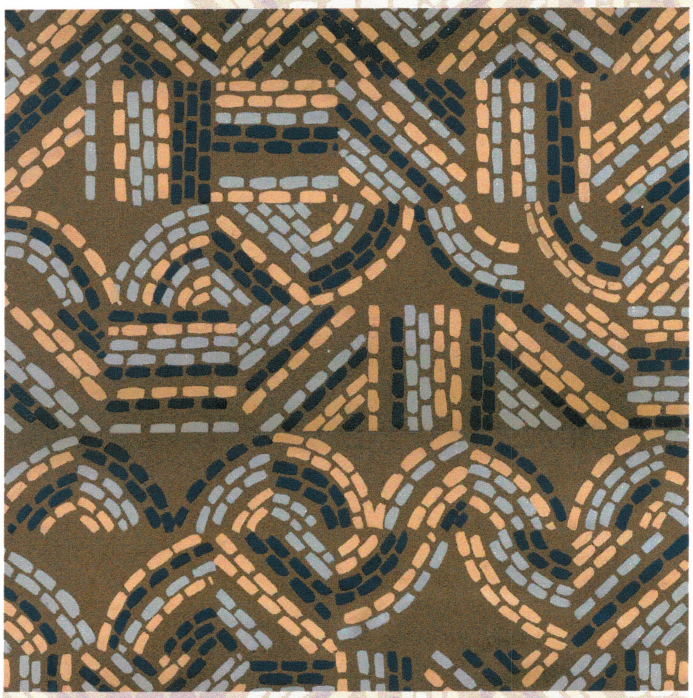
Uiteindelijk loopt elk systeem vast

Softwaresystemen moeten evolueren. Evolutie leidt echter tot erosie van de interne structuur. En die erosie, concludeert Arie van Deursen in zijn inaugurele rede, maakt evolutie uiteindelijk onmogelijk. Dit is de software-evolutieparadox. Wat we ook doen, vroeg of laat manifesteert zich deze paradox. Aspect-georiënteerde software-ontwikkeling is mogelijk een uitweg uit de impasse.

Een goede modularisering is een belangrijk hulpmiddel bij evolutie. Helaas is sommige functionaliteit niet of nauwelijks binnen een enkele module te isoleren, maar inherent 'alomtegenwoordig'. Een bekend voorbeeld is 'logging': dit kan niet door één enkele component gedaan worden, maar moet door elke component gebeuren. Andere voorbeelden zijn uniforme foutafhandeling, efficiënt geheugenbeheer of beveiliging.
Dergelijke alomtegenwoordige functies ('cross-cutting' genoemd, omdat zij de modularisering doorsnijden) hebben twee voor evolutie vervelende gevolgen. Allereerst zijn zij verspreid ('scattered') geïmplementeerd en daarom zijn ze zelf moeilijk aan te passen. Wie een andere loggingstrategie wil, loopt kans dat hij alle componenten moet wijzigen. Ten tweede zijn zij verweven ('tangled') met andere functies: wie een component wil aanpassen moet toch ook iets weten van logging, daar immers elke component aan logging doet.
Aspect-georiënteerde software-ontwikkeling is een veelbelovende onderzoeksrichting om de problemen van verspreiding en verwevenheid te voorkomen. 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 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.
- Houdt telkens wanneer u een ingrediënt toevoegt aan uw gerecht bij in uw schrift hoeveel calorieën het bevat.

De plek waarop ingegrepen wordt (overal waar uien gesneden worden) is een 'joinpoint', en de extra actie die daar ondernomen moet worden (het raam openzetten) is een 'advies'. 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 naar diverse plekken in het kookboek gekopieerd worden. Door gebruik te maken van advies en joinpoints wordt het mogelijk alle code die bij alomtegenwoordige functionaliteit hoort in een enkel aspect te vangen, dat vervolgens ingrijpt op alle andere componenten. De functionaliteit is nu geïsoleerd in een enkel aspect (geen verspreiding) en de andere componenten kunnen begrepen worden zonder te weten van het bestaan van dit aspect (geen verwevenheid).

Voor meer informatie: http://www.aosd.net/.



Programmatuur-ontwikkeling is te vergelijken met een mozaïek. Het verplaatsen of weghalen van steentjes kan de structuur makkelijk verstoren. Zoals het aanpassen van een regel code eenvoudig kan leiden tot het verbreken van de softwarestructuur. FOTO: GETTY IMAGES

Onze beschaving draait op software, stelt Bjarne Stroustrup, de ontwerper van C++. Software engineers zijn niet alleen nodig om deze software te maken, maar steeds meer om hem aan te passen. Niet alleen van Word, Windows XP of Linux verschijnen met de regelmaat van de klok nieuwe versies, maar van vrijwel alle softwaresystemen. Hiervoor zijn een aantal oorzaken aan te wijzen.
1. De oude versie bevat vaak fouten.
2. 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 etcetera.
3. Eerdere versies van een softwareproduct worden vaak bewust eenvoudig 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. Vaak wordt ook gesproken over software-onderhoud, maar deze benaming is minder geschikt: de eerdergenoemde oorzaken geven aan dat het veranderen van software van een andere orde is dan bijvoorbeeld het vervangen van banden of het schilderen van kozijnen.
Software-evolutie trok in de jaren zeventig de aandacht van Belady en Lehman, twee onderzoekers van IBM die werkten aan de ontwikkeling van het besturingssysteem voor de IBM 360-mainframes, een project van vijfduizend mensjaar. Op basis van dit project formuleerden zij een aantal 'wetten van de software-evolutie'. De eerste hiervan, 'de wet van de voortdurende verandering', stelt dat een softwaresysteem dat werkelijk gebruikt wordt een voortdurend proces van verandering ondergaat, totdat het goedkoper wordt om het systeem weg te gooien en opnieuw te beginnen.
Het wijzigen van software heeft echter veelal onbedoelde gevolgen. Een tweede observatie van Belady en Lehman was dat de 'entropie' (dat wil zeggen de mate van wanorde of ongestructureerdheid) van een softwaresysteem 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'.
Om ons een voorstelling te maken van het ontstaan van deze entropie kunnen we programmatuurontwikkeling vergelijken met het maken van een mozaïek. Zo'n mozaïek heeft een structuur die bestaat uit bijvoorbeeld gekleurde vlakken of lijnen. Het verplaatsten, toevoegen of weghalen van mozaïeksteentjes kan maar al te gemakkelijk de structuur verstoren. Ook programma's hebben structuur, die bestaat uit bijvoorbeeld lagen, modules of tijdsafhankelijkheden. Een regel code aanpassen kan eenvoudig leiden tot het verbreken van deze structuur.
De kans dat dit gebeurt is des te groter omdat software vaak zeer omvangrijk is en veelal jaarlijks groeit. Bovendien worden de aanpassingen door diverse programmeurs tegelijkertijd gedaan. Zo werken er bij chipmachinefabrikant ASML vierhonderd 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 veertig steentjes van een vierkante centimeter dan komt dit overeen met een jaarlijks groeiend mozaïek ter grootte van (momenteel) tien voetbalvelden waar vierhonderd mensen continu veranderingen in aanbrengen.
Nu gaan software engineers natuurlijk niet zomaar 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, dat wil zeggen de ingewikkeldheid van de gekozen oplossing. Wie bestaande softwaresystemen analyseert, zal vast moeten stellen dat de meeste van deze systemen van binnen overmatig complex zijn.

Meeste systemen zijn overmatig complex

Evolutieparadox
Nu zou een slechte interne structuur niet zo erg zijn wanneer de eindgebruiker er niets van zou merken. Die interne structuur is echter van groot belang 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 in een tegenstrijdigheid. Evolutie moet, maar leidt tot structuurerosie, waardoor evolutie moeilijker, en uiteindelijk onmogelijk wordt. We kunnen dit samenvatten als de 'software-evolutieparadox': evolutie belemmert verdere evolutie.
De meest voor de hand liggende reactie op deze paradox is preventie: we moeten proberen erosie te voorkomen. Als je nu maar netjes gestructureerd, objectgeoriënteerd, testgestuurd of volgens methode XYZ ontwikkelt, dan krijg je wel een goed aanpasbaar systeem. En is je systeem slecht gestructureerd, dan is het de schuld van de programmeur die de desbetreffende methode niet goed heeft toegepast. Ook het software-engineeringonderwijs is veelal volgens die lijn opgezet: we leren de student hoe hij netjes moet programmeren teneinde problemen te voorkomen.
Maar zonder ook maar iets af te doen aan het belang van gedisciplineerde software-ontwikkeling, zullen we moeten erkennen dat alleen werken aan pogingen entropie te voorkomen onvoldoende is: uiteindelijk zal de paradox zich manifesteren. Dit heeft twee belangrijke consequenties voor het vakgebied software engineering.
De eerste is dat structuurherstel noodzakelijk is om wijzigingen kosten-effectief en op een betrouwbare manier door te voeren. Bij structuurherstel maken we 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 heet dit 'renovatie', doen we het in kleine stapjes dan spreken we van 'refactoring'. Helaas zijn met de huidige stand der techniek de grote stappen bijzonder tijdrovend, risicovol en duur, en leveren de kleine stappen maar beperkte winst op. Meer onderzoek is noodzakelijk om refactoring en renovatie beter te ondersteunen.

Onderzoek

**Welke rol kan aspect-georiënteerde software-ontwikkeling spelen bij het doorgronden of opnieuw structureren van overgeëvolueerde systemen? Hier wordt onderzoek naar gedaan aan zowel de TU Delft als aan het Amsterdamse Centrum voor Wiskunde en Informatica. Dit betreft enerzijds open source Java-code en anderzijds industriële C-code van ASML, dit laatste in het kader van een samenwerking met onder meer het Embedded Systems Institute uit Eindhoven.
Een eerste vraag is hoe de aspect-georiënteerde theorie kan helpen bij het doorgronden van bestaande systemen. Dit komt neer op het begrijpen van een bestaand systeem in termen van alomtegenwoordige functionaliteit en op het inzichtelijk maken van de bijbehorende problemen van 'verspreiding' en 'verwevenheid'. Een van de mogelijkheden is het inzetten van clone-detectietechnieken, waarmee volledig automatisch gedupliceerde stukken code gevonden kunnen worden. Op dit moment wordt bestudeerd in hoeverre dergelijke technieken gebruikt kunnen worden om verspreid geïmplementeerde functionaliteit bij elkaar te vinden.
Een tweede mogelijkheid bestaat uit het gebruik van metrieken die de verspreiding in een getal uitdrukken.
Een tweede vraag is of de op deze manier gevonden doorsnijdende verantwoordelijkheden met behulp van een aspect geïmplementeerd kunnen worden, hoe dit moet, en wat de bijbehorende voor- en nadelen zijn.**

Voor meer informatie: http://www.esi.nl/ideals of. http://swerl.tudelft.nl/.

Veranderen van software is iets anders dan het verwisselen van banden

Een tweede consequentie is dat we er niet aan ontkomen slecht gestructureerde systemen aan te passen. Uit onderzoek blijkt dat het merendeel van de tijd benodigd voor het doorvoeren van een wijziging, zit in het doorgronden van de werking van het systeem. Naarmate de structuur van het systeem slechter is, neemt deze tijd toe. Derhalve is het noodzakelijk technologie te ontwikkelen waarmee bestaande systemen, ondanks de toegenomen entropie, gemakkelijker begrepen kunnen worden. Dit resulteert in gereedschap voor 'software-exploratie', dat de software engineer ondersteunt bij het op verschillende niveaus verkennen en doorgronden van een softwaresysteem.

Fundamenteel
Willen we voortgang op het gebied van structuurherstel en exploratie boeken, dan dienen we een aantal fundamentele vragen te beantwoorden. Hoe begrijpt een programmeur zijn programma? Hoe kunnen we hem daarbij helpen? 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? Kunnen we de invloed van evolutie op de structuur meten, monitoren of voorspellen?
Dergelijke vragen openen een nieuw perspectief op allerlei ontwikkelingen in de software engineering. Een voorbeeld wordt gegeven in bijgevoegde kaders, waarin allereerst aspect-georiënteerd programmeren wordt uitgelegd en vervolgens wordt gekeken hoe dit kan worden toegepast op bestaande software.
Onder het motto 'er is niets zo praktisch als goede theorie' hebben ook bedrijven veel te winnen bij een meer fundamentele aanpak van de problemen rondom evolutie, problemen waar ze vaak dagelijks mee te maken hebben. Bedrijven die meewerken aan dergelijk onderzoek leveren de experimentele data in de vorm van hun softwaresystemen aan, en kunnen de onderzoeksvragen sturen in de richting van de voor hen meest urgente problemen.
Ook studenten informatica komen vaak al tijdens hun afstuderen in aanraking met interessante evolutieproblemen. Om de studenten te wapenen tegen dit soort problemen is het essentieel onderzoek op het gebied van evolutie te integreren in het software-engineering:curriculum. Zo kunnen studenten de ontwikkelde methoden en technieken toepassen in de praktijk, wat weer helpt om de experimentele basis van het onderzoek te vergroten en de resultaten te verfijnen en te verbeteren.

ARIE VAN DEURSEN
AG - 11-03-'05

Arie van Deursen is hoogleraar software engineering aan de Technische Universiteit Delft en onderzoeksleider aan het Centrum voor Wiskunde en Informatica te Amsterdam (http://www.cwi.nl/~arie/). Dit is een aangepaste en ingekorte versie van zijn intreerede, uitgesproken op 23 februari 2005 ter gelegenheid van de aanvaarding van het ambt van hoogleraar aan de Technische Universiteit Delft.