Component-Based Development
en
Betrouwbaarheid van Software

door S. de Gilde

Component-Based Development

en

Betrouwbaarheid van Software

door S. de Gilde
<svendegilde@hotmail.com>

onder begeleiding van dr. ir. J. van den Berg

Afstudeerscriptie Bestuurlijke Informatiekunde
Faculteit der Economische Wetenschappen
Erasmus Universiteit Rotterdam

Februari 2005

Ondersteund door Matthijs en Vrijenhoek Software Development (MVSD)

Inhoudsopgave

Voorwoord

Met het schrijven van deze scriptie komt er een einde aan mijn studie Bestuurlijke Informatiekunde aan de Erasmus Universiteit, waarvan het verloop zonder twijfel ongebruikelijk kan worden genoemd. Na in 1991 begonnen te zijn met de studie Econometrie, ben ik na het behalen van mijn propedeuse overgestapt naar de studie Economie, met als afstudeerrichting Bestuurlijke Informatiekunde. Diverse verhuizingen en perioden waarin ik afwisselend veel heb gewerkt en weinig heb gestudeerd en andersom, hebben elkaar afgewisseld en ook het kopen en renoveren van een schitterend huis met mijn vriendin heeft mijn studieverloop sterk beïnvloed.

Tijdens mijn studie en werk ben ik gefascineerd geraakt door hergebruik van software. Het ideaal dat aanzienlijke delen van nieuwe systemen niet opnieuw hoefden te worden ontwikkeld, maar dat er gebruik gemaakt kon worden van reeds ontwikkelde en geteste delen en de implicaties die dit zou hebben voor aspecten als de kwaliteit en de kortere doorlooptijd, heeft me gegrepen en niet meer losgelaten. De komst van Java 2 Enterprise Edition en later Microsoft .NET heeft de mogelijkheden om dit ideaal te realiseren een stuk dichterbij gebracht en het werken met deze technologieën heeft mijn fascinatie alleen maar aangewakkerd, wat uiteindelijk heeft geresulteerd in het schrijven van deze scriptie.

Dit voorwoord biedt me de gelegenheid een aantal personen te bedanken. Hierbij wil ik beginnen met mijn begeleider, Jan van den Berg. Dat deze scriptie zonder hem niet tot stand gekomen zou zijn, is een understatement. Met een altijd enthousiaste en positieve instelling, stimulerende discussies en het uitstralen van vertrouwen - zelfs wanneer dit vanwege omstandigheden vrijwel onmogelijk moet zijn geweest - heeft hij mij weten te motiveren door te gaan. Daarnaast wil ik mijn werkgevers, Marcel Matthijs en Mineke Vrijenhoek, bedanken voor het bieden van de faciliteiten om mijn onderzoek uit te voeren, diverse versies te reviewen en me te blijven motiveren mijn scriptie af te ronden. Tevens wil ik mijn ouders, Ali en Gert de Gilde, bedanken omdat ze altijd zijn blijven geloven in mijn kunnen. Een bijzonder woord van dank wil ik richten aan mijn schoonouders, Hannie en Ger van de Wetering, die er alles aan hebben gedaan om me tijdens mijn studie te ontlasten, waarbij ze zelfs de verbouwing van ons huis hebben afgemaakt terwijl ik aan het werk was. Tot slot wil ik mijn vriendin, Maris van de Wetering, bedanken voor de wijze waarop ze mij tijdens mijn studie heeft gesteund. Ik had mijn studie en deze scriptie in het bijzonder niet zonder jouw hulp kunnen afronden!

Sven de Gilde

Zandvoort, 1 februari 2005

1 Inleiding

1.1 Achtergrond

Onze samenleving raakt steeds meer doordrongen van computers. Gevolg hiervan is een groeiende afhankelijkheid van deze computers en de applicaties en systemen die erop draaien. Hiermee groeit ook het belang van de kwaliteit van de software. Slechte software zal in toenemende mate leiden tot ergernis en problemen. Voorbeelden hiervan zijn aperte fouten (taal- en interpretatiefouten), beperkte functionaliteit of de gevolgen van virusaanvallen of hackpogingen die mogelijk zijn door de aanwezigheid van kwetsbaarheden in de software.

De wijze waarop software wordt gemaakt is volop in beweging. Binnen het vakgebied dat zich met de gestructureerde ontwikkeling van software bezighoudt – Software Engineering – kunnen op diverse vlakken ontwikkelingen worden waargenomen. Zo zijn er verschillende modellen gepresenteerd die de levenscyclus van software beschrijven. Dit is de cyclus die – net als bij andere producten – de fasen van het product beschrijft en in hoofdlijnen bestaat uit de fasen analyse, ontwerp, implementatie, oplevering, onderhoud en afstoting. Ook zijn er in de loop van de tijd diverse programmeer-paradigma's opgekomen die invloed hebben op de manier waarop software wordt ontwikkeld. Eén van deze paradigma's is de componentgewijze ontwikkeling (component-based development of kortweg CBD) van software. Hierbij wordt een systeem opgesplitst in relatief zelfstandige delen en wordt er geprobeerd maximaal gebruik te maken van reeds ontwikkelde en geteste softwarecomponenten.

Vanuit de gedachte dat de kwaliteit van software steeds belangrijker wordt, is het interessant om vast te stellen of de invloed die het toepassen van CBD heeft op de kwaliteit van software, vastgesteld kan worden. Omdat het begrip kwaliteit heel breed is, is er gekozen voor het deelaspect betrouwbaarheid. Hieronder wordt, vooruitlopend op de definitie en operationalisering van dit begrip in Hoofdstuk 5, de kwaliteit vanuit het gezichtspunt van de gebruiker verrstaan. Het onderwerp van deze scriptie is daarom de invloed van het toepassen van CBD op de betrouwbaarheid van software.

1.2 Doelstelling

Deze scriptie heeft als doel om antwoord te geven op de vraag of het toepassen van CBD binnen het softwareontwikkelingsproces invloed heeft op de betrouwbaarheid van het eindresultaat van dit proces, te weten de software en indien dit het geval is, wat deze invloed dan is.

Om dit doel te realiseren wordt er antwoord gegeven op de volgende vraag:

Wordt de betrouwbaarheid van software verbeterd door gebruik te maken van Component-Based Development?

Probleemstelling

Deze probleemstelling vormt de centrale vraag van deze scriptie. Aan de hand van de volgende doelstellingen wordt een poging gedaan om antwoord te geven op de in de probleemstelling geformuleerde vraag:

1. Het verschaffen van inzicht in de kenmerken waarin component-based development verschilt van andere vormen van softwareontwikkeling.
2. Het vaststellen van de gevolgen van deze verschillen voor de betrouwbaarheid van de software.

Doelstellingen 1 en 2

Het begrip betrouwbaarheid is hierin nog vrij vaag en moet nader worden toegelicht. Het is mogelijk te kijken naar de betrouwbaarheid van een bepaalde versie van een softwaresysteem op een bepaald moment. Hierbij worden uitsluitend kenmerken van die specifieke versie van het systeem beschouwd, waardoor gesproken kan worden over de statische betrouwbaarheid van die versie. Daarentegen kan ook gekeken worden naar de betrouwbaarheid van de verschillende, opvolgende, versies van het systeem. Indien de verschillende versies van het systeem naast elkaar worden gezet, vormen de opvolgende betrouwbaarheden van al deze versies de ontwikkeling of evolutie van de betrouwbaarheid van het systeem. Deze evolutie van de betrouwbaarheid van het systeem kan de dynamische betrouwbaarheid van dit systeem worden genoemd.

Terugkerend naar het doel van de scriptie kan worden gesteld dat de laatste doelstelling specifieker op de onderstaande wijze kan worden geformuleerd.

2. Het vaststellen van de gevolgen van deze verschillen voor de dynamische betrouwbaarheid van de software.

Doelstelling 2

1.3 Methodologie

Zoals reeds gesteld is het onderwerp van deze scriptie de invloed van CBD op de betrouwbaarheid van software. Om deze invloed vast te kunnen stellen, dient er vergelijking plaats te vinden met ontwikkeling van software zonder gebruikmaking van CBD. Deze wijze van ontwikkeling wordt vanaf nu aangeduid als ‘conventionele ontwikkeling’, een op deze wijze ontwikkeld systeem een ‘conventioneel systeem’.

Om de gevolgen van de toepassing van CBD op de betrouwbaarheid van software te kunnen vaststellen, is een aantal zaken verduidelijkt. Zo is het verschil tussen CBD en conventionele ontwikkeling toegelicht. Daarnaast is duidelijk gemaakt wat onder betrouwbaarheid van software wordt verstaan. Hiertoe is een literatuuronderzoek verricht waarvan de bevindingen in deze scriptie zijn beschreven.

Nadat duidelijk is geworden wat onder bovenstaande begrippen verstaan wordt, zijn de specifieke aspecten van CBD, die relevant zijn voor de betrouwbaarheid, geanalyseerd. De veronderstellingen die uit deze analyse volgden, zijn geformuleerd als hypothesen. Deze hypothesen zijn vervolgens geoperationaliseerd, wat inhoudt dat ze observeerbaar ofwel meetbaar zijn gemaakt. Hierbij speelde de beschikbaarheid van meetgegevens een cruciale rol. De operationalisering was noodzakelijk om de statische en dynamische betrouwbaarheid van de twee methoden van softwareontwikkeling te kunnen vaststellen.

Vervolgens kon van zowel conventionele softwareontwikkeling als CBD worden vastgesteld welke statische en dynamische betrouwbaarheid hiermee is gerealiseerd. Aangezien het niet mogelijk was om alle ontwikkelprojecten die door MVSD zijn uitgevoerd te bekijken, is er gekozen voor het onderzoeken van twee case studies die als representatief worden beschouwd voor deze twee vormen van softwareontwikkeling. Daarbij bleek er sprake te zijn van een praktisch probleem, te weten de beperkte beschikbaarheid van gegevens. Deze restrictie heeft geleid tot een specifieke operationalisering van het betrouwbaarheidsbegrip, welke gebruikt is om beide casestudies te analyseren.

Aan de hand van de resultaten van de casestudies zijn de hypothesen getoetst. De conclusies die hieruit zijn getrokken, stelden ons in staat de probleemstelling te beantwoorden en aanbevelingen te doen voor nader onderzoek.

1.4 Structuur

Deze scriptie bestaat uit drie delen: een theoretisch deel, een toetsend deel en een concluderend deel.

Het theoretische deel beslaat hoofdstukken twee, drie en vier. Hoofdstuk twee bevat een inleiding tot systeemontwikkeling en beschrijft het vakgebied van software engineering. Daarnaast worden de belangrijkste ontwikkelmodellen beschreven en tot slot worden de ontwikkelingen op het gebied van het programmeren toegelicht. Hoofdstuk drie richt zich op de betrouwbaarheid van software. De begrippen softwarekwaliteit en softwarebetrouwbaarheid worden gedefiniëerd en de relatie hiertussen wordt beschreven. Vervolgens komt de wijze waarop kwaliteit en betrouwbaarheid gerelateerd zijn aan systeemontwikkeling aan de orde. Ook worden de belangrijkste modellen voor het beoordelen van softwarekwaliteit toegelicht. Hoofdstuk vier richt zich tot slot op de aspecten van betrouwbaarheid die specifiek van toepassing zijn op CBD. Hierbij wordt het totale spectrum van kwaliteit ingeperkt tot de betrouwbaarheidsaspecten die relevant zijn voor het onderzoek dat in het tweede deel van deze scriptie besproken wordt. Deze inperking leidt tot het opstellen van hypothesen die voorspellingen doen over de invloed van CBD op de betrouwbaarheid.

Het toetsende deel beslaat de hoofdstukken vijf en zes. Hoofdstuk vijf beschrijft de opzet van het onderzoek. In hoofdstuk zes worden de resultaten van het onderzoek gepresenteerd en worden deze geïnterpreteerd. Deze interpretatie bestaat uit de samenvatting van de vastgestelde kenmerken en de gevolgen die hier direct uit voortvloeien.

Het concluderende deel is beschreven in hoofdstuk zeven. Hierin worden de onderzoeksresultaten teruggekoppeld naar de hypothesen en worden er conclusies getrokken met betrekking tot de doelstellingen en probleemstelling van de scriptie. Tot slot worden er aanbevelingen gegeven voor nader onderzoek.

2 Softwareontwikkeling in historisch perspectief

2.1 Inleiding

Voordat de kwaliteitsaspecten van software in hoofdstuk drie aan de orde komen, kijken we eerst naar het ontwikkelen van software. Paragraaf 2.2 van dit hoofdstuk gaat in op het vakgebied software engineering en zet de historische fasen hierbinnen op een rij met hun belangrijkste kenmerken.

Bij de ontwikkeling van software is een levenscyclus vast te stellen waarbinnen de software verschillende fasen doorloopt. Er zijn verschillende modellen die deze levenscyclus beschrijven en in paragraaf 2.3 worden deze aan de hand van het soort model in chronologische volgorde besproken, waarbij ze worden gerelateerd aan de in de voorgaande paragraaf geïntroduceerde fasen binnen de software engineering.

Tot slot worden in paragraaf 2.4 de belangrijkste zaken uit dit hoofdstuk kort samengevat.

2.2 Softwareontwikkeling

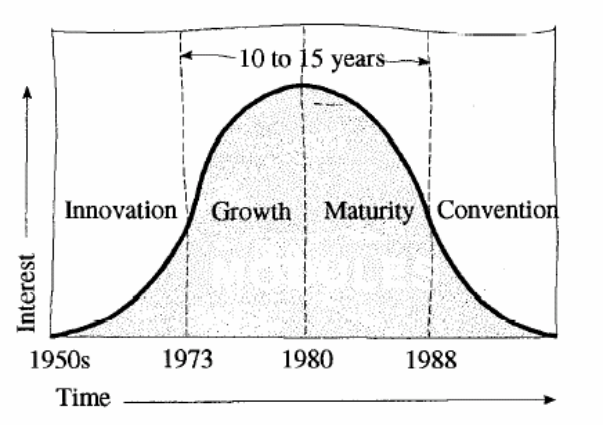
2.2.1 Watermetafoor

De evolutie van Software Engineering kan worden beschreven door gebruik te maken van de watermetafoor [Raccoon, 1997]. De watermetafoor maakt gebruik van de begrippen *wave*, *stream* en *tide*, welke in de volgende alinea's kort toegelicht worden.

Wave

Technologische ontwikkelingen maken een evolutie door die onder te verdelen is in een aantal fasen. Kijkend naar de mate van interesse die er vanuit de maatschappij is in een technologische ontwikkeling, zijn achtereenvolgens de volgende fasen te onderscheiden: innovatie, groei, volwassenwording en conventie. In de innovatiefase wordt de ontwikkeling opgepikt door een groep die het belang ervan onderkent. Bedrijven zien het potentiële belang ook, maar investeren er nog niet in. Na deze periode, die zo'n vijf tot twintig jaar kan duren, leidt een toename van de populariteit in de groeifase tot een steeds groter enthousiasme en een uitgebreid gebruik van de technologie. In de fase van volwassenwording raakt het begrip uitgebalanceerd en worden de beperkingen en grenzen van de technologie duidelijk. De groeifase en de fase van volwassenwording duren samen gemiddeld zo'n tien tot vijftien jaar. Uiteindelijk raakt in de conventiefase het basisidee volkomen geaccepteerd en worden er nieuwe technologieën ontwikkeld die verdergaan op de tekortkomingen van de huidige technologie. Indien deze fasen uitgezet worden tegen de mate van maatschappelijke interesse, ontstaat een figuur die lijkt op

een golf. Een voorbeeld hiervan is de volgende figuur, die de maatschappelijke interesse in de modulaire organisatie van software weergeeft.



Figuur 1: Fasen *wave* [Raccoon, 1997]

Stream

Indien verschillende, opeenvolgende golven betrekking hebben op hetzelfde thema, doordat ze zich richten op hetzelfde specifieke probleem, wordt deze reeks golven een stroom genoemd. Zo worden de ontwikkelingen op het gebied van hardware de '*hardware economics stream*' genoemd.

Tide

Indien er sprake is van meerdere waves met verschillende thema's, maar met overeenkomstige denkbeelden en opvattingen in een bepaalde tijd, wordt dit een tide genoemd. Een voorbeeld is de object tide, waarbinnen de denkbeelden en opvattingen over objecten in alle verschillende thema's, zoals organisatie van software en strategieën voor het ontwikkelen ervan, overeenkomstig zijn.

De vorderingen op het gebied van de softwareontwikkeling worden beschreven door de tides '*naive*', '*function*', '*structured programming*', '*module*', '*object*' en '*patterned programming*' [Raccoon, 1997].

2.2.2 Softwareontwikkeling

Binnen de softwareontwikkeling volgen verschillende paradigma's elkaar op. Een paradigma is het totaal van opvattingen, waarden, en gewoonten die een groep wetenschappers gemeenschappelijk heeft, een soort overkoepelende visie op een bepaald gebied. Kuhn [Kuhn, 1996] beschrijft wetenschapsontwikkeling als een cyclisch proces waarbinnen de fasen 'normale wetenschap' en 'revolutie' elkaar afwisselen [Veerman et al., 1994]. Tijdens de fase 'normale wetenschap' bestaat er een paradigma waarbinnen het aantal onoplosbare problemen beperkt is. Dit aantal neemt toe totdat er

een punt bereikt is, waarop er grote scepsis ontstaat over het paradigma. De wetenschapsontwikkeling bevindt zich in een crisis. Vervolgens wordt er in de fase 'revolutie' een vervangend paradigma gezocht dat de problemen, die volgens het oude paradigma onoplosbaar waren, wél kan oplossen. Zodra zo'n vervangend paradigma is gevonden, begint de fase van 'normale wetenschap' weer.

De paradigma's die binnen de softwareontwikkeling worden onderkend, hebben onder andere betrekking op de wijze waarop geprogrammeerd wordt en hoe software georganiseerd wordt. [Raccoon, 1997]. De onderkende paradigma's worden in het navolgende overzicht op een rij gezet.

Naive tide

Het 'Naive' getijde loopt van 1945 tot en met 1955 en kenmerkt zich door een tijdsgeest waarin computers 'neat' worden gevonden. In deze periode verschijnen er diverse mainframes, die voornamelijk voor onderzoek werden gebruikt. De ENIAC uit 1945 wordt veelal als de eerste computer beschouwd, hoewel dit feitelijk nog een calculator was omdat iedere stap door een gebruiker geïnstrueerd moest worden [Augarten, 1985]. De toename van de hoeveelheid beschikbaar geheugen en het opnemen van programma-instructies naast de gegevens in dit geheugen, maakte het mogelijk de computer beslissingen te laten nemen op basis van condities. Dit werd voor het eerst volledig electronisch gedaan in 1948 door de Mark 1 computer van de universiteit van Manchester. Oorspronkelijk bestond het programma nog uit instructies in de vorm van nullen en enen. Deze binaire vorm, ook wel machinetaal genoemd, was voor de programmeur lastig leesbaar.

De ontwikkeling van de assembler door Alan Turing leidde tot een aanzienlijke verbetering [Clements, 1994][Leventhal, 1981]. In essentie kon een programmeur nu symbolische namen als ADD en MOV, *mnemonics* geheten, gebruiken in plaats van de binaire instructies die ze representeren. Daarnaast kon hij logische namen, zoals A en B, kiezen voor constanten of variabelen in plaats van hun fysieke adressen. De assembler nam vervolgens het administratieve werk voor zijn rekening door de assembly broncode met deze mnemonics en symbolische namen om te zetten in binaire code. Hoewel de programma's zelf even complex bleven, fungeerde de assembler als een laag die deze complexiteit abstraheerde, wat het programmeren aanzienlijk vereenvoudigde.

Bij de eerste computers bestond de software uit opvolgende statements, conform de stapsgewijze werking van de hardware. Alle gegevens zijn hierbij globaal, wat inhoudt dat alle statements bij de gegevens kunnen. Initieel werden de statements geschreven in machinetaal, maar na de introductie van de assembler in 1949 ("Initial Order" op EDVAC) werden hiervoor voornamelijk nog assemblytalen gebruikt. Deze wijze van programmeren wordt aangeduid als 'statement-oriented programming'.

Function tide

De Function tide loopt ongeveer van 1956 tot en met 1966. De UNIVAC computer uit 1951 maakte gebruik van programma's die werden geschreven in short code, bestaande

uit alfanumerieke instructies, zoals *x* = *a* + *b*. Deze short code werd door een programma, dat *interpreter* wordt genoemd, regel voor regel omgezet in machinecode en uitgevoerd. Ondanks deze verbetering ten opzichte van assembly was de uitvinding van de *compiler* door Grace Hopper in datzelfde jaar veel belangrijker. Een compiler zet een programma in een logische taal in zijn geheel om in machinetaal. Hierdoor kan het omzetten en uitvoeren worden gescheiden en hoeft de omzetting slechts éénmaal plaats te vinden.

Deze beide ontwikkelingen, samen met de opkomst van de eerste commerciële computers, zoals de Ferranti Mark 1 in 1951, waren de innovatiefasen van verschillende technologische golven die samen het 'Function' getijde vormden. Kenmerkend voor dit getijde is de grote interesse in functies. Een belangrijke ontwikkeling is de eerste hogere programmeertaal door de ontwikkeling van de Fortran compiler door IBM in 1957.

De komst van deze FORTRAN compiler leidde tot het gebruik van functies, die verzamelingen bij elkaar horende statements bevatten. Later werd het concept functie verfijnd in de talen Algol en Pascal. Hierbij worden globale en locale gegevens van elkaar gescheiden en is het mogelijk de toegang tot de locale gegevens af te schermen. Deze wijze van programmeren wordt aangeduid als 'function-oriented programming'.

Structured Programming tide

De ontwikkeling van de eerste minicomputer van Digital Equipment Corporation (DEC) in 1963 en de System/360 computerfamilie van IBM het jaar erop waren voorlopers van de volgende ontwikkelingsgolven, leidend tot het 'Structured Programming' getijde, dat in 1967 begon en doorliep tot ongeveer 1977.

Doordat computers steeds goedkoper worden kunnen steeds meer bedrijven ze aanschaffen. Functies die voorheen met de hand werden uitgevoerd worden in toenemende mate geautomatiseerd. Er komen systemen die groter zijn dan tot nu toe gebruikelijk was. Dit leidt ertoe dat de vraag naar software enorm stijgt. De productiviteit wordt een belangrijk aspect van systeemontwikkeling.

Functies zijn inmiddels verworden tot conventie en de nadruk verschuift naar het goede gebruik ervan. De technologie is zo ver dat compilers voor Fortran en Algol loops kunnen optimaliseren, waardoor het mogelijk wordt om efficiënte applicaties te ontwikkelen zonder gebruik te maken van oncontroleerbare sprongen door middel van het goto-statement. Eén van de grote namen binnen dit paradigma is Dijkstra, welke onder andere bekend om een artikel dat is gepubliceerd als "The goto statement considered harmful". Het besef bestaat dat de control-flow van een programma uitsluitend bepaald dient te worden door concatenatie (opvolgen van statements), alternatie (conditionele uitvoering van statements) en repetitie (herhalen van statements).

Ontwerpen maken gebruik van stapsgewijze verfijning. Hierbij wordt een programma steeds verder uitgewerkt in kleinere delen die na getest te zijn weer worden samengevoegd tot een geheel programma. Algoritmes komen ook steeds meer in de belangstelling. Er ontstaat een duidelijk beeld van efficiëntie van algoritmes, zodat uit verschillende algoritmes het meest efficiënte alternatief kan worden gekozen.

Een laatste, maar niet minder belangrijk kenmerk van dit paradigma is het bewijzen van de formele juistheid van programma's, met behulp van invarianten en pre- en postcondities. Tegenwoordig wordt dit formele bewijs met name nog toegepast bij kritieke systemen die te kostbaar of risicovol zijn om in de praktijk te testen.

Module tide

De introducties van de Apple II computer in 1977 en de personal computer van IBM in 1981 markeerden het begin van het 'Module' getijde, dat doorliep tot ongeveer 1988. Hierbinnen heersten twee algemene denkbeelden.

Het ene denkbeeld bestond uit de opvatting dat zaken gegroepeerd dienden te worden. Algoritmes worden gegroepeerd binnen in deze periode populair wordende Abstracte Data Types (ADT's). Vanaf het begin van de jaren zeventig kwam er een steeds grotere nadruk op het samenvoegen van functies in modules, met als doel het optimaliseren van de werking van groepen functies. Programma's worden gezien als groepen modules. Gegevens kunnen worden gedeeld op globaal, module- en functieniveau, waarbij het mogelijk is het delen van gegevens door functies te beperken. Voorbeelden van moduletalen zijn C, Modula en Ada.

Het andere denkbeeld is de nadruk op de juistheid, waarbij de correctheid van algoritmes uitgebreid wordt naar hele programma's. Op het vlak van de programmeeromgevingen verschijnen er veel algemene hulpprogramma's, zoals vi, grep, en make op het Unix-platform. Binnen dit getijde groeit de behoefte aan hogere productiviteit en worden er steeds grotere en complexere projecten uitgevoerd, veelal in opdracht van defensie.

Object tide

Dit tijdperk loopt van 1989 tot ongeveer 1999 en kent twee grote denkbeelden: objecten en iteratie.

Functies en modules zijn geschikt om programma's met een grootte tot 50.000 à 100.000 regels code, effectief te organiseren [Raccoon, 1997]. Vanwege steeds complexere systemen en een toenemend aantal strengere eisen aan software voldoen deze wijzen van organisatie echter steeds minder goed. Objectoriëntatie wordt gezien als een oplossing voor dit probleem. Een programma kan worden gezien als de combinatie van enerzijds een verzameling gegevens en anderzijds het totaal van functionaliteit dat deze gegevens bewerkt. Waar traditioneel de nadruk lag op de functionaliteit van het systeem, is deze nadruk nu verschoven naar het object, dat de gegevens en de bewerkingen die hierop uitgevoerd dienen te worden (de functionaliteit), in zich combineert. Toepassing van objectoriëntatie leidt tot systemen die eenvoudiger aanpasbaar zijn aan veranderende eisen, eenvoudiger onderhoudbaar zijn, robuuster zijn en meer gebruik maken van hergebruik van code en ontwerp [Bahrami, 1999]. Er komen objectgeoriënteerde talen als Eiffel en C++ en objectoriëntatie dringt door in systemen, ontwikkelstrategieën, hulpprogramma's, notaties, methodologieën en zelfs algoritmes.

Iteratie is het tweede grote thema binnen deze periode. Iteratieve ideeën komen terug in softwarelevenscyclusmodellen als het Spiral model van Boehm, maar ook in applicaties

waarvan regelmatig nieuwe versies worden opgeleverd. Hergebruik van software is een belangrijk doel.

Een toenemende complexiteit, deels veroorzaakt door steeds grotere programma's, leidt er in toenemende mate toe dat procedureel programmeren minder goed voldoet. Objectoriëntatie vermindert dit probleem, doordat de basisconstructies uit een objectgeoriënteerde taal kunnen worden uitgebreid met objecten die sterk lijken op de conceptuele zaken uit de onderliggende applicatie. Hierdoor lijkt de taal de onderliggende applicatie rechtstreeks te ondersteunen, met als resultaat dat de softwareontwikkeling eenvoudiger, sneller en natuurlijker wordt [Bahrami, 1999]. Voorbeelden van objectgeoriënteerde talen zijn Simula uit de jaren zestig, Smalltalk uit de jaren zeventig, C++ en Eiffel uit de jaren tachtig.

Objecten combineren zowel de gegevens als de logica in de vorm van bewerkingen die op deze gegevens toegestaan zijn. Hiertoe beschikken ze over eigenschappen en methodes. De eigenschappen van een object beschrijven de toestand waarin dit object zich bevindt. Voorbeelden van eigenschappen van het object mens zijn naam, geboortedatum en geslacht. Het gedrag van een object wordt beschreven door de methodes waartoe het object in staat is. Voorbeelden van methodes van het object vliegtuig zijn 'start', 'stijg op', 'land', 'versnel', 'vertraag', 'stijg' en 'daal'.

Een object is een instantie van een klasse. De klasse is de sjabloon dat definieert over welke eigenschappen en methodes het object beschikt. Dit concept is te verduidelijken indien een parkeerplaats met auto's wordt bekeken. Alle auto's zijn objecten van de klasse auto. Elke afzonderlijke auto is te modelleren als een object met specifieke eigenschappen, bijvoorbeeld een rode auto met vier deuren en een trekhaak.

Objectgeoriënteerde programmeertalen beschikken over een viertal fundamentele eigenschappen: *information hiding*, *data abstraction*, *inheritance* en *polymorfisme*.

Information hiding is het principe dat alle interne eigenschappen en methodes van een object niet zichtbaar of toegankelijk zijn voor de buitenwereld. De eigenschappen die wel zichtbaar zijn, vormen de interface van het object. Recente programmeertalen hebben hiertoe de beschikking over 'access modifiers', die de toegang vastellen. Veelgebruikte vormen zijn 'public' voor algemene toegang, 'protected' indien uitsluitend objecten van de huidige klasse of klassen gebaseerd op deze huidige klasse toegang hebben en 'private' indien alleen het object zelf toegang heeft.

Data abstraction houdt in dat de interne gegevens van een object niet rechtstreeks te benaderen zijn, maar dat dit uitsluitend mogelijk is via de beschikbare eigenschappen en bewerkingen. Deze vormen samen de interface van het object. Hierdoor wordt de functionaliteit van het object (wát het object doet) gescheiden van de wijze waarop deze functionaliteit wordt geleverd (de implementatie, ofwel hóe het object dit doet). Het voordeel hiervan is tweezijdig. Enerzijds kan de implementatie van een object worden aangepast (bijvoorbeeld bij een optimalisatie) zonder dat applicaties waarin het object wordt gebruik aangepast hoeven te worden. Dit bevordert de onderhoudbaarheid van een systeem. Anderzijds kan de gebruiker van het object zich concentreren op de

functionaliteit, zonder zich te hoeven verdiepen in de wijze waarop deze functionaliteit wordt geleverd. Dit reduceert de complexiteit van het systeem.

Objecten kunnen worden gebaseerd op bestaande objecten. De klasse die dient als basis voor het nieuwe object wordt de basisklasse of superklasse genoemd. De klasse die hiervan wordt afgeleid heet de afgeleide klasse of subklasse. Een object van de afgeleide klasse overerft alle eigenschappen en methodes van de basisklasse. Dit houdt in, dat deze eigenschappen en methodes op het afgeleide object aan te roepen zijn, zonder dat ze opnieuw geprogrammeerd hoeven te worden. Een verzameling van elkaar afgeleide klassen wordt een klassen hiërarchie genoemd. Figuur 2 toont twee klassen. De basisklasse voertuig bezit de eigenschap snelheid en de methodes versnel en vertraag. De afgeleide klasse Auto beschikt automatisch over deze eigenschappen en methodes en kan zelf additionele eigenschappen en methodes definiëren, bijvoorbeeld respectievelijk aantal wielen en toeter.

Figuur 2: Overerving

Overerving leidt tot een verhoging van hergebruik en stimuleert de onderhoudbaarheid van de code, omdat functionaliteit slechts op één plaats (in een basisklasse) staat en niet in alle afgeleide klasse wordt gedupliceerd.

Doordat afgeleide klassen altijd tenminste over alle eigenschappen en methodes beschikken waar de basisklasse over beschikt, kunnen deze afgeleide objecten overal worden gebruikt waar objecten van de basisklasse worden verwacht. Omdat een specifieke implementatie van deze afgeleide klassen afwijkend kan zijn van de implementatie van de basisklasse, wordt gesproken over polymorfisme. Het voordeel van polymorfisme is dat elke klasse zijn eigen implementatie kan hebben, terwijl op al deze objecten dezelfde operatie kan worden aangeroepen. Dit resulteert in een beter onderhoudbaar systeem.

Resumerend kan worden gesteld dat objectgeoriënteerd programmeren leidt tot de volgende voordelen ten opzichte van procedureel programmeren [Bahrami, 1999].

- Het ontwikkelde systeem is door het hogere abstractieniveau minder complex dan de procedurele variant. De scheiding van interface en implementatie en de opdeling van het systeem in objecten, leiden ertoe dat het systeem beter onderhoudbaar is. Een bijkomend voordeel is dat objectoriëntatie in alle fasen van het ontwikkelproces kan worden toegepast. Hierdoor hoeft er bij de

overgang van de ene fase naar de andere geen ingrijpende vertaalslag plaats te vinden, wat voorheen wel het geval was.

- Reeds bestaande functionaliteit kan worden hergebruikt door gebruik te maken van overerving.
- Gebruik van goed geteste klassen voor overerving leidt tot een grotere betrouwbaarheid van het systeem.

Patterned programming tide

Het huidige getijde, dat is begonnen in 2000 en naar verwachting doorloopt tot ongeveer 2010, wordt gekenmerkt door een enorme toename van de aanwezigheid van computers in het dagelijks leven. Digitale agenda's en routeplanners in auto's zijn hiervan slechts enkele voorbeelden. Er wordt op steeds grotere schaal gebruik gemaakt van het internet via steeds snellere verbindingen. De afzetmarkt voor software wordt hierdoor alleen maar groter, leidend tot een grotere drang naar productiviteit op het gebied van systeemontwikkeling.

Inmiddels zijn de beperkingen van objecten bekend geworden en ligt de nadruk op het goed gebruiken van objecten en het vinden van oplossingen waar deze te kort schieten. Objecten alleen zijn onvoldoende om de beloften van meer hergebruik, betere onderhoudbaarheid en hogere kwaliteit waar te maken. De nadruk wordt gelegd op de middelste niveaus van systeemontwikkeling [Raccoon,1997]. De hoogste niveaus betreffen de programmastructuur als geheel. De laagste niveaus betreffen de individuele coderegels. Deze niveaus worden over het algemeen goed begrepen. Het zijn met name de niveaus er tussenin waar nog veel verbetering mogelijk is.

Er wordt veel gebruik gemaakt van raamwerken en sjablonen zodat objecten goed met elkaar kunnen samenwerken. Daarnaast worden patterns – conceptuele structuren - gebruikt op alle niveaus. Deze niveaus lopen van het ontwerpniveau, waar het probleem wordt gedefiniëerd, tot en met het implementatieniveau, waar de – door technologie mogelijk gemaakte – oplossing wordt bepaald. Optimalisatie bevindt zich niet langer op het niveau van functies, maar op het dynamisch selecteren van de meest efficiënte functie, gegeven de omstandigheden.

Conceptueel bestaat inmiddels het besef dat er sprake is van continue verandering en dat softwareontwikkeling hier goed mee om dient te gaan. Hiervan is de populariteit van Agile Development (zie Lightweight Modellen in sectie 2.3.3) een goed bewijs. Tevens heerst de opvatting dat er continue prioriteiten gesteld dienen te worden en dat de meest belangrijke zaken als eerste opgelost dienen te worden. De rol die de gebruiker hierbij speelt is gegroeid. Deze zorgt continu voor terugkoppeling in de vorm van aanvragen tot verbetering of verandering en verzoekt op correctie van geconstateerde fouten. Deze worden onmiddellijk opgenomen in de prioriteitsstelling. De gebruiker heeft nog steeds behoefte aan bruikbaarheid: indien het geleverde nut van een applicatie opweegt tegen de beperkingen, is de gebruiker tevreden.

Hulpmiddelen die bij het programmeren gebruikt worden zijn specifiek gericht op een aspect van het ontwikkelproces, zoals interface ontwerp of testen.

Gaandeweg werd duidelijk dat de voordelen van object-georiënteerd programmeren niet voldeden aan de heersende verwachtingen op het gebied van onderhoudbaarheid, hergebruik en betrouwbaarheid. Het modelleren van goede klassen bleek lastig, waardoor snelle oplevering niet altijd mogelijk was en wat tevens de onderhoudbaarheid van de software beïnvloedde [Florijn, 1996]. Individuele klassen waren te gedetailleerd en specifiek en moeten doorgaans meegecompileerd worden in een applicatie. Ook was voor het gebruik uitgebreide kennis van de klassen nodig, zodat doorgaans de broncode moest worden bekeken. [Sommerville, 2004] Ook bleek hergebruik niet eenvoudig te realiseren was, maar dat daarvoor naast technologische maatregelen ook organisatorische maatregelen nodig waren. [Florijn, 1996] Dit inzicht resulteerde in het ontstaan van een ander paradigma. Omdat de voordelen van OOP wel werden onderkend, is de oplossing gezocht in het vergroten van deze voordelen. Hierbij wordt in grote mate gebruik gemaakt van patterns. Een pattern is instructieve informatie die de essentiële structuur en het essentiële inzicht omvat van een bewezen succesvolle familie van oplossingen voor een terugkerend probleem dat optreedt binnen een zekere context en systeem van krachten.

De toepassing van software frameworks helpt objecten samen te werken. Een framework levert in het algemeen een generieke oplossing voor een bepaald probleem en kan in principe toegepast worden op alle niveaus van ontwikkeling. Software frameworks of kortweg frameworks zijn herbruikbare ontwerpen in de vorm van een verzameling abstracte klassen die op een bepaalde manier met elkaar samenwerken. Hierbij worden meestal enkele design patterns gebruikt. Om deze reden worden frameworks ook wel gezien als implementaties van systemen van design patterns. Een duidelijke definitie van een framework is de volgende.

> A framework is a set of cooperating classes that make up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

Definitie 1: Framework [Gamma et al., 1995]

Voorbeelden van frameworks zijn het .NET framework van Microsoft en het J2EE framework van Sun.

2.2.3 CBD

Eén van de ontwikkelingen binnen het 'patterned programming tide' is CBD. De beperkingen van objecten worden bij componenten verder ondervangen en de mogelijkheden van voorzieningen als frameworks hebben ervoor gezorgd componenten eenvoudiger te gebruiken zijn. De opkomende populariteit van gedistribueerde systemen [Sommerville, 2004] heeft geleid tot de acceptatie van CBD. Coulouris noemt de volgende voordelen van gedistribueerde systemen [Sommerville, 2004]:

1. Resource sharing – Resources (hardware en software) kunnen binnen het gedistribueerde netwerk worden gedeeld.
2. Openness – Door gebruik te maken van standaard protocollen, kunnen systemen van verschillende fabrikanten met elkaar samenwerken.
3. Concurrency – Processen kunnen op hetzelfde moment op verschillende computers op het netwerk draaien en gegevens uitwisselen over het netwerk.
4. Scalability – In principe kan de capaciteit van het systeem worden uitgebruid door hardware toe te voegen.
5. Fault tolerance – Vanwege de mogelijkheid dat processen tegelijkertijd op meerdere machines kunnen draaien, kan het uitvallen van een deel van de hardware worden opgevangen door de delen die wel blijven functioneren.

Naast deze voordelen noemt Coulouris ook nadelen van gedistribueerde systemen [Sommerville, 2004]:

1. Complexity – Gedistribueerde systemen zijn complexer dan gecentraliseerde systemen, wat het moeilijk maakt systeemgedrag te voorspellen.
2. Security – Omdat het systeem benadert wordt via meerdere computers en er gebruik wordt gemaakt van meer netwerkverkeer, is het lastiger te voorkomen dat gegevens worden gelezen of gewijzigd en is het systeem moeilijker te beschermen tegen denial-of-service[1] aanvallen.
3. Manageability – Doordat een gedistribueerd systeem kan bestaat uit verschillende typen computers, waarop verschillende besturingssystemen draaien, kan een fout op het ene systeem onverwachte resultaten veroorzaken op het andere systeem.
4. Unpredictability – Vanwege de sterk varierende belasting van het systeem, is de reactiesnelheid moeilijk te voorspellen.

Componenten worden gezien als een geschikt alternatief voor de implementatie van gedistribueerde systemen, zoals blijkt uit het volgende citaat.

[1] Een denial-of-service (DOS) aanval houdt in, dat een specifieke service met een enorme frequentie wordt benaderd, waardoor de service overbelast raakt en crasht.

> "Het denken in en werken met van componenten is een voorwaarde voor de ontwikkeling van gedistribueerde systemen."

Citaat 1: Componenten als voorwaarde voor gedistribueerde systemen [Florijn, 1996]

Nu duidelijk is waarom CBD is ontstaan, wordt het tijd een aantal begrippen te verduidelijken. Hierbij beginnen we gelijk bij het centrale begrip component. We bekijken eerst een aantal definities van het begrip software component.

> A *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Definitie 2: Software component [Szyperski, 1997]

Een component heeft vastgelegde interfaces en expliciet gemaakte afhankelijkheden (eisen aan de omgeving). Een component kan onafhankelijk worden ingezet en samengesteld door derden. Een andere definitie is de volgende.

> A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

Definitie 3: Software component [Heineman et al., 2001]

Deze definitie relateert een component aan de begrippen component model en composition standard. Hieronder wordt het volgende verstaan.

> A *component model* defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.

Definitie 4: Component model [Heineman et al., 2001]

Een component model definieert een aantal standaarden met betrekking tot interactie en compositie. Hierdoor kunnen componenten die aan dit model conformeren met elkaar samenwerken. Voorbeelden van component modellen zijn het .Net Component Model van Microsoft, het CORBA Component Model van de Object Management Group (OMG) en het Java Bean Component Model van Sun.

Een component model implementation zorgt ervoor dat componenten die aan het bijbehorende component model conformeren, uitgevoerd kunnen worden. Een voorbeeld is het .Net Framework van Microsoft voor .Net componenten.

> A *software component infrastructure* is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

Definitie 5: Software component infrastructure [Heineman et al., 2001]

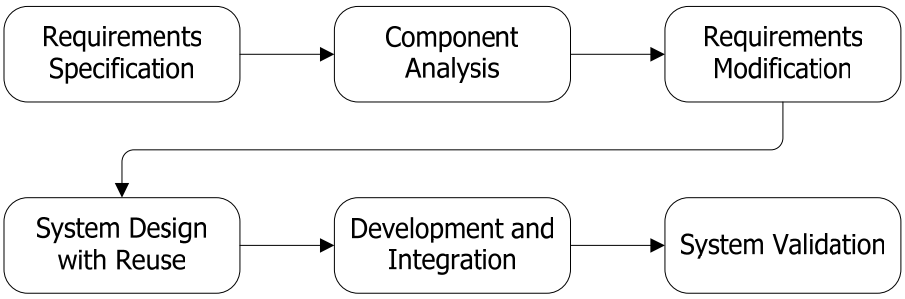
Een laatste definitie is de volgende.

> A component is a software unit whose functionalities and dependencies are completely defined by a set of public interfaces. Components can be combined with other components without reference to their implementation and can be deployed as an executable unit.

Definitie 6: Component [Sommerville, 2004]

Samenvattend is een software component waarvan de functionaliteit en de afhankelijkheden zijn vastgelegd in diens interfaces, die gebruikt kan worden zonder de details van de implementatie te kennen, die geassembleerd kan worden samen met andere componenten en die onafhankelijk ingezet kan worden. In het vervolg van deze scriptie zullen we deze definitie hanteren.

Nu duidelijk is wat onder component moet worden verstaan, is het tijd te verduidelijken wat onder het begrip CBD wordt verstaan. Bij CBD wordt een systeem, indien mogelijk, samengesteld uit reeds bestaande componenten, welke niet specifiek voor dit systeem zijn ontwikkeld en generiek van aard zijn. Vooruitlopend op de levenscyclusmodellen die in 2.3 worden beschreven, wordt het proces dat hierbij wordt gevolgd weergegeven in Figuur 3.



Figuur 3: Proces CBD, aangepast van [Sommerville, 2004]

Bij de stap *Requirements Specification* worden – net als bij conventionele ontwikkeling – de systeemeisen vastgesteld. Deze worden echter niet zo gedetailleerd uitgewerkt. Op basis van deze eisen wordt in de stap *Component Analysis* gezocht naar componenten die voldoen aan de opgestelde eisen. Op dit zoeken komen we later terug. Doorgaans voldoen de gevonden componenten niet volledig aan de gestelde eisen. Hierop worden in de volgende stap, *Requirements Modification*, gekeken of de eisen aangepast kunnen worden aan de gevonden componenten. Een reden om de eisen aan te passen is bijvoorbeeld dat er een component is gevonden dat niet geheel aan de eisen voldoet. Gebruik van dit component kan de ontwikkeling echter aanzienlijk versnellen, zodat de opdrachtgever mogelijk liever dit alternatief kiest dan dat hij het niet gevonden component laat bouwen. Als er geen compromis bereikt kan worden, kan opnieuw worden gezocht naar componenten. Nadat de eisen aangepast zijn, kan in stap *System Design with Reuse* een framework van het systeem worden ontworpen of hergebruikt, waarbij rekening wordt gehouden met de geselecteerde componenten. Indien voor bepaalde functionaliteit geen geschikte componenten zijn gevonden, kan het ontwerp van deze componenten worden gemaakt. Ook is het mogelijk dat bepaalde gevonden componenten hier toch niet blijken te voldoen, waardoor wordt teruggegaan naar de stap Component Analysis. In de volgende stap, *Development and Integration*, worden de ontworpen componenten – waar geen geschikte alternatieven voor bestonden – ontwikkeld en samen met de gevonden interne en externe componenten geïntegreerd tot een systeem. Hierbij wordt vaak gebruik gemaakt van 'glue code' om verschillen in verwachte en geleverde interfaces op te lossen. Tot slot wordt in stap *System Validation* gecontroleerd dat het systeem voldoet aan de eisen.

We moeten nog even terugkomen op de stap Component Analysis. Deze stap omvat het zoeken naar componenten die voldoen aan de gestelde eisen. Hierbij wordt doorgaans eerst binnen de ontwikkelorganisatie gezocht of er een geschikte kandidaat is. Zo niet, wordt er pas extern gezocht. Een belangrijke overweging bij dit externe zoeken zijn de risico's die zijn verbonden aan de aanschaf van externe componenten, zoals kwaliteit, huidige en toekomstige ondersteuning en prijsontwikkeling voor toekomstige versies. Indien er geen geschikt extern component gevonden kan worden, dan pas wordt het component intern ontwikkeld. Alleen indien de verwachting bestaat dat het component vaker toegepast kan worden, wordt het als generiek component ontworpen en gebouwd, vanwege de hogere complexiteit en kosten die dit met zich meebrengt. Anders wordt er een specifiek component van gemaakt. Deze wijsheid [Allen, 2001] staat ook wel bekend als:

> "Reuse before you buy before you build."

Citaat 2: CBD wijsheid [bron onbekend]

2.3 Softwarelevenscyclusmodellen

Softwareontwikkeling betreft het proces dat leidt tot de productie van software. Binnen dit proces doorloopt het product software verschillende fasen, zodat gesproken kan worden van de levenscyclus van software, zoals in de volgende definitie duidelijk wordt.

> "The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirements phase, test phase, installation and check-out phase, operation and maintenance phase, and sometimes, retirement phase."

Definitie 7: Software Engineering [IEEE Standard Glossary of Software Engineering]

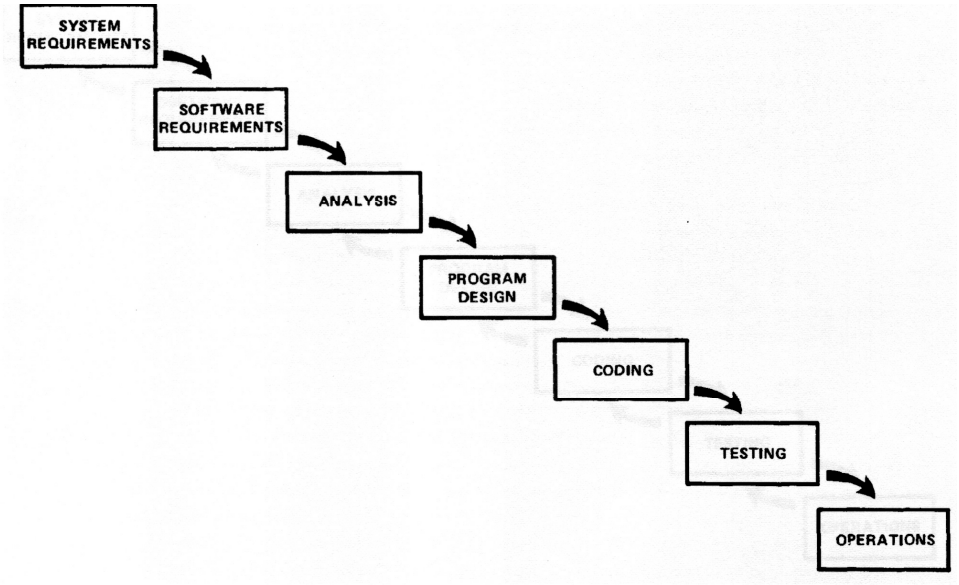
Modellen die het proces van softwareontwikkeling beschrijven worden om die reden ook wel software levenscyclusmodellen of softwareontwikkeling levenscyclusmodellen genoemd (in het Engels respectievelijk software life cycle models of software development life cycle models (SDLM)).

Een SDLM geeft dus aan hoe het ontwikkeltraject van software eruit ziet. Een model doorloopt hierbij alle fasen van de aanvang van een softwareproject tot en met de oplevering en ondersteuning en eventueel de buitengebruikstelling. Er zijn verschillende typen levenscyclus modellen, met verschillende kenmerken. Afhankelijk van de kenmerken van het softwareproject is het ene model meer geschikt dan het andere model.

2.3.1 Waterfall model

Een van de meest bekende modellen is het Waterfall model van Winston Royce uit 1970. De fasen van dit model zijn vastgesteld op basis van de verschillende activiteiten van het ontwikkelproces. Alle fasen worden in de originele vorm van het model in een strikte volgorde – sequentieel – doorlopen, waarbij een volgende fase pas wordt gestart nadat de vorige is afgerond.

De originele vorm van het model bestaat uit de fasen 'system requirements', 'software requirements', 'analysis', 'planning', 'program design', 'coding', 'testing' en 'operations'.



Figuur 4: Waterfall model [Royce, 1970]

In de fase 'System requirements' wordt door enkele leden van het ontwikkelteam overlegd met de klant en met de toekomstige gebruikers. Hierbij worden de systeemeisen zo nauwkeurig mogelijk expliciet gemaakt. Zo nodig worden hierbij bestaande systemen doorgelicht inclusief hun documentatie en worden interviews gehouden. In de fase 'software requirements' worden op basis van de verzamelde systeemeisen de softwarematige eisen vastgesteld. In de analysefase worden de eisen uit de voorgaande fasen vertaald in een specificatie die aangeeft wat het systeem precies doet (en wat niet). Deze specificatie dient – evenals de voorgaande documenten – volledig te zijn. Ook mag ze geen tegenstellingen en onduidelijkheden bevatten. Bij het plannen worden de benodigde activiteiten, om tot realisatie van de specificatie te komen, ingepland op basis van de beschikbare capaciteit. Zodra deze planning is afgerond wordt het ontwerp gemaakt. Nadat het ontwerk is goedgekeurd wordt het gecodeerd. Na oplevering van de programmacode, inclusief de bijbehorende documentatie, wordt het systeem getest en worden gevonden problemen herleid tot de veroorzaker. De fase waarin geconstateerde defecten zijn ontstaan, moet opnieuw (deels) worden uitgevoerd, met alle opvolgende fasen. Zodra het testen is afgerond kan het systeem in gebruik worden genomen.

Het model kenmerkt zich door het nastreven van volledigheid en juistheid. Hiervoor wordt aan het einde van elke fase een zorgvuldige controle uitgevoerd. Tevens is het model sterk documentgebaseerd: in iedere fase maakt de documentatie deel uit van het eindproduct van die fase. Deze documentatie wordt dus ook in iedere fase onderworpen aan de controle. Pas nadat het product van de huidige fase is goedgekeurd, wordt deze fase afgesloten en kan een begin worden gemaakt met de volgende fase.

Het model is gebaseerd op de aannames dat de eisen bekend en stabiel zijn en geen grote risico's met zich mee brengen, men bekend is met het probleemgebied en er voldoende tijd is om de ontwikkeling sequentieel uit te voeren. [Boehm, Port, 1999]

In de loop der jaren is er op een aantal punten kritiek geuit op het Waterfall model.

- Allereerst is het model sterk gebaseerd op een situatie waarin alle informatie bekend is gedurende de uitvoering van een fase. De systeemeisen van het volledige systeem zijn echter zelden bekend op het moment dat deze opgeleverd worden. [Jacobson, 1993]
- Aangezien het inzicht in de volledigheid van een fase pas bij het uitvoeren van latere fasen duidelijk wordt, is de voortgang van de ontwikkeling zeer moeilijk vast te stellen.
- Tussentijdse wijzigingen van het systeem kunnen tot gevolg hebben dat voorgaande fasen (deels) opnieuw uitgevoerd moeten worden, wat doorgaans leidt tot aanzienlijke vertraging.
- Klanten kunnen pas na oplevering terugkoppelen of het systeem voldoet aan hun verwachtingen. [Schach, 1992]

2.3.2 Iteratief model

Indien de kritiek op het Waterfall model van Royce bekeken wordt in samenhang met de aannames die aan het model ten grondslag liggen, blijkt dat het model in de praktijk veelal in situaties wordt toegepast waar niet aan deze aannames is voldaan. Royce gaf overigens al in hetzelfde artikel, waarin hij het oorspronkelijke model introduceerde, aan dat er ontwikkelrisico's zijn die door de volgende vijf toevoegingen aan het model grotendeels kunnen worden geëlimineerd [Royce, 1970]:

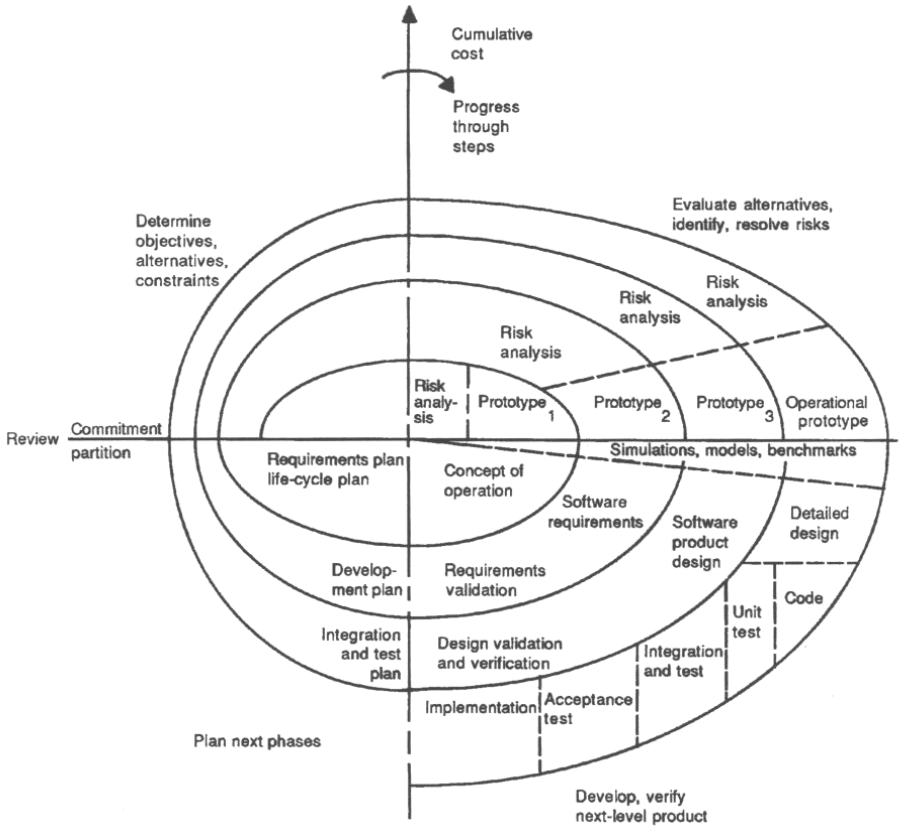
1. Voer een voorlopige ontwerpslag uit voordat de analysefase wordt begonnen, om te voorkomen dat in de analysefase niet-functionele eisen, zoals de benodigde resources, buiten beschouwing worden gelaten.
2. Zorg voor een uitvoerige documentatie van het ontwerp, zodat de opvolgende fasen hiervan gebruik kunnen maken.
3. Doorloop de fasen van voorlopig ontwerp tot gebruik tenminste één keer extra, voordat de eerste versie wordt opgeleverd, in de vorm van een pilot. Hierdoor wordt tenminste één mogelijkheid geleverd om de belangrijkste aspecten van het systeem met nauwkeurigheid te verifiëren aan de hand van een concrete, werkende versie in plaats van een ontwerp.
4. Leg extra nadruk op het testen door het testen over te laten aan specialisten, analyse en code visueel te controleren, te proberen alle logische paden te controleren en het testen geautomatiseerd uit te voeren.
5. Betrek de gebruiker vroegtijdig in de ontwikkeling.

Het verbeterde Waterfall model dat zo ontstaat, bevat al eigenschappen van iteratieve ontwikkeling [Larman, Basili, 2003]. Dit is niet verwonderlijk, omdat de oudste specifieke beschrijving van iteratieve ontwikkeling uit 1968 stamt en kenmerken van iteratieve ontwikkeling al een aantal jaren eerder zichtbaar waren [Larman, Basili, 2003]. In de

zeventiger jaren zijn onder andere door IBM diverse grote applicaties ontwikkeld waarbij gebruik werd gemaakt van iteratieve en incrementele methoden. Eind jaren tachtig realiseert men zich dat traditionele modellen – zoals het Waterfall model in zijn originele vorm – niet geschikt zijn voor effectievere benaderingen zoals prototyping en hergebruik van software [Boehm, 1988]. In 1988 introduceert Boehm het Spiral Model, dat is geëvolueerd uit ervaringen met verfijningen van het Waterfall model, toegepast op grote overheidsprojecten. Kenmerkend van dit model is de risicogebaseerde benadering. Hierbij worden de volgende stappen doorlopen.

- Identificatie van doelen (functionele- en niet-functionele-), alternatieven (verschillende ontwerpen, hergebruik of aanschaf) en restricties (kosten, tijd, interface).
- Evaluatie van alternatieven op basis van risico-analyse m.b.v. diverse technieken, waaronder prototyping.
- Doorgaan met elimineren van risico's totdat deze allemaal acceptabel zijn.
- Ontwikkelen en verifiëren van volgende niveau product.

Voorafgaand aan de uitvoering van de spiral, wordt een hypothese opgesteld die stelt dat een specifieke operationele missie (of verzameling missies) verbeterd kan worden door toepassing van software. Indien deze hypothese gefalsificeerd kan worden, wordt de spiral niet (of niet verder) uitgevoerd [Boehm, 1988]. Een overzicht van het model wordt weergegeven in Figuur 5.



Figuur 5: Spiral Model [Boehm, 1988]

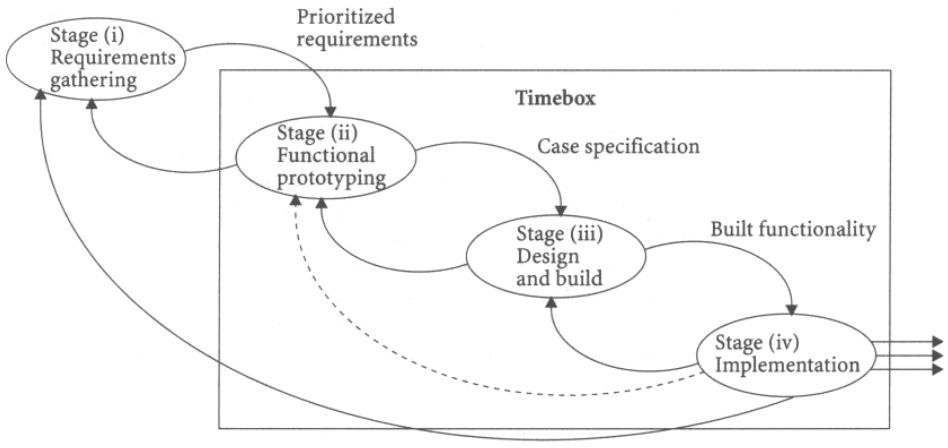
Dit model kent de volgende voordelen:

- Gedetailleerde specificaties worden pas opgesteld nadat de elementen van het ontwerp met een hoog risico gestabiliseerd zijn.
- Prototyping kan in elke fase worden toegepast voor de reductie van risico's.
- Detail is alleen dan noodzakelijk wanneer de afwezigheid ervan het project in gevaar brengt.
- Er wordt gekeken naar de mogelijkheid om bestaande software te hergebruiken.

Naast deze voordelen worden er ook een aantal nadelen onderkend:

- Het is vooral geschikt voor interne projecten vanwege de flexibiliteit hiervan. Bij softwarecontractacquisitie is deze flexibiliteit veelal beperkt.
- Het is afhankelijk van expertise op het gebied van risico-assessments.
- De modelstappen moeten verder worden uitgewerkt.

Ten gevolge van de steeds grotere dynamiek in het bedrijfsleven blijft de behoefte aan de snelle oplevering van systemen stijgen. In 1991 werd door James Martin Rapid Application Development (RAD) geïntroduceerd. Deze wijze van ontwikkeling is gebaseerd op de incrementele oplevering van functionaliteit, waarbij de meest belangrijke functionaliteit het eerst wordt opgeleverd en additionele functionaliteit pas later. De levenscyclus van RAD wordt beschreven door de fasen: 'initiation and viability', 'requirements gathering', 'functional prototyping', 'design and build' en 'implementation', waarvan de laatste vier worden weergegeven in de volgende figuur.



Figuur 6: RAD [Skidmore, Eva, 2004]

Tijdens 'initiation and viability' wordt bekeken of het project haalbaar is. Daarna worden in de fase 'requirements gathering' de systeemeisen op hoog niveau vastgesteld en wordt het systeem opgedeeld in afgebakende delen functionaliteit. Daarna worden er prioriteiten toegekend aan deze functionele delen. In de volgorde van deze prioriteiten worden daarna per functioneel deel van het systeem de overige drie fasen in een timebox uitgevoerd.

Timeboxing is een management techniek waarbij een variabele, geprioriteerde hoeveelheid werk wordt uitgevoerd in een vaste hoeveelheid tijd, welke timebox wordt genoemd. Indien niet al het werk binnen deze timebox uitgevoerd kan worden, vervalt het deel van het werk met de laagste prioriteit. Dit verschil in prioriteiten is essentieel, omdat er geen werk met de laagste prioriteit vastgesteld kan worden als ze allemaal dezelfde prioriteit zouden hebben.

Bij de vaststelling van de prioriteiten wordt vaak gebruikgemaakt van het MoSCoW-principe. Hierbij staat het acroniem voor de verschillende groepen prioriteiten:

- Must do – essentieel voor het slagen van het project
- Should do – belangrijk, voegt waarde toe aan het project
- Could do – individuele wens, voegt weinig waarde toe aan project
- Won't do – voegt niets toe aan project

Kader 1: Timeboxing

Tijdens de fase 'functional prototyping' worden de lower-level requirements verzameld, bestaande uit de functionele en niet-functionele requirements. Hierbij wordt gebruik gemaakt van verschillende prototypes, die door de eindgebruiker worden geëvalueerd en op grond van zijn feedback worden aangepast totdat de requirements duidelijk zijn. Van de eindgebruiker wordt fulltime medewerking verwacht. In deze fase worden een gedetailleerde verzameling prototypes van de requirements, een verzameling niet-functionele requirements, review documenten en een planning voor de design and build-fase opgeleverd.

Gedurende 'design and build' worden de requirements, prototypes en documentatie vertaald in een ontwerp. Hierbij worden bij voorkeur met behulp van CASE tools – geautomatiseerde ondersteuning voor softwareontwikkeling – databasetabellen en programmacode gegenereerd. Vervolgens wordt het ontwerp getest en gebouwd.

In de fase 'implementation' wordt de documentatie opgeleverd en worden de gebruikers getraind. De gegevens worden aangemaakt of geconverteerd en het systeem wordt overgedragen.

De naamgeving voor iteratieve modellen is niet consequent. Zo worden ook de benamingen incrementeel, spiral, evolutionary en jaccuzi gebruikt [Fowler, 2003]. Ook wordt regelmatig de benaming Iterative & Incremental Development (IID) gebruikt. Feitelijk zijn deze termen verschillend, waarbij iteratief slaat op het proces dat herhaaldelijk dezelfde (reeks) activiteiten uitvoert en incrementeel op het product dat geleidelijk aan wordt verbeterd door wijzigingen en toevoegingen [Nørbjerg, 2002].

Kenmerkend van iteratieve modellen is:

- de afwezigheid van uitvoerige specificatie vooraf;
- meerdere iteraties met uitbreidingen en/of verbeteringen bestaande uit afgebakende functionaliteit, waarbij het eindproduct van productiekwaliteit is;
- gebruik van een volledige cyclus van analyse, ontwerp, coderen en testen binnen iteraties, vaak gebruikmakend van time-boxing;
- feedback van gebruikers.

Een gevolg van IID is, dat onderdelen van systemen onafhankelijk van elkaar worden opgeleverd. Omdat deze onderdelen naadloos met elkaar moeten samenwerken, worden er bij IID extra eisen gesteld aan goed gedefiniëerde interfaces.

2.3.3 Lightweight Modellen

De tot nu toe behandelde modellen worden als traditionele modellen aangeduid en worden vaak gezien als log, bureaucratisch en ongeschikt voor het snelle tempo van veel projecten [Wolak, 2001]. Vanwege deze eigenschappen worden ze ook ‘heavy methodologies’ genoemd [Fowler, 2000]. In een reactie hierop is een nieuwe generatie ontwikkelmethodologieën ontstaan, die een balans bieden tussen geen proces en teveel proces [Fowler, 2000]. Deze generatie modellen wordt omdat ze een alternatief bieden voor de heavy methodologieën aangeduid als ‘lightweight’ [Fowler, 2000] en later als ‘agile’ [Fowler, april 2003].

Deze typering kan worden geillustreerd aan de hand van het Agile Manifesto. Dit manifest is ontwikkeld tijdens een bijeenkomst van een aantal ‘industry leaders’ op het gebied van softwareontwikkeling en bevat het volgende:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> **Individuals and interactions** over processes and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

Agile Manifesto [Agile Alliance, 2001]

Onder Agile Development vallen een aantal verschillende ontwikkelmethoden, waaronder[2] Adaptive Software Development (ASD), Crystal, Dynamic Systems Development Method (DSDM), eXtreme Programming (XP), Feature Driven Development (FDD) en Scrum. Deze worden hieronder kort besproken.

ASD

Binnen ASD wordt gerealiseerd dat verandering altijd voorkomt. In plaats van verandering te negeren wordt verandering hier omarmd door dit goed te managen. Er worden drie overlappende componenten onderscheiden, te weten [Fowler, april 2003][Highsmith, 2002]:

- speculation, waarin rekening wordt gehouden met de onzekerheid van het plannen;
- collaboration, waarbij samengewerkt wordt door verschillende teamleden om alle kennis en ervaring samen te voegen;
- learning, waarin de kennis wordt getoetst en reviews plaatsvinden.

Er wordt gebruik gemaakt van iteratieve ontwikkeling, planning op basis van features, nauwe betrokkenheid van de klant en een intensieve samenwerking met het management [Conn, 2004].

Crystal

De Crystal methode is ontwikkeld na onderzoek hoe softwareontwikkeling in de praktijk werkt in tegenstelling tot hoe het in theorie zou moeten werken. Crystal bestaat feitelijk

[2] Bij deze selectie heb ik me beperkt tot de ontwikkelmethoden die zowel worden genoemd in [Fowler, april 2003] als op de website van Agile Alliance, een bekende non-profit organisatie die het gebruik van deze methoden ondersteunt, welke te vinden is op http://www.agilealliance.com/.

uit een familie methoden waarvan de keuze afhankelijk is van twee variabelen: het aantal medewerkers aan een project en de impact van defecten in een project [Fowler, april 2003]. Iedere deelnemer aan het ontwikkelproces krijgt taken toegekend op basis van zijn of haar capaciteiten en talenten [Conn, 2004].

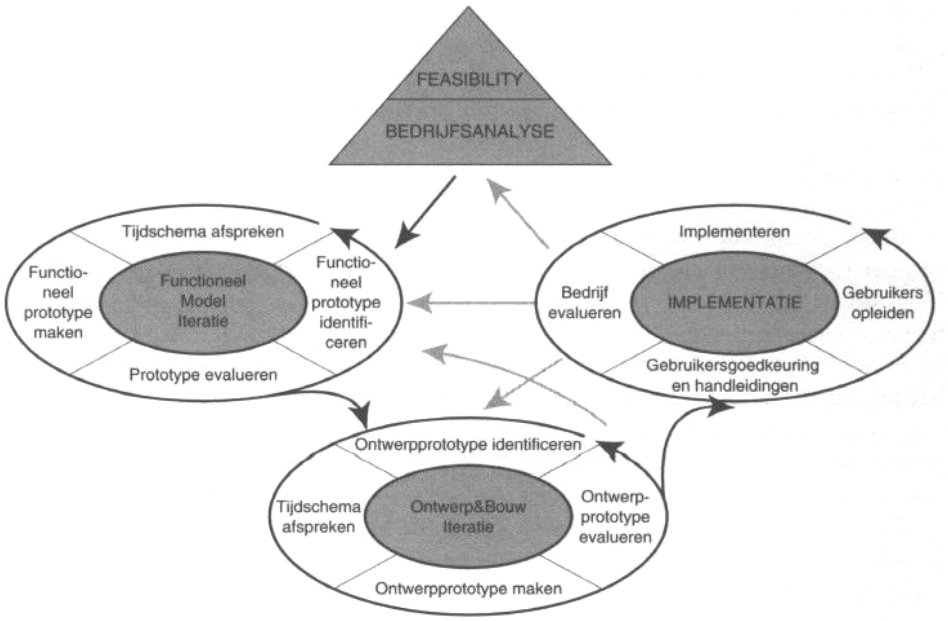
DSDM

DSDM is in 1994 in Groot-Brittannië opgericht en is een raamwerk van sturingsmechanismen bij RAD, aangevuld met richtlijnen. Het definieert een proces met een aantal producten op globaal niveau, welke kunnen worden aangepast aan iedere techniek en bedrijfsomgeving. [Stapleton, 1999] Het is volledig gedocumenteerd en voorziet in trainingen en certificeringen.

DSDM is gebaseerd op de volgende negen principes [Stapleton, 1999].

1. Actieve betrokkenheid van gebruikers is noodzakelijk.
2. DSDM-teams moeten gemachtigd zijn besluiten te nemen.
3. Frequente oplevering van producten is van wezenlijk belang.
4. Geschiktheid van bedrijfsdoeleinden is het essentiële criterium voor de acceptatie van producten.
5. Iteratieve en incrementele ontwikkeling is noodzakelijk om te convergeren tot een juiste bedrijfsoplossing.
6. Alle wijzigingen tijdens de ontwikkeling zijn terug te draaien.
7. Eisen worden op hoog niveau vastgelegd.
8. Testen is geïntegreerd in de levenscyclus.
9. Een samenwerkende en coöperatieve houding van alle belanghebbenden is essentieel.

De fasen die bij DSDM worden onderscheiden zijn: Haalbaarheidsonderzoek, Bedrijfsanalyse, Functioneel Model Iteratie, Ontwerp & Bouw Iteratie en Implementatie. De eerste twee fasen worden sequentieel uitgevoerd, de overige drie fasen zijn iteratief, zoals te zien is in de volgende figuur. Bij de iteraties wordt gebruik gemaakt van time-boxing met een prioritering van eisen.



Figuur 7: DSDM Proces [Stapleton, 1997]

FDD

FDD is een minimalistische aanpak van het ontwikkelproces in vijf stappen [Conn, 2004]. De eerste drie stappen zijn het bouwen van een algemeen object model, het opstellen van een lijst van gewenste eigenschappen en het maken van een planning per eigenschap, welke aan het begin van een project worden uitgevoerd. De volgende twee stappen zijn het maken van een ontwerp per eigenschap en het bouwen per eigenschap, welke iteratief worden uitgevoerd [Fowler, april, 2003]. Elke stap in dit proces is kort en nauwkeurig gedocumenteerd [Conn, 2004] .

Scrum

Scrum is een projectmanagement framework waarbinnen softwareontwikkeling plaatsvindt in iteraties van dertig dagen. Deze iteraties worden 'scrums' genoemd en hierbinnen wordt gebruik gemaakt van time-boxing. Eén van de belangrijkste kenmerken van Scrum is dat er dagelijks een kwartier is ingedeeld waarbinnen het ontwikkelteam overlegd teneinde de coördinatie en integratie te garanderen [Conn, 2004].

XP

XP is de meest populaire agile methode [Conn, 2004], uitgevonden door Kent Beck. Het is gebaseerd op de waarden moed, eenvoud, terugkoppeling en gemeenschap. XP is ontworpen voor gebruik in kleine teams die software ontwikkelen onder grote tijdsdruk in een dynamische omgeving met snel veranderende requirements. Daarnaast gaat XP om met projectrisico's, zoals een vaste opleverdatum, een nieuwe uitdaging voor het ontwikkelteam of een nog nooit toegepaste oplossing in de software-industrie, door gebruik te maken van de volgende 'key practices'.

- Het planning process, waarin de door de klant bepaalde bedrijfswaarde van gewenste features wordt afgezet tegen de kostenschattingen van de programmeurs om te bepalen wat prioriteit heeft.
- Kleine releases om het systeem snel in productie te zetten en frequent te verbeteren.
- Metaforen voor het vergemakkelijken van de ontwikkeling en communicatie.
- Het meest eenvoudige ontwerp dat voldoet aan de huidige eisen.
- Test first: eerst schrijven van testen die functionele eisen representeren, daarna pas de software die aan deze testen voldoet. Op dezelfde wijze leveren klanten acceptatietesten aan.
- Continue verbetering van het systeemontwerp door toepassing van refactoring (zie Kader 2).
- Pair programming, waarbij alle code wordt geschreven door twee programmeurs tegelijk, zodat er geen code review plaats hoeft te vinden.
- Collectief bezit van de code, zodat deze door iedereen gewijzigd mag worden zonder overleg vooraf.
- Continue integratie, waarbij de code meerdere keren per dag wordt geïntegreerd en gebuild, zodat alle programmeurs bij blijven en er snelle voortgang gemaakt kan worden.
- Geen excessief overwerk, om te voorkomen dat vermoeide programmeurs fouten maken.
- Een klant op locatie die requirements vaststelt, prioriteiten en vragen van programmeurs beantwoordt, met als gevolg een verbetering van de communicatie en reductie van papieren documenten.
- Standaard wijze van coderen, zodat de code van elke programmeur begrijpbaar is voor elke andere.

Refactoring is een gedisciplineerde techniek die bestaande software herstructureert, waarbij de interne structuur wordt aangepast zonder de functionaliteit aan te passen. Dit wordt gerealiseerd door het toepassing van reeksen kleine veranderingen, bijvoorbeeld de eliminatie van duplicate code door deze in een functie te zetten en vanaf de oorspronkelijke locaties deze functie aan te roepen. Na iedere reeks van kleine veranderingen wordt gecontroleerd dat de software nog correct functioneert. Refactoring leidt tot duidelijker en beter onderhoudbare code. [Fowler et al., 1999]

Kader 2: Refactoring [Fowler et al., 1999]

Naast genoemde methoden, wordt het Rational Unified Process (RUP) regelmatig genoemd als agile methode, terwijl dit in feite niet zo is [Fowler, april 2003]. Het RUP is een process framework dat een grote variëteit aan processen ondersteunt. Processen die ingericht zijn conform dit framework leveren functionaliteit in kleine increments en zijn gebaseerd op use cases. Deze processen zijn niet per definitie agile, maar kunnen wel zo ingericht worden. Een voorbeeld hiervan is het dX proces, dat identiek is aan XP [Booch et al., 1998]. RUP maakt gebruik van de modelleertaal Unified Modeling Language (UML), waarin ontwerpen gemodelleerd kunnen worden.

De agile methoden hebben de volgende kenmerken met elkaar gemeen [Nørbjerg, 2002].

- Vroegtijdige vaststelling van de centrale systeemeigenschappen door middel van prototyping.
- Herhaalde, korte, ontwikkelcycli met als doel snelle oplevering van een werkend systeem.
- Het eindproduct wordt opgeleverd in een reeks los van elkaar vrijgegeven deelproducten (increments).
- De eisen evolueren gedurende de ontwikkeling.

Vanwege het feit dat de agile methoden nog zo jong zijn, is het moeilijk vast te stellen wat de langetermijn-gevolgen ervan zijn.

2.4 Conclusie

Softwareontwikkeling heeft in zijn relatief korte bestaansperiode een flinke voortgang meegemaakt. Op verschillende gebieden zijn er grote vorderingen geweest, die allemaal hun nut hadden en nog steeds hebben.

De automatiseringsgraad neemt toe. Programma's en systemen worden steeds groter en complexer. Gebruikers worden kritischer en stellen hogere eisen aan betrouwbaarheid. Anderzijds leidt de vraag naar snelle oplevering tot een hoge tijdsdruk. Er is grote behoefte aan productiviteitsgroei binnen de softwareontwikkeling.

Software wordt tegenwoordig veelal ontwikkeld met gebruikmaking van iteratieve ontwikkelmodellen, waarbij de klant intensief wordt betrokken. Verder wordt er veel gebruik gemaakt van object-oriëntatie, patterns, frameworks en componenten. Deze wijze van ontwikkeling wordt geschikt geacht om tegemoet te komen aan de eisen die er vandaag de dag aan worden gesteld.

Na dit historische overzicht van de ontwikkeling van software, kijken we in het volgende hoofdstuk naar de kwaliteit van software.

3 Softwarekwaliteit

3.1 Inleiding

In hoofdstuk 1 is gesteld dat de betrouwbaarheid van software in toenemende mate van belang is. De betrouwbaarheid van software is slechts één van de aspecten van de kwaliteit van software. Het huidige hoofdstuk gaat in op deze kwaliteit.

Paragraaf 3.2 licht het belang van de kwaliteit van software toe. Om te kunnen komen tot uitspraken over de kwaliteit van software en de beïnvloeding hiervan, zal duidelijk moeten zijn wat onder softwarekwaliteit wordt verstaan. In paragraaf 3.3 worden de heersende opvattingen over de definitie van softwarekwaliteit op een rij gezet. Vervolgens gaat paragraaf 3.4 in op de bestaande technieken om de gewenste softwarekwaliteit binnen het softwareontwikkelingsproces te realiseren. Paragraaf 3.5 behandelt vervolgens de kwaliteitsevaluatie, welke buiten het softwareontwikkelings-proces plaatsvindt. Hiertoe worden enkele kwaliteitsmodellen voor software geïntroduceerd. In paragraaf 3.6 worden de belangrijkste bevindingen uit dit hoofdstuk kort samengevat.

3.2 Belang

Voordat ingegaan wordt op de precieze betekenis van het begrip softwarekwaliteit is het goed om vast te stellen waarom softwarekwaliteit belangrijk is.

In de beginperiode waarin voornamelijk met mainframes werd gewerkt, was er sprake van relatief dure hardware met beperkte verwerkingscapaciteit. Programma's werden na elkaar uitgevoerd en er ontstonden wachtrijen, waarbij programma's overdag werden aangeleverd en 's nachts werden uitgevoerd. Fouten in de software werden daarom vaak pas 's ochtend ontdekt en opgelost, waarna het aangepast programma weer werd aangeleverd en pas de volgende nacht werd uitgevoerd. Kwalitatief slechte software kon dus tot grote vertragingen leiden.

In de afgelopen jaren is de maatschappij steeds afhankelijker geworden van computersystemen. Zo zijn vrijwel alle telecommunicatienetwerken afhankelijk van computersystemen waarop ze draaien, hebben de meeste bedrijven veel van hun systemen geautomatiseerd en zijn aandelenbeurzen vrijwel geheel afhankelijk van software. Daarnaast is er ook apparatuur die niet direct wordt geassocieerd met computers, maar waar wel software in verwerkt zit. Men spreekt hier over 'embedded software'. Voorbeelden hiervan zijn mobiele telefoons, digitale agenda's, mp3-spelers, televisietoestellen en navigatiesystemen in auto's en in toenemende mate onderdelen van auto's zoals het remsysteem en het motormanagement.

Indien de hedendaagse software van slechte kwaliteit is, kunnen de betreffende systemen minder goed of zelfs helemaal niet functioneren. De gevolgen hiervan zijn vooraf moeilijk of helemaal niet in te schatten. Humprey noemt naast minder ernstige gevolgen als ongemak, inefficiëntie en ergernis ook ernstiger gevolgen als het negatief

beïnvloeden van de winstgevendheid, grote economische ontwrichtingen en de dood [Humprey, 1999].

Helaas zijn er legio voorbeelden van deze gevolgen. Tussen 1985 en 1987 stierven drie patiënten aan de gevolgen van een overdosis straling die ze ontvingen tijdens hun bestralingstherapie met de Therac 25 machine. Een fout in de software leidde ertoe dat – door het invoeren van een specifieke reeks instructies – de intensiteit van de toegediende straling een factor 100 te hoog was. Enkele andere patiënten hielden blijvende verwondingen over aan de overdosis straling [Mellor, 1992]. In 1991 leidde een fout in de software ertoe dat een raket van een Patriot luchtafweerinstallatie een SCUD raket niet onderschepte. Deze laatste raakte een Amerikaanse basis waarbij 28 militairen om het leven kwamen. Ten gevolge van de stroomuitval in het noordoosten van de Verenigde Staten en Canada in 2003 komen 3 mensen om het leven. De economische schade is enorm. De massale stroomuitval in meerdere centrales was het gevolg van het niet adequaat afhandelen van foutindicaties van de stroomuitval in de eerste centrale.

Dat deze gevolgen ongewenst zijn, is duidelijk. Verhoging van softwarekwaliteit zal de kans dat deze gevolgen zich voordoen reduceren. Hiermee is het belang van softwarekwaliteit alsmede het inzicht in de manieren waarop deze kwaliteit verhoogd kan worden, aangetoond.

3.3 Definitie van softwarekwaliteit

Voordat we vaststellen wat we verstaan onder kwaliteit van software, zullen we eerst kijken wat er wordt verstaan onder de begrippen 'kwaliteit in het algemeen' en 'software'.

Definitie van kwaliteit

Waarschijnlijk zijn er weinig begrippen waarvan zoveel definities bestaan als het begrip kwaliteit. Een verklaring hiervoor is dat kwaliteit in sterke mate afhankelijk is van het perspectief van waaruit kwaliteit wordt benaderd. Evans en Lindsay noemen vijf criteria van waaruit kwaliteit kan worden benaderd [Evans et al., 1999], te weten het transcedente criterium, het productgebaseerde criterium, het gebruikersgebaseerde criterium, het waardegebaseerde criterium en het productiegebaseerde criterium. Het transcedente criterium beschouwt kwaliteit als superioriteit of excellerend. Het productgebaseerde criterium ziet kwaliteit als functie van specifieke, meetbare producteigenschappen. Het gebruikersgebaseerde criterium ziet kwaliteit als geschiktheid voor bedoeld gebruik. Twee definities die binnen dit criterium vallen zijn:

> Quality is what the customer says it is.

Definitie 8: Kwaliteit [Feigenbaum, 1983]

en

> Quality is fitness for use (which has five major dimensions, quality of design, quality of conformance, availability, safety, and field use).

Definitie 9: Kwaliteit [Juran, 1995]

Waardegebaseerde criteria beschouwen kwaliteit als de verhouding tussen prestatie en prijs. Productiegebaseerde criteria zien kwaliteit tot slot als het conformeren aan specificaties. De volgende definitie van Crosby valt binnen dit criterium:

> Quality is conformance to requirements.

Definitie 10: Kwaliteit [Crosby, 1979]

Een toonaangevende organisatie die zich bezighoudt met standaardisatie, de International Organization for Standardization (ISO), definieert kwaliteit als volgt:

> The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs.

Definitie 11: Kwaliteit [ISO 8402, 1994]

Reeves en Bednar [Reeves & Bednar, 1994] stellen dat er twee definities zijn die het vaakst voorkomen in de literatuur. De ene definitie definieert kwaliteit als het conformeren aan specificaties. Deze definitie wordt vooral voor producten gebruikt. De andere definitie stelt dat kwaliteit het voldoen aan of het overtreffen van verwachtingen is. Deze definitie betreft doorgaans diensten.

Samenvattend kunnen we stellen dat er veel definities van het begrip kwaliteit bestaan. De gemene deler van deze definities is, dat kwaliteit wordt bepaald door de mate waarin een product of dienst met een – mogelijk brede – verzameling karakteristieken in staat is om in impliciete of expliciete behoeften te voorzien. Deze opvatting van het begrip kwaliteit zal in het vervolg van deze scriptie worden gehanteerd.

Definitie van software

Voordat kan worden gedefiniëerd wat precies onder kwaliteit van software wordt verstaan, dient duidelijk te zijn wat onder de term 'software' valt. Ook van dit begrip bestaan veel definities. Humphrey geeft de volgende definitie:

> Software can be viewed as executable knowledge.

Definitie 12: Software [Humphrey, 1989]

In deze context is software kennis, die op een dusdanig precieze wijze is gedefiniëerd en gestructureerd, dat deze door een computer kan worden uitgevoerd. Een andere definitie van software wordt gegeven door ISO/IEC:

> (Software is…) All or part of the programs, procedures, rules, and associated documentation of an information processing system.

Definitie 13: Software (ISO/IEC 2382-1: 1993)

Een andere, veelgebruikte, definitie wordt gegeven door IEEE:

> (Software is…) Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Definitie 14: Software (IEEE Std 610.12-1990)

Deze laatste twee definities beschouwen software als het totaal van programma's (implementaties van de functionaliteit), procedures, regels (beschrijvingen van de te volgen stappen en richtlijnen bij gebruik en onderhoud) en documentatie (de neerslag van de procedures en regels). Het verschil tussen deze definities dat de laatste definitie gegevens ook rekent tot software, terwijl gegevens hier in de voorlaatste definitie niet toe worden gerekend.

In het vervolg van deze scriptie hanteer ik Definitie 13, wat inhoudt dat gegevens niet tot de software worden gerekend. Dit standpunt komt vaker voor [Bocij et al., 1999][Sommerville, 2004].

Definitie van softwarekwaliteit

Vanwege de vele, verschillende, definities van het begrip kwaliteit is het onwaarschijnlijk dat het begrip softwarekwaliteit eenduidig gedefiniëerd kan worden. Een voorbeeld van een definitie is de volgende:

> Softwarekwaliteit is de mate waarin software correct en robuust is.

Definitie 15: Softwarekwaliteit [Koldijk, 1999]

Correct is hierin synoniem voor het voldoen aan de specificaties, terwijl met robuust wordt bedoeld in hoeverre de software zich op een redelijke manier gedraagt indien zich situaties voordoen die niet door de specificaties worden gedekt. In feite kan ieder definitie van het begrip kwaliteit worden toegepast op het begrip software. Zo kan het kwaliteitsbegrip van ISO 8402 als volgt worden toegepast op softwarekwaliteit:

> Softwarekwaliteit is het totaal van eigenschappen en karakteristieken van software dat bijdraagt aan diens vermogen om te voldoen aan gestelde en geïmpliceerde behoeften.

Definitie 16: Softwarekwaliteit (afgeleid van ISO 8402)

Deze definitie impliceert dat er een aantal aspecten bekend zijn. Zo moeten de gestelde en geïmpliceerde behoeften bekend zijn. Daarnaast moeten de eigenschappen en karakteristieken van de software bekend zijn en tot slot moet bekend zijn wat het vermogen van deze eigenschappen en karakteristieken is om aan de gestelde behoeften te voldoen. Het onvermogen van software om hieraan te voldoen, kan dus worden gezien als een tekortkoming van de software. Naarmate dit onvermogen toeneems, is de software van slechtere kwaliteit.

Het is niet mogelijk om een definitie van softwarekwaliteit te geven die in iedere situatie toepasbaar is. Er moet rekening worden gehouden met het perspectief van waaruit wordt gekeken. Hendriks onderkent bij het ontwikkelen van software vier invalshoeken, te weten proces, product, people en performance [Hendriks, 2000]. Binnen deze scriptie is de keuze gemaakt de softwarekwaliteit vanuit de invalshoek van het product te beschouwen.

Concluderend kan worden gesteld dat softwarekwaliteit veel definities kent. De beoordeling van de mate waarin aan één van deze definities van softwarekwaliteit wordt voldaan, is pas mogelijk nadat dit begrip is geoperationaliseerd binnen een bepaalde context, hier gezien vanuit het softwareproduct. Hiervoor kan gebruik worden gemaakt van een softwarekwaliteitsmodel, waarvan in paragraaf 3.5 enkele varianten worden besproken.

3.4 Kwaliteitsborging

In paragraaf 3.2 is aangetoond dat softwarekwaliteit belangrijk is. Daarna is in paragraaf 3.3 vastgesteld wat onder softwarekwaliteit verstaan kan worden. Voordat gekeken wordt naar de beoordelingsmethoden van kwaliteit kijken we hoe deze kwaliteit tot een vastgesteld niveau te brengen is, een methode die bekend staat als kwaliteitsborging. Bij kwaliteitsborging worden binnen het softwareontwikkelingsproces activiteiten uitgevoerd die tot doel hebben de kwaliteit van software te garanderen tot een vastgesteld niveau.

Aangezien kwaliteit een belangrijke eigenschap van software is, kan worden verwacht dat kwaliteitsborging onderdeel uitmaakt van de diverse ontwikkelmodellen voor software. Dit blijkt ook zo te zijn indien we de verschillende modellen nader bekijken.

Het Waterfall model bevat aan het einde van iedere fase een controle voordat doorgegaan wordt met de volgende fase. De fasen waarin de haalbaarheid en de requirements worden bepaald eindigen na de validatie hiervan [Boehm, 1988]. In de ontwerpfasen vindt verificatie van de ontwerpen plaats [Boehm, 1988]. In de coderingsfase wordt het eindresultaat met behulp van unit tests gecontroleerd [Boehm, 1988]. Aan het einde van de integratiefase vindt er een product verification plaats

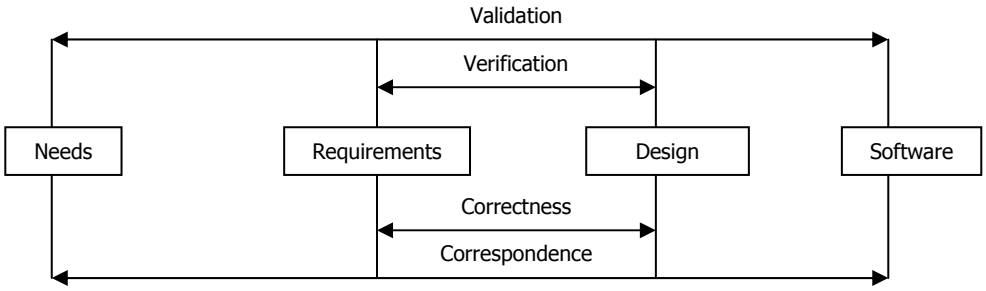
[Boehm, 1988], een activiteit die tegenwoordig als integratietest wordt aangeduid. Na de implementatie vindt een systeemtest plaats en tijdens de fase operations wordt er een revalidatie uitgevoerd [Boehm, 1988].

Het spiral model van Boehm bevat de volgende kwaliteitsactiviteiten [Boehm, 1988]:

- validatie van requirements;
- validatie en verificatie van het ontwerp;
- unit testing na codering;
- integratietest na integratie;
- acceptatietest nadat het systeem volledig is.

Bij RAD wordt door het laten controleren en goedkeuren van de prototypes nog meer de nadruk gelegd op de validatie van de requirements door de klant.

De begrippen validatie en verificatie kunnen worden verduidelijkt aan de hand van de beschrijving van Blum [Bahrami, 1999]. Hierbij worden de begrippen correspondence, validation, correctness en verification gebruikt. Hij stelt dat validation de activiteit is die de correspondence voorspelt. Dit is de mate waarin het opgeleverde systeem aan de behoeften van de operationele omgeving voldoet, welke beschreven zijn in de requirements. De correspondence kan pas worden vastgesteld nadat het systeem in gebruik is genomen. Correctness is de consistentie waarmee de requirements zijn overgenomen in de specificatie van het ontwerp. Verificatie is de activiteit die de mate van correctness vaststelt.



Figuur 8: Kwaliteitsmaatstaven en activiteiten [Bahrami, 1999]

Boehm stelt dat verificatie antwoord geeft op de vraag of het product juist wordt gebouwd, terwijl validatie antwoord geeft op de vraag of het juiste product wordt gebouwd [Boehm, 1984].

De agile methodologieën maken tot slot gebruik van de volgende activiteiten die de kwaliteit moeten garanderen:

- code reviews (eventueel gelijktijdig met het maken van de code in geval van pair programming);
- geautomatiseerd testen waarbij de code na iedere wijziging aan de hand van functionele tests wordt gecontroleerd op een juiste werking;
- continue integratie, waarbij het builden van de code geautomatiseerd wordt uitgevoerd zodra een programmeur zijn gewijzigde code heeft ingecheckt in het versiebeheersysteem.

Bij het streven naar kwalitatief hoogwaardige software wordt getracht het aantal fouten in de software te reduceren. Deze fouten kunnen worden ingedeeld in syntactische fouten, runtime fouten en logische fouten. Het elimineren van syntactische fouten vindt plaats via het proces van het debuggen. Dit gebeurt met behulp van compilers, welke syntactische fouten met een hoge nauwkeurigheid kunnen vaststellen. Het elimineren van logische en runtime fouten wordt nagestreefd door middel van het proces van testen. [Bahrami, 1999] In tegenstelling tot syntactische fouten is het verwijderen van runtime en vooral logische fouten veel moeilijker. Doorgaans blijven er na het testen nog een aantal fouten achter, waarvan een deel na oplevering alsnog wordt geconstateerd. Hier komen we in 5.3 op terug.

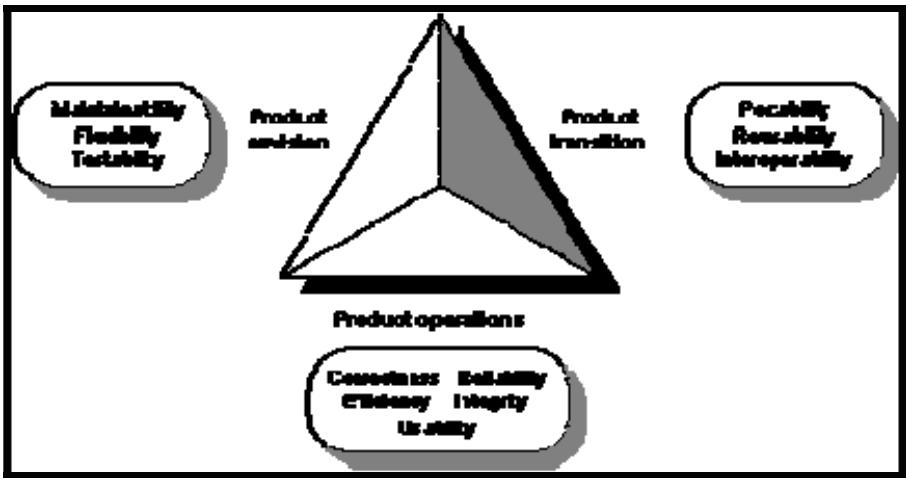
Samenvattend kan worden gesteld dat er binnen het softwareontwikkelingsproces een veelvoud aan technieken en activiteiten zijn die de kwaliteit moeten garanderen. Sommige van deze technieken worden al lange tijd uitgevoerd, zoals visuele controles. Andere technieken zijn pas in de relatief nieuwe agile methoden expliciet opgenomen, zoals continue integratie.

3.5 Kwaliteitsevaluatie

Er zijn sinds eind jaren zeventig diverse modellen ontwikkeld waarmee de kwaliteit van software vastgesteld kan worden. Van deze modellen behandel ik drie veelgebruikte modellen [Hyatt, Rosenberg, 1996].

General Electric

Het General Electric (GE) model of Factor Criteria Metric (FCM) model van James McCall uit 1977 benadert softwarekwaliteit vanuit drie perspectieven of factoren. Deze factoren bestaan elk uit een aantal criteria. Vooruitlopend op de volgende paragraaf kunnen voor deze criteria metrieken worden gebruikt die meten in hoeverre aan de criteria is voldaan.



Figuur 9: GE kwaliteitsmodel [McCall et al., 1977]

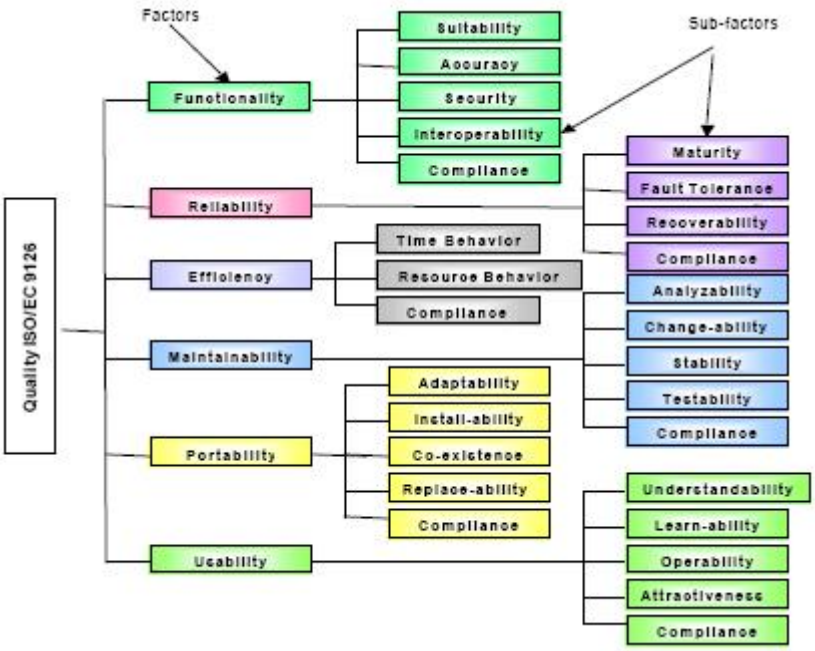
Het eerste perspectief, *product operations*, bekijkt de effectiviteit van softwareoperaties in het uitvoeren van een verzameling taken, zoals het gemak waarmee gegevens ingevoerd kunnen worden en het gemak en de betrouwbaarheid van de uitvoer. Binnen dit perspectief onderscheidt hij de attributen (hij noemt deze criteria) *usability, correctness, reliability, integrity* en *efficiency*. Het perspectief *product revision* bekijkt kwaliteit vanuit het gezichtspunt van het gemak waarmee de performance van de software verbeterd, gewijzigd en onderhouden kan worden. De binnen dit perspectief onderscheiden softwareattributen of factoren zijn maintainability, flexibility en testability. Het laatste perspectief, *product transition*, beschouwt het gemak waarmee software kan ingezet kan worden binnen andere omgevingen. Hierbij worden de attributen interoperability, reusability en portability genoemd.

Boehm

Een ander bekend model is het model van Boehm uit 1978. Dit model komt in grote lijnen overeen met het hiervoor genoemde model van McCall. Het decomposeert kwaliteit ook in verschillende aspecten en bekijkt aspecten van kwaliteit die voor de gebruiker van belang zijn. Ten opzichte van het model van McCall definieert het model van Boehm een bredere verzameling karakteristieken, waaronder hardware, en 19 criteria in tegenstelling tot de 11 van McCall's model.

ISO-9126

In 1991 heeft de International Standards Organisation (ISO) de ISO-9126 standaard gepubliceerd, welke als doel heeft te komen tot een uniforme wijze van vaststelling van de kwaliteit van een softwareproduct. Kwaliteit wordt hierin in eerste instantie gedecomposeerd in de zes attributen *functionality, reliability, usability, efficiency, maintainability* en *portability*. Vervolgens wordt van elk attribuut een verdere decompositie gegeven in subattributen.



Figuur 10: ISO-9126 model [Kececi, Abran, 2001]

Het attribuut *functionality geeft aan* of de software beschikt over de vereiste functies en is onderverdeeld in de subattributen *suitability*, *accuracy*, *security*, *interoperability* en *compliance*.

Het attribuut *reliability* geeft aan hoe betrouwbaar de software is en bestaat uit de subattributen *maturity*, *fault tolerance*, *recoverability* en *compliance*.

Het attribuut *efficiency* geeft aan hoe efficiënt de software is en bevat de subattributen *time behaviour*, *resource behaviour* en *compliance*.

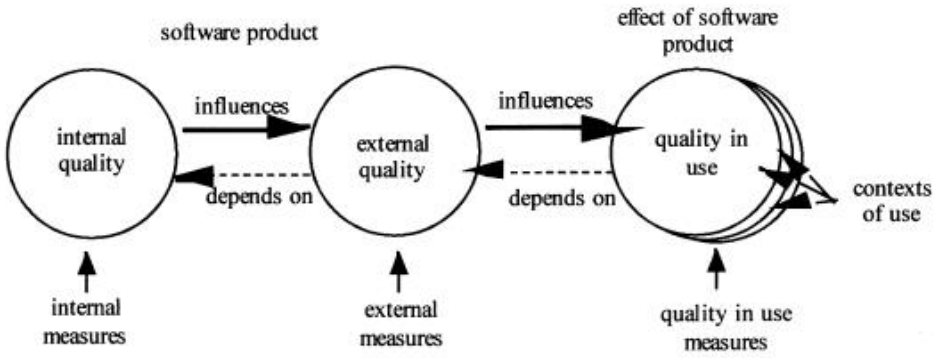
Maintainability geeft een indicatie voor het gemak waarmee wijzigingen kunnen worden doorgevoerd in de software en is onderverdeeld in de subattributen *analyzability*, *changeability*, *stability*, *testability* en *compliance*.

Portability indiceert het gemak waarmee de software naar een andere omgeving overgezet kan worden. Dit attribuut bestaat uit de subattributen *adaptability*, *installability*, *co-existence*, *replacability* en *compliance*.

Usability beschrijft hoe eenvoudig het is om de software te gebruiken. De subattributen die hierbij horen zijn *understandability*, *learnability*, *operability*, *attractiveness* en *compliance*.

Het begrip softwarekwaliteit wordt binnen ISO-9126 uitgesplitst in de kwaliteit van het proces en de kwaliteit van het product. Deze laatste is onderverdeeld in interne kwaliteit, welke bepaald wordt door de statische eigenschappen van de code, externe kwaliteit, bepaald door de dynamische eigenschappen van de code wanneer deze uitgevoerd wordt en door gebruikskwaliteit, welke bepaald wordt door de mate waarin de software tegemoet komt aan de behoeften van de gebruiker. Er is sprake van een voorwaardelijke relatie tussen de kwaliteitsaspecten. De interne kwaliteit beïnvloedt de externe kwaliteit.

De externe kwaliteit is dus afhankelijk van de interne kwaliteit en beïnvloedt op zijn beurt de gebruikskwaliteit. Deze gebruikskwaliteit is hierdoor afhankelijk van de externe kwaliteit. Onderstaande figuur geeft een overzicht van de kwaliteitsaspecten en hun relaties.



Figuur 11: ISO-9126 kwaliteitsaspecten [Bevan, 1999]

Elke van deze kwaliteitsaspecten heeft metrieken die gericht zijn op de specifieke eigenschappen van dit aspect. De interne metrieken richten zich op de statische eigenschappen van de code, zoals padlengte en aantal regels code. Deze worden meestal gemeten door het uitvoeren van inspecties. Externe metrieken zijn daarentegen gericht op de dynamische eigenschappen van de software als onderdeel van het systeem. Deze hebben betrekking op het testen en meten van het gedrag van het systeem. De gebruikskwaliteitsmetrieken meten de mate waarin de software voorziet in de behoeften van de gebruiker, zoals effectiviteit, productiviteit en tevredenheid.

Bij het toepassen van ISO-9126 wordt, aan de hand van de kwaliteitsvereisten van de gebruiker, vastgesteld welke kwaliteitskarakteristieken en subkarakteristieken relevant zijn. Voor de relevante karakteristieken worden vervolgens relevante metrieken vastgesteld voor de relevante kwaliteitsniveaus.

Er bestaat kritiek op de wetenschappelijke onderbouwing van de genoemde hiërarchische kwaliteitmodellen, onder andere wordt de keuze voor de gebruikte factoren niet onderbouwd [Kitchenham et al., 1996]. Ondanks deze kritiek is de praktische toepasbaarheid ervan overduidelijk door de hoeveelheid publicaties over deze modellen [van Veenendaal et al., 1997].

3.6 Conclusie

Softwarekwaliteit speelt een belangrijke rol in de maatschappij en het belang hiervan zal alleen maar toenemen. Het begrip softwarekwaliteit is op veel verschillende wijzen gedefiniëerd en voorafgaand aan het meten ervan zal een operationalisering moeten plaatsvinden van dit begrip. Er zijn verschillende modellen die de kwaliteit van software meten. Het merendeel van deze modellen gebruikt de gemeenschappelijke methode die kwaliteitsfactoren vaststelt, criteria vaststelt die deze factoren representeren en

metrieken voor deze criteria bepaalt. Van elk van deze metrieken kan een waarde worden vastgesteld die de waarde van de criteria bepaalt, waarmee de waarde van de factoren en dus van de kwaliteit van de software kan worden bepaald.

Het invoeren van genoemde hiërarchische kwaliteitsmodellen is lastig. Het zijn zogenoemde referentiemodellen die op basis van de specifieke omstandigheden moeten worden ingevuld. Hierbij moeten relevante factoren worden aangewezen en moeten de gebruikte eigenschappen en metrieken worden bepaald. De interpretatie van de meetresultaten is zonder referentiekader – in de vorm van kengetallen of eerdere meetresultaten – lastig, zodat een eindoordeel zonder dit kader moeilijk te vellen is.

Bij het operationaliseren van het betrouwbaarheidsbegrip in 5.4 zal rekening worden gehouden met de noodzaak van een referentiekader in de vorm van een conventioneel systeem.

4 Aspecten van CBD die de betrouwbaarheid van software beïnvloeden

4.1 Inleiding

Toepassing van CBD heeft – ten opzichte van conventionele ontwikkeling – tot gevolg dat het ontwikkelproces op een andere wijze wordt ingevuld. Omdat hergebruik bij CBD centraal staat, wordt eerst in algemene zin naar dit begrip hergebruik gekeken. Daarna wordt dit begrip bekeken in samenhang met CBD. Ook worden de specifieke aspecten van CBD, die de betrouwbaarheid beïnvloeden, op een rij gezet.

In paragraaf 4.2 wordt een overzicht gegeven van hergebruik in het algemeen. Paragraaf 4.3 gaat daarna in op hergebruik van componenten en de gevolgen daarvan, waarbij twee hypothesen worden opgesteld. In paragraaf 4.4 komen vervolgens specifieke aspecten van componenten aan de orde die invloed hebben op de betrouwbaarheid. Tot slot wordt in paragraaf 4.5 een overzicht gegeven van de bevindingen in dit hoofdstuk.

4.2 Hergebruik in het algemeen

Het streven naar hergebruik van software is al zo oud als het gebruiken van software zelf. Onder hergebruik wordt het meerdere malen gebruiken van dezelfde functionaliteit verstaan. De wijze waarop hergebruik plaats kan vinden is uiteraard beïnvloed door de voortgang in het ontwikkelen van software. Hergebruik werd oorspronkelijk toegepast met behulp van 'sprongen' in de broncode. Later werd deze functionaliteit in een functie geplaatst en werd deze functie vanuit meerdere plaatsen aangeroepen. De komst van objectoriëntatie introduceerde een nieuw mechanisme voor hergebruik: overerving. Hierbij wordt de functionaliteit geïmplementeerd als methode van een klasse, wat vergelijkbaar is met de hierboven beschreven functie. Wanneer een andere klasse wordt gebaseerd op deze klasse, erft deze alle eigenschappen en methodes van de basisklasse. Bij hergebruik van componenten wordt een component aangeroepen binnen een systeem, meestal door een ander component.

Er worden twee vormen van hergebruik onderkend: [Poulin, 1996]

- opportunistisch hergebruik en
- gestructureerd hergebruik.

Bij opportunistisch hergebruik wordt er zonder goed over de gevolgen na te denken bestaande functionaliteit gekopieerd en is er dus geen sprake van een gestructureerd proces. Deze vorm van hergebruik wordt ook wel *white-box reuse,* cut-and-paste reuse of ad-hoc hergebruik genoemd. Hoewel dit op korte termijn een productiviteitswinst oplevert, leidt deze vorm van hergebruik op lange termijn tot een lagere productiviteit en meer fouten, omdat de functionaliteit nu niet langer op één, maar op twee (of meer) locaties moet worden bijgehouden, inclusief de bijbehorende documentatie en het testen na iedere wijziging [Mili et al., 2001]. Het vergeten van één van deze kopieën is een veel

voorkomende bron van problemen bij onderhoud aan systemen waarbij gebruik is gemaakt van opportunistisch hergebruik.

Bij gestructureerd hergebruik wordt het hergebruik gepland en formeel geïntegreerd binnen het ontwikkelproces. De nadruk ligt hierbij op hergebruik zonder aanpassing van de broncode. Deze vorm wordt ook wel *black-box reuse* genoemd. Er zijn een aantal redenen waarom deze vorm van hergebruik te prefereren is boven de eerstgenoemde wijze. Ten eerste wordt de hergebruikte code gecentraliseerd en vanaf deze centrale locatie meerdere malen aangeroepen. Hierdoor kunnen wijzigingen ten gevolge van veranderende functionaliteit of geconstateerde defecten veel sneller worden gelokaliseerd en doorgevoerd. Een andere reden is dat het concept van de black box ertoe leidt dat er bij de ontwikkeling van het systeem meer nadruk ligt op wat de geleverde functionaliteit is en het minder belangrijk – of zelfs irrelevant – is om te weten hoe deze functionaliteit precies wordt geleverd. Deze verschuiving van de aandacht leidt tot een aanzienlijke reductie van de complexiteit van het systeem.

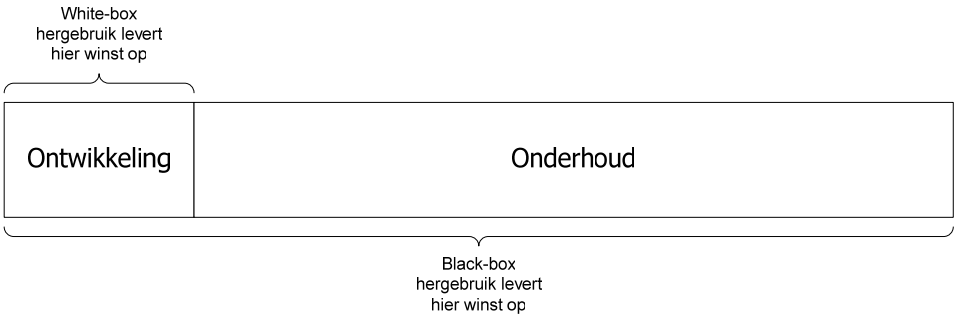
Naast black-box reuse en white-box reuse worden ook nog de volgende vormen van hergebruik onderkend:

- Glass-box reuse is een vorm van hergebruik waarbij de interne werking (de implementatie) wel zichtbaar is, zoals bij white-box reuse, maar het is bij deze vorm van hergebruik niet mogelijk deze implementatie aan te passen.
- Grey-box reuse is een vorm van hergebruik waarbij een gecontroleerd deel van de implementatie onthuld wordt. Dit deel kan worden gezien als een specificatie van de werking [Szyperski, 1997].

Beide vormen van hergebruik vallen onder structureel hergebruik. Omdat deze vormen van hergebruik niet algemeen worden gebruikt, wordt er binnen deze scriptie niet verder op ingegaan.

Kader 3: Glass-box reuse en Grey-box reuse

Een doorslaggevend argument voor black-box reuse wordt geïllustreerd door Figuur 12. Deze figuur laat zien dat hergebruik waarbij kleine aanpassingen worden gemaakt (kenmerkend voor white-box reuse), uitsluitend in de ontwikkelfase winst oplevert, terwijl hergebruik zonder aanpassing in de ontwikkelfase winst oplevert, maar dat deze winst in de onderhoudsfase veelal een grotere winst oplevert. Aangezien 60% tot 80% van de geleverde inspanningen tijdens de levenscyclus van software bestaat uit onderhoud [Poulin, 1996][Sommerville, 2004], kijken we in het vervolg van deze scriptie uitsluitend naar structureel hergebruik.



Figuur 12: Voordelen van hergebruik, aangepast van [Poulin, 1996]

Een ander belangrijk punt bij hergebruik is het verschil tussen hergebruik en gebruik. Het is algemeen bekend dat functionaliteit die meerdere malen wordt gebruikt slechts op één plaats dient te worden gedefiniëerd en vanaf meerdere locaties dient te wordt aangeroepen. Dit is één van de kenmerken van 'good programming' [Poulin, 1996]:

> In traditional procedural development, programmers use language features such as macros, functions, and procedures when they repeatedly execute the same operation. The programmer may use techniques (depending on the implementation language) such as variable macros (a method for overloading macro definitions), generic packages (a method for overloading abstract data types in Ada), or parameterized functions (a method for passing logic into a routine through the routine's interface). We consider the use of these language features and techniques "good programming" or "good design".

Citaat 3: Good programming [Poulin, 1996]

Indien deze functionaliteit gerepresenteerd wordt door een component, kan onderscheid worden gemaakt tussen gebruik of hergebruik van het component op grond van de herkomst van het component [Poulin, 1996]. Hierbij wordt onderscheid gemaakt tussen intern hergebruik en extern hergebruik. Er is sprake van intern hergebruik indien het component afkomstig is van dezelfde organisatie of hetzelfde project waarin het is ontwikkeld. Bij extern hergebruik is het component afkomstig van buiten deze omgeving.

In het vervolg van deze scriptie rekenen we intern hergebruik, dus hergebruik van een component bínnen de omgeving waarin het is ontwikkeld, niet als hergebruik maar als gebruik. De reden hiervoor is dat het component specifiek is ontwikkeld voor het betreffende project en het nog niet is bewezen dat het component geschikt is voor hergebruik. Dit laatste zal pas blijken indien het component generiek genoeg is om in een ander project of binnen een andere organisatie te worden hergebruikt.

4.3 Hergebruik van componenten

Het centrale thema binnen CBD is hergebruik. Componenten zijn vanwege hun eigenschappen uitermate geschikt voor hergebruik. Voordat ingegaan wordt op de gevolgen voor kwaliteit zullen we het ontwikkelproces van CBD nader bekijken ten opzichte van het ontwikkelproces waarbij geen hergebruik plaatsvindt.

Het ontwikkelproces bij CBD maakt onderscheid tussen *ontwikkelen voor hergebruik* en *ontwikkelen met hergebruik*. Deze worden beide besproken.

4.3.1 Ontwikkelen voor hergebruik

Het onderkennen van de mogelijkheid tot hergebruik kan plaatsvinden zodra een bepaalde functionaliteit voor de tweede keer wordt gebruikt. Het is echter veelal niet mogelijk om specifieke functionaliteit direct opnieuw te gebruiken. Eén van de redenen hiervoor is dat de functionaliteit vaak relaties heeft met andere delen van het systeem, die niet opnieuw worden gebruikt. Deze relaties moeten eerst zoveel mogelijk worden geïsoleerd van de rest van de gewenste functionaliteit, omdat ze het hergebruik van de gewenste functionaliteit belemmeren. Een tweede reden is dat de functionaliteit vaak te specifiek is. Om geschikt te zijn voor hergebruik zal deze functionaliteit eerst voldoende generiek moeten worden gemaakt.

Onder ontwikkelen voor hergebruik wordt het ontwikkelen van herbruikbare softwarecomponenten verstaan, waarbij de volgende activiteiten worden onderkend [Poulin, 1996]:

- Uitvoeren van domeinanalyse.
- Generaliseren voor additionele activiteiten.
- Toevoegen van additionele documentatie.
- Testen om het component voldoende betrouwbaar te maken.
- Testen voor additionele potentiële toepassing.
- Voorbereiden voor distributie.

De ontwikkeling van een component kan worden gezien als een normaal ontwikkelproject, waarbij het component het op te leveren eindproduct is. Er is hierbij geen voorgeschreven of meest optimaal ontwikkelmodel, dus er kan een model worden gekozen op grond van de aard van de functionaliteit van het component en de kenmerken van het verwachte toepassingsgebied waarin het component wordt ingezet.

Hoewel objectoriëntatie wordt beschouwd als een geschikte methode voor het omgaan met complexiteit, dat hergebruik zelfs ten doel heeft [Bahrami, 1999], is toepassing hiervan zeker niet noodzakelijk. Het is gebruikelijk dat een component objecten instantieert en referenties naar deze objecten toegankelijk maakt voor gebruikers ('clients') van het component. Zelfs indien dit het geval is, is het niet vanzelfsprekend dat het component intern volledig objectgeoriënteerd gestructureerd is. Intern zou er ook

gebruik gemaakt kunnen worden van traditionele programmeerbenaderingen als functional programming of assembly [Szyperski, 1997]. Aangezien de interne werking van een component volledig is afgeschermd, is dit bezien vanuit de consument van het component – degene die componenten assembleert tot een systeem – volkomen irrelevant.

Indien de ontwikkeling van een component wordt vergeleken met de ontwikkeling van dezelfde functionaliteit binnen één specifiek systeem, kan worden gesteld dat het ontwikkelen van een herbruikbaar component complexer is. Dit komt ten eerste door de grotere generiekheid en ten tweede omdat de toekomstige toepassingen van het component niet bekend zijn. Dit laatste leidt ertoe dat het onmogelijk is het component binnen alle mogelijke systemen waarin het wordt gebruikt te testen.

Op grond van de grotere complexiteit bij de ontwikkeling van een component kan verondersteld worden, dat de betrouwbaarheid van een herbruikbaar component initieel minder groot is dan de kwaliteit van dezelfde functionaliteit wanneer deze niet als herbruikbaar component wordt geïmplementeerd. Deze veronderstelling is tevens de eerste hypothese:

De initiële betrouwbaarheid van een herbruikbaar component is lager dan de initiële betrouwbaarheid van een vergelijkbaar specifiek component.

Hypothese 1: Initiële betrouwbaarheid van een herbruikbaar component

4.3.2 Ontwikkelen met hergebruik

Bij ontwikkeling met hergebruik wordt bij het maken van het ontwerp na de analysefase en het verzamelen van de requirements de nadruk gelegd op de scheiding van het totaal aan functionaliteit in verzamelingen van bij elkaar horende functionaliteit. Vervolgens wordt op basis van deze functionaliteit gezocht naar componenten die deze functionaliteit bieden. Hierbij worden doorgaans eerst de verzameling componenten binnen het bedrijf bekeken. Indien hier geen geschikte componenten tussen zitten, wordt gekeken naar bestaande componenten buiten het bedrijf. Indien alle geschikte kandidaten verzameld zijn, wordt geëvalueerd welk component het meest geschikt is. Eventueel wordt er hierbij gekozen voor het intern ontwikkelen van een component. Na de selectie van alle bruikbare componenten (en eventuele ontwikkeling van nieuwe componenten) worden de componenten geassembleerd tot een werkend systeem. Dit gebeurt door de componenten met behulp van 'glue code' aan elkaar te koppelen.

De centrale vraag hierbij is natuurlijk waarom deze wijze van ontwikkelen zou leiden tot betere betrouwbaarheid. Herbruikbare componenten worden in meerdere systemen ingezet. Hierdoor is er sprake van een grotere gebruiksintensiteit dan bij specifieke componenten het geval is. Op basis van deze grotere gebruiksintensiteit veronderstellen we tevens een grotere onderhoudsintensiteit met betrekking tot correctief onderhoud. Dit laatste begrip wordt in Kader 4 toegelicht.

Een veelgebruikte typologie van onderhoud is de volgende van Swanson uit 1976 [Chapin et al., 2001]:

- Perfective Maintenance — Maintenance with the intention to perfect the system in terms of its performance, processing efficiency, or maintainability.
- Adaptive Maintenance — Maintenance with the intention to adapt the system to changes in its data environment or processing environment.
- Corrective maintenance — Maintenance with the intention to correct processing, performance, or implementational failures of the system.

De inhoud van deze begrippen wordt vaak door elkaar gehaald [Chapin et al., 2001]. In deze scriptie maken we gebruik van de typologie van Swanson, waarbij we de termen perfectief, adaptief en correctief onderhoud aanhouden.

Kader 4: Onderhoudstypologie voor software

Hierdoor lijkt het aannemelijk dat de betrouwbaarheid van herbruikbare, generieke componenten sneller stijgt dan het geval is bij specifieke componenten. Omdat herbruikbare componenten – zoals de naam al zegt – in meerdere systemen worden gebruikt, is het aannemelijk dat er een hogere prioriteit wordt gegeven aan het oplossen van defecten in herbruikbare componenten dan aan het oplossen van defecten in specifieke componenten. Op grond hiervan verwachten we dat het aantal defecten van CBD systemen sneller daalt dan het aantal defecten van conventionele systemen, wat leidt tot de tweede hypothese:

De snelheid waarmee de betrouwbaarheid van software toeneemt is hoger indien gebruik gemaakt wordt van CBD.

Hypothese 2: Snelheid toename betrouwbaarheid CBD-systeem t.o.v. conventioneel systeem

4.4 Karakteristieke betrouwbaarheidsaspecten van componenten

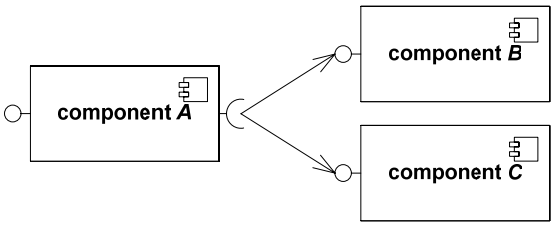
Bij CBD wordt een systeem opgebouwd met behulp van componenten. Deze vervullen de functionaliteit van het systeem doordat ze met elkaar samenwerken. Bij dit samenwerken zijn een aantal aspecten aan de orde, die meer problemen kunnen opleveren dan bij conventionele, monolitische systemen het geval was. Szyperski noemt asynchrone

communicatie en multithreading [Szyperski, 1997]. Sommerville voegt hier nog 'emergent properties' aan toe [Sommerville, 2004]. Deze aspecten worden in de rest van deze paragraaf besproken.

4.4.1 Asynchrone communicatie

Asynchrone communicatie is een mechanisme dat componenten in staat stelt met elkaar samen te werken zonder dat deze componenten vooraf weten met welke andere componenten deze samenwerking precies plaatsvindt. De werking van dit mechanisme kan als volgt worden omschreven.

Een specifieke verandering in de toestand van een component *A* wordt kenbaar gemaakt door het aanroepen van een event. Componenten kunnen doorgaans meerdere events genereren. Andere componenten kunnen zich aanmelden bij component *A*, zodat zij een bericht krijgen van component *A* indien zich daar events voordoen. Wanneer zij dit bericht ontvangen, kunnen ze hierop reageren door zelf specifieke activiteiten uit te gaan voeren. Dit mechanisme is weergegeven in Figuur 13.



Figuur 13: Multicasting van een event

Ter verduidelijking van dit mechanisme kan het voorbeeld van het hernoemen van een bestand worden gegeven. Indien er twee instanties van Windows Explorer (componenten *B* en *C*) dezelfde map weergeven, luisteren deze instanties naar de events van de bestanden in deze map. Zij hebben zich hiervoor geregistreerd bij het FileSystem (component *A*). Zodra van een bestand in deze map de naam gewijzigd wordt, verandert de toestand van dit bestand. Op grond van deze toestandswijziging genereert het FileSystem component het event NameChanged. Op grond van dit event zullen beide instanties van Windows Explorer hun beeld verversen, zodat de nieuwe naam van het bestand wordt weergegeven.

Dit mechanisme introduceert vier mogelijke probleemsituaties, die moeten worden ondervangen [Szyperski, 1997]:

1. Het standaard distributiemechanisme van events is multicasting, waarbij het event aan meerdere ontvangers tegelijk wordt doorgegeven. Gedurende het uitvoeren van deze multicast bevindt het systeem zich in een inconsistente toestand. Instanties van componenten moeten deze inconsistente toestand zelf constateren door het opvragen van de toestand van andere componenten.

2. Ontvangers van events kunnen op grond van deze ontvangst zelf ook events doorgeven. De volgorde waarin deze verschillende multicasts moeten worden uitgevoerd kan diverse problemen opleveren. Zo is het mogelijk, dat de multicast eerst moet worden doorgegeven van component A naar componenten B en C, voordat deze hun events doorgeven.
3. Tijdens het uitvoeren van een multicast kan de verzameling ontvangers veranderen. Deze mogelijkheid moet worden afgevangen.
4. Indien ontvangers van een event een probleem tegenkomen, kunnen ze een exception teruggeven. De wijze waarop deze exceptions worden afgehandeld bij multicasting is lastig.

Concluderend kan worden gesteld dat het mechanisme van asynchrone communicatie de complexiteit in sterke mate vergroot. Dit resulteert in complexere software die de componenten met elkaar laat communiceren (de 'glue code'), waardoor het aannemelijk is dat de betrouwbaarheid daalt.

4.4.2 Multithreading

Multithreading is het concept waarbij meerdere sequentiële activiteiten tegelijk worden uitgevoerd, waarbij elke synchrone activiteit een thread wordt genoemd. Multithreading wordt toegepast om bij meerdere gelijktijdige gebruikers van een systeem de performance beter te verdelen over de verschillende gebruikers. Het is niet de snelste wijze van het verwerken van aanvragen[3], maar geeft de gebruikers wel de indruk dat ze minder lang hoeven te wachten.

Bij multithreading kan iedere thread de waarde van een variabele lezen of schrijven. Indien meerdere threads dit tegelijk doen, is er sprake van *concurrent access*. Nu is er bij concurrent reads geen probleem, omdat de waarde van de variabele niet verandert, maar het is niet toegestaan dat twee threads de waarde van de variabele tegelijk schrijven, omdat de waarde van de ene thread overschreven zou worden door de andere thread. Ook is het niet toegestaan dat de ene thread de waarde van de variabele leest terwijl de andere thread de waarde verandert. De gelezen waarde zou in dit geval niet meer geldig zijn, omdat deze inmiddels veranderd kan zijn.

Een oplossing voor dit probleem wordt gegeven door de toegang tot een variabele te blokkeren indien nodig. Deze techniek wordt locking genoemd en de threads heten dan te zijn gesynchroniseerd. Er zijn verschillende vormen van locking, met elk verschillende gevolgen voor de performance van het systeem.

Locking kan echter een nieuw probleem veroorzaken indien meerdere threads op elkaar aan het wachten zijn. Deze situatie wordt deadlock genoemd en doet zich in de volgende

[3]De snelste verwerkingswijze wordt gerealiseerd door geen multithreading te gebruiken en de verschillende, gesorteerde, activiteiten sequentieel uit te voeren. Hierbij moeten de activiteiten gesorteerd zijn op oplopende (verwachte) verwerkingstijden.

situatie voor[4]. Stel dat er twee threads zijn, T_1 en T_2 en twee variabelen, v_1 en v_2. Beschouw daarna de volgende situatie waarin in twee stappen het volgende gebeurt. In stap 1 Beschrijft T_1 de waarde van v_1 en beschrijft T_2 de waarde van v_2. Hierbij krijgen deze threads een lock op de variabele wiens waarde ze schrijven. In stap 2 wil T_1 de waarde van variabele v_2 lezen. Omdat T_2 echter een lock op deze variabele heeft, wacht T_1 totdat deze lock wordt vrijgegeven en de waarde van v_2 gelezen kan worden. Tegelijkertijd wil T_2 de waarde van variabele v_1 lezen. Deze is echter gelockt door T_1, dus T_2 wacht totdat de lock op v_1 wordt opgeheven, zodat de waarde hiervan gelezen kan worden. Beide threads wachten nu totdat de ander klaar is, wat niet zal gebeuren. In deze situatie zal een concurrency mechanisme in moeten grijpen, bijvoorbeeld door na het constateren van een deadlock één van de twee threads te stoppen. Transacties zijn een mogelijkheid om de complexiteit van multithreading te reduceren, omdat een transactie kan worden teruggedraaid indien er iets misgaat (bijvoorbeeld in geval van een deadlock) en de transactie later opnieuw kan worden uitgevoerd.

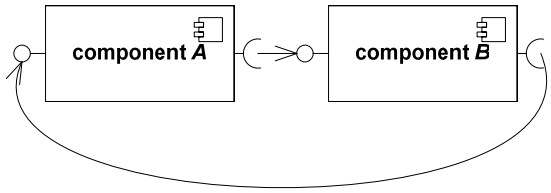
Multithreading is zonder gebruik te maken van componenten al complex. Indien toegepast met componenten, wordt de situatie nog complexer vanwege het feit dat er bij de ontwikkeling van componenten geen rekening kan worden gehouden met de componenten waarmee samengewerkt wordt. Deze grotere complexiteit zal de betrouwbaarheid waarschijnlijk niet ten goede komen.

4.4.3 Re-entrance

In traditionele systemen was sprake van een strikte verdeling in lagen, waarbij een laag uitsluitend de laag direct eronder mocht benaderen. In de huidige, gedistribueerde systemen is geen sprake meer van zo'n gelaagde architectuur. In principe mag elk component elk ander component benaderen.

We hebben bij asynchrone communicatie gezien dat een component tijdens de overgang van de ene toestand in de andere in een inconsistente toestand verkeert. Dit is bijvoorbeeld het geval indien er voor een overgang van de ene toestand naar de andere meerdere methods uitgevoerd moeten worden. Op zich is dit geen enkel probleem, zolang deze tijdelijke, inconsistente, toestand, verborgen blijft voor de rest van het systeem. Het is echter mogelijk, dat een ander component dit eerste component, dat in de inconsistente toestand verkeert, benadert. De situatie die optreedt wanneer het eerste component zich nog in een inconsistente toestand bevindt, terwijl het tweede component het eerste benadert, wordt re-entrance genoemd. Figuur 14 toont een voorbeeld van re-entrance.

[4] In de praktijk wordt meestal gebruik gemaakt van het ingewikkelder two-phase locking, waarbij verschil wordt gemaakt tussen schrijf- en leeslocks. Ter illustratie is het eenvoudiger single type locking beschreven.



Figuur 14: Component re-entrance

Component A roept hier een event aan vanwege een toestandswijziging, terwijl het nog in een inconsistente toestand verkeert. Component B heeft zich aangemeld en ontvangt het event van component A. In de afhandeling van dit event, roept component B rechtstreeks component A aan, wat nog steeds in de inconsistente toestand verkeert.

Het correct afhandelen van deze situatie is zeer complex. Vanwege de aard van CBD-systemen, zal deze situatie bij dit type systemen vaker voorkomen dan bij conventionele systemen. Hierdoor is het aannemelijk, dat de betrouwbaarheid van dit soort systemen minder groot is.

4.4.4 Specificaties (Contracten)

Een interface is een verzameling operaties die een gebruiker van een component hierop kan aanroepen [Szyperski, 1997]. Via deze interface wordt de functionaliteit van het component aangeroepen. Hierdoor fungeert de interface als het mechanisme om componenten met elkaar te verbinden.

Een specificatie beschrijft de betekenis van een interface [Szyperski, 1997]. Deze specificatie is bij CBD van groter belang dan bij conventionele ontwikkeling, omdat de producent van een herbruikbaar component niet direct communiceert met de consument van dit component. Hierdoor functioneert de specificatie als een contract en het conformeren aan deze specificatie is een voorwaarde voor samenwerking. De producent van het component zorgt ervoor dat de implementatie doet wat er in de specificatie staat beschreven. De consument zorgt ervoor dat de interface wordt aangeroepen conform de specificatie.

Een contract dient in elk geval de volgende onderdelen te bevatten [Szyperski, 1997]:

- Beschrijving van de interface, bij voorkeur met pre- en postcondities.
- Functionele requirements.
- Niet-functionele requirements (beslag op resources als tijd en geheugenbeslag).
- Manier waarop exceptions worden afgehandeld.

Vanwege het grotere belang van contracten bij CBD, is het bij deze vorm van softwareontwikkeling belangrijker dat deze contracten juist en volledig zijn. Per direct is hiervan echter niet te zeggen of dit vermoedelijk zal leiden tot een hogere of lagere betrouwbaarheid.

4.4.5 Emergent properties

Emergent properties zijn eigenschappen van een systeem die niet herleid kunnen worden tot eigenschappen van individuele componenten die deel uitmaken van het systeem, maar eigenschappen van het systeem als geheel zijn.

Deze eigenschappen zijn vaak niet of niet volledig te voorspellen voordat het systeem in zijn geheel is opgeleverd, maar kunnen zodra dit is gebeurd, worden gemeten. Voorbeelden hiervan zijn performance, bruikbaarheid en falende componenten die fouten doorgeven aan andere delen van het systeem [Sommerville, 2004].

Door deze emergent properties is het gedrag van een CBD systeem niet precies vooraf te voorspellen. Dit zou kunnen leiden tot lagere betrouwbaarheid.

4.5 Conclusie

Indien gekeken wordt naar de aspecten van CBD die de betrouwbaarheid beïnvloeden, kunnen zowel positieve als negatieve invloeden geconstateerd worden.

Een sterke positieve invloed is gelegen in het feit dat een herbruikbaar component vanwege de veelvoud van systemen waarin het wordt gebruikt, veel sneller een hoge mate van betrouwbaarheid zal realiseren.

De negatieve invloeden zijn vooral gelegen in de grotere complexiteit, welke wordt veroorzaakt door de grotere mate van generiekheid van een herbruikbaar component ten opzichte van een specifiek component, en verder door asynchrone communicatie, multithreading, re-entrance en emergent properties.

Vanwege deze tegengestelde invloeden zijn we niet direct in staat de resulterende invloed van CBD op de betrouwbaarheid vast te stellen. Hierdoor zullen we moeten komen tot een operationalisering van het betrouwbaarheidsbegrip, die ons in staat stelt de volgende hypothesen te toetsen.

1. De initiële betrouwbaarheid van een herbruikbaar component is lager dan de initiële betrouwbaarheid van een vergelijkbaar specifiek component.

2. De betrouwbaarheid van herbruikbare, generieke componenten stijgt sneller dan die van specifieke componenten.

Hypothesen

Deze operationalisering en de toetsing die daaruit volgt, zijn het onderwerp van de hoofdstukken vijf en zes.

5 Praktische toets: Ontwerp

5.1 Inleiding

In het voorgaande hoofdstuk zijn twee hypothesen opgesteld die iets zeggen over de veronderstelde invloed van CBD op de betrouwbaarheid van ontwikkelde software. Deze hypothesen worden getoetst om vast te kunnen stellen of ze beide juist blijken te zijn. Als aangetoond kan worden dat ze onjuist zijn spreken we van falsificatie.

Om te onderzoeken of de hypothesen gefalsificeerd kunnen worden maken we in deze scriptie gebruik van een toets, waarin de gewenste gegevens worden verzameld en geanalyseerd. De redenering die is gevolgd bij het ontwerpen van deze toets, wordt, samen met het resultaat van het ontwerp (de toets zelf), in dit hoofdstuk beschreven. Hierbij is rekening gehouden met de praktische uitvoerbaarheid ervan, op grond van de beschikbaarheid van gegevens. Door het bepalen van de wijze waarop de toets plaatsvindt, is er sprake van een operationalisering van het betrouwbaarheidsbegrip. Dit betekent dat we nu met gegevens uit de praktijk metingen kunnen verrichten die iets zeggen over de betrouwbaarheid. De operationalisering van het betrouwbaarheidsbegrip maakt de hypothesen toetsbaar.

De structuur van dit hoofdstuk is als volgt. In paragraaf 5.2 wordt een overzicht gegeven van meettheorie, zodat de gebruikte begrippen in het ontwerp van de toets duidelijk zijn. De projecten die aan de toets onderworpen zijn, worden in paragraaf 5.3 geïntroduceerd. Paragraaf 5.4 beschrijft de keuze van de eenheden waarin de resultaten van de toets worden uitgedrukt en de reden waarom voor deze eenheden is gekozen. Paragraaf 5.5 beschrijft vervolgens hoe de toets wordt uitgevoerd en paragraaf 5.6 bevat een korte samenvatting van dit hoofdstuk.

5.2 Meettheorie

Deze paragraaf gaat kort in op meettheorie. Deze theorie beschrijft hoe het proces van meten ingericht moet worden om metingen consistent uit te kunnen voeren en ervoor te zorgen dat de meetgegevens zo correct mogelijk worden geïnterpreteerd. In de bespreking van de meettheorie worden de basisbegrippen van het meten geïntroduceerd en wordt de samenhang tussen deze begrippen geïllustreerd aan de hand van een algemeen model. Daarna worden de randvoorwaarden van het meten toegelicht.

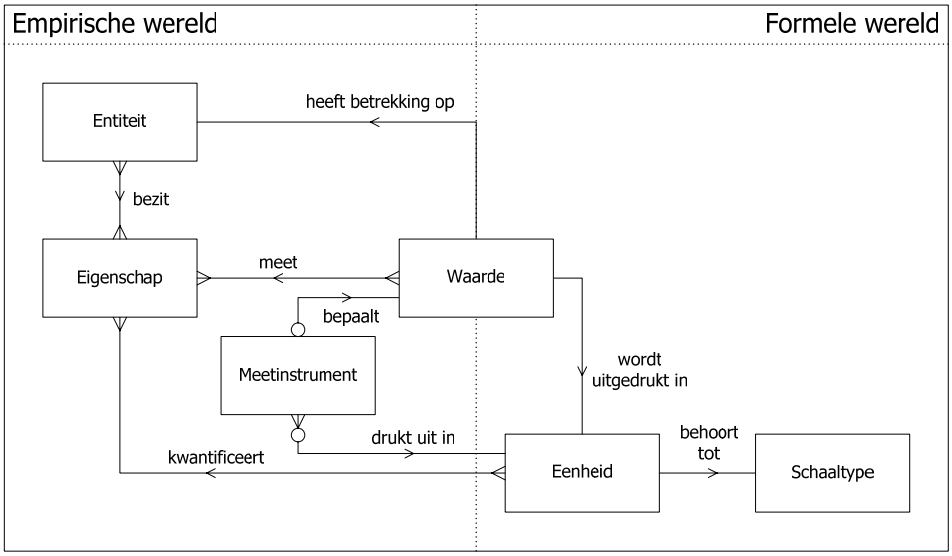
Een korte en bondige definitie van meten is de volgende [Kaner et al., 2004]:

> Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping to characterize an attribute.

Definitie 17: measurement [Fenton et al., 1997]

5.2.1 Het meetproces

Vrij vertaald is meten dus het uitdrukken van eigenschappen van entiteiten in getallen of symbolen. De entiteiten en hun eigenschappen bestaan in de empirische wereld, in de realiteit. De gemeten waarden zijn afbeeldingen van deze eigenchappen in de formele, wereld. Een model dat het proces van meten beschrijft is weergegeven in Figuur 15.



Figuur 15: Structureel model voor meten, aangepast van [Kitchenham, 1996]

Dit model wordt aan de hand van de gebruikte begrippen toegelicht.

Entiteit (entity)

Een entiteit is een object waarin we geïnteresseerd zijn. Voorbeelden zijn: een persoon, een auto en een softwaresysteem.

Eigenschap (attribute)

Een eigenschap is een kenmerk van een entiteit. Een entiteit heeft meestal meerdere eigenschappen. Zo bezit de entiteit persoon onder andere de eigenschappen lengte en gewicht. Andersom kan een eigenschap ook van toepassing zijn op meerdere entiteiten. Denk bijvoorbeeld aan de eigenschap lengte, welke van toepassing is op zowel de entiteit auto als de entiteit persoon.

Eenheid (unit)

Een eenheid stelt ons in staat de waarde van een eigenschap weer te geven, zodat de uitkomsten van het meten op uniforme wijze vastgelegd kunnen worden. Een eigenschap kan vaak in meerdere eenheden uitgedrukt worden. Zo kan de eigenschap temperatuur uitgedrukt worden in de eenheden graden Celcius, graden Fahrenheit of graden Kelvin.

Andersom kunnen meerdere eigenschappen worden uitgedrukt in dezelfde eenheid. Een voorbeeld van een dergelijke eenheid is het aantal gevonden fouten in software. Enerzijds wordt deze eenheid gebruikt om uit te drukken hoe goed er getest is. Anderzijds drukt deze eenheid uit hoe slecht er ontwikkeld is. Een voorbeeld van een dergelijke eenheid is het aantal afgekeurde eindproducten in een productieproces (de uitval). Naarmate de kwaliteitscontrole beter is, zullen meer eindproducten die niet aan de kwaliteitseisen voldoen, worden afgekeurd. De eenheid kan dus worden gebruikt om uit te drukken hoe goed de kwaliteitscontrole is. Anderzijds zal een slechter productieproces ertoe bijdragen dat het aantal eindproducten van onvoldoende kwaliteit groter is, zodat deze eenheid een indicator is voor de kwaliteit van het productieproces.

Een eenheid moet volledig gedefiniëerd zijn door te refereren aan een standaard of aan een theoretisch model [Kitchenham, 1996]. Hiermee wordt voorkomen dat meetresultaten verkeerd worden geïnterpreteerd.

Schaaltype (scale type)

Eenheden kunnen worden geclassificeerd op basis van het type schaal waarin zij uitgedrukt worden. De aanduiding voor dit type is *schaaltype*. Een eenheid behoort tot een schaaltype. Een schaaltype beperkt welke wiskundige en statistische bewerkingen toegestaan zijn bij het verwerken van de meetwaarden [Kitchenham, 1996]. De meest voorkomende schaaltypen zijn: [Kitchenham, 1996]: *nominaal*, *ordinaal*, *interval* en *ratio*. Deze schaaltypen worden in Tabel 1 toegelicht.

Schaaltype	Omschrijving	Voorbeelden
Nominaal	Verzameling categorieën die een classificatie voorstelt.	Soorten auto's: cabriolet, stationwagen, hatchback
Ordinaal	Geordende verzameling categorieën die een classificatie voorstelt.	Intensiteit telefoongebruik: vaak, soms, nooit
Interval	Numerieke waarden waarbij de afstand tussen opvolgende getallen even groot is, zonder *echte* nulwaarde.	Temperatuur gemeten in graden Celcius of Fahrenheit
Ratio	Intervalschaal met *echte* nulwaarde (zodat 2 eenheden evenveel is als 2 maal één eenheid).	Temperatuur gemeten in graden Kelvin, het aantal auto's op een parkeerplaats

Tabel 1: Schaaltypen

Waarde (value)

Een waarde meet een eigenschap van een entiteit en is daarmee van toepassing op deze entiteit. Een waarde wordt uitgedrukt in een eenheid.

Meetinstrument (measurement instrument)

Er kan gebruik gemaakt worden van een meetinstrument om de waarde van een eigenschap te bepalen. Een meetinstrument maakt gebruik van een eenheid om deze waarde in uit te drukken.

5.2.2 Vergelijkbaarheid van meetresultaten

Meetresultaten zijn niet zonder meer vergelijkbaar. De vergelijkbaarheid van meetresultaten is echter wel een fundamentele voorwaarde van wetenschappelijk onderzoek.

> The comparability of measurements made in differing circumstances by different methods and investigators is a fundamental pre-condition for all of science.

Citaat 4: Comparability of measurements [Dorans et al., 2000]

Indien niet aan de voorwaarde van vergelijkbaarheid is voldaan, kan geen enkele wetenschappelijke waarde worden gehecht aan vergelijkingen tussen meetresultaten.

Bekijk ter illustratie van de onderdelen van deze voorwaarde het opmeten van iemands bloeddruk. Verschillende omstandigheden kunnen leiden tot verschillende resultaten. Zo zal de bloeddruk van iemand die net een marathon heeft gelopen, net als die van iemand die net een grote hoeveelheid drop heeft geconsumeerd, hoger liggen. Ook is het moment op de dag waarop de meting wordt uitgevoerd van invloed. Tevens is de dikte van de arm van invloed: dikkere armen leidt tot een hogere gemeten waarde. Daarnaast kan de methode waarop de meting wordt uitgevoerd van invloed zijn. Zo zal een bloeddrukmeter die wordt verkocht in winkels met consumentenelectronica mogelijk andere meetresultaten opleveren dan de bloeddrukmeters die in ziekenhuizen worden gebruikt. Tot slot is degene die het onderzoek uitvoert van invloed. De mate van nauwkeurigheid waarmee deze, bijvoorbeeld door te luisteren door een stethoscoop, vaststelt bij welke druk het bloed gaat stromen (de bovendruk) of weer normaal stroomt (de onderdruk) is hier van belang.

Voordat meetresultaten met elkaar vergeleken kunnen worden, zal moeten worden vastgesteld of deze vergelijkbaar zijn. Omdat we in ons onderzoek voor beide onderzoeksobjecten dezelfde methode gebruiken en het onderzoek uitgevoerd wordt door één persoon, kunnen we ons beperken tot het controleren van de vergelijkbaarheid van de omstandigheden. Deze stelling is geformuleerd in Voorwaarde 1.

De vergelijkbaarheid van entiteiten waarvan de waarden van hun eigenschappen worden gemeten is een voorwaarde voor de vergelijkbaarheid van de meetresultaten.

Voorwaarde 1: Vergelijkbaarheid entiteiten

De praktische invulling van deze vergelijkbaarheid verschilt per situatie, maar dient in elk geval in beschouwing te worden genomen voordat de meetresultaten met elkaar vergeleken kunnen worden.

5.3 Selectie onderzoeksobjecten

Vanwege de centrale rol die het onderzoek inneemt binnen deze scriptie moet de selectie van de onderzoeksobjecten worden toegelicht. In paragraaf 5.3.1 wordt eerst gekeken naar de voorwaarden waaraan de systemen dienen te voldoen. Daarna volgt in paragraaf 5.3.2 de selectie van de systemen. Tot slot wordt in paragraaf 5.3.3 gekeken in hoeverre de systemen voldoen aan de criteria en wordt besloten of de systemen geschikt zijn voor het onderzoek.

5.3.1 Selectiecriteria

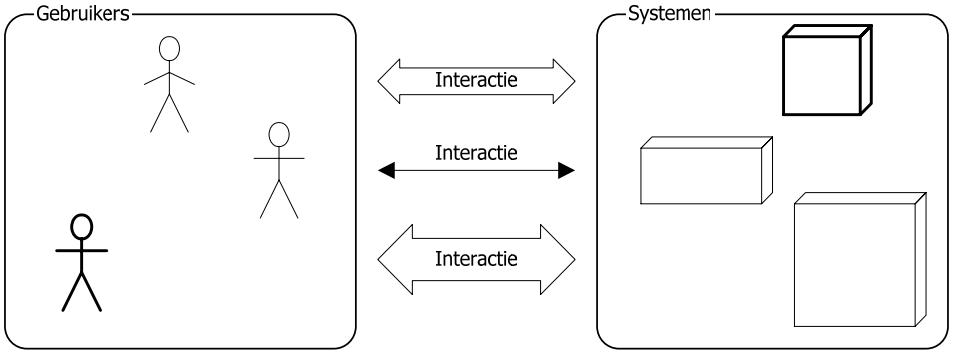
De essentie van de toets is het vaststellen of de betrouwbaarheid van software wordt beïnvloed door toepassing van CBD. Anders gezegd, gaan we de waarden van de eigenschap betrouwbaarheid van de entiteit softwaresysteem meten. Omdat we de invloed van de ontwikkelmethodiek CBD willen vaststellen, moeten we metingen uitvoeren op een systeem dat *niet* is ontwikkeld met CBD en een systeem dat *wel* is ontwikkeld met CBD. De systemen zullen we vanaf nu aanduiden met respectievelijk *conventioneel systeem* en *CBD-systeem*.

Idealiter willen we naast het verschil dat we onderzoeken verder volkomen gelijke omstandigheden, in het Latijn aangeduid als *ceteris paribus*. Specifiek willen we twee systemen die naast het genoemde verschil in ontwikkelmethodiek exact identiek zijn. Deze gelijkheid houdt allereerst in dat beide systemen dezelfde functionaliteit zouden moeten hebben, van dezelfde grootte, geschreven in dezelfde programmeertaal en gecompileerd met dezelfde compiler zouden moeten zijn en dezelfde gebruiksinterface zouden moeten hebben. Daarnaast zouden de omstandigheden met betrekking tot het gebruik identiek moeten zijn. Hiervoor moeten de platformen waarop de systemen worden gebruikt identiek zijn, moet de gebruiksintensiteit hetzelfde zijn en moeten de systemen gelijktijdig gebruikt worden door dezelfde gebruikers met dezelfde gegevens. Tot slot zou de onderhoudsintensiteit moeten overeenkomen, wat inhoudt dat er exact evenveel inspanning wordt besteed aan onderhoud van beide systemen.

Om te voldoen aan deze omstandigheden zouden er twee systemen moeten worden gebouwd die met betrekking tot functionaliteit identiek zijn en alleen qua ontwikkelingsmethodiek verschillen. Deze systemen zouden op uniforme wijze moeten worden aangeroepen met identieke gegevens, bijvoorbeeld door de activiteiten van een

specifieke gebruiker gedurende een vaste periode zeer nauwkeurig vast te leggen (inclusief de wachttijden tussen deze activiteiten) en deze activiteiten vervolgens op identieke wijze uit te laten voeren op beide systemen.

In de praktijk is bovenstaande situatie slechts met zeer veel inspanning en hoge kosten te realiseren. Aangezien het onderzoeksbudget dit niet toeliet, hebben we een aantal factoren bepaald die invloed uitoefenen op de vergelijkbaarheid van systemen. Daarbij is gebruik gemaakt van de interactie tussen gebruikers en systemen, zoals weergegeven in Figuur 16. De hierbij gebruikte redeneringen lopen vooruit op de operationalisering van de betrouwbaarheid in paragraaf 5.4.



Figuur 16: Interactie tussen gebruikers en systemen

In het overzicht van systeemgebruik zien we een aantal verschillende gebruikers interacteren met een aantal verschillende systemen en ook bij de interactie tussen deze gebruikers en systemen zijn verschillen te onderkennen. Hierbij zijn een drietal aspecten van grote invloed op de vergelijkbaarheid van de betrouwbaarheid van de systemen. Dit zijn de aspecten uitgebreidheid, complexiteit en gebruiksintensiteit. Aangezien de aspecten uitgebreidheid en complexiteit veel overeenkomsten hebben, worden deze samen besproken, waarna de gebruiksintensiteit aan de orde komt.

Uitgebreidheid en complexiteit

Het eerste twee aspecten zijn de uitgebreidheid en de complexiteit van het systeem. Het is aannemelijk dat, naarmate de functionaliteit van een systeem uitgebreider en complexer is, de kans op defecten groter is. Deze aspecten hebben dus betrekking op het ontstaan van defecten. Er zijn verschillende factoren die invloed uitoefenen op de uitgebreidheid en complexiteit (en daarmee de functionaliteit) van software, zoals de gekozen systeemarchitectuur, de gebruikte programmeertaal en de aanwezige ontwikkelomgeving.

Gebruiksintensiteit

Het derde en laatste aspect is de gebruiksintensiteit. De mate van intensiteit waarmee een systeem wordt gebruikt, heeft geen invloed op de hoeveelheid defecten dat in het systeem aanwezig is[5]. Het is echter aannemelijk dat, naarmate een systeem intensiever en gevarieerder wordt gebruikt, de in het systeem aanwezige defecten sneller geconstateerd worden. Onder gevarieerder wordt een groter aantal verschillende combinaties van invoervariabelen en na elkaar uitgevoerde combinaties van aanroepen bedoeld. Een tweetal factoren, dat van invloed is op de gebruiksintensiteit, is het aantal gebruikers en de hoeveelheid interactie (het aantal aanroepen) tussen deze gebruikers en de systemen. Daarnaast is de hoeveelheid systemen waarin een generiek component wordt gebruikt, van invloed op deze gebruiksintensiteit.

Omdat defecten in veelgebruikte systemen storender zijn dan defecten in minder intensief gebruikte systemen, vermoeden we dat de onderhoudsintensiteit in sterke mate positief is gerelateerd aan de gebruiksintensiteit. Een hoge gebruiksintensiteit leidt dus tot een onderhoudsintensiteit.

Samenvattend zijn we op zoek naar een conventioneel systeem en een CBD-systeem die met betrekking tot de aspecten functionaliteit en gebruiksintensiteit vergelijkbaar zijn. Deze vergelijkbaarheid kan direct zijn, waarbij de functionaliteit en de gebruiksintensiteit rechtstreeks met elkaar overeenkomen. Indien er geen directe vergelijking mogelijk is doordat deze aspecten te verschillend zijn, is het wellicht mogelijk om ze na het uitvoeren van een correctie te vergelijken.

5.3.2 Selectie systemen

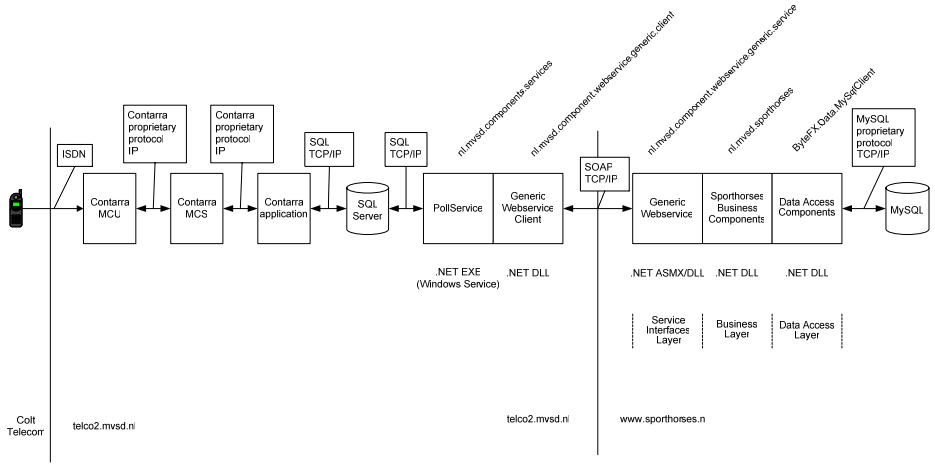
Uit alle systemen die beschikbaar waren binnen het bedrijf waar ik onderzoek heb gedaan, zijn twee systemen geselecteerd die met betrekking tot de criteria functionaliteit en gebruiksintensiteit zoveel mogelijk overeenkomen. Deze selectie wordt in het vervolg van deze sectie beschreven.

CBD-systeem

Bij de selectie van het systeem dat is ontwikkeld met behulp van CBD deed zich een probleem voor, aangezien de toepassing van generieke componenten pas recentelijk is ingevoerd, was er slechts één systeem dat aan dit criterium voldeed. Kortom, er viel weinig te selecteren en we waren genoodzaakt het betreffende systeem te kiezen als CBD-systeem. Het betreffende systeem dient ter ondersteuning van een ander systeem dat klanten in staat stelt advertenties te plaatsen op een website. Het gekozen CBD-systeem stelt de klanten in staat om een eerder geplaatste advertentie bovenaan te plaatsen. De klant belt naar een 0900-nummer en toetst een via de e-mail ontvangen code, behorende bij zijn advertentie, in.

[5] We gaan er hierbij van uit, dat het systeem niet zodanig zwaar wordt belast, dat het uitsluitend vanwege deze belasting faalt.

Het systeem bestaat uit een aantal verschillende onderdelen, welke in Figuur 17 zijn weergegeven[6].



Figuur 17: Architectuur CBD-systeem

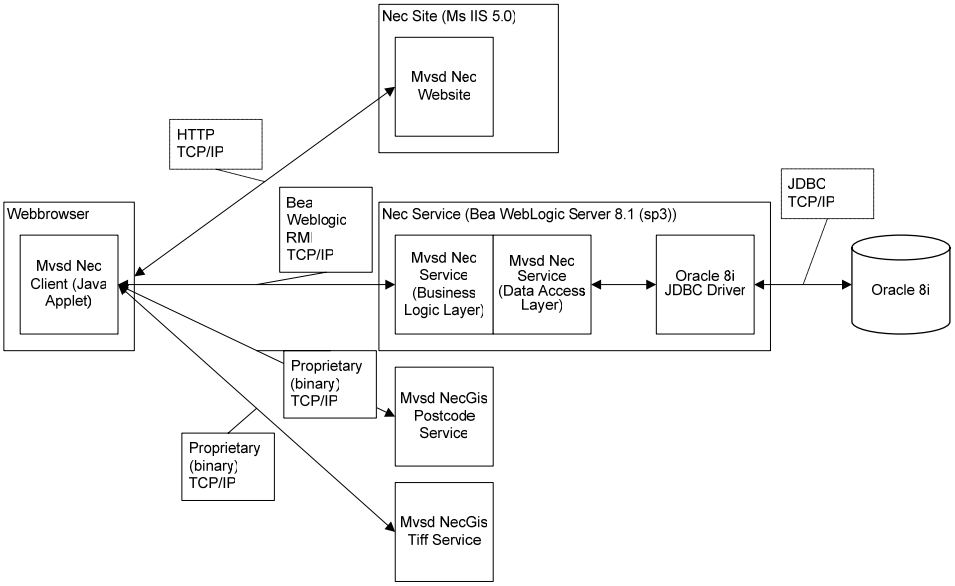
De voice-response-applicatie ontvangt het telefoontje en de ingevoerde code. De applicatie schrijft deze code weg in een database. Een Windows Service controleert periodiek of er codes zijn ingevoerd in deze database. Indien dit het geval is wordt er voor iedere code een generieke webservice client aangeroepen. Deze client roept via het internet de generieke webservice aan. Dit component roept de server business component aan. Het server business component plaatst de advertentie, door middel van de meegegeven code, bovenaan.

Conventioneel systeem

Bij de selectie van een systeem dat niet ontwikkeld is door middel van CBD waren er een aantal alternatieven. Op basis van de overeenkomst met de architectuur van het CBD-systeem en de beschikbaarheid van gegevens over defecten is hier gekozen voor het systeem NEC. NEC staat voor Nieuw Elektronisch Contract. Dit systeem is gebouwd om het maken van contracten voor telefoniediensten (0800 en 0900 diensten) gemakkelijker en overzichtelijker te maken. Het biedt een grafische interface waarmee de verschillende onderdelen van deze contracten, zoals functionaliteit en klantgegevens, gemakkelijker met elkaar verbonden en ingevuld kunnen worden. De architectuur van NEC is weergegeven in Figuur 18[7].

[6] In Bijlage 2 is een grotere afbeelding van de architectuur van het CBD-systeen opgenomen.

[7] In Bijlage 6 is een grotere afbeelding van de architectuur van het conventionele systeem opgenomen.



Figuur 18: Architectuur conventioneel systeem

Bij het gebruik van het systeem zijn twee fasen te onderkennen. De eerste fase omvat het opstarten, waarbij een Java applet wordt geladen. De tweede fase bestaat uit het gebruik van het systeem via dit applet.

In de opstartfase wordt een Java applet geladen. Het uitvoeringstraject van deze fase verloopt als volgt [van den Berg et al., 1998]: De gebruiker opent een HTML-pagina in zijn Java-enabled webbrowser. Achter de schermen stuurt de browser een verzoek om de pagina naar de webserver. Deze stuurt hierop de HTML-pagina, waarin zich een verwijzing naar een Java applet (een class-bestand) bevindt, terug naar de browser. Bij het laden van de ontvangen pagina ziet de browser de verwijzing naar het applet en stuurt de browser een verzoek om dit applet naar de webserver. De webserver stuurt hierop het applet terug naar de webbrowser. Deze browser ontvangt het applet, waarna de Java Virtual Machine (JVM) binnen de webbrowser de bytecode van het applet transformeert in uitvoerbare code en het applet uitvoert.

In de tweede fase maakt de gebruiker gebruik van het systeem via het applet, dat met de EJB-server communiceert. Deze communicatie vindt plaats over het internet[8], gebruikmakend van het RMI-protocol. De EJB-server slaat zijn gegevens op in een Oracle 8i database.

[8] Feitelijk verloopt de communicatie via een WAN dat niet door buitenstaanders te gebruiken is. Er is dus geen sprake van een openbaar netwerk, zoals het internet. Omdat er bij de communicatie gebruik gemaakt wordt van dezelfde communicatieprotocollen als die waar het internet gebruik van maakt, spreken we voor het gemak toch over internet.

5.3.3 Beoordeling vergelijkbaarheid

De vergelijkbaarheid van de geselecteerde systemen is beoordeeld op basis van de in sectie 5.3.1 opgestelde criteria, te weten uitgebreidheid, complexiteit en gebruiksintensiteit.

Uitgebreidheid

Met betrekking tot de uitgebreidheid kan direct gesteld worden dat deze niet geheel vergelijkbaar is. Het conventionele systeem heeft een vrij uitgebreide functionaliteit, terwijl deze bij het CBD-systeem zeer beperkt is. Met andere woorden, het conventionele systeem is rijk aan features, terwijl het CBD-systeem dit niet is. Hierdoor komt de vergelijkbaarheid in gevaar.

Het verschil in uitgebreidheid uit zich nog op een andere wijze. Door de uitgebreidere functionaliteit van het conventionele systeem ten opzichte van het CBD-systeem, is dit conventionele systeem beduidend groter. Indien we de systemen toch willen vergelijken, zullen we een correctie moeten toepassen voor dit verschil.

Complexiteit

Met betrekking tot de complexiteit kunnen we stellen dat de twee systemen voldoende vergelijkbaar zijn. Beide systemen hebben een gedistribueerde architectuur die het systeem verdeeld over een aantal verschillende servers. Beide systemen worden gebruikt over het internet en bij beide systemen wordt gebruik gemaakt van verschillende samenwerkende technologieën (het conventionele systeem maakt gebruik van EJB's en applets, het CBD-systeem maakt gebruik van een telefonieapplicatie, een Windows service, een webservice en een Windows applicatie).

Doordat de systemen van vergelijkbare complexiteit zijn, stellen we dat de systemen toch vergelijkbaar zijn, mits het onderkende verschil in grootte gecompenseerd wordt.

Gebruiksintensiteit

Voordat we ingaan op de vergelijkbaarheid van de gebruiksintensiteit van beide systemen, moet worden opgemeld dat gebruiksintensiteit lastig is vast te stellen. Dit komt met name doordat er geen gegevens voorhanden zijn met betrekking tot bijvoorbeeld de hoeveelheid processortijd die door beide systemen afzonderlijk wordt gebruikt. De systemen draaien op externe servers, waarvan het operationeel beheer niet door MVSD wordt uitgevoerd. Zelfs als dat wel het geval zou zijn, draaien er meerdere systemen naast elkaar op dezelfde servers, zodat het aandeel van de twee onderzochte systemen nog steeds moeilijker vast te stellen zou zijn.

Desondanks proberen we vast te stellen of de gebruiksintensiteit vergelijkbaar is. We hebben in sectie 5.3.1 de aspecten aantal gebruikers, hoeveelheid interactie en hoeveelheid systemen waarin de componenten zijn opgenomen genoemd. Het aantal gebruikers van beide systemen komt in grote lijnen overeen. De hoeveelheid interactie is

bij het conventionele systeem een stuk hoger dan bij het CBD-systeem. Per gebruiker wordt het systeem veel vaker benaderd. Tot slot zijn er bij het conventionele systeem geen generieke componenten die in andere systemen worden toegepast. De generieke componenten van het CBD-systeem worden naast dit systeem wél in een aantal andere systemen toegepast.

Samengevat is het aantal gebruikers vergelijkbaar. De hoeveelheid interactie is bij het conventionele systemen hoger dan bij het CBD-systeem, wat ertoe leidt dat defecten bij het conventionele systeem vermoedelijk sneller zullen worden gevonden. Daarentegen is het aantal generieke componenten dat naast de bestudeerde systemen in andere systemen wordt gebruikt, bij het CBD-systeem hoger dan bij het conventionele systeem. Dit laatste leidt ertoe dat defecten in generieke componenten van het CBD-systeem mogelijk weer sneller zullen worden geconstateerd dan bij het conventionele systeem. De laatste twee aspecten hebben een tegengesteld effect, zodat ze elkaar in sterke mate opheffen. Ondanks het feit dat de gebruiksintensiteit niet gelijk is, stellen we – op basis van bovenstaande argumenten – dat deze wel als voldoende vergelijkbaar beschouwd kan worden.

Conclusie

Op basis van de vergelijkbaarheid van de aspecten uitgebreidheid, complexiteit en gebruiksintensiteit stellen we dat de systemen ondanks de verschillen vergelijkbaar zijn, mits correctie plaats vindt ten aanzien van het aspect van 'uitgebreidheid' van de software. Hierbij dient te worden aangetekend dat we het hierbij hebben over een voldoende mate van vergelijkbaarheid in het kader van dit verkennende onderzoek. Om deze reden moet extra zorg worden betracht bij het generaliseren van de uitkomsten van het onderzoek. Deze generalisaties zijn algemene uitspraken over de invloed van CBD op de betrouwbaarheid.

5.4 Operationalisering betrouwbaarheid

Nu duidelijk is hoe meten in het algemeen in zijn werk gaat, gaan we de algemene begrippen van de meettheorie toepassen op de betrouwbaarheid van software. We willen weten wat de invloed van CBD op de betrouwbaarheid van software is. De entiteit is dus het softwaresysteem en de eigenschap waar we naar kijken is de betrouwbaarheid van dit systeem.

Een omschrijving van het begrip betrouwbaarheid is de volgende.

> Software reliability concerns the frequency with which a software product fails when it is being used – a concept of understandable importance to the user.

Citaat 5: Software reliability [Kitchenham, 1996]

De betrouwbaarheid heeft te maken met de frequentie waarmee de software faalt tijdens het gebruik. Het falen van een systeem is het niet voldoen aan de bedoelde werking van het systeem. Deze werking wordt de functionaliteit van het systeem genoemd. De bedoelde werking is gespecificeerd in het functionele ontwerp. Hierbij wordt voor het gemak aangenomen dat deze functionele specificatie volledig en compleet is[9]. Voorbeelden van het falen van een systeem zijn het uitvoeren van een wiskundige berekening waarbij een onjuiste waarde wordt teruggegeven of het vastlopen van een dialoogscherm. Indien een systeem faalt, is er sprake van het niet voldoen aan de functionele specificatie.

Falen of *failure* van een systeem kan gezien worden als een mogelijk gevolg van het optreden van een defect [Kan, 1995]. Defecten zijn tekortkomingen binnen het systeem die veroorzaakt zijn door onjuist menselijk handelen. Defecten zijn situaties waarin een afwijking optreedt ten opzichte van de specificaties, welke staan beschreven in het technische ontwerp. Er zijn veel voorbeelden van defecten, zoals het toekennen van een verkeerde waarde aan een variabele, het vergeten een variabele een waarde te geven voordat deze waarde wordt gebruikt, het berekenen van de som in plaats van het verschil of het gebruiken van een onjuist karakter in een tekst. Een defect kan één of meerdere failures tengevolge hebben, maar het is ook mogelijk dat een defect niet direct leidt tot het falen van een systeem, waardoor het defect lang onopgemerkt blijft.

De gegeven omschrijving van de eigenschap betrouwbaarheid is echter niet duidelijk genoeg, dus zullen we het begrip verder verduidelijken. Bij betrouwbaarheid wordt doorgaans onderscheid gemaakt tussen twee verschillende soorten systemen: *safety-critical systemen* en *non-safety-critical systemen* [Kan, 1995].

- Safety-critical systemen zijn systemen wier functioneren essentieel is en waarbij het optreden van defecten de veiligheid in gevaar brengt. Voorbeelden zijn de vluchtleidingscontrolesystemen en systemen die gebruikt worden in vliegtuigen en wapens [Kan, 1995]. Safety-critical systemen zijn vaak real-time systemen. De betrouwbaarheid van deze systemen wordt doorgaans gemeten in mean time between failures (MTBF) of mean time to failure (MTTF), de gemiddelde tijd die verstrijkt voordat een storing optreedt [Kan, 1995].
- Non-safety-critical systemen zijn systemen wier falen de veiligheid niet in gevaar brengt. Hieronder vallen alle overige systemen, waaronder veel commerciële softwaresystemen. Voorbeelden zijn administratieve systemen en kantoorautomatiseringssystemen. De betrouwbaarheid van deze systemen wordt doorgaans gemeten in het aantal defecten dat zich in de software bevindt, afgezet tegen de grootte van de software [Kan, 1995].

---

[9] In de praktijk blijkt dat er vaak onduidelijkheid bestaat over de functionele specificatie tussen de opdrachtgever en de producent van de software. Een belangrijke reden hiervoor is, dat de opdrachtgever en de producent beide impliciete veronderstellingen hebben over vanzelfsprekende functionaliteit (algemeen geaccepteerde verwachtingen), maar dat deze niet volledig met elkaar overeen komen. Het volledig en compleet maken van de functionele specificatie is een langdurig proces en vaak wordt een compromis gesloten tussen volledigheid en beknoptheid.

Er zijn twee redenen waarom MTBF en MTTF niet vaak worden gebruikt bij non-safety-critical systemen [Kan, 1995].

- Er is bij deze systemen in tegenstelling tot systemen uit de eerste categorie veel minder sprake van een strak gedefiniëerd gebruiksprofiel. Een gebruikersprofiel is de verzameling operaties die uitgevoerd kan worden door de software samen met de kans waarmee elk aangeroepen wordt [Lyu, 1996]. De meeste safety-critical systemen hebben een gebruiksprofiel waarbinnen weinig variatie voorkomt: iedere gebruiker van het systeem maakt gebruik van ongeveer dezelfde operaties met vergelijkbare frequenties. Gemeten waarden van MTBF en MTTF zijn hierdoor representatief voor de totale gebruikersgroep. Systemen uit de andere categorie hebben vaak totaal verschillende soorten gebruikers, die de systemen op een heel andere manier gebruiken. Een voorbeeld hiervan is een tekstverwerkingsprogramma dat wordt gebruikt door professionele editors en door personen die incidenteel een brief typen. Door het verschil in gebruik is hier minder sprake van een uniform gebruiksprofiel en zegt een gemeten MTBF of MTTF niet zoveel over een specifieke gebruiker.
- Het verzamelen van gegevens die het vaststellen van MTTF of MTBF mogelijk maakt, is veel duurder dan het verzamelen van het aantal defecten, omdat élke failure vastgelegd moet worden en de gegevens veel nauwkeuriger moeten zijn.

De twee systemen die we onderzoeken bevinden zich beide in de tweede categorie, te weten non-safety-critical systemen. Als gevolg hiervan gaan we de betrouwbaarheid meten in het aantal defecten ten opzichte van de softwaregrootte.

De betrouwbaarheid van software wordt gemeten in het aantal defecten ten opzichte van de grootte van de software.

Operationalisering 1: betrouwbaarheid van software

Deze eerste operationalisering bevat twee elementen die verder geoperationaliseerd moeten worden, te weten het aantal defecten en de softwaregrootte.

5.4.1 Het aantal defecten

Bij de vaststelling van het aantal defecten moet vastgesteld worden wat precies onder een softwaredefect wordt verstaan. Hierbij wordt eerst vastgesteld wat in het algemeen onder defecten wordt verstaan, daarna wordt vastgesteld welke defecten we wél en welke defecten we niet meetellen.

Voor de vaststelling van de soorten defecten die we wel en niet meetellen is het noodzakelijk de defecten in te delen in de fase van het ontwikkelproces waarin het defect wordt geconstateerd en in welke fase het is opgelost. Defecten kunnen in elke fase van

het ontwikkelproces ontstaan. Vaak komen deze defecten naar voren als gevolg van controleactiviteiten die in het ontwikkelproces worden uitgevoerd. Voorbeelden van deze controleactiviteiten en de fasen waarin ze voorkomen zijn inspecties van het ontwerp in de ontwerpfase en code reviews en unit tests in de ontwikkelfase. Ook is het mogelijk dat deze defecten niet worden opgemerkt door controleactiviteiten en zich nog bevinden in het systeem dat wordt opgeleverd aan de gebruiker.

Het gaat ons om de betrouwbaarheid van het systeem in de ogen van de gebruiker. Dit houdt in dat we uitsluitend kijken naar defecten die voorkomen in versies van de systemen die gebruikt worden door eindgebruikers. Dit criterium leidt ertoe dat defecten die worden ontdekt en opgelost tijdens het ontwikkelingsproces voorafgaand aan de fase van oplevering niet meetellen.

Deze constatering leidt tot de volgende, specifiekere operationalisering van het begrip betrouwbaarheid.

De betrouwbaarheid van software wordt gemeten in het aantal defecten dat geconstateerd wordt in de versie van de software die is opgeleverd aan eindgebruikers ten opzichte van de grootte van de software.

Operationalisering 2: Betrouwbaarheid van software

Issue tracking systeem

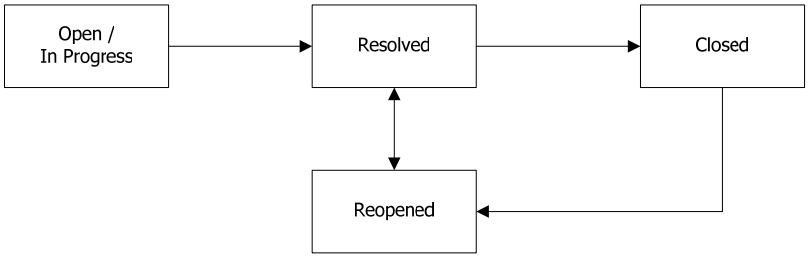
Bij de vaststelling van de geconstateerde defecten maken we gebruik van het issue tracking systeem Jira van Atlassian Software, dat vanaf medio mei 2003 wordt gebruikt. Dit systeem stelt gebruikers in staat om *issues* bij te houden, onderwerpen waarvan het belangrijk is om de voortgang ervan te volgen. Binnen het systeem wordt onderscheid gemaakt tussen de volgende typen issues.

- Bugs — Problemen die de functionaliteit van het systeem schaden of beperken.
- Improvements — Verbeteringen van bestaande functionaliteit.
- New Features — Nieuwe eigenschappen die nog niet in het product aanwezig zijn.
- Problems — Situaties die onwenselijk zijn, maar waarvan nog niet is vastgesteld hoe deze opgelost dienen te worden.
- Tasks — Activiteiten die uitgevoerd dienen te worden (zoals research).
- Questions — Situaties waarover onduidelijkheid bestaat (zoals het al dan niet voldoen aan een specifieke standaard).

Aan deze issues is een status gekoppeld, welke gewijzigd kan worden. De volgende statussen worden onderscheiden.

- Open — Het issue is nog niet beëindigd.
- In Progress — Er wordt aan het issue gewerkt.
- Resolved — Het issue is opgelost en dient te worden geverifieerd.
- Reopened — Het issue was opgelost, maar de oplossing blijkt tijdens verificatie niet te voldoen of het issue doet zich na verloop van tijd opnieuw voor.
- Closed — Het issue is opgelost en geverifieerd.

Over het algemeen heeft een issue initieel de status 'Open'. Zodra eraan gewerkt wordt, kan de status 'In Progress' aan het issue worden toegekend, maar dit komt in de praktijk zelden voor. Zodra het issue beëindigd is, wordt de status 'Resolved' eraan toegekend. Vervolgens kan het issue worden heropend of gesloten. Ook kan het indien het al gesloten is opnieuw worden geopend. De mogelijke statusovergangen zijn weergegeven in Figuur 19.



Figuur 19: Statusovergangen issue tracking system

Bij het oplossen of sluiten van een issue wordt één van de volgende oplossingen gespecificeerd.

- Fixed — Het issue is opgelost.
- Won't Fix — Het issue kan nu en in de toekomst niet worden opgelost.
- Duplicate — Het issue is een duplicaat van een reeds bestaand issue.
- Incomplete — De gegevens in het issue zijn onvoldoende om het op te lossen.
- Cannot Reproduce — Het issue is niet reproduceerbaar.
- Rejected — Het issue is geweigerd (bijvoorbeeld omdat het gaat om een gebruikersfout).
- Deferred — Het oplossen van het issue is uitgesteld.

Bij de vaststelling van het aantal defecten kijken we telkens naar een gegeven versie van het systeem. Van dit systeem tellen we issues van het type 'Bug' met de status 'Open',

'In Progress' of 'Reopened', omdat dit de enige bekende defecten zijn die in deze versie bekend en onopgelost zijn.

> Het aantal defecten wordt bepaald door de issues van het type 'Bug' met de status 'Open', 'In Progress' of 'Reopened' uit het issue tracking systeem per versie van een softwaresysteem vast te stellen.

Operationalisering 3: Aantal defecten per systeemversie

Een moeilijkheid bij het verwerken van de issues is, dat er soms twee issues worden ingevoerd alszijnde één issue. Een voorbeeld is een issue dat is ingevoerd als type Bug, waarbij in de beschrijving een oplossing is aangegeven die de functionaliteit van het systeem een klein beetje wijzigt. Daar waar dit mogelijk was, zijn deze dubbele issues gesplitst.

5.4.2 Prioriteiten

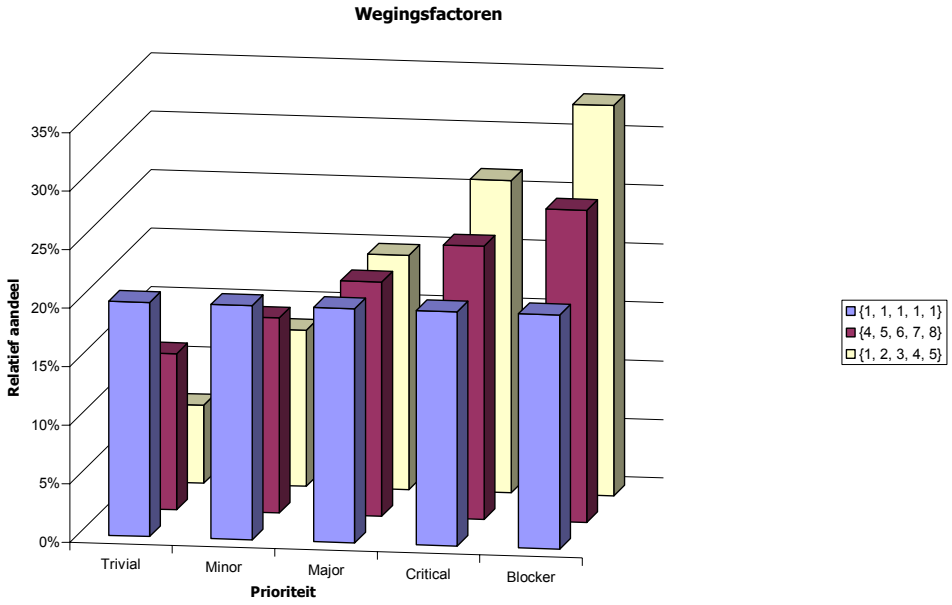
De issues worden in het issue tracking systeem vastgelegd met een prioriteit. Hierbij wordt een ordening gebruikt van minst belangrijk naar meest belangrijk van de prioriteiten 'Trivial', 'Minor', 'Major', 'Critical' en 'Blocker'. Door deze ordening van categorieën spreekt men over een ordinale schaal.

Een belangrijke eigenschap van ordinale schalen is dat er onderscheid wordt gemaakt tussen prioriteiten en dat er een ordening bestaat tussen minder belangrijk en meer belangrijk. Wat de mate van het verschil in belangrijkheid is, wordt niet vastgesteld. Zo staat bijvoorbeeld niet vast dat een issue uit een categorie 'Major' twee keer zo belangrijk is als een issue uit de categorie 'Minor'.

Om het verschil in belang van de verschillende prioriteiten mee te nemen in het onderzoek, maken we gebruik van wegingsfactoren. Hiermee stellen we arbitrair vast hoeveel belangrijker een issue van een prioriteit is ten opzichte van een issue van een andere prioriteit. Het is van groot belang dat we ons blijven realiseren dat de resulterende waarden beïnvloed worden door de arbitrair gekozen wegingsfactoren. Een andere verzameling wegingsfactoren leidt vermoedelijk tot andere waarden.

Om te komen tot enigszins betrouwbare waarden, maken we gebruik van een drietal verschillende verzamelingen wegingsfactoren voor de prioriteiten 'Trivial', 'Minor', 'Major', 'Critical' en 'Blocker'. De wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\}$ drukken de verhouding van het belang uit van de prioriteit, $w_{Trivial}:w_{Minor}:w_{Major}:w_{Critical}:w_{Blocker}$, uit.

Deze gekozen verzamelingen wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\}$ zijn respectievelijk {1, 1, 1, 1, 1}, {4, 5, 6, 7, 8} en {1, 2, 3, 4, 5}. In Figuur 20 wordt een grafische weergave gegeven van het relatieve belang van deze wegingsfactoren.



Figuur 20: Relatieve aandelen wegingsfactoren

De redenering achter deze keuze is als volgt. De verzameling $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 1, 1, 1, 1\}$ is de situatie wanneer er geen onderscheid gemaakt zou worden tussen de prioriteiten. Alle prioriteiten tellen even zwaar mee. Dit is als het ware de ondergrens voor weging. Bij de verzameling $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{4, 5, 6, 7, 8\}$ is de toename bij iedere opvolgende prioriteit constant en telt de meest belangrijke prioriteit twee keer zo zwaar mee als de minst belangrijke prioriteit. Bij de verzameling $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 2, 3, 4, 5\}$ is de toename bij iedere opvolgende prioriteit ook constant, maar hier telt de belangrijkste prioriteit vijf maal zo zwaar mee als de minst belangrijke prioriteit. Indien de belangrijkste prioriteit nog zwaarder mee zou tellen, zou het aandeel van de lagere prioriteiten te klein worden.

5.4.3 De grootte van software

Voor de eenheid waarin de grootte van de software wordt uitgedrukt zijn verschillende mogelijkheden. Er zijn maatstaven die vroeg in het ontwikkelproces gebruikt kunnen worden en die om die reden gebruikt kunnen worden voor het schatten van de definitieve grootte, zoals Albracht function points, Mark II function points en DeMarco's Bang metrics [Kitchenham, 1996]. Kritiek op deze maatstaven is de hoeveelheid training die nodig is om deze maatstaven goed te kunnen gebruiken en de subjectiviteit die ermee gepaard gaat [Kitchenham, 1996].

Maatstaven die laat in het ontwikkelproces gebruikt kunnen worden en iets zeggen over de code worden ondergebracht in de categorie 'code measures' [Kitchenham, 1996]. De naam zegt al dat deze maatstaven een indicatie geven van de grootte van code. Drie maatstaven die in deze categorie vallen zijn '*lines of code*', '*bytes of object code*' en *'lexical elements count'* [Kitchenham, 1996]:

- 'Lines of Code' (LOC) is een maatstaf waarbij het aantal regels in de broncode wordt geteld. Een voordeel van deze maatstaf is de eenvoudige wijze waarop deze gemeten kan worden. Een nadeel is de eenvoudige wijze waarop de programmeerstijl de maatstaf kan beïnvloeden, bijvoorbeeld door een instructie in de code over meerdere regels te verspreiden. Een aandachtspunt bij deze maatstaf is dat het begrip LOC ondubbelzinnig omschreven moet worden [Kitchenham, 1996].
- De maatstaf 'bytes of object code' meet de grootte van de software nadat deze door een compiler is omgezet in uitvoerbare code. Een voordeel van deze maatstaf is dat deze eenvoudg te verzamelen is [Kitechenham, 1996].
- De 'lexical elements count' telt het aantal elementen waaruit de broncode bestaat. Een nadeel van deze maatstaf is dat deze relatief lastig te verzamelen is en niet veel extra informatie oplevert [Kitchenham, 1996].

Genoemde voor- en nadelen hebben geleid tot een keuze voor de maatstaf LOC. De reden voor het afvallen van de maatstaf 'lexical elements count' is het argument dat deze relatief lastig te verzamelen is, terwijl de praktische toepasbaarheid van de toets in sterke mate meespeelt. Het belangrijkste argument voor het afvallen van de maatstaf 'bytes of object code' is het feit dat er bij de twee onderzoeksprojecten gebruik is gemaakt van twee verschillende programmeertalen, te weten Java en C#. Ondanks dat beide talen in grote mate overeen komen, zal gelijke functionaliteit vermoedelijk toch resulteren in verschillende aantallen bytes of object code. Hierdoor is afgezien van het gebruik van deze maatstaf. De maatstaf LOC is de gemeenschappelijke noemer voor het meten van de softwaregrootte [Kan, 1995] en deze maatstaf is ondanks zijn tekortkomingen een betrouwbare indicator voor de globale geleverde inspanning [Poulin, 1996]. Het nadeel van beïnvloeding van de maatstaf door aanpassing van de programmeerstijl is slechts in beperkte mate aanwezig, omdat de meting achteraf, dus na ontwikkeling van de onderzochte systeemversies, heeft plaatsgevonden en de resultaten van de meting niet tijdens de ontwikkeling beschikbaar waren. Op grond hiervan gaan we ervan uit dat LOC een goede maatstaf is voor de grootte van de software mits we het begrip ondubbelzinnig omschrijven.

Er worden veel verschillende variaties gebruikt voor het tellen van LOC. Jones onderkent de volgende soorten [Kan, 1995]:

- only executable lines;
- executable lines and data definitions;
- executable lines, data definitions and comments;
- executable lines, data definitions, comments and job control language;
- physical lines on an input screen;
- lines terminated by logical delimiters.

Er wordt onderscheid gemaakt tussen *fysieke regels* en *logische regels* code [Kan, 1995].

- Een fysieke regel code is een regel code welke loopt van het begin van een regel tot het eerste newline karakter. Het begin van een regel is hierdoor het eerste karakter volgend op het vorige newline karakter.
- Een logische regel code loopt van het begin van een instructie tot het einde van een instructie. In veel programmeertalen, zoals C, C++, Java en C# wordt dit einde aangegeven met het karakter ';'. Een voorbeeld van een bedrijf dat de softwaregrootte uitdrukt in het aantal logische LOC is IBM [Kan, 1995].

Vanwege de eenvoud waarmee de metingen uit te voeren zijn kiezen we in het kader van deze scriptie voor fysieke regels. De gehanteerde definitie van LOC is gegeven in Definitie 18. Een regel met zowel code als commentaar wordt door deze definitie meegeteld als een coderegel.

Een LOC is een regel broncode die niet leeg is en niet uitsluitend commentaar bevat.

Een regel broncode loopt vanaf het eerste karakter na het vorige newline karakter tot het eerste newline karakter.

Een lege regel is een regel broncode die uitsluitend spaties en tabs bevat.

Definitie 18: Line of Code (LOC)

Voor een individueel broncodebestand is nu vast te stellen uit hoeveel regels code dit bestand bestaat. Hiermee weten we echter nog niet de grootte van het totale systeem. Hiertoe maken we gebruik van het volgende model [Fenton, 1991].

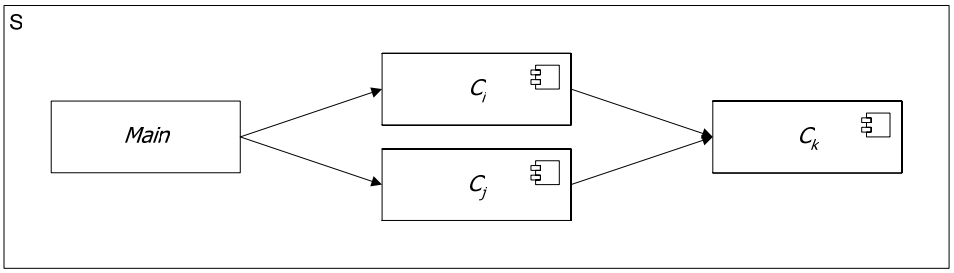
Zij S een systeem dat is ontstaan door één of meer bestanden met broncode te compileren, waarbij mogelijk componenten worden aangeroepen. Stel dat er een functie *SourceFiles* bestaat die van systeem S de verzameling bestanden met broncode teruggeeft. Stel dat er tevens een functie *Components* bestaat die van systeem S de verzameling van afzonderlijke componenten teruggeeft, zoals weergegeven in Vergelijking 1.

$$Components(S) = \{C_1, \ldots, C_n\}$$

Vergelijking 1: Components(S)

Dan bevat *Components(S)* alle componenten die potentieel worden aangeroepen door S. Dit wordt geïllustreerd aan de hand van Figuur 21. Hierbinnen roept de main functie van S mogelijk de componenten C_i en C_j aan. Deze mogelijkheid is het gevolg van de

afhankelijk van de invoer van de gebruiker of de configuratie van het systeem. Beide componenten roepen op hun beurt mogelijk component C_k aan. De aanroep van functie *Components* met argument *S* levert de verzameling $\{C_i, C_j, C_k\}$ op. Essentieel hierbij is dat het gebruik van functie *Components* er hierbij voor zorgt dat componenten, die potentieel meerdere malen worden gebruikt, nooit meer dan één keer worden meegeteld.



Figuur 21: aanroep componenten binnen S

Stel dat er tevens een functie *Size* bestaat die van een systeem *S* de grootte bepaalt, zoals weergegeven in Vergelijking 2. Deze functie telt de grootte van de bronbestanden van *S* en de grootte van de componenten die *S* potentieel aanroept.

$$Size(S) = \sum_{sf \in SourceFiles(S)} Size(sf) + \sum_{c \in Components(S)} Size(c)$$

Vergelijking 2: Size(S)

5.4.4 Additionele restricties

De nauwkeurigheid van de gemeten betrouwbaarheid is in essentiële mate afhankelijk van de grootte van de software en de geconstateerde defecten. Hiertoe is informatie nodig over zowel de grootte van de broncode als de geconstateerde defecten. Bij het zoeken naar deze gegevens over componenten van derden ('third party components') bleek dat deze gegevens geheel niet of slechts in zeer beperkte mate beschikbaar worden gesteld. Op grond hiervan is besloten om het onderzoek te beperken tot code die in eigen beheer is ontwikkeld, omdat hierdoor de beschikbaarheid van de benodigde gegevens met zekerheid is gegarandeerd.

5.5 Betrouwbaarheidstrend

Tot nu toe hebben alle geïntroduceerde begrippen van de toets betrekking gehad op statische eigenschappen. Dit zijn eigenschappen die van toepassing zijn op een gegeven versie van een softwaresysteem. In hoofdstuk 2 hebben we gezien dat softwareontwikkeling ook betrekking heeft op het onderhouden van software, waarbij

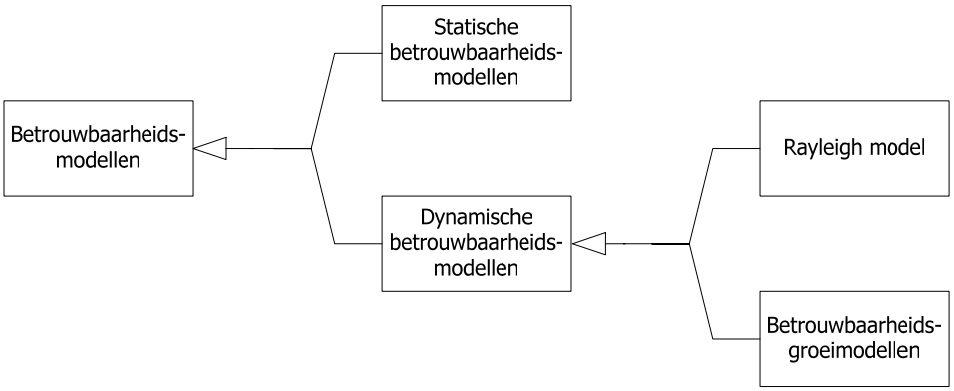
onderscheid werd gemaakt tussen adaptief, perfectief en correctief onderhoud. Dit onderhoud heeft ten gevolg dat de eigenschappen die de betrouwbaarheid van het systeem bepalen, de grootte en het aantal defecten, zal veranderen. Met betrekking tot deze eigenschappen kan het volgende worden vastgesteld.

- De grootte van de software zal wijzigen door onderhoud. Uitbreidingen van het systeem zullen doorgaans leiden tot meer code. Onderhoud van de code kan echter ook leiden tot minder code, bijvoorbeeld doordat de huidige code wordt geherstructureerd zonder de functionaliteit te veranderen. Dit laatste staat bekend als 'refactoring' [Fowler et al., 1999].
- Het aantal defecten in de code zal mogelijk veranderen door het onderhoud. Defecten die zich in de code bevinden kunnen worden vastgesteld, wat een stijging van het aantal geconstateerde defecten in de huidige versie tot gevolg heeft. Bij corrigerend onderhoud worden geconstateerde defecten verwijderd, wat leidt tot een lager aantal defecten in de volgende versie. Hierbij dient te worden aangetekend dat iedere toevoeging aan de code of wijziging in de code het risico van introductie van nieuwe defecten met zich meebrengt, een kenmerk dat bekend staat als 'defect injection' [Humphrey, 1994a], [Humphrey, 1994b], [Kan, 1995].

Vanwege de rol die de verschillende soorten onderhoud spelen, moet er onderscheid worden gemaakt tussen correctief onderhoud enerzijds en adaptief en perfectief onderhoud anderzijds.

- Correctief onderhoud is het verwijderen van defecten. Deze vorm van onderhoud leidt over het algemeen niet tot een substantiële toename van de grootte, maar wel tot een afname van het aantal defecten.
- Adaptief en perfectief onderhoud leidt tot een substantiële toename van de grootte. Tevens leidt het doorgaans tot een toename van het aantal defecten vanwege het verschijnsel 'defect injection'.

Modellen die de betrouwbaarheid van software beschrijven, worden betrouwbaarheidsmodellen genoemd. Deze modellen schatten de betrouwbaarheid van software die beschikbaar wordt gesteld aan gebruikers. [Kan, 2002] Figuur 22 geeft een onderverdeling van deze modellen weer. Betrouwbaarheidsmodellen zijn te classificeren in twee categorieën, te weten statische modellen en dynamische modellen.



Figuur 22: Categorieën betrouwbaarheidsmodellen

Bij statische modellen wordt de betrouwbaarheid van de software verklaard aan de hand van eigenschappen van het softwareproduct, het ontwikkelingsproces of het projectteam, zoals grootte, aantal bestede uren, opleidingsniveau en percentage overwerkuren. Er wordt gebruik gemaakt van gegevens van vorige – vergelijkbare – projecten om de coëfficiënten in het model te schatten.

Bij dynamische modellen worden de parameters daarentegen geschat op basis van meetgegevens die tot nu toe zijn verzameld van het softwareproduct dat wordt onderzocht. Het resulterende model is hierom specifiek voor de software die wordt onderzocht.

Dynamische modellen zijn in het algemeen beter geschikt dan statische modellen wanneer er analyse plaatsvindt op productniveau met als doel het vaststellen van de betrouwbaarheid van software [Kan, 2002]. Daarom beperken we ons tot betrouwbaarheidsmodellen uit de categorie dynamische modellen. Hiervan bestaan weer twee categorieën, te weten modellen die het gehele ontwikkelproces bekijken en modellen die uitsluitend kijken naar de formele testfase. Een model uit de eerste categorie is het Rayleigh model. Modellen uit de tweede categorie worden betrouwbaarheidsgroeimodellen genoemd.

Omdat binnen het bedrijf waarin dit onderzoek wordt uitgevoerd gedurende de onderzoeksperiode defecten in de fasen voorafgaand aan de testfase vrijwel niet zijn vastgelegd, beperken we ons tot de categorie betrouwbaarheidsgroeimodellen.

De naam van deze categorie is afkomstig van de werkwijze na ontwikkeling van de software. Indien de software in deze periode faalt, worden de defecten die hiertoe leiden geidentificeerd en opgelost. Omdat er slechts een eindig aantal defecten in de software zit en er over het algemeen bij het oplossen van defecten niet méér defecten worden geintroduceerd dan er worden opgelost, neemt het aantal defecten af en hierdoor de betrouwbaarheid toe.

Het exponentiële model is het meest eenvoudige en meest belangrijke betrouwbaarheidsgroeimodel [Kan, 2002]. Hierdoor is dit model geschikt voor dit verkennende onderzoek. Allereerst definiëren we de Defect Density (DD) als volgt:

Defect Density (DD) = # defecten / LOC

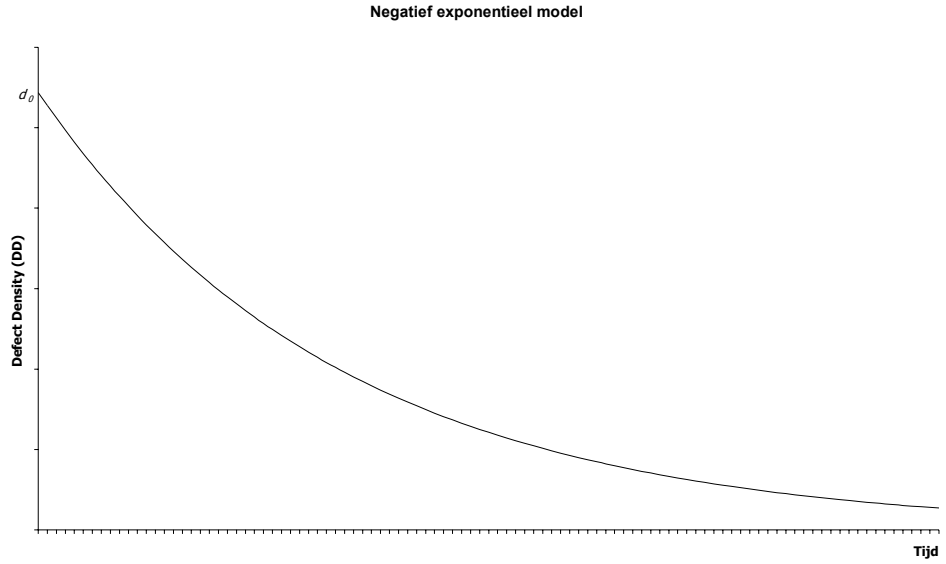
Operationalisering 4: Defect Density (DD)

De algemene vorm van het exponentiële model bestaat uit de in Formule 1 gegeven negatief exponentiële functie.

$$d(t) = d(0)e^{-\alpha t} = d_0 e^{-\alpha t}; \alpha > 0, t \geq 0$$

Formule 1: Negatief exponentieel model

Vanwege het minteken in de macht en de niet-negatieve waarden van zowel α als *t*, spreken we over een negatief exponentiële functie. De functie *d*(*t*) geeft de DD aan op tijdstip *t*, bij een constante waarde van α. Figuur 23 bevat een grafische weergave van het model.



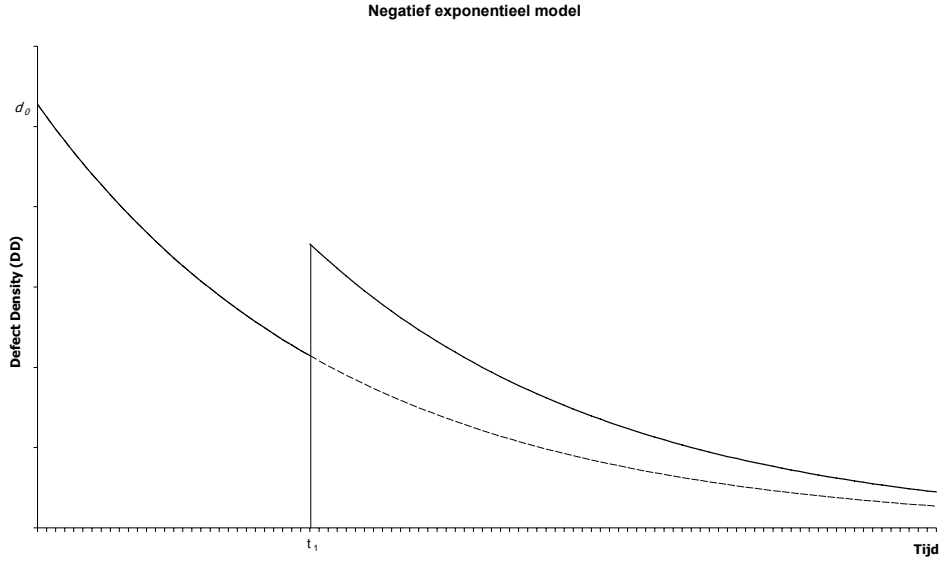
Figuur 23: Negatief exponentieel model

De onderliggende aanname van dit model is, dat bij toepassing van correctief onderhoud, de afname van de defect dichtheid – en daarmee met het aantal defecten – op elk tijdstip (voor elke *t*) evenredig is met het aantal aanwezige fouten. Dit brengt ons tot de volgende aanname.

Er wordt uitsluitend correctief onderhoud uitgevoerd op de software binnen de onderzoeksperiode.

Aanname 1: Geen adaptief en perfectief onderhoud

Indien niet aan Aanname 1 is voldaan, kan de functie een discontinuiteit vertonen. Dit kan als volgt worden geïllustreerd. Stel dat op tijdstip t_1 een grote aanpassing aan het systeem plaatsvindt, bestaande uit nieuwe en/of gewijzigde functionaliteit. Er is dan sprake van adaptief en/of perfectief onderhoud. Hierbij is veel code toegevoegd en aangepast. Er zijn dan hoogst waarschijnlijk nieuwe defecten toegevoegd aan het systeem. Hierdoor ligt de DD vanaf t_1 hoger dan ervoor. Dit wordt weergegeven in Figuur 24.



Figuur 24: Exponentieel model met adaptief en/of perfectief onderhoud

Aanname 1 zorgt ervoor dat de meetresultaten niet worden vertekend door discontinuïteiten zoals in Figuur 24. Indien aan deze aanname is voldaan spreken we over een stabiel systeem. Ik probeer op grond van de volgende drie indicatoren een beeld te vormen van deze stabiliteit:

- ΔLOC — Toevoegingen aan en uitbreidingen van het systeem leiden tot een substantiële toename van het aantal gewijzigde en toegevoegde coderegels en daarmee waarschijnlijk tot meer defecten.
- #APMI — Adaptive en perfective maintenance issues (APMI) hebben een grotere bijdrage aan het aantal gewijzigde en toegevoegde coderegels en daarmee aan het aantal nieuwe defecten dan corrective maintenance issues.

- #testuren Meer testuren leidt niet direct tot meer defecten. Ondanks dat meer testen ertoe kan leiden dat er meer – tot dan toe verborgen – defecten worden gevonden, stijgt het totale aantal defecten er niet door. De reden voor opname van het aantal testuren als indicator is gelegen in het feit dat substantieel meer testen aangeeft dat er blijkbaar belangrijke functionaliteit is toegevoegd of gewijzigd. Deze indicator is dus indirect.

Om de waarde van α in het model vast te stellen, moet de DD op zoveel mogelijk momenten worden vastgesteld. Hiertoe voeren we de reeds beschreven metingen van grootte en aantal defecten uit per opgeleverde versie.

Deze waarden moet vervolgens worden uitgezet tegen de tijd. Om de helling van de curve in het model vast te stellen, maken we gebruik van een transformatie van de functie d(t).

$$g(t) = \ln(d(t)) = \ln(d_0 e^{-\alpha t}) = \ln(d_0) + \ln(e^{-\alpha t}) = g_0 - \alpha t;\ \alpha > 0, t \geq 0$$

Formule 2: Transformatie van *d(t)*

Hierbij is g_0=ln(d_0)

Indien we de waarden ln(DD) uitzetten tegen de tijd, kunnen we door middel van lineaire regressie de waarde van α vaststellen.

5.5.1 De verwachte invloed van CBD op α

Laten we nog even terugkijken naar de in hoofdstuk 4 opgestelde hypothesen.

1. De initiële betrouwbaarheid van een herbruikbaar component is lager dan de initiële betrouwbaarheid van een vergelijkbaar specifiek component.

2. De betrouwbaarheid van herbruikbare, generieke componenten stijgt sneller dan die van specifieke componenten.

Hypothesen

Door middel van het geïntroduceerde model is het mogelijk een operationalisering te geven van deze hypothesen. Hypothese 1 stelt, dat de initiële betrouwbaarheid bij toepassing van CBD lager ligt dat bij conventionele ontwikkeling. Met andere woorden, de initiële DD ligt bij CBD hoger dan bij conventionele ontwikkeling. Deze initiële DD wordt weergegeven door de waarde d_0. We kunnen dus de volgende operationalisering geven van hypothese 1.

$$d_0^{CBD} > d_0^{conventioneel}$$

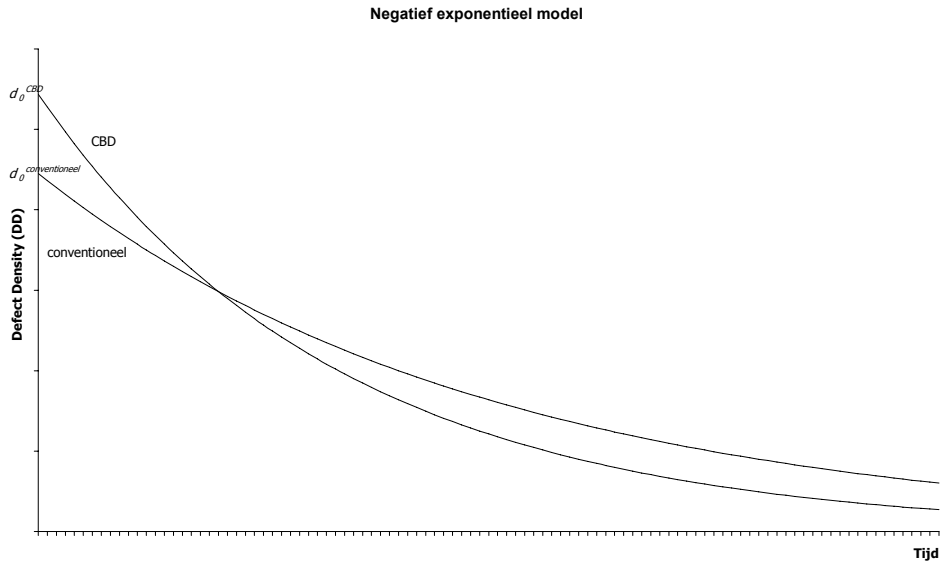
Operationalisering van hypothese 1

De mate van stijging van de betrouwbaarheid is binnen ons model vastgelegd in de waarde van parameter α. Hoe groter de waarde van α, hoe sneller de stijging van de betrouwbaarheid. Omdat een conventioneel systeem uitsluitend gebruik maakt van specifieke componenten en een CBD-systeem, met uitzondering van de glue code, zoveel mogelijk van generieke componenten, kunnen we de volgende operationalisering geven van hypothese 2.

$$\left|\alpha^{CBD}\right| > \left|\alpha^{conventioneel}\right|$$

Operationalisering van hypothese 2

Beide geoperationaliseerde hypothesen zijn grafisch weergegeven in Figuur 25.



Figuur 25: Geoperationaliseerde hypothesen

Deze figuur laat zien dat d_0^{CBD} hoger ligt dan $d_0^{conventioneel}$. Tevens zien we dat, voor elke waarde van t, d_t^{CBD} stijler is dan $d_t^{conventioneel}$, met andere woorden, $|\alpha^{CBD}|>|\alpha^{conventioneel}|$.

Voor ons onderzoek, het toetsen van hypothesen 1 en 2, moeten we dus voor beide systemen vaststellen wat de initiële DD is. Daarnaast moeten we voor beide systemen de trendlijn $d(t)$ bepalen. Dit doen we door de DD per versie uit te zetten tegen de tijd.

Omdat er gebruik gemaakt wordt van verschillende prioriteiten van defecten, zullen de defecten worden gewogen met de in 5.4.2 besproken verzamelingen wegingsfactoren. In dit geval wordt er niet gesproken over DD, maar over Weighted Defect Density (WDD), waaronder het volgende wordt verstaan.

Weighted Defect Density (WDD) = gewogen # defecten / LOC

Operationalisering 5: Weighted Defect Density (WDD)

5.6 Conclusie

In dit hoofdstuk hebben we het ontwerp van de toets geïntroduceerd. Dit ontwerp bestaat uit een aantal stappen, die toegepast kunnen worden op een softwaresysteem. Uitvoering van de toets leidt tot de vaststelling van de indicator voor de betrouwbaarheid van het systeem.

In de eerste stap wordt het systeem geanalyseerd en worden de in eigen beheer ontwikkelde componenten vastgesteld die worden meegenomen in het onderzoek. De tweede stap bestaat uit de identificatie van de versies waarvan zowel de broncode als de gegevens over issues beschikbaar zijn. In de derde stap worden de metingen met betrekking tot grootte, aantal defecten, aantal adaptieve en perfectieve onderhoudsissues en testuren verzameld. De vierde stap bestaat uit het vaststellen en selecteren van de geschikte perioden aan de hand van de stabiliteitscriteria en de gemeten waarden. In de vijfde stap wordt van iedere opvolgende versie binnen de geschikte perioden het gewogen aantal defecten gedeeld door het aantal regels code. De gewogen defect dichtheid (WDD) is de indicator voor de betrouwbaarheid. Stap zes bestaat uiteindelijk uit het vaststellen van de waarde van de initiële betrouwbaarheid d_0 en de indicator voor de betrouwbaarheidstoename α door de individuele betrouwbaarheidsindicatoren uit te zetten tegen de tijd en hier een trendlijn in vast te stellen.

6 Praktische toets: Uitvoering

6.1 Inleiding

In dit hoofdstuk wordt de uitvoering van de toets beschreven en worden de resultaten hiervan gepresenteerd. De uitvoering van de toets volgt hierbij de werkwijze die is beschreven in hoofdstuk 5.

In paragraaf 6.2 worden de twee systemen geanalyseerd. Hierbij wordt beargumenteerd welke componenten van de systemen wél en welke componenten niet worden meegenomen in het onderzoek. Ook wordt toegelicht hoe de verschillende versies van beide systemen zijn bepaald en welke versies meegenomen zijn in het onderzoek.

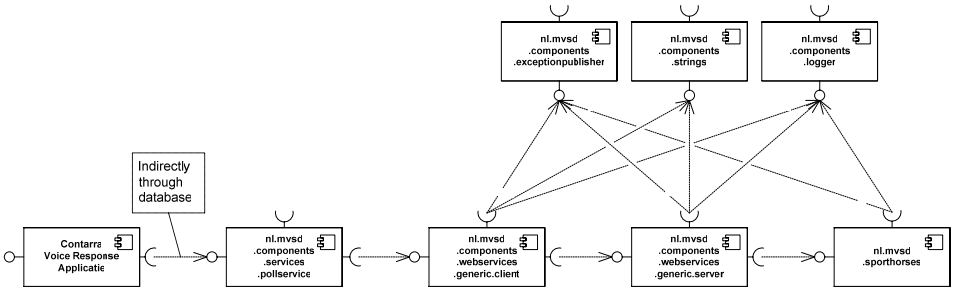
In paragraaf 6.3 worden de resultaten van de toets beschreven. Paragraaf 6.4 bevat de conclusies die op basis van de meetresultaten getrokken kunnen worden. Tot slot worden in paragraaf 6.5 de bevindingen van dit hoofdstuk kort samengevat.

6.2 Analyse systemen t.b.v. meten

Voordat de metingen van de grootte van de systemen plaats kunnen vinden worden de componenten waaruit de systemen bestaan geïdentificeerd. Omdat gegevens over defecten niet consequent beschikbaar zijn bij componenten van derden, worden uitsluitend componenten die in eigen beheer zijn ontwikkeld meegeteld. Deze componenten zullen bij beide systemen worden onderscheiden.

CBD-systeem

Het CDB-systeem bestaat uit een flink aantal componenten, waarvan een deel in eigen beheer is ontwikkeld en een deel dat door derden is ontwikkeld, zoals componenten van het Microsoft .NET Framework, Microsoft Application Blocks en componenten van een andere organisatie. Het totaaloverzicht van componenten is te vinden in Bijlage 3. Zoals in sectie 5.4.4 aangegeven, worden uitsluitend componenten die in eigen beheer ontwikkeld zijn, meegenomen in de toets. Deze componenten en hun relaties zijn weergegeven in Figuur 26. De pijlen geven in deze figuur de mogelijke aanroepen tussen de componenten aan.



Figuur 26: Componenten CBD-systeem

Component 'Contarra Voice Response Applicatie' is een component dat is ontwikkeld in de ontwikkelomgeving Contarra, waarbij gebruik is gemaakt van een procedurele taal. De rest van het systeem in ontwikkeld in de ontwikkelomgeving Microsoft Visual Studio 2003, in de objectgeoriënteerde taal C#. In hoofdstuk 5 is beargumenteerd dat beide systemen zoveel mogelijk vergelijkbaar moeten zijn om te voorkomen dat de toets teveel wordt beïnvloed door doorwerkende verschillen. Om deze reden is het genoemde Contarra component niet meegenomen in het onderzoek.

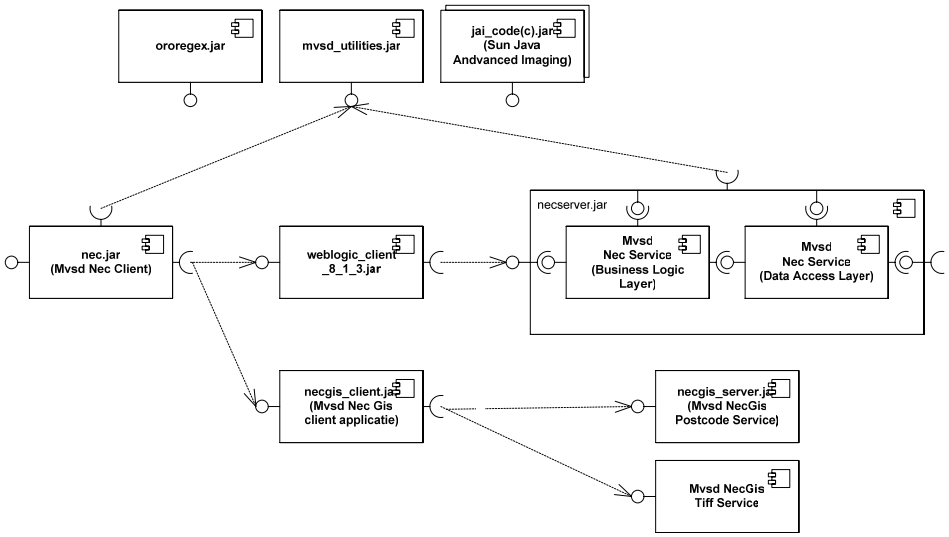
Bij de vaststelling van de versies van het CBD-systeem is gekeken naar de momenten waarop er van één of meer afzonderlijke componenten een nieuwe versie is opgeleverd. Bij het vervangen van een oude versie van een component door een nieuwe versie, wordt een versie met specifieke waarden voor eigenschappen als grootte en aantal defecten vervangen door een versie met mogelijk andere waarden voor deze eigenschappen. Hierdoor wijzigen mogelijk de waarden van de eigenschappen voor het gehele systeem. Er kan dus worden gesproken van een nieuwe versie van het systeem op het moment dat één of meer componenten van het systeem wordt vervangen door een nieuwe versie. In Tabel 2 worden de onderkende versies weergegeven. Versies CBD-systeem geeft aan hoe deze versies zijn onderkend.

Versie	Datum	Dag
1	31-08-2004	48
2	09-09-2004	57
3	15-09-2004	63
4	20-09-2004	68
5	04-10-2004	82
6	05-10-2004	83
7	19-10-2004	97

Tabel 2: Versies CBD-systeem

Conventioneel systeem

De componenten waaruit het conventionele systeem bestaat, zijn weergegeven in Figuur 27.



Figuur 27: Componenten conventioneel systeem

Van de getoonde componenten worden ororegex.jar, jai_code.jar, jai_codec.jar en weblogic_client_8_1_3.jar buiten beschouwing gelaten, omdat dit componenten van derden zijn. Hierdoor worden uitsluitend de componenten nec.jar, necgis_client.jar, necgis_server.jar, Mvsd NecGis Tiff Service, Mvsd Nec Service (Business Logic en Data Access Layer) en mvsd_utilities.jar meegenomen in het onderzoek.

Van de genoemde componenten zijn de uitgeleverde versies vastgesteld aan de hand van de gegevens in het issue tracking systeem. Bij de ingebruikname van dit systeem zijn er geen gegevens over eerder uitgeleverde versies opgenomen. Hierdoor vallen de versies vóór de eerste versie in het issue tracking systeem, te weten versie 2.2 van 19-11-2002 buiten het onderzoek.

Vervolgens is vastgesteld van welke versies de broncode beschikbaar was in het versiebeheersysteem CVS. De versies vóór versie 2.3 bleken niet aanwezig te zijn in dit systeem, waardoor ze buiten het onderzoek zijn gelaten. Versiebeheersystemen bieden doorgaans de mogelijkheid om een label toe te kennen aan een verzameling bestanden. Binnen CVS wordt dit tagging genoemd. Niet alle versies die onderkend worden in het issue tracking systeem bleken zo'n label te bevatten in het versiebeheersysteem. Dit probleem is ondervangen door in deze gevallen de versie op te halen die aanwezig was op de dag volgend op de opleverdatum van de versie. Er is bewust gekozen de bestanden van de volgende dag op te halen, omdat anders de wijzigingen van de dag van oplevering niet in de opgehaalde versie zouden zitten. De gevolgde procedure voor het ophalen van de bestanden is weergegeven in Bijlage 9.

Nadat alle versies van het issue tracking systeem vanaf versie 2.3 van 25-07-2003 waren opgehaald, bleek dat sommige opvolgende versies volkomen identiek waren. Twee

versies worden identiek genoemd indien alle bestanden van beide versies identiek aan elkaar zijn. De meest waarschijnlijke verklaring hiervoor is dat de ontwikkelaar de opgeleverde versie heeft samengesteld uit de bestanden op zijn ontwikkelmachine, zonder de wijzigingen in deze bestanden door te voeren in het versiebeheersysteem. Indien twee volkomen identieke, opvolgende, versies bestaan, is de tweede versie buiten het onderzoek gelaten.

Een overzicht van alle versies en hun opleverdata is weergegeven in Bijlage 6. Tabel 3 bevat een overzicht van alle versies die zijn meegenomen in het onderzoek.

Versie	Datum
2.3	25-07-2003
2.3.1	04-08-2003
2.3.2	26-09-2003
2.3.4	15-10-2003
2.3.4.100	17-02-2004
2.3.5.1	25-02-2004
2.3.5.2	01-03-2004
2.3.5.3	09-03-2004
2.3.5.4	05-04-2004
2.3.5.100	11-02-2004
2.3.5.102	04-03-2004
2.3.5.103	08-03-2004
2.3.5.104	09-03-2004
2.3.5.107	23-03-2004
2.3.5.108	25-03-2004
2.3.5.111	08-04-2004
2.4.1	05-05-2004
2.4.1.4	15-05-2004
2.4.1.5	19-05-2004
2.4.1.6	21-05-2004
2.4.1.7	25-05-2004
2.4.1.8	26-05-2004
2.4.1.10	27-05-2004
2.4.1.12	21-06-2004
2.4.1.13	27-06-2004
2.4.1.15	05-07-2004
2.4.2	29-07-2004

Versie	Datum
2.4.2.1	12-08-2004
2.4.2.2	24-08-2004

Tabel 3: Onderzochte versies conventioneel systeem

6.3 Meetresultaten

Van beide systemen zijn de meetgegevens verzameld van alle versies die meegenomen zijn in het onderzoek. Het verzamelen van de gegevens met betrekking tot LOC, ΔLOC, adaptive en perfective maintenance issues en defecten wordt voor de systemen beschreven.

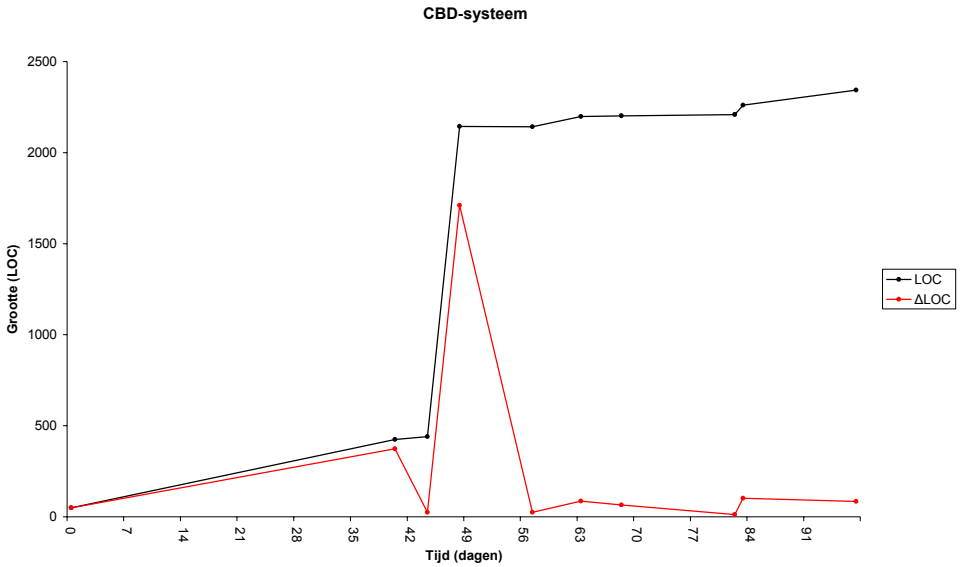
CBD-systeem

De meetresultaten met betrekking tot LOC, ΔLOC, adaptive en perfective maintenance issues en defecten zijn bij dit systeem verzameld per component. Deze resultaten zijn weergegeven in Bijlage 5. Deze gegevens zijn vervolgens geaggregeerd en weergegeven in Tabel 4.

Versie	Datum	Dag	LOC	ΔLOC
	14-07-2004	0	50	50
	23-08-2004	40	424	374
	27-08-2004	44	441	24
1	31-08-2004	48	2.144	1710
2	09-09-2004	57	2.142	25
3	15-09-2004	63	2.198	86
4	20-09-2004	68	2.202	66
5	04-10-2004	82	2.210	12
6	05-10-2004	83	2.261	102
7	19-10-2004	97	2.344	84

Tabel 4: Meetgegevens CBD-systeem

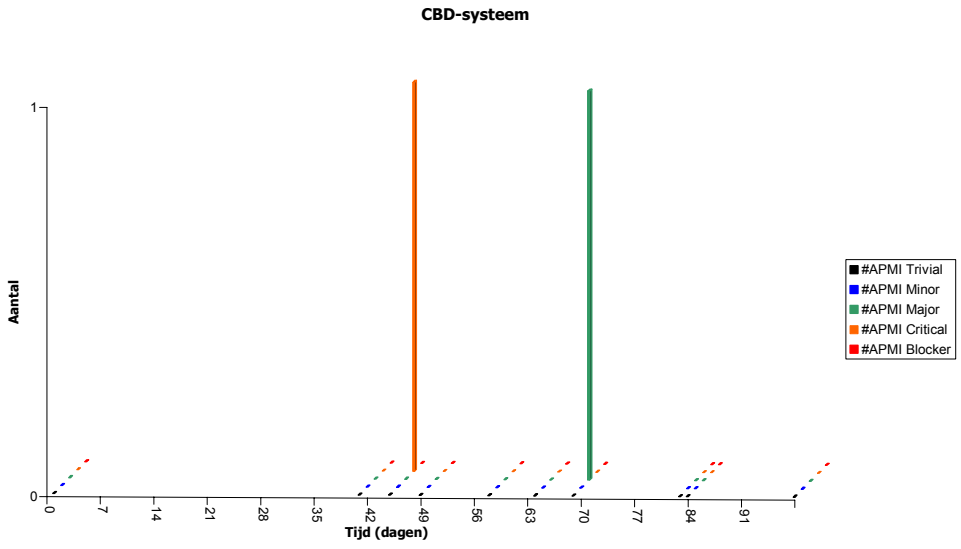
Ter indicatie van de stabiliteit van het systeem is als eerste het verloop van de grootte ervan in de tijd weergegeven in Figuur 28. Dag 0 is de datum van de eerste oplevering, te weten 14-07-2004.



Figuur 28: Ontwikkeling grootte CBD-systeem

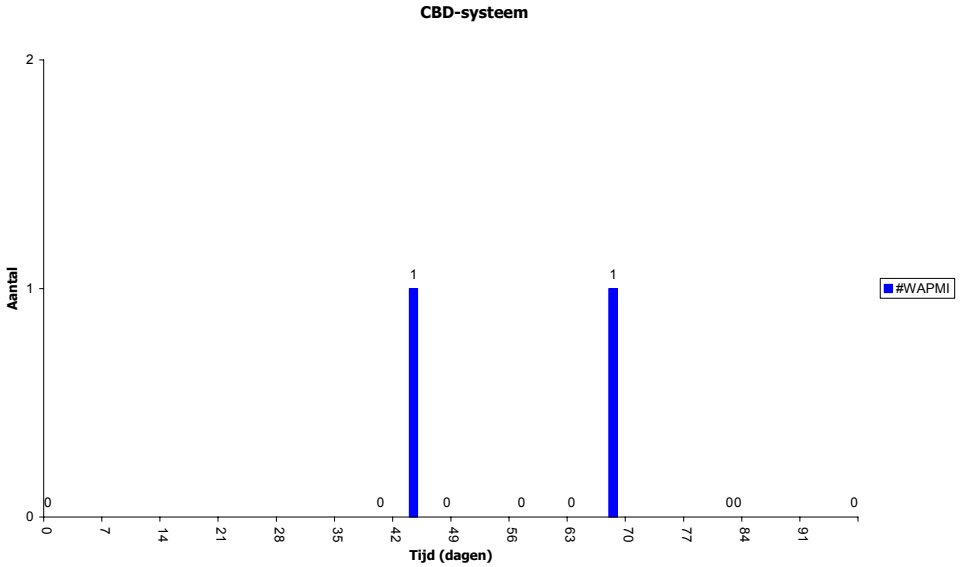
Er is duidelijk te zien dat het systeem vóór dag 48 (31-08-2004) sterk toeneemt in grootte, terwijl de grootte vanaf die dag slechts beperkt stijgt. Dit duidt erop dat het systeem vanaf dat moment niet in grote mate verandert.

De tweede indicator voor de stabiliteit van het systeem is het verloop van het aantal opgeleverde adaptive en perfective maintenance issues (APMI). Dit verloop is verzameld aan de hand van de gegevens in het issue tracking systeem en, gespecificeerd per prioriteit, weergegeven in Figuur 29. Hierin is duidelijk te zien dat er gedurende de levensduur van het systeem slechts twee 'issues' zijn opgeleverd, wat een sterke indicatie is voor de stabiliteit van het systeem.

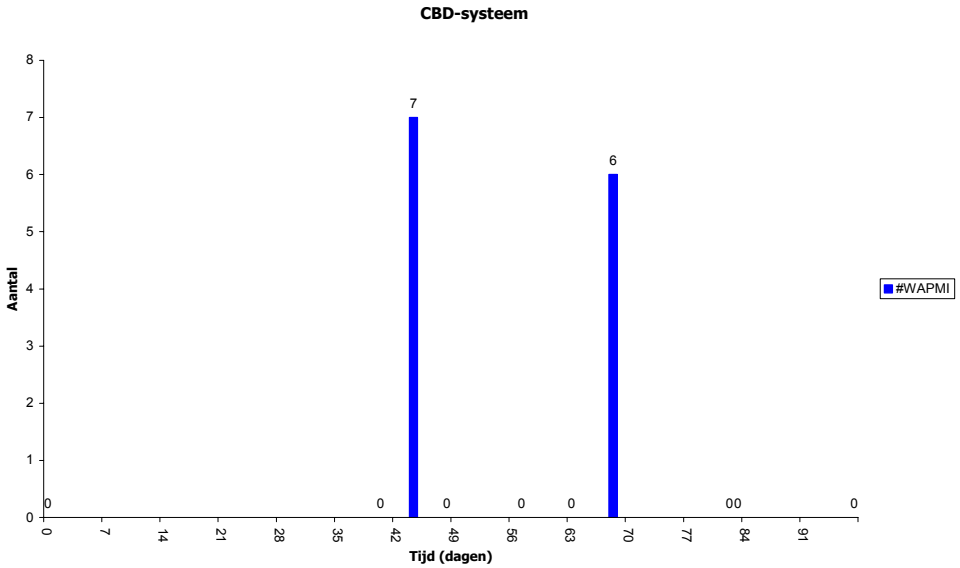


Figuur 29: Verloop APMI bij CBD-systeem

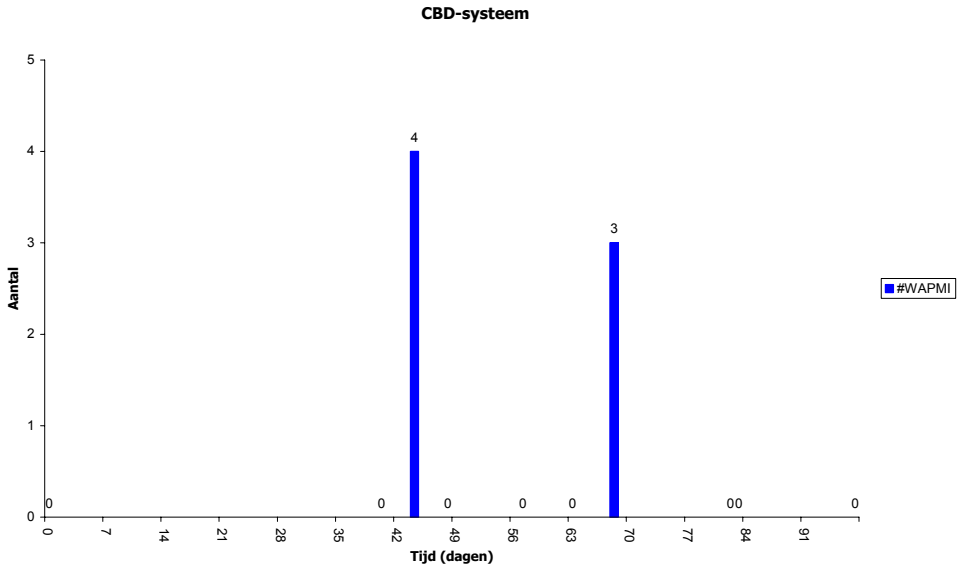
Het verloop van de gewogen adaptive en perfective maintenance issues (WAPMI) is weergegeven in Figuur 30 tot en met Figuur 32. Hierbij zijn respectievelijk de in sectie 5.4.2 vastgestelde wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 1, 1, 1, 1\}$, {4, 5, 6, 7, 8} en {1, 2, 3, 4, 5} gebruikt. Hieruit blijkt ook dat het systeem vrij stabiel is.



Figuur 30: Ontwikkeling WAPMI bij CBD-systeem met wegingsfactoren {1, 1, 1, 1, 1}

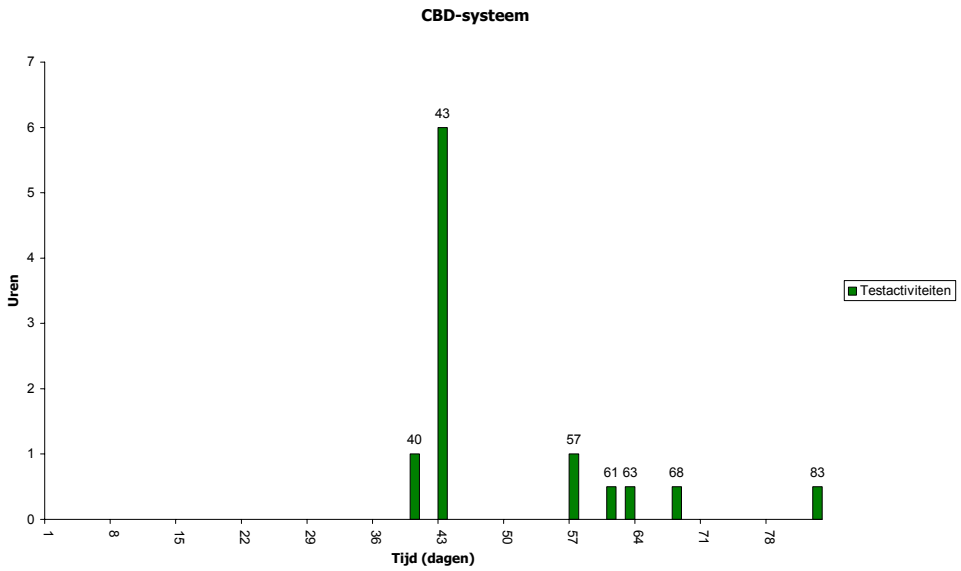


Figuur 31: Ontwikkeling WAPMI bij CBD-systeem met wegingsfactoren {4, 5, 6, 7, 8}

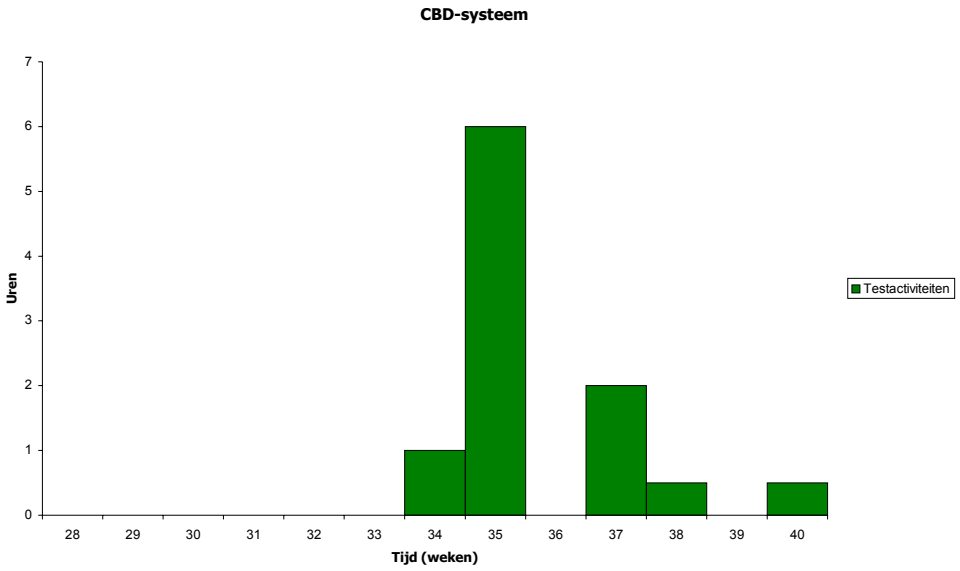


Figuur 32: Ontwikkeling WAPMI bij CBD-systeem met wegingsfactoren {1, 2, 3, 4, 5}

De laatste indicator voor stabiliteit is het aantal uren dat is besteed aan testactiviteiten. Dit aantal is in Figuur 33 en Figuur 34 respectievelijk per dag en per week weergegeven. Uit deze figuren blijkt dat er, met uitzondering van dag 43 (26-08-2004), niet veel is getest. Het aantal uren dat is besteed aan testactiviteiten is na deze dag zelfs niet hoger dan één. Tengevolge hiervan verwachten we dat het aantal defecten na dag 43 niet ineens sterk stijgt doordat er, tengevolge van veel testen, veel (tot dat moment verborgen) defecten worden ontdekt.



Figuur 33: Testuren CBD-systeem per dag



Figuur 34: Testuren CBD-systeem per week

Uit de meetgegevens van de ontwikkeling van de grootte, de opgeleverde adaptive en perfective maintenance issues en de testactiviteiten kan worden vastgesteld dat er vanaf versie 1 van dag 0 (31-08-2004) tot en met versie 7 van dag 97 (19-10-2004) sprake is van een stabiel systeem. Op grond hiervan kan de waarde van de indicator voor de snelheid van de betrouwbaarheidsgroei, α_{CBD}, gedurende deze periode worden vastgesteld.

De volgende stap is het per versie vaststellen van de WDD. Dit is weer gedaan met de respectievelijke wegingsfactoren {1, 1, 1, 1, 1}, {4, 5, 6, 7, 8} en {1, 2, 3, 4, 5} voor de prioriteiten {Trivial, Minor, Major, Critical en Blocker}, welke zijn weergegeven in Tabel 5.

Versie	Datum	Dag	WDD {1, 1, 1, 1, 1}	WDD {4, 5, 6, 7, 8}	WDD {1, 2, 3, 4, 5}
1	31-08-2004	48	2,7985	17,2575	8,8619
2	09-09-2004	57	2,8011	17,2736	8,8702
3	15-09-2004	63	1,3649	8,6442	4,5496
4	20-09-2004	68	1,3624	8,6285	4,5413
5	04-10-2004	82	1,3575	9,0498	4,9774
6	05-10-2004	83	0,4423	3,0960	1,7691
7	19-10-2004	97	0,4266	2,9863	1,7065

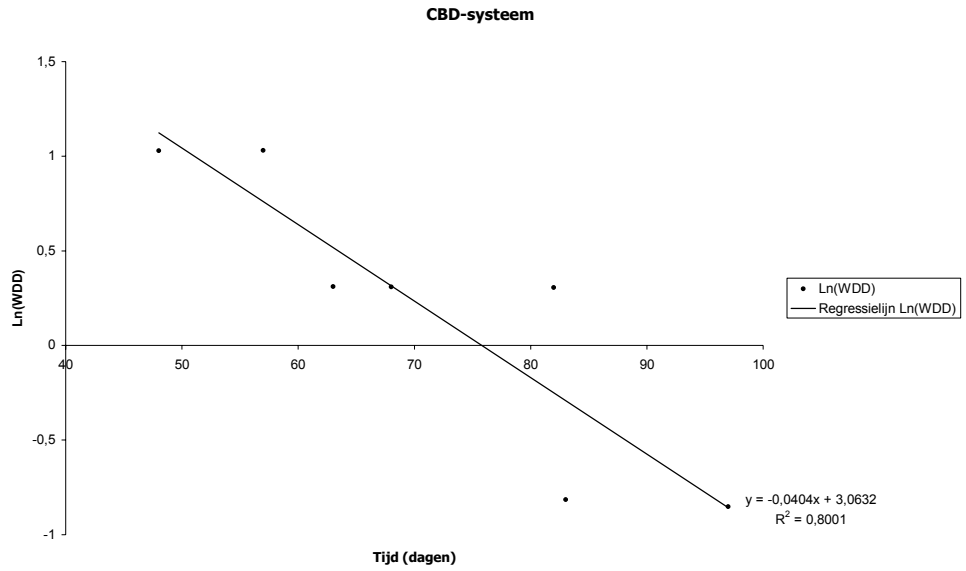
Tabel 5: WDD van CBD-systeem

Teneinde de snelheid van betrouwbaarheidsgroei voor het CBD-systeem, α_{CBD}, te bepalen is van deze waarden het natuurlijk logaritme vastgesteld, weergegeven in Tabel 6.

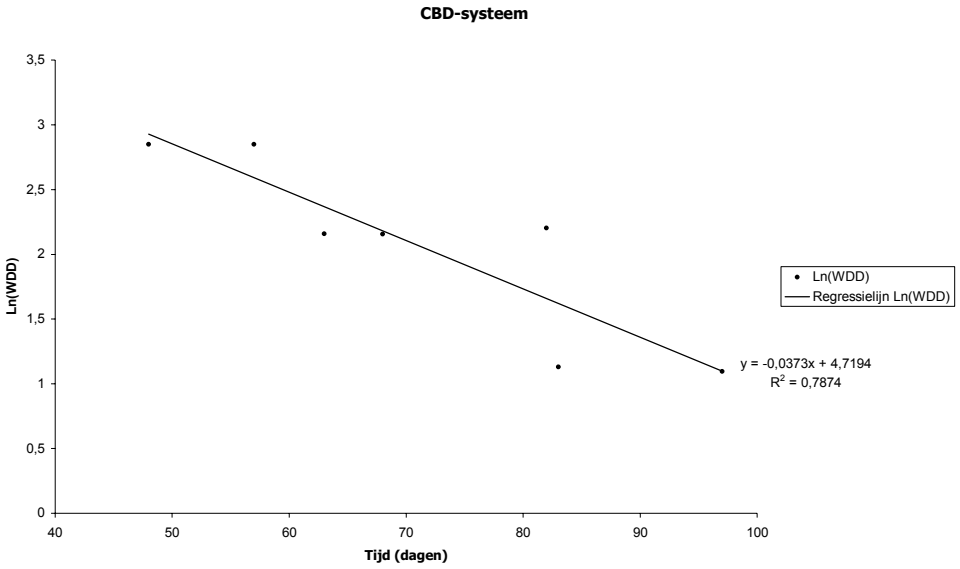
Versie	Datum	Dag	Ln(WDD) {1, 1, 1, 1, 1}	Ln(WDD) {4, 5, 6, 7, 8}	Ln(WDD) {1, 2, 3, 4, 5}
1	31-08-2004	48	1,0291	2,8482	2,1818
2	09-09-2004	57	1,0300	2,8492	2,1827
3	15-09-2004	63	0,3111	2,1569	1,5150
4	20-09-2004	68	0,3092	2,1551	1,5132
5	04-10-2004	82	0,3056	2,2027	1,6049
6	05-10-2004	83	-0,8158	1,1301	0,5705
7	19-10-2004	97	-0,8519	1,0941	0,5344

Tabel 6: Ln(WDD) van CBD-systeem

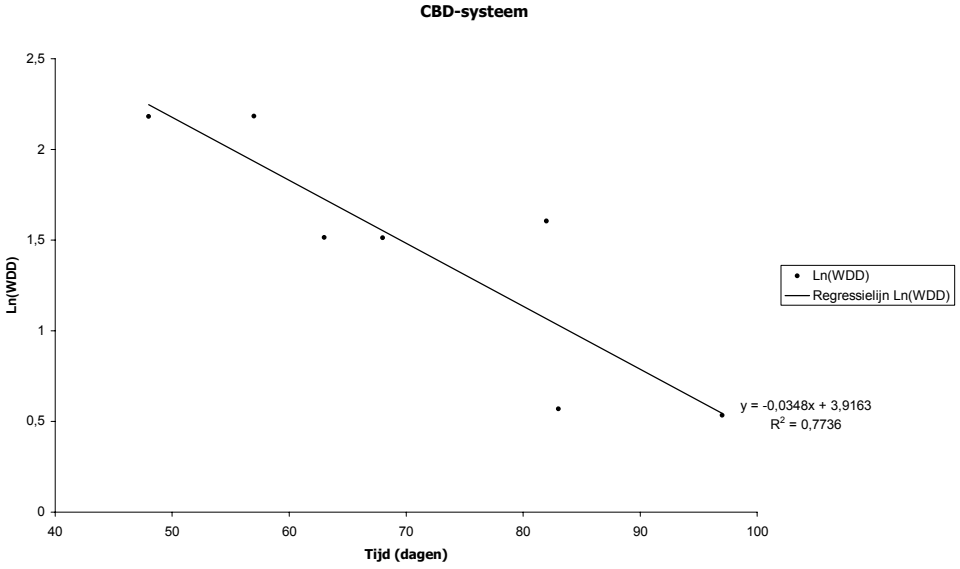
Deze waarden zijn uitgezet tegen de tijd in Figuur 35 tot en met Figuur 37. Vervolgens is met behulp van lineaire regressie, voor elke combinatie van wegingsfactoren, de hellingshoek α_{CBD} en het betrouwbaarheidsindicator R^2_{CBD} vastgesteld. De waarde α_{CBD} geeft aan hoe snel de betrouwbaarheid van het CBD-systeem toeneemt in de tijd. De indicator R^2_{CBD} geeft aan hoe betrouwbaar deze waarde is. De gemeten waarden zijn weergegeven in Tabel 7.



Figuur 35: Ontwikkeling van Ln(WDD) van CBD-systeem bij wegingsfactoren {1, 1, 1, 1, 1}



Figuur 36: Ontwikkeling van Ln(WDD) van CBD-systeem bij wegingsfactoren {4, 5, 6, 7, 8}



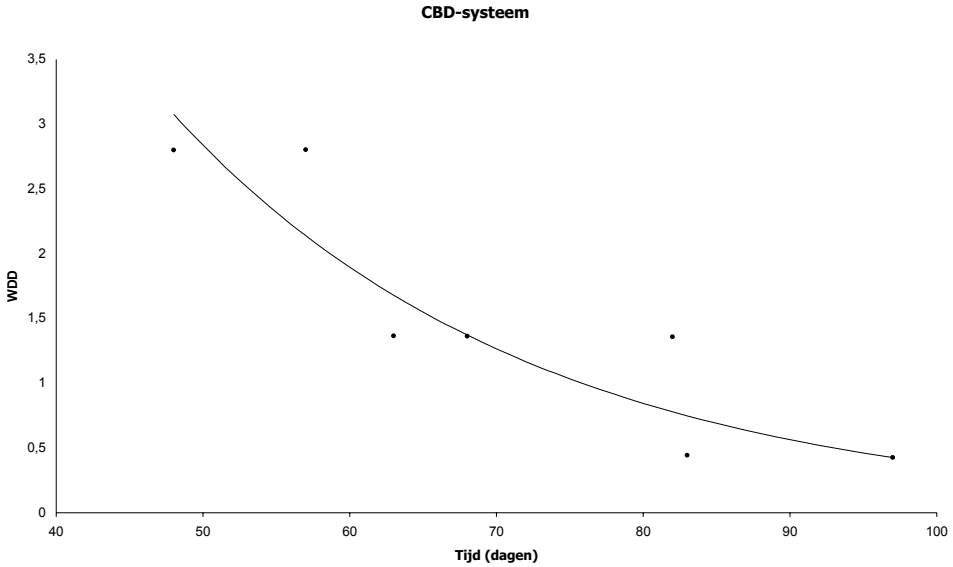
Figuur 37: Ontwikkeling van Ln(WDD) van CBD-systeem bij wegingsfactoren {1, 2, 3, 4, 5}

Wegingsfactoren $\{W_{Trivial}, W_{Minor}, W_{Major}, W_{Critical}, W_{Blocker}\}$	a_{CBD}	R^2_{CBD}
{1, 1, 1, 1, 1}	0,0404	0,8001
{4, 5, 6, 7, 8}	0,0373	0,7874
{1, 2, 3, 4, 5}	0,0348	0,7736

Tabel 7: Snelheid betrouwbaarheidsgroei van CBD-systeem

Uit Tabel 7 blijkt dat de snelheid van de betrouwbaarheidsgroei, α_{CBD}, afneemt naarmate de ernstiger effecten zwaarder meewegen dan de minder ernstige defecten. Hierbij is een lichte daling van de betrouwbaarheid te zien, welke respectievelijk 89%, 89% en 88% is[10].

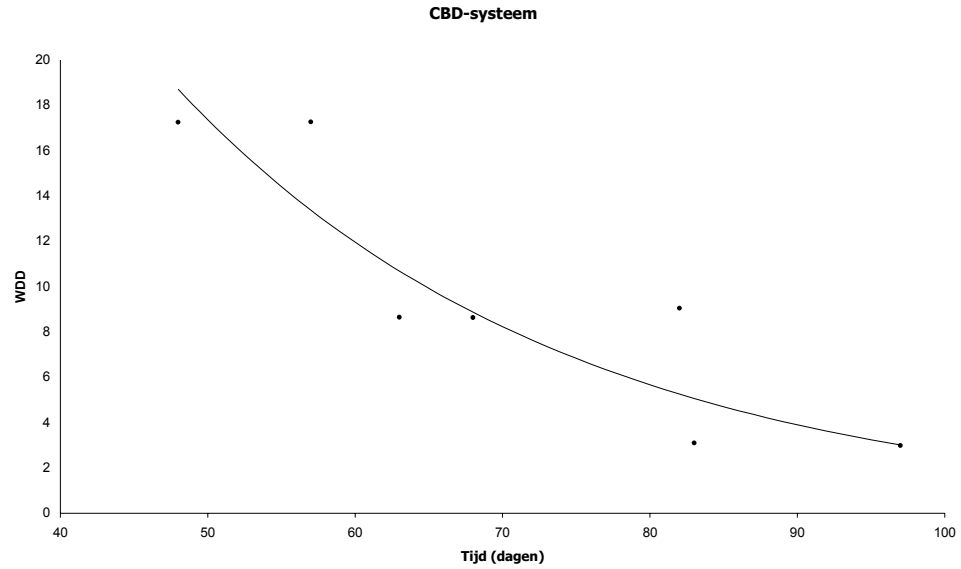
Om visueel te controleren of de gemeten waarden voor α_{CBD} juist zijn, zijn de curven van de WDD behorende bij deze indicatoren voor elke combinatie van wegingsfactoren samen met de gemeten waarden uitgezet in Figuur 38 tot en met Figuur 40. Uit deze figuren blijkt dat de vastgestelde waarden voor α_{CBD} redelijk overeenkomen met de gemeten waarden van de WDD.



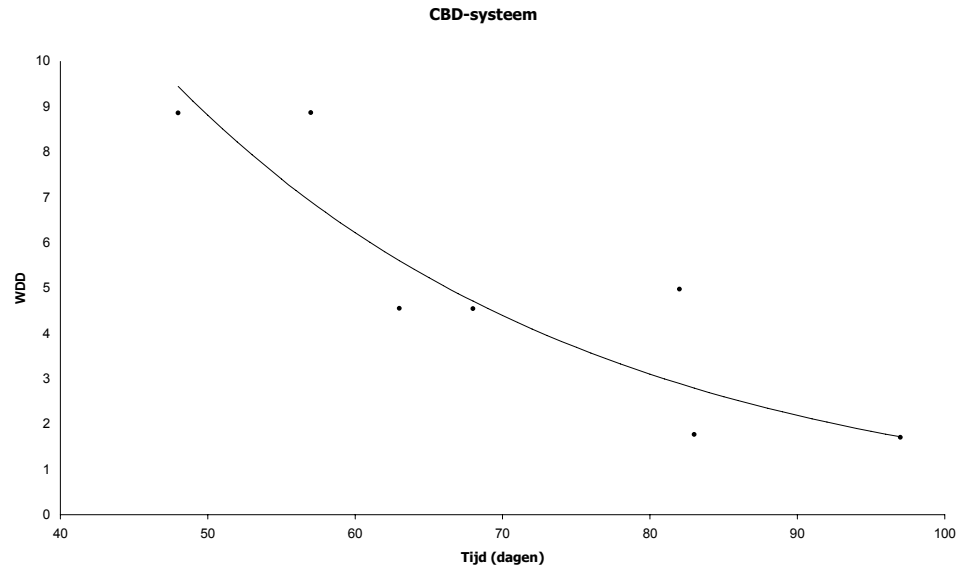
Figuur 38: Ontwikkeling van WDD van CBD-systeem bij wegingsfactoren {1, 1, 1, 1, 1}

[10] De betrouwbaarheid van de gemeten waarde van α wordt uitgedrukt in de variabele *R*. Bij een waarde van 0,8001 voor R^2 geldt dat de betrouwbaarheid gelijk is aan:

$$R = \sqrt{R^2} = \sqrt{0{,}8001} \approx 0{,}8945 \approx 89\%$$



Figuur 39: Ontwikkeling van WDD van CBD-systeem bij wegingsfactoren {4, 5, 6, 7, 8}



Figuur 40: Ontwikkeling van WDD van CBD-systeem bij wegingsfactoren {1, 2, 3, 4, 5}

Conventioneel systeem

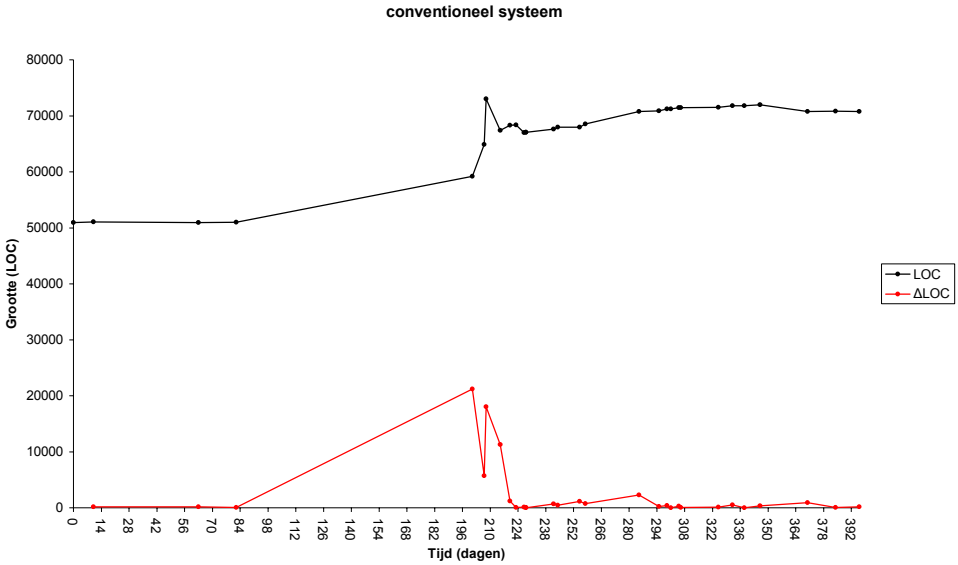
In tegenstelling tot bij het CBD-systeem, kan de grootte van de afzonderlijke versies van het conventionele systeem direct worden gemeten. De versies worden in zijn geheel bijgehouden in het versiebeheersysteem. Hierdoor is het niet nodig om de metingen uit te voeren voor alle versies van de afzonderlijke componenten, maar kunnen van iedere systeemversie direct de gegevens met betrekking tot de grootte, het aantal adaptive en perfective maintenance issues en het aantal testuren worden gemeten. Deze drie

gegevens geven een indicatie voor de stabiliteit van het systeem. De meetresultaten met betrekking tot de eerste stabiliteitsindicator, de codegrootte, zijn weergegeven in Tabel 8.

Versie	Label	Datum	Dag	LOC	ΔLOC
2.3		25-07-2003	0	50.970	n.v.t.
2.3.1		04-08-2003	10	51.036	167
2.3.2		26-09-2003	63	50.927	172
2.3.4		15-10-2003	82	50.976	55
2.3.5.100		11-02-2004	201	59.223	3.532
2.3.4.100		17-02-2004	207	64.845	26.818
2.3.5	nec-2-3-5	18-02-2004	208	72.981	18.048
2.3.5.1		25-02-2004	215	67.383	11.268
2.3.5.2		01-03-2004	220	68.335	1.197
2.3.5.102		04-03-2004	223	68.367	11.073
2.3.5.103		08-03-2004	227	66.994	117
2.3.5.104		09-03-2004	228	67.064	78
2.3.5.3		09-03-2004	228	67.064	233
2.3.5.107		23-03-2004	242	67.640	716
2.3.5.108		25-03-2004	244	67.976	441
2.3.5.4		05-04-2004	255	67.976	1.156
2.3.5.111		08-04-2004	258	68.527	744
2.4.1		05-05-2004	285	70.760	2.310
2.4.1.4		15-05-2004	295	70.890	232
2.4.1.5		19-05-2004	299	71.225	396
2.4.1.6	nec-2-4-1-6	21-05-2004	301	71.228	16
2.4.1.7	nec-2-4-1-7	25-05-2004	305	71.461	272
2.4.1.8		26-05-2004	306	71.437	64
2.4.1.11	nec-2-4-1-11	14-06-2004	325	71.532	117
2.4.1.12		21-06-2004	332	71.780	518
2.4.1.13		27-06-2004	338	71.780	2
2.4.1.15	nec-2-4-1-15	05-07-2004	346	71.964	317
2.4.2	nec-2-4-2	29-07-2004	370	70.781	940
2.4.2.1	nec-2-4-2-1	12-08-2004	384	70.804	39
2.4.2.2	nec-2-4-2-2	24-08-2004	396	70.784	181

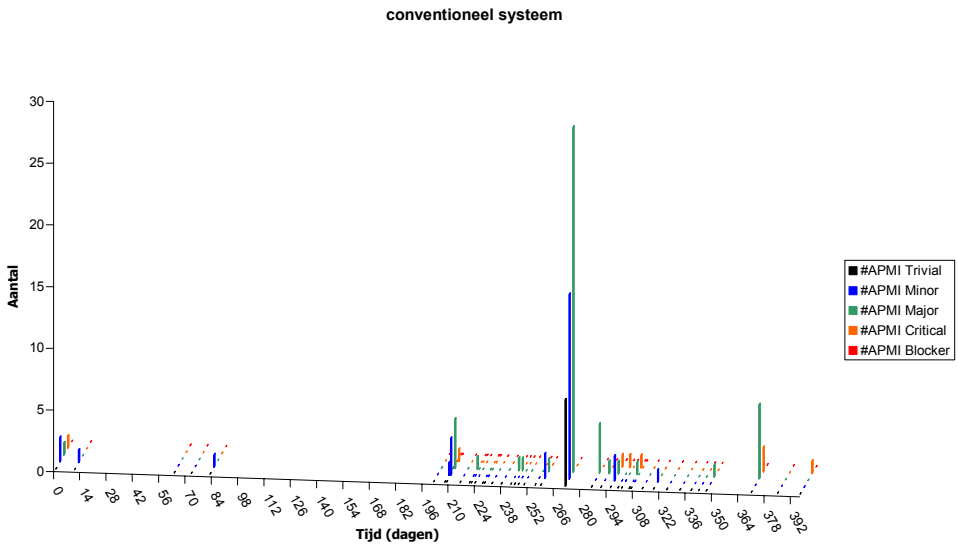
Tabel 8: Grootte conventioneel systeem

Een grafische weergave van de ontwikkeling van deze grootte is weergegeven in Figuur 41. Wat direct opvalt is de sterke toename in grootte in de versies die zijn opgeleverd tussen dag 201 (11-02-2004) en dag 215 (25-02-2004) en de kleinere toename op dag 285 (05-05-2004), wat wijst op instabiliteit van het systeem. Tengevolge van deze hoge waarden van ΔLOC verwachten we een toename van de defecten in deze versies.



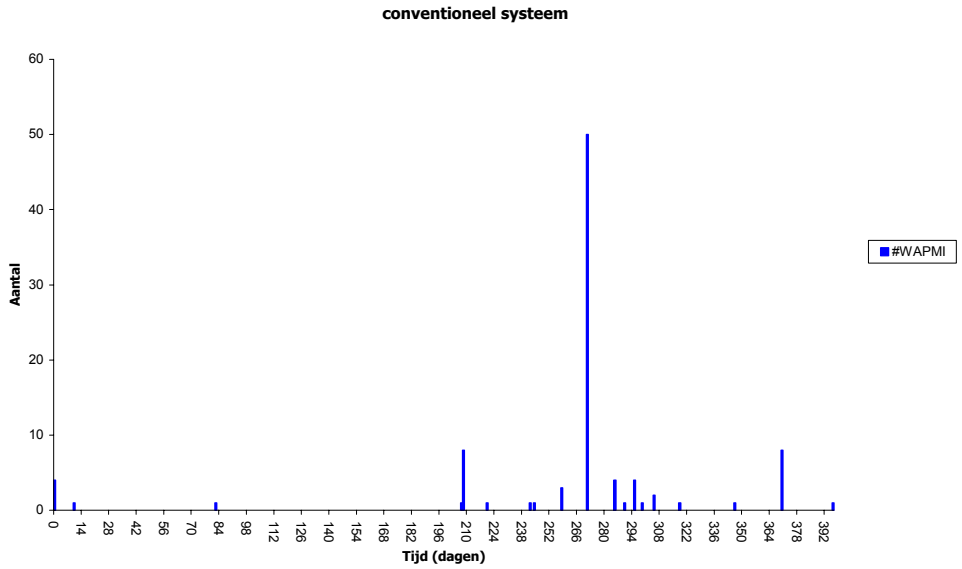
Figuur 41: Grootte conventioneel systeem

De tweede indicator voor systeemstabiliteit is het aantal opgeleverde adaptieve en perfectieve maintenance issues (APMI). In Figuur 42 is de verdeling van de APMI over de tijd weergegeven.

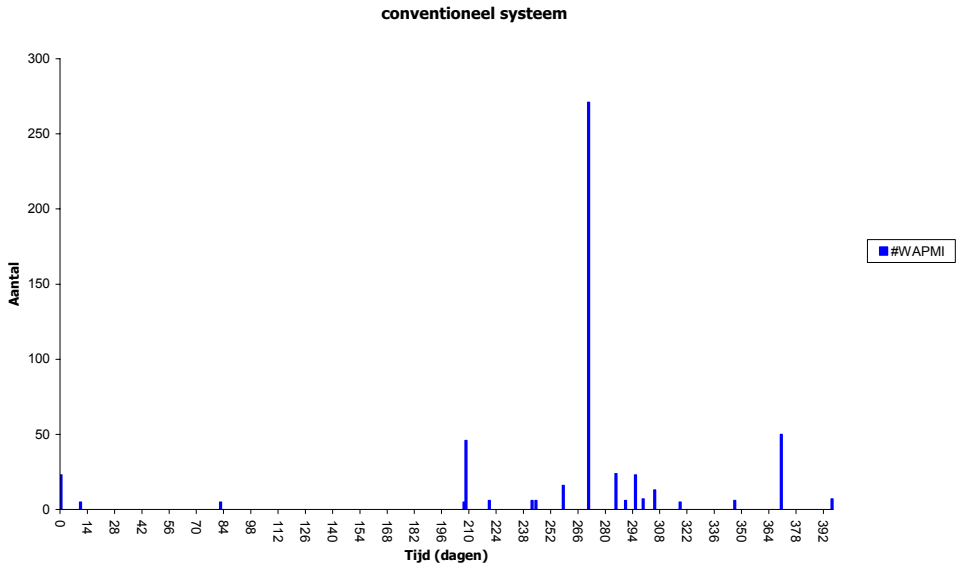


Figuur 42: APMI van conventioneel systeem

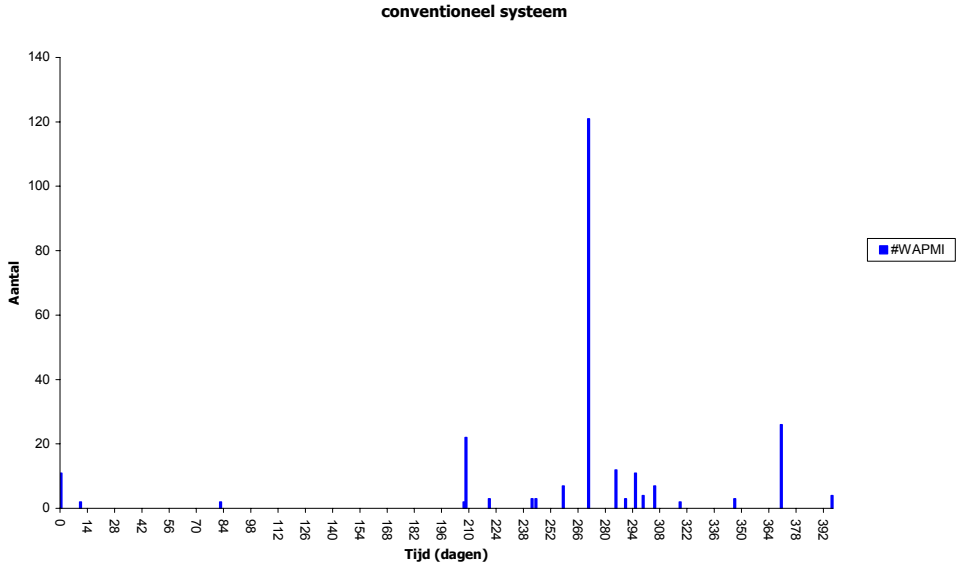
Het verloop van het aantal gewogen adaptive en perfective maintenance issues (WAPMI) is weergegeven in Figuur 43 tot en met Figuur 45, waarbij als wegingsfactoren voor de prioriteiten, {$w_{Trivial}$, w_{Minor}, w_{Major}, $w_{Critical}$, $w_{Blocker}$}, respectievelijk {1, 1, 1, 1, 1}, {4, 5, 6, 7, 8} en {1, 2, 3, 4, 5} is gebruikt. Er is duidelijk te zien, dat er bij een flink aantal versies behoorlijk veel adaptive en perfective maintenance issues zijn opgeleverd. Dit indiceert een betrekkelijk hoge mate van instabiliteit van het systeem. Een hoog aantal opgeleverde issues zal naar verwachting leiden tot een hoger aantal defecten.



Figuur 43: Ontwikkeling WAPMI bij conventioneel systeem met wegingsfactoren {1, 1, 1, 1, 1}

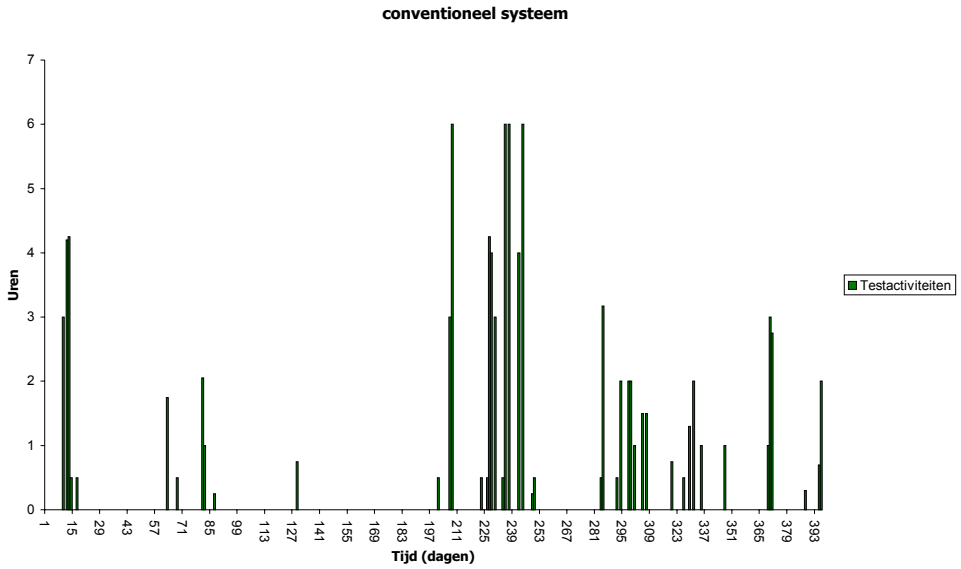


Figuur 44: Ontwikkeling WAPMI bij conventioneel systeem met wegingsfactoren {4, 5, 6, 7, 8}

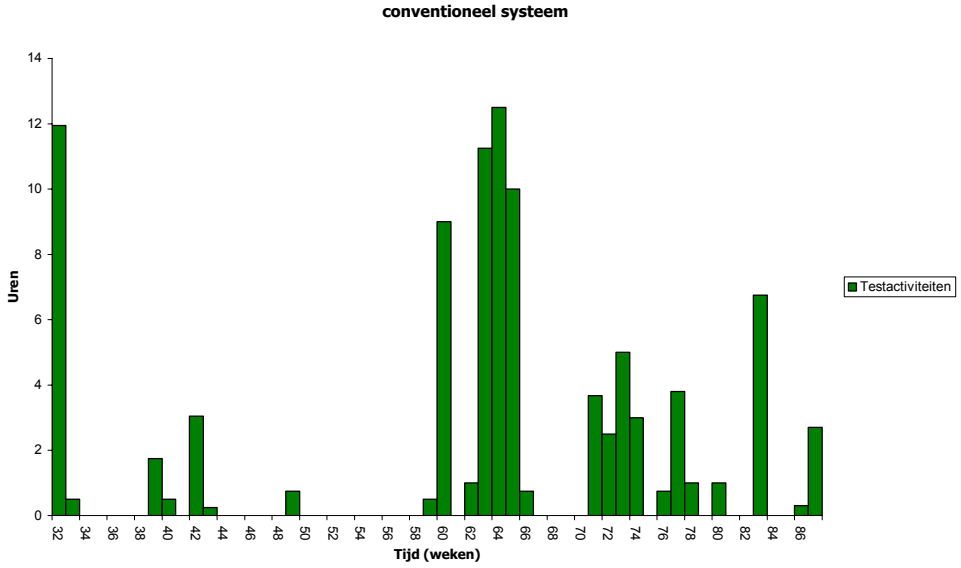


Figuur 45: Ontwikkeling WAPMI bij conventioneel systeem met wegingsfactoren {1, 2, 3, 4, 5}

De derde en laatste indicator voor systeemstabiliteit is het aantal uren dat gedurende de onderzoeksperiode is besteed aan testactiviteiten. Dit aantal is in Figuur 46 per dag en in Figuur 47 per week weergegeven. Er is duidelijk te zien dat er hier geen sprake is van testactiviteiten die gelijkmatig over de tijd – of over de versies – zijn verdeeld. Op grond van de gegevens verwachten we dat het aantal defecten door het bovengemiddeld testen in weken 32, 42, 60, 63 t/m 65, 71 t/m 74, 77, 83 en 87 zal stijgen.



Figuur 46: Testuren conventioneel systeem per dag



Figuur 47: Testuren conventioneel systeem per week

De meetresultaten van de ontwikkeling van de grootte, het aantal opgeleverde adaptive en perfective maintenance issues en de testactiviteiten geven een indicatie over de stabiliteit van het systeem in de onderzoeksperiode. Grote veranderingen in grootte of het aantal opgeleverde issues leiden naar verwachting tot meer gevonden defecten. Daarnaast is een bovengemiddelde testinspanning een indicatie dat er veel functionaliteit aan het systeem is toegevoegd of aangepast, wat naar verwachting leidt tot meer defecten. Op grond van deze instabiliteit concluderen we dat een groot deel van het systeem niet stabiel is en daarom niet geschikt om de parameter $\alpha_{conventioneel}$, de indicator voor de snelheid waarmee de betrouwbaarheid van het systeem toeneemt, vast te

stellen. De perioden die als ongeschikt worden geacht en de reden hiervoor staan beschreven in Bijlage 8.

De enige periode die op grond van de stabiliteitscriteria geschikt is, is de periode van dag 242 (23-03-2004) tot en met dag 258 (08-04-2004). Van de versies die gedurende deze periode zijn opgeleverd, is de WDD vastgesteld, met de respectievelijke wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 1, 1, 1, 1\}$, {4, 5, 6, 7, 8} en {1, 2, 3, 4, 5}. De waarden zijn weergegeven in Tabel 9.

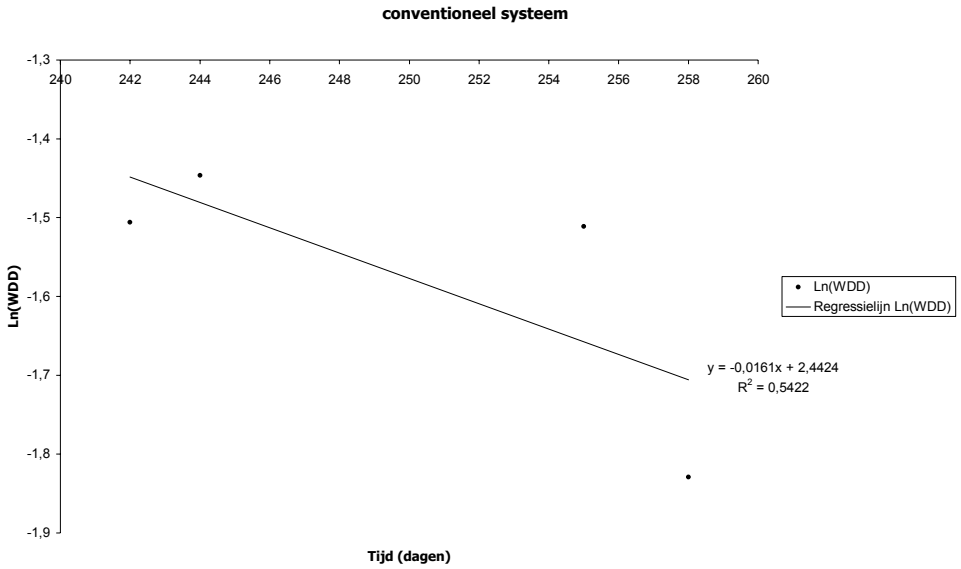
Versie	Datum	Dag	WDD {1, 1, 1, 1, 1}	WDD {4, 5, 6, 7, 8}	WDD {1, 2, 3, 4, 5}
2.3.5.107	23-03-2004	242	0,2218	1,3601	0,6949
2.3.5.108	25-03-2004	244	0,2354	1,2652	0,5590
2.3.5.4	05-04-2004	255	0,2207	1,1769	0,5149
2.3.5.111	08-04-2004	258	0,1605	0,9048	0,4232

Tabel 9: WDD van conventioneel systeem

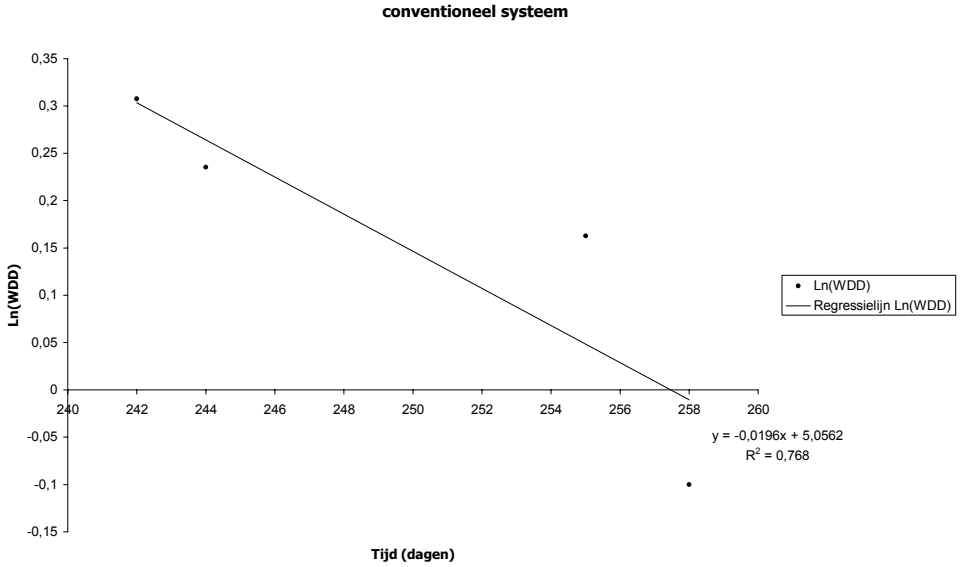
Voor het vaststellen van de snelheid van de betrouwbaarheidsgroei is van deze waarden het natuurlijk logaritme vastgesteld en weergegeven in Tabel 10. Deze waarden zijn in Figuur 48 tot en met Figuur 50 uitgezet tegen de tijd met respectievelijke wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 1, 1, 1, 1\}$, {4, 5, 6, 7, 8} en {1, 2, 3, 4, 5}. In deze figuren is tevens de trendlijn weergegeven. De vastgestelde waarden voor de snelheid van de betrouwbaarheidsgroei zijn weergegeven in Tabel 11.

Versie	Datum	Dag	Ln(WDD) {1, 1, 1, 1, 1}	Ln(WDD) {4, 5, 6, 7, 8}	Ln(WDD) {1, 2, 3, 4, 5}
2.3.5.107	23-03-2004	242	-1,5061	0,3076	-0,3641
2.3.5.108	25-03-2004	244	-1,4466	0,2352	-0,5816
2.3.5.4	05-04-2004	255	-1,5111	0,1629	-0,6638
2.3.5.111	08-04-2004	258	-1,8293	-0,1001	-0,8599

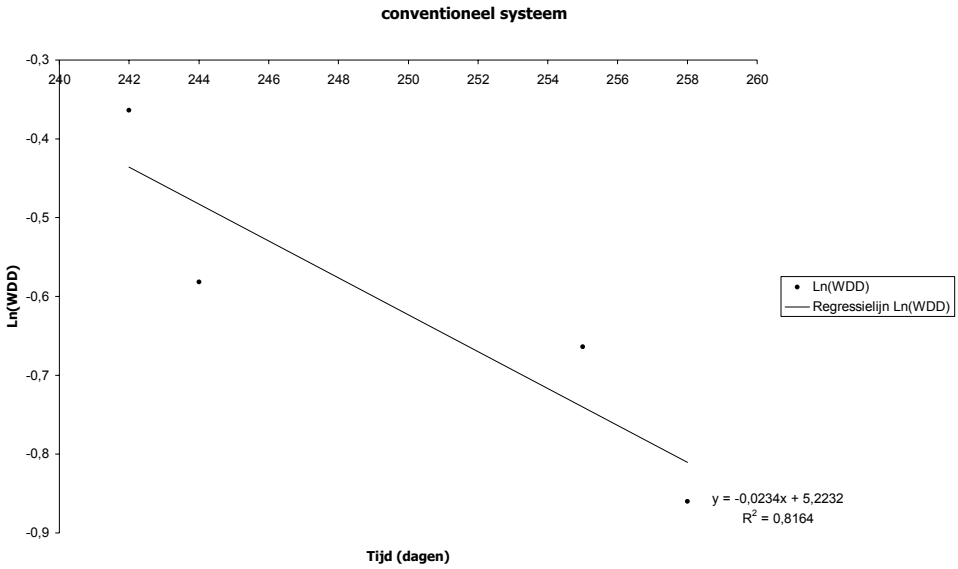
Tabel 10: Ln(WDD) van conventioneel systeem



Figuur 48: Ontwikkeling van Ln(WDD) van conventioneel systeem bij wegingsfactoren {1, 1, 1, 1, 1}



Figuur 49: Ontwikkeling van Ln(WDD) van conventioneel systeem bij wegingsfactoren {4, 5, 6, 7, 8}



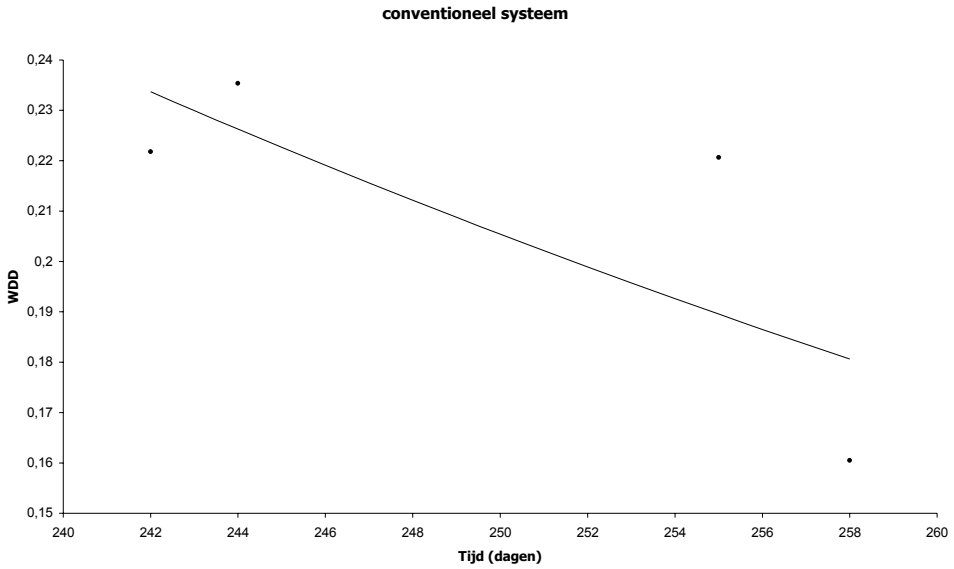
Figuur 50: Ontwikkeling van Ln(WDD) van conventioneel systeem bij wegingsfactoren {1, 2, 3, 4, 5}

Wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\}$	$a_{conv.}$	$R^2_{conv.}$
{1, 1, 1, 1, 1}	0,0161	0,5422
{4, 5, 6, 7, 8}	0,0196	0,7680
{1, 2, 3, 4, 5}	0,0234	0,8164

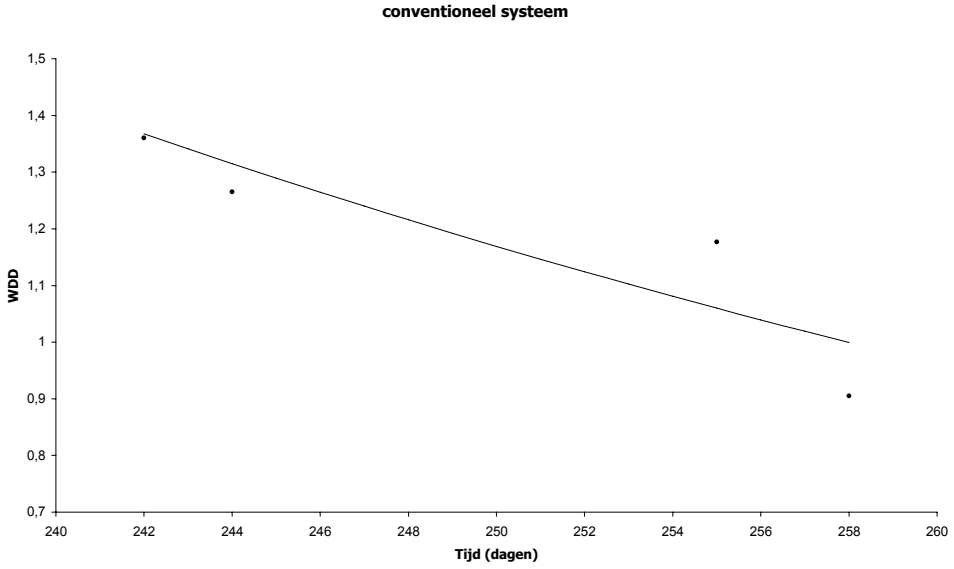
Tabel 11: Snelheid betrouwbaarheidsgroei van conventioneel systeem

Uit Tabel 11 blijkt dat de snelheid van de betrouwbaarheidsgroei toeneemt naarmate de ernstiger effecten zwaarder meewegen dan de minder ernstige defecten. De betrouwbaarheid van de gemeten indicator vertoont ook een stijgend verloop. Bij $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 1, 1, 1, 1\}$ is de betrouwbaarheid met 74% aan de lage kant.

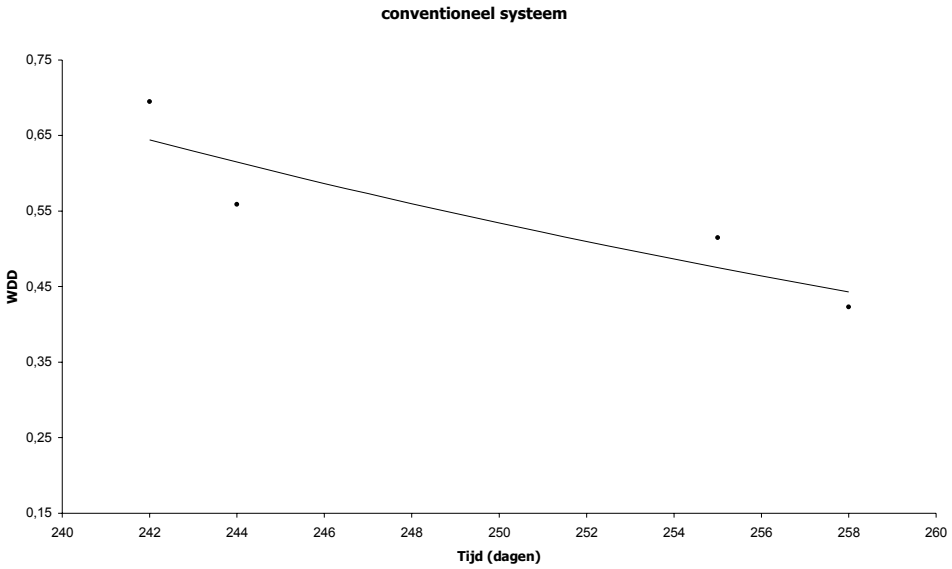
Ter controle zijn de gevonden waarden voor de betrouwbaarheidsindicator $a_{conventioneel}$ toegepast op de curven voor de meetresultaten van WDD in Figuur 51 tot en met Figuur 53. De gevonden waarden lijken redelijk goed te kloppen. Wat opvalt is dat de kromming van de curven vrij beperkt is. Een verklaring hiervoor kan zijn, dat dit systeem al langer in gebruik is en de WDD al verder is gedaald, waardoor de afname van de foutdichtheid – vanwege de eigenschappen van de exponentiële verdeling – steeds kleiner wordt en de curve daardoor steeds rechter.



Figuur 51: Ontwikkeling van WDD van conventioneel systeem bij wegingsfactoren {1, 1, 1, 1, 1}



Figuur 52: Ontwikkeling van WDD van conventioneel systeem bij wegingsfactoren {4, 5, 6, 7, 8}



Figuur 53: Ontwikkeling van WDD van conventioneel systeem bij wegingsfactoren {1, 2, 3, 4, 5}

6.4 Conclusie

Op basis van de gemeten waarden in de voorgaande paragrafen kunnen direct een aantal zaken worden vastgesteld. De grootte van de software vertoont bij beide projecten een stijgend verloop. Op basis van de gemeten waarden voor de grootte, aantal adaptive en perfective maintenance issues en het aantal uren besteed aan testactiviteiten kan worden gesteld dat het CBD-systeem gedurende de onderzoeksperiode 'stabiel' is. Binnen deze periode is de snelheid van betrouwbaarheidsgroei, α^{CBD}, vastgesteld voor verschillende wegingsfactoren. Bij het conventionele systeem is helaas geen sprake van stabiliteit over de gehele onderzoeksperiode. Er zijn diverse versies waarvan de gewijzigde regels code, het aantal opgeleverde adaptive en perfective maintenance issues of het aantal bestede uren aan testactiviteiten zo hoog is, dat er sprak is van een discontinuïteit van de curve die de ontwikkeling van de betrouwbaarheid aangeeft. Hierdoor kunnen slechts de delen van de onderzoeksperiode worden bekeken waarover deze curve continu is. Een bijkomend probleem is dat hierdoor een aantal continue segmenten van de curve ontstaan waarop slechts één of twee versies liggen. Dit aantal is te laag om een regressie te kunnen uitvoeren, bij één punt omdat dit onmogelijk is, bij twee punten omdat de regressie dan teveel wordt beïnvloed door uitschieters. Uiteindelijk is slechts één deel van de onderzoeksperiode geschikt gebleken, zodat alleen binnen dit deel de snelheid van de betrouwbaarheidsgroei, $\alpha^{conventioneel}$, is vastgesteld voor verschillende wegingsfactoren.

De gevonden waarden voor α is voor beide systemen in Tabel 12 weergegeven.

Wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\}$	$\alpha^{conv.}$	$R^2_{conv.}$	α^{CBD}	R^2_{CBD}	$\alpha^{CBD}: \alpha^{conv.}$
{1, 1, 1, 1, 1}	0,0161	0,5422	0,0404	0,8001	2,5
{4, 5, 6, 7, 8}	0,0196	0,7680	0,0373	0,7874	1,9
{1, 2, 3, 4, 5}	0,0234	0,8164	0,0348	0,7736	1,5

Tabel 12: Betrouwbaarheidsindicatoren α^{CBD} en $\alpha^{conventioneel}$

Er is een aantal zaken opvallend aan deze meetgegevens. Ten eerste is duidelijk te zien dat α^{CBD} bij elke combinatie van wegingsfactoren hoger is dan $\alpha^{conventioneel}$. Aan kolom '$\alpha^{CBD}: \alpha^{conventioneel}$', die de verhouding tussen α^{CBD} en $\alpha^{conventioneel}$ aangeeft, is te zien dat deze waarde anderhalf tot twee-en-een-half keer zo hoog is. Vanwege de lage waarde van $R^2_{conventioneel}$ bij wegingsfactoren $\{w_{Trivial}, w_{Minor}, w_{Major}, w_{Critical}, w_{Blocker}\} = \{1, 1, 1, 1, 1\}$ moeten we vooral met deze waarde van $\alpha^{conventioneel}$ voorzichtig zijn. Daarnaast valt op dat de verhouding tussen α^{CBD} en $\alpha^{conventioneel}$ sterk beïnvloed wordt door de keuze van de wegingsfactoren. Het soort defect speelt dus een grote rol. Tot slot neemt $\alpha^{conventioneel}$ toe naarmate ernstiger defecten relatief zwaarder meetellen, terwijl α^{CBD} in dat geval juist afneemt.

Met betrekking tot de initiële betrouwbaarheid, d_0 ofwel WDD_0, zijn de resultaten minder betrouwbaar. De meetgegevens van de eerste versies van beide systemen zijn weergegeven in Tabel 13.

Systeem	Datum	WDD_0 {1, 1, 1, 1, 1}	WDD_0 {4, 5, 6, 7, 8}	WDD_0 {1, 2, 3, 4, 5}
CBD	31-08-2004	2,7985	17,2575	8,8619
conventioneel	25-07-2003	0,1373	0,7455	0,3335

Tabel 13: Meetgegevens initiële betrouwbaarheid

Op het eerste gezicht lijken de gegevens aan te tonen dat d_0^{CBD} een stuk hoger ligt dan $d_0^{conventioneel}$. Indien we nader kijken naar de versies van het conventionele systeem, dan blijkt dat we hier echter niet de waarde d_0 hebben gemeten, maar de waarde $d(t)$ op een tijdstip dat later ligt dan t_0. Het betreft hier versie 2.3, zodat we hier niet kunnen spreken van $d_0^{conventioneel}$. Omdat deze waarde onbekend blijft, kunnen we deze niet vergelijken met d_0^{CBD}.

7 Conclusie

7.1 Inleiding

Nu de resultaten van het onderzoek bekend zijn, wordt het tijd om antwoord te geven op de probleemstelling. Aan de hand van de in hoofdstuk 4 opgesteld hypothesen trekken we hierover in paragraaf 7.2 de conclusies. Daarna kijken we in paragraaf 7.3 terug op het onderzoek, wat ten doel had om vast te stellen of de invloed van CBD op de betrouwbaarheid van software vastgesteld kon worden. Het betrof een verkennend onderzoek, waarmee geprobeerd werd een kwalitatieve invloed vast te stellen. Desondanks zijn er vanwege de uitvoerbaarheid van het onderzoek en de beschikbaarheid van gegevens noodgedwongen enkele beperkende voorwaarden vastgesteld, welke de bruikbaarheid van het onderzoek en de situaties waarin het onderzoek relevant is, inperken. Deze beperkingen geven aanleiding tot suggesties voor verbetering. Daarnaast roept het onderzoek ook nieuwe vragen op. Met deze suggesties en vragen wordt in paragraaf 7.3 afgesloten.

7.2 Conclusie

Aan het begin van deze scriptie is de volgende centrale probleemstelling opgesteld.

> Wordt de betrouwbaarheid van software verbeterd door gebruik te maken van component-based development?

Probleemstelling

Op basis van deze probleemstelling zijn de volgende twee doelstellingen geformuleerd die deze vraag moeten beantwoorden.

1. Het verschaffen van inzicht in de kenmerken waarin component-based development verschilt van andere vormen van softwareontwikkeling.
2. Het vaststellen van de gevolgen van deze verschillen voor de dynamische betrouwbaarheid van de software.

Doelstellingen 1 en 2

Door het uitvoeren van een literatuuronderzoek is een duidelijk beeld ontstaan over de verschillen tussen CBD en conventionele softwareontwikkeling. Hierbij is gekeken naar de kwaliteit van software en de specifieke aspecten van CBD die deze kwaliteit beïnvloeden. Dit ontstane beeld verschaft het inzicht waarnaar verwezen wordt in doelstelling 1, zodat aan deze doelstelling is voldaan.

Teneinde doelstelling 2 te realiseren, is het begrip betrouwbaarheid geoperationaliseerd. Deze operationalisering heeft geleid tot de volgende twee hypothesen:

1. De initiële betrouwbaarheid van een herbruikbaar component is lager dan de initiële betrouwbaarheid van een vergelijkbaar specifiek component.

2. De snelheid waarmee de betrouwbaarheid van software toeneemt is hoger indien gebruik gemaakt wordt van CBD.

Hypothesen

Teneinde deze hypothesen te kunnen toetsen is het begrip betrouwbaarheid nader geoperationaliseerd. Hierbij is rekening gehouden met de praktische haalbaarheid, waarbij het al dan niet aanwezig of reconstrueerbaar zijn van gegevens een cruciale rol speelde. Vervolgens is een model geintroduceerd dat een verondersteld verband tussen de betrouwbaarheid en de tijd bevatte. De introductie van dit model heeft geleid tot de volgende operationalisering van de hypothesen.

$$d_0^{CBD} > d_0^{conventioneel}$$

$$\left|\alpha^{CBD}\right| > \left|\alpha^{conventioneel}\right|$$

Geoperationaliseerde hypothesen

Het uitvoeren van de toets heeft geleid tot een vaststelling van meetresultaten. Aan de hand van deze meetresultaten kunnen we nu vaststellen of de hypothesen gefalsificeerd kunnen worden of dat deze overeind blijven.

Met betrekking tot hypothese 1 hebben we, ondanks de gemeten waarden voor d_0^{CBD} en $d_0^{conventioneel}$, moeten constateren dat er onvoldoende meetresultaten beschikbaar waren om hierover uitspraken te kunnen doen. We kunnen hypothese 1 niet verwerpen, maar we kunnen ook niet aantonen dat de hypothese juist is.

Met betrekking tot hypothese 2 kunnen we meer vaststellen. Onder de aanname dat de systemen gedurende de onderzoeksperiode stabiel moeten zijn, hebben we voor beide systemen indicatoren kunnen vaststellen voor de snelheid van de betrouwbaarheidstoename. Hierbij hebben we een drietal verzamelingen wegingsfactoren gebruikt. De indicatoren geven voor elke verzameling wegingsfactoren met gemiddeld tot hoge betrouwbaarheid aan dat de snelheid van de betrouwbaarheidstoename bij het CBD-systeem, α^{CBD}, hoger ligt dan de snelheid van de betrouwbaarheidstoename bij het conventionele systeem, $\alpha^{conventioneel}$. We hebben met andere woorden aangetoond dat geldt: $|\alpha^{CBD}|>|\alpha^{conventioneel}|$. We hebben hiermee vastgesteld, dat we geen reden hebben om hypothese 2 te verwerpen. De betrouwbaarheid van software welke is ontwikkeld met

gebruikmaking van CBD groeit sneller dan die van conventionele software. De grootte van dit verschil is afhankelijk van de keuze voor de wegingsfactoren van de verschillende categorieën defecten.

De vraag is nu of we met deze vaststellingen de centrale probleemstelling kunnen beantwoorden. Vanwege het feit dat de betrouwbaarheid van software ontwikkeld met CBD sneller groeit dan de betrouwbaarheid van software die op conventionele wijze is ontwikkeld, kunnen we stellen deze hypothese op basis van onze bevindingen niet verworpen kan worden.

Wel moet hierbij de opmerking worden geplaatst dat de vastgestelde waarden van α^{CBD} en $\alpha^{conventioneel}$ gebaseerd zijn op betrekkelijk weinig gegevens en dat er na dit verkennende onderzoek nader onderzoek moet volgen om deze constatering te bevestigen.

7.3 Verder onderzoek

Zoals in de inleiding van dit hoofdstuk gesteld is, zijn er suggesties voor het verbeteren of uitbreiden van het onderzoek. Ook zijn er nieuwe vragen die naar aanleiding van de gevonden antwoorden opgeroepen worden. De volgende suggesties en vragen vormen derhalve het einde van deze scriptie en mogelijk het begin van nieuw onderzoek.

Loslaten van de voorwaarde van een stabiel systeem

Een grote beperking van het onderzoek was de voorwaarde om slechts te kijken naar periodes waarin het systeem stabiel is, waarmee werd geïmpliceerd dat de hoeveelheid adaptief en perfectief onderhoud aan het systeem minimaal moest zijn. Dat dit een zeer beperkende voorwaarde is, blijkt uit het gegeven dat van de 29 versies van het conventionele systeem slechts vier opvolgende versies stabiel voldeden aan deze voorwaarde. Indien de invloed van adaptief en perfectief onderhoud ook wordt opgenomen in het model (bijvoorbeeld door de relatie tussen de hoeveelheid gewijzigde code en het gewogen aantal opgeleverde adaptive en perfective maintenance issues enerzijds en het verwachte aantal defecten anderzijds te kwantificeren), stijgt de toepasbaarheid van het onderzoek enorm.

Uitbreiden onderzoek tot economisch model

Het onderzoek heeft vastgesteld dat CBD leidt tot een snellere betrouwbaarheidsgroei dan conventionele ontwikkeling. Hiermee is echter nog niet aangetoond dat het op grond hiervan een economisch verantwoorde keuze is om gebruik te gaan of blijven maken van CBD. Mogelijk leidt CBD tot zoveel extra inspanning dat de toename in kosten de toename in opbrengst overtreffen. Om dit te onderzoeken moeten alle relevante kosten en baten van het toepassen van CBD worden opgenomen in het model. Aspecten als de economische schade van een slecht image tengevolge van defecten, het overstappen van consumenten naar andere leveranciers tengevolge van defecten en de economische

waarde van aansprakelijkheidsrisico's tengevolge van schade opgelopen door defecten spelen hierbij een rol.

Bevestigen verkennende onderzoeksresultaten

Aangezien dit onderzoek van een verkennende aard is, is het belangrijk dat de resultaten in nadere onderzoeken worden bevestigd. Ondanks de summiere gegevens die beschikbaar waren, geven de resultaten toch een duidelijke indicatie van een positieve invloed van CBD op de betrouwbaarheidsontwikkeling. Toch dient het onderzoek zonder meer vaker of met betere onderzoeksobjecten te worden uitgevoerd voordat gesproken kan worden over een sterk aangetoond verband. Aspecten die kunnen worden verbeterd zijn bijvoorbeeld meer gegevens, maar ook onderzoeksobjecten die qua grootte en gebruik dichter bij elkaar liggen.

Invloed van standaardsoftware

Dit onderzoek is uitgevoerd met twee maatwerkprojecten. Omdat maatwerk op basis van specifieke wensen van de opdrachtgever wordt ontwikkeld, wordt de geleverde functionaliteit vrijwel allemaal gebruikt. Standaardsoftware verschilt in diverse aspecten van maatwerksoftware. Standaardsoftware wordt meestal op grotere schaal gebruikt. Ook bevat standaardsoftware doorgaans een veel bredere hoeveelheid functionaliteit, omdat een poging wordt gedaan een zo groot mogelijke markt met de standaardsoftware te bereiken. Een gevolg hiervan is, dat naar verwachting een groter deel van de geleverde functionaliteit niet gebruikt wordt door de 'gemiddelde' gebruiker.

Het op grotere schaal gebruiken van de software leidt vermoedelijk tot een snellere groei van de betrouwbaarheid. Echter, doordat de 'gemiddelde' gebruiker vermoedelijk niet de volledige functionaliteit zal benutten, is het mogelijk dat sommige defecten langer onopgemerkt in de software achterblijven. Dus ondanks dat CBD vermoedelijk ook – of juist – bij de ontwikkeling van standaardsoftware leidt tot een snellere toename van de betrouwbaarheid, is het interessant het netto effect van bovenstaande effecten te onderzoeken.

Bijlagen

Bijlage 1 Afkortingen

ASD Adaptive Software Development

Agile ontwikkelmethodologie, ontwikkeld door Jim Highsmith, welke gericht is op het vroegtijdig onderkennen van en inspelen op verandering. (website: http://www.adaptivesd.com/)

CBD Component-based development

Het ontwikkelproces waarbij software wordt geassembleerd met behulp van kant en klare componenten.

CBSD Component-based software development

Zie CBD

CBSE Component-based software engineering

Wijze van het ontwikkelen van software waarbij reeds ontwikkelde componenten worden geassembleerd tot gehele applicaties of systemen.

CVS Concurrent Versioning System

Open source versiebeheersysteem waarmee wijzigingen in broncode kunnen worden beheer, ontwikkeld door Brian Berliner in 1989, gebaseerd op scripts van Dick Grune. (websites: http://www.gnu.org/software/cvs/ en https://www.cvshome.org/) CVS is een command-line applicatie, maar er zijn diverse grafische add-ons beschikbaar zoals WinCVS voor Windows. (website: http://www.wincvs.org/)

Zie ook SCC

DD Defect Density

De foutdichtheid van software, uitgedrukt in het aantal fouten ten opzichte van de grootte van software.

DSDM Dynamic Systems Development Method

Agile ontwikkelmethodologie ontwikkeld door een consortium van bedrijven, oorspronkelijk gebaseerd op RAD en iteratief ontwikkelen. Onderliggende principes zijn actieve interactie met gebruikers, frequente oplevering, gevolmachtigde teams en testen gedurende de ontwikkelcycli. (website: http://www.dsdm.org/)

EJB Enterprise JavaBeans

Gedistribueerd component model van Sun Microsystems voor objecten die business logica representeren, geschikt voor multi-tier client/server systemen. (website: http://java.sun.com/products/ejb/)

FDD Feature Driven Development

Agile ontwikkelmethodologie ontwikkeld door Jeff De Luca en Peter Coad, waarin in iteraties van twee weken tastbare functionaliteit wordt opgeleverd. Initieel wordt een project opgedeeld in features, welke worden ingepland. Per iteratie wordt vervolgens een feature ontworpen, gebouwd en opgeleverd, waarbij veel in teamverband wordt gewerkt. (website: http://www.featuredrivendevelopment.com/)

IEC International Electrotechnical Commission

Toonaangevende organisatie die zich bezighoudt met het opstellen van standaarden. (website: http://www.iec.ch)

IEEE Institute of Electrical and Electronics Engineers, Inc.

's Werelds grootste professionele gemeenschap gericht op technische aangelegenheden. (website: http://www.ieee.org/)

ISO International Organization for Standardization

Toonaangevende organisatie die zich bezighoudt met het opstellen van standaarden. (website: http://www.iso.org)

JAI Java Advanced Imaging

Verzameling functionaliteit voor het manipuleren van plaatjes in Java applicaties en applets.

JIT Just In Time

Methodiek waarbij verwerking op het laatste moment plaatsvindt.

JVM Java Virtual Machine

Softwarematige computer die Java byte-code – code in machine-onafhankelijke vorm – welke is opgeslagen in een class-bestand, omzet en uitvoert. Dit omzetten kan gedaan worden door een run-time interpreter, of een Just-In-Time (JIT) compiler, welke er machine-afhankelijke code van maakt.

KLOC Kilo Lines Of Code

Het aantal LOC in duizendtallen.

LOC Lines of Code

Het aantal coderegels van software. Eenheid waarin de grootte van software kan worden uitgedrukt.

MTBF Mean Time Between Failures

De gemiddelde tijd die een apparaat in bedrijf is voordat een storing optreedt.

MTTF Mean Time To Failure

De tijd die een apparaat in bedrijf is voordat de eerste storing optreedt.

RAD Rapid Application Development

Ontwikkelmethodologie die in 1991 is geintroduceerd door James Martin. Door middel van een aantal technieken als prototyping, iteratief ontwikkelen en timeboxing wordt gestreefd naar snelle oplevering.

RMI Remote Method Invocation

Technologie die Java-programma's in staat stelt om objecten van andere Java programma's op andere computers te benaderen.

SCC Source Code Control

Systematiek om verschillende versies van broncode te archiveren en op te halen. Deze omvat meestal uitgebreide mogelijkheden, zoals het splitsen van een versie in meerdere vertakkingen (branching), waarbij één vertakking uitsluitend correcties van defecten bevat en een andere versie nieuwe functionaliteit. Deze vertakkingen kunnen op een later punt weer samengevoegd worden (merging). Source code control systemen worden ook versiebeheersystemen genoemd. Voorbeelden zijn Microsoft SourceSafe en CVS.

WAPMI Weighted Adaptive and Perfective Maintenance Issues

De gewogen onderhoudsissues die betrekking hebben op het toevoegen van functionaliteit aan het systeem of het verbeteren van bestaande functionaliteit. De wegingsfactoren geven hierbij het belang aan dat wordt gehecht aan de verschillende categorieën prioriteiten.

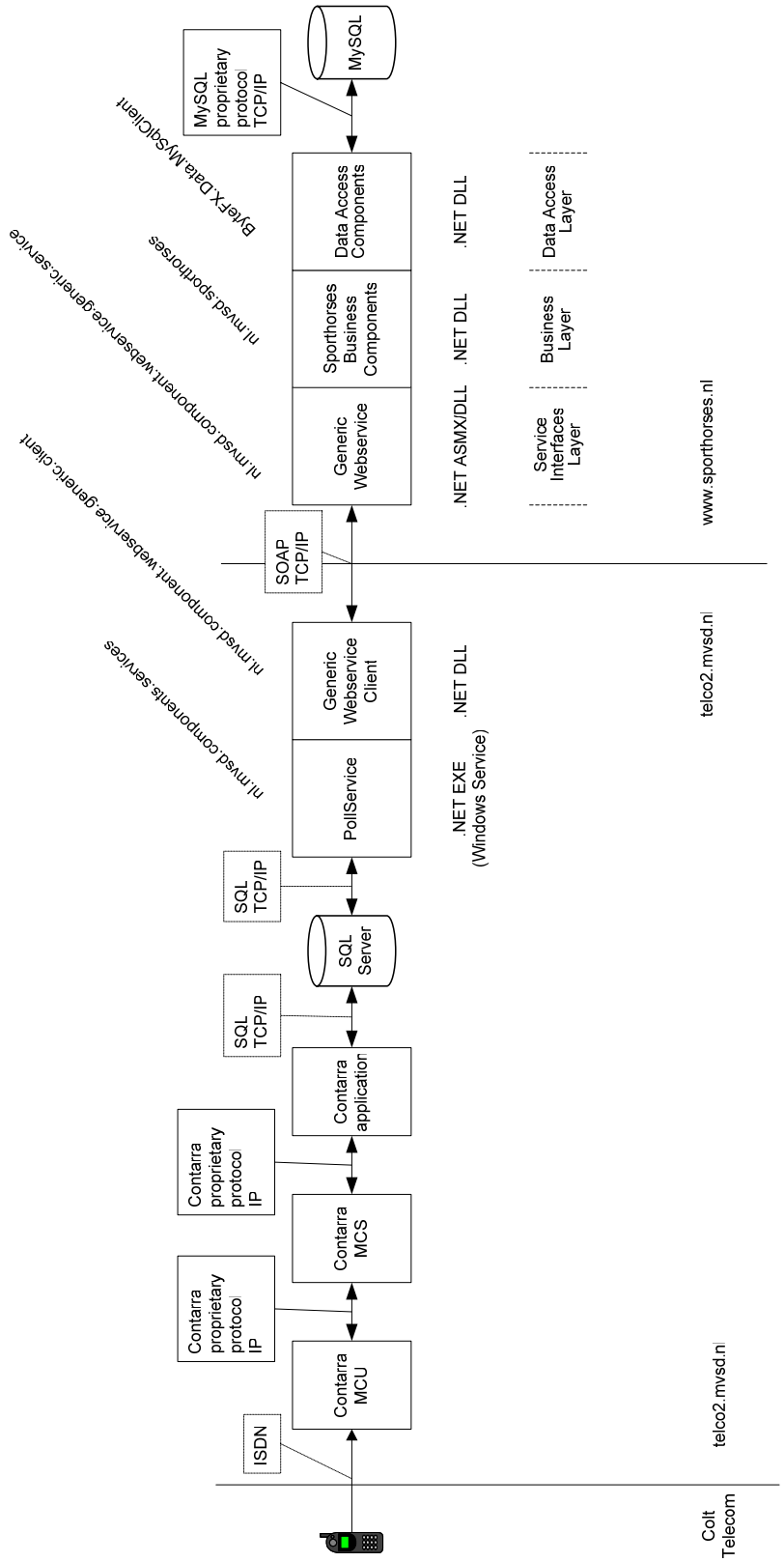
WDD Weighted Defect Density

De gewogen dichtheid van defecten in de software. De wegingsfactoren geven hierbij het belang aan dat wordt gehecht aan de verschillende categorieën prioriteiten.

XP eXtreme Programming

Meest bekende Agile ontwikkelmethodologie, ontwikkeld door Kent Beck. XP is gebaseerd op de waarden communicatie, terugkoppeling, eenvoud en lef en maakt gebruik van een veelvoud aan technieken als kleine releases, geautomatiseerd testen van alle code, refactoring en continue integratie. (website: http://www.extremeprogramming.org/)

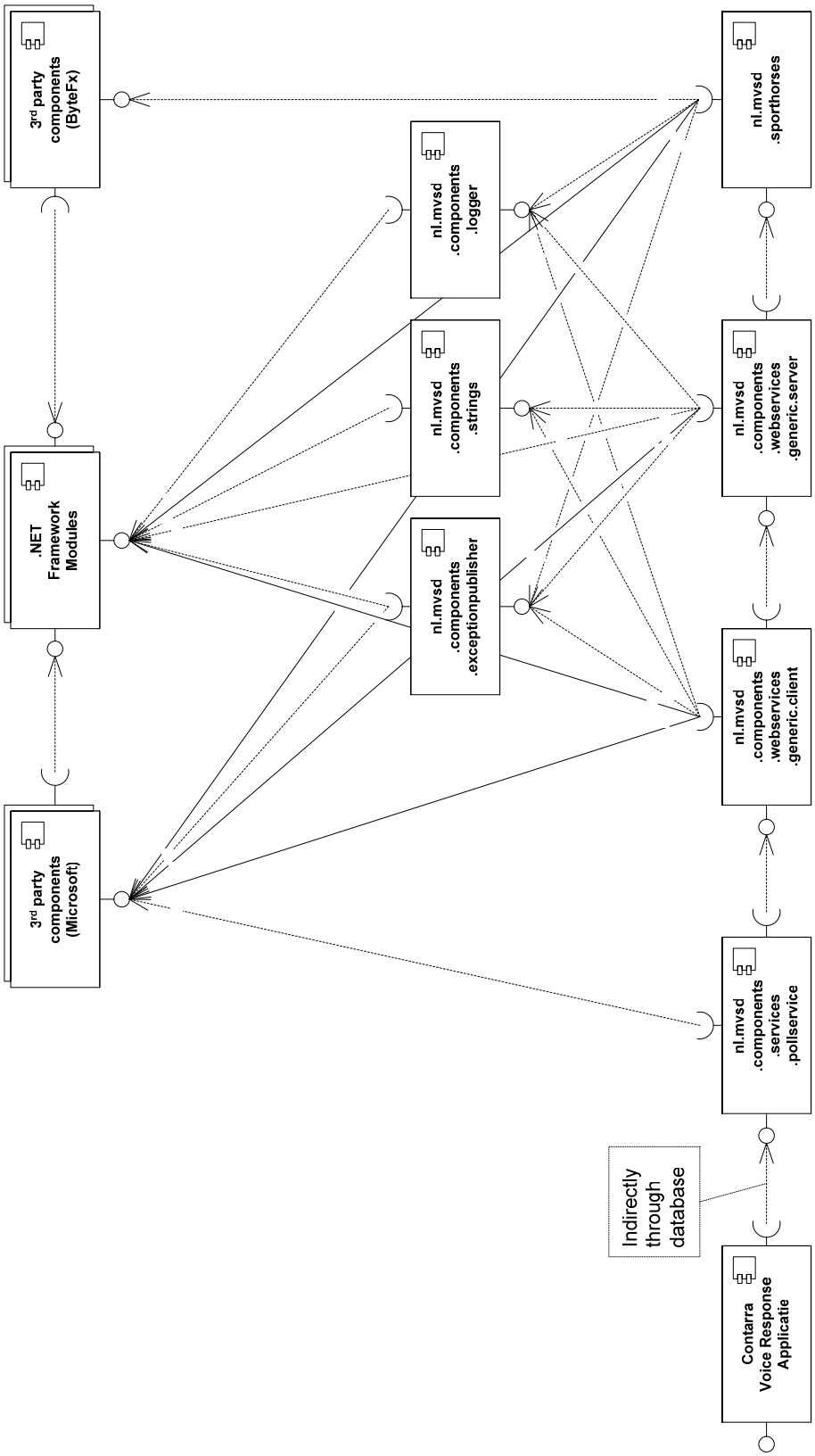
Bijlage 2 Architectuur CBD-systeem



Figuur 54: Architectuur CBD-systeem

Bijlage 3 Componenten CBD-systeem

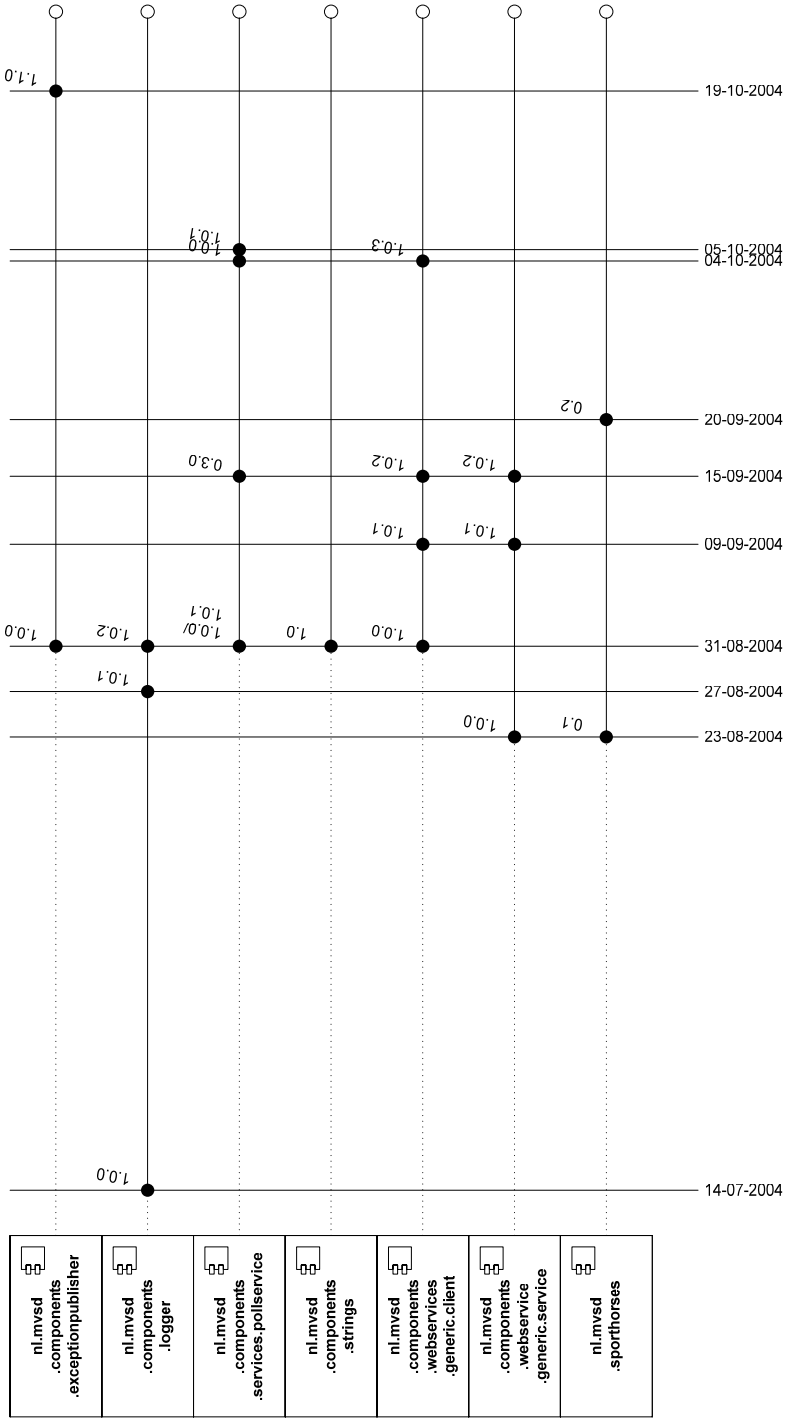
De componenten van het CBD-systeem worden in Figuur 55 weergegeven.



Figuur 55: Totaaloverzicht componenten CBD-systeem

Bijlage 4 Versies CBD-systeem

De versies van het CBD-systeem worden bepaald door de versies van de componenten. Indien één of meer componenten worden vervangen door nieuwe versies, wordt een nieuwe versie van het systeem onderkend. In Figuur 56 worden de componenten van het CBD-systeem weergegeven.



Figuur 56: Versies CBD-systeem

De componenten staan naast elkaar, de tijd is uitgezet over de verticale as. Een nieuwe versie van een component wordt weergegeven door een gesloten rondje (●). Deze versie blijft in gebruik totdat er een nieuwere versie in gebruik wordt genomen. De data waarop wijzigingen van componenten hebben plaatsgevonden zijn 14-07-2004, 23-08-2004, 27-08-2004, 31-08-2004, 09-09-2004, 15-09-2004, 20-09-2004, 04-10-2004, 05-10-2004 en 19-10-2004. Aangezien de eerste versie van het systeem waarin een substantieel deel van alle componenten was opgenomen pas op 31-08-2004 is opgeleverd, wordt deze versie van het systeem beschouwd als de eerste versie.

Bijlage 5 Meetgegevens componenten CBD-systeem

De meetgegevens van het CBD-systeem zijn per component verzameld voor de volgende componenten:

- nl.mvsd.components.exceptionpublisher
- nl.mvsd.components.logger
- nl.mvsd.components.services.pollservice
- nl.mvsd.components.strings
- nl.mvsd.components.webservices.generic.client
- nl.mvsd.components.webservices.generic.service
- nl.mvsd.sporthorses

Per component worden de naam van het project in het issue tracking systeem, de naam van het project in het versiebeheersysteem en de referenties naar andere componenten gegeven. Vervolgens wordt van elke versie van het component de gegevens met betrekking tot totale grootte van de code in LOC, grootte van de nieuwe en gewijzigde code in LOC, en aantal defecten gegeven. Daarna wordt een overzicht van de issues van het component gegeven, bestaande uit nummer, type en prioriteit. Hierbij wordt tevens aangegeven of een issue is meegenomen in het onderzoek. Redenen voor het niet opnemen van een issue zijn bijvoorbeeld het niet kunnen reproduceren van een defect, aanpassingen die betrekking hebben op configuratie van het component en aanpassingen die geen invloed hebben de werking van het component, zoals het veranderen van de namespace[11] van het component. Ook is het mogelijk dat de issues van twee componenten in het issue tracking systeem onder hetzelfde project worden bijgehouden, zodat issues slechts bij één component worden meegeteld. Tot slot is het aantal defecten per versie van het component gegeven.

[11] Een namespace is een groepering van gegevenstypen [Robinson et al., 2002].

Component nl.mvsd.components.exceptionpublisher

Naam	Waarde(n)
Project in issue tracking systeem	MVSD .NET Components – ExceptionPublisher (COMEXH)
Project in versiebeheersysteem	MvsdComponents -> ExceptionPublisher
Referenties naar andere componenten	▪ Microsoft.ApplicationBlocks.ExceptionManagement ▪ Microsoft.ApplicationBlocks.ExceptionManagement .Interfaces ▪ System ▪ System.Data ▪ System.Windows.Forms ▪ System.XML

Tabel 14: Gegevens component nl.mvsd.components.exceptionpublisher

Versie	LOC	ΔLOC	#defecten
1.0.0 (31-08-2004)	275	275	1
1.1.0 (19-10-2004)	358	84	1

Tabel 15: Meetgegevens component nl.mvsd.components.exceptionpublisher

Issue	Type	Prioriteit
COMEXH-1	Bug	ja
COMEXH-2	Niet meegeteld	n.v.t.
COMEXH-3	Niet meegeteld	n.v.t.

Tabel 16: Issues component nl.mvsd.components.exceptionpublisher

Defect	1.0.0 (31-08-2004)	1.1.0 (19-10-2004)
COMEXH-1	ja	ja
Totaal	**1**	**1**

Tabel 17: Defecten component nl.mvsd.components.exceptionpublisher

Component nl.mvsd.components.logger

Naam	Waarde(n)
Project in issue tracking systeem	MVSD .NET Components – Logger (COMLOG)
Project in versiebeheersysteem	MvsdComponents -> Logger
Referenties naar andere componenten	▪ System ▪ System.Data ▪ System.XML

Tabel 18: Gegevens component nl.mvsd.components.logger

Versie	LOC	ΔLOC	#defecten
1.0.0 (14-07-2004)	50	50	0
1.0.1 (27-08-2004)	67	24	0
1.0.2 (31-08-2004)	78	18	0

Tabel 19: Meetgegevens component nl.mvsd.components.logger

Issue	Type	Prioriteit
COMLOG-1	Improvement	Critical
COMLOG-2	niet meegeteld	n.v.t.

Tabel 20: Issues component nl.mvsd.components.logger

Vanwege de afwezigheid van defecten bij dit component worden deze niet per versie weergegeven.

Component nl.mvsd.components.services.pollservice

Naam	Waarde(n)
Project in issue tracking systeem	Presta Webdesign Sporthorses.nl (SPH)
Project in versiebeheersysteem	SportHorses -> pollservice
Referenties naar andere componenten	▪ Microsoft.ApplicationBlocks.Data ▪ Microsoft.ApplicationBlocks.ExceptionManagement ▪ Microsoft.ApplicationBlocks.ExceptionManagement. Interfaces ▪ nl.mvsd.components.exceptionpublisher ▪ nl.mvsd.components.logger ▪ nl.mvsd.components.strings ▪ nl.mvsd.components.webservices.generic.client ▪ System ▪ System.Configuration.Install ▪ System.Data ▪ System.Management ▪ System.ServiceProcess ▪ System.XML

Tabel 21: Gegevens component nl.mvsd.components.services.pollservice

Versie	LOC	ΔLOC	#defecten
1.0.0 (31-08-2004)	220	220	2
1.0.1 (31-08-2004)	220	8	2
0.3.0 (15-09-2004)	227	8	2
1.0.0 (04-10-2004)	235	12	2
1.0.1 (05-10-2004)	239	5	0

Tabel 22: Meetgegevens component nl.mvsd.components.services.pollservice

Issue	Type	Prioriteit
SPH-2	Improvement	Major
SPH-3	Bug	Major
SPH-4	niet meegeteld	n.v.t.
SPH-5	Bug	Critical
SPH-6	niet meegeteld	n.v.t.
SPH-7	niet meegeteld	n.v.t.
SPH-8	niet meegeteld	n.v.t.
SPH-9	Bug	Major

Tabel 23: Issues component nl.mvsd.components.services.pollservice

Defect	1.0.0 (31-08-2004)	1.0.1 (31-08-2004)	0.3.0 (15-09-2004)	1.0.0 (04-10-2004)	1.0.1 (05-10-2004)
SPH-3	ja	ja	ja	nee	nee
SPH-5	nee	nee	nee	ja	nee
SPH-9	ja	ja	ja	ja	nee
Totaal	**2**	**2**	**2**	**2**	**0**

Tabel 24: Defecten component nl.mvsd.components.services.pollservice

Component nl.mvsd.components.strings

Naam	Waarde(n)
Project in issue tracking systeem	MVSD .NET Components – Strings (COMSTR)
Project in versiebeheersysteem	MvsdComponents -> Strings
Referenties naar andere componenten	▪ System ▪ System.Data ▪ System.XML

Tabel 25: Gegevens component nl.mvsd.components.strings

Versie	LOC	ΔLOC	#defecten
1.0 (31-08-2004)	630	630	0

Tabel 26: Meetgegevens component nl.mvsd.components.strings

Vanwege de afwezigheid van issues wordt hiervan geen overzicht gegeven. Er zijn dus geen defecten, zodat deze ook niet per versie worden weergegeven.

Component nl.mvsd.components.webservices.generic.client

Naam	Waarde(n)
Project in issue tracking systeem	MVSD .NET Components – Generic Webservice Client (COMGWC)
Project in versiebeheersysteem	MvsdComponents -> Generic Webservice -> Client
Referenties naar andere componenten	▪ Microsoft.ApplicationBlocks.ExceptionManagement ▪ Microsoft.ApplicationBlocks.ExceptionManagement. Interfaces ▪ nl.mvsd.components.exceptionpublisher ▪ nl.mvsd.components.logger ▪ nl.mvsd.components.strings ▪ System ▪ System.Data ▪ System.EnterpriseServices ▪ System.Messaging ▪ System.Web ▪ System.Web.Services ▪ System.XML

Tabel 27: Gegevens component nl.mvsd.components.webservices.generic.client

Versie	LOC	ΔLOC	#defecten
1.0.0 (31-08-2004)	567	567	2
1.0.1 (09-09-2004)	566	14	2
1.0.2 (15-09-2004)	588	45	0
1.0.3 (05-10-2004)	635	97	0

Tabel 28: Meetgegevens component nl.mvsd.components.webservices.generic.client

Issue	Type	Prioriteit
COMGWC-1	Bug	Critical
COMGWC-2	Bug	Minor

Tabel 29: Issues component nl.mvsd.components.webservices.generic.client

Defect	1.0.0 (31-08-2004)	1.0.1 (09-09-2004)	1.0.2 (15-09-2004)	1.0.3 (05-10-2004)
COMGWC-1	ja	ja	nee	nee
COMGWC-2	ja	ja	nee	nee
Totaal	**2**	**2**	**0**	**0**

Tabel 30: Defecten component nl.mvsd.components.webservices.generic.client

Component nl.mvsd.components.webservices.generic.service

Naam	Waarde(n)
Project in issue tracking systeem	MVSD .NET Components – Generic Webservice Service (COMGWS)
Project in versiebeheersysteem	MvsdComponents -> Generic Webservice -> Service
Referenties naar andere componenten	▪ Microsoft.ApplicationBlocks.ExceptionManagement ▪ Microsoft.ApplicationBlocks.ExceptionManagement. Interfaces ▪ nl.mvsd.components.exceptionpublisher ▪ nl.mvsd.components.logger ▪ nl.mvsd.components.strings ▪ System ▪ System.Data ▪ System.Web ▪ System.Web.Services ▪ System.XML ▪ System.Drawing

Tabel 31: Gegevens component nl.mvsd.components.webservices.generic.service

Versie	LOC	ΔLOC	#defecten
1.0.0 (23-08-2004)	230	230	1
1.0.1 (09-09-2004)	229	11	1
1.0.2 (15-09-2004)	256	33	0

Tabel 32: Meetgegevens component nl.mvsd.components.webservices.generic.service

Issue	Type	Prioriteit
COMGWS-1	Bug	Major
COMGWS-2	Bug	Major
COMGWS-3	niet meegeteld	n.v.t.

Tabel 33: Issues component nl.mvsd.components.webservices.generic.service

Defect	1.0.0 (23-08-2004)	1.0.1 (09-09-2004)	1.0.2 (15-09-2004)
COMGWS-1	ja	nee	nee
COMGWS-2	nee	ja	nee
Totaal	**1**	**1**	**0**

Tabel 34: Defecten component nl.mvsd.components.webservices.generic.service

Component nl.mvsd.sporthorses

Naam	Waarde(n)
Project in issue tracking systeem	Presta Webdesign Sporthorses.nl (SPH)
Project in versiebeheersysteem	SportHorses -> sporthorses
Referenties naar andere componenten	▪ ByteFX.MySqlClient ▪ ByteFX.MySqlClient.Design ▪ Microsoft.ApplicationBlocks.ExceptionManagement .Interfaces ▪ Microsoft.ApplicationBlocks.ExceptionManagement ▪ nl.mvsd.components.exceptionpublisher ▪ nl.mvsd.components.logger ▪ nl.mvsd.components.strings ▪ System ▪ System.Data ▪ System.Web ▪ System.Web.Mobile ▪ System.XML

Tabel 35: Gegevens component nl.mvsd.sporthorses

Versie	LOC	ΔLOC	#defecten
0.1 (23-08-2004)	144	144	0
0.2 (20-09-2004)	148	66	0

Tabel 36: Meetgegevens component nl.mvsd.sporthorses

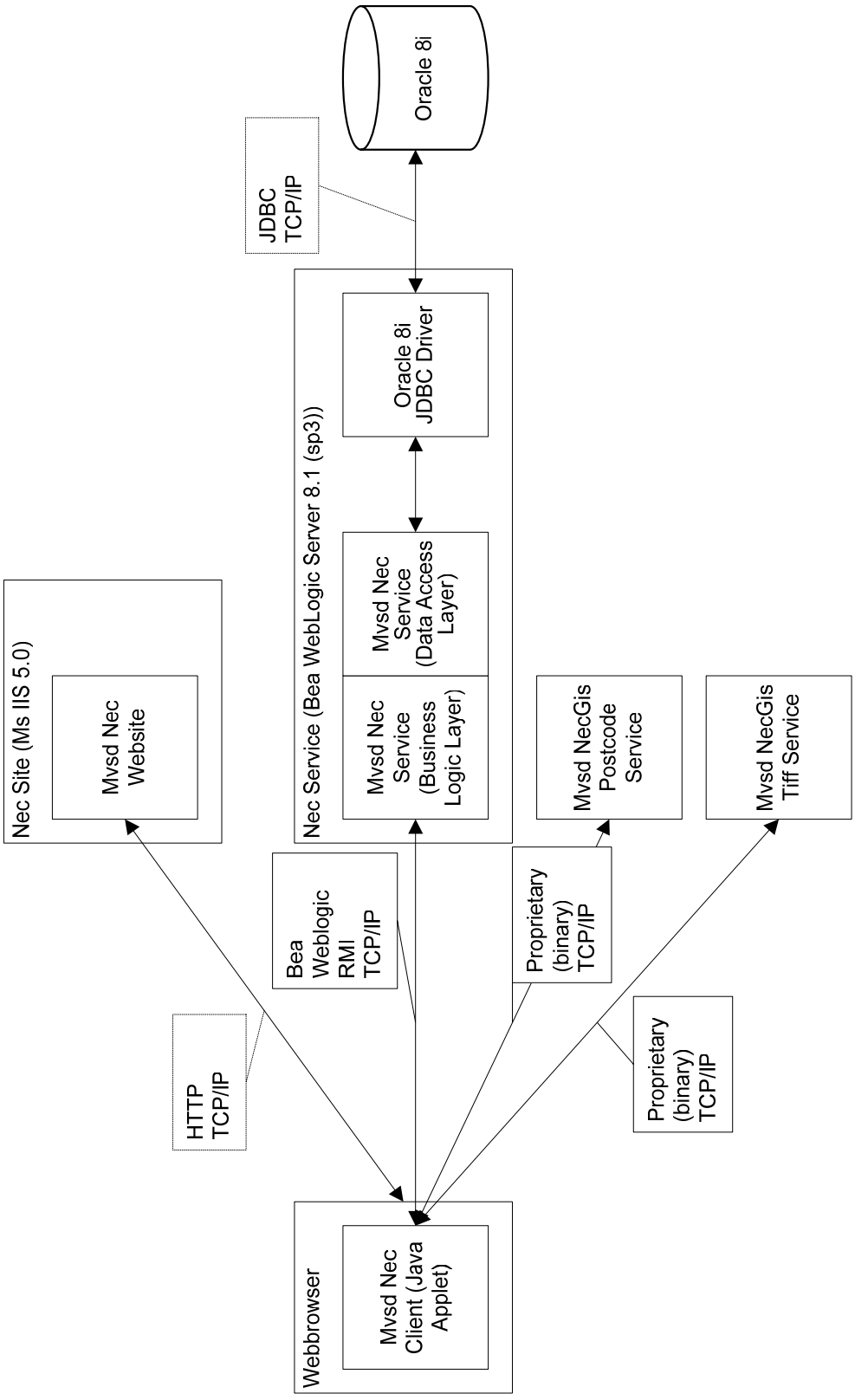
Issue	Defect	Prioriteit
SPH-2	niet meegeteld[12]	n.v.t.
SPH-3	niet meegeteld[12]	n.v.t.
SPH-4	niet meegeteld	n.v.t.
SPH-5	niet meegeteld[12]	n.v.t.
SPH-6	niet meegeteld	n.v.t.
SPH-7	niet meegeteld	n.v.t.
SPH-8	niet meegeteld	n.v.t.
SPH-9	niet meegeteld[12]	n.v.t.

Tabel 37: Issues component nl.mvsd.sporthorses

Vanwege de afwezigheid van defecten worden deze niet per versie weergegeven.

[12] Dit issue is hier niet meegeteld omdat het al is meegeteld bij component nl.mvsd.component.services.pollservice.

Bijlage 6 Architectuur conventioneel systeem



Figuur 57: Architectuur conventioneel systeem

Bijlage 7 Versies conventioneel systeem

De versies van het conventionele systeem die opgeleverd zijn vanaf het moment waarop issues werden bijgehouden én waarvan gegevens beschikbaar zijn in het versiebeheersysteem zijn weergegeven in Tabel 38. Hierin zijn versienummer en opleverdatum van elke versie indien bekend weergegeven. Indien een versie niet in meegenomen in het onderzoek is in kolom Opmerking de reden hiervan weergegeven. De twee oorzaken hiervoor zijn het niet bekend zijn van een opleverdatum en het niet beschikbaar zijn van broncode. Dit laatste is geconstateerd indien de opvolgende versies identiek aan elkaar zijn.

Versie	Datum	Opmerking
2.3	25-07-2003	
2.3.1	04-08-2003	
2.3.2	26-09-2003	
2.3.3	01-10-2003	niet opgenomen: code identiek aan 2.3.2
2.3.4	15-10-2003	
2.3.4.100	17-02-2004	
2.3.4.101	18-02-2004	niet opgenomen: code identiek aan 2.3.4.100
2.3.5	18-02-2004	niet opgenomen: code identiek aan 2.3.4.101
2.3.5.1	25-02-2004	
2.3.5.2	01-03-2004	
2.3.5.3	09-03-2004	
2.3.5.4	05-04-2004	
2.3.5.100	11-02-2004	
2.3.5.101		niet opgenomen: opleverdatum onbekend
2.3.5.102	04-03-2004	
2.3.5.103	08-03-2004	
2.3.5.104	09-03-2004	
2.3.5.105	09-03-2004	niet opgenomen: code identiek aan 2.3.5.104
2.3.5.106	11-03-2004	niet opgenomen: code identiek aan 2.3.4.105
2.3.5.107	23-03-2004	
2.3.5.108	25-03-2004	
2.3.5.109	29-03-2004	niet opgenomen: code identiek aan 2.3.4.108
2.3.5.110	30-03-2004	niet opgenomen: code identiek aan 2.3.4.109

Versie	Datum	Opmerking
2.3.5.111	08-04-2004	
2.4		niet opgenomen: opleverdatum onbekend
2.4.1	05-05-2004	
2.4.1.3	13-05-2004	niet opgenomen: code identiek aan 2.4.1
2.4.1.4	15-05-2004	
2.4.1.5	19-05-2004	
2.4.1.6	21-05-2004	
2.4.1.7	25-05-2004	
2.4.1.8	26-05-2004	
2.4.1.9	26-05-2004	niet opgenomen: code identiek aan 2.4.1.8
2.4.1.10	27-05-2004	
2.4.1.11	14-06-2004	niet opgenomen: code identiek aan 2.4.1.10
2.4.1.12	21-06-2004	
2.4.1.13	27-06-2004	
2.1.4.14	28-06-2004	niet opgenomen: code identiek aan 2.4.1.13
2.4.1.15	05-07-2004	
2.4.2	29-07-2004	
2.4.2.1	12-08-2004	
2.4.2.2	24-08-2004	
2.4.2.3	28-10-2004	niet opgenomen: code identiek aan 2.4.2.2
2.4.3	27-09-2004	niet opgenomen: code identiek aan 2.4.2.3

Tabel 38: Versies conventioneel systeem in issue tracking systeem

Bijlage 8 Ongeschikte perioden conventioneel systeem

De periode van het conventionele systeem die op grond van de drie stabiliteitscriteria (beperkte verandering grootte, beperkt aantal opgeleverde adaptive en perfective maintenance issues en niet meer dan gemiddelde testactiviteit) niet geschikt zijn voor het onderzoek worden in Tabel 39 weergegeven.

Startdatum	Einddatum	Reden
25-07-2003	15-10-2003	Veel testactiviteit op 04-08-2004
11-02-2004	18-02-2004	Hoge ΔLOC
25-02-2004	23-03-2004	▪ Hoge ΔLOC op 04-03-2004 ▪ Veel testactiviteit tussen 08-03-2004 en 23-03-2004
21-04-2004	21-04-2004	Nieuwe versie met veel issues
05-05-2004	26-05-2004	▪ Diverse nieuwe issues ▪ Hoge ΔLOC ▪ Gemiddeld testen
14-06-2004	21-06-2004	Te weinig meetpunten
27-06-2004		1 issue, gemiddeld testen
27-06-2004	05-07-2004	Te weinig meetpunten
29-07-2004		▪ Hoge ΔLOC ▪ Veel nieuwe issues
12-08-2004		Veel testen voorafgaand aan deze versie
12-08-2004	24-08-2004	Te weinig meetpunten

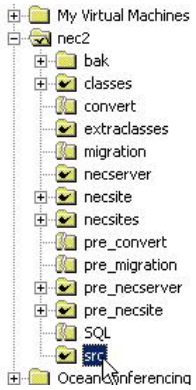
Tabel 39: Ongeschikte perioden conventioneel systeem

Bijlage 9 Procedure verkrijgen specifieke versie WinCvs

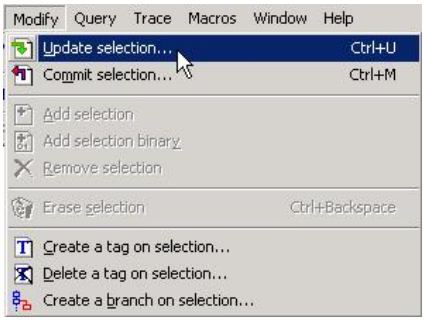
Voor het verkrijgen van een specifieke versie van een project dat met WinCvs wordt beheerd is de volgende procedure gevolgd. De gebruikte versie van WinCvs is 1.2.

1. Verwijder de map "nl\" uit de lokale map "\nec2\src\". Hierdoor kunnen er geen oude versies van bestanden achterblijven die de meetresultaten beïnvloeden.

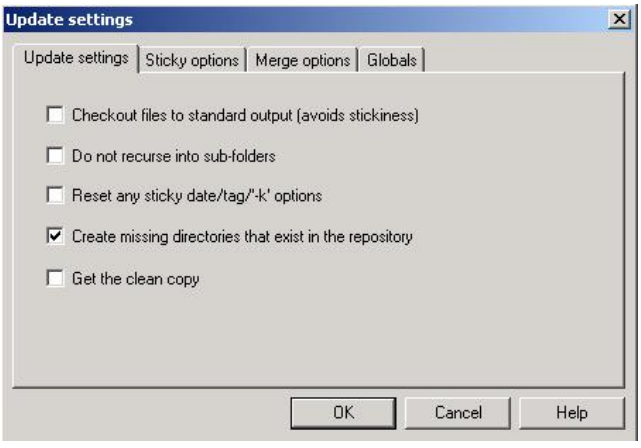
2. Selecteer in WinCvs de map die de bronbestanden bevat.



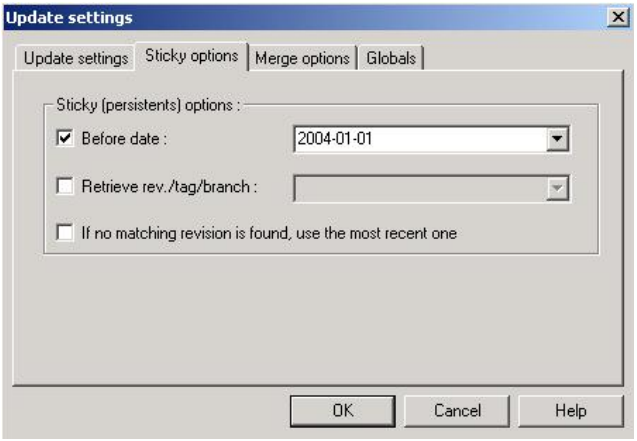
3. Open het dialoogvenster "Update Settings" door in het menu "Modify", "Update Selection…" te selecteren.



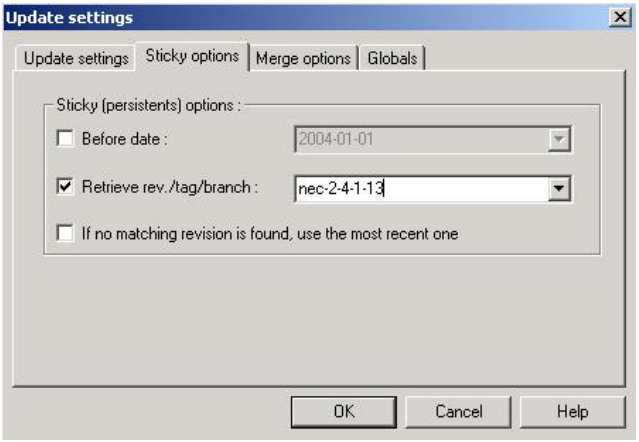
4. Selecteer in tabblad "Update settings" de optie "Create missing directories that exist in the repository".



5. Selecteer in tabblad "Sticky options" de versie door de datum of versienummer op te geven. Bij een datum wordt in het veld "Before date:" de datum volgend op de dag van oplevering van een versie opgegeven in het formaat yyyy-mm-dd.



Bij een versienummer wordt in het veld "Retrieve rev./tag/branch:" het versienummer opgegeven.



6. Haal de versie op door op de knop "OK" te klikken.

Bijlage 10 Procedure verzamelen bestede testuren

Het verzamelen van de aantallen uren die zijn besteed aan testactiviteiten per systeem is uitgevoerd door middel van het direct uitvoeren van sql select-statements op de database waarin het urenverantwoordingssysteem zijn gegevens opslaat. De algemene vorm van deze query is weergegeven in Code snippet 1. Hierin is [verzameling projectnummers] de verzameling van nummers van projecten waaronder de uren van respectievelijk het CBD-systeem en het conventionele systeem worden geregistreerd. De testactiviteiten zijn geselecteerd door de nummers van taken die onder testen vallen in te vullen bij in [verzameling nummer van testtaken].

```
SELECT
	h.[Date],
	SUM(h.Hours)
FROM
	tblProject AS p
	INNER JOIN tblprojecttask AS pt
	ON pt.projectno = p.[no]
	INNER JOIN tblTask AS t
	ON pt.TaskNo = t.[No]
	INNER JOIN tblHour AS h
	ON
	(
		h.ProjectNo = p.[No]
		AND h.TaskNo = t.[No]
	)
WHERE
	p.[No] IN ([verzameling projectnummers])
	AND t.[No] IN ([verzameling nummers van testtaken])
GROUP BY
	h.[Date]
ORDER BY
	h.[Date]
```

Code snippet 1: Algemene query voor ophalen besteden testuren.

De specifieke invulling van deze algemene query voor het CBD-systeem en het conventionele systeem zijn weergegeven in respectievelijk Code snippet 2 en Code snippet 3.

```
SELECT
	h.[Date],
	SUM(h.Hours)
FROM
	tblProject AS p
	INNER JOIN tblprojecttask AS pt
	ON pt.projectno = p.[no]
	INNER JOIN tblTask AS t
	ON pt.TaskNo = t.[No]
	INNER JOIN tblHour AS h
	ON
	(
		h.ProjectNo = p.[No]
		AND h.TaskNo = t.[No]
	)
WHERE
	p.[No] IN (270)
	AND t.[No] IN (53, 117, 51)
GROUP BY
	h.[Date]
ORDER BY
	h.[Date]
```

Code snippet 2: Query voor het ophalen van bestede testuren aan het CBD-systeem

```
SELECT
	h.[Date],
	SUM(h.Hours)
FROM
	tblProject AS p
	INNER JOIN tblprojecttask AS pt
	ON pt.projectno = p.[no]
	INNER JOIN tblTask AS t
	ON pt.TaskNo = t.[No]
	INNER JOIN tblHour AS h
	ON
	(
		h.ProjectNo = p.[No]
		AND h.TaskNo = t.[No]
	)
WHERE
	p.[No] IN (197, 233, 243, 266, 268)
	AND t.[No] IN (53, 117, 51)
GROUP BY
	h.[Date]
ORDER BY
	h.[Date]
```

Code snippet 3: Query voor het ophalen van aan het conventionele systeem bestede testuren

Bronnen

Agile Alliance, 2001 — Agile Alliance (Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas), *Agile Manifesto*, 2001

http://agilemanifesto.org/

Allen, 2001 — Paul Allen, Reuse in the Component Marketplace, Cutter Consortium, November 20th, 2001

http://www.cutter.com/research/2001/edge011120.html

downloaddatum: 30-01-2005

Augarten, 1985 — Stan Augarten, *Bit by bit, an illustrated history of computers*, Unwin Paperbacks, London, 1985, ISBN 0-04-001007-4

Bahrami, 1999 — Ali Bahrami, *Object-oriented systems development*, Irwin/McGraw-Hill, Singapore, International Edition, 1999, ISBN 0-256-25348-X

van den Berg et al., 1998 — Jan van den Berg, Mark Kilsdonk, Ed Ridderbeekx, *Beveiligingsrisico's van Java applets*, Rotterdam Institute for Business Economic Studies (RIBES), ISSN 1380-796X, Issue 9850, 1998, ISBN 90-5086-308-6

Bevan, 1999 — Nigel Bevan, *Quality in use: Meeting user needs for quality*, The Journal of Systems and Software, Volume 49, Issue 1, 15 december 1999, pp. 89-96

Bocij et al., 1999 — Paul Bocij, Dave Chaffey, Andrew Greasley, Simon Hickie, *Business Information Systems: Technology, Development and Management*, Pearson Education Limited, Harlow, 1999, ISBN 0-273-63849-1

Boehm, 1984 — Barry W. Boehm, *Verifying and validating software requirements and design specifications*, Software, January 1984, pp. 75-88.

Boehm, 1988 — Barry Boehm, *A Spiral Model of Software Development and Enhancement, IEEE Computer, May 1988*, Volume 21, issue 5, pp. 61-72., ISSN 0018-9162

http://uweb.txstate.edu/~mg43/CS5391/Papers/ProcsReqs/SpiralModel.pdf

downloaddatum: 19-10-2004

Boehm, Port, 1999 — Barry Boehm, Dan Port, *Escaping the software tar pit: model clashes and how to avoid them*, ACM Software Engineering Notes, January, 1999, Volume 24, issue 1, pp. 36-48., ISSN 0163-5948

http://sunset.usc.edu/TechRpts/Papers/usccse98-517/usccse98-517.pdf

downloaddatum: 12-09-2004

Booch et al., 1998 Grady Booch, Robert C, Martin, James Newkirk, *Preliminary chapter of Object Oriented Analysis and Design with Applications, 2nd edition*, Addison Wesley Longman, Inc., 1998

http://www.objectmentor.com/resources/articles/RUPvsXP.pdf

downloaddatum: 17-10-2004

Chapin et al., 2001 Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, Wui-Gee Tan, *Types of software evolution and software maintenance*, Journal of Software Maintenance and Evolution: Research and Practice, Volume 13, Issue 1, pp. 3-30, Februari 27th 2001, ISSN 1532-060X

http://wwwhome.cs.utwente.nl/~strijk/BIT/ADSA/Software%20Evolution/%5B6%5D%20-%20Chapin%202000.pdf

downloaddatum: 26-01-2005

Clements, 1994 Alan Clements, *68000 Family Assembly Language*, PWS Publishing Company, Boston, 1994, ISBN 0-534-93275-4

Conn, 2004 Samuel S. Conn, *A New Teaching Paradigm in Information Systems Education: An Investigation and Report on the Origins, Significance, and Efficacy of the Agile Development Movement*, Information Systems Education Journal, Volume 2, Number 15, February 11, 2004, ISSN 1545-679X

http://isedj.org/2/15/ISEDJ.2(15).Conn.pdf

downloaddatum: 22-10-2004

Crosby, 1979 Phillip B. Crosby, *Quality is Free*, New York, McGraw-Hill, 1979, ASIN 0070145121

Dorans et al., 2000 Neil J. Dorans, P. W. Holland, *Population invariance and the equatability of tests: Basic theory and the linear case*, Journal of Educational Measurement, Volume 37, Number 4, Winter 2000, pp. 281-306

Eliëns, 2000 Anton Eliëns, *Principles of Object-Oriented Software Development*, Addison-Wesley, 2nd edition, ISBN 0-201-39856-7

http://www.cs.vu.nl/~eliens/poosd/1.html

Evans et al., 1999 James R. Evans, William M. Lindsay, *The Management and Control of Quality*, 4th edition, South-Western College Publishing, Cincinnati, 1999, ISBN 0-324-0045-6 / 0-538-88242-5

Feigenbaum, 1983 Armand V. Feigenbaum, *Total Quality Control*, 3rd edition, New York, McGraw-Hill, 1983, ASIN 0070203539

Fenton et al., 1997 Norman E. Fenton, Shari Lawrence Pfleeger, *Software Metrics: A Rigorous and Practical Approach, 2nd Edition*, PWS Publishing, Boston, 1997, ISBN 0-534-95425-1

Florijn, 1996 — Gert Florijn, *OO – een hernieuwde kennismaking? (Objectoriëntatie leidt niet vanzelf tot massaal hergebruik)*, Automatiseringsgids, nummer 16, 1996

http://www.serc.nl/resources/publicaties/artikelen/oo-autgids-96.pdf

downloaddatum: 30-01-2005

Fowler, 2000 — Martin Fowler, *Put Your Process on a Diet*, Software Development Online, http://www.sdmagazine.com/, December 2000

http://www.sdmagazine.com/articles/2000/0012/0012a/0012a.htm

downloaddatum: 22-10-2004

Fowler, april 2003 — Martin Fowler, *The New Methodology*, April 2003

http://www.martinfowler.com/articles/newMethodology.html

downloaddatum: 22-10-2004

Fowler, september 2003 — Martin Fowler, *UML Distilled, Third Edition*, Addison-Wesley, September 2003, ISBN 0-321-19368-7

Fowler et al., 1999 — Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, 1st edition, Addison-Wesley Publishing Company, Reading, Massachusetts, June 28, 1999, ISBN 0-201-48567-2

Gamma et al., 1995 — E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley Professional, January 15, 1995, ISBN 0-201-16336-2

Heineman et al., 2001 — George T. Heineman, William T. Councill, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, May 2001, ISBN 0-201-70485-4

Hendriks, 2000 — Paul Hendriks, *Kwaliteitszorg van softwareontwikkeling*, Informatie, november 2000, pp. 20-27, tenHagen & Stam bv, ISSN 00199907

Highsmith, 2002 — Jim Highsmith, *What Is Agile Software Development?*, CrossTalk, October 2002

http://www.adaptivesd.com/articles/cross_oct02.pdf

downloaddatum: 26-01-2005

Humphrey, 1989 — Watts S. Humphrey, *Managing the Software Process*, Addison-Wesley Professional, Reading, January 1st, 1989, ISBN 0-201-18095-2

Humphrey, 1994a — Watts S. Humphrey, *A Personal Commitment to Software Quality*, American Programmer Journal, December 1994

http://www.sei.cmu.edu/pub/documents/articles/pdf/psp.qual.pdf

downloaddatum: 06-04-2004

Humphrey, 1994b — Watts S. Humphrey, *A Discipline for Software Engineering*, 1st edition, Addison-Wesley Professional, December 1994, ISBN 0-2091-54610-8

Humprey, 1999 — Watts S. Humprey, *Bugs or Defects?*, SEI Interactive, maart 1999

http://www.sei.cmu.edu/news-at-sei/columns/watts_new/1999/March/watts-mar99.pdf

downloaddatum: 03-05-2004

Hyatt, Rosenburg, 1996 — Lawrence E. Hyatt, Linda H. Rosenburg, Ph. D., *A software quality model and metrics for identifying project risks and assessing software quality*, 8th Annual Software Technology Conference, Utah, April 24, 1996

http://satc.gsfc.nasa.gov/support/STC_APR96/qualtiy/stc_qual.pdf

downloaddatum: 22-04-2004

ISO-8402, 1994 — ISO 8402, *Quality management and quality assurance -- Vocabulary*, 1994

Jacobson, 1993 — Ivar Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, 4th printing, 1993, ISBN 0-201-54435-0

Juran, 1995 — Joseph M. Juran, *Managerial Breakthrough, Revised Edition*, McGraw-Hill, New York, July 1, 1995, ISBN 0-07-113404-2

Kan, 1995 — Stephen H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995, ISBN 0-201-63339-6

Kan, 2002 — Stephen H. Kan, *Metrics and Models in Software Quality Engineering, 2nd Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, September 16th, 2002, ISBN 0-201-72915-6

Kaner et al., 2004 — Cem Kaner, Walter P. Bond, *Software Engineering Metrics: What Do They Measure and How Do We Know?*, 10th International Software Metrics Symposium, Metrics 2004

http://www.kaner.com/pdfs/metrics2004.pdf

downloaddatum: 21-11-2004

Kitchenham, 1996 — Barbara A. Kitchenham, *Software Metrics: Measurement for Software Process Improvement*, NCC Blackwell Ltd., Oxford, 1996, ISBN 1-85554-820-8

Kitchenham et al., 1996 — Barbara A. Kitchenham, Shari Lawrence Pleeger, *Software Quality: The elusive target*, IEEE Software, volume 13, issue 1, January 1996, pp. 12-21, ISSN 0740-7459

http://staff.science.uva.nl/~marx/teaching/0405/se/target.pdf

downloaddatum: 21-11-2004

Koldijk, 1999 — Frank Koldijk, *Een exotisch curiosum: Krachtige programmeertaal Eiffel wil niet aanslaan*, Computable, VNU Business Publications, jaargang 1999, nummer 7, 19 februari 1999, p. 37, ISSN 0169-3786

http://www.computable.nl/artikels/archief9/d07ag9qt.htm

downloaddatum: 25-01-2005

Kuhn, 1996 — Thomas S. Kuhn, *The structure of scientific revolutions*, The University of Chicago Press, Chicago, 3rd edition, 1996, ISBN 0-226-45808-3

Larman, Basili, 2003 — Craig Larman, Victor R. Basili, *Iterative and Incremental Development: A Brief History*, IEEE Computer, volume 36, issue 6, june 2003, pp. 47-56, ISSN 0018-9162

http://www2.umassd.edu/SWPI/xp/articles/r6047.pdf

downloaddatum: 17-10-2004

Leventhal, 1981 — Lance A. Leventhal, *6809 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, 1981, ISBN 0-931988-35-7

Lyu, 1996 — Michael R. Lyu (editor), *Software Reliability Engineering*, McGraw-Hill Book Company, New York, April 1st, 1996, ISBN 0-07-039400-8

http://www.cse.cuhk.edu.hk/~lyu/book/reliability/

Mellor, 1992 — Peter Mellor, *Failures, Faults and Changes in Dependability Measurement*, Journal of Information and Software Technology, volume 34, issue 10, pp. 640-654, October 1992, ISSN 0950-5849

http://www.cs.york.ac.uk/hise/safety-critical-archive/1997/0049.html

downloaddatum: 03-05-2004

Mili et al., 2001 — Hafedh Mili, Ali Mili, Sherif Yacoub, Edward Addy, *Reuse based software engineering*, John Wiley & Sons Inc, [place], December 1st, 2001, ISBN 0-471-39819-5

Nørbjerg, 2002 — Jacob Nørbjerg, *Managing incremental development: combining flexibility and control*, Proceedings of the Xth European Conference on Information Systems, Gdansk, Poland, 6 - 8 June, 2002

http://web.cbs.dk/staff/noerbjerg/publications_files/INCDEV.pdf

downloaddatum: 19-10-2004

Poulin, 1996 — Jeffrey S. Poulin, *Measuring Software Reuse : Principles, Practices, and Economic Models*, Addison Wesley, Reading, Massachusetts, November 1996, ISBN 0-201-63413-9

Raccoon, 1997 — L.B.S. Raccoon, *Fifty years of progress in software engineering*, Software Engineering Notes, Volume 22, Issue 1, January 1997, pp. 88-104, ISSN 0163-5948, ACM Press, New York

http://uweb.txstate.edu/~mg43/CS5391/Papers/Introduction/progress50years.pdf

downloaddatum: 07-10-2004

Reeves & Bednar, 1994 — Reeves, Carol A. and Bednar, David A., *Defining Quality: Alternatives and Implications*, Academy of Management Review, volume 19, issue 3, pp. 419-445, july 1994, ISSN 0363-7425

Reilly et al., 1993 — Edwin D. Reilly, Anthony Ralston, *Encyclopedia of Computer Science*, *third edition*, Chapman & Hall, London, 1993

Robinson et al., 2002 — Simon Robinson, Ollie Cornes, Jay Glynn, Burton Harvey, Graig McQueen, Jerod Moemeka, Christian Nagel, Morgan Skinner, Karli Watson, *C# voor Professionals*, Academic Service, Schoonhoven, 2002, ISBN 90-395-1962-5

Royce, 1970 — Winson W. Royce, *Managing the Development of Large Software Systems*, Proceedings, IEEE WESCON, August 1970, pp. 1-9, pp. 328-338

Skidmore, Eva, 2004 — Steve Skidmore & Malcolm Eva, *Introducing Systems Development*, Palgrave MacMillan, Hampshire, 2004, ISBN 0-333-97369-0

Sommerville, 2004 — Ian Sommerville, *Software Engineering*, Pearson Addison-Wesley, Harlow, 7th edition, 2004, ISBN 0-321-21026-3

Stapleton, 1997 — Jennifer Stapleton, *DSDM: de methode in de praktijk*, Academic Service, Schoonhoven, 1999, ISBN 90-395-1091-1

Szyperski, 1997 — Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley Longman Limited, Edinburg Gate, Harlow, Essex, December 19, 1997, ISBN 0-201-17888-5

van Veenendaal et al., 1997 — E.P.W.M. van Veenendaal, J.J.M. Trienekens, *Testen op basis van de kwaliteitsbehoeften van gebruikers*, Informatie, juli/augustus 1997, jaargang 39, ISSN 00199907

http://www.improveqs.nl/pdf/gebruikers.pdf

downloaddatum: 27-01-2005

Veerman et al., 1994 — C.P. Veerman & J.P.J.M Essers, *Wetenschap en wetenschapsleer: een inleiding*, Eburon, Delft, 3e druk, 1994, ISBN 90-5166-019-7

van Vliet, 1988 — Prof. Dr. J.C. van Vliet, *Software Engineering*, Stenfert Kroese, Leiden, tweede, herziene druk, 1988, ISBN 90-207-1646-8

Wolak, 2001 Ronald G. Wolak, *DISS 725 – System Development: Research Paper 1; SDLC on a Diet,* April 2001

http://www.itstudyguide.com/papers/rwDISS725researchpaper1.htm

downloaddatum: 22-10-2004

Figuren

Tabellen

Citaten