Goede grip op software-afhankelijkheden

Met Design Structure Matrix

Bij softwareontwikkeling is een hoog innovatietempo een vereiste. Door de aanhoudende tijdsdruk is er veelal weinig aandacht voor de softwarearchitectuur, waardoor deze gaandeweg complexer word. Deze verhoogde complexiteit kan leiden tot 8x meer defects, een 50% lagere productiviteit en een hoger personeelsverloop. Gebruik van een Design Structure Matrix maakt complexe afhankelijkheidsstructuren inzichtelijk en kan daardoor weer grip geven op softwareafhankelijkheden.

Inleiding

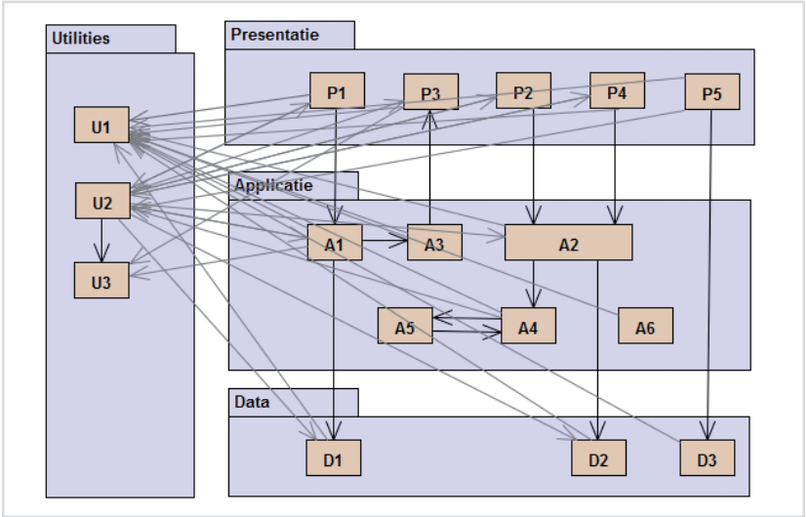
Idealiter is software eenvoudig aan te passen, begrijpelijk, betrouwbaar en herbruikbaar. In de praktijk blijft dit vaak een ideaal en wordt de software in de loop van de tijd steeds meer rigide, ondoorzichtig en fragiel. Hierdoor worden aanpassingen steeds lastiger om door te voeren. Ook zijn componenten moeilijk te isoleren en daardoor niet herbruikbaar en individueel testbaar. Al deze problemen worden veroorzaakt door softwareafhankelijkheden.

Veel softwareprojecten worstelen met softwareafhankelijkheden. Een van de redenen is dat de softwarearchitectuur vaak is vastgelegd op een conceptueel niveau dat onvoldoende precisie heeft ten aanzien van welke afhankelijkheden zijn toegestaan en welke niet. Daarnaast kunnen aanpassingen op broncodeniveau onbedoeld afhankelijkheden toevoegen, waardoor de gedocumenteerde en daadwerkelijke architectuur van elkaar gaan afwijken. Ook is de veelgebruikte UML-notatie niet geschikt voor representatie van grote hoeveelheden afhankelijkheden.

Op een gegeven moment kan de complexiteit een niveau bereiken dat compleet herschrijven goedkoper is. Vaak ontbreekt hiervoor echter de tijd en zijn de risico's onaanvaardbaar hoog. Om die reden is het definiëren van een software architectuur in termen van bestaande broncode elementen via een Design Structure Matrix (DSM) een zeer aantrekkelijk alternatief.

Als voorbeeld toont **Figuur 1** een UML-classdiagram met daarin een aantal typische situaties. Zo is er een cyclische relatie tussen A4 en A5, een relatie van A3 naar P3 tegen de gewenste layering in, een relatie van P5 naar D3 die een laag overslaat en een ongebruikte class A6 zonder ingaande relaties. Verder hebben U1 en U3 veel ingaande relaties en heeft U2 veel in- en uitgaande relaties. Met name deze relaties maken het geheel erg onoverzichtelijk. Om die reden worden in een UML-diagram vaak niet alle relaties weergegeven, maar alleen de meest essentiële, waardoor het slechts een incomplete view van het model is.

Johan van den Muijsenberg is senior softwareconsultant bij Alten PTS.



Figuur 1: Elementen met veel in- en uitgaande relaties maken een UML-diagram onoverzichtelijk.

Belangrijke voordelen

Met behulp van een Design Structure Matrix (DSM) kan een structuur worden gemaakt van elk systeem dat een hiërarchie van elementen heeft en worden ook de relaties tussen deze elementen weergegeven. Algoritmes om de structuur te analyseren en te herordenen, vormen een essentieel onderdeel van deze methodiek. Hoewel het oorspronkelijk is bedacht voor procesoptimalisatie zijn DSM's ook toepasbaar voor de visualisatie en analyse van productarchitecturen. Dat maakt het zeer geschikt voor software.

Figuur 2 geeft het ontwerp uit **Figuur 1** weer in een DSM. Deze bestaat uit een matrix met in de rijen en kolommen dezelfde elementen. De hiërarchie van packages en classes is zichtbaar aan de linkerzijde. In de cellen staan de relaties tussen de elementen. Dat kolom 5 (P4) van rij 16 (A2) bijvoorbeeld gevuld is, betekent dat P4 afhankelijk is van A2. Dit is in overeenstemming met het UML-diagram van Figuur 1. Het nummer in de cel representeert de sterkte van de relatie en is meestal het aantal afhankelijkheden tussen de elementen. De cellen op de diagonaal representeren de relatie van een element met zichzelf en zijn meestal leeg.

Het kost weliswaar enige gewenning om de matrices te lezen, maar daarna zijn er een aantal belangrijke voordelen ten opzichte van UML. Ten eerste kan een DSM alle relaties in een enkele view bevatten. Waar weergave van alle relaties in een UML-diagram onoverzichtelijk kan zijn, zoals in Figuur 1, vormt dit geen enkel probleem in een DSM, zoals te zien is in de eerste drie rijen en kolommen van **Figuur 2**.

Ten tweede is een DSM schaalbaar. Doordat de hiërarchie geheel of gedeeltelijk inklapbaar is, is het mogelijk om een systeem met duizenden classes weer te geven. Als we de DSM in Figuur 2 volledig inklappen, krijgen we de matrix in **Figuur 3**. De relatiesterktes van de ingeklapte cellen worden daarbij eenvoudig samengevoegd. De DSM wordt hierdoor compacter, maar blijft inhoudelijk nog steeds correct.

Ten derde kunnen we de layering van de software eenvoudig inzichtelijk maken door een partitioneringsalgoritme toe te passen. Een dergelijk algoritme probeert de DSM zo te herordenen dat zoveel mogelijk relaties onder de diagonaal komen. Toepassing op **Figuur 3** levert de matrix op die is weergegeven in **Figuur 4**. Hierin is te zien dat elementen met veel ingaande relaties (providers) naar de onderzijde zijn verschoven, terwijl de elementen met veel uitgaande relaties (consumers) omhoog zijn gegaan. Relaties die het algoritme boven de diagonaal zet, zijn een indicatie voor cyclische afhankelijkheden in de software. Deze zijn ongewenst, omdat de betrokken elementen niet onafhankelijk zijn en aantoonbaar meer fouten bevatten.

Ten vierde maakt een DSM het gemakkelijk om afhankelijkheidspatronen te herkennen. Figuur 5 laat zien dat er problemen met de layering zijn, omdat P5 in de presentatielaag rechtstreeks D3 uit de datalaag gebruikt (1) en er een afhankelijkheid in ongewenste richting is, omdat A3 in de applicatielaag P3 in de presentatielaag gebruikt (2). Verder is te zien dat A1 en A2 de publieke interfaces zijn en dat A3, A4 en A5 interne elementen zijn van de appli-

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Utility	U3	1	.	2				2					13				11	6
	U2	2		.		13	13		3	4					2			9
	U1	3			.	10	10	2		2	2	9	4	13		3	2	5	2
Presentatie	P5	4				.												
	P4	5		1			.											
	P3	6		1				.								1		
	P2	7		1					.									
	P1	8		1						.								
Data	D3	9				4					.							
	D2	10		1								.					7	
	D1	11		1									.					4
Applicatie	A6	12												.				
	A5	13													.	1		
	A4	14												5	.		2	
	A3	15														.		1
	A2	16		1		1		1									.	
	A1	17		1					1									.

Figuur 2: Waar weergave van alle relaties in een UML-diagram onoverzichtelijk kan zijn, vormt dit geen enkel probleem in een DSM.

		1	2	3	4
Utility	1	.	59	15	66
Presentatie	2	4	.		1
Data	3	2	4	.	11
Applicatie	4	2	3		.

Figuur 3: In een DSM is de hiërarchie ook geheel of gedeeltelijk inklapbaar. De matrix is dan compacter, maar inhoudelijk nog steeds correct.

OP EEN GEGEVEN MOMENT KAN DE COMPLEXITEIT EEN NIVEAU BEREIKEN DAT COMPLEET HERSCHRIJVEN GOEDKOPER IS

		1	2	3	4
Presentatie	1	.	1		4
Applicatie	2	3	.		2
Data	3	4	11	.	2
Utility	4	59	66	15	.

Figuur 4: Door een partitioneringsalgoritme toe te passen, is de layering van de software eenvoudig inzichtelijk te maken in een DSM.

catielaag, omdat ze buiten (3) respectievelijk alleen binnen deze laag worden gebruikt (4), en dat A6 een ongebruikt element is, omdat het geen enkele ingaande relatie heeft (5). U1, U2 en U3 zijn elementen die door de hele software heen worden gebruikt (6). U2 is echter problematisch, omdat het behalve ingaande relaties ook uitgaande relaties heeft (7).

Tot slot is een DSM te gebruiken om de afhankelijkheidsstructuur te verbeteren. In de matrix kunnen we een element verplaatsen naar een andere component of laag, combineren met andere elementen of splitsen en vervolgens alle afhankelijkheden herberekenen om te kijken of dit een betere afhankelijkheidsstructuur oplevert. Stel bijvoorbeeld dat P3 in Figuur 5 eigenlijk thuishoort in de applicatielaag. Dan kan dat element daar naar toe worden verplaatst. Na herberekening van de afhankelijkheden is te zien dat de cyclische relatie tussen de presentatie- en applicatielaag verdwenen is. De voordelen van zo'n impactanalyse komen vooral tot hun recht bij verbeteringsscenario's die zich afspelen op architectuurniveau en dus meerdere componenten raken. Zonder het gebruik van een DSM zijn dergelijke analyses onbetrouwbaar, omdat de ontworpen software en de daadwerkelijke softwarearchitectuur vaak niet overeenkomen.

Toepassingsgebied

DSM's zijn zeer effectief in situaties waar een goed inzicht in de daadwerkelijke softwareafhankelijkheden van essentieel belang is. Te denken valt hierbij aan grote functionele uitbreidingen, outsourcing onderzoek, technologie migratie, verbetering van de testbaarheid en verhoging van het hergebruik tussen productvarianten. Daar waar een goed gedocumenteerde software architectuur ontbreekt, kan deze in kaart gebracht worden met een DSM analyse van de broncode.

Architectuur elementen en toegestane relaties tussen deze elementen kunnen gedocumenteerd worden door middel van een DSM. Een dergelijk definitie maakt de toegestane afhankelijkheden expliciet en vervult een brugfunctie tussen de conceptuele architectuur en de code. Door automatische validatie van de afhankelijkheidsregels in het softwarebouwproces en eventueel IDE kan degradatie van de architectuur worden voorkomen.

Tooling

Hoewel de aanpak veelbelovend is, zijn er op dit moment niet veel tools die DSM ondersteuning bieden.
Beschikbaarheid van adequate open source tools zou de drempel voor de inzet van DSM's verlagen. Helaas bieden de op dit moment beschikbare open source tools onvoldoende ondersteuning, waardoor men gedwongen is om gebruik te maken van commerciële tools.

Als de inzet beperkt blijft tot architectuur analyse en bewaking dan biedt JArchitect voldoende features.
Indien echter architectuur refactoring noodzakelijk is, dan is Lattix de beste keuze. Structure101 ondersteunt ook architectuur refactoring, maar echter niet via een DSM. Daarnaast mist Structure101 de mogelijkheid om afhankelijkheden via een hiërarchische matrix te visualiseren, hetgeen essentieel is. De Intelij Ultimate IDE biedt standaard een beperkte DSM functionaliteit.

HET KOST WELISWAAR ENIGE GEWENNING OM DE MATRICES TE LEZEN, MAAR DAARNA ZIJN ER EEN AANTAL BELANGRIJKE VOORDELEN TEN OPZICHTE VAN UML

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Presentatie	P5	1	.															7	
	P4	2		.														1	
	P3	3			.						1	2						1	
	P2	4				.												1	
	P1	5					.											1	
Applicatie	A6	6	5					.											
	A5	7							.	1									
	A4	8							5	.		2							
	A3	9						4			.		1						
	A2	10		1		1						.						1	
	A1	11	3				1						.					1	
Data	D3	12	4	1										.					
	D2	13										7			.			1	
	D1	14											4			.		1	
Utility	U3	15	6		2			13				11	6				.	2	
	U2	16	13	13		3	4			2			9					.	
	U1	17	10	10	2		2	13		3	2	5	2	2	9	4			.

Figuur 5: Een DSM maakt het bovendien gemakkelijk om afhankelijkheidspatronen te herkennen.

ArgoUML Case

Om de aanpak te illustreren is een DSM analyse van de open source applicatie ArgoUML uitgevoerd. De uitwerking van deze analyse is te vinden op de website van de Dutch Java User Group (**http://www.nljug.org/**).

Conclusie

De afgelopen periode heb ik een aantal codebases op de boven beschreven wijze doorgelicht. De aanpak heeft zich bewezen als een zeer effectieve manier om inzicht te krijgen in de afhankelijkheidsstructuur van software. Vervolgens kan men middels de DSM een aantal mogelijke refactoring scenario's simuleren ter verbetering van de software structuur. Op deze manier kan men een goede impact analyse doen alvorens tot code aanpassingen over te gaan. Ten slotte kan men met de DSM afhankelijkheidsregels definiëren, waarmee de software architectuur bewaakt kan worden.

Het is op deze manier mogelijk afhankelijkheden sterk te reduceren en vervolgens onder controle te houden. Daarmee wordt de architectuur beter communiceerbaar en de productiviteit en kwaliteit verhoogd. ■

	Lattix	Structure101	Intelij Ultimate	J A r c h i t e c t	SonarCube	Dtangler/ DSMViewer
Hoofddoel applicatie	Software architectuur analyse en refactoring	Software architectuur analyse en refactoring	Java IDE	Software architectuur en code analyse	Web based continuous inspection	Software architectuur analyse
Visualisatie afhankelijkheden						
Hiërarchisch matrix	Ja	Nee	Ja	Ja	Nee	Nee
Cyclische afhankelijkheden	Ja	Ja	Ja	Ja	Ja	Ja
Indirect afhankelijkheden	Nee	Nee	Nee	Ja	Nee	Nee
Afhankelijkheden tot op methode/attribuut nivo	Ja	Ja	Nee	Ja	Nee	Nee
Architectuur analyse en assessment						
Aantal partitionerings algoritmes	11	1	1	1	1	1
Aantal software metrieken	24	2	0	82	64	0
mpact analyse	Ja	Nee	Nee	Ja	Nee	Nee
Architectuur verbetering						
Simuleer refactoring	via DSM	via LSM	Nee	Nee	Nee	Nee
Architectuur bewaking						
Definitie afhankelijkheid regels	via DSM	via architectuur diagram	Nee	via code query taal	Nee	Nee
IDE integratie	Eclipse	Eclipse/ Intelij	n.v.t.	Nee	n.v.t.	Nee
Build integratie	Ja	Ja	Nee	Ja	Ja	Nee
Metrics trends	Ja	Ja	Nee	Ja	Ja	Nee
Overig						
Aanpasbaarheid	via Groovy	Nee	Nee	via code query taal	Nee	Nee
Open source licentie	Nee	Nee	Nee	Nee	Ja	Ja