# Geautomatiseerd toetsen van acceptatiecriteria

De acceptatie van maatwerkprogrammatuur is vaak gebaseerd op functionele requirements. Kwaliteitsaspecten zoals de onderhoudbaarheid van de programmatuur verdienen echter ook aandacht. Na inproductiename van de programmatuur is de schade – zeker in geval van uitbestede ontwikkelprocessen – heel lastig te herstellen. Geautomatiseerde toetsing van acceptatiecriteria biedt een instrument voor regie en verlicht en objectiveert het werk van applicatiebeheerders.

***Paul Jansen***

Steeds meer bedrijven besteden het maken van maatwerkapplicaties uit aan gespecialiseerde softwarebedrijven. Deze aanpak buiten de deur bespaart kosten, door efficiëntere inzet van personeel en andere bedrijfsmiddelen. Vaak is er echter geen zicht op de kwaliteit die wordt gerealiseerd. Deze moet dan aan het einde van het ontwikkelproces bepaald worden. Er blijft dan meestal alleen een acceptatie over die gericht is op de functionaliteit die vanuit een gebruikersperspectief wordt gemeten. Als de applicatie eenmaal in productie is, dan blijkt vaak dat functionele gebreken snel zijn op te lossen. Het kost wel veel tijd en geld om kwaliteitsgebreken, zoals het niet beschikbaar zijn en slechte performance, te verhelpen. Er dient daarom een strakkere regie te worden gevoerd bij het laten bouwen en in beheer nemen van een applicatie.

Bij outsourcing is het dus de verantwoordelijkheid van de afnemer om vooraf heldere specificaties op te stellen. Daarnaast dient hij na oplevering een gedegen acceptatieprocedure te volgen. Vaak realiseert men zich te laat dat competenties op dit vlak aanwezig moeten zijn om van outsourcing een succes te maken.

Acceptatie kan sneller en objectiever worden uitgevoerd als de toetsing van acceptatiecriteria wordt geautomatiseerd. Een bijkomend voordeel van deze automatisering is dat de acceptatiecriteria al tijdens de ontwikkelfase kunnen worden getoetst en gevolgd. Dat voorkomt verrassingen in een later stadium. Het Eindhovense bedrijf TIOBE (zie kader 'TIOBE') levert gespecialiseerde tools om softwarekwaliteit automatisch te meten en te verhogen.

## GSA-stappenplan

De aanpak van TIOBE kan worden verduidelijkt met het GSA-stappenplan[1,2] (figuur 1). GSA staat voor Generieke en Specifieke Acceptatiecriteria. Steeds meer organisaties hanteren dit stappenplan om op basis van onderkende risico's acceptatiecriteria te bepalen voor nieuw te ontwikkelen informatiesystemen of wijzigingen aan bestaande informatiesystemen. Binnen dit stappenplan worden zeven stappen doorlopen (zie kader 'De GSA-stappen'). De belangrijkste stappen die van toepassing zijn bij de acceptatie van maatwerkprogrammatuur zijn de risicoanalyse (GSA 3.2), het opstellen van de acceptatiecriteria (GSA 4.3 en GSA 4.4) en het toetsen hiervan (GSA 7.5).

| R# | Bedreiging | Kans | Impact | Risico |
|---|---|---|---|---|
| R1 | Dekkingsgraad:<br>De programmatuur is niet volledig getest, waardoor de kwaliteit van het te accepteren object niet vastgesteld is. | H/M/L | De gebreken worden pas in productie zichtbaar. | H/M/L |
| R2 | Instabiliteit:<br>Niet alle mogelijke paden voor alle mogelijke waarden worden getest, waardoor de programmatuur in productie instabiel blijkt te zijn. | H/M/L | De beschikbaarheids- en/of performance-SLA-normen worden niet gehaald. | H/M/L |
| R3 | Uitbreidbaarheid:<br>Het uitbreiden van de programmatuur is lastig vanwege de complexiteit van de sourcecode en is derhalve kostbaar. | H/M/L | De doorlooptijd van wijzigingen is te lang en de wijzigingen kosten te veel geld. | H/M/L |
| R4 | Onderhoudbaarheid:<br>De programmatuur is niet onderhoudbaar. | H/M/L | De oplostijd van incidenten is te lang. Er treden veel gevolgincidenten op na het doorvoeren van de wijzigingen. | H/M/L |

**Tabel 1** Bedreigingen en hun impact

***Risicoanalyse (stap 3)***

Bij het in productie nemen van maatwerkprogrammatuur moeten vier prominente risico's bewust worden genomen of zijn beheerst. Elk risico (hoog, middel en laag) wordt vastgesteld op basis van de bedreiging, de impact van de bedreiging en de kans (hoog, middel en laag) dat de bedreiging werkelijkheid wordt. Een risico is dus de kans maal de impact. In tabel 1 worden vier bedreigingen en de bijbehorende impact benoemd.

*Dekkingsgraad (R1)*

Bij de acceptatie is een belangrijke vraag of de uitgevoerde tests wel betrouwbaar genoeg zijn. Vaak zijn de systeem- en integratietests die worden uitgevoerd voor een systeem het belangrijkste kwaliteitsvangnet voor de uiteindelijke overdracht en vrijgave. De vraag is echter hoeveel procent van de sourcecode van het opgeleverde systeem daadwerkelijk op de proef wordt gesteld door deze tests.

*Instabiliteit (R2)*

Hoe vaak zal de programmatuur crashen? Daar waar tests de functionele aspecten van een applicatie borgen, zullen op de een of andere manier ook de non-functionele aspecten moeten worden gegarandeerd. Het gaat hierbij om de stabiliteit van de programmatuur.

## TIOBE

Het Eindhovense TIOBE is marktleider in Nederland op het gebied van geautomatiseerd kwaliteitsmetingen voor de *embedded software*-industrie. Het bedrijf is opgericht in oktober 2000 met behulp van een investering van het Zwitserse bedrijf Synspace en enkele private financiers. De afkorting TIOBE staat voor *the importance of being earnest*. Dit is ook de naam van een beroemd toneelstuk van Oscar Wilde. De meeste medewerkers van TIOBE zijn oorspronkelijk afkomstig van Philips Research. Daar hebben zij zich verdiept in het automatisch analyseren van broncode. Deze kennis wordt nu ingezet om klanten te helpen.

*Uitbreidbaarheid (R3)*

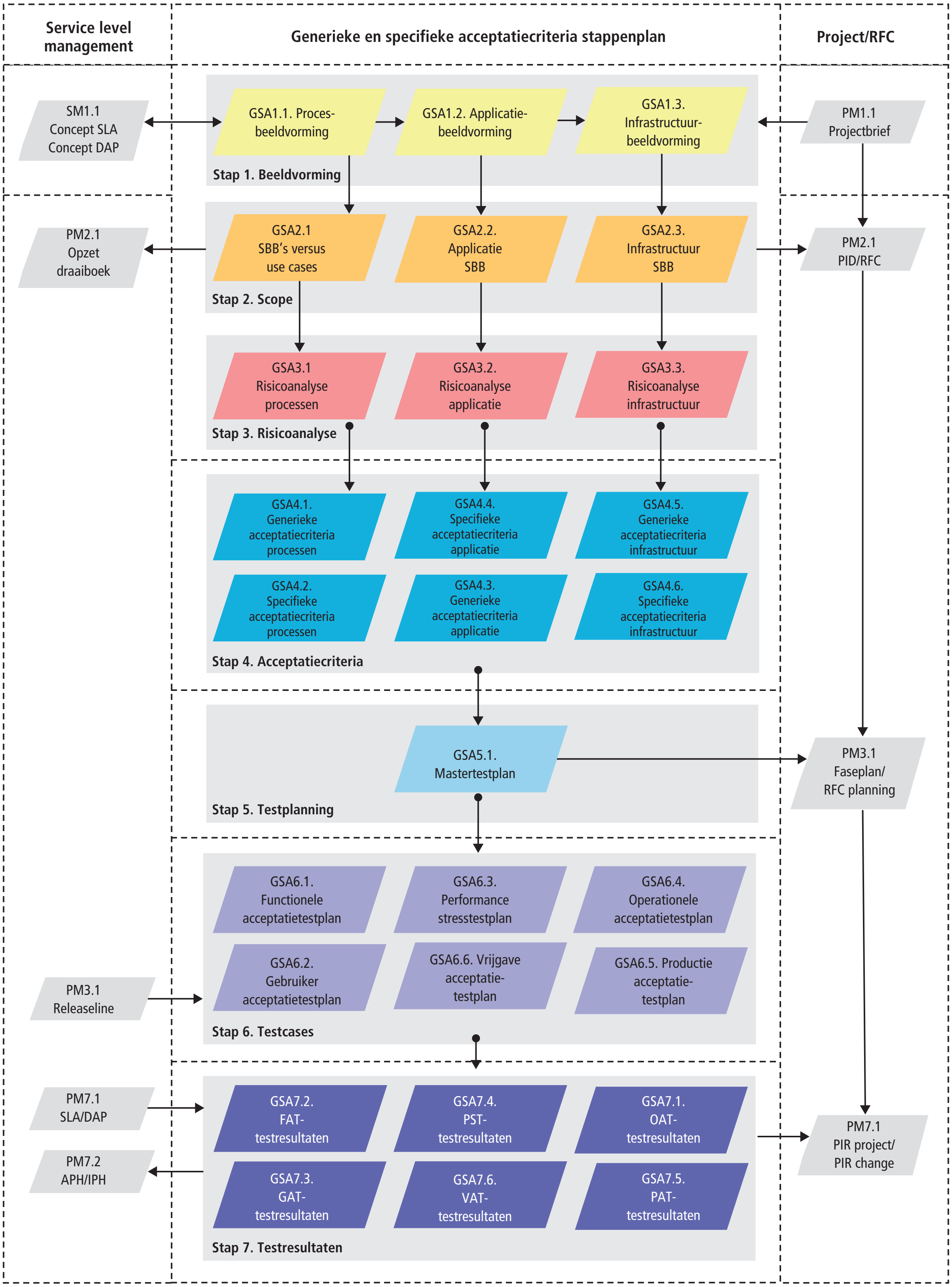
Is de opgeleverde programmatuur uitbreidbaar? De vorige twee risico's hadden betrekking op de externe kwaliteit van de software. Dit is de kwaliteit zoals deze door de eindgebruiker wordt ervaren. Naast de externe kwaliteit speelt ook de interne kwaliteit een belangrijke rol. Wanneer het veel geld kost om een product met een goede externe kwaliteit aan te passen, wordt de rekening uiteindelijk tijdens de onderhoudsfase betaald.

*Onderhoudbaarheid (R4)*

Is de programmatuur begrijpelijk/onderhoudbaar? De norm die aan de onderhoudbaarheidseis gesteld wordt, hangt af van factoren zoals de levensduur, de mate waarin deze ondersteunend is voor een bedrijfskritisch proces, de mate van veranderende behoeften et cetera.

*Acceptatiecriteria (stap 4)*

Voor elk van de onderkende risico's moet gekozen worden tussen risico's nemen en ze beheersen. Voor het beheersen van de risico's moeten tegenmaatregelen worden gedefinieerd. Deze tegenmaatregelen dienen als criteria op basis waarvan het informatiesysteem geaccepteerd



**Figuur 1** Het Generieke en Specifieke Acceptatiecriteria-stappenplan[3,4]

wordt. De acceptatiecriteria zijn dus bedoeld om te toetsen in welke mate de risico's beheerst zijn. In tabel 2 staan bedreigingen genoemd met hun acceptatiecriterium en automatische metriek.

*Dekkingsgraad (SA-01)*
Het risico dat programmatuur niet volledig is getest, kan beheerst worden door het meten van de dekkingsgraad van het testen, de zogeheten *test coverage*. Deze indicator geeft aan hoeveel procent van de opgeleverde code wordt geraakt door een uitgevoerde test. Als de test coverage laag is, zitten er veel ongebruikte features in de over te dragen applicatie en/of is veel functionaliteit gewoon niet getest.
Er bestaan veel verschillende soorten test coverage, waarvan *statement coverage* en *branch coverage* de bekendste zijn (zie kader 'Test coverage'). 100 procent test coverage voor systeem- en integratietesten is na te streven, maar dit is voor grote softwaresystemen praktisch onhaalbaar. Testcoveragegoeroe Steve Cornett stelt daarom dat 80 procent een realistische doelstelling is.[5] Om test coverage te meten dient codecoveragegereedschap gebruikt te worden. Enigszins simpel gezegd voegt dit soort tooling printstatements toe aan een applicatie. Daarmee kan bijgehouden worden welke code geraakt wordt en welke niet. Bekende voorbeelden van testcoveragegereedschappen zijn Cobertura voor Java en NCover voor C#.

*Instabiliteit (SA-02)*
Instabiliteit van een informatiesysteem is desastreus voor SLA-normen zoals beschikbaarheid en performance. Om dit risico in kaart te brengen wordt meestal een combinatie van metrieken gebruikt. De bekendste zijn checks op crashes als gevolg van zogeheten *null dereferences*, *array out of bounds exceptions*, oneindige loops en delen door nul. Dit type fouten in programmatuur is helaas niet op een waterdichte manier vast te stellen. Om crashes te

## *De GSA-stappen*

Stap 1: Beeld. De eerste stap is het bepalen van het beeld van de bedrijfsprocessen waarvoor het informatiesysteem wordt gebouwd of veranderd. Vervolgens wordt een plaat gemaakt van de applicatie en de omgeving waarmee de applicatie communiceert. Als derde onderdeel van de beeldvorming wordt een plaat gemaakt van de infrastructuur die wordt geraakt door het informatiesysteem.

Stap 2: Scope. In deze stap wordt een decompositie gemaakt van de applicatieplaat en de infrastructuurplaat in de vorm van bouwstenen (System Building Blocks). Vervol gens worden de *use cases* van het informatiesysteem afgebeeld op de onderkende bouwstenen.

Stap 3: Risico's. De risico's worden op basis van een gestructureerde analyse bepaald en toegekend aan een eigenaar. Voor elk risico wordt een te nemen preventieve en/of correctieve tegenmaatregel gedefinieerd. Tevens worden de risico's geclassificeerd per bouwsteen (stap 2).

Stap 4: GSA. Deze stap is bedoeld om vast te stellen wanneer de risico's zijn beheerst. Dit gebeurt door functionaliteiteisen, kwaliteiteisen en beheereisen te formuleren in de vorm van zowel generieke als specifieke acceptatiecriteria. Tevens worden de testsoort, de teststrategie en de testplanning in tijd & geld per acceptatiecriterium bepaald.

Stap 5: Testplan. In deze stap wordt het mastertestplan opgesteld. Hierin staan het testdomein, de testbasis, de testorganisatie, de overall testplanning, de testspecificatiewijze, de bevindingenprocedure en de risicodekkingsgraadanalyse beschreven.

Stap 6: Testcase. De acceptatietestplannen beschrijven de testscenario's/testcases om vast te stellen of aan de acceptatiecriteria wordt voldaan. Het betreft: Operationeel Acceptatie Testplan (OAT), Functioneel Acceptatie Testplan (FAT), Gebruikers Acceptatie Testplan (GAT), Performance StressTestplan (PST) en Productie Acceptatie Testplan (PAT) en Vrijgave Acceptatie Testplan (VAT).

Stap 7: Testrun. Dit is de laatste stap in het GSA-stappenplan. Hier worden testscenario's/testcases die in stap 6 zijn bepaald uitgevoerd ter acceptatie van het nieuwe of gewijzigde informatiesysteem. De testresultaten worden voorgelegd aan de projectmanager en de changemanager. De changemanager bespreekt deze resultaten in het CAB met de bij de acceptatie betrokken stakeholders.

| R# | Bedreiging | GSA | Acceptatiecriterium | Automatische metriek |
|---|---|---|---|---|
| R1 | Dekkingsgraad | SA-01 | De test coverage bedraagt minimaal 80 procent. | Test coverage |
| R2 | Instabiliteit | SA-02 | De programmatuur bevat geen enkele dynamische overtreding, zoals vastgesteld door de gehanteerde tooling. | *null dereference*, *array out of bounds exceptions*, oneindige loops en delen door nul |
| R3 | Uitbreidbaarheid | SA-03 | De gemiddelde cyclomatische complexiteit voor functies is kleiner dan 5. | Cyclomatische complexiteit |
| R4 | Onderhoudbaarheid | SA-04 | De confidencefactor is minimaal 80 procent | Regels uit de gehanteerde codeerstandaard |

**Tabel 2** Bedreigingen met hun acceptatiecriterium en metriek

voorkomen moeten alle mogelijke paden voor alle mogelijke waarden worden afgetest.

In 2000 heeft er op dit terrein een doorbraak plaatsgevonden, onder aanvoering van Patrick Cousot van het ESN in Parijs.[6] Met behulp van zijn technologie, abstracte interpretatie geheten, is het mogelijk om een redelijke overdekking van alle paden uit te rekenen zonder een applicatie ook daadwerkelijk uit te voeren. Abstracte interpretatie probeert net als schaakcomputers het aantal mogelijkheden in te perken door onwaarschijnlijke combinaties uit te sluiten. In de afgelopen jaren is abstracte interpretatie volwassen geworden en wordt zij in toenemende mate ingezet om de stabiliteit van systemen door te meten.

Voorbeelden van producenten van gereedschappen die in staat zijn om omissies automatisch te detecteren en dus bovengenoemde crashes te voorkomen, zijn Klocwork en Coverity. Het acceptatiecriterium dat aantoont of dit risico beheerst is, luidt: de programmatuur bevat geen enkele dynamische overtreding, zoals vastgesteld door de gehanteerde tooling.

### *Uitbreidbaarheid (SA-03)*

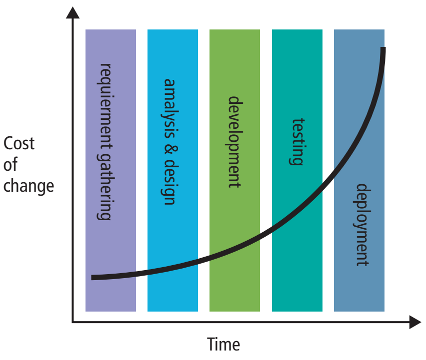
Om te bepalen of programmatuur eenvoudig uitbreidbaar is, is het belangrijk vast te stellen wat de impact van wijzigingen is. Wanneer een kleine aanpassing van de code invloed heeft op veel andere onderdelen van de programmatuur, dan is deze moeilijk uitbreidbaar.

Dit komt meestal doordat de executiepaden in de programmatuur ondoorzichtig zijn en deze veel onderlinge afhankelijkheden bevat. Deze ondoorzichtigheid is op verschillende manieren te meten.

Er kan worden gekeken naar ongecontroleerde sprongen in de code. Het veelvuldig gebruik van het *go to*-statement in veel talen is een berucht voorbeeld.

Ook modernere varianten zoals *continue* en *break* leiden tot onduidelijkheid.

Ook het aantal afhankelijkheden tussen programmamodules is een maat voor ondoorzichtigheid. Als modules veel onderlinge verbanden hebben, betekent dat dat een wijziging aan de ene module ook een wijziging aan een andere module kan vereisen.



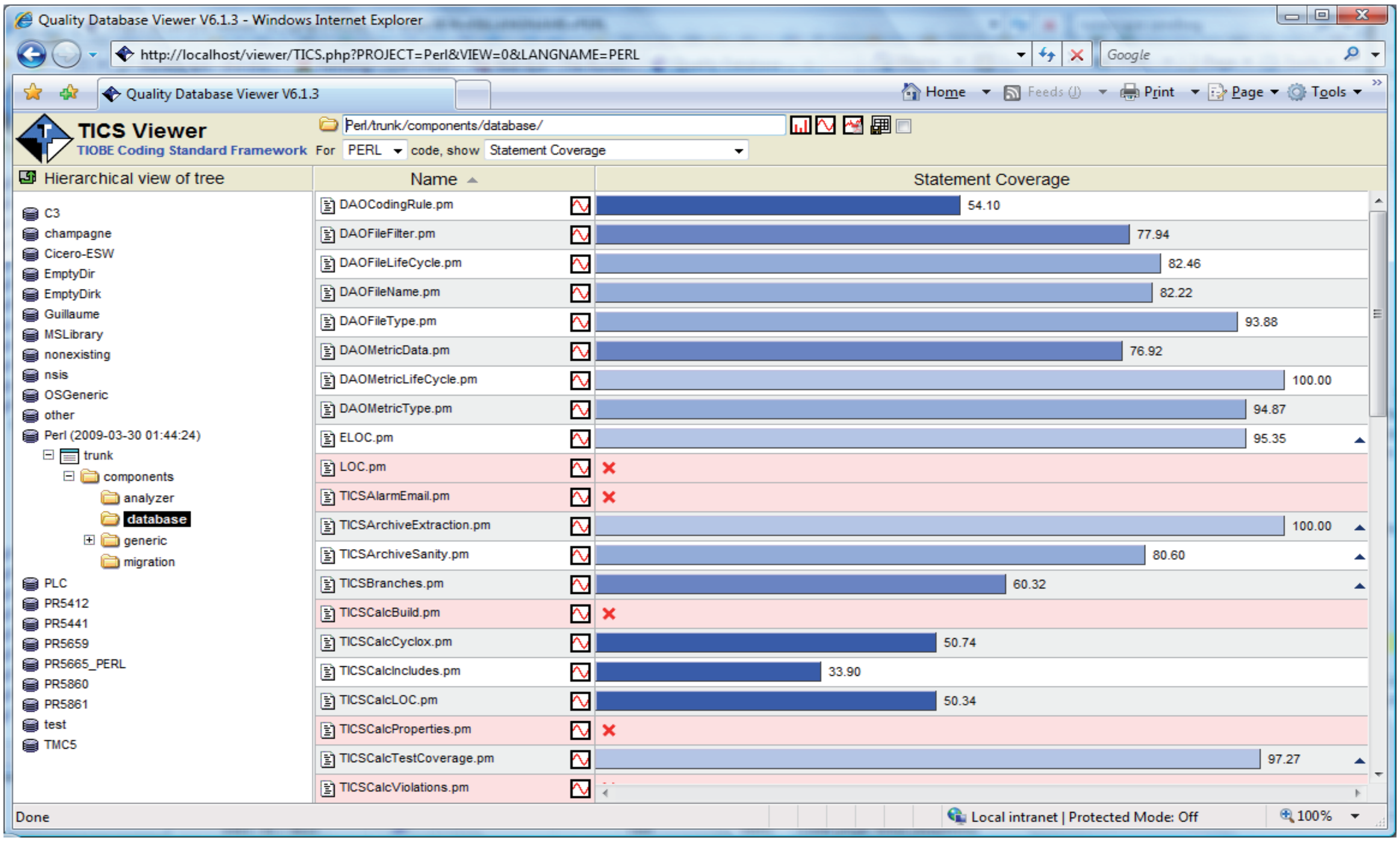
**Figuur 2** De curve van Barry Boehm

Dan is er nog de cyclomatische complexiteit, ontwikkeld door Thomas McCabe.[7] Kort gezegd meet deze metriek het aantal executiepaden per functie. Als er heel veel manieren zijn om een functie te doorlopen, is het gedrag van de functie complex en dus moeilijk aan te passen.

De cyclomatische complexiteit is het meest gehanteerde acceptatiecriterium om uitbreidbaarheid te toetsen. Indien deze per programmaklasse wordt gemeten, is een gemiddeld aantal paden van tien toelaatbaar; wanneer per functie wordt gemeten, is een gemiddelde van vijf nog aanvaardbaar. Om cyclomatische complexiteit te meten worden meestal codecheckers gebruikt. Deze lezen programmatuur in en slaan deze dan op in een abstracte structuur. Een veelgebruikt gereedschap op dit gebied voor zowel Java als C# is SourceMonitor.

### *Onderhoudbaarheid (SA-4)*

De onderhoudbaarheid van een applicatie geeft aan in hoeverre een stuk software begrijpelijk en toegankelijk is voor een andere ontwikkelaar of applicatiebeheerder. Omdat hier vaak persoonlijke voorkeuren aan ten grondslag liggen, is de onderhoudbaarheid niet zomaar met behulp van een paar metrieken te bepalen. Een holistische aanpak wordt daarom aangeraden, waarbij een codeerstandaard wordt opgesteld met



| Name | Statement Coverage |
|---|---|
| DAOCodingRule.pm | 54.10 |
| DAOFileFilter.pm | 77.94 |
| DAOFileLifeCycle.pm | 82.46 |
| DAOFileName.pm | 82.22 |
| DAOFileType.pm | 93.88 |
| DAOMetricData.pm | 76.92 |
| DAOMetricLifeCycle.pm | 100.00 |
| DAOMetricType.pm | 94.87 |
| ELOC.pm | 95.35 |
| LOC.pm | ✗ |
| TICSAlarmEmail.pm | ✗ |
| TICSArchiveExtraction.pm | 100.00 |
| TICSArchiveSanity.pm | 80.60 |
| TICSBranches.pm | 60.32 |
| TICSCalcBuild.pm | ✗ |
| TICSCalcCyclox.pm | 50.74 |
| TICSCalcIncludes.pm | 33.90 |
| TICSCalcLOC.pm | 50.34 |
| TICSCalcProperties.pm | ✗ |
| TICSCalcTestCoverage.pm | 97.27 |
| TICSCalcViolations.pm | |

**Figuur 3** Screenshot TICS Viewer Test Coverage

normaliter meer dan honderd verschillende regels.
Met deze regels kan bijvoorbeeld worden gecheckt of commentaar is toegevoegd op de juiste plekken, of de gehanteerde programmeerstijl consistent is en of de juiste naamconventies zijn gebruikt. Het meten van het aantal overtredingen van een codeerstandaard is dan een afspiegeling van de onderhoudbaarheid van de programmatuur. Niet elke overtreding weegt echter even zwaar en elke codeerstandaard is anders.

### *Confidencefactor*

Met de confidencefactor[8] die TIOBE Software samen met architecten van Philips en Océ heeft ontwikkeld, is toch een overkoepelend beeld te krijgen van de mate waarin een applicatie voldoet aan een codeerstandaard. Deze factor geeft met een percentage tussen 0 en 100 aan hoe onderhoudbaar de code is. Wanneer een applicatie een confidence factor van 100 procent heeft, is er geen enkele overtreding van de codeerstandaard; een confidencefactor van 0 procent betekent dat zo'n beetje op elke plek elke mogelijke overtreding is gemaakt die gemaakt kon worden. De factor houdt rekening met de grootte van de applicatie, de zwaarte van de geconstateerde overtredingen en de hoeveelheid regels waaruit een standaard bestaat.
De factor wordt normaliter elke dag berekend. 's Nachts wordt de overdag aangemaakte en/of aangepaste sourcecode geanalyseerd op kwaliteitscriteria. De crux van deze factor is dat de regievoerder op eenvoudige wijze grip krijgt op een deel van de kwaliteitbeheersing, zonder dat hij diepgaande kennis nodig heeft van applicatietechnologie. Om afwijkingen van de confidencefactor te beoordelen is specifieke vakkennis nodig. Deze is het domein van de systeemontwikkelaars en applicatiebeheerders.
Al een aantal jaar controleert TIOBE elke dag meer dan honderd miljoen regels programmatuur voor meer dan 250 applicaties op softwarefouten. Op basis van deze gegevens blijkt dat een confidencefactor van 80 procent realistisch is zonder onderhoudbaarheid een doel op zich te maken.
Veelgebruikte gereedschappen om codeerstandaarden automatisch te checken zijn bijvoorbeeld PMD en FindBugs voor Java en FxCop voor C#.

### *Toetsing (stap 7)*

De acceptatiecriteria zijn zodanig gekozen dat ze automatisch gemeten kunnen worden. Dit heeft als grote voordeel dat er geen noemenswaardige inspanning

## Test coverage

Het meten van test coverage is een steeds meer voorkomende techniek om te bepalen of uitgevoerde tests voldoende dekkingsgraad hebben. Dit kunnen unittests zijn, maar ook integratie- of systeemtests. Met behulp van speciale tools wordt het te testen systeem op een speciale manier gebouwd (geïnstrumenteerd), waardoor precies kan worden bijgehouden welke stukken broncode tijdens het testen niet worden geraakt.

De twee belangrijkste vormen van test coverage zijn statement coverage, het percentage statements dat wordt geraakt tijdens testen, en branch coverage, het percentage mogelijke beslissingen die in een functie zijn afgetest.

Onderstaand pseudocode voorbeeld maakt de twee verschillende vormen van test coverage duidelijk.

```
int KlantenKorting(int aantal, int krediet) {
 int korting = 1;
 if (krediet <= 0) {
 korting = 0;
 }
 if (aantal > 50) {
 korting = 2;
 }
 return korting;
}
Om deze functie te testen, kun je de volgende unittest schrijven:
boolean TestKlantenKorting() {
 return
 KlantenKorting(0, 0) == 0 && // test 1
 KlantenKorting(61, 3) == 2 // test 2
 ;
}
```

De statement coverage is hier 100 procent, omdat test 1 de statements van de eerste *if*-tak doorloopt en test 2 die van de tweede. De branch coverage is echter maar 50 procent. Van de vier mogelijke paden door de code worden er maar twee getest. Er is geen test die allebei de if-takken doorloopt (bijvoorbeeld KlantenKorting(80, -3)) of die geen enkele if raakt (bijvoorbeeld KlantenKorting(1, 1)).

en kennis van beheerders nodig is om de acceptatiecriteria te toetsen. Deze toetsing kan dus hoogfrequent worden uitgevoerd. Met andere woorden: door de juiste tooling te gebruiken is het mogelijk om de risico's gedurende het gehele ontwikkeltraject van de programmatuur tot aan uiteindelijke acceptatie te beheersen. Het wel of niet voldoen aan de acceptatiecriteria en de mate waarin hiervan wordt afgeweken kunnen namelijk al tijdens de ontwikkelfase worden vastgesteld.

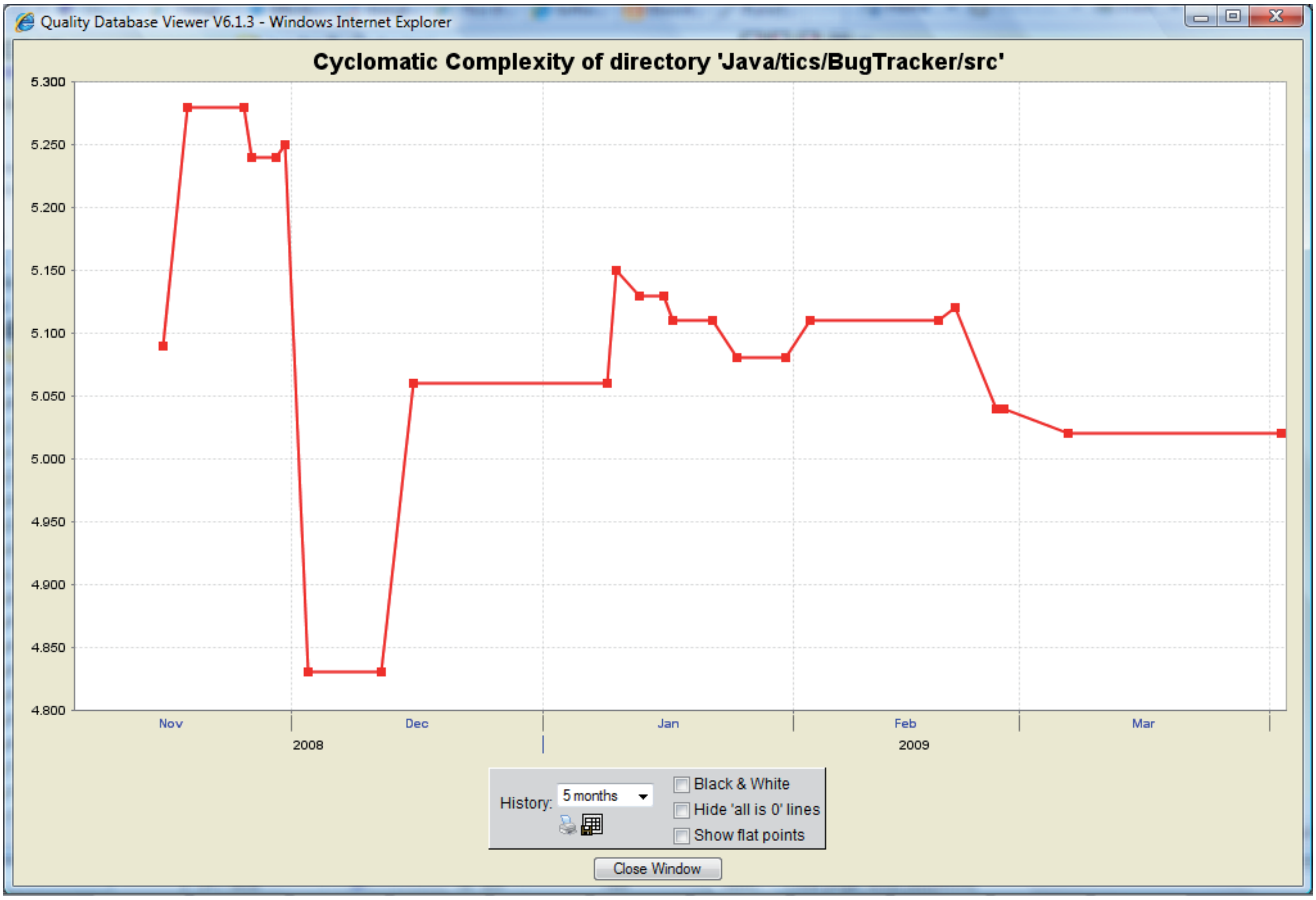
Dergelijke geautomatiseerde toetsing van acceptatiecriteria tijdens ontwikkeling leidt ertoe dat er al in een heel vroeg stadium kan worden bijgestuurd indien de gestelde normen dreigen te worden overtreden. Uit onderzoek van Barry Boehm[9] is gebleken dat de kosten van *rework* op deze manier exponentieel kunnen worden verminderd: rework kost tien keer meer per latere fase waarin een afwijking van een van de criteria wordt hersteld (figuur 2).

Een belangrijke voorwaarde voor het continu toetsen van acceptatiecriteria is dat hiervoor tooling beschikbaar is. De kwaliteitsdata die deze metingen opleveren, moeten wel nog worden bewerkt voordat deze bruikbaar is.

Eerst moeten de kwaliteitsdata uit de broncode worden geëxtraheerd. Deze data-acquisitie gebeurt met compilertechnieken. Eerst wordt de broncode ingelezen en in een abstract model vertaald. Vervolgens vinden er allerlei berekeningen plaats op dit abstracte model. Alle metrieken die in GSA-stap 4 aan bod zijn gekomen kunnen op deze manier worden bepaald. Vervolgens moeten de verzamelde kwaliteitsdata worden opgeslagen. Zo kan de kwaliteit van de programmatuur en het wel of niet halen van de acceptatienormen worden gevolgd gedurende de ontwikkelfase. Vervolgens moeten de data nog helder en toegankelijk worden gevisualiseerd. Dit zorgt ervoor dat de gemeten waarden ook daadwerkelijk worden gebruikt voor verbetering.

De meeste bedrijven richten zich bij het inrichten van een acceptatiesysteem alleen op de data-acquisitie. In de praktijk is gebleken dat dit onvoldoende houvast biedt om het toetsen van de acceptatiecriteria te verankeren in de ontwikkelorganisatie. De belangrijkste reden hiervoor is dat de gemeten data zonder dataopslag en datavisualisatie niet centraal beschikbaar zijn.

Een integraal acceptatiesysteem doorloopt bovenstaande drie fasen op gezette tijden (bijvoorbeeld elke nacht). De gewijzigde programmatuur wordt opnieuw geanalyseerd en de resultaten hiervan worden in een database opge-



History: 5 months

☐ Black & White
☐ Hide 'all is 0' lines
☐ Show flat points

**Figuur 4** Screenshot TICS viewer Trends Cyclomatic Complexity

slagen. Via een webapplicatie kunnen de relevante gegevens dan ontsloten worden.

## TICS

Het TICS-framework van TIOBE is een voorbeeld van een systeem dat alle onder GSA stap 4 gedefinieerde acceptatiecriteria automatisch volgt. In figuur 3 is een screenshot van de TICS-kwaliteitsviewer te zien. Links is de applicatiestructuur te zien. Op topniveau staan alle projecten waarvoor de acceptatiecriteria gevolgd worden. Het project dat nu geopend is heet Perl. Binnen het project is de folderstructuur te zien. Uiteindelijk zijn op het laagste niveau ook de individuele files te zien waarin de broncode is te vinden. Voor elk van de projecten, folders en files kunnen de acceptatiecriteria worden opgevraagd.

De screenshot laat de statement coverage zien van alle files en folders in de folder ‘database’ van het Perl-project. Hieruit is af te lezen dat de file ‘DAOMetricLifeCycle.pm’ een statement coverage heeft van 100 procent, terwijl ‘DAOCodingRule.pm’ slechts een coverage kent van 54,10 procent. Verder is te zien dat voor sommige files geen test coverage kon worden berekend (rood kruis). Dit kan bijvoorbeeld komen doordat de bijbehorende tests faalden of doordat er nog helemaal geen tests zijn voor de betreffende file.
Aangezien alle kwaliteitsdata in de database zijn opgeslagen, kunnen er ook trends worden getoond. Figuur 4 laat de cyclomatische complexiteit (gemiddeld aantal executiepaden per functie) zien van een bepaalde folder van een project. Uit de trend is af te leiden dat de cyclomatische complexiteit langzaam afneemt, wat een goed teken is. De norm van 5 is nog niet gehaald, maar men zit er nog maar net boven, namelijk op 5,03.

## Best practices

Het is aan te raden de volgende, op de praktijk gebaseerde adviezen voor het geautomatiseerd accepteren van informatiesystemen op te volgen.

- Begin in een zo vroeg mogelijk stadium met het geautomatiseerd meten van de acceptatiecriteria en sla de meetwaarden op in een database.

- Laat zowel de ontvangende partij als de leverende partij de metingen uitvoeren.
- Meet de acceptatiecriteria tijdens zowel de ontwikkelfase als de beheerfase.
- Zorg ervoor dat de publicatie van de meetgegevens bekend en toegankelijk is.
- Hanteer de confidencefactor als acceptatiecriterium bij het uitbesteden van applicatiebouw en als SLA-norm voor het applicatiebeheer.
- Zorg dat de confidencefactor bij een change en/of release in elk geval niet lager wordt bij het herstellen van bestaande programmatuur met een confidencefactor van minder dan 80. Het is vaak te kostbaar om de kwaliteit in zo'n geval in één keer te verbeteren.
- Hanteer de confidencefactor niet als doel, maar als middel om de kwaliteit van applicatieprogrammatuur te meten.

*Hierbij dank ik Steven van der Linden, Louis van Hemmen, Leon Meuldijk, Marijke Somers, Mark Smalley, Stephen Klomp en Bart de Best voor het reviewen van dit artikel.*

## Voetnoten

1 Best, B de. 'Afleiden en toepassen acceptatiecriteria', IT Beheer Magazine 2005, nr. 7.
2 Best, B de. 'Acceptatiecriteria in de praktijk', IT Beheer Magazine 2006, nr. 1.
3 Best, B de. 'Risicobeheersing bij nieuwe functionaliteit', IT Beheer Magazine 2007, nr. 1.
4 Best, B de. 'Verzekeraar beheerst de risico's van SOA', IT Beheer Magazine 2007, nr. 9.
5 Cornett S. 'Minimum Acceptable Code Coverage'. www.bullseye.com/minimum.html.
6 Cousot P. 'Semantics and Abstract Interpretation', www.di.ens.fr/~cousot/abstract_interpret.shtml.
7 McCabe T. 'A Complexity Measure' in: IEEE Transactions on Software Engineering 1976, vol. 2, nr. 4.
8 Jansen P, Krikhaar R, Dijkstra F. 'Towards a Single Software Quality Metric', www.tiobe.com/content/paperinfo/DefinitionOfConfidenceFactor.html.
9 Boehm B. Software Engineering Economics, Prentice Hall, 1981.

## Literatuur

- B. de Best, *Acceptatiecriteria*, SDU uitgevers, 2006, ISBN 90 395 24998.
- B. de Best, *Beheren onder Architectuur*, NGN, 2008, ISBN 9789081338011.

*Paul Jansen is managing director van TIOBE (paul.jansen@tiobe.com).*