Requirements, informatica versus architectuur

Bachelorscriptie Daan Pijper 0512354

Abstract

De informatica en architectuur zijn twee verschillende vakgebieden. Toch zijn er overeenkomsten. Zowel de informatica en architectuur houden zich bezig met het ontwerpen van objecten die gebruikt gaan worden. En de mensen die betrokken zijn bij het gebruik van deze objecten hebben eisen aan dat gebruik. Binnen de informatica worden deze eisen requirements genoemd en heet het vakgebied dat zich bezig houdt met het bepalen van de requirements requirements engineering. Geïnspireerd door eerder onderzoek naar de fysieke architectuur door ICT architecten door o.a. Daan Rijsenbrij [12] word er in dit artikel onderzoek gedaan naar hoe er met requirements word omgegaan tijdens het ontwerp van architecten.

Binnen deze scriptie wordt een beeld geschept van requirements engineering binnen de informatica door in te gaan op algemene eigenschappen van requirements engineering en door een specifieke methode, volgend uit het boek *Use Cases, Requirements in Conrtext* in te gaan. Ook wordt behandeld hoe requirements engineering terug komt in de ontwikkelmethode *Agile Development*.

Om dit te kunnen vergelijken met de architectuur wordt er uitgelegd wat de basisdefinitie van architectuur is en hoe architecten bepalen wat zij moeten maken. Daarnaast wordt er beargumenteerd dat alle regelgeving betreffende het bouwplan eigenlijk de requirements van de overheid representeren. In de analyse worden de methode van de informatica vergeleken met hoe de architecten omgaan met de bouwwet en hoe zij bepalen wat ze gaan ontwerpen.

Er kan uiteindelijk op drie gebieden een conclusie worden getrokken. De architectuur kent vele wetten en regelgeving over wat gebouwd mag worden. De informatica niet. Maar dit is niet vanzelfsprekend en de informaticus kan veel leren van de architect over hoe om te gaan met zoveel wetten en regels. Ook kan de overheid kijken naar zijn wetten en regels betreffende de bouw als zij wetten en regels willen opstellen voor de ICT.

Architecten houden zich naast het nut en de degelijkheid zich ook expliciet bezig met het schoon en de belevenis van hun gebouw. Informatici niet. Dit artikel probeert duidelijk te maken dat het niet vanzelfsprekend is dat schoon en belevenis niet relevant zijn voor de informatica en dat de informatica zich niet expliciet bezig houd met schoon en belevenis. Verder onderzoek naar waarom hier geen aandacht aan word besteedt en, als dit niet terecht blijkt te zijn, hoe er dan wel expliciet rekening mee kan worden gehouden is nodig.

Architecten en informatici gebruiken verschillende methode om onderzoek te doen naar de eisen die aan hun ontwerp worden gesteld. Het is voor informatici aan te raden om te kijken naar welke methode architecten gebruiken en welke zij misschien ook kunnen gebruiken.

Inleiding

In dit artikel zal een vergelijking worden gemaakt tussen requirements engineering binnen de informatica en vergelijkbare methode binnen de fysieke architectuur. Het uiteindelijke eindresultaat van dit artikel zal dan ook een beeld geven van de verschillen en overeenkomsten binnen zowel de methode als het doel van requirements engineering binnen beide vakgebieden.

Voordat ik dieper zal ingaan op het waarom van deze vergelijking zal ik eerst even kort uiteenzette wat requirements in het algemeen zijn. Ik zal beginnen met een quote waarmee een boek voor een methode van requirements engineering begint met zijn omschrijving van requirements. "Requirements are the effects that the computer is to exert in the problem domain, by virtue of the computer's programming."[1] Deze specifieke quote slaat op requirement binnen een context van computers. Maar in principe is deze definitie toepasbaar op elk artefact. In plaats van ''computer'' en ''computer's programming'' praat je dan over ''artefact'' en ''artefact's nature''. Requirements gaan dus over de effecten die een artefact moet hebben op het probleemgebied. Een andere uitleg die in dit boek wordt gebruikt is "A requirement is something that a computer application must do for its users"[1] Binnen deze definitie gaat het over eisen aan de functionaliteit van een artefact.

Hoewel de definitie per methode enigszins varieert gaat het bij requirements in het algemeen over wat een artefact moet doen en dan vooral over de eisen die vooraf gesteld worden aan het doen van het artefact. En requirements engineering gaat dan over het verkrijgen van deze eisen bij alle verschillende stakeholders en het documenteren op zo een manier dat de requirements ook gewaarborgd kunnen worden bij het ontwerp en de ontwikkeling van het artefact. Het uiteindelijke doel van requirements engineering is daarmee zorgen dat er daadwerkelijk wordt gemaakt wat de stakeholders ook willen. En hoewel het simpel klinkt is het maken van goede requirements een enorme uitdaging.[1] Slechte requirements hebben veel software projecten doen laten falen.[1]

Het doel van dit artikel is door middel van deze vergelijking nieuwe inzichten te krijgen binnen requirements engineering. Hoewel requirements engineering als probleemgebied alleen binnen het vakgebied van de informatica voorkomt worden er aan alle artefacten eisen gesteld. En alle artefacten hebben stakeholders met ieder zijn eigen eisenpakket. De uitdaging om als ontwerper het juiste beeld van deze eisen te krijgen en de uitdaging om aan deze eisen te voldoen zijn dus universeel voor het ontwerp van elk artefact. Binnen de fysieke architectuur is men al millennia bezig met het ontwerpen van artefacten. Dit is het oudste vakgebied waarin artefacten worden ontwikkeld en daarmee mogelijk het meest ontwikkelde vakgebied. Daarom wil ik hun methode van omgaan met klanteisen bestuderen en vergelijken met die binnen de informatica. Met de resultaten van deze vergelijking hoop ik nieuwe inzichten te verwerven die requirements engineering dan wel nu of in de toekomst kunnen helpen om zichzelf verder te ontwikkelen.

Methode

Er is weinig voorgaand onderzoek gedaan dat twee (deel)vakgebieden die, hoewel ze ook veel overeenkomsten hebben, zoveel van elkaar verschillen met elkaar vergelijkt. Dit komt omdat er weinig gebieden zijn waarover van te voren kan worden gezegd dat ze op elkaar lijken. Daarom kan er van tevoren moeilijk een indeling worden gemaakt van welke deelaspecten precies met elkaar vergeleken worden en hoe deze deelaspecten zullen worden weergegeven. Omdat beide (deel)vakgebieden sterk verschillende achtergronden hebben is er ook een groot verschil in

terminologie wat het vinden van overeenkomende elementen moeilijk maakt. Er is dus weinig onderzoek waar ik mijn methode op kan baseren.

De vergelijkbare onderzoeken die ook sterk verschillende onderwerpen vergelijken variëren ook sterk van elkaar. De enige overeenkomst is dat alle onderzoeken proberen de onderwerpen te vergelijken is dat ze een zo volledig mogelijke uiteenzetting van de te vergelijke onderwerpen geven waarna op basis van deze uiteenzetting de overeenkomsten en verschillen worden geanalyseerd. Hoe de analyse er precies uit zal zien is meestal niet duidelijk totdat beide uiteenzettingen gemaakt zijn.

Voor dit artikel zal de methode vergelijkbaar zijn. Er zal voor zowel de requirements binnen de informatica als requirements binnen de fysieke architectuur een uiteenzetting worden gemaakt waarna later in een analyse in zal worden gegaan op de overeenkomsten en verschillen. De uiteenzetting over requirements binnen de informatica zal als eerst worden behandeld en bestaan uit de volgende onderdelen:

- Inleiding
- Doel en uitdagingen van requirements: Het doel van deze sectie is de lezer een beeld te geven van het waarom van requirements. Ook zullen de uitdagingen worden beschreven om aan te tonen waarom requirements achterhalen moeilijk is en zijn eigen (deel)vakgebied heeft. In dit deel van de uiteenzetting zal worden ingegaan op het doel wat requirements binnen de informatica hebben, de traditionele plaats van requirements in het software ontwikkelproces In en de universele uitdagingen bij het opstellen van requirements. Ieder deze onderwerpen heeft weer een eigen sectie.
- Requirements in Agile Development: Binnen deze sectie zullen requirements worden behandeld in de context van Agile Development. Agile Development is een recente ontwikkelmethode die sterk verschild van de meer traditionele ontwikkelmethode waarbinnen requirements engineering is ontstaan. Binnen agile development zelf worden requirements niet bij naam genoemd en word er zelfs enigszins afstand genomen van de term requirements. In deze sectie wordt aangetoond dat requirements toch nog een plaats hebben binnen agile development. Ze worden alleen niet bij naam genoemd omdat agile development afstand wil nemen van de meer traditionele ontwikkelmethode. Ook is de aanpak en rol van requirements anders. Door dit te behandelen wil ik de lezer laten zien dat requirements in sterk verschillende vormen kunnen terug komen.
- Use Cases Methode: In deze sectie wordt een uiteenzetting gegeven van de methode voor requirements engineering die bij Use Cases hoort. Deze methode wordt veel gebruikt in de praktijk. Door toch een specifieke methode volledig uit te werken word er een beter beeld gecreëerd van wat requirements engineering binnen de informatica nou precies inhoud en hoeveel moeite het kost.

Deze uiteenzetting is zeker niet volledig maar gezien de beperkte tijd en ruimte denk ik dat dit de beste uiteenzetting is. Het waarom van requirements word uitgelegd in combinatie met meer traditionele aanpak. Er wordt ingegaan op een minder conventionele context en plaats voor requirements zodat er een breder gebied word gepresenteerd waar later vergelijkingen met requirements binnen de fysieke architectuur kunnen worden gemaakt. Als laatste word er ingegaan op een concreet voorbeeld van een methode.

De uiteenzetting over de fysieke architectuur is van tevoren minder duidelijk gespecificeerd. Uit [6]volgt dat het Royal Institute of British Architects wel aandacht heeft voor requirements en ze ook noemt als belangrijk. Alleen voorgaand literatuuronderzoek heeft geen concrete methodes opgeleverd die worden behandeld. De onderverdeling die ik in het hoofdstuk zal hanteren is als volgt:

- Bouwwetten: In deze sectie worden de bouwwetten die de overheid oplegt aan architecten worden behandeld. Deze wetten betreffen voor een groot deel eisen of manieren waarop de overheid eisen kan stellen aan de architect. Aangezien de overheid een stakeholder is in alle gebouwen in Nederland omdat deze op Nederlands grondgebied worden gebouwd zijn deze Bijvoorbeeld wetten de requirements van de overheid. Er zijn verschillende vormen van bouwwetten en verschillende instanties met bepaalde verantwoordelijkheden. Daarom zal deze sectie nog worden onderverdeeld in de volgende deelhoofdstukken waarin de gevolgen van iedere wet of regelgeving voor requirements zal worden besproken:
    - Bestemmingsplan
    - Welstand
    - Buren
    - Bouwtechnische eisen
- Fysieke architectuur: In deze sectie zal ingegaan worden op verschillende definities van architectuur en hoe bij het ontwerpen door architecten requirements terug komen. Ook zullen aspecten van het ontwerp worden behandeld waar geen requirements in worden gebruikt. Het hoe en vooral het waarom zal hier worden behandeld.

De analyse van verschillen en overeenkomsten zal in het laatste hoofdstuk plaats vinden. Het is moeilijk om van te voren vast te leggen hoe dit hoofdstuk eruit zal zien. De voorafgaande literatuurstudies hebben maar een zeer beperkt beeld van requirements in de fysieke architectuur opgeleverd. Daarom kan er niet van tevoren een volledige indeling worden gemaakt. Wel is duidelijk dat er een sectie zal zijn die ingaat op het aanwezig zijn van bouwwetten binnen de fysieke architectuur. Verder zal de sectie “Doel en uitdagingen van requirements” als uitgangspunt dienen voor de analyse. De andere secties binnen het hoofdstuk over requirements binnen de informatica zullen gebruikt om meer specifieke voorbeelden van verschillen en overeenkomsten toe te lichten.

Requirements binnen de informatica

Inleiding

In dit hoofdstuk zal uitleg worden gegeven over requirements binnen de informatica. Eerst zal er worden ingegaan op het hoe en waarom van requirements en de uitdagingen voor requirements engineering. Opvolgend zal er kort worden ingegaan op de positie die requirements engineering traditioneel in software ontwikkelmethode inneemt. Verder zal de positie van requirements engineering binnen de software ontwikkelmethode agile development worden beschreven. Deze wijkt af van de traditionele positie van requirements binnen software ontwikkelmethode. Als laatste zal er uitgebreid worden ingegaan op de requirements methode van use cases aan de hand van het boek *Use Cases, Requirements in context*[1]. Dit is een veelgebruikte en redelijk recente methode van requirements engineering. Door hier dieper op in te gaan zal er een concreter beeld van requirements binnen de informatica worden gemaakt.

Doel en uitdagingen van requirements

Het doel van requirements

Requirements engineering is een vakgebied dat al lang onderdeel uit maakt van de informatica. Het is altijd een probleem geweest om software te maken die achteraf daadwerkelijk aansluit bij de wensen van de stakeholders. Daarom is er binnen de informatica veel aandacht voor het vinden van de goede methode om requirements naar boven te krijgen. Meestal beschrijven deze methodes structuren en manieren om de requirements te documenteren. Maar veel methodes in Requirements Engineering gaan ook over het verkrijgen van requirements bij de stakeholders. Denk hierbij aan workshops, groeps- of individuele interviews of literatuurstudies. Omdat hier binnen de sociale wetenschappen al veel onderzoek naar gedaan is word hier wel minder aandacht aan besteed binnen de methodes.

Het documenteren van de requirements word niet alleen gebruikt om ervoor te zorgen dat het op te leveren systeem als goed zal worden ontvangen. In veel gevallen komen de requirements ook in een of andere vorm terug in het contract. De requirements worden dan gebruikt om te bepalen wat het systeem bij oplevering allemaal moet kunnen.

Omdat de belangrijkste informatiebronnen voor requirements de toekomstige gebruikers of andere stakeholders zijn is het belangrijk dat de requirements enigszins begrijpbaar zijn voor de gebruikers of begrijpbaar te maken zijn zodat de gebruikers tijdens het opstellen van de requirements ook feedback kunnen geven op tussenresultaten.[1] En als requirements in het contract komen moeten ze begrijpbaar zijn voor de opdrachtgever die akkoord moet gaan met het contract. Het begrijpbaar houden of maken voor de gebruikers is dan ook een belangrijk punt binnen de methodes.

Requirements engineering in software ontwikkelmethode

Requirements engineering werd geïntroduceerd in de softwareontwikkeling toen de meest gebruikelijke software ontwikkelmethode het watervalmodel was. Het watervalmodel is een traditionele software ontwikkelmethode waarin het ontwikkeltraject een vaste sequentie volgt. Er wordt bijvoorbeeld eerst een requirements document opgesteld, dan een functioneel ontwerp, dan een technisch ontwerp, dan de implementatie en als laatste volgt het testen. Ieder van deze fases word eerst afgerond voordat aan de andere word begonnen. Zoals in het voorbeeld al duidelijk word is requirements engineering vaak de eerste stap. Het eerste wat er gedaan word nadat de opdrachtgever zijn opdracht heeft gegeven is inventariseren wie de belanghebbende zijn en beginnen met requirements engineering. Als het requirements document eenmaal af is en ondertekend door de opdrachtgever staan de requirements vast en kan er begonnen worden met het functionele ontwerp. Als verder in het ontwikkelproject toch nog requirements naar boven komen is het te laat en kunnen deze niet meer in het requirements document worden opgenomen. Deze methode werd lang gebruikt en het voordeel is dat de ontwikkelaars een volledige, ondertekende lijst van requirements hebben waar ze de rest van het ontwikkeltraject op kunnen vertrouwen. Maar in werkelijkheid is het haast niet te voorkomen dat er later requirements veranderen of bijkomen. Ook weet je zeker dat je alle requirements naar boven heb gehaald, de situatie kan altijd veranderen en mensen kunnen van mening veranderen.[2] Hoewel ik verder in dit hoofdstuk zal aantonen dat dit geen probleem hoeft te zijn voor requirements engineering is het wel belangrijk om te realiseren dat requirements begonnen zijn in deze ontwikkelcontext.

Universele uitdagingen voor requirements engineering

Er zijn vele uitdagingen binnen requirements engineering. *Use Cases, Requirements in context*[1] komt met een uitgebreide lijst van uitdagingen binnen requirements engineering gebaseerd op eigen ervaring van de auteur en andere literatuur. Hoewel deze lijst uit literatuur komt voor een specifieke methode geld hij voor requirements engineering in het algemeen. Hij wordt in de literatuur gepresenteerd als motivatie voor de te introduceren methode.

De eerste uitdaging is uitzoeken wat de gebruikers nodig hebben. Het standaardantwoord op deze vraag is: Vraag het aan de gebruiker. Helaas is de oplossing niet zo eenvoudig. Gebruikers kunnen deze vraag namelijk helemaal niet beantwoorden. Het antwoord is veel complexer dan het in eerste instantie lijkt. Vaak gaat de impact van een softwaresysteem veel verder dan alleen ondersteunen van de huidige werkmethode. De functie van de werknemer zal altijd wel iets veranderen door het softwaresysteem en hetzelfde geld voor de gehele bedrijfsvoering. Dus de gebruiker moet al begrijpen hoe zijn werk gaat veranderen voordat hij deze vraag kan beantwoorden. Daarbij moet de gebruiker ook nog verder met zijn huidige werk. Hij krijgt er nu een extra taak bij. Het informeren van de requirements engineer. En daar zit de gebruiker vaak niet op te wachten. En er zijn nog veel meer problemen. Steve McConnel maakte in zijn boek *Rapid Development* (McConnel 1996) een lijstje waarop gebruikers requirements engineering moeilijk kunnen maken.

- Gebruikers begrijpen niet wat ze willen.
- Gebruikers willen niet toezeggen aan geschreven requirements.
- Gebruikers eisen nieuwe requirements als de kosten en uitvoering al vast staan.
- Communicatie met de gebruikers is langzaam.
- Gebruikers nemen vaak niet deel aan de recensie of kunnen niet deel nemen.
- Gebruikers zijn technisch niet bekwaam.
- Gebruikers begrijpen het software ontwikkelproces niet.

Deze lijst geeft het idee dat gebruikers alleen maar lastig zijn. Dit is niet waar. Alleen gebruikers kunnen niet al hun tijd besteden aan het beantwoorden van vragen. En je kan de gebruikers ook niet precies uitleggen hoe software en softwareontwikkeling werkt. Dat willen ze niet en daar hebben ze geen tijd voor. Daarom is het een uitdaging om de gebruikers zo ver te krijgen dat ze voldoende mee willen en kunnen werken aan het requirements engineering proces.[1]

De tweede uitdaging is het documenteren van de gebruikers behoeften. De uitdaging hier is het verifiëren van de requirements die opgeschreven zijn. Het is moeilijk om te bepalen of de requirements volledig zijn en of er geen conflicten zijn.[1] Zeker bij grote softwareprojecten. Dit werd ooit nog moeilijker gemaakt door het gebrek aan goede methode om dit te documenteren. Tegenwoordig zijn er onder andere use cases die hierbij helpen.

Het derde probleem is dat er te vroeg ontwerpbeslissingen worden genomen. Het is niet de bedoeling, dat er binnen de requirements al over het hoe wordt gepraat. Maar helaas is het voor mensen heel moeilijk om over wat na te denken zonder aan hoe te denken. Mensen zijn gewend om in oplossingen te denken. Dit probleem wordt nog groter wanneer het ontwerpers zijn die de requirements opstellen. Zij zijn al helemaal gewend in oplossingen te denken. Wat ook een probleem kan worden is wanneer de requirements engineers weinig vertrouwen hebben in de ontwerpers. De requirements engineers zullen dan al ontwerpbeslissingen nemen om te voorkomen dat de

ontwerpers het fout doen.[1] Vertrouwen en bekendheid tussen de ontwerpers en de requirements engineers zijn dus ook belangrijk.

Het vierde probleem is het oplossen van conflicterende requirements. Omdat je vaak veel verschillende belanghebbende hebt is het onmogelijk om te voorkomen dat er requirements zullen komen die met elkaar conflicteren. Zeker wanneer je de klassieke methode voor requirements gebruikt, namelijk een lijst, kan dit probleem groot zijn. Er moet binnen de methode en documentatie een manier zijn om conflicten op te lossen zodat uiteindelijk het aantal conflicterende requirement zo klein mogelijk is.[1]

Het vijfde probleem zijn overbodige requirements. Zeker als er verschillende requirements engineers zijn maar ook als er gewoon veel requirements zijn kunnen er uiteindelijk requirements komen die al behandelt zijn binnen andere requirements. Hoewel dit in eerste instantie niet een groot probleem hoeft te zijn kan dit later tot verwarring leidden.[1] Het lijkt dan alsof er nog iets mist in de software terwijl het eigenlijk via een eerdere requirements al behandeld zijn. Je loopt dan het risico dat je requirements zo gaat interpreteren dat er toch nog iets nieuws wordt toegevoegd aan de software wat niet nodig of zelfs schadelijk is.

Het zesde probleem is omgaan met volume. Er kunnen heel veel requirements zijn. Dit is op zich niet erg omdat je dan zeker weet dat je alles hebt. Maar achteraf moet je altijd controleren op redundantie, conflicten en ontwerpbeslissingen. Ook is het aan te raden om te kijken of vergelijkbare requirements naar een hoger abstractie niveau kunnen worden gebracht. Sommige methodes, zoals use cases, staan ook deel-requirements toe wat helpt bij de overzichtelijkheid.

Als laatste is er het waarborgen van traceerbaarheid. De requirements moeten begrijpbaar zijn voor de ontwikkelaars door het hele ontwikkeltraject van het softwaresysteem. Men moet kunnen achterhalen waar requirements vandaan komen om ze volledig te kunnen begrijpen en om te controleren of ze daadwerkelijk ook geïmplementeerd is. Hoewel dit eigenlijk niet te doen is binnen de huidige softwareprojecten zou het in een ideale situatie een goed controlespoor binnen het ontwikkelproces voorzien. Als dit niet het geval is dan loop je het risico dat verder in het ontwikkelproces andere ontwikkelaars functionaliteit gaan toevoegen die niet nodig of zelfs niet gewenst is. Gezien het uitblijven van de ideale situatie is het nu vooral de taak van het management om te controleren of men zich aan de requirements houd. [1]

Requirements in Agile Development

Agile software development is een relatief nieuwe en populaire ontwikkelmethode binnen de software ontwikkeling. Er zijn verschillende methodes ontwikkelt die allemaal als doel hebben software sneller op te leveren en te zorgen dat de software voldoet aan de voordurend veranderende klanteisen. De agile methodes proberen dit te bereiken door een hechte samenwerking tussen de ontwikkelaars en mensen uit de business en door vaak werkende software op te leveren.[2]

Requirements engineering komt juist uit de traditionele hoek waar het ontwikkelproces sterk gefaseerd is. Eest worden de requirements opgesteld, daarna een technisch ontwerp, het daadwerkelijk implementeren en uiteindelijk testen. Omdat requirements zwaar leunen op documentatie en agile development meer op persoonlijke samenwerking en adaptief ontwerpen word soms gesteld dat agile development en requirements niet samengaan.[2] Toch zal blijken dat

requirements wel degelijk een plaats hebben binnen agile development.[2] Eigenlijk komen requirements en veel methodes uit de requirements engineering weer terug binnen agile development.

Vergeleken met de traditionele software ontwikkelmethoden lijkt agile development meer code georiënteerd en minder documentatie georiënteerd. Binnen de meer traditionele methode wordt er vaak naast de code ook veel documentatie opgeleverd zoals een requirements document. Binnen agile development word er aanzienlijk minder documentatie opgeleverd. Maar het verschil in hoeveelheid documentatie tussen de twee methodes is eigenlijk een symptoom van twee diepgaandere verschil. Deze zijn:

- Agile methodes zijn adaptief in plaats van voorspellend. Binnen de traditionele methodes word het ontwikkelproces van tevoren geplanned. Agile methodes zijn ontwikkeld met het doel om te kunnen gaan met frequente veranderingen. Iets waar traditionele methodes vaak moeite mee hebben.[3]
- Agile methodes zijn mens georiënteerd in plaats van proces georiënteerd. Agile methodes vertrouwen op expertise, competenties en samenwerking in plaats van rigoureuze, document gecentreerde processen om kwaliteitssoftware op te leveren.[3]

Er zijn verschillende methode van agile development. Het artikel *Requirements Engineering and Agile Software Development*[2] heeft voor 5 methodes, Extreme Programming (XP), Agile Modeling (AM), Scrum, the Crystal Methodologies, Feautere Driven Development (FDD), Dynamic System Development Method (DSDM) en Adaptive Software Development (ASD) onderzocht of en hoe requirements terugkomen binnen deze methodes. Binnen de methode Scrum wordt er bijvoorbeeld een zogenaamde Product Backlog bijgehouden. Dit is een geprioriteerde lijst waar alle functies, aanpassingen, bugs en kenmerken worden bijgehouden. De functies en kenmerken komen voort uit de requirements die noodzakelijk of nuttig zijn voor het programma. Scrum heeft dan iedere dertig dagen een sprint waarin de prioriteiten voor die dertig dagen uit de backlog worden gehaald. [4;5] Hoewel requirements niet los worden benoemd en gedocumenteerd komen ze binnen deze methode wel degelijk terug. En hetzelfde geld voor de andere methodes.

Het artikel *Requirements Engineering and Agile Software Development*[2] stelt een aantal requirements engineering technieken voor bij de agile aanpak. Deze bestaat uit een analyse van mogelijke synergies tussen requirements engineering en agile development.

- **Klantenbetrekking**: Agile methodes gaan vaak uit van een ideale klant. De klant weet precies wat hij wil en kan ook uitdrukken wat hij wil. Ook is de klant instaat om bindende en correcte beslissingen te maken. Zelfs binnen de agile methodes die groeps elicitations kennen is het niet gegarandeerd dat alle achtergronden en belangen aanwezig zijn. Binnen de agile methodes komen de requirements dus wel degelijk naar boven. Er word alleen niet meteen gegarandeerd dat alle requirements naar boven komen. Daar staat tegenover dat binnen agile methodes klantbetrekking door het hele ontwikkelproces blijft plaatsvinden. Methodes uit de requirements engineering zouden wel kunnen helpen om te garanderen dat alle belanghebbende ook betrokken worden bij het ontwikkelproces.
- **Interviews**: Alle agile methodes stellen dat de beste manier om informatie van klanten te krijgen praten is. De klantinteractie die plaats vindt binnen requirements engineering is ook

vaak in de vorm van een interview. Agile Development en Requirements engineering zien allebij praten met de gebruikers en klant als belangrijke bron voor informatie.

- **Prioriteiten**: Binnen agile development worden altijd prioriteiten gesteld. Meestal worden hoogste prioriteiten eerst uitgevoerd waarna nieuwe inzichten vanzelf zouden moeten komen. De prioriteiten worden regelmatig bijgewerkt door het gehele ontwikkelproces. Binnen de context van requirements zou dit betekenen dat door eerdere requirements te implementeren nieuwe requirements naar boven zouden moeten komen. Hoewel dit tegen het gefaseerde ontwikkelen dat waarmee traditioneel requirements engineering word geassocieerd ingaat word in *Use Cases, requirements in context*[1] al gesteld dat requirement eigenlijk nooit af zijn. Binnen agile development is dit ook het geval.
- **Modeleren**: Hoewel modellen ook worden gebruikt binnen agile development worden ze voor een ander doel gebruikt dan binnen requirements engineering. Binnen agile development worden modellen meestal gebruikt om een klein deel van het systeem uit te leggen gedurende de ontwikkeling. Deze modellen worden alleen gebruikt tijdens de uitleg en hebben een beperkte levensduur. Binnen vele methodes in requirements engineering worden modellen gebruikt als een manier van documentatie. Op verschillende abstractieniveaus worden delen van het systeem gemodelleerd en gedocumenteerd. Hoeveel en wat gemodelleerd word varieert per methode. FDD, een agile methode waarin in korte cycli het ontwerp en de implementatie van het systeem worden uitgevoerd gebruikt modelleren wel op een manier vergelijkbaar met requirements engineering. [2]
- **Documentatie**: Het maken van een compleet en consistent requirements document word binnen agile development gezien als onhaalbaar of op zijn minst niet kosten effectief. Sommige agile methode zoals Scrum, Crystal of DSDM adviseren het gebruik van een requirements document maar het wordt aan het ontwikkelteam overgelaten hoe uitgebreid en gedetailleerd dit document word. Binnen agile development word de keuze gemaakt documentatie minimaal te houden en zoveel mogelijk informatie verbaal uit te wisselen. Dit levert op de lange termijn problemen op bij bijvoorbeeld het informeren van nieuwe projectleden of het achteraf evalueren van het ontwikkelproces. Maar het maakt het ontwikkelproces wel sneller. Omdat requirements engineering methode vaak uitgebreid documenteren passen de traditionele methode vaak niet goed in agile development. Binnen agile development volgt de implementatie vaak veel sneller na het ontwerp waardoor er minder noodzaak is om de requirements uitgebreid te documenteren.
- **Validatie**: Er worden regelmatig acceptatietesten en review sessies gehouden binnen agile methodes om requirements te valideren. Deze tests moeten dan aantonen of het project de juiste richting ingaat en of het project op schema ligt. Door steeds deelsystemen of prototypes te maken en door regelmatig te testen of deze de juiste richting opgaan probeert agile development te komen tot het juiste eindproduct. Traditioneel werd dit gedaan door eerst een uitgebreid ontwerp te maken die direct in geheel werd geïmplementeerd. Hierdoor word de vorm van requirements engineering natuurlijk anders dan in de traditionele methodes. Maar requirements blijven nog steeds nodig om tot de deelsystemen of prototypes te komen.
- **Management**: Traditioneel wordt de uitgebreide documentatie van requirements gebruikt om te kijken waarom veranderingen gemaakt werden. Binnen agile development is dit niet mogelijk omdat de documentatie ontbreekt. Dit is ook de reden waarom projecten gemaakt

met agile development gebaseerd zijn op vaste koste en een beperkte tijd in plaats van beperkte koste en een vast doel.[2]

- **Non-functional requirements**: Hoe om te gaan met non-functional requirement is binnen agile methoden vaak slecht gedefinieerd. Gebruikers en klanten die gevraagd word wat zij willen dat het systeem doet denken vaak niet na over resources, onderhoud, veiligheid prestaties of overdraagbaarheid. De ontwikkelaars moeten bedenken welke non-fucntionals gewenst zijn gezien de input van klant en gebruikers. Het artikel *Requirements Engineering and Agile Software Development*[2] beargumenteerd dan ook dat dit een zwakte van agile development is.

Requirements spelen ook binnen agile development een belangrijke rol. Omdat agile development probeert in samenwerking met klant en gebruikers in snelle iteraties een systeem te maken vertrouwt agile development meer op direct contact tijdens de ontwikkeling om de requirements naar boven te krijgen dan dat ze van tevoren worden achterhaald en gedocumenteerd. Requirements zijn binnen agile development hierdoor meer informeel en zijn minder expliciet aanwezig. Toch zijn ze wel degelijk aanwezig en de behoefte aan goed klantcontact binnen agile development geeft aan dat je goed moet praten met de klant en gebruiker om tot het gewenste systeem te komen.

Use Cases Methode

Use Cases zijn de eerste echte tool die de specificatie en communicatie problematiek aankaart binnen requirements engineering.[1] Voor use cases was requirements engineering vaak het opstellen van een lijstje requirements en ontbrak een echte voorgeschreven methode om ze op te schrijven. Het resultaat was vaak een lange lijst aan requirements in de vorm van een paar zinnen. Met use cases wordt er geprobeerd een meer gestructureerde en leesbare documentatie van requirements te maken. Maar use cases gaat niet alleen over de vorm van requirements documentatie maar ook over hoe je deze requirements moet verzamelen en hoe een requirement op zich er uit moet zien.

Requirements volgens Use Cases, Requirements in Context [1]

Requirements beschrijven wat een computer systeem moet doen voor zijn gebruikers. Een requirements bepaald wat een computer systeem kunnen moet doen om zijn bestaan te rechtvaardigen. Het gaat bij requirements heel duidelijk om het wat en niet om het hoe. Ook is het belangrijk dat het waarom duidelijk wordt in de requirements. Dit omdat er uiteindelijk een technische oplossing uit de requirement moet komen. Door ook duidelijk te maken waarom wat moet kunnen is het makkelijker om de juiste technische oplossing te bedenken waarmee aan dit wat wordt voldaan. Onder zullen een aantal voorbeelden van foute requirements behandeld worden om een beeld te geven wat deze beperking precies inhoud. Deze voorbeelden komen uit '*Use Cases Requirements in Context'*.[1] Ondanks het feit dat dit uit een methode voor use cases komt is het, ook binnen de methode, het idee dat dit ongeacht de vorm voor alle requirements geld.

"Het systeem moet over de capaciteit beschikken om alle klant transacties binnen een fiscaal jaar te bewaren"

Het probleem met deze requirement is dat hij te vaag is. Het is niet duidelijk wat een fiscaal jaar is, alleen dat het een impact heeft op het systeem. Het is ook niet duidelijk wat klant transacties precies

zijn. Het wordt alleen duidelijk dat er in data in het systeem moet komen, maar wat voor data wordt niet duidelijk. Een requirement moet dus een mate van duidelijkheid en specifiekheid hebben.

"Het systeem zal beperkte remote inquiry access (door middel van een dail-in) om plaatjes en data onafhankelijk in te kunnen zien voorzien."

Deze requirement is weer te vaag. Er wordt niet uitgelegd hoe beperkt de verbinding moet zijn. Ook wordt niet duidelijk hoe remote de verbinding moet zijn. Impliceert remote overal op de wereld via een mobiele verbinding of over in het kantoorgebouw en de directe omgeving.

"Het systeem zal documenten automatisch voor dat ze gedistribueerd worden voorzien van een barcode. Minimaal zullen de barcodes gebruikt worden om te identificeren bij welke opdrachten rij precies dit document hoort."

Deze requirement maakt een aantal technische aannames over hoe het moet worden opgelost. Barcodes zijn een oplossing voor het probleem, geen requirement. Het systeem moet documenten individueel kunnen herkennen maar dit kan met meer dan alleen barcodes. Requirements moeten over het wat gaan, niet over het hoe. Deze requirement bevat technische details en beperkt daardoor de mogelijkheden.

"Het systeem moet een intekening maken in het journaal bestand iedere keer als er een letter is gemaakt."

Deze requirement is meer georiënteerd op "wat te doen?" in plaats van "om dit te bereiken?". Het legt niet uit waarom waardoor het niet mogelijk is om alternatieven technische implementaties te bedenken. Wat men eigenlijk wil met deze requirement is de mogelijkheid om bij te houden welke letters precies gemaakt zijn. Hoe kan later bij het ontwerp worden besloten.

Dus het is belangrijk dat requirements alleen over het wat gaan en ook iets van waarom bevatten. Het is een zonde als requirements de oplossing al in een technische richting sturen. Toch moeten requirements ook weer specifiek genoeg zijn zodat de technische ontwerper alleen over de realisatie hoeft na te denken. Wat er precies moet worden gerealiseerd moet al volledig duidelijk zijn binnen de requirements. [1]

Een tweede eigenschap van requirements die in het algemeen zou moeten gelden is het onderscheid tussen functionele requirements en niet functionele requirements.
Functionele requirements gaan over wat gebruikers nodig hebben om met het systeem te werken. Functionele requirements beschrijven functies en mogelijkheden. Dit zijn over het algemeen de requirements waar men in eerste instantie aan denkt.[1] Denk hierbij aan requirements zoals:

- Het systeem zal orders van producten accepteren en een notificatie voorzien aan de magazijnbeheerder om te bepalen of er voldoende inventaris is om aan de order te voldoen.
- Het systeem zal onderdelen die niet meer in voorraad zijn vervangen door vergelijkbare onderdelen zoals gespecificeerd door de inventarismanager.[1]

Use Cases zijn vooral bedoeld om functionele requirements te documenteren.

Niet functionele requirements gaan over de verborgen onderdelen van het systeem die, ondanks wat de gebruikers denken, belangrijk zijn voor het systeem. Deze requirements gaan niet over de

functionaliteiten van het systeem. Niet functionele requirements gaan over de universele, vage factoren of principes binnen het systeem die belangrijk zijn voor het uiteindelijke succes. Het gaat vaak over termen die eindigen op *–heid* zoals snelheid, veiligheid of schaalbaarheid. Schaalbaarheid gaat bijvoorbeeld over de eis dat een systeem bij een veel grotere werklast niet aanzienlijk langzamer is dan bij een kleinere werklast.

Use Cases

Use Cases zijn onderdeel van UML, de Unified Modeling Language. UML is een modeleertaal voor systeemspecificaties. Deze bestaat uit verschillende onderdelen waarvan use cases er een is. UML wordt momenteel door OMG (Object Modeling Group) gehanteerd als industriestandaard. Dus use cases zijn momenteel onderdeel van de industriestandaard gegeven door OMG. Use Cases zijn binnen UML de methode om requirements te documenteren. Uiteindelijk bestaat het geheel van use cases uit een use case diagram, die de relatie tussen use cases aangeeft, scenario's, die een concrete invulling geven aan use cases, en de use cases zelf. Maar voordat ik in ga op deze drie onderdelen zal ik eerst de specifieke doelen van use cases behandelen.

Er zijn vier doelen die use cases proberen te halen.

- Use Cases zijn bedoeld om interacties tussen het systeem en entiteiten buiten het systeem weer te geven. De interacties moeten waarden hebben voor de externe entiteit binnen de reactie of voor een externe entiteit buiten de interactie. In het laatste geval moet je denken aan een motivatie rubriek binnen een formulier voor financiële transacties. Dit is niet interessant voor de entiteit die deze transacties in het systeem moet invoeren dus niet relevant voor de entiteit die deelneemt in de interactie met het systeem. Maar een manager zou via een andere interactie achteraf kunnen contoleren of hij deze transactie wel terecht vind. Maar het eerste doel van use cases is dat de interacties beschreven relevant zijn voor een entiteit buiten het systeem.[1]
- Het tweede doel van use cases is dat het geen specifieke implementatietaal heeft. Het is niet de bedoeling dat use cases al bepalen hoe een requirement geïmplementeerd moet worden binnen het systeem. Requirements moeten alleen beschrijven wat er geïmplementeerd moet worden. Hierdoor wordt gewaarborgd dat requirements geen specifieke implementatietaal afdwingen. Er is wel een kanttekening. De levenscyclus van use cases eindigt niet na de requirementsfase. In latere ontwikkelingsfase kunnen use cases ook bij de ontwikkeling helpen. Hoe precies valt buiten het doel van dit artikel. En dan kunnen er wel implementatie eisen in staan.[1]
- Use Cases moeten op zijn minst voor een deel begrijpbaar zijn voor de gebruikersgroep. Probeer zo veel mogelijk worden te gebruiken die de gebruikers ook kunnen begrijpen. Dit is niet altijd mogelijk voor de meer gedetailleerde use cases. Zorg dan ook dat er verschillende niveaus van use cases zijn waarbij de algemenere niveaus begrijpbaar zijn voor de gebruiker. Dan kunnen de gedetailleerde niveaus wat technischer zijn.
- Het laatste doel is een volume begrijpbaar voor de gebruiker. Het maximaal aantal gewenste use cases varieert tussen de 70 a 80 tot maar 28 afhankelijk van welke expert je het vraagt. Het is alleen wel belangrijk de hoeveelheid use cases zo klein mogelijk te houden. [1]Kijk achteraf altijd of use cases samengevoegd of geschrapt kunnen worden. Want met teveel

use cases wordt het onoverzichtelijk. En de gebruiker moet het ook kunnen begrijpen want hij moet er akkoord mee gaan.

Use Cases kunnen zelf niet iets doen. Dus voordat we iets aan use cases hebben moet er iets zijn wat de use cases activeert. Dit zijn actors. Een actor is iets buiten de computer applicatie wat de use case activeert. Dit kan een persoon zijn, een andere computer of iets meer abstracts zoals een specifieke datum en tijd. Binnen een use case diagram moeten de actors zichtbaar zijn. Maar alleen de actors die direct het systeem beïnvloeden of direct door het systeem worden beïnvloed. Het moeilijke met actors is bepalen welke te tonen in het use case diagram. [1] Als bijvoorbeeld een leverancier aan een administratief medewerker een factuur geeft waarna de administratieve medewerker weer de factuur invoert in het systeem, moet je dan alleen de administratieve medewerker tonen als actor of ook de leverancier. In dit geval is de leverancier een secondary actor. Binnen [1] wordt aangehouden dat je deze alleen moet meenemen als ze daadwerkelijk invloed uitvoeren op het systeem.
Ook andere computers kunnen actors zijn. Stel dat jou computer een fabrieksrobot aanstuurt dan is de fabrieksrobot waar de computer een dienst aanbied. Andersom kan de fabrieksrobot ook als actor de computer beïnvloeden door bijvoorbeeld informatie terug te koppelen die de computer gebruikt om het gehele fabricageproces aan te sturen.[1]
Een derde mogelijkheid is een externe impuls zoals tijd of omgevingsfactoren. De temperatuur kan de actor zijn voor een thermostaat die onder zoveel graden aanslaat.
Het is belangrijk om op te merken dat actors buiten het systeem vallen. Daarom is het verstandig om actors als aannames te zien. Ze maken interactie met het systeem maar ze worden er niet door geautomatiseerd. Actors kunnen een grote invloed hebben op de use cases. Zij vormen vaak een beginpunt voor het use case diagram of een model want bij hun begint het werkproces. Daarom is het erg belangrijk op de juiste actors bij de juiste use cases te plaatsen.
Actors worden uiteindelijk met pijltjes aan de bijbehorende use cases verbonden. Bij een dergelijke verbindingspijl komt ook een label te staan. Dit is de rol van de actor. Denk hierbij aan een label zoals administratief medewerker of human resource manager. Rollen zijn er dus om aan te geven welke rol de actor binnen die use case uitvoert. Rollen zijn niet een op een gekoppeld aan personen. Binnen een use case kan de administratieve medewerker een rol spelen als medewerker(denk aan een prikklok) terwijl bij een andere use case hij de rol financieel administratief medewerker speelt.

Use case diagrams kennen ook associaties. Deze kunnen tussen use case en actor, use cases of actors lopen. Er zijn drie soorten associaties. Een generalisatie, een extend en een include. [1]

- Generalisatie is een concept geleend van de object georiënteerde wereld. Wanneer een aantal use cases samen iets gemeen hebben waardoor ze op een hoger niveau als een use case voorkomen, dan zijn ze gegeneraliseerd en is het hogere niveau use case de generalisatie. Audi en BMW kunnen dan gegeneraliseerd worden naar automerken. Een ander voorbeeld van een generalisatie is de medewerker binnen het voorbeeld dat ik iets eerder heb aangehaald. Je kunt dan meteen zien hoe generalisaties nuttig zijn. Door de generalisatie medewerker [1]
- Extend wordt gebruikt om aan te geven wanneer een speciale use case voortkomt uit een andere use case. Dit wordt gebruikt om om te gaan met bijzondere omstandigheden en uitzonderingen. Een voorbeeld van een extend is noteer herhalende klantafspraak ten opzichte van noteer klantafspraak. Door een extend te gebruiken vermijd je dat noteer

klantafspraak ook rekening moet houden met de mogelijkheid dat het een herhalende klantafspraak is.[1]

- De include associatie geeft de mogelijkheid om het kopiëren van stappen binnen use cases te voorkomen. Als je meerdere use cases hebt waar voor een deel hetzelfde gebeurt kun je include gebruiken om te voorkomen dat je steeds hetzelfde herhaalt in de use cases.[1]

Hoewel extend en include in eerste instantie nuttig lijken is het aan te raden ze niet te gebruiken. Je kunt use cases ook vaak prima maken zonder en het maakt de associaties alleen maar ingewikkelder. Het gebruik vergroot het risico op fouten. Alleen wanneer je use cases van een al bestaand systeem moet aanpassen of uitbreiden wanneer het systeem zelf aangepast wordt kan het gebruik van extend en include aangeraden worden.[1]

Use Case template

Hoewel de UML waarop use cases gebaseerd zijn en use case diagrams bestaan uit diagrammen bestaan use cases uit tekst. Om use cases toch nog een standaard te geven is er het use case template. Een specifieke use case wordt uitgewerkt binnen deze template. Het is heel goed mogelijk om een eigen template te maken. Zolang de template maar binnen het project functioneert als standaard. Er zijn ook al bestaande templates zoals het RUP template uit *Use Cases Requirements in Context*[1]. Hieronder zal ik deze template weergeven en de verschillende secties behandelen.

Use Case Name:	
Iteration:	
Summary:	
Basic Course of Events	
Alternative Paths:	
Exception Paths:	
Extension Point:	
Triggers:	
Assumptions:	
Preconditions:	
Postconditions:	
Related Business Rules	
Author:	
Date:	

- De Use Case Name geeft de naam van de use case weer. Dit is de identificatie van de use case en daarom moet de naam uniek zijn. Het heeft de voorkeur om namen te gebruiken. Nummers kunnen afschrikwekkend werken voor gebruikers.
- De iteratie geeft aan in welke van de drie fase de use case is. De mogelijke fase zijn Facade, Filled en Foces. Deze iteraties zullen verder in het hoofdstuk nog worden behandeld.
- De summary geeft in een paar zinnen weer welke interactie er plaats vind binnen de use case. De summary mag context bevatten die andere secties niet bevatten.
- De basic course of events is het belangrijkste deel van de use case. Hierin worden de stappen beschreven die de actor en het systeem moeten maken om de doel van de use case te behalen. De actor begint met de eerste stap waarop het systeem weer reageert. Dit gaat door tot het doel van de use case is behaald. De basic course of events beschrijft het meest simpele, meest correcte pad. Er wordt vanuit gegaan dat er geen fouten worden gemaakt

door zowel de actor als het systeem. Ook word er vanuit gegaan dat het pad beschreven in de basic course of events het meest gebruikte pad is. Hoewel de use case in principe bestaat uit tekst kan het bij de basic course of events geen kwaad om ter ondersteuning diagrammen te gebruiken. Denk hierbij aan flowcharts of Petri nettenwerken.
Een simpel voorbeeld van een basic courcse of events is:

1. De actor voert zijn gebruikersnaam en paswoord in.
2. Het systeem controleert de correctheid van de gegevens
3. Het systeem bevestigt de correctheid van deze gegevens.
4. De actor neemt toegang tot het systeem.

- Binnen alternative paths worden de minder voorkomende paden beschreven waarmee het doel van de use case kan worden behaald. Alternative paths beginnen vaak wanneer er een ongewone conditie plaats vindt. Denk bijvoorbeeld aan de keuze tussen betalen met cash, credit card of pinpas. De meest voorkomende methode van betalen is dan de basic course of events terwijl de twee alternatieven alternative paths zijn. Alternative paths hoeven niet steeds weer bij het begin te beginnen. Er moet wel worden aangegeven bij welke stap binnen de basic cource of events het alternative path begint. Als het niet duidelijk is welk pad nou de basic cource of events is en welke nou de alternative path is kies dan de simpelste als basic course of events.
- Exception paths lijken op alternative paths. Exception paths verschillen van alternative paths in het feit dat zij aangeven wat er gebeurd nadat er een fout heeft plaats gevonden. Een voorbeeld van een exception path is het pad dat word genomen nadat een actor een incorrecte datum heeft ingevoerd.
- Extension points worden gebruikt om aan te geven waar in een use case een extend plaats vind. De extend is al eerder in dit hoofdstuk behandeld.
- Triggers beschrijven de criteria waarmee een use case kunnen worden gestart. Dit kan bijvoorbeeld een tijd zijn of het succesvol afronden van een andere use case.
- Onder assumptions staan de aannames die voor deze use case gelden. Hier staan alle dingen waar binnen deze use case vanuit word gegaan dat ze waar zijn terwijl dit niet vanzelfsprekend is. Dit is een erg belangrijk onderdeel binnen de use cases omdat er moet worden gezorgd dat deze aannames daadwerkelijk gelden wil deze use case later door het systeem worden vervuld.
- Bij preconditions staan alle dingen die moeten gelden voordat de use case in werking kan treden. Het betreft hier zaken buiten het bereik van de use case. Een voorbeeld hiervan is de aanwezigheid van een database voor een use case die het wegschrijven van persoonsgegevens naar een database beschrijft.
- De postconditions geven aan wat er is veranderd nadat de use case is uitgevoerd. Net als preconditions betreffen postconditions zaken buiten het bereik van de use case. Bij het voorbeeld uit het vorige kopje zou de postconditie zijn dat de persoonsgegevens in de database staan.
- Business rules zijn de geschreven en ongeschreven regels die bepalen hoe een bedrijf zijn business voert. Binnen de sectie related business rules kan er worden verwezen naar business rules die gerelateerd zijn aan de use case.

- Onder het kopje author komt de auteur van de use case te staan. Deze staat onderaan de use case omdat binnen RUP is besloten dat de meest kritieke informatie bovenaan komt te staan.
- De datum komt als laatste. Hier worden de data bijgehouden waarop de verschillende iteraties zijn afgerond.

Scenario's

Use Cases kunnen niet het hele verhaal vertellen. Om op de meer gedetailleerde interacties te focussen zijn er scenario's. Scenario's zijn een realisatie van een use case. Dit houdt in dat ze een specifiek pad binnen de use case weergeven op een meer specifiek niveau. [1] Hieronder zal een voorbeeld gegeven worden van (een deel van) een use case en bijbehorend scenario.

Use Case Name:	
Basic Course of Events:	1. De maatschappelijk werker voert de naam en werksituatie van de deelnemer in. 2. Het systeem reageert door te stellen of de deelnemer recht heeft op financiële steun.
Alternative Path	Bij stap 1, als de deelnemer zich al eerder heeft aangemeld en hij dit zelf aangeeft, dan zoekt de maatschappelijk werker naar eerdere data

Nu het bijbehorende scenario:

1. De maatschappelijk werker vraagt Edward Trueman of hij zich al eerder heeft aangemeld voor steun en of hij deze heeft ontvangen. Mr. Trueman antwoord dat hij zich al eerder heeft aangemeld.
2. De maatschappelijk weker zoekt op Mr. Truemans naam in het systeem.
3. Het systeem levert de eerdere gegevens over Mr. Trueman op, die duidelijk maken dat hij op december 1997 al eerder een aanvraag voor steun heeft ingediend vanwege zijn eerdere part-time aanstelling bij Boeing Aerospace als fabriekswerker.

Scenario's kunnen voor twee doelen gebruikt worden. Ten eerste kunnen ze worden gebruikt om onmiddellijk terugkoppeling te geven naar de gebruikers en analisten om te controleren of de use cases de behoefte van de gebruikers goed weerspiegelen. Ten tweede kunnen ze tijdens het testen worden gebruikt of de requirements terug te vinden zijn in het computer systeem. Om dit samen te vatten, scenario's zijn instanties van use cases die gebruikt worden om use cases begrijpbaar te maken voor de gebruikers zodat ze gecontroleerd kunnen worden.

Iteraties binnen Use Cases Methode

Zoals eerder vernoemd word binnen use cases het requirements document in drie fases ontwikkeld. De drie iteraties van requirements documenten zijn facade, filled en focus.[1]

De eerste iteratie is de facade iteratie. Binnen deze iteratie wordt een inventaris gemaakt van alle belangrijke interacties die de actors zullen gaan hebben met het systeem. De use cases die binnen de facade iteratie worden gemaakt hoeven alleen een naam, een korte beschrijving en de initiërende en andere gerelateerde actors te bevatten. [1] Dit is misschien een van de moeilijkste iteraties omdat er mogelijk nog geen duidelijk concept is van de applicatie die moet worden ontwikkeld.

De eerste stap binnen de facade iteratie is het bepalen van de mission, vision en values van de applicatie. Hierin betreft mission het doel van het project, vision een eerste beeld van het eindproduct en values de principes die binnen het project gelden. De mission, vision en values behoren van de opdrachtgever zelf te komen of op zijn minst in zijn naam te zijn.

De tweede stap is het doornemen van de al bestaande documentatie. Het doel hiervan is om jezelf bekend te maken met de voorgeschiedenis van het project. Wie zijn voorstanders van het project en wie zijn tegenstanders? Welke mogelijkheden zijn er bij voorbaat al afgeschoten en door wie? Tot welk niveau management is dit project doorgedrongen? Hoe lang ligt het idee voor dit project er al? Door deze vragen te beantwoorden kun je al eventuele requirements vinden die in eerste instantie niet duidelijk zijn. Ook kun je mogelijk hiermee struikelblokken voorspellen. [1]

De derde stap is de blik die de opdrachtgever zelf op het systeem heeft te achterhalen. Uiteindelijk is hij of zij de persoon die de beslissing maakt. Daarom is deze stap cruciaal. Als dit niet duidelijk is loop je het risico dat de requirements alsnog worden afgekeurd door de oprachtgever.[1]

De vierde stap is alle documentatie van het bedrijfsproces doornemen. Documentatie over het bedrijfsproces dat relevant is voor het te maken systeem kan zeer waardevol zijn. Het kan mogelijk zelfs al hele use cases opleveren. [1]

Stap vijf bestaat uit het identificeren van de alle mogelijke gebruikers van het systeem en eventuele klanten of andere relevante stakeholders. Dit is nodig om te bepalen wie geïnterviewd moeten worden als informatiebron voor de use cases. [1]

Stap zes is het daadwerkelijk interviewen van de stakeholders. Er zijn verschillende manieren om dit te doen. Dit kan een op een of in een workshop met alle stakeholders tegelijk. [1]

Als de eerste interviews zijn gehouden kan er een definitieve lijst van stakeholders worden opgesteld. Na de interviews weet je namelijk zeker of de eerder geïdentificeerde stakeholders ook echt stakeholders zijn.[1]

Stap 8 is het vinden van de actors. Dit zijn de mensen en systemen die daadwerkelijk interactie hebben met het systeem. Probeer elke stakeholder te vragen wie hij of zij denkt dat de actors zijn.[1]

Stap 9 is het maken van een use case survey. Het resultaat hiervan is een verzameling use case templates met alleen namen, korte beschrijvingen en actors. Dit is de eerste poging om alle use cases te identificeren. Het aantal use cases mag zeker nog veranderen bij het komen tot de uiteindelijke requirements.[1]

Stap 10 bestaat uit het identificeren van de non functional requirements. Eerder in dit hoofdstuk is uitgeld wat deze precies inhouden. In latere iteraties worden deze nog verfijnd.[1]

Stap 11 is het opstellen van een catalogus van business rules. Binnen de context van requirements gaat het om de business rules die het maken van de use cases voor dit systeem beperken. Deze catalogus hoeft in de facade iteratie nog niet volledig te zijn.[1]

Stap 12 is het maken van een risico analyse. Nu er een beeld is van de stakeholders en mogelijke use cases kan er een analyse worden gemaakt van de risico’s bij het opstellen van de requirements.[1]

Stap 13 is het maken van een statement of work. Dit moet uiteindelijk dienen als contract waarin staat wat er gedaan moet worden bij deze opdracht.[1]

Stap 14 en 15 betreffen het experimenteren met user interface metaphors en storyboards. Hiermee probeer je de stakeholders(vooral de gebruikers) al een eerste idee te geven van hoe het systeem eruit ziet. User interfaces zijn wat de gebruiker zelf ziet en door een idee te schetsen van hoe dit met deze use cases eruit zal zien krijgen de gebruikers een beter beeld van het ontwerp wat er nu ligt.[1]

Stap 16 is het verkrijgen van goedkeuring voor deze iteratie van de opdrachtgever.[1]

De volgende iteratie is de filled iteratie. Het doel van deze iteratie is om vanuit de use case survey te komen tot een zo volledig mogelijke verzameling use cases. Het idee achter de filled iteratie is dat het beter is te veel use cases te hebben dan te weinig. Dan weet je tenminste zeker dat alle requirements ook naar boven zijn gekomen. Later kun je de verzameling requirements altijd doen inkrimpen. Net als de facade iteratie is de filled iteratie weer onder te verdelen in stappen.[1]

Stap 1 is het opdelen van use cases uit de facade iteratie die te generiek zijn. Er zullen use cases zijn die alleen niet alle situaties dekken. Deze moeten opgedeeld worden. Binnen deze stap moet ook het use case diagram voor het eerst uitgewerkt worden.[1] Alle nieuwe use cases en de relatie met andere use cases moeten gedocumenteerd worden en het use case diagram is hier een geschikte tool voor. Binnen deze stap moet er ook bepaald worden wat de scope moet zijn van de gemiddelde use case. Dit wordt use case granularity genoemd. Dit is afhankelijk van de grootte van de applicatie.[1]

Stap 2 is het maken van filled use cases. Het doel is om voor alle use case het template volledig in te vullen. In het geval van het voorbeeld voor een template dat in dit hoofdstuk is gegeven kan de volgorde van alle onderdelen in zekere mate aangehouden worden bij het filled maken van de use cases.[1] Het is belangrijk dat er een goed woordenboek is voordat dit alles word gedaan om inconsistenties in terminologie te voorkomen.[1]De grootste en belangrijkste stap bij het filled maken van de use cases is het maken van de basic course of events. Hier worden de specifieke stappen van de interactie tussen actor en systeem voor deze use case beschreven. Samen met de exception en alternarive paths vormt dit het functioneel ontwerp dat binnen de use case.

Stap 3 is het verzamelen en toevoegen van business rules aan de business rules catalogus. Het is in de filled iteratie de bedoeling dat simpelweg alle business rules die men tegenkomt bij het maken van de use cases toevoegd.

Stap 4 is het testen van de filled use cases. Onderdeel hiervan is controleren op volledigheid en inconsistenties binnen en tussen de use cases. Dit moet gedaan worden voordat de use cases getoond worden aan de stakeholders. [1] Anders krijg je hier ook feedback op terwijl je deze problemen zelf al had kunnen vinden en oplossen.
Daarna moeten de filled use cases worden voorgelegd aan de stakeholders. Dit kan door middel van scenario's. Deze zijn eerder in dit hoofdstuk al behandeld. Elke use case kan verschillende scenario's bevatten en het is belangrijk ze allemaal aan de relevante stakeholders te tonen. De vraag die je hierbij stelt aan de stakeholders is of de use cases ook daadwerkelijk het beoogde systeem weergeven.[1]

Stap 5 is het verwijderen van een aantal dingen. Hoewel het beter is te veel use cases te hebben dan te weinig moet het tijdens de filled iteratie niet zo zijn dat de scope van de use cases te veel word uitgebreid. Zorg dat de use cases zich beperkten tot wat en niet hoe.[1] En zorg ook dat de scope van het beoogde systeem niet opeens groter is geworden. Als dit gedaan is kan men naar de volgende stap. Daar kan de verzameling use cases nog verder worden ingeperkt tot een coherent geheel.

De derde en laatste iteratie is de focused iteratie. Het doel van de focused iteratie is het inperken van het grote geheel aan use cases tot een verzameling duidelijke project requirements. De focused iteratie scheid het essentiële van wat leuk is om te hebben. Aan het eind van deze iteratie is er een systeem gedefinieerd en voldoende informatie verzamelt om tot een succesvolle applicatie te komen. Deze definitie bevat alles en dan ook alleen alles wat er in moet staan.[1]Net als de filled iteratie bestaat deze iteratie uit de use case methode uit 5 stappen.

Binnen stap 1 word de relatie van een use case met alle andere use cases bekeken. Dit word gedaan met het doel om use cases samen te voegen. Soms moeten een aantal use cases gegeneraliseerd worden en soms kan een use case onderdeel worden van een andere use case. Dit is een cruciaal onderdeel van deze iteratie.[1] Als dit niet goed gedaan word is de verzameling use cases onoverzichtelijk en loop je het risico dat use cases die eigenlijk dezelfde interactie beschrijven verschillend worden geïnterpreteerd binnen de verdere ontwikkeling van de applicatie puur en alleen omdat er twee use cases zijn.

Stap 2 is het aanbrengen van focus in iedere use case. Iedere use case moet compleet, precies en beknopt zijn.[1] Alle nodige informatie moet er in staan maar alle nutteloze informatie moet eruit. Later in het ontwikkelproces moeten andere ontwerpers aan de hand van jou use cases het systeem verder ontwikkelen en je wil niet hun tijd verdoen.

Stap 3 is het omgaan met scope creep. Omdat requirements constant veranderen is er een kans dat de scope van de beoogde applicatie veranderd. Het kan bijvoorbeeld zijn dat een systeem wat bedoeld was als hulpmiddel van de financiële administratie om hun administratie bij te houden nu in het ontwerp ook door het management word gebruikt om de financiële administratie te controleren. Wees bewust van dergelijke veranderingen en ga na of ze gewenst zijn of niet. Zo niet, elimineer de use cases die hier verantwoordelijk voor zijn.[1]

Tijdens het ontwerpen van de use cases kom je geheid dingen tegen die niet in de use cases thuis horen maar wel relevant zijn voor de verdere ontwikkeling van het systeem. In stap 4 moeten deze dingen worden gedocumenteerd. Het gaat hier vaak over aannames die voor het gehele systeem gelden. Het is ook belangrijk om deze aannames te presenteren aan de gebruikers. De scope van de use cases word sterk beïnvloed door deze aannames en zonder deze aannames kunne gebruikers en latere ontwerpers een verkeerd beeld krijgen van de use cases.[1]

Stap 5 is de review. Het is belangrijk dat iemand anders werk binnen deze iteratie controleert op fouten of dingen die missen. Aan het eind van deze iteratie zijn de use cases klaar dus men moet alles doen om te garanderen dat er geen fouten of gemiste kansen meer in de use cases zitten.

Na deze stap zijn alle iteraties afgerond en is als het goed is het requirements ontwikkel proces afgerond. De functionele requirements zijn allemaal gedocumenteerd in use casus en de non-

functionele requirements staan in een apart document. De volgende stap binnen de ontwikkelfase van het systeem kan dan worden uitgevoerd aan de hand van deze requirements.

Requirements in de architectuur

Inleiding

In dit hoofdstuk zal uitgelegd worden waar binnen de fysieke architectuur en gerelateerde zaken uit de bouw requirements terug komen. Dit hoofdstuk zal onder andere ingaan op de verschillende wetten waar architecten rekening mee moeten houden bij hun ontwerp. Hoewel in eerste instantie de relatie tussen wetten en requirements niet vanzelfsprekend lijkt valt het tegendeel te beargumenteren. Een aantal wetten stelt eisen aan de functies van een gebouw. Deze eisen zijn opgesteld door de overheid en gedocumenteerd in de wetten. Binnen de context van requirements is de overheid een stakeholder en zijn de eisen die voortkomen uit de wetten requirements. Stel dat de welstandsnota van de gemeente vereist dat een nieuw gebouw op het marktplein een tweevoudige deur heeft. Dan is het hebben van een tweevoudige deur een requirement van de overheid. Alleen is de overheid een stakeholder die middelen heeft om te garanderen dat zijn requirements opgenomen worden in het ontwerp. Verder in dit hoofdstuk zullen verschillende middelen worden behandeld waarmee de overheid requirements kan verplichten of andere stakeholders wettelijke middelen kan geven om bepaalde requirements te verplichten.

De fysieke architectuur

Het is moeilijk om een algemene eenduidige definitie te bepalen voor fysieke architectuur. In Compton's Encyclopaedia vinden we een meer fundamentele definitie van architectuur.

Deze luidt: *By the simplest definition, architecture is the design of buildings, executed by architects. However, it is more. It is the expression of thought in building. It is not simply construction, the piling of stones or the spanning of spaces with steel girders* [12].

Een andere definitie luidt: *It is the intelligent creation of forms and spaces that in themselves express an idea*[12].

Vitruvius, de bouwmeester van Julius Caesar en Augustus, onderscheidde in de eerste eeuw voor Christus drie aspecten aan architectuur: 'utilitas','firmitas' en 'venustas'. Utilitas staat voor de gebruiksaspecten: doelmatigheid, nuttigheid en deugdelijkheid. Firmitas staat voor fysieke zaken als: duurzaamheid, vastheid en sterkte. En venustas staat voor bekoorlijkheid: uiterlijk schoon, dus de beleving.[Daan Rijsenbrij, 12] Als we dit terugkoppelen naar requirements zoals deze binnen de informatica worden gebruikt, dan zouden functionele requirements utilitas moeten betreffen, aangezien zij betrekking hebben tot de gewenste functies aan het computersysteem en daarmee het gewenste nut moeten bepalen. Non-functional requirements betreffen zaken zoals veiligheid, stabiliteit en snelheid. Dit komt overeen met firmitas, de 'fysieke' aspecten aan het computersysteem. En venustas komt nergens terug in de requirements.

Als we uitgaan van deze twee definities van architectuur, dan zouden requirements de ideeën zijn van de verschillende stakeholders. De overheid, de bewoners, omwonende, en vele andere stakeholders hebben allemaal ideeën over hoe het gebouw moet zijn. Deze ideeën hoeven niet volledig te zijn. Een buur zou zich in zijn ideeën kunnen beperkten tot een gebouw waarvan het

uiterlijk hem bevalt en wat hem geen verdere overlast veroorzaakt. Maar het zal hen weinig uitmaken hoe het is om te werken en leven in het gebouw. Het is dan uiteindelijk aan de architect om de ideeën van alle verschillende stakeholders te interpreteren en terug te laten komen in het uiteindelijke volledige ontwerp.

Het architectenbureau MVRDV probeert rekening te houden met de verschillende stakeholders door middel van zogenaamde datascapes. Datascapes visualiseren de ruimtelijke consequenties van de verschillende belangen van alle betrokken partijen. Iedere datascape is hierbij het ideale ontwerp voor een bepaalde partij. Over de verschillen en tegenstellingen tussen al die datascapes word weliswaar in het ontwerpproces met de verschillende partijen net zo lang onderhandeld tot het ontwerp kan worden gerealiseerd, maar ze worden nooit gladgestreken of in de vormgeving tot een synthese gebracht. Hiermee probeert MVRDV zijn gebouwen een brutale en pragmatische, licht anarchistische uitstraling te geven. [[13] pg. 123] MVRDV gaat dus een stap verder dan requirements en maakt meteen een ontwerp per stakeholder in plaats van meteen de requirements te verenigen in een ontwerp. Toch gebruikt MVRDV deze datascapes uiteindelijk om tot een ontwerp te komen samen met alle partijen waar alle partijen in zekere mate tevreden mee kunnen zijn.

Maar MVRDV is uniek in deze aanpak. Andere architecten of architectenbureaus missen een aanpak waar zo duidelijk rekening wordt gehouden met verschillende stakeholders. Dit betekent niet dat andere architecten geen rekening houden met de verschillende partijen en de omgeving waarin het gebouw moet worden gebouwd. Jo Coenen, rijksbouwmeester van 2000 tot 2004, onderstreept het belang van onderzoek naar de omgeving waarin het gebouw moet komen te staan. *Het gaat erom de bestaande kwaliteiten uit te buiten.*[Jo Coenen [14] pg.42] Dit vergt diepgaand onderzoek. Om de kwaliteit van een plaats te bepalen, moet je altijd ruim de tijd te nemen. *Je kunt een locatie pas kennen als je er eerst heen bent gegaan, en er dan heel vaak bent teruggekeerd.*[Jo Coenen [14] pg.42] Onderzoek naar de omgeving is voor architecten erg belangrijk. En de omgeving beperkt zich niet alleen tot omliggende gebouwen en de natuur. Ook mensen en hun werk zijn onderdeel van de omgeving. *Wel kijk ik op een drukke plek in een stad ook expliciet naar wat mensen er doen. En dat doe ik dan niet even, maar gewoon een hele dag*[Jo Coenen [14] pg.42] Hoewel het onderzoek dat architecten doen naar hun omgeving niet per se in de vorm van requirements word gedaan is onderzoek naar de omgeving belangrijk. Hoewel het MVRDV met zijn datascapes net als bij requirements binnen de informatica open de verschillende eisen van alle partijen bespreken en hierover onderhandelen, kan een architect ook zelf een onderzoek doen zonder direct vragen te stellen aan alle betrokken partijen. Dit wordt ook gedaan door het gedrag en karakter van de betrokken partijen te onderzoeken door middel van observatie

Requirements beperken zich tot de eisen van alle betrokken partijen bestaand uit mensen. Binnen de informatica worden requirements traditioneel achterhaald door mensen die deze verschillende partijen vertegenwoordigen vragen te stellen en uit te dagen antwoorden te geven. *De nieuwe gemeenschappelijke bibliotheek van de Technische Universiteit Delf is direct achter de aula van Van den Broek & Bakema gerealiseerd: een nuks, brutalistisch betonnen gebouw uit het begin van de jaren zestig, dat eigenlijk geen ander gebouw naast zich duldt*[[13] pg.107] Mecanoo, het architectenbureau dat de nieuwe bibliotheek moest bouwen, heeft dit opgelost door een gebouw te maken door een gebouw te maken dat van buitenaf niet als zodanig word erkend. Het gebouw is onder gras verborgen. Dit voorbeeld toont aan dat architecten niet alleen rekening moeten houden met de eisen van alle betrokken menselijke partijen. Ook de directe omgeving moet bestudeerd

worden om tot een goed gebouw te komen. Alleen mensen interviewen om tot een volledige documentatie van alle requirements te komen is niet genoeg. De volledige omgeving moet worden bestudeerd en bij gebouwen, objecten die zich altijd in de openbare ruimte bevinden, maken de natuur en omliggende bouwwerken altijd deel uit van de omgeving. Requirements alleen zijn niet genoeg. Er moet ook onderzoek gedaan worden naar niet menselijke onderdelen van de omgeving. En omdat iedereen beperkte tijd en middelen heeft moet er een keuze worden gemaakt hoeveel aandacht te besteden aan requirements.

De aspecten utilitas en firmitas van de architectuur worden binnen het concept van requirements gedekt. Maar venustas, het schoon en de belevenis van gebouwen, komt niet terug in het idee van requirements zoals gepresenteerd in de informatica. Toch is venustas een belangrijk onderdeel van de architectuur. Het uiterlijk schoon en de belevenis van gebouwen is iets waar architecten veel waarde aanhechten. Het is wat architectuur een kunst maakt. En omdat architectuur kunst is maakt het ook onderdeel uit van de cultuur. De nationale identiteit van de architecten word weer gereflecteerd in hun eigen stijl. *Ook in de architectuur, de mode en het design speelt de gedachte die door het citaat van Calvijn word verwoord, op de achtergrond nog steeds een rol.*[[13] pg. 16] Het citaat van Calvijn, ‘ Laten zij die over vloed hebben eraan denken dat ze zijn omgeven met doornen, en laten ze goed oppassen dat ze er niet door geprikt worden’ [[13] pg. 15], heeft grootte invloed op het Nederlands cultuurgoed. Nederlanders geven bijvoorbeeld van alle Europeanen per jaar het minste geld uit aan kleding.[[13] pg. 16] Dit citaat leidt ertoe dat in Nederland het niet de gewoonte is om door het uiterlijk van o.a. een gebouw welvaart te laten blijken. Het is niet zo dat venustas, het uiterlijk schoon en het idee, dat weer neerslag heeft in de cultuur, dat erdoor uitgedragen word als onbelangrijk word beschouwd. *Dat leidde er in 1990 zelfs toe dat de minister van WVC en VROM samen een architectuurnota het licht deden zien, waarin een overheidspolitiek werd uitgestippeld om de ‘culturele component’ van de architectuur te stimuleren.*[[13]pg.13/14] Hoewel het probleem waarop deze architectuurnota een reactie was niet relevant is voor het argument is het wel relevant om op te merken dat de overheid zelf het culturele aspect van architectuur belangrijk genoeg vind om er een nota over te maken.

Venustas is aspect van architectuur gelijkwaardig aan de andere aspecten. Alleen requirements houden zich hier helemaal niet bezig. Architecten hebben ook niet de gewoonte om aan stakeholders te vragen welke stijl zij voor hun gebouw wensen. Elke architect heeft zijn eigen stijl, onderbouwd door een beargumenteerde reactie op de huidige cultuur, en de opdrachtgever neemt de stijl van de architect mee in zijn keuze. Of het mogelijk is om ook requirement op te stellen voor venustas blijft een vraag. Maar elk kunstenaar zal je vertellen dat een kunstenaar een eigen stijl nodig heeft en deze niet volledig kan schikken naar de eisen van ene opdrachtgever.

Bouwplan

De overheid heeft verschillende regels en wetten waar gebouwen aan moeten voldoen. Vandaar dat voordat er kan worden begonnen aan het uitvoeren van een ontwerp van een gebouw eerst het ontwerp moet worden goedgekeurd. Meestal zijn het de gemeente of de provincie die deze regels en wetten tot detail invoeren. Maar soms, zoals bij de bouwtechnische eisen, vult de centrale overheid deze wetten en regels in. Binnen de context van de bouwwet heet dit ontwerp het bouwplan. Een bouwplan is niet altijd onderhevig aan de zelfde hoeveelheid wetten en regels. Afhankelijk van de locatie en de functie van het gebouw gelden er verschillende regels, soms meer en soms minder. Het bestemmingsplan bepaalt bijvoorbeeld op welk grondgebied binnen de gemeente welk soort

gebouw mag worden gebouwd. Op sommige percelen zijn er strenge eisen wat betreft de functie, technische aspecten, of, volgend uit de welstandsnota, het uiterlijk van de gebouwen terwijl op andere percelen bijna vrij kan worden gebouwd. In dit deelhoofdstuk zal in worden gegaan op de verschillende wetten en regels waar een bouwplan aan moet voldoen. Na een korte uitleg van deze wet zal deze wet uitgelegd worden als een requirement waarbij het doel van de wet en de te vertegenwoordigen stakeholder worden behandeld.

Bestemmingsplan

In een bestemmingplan staat wat er met de ruimte in een gemeente mag gebeuren. Bijvoorbeeld of er op een bepaalde plek een fabriek mag worden gebouwd of kantorencomplex etc. De gemeente stelt het bestemmingsplan op. Het bestemmingsplan geld voor burgers, bedrijven en de gemeente zelf. Bouwplannen zullen altijd moeten aansluiten op het bestemmingsplan. Anders zal er geen vergunning worden uitgegeven.[7]

Het bestemmingsplan komt voort uit de wet ruimtelijke ordening die opgesteld word door de centrale overheid. Deze wet geeft voornamelijk gemeente en de mogelijkheid om bestemmingsplannen op te stellen en geeft aan wie allemaal voor wat verantwoordelijk is. Naast het bestemmingsplan volgt uit de wet ruimtelijke ordening ook nog een structuurvisie. Deze wordt opgesteld binnen de provincie in samenwerking met gemeente en centrale overheid. De structuurvisie bevat hoofdlijnen van de voorgenomen ontwikkeling van een gebied, evenals de hoofdzaken van het door de provincie te voeren ruimtelijk beleid. Het bestemmingsplan van een gemeente moet weer passen in deze structuurvisie.[8]

Het bestemmingsplan zelf bestaat uit een kaart en een document. Op de kaart wordt aangegeven hoe de grond is verdeeld en welke bestemmingen de percelen hebben. In het bijbehorend document worden in de begripsbepaling de verschillende bestemmingen gedefinieerd en in de verder e hoofdstukken de procedures voor wijziging, bezwaar of ontheffing bij het bestemmingsplan gedefinieerd.

Het bestemmingsplan bevat dus eisen aan de functies die een gebouw op een bepaalde locatie mag vervullen, volgent uit besluiten van de gemeente en wetten en regels van hogere autoriteiten. Door middel van het bestemmingsplan proberen de overheden een ontwerp te maken voor Nederland waarin locaties gekoppeld worden aan functies. Binnen de context van requirements is de gemeente een stakeholder en bevat het bestemmingsplan een deel van haar requirements. Het verschil met requirements zoals we kennen uit de informatica is dat de requirements in het bestemmingsplan al van te voren gedocumenteerd zijn en niet onderhandelbaar zijn. Verder volgen deze requirements uit de locatie waar het project zal plaatsvinden. De gemeente kan dit eisen omdat de zij, in naam van het volk, de locatie beheren en er op toe moeten zien dat de indeling van de gemeente bevorderlijk is. Voor het project moet er dus een locatie gezocht worden met een functie, volgend uit het bestemmingsplan, die past bij de functie die de opdrachtgever en architect voor ogen hebben. Er valt weinig te onderhandelen met de stakeholder, de gemeente, anders dan het overtuigen van de gemeente dat de functie van het project ook overeen komt met de functie die aan de locatie is toegewezen. Het bestemmingsplan bevat eisen van een stakeholder, en kan hierdoor inderdaad gezien worden als een vorm van requirements. De vorm is alleen wel sterk verschillend van wat men binnen de informatica gewend is.

Welstand

Welstand betekent dat het bouwplan architectonisch past in de omgeving. In de welstandsnota legt de gemeente zo concreet mogelijk vast welke welstandscriteria gelden voor verschillende delen van de gemeente. Alleen bij vergunningsvrij bouwen hoeft een bouwplan niet te voldoen aan de criteria uit de welstandsnota. Wel geld dat het bouwplan niet sterk mag afwijken van de redelijke eisen van welstand. Ook moet er rekening gehouden worden met de buren. Maar dit zal verder worden behandeld in het volgende deelhoofdstuk.[7]

De woningwet verplicht gemeente een welstandsnota op te stellen. Hieruit volgt dat de welstandscommissie advies uitbrengt aan de burgemeester en wethouders ten aanzien van de vraag of het uiterlijk of de plaatsing van een bouwwerk of standplaats, waarvoor een aanvraag voor bouwvergunning is ingediend, in strijd is met redelijke eisen van welstand. [9, woningwet] De welstandscommissie is een onafhankelijke commissie benoemd door de gemeenteraad. De bouwverordening bevat de voorschriften betreffende de samenstelling, inrichting en werkwijze van de welstandscommissie. Ook kan er een stadsbouwmeester worden aangesteld in plaats van een welstandscommissie. Deze bouwverordening wordt opgesteld door de gemeente.

De welstandscommissie controleert, maakt en geeft advies over de welstandsnota. Dit wordt gedaan in naam van en voor de gemeente. Binnen de context van requirements zou de gemeente de stakeholder zijn van de requirements die volgen uit de welstandsnota en is de welstandscommissie verantwoordelijk voor de documentatie en beheer. Zoals van een overheid valt te verwachten vertegenwoordigd de gemeente als stakeholder in het bouwproject een deel van de belangen van de burger. Net als bij het bestemmingsplan is de welstandsnota een ononderhandelbare set requirements die van tevoren al gedocumenteerd is en waarvan het beheer buiten het project valt. De mate waarin een bouwproject moet voldoen aan de welstandsnota kan variëren maar een bouwplan moet gecontroleerd worden op overeenkomst met de welstand. Net als het bestemmingsplan bevat de welstand ook requirements, alleen op een ander gebied. En net als het bestemmingsplan is de vorm sterk verschillend van wat men in de informatica is gewend.

Buren

Buren hebben rechten en plichten ten opzichte van elkaar. Deze zijn wettelijk vastgelegd. Daarom moet er bij een bouwproject rekening worden gehouden met de buren. Hoewel de gemeente niet controleert of het bouwplan in overeenstemming is met de buren. Buren hebben wel de mogelijkheid om bezwaren in te dienen tegen een bouwplan. In [9] staat wettelijk vastgelegd wat de rechten en plichten van buren zijn. Artikel 37(J) luidt bijvoorbeeld:
Artikel 37 (J): De eigenaar van een erf mag niet in een mate of op een wijze die volgens artikel 162 van Boek 6 onrechtmatig is, aan eigenaars van andere erven hinder toebrengen zoals door het verspreiden van rumoer, trillingen, stank, rook of gassen, door het onthouden van licht of lucht of door het ontnemen van steun.(Red: zie ook art. 13 bk 3 BW; artt. 1 lid 2, 21, 40 bk 5 BW; artt. 168, 174 lid 1 bk 6 BW; artt. 159, 161 ONBW)
Dit artikel verplicht een buur dat hij eigenaars van andere erven geen hinder toebrengt door verspreiden van onder andere rook of gassen.

Binnen de context van requirements zijn de buren de stakeholders en bepaalt [9] op welk gebied de burger requirements mag stellen waarbij de wet verplicht dat het bouwproject aan deze requirements voldoet. Artikel 37(J) geeft buren bijvoorbeeld de wettelijke macht om te zorgen dat er

geen overlast wordt veroorzaakt door een aangrenzend bouwproject, zoals bepaald in artikel 37(J) en gerelateerde artikelen. De ontwerpers van het bouwproject wordt dus verplicht om in de uiteindelijke requirements eisen van de buren op het gebied van overlast mee te nemen. Buren zijn dus stakeholders die het wettelijk recht hebben dat requirements betreffende bepaalde gebieden opgenomen worden in het ontwerp.

Bouwtechnische eisen

Ook op technisch gebied stelt de overheid eisen aan gebouwen en bouwprojecten. Technische details zijn in principe geen onderdeel van de requirements. Maar non-functional requirements op het gebied van onder andere veiligheid kunnen wel leiden tot technische eisen. De bouwtechnische eisen van de overheid staan in het bouwbesluit 2003. Dit betreft technische eisen op het gebied van constructie, veiligheid, gezondheid, energiezuinigheid en milieu. Het bouwbesluit bevat in zijn volledigheid alle eisen voor het hele land. In tegenstelling tot het bestemmingsplan en de welstand word het bouwbesluit niet verder uitgewerkt door andere overheden. Het is al volledig uitgewerkt door de centrale overheid.[7][10]

Binnen de context van requirements is het bouwbesluit op zich niet zo interessant. Wel zijn de onderliggende non-functional requirements. Helaas zijn deze moeilijk te achterhalen omdat de overheid in een keer het bouwbesluit volledig heeft uitgewerkt met beperkte verwijzingen naar de achterliggende motivatie. Wel is het interessant om te zien dat een stakeholder, namelijk de overheid, in zijn taak om de burger te vertegenwoordigen een zo'n uitgebreid pakket van technische eisen opstelt.

Analyse verschillen en overeenkomsten

ICT en bouw

Voordat kan worden begonnen met een analyse van verschillen en overeenkomsten tussen requirements binnen de informatica en architectuur moet er eerst gekeken worden naar de verschillen tussen ICT en bouw op zich. De bouw houdt zich bezig met gebouwen. En gebouwen hebben een aantal fundamentele eigenschappen, namelijk:

- Gebouwen nemen fysieke ruimte in beslag.
- Gebouwen zijn zichtbaar in deze fysieke ruimte.
- Hoewel gebouwen andere gebouwen kunnen bevatten zijn gebouwen nooit binnen een ander artefact waardoor gebouwen altijd zichtbaar zijn in de omgeving.

Omdat gebouwen zo aanwezig zijn in de omgeving beïnvloeden gebouwen de omgeving. En vaak in dusdanige maat dat hun impact op de omgeving gevolgen heeft voor het algemene gebied. Hierdoor word het algemene publiek een stakeholder in alle gebouwen die vertegenwoordigd word door de overheid. In Nederland en bijna alle landen in de wereld is de overheid mede-eigenaar van al het grond en daarmee mede-eigenaar van de grond waarop een gebouw gebouwd gaat worden. Dit heeft tot gevolg dat de overheid de wettelijke macht heeft om eisen te stellen aan gebouwen.

De ICT houdt zich bezig met het ontwikkelen van computersystemen. Soms betreft dit combinaties van hardware en software maar soms alleen software. Hardware neemt net als een gebouw fysieke ruimte in beslag. Maar deze is vaak kleiner dan de fysieke ruimte die een gebouw inneemt. En

hardware is vaak opgenomen in een ander object dat fysieke ruimte inneemt, zoals een gebouw, waardoor deze niet zichtbaar is van buitenaf. Software is echter fundamenteel anders dan gebouwen. Software neemt namelijk geen fysieke ruimte in. Software is de programmering van een specifiek gedraging van hardware. Het is zelf niet zichtbaar en kan alleen waargenomen worden door gedragingen van hardware. Als combinatie van hardware en software hebben computersystemen de volgende eigenschappen:

- Hardware neemt fysieke ruimte in beslag, software niet.
- Hardware is zichtbaar in de fysieke ruimte, software is zichtbaar aan de hand van de gedragingen van hardware.
- Hardware is vaak opgenomen in andere fysieke objecten waardoor het zelden zichtbaar is in de omgeving.

Computersystemen zijn veel makkelijker te verbergen dan gebouwen. Vaak kunnen ze alleen gezien worden door de mensen die ze mogen zien. Ze zijn lang niet zo aanwezig in de publieke ruimte als gebouwen waardoor de overheid veel minder inspraak heeft op computersystemen. Hierdoor hoeft een informaticus veel minder rekening te houden met de overheid.

Toch is het verschil niet zo duidelijk als het lijkt. Hoewel het internet niet meer fysieke ruimte inneemt dan de servers waar zij opstaat, is zij toch toegankelijk voor iedereen die toegang heeft tot een computer met internet. In een land als Nederland, waar bijna iedereen die toegang heeft, is het internet dus niet veel minder publiek dan de omgeving. En hoewel toegang tot websites beperkt kan worden zijn ze altijd enigszins zichtbaar. Toch heeft de overheid lang niet de invloed op het internet als zij op de omgeving is. Hoewel het niet het doel is van dit artikel om diep in te gaan op cyberspace en zijn plaats in de samenleving is het wel belangrijk dit op te merken omdat het een aantal parallellen tussen de bouw en ICT oplevert.

Hoewel de artefacten die de ICT en de bouw opleveren fundamentele verschillen hebben zijn er ook overeenkomsten. Deze overeenkomsten kunnen helpen om bij de vergelijking nieuwe inzichten te verkrijgen in requirements. In de verdere analyse zal ingegaan worden op verschillende overeenkomsten en verschillen om tot nieuwe inzichten te komen.

Venustas

De drie aspecten van architectuur zijn venustas, utilitas en firmitas. Firmitas, de fysieke degelijkheid, en utilitas, het nut, worden gedekt door respectievelijk non-functional en functional requirements. Alleen venustas wordt nergens behandeld in requirements. Sterker nog, er is binnen de informatica weinig tot geen aandacht voor venustas op een hoog niveua. Binnen de opleidingen informatica en informatiekunde aan de RU word hier in ieder geval bijna geen aandacht aan besteed. Zijn computersystemen dan fundamenteel anders dan gebouwen opdat venustas niet relevant is bij computersystemen en bij gebouwen wel?

Deze vraag kent vele aspecten en het is niet de intentie van dit artikel om alle aspecten van de ICT te behandelen om een antwoord te geven. Maar het is wel de intentie om deze vraag binnen de context van requirements te beantwoorden. Het doel van requirements is te achterhalen wat alle eisen van alle stakeholders zijn, zodat er een ontwerp kan worden gemaakt waar zoveel mogelijk stakeholders tevreden mee zijn. Maar mensen beoordelen niet alleen op utilitas en firmitas. Hoe een computersysteem eruit ziet en hoedanig werken met het systeem ervaren word zijn wel degelijk

aspecten waarop een computersysteem zal worden beoordeeld, zeker door de mensen die er later mee moeten werken. Interface design is ook wel degelijk een onderdeel van het ontwerp van computersystemen. De user interface voor het computersysteem komt (meestal) niet zomaar uit het niets. Er wordt een ontwerp gemaakt en als het goed is word dit ontwerp ook aan toekomstige gebruikers voorgelegd zodat zij voor het finale product nog een oordeel kunnen geven over het uiterlijk. Hiervoor is de usability test of bruikbaarheidtest. Daar word door een testpubliek getest of het systeem bruikbaar is. De belevenis en het schoon worden hierbij beoordeeld.

Maar als venustas wel degelijk van belang is bij computersystemen, waarom wordt er bij requirements dan helemaal geen aandacht aan besteed terwijl firmitas en (vooral) utilitas de aandacht krijgen? Use Cases geven bijvoorbeeld een duidelijk ontwerp van wat het computersysteem moet kunnen. En non-functionals worden in overeenstemming met de use case methode ook gedocumenteerd. Maar de ontwerpers en ontwikkelaars die verder moeten met de requirements documentatie hebben geen idee wat voor eisen er aan het uiterlijk en de belevenis van het computersysteem zijn. Het citaat van Calvijn en de gevolgen hiervan voor de Nederlandse architectuur geven aan wat voor impact nationale cultuur kan hebben op het uiterlijk van een gebouw. Waarom zou het voor een computersysteem dan niet uitmaken of het in Nederland of India gaat worden gebruikt? De nationale cultuur kan grote impact hebben op de eisen wat betreft uiterlijk en belevenis. En cultuur houdt niet op bij de nationale grenzen. Bedrijven hebben ook hun eigen culturen en die hebben ook invloed op venustas.

De bedrijfscultuur en de consequenties van de cultuur voor de stijl zouden ook bij de requirements gedocumenteerd kunnen worden. Net als bij functionele en non-functionele eisen kunnen ook eisen aan het uiterlijk naar gevolg van verschillende culturen in het bedrijf conflicteren. Om een uiterlijk en belevenis te creëren in het computersysteem waar iedereen tevreden mee is zouden de verschillende culturen en de gevolgen voor de stijl net als de verschillende eisen op andere gebieden kunnen worden gedocumenteerd. En net als voor alle andere eisen zou er dan in het requirements document kunnen worden besloten over een uiteindelijke stijl voor het computersysteem. Bij een open bedrijfscultuur zou er bijvoorbeeld besloten kunnen worden alle interfaces voor alle functies dezelfde stijl te geven zodat het computersysteem het idee wekt dat, ondanks de andere uit te voeren taak, iedereen gelijk is. Bij een sterk hiërarchisch, gesloten bedrijfscultuur is het tegenovergestelde misschien gewenst. Verschillende functies moeten duidelijk verschillende interfaces hebben om de differentiatie tussen functies bij het werken met het computersysteem terug te laten komen.

Waarom hier geen aandacht aan word besteed binnen requirements, of meer algemeen in de beginfase van het ontwerp, binnen de informatica kan ik niet beantwoorden. En als er geen antwoord is moet er gekeken worden hoe venustas dan wel meegenomen kan worden binnen de beginfase van het ontwerp. Architectuur laat ons zien dat venustas belangrijk is en binnen requirements word hier geen aandacht aan besteed. Is er een reden dat venustas niet belangrijk is of is het een gebrek van de ICT dat hier geen aandacht aan word besteed?

Is alleen vragen genoeg?

Binnen de requirements engineering methode worden requirements hoofdzakelijk boven water gehaald door interviews met medewerkers. Men vraagt aan de medewerkers wat zij belangrijk vinden. Workshops is de methode die nog het meest afwijkt van vragen maar ook hier word er vaak

expliciet gevraagd keuzes te motiveren en word er achteraf gevraagd waarom een bepaalde actie is uitgevoerd. In geen van deze gevallen worden de stakeholders in een natuurlijke situatie onderzocht.

Architecten gebruiken echter zelden interviews voor hun ontwerp. Communicatie beperkt zich meestal tot communicatie met de opdrachtgever en medeontwerpers. Als bijvoorbeeld Jo Coenen een gebouw in een stad wil bouwen onderzoekt hij het gebruik van de omgeving door de mensen in hun dagelijkse bezigheden te observeren. Hij gaat niet mensen vragen wat zij dagelijks doen. Het voordele hiervan is dat mensen niet altijd antwoorden geven die overeenkomen met de werkelijkheid. Als de mensen direct geobserveerd worden zal dit de correctheid van de onderzoeksresultaten ten goede komen. Nu is het natuurlijk een stuk minder storend wanneer een architect midden in de drukke stad rondloopt en observaties maakt dan wanneer er in een toch meer gesloten omgeving iemand, terwijl je aan het werk bent, jou loopt te observeren.

Daarnaast kijken architecten ook veel naar eerdere bouwwerken en hoe deze tot stand zijn gekomen. Door alle problemen en alles wat goed is gegaan te analyseren kan een architect bepalen hoe hij bij zijn project zijn werk het beste kan aanpakken. De architect is zich dan bewust van de valkuilen waardoor zijn collega's eerdere fouten hebben gemaakt. Waar informatici meer vertrouwen op van te voren bepaalde methodes vertrouwen architecten meer op hun kennis van eerdere projecten en het verloop van het ontwerp. Er zijn veel autobiografische boeken over architecten waarin zij hun blik op de architectuur en hun blik op collega's uitleggen. Binnen de informatica is dit minder en zijn er meer boeken die een aanpak of methode voorschrijven.

Het is niet de bedoeling om een waardeoordeel te geven aan de verschillende aanpak. Maar er zijn meer manieren om requirements te achterhalen dan alleen vragen. Observeren of eerdere vergelijkbare projecten bestuderen zijn ook manieren om te achterhalen welke requirements er gelden. Nu is observatie voor informatici moeilijker dan architecten. Waar bij architecten de omgeving vaak statisch is, is deze bij informatici vaak zo veranderlijk dat de informatici zelf de verandering niet bij kunnen benen. Hoewel er misschien andere en meer complicaties zijn bij andere onderzoeksmethoden dan interviews e.d. dan voor architecten is de informatica momenteel eenzijdig in zijn aanpak om onderzoek te doen naar requirements. Er zijn meer methode om de omgeving te onderzoeken en het is de moeite waard om hiernaar te kijken.

Requirements in wetsteksten

Het eerste verschil tussen de informatica en de architectuur is dat architecten bij hun ontwerp rekening moeten houden met een groot aantal wetsteksten die de overheid als stakeholder de mogelijkheid geven om eisen aan het te ontwerpen bouwwerk te stellen. Hoewel informatici wel degelijk rekening moeten houden met de wet, de overheid heeft bij het ontwerp van software geen wettelijke middelen om ontwerpen af te keuren opdat zij geen rekening houden met de specifieke requirements van de overheid. Binnen de architectuur heeft de overheid een bestemmingsplan en welstandsnota om eisen te stellen aan het bouwproject op basis van zijn locatie. Ook garandeert de overheid dat burgers op bepaalde gebieden inspraak hebben. Verder heeft de overheid ook een uitgebreid pakket aan technische eisen waar verplicht aan moet worden voldaan die een aantal non-functional requirements garanderen.

Gevolg van deze wetten en regelgeving is dat een groot deel van de requirements bij een bouwproject al van te voren zijn vastgelegd. De architect moet zijn bouwplan laten goedkeuren door de overheid, die kijkt of haar requirements wel terug gevonden kunnen worden in het plan. Ook

kunnen burgers bezwaar maken tegen het plan als zij zich als buur benadeeld voelen op punten die door de wet beschermd zijn. Het proces van requirements engineering wordt hierdoor voor een deel anders. Het is van te voren duidelijk dat de overheid een stakeholder is en wat haar requirements zijn. Verder zijn deze requirements niet onderhandelbaar. Het is een uitdaging voor de architect om een ontwerp te maken wat een compromis is tussen de eisen van de overheid en andere stakeholders. Een informaticus is veel vrijer dan een architect bij het ontwerpen van een systeem. De informaticus heeft de vrijheid te bepalen aan welke requirements in het ontwerp wel en niet word voldaan. Hij moet ook een compromis maken tussen verschillende requirements van verschillende stakeholders maar hij wordt niet verplicht door de overheid bepaalde requirements op te nemen.

Hoewel de informaticus vrijer is dan de architect in de keuze welke requirements hij wel of niet meeneemt zorgt het gebrek aan verplichtingen van de overheid er ook voor dat het de informaticus meer werk kost om de requirements vast te leggen. Voor de architect is het duidelijk dat de strekking van de wet terug moet komen in het ontwerp. Als eventuele requirements van andere stakeholders conflicteren met de wet is dat jammer voor die stakeholders. De architect moet zich aan de wet houden. De informaticus zal zich nooit achter de wet kunnen verschuilen. Er is geen wet die de informaticus verplicht dingen in zijn ontwerp op te nemen dus hij zal zijn keuzes altijd moeten beargumenteren naar verschillende stakeholders. De architect zou bepaalde stakeholders moeten teleurstellen omdat hun ideale ontwerp tegen de wet ingaat. De wet zal de informaticus nooit tegenhouden bij het maken van het ontwerp. Alleen de opdrachtgever zou dit kunnen doen.

Niet alle stakeholders hebben evenveel macht en de machtigste stakeholders zijn lang niet altijd de stakeholders die het beste ontwerp voor ogen hebben. Een informaticus kan door zijn opdrachtgever gedwongen worden een ontwerp te maken waarvan duidelijk is dat het resulterende systeem onveilig zal zijn of waarvan het gebruik nadelig zal zijn voor de uiteindelijke gebruikers. Binnen de bouw zorgt de wet ervoor dat gebouwen veilig zijn en voorkomt de wet dat opdrachtgevers in een poging kosten te besparen onveilige gebouwen neerzetten. Ook beschermt de wet de buren waardoor buren kunnen voorkomen dat er een gebouw wordt gebouwd wat voor hen overlast zal veroorzaken. Voor het maken van computersystemen zijn dergelijke wetten er nog niet. Niks weerhoud een opdrachtgever ervan om systemen te wensen die onveilig zijn anders dan zijn eigen verstand. En hoewel een informaticus de opdrachtgever kan trachten te overtuigen dat het verstandig is om alle stakeholders, en op zijn minst de toekomstige gebruikers, aan te horen en hun requirements mee te nemen kan hij dit de opdrachtgever niet verplichten.

Er zijn wel een aantal fundamentele verschillen die kunnen verklaren waarom de architect met veel meer wetten rekening moet houden dan de informaticus. De welstandsnota bestaat omdat de meeste gebouwen in de openbare ruimte zichtbaar zijn en daardoor door iedereen gezien kunnen worden. De overheid, als vertegenwoordiger van het algemeen publiek, stelt daarom eisen aan het uiterlijk van gebouwen opdat gebouwen in hun omgeving passen. Maar informatiesystemen zijn vaak niet publiek toegankelijk en niet zichtbaar in de publieke ruimte. De overheid heeft er dan ook geen belang bij om, anders dan voor hun eigen informatiesystemen, te zorgen dat deze in hun omgeving passen. Dit is aan de organisatie waarbinnen het informatiesysteem een rol speelt. Het bestemmingsplan is er omdat de overheid mede-eigenaar van alle grond is en omdat gemeente voor hun grondgebied een plan willen kunnen opstellen om de ontwikkeling van hun gemeente te kunnen sturen. De overheid is geen eigenaar van informatiesystemen of eventuele cyberspace waarbinnen de informatiesystemen zullen worden gemaakt. Een bestemmingsplan voor informatiesystemen zal

op nationaal niveau dan ook niet mogelijk zijn. Omdat informatiesystemen zich meer in de private sfeer begeven dan gebouwen heeft de overheid minder belang bij het onder controle houden van de informatiesystemen. Daarom zijn er meer wetten en regels nodig die bepalen of een bouwplan word goedgekeurd dan het ontwerp van een informatiesysteem.

Toch kunnen wetten en regels ook nut hebben bij het ontwerpen van informatiesystemen. De wet beschermt buren voor overlast van gebouwen door buren middelen te geven om de bouw van gebouwen tegen te houden. Een werkplaats waar 24 uur lang gewerkt word kan veel overlast veroorzaken door lawaai van het werken. Buren worden uit hun slaap gehouden wat ernstige gevolgen voor hun gezondheid kan hebben. In het geval van informatiesystemen zou de wet werknemers inspraak kunnen geven bij het ontwerp van informatiesystemen zodat de informatiesystemen die de werknemers later gaan gebruiken prettig zijn om mee te werken. Nu is het aan de opdrachtgever om te bepalen of hij zijn werknemers inspraak geeft en hoe zwaar hij deze laat mee wegen.

Het internet is een apart geval. Iedereen die wil kan toegang krijgen tot het internet. Daarmee is het in principe een publiek domein. Hoewel er wel degelijk organisaties zijn die delen van het internet beheren beperken deze zich tot meer technische aspecten en is het publiek vrij om materiaal op het internet te zetten. Overheden kunnen niet voorkomen dat men materiaal op het internet plaatst en alleen die materialen die door het merendeel van de overheden als illegaal worden beschouwd kunnen echt verwijderd worden. Het is dus niet mogelijk voor een overheid om het internet te vormen als de publieke ruimte. Sommige overheden trachten toegang te blokkeren tot materiaal waarvan zij niet willen dat het publiek deze ziet. Maar dit moet per site gebeuren en is zelfs voor de machtigste overheden een uitdaging.[11] Het is niet het doel van deze scriptie om in te gaan op de vraag of het technisch dan wel niet mogelijk is voor nationale overheden om het internet te controleren. Maar uit het beleid dat de overheid in de bouw volgt blijkt de overheid wel de behoefte te hebben controle uit te oefenen op publieke ruimte. Om de ontwikkeling van de publieke ruimte te sturen, zoals het bestemmingsplan of om de kwaliteit te garanderen door middel van het bouwbesluit. En zoals in het geval van de Chinese overheid duidelijk word[11] hebben overheden ook de behoefte het publieke internet te regeren. En hoewel de controle die de Chinese overheid over het internet wil uitoefenen door de meeste mensen in de Westerse samenleving als ongewenst word ervaren is men in de Westerse samenleving niet vreemd van overheden die zaken onder controle proberen te houden. Als de overheid de mogelijkheid heeft om het internet te regeren is het dus mogelijk om te kijken naar hoe de overheid de bouw regeert als voorbeeld van het controleren van objecten in een openbare ruimte. De overheid zou bijvoorbeeld een plan kunnen opstellen om de ontwikkeling van het internet en domeinen binnen het internet te sturen, net als een bestemmingsplan de bijdrage van de bouw aan de ontwikkeling van een gemeente controleert.

Maar wetten en regels zijn niet alleen interessant voor de overheid. Door de welstandsnota probeert een gemeente het uiterlijk van gebieden consistent te houden en een bepaalde uitstraling te behouden. Bedrijven zouden voor hun software, zowel intern als extern, een eigen welstandsnota kunnen opstellen om de uniformiteit tussen hun verschillende systemen, en via het internet naar buiten toe, te behouden. De overheid biedt steeds meer diensten via het internet aan, aan de burger. Een soort van welstandsnota voor internetsites van de overheid zou kunnen helpen om deze sites een uniform uiterlijk te geven. Dit kan de burger helpen om een site van de overheid als echt te herkennen. Hoewel de wetten en regels behandeld in deze scriptie van de overheid zijn kunnen

wetten en regels zich beperkten tot kleinere organisaties. De wetten van de overheid kunnen helpen om te bepalen hoe requirements die binnen een organisatie in het algemeen gelden gedocumenteerd en gehandhaafd kunnen worden.

Conclusie

Hoewel architectuur op vele fronten fundamenteel anders is dan de informatica zijn er wel degelijk overeenkomsten. De eisen van alle betrokken partijen, requirements binnen de informatica, komen in beide vakgebieden in een of andere vorm voor. Maar hoewel de architectuur bij zijn ontwerpt zich richt op de drie aspecten van de architectuur, utilitas, firmitas en venustas, richt lijkt de informatica zich, in ieder geval binnen requirements, alleen maar bezig te houden met utilitas en firmitas of in ieder geval veel minder met venustas. Waarom word in dit artikel niet duidelijk en de vraag is of dit voor het vakgebied informatica wel duidelijk is. Het is zeker aan te raden om verder onderzoek te doen of venustas wel of niet belangrijk is en als dit wel belangrijk is, hoe de informatica venustas moet bevatten in hun ontwerpen.

Een ander aspect waarop de informatica en architectuur van elkaar verschillen is de manier waarop onderzoek word gedaan naar de eisen aan het te ontwerpen object. Informatici vertrouwen bij het onderzoeken naar de requirements vooral op interviews of andere vraagvormen. Architecten vertrouwen vaker op observaties. Ook baseren zij hun ontwerp vaker op ervaringen opgedaan door eerdere ontwerpen van collega's. Er zijn meer methode dan interviews of andere vraagvormen en hoewel er wel een paar argumenten zijn om interviews te kiezen boven andere methode zijn die niet overtuigend genoeg om alleen maar interviews te gebruiken. Nu word er wel meer dan alleen interviews gebruikt binnen de informatica bij het achterhalen van requirements maar interviews overheersen in veel methodes en het is belangrijk te onthouden dat er andere mogelijkheden zijn.

Architecten moeten rekening houden met veel wetten en regels van de overheid. Binnen de informatica is dit aanzienlijk minder. Hoewel er nog veel moet veranderen voordat er net zo veel regelgeving is voor ICT als voor de bouw, en het de vraag is of dit ooit zal gebeuren, is het wel belangrijk te realiseren dat er veel meer regelgeving mogelijk is en dit grote effecten op het ontwerpproces kan hebben. De overheid zou naar de bouw kunnen kijken wat ze allemaal wensen te reguleren en wat niet binnen de ICT en de informatici kunnen, als het duidelijk word dat er veel meer regelgeving is, naar de architecten kijken hoe zij hier mee omgaan tijdens hun ontwerp en hoe zij de regelgeving proberen te beïnvloeden.

Er zijn veel verschillen tussen het vakgebied van informatici en architecten. Toch zijn er ook overeenkomsten en in de toekomst mogelijk meer overeenkomsten. Het is zeker de moeite waard om te kijken naar wat architecten of andere ontwerpers doen zodat de informatica zich op zijn minst bewust is van alle mogelijkheden. Ik hoop dat dit artikel helpt te overtuigen dat dit inderdaad het geval is en dat dit zal leiden tot verder onderzoek.

Bronvermelding

[1]:Daryl Kulak, Eamonn Guiney; *Use Cases, Requirements in Context*; Addison-Wesley, 2004

[2]:Frauke Paetsch, Dr. Armin Eberlein, Dr. Frank Maurer; *Requirements Engineering and Agile Software Development*; 12th IEEE International Workshops on Enabling Technologies, 2003

[3]:Martin Fowler; *The new methodology*; http://www.martinfowler.com/articles/newMethodology.html, 2003

[4]:Ken Schwaber, Mike Beedle; *Agile Software Development with Scrum*; Prentice Hall, 2001

[5]:Pekka Abrahamsson, Outi Salo, Jussi Rankainen, Juhani Warsta; *Agile software development methods – Review and analasys*; 2002, VTT Electronics

[6]:Keith Snook, *RIBA Quality Management System: Procedures Manual*;, RIBA Quality Management Toolkit, December 2006

[7]:*Gids bouwregelgeving*; http://www.vrom.nl/gidsbouwregelgeving

[8]:*Wet Ruimtelijke Ordening 566*; Staatsblad van het Koninkrijk der Nederlanden, Oplage 2006

[9]:*Titel 4. Bevoegdheden en verplichtingen van eigenaars naburige erven*; Burgerlijk Wetboek, Boek 5

[10]:Ministerie van VROM; *Bouwbesluit online*

[11]: Assafa Endeshaw; *Internet Regulation in China: The Never-ending Cat and Mouse game*; Information & Communication Technology Law, Vol. 13, No. 1, 2004

[12]: Daan Rijsenbrij; *Architectuur, een begripsbepaling*; https://lab.cs.ru.nl/algemeen/Architectuur/Rijsenbrij/1._Architectuur_-_een_begripsbepaling; Digitale werkplaats III

[13]: Bart Lootsma; *Superdutch*; Thames & Hudson, 2000

[14]: Hilde de Haan, *Jo Coenen, van standsontwerp tot architectonisch detail*; NAi Uitgevers; 2004