Software Engineering

Rick van Rein

In dit artikel ga ik in op de doelen van software engineering. Ik begin met een korte inleiding in object georiënteerde taaleigenschappen, en dan stap ik over op de concepten die er in de wereld van software engineers leven. Daarbij geef ik speciale aandacht aan de activiteiten die onze faculteit op dit gebied ontplooit.

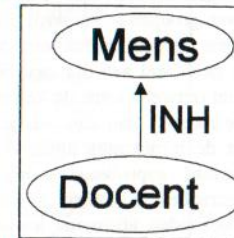
1. De doelen van software engineering

Software engineering is een stroming van de informatica die probeert de complexiteit van het bouwen van software te overwinnen. Meestal gaat dat hand in hand met object oriëntatie, omdat dat taalconstructies aanbiedt die de wensen van software engineering kunnen uitdrukken.

Omdat programmatuur steeds complexer wordt, dreigen wij, als simpele mensen, het overzicht te verliezen. Er spelen domweg te veel aspecten tegelijk mee. Daarnaast is het systeem, als het eenmaal werkt, meestal niet volgens wens, en moet het vroeg of laat worden aangepast. En dan is er nog het probleem dat systemen soms erg op elkaar lijken, en dat het dan wel erg prettig is als je zo veel mogelijk code en ontwerp uit een eerder gebouwd systeem kunt hergebruiken; maar hoe doe je dat goed? Een handvol problemen, die de software engineering probeert te overwinnen. Dat is moeilijk, maar er zit schot in.

De software engineering probeert systemen zo te bouwen dat hun complexiteit beheersbaar blijft. Hiervoor wordt gebruik gemaakt van hokjesdenken: zodra je een onderscheid tussen twee dingen kunt vinden, maak je aparte hokjes voor ze. Maar als ze iets gemeenschappelijks hebben, stop je ook een deel van het gedrag in een hokje dat ze delen. Object georiënteerde programmeertalen bieden voor zulke wensen *classes* aan, die in een *class hierarchy* kunnen worden gecombineerd. Een class kan het gedrag van een andere class overerven via een *inheritance* relatie, en daar een uitbreiding op maken. Zo kan er bijvoorbeeld een class Docent zijn, die erft van class Mens, zoals in figuur 1a. Alles wat een Mens kan en doet, kan een Docent ook doen, maar een Docent doet meer dan dat alleen, bijvoorbeeld les geven. Figuur 1b geeft daarom de operaties bij de twee classes weer.

De hokjesgeest van de software engineer maakt het ook iets gemakkelijker om fouten op te sporen in een programma. Maar ook de plaatsen waar uitbreidingen moeten komen zijn gemakkelijker aan te wijzen. Meestal is er namelijk een hokje dat voorbestemd lijkt voor zo'n uitbreiding, en anders is het vaak gemakkelijk om een hokje toe te voegen. Deze *extensibility* is een belangrijke voorwaarde als we ontwerpen flexibel willen houden. Het is dan ook een belangrijk doel van de software engineering.



Figuur 1a

Class	Operaties
Mens	Lopen Eten Praten
Docent	Lesgeven

Figuur 1b

Een nog belangrijker doel is *reuse*. Dit houdt in, dat delen van oude code en oude ontwerpen nieuw leven wordt ingeblazen in een nieuwe situatie. In imperatieve talen zoals Pascal en C kennen we het gebruik van libraries met standaard routines, en dit lijkt er een beetje op. Het probleem daarbij is alleen, dat de code die daar in zit alleen in zijn geheel herbruikt kan worden. Verder kan het zijn dat library routines die per si moeten samenwerken niet helemaal volgens hun specificatie worden gebruikt in een programma, met een bug tot gevolg. En een ander probleem is het geval waarin een oud stuk code nèt niet voldoet aan de verwachtingen of wensen in een specifieke situatie. Dan moet je alsnog de code herschrijven. Software engineering probeert ook dit probleem aan te pakken.

Op het gebied van reuse en extensibility moet nog veel gebeuren, nog heel veel. Maar een klein lichtpuntje lijkt er wel te zijn, en dat is het gebruik van zogenaamde *frameworks*. Dat zijn, volgens een officiële definitie, *abstracte, object georiënteerde applicaties*. Beetje vaag. Hiermee wordt bedoeld, dat het nog niet helemaal af is: de grote lijn van een applicatie zit er wel in, maar de details moeten nog verder worden uitgewerkt. Dat kan bijvoorbeeld door het hierboven beschreven inheritance-mechanisme.

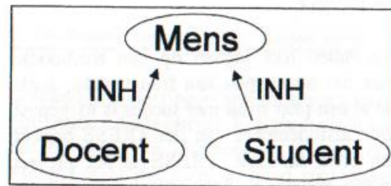
We zullen hier ingaan op een methoodje voor het ontwerpen van frameworks, zoals die al een paar maal met succes is toegepast door afstudeerders van het TRESE-project van onze faculteit. TRESE is het project waarin onderzoek naar object oriëntatie en software engineering wordt gedaan. Misschien ken je de vakken "object georiënteerde systemen" en "software analyse en ontwerp", die worden gegeven vanuit TRESE.

2. Abstracties en reflectie

Zoals gezegd: software engineers hebben een extreme vorm van een hokjesgeest, ze zoeken altijd allerlei manieren om een probleem op te delen. Daarbij zijn twee algemene opdelingsprincipes van groot belang: abstractie en reflectie.

Een voorbeeld van abstractie is het eerder gegeven voorbeeld van de Docent die een

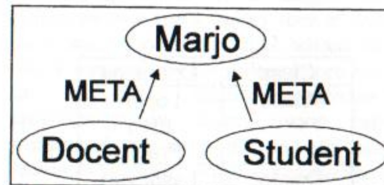
Mens was. Een Mens is een abstractie van een Docent. Maar wat betekent dat nu eigenlijk precies? Een Mens is een algemener begrip, het omvat meer gevallen dan alleen maar een Docent. Bijvoorbeeld ook een Student, zie figuur 2. De Mens heeft een aantal eigenschappen, maar alleen die eigenschappen die niet te maken hebben met de rol die hij in de collegezaal heeft. We zeggen dan ook, dat Mens abstraheert van het collegezaal-gedrag. *Abstractie* is dan ook het weggooien van irrelevante gegevens. Door aan te geven van welk aspect wordt geabstraheerd, geef je aan wat je weggooit.



Figuur 2

Reflectie is een moeilijker begrip, en subtiel iets anders dan abstractie. Eigenlijk is *reflectie* niets meer dan overzicht. Neem bijvoorbeeld een class Marjo, die de roosters maakt. Daarbij moet rekening worden gehouden met de tijden waarop een Student en een Docent beschikbaar zijn. Zo'n class Marjo pleegt reflectie op het rooster-aspect van de Student en Docent, zie figuur 3. Reflectie heet ook wel het optillen naar *meta-niveau.* Dit houdt in dat in die meta-class een samenvatting of ander overzicht wordt gemaakt van de rooster-aspecten van de andere classes, en dat die classes daarvan afhankelijk worden beïnvloed.

Bij het bouwen van frameworks is de kunst om de juiste abstracties te vinden. Er zijn namelijk eng veel mogelijkheden. Probeer bijvoorbeeld maar eens de lijst woorden in figuur 4 in twee groepen te verdelen. Er zijn ontzettend veel mogelijkheden om dit te doen, en dat geeft wel aan dat ook bij het maken van een ontwerp soms de keuze van een abstractie moeilijk kan zijn. Als je naar onderzoek in de informatica kijkt, valt op dat er inderdaad geprobeerd wordt om geschikte abstracties te vinden. Meer dan eens is een gevonden abstractie aanleiding tot een publikatie.



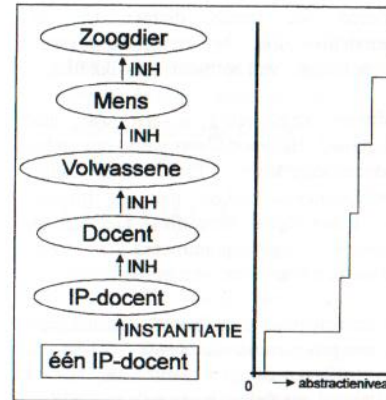
Figuur 3

LIED	STOEL	SPATIE
OMA	BONNE	OBJECT
ROOS	ROODHARIG	TITEL

Figuur 4

Veelal blijft er toch een probleem met abstracties en frameworks die volgens gangbare methoden worden ontwikkeld. Als je een plaatje maakt van het abstractieniveau van een class, dan lijkt het op figuur 5; het framework verzorgt een geleidelijk afdaling in abstractieniveau, zodat er altijd een duidelijke plaats is om een abstractie een klein beetje aan te passen; zodra je echter een werkelijke applicatie gaat maken is er vaak een verdere uitwerking van de framework classes nodig, en de sprong in abstractieniveau die dan ontstaat is vaak erg

groot, omdat toch nog relatief veel details moeten worden ingevuld.



Figuur 5

3. Compositie

Binnen TRESE heeft men andere ideeën over het ontwerpen van software, die bovengenoemde bezwaren misschien kunnen overwinnen. Kort gezegd komt de droom van het TRESE project neer op het *componeren* van software, waarmee wordt bedoeld dat componenten worden samengesteld. Iets preciezer komt het neer op de wens om standaard problemen in de informatica te identificeren, en voor alle oplossingen een stuk software in de vorm van een framework te gieten. Elke oplossing is dan een apart component.

Gegeven een complex probleem willen we dan de aspecten eraan uitzoeken, en de standaard oplossingen voor de aldus gevonden deelproblemen willen we dan samenstellen tot de oplossing voor het gehele probleem. Het hoeft geen betoog dat dit "samenstellen" zo veel mogelijk automatisch moet gaan. Een methode ervoor is bijvoorbeeld inheritance uit meerdere abstractere classes, of sequencing van onze *composition filters*.

De componenten moeten liefst zo klein mogelijke stukjes code vertegenwoordigen, zodat ze zo algemeen mogelijk inzetbaar zijn. Een probleem waar je dan mee te maken krijgt is alleen dat dit betekent dat je voor een gegeven probleem meestal extra veel componenten moet samenstellen, en dat maakt het werk toch weer complexer.

Merk wel op, dat de abstractiestap niet meer zo groot is als in figuur 5. Wanneer we in een toepassing de classes maken door combinatie van meerdere abstracte classes, dan kunnen die elk afzonderlijk van een ander aspect abstraheren. Hierdoor is hetgeen uiteindelijk nog moet worden toegevoegd tot een minimum te beperken, en misschien wel tot niets te reduceren.

De truc die we in TRESE af en toe toepassen, is na te gaan welke *views*, of invalshoeken, we op een ontwerp-probleem kunnen vinden. Binnen elke view maken we dan een aparte inheritance hiërarchie. Deze *multiple view* aanpak voor software design lijkt erg vruchtbaar, en lijkt ook hergebruik van code beter mogelijk te maken. Verder wordt er momenteel gewerkt aan een softwarematige ondersteuning voor multiple views, en als alles naar verwachting uitpakt dan zal die ondersteuning er zorg voor dragen dat zelfs (delen van) ontwerpen herbruikbaar worden.

Dit is overigens een fundamentele uitbreiding ten opzichte van het meeste gangbare onderzoek; daar probeert men alle ontwerp-informatie in één inheritance hiërarchie te stoppen. Dat resulteert direct in het probleem dat in figuur 5 werd uitgelegd.

Noodzakelijk voor hergebruik van het ontwerp is natuurlijk wel een goede administratie van de ontwerp-beslissingen. Dit is wat we bij TRESE *hermeneutics* noemen; het idee daarvan is dat de software waarmee de software engineer werkt kan onderbouwen waarom iets op een bepaalde manier in elkaar zit, zodat je bij wijzigingen aan een ontwerp beter weet wat je doet. Vooral in een omgeving met meerdere ontwerpers die aan één ontwerp samenwerken lijkt dit erg nuttig. De views lijken een geschikt aankoppelpunt om zulke informatie in op te slaan.

4. Conclusie

Software engineering stelt zichzelf zeer hoge doelen, die voorlopig nog lang niet zijn verwezenlijkt. Ook het onderzoek van de TRESE groep van onze faculteit is nog verre van af. Echter, in kleine stapjes naderen we steeds dichter tot ons uiteindelijke doel: het beheersbaar maken van het traject van software-ontwikkeling.

Software engineering is een wat apart vakgebied. Het modelleert werk van andere onderzoeksgroepen met haar eigen modelleringstechnieken, met als resultaat dat oplossingen (hopelijk) niet telkens opnieuw uitgeprogrammeerd of zelfs ontworpen hoeven te worden.

Object oriëntatie is een stapel eigenschappen die een programmeertaal heeft; het blijkt dat de taaleigenschappen uit die wereld uitstekend geschikt zijn voor de modellering van de ideeën uit de software engineering, al doet het vele onderzoek naar dit vakgebied vermoeden dat het meest optimale taalmodel nog niet is gevonden. Dat zal ook nog wel even op zich laten wachten, want het gaat hand in hand met de software engineering. En zoals gezegd, is daar nog wel wat werk te doen...

Agenda

1 februari	**Borrel**
8 februari	**Snookeravond**
15 maart	**Borrel**
30/31 maart	**Ouderdagen INF/BIT**
5 april	**Borrel**

Virtual Reality in de Vrijhof

Op 22 en 29 november was er in de vrijhof een symposium over Virtual Reality. Hieronder volgt de tekst van de wervingsfolder die toen is uitgedeeld.

VIRTUAL REALITY

De werkelijkheid is niet meer wat zij geweest is. Met steeds vernuftiger apparatuur zetten wetenschappers de wereld naar hun hand. Voor steeds meer bedrijven ligt in, letterlijk vertaald, 'schijnbare realiteit' een wenkend toekomstperspectief.

Dit programma voert u door middel van woord en beeld naar cyberspace en andere onbekende oorden.

Kom dat horen en laat u verrassen door de nieuwste ontwikkelingen op het terrein van hard- en software.

Kom dat zien en geloof uw ogen niet.

Deze cyclus is tot stand gekomen in een samenwerkingsverband van Inter-*Actief*, de studievereniging van de faculteit Informatica, en het *Studium Generale* van de Universiteit Twente.

Samenstelling en regie zijn in handen van Ilja Heitlager, Henk Procee, Herman Schipper, Lodewijk Smit, Emiel Voskuilen en Tjarda de Vries.

Marcel Wierda:

De realiteit rondom de Virtuele Omgeving

Cyberspaces, Wegwerpwerelden, Virtuele Realiteiten, Imaginaire Werelden: sinds een tiental jaren trekken 'ze' de aandacht. Hoe is de stand van zaken nu? Middels een korte analyse van wat een virtuele omgeving is, zowel als systeem voor psychologische zinsbegoocheling als hoe ze er technisch uitzien, zal worden aangegeven wat de sterke en zwakke kanten van de Virtuele Omgeving zijn. Wellicht kan met een dergelijk overzicht een volgende generatie van prognoses gegeven worden. Met name zal worden ingegaan op de samensmelting van Artificial Intelligence en Virtuele Omgevingen, een symbiose die werkelijk meer zou kunnen zijn dan de som der delen.

Drs. M. Wierda is werkzaam aan het Verkeerskundig Studiecentrum van de Rijksuniversiteit Groningen, alwaar hij waarschijnlijk de eerste virtuele verkeersomgeving ter wereld heeft gebouwd. Hij publiceert geregeld over diverse aspecten van Virtual Reality.

Hans Hubers:

Multimedia presentaties in de architectuur

In de wereld van de architectuur worden computerpresentaties meer en meer een interessant alternatief voor de gebruikelijke