t

# Automatiseren onderhoud complex maar lucratief

## Grammatica essentieel voor analyseren broncode

Het merendeel van het onderhoud aan grote softwaresystemen wordt handmatig uitgevoerd. Dit proces is foutgevoelig en kost veel tijd. Een geautomatiseerde aanpak is vaak sneller, nauwkeuriger en dus kostenbesparend.

*Steven Klusener en Niels Veerman*

De snelle ontwikkelingen in de softwaretechnologie zijn voorbijgegaan aan het onderhoud, beheer en aanpassen van operationele software. Veel softwaresystemen zijn nog ontwikkeld in de jaren '70 en '80. Deze legacysystemen zijn ontoegankelijk en statisch omdat zij geschreven zijn in verouderde programmeertalen maar vooral ook met verouderde programmeerconventies. De onderhoudsprogrammeurs zijn meestal niet bij de ontwikkeling van het originele systeem betrokken geweest en hebben vaak slechts een globaal idee van de werking van het systeem. Het is dan ook niet verwonderlijk dat softwareonderhoud een kostbare aangelegenheid is. Zo werken er in Nederland naar schatting meer dan tienduizend it'ers aan softwareonderhoud. Uit diverse studies blijkt dat meer dan de helft van de kosten van software besteed wordt na oplevering; van deze kosten wordt weer de helft besteed aan het analyseren van de code, een kwart aan testen en slechts een 5% aan het doorvoeren van de eigenlijke wijziging (zie figuur 1). Softwareonderhoud kan efficiënter worden ingericht door de inzet van automatische hulpmiddelen. Hierbij is het essentieel dat deze hulpmiddelen snel kunnen worden aangepast aan de kenmerken van de onderhavige onderhoudsproblematiek, en dat de software-engineers die deze tooling ontwikkelen de onderhoudsproblematiek in detail begrijpen.

### Veranderingen

Softwaresystemen zijn constant onderhevig aan veranderingen en worden daardoor steeds complexer. Deze aanpassingen blijven niet beperkt tot het uitbreiden van een datum voor het jaar 2000 of invoering van de euro. Vele kleinere veranderingen zijn aan de orde van de dag. Bijvoorbeeld een nieuwe versie van een besturingssysteem, een databasemigratie, een verandering in de renteberekening, of het zoeken en verwijderen van fouten. Deze aanpassingen worden gedaan door (verschillende) onderhoudsprogrammeurs en resulteren in groeiende, steeds complexer wordende systemen. Iedere aanpassing vergt hierdoor steeds meer tijd en brengt dus hogere kosten met zich mee.

### Onderhoudsproblematiek

De onderhoudsproblematiek, vooral die van legacysystemen, verschilt in grote mate van de software-engineering zoals onderwe-

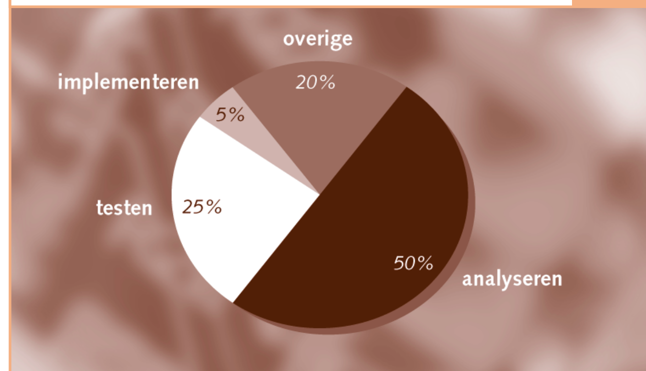
## Samenvatting

**Geautomatiseerde softwaretransformatie is efficiënter, sneller en nauwkeuriger dan handmatig onderhoud. Eerst moet het probleem worden geanalyseerd. Ligt de probleemdefinitie vast, dan kan op basis van *generieke taaltechnologie* een migratietool worden gebouwd. Met behulp van deze technologie is de transformatie een kwestie van zoek-en-vervangopdrachten.**

zen wordt op de universiteiten en hogescholen. Talen als Cobol en jcl's (job control language) zijn de standaard bij het merendeel van de grote softwaresystemen, en niet de moderne objectgeoriënteerde talen als Java of C++, of zelfs de meer klassieke procedurele talen als Pascal en C. Eenvoudige concepten uit het abstractiebeginsel van de software-engineering, zoals procedures en lokale variabelen (in Pascal en C) of de meer geavanceerde objectoriëntatie uit Java en C++ zijn niet beschikbaar in de Cobol-varianten Cobol'74 en Cobol'85. In deze talen zijn de legacysystemen vaak geschreven. (Deze concepten zijn overigens wel tot op zekere hoogte beschikbaar in de laatste versie van Cobol, Cobol2000, maar introductie van deze concepten in de legacysystemen zal vele complexe aanpassingen vereisen.) Hierdoor kan een analyse van een Cobol-fragment meer tijd kosten dan een fragment in een andere taal. De onderhoudsprogrammeur moet immers uitzoeken welke plaats het fragment inneemt in de control-flow van het gehele programma, en waar de variabelen die erin voorkomen nog meer worden gebruikt (alle variabelen zijn immers globaal).

De data van een legacysysteem zijn veelal niet opgeslagen in relationele databases maar in hiërarchische databases als IMS of zelfs in een groot aantal platte files. Merk op dat hiërarchische databases, net als Cobol, al sinds tientallen jaren als verouderd beschouwd worden en daarom niet worden onderwezen.

**Onderverdeling van onderhoudskosten** 1




Vooral de echt grote databases, van bijvoorbeeld banken en overheidsinstanties met persoonsgegevens van miljoenen mensen, zijn vaak nog hiërarchisch; zo wordt gezegd dat de meerderheid van de data wereldwijd nog worden bijgehouden in hiërarchische databases. *Sourcecode* (broncode) van deze complexe systemen moet worden aangepast in eenvoudige teksteditors. De onderhoudsprogrammeur heeft geen 'intelligente' programmeeromgeving zoals Delphi, Visual Studio, of Visual Age tot zijn beschikking, waardoor het aanpassen van de naam van een variabele al een tijdrovende actie wordt. Verder zijn legacysystemen over het algemeen groot; zij bestaan veelal uit duizenden programma's van elk duizenden (of zelfs tienduizenden) regels code, hun totale omvang varieert tussen de honderdduizend en enkele miljoenen regels code. Hierbij kunnen nog duizenden *copybooks* (*include files*), schermdefinities, record- en tabeldefinities en batch-jobs komen.

Er zijn nog meer verschillen met de modernere software-engineering, zo kunnen voor documentatie, testen, versiebeheer en vele andere aspecten soortgelijke verschillen worden aangegeven. Deze verschillen leveren ook een cultuurverschil op tussen de it'ers die zich met systeemonderhoud bezighouden en de it'ers die zich met 'moderne' softwaretechnologie bezighouden. Om het onderhoudsproces effectiever in te richten is het van belang om dit cultuurverschil te overbruggen. Dit kan onder meer bereikt worden door over te gaan van handmatig naar grotendeels geautomatiseerd softwareonderhoud.

### Risico's

Het analyseren en aanpassen van legacysoftwaresystemen voor onderhoud gebeurt vooralsnog grotendeels handmatig. Echter, bij deze aanpak speelt een aantal risico's. In het begin van een aanpassingstraject gaat men bij de probleemdefinitie vaak uit van de reguliere gevallen die veranderd moeten worden. Pas bij de daadwerkelijke uitvoering komen de uitzonderingsgevallen aan het licht. Hierna wordt de probleemdefinitie

aangepast. Dit houdt echter wel in dat men telkens opnieuw moet beginnen.
De probleemdefinitie kent ook vaak interpretatieverschillen zonder dat dit de betrokkenen duidelijk is. Binnen een team van onderhoudsprogrammeurs levert dit verschillende resultaten op, die pas laat in het project naar voren kunnen komen. Komen deze interpretatieverschillen eenmaal naar voren, dan moet de probleemdefinitie worden aangescherpt en kan men weer opnieuw beginnen.
Verder is uit de literatuur bekend dat bij ongeveer 10% van de tekstaanpassingen een typefout wordt gemaakt, wat de kans op fouten bij een handmatige aanpassing groot maakt. Om de genoemde risico's enigszins op te vangen moet er een uitgebreid testproces worden ingericht, dat dikwijls tijdrovender is dan de uitvoering van de aanpassing zelf. Al met al kost een handmatige aanpak vaak niet alleen veel tijd maar is het ook nog eens foutgevoelig.

### Automatische hulpmiddelen

Veel van de analyses en aanpassingen voor onderhoud zijn met de juiste technologie gedeeltelijk of geheel te automatiseren. Vooral bij grote systemen (meer dan 10 duizend tot meer dan 1 miljoen regels sourcecode) is een automatische benadering interessant omdat het vaak om kleine veranderingen gaat, die op vele plaatsen moeten worden uitgevoerd.
De voordelen van een automatische aanpak zijn vooral efficiëntie, snelheid en nauwkeurigheid. Voordat de automatische hulpmiddelen ontwikkeld worden, moet de probleemdefinitie tot in detail duidelijk zijn. Bij eventuele nieuwe inzichten zijn de hulpmiddelen snel aan te passen. Bovendien is het aantal regels code dat uiteindelijk wordt ingetypt om een automatische aanpassing te specificeren aanzienlijk kleiner dan het aantal dat nodig is bij een handmatige aanpak, wat de kans op typefouten verlaagt.
Aan de Vrije Universiteit in Amsterdam doet een team in nauwe samenwerking met het bedrijfsleven, onderzoek naar grootschalige automatische transformatieprojecten zoals hierboven beschreven. Zij onderzoeken onder meer het automatisch aanpassen van grote systemen, het analyseren en herstructureren van sourcecode, en het verkrijgen en aanpassen van grammatica's die de verschillende programmeertalen beschrijven. Grammatica's vormen een essentieel onderdeel bij het analyseren en aanpassen van sourcecode.

In projecten zijn zowel analyse als aanpassing noodzakelijk. In een reguliere aanpak werkt een migratie-expertiseteam hiervoor samen met een systeemexpertiseteam. In overleg met het management doorlopen zij aantal stappen om tot een geautomatiseerd resultaat te komen. Het migratie-expertiseteam ontwikkelt de automatische tooling en het systeemexpertiseteam onderhoudt het systeem.

### Probleemdefinitie

Uiteraard begint een aanpassingsproject bij het vaststellen van een algemene probleemdefinitie, zoals 'alle productcodes moeten met één positie worden uitgebreid', of 'de software moet worden aangepast voor een migratie van de database van versie X naar versie Y'. Bij het voorbeeld van de productcodes kan men zich wat voorstellen, maar in het tweede voorbeeld zijn meer details nodig over de taalconstructies die in de software moeten worden aangepast. Dit gebeurt in overleg met het management, dat globaal het budget aangeeft, en de onderhoudsprogrammeurs van het systeemexpertiseteam, die bijvoorbeeld aangeven wat de databasemigratie inhoudt.

### Quickscan

Het migratie-expertiseteam bestudeert vervolgens de sourcecode van het systeem en waar nodig de technische handleidingen. Voor het bestuderen van de sourcecode kan in korte tijd speciale tooling worden ontwikkeld om de sourcecode te scannen. Het doel van de quickscan is de probleemdefinitie in meer detail vast te stellen; de diverse uitzonderingssituaties worden geformuleerd, en er wordt een schatting gemaakt hoe vaak zij voorkomen. Deze quickscan moet een korte doorlooptijd hebben. Het resultaat is een overzicht van de verschillende varianten waarin het probleem voorkomt (zie het voorbeeld in figuur 2). Van elke variant wordt aangegeven of een automatische aanpak mogelijk is, welke inspanning nodig is voor het ontwikkelen en afstemmen van de tooling en hoe vaak de variant zich voordoet.

### Taakverdeling

In overleg met beide teams bepaalt het management welke varianten worden uitbesteed aan het migratie-expertiseteam, dat eerst nog de benodigde tooling ontwikkelt. Voor de veel voorkomende gevallen zal het kosteneffectiever zijn om de aanpassing aan het migratie-expertiseteam uit te besteden. Een moeilijke aanpassing die een complexe toolontwikkeling vergt maar die slechts een handvol keren moet worden uitgevoerd, zal echter goedkoper handmatig door de eigen onderhoudsprogrammeurs kunnen worden uitgevoerd. Ook is het mogelijk dat het migratie-expertiseteam een bepaalde analyse automatiseert, maar dat het systeemexpertiseteam de eigenlijke aanpassing uitvoert, op basis van de geautomatiseerde analyseresultaten.

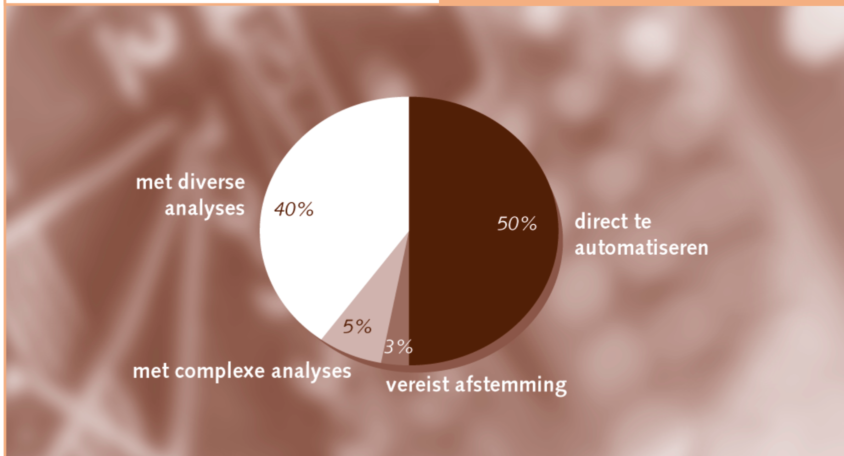
## *Generieke taaltechnologie*

Als de probleemdefinitie in detail vastligt en de taken zijn verdeeld, kan het migratie-expertiseteam de benodigde tooling ontwikkelen. *Generieke taaltechnologie* is hiervoor de aangewezen basis. De ontwikkeling van deze tooling in de standaard programmeertalen neemt anders te veel tijd in beslag of is simpelweg niet mogelijk.

Voordat een aanpassing of analyse kan worden uitgevoerd moet de sourcecode worden geparst (ontleed), zoals men zinnen ontleedt in een onderwerp, persoonsvorm, lijdend voorwerp et cetera. Dit gebeurt aan de hand van een grammatica die de betreffende taal beschrijft. Van sommige talen is de grammatica verkrijgbaar via het internet, maar deze vereisen aanpassingen en zijn meestal niet direct te gebruiken. Bij het opstellen of aanpassen van een grammatica is het goed om rekening te houden met de wijziging die in de sourcecode moet worden doorgevoerd. Het parsen vertaalt de sourcecode in een *parse tree* – een boomstructuur waarin de elementen van de code gerepresenteerd zijn in de structuur zoals ze in de code voorkomen, geclassificeerd naar hun soort. Een voorbeeld van een soort is een statement. Het voordeel van de parse tree is dat nu gezocht kan worden naar bepaalde taalconstructies, bijvoorbeeld een bepaald type statement, om deze vervolgens aan te passen.

De eigenlijke analyses en aanpassingen kunnen met behulp van generieke taaltechnologie worden uitgeschreven als transformatieregels met patronen aan de linker en rechterkant. Als de linkerkant in de (parse tree van de) sourcecode voorkomt, dan wordt deze vervangen door de rechterkant. Door in de patronen variabelen te gebruiken om taalconstructies te representeren, kan worden volstaan met een minimum aantal patronen. In elke moderne teksteditor kan men zoek-en-vervangopdrachten geven, waarbij het ene woord wordt vervangen door het andere. De patronen in de generieke taaltechnologie kunnen worden beschouwd als 'intelligente' patronen, waarbij rekening gehouden wordt met de structuur van de taal (de grammatica) en de context waarin de aanpassing en analyse moeten worden uitgevoerd.

Resultaten van een quickscan 2




### Automatische toepassing

Zodra de aanpassingen zijn gespecificeerd en de tooling is ontwikkeld, kan deze op de sourcecode worden toegepast. Afhankelijk van de grootte van het systeem en de complexiteit van de aanpassing kan dit variëren van minuten tot enkele uren voor een geheel systeem. Tot slot kan het migratie-expertiseteam nog enkele geautomatiseerde controles uitvoeren die specifiek voor deze aanpassing zijn ontwikkeld.
De ontwikkeling en de afstemming van de tooling kan enige tijd in beslag nemen. Gedurende deze periode kan het reguliere onderhoud (het herstellen van productiefouten) gewoon doorgaan. Op het moment dat de tooling beschikbaar is maakt het migratie-expertiseteam een afspraak met het systeemexpertiseteam over het moment waarop zij de sourcecode aanleveren. Vervolgens leveren zij de aangepaste sourcecode op, zodat het regulier onderhoud niet 'bevroren' hoeft te worden en eventuele versieproblematiek wordt vermeden.

### Testen en opleveren

Na dit traject test het systeemexpertiseteam de sourcecode. Het migratie-expertiseteam heeft vaak niet de beschikking over de compiler en de hardware waarop het systeem draait. Daarom moet het systeemexpertiseteam de opgeleverde sourcecode eerst compileren en eventuele meldingen rapporteren. Vervolgens kan het migratieteam de systeemtests uitvoeren.
Het aanleveren, opleveren en testen van de sourcecode van een systeem neemt vaak meer dan één iteratie in beslag. Het is raadzaam om minstens één proefiteratie uit te voeren, om eventuele compileerfouten te kunnen rapporteren. Pas als er geen compileerfouten meer optreden en het systeemexpertiseteam diverse controles heeft uitgevoerd (bijvoorbeeld het controleren van de layout) kunnen ze afspraken maken over de definitieve oplevering.

### Conclusie

Het automatisch aanpassen van grote softwaresystemen biedt veel voordeel ten opzichte van handmatige aanpassing. Vooral voor aanpassingen die op veel plaatsen binnen een grote hoeveelheid sourcecode nodig zijn is een automatische aanpassing zeer waardevol. Een handmatige aanpak is in zulke gevallen erg foutgevoelig en kost veel tijd.
Als eenmaal in de tooling is geïnvesteerd zijn aanpassingen snel en nauwkeurig uit te voeren. Het ontwikkelen van deze tooling is een technisch complexe aangelegenheid. Er zijn diverse gespecialiseerde bedrijven die een dergelijke afgestemde tooling in licentie kunnen aanbieden of die het bulkgedeelte van het aanpassingproject voor hun rekening kunnen nemen.

### Literatuur

Boehm, B.W. (1981). *Software-engineering Economics*. Prentice-Hall.
Jones, C. (2000). *Software Assessments, Benchmarks, and Best Practices*. Addision-Wesley.
Tilley, S.R. & Smith, D.B. (1996). *Perspectives on Legacy System Reengineering*. Software Engineering Institute.

*Dr. Steven Klusener*
*werkt bij de Software Improvement Group en is twee dagen per week gedetacheerd bij afdeling Informatica van de Faculteit der Exacte Wetenschappen aan de Vrije Universiteit Amsterdam. Hij schrijft zijn bijdrage op persoonlijke titel. E-mail: steven@software-improvers.com.*

*Drs. Niels Veerman*
*is aio bij de afdeling Informatica van de Faculteit der Exacte Wetenschappen aan de Vrije Universiteit Amsterdam. E-mail: nveerman@cs.vu.nl.*