het aanmerkelijk moeilijkere foutafhandelingsidioom.

Essentieel bij een herstructurering naar een aspect-georiënteerde taal is dat er geen fouten in de software door worden geïntroduceerd. Om deze reden werken we momenteel aan testtechnieken die speciaal zijn toegespitst op het vinden van fouten die programmeurs zouden kunnen maken bij de overgang naar een aspect-georiënteerde programmertaal (Van Deursen *et al.* 2005). Dit resulteert in een aspect-georiënteerd foutmodel met bijbehorende *test adequacy criteria* en afdekkingsdoelen, die gebruikt kunnen worden om de kans dat de meest waarschijnlijke fouten gevonden worden te maximaliseren.

7 Perspectief

Hier wil ik het wat betreft de aspecten voor vanmiddag bij laten, hopend dat u een indruk heeft gekregen van het soort onderzoek dat ik doe op het gebied van software-evolutie. Al dat onderzoek wordt gedreven door nieuwsgierigheid naar werkende software-systemen. Welke technieken zijn met succes toegepast? Waarom? Is dit succes herhaalbaar? Maar ook: wat zijn de problemen die we steeds terug zien komen bij bestaande software-systemen? Zijn deze problemen te verminderen, op te lossen of te voorkomen?

Door bestaande systemen in detail te bestuderen verkrijgen we inzicht in de problemen die zich daadwerkelijk voordoen bij software-ontwikkeling en kunnen we pogen deze bij de wortel aan te pakken. Opvallend hierbij is de centrale rol die de evolutieparadox speelt: Elk werkend systeem is het slachtoffer van overmatige interne complexiteit als gevolg van evolutie.

Die software-evolutie paradox noopt ons de uitgangspunten van programmatuurontwikkeling opnieuw te doordenken. Dit betreft niet alleen het code-niveau (waarop ik me in het voorafgaande over aspecten vooral heb gericht), maar alle facetten van software-ontwikkeling, zoals bijvoorbeeld requirements, software-architectuur, software-proces of het testen van programmatuur. Voor enkele recente Delftse resultaten op deze gebieden verwijs ik u naar (Graaf *et al.* 2005, Lormans *et al.* 2004).

Onder het motto "er is niets zo praktisch als goede theorie" [9] hebben ook bedrijven grote belangstelling voor een meer fundamentele aanpak van de problemen rondom evolutie, problemen waar ze vaak dagelijks mee te maken hebben. We werken daarom met allerlei bedrijven op verschillende manieren samen. Soms gaat het op kleine schaal via bijvoorbeeld afstudeerders; In andere gevallen, zoals bij het eerder genoemde ASML, betreft het een langdurige samenwerking gericht op het uitwerken van specifieke problemen, methoden en technieken. Ook bij drie recent aan ons toegekende projectsubsidies zijn diverse bedrijven langdurig betrokken waaronder Philips, LogicaCMG en gespecialiseerde kleine bedrijven zoals Backbase en West Consulting.

Mijn onderzoek op het gebied van software-evolutie is begonnen als een activiteit op het Centrum voor Wiskunde en Informatica, in het kader van het Resolver project dat we uitvoerden voor ABN Amro en PinkRoccade. De hieruit voortgekomen resultaten vormden een beginpunt voor internationaal erkend evolutie-onderzoek en voor een spin-off bedrijf, de Software Improvement Group.

[9] Motto overgenomen uit het ten geleide van Keuning en Eppink, *Management & Organisatie: Theorie en Toepassing*, Stenfert Kroese, 1996

Mijn benoeming in Delft levert een unieke kans op om dit onderzoek verder uit te bouwen, niet alleen door de extra onderzoekscapaciteit, maar zeker ook door de interactie met studenten. Veel studenten willen bij een groot bedrijf afstuderen en in negen van de tien gevallen blijken ze dan te maken te krijgen met interessante evolutie-problemen. Door ons onderzoek te integreren in het Delftse software engineering onderwijs zijn studenten ook gewapend tegen dit soort problemen, zijn ze gewend kritisch na te denken over dergelijke problemen en kunnen ze de door ons voorgestelde methoden en technieken in de praktijk toetsen. Zo vergroten we de experimentele basis voor ons onderzoek, kunnen we onze onderzoeksresultaten verder verfijnen en verbeteren en kunnen we onze vraagstelling richten op de meest urgente problemen uit de software engineering praktijk.

8 Tot besluit

Geachte toehoorders
Omwille van de tijd moet ik het hierbij laten. Aan de hand van vragen aan u heb ik gesproken over het belang van evolutie, de hieruit voortvloeiende entropie, en de evolutie-problemen die dit met zich meebrengt. Ik heb gepleit voor onderzoek dat verder gaat dan entropiepreventie, en dat zich richt op bijvoorbeeld software-exploratie en herstructurering. Hoe dit kan heb ik geïllustreerd aan de hand van ons onderzoek op het gebied van aspect-georiënteerde softwareontwikkeling. Dergelijk onderzoek vereist enerzijds een nieuwsgierigheidsgedreven fundamentele aanpak, en anderzijds concrete verfijning en toetsing in samenwerking met het bedrijfsleven. Ten slotte heb ik uiteengezet waarom dergelijk onderzoek geïntegreerd dient te worden in het universitaire softwareengineering onderwijs.

9 Dankwoord

Ik wil afsluiten door allen te bedanken die eraan bij hebben gedragen dat ik hier sta. Daartoe hoort zonder meer het Amsterdamse Centrum voor Wiskunde en Informatica, een onderzoeksinstituut van wereldklasse. Dank aan de hele SEN-1 groep, en in het bijzonder aan Paul Klint, promotor en leermeester in alle positieve betekenissen van het woord; en aan Jan Heering, die altijd door blijft vragen tot hij snapt hoe het zit en nog bij het proeflezen van deze rede bij een alinea schreef: *Wat betekent dat nu precies?*

Aan de TU Delft wil ik Jan van Katwijk, Henk Sips en Erik Jansen bedanken voor het vertrouwen en de hulp bij management-gerelateerde zaken; en de software engineering groep voor de warme ontvangst. In het bijzonder wil ik Leon Moonen noemen die moest zien te overleven in de chaotische tijd dat we samen het Software Evolution Research Lab SWERL op aan het zetten waren, en Arjan van Gemund, de Delftse routinier die het thema embedded software binnen de groep vorm aan het geven is.

Een speciaal woord van dank aan mijn promovendi, die het aan hebben gedurfd meerdere jaren met mij in zee te gaan: Tobias Kuipers, Leon Moonen, Merijn de Jonge, Marius Marin, Magiel Bruntink, Bas Graaf, en Marco Lormans: Bedankt voor de goede samenwerking en de mooie resultaten! Verder gaat mijn dank uit naar Remco van Engelen, Jan Heering, Tom Tourwé en Hans Vonk en Machteld Vonk voor de feedback op eerdere versies van de rede.

Mijn vrienden en familie hebben er het meest onder te lijden wanneer ik me weer eens laat meeslepen door mijn werk. Dank voor jullie geduld, interesse en onmisbare gezelligheid. Mijn schoonouders, Hans en Alida Vonk, wil ik danken voor hun voortdurende belangstelling en steun, niet in de laatste plaats als vaak ingezette breng-, haal- en oppasdienst. Een speciaal woord van dank voor mijn vader en moeder, voor wie de wil om kennis te vergaren en kennis te delen iets volstrekt vanzelfsprekends is.

Thuis op de Brantwijk, ten slotte, maken we er een opwindende tijd van door twee drukke banen te combineren met een tweeling van één en een dochter van vier. Lieve David, Sebastiaan en Julia: niets is leuker dan niet werken en met jullie spelen. Lieve Machteld: dankjewel voor alles!

Ik heb gezegd.

Referenties

L. A. Belady en M. M. Lehman, 1976. A Model of Large Program Development. *IBM Systems Journal*, 15-3 (1976), 225–252.

G. Booch, 2004. Handbook of Software Architecture. Draft, zie `http://www.booch.com/architecture/`.

M. Bruntink, A. van Deursen, R. van Engelen, en T. Tourwé, 2004. An Evaluation of Clone Detection Techniques for Identifying Cross-Cutting Concerns. In *Proceedings International Conference on Software Maintenance (ICSM 2004)*, p. 200–209. IEEE Computer Society.

M. Bruntink, A. van Deursen, en T. Tourwé, 2005. Isolating Crosscutting Concerns in System Software. Technical Report SEN-R0504, CWI.

M. Campbell-Kelly, 2003. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. MIT Press.

E.J. Chikofsky en J.H. Cross, 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7-1 (1990), 13–17.

T. A. Corbi, 1989. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28-2 (1989), 294–306.

A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, en C. Riva, 2004. Symphony: View-Driven Software Architecture Reconstruction. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, p. 122–134. IEEE Computer Society.

A. van Deursen, P. Klint, en C. Verhoef, 1999. Research Issues in Software Renovation. In *Fundamental Approaches to Software Engineering (FASE '99)*, J.-P. Finance, redactie, Lecture Notes in Computer Science, p. 1–21. Springer-Verlag.

A. van Deursen en T. Kuipers, 1999. Building Documentation Generators. In *Proceedings International Conference on Software Maintenance*, p. 40–49. IEEE Computer Society.

A. van Deursen, M. Marin, en L. Moonen, 2005. A Systematic Aspect-Oriented Refactoring And Testing Strategy, and its Application to JHotDraw. Technical report, CWI and Delft University of Technology.

A. van Deursen en E. Stroulia, 2005. Editors' Introduction: 10th WCRE Working Conference on Reverse Engineering. *IEEE Transactions on Software Engineering*, 31-2 (2005).

T. Elrad, M. Akşit, G. Kiczales, K. Lieberherr, en H. Ossher, 2001. Discussing aspects of AOP. *Communcations of the ACM*, 44-10 (2001), 33–38.

M. Fowler, 1999. *Refactoring: Improving the Design of Existing Code.* Addison Wesley.

E. Gamma, R. Helm, R. Johnson, en J. Vlissides, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

M. W. Godfrey en Q. Tu, 2000. Evolution in Open Source Software: A Case Study. In *Proceedings International Conference on Software Maintenance (ICSM'00)*, p. 131–142. IEEE Computer Society.

B. Graaf, H. van Dijk, en A. van Deursen, 2005. Evaluating an Embedded Software Reference Architecture: Experience Report. In *Proceedings 9th European Conference on Software Maintenance and Reengineering (CSMR 2005).* IEEE Computer Society.

J. Hannemann en G. Kiczales, 2002. Design Pattern Implementation in Java and AspectJ. In *Proceedings 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, p. 161–173. ACM Press.

M. Lormans, H. van Dijk, A. van Deursen, E. Nöcker, en A. de Zeeuw, 2004. Managing Evolving Requirements in an Outsoucring Context: An Industrial Experience Report. In *Proceedings International Workshop on Principles of Software Evolution (IWPSE'04)*, p. 149–158. IEEE Computer Society.

M. Marin, A. van Deursen, en L. Moonen, 2004. Identifying aspects using fan-in analysis. In *Proceedings 11th Working Conference on Reverse Engineering (WCRE)*, p. 132–141. IEEE Computer Society.

L. Moonen, 2002. *Exploring Software Systems.* Dissertatie, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam.

T. Tourwé, J. Brichau, en K. Gybels, 2003. On the Existence of the AOSD-Evolution Paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT).* DAIMI, Aarhus University.