Requirements Engineering

Hans van Vliet (*), Sjaak Brinkkemper (**)
(*) Hoogleraar Software Engineering, Vrije Universiteit, Amsterdam
(**) Senior Process Engineer, Baan Research and Development, Barneveld

Abstract

Tijdens de requirements engineering fase worden de eisen van gebruikers en andere belanghebbenden geïdentificeerd en gedocumenteerd. We onderscheiden hierbij drie deelprocessen:

- Elicitatie: het *begrijpen* van het probleem waarvoor een (geautomatiseerde) oplossing wordt gezocht.
- Specificatie: het *beschrijven* van het probleem.
- Validatie: het onderling *eens worden* over het probleem.

De essentie van elke van deze fasen wordt in dit artikel kort toegelicht. We besluiten met een korte geannoteerde bibliografie voor de geïnteresseerde lezer.

1. Inleiding

Tijdens de requirements engineering fase worden de eisen van gebruikers en andere belanghebbenden ("stakeholders") zorgvuldig geïdentificeerd en gedocumenteerd. Deze eisen betreffen zowel de functionaliteit -- wat moet het systeem de toekomstige gebruikers gaan bieden -- alsook de kwaliteit -- hoe snel moet het zijn, hoe betrouwbaar, in hoeverre gaat het om privacy-gevoelige zaken, enz.

Requirements zijn eisen aan het toekomstig systeem of aan het ontwikkelproject. Een veelgebruikte indeling van requirements is [Robertson 99]:

- Functionele requirements: acties die het systeem moet kunnen uitvoeren; bijvoorbeeld *“Het catalogussysteem moet weergeven of een boek beschikbaar, uitgeleend, of in bestelling is”*
- Niet-functionele requirements: eigenschappen of kwaliteiten die het systeem moet hebben; bijvoorbeeld: *“Het catalogussysteem moet binnen 3 seconden respons geven op ieder willekeurig informatieverzoek*”.
- Randvoorwaarden: zijn globale eisen die de context van het gehele product en project betreffen, zoals de doelstelling van het product en de typering van de toekomstige gebruikersgroepen; bijvoorbeeld *“Het catalogussysteem zal gebruikt worden door medewerkers en bezoekers van de bibliotheek”.*
- Project Issues: eisen die betrekking hebben op het project; bijvoorbeeld *“Het catalogussysteem dient op 1 maart 2002 operationeel te zijn”.*

We gebruiken met opzet de term "requirements engineering", in plaats van bijvoorbeeld "requirements analyse", om te benadrukken dat het een constructief proces is van analyseren, documenteren, en toetsen van de verzamelde kennis. In principe stellen we ons in deze fase nog niet de vraag *hoe* een en ander gerealiseerd moet worden; dat komt tijdens de ontwerpfase aan de orde. De scheiding tussen het *wat* en het *hoe* is echter heel lastig te trekken.

Ook de scheiding tussen de *fasen* requirements engineering en (architectuur) ontwerp is moeilijk precies te maken. Soms is het lastig in te zien hoe een bepaalde eis gerealiseerd kan worden, en maakt men een ontwerp, of zelfs een compleet prototype, om te zien hoe een en ander uitpakt. Ook kan, zonder een architectuurontwerpstap, veelal niet beoordeeld worden of en hoe aan de gestelde kwaliteitseisen kan worden voldaan, of hoe afwegingen tussen kwaliteitseisen uitpakken.

Wanneer we requirements engineering definiëren als een proces voor het identificeren, documenteren, en valideren van gebruikerseisen, veronderstellen we stilletjes dat er ook zo'n *gebruiker* beschikbaar is om in dit proces te participeren. Veel software-ontwikkeling echter is *marktgedreven* in plaats van *klantgedreven*. Bijvoorbeeld zullen we een systeem voor bibliotheekautomatisering niet gauw ontwikkelen voor één specifieke bibliotheek. In plaats daarvan ontwikkelen we een `generiek' bibliotheeksysteem. De eisen voor zo'n generiek systeem krijgen we boven tafel door meer algemeen te bestuderen wat er in een bibliotheek gebeurt, terwijl afwegingen betreffende functionele en kwaliteitseisen gebaseerd zijn op onder meer marktoverwegingen. Bijvoorbeeld kunnen we besluiten dat ons systeem geschikt moet zijn voor de doorsnee openbare bibliotheek in ons land, maar dat het niet hoeft te werken voor de Koninklijke Bibliotheek. Ook kunnen we besluiten dat het systeem zeker moet kunnen samenwerken met financieel systeem Y, omdat dat in bijna alle openbare bibliotheken wordt gebruikt.

We onderscheiden binnen requirements engineering drie deelprocessen [Loucopoulos95]:

- **Requirements elicitatie**. In het algemeen is de requirements analist geen expert in het domein dat hij moet analyseren. Door interactie met bijvoorbeeld professioneel bibliotheekpersoneel moet hij zich een beeld vormen van wat daar speelt. Requirements elicitatie gaat dus over het *begrijpen* van het probleem. Dit proces wordt gecompliceerd door het feit dat er verschillende disciplines bij betrokken zijn. Veelal kan de analist niet volstaan met een rol als buitenstaander, die alleen maar vraagt en documenteert wat de betrokkenen willen. Hij moet misschien een keuze maken tussen conflicterende eisen, of een standpunt innemen in een machtsstrijd tussen betrokkenen.
- **Requirements specificatie**. Nadat het probleem is begrepen, moet het *beschreven* worden. Er zijn nogal wat technieken voor het vastleggen van requirements, variërend van informeel (natuurlijke taal en plaatjes) tot zeer formeel (wiskundig).
- **Requirements validatie en verificatie**. Nadat het probleem is beschreven, moeten de verschillende belangstellenden het *eens worden*. We moeten vaststellen dat de juiste eisen zijn vastgelegd (validatie), en dat deze eisen juist zijn vastgelegd (verificatie).

Vanzelfsprekend vindt er iteratie en terugkoppeling plaats tussen deze processen. Voor marktgedreven systeemontwikkeling komt de ontwikkeling van specifieke technieken voor requirements engineering pas recentelijk op gang. Het essentiele verschil ligt in de planning van de requirements voor de opeenvolgende releases van het softwareproduct. Tijdens requirements specificatie worden capaciteitsschattingen aan de requirements toegevoegd, zodat een optimale selectie van de requirements gemaakt kan worden die past binnen de beschikbare capaciteit en het tijdvak tot aan de eerstvolgende release-datum.

Management heeft tot taak het requirements engineering proces zorgvuldig te plannen en te sturen. Dit houdt dus in het kiezen van een bepaalde aanpak voor elk van de hierboven genoemde deelprocessen, het inschatten en alloceren van de benodigde menskracht hiervoor (hoe lang en uitgebreid gaan we requirements trachten te achterhalen, met welke betrokkenen praten we daarover, enz.), en het zorgen voor een adequate interactie met de opdrachtgever en toekomstige gebruikers over de specificaties. Het succes van het gehele automatiseringsproject is in belangrijke mate afhankelijk van de kwaliteit van de requirements engineering fase. Slechte of onhelder geformuleerde eisen leiden in latere fasen tot ontevreden klanten, veel fouten, en veel extra werk.

2. Requirements elicitatie

Tijdens requirements engineering modelleren we een deel van de werkelijkheid. Het deel van de werkelijkheid dat we modelleren wordt ook wel Universe of Discourse (UoD) genoemd. Voorbeelden zijn: een bibliotheek, een lift, de loonadministratie van een bedrijf.

Het model dat we maken tijdens requirements engineerng is een *expliciet* model. Het voorvoegsel `expliciet' duidt erop dat we het model moeten kunnen communiceren met andere betrokkenen, zoals de gebruiker en de opdrachtgever. Opdat alle betrokkenen weten waarover ze praten moet het model **alle** relevante informatie bevatten. En daar zit een van de

grootste problemen: hoe zorgen we er voor dat het model alle relevante informatie bevat, en dat alle niet-relevante informatie is weggelaten?

Bij ons bibliotheekautomatiseringsproject bijvoorbeeld is het goed mogelijk dat we over het hoofd zien dat de naam van de auteur zoals die op de kaft van een boek staat vermeld wel eens anders kan zijn dan de zogenoemde kanonieke auteursnaam. Dit treedt met name op bij schrijvers uit landen waar men een ander schrift hanteert. Op Nederlandse vertalingen van boeken van Θo€?% staat `Tsjechow', en op Engelse vertalingen van dezelfde auteur staat `Chekhov'. We willen in zo'n geval in ons systeem twee namen opnemen: een naam zoals gespeld op de kaft, en een `kanonieke' naam die in allerlei zoekopdrachten wordt gebruikt.

Subtiele verschillen tussen de opvatting die de analist heeft van termen en concepten, en hun echte betekenis in het domein, kunnen grote gevolgen hebben. Zoiets kan met name gemakkelijk optreden in domeinen die we wel denken te kennen, zoals in ons voorbeeld een bibliotheek. Andere voorbeelden van zaken waar we in ons voorbeeld op moeten letten, zijn:

- Een medewerker van een bibliotheek kan ook gewoon lid van de bibliotheek zijn, en die verschillende rollen van een en dezelfde persoon moeten we dus scheiden;
- Er is een verschil tussen een boek (zoals bijvoorbeeld geïdentificeerd door een ISBN nummer), en de (fysieke) exemplaren van een boek zoals die in de bibliotheek aanwezig zijn;
- Het is niet voldoende om als status van een boek alleen de waarden `aanwezig' en `uitgeleend' te onderscheiden. Bijvoorbeeld kan een boek, of beter gezegd een exemplaar van een boek, ook verloren zijn, of in reparatie, of in bestelling.

De personen die betrokken zijn bij een bepaald domein hebben een *impliciet* model van dat domein. Zo'n impliciet model bevat de achtergrondkennis die door de betrokkenen gedeeld wordt. Voor betrokkenen is die kennis vanzelfsprekend. Dat uit zich in uitspraken die beginnen met "natuurlijk …" ("Natuurlijk is er een verschil tussen een boek en een exemplaar van een boek"). Een deel van dit impliciete model is niet onder woorden gebracht. Het gaat hier bijvoorbeeld om kennis die deskundigen routinematig gebruiken. Het impliciete model bevat ook gewoontes, vooroordelen, en inconsistenties.

De analist zet een impliciet model om in een expliciet model. Hierbij doen zich twee soorten problemen voor: analyse-problemen en onderhandelings-problemen. Analyse-problemen treden op omdat een deel van het impliciete model niet geverbaliseerd is, omdat dit impliciete model niet constant in de tijd is, de gebruiker en de analist verschillend vakjargon spreken. Onderhandelings-problemen treden op omdat sommige betrokkenen het proces kunnen tegenwerken (ze kunnen bijvoorbeeld hun baan verliezen, of minder interessant werk krijgen), of omdat de impliciete modellen van mensen niet hetzelfde hoeven zijn, of betrokkenen tegenstrijdige belangen hebben.

2.1. Analyseproblemen

Het probleem dat een geautomatiseerd systeem moet oplossen ligt bij de gebruiker. Die moet dat probleem correct en volledig beschrijven, ten overstaan van iemand die in het algemeen geen diepe kennis van het betrokken vakgebied heeft. De analist moet dus de taal van dat vakgebied leren, bekend worden met de terminologie, concepten en procedures. Vooral in grote projecten is de benodigde kennis vaak verspreid over een groot aantal personen, wat al snel kan leiden tot integratie- en coördinatieproblemen. Nog niet zo lang geleden is een missie naar de planeet Mars mislukt omdat de ene helft van de programmatuur "dacht" dat afstanden werden uitgedrukt in kilometers, terwijl de andere helft uitging van de mijl als eenheid van afstand.

Het achterhalen van correcte en volledige informatie is geen sinecure. Het gewoonweg vragen aan de toekomstige gebruiker werkt vaak niet. We krijgen dan al gauw een onvolledig en onnauwkeurig beeld. Een van de redenen hiervoor is dat het menselijk geheugen nogal beperkt is. We herinneren ons recente kwesties, en dan nog niet al te veel. Het menselijk korte-termijn geheugen is beperkt van omvang. Zoals we zonder ezelsbruggetje geen

getallen met meer dan 7-10 cijfers kunnen onthouden, zo ook kunnen we uit ons hoofd maar een beperkt aantal functies van een toekomstig systeem formuleren.

Mensen kunnen ook niet goed rationeel denken. Ze vereenvoudigen dingen, en gebruiken een model dat niet helemaal overeenkomt met de werkelijkheid. Ook opleiding, ervaring e.d. kleuren ons model van de werkelijkheid, en dus ook de requirements die we opstellen.

We kunnen ook niet echt van de gebruiker verwachten dat hij in een heel vroegtijdig stadium zijn eisen precies formuleert. Een belangrijke reden voor automatisering is dat we niet tevreden zijn met de huidige situatie. Gewoon de huidige situatie automatiseren is dus vast niet de oplossing. We willen iets anders, maar weten niet goed wat. (Of automatisering dan de oplossing is, is overigens de vraag. Veel automatiseringsproblemen zijn eigenlijk organisatieproblemen.) We zijn vaak pas in staat nauwkeurig aan te geven wat we willen wanneer we een geautomatiseerd systeem hebben en daarmee werken. Dit is een van de redenen van het zo omvangrijke onderhoudsprobleem. Prototyping, iteratieve en evolutinaire ontwikkelmethoden erkennen dit leerproces, en verdienen dus veelal de voorkeur boven de meer traditionele, lineaire ontwikkelmodellen.

Door een nauwkeurige analyse kunnen we proberen ons een precies beeld van de huidige en toekomstige gebruikerswensen te vormen. Hoeveel aandacht we hier ook aan geven, de mate waarin we de toekomst kunnen voorspellen is zeer beperkt. Requirements engineering lijkt wel een beetje op weersvoorspelling. Het weer van volgend jaar weten we niet. Wat de gebruiker volgend jaar wil, weten we ook niet. In die zin zullen de geïdentificeerde requirements nooit volledig zijn.

2.2. Onderhandelingsproblemen

De meeste requirements engineering methoden, en eigenlijk geldt dit voor software-ontwikkelingsmethoden in het algemeen, zijn Tayloriaans van aard. Rond 1900 introduceerde Taylor het begrip `wetenschappelijk management'. Hierin worden taken recursief opgedeeld in eenvoudiger taken, en voor elke taak is er een beste manier om die uit te voeren. Door zorgvuldige observatie en analyse kan die beste manier gevonden worden, en vervolgens vastgelegd in procedures en regels. Menige lopende band is zo, en met succes, opgezet. Het equivalent bij requirements engineering is dat we gebruikers interviewen en observeren, teneinde de `echte' wensen te achterhalen. Als dat gebeurd is gaan de experts aan het werk om deze wensen te implementeren. Gedurende dit laatste proces hoeven we niet meer te praten met de gebruikers; we weten immers precies wat ze willen. Dit is een functionele, rationele, kijk op software ontwikkeling. De onderliggende aanname is dat er één objectieve waarheid is, die we alleen maar hoeven te ontdekken tijdens requirements engineering.

In puur technische domeinen, zoals bijvoorbeeld automatisering van een gsm of scheerapparaat, werkt dit wellicht. In domeinen waar ook *mensen* voorkomen werkt het meestal minder goed. In die gevallen is de analist geen passieve buitenstaander. Hij doet actief mee bij het vormgeven van het UoD.

De volgende gedachtengang helpt ons zicht te krijgen op alternatieven voor de Tayloriaanse werkwijze. Als we iets bestuderen, hanteren we bepaalde aannames omtrent het onderwerp van studie. We noemen zo'n verzameling aannames ook wel paradigma. In ons gebied betreffen deze aannames de wijze waarop de analist zijn kennis opdoet (epistemiologische aannames) en hun kijk op de wereld (ontologische aannames).

De aannames omtrent kennis resulteren in een dimensie met objectief en subjectief als uitersten. Bij het objectivistische uitgangspunt hanteert de analist modellen en methoden uit de natuurwetenschappen om tot de enige waarheid te komen. Bij het andere, subjectivistische, uiterste, gaat het erom te begrijpen hoe een individu de wereld waarin hij verkeert, creeert, modificeert, en interpreteert. De aannames over de wereld om ons heen resulteren in een orde-conflict dimensie. Aan de orde-zijde ligt de nadruk op regelmaat, stabiliteit, integratie, consensus. Aan de conflict-zijde gaat het om verandering, desintegratie, tegenstand.

Deze twee dimensies en hun uiterste waarden geven zo vier paradigma's voor requirements engineering en, meer in het algemeen, systeemontwikeling:

- **Functionalisme** (objectief + orde). De ontwikkelaar is een expert die op zoek is naar meetbare oorzaak-gevolg relaties. We geloven dat er een empirische werkelijkheid is die los staat van de observator. Systemen worden ontwikkeld ter ondersteuning van rationele bedrijfsprocessen. Hun effectiviteit kan objectief gemeten worden, net zoals dat in andere ingenieurs-disciplines gebeurt.
- **Sociaal-relativisme** (subjectief + orde). De analist is hier veranderingsagent. De werkelijkheid is niet iets vaststaands `om ons heen', maar wordt door de mens geconstrueerd. De analist ondersteunt het leerproces van de betrokkenen, en functioneert als een soort vroedvrouw.
- **Radicaal-structuralisme** (objectief + conflict). De fundamentele aanname in dit paradigma is dat systeemontwikkeling ingrijpt in het conflict tussen twee of meer sociale klassen, op het gebied van macht, prestige, middelen. Vaak worden systemen ontwikkeld ter ondersteuning van de baas, en ten koste van de werknemers. Om het machtsevenwicht te herstellen, zou de analist als vakbondsman moeten optreden; eisen worden dan opgesteld in samenspraak tussen gebruikers en de analist. Deze aanpak zou moeten leiden tot systemen die het vakmanschap en goede werkomstandigheden helpen bevorderen.
- **Neohumanisme** (subjectief + conflict). Centraal thema hierbij is emancipatie, De analist is een soort sociaal therapeut die probeert om, in een open discussie, de verschillende groepen belanghebbenden tot elkaar te brengen.

Deze paradima's vertegenwoordigen natuurlijk extremen. In werkelijkheid zal er een zekere mix van aannames worden gehanteerd. Feit is echter dat in de meeste systeemontwikkelingstechnieken het functionalisme de overhand heeft. Dit is in veel technische domeinen zeker niet verkeerd. Als we programmatuur voor de besturing van een kopieerapparaat moeten ontwikkelen, kunnen we met een gerust hart een dergelijke benadering kiezen. Voor een systeem dat mensen in hun werk ondersteunt, ligt de zaak anders. In dat geval is gebruikersparticipatie zeer belangrijk. Middels een open dialoog kunnen gebruikers invloed uitoefenen op het toekomstige systeem. Het is in dit geval onder meer de taak van de analist om de verschillende wensen van betrokkenen met elkaar in overeenstemming te brengen. Terugkoppeling is in dit geval belangrijk om gebruikers gelegenheid tot bijsturing te geven. De toekomstige gebruikers moeten immers met het systeem gaan werken. Een ontevreden gebruiker zal trachten het systeem te negeren. Op zijn best komt hij meteen met een lijst nieuwe wensen.

Een zeer illustratief voorbeeld van de mogelijke effecten van een radicale functionele werkwijze betreft de automatisering van het Londense ambulance verkeer. Hoewel dit systeem nogal wat invloed zou hebben op de werkwijze van het ambulance personeel, werden zij nauwelijks betrokken bij het formuleren van de eisen. Het management stelde vast wat het systeem moest gaan doen, en het personeel had maar te volgen. Enkele gevolgen hiervan:

- Het systeem wees de dichtsbijzijnde vrije ambulance toe aan een ongeluk, onafhankelijk van de thuisbasis van die ambulance. Als gevolg hiervan raakten ambulances regelmatig steeds verder van hun thuisbasis verwijderd. Het ambulancepersoneel kwam dan terecht in buurten waar ze de weg niet meer wisten. Ook moesten ze dan, in hun eigen tijd, een veel langere rit naar hun thuisbasis maken.
- Het systeem nam het personeel ook de vrijheid af om met een willekeurige ambulance weg te rijden. Dit leidde tot allerlei problemen als iemand zich daar niet aan hield, bijvoorbeeld omdat de hem toegewezen ambulance ingeparkeerd stond en hij zijn taak als redder van mensenlevens belangrijker achtte dan gehoorzaamheid aan de baas. Een volgende chauffeur ging dan tevergeefs op zoek naar de hem toegewezen ambulance.

2.3. Technieken voor requirements elicitatie

Er zijn twee belangrijke informatiebronnen voor het elicitatieproces: de gebruikers en het applicatiedomein. We nemen hierbij aan dat er zoiets is waarop we de requirements kunnen

baseren. In marktgedreven software ontwikkeling is dat vaak niet het geval, en requirements engineering lijkt dan meer op het `uitvinden' van requirements, gestuurd door marketing en verkoop overwegingen. In figuur 1 wordt een aantal elicitatietechnieken opgesomd. Hierbij is aangegeven of de gebruiker of het domein de belangrijkste inspiratiebron is. Ook is aangegeven of de techniek vooral geschikt is om de huidige dan wel de toekomstige situatie te analyseren. Je kun een tapijt beter in twee richtingen stofzuigen dan in een; dan haal je meer vuil op. Op dezelfde wijze is het raadzaam meer dan een requirements elicitatietechniek te gebruiken.

Techniek	Belangrijkste bron van informatie		Vooral te gebruiken voor	
	domein	gebruiker	heden	toekomst
Interview		X	X	
Delphi techniek		X	X	
Brainstorm		X		X
Taakanalyse		X	X	
Scenario analyse		X	X	X
Ethnografie	X		X	
Formulier analyse	X		X	
Analyse natuurlijke-taal	X		X	
Synthese bestaand systeem	X		X	
Domeinanalyse	X		X	
Gebruik referentiemodel	X		X	
Business Process Redesign	X		X	X
prototyping		X		X

Figuur 1: Een aantal requirements elicitatie technieken

- **Vragen**. We kunnen simpelweg de gebruiker vragen wat hij wil. Dat vragen kan via een interview, brainstorm, vragenlijst. Een open interview is het eenvoudigst, maar heeft ook alle bezwaren als in sectie 2 opgesomd. In een groepsdiscussie zie je vaak dat enkele deelnemers de boventoon voeren en zo het resultaat sterk beïnvloeden. De Delphi-techniek kan dan uitsluitsel geven. De Delphi-techniek is een iteratieve techniek waarin informatie schriftelijk wordt uitgewisseld totdat er een consensus is bereikt.
- **Taakanalyse**. Werknemers in een bepaald domein voeren zekere taken uit, zoals het behandelen van een verzoek om een boek te reserveren, het uitlenen van een boek, en het inschrijven van een nieuw lid. Hoog-niveau taken kunnen worden opgedeeld in subtaken. Bijvoorbeeld kan de taak `leen een boek' worden opgesplitst in:
  - Controleer lidmaatschap
  - Controleer maximum aantal te lenen boeken
  - Registreer dat dit boek nu is uitgeleend
  - Geef de klant een briefje met de uiterste retourdatum

  Taakanalyse is een techniek om een hierarchie van taken en subtaken af te leiden. Elke andere genoemde elicitatietechniek kan gebruikt worden om de benodigde informatie boven tafel te krijgen. Er is geen duidelijk criterium om te stoppen met verder opsplitsen van taken. Vuistregel om te stoppen is dat gebruikers op zeker ogenblik `weigeren' meer gedetailleerde informatie te geven. Taakanalyse wordt vaak toegepast bij het bepalen van de systeem-gebruikers dialoog, maar kan ook heel goed in een eerder stadium worden gebruikt.
- **Scenario-gebaseerde analyse**. Bij interviews en taakanalyse zijn we op zoek naar algemeen geldende plannen. Bij scenario-gebaseerde technieken zijn we op zoek naar concrete voorbeelden. Bijvoorbeeld kan de analist kijken hoe een specifiek boek wordt uitgeleend, en de stappen daarin precies noteren. Als hulpmiddel kun je de medewerker hardop laten vertellen wat hij doet. Scenario-gebaseerde analyse wordt vaak gebruikt in object-georiënteerde ontwikkelmethoden, en wordt dan meestal `use-case analyse' genoemd. Een van de lastige aspecten van scenario-gebaseerde analyse is het aanbrengen van structuur in het proces: welke scenarios zijn belangrijk, wanneer heb je

er genoeg, is het zinvol nog verder te zoeken naar scenarios, e.d., zijn vragen die lastig zijn te beantwoorden.

- **Ethnografie**. Een nadeel van bijvoorbeeld interviews is dat de analist zijn wereldbeeld aan de gebruiker opdringt. Bijvoorbeeld kan de analist het volgende vragen: "Als een lid van de bibliotheek een boek wil lenen en nog een boete heeft uitstaan, weiger je dan het verzoek, of willig je dat verzoek toch in?" De praktijk van de bibliotheek medewerker is wellicht complexer. Misschien willigt hij het verzoek in als een deel van de boete wordt betaald, of als hij het lid wel kent en vertrouwt.
  Ethnografische methoden zouden dat soort tekortkomingen niet hebben. Bij ethnografie worden mensen bestudeerd in hun natuurlijke omgeving. De methode is vooral bekend uit de sociologie, waarin primitieve stammen worden bestudeerd door geruime tijd met hen samen te leven. Als ethnografie wordt toegepast in requirements engineering, wordt de analist een soort leerling die meeloopt met de domeinexperts.
- **Analyse van formulieren**. Veel domeinspecifieke informatie is vastgelegd in allerlei formulieren. Formulieren geven ons nuttige informatie over de gegevensobjecten in een domein, hun eigenschappen en hun onderlinge relaties. Bijvoorbeeld, om de proceedings van een conferentie te bestellen, zou men gevraagd kunnen worden een formulier in te vullen waarop onder meer de plaats wordt gevraagd waar de conferentie plaatsvond. Ik weet misschien niet precies hoe de laatste internationale software engineering conferentie heette, maar ik weet wel dat hij in Toronto plaatsvond. Dit soort vragen kan alleen beantwoord worden als deze onderdelen van het aanvraagformulier ook in het onderliggende datamodel zijn opgenomen. Zo'n formulier wijst dus direct op een nuttige requirement.
- **Beschrijving in natuurljke taal**. Net als formulieren geven allerlei documenten in natuurlijke taal veel nuttige informatie over het te modeleren domein. Vaak zijn deze documenten geschreven voor de domeinexpert, en dus vereisen ze nogal wat achtergrondkennis. Net als de documentatie van software hebben dit soort documenten de neiging niet aangepast te worden bij veranderingen, zodat ze niet altijd up-to-date zijn.
- **Synthese van een bestaand systeem**. We kunnen ook de eisen voor een systeem bepalen door die van een vergelijkbaar systeem als uitgangspunt te nemen. Een variant hierop is **domein engineering**. Dit is een soort techniek op meta niveau, waarbij we niet de eisen aan één bepaald systeem zoeken, maar de eisen aan een familie gelijksoortige systemen. Domein analyse kan ok gebruikt worden om voor een bepaalde klasse systemen een `referentie' model te bepalen. Zo'n referentiemodel geeft ook vaak aanleiding tot een bijbehorende referentie-architectuur, of framework dat als uitgangspunt voor realisatie dient.
- **Business Process Redesign (BPR)**. BPR is een techniek om op radicale wijze de bedrijfsprocessen opnieuw te ontwerpen, teneinde een doorbraak te krijgen met betrekking tot bijvoorbeel kosten, kwaliteit, of gebruikerstevredenheid. BPR is niet echt een requirements elicitatietechniek. Het benadrukt echter wel een essentieel aspect: bedrijfsprocessen moeten worden aangestuurd door ICT, en niet andersom. Een complete BPR-slag is bij veel automatiseringsproblemen niet nodig, maar het opnieuw overdenken van bestaande processen en procedures is nuttig, en wordt al te vaak overgeslagen.
- **Prototyping**. Soms is de opdrachtgever onzeker over de requirements van het te bouwen systeem. Het is dan lastig, zo niet onmogelijk, in een keer het juiste systeem te bouwen, zodat we kunnen besluiten prototypes te gebruiken. Beginnend met een eerste verzameling requirements wordt een eerste versie van het systeem gebouwd. Hiermee kunnen gebruikers ervaring opdoen, en die ervaring wordt gebruikt om verdere eisen te formuleren. Op die wijze groeit het systeem naar zijn uiteindelijke vorm toe.

Van deze technieken is vragen de minst zekere, en prototyping de minst onzekere. Het hangt af van de ervaring van de gebruikers en de analist, de stabiliteit van de omgeving, de complexiteit van het probleem en de bekendheid van de betrokkenen daarmee, weke techniek je het beste kunt gebruiken. Bij een goed-begrepen probleem, met ervaren analisten, kan een interview met gebruikers volstaan. Bij een slecht-begrepen probleem op onbekend terrein is het verstandiger eerst een of meer prototypes te ontwikkelen. Enige voorzichtigheid is hier op zijn plaats. We zijn al te snel geneigd onszelf en onze kennis te overschatten, met name waar het de extrapolatie naar een "iets" andere omgeving betreft.

Een Nederlands bedrijf, goed in procesautomatisering, had met succes een boerderij geautomatiseerd. Elke koe kreeg een chip in haar oor, en werd vervolgens bij al haar activiteiten gevolgd. De toevoer van water en voer werd automatisch geregeld en aangepast, de hoeveelheid en kwaliteit van de melk werd gemeten, enz. Dezelfde techniek werd vervolgens, met succes, toegepast bij varkens. Daarna werd het geprobeerd bij geiten. Er werd, voor vele miljoenen, een volautomatische geitenboerderij opgezet. Maar helaas, dat werkte niet echt goed. Geiten eten alles, ook de chips uit elkaars oren.

3. Het requirements specificatie document

Het eindproduct van requirements engineering is een requirements specificatie. Het is een reconstructie achteraf van de resultaten van deze fase. Het doel is de resultaten naar anderen te communiceren. Het is een ijkpunt voor diverse partijen. De klant ziet hier beschreven waar hij om gevraagd heeft. De ontwerpers zien hier wat ze moeten gaan realiseren. De testers zien hier waartegen ze het systeem moeten valideren.

De requirements specificatie moet aan nogal wat eisen voldoen:

- Hij moet leesbaar en begrijpelijk zijn. Diverse belanghebbenden baseren hier immers hun beslissingen op.
- Hij moet correct zijn, dat wil zeggen dat de betekenis in het UoD op een adequate manier wordt gerepresenteerd. Correctheid is helaas niet te bewijzen, en daarom moet het document op andere wijze getoetst worden tegenover bijvoorbeeld gebruikerswensen.
- Hij moet voor maar één uitleg vatbaar zijn. Requirements moeten maar op een manier kunnen worden geïnterpreteerd, en dat is per definitie lastig bij natuurlijke taal.
- Hij moet volledig zijn. Alle belangrijke zaken met betrekking tot functionaliteit, prestaties, e.d., moeten zijn gedocumenteerd. Voor alle functies moet zowel worden aangegeven wat het systeem moet doen bij juiste invoer, en hoe het moet reageren bij onjuiste invoer. Met name zogenoemde "TBD" zinsneden ("to be determined") zijn heel gevaarlijk.
- Hij moet (intern) consistent zijn. Requirements kunnen zowel logisch als temporeel tegenstrijdig zijn. Ook het gebruik van verschillende termen voor een en hetzelfde kan leiden tot tegenstrijdigheden.
- Requirements moeten geordend worden qua belang of stabiliteit. Meestal zijn sommige requirements belangrijker dan andere. Vaak is een eenvoudige ordening, als `essentieel', `belangrijk' en `handig' al voldoende. Stabiliteit kan aangegeven worden met een kans, of de hoeveelheid verwachte veranderingen in een bepald tijdsbestek. Dit soort informatie nodigt gebruikers uit beter over hun eisen na te denken. Het biedt ontwikkelaars de mogelijkheid hun aandacht beter te richten.
- Requirements moeten verifieerbaar zijn. Het moet uiteindelijk mogelijk zijn om te bepalen of het systeem aan de eisen voldoet. Zinsneden als `het systeem moet gebruikersvriendelijk' zijn, zijn niet te testen. Ook het gebruik van vage uitdrukkingen die niet zijn te kwantificeren, als in `de responstijd dient *meestal* onder de 2 seconden te liggen' kunnen beter worden voorkomen.
- Requirements moeten traceerbaar zijn. De oorsprong en rationale van elke requirement moet terug te vinden zijn.

Een mogelijke richtlijn voor de structuur van het requirements document wordt gegeven in IEEE Standaard 830, weergegeven in figuur 2. De precieze structuur van zo'n document is niet wezenlijk. Belangrijk is dat de inhoud voldoet aan de hierboven gegeven eisen.

1. Inleiding
   - 1.1 Doel
   - 1.2 Reikwijdte
   - 1.3 Definities en afkortingen
   - 1.4 Referenties
   - 1.5 Overzicht
2. Globale beschrijving
   - 2.1 Doel van het product
   - 2.2 Product functies

2.3 Gebruikers karakteristieken
2.4 Beperkingen
2.5 Aannames, afhankelijkheden
2.6 Deelverzamelingen van de requirements
3. Specifieke requirements
3.1 Requirements externe interface
3.1.1 Gebruikersinterfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communicatie interfaces
3.2 Functionele requirements
3.2.1 Gebruikersklasse 1
3.2.1.1 Functionele requirement 1
3.2.1.2 Functionele requirement 2
3.2.1.3 …
3.2.2 Gebruikersklasse 2
3.2.3 …
3.3 Prestatie requirements
3.4 Beperkingen aan het ontwerp
3.5 Overige requirements

Figuur 2: Globale structuur requirements engineering document volgens IEEE Standaard 830

Voor elk niet-triviaal systeem zal de sectie `Specifieke requirements' behoorlijk omvangrijk zijn. Het is dus handig hier enige verdere structuur in aan te brengen. In figuur 2 is gekozen voor een verdere opsplitsing in soorten gebruikers (bijvoorbeeld zouden we in ons bibliotheekvoorbeeld een onderscheid kunnen maken tussen functies voor leden van de bibliotheek, en functies die alleen voor het personeel bedoeld zijn). We zouden ook kunnen kiezen voor een opdeling in klassen functies, modus van het systeem (bijvoorbeeld operationeel gebruik versus opleiding), of naar het type object waarop de functies betrekking hebben.

Het IEEE raamwerk gaat ervan uit dat er een volledige verzameling requirements is. Bij een prototyping aanpak kan een dergelijk document dus pas achteraf gemaakt worden. Het beschrijft het **eind**product van het requirements engineering proces, en is minder geschikt als **tussen**product. Een situatie die zich bijvoorbeeld voor zou kunnen doen tijdens het requirements engineering proces voor ons bibliotheeksysteem is de volgende. Het bibliotheeksysteem moet om kunnen gaan met boetes voor te laat inleveren van boeken. Jan, een van de baliemedewerkers, is van mening dat deze boetes zo snel mogelijk geïnd moeten worden. De dienstverlening gaat immers achteruit als boeken niet uitleenbaar zijn omdat ze te lang bij iemand thuis blijven liggen. Claire, de directeur van de bibliotheek, denkt daar heel anders over; zij wil de boetes zo laat mogelijk innen. Boetes voor te laat inleveren vormen een mooie extra bron van inkomsten, en die heeft de bibliotheek hard nodig. Het is zeer handig om dit soort conflicterende eisen, zeker in de vroege fasen van het proces, expliciet te kunnen vastleggen in het requirements document. Hypertekst-achtige representaties bijvoorbeeld lenen zich daar goed toe.

4. Technieken voor requirements specificatie

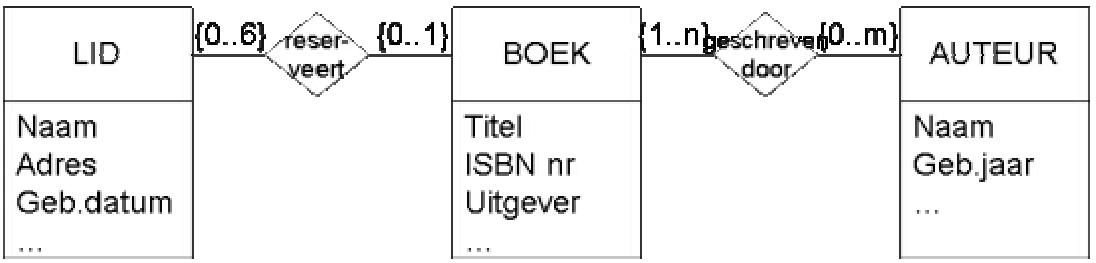
het requirements specificatie document heeft twee belangrijke doelgroepen: de gebruikers en de ontwikkelaars. Voor de gebruikers beschrijft het wat het systeem hen zal gaan leveren. Voor de ontwikkelaars beschrijft het wat ze moeten gaan realiseren. De gebruiker is het meest gebaat met een document dat zijn taal spreekt. In ons biliotheekvoorbeeld gaat het dan over `leden', `boeken', `titelbeschrijving', enz. De ontwikkelaar heeft zijn eigen taal. Daarin komen begrippen voor als `file' en `record'. Hierin moet een keuze worden gemaakt, en het is in het algemeen verstandig de keus te laten vallen op de taal van de gebruiker.

Er zijn nogal wat technieken ontwikkeld om het requirements engineering proces te ondersteunen. Meestal is de representatie een verzameling semantische netwerken. Zo'n representatie bevat dan verschillende typen knopen en verbindingen tussen knopen. Die worden visueel van elkaar onderscheiden door knopen als cirkels, vierkanten, enz. af te beelden. Deze knopen duiden dan dingen aan als processen, opslagplaatsen voor gegevens, objecten. De verbindingen als lijnen, al of niet gestippeld, al of niet met een aanduiding van de richting, enz. Deze verbindingen duiden dan relaties aan als `is een deel van', `stuurt gegevens naar', `volgt op', enz. Als voorbeelden zullen we hieronder kort ingaan op het entity-relatie diagram en de eindige automaat.

4.1 Gegevensmodellering

In gegevens-intensieve systemen is de structuur van de gegevens belangrijk. Als voorbeeld van een techniek voor het modelleren van gegevens bespreken we hier entitie-relatie modellering (ERM), ontwikkeld in de 60-er jaren door P. Chen. ERM beoogd de logische structuur van het UoD te modelleren. Deze modellen worden grafisch weergegeven in entitie-relatie diagrammen (ERD). Centrale begrippen in ERM zijn: entiteit, entiteit-type, attribuut, attribuut-type en relatie.

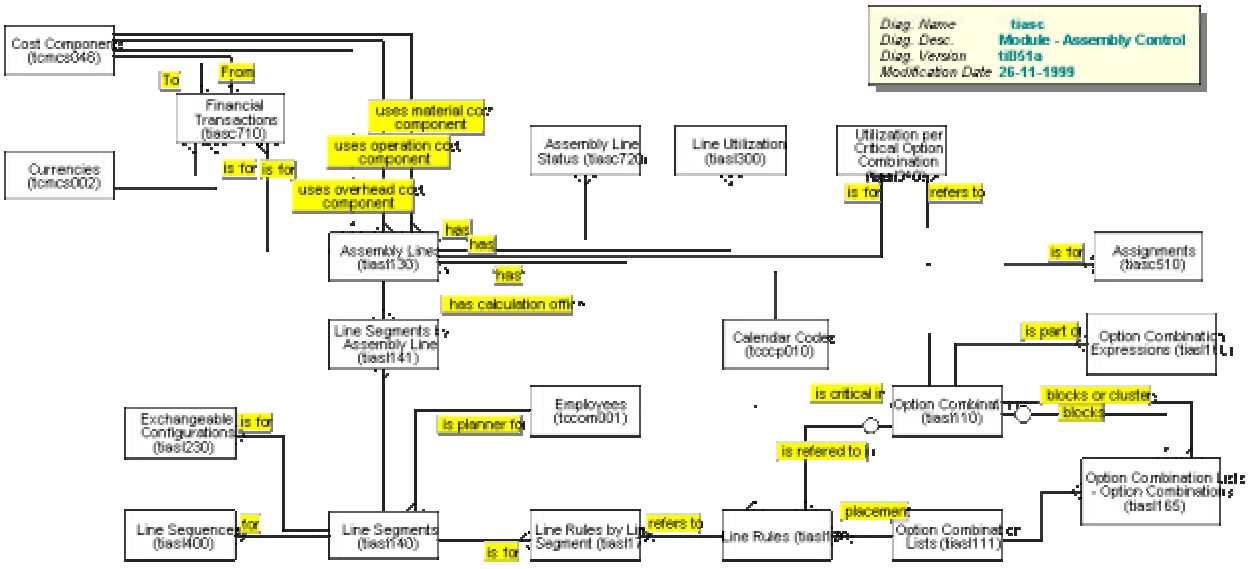
Een entiteit is een `ding' dat uniek kan worden geïdentificeerd. Dat kan om tastbare zaken gaan, zoals een exemplaar van een boek, maar ook om ontastbare zaken, zoals een afdeling van een bedrijf. Een entiteit wordt in een ERD veelal met een rechthoek aangeduid. Entiteiten hebben eigenschappen, en die worden attributen genoemd. Zo kunnen `Cees' en `57' attributen van entiteit `Van der Hoeven' zijn. Attributen worden opgesomd onder de naam van de entiteit waar ze bij horen. Entiteiten zijn met elkaar verbonden via relaties. De relatie `reserveert' bijvoorbeeld verbindt entiteiten van type 'boek' en `lid van de bibliotheek'. Een relatie is meestal binair, d.w.z. verbindt twee entiteiten. Tenslotte heeft zo'n relatie vaak een bepaalde beperkingen wat betreft cardinaliteit. Iemand kan bijvoorbeeld maximaal 6 boeken lenen, en een exemplaar van een boek kan aan hoogstens een persoon uitgeleend zijn. In figuur 3 is een eenvoudig entitie-relatie diagram weergegeven waarin de relaties `reserveren' en 'geschreven door' centraal staan.



Figuur 3: Entitie-relatie diagram

Huidige ERM-technieken hebben veel gemeen met technieken die gebruikt worden bij object-georiënteerde analyse-methoden. Omgekeerd, het bekende klasse-diagram uit object-georiënteerde technieken, en de bijbehorende representatie in bijvoorbeeld UML, bevatten veel elementen van ERM en ERD.

In de praktijk van systeemontwikkeling zijn de gegevensmodellen vaak groot en complex. Om een indruk te geven van deze complexiteit tonen we in figuur 4 een deel van het ERM van het Productie-automatiseringspakket van Baan, waarin kosten van werkplekken aan productielijnen bijgehouden worden.



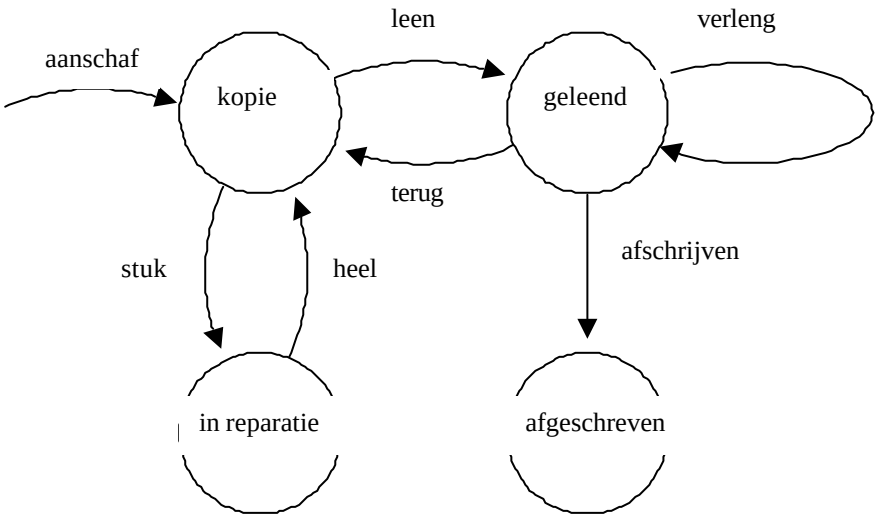
Figuur 4: Een ingewikkelder entitie-relatie diagram

4.2. Procesmodellering

Op elk moment in de tijd is onze bibliotheek in een of andere toestand. Die toestand wordt bepaald door dingen als:

- De verzameling beschikbare boeken
- De verzameling bestelde, maar nog niet ontvangen, boeken
- De verzameling bibliotheekmedewerkers
- De verzameling leden van de bibliotheek
- De verzameling uitgeleende boeken.

Elke actie, of het nu het terugbrengen van een boek betreft, of de inschrijving van een nieuw lid, of de bestelling van een boek, transformeert de huidige toestand *s* in een nieuwe toestand *s'*.



Figuur 5: Een toestands-overgang-diagram

Diverse specificatietechnieken modelleren een systeem in termen van toestanden en toestandsovergangen. Een eenvoudig formalisme voor het specificeren van toestanden en toestandsovergangen is de eindige automaat. Zoals de naam al aanduidt bestaat zo'n model uit een eindig aantal toestanden en overgangen daartussen. Grafisch wordt een eindige

automaat aangeduid in een toestands-overgang-diagram. Figuur 5 geeft daarvan een voorbeeld. In dit voorbeeld wordt slechts een klein stukje van onze bibliotheek gemodelleerd. Wanneer we een systeem in één groot toestandsdiagram modelleren, wordt dat een heel groot en onleesbaar iets. Een notatie genaamd `statecharts' geeft de mogelijkheid dit soort diagrammen hiërarchisch op te bouwen, zodat een verzameling toestanden op een hoger niveau als één toestand kan worden gezien. Statecharts vormen weer de basis voor het `state diagram' uit UML en andere object-geörienteerde modelleringsmethoden.

5. Verificatie en validatie

We moeten al tijdens de requirements engineering fase beginnen met het toetsen van de beslissingen die vastgelegd worden in de requirements specificatie. De requirements specificatie weerspiegelt het wederzijdse begrip dat gebruikers en ontwikkelorganisatie hebben van het probleem. Belangrijke vraag is dan of alles beschreven is, en of het juist beschreven is. Valideren van de requirements specificatie houdt dus in het controleren van de correctheid, volledigheid, juistheid van dit document. Betrokkenheid van de gebruikers, de probleemeigenaar, is hierbij onontbeerlijk. De meeste testtechnieken die in dit stadium worden gebruikt ijn informeel. De documenten en beschrijvingen die ons ter beschikking staan zijn immers ok veelal informeel: natuurlijke taal, plaatjes, e.d.

Naast het "testen" van de specificatie kunnen in dit stadium ook voorbereidingen worden getroffen voor latere testfasen, zoals het opstellen van een testplan, en het genereren van tests uit de specificatie. Als er voor onderdelen van de specificatie geen tests met een eensluidend criterium kunnen worden afgeleid, is dit een zeker teken dat de specificatie op dat punt niet goed is.

6. Quo vadis[Sjaak1]

Het is opvallend dat de meeste requirements engineering technieken die heden ten dage veel gebruikt worden, alsook `moderne' technieken als de Unified Modeling Language (UML), hun wortels in een ver verleden hebben, zoals de ERM techniek die stamt uit de jaren 60. Natuurlijk zijn er ook meer recente ontwikkelingen en gebieden waarop nog veel onderzoek gaande is. We volstaan hier met een korte opsomming:

- Niet-functionele eisen. We hebben ons in het voorgaande bijna uitsluitend gericht op het specificeren van functionele eisen. Niet-functionele eisen, zoals die betreffende betrouwbaarheid, security ed, zijn vaak even belangrijk, maar veel lastiger precies vast te leggen. Ook is het moeilijk vooraf te bepalen of een bepaalde combinatie van biet-functionele (en functionele) eisen een haalbare zaak is. Het is dan nodig tenminste een architectuurontwerp te maken, en dat ontwerp nader te analyseren op haalbaarheid.
- Conflicterende eisen. Verschillende belanghebbenden kunnen tegenstrijdige eisen hebben. In plaats van deze tegenstrijdigheden in een vroegtijdig stadium op te lossen en verder te werken met een consistente verzameling eisen, kan men tegenstrijdige eisen accepteren, ze allemaal opslaan, en nader analyseren om tot een adequate verzameling te realiseren eisen te komen.
- Use cases. Gebruikers kunnen vaak slecht hun eisen precies formuleren. Ze kunnen echter wel goede voorbeelden geven van gewenst gedrag van een systeem. Deze voorbeelden worden use-cases genoemd. Onderzoek is erop gericht technieken te vinden voor het vastleggen en analyseren van use-cases, en zelfs het afleiden van requiremenst uit een verzameling use-cases.
- Prioritering van eisen. Bij elk project is er een spanningveld tussen de beschikbare tijd en middelen tegenover de totale lijst met requirements. Technieken voor requirements prioritering helpen categorisaties te maken tussen absoluut noodzakelijk, belangrijk, gewenst, en niet-essentieel. In de productsoftware industrie wordt momenteel gewerkt aan waardetoekenning aan requirements, d.w.z. welke requirements voegen de meeste verkoop- en gebruikswaarde toe aan de volgende release.

7. Referenties

[Chen76] P.P. Chen, *The Entity-Relationship Model: Toward a Unifying View of Data,* ACM Transactions on Data Base Systems, **1** (1), 1976, pp 9-36.

- Het beroemde artikel waarin het Entity-Relationship Model wordt geïntroduceerd.

[Davis93] A.M. Davis, *Software Requirements: Objects, Functions and State*, Prentice-Hall, 1993.

- Geeft een vrij compleet overzicht van de `klassieke'requirements engineering technieken.

[Fowler99] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 1999.

- Beknopte, goed leesbare bespreking van UML.

[Harel88] D. Harel, *On Visual Formalisms*, Communications of the ACM, **31** (5), 1988, pp 514-530.

- Artikel waarin `statecharts' worden geïntroduceerd. Later in een of andere vorm in vele modelleringstechnieken opgenomen.

[Hirschheim89] R. Hirschheim, H.K. Klein, *Four Paradigms of Information Systems Development*, Communications of the ACM, **32** (10), 1989, pp 1199-1216.

- Heel goed artikel waarin de vier paradigma's uit sectie 2.2 worden geïntroduceerd.

[IEEE93] IEEE Recommended Practice for Software Requirements Specifications, IEEE Standard 830, 1993.

- Uitgebreide beschrijving van de standaard die in sectie 3 wordt beschreven.

[Kotonya97] G. Kotonya, I. Sommerville, *Requirements Engineering, Processes and Techniques*, John Wiley & Sons, 1997.

- Vrij praktisch boek, waarin veel technieken worden besproken.

[Loucopoulos95] P. Loucopoulos, V. Karakostas, *Systems Requirements Engineering*, McGraw-Hill, 1995.

- Niet al te dik boek, waarin de verschillende fasen (elicitatie, specificatie en validatie) goed belicht worden.

[Robertson99] S. Robertson, J. Robertson, *Mastering the Requirements Process*. ACM Press, Addison Wesley, 1999.

- Goed toegankelijk boek met overzicht van allerlei soorten requirements, geschreven door zeer ervaren RE consultants

[vanVliet00] H. van Vliet, *Software Engineering: Principles and Practice*, John Wiley & Sons, 2000.

- Heel aardig boek over software engineering. Dit artikel is grotendeels gebaseerd op hoofdstuk 9 uit dit boek.

Page: 12

[Sjaak1] Heeft het zin om deze hier achter te plaatsen? We zouden hier ook een korte beschrijving van hedendaagse onderzoeksspeerpunten op het gebied van RE kunnen opnemen. De samenvatting hoort aan het begin.