

Complexiteit zonder fouten is utopie

Om de kwetsbaarheid van softwaresystemen aan te pakken is het belangrijk naar de broncode te kijken. Het toepassen van broncodemetrieken, concludeert Arie van Deursen, helpt bij het in de hand houden van fragiliteit. Peter Kaptein wijst op het belang van codegeneratoren. Dit is de zesde aflevering van een serie over fragile software. Eerdere artikelen verschenen 5, 12, 19 en 26 november en 3 december.



ILLUSTRATIE: CLIFFHANGER

Voorkomen is beter dan genezen

Wanneer is een softwaresysteem fragiel? Fragiel betekent breekbaar, broos, en makkelijk (per ongeluk) kapot te maken. Maar programmatuur per ongeluk kapot maken: Wanneer doe je dat? Dit gebeurt eigenlijk alleen tijdens onderhoud. Je probeert een nieuwe use case toe te voegen aan een bestaand systeem en moet daartoe op enkele plekken wijzigingen aanbrengen. Daarbij is het moeilijk te overzien wat precies de afhankelijkheden zijn tussen allerlei modules, databestanden en processen, en het gebeurt dan snel dat je een essentieel verband tussen bijvoorbeeld twee componenten per ongeluk verbreekt. Zo bezien is fragiliteit een speciale vorm van slechte onderhoudbaarheid. Een fragiel systeem is dermate moeilijk te begrijpen dat de kans groot is dat een software engineer ongewild allerlei bestaande functionaliteit kapot maakt bij het doorvoeren van een enkele kleine wijziging.

Fragiliteit is gerelateerd aan complexiteit. Immers, hoe complexer een systeem is, hoe moeilijker het is aan te passen. Veel systemen zijn echter overmatig complex: de interne complexiteit is vele malen groter dan nodig zou zijn om het externe gedrag (de functionaliteit) te realiseren. Soms is dit met reden gedaan (om extra flexibiliteit te verkrijgen), maar doorgaans is de extra complexiteit in de loop der tijd ontstaan, en is het oorspronkelijk zuivere ontwerp geërodeerd door de vele wijzigingen die het systeem heeft moeten ondergaan. Over het algemeen is deze extra complexiteit de belangrijkste veroorzaker van de fragiliteit van softwaresystemen. We kennen fragile systemen als 'legacy': systemen die we van onze voorgangers gekregen hebben en die een geschiedenis van jaren slecht onderhoud achter zich hebben. Maar fragiliteit is niet alleen een kenmerk van oude systemen: ook een nieuwe applicatie met een ondoordacht ontwerp kan zeer duur in onderhoud zijn.

Om die reden definiëren Brodie en Stonebraker (in 'Migrating Legacy Systems', 1995) een legacystelsel ook niet als een oud systeem, maar als een 'systeem dat zich verzet tegen verandering'. Als kenmerk geven ze aan dat dergelijke

systemen over het algemeen 'brittle' zijn: broos, breekbaar en fragiel.

Kadaster

Valt er iets te doen aan fragiliteit? Een mogelijkheid is renovatie. Zo heeft bijvoorbeeld het Kadaster besloten zijn primaire informatiesystemen te vernieuwen, en daartoe een incrementele aanpak geformuleerd die de bestaande systemen stap voor stap omzetten in een componentgebaseerde oplossing. Waar mogelijk wordt het renoveren van een onderdeel gecombineerd met het opleveren van nieuwe of aangepaste functionaliteit, om zo het draagvlak voor het

Broncodemetrieken leveren inzicht in kwetsbaarheid systemen

uitvoeren van de renovatie bij de diverse belanghebbenden te waarborgen. Het belang van een dergelijke incrementele renovatie-aanpak wordt verder toegelicht in het eerder genoemde werk van Brodie & Stonebraker over legacy-systemen.

Maar, is het wellicht ook mogelijk de kwaal te voorkomen in plaats van te genezen met dure renovaties? Hier biedt een onderdeel van de opkomende wendbare (agile) methodes zoals extreem programmeren en Scrum een mogelijke oplossing. Centraal in extreem programmeren staat 'continuus refactoring': het voortdurend bijhouden van het ontwerp en de code van een systeem. In elke iteratie is expliciet tijd beschikbaar om verbeteringen, en met name simplificaties, aan de interne structuur van de programmatuur aan te brengen. Het motto hierbij is: 'Do the simplest

thing that could possibly work.'

Het nu al klassieke boek over refactoring van Martin Fowler (Addison-Wesley, 1999) bevat een reeks van kleine verbeteringsstappen op ontwerp- en code-niveau. Ook grotere aanpassingen aan het ontwerp worden als keten van dergelijke kleine stappen geformuleerd. Fowler erkent wel dat het herfactoriseren in de praktijk vaak achterwege wordt gelaten. Hij omschrijft dit als het opbouwen van een schuld. Je kunt ermee weggelopen, bijvoorbeeld vlak voor een urgente release van je product. Maar zolang je niet terugbetaalt en de benodigde vereenvoudigingen aanbrengt,

betaal je rente in de vorm van de extra kosten die elke wijziging met zich meebrengt.

Metrieken

De fragiliteit van een systeem in ontwikkeling of onderhoud kan alleen maar onder controle worden gehouden als deze tastbaar wordt gemaakt. Helaas bestaat er geen metriek die de fragiliteit van een systeem uitdrukt in een cijfer tussen 1 en 10. Toch zijn er wel mogelijkheden. Allereerst kan een poging worden ondernomen om achteraf de gevolgen van fragiliteit te meten. Hierbij kan het gaan om tijdsduur of kosten per afgehandeld veranderingsverzoek of probleemrapport, het aantal openstaande problemen, het aantal nieuwe problemen dat per release geïntroduceerd wordt enzovoorts. Dergelijke metingen kunnen helpen bij de bewustwording

(we moeten er wat aan doen) en strategie bepaling (welke prioriteiten moeten we stellen?).

Wat je op deze manier meet zijn onderhoudskosten achteraf. Maar dergelijke metingen zeggen weinig over de oorzaak (het kan fragiliteit zijn, maar ook een slecht proces). Bovendien helpt informatie achteraf niet bij het voorkomen van problemen. Daarom is het belangrijk ook te kijken naar de broncode van het systeem zelf. Er zijn diverse broncodemetrieken bekend (zoals koppeling of McCabe) waarvan empirisch is aangetoond dat deze gecorreleerd zijn aan onderhoudbaarheid en fragiliteit. Dergelijke metrieken kunnen volautomatisch uit de broncode worden afgeleid. Het bijhouden hiervan brengt dus weinig kosten met zich mee, maar levert wel inzicht op in de kwetsbaarheid van een systeem.

De absolute uitkomsten van dergelijke metingen en metrieken zijn vaak moeilijk te interpreteren, en niet altijd bijzonder betekenisvol. Maar de trends hiervan over een langere levensduur van het systeem vormen wel degelijk een indicatie of het de goede of de slechte kant opgaat met de fragiliteit, en of deze bijvoorbeeld in de huidige release sneller is gestegen dan in alle voorgaande. Het is dus zaak een systeem voortdurend te monitoren, in plaats van eenmalig een puntmeting uit te voeren. Het actief monitoren van onderhoudbaarheid en fragiliteit kan op individueel projectniveau, om zo de kwaliteit van een enkel ontwikkeld systeem in de gaten te houden. Beter nog is het dit ook op hoger managementniveau te doen, ten einde de fragiliteit van de volledige softwareportfolio van een organisatie te monitoren, en zo de kosten te beheersen die onvermijdelijk komen bij overmatige fragiliteit.

ARIE VAN DEURSEN

AG-10-12-'04

Arie van Deursen is onderzoeksleider op het gebied van softwarerenovatie aan het Centrum voor Wiskunde en Informatica (Amsterdam) en deeltijdhoogleraar software engineering aan de Technische Universiteit Delft. Hij is mede-oprichter van de Software Improvement Group BV, te Dienen.

Codegenerator belangrijk wapen

Vanwege economische en idealistische stromingen (om twee factoren te noemen) is er constant sprake van een concurrentiestrijd en ontwikkeling van nieuwe inzichten. Het is onder meer daardoor onvermijdelijk dat software steeds complexer wordt en door die toenemende complexiteit steeds fragieler. Software moet beter, sneller, mooier, constant nieuwe functionaliteiten blijven bieden en veel eenvoudiger in gebruik worden om niet ten onder te gaan. De risico's zijn navenant. Door het gebruik van een codegenerator kan men die complexiteit terugbrengen naar het werkveld van structuren, eenvoudige patronen en een verzameling regels die voortvloeien uit – bijvoorbeeld – een databasemodel.

Elk softwarepakket bevat fouten. Constructiefouten, ontwerpfouten, onvolledige afhandeling van situaties. Deze fouten zijn primair het gevolg van de mens: vermoeidheid, onvolledig begrip van de situatie, tijdgebrek, een slecht ontwerp, onvoldoende ervaring etcetera. Het overgrote deel van ontwikkelingsprojecten heeft een beperkt budget. Of dit nu een paar duizend- of een paar miljoen euro is. Verder bestaat er geen oneindige bron van experts die per direct beschikbaar zijn, blijven specificaties en inzichten gedurende het ontwikkelingsproces veranderen omdat de wereld niet stilstaat en is testen te vaak een ondergeschoven kindje.

Gebruikers moeten uitstapjes kunnen maken naar andere onderdelen van het pakket, daar specifieke handelingen kunnen uitvoeren, begrijpen waar ze zijn en bij terugkeer naar het oorspronkelijke scherm al die veranderingen bijvoorbeeld terug kunnen vinden in selectielijsten om verder te kunnen werken: 'Ik voer even uw gegevens in. Wie is uw tandarts? OK, die hebben we niet in het systeem zitten. Heeft u een moment? Dan voeg ik die even toe.'

Waar abcd de basisacties zijn, volgt de realiteit vaak paden als cabdad of cabbad volgens de associatieve logica van de ge-

de applicatiecode en vormen een conclusie van patronen, structuren, definities en toegepaste logica.

Met deze vijf onderdelen is in de praktijk via de codegenerator ongeveer 70 tot 90 procent van alle code voor schermen en interacties kant-en-klaar op te leveren. Wat overblijft zijn de applicatie- en bedrijfslogica, verpakt in businesslogica.

Gegeneerde code achteraf aanpassen is gevaarlijk. Elke nieuwe run kan deze code volledig wegvagen, waardoor specifieke aanpassingen verloren gaan. De businesslogica (denk aan workflowprocessen en financiële transacties) wordt daarom in de betere toepassingen als een externe module aangeropen: 'Onderbreek het proces, roep module XYZ aan, voer de handelingen uit en vervolg de stappen in mijn gegeneerde code als je klaar bent.'

Oplossingen

Wat lossen we op? Ten eerste: de complexiteit van de applicatie wordt voor een groot deel teruggebracht naar het werkveld van definities en structuren. Code wordt grotendeels automatisch gegeneerd. Applicatielogica (ook non-linear) is in veel gevallen impliciet aan de structuur van de database en wordt voor het grootste deel automatisch aangemaakt. Ten tweede: het ontwerptraject wordt verkort omdat veel zaken al worden gedekt door het grondwerk dat geleverd wordt door de codegenerator. Ten derde: wijzigingen in de applicatiestructuur hebben niet meer de grote gevolgen zoals bij handwerk het geval is. Het leeuwendeel van de veranderingen wordt opvangen en doorgevoerd door de codegenerator. Businesslogica is extern en duidelijk omlind. Ten vierde: programmeurs kunnen zich primair richten op het bouwen en testen van de applicatie- en processpecifieke businesslogica, wat de kwaliteit van die onderdelen ten goede komt, omdat er minder werk is aan 'copy, paste & change'-processen. Codegeneratoren vormen slechts één van de mogelijke invullingen. Runtime-op-

Gegeneerde code achteraf aanpassen is gevaarlijk

bruiker: wat vaak lijnrecht tegen de logische gegevensstroom in de database gaat. Veel handwerk om te programmeren. De patronen volgen echter in grofweg 98 van de 100 praktijkgevallen de paden van het datamodel waarop de applicatie is gebaseerd.

Patronen

Doordat veel van de code een herhaling is van patronen bestaat veel programmeren en bouwwerk op dit moment nog steeds uit 'copy, paste & change', ook met betrekking tot schermen en formulieren en veel complexe interactiemodellen. Gelukkig is een groot deel van het proces te automatiseren. Overal zijn herhalende patronen te vinden. Ook in de meer complexe interacties tussen mens, machine en gegevens. En zo komen we bij het principe van de codegenerator. Het geheim van deze generator bestaat uit vijf onderdelen:

- **Patronen** Schermen, schermopbouw, basislogica en interacties tussen elementen volgen allemaal patronen die te vertalen zijn in templates en voorwaarden en regels voor de uitvoer van die templates.
- **Structuren** Een van de veelgebruikte bronnen voor het genereren van code is het datamodel van de database. Hieruit is het merendeel van de applicatiestructuur en het interactiemodel direct te bepalen.
- **Definities** Dit zijn de bepalingen van de specifieke gedragingen en presentatievormen van schermen, velden en stukken code; veelal afkomstig uit het functioneel ontwerp.

- **Logica** De logica bepaalt wanneer welke code-elementen uit de templates worden gekozen en waar en hoe ze zullen worden ingezet. Bijvoorbeeld 'als het invoerveld numeriek is, moet alleen de controle voor numerieke velden worden toegepast op de variable'.
- **Templates** Deze bevatten het 'DNA' van

lossingen waarin de logica realtime wordt uitgevoerd in plaats van via gegeneerde code, is een andere. In de basis leveren ze echter allemaal hetzelfde: het grondwerk van een applicatie.

Steeds meer codegeneratoren leveren volledige applicaties met schermen en menustructuren, waarbij de mate van vrijheid in opbouw en presentatie per applicatie kan verschillen en basistructuren binnen een dag kunnen worden afgeleverd. Vooral de interactiemodellen die standaard worden aangeboden, zijn momenteel nog vrij basaal, maar dat zal over drie, vijf en tien jaar geheel anders zijn. Software ontstaat per definitie vanuit fouten die langzaam en structureel worden opgelost. Complexe software zonder fouten is daarom een utopie. Als fouten niet voorkomen kunnen worden, is het in ieder geval van belang dat ze zo snel mogelijk worden opgelost. Het liefst binnen een dag. Dit is wat ik 'snelheid van schakelen' noem. De belangrijkste wapens die een codegenerator hiervoor biedt zijn:

- reductie van de daadwerkelijke broncode;
- onmiddellijke en applicatiebrede toepassing van simpele wijzigingen.

PETER KAPTEIN

AG-10-12-'04

Peter Kaptein (peter@InstantInterfaces.nl) is eigenaar van Instant Interfaces en ontwikkelaar van oplossingen voor het versnellen van bouwtrajecten van software.