Complexiteit en beheersbaarheid van software

5.7 *COMPLEXITEIT EN BEHEERSBAARHEID VAN SOFTWARE*

prof.dr. Th.M.A. Bemelmans en dr.ir. F.J. Heemstra

5.7.1 SAMENVATTING

Grote software-projecten zijn vaak complex met alle gevolgen vandien voor de beheersing van dergelijke projecten. In dit artikel wordt een omschrijving gegeven

van complexiteit, worden factoren die complexiteit veroorzaken, geïdentificeerd en worden besturingsconcepten om complexiteit te beheersen, toegelicht.

5.7.2 COMPLEXITEIT

Algemeen bekend moge zijn dat automatisering een hoge vlucht heeft genomen in de afgelopen decennia. Automatisering wordt nu toegepast in drie bijna alles omvattende hoofddomeinen, te weten [108]:

- bestuurlijke (administratieve) toepassingen;
- primaire procesautomatisering, waaronder Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) en Computer Integrated Manufacturing (CIM);
- automatisering in eindprodukten ('embedded software') en einddiensten. Voorbeelden: consumenten-elektronica, allerlei dienstverlening met telematica waaronder Electronic Data Interchange (EDI), enz.

Software-ontwikkeling is een belangrijke industriële activiteit geworden. Een van de kernproblemen bij software-ontwikkeling is de beheersbaarheid. Kosten en doorlooptijden worden regelmatig fors overschreden. In de beginperiode van automatisering was dat geen ramp, immers automatisering speelde zich af in de 'marge van de bedrijfsvoering'. Dat beeld is drastisch veranderd. Software-ontwikkeling behoort tot een van de kernactiviteiten van de hedendaagse industrie of dienstverlening, en zodoende is men strategisch afhankelijk van het goed functioneren van de automatisering. Forse overschrijdingen van ontwikkel- of onderhoudskosten, dan wel van doorlooptijden zijn daarmee ongeoorloofd geworden [43, 109, 110].

In dat perspectief staat complexiteit van software en software-projecten opnieuw in de belangstelling. Immers: hoe complexer een produkt of een project, hoe moeilijker de beheersing, en dus hoe onzekerder bijvoorbeeld de kosten en de doorlooptijd.

Complexiteit is dus belangrijk. De volgende vraag is wat complexiteit precies is en hoe men dat kan meten. Lange tijd heeft men volstaan met een puur subjectieve benadering. Of iets wel of niet complex was, liet men over aan de perceptie van de persoon en aan iemands persoonskenmerken. Duidelijk moge zijn dat een dergelijke benadering weinig 'operationele zoden aan de dijk' zet voor een adequate beheersing van software-ontwikkeling en -onderhoud.

Er zijn diverse pogingen ondernomen om meer objectieve maatstaven te ontwikkelen voor de complexiteit van software. In grote lijnen kan men deze maatstaven verdelen in maten voor complexiteit die zijn gerelateerd aan de *omvang* van een software-programma, en in maten voor complexiteit die zijn gerelateerd aan de *structuur* van zo'n programma.
De meest simpele maat voor de omvang is het aantal regels programmacode: hoe meer regels, hoe groter de complexiteit. Een ernstig nadeel van deze simpele metriek is dat men het verschil in uitdrukkingskracht van de verschillende programmeertalen volledig verwaarloost. Een programma in de taal COBOL is nu eenmaal langer dan hetzelfde programma in de taal PASCAL of APL.

Een ander bezwaar is dat men geen onderscheid maakt naar soorten regels programmacode. Echter: de ene regel is niet de andere. Er zijn eenvoudige regels programmacode en er zijn complex samengestelde regels programmacode. Om aan deze bezwaren tegemoet te komen, heeft Halstead een meer verfijnde metriek ontwikkeld [111]. In zijn theorie is de complexiteit en de daarmee samenhangende programmeerinspanning afhankelijk van de omvang van een programma, en van het niveau van structurering. Hoe groter de omvang, hoe hoger de programmeerinspanning; hoe hoger (hoe compacter) het niveau van structurering, hoe lager de inspanning. Zowel omvang als niveau van structurering zijn functies van het aantal verschillende operatoren (lees bewerkingen) in een programma en het aantal verschillende operanden (dus variabelen en constanten) in een programma.
De precieze functies staan beschreven in [112] en [111].

Een poging om een maat voor complexiteit te ontwikkelen, gebaseerd op de structuur van een programma, is de 'cyclomatische complexiteit' van McCabe [112, 111]. De 'control flow' van een programma wordt daarbij afgebeeld als een graaf (verzameling knooppunten verbonden door lijnen). Het aantal lineair onafhankelijke paden in deze graaf bepaalt dan de cyclomatische complexiteit. In wezen zegt deze metriek iets over het aantal verschillende paden dat een programma kan doorlopen tijdens de uitvoering. Hoe meer alternatieve paden, hoe hoger de cyclomatische complexiteit.

Hoewel voorgaande maten voor complexiteit verdienstelijke pogingen zijn om de complexiteit van software objectief te bepalen, blijken deze maten te weinig genuanceerd en te weinig onderbouwd om daarmee reële schattingen te maken van bijvoorbeeld ontwikkelinspanning, onderhoudslast, enz. Voorlopig moet men daarom genoegen nemen met benaderingen van complexiteit uit verschillende invalshoeken. We zullen dat in het navolgende toelichten.

5.7.3 INVLOED VAN COMPLEXITEIT OP PRODUKTIVITEIT

Een van de eerste empirische studies naar de samenhang tussen complexiteit en produktiviteit, is een onderzoek van Walston en Felix rond 1975 [113]. In dat onderzoek zijn zestig zeer uiteenlopende software-projecten geanalyseerd en is nagegaan welke factoren invloed hebben op de produktiviteit van het ontwikkelteam. Produktiviteit is daarbij vertaald naar het aantal uiteindelijk opgeleverde regels programmacode per mensmaand. Over alle typen projecten was die produktiviteit gemiddeld om en nabij de 300 regels programmacode per mensmaand, ofwel nauwelijks 3000 regels programmacode per geïnvesteerd mensjaar. Wanneer men dat relateert aan hedendaagse software-toepassingen van een miljoen regels programmacode of meer – dat is geen uitzondering – zou het om projecten gaan met een inzet van 300 mensjaren of meer! Nu is er sinds de studie van Walston en Felix het nodige verbeterd aan software-methoden en -technieken, maar een feit blijft dat het ontwikkelen van software al snel leidt tot zeer omvangrijke projecten met veel menskracht en lange doorlooptijden.
Interessanter dan de absolute produktiviteit, zijn de variaties in deze produktiviteit door allerlei complexiteitsverhogende factoren. Tabel 5.2 geeft daarvan een sum-

mier overzicht. In de tabel zijn de negen meest zwaarwegende factoren genoemd. De invloed van elke factor op de produktiviteit (in regels programmacode per mensmaand) is bekeken bij variatie in de betreffende factor. Zo geldt bij een normale complexiteit van de mens/machine-interface een produktiviteit van 295 regels programmacode. Verlaagt men de eisen aan die mens/machine-interface aanzienlijk, dan stijgt de produktiviteit met bijna 70% tot 500 regels programmacode. Verhoogt men de eisen aanzienlijk, dan keldert de produktiviteit met 60% tot slechts 124 regels programmacode. Op overeenkomstige wijze zijn de invloeden van de overige factoren na te gaan.

	Produktiviteit (in regels code per mensmaand)		
1 Complexiteit van de mens/machine-interface	laag 500	normaal 295	hoog 124
2 Betrokkenheid gebruikers bij definitie van eisen	nauwelijks 491	matig 267	intensief 205
3 Ervaring en kennis van projectmedewerkers	weinig 132	gemiddeld 257	veel 410
4 Specifieke ervaring met betrekking tot concrete toepassing	weinig 146	gemiddeld 221	veel 410
5 Ervaring met gehanteerde programmeertaal	weinig 122	gemiddeld 225	veel 385
6 Percentage programmeurs dat betrokken was bij het opstellen van het logisch ontwerp	<25% 153	25-50% 242	>50% 391
7 Computergeheugen als beperkende factor bij het ontwerp	weinig 391	enigszins 277	hoog 193
8 Aandeel van het totale project dat werd uitgevoerd volgens de organisatie 'Chief Programmer Team' (waarbij het projectteam wordt begeleid door een ervaren senior)	0-33% 219	34-66% 302	>66% 408
9 Complexiteit van de totale toepassing	laag 349	gemiddeld 345	hoog 168

Tabel 5.2 Produktiviteitsbeïnvloedende factoren bij software-ontwikkeling

Tabel 5.2 maakt duidelijk dat de complexiteit van software en van het project duidelijk invloed hebben op de produktiviteit. Variaties in die complexiteit leiden tot dramatisch grote verschuivingen in produktiviteit met alle kosten en tijdconse-

quenties vandien. Ordent men de in de literatuur genoemde factoren, die de complexiteit en daarmee de produktiviteit bepalen, dan zijn deze factoren:
- produktfactoren, die samenhangen met het produkt;
- procesfactoren, die samenhangen met het produktieproces;
- middelenfactoren, die samenhangen met de middelen.

Onder produktfactoren vallen die zaken, die te maken hebben met het uiteindelijke software-systeem zoals dat moet worden opgeleverd. In tabel 5.2 zijn dat de factoren 1, 7 en 9. Andere factoren die men vaak in de literatuur noemt, zijn:
- *omvang*, bijvoorbeeld gemeten in regels programmacode;
- *type applicatie*. Zo signaleert Jones [43] dat de complexiteit van een real-time-toepassing vele malen hoger is dan een even grote, maar batch-georiënteerde toepassing;
- *structuur* waaronder 'modulariteit', aantal interfaces tussen modulen, gestructureerd ontwerp en programmeren, enz.;
- zwaarte van de *kwaliteits- ofwel performance-eisen* zoals gebruiksvriendelijkheid, beveiliging, integriteit, onderhoudbaarheid, aanpasbaarheid, compatibiliteit, enz. Als men wil voldoen aan gerechtvaardigde kwaliteitseisen, is het geen uitzondering dat de inspanning 5 tot 10 keer zo zwaar wordt als de inspanning bij het 'kale software-probleem'.

Onder produktieprocesfactoren vallen die zaken, die te maken hebben met de wijze waarop software wordt ontworpen, geconstrueerd, getest en ingevoerd. In tabel 5.2 verwijzen de factoren 2, 6 en 8 naar het produktieproces. Factor 6 geeft aan dat de produktiviteit stijgt naarmate er minder *(communicatie)schotten* bestaan tussen de diverse specialisten. Factor 8 verwijst naar een specifieke vorm van projectorganisatie waarbij een ervaren senior voortdurend *inhoudelijk toezicht* houdt op het werk van zijn teamgenoten. De meest complexiteitsverhogende en produktiviteitsbeïnvloedende factor is gebruikersparticipatie (factor 2). Daarmee wordt geen pleidooi gehouden om vooral gebruikers 'buiten de deur' te houden, integendeel. Wel wordt stelling genomen tegen literatuur, waarin ongenuanceerd en ongemotiveerd wordt 'geschreeuwd' om gebruikersparticipatie. Men doet het dan voorkomen alsof alle problemen zijn opgelost wanneer men de gebruikers erbij betrekt. In veel gevallen blijkt het tegendeel waar te zijn. Gebruikers voelen zich 'met de haren erbij getrokken' omdat technische problemen en opties niet de hunne zijn. Of het is fysiek onuitvoerbaar alle gebruikers erbij te betrekken. Bij het laatste kan men denken aan infrastructurele voorzieningen, zoals netwerken in een organisatie, of aan consumentenelektronica waarbij men onmogelijk elke afnemer kan kennen. De moraal is dat men er wijs aan doet expliciet te overwegen waar, wanneer en waarom men gebruikers wil laten participeren. 'Jan en alleman' laten meepraten, is voor betrokkenen onbevredigend en leidt niet zelden tot een zodanig complexe organisatie dat het software-project in korte tijd 'muurvast' zit.

Onder middelenfactoren vallen die zaken, die als middelen worden ingezet bij de software-ontwikkeling, zoals menskracht, (4e generatie-)hulpmiddelen bij de ontwikkeling, budgetruimte en toegewezen tijd. Alle soorten middelen die men inzet, beïnvloeden de produktiviteit in aanzienlijke mate. Zo blijkt uit tabel 5.2 dat de kennis en ervaring van de projectmedewerkers de produktiviteit beïnvloeden (zie

factor 3, 4 en 5). Het middel 'toegewezen tijd' verdient enige toelichting. Bij het bepalen van een gewenste doorlooptijd gaat men vaak van de foutieve veronderstelling uit dat er een lineair verband bestaat tussen inzet van middelen en doorlooptijd. Men noemt dat ook wel het 'Chinese Monkey Syndroom'. Simpel gesteld: een karwei van een manjaar kan men laten doen:

a. door één persoon, doorlooptijd één jaar;
b. door twee mensen, doorlooptijd 0,5 jaar;
c. door drie mensen, doorlooptijd 0,33 jaar;
d. enz.

Vooral Putnam heeft gewezen op dit syndroom. Hij stelt dat een project weliswaar verkort kan worden door de inzet van meer mensen, maar dat dit slechts geldt tot een bepaalde limiet. Putnam citeert daarbij de wet van Brooks: 'Adding people to a late project only makes it later. The reason is clear. As the number of people on a project increases arithmetically, the number of human interactions increases geometrically. More and more time must be spent on human communication and less on productive works' [111].

5.7.4 BEHEERSBAARHEID EN COMPLEXITEIT

In veel vakgebieden is zo langzamerhand duidelijk geworden dat er niet zoiets als een allesomvattende, steeds geldende oplossing bestaat. Oplossingen en manieren om tot een oplossing te komen, verschillen per specifieke situatie en zijn dus situatiegebonden (contingentietheorie). Een en ander dringt ook door in de software engineering. Was vroeger *de* projectorganisatie het panacee, langzamerhand krijgt men oog voor een typologie van beheersingsituaties. Als men uitgaat van het hiervoor gemaakte onderscheid in complexiteit van produkt, produktieproces en middelen, en uitgaat van slechts twee mogelijke waarden van complexiteit (hoog versus laag), kan men in principe $2^3 = 8$ verschillende beheersingsituaties onderkennen. Van deze beheersingsituaties hebben er vier echte realiteitswaarde. Voor een onderbouwing daarvan wordt verwezen naar [109]. De vier beheersingsituaties zijn in trefwoorden samengevat in fig. 5.4.

Situatie 1 kenmerkt zich door een lage complexiteit van produkt, produktieproces en middelen. Welk software-produkt moet worden gemaakt en hoe dat moet gebeuren, is precies bekend. Er zijn voldoende middelen van voldoende kwaliteit aanwezig. Zo'n beheersingsituatie is relatief eenvoudig. Het type probleem is een *realisatieprobleem*: men moet het gevraagde software-produkt gewoon maken! Het doel van de besturing ligt in een *zo efficiënt mogelijke inzet* van middelen in een zo kort mogelijke doorlooptijd. Aangezien in deze situatie produkt, produktieproces en middelen bekend zijn, kan men genormeerd werken. Men kent *standaards* voor het produkt ('output'), de wijze van ontwikkelen (produktieproces) en voor de middelen ('input'). De ontwikkeling verloopt recht toe, recht aan (*lineair*) in een strakke fasering, projectmatig en welhaast *strikt rationeel*. Begroten vooraf kan door een *begrotingsmodel* door te rekenen dat wordt gevoed door, en gecalibreerd door projectervaring uit het verleden. Een begroting is taakstellend: men beheerst het project qua tijd, geld, organisatie, informatie en kwaliteit.

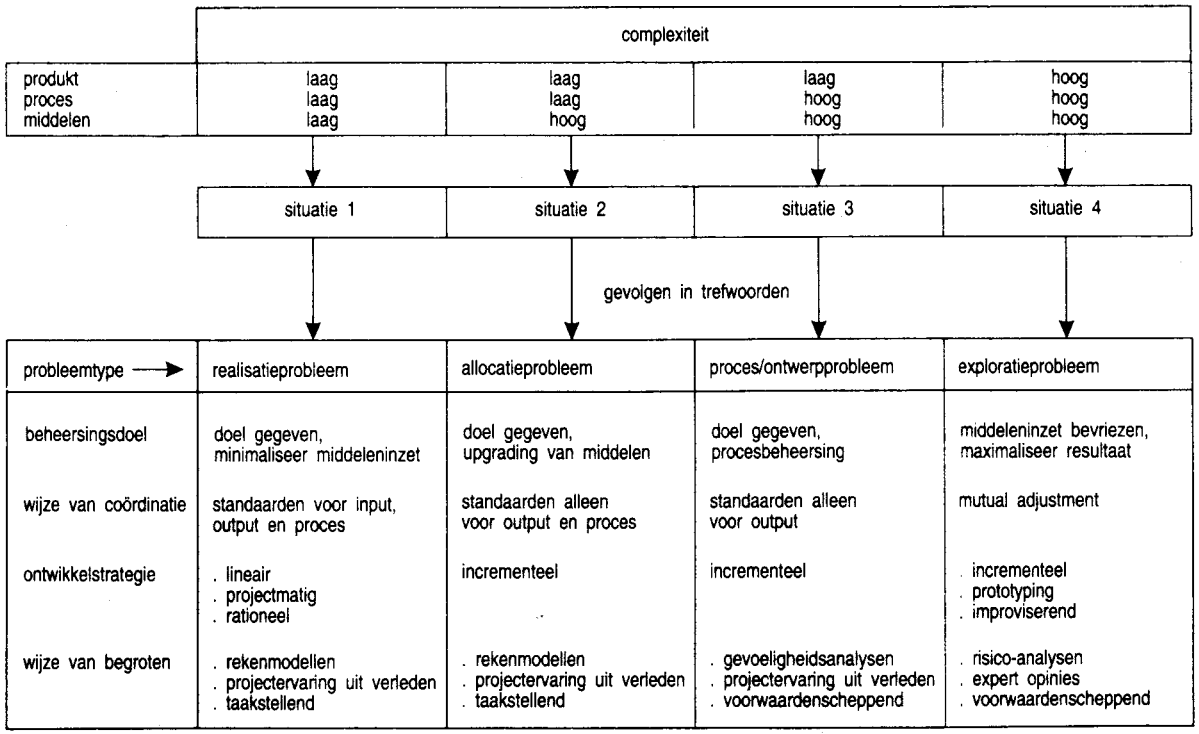

Fig. 5.4 Vier beheersingsituaties

Beheersingsituatie 1 is weliswaar zwart-wit getekend, maar bedoeld om aan te geven hoe relatief eenvoudig de beheersing in dit geval is.

Tegenover situatie 1 staat situatie 4 met een hoge complexiteit van produkt, produktieproces en middelen. Hier weet men vooraf niet precies welk software-produkt met welke kwaliteit moet worden ontwikkeld. In een project moet dat vaak eerst worden onderzocht. In dat geval zal eveneens onzeker en onduidelijk zijn hoe men dat software-produkt kan maken (produktieproces) en welke middelen en hoeveel men ervan nodig heeft. Het type probleem is niet de realisatie, 'het doen', maar de *exploratie*, het 'denken' en 'ontdekken'. Hergebruik van eerder bedachte oplossingen of eerder gemaakte software is hier veel minder of zelfs niet aan de orde. Dus men kan de output, de input en het produktieproces niet of nauwelijks standaardiseren. Men bereikt geen overeenstemming door het principe 'baas boven baas' ('direct supervision'), maar door onderlinge afstemming ('mutual agreement') in een voortdurend zoeken en tasten. Strikt projectmatig sturen is hier uit den boze en draait de noodzakelijke creativiteit de nek om. Anderzijds is het geven van een vrijbrief aan de projectmedewerkers (qua tijd en budget) ook niet verstandig. Daarom geldt hier dat men niet taakstellend begroot, maar voorwaardenscheppend. Het management reserveert een bepaald bedrag en een bepaalde tijd voor een onderzoekteam (of voor meer teams bij parallel onderzoek!). Binnen dat budget en de toegemeten tijd moet een team trachten een zo maximaal mogelijk resultaat te bereiken. Hoe en met welke middelen moet het team binnen de kaders van het toegewezen budget in geld en tijd zelf bepalen. Het mag duidelijk zijn dat hier een lineaire, projectmatige aanpak faalt. Men kan niet recht toe, recht aan op het doel afstevenen. Men moet immers dat doel grotendeels nog bepalen. Vandaar een incrementele, evolutionaire werkwijze ondersteund met prototypen en probeersels. Men werkt niet strikt rationeel, maar improviseert en zoekt.

De situaties 2 en 3 zijn 'grijstinten' van de hiervoor geschetste extreme situaties. In situatie 2 is sprake van een hoge complexiteit van middelen omdat de middelen er niet zijn, of onzeker is of deze beschikbaar komen in de gewenste kwantiteit en kwaliteit. Hier is de kern van het probleem van beheersing het *allocatievraagstuk* met alle maatregelen die in dat kader passen (werven, trainen en opleiden, uitbesteden, enz.).
Situatie 3 is complexer omdat niet alleen de middelen geheel of gedeeltelijk onbekend zijn, maar ook de activiteiten in het produktieproces naar soort en volgorde geheel of gedeeltelijk onbekend zijn. Het probleem van beheersing verschuift dan van een probleem van allocatie naar een probleem van *procesontwerp- en procesbeheersing*. Daarmee wordt de beheersing van zo'n situatie meer complex en gaat meer lijken op de extreme situatie 4.

5.7.5 SLOTBESCHOUWING

In dit artikel hebben we beschreven dat complexiteit van software en van softwareprojecten een hoogst relevant onderwerp is. Complexiteit heeft immers grote repercussies voor de kosten en de doorlooptijd van projecten. Toen automatisering

zich slechts manifesteerde in de marge van de bedrijfsvoering, was het overschrijden van een budget weliswaar vervelend, maar niet 'dodelijk'. Die situatie is grondig gewijzigd. Voor het meeste organisaties is software tegenwoordig van cruciaal (strategisch) belang.

Complexiteit blijkt een weerbarstig onderwerp als men objectieve maatstaven en criteria wil aanleggen. Veel factoren blijken de complexiteit te bepalen en daarmee de produktiviteit van software engineers en de beheersbaarheid van softwareprojecten. In die zin staat software engineering niet aan het einde, maar aan het begin van een ontwikkeling. Het zal nog veel onderzoek vergen voordat men fabrieksmatig software kan fabriceren volgens de produktiewijzen en met de produktie-organisaties, waarmee we al decennia vertrouwd zijn bij de fabricage van traditionele produkten.