Master Thesis

Kwaliteitsverbetering in de Software Fabriek

Radboud Universiteit Nijmegen

Datum:	17 juli 2008
Auteur:	Ing. Ester Sprik
Afstudeernummer:	593
Begeleiders:	Dr. Patrick van Bommel
	Prof. Dr. Ben Dankbaar
	Ir. Jasper Kloost

Voorwoord

Ik heb deze scriptie geschreven tijdens mijn afstudeeronderzoek bij Thinkwise software. Hiermee studeer ik af aan de Radboud Universiteit te Nijmegen. Het afstudeeronderzoek zal op 31 juli 2008 afgesloten worden met een afstudeerpresentatie. Bij deze heet ik u van harte welkom deze presentatie bij te wonen.

Ik wil bij deze graag iedereen bedanken die mij geholpen heeft met het bereiken van dit eindresultaat. Een paar mensen in het bijzonder:

Mijn begeleiders van de universiteit: Patrick van Bommel en Ben Dankbaar. Ondanks dat informatica en management twee totaal verschillende richtingen zijn, hoop ik dat ze dit onderzoek allebei interessant vonden.

Mijn collega's bij Thinkwise en in het bijzonder mijn begeleider Jasper Kloost. Bedankt voor de steun, de tijd en de gezelligheid.

Mijn familie: mijn ouders, Amanda en Lisanne. Zonder hen was het niet gelukt.

Ester Sprik

Samenvatting

Thinkwise software (Thinkwise) ontwikkelt software sneller en kosteneffectief met de behulp van Thinkwise Software Factory (TSF). In dit project is onderzocht hoe de kwaliteit van de elektronische bouwtekening (MSD) verhoogd kan worden, zodat het eindproduct geautomatiseerd gebouwd met de TSF van betere kwaliteit is. Het onderzoek is opgedeeld in verbetering van het kwaliteitsproces en verbeteringen op gebied van het ontwikkelproces. Voor het verbeteren van het kwaliteitsproces is gekeken naar wat kwaliteit inhoud. De geschiedenis van kwaliteit en kwaliteitsbeheer ligt bij massaproductie. Bij massaproductie kan ervoor gekozen worden om elk product te controleren, wat veel tijd en geld kost, of om slechts een deel te controleren, wat minder geld kost, maar wel foutgevoeliger is. Hiertussen moest een goede balans gevonden worden. Software verschilt veel met massaproductie, maar toch zijn er een aantal cruciale overeenkomsten. Zo kan software niet volledig getest worden aangezien dit te lang duurt. Daarom worden er vaak delen getest. De gevolgen van fouten kunnen echter zo duur zijn, dat men het er soms wel voor over heeft om erg lang te testen. Er moet daarom een balans gezocht worden voor de hoeveelheid testen, zodat de kwaliteit van voldoende niveau is.

Voor het verbeteren op gebied van het ontwikkelproces zijn er 4 punten gedefinieerd waar de kwaliteit verbeterd zou kunnen worden. Protocollen, Impact Analyse, Valideren en Testen. Protocollen zijn gedragsvoorschriften, die ervoor zorgen dat acties op een eenduidige manier uitgevoerd worden. In de MSD kunnen protocollen ontwikkeld worden op verschillende niveaus. Deze protocollen zorgen ervoor dat incorrecte data niet opgeslagen mogen worden. Protocollen kunnen ook gedefinieerd worden op het gebied van code, door een functie te maken die automatisch de opmaak aanpast aan de standaard. Deze standaard kan per project verschillen, maar is vervolgens wel over het hele project gelijk.

Impact analyse is een manier om tussen versie's te analyseren welke delen code aangepast moeten worden. Door na een wijziging aan te geven waar de code mogelijk niet meer correct is, wordt de software beter te onderhouden. De functionaliteit wordt ook beter omdat met behulp van een tool de plekken aangewezen worden, waar de functionaliteit niet meer correct is. Deze kan vervolgens aangepast worden. Voor de impact analyse is een ontwerp en een prototype gemaakt, zodat dit ingezet kan gaan worden bij Thinkwise.

De MSD kan, voordat het eindproduct gebouwd wordt, gevalideerd worden. Door het ontwikkelen van een testproject, zijn een aantal situaties ontdekt, die niet zouden mogen voorkomen en daarom tijdens het valideren afgevangen moeten worden. Tijdens het onderzoek naar de impact analyse is gebleken dat de MSD ook op een andere manier gevalideerd kan worden. Hierbij wordt niet gekeken naar situaties in de bouwtekening, maar verschillen tussen de code en de bouwtekening.

Als laatste is aandacht besteed aan het testen. Dit kan opgedeeld worden in testen van de GUI's en testen van de business rules. Voor het testen van de GUI's is een testproject gemaakt. In dit SF-project moeten de meest uiteenlopende situaties zitten zodat de GUI's getest kunnen worden. Voor het testen van business rules moeten de broncode (zoals triggers, stored procedures, etc.) getest worden. Uiteindelijk zouden testcases in templates opgeslagen moeten worden, zodat deze net als de broncode gegenereerd kunnen worden.

Inhoud

1 Inleiding

Dit project is uitgevoerd ter afronding van de Master studie aan de Radboud Universiteit te Nijmegen. Tijdens de studie informatica is gekozen om met de variant *'Management en Toepassing'* te specialiseren op het vakgebied *'Software Constructie'*. Om deze Master Thesis te voltooien is naar een opdracht bij een bedrijf gezocht. Deze opdracht is gevonden bij Thinkwise Software. Het project is begonnen in februari 2008 en is geëindigd in juli 2008.

Thinkwise software (Thinkwise) is begonnen met het ontwikkelen van bedrijfskritische database applicaties. Later is de nadruk gelegd op het ontwikkelen van software (de Thinkwise software fabriek) waarmee sneller software gebouwd kan worden. Door gebruik van de software factory, worden deze informatiesystemen sneller en kosteneffectief gebouwd. De software factory is uitgegroeid tot een ontwikkelomgeving waarmee met een factor 5 tot 10 sneller ontwikkeld kan worden. Tegenwoordig richt Thinkwise zich vooral op het ondersteunen van klanten die met de software factory zelf ontwikkelen.

Het is ondertussen bewezen dat met de software factory van Thinkwise sneller ontwikkeld kan worden, daarom wordt nu de focus gelegd op hogere kwaliteit van de software. Doordat de software gegenereerd wordt, is er minder kans op menselijke fouten. Toch moet er nog steeds een deel door mensen ontwikkeld worden. Doordat er door mensen ontwikkeld wordt, kunnen er alsnog fouten ontstaan. Deze zijn wel minder dan met traditioneel bouwen. Hoe kunnen deze fouten, wanneer ze eenmaal gemaakt zijn, eruit gehaald worden? Hoe kan voorkomen worden dat fouten gemaakt worden? Hoe kan duidelijk gemaakt worden dat de kwaliteit inderdaad hoger is dan gemiddeld?

Aan de hand van deze vragen is dit onderzoek naar kwaliteit uitgevoerd. In dit document worden de opzet, uitvoering en resultaten van het onderzoek weergegeven. Hiervoor wordt eerst de probleemstelling en werkwijze weergegeven. Om onderzoek naar kwaliteit in de software factory te kunnen begrijpen, moet eerst duidelijk zijn wat een software factory is. Daarna wordt aangegeven (in hoofdstuk 3) wat de software factory van Thinkwise zo speciaal maakt. Vervolgens wordt gekeken wat onder kwaliteit verstaan wordt (hoofdstuk 4), daarna kan pas gekeken worden hoe de kwaliteit kan verbeteren.

In de daaropvolgende hoofdstukken wordt weergegeven op welke manieren de kwaliteit verhoogd kan worden en hoe dit bij Thinkwise toepast kan worden. Uiteindelijk wordt afgesloten met een conclusie en aanbevelingen.

2 Probleemstelling en werkwijze

Greenfield en Short [1] geven aan dat in de Verenigde Staten meer dan $250 miljard besteed wordt aan software ontwikkeling. Van de projecten zijn slechts 16% op tijd en binnen budget. Daarnaast wordt 31% van de projecten gestopt, hoofdzakelijk vanwege kwaliteitsproblemen. De kwaliteit van software kan dus nog flink verbeterd worden. Kwaliteit is echter een ruim begrip, wat de ene goede kwaliteit vindt, is voor de andere niet goed genoeg. Hieronder is weergegeven wat de probleemstelling is en welke werkwijze gebruikt wordt om de kwaliteit van de software van Thinkwise te verbeteren.

2.1 Probleemstelling

Met de software van Thinkwise wordt een elektronische bouwtekening ontwikkeld. Vanuit deze elektronische bouwtekening wordt het eindproduct automatisch gebouwd. Doordat het product geautomatiseerd gebouwd wordt, zullen er minder fouten ontstaan dan wanneer dit handmatig gedaan wordt.

Omdat nog steeds een deel van de software door mensen ontwikkeld wordt, kunnen er in de bouwtekening nog fouten zitten. Hierdoor zitten er ook in het eindproduct nog steeds fouten (bugs). De bugs kunnen met de techniek van Thinkwise eerder in het ontwikkelproces opgespoord worden, zelfs al voordat het daadwerkelijke eindproduct gemaakt is. Dit is vergelijkbaar met de auto-industrie. Daar wordt een crashtest eerst in het ontwerp doorgerekend en wanneer een gebouwde auto deze test daadwerkelijk ondergaat, worden er bijna geen fouten meer gevonden.

Bij het testen van het eindproduct ligt daarom de nadruk op de invulling van de elektronische bouwtekening. Wanneer de kwaliteit van de elektronische bouwtekening hoger wordt, dan wordt het eindproduct ook van betere kwaliteit. De hoofdvraag die tijdens dit onderzoek gesteld wordt is:

Hoe kan de kwaliteit van de elektronische bouwtekening verhoogd worden, zodat het eindproduct, geautomatiseerd gebouwd met Thinkwise technologie, van betere kwaliteit is?

Omdat dit een abstracte hoofdvraag is, zijn er een aantal deelvragen geformuleerd die aangeven op welke onderdelen specifiek gericht zal worden. Dit resulteert in de volgende deelvragen:

1. *Hoe kunnen protocollen ervoor zorgen dat tijdens de invoer van de elektronische bouwtekening slechte kwaliteit vermeden wordt?*
Het gaat hierbij om controles die ervoor zorgen dat fouten niet in de bouwtekening komen.
2. *Hoe kan de bestaande validatie uitgebreid worden, zodat slechte kwaliteit eerder in het proces zichtbaar wordt?*
Traditioneel worden fouten pas gevonden tijdens het testen van het eindproduct. Als er een ontwerp is, dan is een review meestal de enige methode om dit ontwerp te valideren. Doordat de elektronische bouwtekening in een database opgeslagen is, kan deze met automatische controles gevalideerd worden. Deze functionaliteit is al voor een deel aanwezig in de Thinkwise technologie.
3. *Hoe kan een change impact analyse gebruikt worden om ervoor te zorgen dat aanpassingen in de elektronische bouwtekening en het eindproduct kwalitatief beter zijn?*

Soms zijn er wijzigingen die ervoor zorgen dat andere delen niet meer werken. Door alle effecten van deze wijzigingen in kaart te brengen kunnen deze eerder geïdentificeerd en aangepast worden. Dit zorgt ervoor dat onderhoud (zelfs nog voordat het product in gebruik is genomen) tijdens de overgang tussen versies makkelijker wordt en dat het eindproduct na de aanpassingen kwalitatief beter is.

4. *Hoe kan het eindproduct, dat gebouwd wordt met Thinkwise technologie, getest worden?*

Bij Thinkwise wordt alleen nog maar handmatig getest. Door dit te automatiseren kan het testen sneller en gemakkelijker uitgevoerd worden.

5. *Hoe kan de kwaliteit van een eindproduct, dat gebouwd wordt met Thinkwise technologie, gemanaged worden?*

De nadruk ligt hierbij op de betekenis van kwaliteit in de software ontwikkeling en verbetering van het kwaliteitsproces.

2.2 Werkwijze

Er zijn twee manieren waarop gekeken kan worden of software van betere kwaliteit is. Ten eerste kan gemeten worden wat de kwaliteit van een bepaalde product is, vervolgens kan een techniek geïntroduceerd worden en naar verloop van tijd kan de kwaliteit opnieuw gemeten worden om te controleren of deze verbeterd is. Door dit met verschillende technieken uit te voeren, wordt bepaald welke techniek het beste gebruikt kan worden om de kwaliteit te verbeteren. Deze methode heeft echter als nadeel dat er een manier gevonden moet worden om kwaliteit te meten. Daarnaast moet een geschikte casus gevonden worden die uitgevoerd kan worden. Als deze casus uitgevoerd wordt, dan moet vervolgens een lange periode gewacht worden voordat het resultaat geanalyseerd kan worden. Daarna zal dezelfde casus nog een keer met een andere techniek uitgevoerd moeten worden of zal een andere casus uitgevoerd moeten worden. Het twee keer uitvoeren van dezelfde casus zal voor geen enkel bedrijf rendabel zijn en zal daarom niet snel uitgevoerd worden. Het uitvoeren van het experiment met twee verschillende casussen zorgt ervoor dat de situaties anders zijn en de resultaten daarom minder betrouwbaar zijn.

De tweede manier om kwaliteit te verbeteren, is door te kijken naar kenmerken van slechte kwaliteit. Als deze kenmerken uit de software gehaald worden, betekent dit dat de kwaliteit beter wordt, tenzij de aanpassingen de kwaliteit weer verslechteren. In dit onderzoek is ervoor gekozen de tweede manier te hanteren. Er zal echter alleen gericht worden op het herkennen van slechte kwaliteit in het ontwikkelproces. Als deze punten geïdentificeerd zijn, is de ontwikkelaar die het gebruikt er zelf verantwoordelijk voor om te zorgen dat deze punten op een goede manier aangepast worden.

Dit onderzoek is daarom van explorerende aard. Het is niet de bedoeling dat de beste techniek gekozen wordt uit een lijst met alle mogelijke technieken, om de kwaliteit te verbeteren. Het is juist de bedoeling de verschillende technieken, die geschikt zijn om op de software van Thinkwise toe te passen, weer te geven en aan te geven hoe deze toegepast kunnen worden.

Het onderzoek kan opgedeeld worden in twee onderdelen:

1. Verbetering van het kwaliteitsproces. Het proces dat ingezet wordt om structureel kwaliteit te verbeteren.
2. Verbeteringen op het gebied van het ontwikkelproces. Dit zijn technieken die door ontwikkelaars of testers gebruikt kunnen worden om de elektronische bouwtekening te

verbeteren. Als de elektronische bouwtekening verbeterd, dan wordt daardoor ook het eindproduct beter.

2.2.1 Verbetering van het kwaliteitsproces

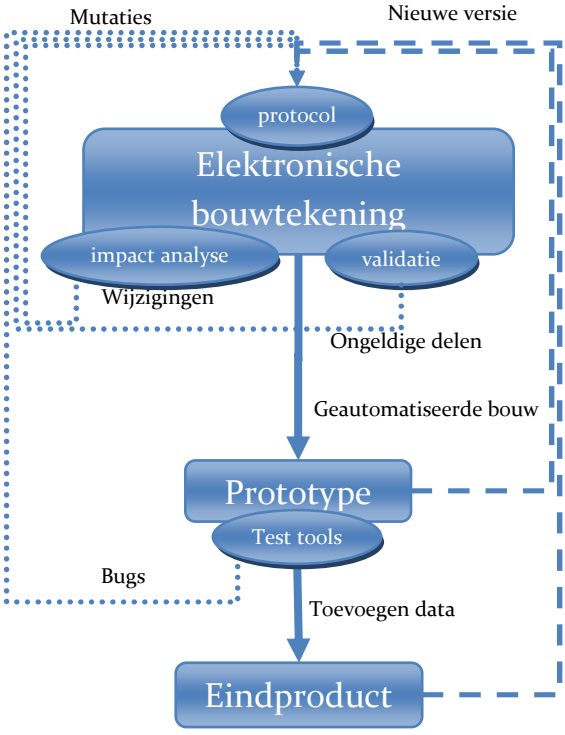
Om eerst het niveau van de kwaliteit op een constant niveau te houden en later te verbeteren, moet een kwaliteitsproces ontwikkeld zijn. Dit kan een heel simpel proces zijn, bijvoorbeeld door af te spreken dat een ontwikkelaar zijn eigen werk test. Het kan ook een heel complex proces zijn, zoals bij een raket van miljarden nodig is. Daarom zal uitgezocht worden welke modellen er zijn die het kwaliteitproces (van software) beschrijven en hoe deze modellen toegepast kunnen worden bij Thinkwise. Hiervoor wordt eerst gekeken naar de oorsprong van kwaliteit in het algemeen. Vervolgens kan gekeken worden wat er anders is aan kwaliteit van software ontwikkeling.

2.2.2 Kwaliteitsverbetering in het ontwikkelproces

Er zijn verschillende manieren waarop de kwaliteit verbeterd kan worden. Daarom is in dit onderzoek gestart met het maken van een architectuur waaraan de mogelijke technieken opgehangen kunnen worden. Aan de hand van deze architectuur zijn ook de deelvragen geformuleerd. De architectuur laat zien op welke plaatsen in het proces verbetering van de kwaliteit toegepast kan worden.

Om de kwaliteit te verbeteren, moeten er zoveel mogelijk fouten opgelost of voorkomen worden. In het algemeen is het zo dat hoe langer een fout in een product zit, hoe duurder het wordt om deze fout weer op te lossen. Een fout die in de analyse al gevonden wordt, kost daarom minder tijd en geld om te corrigeren, dan een fout die tijdens de testfase gevonden wordt.

Het doel is daarom om fouten zo snel mogelijk op te sporen, of zelfs te voorkomen dat ze ontstaan. Er zijn vier punten waarop fouten ontdekt kunnen worden, deze staan weergegeven in de architectuur in Figuur 1. Dit zijn: protocollen, impact analyse, validatie en test tools. Bij elk punt krijgt de ontwikkelaar feedback, zodat fouten eerder opgemerkt worden.



Figuur 1: Architectuur

Protocol

Het eerste onderdeel heeft te maken met het voorkomen van fouten. Tijdens het muteren, het toevoegen, wijzigen of verwijderen van gegevens, kunnen er controles gemaakt worden die ervoor zorgen dat de data correct blijft. Een voorbeeld hiervan is dat elke tabelnaam met een hoofdletter moet beginnen of dat een kolomnaam geen spaties mag bevatten. Wanneer een tabel gewijzigd wordt van 'Persoonsgegevens' naar 'Persoon gegevens' zal er feedback aan de gebruiker gegeven worden en zal de wijziging niet opgeslagen worden.

Validatie

Validatie kan op elk moment uitgevoerd worden, maar moet minimaal één keer, voordat een geautomatiseerde bouw plaats vindt, uit zijn gevoerd. Tijdens de validatie worden in de elektronische bouwtekening situaties opgespoord, die in het verleden tot fouten geleid hebben en die met hoge waarschijnlijkheid weer fouten geven. Een voorbeeld hiervan is, als er een tabel is zonder primaire sleutel, of zelfs een tabel zonder kolommen. Bij invoeren moet het mogelijk zijn om tabellen in te voeren, terwijl er nog geen kolommen zijn, maar in het eindproduct is dit incorrect. Door de elektronische bouwtekening te valideren, voordat het product gemaakt wordt, krijgt een ontwikkelaar sneller feedback en kunnen mogelijke fouten sneller opgespoord en gecorrigeerd worden.

Impact analyse

Dankzij de gebruikte technieken kan een product gemaakt worden, terwijl de bouwtekening nog niet helemaal compleet is. De gebruiker krijgt op deze manier een beter beeld van het deel dat al wel ontwikkeld is en kan in een vroeg stadium feedback geven. Als een product gebouwd is, kan een nieuwe versie van de elektronische bouwtekening gemaakt worden waarin alle gegevens gemuteerd kunnen worden en er nieuwe informatie toegevoegd kan worden. Hierop kan een analyse gedaan worden om te kijken of eerder ontwikkelde stukken nog steeds correct zijn. De onderdelen die niet correct zijn, kunnen dan aangepast worden. Een voorbeeld hiervan is een tabel, waarvan het aantal kolommen wijzigt. Op de tabel kan al business functionaliteit en taken zitten. Als er een taak bestaat waarmee de tabel gevuld wordt, zal deze taak ook aangepast moeten worden aan het aantal kolommen. In de huidige situatie komen dit soort fouten pas bij de geautomatiseerde bouw naar boven en moeten dan nog opgelost worden. Door dit in een impact analyse te doen, die op elk moment in het proces uitgevoerd kan worden, worden de fouten eerder opgespoord en zijn de kosten van het onderhoud en de kosten van de fout minder.

Test tools

Op het eindproduct kan op de traditionele manier getest worden. Hierbij moet gekeken worden hoe het testproces gemanaged kan worden. Door de manier van bouwen zal de focus van het testen verschuiven. De nadruk van het testen ligt vooral op de business rules, waarin meer handmatige onderdelen zitten. Het ligt minder op de Grafische User Interface (GUI) en het traditionele datamodel die gebruik maken van de elektronische bouwtekening. Op dit moment wordt nog handmatig getest, maar met behulp van test tools moet het mogelijk zijn dit te automatiseren, zodat ook regressietesten makkelijker uitgevoerd kunnen worden.

3 Software Fabriek

Thinkwise helpt organisaties met het maken van software door middel van de software fabriek (SF). *Software fabriek* of *software factory* is een begrip dat al lang bestaat, maar de laatste jaren weer nieuw leven is ingeblazen.

3.1 Definitie software fabriek

De term *software factory* is als eerst voorgesteld door R.W. Bremer van General Electric (GE) in de jaren '60 [1] [2]. Bremer definieert een sofware factory als volgt:

"A software factory should be a programming environment residing upon and controlled by a computer. Program construction, checkout, and usage should be done entirely within this environment and by using the tools contained in the environment … a factory … has measures and controls for productivity and quality. Financial records are kept for costing and scheduling. Thus, management is able to estimate from previous data … Among the tools to be available in the environment should be compilers for machine-independent languages; simulators, instrumentation, devices, and test cases as accumulated; documentation tools – automatic flow-charters, text editors, [and] indexers; accounting function devices; linkage and interface verifiers; [and] code filters (and many others)" [2]

Tegenwoordig zijn er veel ontwikkelomgevingen die door een computer aangestuurd worden en de beschikking hebben over tools om het ontwikkelen makkelijker te maken. Compilers zijn geschikt voor de meest gangbare machinetalen, debuggers worden gebruikt om de code te simuleren en test cases en documentatie tools zijn beschikbaar. Toch missen er nog een paar dingen die wel geschreven zijn in de definitie van Bremer, zoals financiële en planningsgegevens. Ontwikkelaars moeten nog veel handmatig programmeren, ondanks dat in veel ontwikkelomgevingen wizards aanwezig zijn om een vliegende start te maken in software ontwikkeling.

Een andere definitie die veel gebruikt wordt, is ontwikkeld door M.D. McIlroy bij AT&T: "De systematische herbruikbaarheid van code voor het construeren van nieuwe programma's". In midden jaren '70 en de jaren '80 werd het begrip software factory vooral in Japan veel toegepast, waarbij de nadruk op *"kojo"*, de fabriek gelegd werd. Volgens Cusumano [2] werd in die tijd hetzelfde concept in Amerika ook gebruikt, maar daar werd het niet onder de term "software factory" gecategoriseerd.

Sinds een paar jaar is de term software factory weer opnieuw geïntroduceerd. Microsoft zet softwarefabrieken in tegen IBM, die met Rational Unified Process (RUP) probeert software ontwikkeling sneller en simpeler te maken [3]. De definitie die Microsoft gebruikt is:

"*A Software Factory is a Software Product Line that configures extensible tools, processes, and content using a software factory template based in a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework based components.*" [4]

De term software factory wordt ook wel gebruikt bij outsourcing. Bij outsourcing wordt het ontwikkelen van software code uitbesteed aan andere bedrijven, meestal ook in andere landen. Vervolgens wordt de fabriek, waarin men de software ontwikkelt, een software fabriek

genoemd. Deze definitie zal hier niet gebruikt worden, de definitie die in dit onderzoek gebruikt wordt is:

"Een software fabriek (SF) is software waarmee vanuit een elektronische bouwtekening of model de ontwikkeling en het onderhoud van software geautomatiseerd wordt."

Hoe dit door Thinkwise uitgevoerd wordt, is uitgebreider beschreven in hoofdstuk 3.3 over de software fabriek van Thinkwise.

3.2 Software factories van Microsoft

De kosten van software zijn erg hoog. Dit geldt niet alleen voor het ontwikkelen, maar ook voor het onderhouden van software, zodat deze aan de veranderende eisen van gebruikers blijft voldoen. Volgens Mannaert, Verelst en Ven [5] wordt met software fabrieken geprobeerd om dit probleem te tackelen. Mannaert et al geven geen eigen definitie, maar kijken vooral naar hoe Microsoft software factories onderbrengt. Volgens de auteurs moet er echter eerst nog werk gedaan worden op gebied van software structuur en software verbetering, voordat er daadwerkelijk een generieke software fabriek gemaakt kan worden, die applicaties assembleert. Volgens de schrijvers is standaardisatie ervoor verantwoordelijk dat het nu mogelijk is om op een ander niveau software te evalueren. Niet meer vanaf het code niveau, maar vanaf het architectuur niveau.

Er zijn ook artikelen die wel positief zijn over software fabrieken. Het concept van Microsoft wordt door Greenfield en Short [6] beschreven. Het doel van de software factories van Microsoft is om een persoongeoriënteerde aanpak te gebruiken die ervoor zorgt dat de gebruikte termen en woorden meer liggen bij het probleemdomein en dat de mechanische en deterministische aspecten overgelaten worden aan ontwikkeltools. Dit zorgt ervoor dat projecten beter voldoen aan de wensen van gebruikers.

Volgens Greenfield en Short [6] maakt men vaak de vergelijking met andere industrieën die eerst gebruik maakten van handwerk en later overstapten op geïndustrialiseerde technieken. Deze industrieën konden hierdoor preciezer en goedkoper producten produceren. Dit probeert men met software factories ook te bereiken, omdat er in die industrie veel geld omgaat en weinig kwaliteit opgeleverd wordt. Zoals al bij de probleemstelling weergegeven is, geven Greenfield en Short aan dat in de Verenigde Staten meer dan $250 miljard besteed wordt aan software ontwikkeling. Hiervan zijn slechts 16% van de projecten op tijd en binnen budget. Daarnaast wordt 31% van de projecten gestopt, hoofdzakelijk dankzij kwaliteitsproblemen.

De analogieën die gemaakt worden tussen software en geautomatiseerde industrieën zijn niet altijd juist. Greenfield en Short geven aan dat dit komt, omdat deze analogieën gemaakt worden tussen software en andere industrieën die fysieke producten ontwikkelen. Volgens het artikel zijn dit appels en peren vergelijkingen, omdat het type markt verschilt. Het gaat in de software om unieke applicaties, die worden vergeleken met fysieke producten, die massaproducten zijn. Greenfield en Short maken onderscheid tussen *product line* en *one-off* implementaties. Product Line Development houdt in het ontwikkelen van een set van gelijkwaardige, maar toch verschillende producten. One-off development betekent juist het ontwikkelen van geheel nieuwe applicaties. Dit wordt in het artikel van Demir [4] in de

conclusie tevens als nadeel aangestipt, voor One-Off projecten is de software factories techniek niet geschikt, omdat het teveel tijd, budget en resources kost.

Een andere manier om naar de markt te kijken is door onderscheid te maken tussen economy of scale en economy of scope. Beide termen worden gebruikt voor de realisatie van kostenverlaging bij het produceren van meerdere producten tegelijk, maar bij economy of scale wordt de productie van meerdere implementaties op een enkel ontwerp bedoeld (met bijvoorbeeld verschillende maten en verschillende kleuren), terwijl economy of scope staat voor productie van meerdere ontwerpen en hun bijbehorende implementatie (bijvoorbeeld bedrijven die automotoren maken en daarnaast ook grasmaaiers gaan ontwikkelen). De software factories techniek richt zich vooral op economy of scope om voor groepen van producten de functionaliteit te maken. Greenfield en Short beschrijven hun visie op een toekomst waarin software factories volledig zijn toegepast. Hierin wordt dan met gebruikers onderhandeld over software componenten in het systeem, zoals nu met klanten onderhandeld wordt over de keuken in een nieuw huis. De applicatie ontwikkelaars zullen van een applicatie nog maar 30% hoeven te bouwen, de overige 70% wordt door kant en klare componenten geleverd.

Om het probleem van slechte kwaliteit en van te dure software op te lossen, zijn er door de jaren heen verschillende technieken bedacht. De verschillende auteurs zijn het er niet helemaal over eens wat de positie van de software fabriek is. Demir [4] maakt in zijn artikel een vergelijking tussen Model Driven Architectuur van de Object Management Group (OMG) en de software factories, zoals deze door Microsoft gebruikt worden. Deze technieken behoren beide tot Model Driven Development (MDD). MDD is een software engineering methodologie die zich richt op modellen, automatisering en code generatie. Door Greenfield en Short worden de software factories van Microsoft juist gepositioneerd als de nieuwe ontwikkeling voortbouwend op de object georiënteerde technologieën: Model-Based Development en Component-Based Development. Omdat Greenfield en Short vanuit het standpunt van Microsoft schrijven, zetten ze software factories neer als een op zichzelf staande techniek, die voort bouwt op andere technieken. Het zal door de jaren heen duidelijk moeten worden of het slechts een hype was, die onderdeel is van bestaande technieken of dat het een nieuwe ontwikkeling is die voort bouwt op bestaande technieken en die een basis is voor nieuwe technieken.

Volgens Greenfield en Short onderscheidt het software fabriek concept zich van andere technieken op drie punten: *Abstraction, Granularity* en *Specificity*. Abstractie verbergt eigenschappen van een onderwerp die niet relevant zijn voor sommige doeleinden, terwijl andere eigenschappen benadrukt worden. Door gebruik van software factories is het mogelijk meerdere modellen te maken op meerdere abstractieniveaus tussen executabels en requirements (eisen). Granularity is de maatstaf voor de grote van de software concepten rekening houdend met de abstractie die gebruikt is, die kan oplopen van regels code tot web services. De laatste as is specificiteit en dit definieert de scope van de abstractie. Wanneer een specifiekere abstractie gebruikt wordt, dan kan dit voor minder producten gebruikt worden, maar het draagt wel meer bij aan de ontwikkeling. Wanneer een abstractie minder specifiek wordt, kan het voor meer producten gebruikt worden, maar draagt het minder bij aan de ontwikkeling van die producten.

Met de software factories van Microsoft is het mogelijk meerdere modellen te maken om de abstracties tussen verschillende modellen te overbruggen. Volgens Demir is het maken van modellen voor het specificeren van software systemen niet nieuw, maar deze modellen worden enkel gebruikt voor het documenteren en begrijpen van het systeem. De software factories maken gebruik van Domain Specific Modeling (DSM). Een concept dat gebruikt wordt voor het ontwikkelen van modellen die berekenbaar (computable) zijn. DSM maakt gebruik van Domain Specific Language (DSL). Een taal om in te modelleren, specifiek voor die toepassing of dat domein.

De modellen die met DSL geschreven zijn, kunnen broncode uit een model genereren met behulp van tools. Om de relaties tussen de verschillende modellen aan te geven, wordt een tabel gemaakt waarbij de kolommen verschillende *interesses* (data, activiteiten, locaties, mensen, etc.) voorstellen, terwijl de regels niveaus van abstractie zijn. Elke cel uit deze tabel wordt een *viewpoint* genoemd. Deze tabel met viewpoints kan gegeneraliseerd worden in een graph. Deze graph wordt een software schema genoemd, omdat het de set van specificaties die ontwikkeld moet worden, beschrijft. Het software schema, de processen voor het vastleggen en gebruiken van informatie in de graph en de tools die gebruikt worden om dat te automatiseren, vormen samen een software template. Deze software template kan ingelezen worden in een Interactive Development Environment (IDE) om een specifiek type van het product te produceren. Een volledig ingerichte IDE wordt vervolgens een software factory genoemd.

Demir vergelijkt deze Software Factories met Model Driven Architectuur, gebruik makend van een case studie. Een groot nadeel van Software Factories is dat de kosten van het eerste product dat ontwikkeld wordt veel hoger liggen, omdat er eerst een schema en een code generator gemaakt moeten worden. Dit is volgens de auteur tijdrovend, verfijnd en eist veel expertkennis over het probleem en het oplossingsdomein. Wanneer een deel ontwikkeld is, kunnen belanghebbenden wel meehelpen in het specificatieproces om misinterpretatie en verwarring te voorkomen. Een voordeel dat beide technieken hebben, is dat de kwaliteit en betrouwbaarheid van de software verbetert. Volgens Demir ontstaan er minder fouten in de eindcode, omdat deze gegenereerd is volgens een schema, regels of code-templates. De code voldoet daarom precies aan de specificatie, zoals deze in de modellen beschreven is.

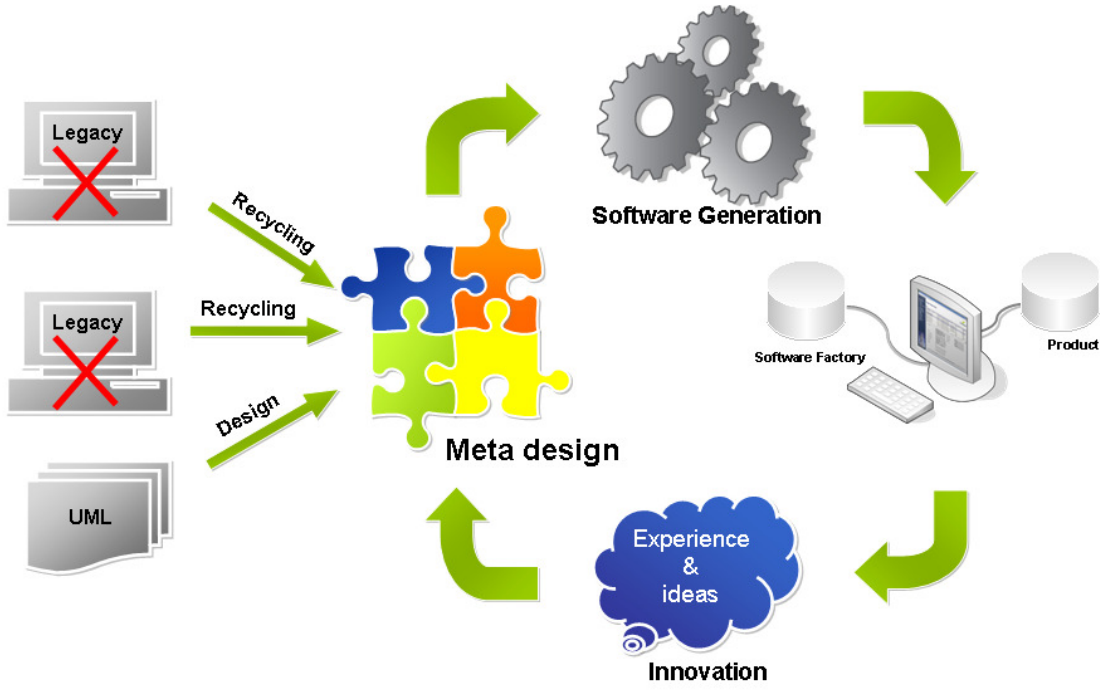
De technieken van Microsoft zijn niet enkel concepten, maar worden ook daadwerkelijk gebruikt en in artikelen beschreven. Bijvoorbeeld door Parigot [7], hij heeft deze techniek gebruikt om een software fabriek genaamd SmartTools in DSL te ontwikkelen. Met SmartTools is het mogelijk om tools te maken voor andere talen, zoals SVG, DTD, XML schema, CSS, WSDL en BPEL. Het framework gebruikt ongeveer 100 000 regels Java code en genereert hiervan 1.000.000 regels eindcode.

3.3 Thinkwise Software Factory

Thinkwise maakt gebruik van het software fabriek concept wat lijkt op het concept, zoals deze gebruikt wordt bij Microsoft. Er is echter een aantal belangrijke verschillen die de Thinkwise Software Fabriek (SF) van de Microsoft software fabrieken onderscheidt.

3.3.1 Algemeen concept

Tegenwoordig maakt bijna elke organisatie gebruik van software in een of andere vorm. Omdat deze software verouderd is of niet meer voldoet, wordt soms besloten om de software te vervangen.



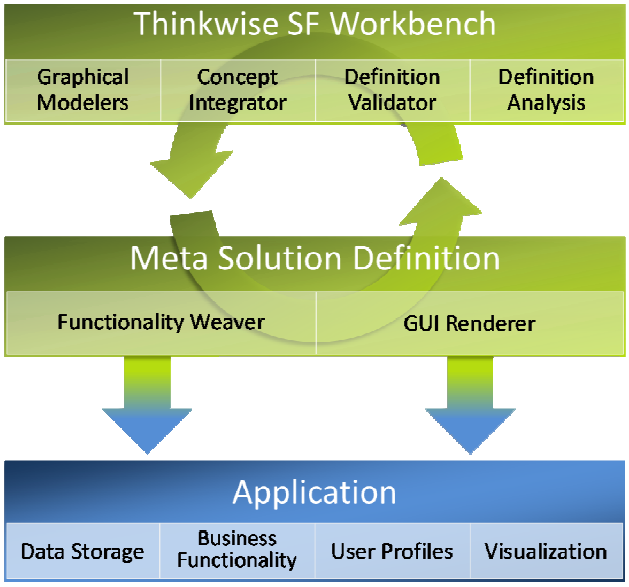
Figuur 2: Het Software Factory concept, bron: Brochure Thinkwise Software Factory

Het bouwproces van de software met de SF verloopt in cycli. Elke cyclus heeft in het ontwerp (Meta design) een eigen versie. Oude systemen (legacy) kunnen gebruikt worden om een snelle start te maken in het meta design. Wanneer de software in gebruik genomen wordt, zal deze legacy software weggegooid worden. De filosofie hierachter is om te houden wat goed is, en te verbeteren waar nodig. Doordat de nieuwe applicatie de goede elementen van de oude software bevat, wordt hij door gebruikers beter geaccepteerd. Naast het recyclen van legacy-systemen is het ook mogelijk om het ontwerp automatisch over te nemen van een ontwerp tool, zoals UML [(8)]. Wanneer het meta design voldoende gevuld is, kan een product gegenereerd worden. Met behulp van dit prototype van het product kunnen gebruikers feedback geven. Deze feedback wordt vervolgens verwerkt in het meta design in een nieuwe cyclus van het bouwproces.

3.3.2 Architectuur

De Meta Solution Definition (MSD), ook wel de elektronische bouwtekening genoemd, is een metamodel waarin de metagegevens opgeslagen worden. In de MSD worden gegevens opgeslagen over onder andere het datamodel, de gebruikersinterface en de business functionaliteit.

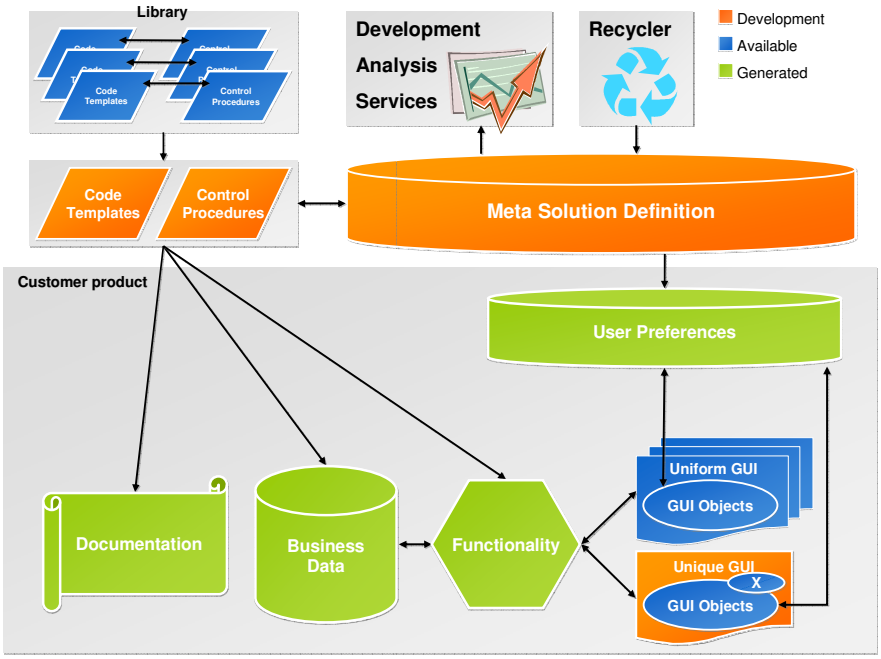
De SF Workbench bevat tools die gebruikt worden om de MSD te vullen, te controleren en te onderhouden. Vanuit een MSD kan met behulp van de Functionality Weaver en de GUI Renderer een applicatie gemaakt worden. Niet alle tools zullen in dit document behandeld worden. De belangrijkste concepten die gebruikt worden zijn Meta definition, GUI objecten en code templates. Deze concepten zorgen voor een grote voorsprong in de software bouw.



Figuur 3: Architectuur SF Workbench, Bron: Website Thinkwise Software

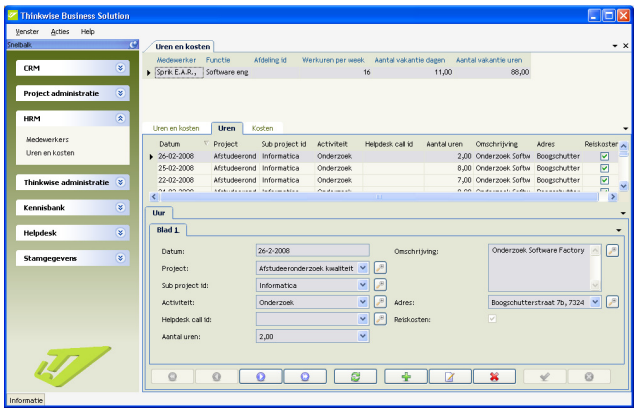
De MSD wordt opgeslagen in een database die de *SF database* genoemd wordt. Met behulp van tools is het mogelijk het model vanuit meerdere kanten te bekijken. In Figuur 4 wordt de architectuur van de Thinkwise software factory weergegeven. De MSD kan gevuld worden door recycling van bijvoorbeeld oude applicaties. Daarnaast kan het ook (aan) gevuld worden door de ontwikkelaars. Uit het MSD kunnen gegevens gehaald worden voor de analyse van de ontwikkeling van de software.

In het MSD staan gegevens opgeslagen over het datamodel. Deze gegevens vormen samen de inhoud van de database en vanuit het datamodel kan dus een database gegenereerd worden. Om de data te manipuleren (gegevens toe te voegen, wijzigen of verwijderen) is een GUI ontwikkeld. Deze Grafische User Interface (GUI) maakt gebruik van een subset van de MSD, deze subset wordt de User Preferences genoemd. De GUI heeft toegang tot de User Preferences en gebruikt de gegevens hieruit om te bepalen wat de opbouw is. Er wordt onderscheid gemaakt tussen uniforme en unieke schermen. Beide schermen kunnen door dezelfde GUI worden uitgelezen. De standaard schermen hebben altijd dezelfde structuur, doordat ze de user preferences uitlezen hoe de schermen eruit moeten zien. De unieke schermen kunnen speciale acties uitvoeren of gegevens op een unieke manier laten zien. Bij het bouwen van



Figuur 4: Architectuur Thinkwise Software Factory, Bron: Brochure Thinkwise Software Factory

unieke schermen kan gebruik gemaakt worden van dezelfde componenten als bij de uniforme schermen. Deze componenten kunnen bijvoorbeeld gegevens uit de database lezen en formulieren hiervan op het scherm zetten,.



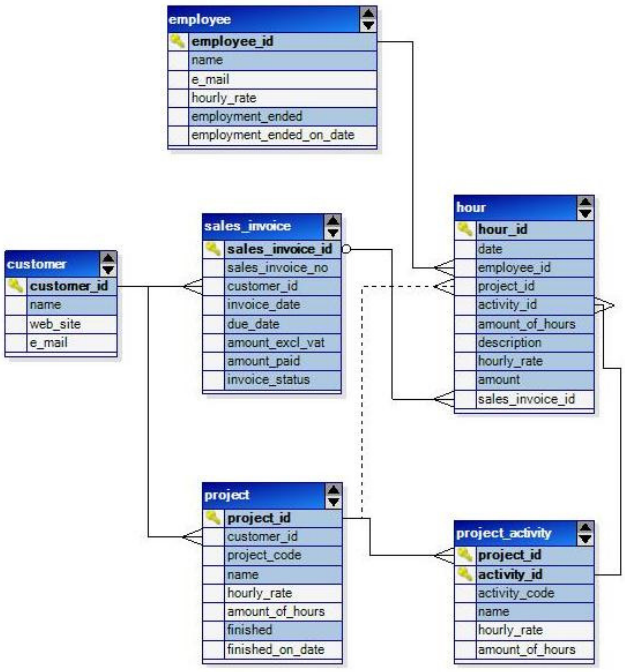
Figuur 5: Uniforme GUI met standaard scherm

Een standaard scherm is te openen via de menubalk aan de linker kant. Een standaard scherm bestaat altijd uit een vaste structuur. Er zijn twee soorten structuren, master/detail en hiërarchie. Bij een master/detail scherm wordt bovenin een lijst getoond (de master) met daaronder de details. De details zijn tabbladen te beginnen met een formulier en daarna de children van de master. Het tabblad van een child is weer een master/detail scherm op zichzelf. Zo is in Figuur 5 het scherm van medewerkers weergegeven. De details geven het tweede tabblad weer waarop de uren van deze medewerker weergegeven worden. Het volgende tabblad geeft de kosten aan die de medewerker gedeclareerd heeft. Het uren scherm bestaat ook weer uit een master/detail scherm. Omdat uren het laatste niveau is, heeft dit scherm enkel een formulier als detail. In dit formulier kan naar het juiste record genavigeerd worden, de gegevens kunnen ververst worden, uren kunnen toegevoegd, gewijzigd of verwijderd worden. Als men klaar is met muteren van een record, dan kan deze opgeslagen of geannuleerd worden.

Het hiërarchie scherm lijkt op het master/detail scherm. Het verschil is echter dat de details op hetzelfde tabblad zitten, zodat niet de lijst met gegevens van de vorige master (in dit geval medewerker en uren) weergegeven wordt. Dit heeft als voordeel dat sneller dieper in de gegevens ingezoomd kan worden en er meer ruimte op het scherm overblijft voor het uiteindelijke formulier.

Naast gegevensopslag en een GUI om deze gegevens te bereiken, bestaat een eindproduct uit specifieke business rules. Business rules zijn operaties, definities en beperkingen die voor een organisatie gelden, zodat deze haar doelen kan bereiken. Om business rules in de applicatie te gebruiken, kan een code template gebruikt worden. In deze code template staat de code die uitgevoerd moet worden. Daarbij wordt een control procedure gemaakt. Deze control procedure geeft aan op welke plekken in de applicatie (bijvoorbeeld in de documentatie, op de database of in de functionaliteit) dat stukje code uitgevoerd moet worden. Door de code in de code templates abstracter te maken, kan deze op meerdere plekken van toepassing zijn. Met de control procedure kan dan aangegeven worden welke condities van toepassing moeten zijn om dat stukje code toe te passen op de verschillende plekken.

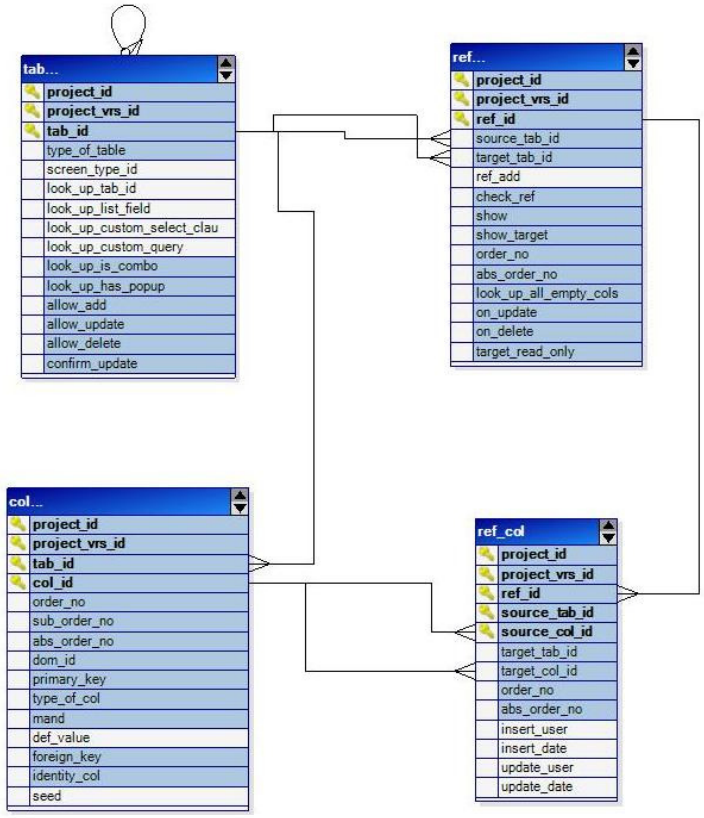
Zoals in Figuur 4 weergegeven is, worden alleen de abstracte (oranje) aspecten ontwikkeld, hieruit wordt het eindproduct (*customer product*) gegenereerd. De software ontwikkelaars moeten de MSD ingeven en de code templates en control procedures schrijven. Hiervoor zijn hulpmiddelen die zorgen dat bestaande software componenten hergebruikt kunnen worden. Bij de MSD is dat de recycler. Bij de control procedures en code templates wordt daar een bibliotheek voor gebruikt. Wanneer gekozen wordt voor unieke schermen, dan zullen deze ontwikkeld moeten worden. De GUI objecten en uniforme schermen zijn al beschikbaar voor elk product.



Figuur 6: ERD model van WORKSHOP project

3.3.3 Meta modelleren

Metadata is de data die de karakteristieken van bepaalde gegevens beschrijft [9]. Een van de tools die gebruikt kan worden om de metadata van het MSD te bekijken, is de ERD modeler. Met hulp van deze ERD modeler is een ERD diagram van een WORKSHOP project gemaakt, weergegeven in Figuur 6. Dit project bestaat uit een gebruiker die zijn gemaakte uren kan boeken op een project activiteit van een project. Dit project wordt gemaakt voor een klant en met deze klant kunnen afspraken gemaakt zijn over de uren.



Figuur 7: Gedeelte van het meta model

Voor dit project is een meta model gemaakt. In een metamodel worden de metagegevens en de relaties tussen de gegevens weergegeven. Een deel van dit meta model is weergegeven in Figuur 7. De meta data wordt opgeslagen in tabellen (*tab*). In deze tabellen zitten kolommen (*col*). Tussen de verschillende tabellen liggen referenties (*ref*). Deze referenties bestaan uit referentiekolommen (*ref_col*) die aangeven op welke kolommen de referenties gebonden worden.

In *tab* zitten zes records met daarin de tabellen *customer*, *employee*, *hour*, *project*, *project_activity* en

sales_invoice. In *col* worden voor alle tabellen de kolommen weergegeven. Bijvoorbeeld voor *customer* bestaan vier records: *customer_id*, *name*, *web_site* en *e_mail*. In *ref* zijn de zeven referenties opgeslagen en in *ref_col* de bijbehorende kolommen.

De MSD kan zelf ook als een klantproject gezien worden. Gegevens over de MSD worden opgeslagen in een meta meta model. Dit meta meta model is ook weer met een SF te bereiken. Ook hier kan een ERD diagram van gemaakt worden. Dit ERD diagram ziet er echter hetzelfde uit als het ERD diagram van het meta model. Alleen de vulling is anders. In de *tab* zitten onder andere de tabellen *tab*, *col*, *ref* en *ref_col*. In *col* zitten de kolommen van deze tabellen, dit zijn onder andere: *project_id*, *project_vrs_id*, *tab_id* en *type_of_table*. *Ref* en *ref_col* zijn nu gevuld met referenties, zoals de referentie tussen *col* en *tab*.

Om een beter inzicht te krijgen van hoe de MSD eruit ziet, is een metamodel van de MSD gemaakt (het meta-metamodel). Dit metamodel is in bijlage III weergegeven. Vanwege de complexiteit is dit model niet leesbaar genoeg en daarom is het opgedeeld in verschillende delen:

- Datamodel
- Gebruikersinterface
- Business functionaliteit
- SF model

In het datamodel wordt weergegeven welke gegevens in het eindproduct opgeslagen zullen worden. Deze gegevens vormen de database. In het model van de gebruikersinterface zijn de gegevens weergegeven die ervoor zorgen dat een GUI voor een specifieke applicatie goed werkt. In de gebruikersinterfaces wordt o.a. opgeslagen welke schermen via het menu geopend moeten worden, welke lay-out de GUI weergeeft en op welke momenten de unieke helpteksten weergeven moeten worden. Het model met business functionaliteit geeft aan hoe de business rules worden opgeslagen in de MSD. Naast de business rules kan in het business functionaliteit model ook aangegeven worden welke processen in de business uitgevoerd moeten worden. Hiervoor kunnen taken, rapportages, prefilters, meldingen en dergelijke gedefinieerd worden. Naast de gegevens die nodig zijn voor het eindproduct, zijn er ook gegevens over het proces opgeslagen. Deze gegevens worden door de verschillende tools gebruikt en zijn weergegeven in het SF model. Dit zijn gegevens voor de validatie, het weergeven van een ER-diagram, onderhoud tussen verschillende versies, controles van naamgeving, e.d.

3.3.4 Microsoft vs. Thinkwise

Greenfield en Short [6] geven aan dat bij een software factory verbeteringen gedaan moeten worden op gebied van abstractie, granularity en specificiteit. De Meta Solution definition wordt gebruikt om een hoog niveau van abstractie te specificeren. Voor specifieke delen die niet op dit niveau gespecificeerd kunnen worden, zijn er de code templates, waar heel specifiek business rules gespecificeerd worden en de GUI objecten die gebruik kunnen maken van unieke schermen die geheel handmatig geprogrammeerd kunnen worden.

De software factories van Microsoft zijn opgedeeld in componenten. Bij zowel de standaard als de unieke schermen wordt gebruik gemaakt van componenten. Zo kunnen alle componenten die in de standaard schermen zitten, ook gebruikt worden door een uniek scherm. Daarnaast

kunnen de nieuwe componenten die voor een nieuw scherm gemaakt worden ook in andere GUI's (van andere producten) hergebruikt worden. De granularity gaat nog verder door het gebruik van code templates. De objecten, zoals triggers en documentatie, worden opgedeeld in kleinere onderdelen. Deze onderdelen maken gebruik van parameters die gevuld worden met gegevens uit de MSD en kunnen daarom ook in andere producten hergebruikt worden.

De specificiteit geeft aan in hoeverre functionaliteit hergebruikt kan worden en gebruikt kan worden voor meerdere producten. Greenfield en Short richten zich op Software Product Lines en geven aan dat hun software factories niet of minder geschikt zijn voor One-off producten. De Thinkwise Software Factory richt zich op bedrijfkritische database applicaties. Doordat de software factory zich richt op een specifiek domein, is de ontwikkeling krachtiger en kunnen de componenten en templates die ontwikkeld worden ook hergebruikt worden. Greenfield en Short geven aan dat applicaties ontwikkelen voor economies of scale al goed ontwikkeld is in de software industrie, maar dat economy of scope nog de nodige aandacht vraagt. Thinkwise maakt unieke applicaties die precies aansluiten op de wensen van de klant en geen scherm van een applicatie is gelijk aan een scherm van een andere applicatie. Maar omdat Thinkwise gespecialiseerd is in bedrijfskritische database applicaties kunnen deze met de software fabriek sneller en goedkoper geproduceerd worden. Daarom is dit economy of scope, net zoals een grasmaaier totaal anders is dan een automotor, maar met kennis van motoren kan een goede grasmaaier motor sneller ontwikkeld worden dan zonder deze kennis.

4 Kwaliteit

Voordat de kwaliteit verbeterd kan worden, moet eerst gedefinieerd worden wat kwaliteit precies inhoudt. Kwaliteit is lastig te beschrijven, mensen weten wel hoe het niet moet en wat dus slechte kwaliteit is, maar goed kwaliteit is veel moeilijker te bereiken.

4.1 Geschiedenis van Kwaliteit

De geschiedenis van kwaliteit heeft haar oorsprong in de wiskundige statistieken. Door statistische methodes op het productieproces los te laten, is het mogelijk om de kwaliteit van de producten beter onder controle te krijgen. Walter A. Shewhart (1891-1967) van Bell labs benadrukt in zijn onderzoek hoe belangrijk metingen en statistieken zijn. Het centrale punt van zijn publicaties is, hoe ervoor te zorgen dat er data over het productieproces opgeslagen wordt, waaruit conclusies getrokken kunnen worden ter verbetering van het productieproces. Voornamelijk gegevens over producten met afwijkingen van de standaard (variaties), zodat deze variaties gereduceerd kunnen worden [10].

In 1927 leert W. Edwards Deming (1900-1993) de concepten van Statistical Quality Control van Shewhart. Deming is de eerste Amerikaan die het belang van kwaliteit op grote schaal in Japan introduceert [11]. Omdat Japan in die tijd een arm land was, weigerde Deming betaald te krijgen voor zijn colleges en seminars. Als waardering is toen de *Deming prize* ontstaan die elk jaar uitgereikt wordt aan bedrijven ter promotie van de constante ontwikkeling van kwaliteitscontrole in Japan [12]. Deming beschreef zijn filosofie in 14 punten. Deze punten werden gezien als actiepunten voor top management ter verbeteren van de kwaliteit. Door de jaren heen zijn deze punten gewijzigd, waardoor inconsistenties in de literatuur zijn ontstaan. Volgens Roa et al [11] zijn de 14 punten van Deming samen te vatten in drie brede filosofie categorieën:

- *Constancy of purpose*
  De organisatie kan zich niet toewijden op het verbeteringsprogramma, wanneer het management telkens blijft veranderen van methode.
- *Continual improvement*
  Managers moeten constant proberen te verbeteren i.p.v. de huidige foutmarges te accepteren.
- *Cooperation between functions*
  Iedereen moet weten welk werk gedaan moet worden, zodat taken niet overlappen of buiten functies vallen.

Joseph M. Juran (1904) deelde deze visie [10]. Hij kwam vier jaar na Deming naar Japan. In 1979 richtte hij het Juran Institute op. Juran drukt zijn ideeën over kwaliteit uit in de vorm van de *Quality Trilogy*. Dit bestaat uit 3 processen die, volgens Juran, van belang zijn voor kwaliteitmanagement [11]:

- Quality planning – identificatie van klanten en hun behoeften
- Quality control – identificatie van de kritische elementen en hun meetbaarheid
- Quality improvement – verbeteren van de kritische elementen ten behoeve van de klanten

Philip B. Crosby (1926-2001) kwam met de kreet "*quality is free*". Hij bedoelt hier niet mee dat het geen geld kost om kwaliteit te bereiken, maar juist dat als berekend wordt welke extra kosten gemaakt worden bij slechte kwaliteit en welke kosten dus niet meer gemaakt zullen worden bij een goede kwaliteit, dan weegt dit op tegen de kosten om de betere kwaliteit te bereiken. Goede kwaliteit kost dus meer om te bereiken, maar zorgt niet voor verlies. Daarom was hij er een voorstander van om foutloze producten te maken. Volgens Crosby zijn er vier absolute standpunten van kwaliteit management:

- Quality means conformance to requirements
- Quality comes from prevention. And prevention as a result of training, discipline, example, leadership, and more.
- Quality performance standard is zero defects. Errors should not be tolerated.
- Quality measurement is the price of nonconformance

De filosofieën van Deming, Juran, Crosby, en anderen, zoals Armand V. Feigenbaum en Koru Ishikawa hebben een grote invloed gehad op de Japanse economie. In 1985 introduceerde de Naval Air Systems Command de term Total Quality management (TQM) voor het beschrijven van de Japanse stijl van management op de kwaliteitsverbetering [13].

4.2 Definitie kwaliteit

Er zijn door verschillende mensen verschillende definities van kwaliteit gegeven. Gravin (1988) was de eerste die deze definities in verschillende categorieën onderbracht [11]. De eerste categorie is *Transcendent approach* en wordt beschreven in een quote van Barbara Tuchman (1980).

"*A condition of excellence implying fine quality as distinct from poor quality…quality is achieving or reaching for the highest standard as against being satisfied with the sloppy or the fraudulent.*" [11]

De tweede categorie, *product-based approach,* richt zich op de eigenschappen en attributen (bijvoorbeeld het materiaal dat gebruikt wordt) die gemeten kunnen worden. Wanneer deze een hogere kwaliteit aangeven, zal het product ook van hogere kwaliteit zijn. *User based approach* richt zich op de gebruiker. Wat beter is voor de gebruiker, is van hogere kwaliteit. Dit komt overeen met de filosofie over kwaliteit van Juran, hij definieert deze manier van kwaliteit als "fitness for use". De vierde categorie is de *manufacturing-based approach*, deze wordt door Crosby beschreven als de "conformance to requirements". Ingenieurs specificeren het product, hoe dichter de implementatie komt bij de requirements, hoe hoger de kwaliteit van het product is. De laatste categorie is *value-based approach*, deze categorie richt zich als enige op het element van de prijs en wordt door Broh (1982) beschreven als:

"*Quality is the degree of excellence at an acceptable price and the control of variability at an acceptable cost*" [11]

Bij deze categorie gaat Garvin er vanuit dat de koop van een product een compromis is tussen kwaliteit en prijs. Hij definieert daarbij acht criteria die door de gebruiker belangrijk gevonden worden bij de kwaliteit van een product.

- Performance – de primaire meetbare eigenschappen
- Features – extra eigenschappen die het product aantrekkelijker maken voor een gebruiker
- Reliability – de waarschijnlijkheid dat het product niet faalt
- Conformance – de precisie waarin het product aan de gespecificeerde standaarden voldoet
- Durability – meet de houdbaarheid van het product (product's life)
- Serviceability – de snelheid waarmee het product weer kan werken nadat het stuk is gegaan
- Aesthetics – de manier waarop de gebruiker reageert op het product (bv. geur, smaak)
- Perceived quality – extra kwaliteit eigenschappen op basis van indirecte metingen.

4.3 Software kwaliteit

De categorieën van kwaliteit die Gravin beschrijft, zijn niet overal op toepasbaar. Roa et al geven aan dat de categorieën slechts geschikt zijn voor producten, voor de kwaliteit van services bestaan weer andere categorieën. Ook de kwaliteit van software heeft raakpunten met de algemene beschrijving, maar kan in deze beschrijving niet geheel toegepast worden. Dit komt omdat software een aantal specifieke karaktereigenschappen heeft die fysieke producten niet hebben. Volgens Brooks [11; 14] zijn dit:

- Complexity – Software products are more complex than perhaps any human constructs are.
- Conformity – There are no "physical laws" in software to conform to.
- Changeability – It is easy to change software, but the consequences are often overlooked.
- Invisibility – Software is invisible and hence we cannot use the powerful tool of geometric abstraction.

Brooks geeft aan dat software complex wordt in grootte, omdat geen twee delen hetzelfde zijn. Als ze, op een level hoger dan statement niveau, hetzelfde zijn dan wordt het herschreven naar één functie die door beide delen aangeroepen wordt. Daarnaast heeft een software systeem enorm veel verschillende statussen (states) waarin het systeem zich kan bevinden, van waaruit het systeem naar veel verschillende andere statussen (states) kan voortbewegen.

In de jaren '60 waren de kosten van hardware de grootste kosten voor het implementeren van informatie technologie. Toen de prijs van hardware begon te dalen, werden de werkelijke kosten van software ontwikkeling duidelijk en verschoof de focus naar planning en controle van software projecten [15]. Brooks [14] gaat in het artikel "No Silver Bullet -... " op zoek naar een reden waarom software niet zoveel productiever, betrouwbaarder en simpeler kan worden als hardware de afgelopen jaren sneller, beter en goedkoper geworden is. In het artikel wordt de vergelijking gemaakt met een weerwolf, die normaal goed beheersbaar is, maar opeens kan veranderen in een monster. Zo kan een software project ook goed lopen en opeens veranderen in een monster dat deadlines mist, budgetten opblaast en producten vol fouten oplevert. Een weerwolf is te doden met een zilveren kogel, zo wordt ook gezocht naar "een kogel", die het monster van software projecten onder controle kan houden, of de reden waarom er niet een dergelijke "silver bullet" bestaat. Volgens Brooks is het moeilijkste aan software ontwikkeling de specificatie, ontwerp en test van de conceptuele structuur, en niet het werk dat gedaan moet worden om het te representeren of te testen of deze representatie werkt.

Runeson en Isacsson [10] geven in hun artikel de volgende overeenkomsten tussen software kwaliteit en kwaliteit in het algemeen aan:

- Processen en afwijkingen zijn statisch en kunnen gemeten en gecontroleerd worden door statische methoden.
- Kwaliteit moet in elke functie ingevuld worden, hoe eerder in het proces, hoe beter.

Daarnaast is er ook een aantal verschillen weergegeven.

- Meer attentie moet gegeven worden aan procesbeheer in software cases, omdat de processen abstracter zijn dan bijvoorbeeld auto bouwprocessen, die altijd zichtbaar zijn.
- Software processen zijn erg afhankelijk van mensen, terwijl andere processen machineafhankelijk zijn. Dit maakt software processen minder precies en controleerbaar.
- Het is moeilijker om grote en stabiele testdata over het ontwikkelproces te verzamelen. Het ontwikkelen van een software product duurt een jaar, terwijl bij andere producten elke dag meerdere stuks gemaakt worden.

Zuser, Heil en Grechenig [16] beschrijven in hun artikel de software quality development en software quality assurance van RUP (Rational Unified Process), MSF (Microsoft Solution Framework) en XP (eXtreme Programming). Ze geven aan dat bij hoge kwaliteit vaak wordt gekeken naar attributen, zoals klant tevredenheid opgesplitst in functional requirements, usability, security, stability, etc. Zuser et al geven aan dat er technieken zijn om kwaliteit te verzekeren door defects (fouten) op te sporen, maar ze wijzen erop dat dit een dure methode is en dat het daarom beter is om kwaliteit te verzekeren door fouten te voorkomen. Dit is in tegenstelling met wat Crosby aangeeft, namelijk dat goede kwaliteit zoveel kosten bespaart dat het opweegt tegen de kosten om de kwaliteit te bereiken. Zuser et al vinden dat software quality assurance ervoor moet zorgen dat er geen defects zijn aan het einde van elke project mijlpaal. Volgens Zuser et al is software kwaliteit te bereiken door drie benaderingen: testen, statische analyse en ontwikkel methodes. Een andere categorie van benaderingen om software te verbeteren is betere kwaliteitsevaluatie, betere kwaliteitsmetingen, betere processen en betere tools. Op proces niveau kan het gebruik van standaards zorgen voor betere kwaliteit.

5 Kwaliteitsysteem

Kwaliteit, zoals dat door Shewhart en anderen beschreven is, geldt voor massaproducten. Er worden statistieken losgelaten op de kosten. Het kost een bepaald bedrag om voor elk geproduceerd product te controleren of het aan de eisen voldoet. Wanneer niet elk product, maar slechts een selectie gecontroleerd wordt, dan zijn de kosten voor het controleren minder. De kans is dan echter groter dat er een product, dat een fout bevat, niet gecontroleerd wordt. Als dit product bij een klant komt, kunnen de kosten om de fout te repareren of de schade te herstellen hoog oplopen. Daarom worden modellen gemaakt om een balans te vinden tussen de hoeveelheid controle van producten en de kans dat producten nog fouten bevatten. Nadat met het model gewerkt is, kan men onder andere de volgende gegevens zien: het aantal dat daadwerkelijk gecontroleerd is, het aantal fouten die tijdens de controle gevonden zijn, het aantal producten dat teruggestuurd is en de kosten die gemaakt werden, omdat er foute producten de klant bereiken. Aan de hand van de statistieken kunnen tactieken ingezet worden om fouten te verminderen, daarna kan het model aangepast worden, zodat er meer of minder gecontroleerd wordt.

Software is het tegenovergestelde van massa productie. Bij massa productie worden elke dag duizenden of miljoenen producten gemaakt. Elk product is precies hetzelfde, echter soms met een kleine afwijkingsmarge. Een software programma is uniek en kost veel tijd om gemaakt te worden. Het kost soms maanden tot jaren, totdat een product klaar is. Als het product vervolgens klaar is, kan het perfect gekopieerd worden, zodat miljoenen met hetzelfde product kunnen werken op verschillende computers. Bij massa productie kan een product makkelijk getest worden, bijvoorbeeld door het meten van gewicht, grootte, buigzaamheid, etc. Software heeft niet een los te identificeren eigenschap waarop het getest kan worden. Omdat software niet een fysiek product is, wordt het getest op het gedrag. Er wordt getest wat er gebeurd als een programma een bepaalde input krijgt. Bijvoorbeeld bij een simpel programmaatje dat het product van twee getallen berekent. Dit programma moet als gedrag hebben dat het een 6 terug geeft wanneer de waardes 2 en 3 ingegeven worden. Door alleen met de waarde 2 en 3 te testen, is het nog niet zeker dat het programma correct werkt. Om er zeker van te zijn dat het programma precies werkt zoals verwacht, moeten alle mogelijke getallen (binnen het bereik van de parameters) gecontroleerd worden.

Het voorbeeld van het product van twee cijfers is deterministisch, dat wil zeggen dat bij dezelfde input, altijd dezelfde output eruit komt. Er zijn ook programma's die dat niet zijn, zoals een spel met een dobbelsteen. Bij een dobbelsteen moet de kans op een 6 altijd gelijk zijn, maar de dobbelsteen mag niet zomaar altijd een 6 geven. Als de dobbelsteen "gegooid" wordt, is de uitkomst van tevoren dus niet te berekenen. Toch kan, door heel vaak de dobbelsteen te gooien, getest worden of elk cijfer wel aan bod komt en dus of de kansberekening goed uitgevoerd wordt.

Het volledig testen van complexe programma's duurt zo lang, dat de mensen die de code oorspronkelijk geschreven hebben, niet meer zouden leven als alle testen klaar zijn. Het testen van een bepaald programma kan zelfs oneindig lang duren. Om zo lang te testen, heeft natuurlijk geen zin, aangezien niemand het product dan kan gebruiken. Het andere uiterste is om het product helemaal niet te testen. Er zijn dan geen kosten gemaakt en er is dan geen tijd

verloren gegaan aan het testen van het programma. Dit wordt in de praktijk wel gedaan wanneer er een grote tijdsdruk is, maar heeft als nadeel dat er nog fouten in het product kunnen zitten. De fouten kunnen grote gevolgen met zich meebrengen. Bijvoorbeeld als het programma dat het product uitrekent, gebruikt wordt om mijn uurloon met het aantal gewerkte uren te vermenigvuldigen en het programma af en toe de helft van het bedrag berekent. Om dit probleem te verminderen, wordt in de software naar verschillende oplossingen gekeken.

Ten eerste probeert men fouten te vermijden. In hoofdstuk o worden standaarden beschreven, die ervoor zorgen dat de software die gemaakt wordt, van betere kwaliteit is. In hoofdstuk o en 8 worden technieken beschreven, die bij Thinkwise ervoor kunnen zorgen dat fouten voorkomen worden in plaats van gevonden bij testen en daarna pas verbetert.

Ten tweede probeert men sneller en automatisch te testen, zodat de testen die uitgevoerd worden sneller resultaat opleveren en minder geld kosten. In hoofdstuk o wordt gekeken hoe de software van Thinkwise, door gebruik te maken van de eigenschappen van de software fabriek, sneller getest kan worden dan normale software.

Als laatste wordt, net zoals dit bij massa productie wordt gedaan, geprobeerd een model te bedenken dat de balans aangeeft tussen kosten die de risico's van niet testen met zich meebrengen en de kosten van het wel testen. Deze optie werkt bij software niet zo goed als bij massaproductie. Omdat de ontwikkelcyclus van software zo lang is, kost het veel meer tijd, voordat een model gevalideerd kan worden. Daarom zal speciaal voor Thinkwise een model opgezet moeten worden dat door het meten van projecten telkens bijgesteld kan worden. Het meten en bijstellen van een model wordt een kwaliteitsysteem genoemd.

5.1 Model

In "managing quality" van Kelemen [17] wordt een model van kwaliteitscontrole weergegeven. Het model bestaat uit de volgende zes stappen:

Stap 1: Define the quality characteristics of the product of service
Stap 2: Decide how to measure each quality characteristic
Stap 3: Set quality standards for each characteristic
Stap 4: Collect and analyze data
Stap 5: Find and correct causes for poor quality
Stap 6: Continue to make improvement

Ondanks dat dit model bedoeld is voor fysieke producten en services, is het toch geschikt om voor software gebruikt te worden, omdat software ook bepaalde kwaliteitseigenschappen heeft. Hiervoor kunnen standaarden neergezet worden en deze kunnen gemeten worden. Het meten van kwaliteit wordt in software gedaan met testen.

5.1.1 Definiëren en meten van kwaliteiteigenschappen

In testen verschilt de software wel van de fysieke producten. Zo kan een fysiek product getest worden door bijvoorbeeld te meten hoe hoog het is of hoe zwaar. Bij het testen van software moet het gedrag nagebootst worden. In hoofdstuk 6.2 worden zes eigenschappen van software kwaliteit beschreven, dit zijn functionality, reliability, usability, efficiency, maintainability en portability. Voor al deze eigenschappen kunnen verschillende test geschreven worden. Zo kan

voor functionality een unit test geschreven worden, die controleert of één klein onderdeel, zoals een methode, voldoet aan de eisen. Voor reliability kan een load test gedaan worden, hierbij wordt een heleboel rekenkracht van het systeem gevraagd. Vervolgens wordt gecontroleerd of het systeem op een goede manier met deze stress omgaat. Op dit moment wordt vooral gefocust op functionaliteit, later kunnen testen geïntroduceerd worden voor de overige eigenschappen van software.

Waar wel rekening mee gehouden moet worden, is het verschil tussen massaproductie, waar de kwaliteitscontrole van Kelemen voor gebruikt wordt, en software. Bij massaproductie is het verstandig om tien keer op een dag een product uit de lijn te pakken en dit te testen, terwijl bij software het verstandiger is om tien nieuwe testen te schrijven dan één test met dezelfde opstelling tien keer uit te voeren.

5.1.2 Opslaan van testen

Een test kan handmatig of automatisch uitgevoerd worden. Testen die automatisch uitgevoerd zijn, kunnen ook automatisch nog een keer herhaald worden. Bijvoorbeeld wanneer een ander deel van de functionaliteit wijzigt, dan kan gecontroleerd worden of de reeds geteste delen nog correct werken. Voor het automatisch testen moet eerst een keuze gemaakt worden welke type testen gebruikt worden. Voor alle soorten testen zijn verschillende tools beschikbaar. Voordat begonnen kan worden met automatisch testen, moet een tool gekozen worden en deze moet ingesteld worden, zodat deze geschikt is voor gebruik.

Daarom is besloten eerst te kijken naar handmatige testen. Wanneer handmatige testen uitgevoerd zijn, dan zijn de testen weg. Dit is totaal onbeheersbaar. Om het proces beter beheersbaar te maken, zouden de testen opgeslagen kunnen worden. Vervolgens kunnen statistieken losgelaten worden op deze gegevens om te beslissen of er genoeg getest is. Als er niet genoeg getest is, kan gekeken worden welke delen dan nog getest zullen moeten worden.

Voor het opslaan van handmatige testen, moet de volgende gegevens opgeslagen worden.

- Welke template is getest
- Op welke tabel deze template getest is
- Op welke type programmaobject (default, update trigger, delete trigger, etc) getest is
- Test gemaakt door
- Test gemaakt op
- Test gereviewd door
- Test gereviewd op
- Welke werkwijze is gebruikt
- Rol van de tester
- Test geslaagd
- Reden niet geslaagd

Template vs. tabel

Als er per template getest wordt, dan heeft dit als voordeel dat de testen opgedeeld worden in kleine, beter te beheersen, brokken. Het nadeel is echter, dat dan bij een test van een template die in drie tabellen voorkomt, maar voor één tabel getest wordt. Er kan dan in de andere tabellen iets mis gaan, doordat er code na of code ervoor uitgevoerd wordt.

Er kan ook per tabel getest worden. Een voordeel hiervan is dat het grotere geheel getest wordt. Daar tegenover staat dat er een heleboel stukken functionaliteit getest moeten worden, terwijl dit maar met één test gecontroleerd wordt. Het is dus minder duidelijk of alle belangrijke delen wel getest zijn.

Als compromis hiertussen kan ook besloten worden om elke template in elke tabel te testen. Hierdoor wordt het testen opgedeeld in beheersbare stukken en worden er niet grote delen overgeslagen. Het nadeel is wel dat dit ervoor zorgt dat er erg veel mogelijkheden zijn om te testen. Neem bijvoorbeeld een template die automatisch op 14 tabellen komt, dan kost het veel tijd om alle 14 templates te testen, terwijl na de vierde template er waarschijnlijk toch geen fouten meer in zitten, omdat de code generiek is. Hetzelfde geldt voor templates die op elke tabel zitten. Het kost dan enorm veel tijd om elke tabel hierop te testen en dit levert waarschijnlijk weinig resultaat op.

Er is voor gekozen om zowel de template als de tabel op te slaan, zo kan het meest gestructureerd en uitgebreid getest worden. Bij de controle of er genoeg getest is, zal er echter rekening mee gehouden moeten worden dat bij een generieke test slechts een selectie getest hoeft te worden.

Naast tabel en template wordt ook opgeslagen welk programmaobject getest is. Een programmaobject is een onafhankelijk deel broncode dat los kan worden gecompileerd, zoals een trigger, een functie of een procedure. Het is mogelijk om een template te schrijven die geldt voor de update en de delete trigger. Wanneer alleen de update trigger getest is, dan kan het zijn dat de delete trigger helemaal niet werkt. Daarom moet ook het object type opgeslagen worden.

Tester

Er moet opgeslagen worden, wie de template getest heeft. Als er dan een fout in de template blijkt te zitten, kan zowel de ontwikkelaar als de tester hierop aangesproken worden. Zo wordt deze fout door de ontwikkelaar niet meer gemaakt en wordt het door de tester in volgende testen wel herkend. Dit verbetert daarom de kwaliteit van het toekomstig werk van de ontwikkelaar en de tester.

Een test moet ook gereviewd kunnen worden. Met een review wordt gekeken naar het ontwerp en de uitwerking van de test. Er kan vervolgens besproken worden welke keuzes er gemaakt zijn en waarom sommige dingen niet en andere dingen wel getest zijn. Het gaat hier vooral om de werkwijze die gereviewd wordt.

De werkwijze moet ook opgeslagen worden, zodat wanneer de test gecontroleerd wordt, gekeken kan worden wat er eigenlijk getest is. Ook heeft het beschrijven van de werkwijze als voordeel dat als de functionaliteit wijzigt, deze werkwijze opnieuw uitgevoerd kan worden om de regressie te testen.

Nadat een bepaalde functionaliteit aangepast is, moet deze opnieuw getest worden. Om te controleren of een test voor of na de wijziging uitgevoerd is, kan een datum van testen opgeslagen worden. Door deze te vergelijken met de datum waarop de template gewijzigd is, kan gecontroleerd worden of de wijzigingen voor of na het testen gemaakt zijn. Het zou

eventueel ook mogelijk zijn om te controleren welke regels veranderd zijn en welke regels opnieuw getest moeten worden. Er is voor gekozen om dit niet te doen, omdat een template het kleinste niveau is waarop getest zou moeten worden. De template moet een klein stukje functionaliteit zijn. Als dit een enorm groot stuk is, kan het beter opgedeeld worden in verschillende templates. Door deze werkwijze is het niet logisch dat een deel van een template goed werkt, terwijl in een ander deel wijzigingen zijn doorgevoerd. Wanneer er wijzigingen in een template gemaakt zijn, zou dus de hele template opnieuw getest moeten worden en niet alleen de regels die wijzigen. Daarom heeft het ook niet zoveel zin om bij te houden welke regels er gewijzigd zijn en hoeveel regels in totaal nog niet getest zijn.

Als laatste moet opgeslagen worden wat de rol van de tester is. Op deze manier kan naast de ontwikkelaar ook de gebruiker van het eindproduct opslaan wat er getest is. Een gebruiker kan natuurlijk alleen per tabel testen, omdat hij of zij niet weet of de functionaliteit de verwacht wordt in de layout, default of een trigger zou moeten zitten.

5.1.3 Verbeteren van kwaliteit

Bij massaproductie kan een reden gevonden worden, waarom de kwaliteit niet aan de eisen voldoet. Dit gaat meestal niet op bij software. Het kan zijn dat een ontwikkelaar slecht werk aflevert en daarom de algemene kwaliteit omlaag haalt, maar meestal is het probleem met slechte kwaliteit van software dat een ontwikkelaar een fout gemaakt heeft of iets over het hoofd gezien heeft. Door te testen kunnen deze fouten boven water komen. Fouten die gevonden zijn moeten daarom aangepast worden, zodat de kwaliteit verbetert.

In plaats van naar een manier te zoeken om kwaliteit te verbeteren, zal er gecontroleerd moeten worden of er wel genoeg getest wordt. Wanneer er niet genoeg getest is, zal een nieuwe cyclus met testen en verbeteren van broncode ingezet moeten worden. Het meten of de functionaliteit genoeg getest wordt, is te vergelijken met de vijfde stap in het model van Keleman, waarbij de slechte punten in het proces worden gevonden en verbeterd.

Er zouden een aantal statistieken gebruikt kunnen worden, om te beslissen of het meten van kwaliteit (testen) genoeg is. Een aantal mogelijke statistieken, die verzameld kunnen worden, zijn:

- Test per template (min. 1)
- Test per template per tabel (min. 1, hiervoor kan een uitzondering gemaakt worden als een template heel generiek is en op veel tabellen voorkomt)
- Test per template per programmaobject type (min. 1)
- Test per tabel (min. 1)
- Aantal testen geslaagd (>90 %)
- Aantal testen niet geslaagd (<10%)
- Aantal testen waarvan de code ondertussen gewijzigd is (<10%)

Voor deze statistieken moet vervolgens aangegeven worden wat de standaard is voor de verschillende statistieken. Bij de statistieken zijn voorbeelden van standaarden erachter weergegeven. Elke template zou minimaal één test moeten hebben. Als een template op meerdere tabellen zit, dan zou de template ook meerdere keren getest moeten worden, tenzij deze template op erg veel tabellen van toepassing is. Bijvoorbeeld, als er 14 tabellen zijn. Er

zouden er dan bijvoorbeeld drie getest moeten worden. Hetzelfde geldt voor een programmaobject type. Elke tabel kan verschillende programmaobject types hebben (*default*, *layout*, *insert trigger*, *update trigger* en *delete trigger*, etc). Voor elk programmaobject zou minimaal één test geschreven moeten worden. Het is ook mogelijk op een tabel de verschillende programmaobjecten niet te gebruiken, daarom zou voor elke tabel toch minimaal 1 test geschreven moeten worden om te controleren of deze tabel wel correct gebruikt kan worden.

De volgende twee controles zijn elkaars complement. Eigenlijk zouden alle testen moeten slagen, maar soms is er functionaliteit die door technische redenen nog niet correct geïmplementeerd kan worden. Wanneer er wel een test voor geschreven zouden zijn, dan kan nooit meer aan 100% geslaagd percentage gekomen worden. Dit heeft als resultaat dat de geschreven testen weggegooid worden, zodat met een vertekend beeld aan de standaard voldaan kan worden. Dit is niet de bedoeling, daarom zou de standaard minder dan 100% moeten zijn. Er moeten echter niet te veel fouten gevonden zijn, die nog niet opgelost zijn. Als dit het geval is, dan moet extra prioriteit gezet worden op het oplossen van bekende fouten, voordat men verder gaat met nieuwe functionaliteit ontwikkelen en testen.

Wanneer een template getest is, kan daarna besloten worden om de template weer aan te passen. De test moet vervolgens opnieuw uitgevoerd worden. Daarom wordt gecontroleerd welk percentage van de templates opnieuw getest zou moeten worden. Dit percentage zou zo laag mogelijk moeten zijn.

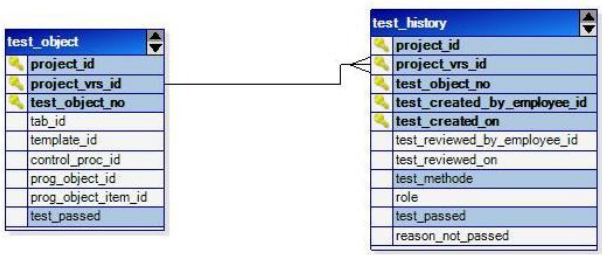
5.1.4 Evalueren van testen

De projectleider zou in het project bij moeten houden of het project voldoet aan de kwaliteitseisen op gebied van testen. Wanneer het project niet voldoet, moet meer getest worden, zodat de kwaliteit van het eindproduct verbeterd.

Omdat software een lang ontwikkelproces heeft, is het lastig om de standaard van de statistieken goed in te stellen. Nu is er een voorbeeldstandaard gezet, maar het is goed mogelijk dat in de praktijk blijkt dat deze standaard niet haalbaar is. Aan de andere kant kan het ook dat de standaard te soepel is en dat de kwaliteit zelfs na behalen van de standaard niet goed genoeg is. De standaard moet daarom geëvalueerd worden, zodat besloten kan worden om het naar beneden of naar boven aan te passen. De managers van het bedrijf zouden periodiek met verschillende projectleiders moeten overleggen, om te controleren of de standaarden haalbaar zijn en deze indien nodig aanpassen.

5.2 Uitwerking

De kwaliteit kan opgeslagen worden in de elektronische bouwtekening van de software fabriek. Omdat de elektronische bouwtekening ook met een software fabriek ontwikkeld is, worden de schermen waarmee de testgegevens opgeslagen worden automatisch opgebouwd. In Figuur 8 is het datamodel weergegeven dat aan de



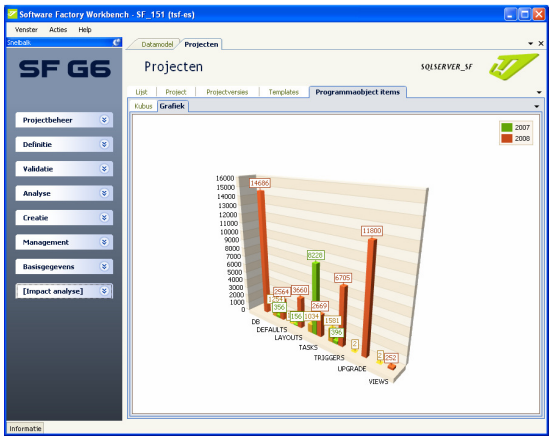
Figuur 8: Datamodel testopslag

elektronische bouwtekening toegevoegd zal moeten worden.

De gegevens die opgeslagen moeten worden zijn onderverdeeld in een test object en een test geschiedenis. De test objecten kunnen automatisch gevuld worden zodat ze aan de eerder beschreven eisen voldoen. Vervolgens kan elke keer dat er getest wordt een record toegevoegd worden aan de testgeschiedenis. De ontwikkelaar geeft aan wanneer er getest is en hoe deze test uitgevoerd is. Vervolgens kan aangegeven worden of de test geslaagd is en als deze niet geslaagd is, dan moet opgegeven worden waarom deze niet geslaagd is.

Als de laatste test niet geslaagd is, dan is het hele object niet geslaagd. Wanneer later een test toegevoegd wordt en deze slaagt wel, dan is het hele object geslaagd. Als er geen geschiedenis van een test is, dan is dat test object dus ook niet geslaagd.

De software fabriek heeft vervolgens ook de mogelijkheid om management informatie over de gegevens te bekijken met behulp van kubussen. Zo kan bijvoorbeeld gekeken worden hoeveel test objecten geen test geschiedenis hebben of hoeveel testen niet geslaagd zijn. Er kan gekeken worden hoe vaak een test uitgevoerd is en hoelang het geleden is dat de testen uitgevoerd zijn. Hierbij kan een grafiek gemaakt worden. Dit is in Figuur 9 weergegeven voor het aantal regels code die de verschillende codegroepen bevatten.



Figuur 9: Grafiek component

6 Kwaliteit standaarden

Om kwaliteit te definiëren en de kwaliteit te verbeteren, zijn er verschillende standaarden ontwikkeld. Dit zijn standaarden over kwaliteit en software kwaliteit, zoals ISO 9000, ISO/IEC 9126, Capability Maturity Model (CMM) en Personal Software Process (PSP). De ISO 9000 series zijn drie protocollen, die kwaliteit in het algemeen beschrijven. De ISO/IEC 9126 is speciaal ontwikkeld voor software, met deze standaard wordt de evaluatie van een software product gemodelleerd. De CMM en PSP zijn beide software kwaliteit standaarden, die zich richten op het software proces. De PSP is een uitgebreide versie van CMM en richt zich op de individuele personen, die aan het software proces deelnemen.

6.1 ISO 9000

De eerst standaard die bekeken is, gaat niet over software, maar over producten in het algemeen. Deze standaard is ontwikkeld toen software nog niet een deel was van elke organisatie.

ISO 9000 is door de International Organization for Standardization (ISO) ontwikkeld. In 1987 heeft het European Committee for Standardization (ECS) de ISO 9000 standaarden geadopteerd. De standaarden zijn bedoeld om op elke industrie toegepast te kunnen worden in elk van de 91 landen die gerepresenteerd worden in de International Organization for Standardization [11]. ISO 9000 bestaat uit drie niveaus, ISO 9003, ISO 9002 en ISO 9001. Bedrijven die op het niveau van ISO 9003 geregistreerd zijn, voldoen aan de kwaliteitseisen voor inspectie en testen. Auditoren controleren het testgereedschap en verzekeren zich ervan dat de gereedschappen regelmatig afgesteld worden.

Bedrijven die geregistreerd zijn op ISO 9002 voldoen aan alle eisen van ISO 9003. Daarnaast zijn de inspectie en testpraktijken doorgevoerd in de organisatie van de productie. Auditoren controleren dat aan Statistical Quality Control gehouden wordt door het gehele proces en dat er inspanningen gedaan worden om de kwaliteit van leveranciers te controleren en verbeteren.

ISO 9001 impliceert dat de kwaliteit praktijken verspreid zijn naar ontwerp en onderhoud services. Dit is de meest uitvoerige registratie van de ISO 9000 series. Een van de testen houdt in dat auditoren een willekeurig pakketje pakken en controleren of deze voldoet aan de laatste wijzigingen die door de techniek zijn uitgegeven [11].

Stevenson en Barnes [18] blikken terug op veertien jaar met ISO 9000. De reden hiervoor is omdat in 2000 een nieuwe en simpelere versie van de ISO 9000 serie is uitgekomen. Het aantal bedrijven dat gecertificeerd is, groeit steeds verder. Volgens Stevenson en Bernes zijn er eind ‘99 343.643 certificaten uitgegeven. Ze geven aan dat er een ISO 9004 is ontstaan om de interne kwaliteit ontwikkeling te begeleiden. Door de jaren heen zijn er nieuwe programma’s bijgekomen, zoals de QS-9000 en de ISO 14000. Stevenson en Barnes geven als kritiek op de certificatie dat het erg lang duurt en veel geld en energie kost. Mensen moeten opgeleid worden en periodiek komen er auditoren langs voor niet goedkope inspecties. Daarnaast zorgt de certificatie voor veel overhead aan documentatie. Volgens Saiedian en McClanahan [15] kost het certificeren van ISO 9000 variërend van een paar maanden tot een paar jaar. Het certificaat is vervolgens 3 jaar geldig. Een punt van kritiek volgens Stevenson en Barnes is dat de methode te formeel is en dat de nadruk daardoor op het certificeren ligt in plaats van op de kwaliteit.

Als belangrijkste voordelen van het certificeren geven ze verbetering van documentatie, bedrijfsstandaarden, bewustwording van kwaliteit en behouden van een markt aandeel.

Stevenson en Barnes geven aan dat ISO 9000 gecertificeerde bedrijven van hun leveranciers verwachten dat zij ook gecertificeerd zijn. Er zijn voorbeelden van situaties waarin groepen organisaties gedwongen werden om zich ook te laten certificeren, omdat ze anders een groot marktaandeel kwijt zouden raken. Stevenson en Barnes geven aan dat in sommige industrieën certificatie grote voordelen oplevert, terwijl in andere industrieën er geen onderscheid gemaakt wordt tussen bedrijven die wel en bedrijven die niet gecertificeerd zijn. Als conclusie geven ze aan dat bedrijven moeten kijken wat certificatie in de markt waarin ze opereren doet. Als certificatie een voordeel oplevert of zelfs vereist is om een markt aandeel te bemachtigen, dan zou het verstandig zijn om gelijk te beginnen. Anders moet het bedrijf er nog eens over nadenken, want het kost veel geld, manuren en documentatie. Voordelen kunnen bereikt worden, maar wel op lange termijn [18].

6.2 ISO/IEC 9126

De vorige ISO standaard is ontwikkeld om toegepast te kunnen worden op zoveel mogelijk producten. Voor software ontwikkeling werden andere eisen gesteld dan bij fysieke producten, daarom heeft ISO ook een standaard speciaal voor software kwaliteit ontwikkeld.

Het Joint Technical Committee 1 van het ISO en het International Electrotechnical Commission (IEC) heeft een reeks standaarden ontwikkeld om zich op het probleem van slechte software product kwaliteit te richten [19]. Deze standaarden zijn bekend als ISO/IEC 9126. Volgens Jung et al. [19; 20] is kwaliteit uit te drukken, zoals dit in ISO 8402 gedaan wordt als:

"Totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs" [19]

Deze ISO/IEC 9126 bestaat uit vier delen: deel 1 Quality Model (2001), deel 2 External metrics (2003), deel 3 Internal metrics (2003) en deel 4 Quality in use metrics (2004). ISO/IEC 9126-1 bestaat uit 6 eigenschappen en 27 subeigenschappen. Deze zijn in tabel 1 weergegeven.

Characteristic	Subcharacteristics
functionality	Suitability, accuracy interoperability, security, functionality compliance
reliability	Maturity, fault tolerance, recoverability, reliability compliance
usability	Understandability, learn ability, operability, attractiveness, usability compliance
efficiency	Time behavior, resource utilization, efficiency compliance
maintainability	Analyzability, changeability, stability, testability, maintainability compliance
portability	Adaptability, installability, replaceability, coexistence, portability compliance

Bron: Table 1 Measuring Software Product Quality: A survey of ISO/IEC 9126 [19]

ISO/IEC 9126-4 bestaat uit: effectiveness, productivity, safety en satisfaction [19]. Een kritiek op de ISO/IEC 9126 standaarden is dat ze niet gebaseerd zouden zijn op strikt en academisch onderzoek, maar meer een instinct of gevoel van experts. Jung probeert met academisch onderzoek een relatie te leggen tussen (een aantal van) de eigenschappen van ISO/IEC 9126-1 en user satisfaction.

Jung et al geven aan dat ze vinden dat de manier waarop de standaard is opgebouwd veel ambiguïteit bevat, maar dat de standaard voor een groot deel wel valide is. Er mist volgens Jung et al nog wel empirisch werk. Hetzelfde onderzoek zou nog meerdere keren uitgevoerd moeten worden, maar dan in andere landen, met grotere gebruikersgroepen of met andere types van applicaties.

6.3 Capability Maturity Model (CMM)

De ISO 9126 standaard richt zich vooral op het software product, er zijn ook standaarden die zich richten op het software proces. Capability Maturity Model is hier een van.

Volgens Runeson en Isacsson [10] is Capability Maturity Model (CMM) de de facto standaard op het gebied van software processen en beoordeling van een organisatie"s volwassenheid (capability maturity). Het doel van CMM is om een continu proces van verbetering te ontwikkelen door middel van foutenvoorkoming, technologie innovatie en proces verandermanagement [15].

De Software Engineering Institute (SEI) is opgericht door het Amerikaanse Department of Defense (DoD). In 1986 heeft het SEI een methode ontwikkelt om leveranciers van software te beoordelen op hun vermogen om volgens planning en binnen budget software te ontwikkelen. In eerste instantie ontwikkelde het SEI, onder leiding van Watts Humphrey een conceptueel kader met een vragenlijst. Hiermee konden organisaties ondersteund worden in het verbeteren van hun software proces. Dit model met vragenlijst is geëvolueerd in het huidige Capability Maturity Model (CMM).

Er zijn vijf niveaus waarin een organisatie zich kan bevinden:

1. *Initial*
   The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.
2. *Repeatable*
   Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3. *Defined*
   The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. Projects use an approved, tailored version of the organization's standard software process(es) for developing and maintaining software.
4. *Managed*
   Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
5. *Optimized*
   Continuous process improvement is facilitated by quantitative feedback from the process and from piloting innovative ideas and technologies. [21]

Organisaties beginnen op het eerste niveau, initial. Vervolgens moet aan een aantal Key Process Areas (KPA) voldaan worden om op een hoger niveau te komen. Runeson en Isacsson

[10] bespreken één van de Key Process Areas, namelijk Software Quality Assurance (SQA). Dit omdat er veel misverstanden bestaan over de definitie van SQA. Zo wordt SQA bijvoorbeeld verward met Quality Assurance volgens ISO 9001. Testen is een vorm van Quality Assurance, maar niet van SQA. Bij SQA wordt het testproces beheerd, er wordt geverifieerd dat testers zich aan de voorgedefinieerde test procedures houden. Runeson en Isacsson definiëren software quality engineering (SQE) als de zorg voor het bouwen van software producten met de benodigde kwaliteit en het beoordelen van het niveau van deze kwaliteit. Vervolgens geven ze aan welke rol SQA speelt in software quality engineering.

Volgens Herbsleb et al [21] kost het bedrijven ongeveer twee jaar om een niveau hoger te komen. Ze geven aan dat maar ongeveer 25% van Level 1 naar Level 2 gaat binnen 21 maanden. Ongeveer 25% doet er zelfs 37 of meer maanden over. Van Level 2 naar Level 3 is makkelijker, 25% van de bedrijven deed dit in 17 maanden of minder en 25% kostten dit 31 maanden of meer.

Saiedian en McClanahan [15] wijzen erop dat ISO 9000 en SEI CMM veel overlap hebben en het kan voor sommige bedrijven wenselijk zijn om volgens beide standaarden goed geëvalueerd te worden. Er zijn daarom al projecten gestart om een combinatie van CMM en ISO 9000 te ontwikkelen, zoals Bootstrap en SPICE (Software Proces Improvement and Capability Determinations).

6.4 Personal Software Process (PSP)

Personal Software Process (PSP) is een persoonlijke versie van CMM. Met PSP gaat men er vanuit dat wanneer een organisatie verder dan CMM level 3 wil komen, individuele teamleden software verbeteringsprocessen moeten implementeren op een persoonlijk niveau [22]. Humphrey [23] geeft aan dat een organisatie gedisciplineerd en efficiënt moet zijn, wil PSP geïmplementeerd kunnen worden. Daarom is het verstandig als organisaties op CMM level 2 of hoger zitten, voordat ze aan PSP beginnen. Bij PSP moet de ontwikkelaar eerst leren om de ontwikkeltijd, ratio van fouten creatie en ratio van foutoplossing te meten in de *Personal Measurement* (PSP0) fase. Vervolgens wordt die data gebruikt om grootte en ontwikkel tijd van nieuwe programma's in te schatten in de *Personal planning* (PSP1) fase. Voor de *Personal Quality* (PSP2) wordt ervoor gezorgd dat de focus op kwaliteit groter wordt, in het bijzonder tijdens het begin van het project. Als laatste fase van de cyclus moet er in de *Scaling Up* (PSP3) fase voor gezorgd worden dat deze metingen verbeterd worden.

6.5 Kwaliteit standaarden voor een software bedrijf

Stevenson en Barnes [18] geven aan dat het invoeren van de ISO 9000 standaarden alleen zinvol is voor bedrijven die hier marktaandeel en klanten mee winnen. Ze geven aan dat het anders veel tijd en energie kost en dat de winst die te behalen valt, pas na langere tijd zichtbaar wordt. Als belangrijkste voordelen van het certificeren geven ze verbetering van documentatie, bedrijfsstandaarden, bewustwording van kwaliteit en behouden van een markt aandeel. Er zal dus marktonderzoek gedaan moeten worden, om te kijken of potentiële klanten gebruik maken van ISO normen en dit ook van hun leveranciers verwachten.

Voor software zijn de ISO 9126 standaarden waarschijnlijk meer geschikt. Deze standaard richt zich op de belangrijkste punten van software, hierdoor worden andere risico's benadrukt dan

bij fysieke producten. Zoals eerder al is aangegeven, zit er volgens Jung et al [19; 20] in deze standaard echter nog veel ambiguïteit en is de standaard daarom nog niet wetenschappelijk verantwoord. Daarom is het misschien verstandiger alleen de punten te pakken die geschikt zijn voor het bedrijf en het product dat dit bedrijf maakt. Op deze manier wordt voorkomen dat een enorme papierwinkel wordt opgestart, maar wordt wel de kwaliteit van het product verbeterd.

De ISO 9000 en ISO 9126 standaarden lijken op elkaar en vallen in dezelfde categorie, namelijk product kwaliteit. CMM en PSP vallen in de andere categorie, die van proces kwaliteit. Het is voor een software bedrijf wel belangrijk om op te schrijven hoe processen lopen. Op deze manier kunnen er geen fouten ontstaan, doordat stappen vergeten zijn. Door betere controle van het proces wordt het product meestal ook beter. Het is alleen niet duidelijk of hiervoor een officiële standaard, zoals CMM en PSP nodig is. Het belangrijkste is dat er nagedacht is over de bedrijfsprocessen. Wanneer er een manier gekozen is, dan is het ook belangrijk dat deze vervolgens ook nageleefd wordt. Hierbij moet men blijven evalueren of de gekozen manier wel de beste is en of andere manieren misschien kwaliteitsverbetering gehaald kan worden. Hierbij geldt ook dat het officieel laten keuren alleen zin heeft als hiervoor specifieke voordelen voor dat bedrijf zijn, zoals een grotere bron van potentiële klanten.

7 Protocollen

Om de kwaliteit van de elektronische bouwtekening te verbeteren, kunnen mechanismes ontwikkeld worden die ervoor zorgen dat fouten voorkomen worden. Er kunnen hiervoor invoerprotocollen geïntroduceerd worden. Een protocol is een gedragsvoorschrift, meestal in de vorm van een aantal uit te voeren stappen [9]. In dit onderzoek zullen protocollen gebruikt worden als gedragsvoorschrift over de manier waarop een elektronische bouwtekening gevuld wordt. Doordat beperkingen gelegd worden op de inhoud van de bouwtekening, worden de foutsituaties zo vroeg mogelijk opgespoord. We kunnen hierbij onderscheid maken tussen voorschiften die gedrag verplichten (moet) en voorschiften die gedrag uitsluiten (mag niet). Omdat Thinkwise tools maakt die het ontwikkelproces verbeteren, is het uiteindelijke doel om de protocollen te automatiseren.

In Figuur 4 op bladzijde 22 is de architectuur van de elektronische bouwtekening weergegeven. De oranje gekleurde objecten worden tijdens elk ontwikkelproces specifiek gebouwd. Voor deze objecten kan mogelijk een protocol gemaakt worden. Dit zijn Meta Solution Definition, Code templates en Control procedures en de GUI.

7.1 Meta Solution Definition

De protocollen van de Meta Solution Definition (MSD) zijn op te splitsen in vier niveaus. Van elk niveau kan weergegeven worden op welke manier de protocollen opgezet worden. Per protocol zal vervolgens een voorbeeld weergegeven worden, waarin het protocol toegepast kan worden, zodat gedrag verplicht of uitgesloten wordt. Het doel van dit onderzoek is niet om een uitputtende lijst met protocollen te geven. Door de protocollen op te delen in niveaus en aan te geven hoe ze op deze niveaus toegepast kunnen worden, kan in de praktijk geconstateerd worden welke protocollen nuttig zijn en kunnen deze protocollen tijdens projecten uitgebreid worden, waar nodig blijkt te zijn.

7.1.1 Model protocollen

De MSD is een model dat opgeslagen wordt in een database en bereikt kan worden met behulp van tooling. De ERD modeler is een tool die gebruikt wordt om het model grafisch weer te geven. Een Entity Relation Diagram (ERD) geeft de entiteiten en haar relaties weer. Met behulp van de ERD modeler is een ERD model gemaakt van de Software Fabriek. Om de leesbaarheid te verhogen is dat model, het metamodel, opgesplitst in vier modellen. Het datamodel is een van de modellen. Het datamodel representeert de tabellen die gebruikt worden om de gegevensstructuur van een klantdatabase in op te slaan, zoals *tab*, *col*, *ref*, *dom*, etc. In de tabel *tab* worden dus alle tabellen van een klantdatabase opgeslagen. Door beperkingen te leggen op de tabel *tab*, worden er beperkingen gelegd op de invoer van de meta modellen van klantprojecten. Een verder beschrijving van dit concept is beschreven in subhoofdstuk 3.3.3 over Meta modelleren.

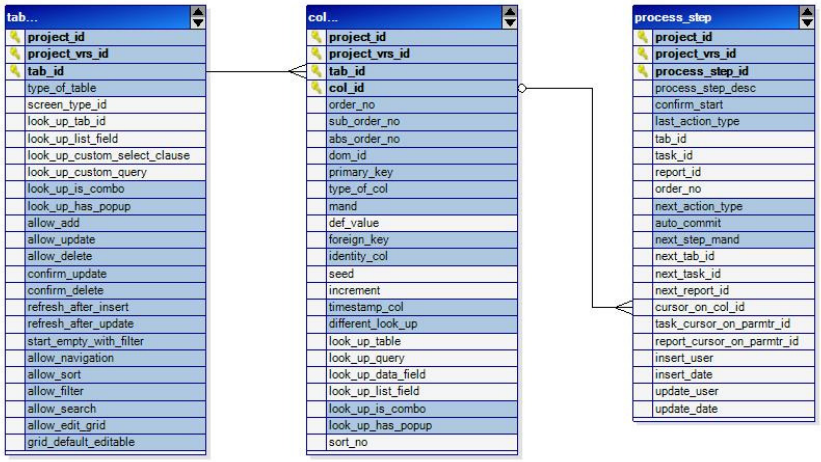
Er zijn verschillende beperkingen die met behulp van het ERD model weergegeven kunnen worden en die op de invoer van het onderliggende niveau spelen. Elke tabel heeft een primaire sleutel. Deze geeft aan dat een combinatie van velden uniek moeten zijn. Zo kan bijvoorbeeld ervoor gezorgd worden dat er niet twee tabellen met dezelfde naam in een project zitten. De primaire sleutel ligt op project_id, project_vrs_id en tab_id, zodat elke tabelnaam uniek is binnen een versie van een project.

In de ERD modeler wordt ook weergegeven welke velden verplicht zijn en welke velden optioneel. Dit wordt aangegeven door de achtergrondkleur van de velden. Verplichte (mandatory) velden zijn gekleurd, terwijl optionele velden wit zijn.

De relaties tussen de tabellen wordt weergegeven door de referenties. Een referentie in een ERD is een relatie tussen twee tabellen. Aan allebei de kanten kan de betrokkenheid van de relatie weergegeven worden. Er zijn verschillende type betrokkenheden:

- 0 of meer, dit wordt weergegeven met een open rondje en drie pootjes.
- 0 of 1, dit wordt weergegeven met een open rondje
- precies 1, dit wordt weergegeven met een rechte lijn
- 1 of meer, dit wordt weergegeven met drie pootjes

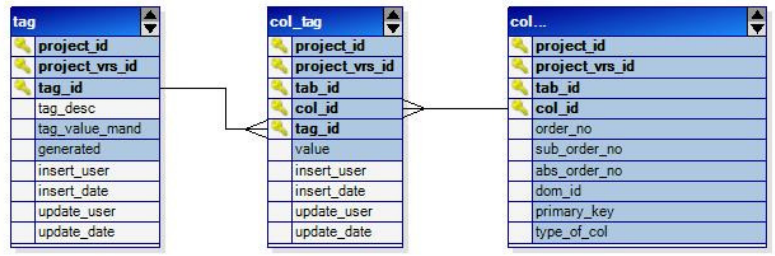
Bij een 0 of meer relatie of een 1 of meer relatie wordt de primaire sleutel automatisch bij de velden van die tabel toegevoegd, dit wordt het foreign key veld genoemd. Als de relatie een 0 of 1 of een 0 of meer relatie is, betekent dit dat het veld waartussen de relatie ligt, niet verplicht is.



Figuur 10: Verschillende relaties

Een voorbeeld is de referentie tussen *tab* en *col*, zichtbaar in Figuur 10. Een tabel kan een of meerdere kolommen hebben en elke kolom hoort bij precies 1 tabel. Bij de kolom wordt daarom ook verplicht weergegeven bij welke tabel deze hoort. De relatie tussen processtap en kolom is optioneel. Bij een processtap kan de cursor in maximaal 1 kolom gezet worden of in helemaal geen kolom (0 of 1). Een kolom kan in verschillende processtappen gebruikt worden.

Door deze manier van modelleren is het niet mogelijk om een veel op veel relatie tussen twee tabellen te hebben. Een veel op veel relatie resulteert namelijk uiteindelijk op fysiek niveau in een koppeltabel, die de twee tabellen aan elkaar koppelt. Een koppeltabel is weergegeven in Figuur 11. *Tag* en *col* hebben een koppeltabel (*col_tag*) waarin bijgehouden wordt wat de waarde is voor een tag die bij een bepaalde kolom hoort.



Figuur 11: Koppeltabel tussen tag en col

7.1.2 Database protocollen

Omdat de MSD opgeslagen is in een database, kunnen de mogelijkheden die databases brengen om beperkingen op te leggen ook gebruikt worden. Deze beperkingen worden op het datamodel gelegd, waarvan ook het ERD diagram gemaakt is. De beschreven beperkingen op model niveau kunnen vertaald worden naar database beperkingen.

Op velden in een tabel kunnen in de database constraints gelegd worden. Zo kunnen de minimale en maximale waarde op databaseniveau vastgelegd worden. Daarnaast kan opgegeven worden dat een kolom een identity kolom is. Dan wordt de kolom automatisch gevuld met een nummer, zodat deze uniek wordt en te identificeren is. Ook kan een standaard waarde ingevoerd worden, deze waarde wordt gebruikt om het veld te vullen, als er geen waarde opgegeven is. Een default waarde kan bijvoorbeeld gebruikt worden bij type_of_table in de tabel *tab*. Standaard wordt in *tab* een tabel opgeslagen, maar in *tab* kunnen ook views en grafieken opgeslagen worden. Wanneer niets aangegeven wordt, dan zal dit van type tabel zijn.

Naast constraints kan er in databases ook gebruik gemaakt worden van triggers. Triggers zijn stukken code die horen bij een bepaalde tabel en die uitgevoerd worden wanneer de data in die tabel muteert. Zo zijn er triggers voor insert (toevoegen), update (wijzigen) en delete (verwijderen). Triggers kunnen gebruikt worden om achteraf, nadat de gebruiker de mutaties opgeslagen heeft, aanpassingen te maken in de data. Er zijn ook triggers die voor de mutaties afgaan. Een voorbeeld van het gebruik van een after trigger is het veld dat het aantal kolommen in een tabel bijgehouden wordt. Wanneer een kolom toegevoegd of verwijderd wordt, wordt dat veld in *tab* ook bijgewerkt. Ook de trace velden, die bijhouden door wie en wanneer een veld gewijzigd is, worden door triggers onderhouden.

7.1.3 SF protocollen

Omdat de MSD zelf ook met de software fabriek gemaakt is, kunnen alle manieren die de software fabriek heeft om beperkingen op te leggen ook gebruikt worden voor het opstellen van protocollen.

Layouts procedure

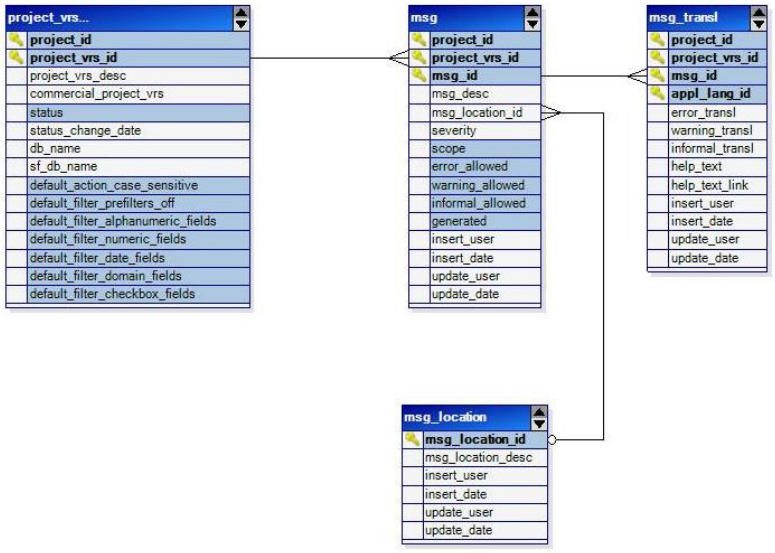
Als een gebruiker aan het muteren is, dan controleert een layout procedure elke keer als een veld verlaten wordt of dit invloed heeft gehad op de layout van het scherm en verandert zo nodig de opmaak van het scherm. Op deze manier kan in de layout procedure aangegeven worden dat als een veld ingevuld wordt, een ander veld verplicht moet worden. Een voorbeeld ervan is te vinden bij de opmaakgegevens van een kolom. Van elke kolom kan opgegeven

worden of deze in de eindapplicatie als veld direct na de vorige kolom geplaatst wordt. De kolom kan echter ook op de volgende rij of zelfs het volgende tabblad weergegeven. Als aangegeven wordt dat deze kolom op een speciaal nieuw tabblad moet komen, dan wordt er dankzij de layout een label zichtbaar en verplicht. Op dit label moet de naam van het volgende tabblad ingevuld worden.

Naast dat een veld verplicht gemaakt wordt, kan deze ook verborgen of alleen lezen worden. Dit wordt in de software fabriek bijvoorbeeld gedaan bij het opgeven van kolommen. Hier kan met een checkbox aangegeven worden of een kolom een identity kolom is. Een identity kolom wordt in de database automatisch gevuld met een nummer. Dit nummer wordt bij elk record automatisch opgehoogd. Een kolom waar al automatisch een nummer in komt, mag geen default waarde hebben. Het veld waarin de default waarde gevuld wordt, wordt daarom ook verborgen, als de identity checkbox aangevinkt wordt. Andersom geldt ook dat het veld seed en incremental verborgen blijven totdat een kolom een identity kolom wordt. Met seed kan opgegeven worden wat het eerste nummer is (meestal 1) en met incremental kan weergegeven worden met hoeveel het getal telkens ophoogt (bijvoorbeeld 1, 3, 5, 7, …). Deze twee velden zijn enkel van toepassing bij identity kolommen en worden daarom bij niet-identity kolommen verborgen.

Defaults procedure

Wanneer een gebruiker het veld verlaat, gaat er ook een default procedure af. Deze default controleert of er door bepaalde wijzigingen, andere velden automatisch gevuld moeten worden en vult vervolgens deze velden. Dit wordt in de SF gebruikt bij referenties. De naam van een referentie is opgebouwd uit de brontabel, de doeltabel en eventueel een toevoeging. Als een nieuwe referentie wordt aangemaakt en de tabelnamen worden gevuld, dan wordt ook de referentienaam gevuld. Een ander voorbeeld is bij een kolom, als deze een primaire sleutel is, moet het veld wel verplicht zijn. Daarom wordt de checkbox of de kolom verplicht is, automatisch aangevinkt (en door de layout procedure alleen lezen gemaakt).



Figuur 12: ERD diagram van meldingen

Processtappen

Processtappen richten zich op de manier waarop gebruikers de gegevens invullen. Doordat gebruikers gestuurd worden in het doorlopen van het proces, zullen ze minder snel gegevens vergeten in te geven. Een processtap kan bijvoorbeeld toegepast worden bij het ingeven van meldingen. Zoals in het ERD diagram van Figuur 12 te zien is, wordt naast een melding, ook een vertaling van deze melding weergegeven. De melding is vaak een korte term die de ontwikkelaar helpt om de verschillende meldingen te herkennen. De melding kan meerdere keren in code gebruikt worden, terwijl hier in de database maar één keer een

gebruikersvriendelijke melding opgegeven hoeft te worden. Een voorbeeld van een melding is *erd_confirmdeletediagram*. Wanneer deze melding voorkomt, moet voor de gebruiker een duidelijke vraag gesteld worden, zoals “Weet u zeker dat u dit diagram wilt verwijderen?”. Hetzelfde kan vervolgens in een andere taal toegevoegd worden. Met een processtap wordt de ontwikkelaar gestuurd, zodat als een melding toegevoegd wordt, daarna een tabblad weergegeven waarop een meldingsvertaling moet worden toegevoegd. Want zonder een vertaling krijgt de eindgebruiker een korte term, waarvan niet altijd duidelijk is hoe de melding verholpen kan worden.

Specifieke (unieke) schermen

Met processtappen kan een gebruiker gestuurd worden in de stappen die doorlopen moeten worden, maar er kunnen geen stappen uitgesloten worden. Met behulp van specifieke schermen kunnen beperkingen gemaakt worden die speciaal aansluiten op specifieke eisen. Het gebruik van specifieke schermen als protocol zal niet vaak gebruikt worden, omdat het meer kost om deze schermen te ontwikkelen en deze schermen vervolgens minder onderhoudbaar zijn dan standaard schermen. De specifieke schermen worden meestal niet gebruikt om beperkingen af te dwingen, maar vooral om het proces en de gebruikersvriendelijkheid te bevorderen. Zo kunnen schermen volledige voor een bepaalde functionaliteit geoptimaliseerd worden. Bijvoorbeeld voor verkopers die de hele dag verkooporders in moeten voeren.

7.1.4 Projectafhankelijke protocollen

De vorige niveaus waren protocollen die gelden voor alle projecten. Er zijn ook regels of afspraken die per project gemaakt worden. Sommige regels worden met de klant afgesproken, andere worden automatisch van elkaar overgenomen en vormen stilzwijgend de werkwijze waarop activiteiten ondernomen worden.

Dit is geen probleem zolang de teamsamenstelling blijft zoals het is. Zodra er andere mensen bijkomen die niet van deze afspraken weten of wanneer er na lifegang onderhoud gepleegd moet worden, kan het maken en vastleggen van deze afspraken fouten voorkomen. Een aantal voorbeelden van afspraken die gemaakt kunnen worden zijn:

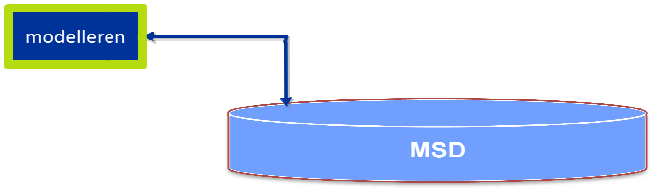
- Elke tabel heeft onzichtbare trace kolommen, genaamd …
- De eerste velden van een tabel zijn altijd de PK-velden
- Als het domein van control checkbox is, dan is het veld altijd verplicht en moet een default gevuld worden.
- Koppeltabellen hebben altijd de naam <tabel1> _ <tabel2>
- Als een veld een identifier en een PK is, dan heet dat veld altijd <tabelnaam> + “_id”
- Alle tabbladen bij een referentie staan uit, tenzij duidelijk is dat het aan moet
- Alle velden zijn verplicht, tenzij duidelijk is dat ze niet verplicht zijn.
- Bij een melding altijd parameters gebruiken
- Bij een melding altijd de Nederlandse versie vullen
- Issues worden gemeld in…
- Specificaties staan beschreven in…
- Alle naamgeving is in het Nederlands
- Elke tabel en kolom begint met een hoofdletter en daarna kleine letters

- Views en enkel views, beginnen met “vw_”
- Velden met een dossier_id en ubdossier_id worden altijd weergegeven op 1 regel.
- Velden met een week en jaar worden altijd weergegeven op 1 regel.
- Velden met aantal links, aantal rechts, aantal totaal geven alleen het label van aantal totaal weer.

Sommige van deze protocollen kunnen wel geïmplementeerd worden met de eerder beschreven technieken, maar omdat ze projectafhankelijk zijn gaan ze niet altijd op, terwijl deze protollen dit wel afdwingen. De project specifieke protocollen zouden per project in een document bijgehouden moeten worden. Bij de start van een nieuw project, zal van een standaard document overgenomen moeten worden welke instellingen en protocollen voor dat project van toepassing zijn. Verder moet gekeken worden hoe dat standaard document uitgebreid kan worden. In een stap verder kunnen de protocollen vastgelegd worden in het MSD. Deze protocollen kunnen in een nieuwe tabel opgeslagen worden. Vervolgens kunnen er triggers geschreven worden die afhankelijk van de tabelinhoud controleren of een record dat gemuteerd wordt aan de eisen van dat project voldoet. Uiteindelijk in de laatste stap kunnen bepaalde tags in een nieuwe versie van de Meta-MSD toegevoegd worden aan bepaalde tabellen, zodat deze verplicht gevuld moeten worden en de triggers hiernaar kunnen kijken voor controles.

7.1.5 Resultaat

De software fabriek heeft door de beschreven technieken een voordeel op gebied van kwaliteit tegenover traditionele software ontwikkeling. Door het gebruik van een elektronische bouwtekening (MSD) kan tijdens het muteren (het modelleren) eisen gesteld worden aan hetgeen dat in de bouwtekening opgeslagen wordt.



Figuur 13: Kwaliteitproces I

Het begin van het kwaliteitsproces wordt weergegeven in Figuur 13. Deze zal in de volgende hoofdstukken verder opgebouwd worden. In Figuur 13 wordt getoond dat het MSD gebruikt wordt om te modelleren, het resultaat hiervan is een verbeterd MSD. Met een groen rand is aangegeven dat in het proces kwaliteitsverbetering bewerkstelligd wordt (door middel van protocollen).

7.2 Code templates en Control procedures

Control procedures en code templates bestaan uit code. Dit kan code in elke taal zijn, maar nu wordt vooral SQL code gebruikt. Protocollen op het gebied van code zouden gerealiseerd kunnen worden door een stijl document. In dit document wordt opgeschreven hoe de code eruit zou moeten zien. Bijvoorbeeld, elke kolom uit de select-clause komt op een nieuwe regel.

Het nadeel van een stijl document is dat deze normaal gesproken bestaat uit tekst. Ontwikkelaars moeten dit document lezen en vervolgens zelf ervoor zorgen dat de code er uit

ziet, zoals in het document beschreven staat. Daarom is gekeken of de opmaak van de code niet automatisch aangepast kan worden aan de afgesproken regels. Hiervoor kan de parser gebruikt worden die bij de impact analyse gebruikt is. Deze wordt verder beschreven in hoofdstuk 0. Deze parser kan naast het parsen de tekst ook opmaken (dit wordt pretty print genoemd). Door het instellen van verschillende parameters, wordt de tekst volgens verschillende regels opgemaakt. Zo kan besloten worden dat regels maximaal 200 karakters breed mag zijn.

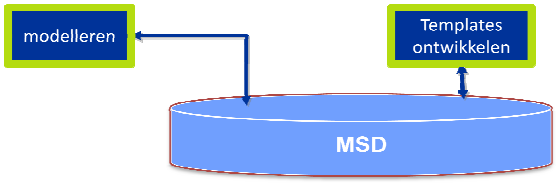
Het zou mooi zijn wanneer elk project dezelfde stijl hanteert. Als alle code automatische opgemaakt wordt bij het opslaan, zodat deze voldoet aan de code. Dit is echter niet mogelijk om twee redenen. Ten eerste stellen verschillende klanten verschillende eisen aan hun code. Zelfs verschillende databases (zoals Microsoft SQL server en DB2) verwachten andere type constructies. Daarom zou het verstandiger zijn als de opmaak instelbaar is. Wanneer dit in de software fabriek gedefinieerd wordt, kan het per project aangepast worden aan de wensen van de klant. Zo is de code altijd hetzelfde voor het hele project.

De tweede reden waarom niet elk project automatisch in dezelfde stijl opgeslagen kan worden, is omdat de code niet altijd syntaxis correct is. Hier zal tijdens de impact analyse verder op in gegaan worden. Nu volstaat om te vertellen dat de code tijdens het genereerproces aangevuld wordt zodat deze correct wordt. Wanneer deze code voor het genereerproces al opgemaakt wordt, kan dit voor fouten zorgen tijdens het genereerproces. Deze stukken code moeten dus bewust niet gewijzigd worden. Daarom is ervoor gekozen het opmaken van code wel met de parser te laten doen, maar de ontwikkelaar moet dit proces per template of control procedure zelf in gang zetten. Zo kan door de ontwikkelaar gekozen worden om het opmaken bij sommige stukken code wel te doen en bij anderen niet.

Het automatisch opmaken van een template zal daarom moeten gebeuren in de functionality modeler. Dit is een tool waarmee de templates en control procedures beheerd kunnen worden. Hierin komt een knop waarmee de opmaak gewijzigd wordt, vervolgens kan de ontwikkelaar beslissen of deze opmaak opgeslagen moet worden. Later zal het mogelijk moeten worden om de template in de functionality modeler te kunnen wijzigen. Het is wel gepland om dit te ontwikkelen, maar niet binnen dit project.

7.2.1 Resultaat

In **Fout! Verwijzingsbron niet gevonden.** wordt weergegeven hoe het kwaliteitsproces itgebreid wordt. Door protocollen op de templates te ontwikkelen, wordt de kwaliteit van de templates verbeterd en gemanaged. Hierbij geldt net als bij het modelleren, dat het MSD gebruikt wordt in het proces om templates te ontwikkelen en het resultaat is een aangepaste MSD.



Figuur 14: Kwaliteitsproces II

7.3 GUI

Naast de code en de elektronische bouwtekening wordt er ook een Grafische User Interface (GUI) gebruikt in een eindproduct. Deze GUI wordt telkens verder ontwikkeld, zodat deze steeds meer functionaliteit omvat. Er zouden ook protocollen bedacht kunnen worden voor het ontwikkelen aan de GUI. Dit is sterk te vergelijken met traditionele applicaties. Er zijn echter geen protocollen bedacht voor de GUI. Deze maakt al gebruik van componenten om te zorgen dat bepaalde brokken functionaliteit altijd hetzelfde werken. Zo werken bijvoorbeeld alle formulieren hetzelfde, omdat er altijd gebruik gemaakt wordt van hetzelfde component (TSFForm).

Ook het opmaken van code kan niet gebruikt worden als protocol. In de gebruikte programmeeromgevingen wordt de code al automatisch opgemaakt. Bijvoorbeeld in Visual Studio, waar de .NET GUI in gemaakt wordt. Hier wordt bij elk sluitaccolade (}) het blok wat tussen de accolades staat opgemaakt.

8 Impact analyse in de literatuur

Wanneer de eisen van de gebruikers veranderen, dan kan dit grote gevolgen hebben voor het systeem. Als deze gevolgen niet goed doorgevoerd worden, dan verslechtert dit de kwaliteit van het systeem. Dit is een probleem dat niet alleen bij Thinkwise, maar ook bij traditioneel ontwikkelen kan spelen. Daarom is gekeken naar hoe dit in de literatuur beschreven wordt en hoe het opgelost kan worden.

8.1 Change Impact Analyse

Om fouten zo vroeg mogelijk op te sporen, kan tussen versies impact analyse gedaan worden. Deze analyse wordt in literatuur vaak aangeduid als *Change impact analysis*. Om aan te geven wanneer impact analyse specifiek voor Thinkwise bedoeld wordt, zal deze afgekort worden als CIA (Thinkwise Change Impact Analyse).

8.1.1 Software evolutie

Bohner [24] geeft aan dat als software systemen groeien in grootte en complexiteit, het steeds moeilijker voor ontwikkelaars wordt om het systeem te begrijpen. Software slijt niet en vertoont geen ouderdomsverschijnselen, zoals dit bij fysieke producten het geval is. Wel worden er telkens nieuwe technologieën ontwikkeld. De eisen van gebruiker veranderen, zodat software niet meer voldoet. Software onderhoud zorgt ervoor dat het aan de nieuwe eisen voldoet, maar de veranderingen zorgen er ook voor dat de kwaliteit van de software verslechtert, tenzij het proces actief gecontroleerd wordt.

Door het generieke karakter van de software fabriek is het mogelijk al snel een prototype weer te geven, nog voordat het datamodel helemaal klaar is. Nadat een prototype gemaakt is, worden aanpassingen en verdere ontwikkelingen in een nieuwe versie van het MSD gemaakt. Gebruikers die met het prototype werken, ontdekken dat er nieuwe en gewijzigde eisen zijn. Zonder het prototype hadden ze deze nieuwe eisen niet kunnen bedenken. Door de nieuwe eisen moet het datamodel en de bijbehorende functionaliteit die al geschreven is, gewijzigd worden. Dit is te vergelijken met het onderhoud van traditionele software. Bij traditionele software komen gebruikers er ook achter dat bepaalde functionaliteit niet (meer) geschikt is om de business te ondersteunen en moet de software vervolgens aangepast worden. Het verschil is echter dat gebruikers nog niet met de software werken en de complete business data meestal nog niet aan de applicatie toegevoegd is.

Bij fysieke producten slijt het product en is het daarom effectiever om een nieuw product te kopen. Omdat software niet zo slijt, wordt het bestaande ontwerp aangepast en ontstaat er software evolutie. Volgens Hewitt en Rilling [25] zijn de veranderingen van gebruikerseisen en van technologie de grootste factoren die van invloed zijn op de software evolutie. Li en Offutt [26] definiëren software evolutie dan ook als de continue ontwikkeling en onderhoud aan een bestaand software systeem.

8.1.2 Definitie change impact analyse

Li en Offutt geven aan dat de focus verschoven is van ontwikkelen naar onderhouden in evoluerende software. Omdat de focus verschoven is, hebben software ontwikkelaars een mechanisme nodig om aan te geven wat het effect is van bepaalde veranderingen. Li en Offutt geven aan dat met change impact analysis de analyse bedoeld wordt die nagaat hoe mogelijke

wijzigingen het systeem kunnen beïnvloeden. Behalve voor software ontwikkelaars is deze informatie ook waardevol voor regressie testen. Alleen de delen die gewijzigd zijn en de delen waar de wijzigingen impact op hebben, moeten opnieuw getest worden. Projectmanagers kunnen met een goede impact analyse beter inschatten hoelang een wijziging gaat duren en hoeveel het gaat kosten. [26]

Naast dat van tevoren de impact berekend wordt, zodat dit de keuze en planning verbetert, kan ook achteraf de change impact analyse gebruikt worden om de kwaliteit te verbeteren (of, zoals eerder gezegd, ervoor te zorgen dat de kwaliteit door onderhoud niet verslechtert). Doordat de systemen bij Thinkwise iteratief groeien, kunnen de mogelijke effecten van wijzigingen moeilijk te overzien zijn. Thinkwise wil graag change impact analyse kunnen uitvoeren op projecten om te kijken welke impact wijzigingen in het MSD hebben op de bestaande business functionaliteit. Door de functionaliteit op te sporen waarop een wijziging mogelijk effect heeft, kan deze functionaliteit zo vroeg mogelijk aangepast worden. Fouten, die zouden ontstaan doordat een wijziging een onverwacht effect heeft op een andere deel van de code, worden hiermee eerder geïdentificeerd.

Bose [27] definieert twee problemen, die ontstaan als men kijkt naar wijzigingen in de architectuur van sofware, namelijk evalueren van de wijzigingen (om te ontdekken welke eigenschappen wijzigen en wat de knelpunten zijn) en het managen van de wijzigingen. Met change impact analyse wordt vooral gekeken naar het eerste probleem. Bohner definieert software change impact analyse als:

"The determination of potential effects to a subject system resulting from a proposed software change."[24]

Hij legt uit dat het basisprincipe van change impact analyse is dat een klein deel van de software aangepast wordt, wat vervolgens effect heeft op veel andere delen van het systeem, zodat een rimpeleffect ontstaat. Hij maakt onderscheid tussen direct en indirecte impact. Twee objecten hebben een directe impact als het object dat gewijzigd wordt een relatie heeft met het andere object. Bij indirecte impact ligt er een pad tussen de twee objecten. Dit pad maakt gebruik van andere objecten. Het type impact is een N-level impact, waarbij N staat voor het aantal relaties tussen de twee objecten.

Figuur 15: 6-level impact relatie

De impact analyse voor Thinkwise richt zich alleen op de wijzigingen in database functionaliteit, daarom is indirecte impact niet van toepassing. Bij indirecte impact wijzigt een component, deze wijziging heeft vervolgens impact op een tweede component. Dit tweede component heeft weer impact op een volgende component. Voor Thinkwise is alleen de impact van de wijziging op de database functionaliteit van belang. Als deze wijziging zou zorgen voor meer veranderingen in het model, dan worden deze veranderingen als nieuwe wijzigingen gezien, die in een nieuwe impact analyse naar voren komt.

8.1.3 Niveau van impact analyse

In de literatuur wordt onderscheid gemaakt tussen impact analyse gebaseerd op broncode en impact analyse gebaseerd op modellen. Het voordeel van impact analyse op broncode niveau is dat het erg nauwkeurig is. Impact analyse op niveau van model is niet zo nauwkeurig, maar heeft als voordeel dat het minder tijd kost en eerder in het ontwikkelproces zitten. [25; 28] Impact analyse op gebied van model kan gedaan worden, voordat de wijzigingen daadwerkelijk in de code doorgevoerd zijn. Zo kan een wijziging makkelijker teruggedraaid worden als de impact te groot is.

Bij de CIA wordt zowel gekeken naar het abstracte niveau van modellen, als naar het specifieke niveau van broncode. De wijzigingen worden in het model bekeken, de impact hiervan wordt vervolgens in de business functionaliteit (op broncode level) geanalyseerd, waardoor het nauwkeurig blijft.

8.2 Methodes van change impact analyse

Er zijn verschillende onderzoeken gedaan naar change impact analyse. Deze onderzoeken belichten telkens een ander techniek voor het ontwikkelen van de software. Zo is er gekeken naar UML, OO, CBS, SOBA en UCM. Hieronder is voor elk punt beschreven wat gebruikt kan worden bij Thinkwise impact analyse en wat juist anders en niet geschikt is.

8.2.1 Impact analyse in UML

Braind et al [28], die impact analyse op het gebied van UML onderzoeken, delen het werk op in een aantal stappen. Eerst wordt de verschillen tussen twee modellen opgezocht (de wijzigingen). Daarna wordt gekeken of de nieuwe diagrammen onderling consistent zijn. Vervolgens wordt gekeken wat de impact van de wijzigingen is en als laatste worden de resultaten gesorteerd op prioriteit. De wijzigingen zijn opgedeeld in 97 categorieën, die weergegeven worden in een *change taxonomy*. Vervolgens worden er *impact analysis rules* gebruikt om automatisch de analyse uit te voeren. Voor elke categorie bestaat een regel, zodat er 97 regels bestaan. Om deze methode te ondersteunen is er een prototype tool gemaakt, genaamd iACMTool.

Het interessante aan deze methode is dat eerst de wijzigingen opgezocht worden en daarna gecontroleerd wordt of de onderlinge diagrammen consistent zijn. Bij Thinkwise is het mogelijk om te controleren of het MSD consistent is met behulp van validatie tools. Dit staat los van de CIA en kan op elk willekeurig moment uitgevoerd worden. Voordat er een nieuw prototype gemaakt wordt, moet het MSD in elk geval valide zijn. Ook het opzoeken van wijzigingen is al voor een deel geïmplementeerd. Dit wordt gebruikt om, bij het ontwikkelen van een volgend prototype, de gewijzigde tabellen en kolommen in de database aan te passen. Braind et al eindigen met een sortering van het eindresultaten. Een sortering naar belangrijkheid is bij CIA niet mogelijk, omdat alle resultaten belangrijk zijn. Alle objecten waar de wijzigingen effect op hebben, kunnen fouten introduceren, waardoor niet mogelijk is om de ene 'fout' boven de andere te plaatsen. Door het resultaat op te delen in een aantal categorieën, wordt het voor een ontwikkelaar wel makkelijker om de impact analyse te verwerken en dus de mogelijke fouten per categorie op te lossen.

8.2.2 Impact analyse in object georiënteerde software

Li en Offutt [26] kijken naar object georiënteerde (OO) software. Een object georiënteerd systeem bestaat uit objecten en classes. Een object is opgedeeld in een reeks eigenschappen (properties) en een reeks handelingen (operations). Vervolgens worden er vier algoritmes beschreven, die de impact van een bepaalde wijziging op het systeem uitrekenen. De algoritmes analyseren de rimpeleffecten door het systeem en geven vervolgens aan welke classes met bijbehorende methodes en gegevens beïnvloed zijn. De algoritmes zoeken alle classes die nog niet gecontroleerd zijn en kijken of deze een relatie hebben met de gewijzigde klasse.

Het interessante aan deze techniek is dat alle code op verschillende manieren nagelopen wordt, om te kijken waar de wijzigingen effect hebben. De technieken waarmee alle classes langs gelopen worden is erg OO-specifiek, maar het idee kan in CIA ook gebruikt worden. Van tevoren is niet bekend welke tabellen in welke functionaliteit gebruikt wordt en daarom zal alle functionaliteit met een algoritme nagelopen moeten worden.

8.2.3 Impact analyse in Component based software

Mao, Zhang en Lu [29] kijken naar change impact analysis voor Component Based Software (CBS). Ze maken gebruik van een matrix om de relaties tussen verschillende componenten weer te geven. Vervolgens rekenen ze uit hoe componenten te bereiken zijn, bijvoorbeeld door via andere componenten te gaan. Aan de hand van een *change ratio* van het component en de *reachability matrix*, berekenen ze de change ratio van het hele systeem. In de berekening maken Mao et al onderscheid tussen wijzigingen in één component en wijzigingen in meerdere componenten.

In de literatuur wordt vaak een voorbeeld gegeven dat men met change impact analyse een keuze wil maken tussen verschillende implementaties. Na berekenen van de impact van beide implementaties, wordt de implementatie met de minste impact gekozen. Dit is een goed voorbeeld van iets dat ***niet*** met de impact analyse van Thinkwise uitgevoerd zal worden. Thinkwise wil graag alle elementen die impact hebben, identificeren. Zodat deze allemaal nagelopen kunnen worden op mogelijke wijzigingen en bugs.

8.2.4 Impact analyse in Service Oriented Business applicaties

Xiao, Guo en Zou [30] maken in hun onderzoek gebruik van Service Oriented Business applications. Business applicaties assisteren gebruikers, zodat ze hun dagelijks werk efficiënter en effectiever uit kunnen voeren. De talen om het bedrijfsproces te specificeren, zoals BPEL4WS kunnen gebruikt worden om verschillende services in een service georiënteerde bedrijfsapplicatie te integreren. Om de kwaliteit te verbeteren en de kosten te reduceren van producten of services, proberen bedrijven steeds hun processen te verbeteren door deze aan te passen. Wanneer bedrijfsprocessen aangepast worden, dan moet het systeem ook onderhouden worden, zodat deze aan de nieuwe eisen voldoet. Xiao et al proberen een methode te ontwikkelen waarbij snel een ruwe schatting gegeven kan worden van de impact en kosten, die wijzigingen van bedrijfsprocessen en de bijbehorende software systemen met zich meebrengen. Als een wijziging waarschijnlijk meer zal kosten dan dat het zal opbrengen, kan gekozen worden om de wijziging niet door te voeren. Xiao et al maken onderscheid tussen

twee niveaus van abstractie: change impact analyse op bedrijfsprocessen niveau en change impact analyse op broncode niveau. De methode is opgedeeld in zes stappen:

1. Identificeer de bedrijfsproces componenten die gewijzigd moeten worden (opgeslagen in de *business component impact set)*.
2. Gebruikt impact analyse om de set uit te breiden met alle bedrijfsproces componenten die aangepast moeten worden, omdat ze een relatie hebben met de te wijzigen componenten.
3. Gebruik de beschikbare links tussen bedrijfsproces componenten en broncode entiteiten om alle componenten uit de set te refereren aan de juiste broncode entiteiten.
4. Sla de broncode entiteiten op in de *source code impact set*.
5. Genereer een *propagation graph* voor elke broncode entiteit in de set. Deze graph geeft aan welke entiteiten aangeroepen worden door de broncode entiteit en welke entiteiten ervan overerven.
6. Bereken de change impact metric gebaseerd op de propagation graph.

Ook deze methode berekent uiteindelijk een ratio en geeft niet alle objecten aan waarop de wijziging invloed heeft. Wel laat deze methode zien, hoe het mogelijk is een koppeling te maken tussen het model niveau en broncode niveau. Het verschil met CIA is wel dat hier beschikbare links zijn om een één op één vertaling van model naar broncode te maken. Bij Thinkwise zijn die links er niet. Deze zullen uit de structuur van de MSD en de structuur van de code gehaald moeten worden.

In een voorbeeld geven Xiao et al aan dat deze methode vooral handig is wanneer een manager meerdere keuzes heeft om een proces te wijzigen. Er kan vervolgens gekeken worden welk proces de kleinste impact heeft en ruw geschat de minste tijd en kosten met zich mee zal brengen. In verder werk willen ze deze schatting verder onderzoeken, zodat ook complexere processen verwerkt kunnen worden.

8.2.5 Impact analyse in Use Case Maps

Hewitt en Rilling [25] presenteren een methode voor het identificeren van de impact die het wijzigen van eisen (requirements) heeft op specificatie (model) niveau. Om de methode uit te beelden, maken ze gebruik van Use Case Maps (UCM). Hewitt en Rilling maken onderscheid tussen *Dependency analysis*, dit wordt gebruikt om te beantwoorden welke en hoe objecten gerelateerd zijn aan elkaar en *Ripple effects*, dit wordt gebruikt om te kijken wat de potentiële effecten zijn van het doorvoeren van een verandering op de rest van het systeem. De afhankelijkheidsanalyse (dependency analysis) wordt opgedeeld in *scenario relationships and dependencies* en *component dependency*.

Dit onderzoek verschilt met CIA in de abstractie. Hewitt en Rilling kijken naar abstracte requirements, terwijl voor CIA op design of zelfs implementatie niveau naar de MSD gekeken wordt. Het rimpeleffect is, zoals eerder al beschreven, niet nuttig, omdat dit sneeuwbaleffect op database niveau niet doorrolt en niet onderzocht hoeft te worden. Van de afhankelijkheidsanalyse is de component analyse erg interessant, omdat de objecten uit de software fabriek ook als losse componenten gezien kunnen worden. Hewitt en Rilling maken onderscheid tussen forward en backward dependence. Componenten zijn forward dependence (afhankelijk) van een component C, als ze direct na C uitgevoerd worden en backward dependence als het component direct voor C uitgevoerd wordt. Deze termen kunnen in de

software fabriek ook gebruikt worden bij de triggers. Stel er is een tabel A met trigger T_a en tabel B met trigger T_b. Dan is A forward dependence van B, als T_b de tabel A gebruikt. A is backward dependence van B, als T_a de tabel B aanroept. Als B wijzigt, dan zijn alle triggers van B makkelijk te vinden, de forward dependence is daarom ook handmatig makkelijker te vinden. De backward dependence is lastiger, omdat alle triggers opgezocht moeten worden die een verwijzing naar B hebben. Hiermee kan een impact analyse een grote rol spelen.

8.3 Type mogelijke wijzigingen

Volgens Bose [27] zijn er drie type wijzingen die kunnen voor komen, namelijk:

a) toevoegen of verwijderen van nieuwe componenten,
b) toevoegen van nieuwe componenten die bestaande componenten verbeteren of uitbreiden en
c) nieuwe connecties tussen componenten of verandering van de banden van de connecties.

Ook Li en Offutt [26] stellen een lijst op met verschillende soorten wijzigingen, zoals wijziging van waarde, van type of van scope en toevoegen of verwijderen van een element. Door verschillende wijzigingen te identificeren, kan gekeken worden of de impact analyse alle belangrijke wijzigingen ondersteunt.

Hieronder worden de type wijzigingen die voor CIA geïdentificeerd zijn weergegeven:

- Wijzigen van een tabelnaam
- Verwijderen van een tabel
- Toevoegen van een kolom
- Wijzigen van een kolomnaam
- Verwijderen van een kolom
- Kolomeigenschappen zijn gewijzigd, zoals:
    - Primary key
    - Verplicht/ niet-verplicht
    - Defaultwaarde
    - Domein
    - Identity
- Wijzigen van een domein
- Verwijderen van een domein
- Toevoegen van domeinelementen
- Wijzigen van een domeinelement
- Verwijderen van een domeinelement
- Wijzigen van een taaknaam
- Verwijderen van een taak
- Toevoegen van een taakparameter
- Wijzigen van een taakparameter
- Verwijderen van een taakparameter

De hoofdonderdelen die gewijzigd kunnen worden, zijn tabellen, domeinen en taken. Als een hoofdonderdeel toegevoegd wordt, dan kan deze nog niet in bestaande functionaliteit gebruikt zijn, aangezien het helemaal nieuw is. Er hoeft dus ook geen impact analyse over plaats te vinden. Bij het wijzigen van deze hoofdonderdelen is vooral de naam van belang. Als bijvoorbeeld een tabel van naam verandert, moet overal waar deze tabel voorkomt de nieuwe naam gebruikt gaan worden. Voor domeinen is de naam niet alleen van belang, maar ook de lengte, precisie en datatype. Wanneer onderdelen verwijderd worden, maar ze worden nog wel gebruikt in functionaliteit, dan kan dit fouten opleveren, zodra de onderdelen aangesproken worden. Daarom moet geanalyseerd worden waar verwijderde hoofdonderdelen impact op hebben.

Onder de hoofdonderdelen hangen subonderdelen, die enkel belangrijk zijn in de context van een hoofdonderdeel. Voor tabellen, domeinen en taken zijn dit respectievelijk kolommen, domeinelementen en taakparameters. Voor deze subonderdelen geldt, dat wijzigen en verwijderen geanalyseerd moeten worden, om dezelfde redenen dat de hoofdonderdelen geanalyseerd worden. Het toevoegen van een subonderdeel kan, in tegenstelling tot bij de hoofdonderdelen, wel voor fouten zorgen in bestaande functionaliteit. Dit omdat subonderdelen voor komen in de context van een hoofdonderdeel. Dit geldt voor tabellen met kolommen. Stel op tabel b zit trigger T_b die tabel a vult. Wanneer aan tabel a een kolom toegevoegd wordt, moet met behulp van CIA duidelijk worden dat trigger T_b gewijzigd moet worden, zodat de nieuwe kolom ook gevuld wordt.

De impact van de wijzigingen kunnen worden weergegeven in het model [25; 28] of in de broncode [26; 30]. Het gaat bij CIA niet om alle broncode, maar alleen om de broncode die speciaal voor het te analyseren project geschreven wordt. Het gaat hierbij vooral om SQL code. Deze SQL code is uiteindelijk te vinden in triggers, views, layout procedures, default procedures, functies en stored procedures. In het MSD wordt opgeslagen hoe deze broncode eruit ziet of waar de broncode opgeslagen wordt. Er is een aantal plaatsen waar dit gebeurt:

- Control procedures met code templates
- De tabel *tab_check_constraint*
- Het veld *look_up_query* uit de tabel *tab*
- Prefilters en prefilter kolommen

De belangrijkste en meest gebruikte plek is bij de control procedures met code templates. Deze zullen daarom ook verder uitgewerkt worden. De CIA kan later uitgebreid worden, zodat de overige plekken waarin code weergegeven wordt ook geanalyseerd kunnen worden.

9 Ontwerp CIA voor Thinkwise

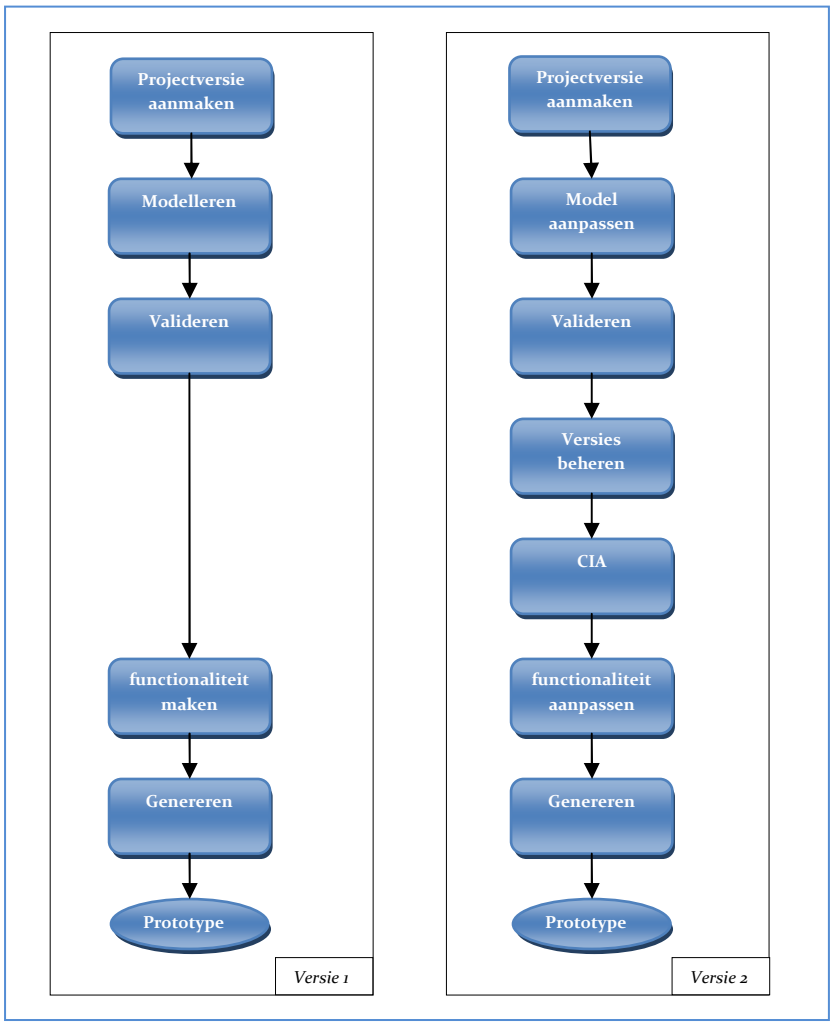
Het doel van Change Impact Analyse (CIA) voor Thinkwise is om in nieuwe versies aan te geven waar (bijvoorbeeld welke templates/control procedures) de wijzigingen (van bijvoorbeeld tabellen, kolommen, domeinen en taken) impact op hebben en wat dus gewijzigd moet worden om fouten te voorkomen. Het analyseren zou ook van tevoren mogelijk moeten zijn, zodat voor een enkele tabel, kolom, domein of taak geanalyseerd kan worden waar het impact op heeft als deze tabel, taak, etc., zou veranderen.

Impact analyse is interessant voor ontwikkelaars, omdat de aanpassingen die gedaan moeten worden eerder in het ontwikkelproces geïdentificeerd worden. Zonder change impact analyse worden mogelijke wijzigingen pas opgespoord wanneer de code uitgevoerd wordt en de ontwikkelaar syntaxis foutmeldingen krijgt. Of de wijzingen resulteren niet in een syntaxis foutmelding en worden pas bij het testen gevonden, omdat de code niet meer werkt zoals verwacht.

Om de Impact analyse te demonstreren is een prototype gemaakt, dit prototype wordt in hoofdstuk o beschreven.

9.1 Impact analyse in het ontwikkelproces

De impact analyse is niet een op zichzelf staand programma, dat los uitgevoerd kan worden. Voordat met impact analyse begonnen kan worden, moet eerst aan een paar condities voldaan worden. Nadat de analyse zelf gedaan is moeten er vervolgacties uitgevoerd worden, zodat de impact analyse goed verwerkt wordt. Daarom wordt hier beschreven hoe impact analyse in het ontwikkelproces past.



Figuur 16: Ontwikkelproces

De impact analyse is bedoeld om tussen versies analyse uit te voeren. Daarom moet er eerst een huidige en een vorige versie bestaan. Om een eerste versie aan te maken moeten de volgende stappen ondernomen worden. Eerst moet een project aangemaakt worden, hierin moeten gegevens over de nieuwe applicatie in de MSD ingevoerd worden, daarna moet het MSD

gevalideerd worden, als alle gegevens goed zijn, kan er code ontwikkeld en gegenereerd worden. Deze code kan weer gebruikt worden om de eerste versie van de applicatie (een eerste prototype) te maken.

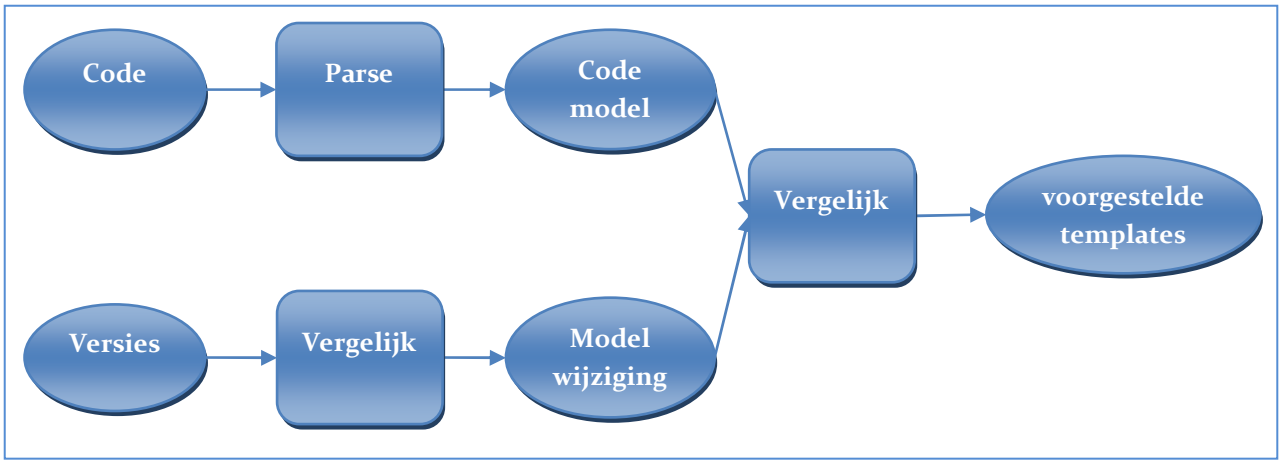
Wanneer de eerste versie van de applicatie bestaat, blijkt meestal dat deze nog niet compleet is en ontstaan er nieuwe wensen. Er moeten dan nieuwe aanpassingen aan de applicatie gedaan worden. Om een tweede of volgende versie van de applicatie te maken moeten de volgende stappen ondernomen worden. Eerst wordt er een nieuwe projectversie aangemaakt. In deze projectversie worden wijzigingen doorgevoerd. Deze gegevens moeten gevalideerd worden. Daarna moet versiebeheer aangemaakt worden, vervolgens kan de impact analyse uitgevoerd worden. Als de gegevens uit de analyse verwerkt zijn, kan de code gegenereerd en uitgevoerd worden, zodat een volgende versie van de applicatie gebruikt kan worden.

Met de impact analyse wordt laten zien op welke plaatsen de wijzigingen impact hebben. Ontwikkelaars moeten vervolgens zelf beslissen of er wijzigingen in de code of in het model aangebracht moeten worden. Als er een deel gecorrigeerd is, kan weer opnieuw een impact analyse uitgevoerd worden. Bij een modelwijziging moet opnieuw het hele proces doorlopen worden, zodat alles bijgewerkt wordt. Dit houdt in dat opnieuw versiebeheer aangemaakt wordt, opnieuw gevalideerd wordt, de nieuwe delen gegenereerd worden en als laatste de impact opnieuw geanalyseerd wordt. Als alles goed gegaan is, dan zorgt de verandering ervoor dat de eerdere melding niet weer voorkomt. Het is echter ook mogelijk dat deze verandering impact heeft op nieuwe stukken code die ook weer aangepast moeten worden. Bij een template wijziging hoeft alleen deze template opnieuw gegenereerd te worden. Het model blijft valide en de wijzingen tussen versies (versiebeheer) blijven ook hetzelfde, deze hoeven dus niet opnieuw uitgevoerd te worden.

9.2 Structuur van de impact analyse

De change impact analyse is opgedeeld in drie fasen, *vergelijken van versies*, *parsen van code naar een model* en *vergelijken van wijzigingen en code model*. In Figuur 17 worden deze fases weergegeven.

In het MSD zijn de verschillende versies opgeslagen. Tussen de verschillende versies kan vergeleken worden welke wijzigingen er gemaakt zijn, dit levert alle model wijzigingen op. De broncode waarop de impact analyse uitgevoerd moet worden, wordt met de code generator (ook wel functionality weaver genoemd) gemaakt door gegevens uit het MSD met code templates te weven. Vervolgens ontstaat door deze code te parsen een code model. In dit model worden de belangrijke elementen uit de code opgeslagen, zodat vervolgens de modellen vergeleken kunnen worden. De objecten die in de code voorkomen en gewijzigd zijn worden aan de ontwikkelaar voorgesteld als aan te passen templates. De ontwikkelaar kan vervolgens kiezen of de templates aangepast moeten worden of dat de voorstellen afgekeurd worden.

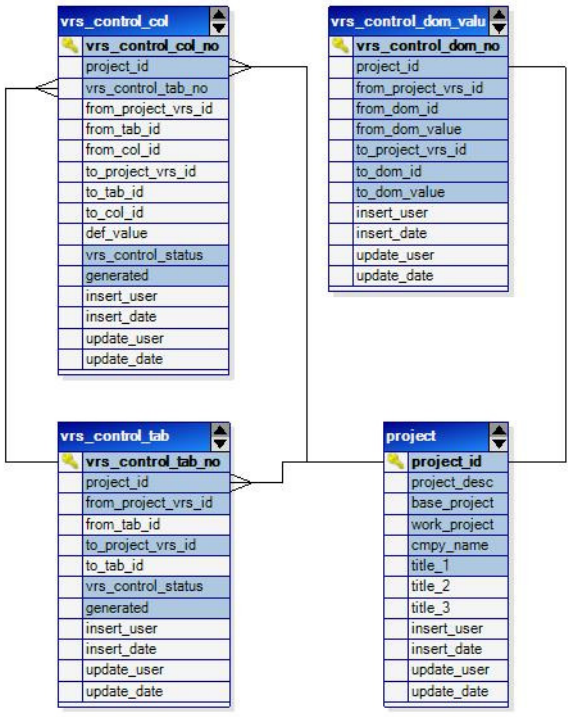


Figuur 17: Ontwerp change impact analyse

In de volgende subhoofdstukken is weergegeven hoe de verschillende fases ingevuld kunnen worden. Het doel van dit onderzoek is niet om een compleet tool neer te zetten die de impact op alle mogelijke manieren kan analyseren. In plaats daarvan is een tool gemaakt die zich vooral richt op de impact die wijzigingen op tabellen en kolommen hebben. Wanneer er keuzes gemaakt zijn, wordt er rekening mee gehouden dat de tool ook uitgebreid kan worden voor meerdere wijzigingen, maar deze wijzigingen zijn nog niet doorgevoerd.

9.3 Fase 1: Vergelijken van versies

Om change impact analyse uit te voeren, moet eerst duidelijk worden welke wijzigingen uitgevoerd zijn. Hiervoor bestaat binnen de software fabriek al een mechanisme, dat bij houdt hoe het datamodel gewijzigd is. Dit wordt versiebeheer genoemd en wordt gebruikt om de database te upgraden van de ene versie naar de volgende.



Figuur 18: Datamodel versiebeheer

9.3.1 Versiebeheer

Versiebeheer wordt niet elke keer bijgewerkt, dit zou de hele applicatie traag maken. Dit wordt gedaan wanneer een versie klaar is en gegenereerd moet worden. Wanneer de taak *"versiebeheer aanmaken"* wordt uitgevoerd, wordt van tabellen, kolommen en domeinelementen het verschil in versies opgeslagen. In Figuur 18 wordt het datamodel van versiebeheer weergegeven.

Alle tabellen worden in *vrs_control_tab* opgeslagen. Een nieuwe tabel is te herkennen, doordat het bijbehorende record geen *from_tab_id* heeft. Een tabel die juist verwijderd wordt, heeft

geen *to_tab_id*. Wijzigingen van de naam van een tabel zijn te herkennen aan records waarbij de *from_tab_id* ongelijk is aan de *to_tab_id*. Hetzelfde geldt voor kolommen en de *col_id*. Tabellen en kolommen hebben ook een vrs_control_status. Dit kan gevuld zijn met de waarde *nieuw*, *goedgekeurd* of *gewijzigd*. Als een object toegevoegd of verwijderd wordt, dan is het record in versiebeheer nieuw. Een object dat niet gewijzigd is, wordt wel weergegeven, maar heeft als status goedgekeurd. Alle andere (gewijzigde) records hebben status gewijzigd.

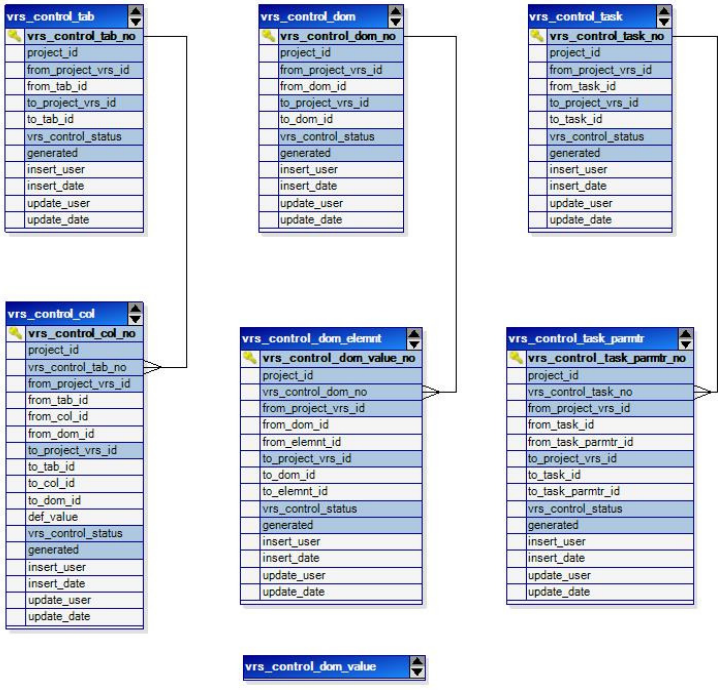
9.3.2 Uitbreidingen versiebeheer

Versiebeheer kan uitgebreid worden, zodat het ook geschikt is voor impact analyse. De tabel voor tabellen en kolommen kunnen gemakkelijk uitgebreid worden. De tabel met domeinwaarden is echter erg specifiek voor database wijzigingen en is daarom niet geschikt. Daarnaast moet voor domeinen en taken nieuwe tabellen toegevoegd worden. In Figuur 19 wordt aangegeven hoe de nieuwe versiebeheer tabellen eruit komen te zien. Elke tabel heeft een "*from_*" en een "*to_*" kolom. Hieruit kan opgemaakt worden of een object (bv. tabel) nieuw is (geen "*from_*"), verwijderd is (geen "*to_*"), gewijzigd is ("*from_*" is anders dan "*to_*") of gelijk gebleven is.

Het bijhouden van objecten die gewijzigd zijn, kan op twee manieren. De eerste manier is door in de tabel voor elk criterium dat kan wijzigen een checkbox op te nemen. Wanneer dat criterium gewijzigd is, wordt de checkbox aangevinkt. Achteraf kan gezocht worden op alle records waarbij de checkbox aangevinkt is.

De tweede manier is om een statusveld op te nemen die aangeeft of het object gewijzigd is. Hierdoor moet achteraf gecontroleerd worden welke conditie van het object gewijzigd is, maar het voordeel is dat er gemakkelijk nieuwe condities toegevoegd kunnen worden.

Hier is gekozen voor de laatste manier, daarom is aan elke tabel het veld *vrs_control_status* toegevoegd. Een wijziging aan het datamodel van de software fabriek betekent voor Thinkwise dat alle klanten over moeten naar de nieuwe projectversie. Een wijziging aan een taak kan gemakkelijk doorgevoerd worden door de SQL van deze taak opnieuw af te schieten bij klanten die gebruik willen gaan maken van de nieuwe functionaliteit. De taak waarmee dit gebeurt, moet vervolgens aangepast worden, zodat deze ook domeinen en taken vergelijkt.



Figuur 19: ERD model nieuw versiebeheer

Versiebeheer zorgt ervoor dat de impact analyse uitgevoerd kan worden tussen versies. Daarnaast zou het ook handig zijn als van tevoren al weergegeven kan worden waar een wijziging impact op zal hebben, zodat een beslissing gemaakt kan worden of de wijziging

doorgevoerd moet worden of niet. Dit heeft minder prioriteit dan de impact analyse tussen verschillende versies. Toch zouden met de uitbreiding van een nieuwe taak de vervolgstappen in de impact analyse aangeroepen kunnen worden met slechts één object als gewijzigd model.

9.4 Fase 2: Parsen van code naar een model

Om te controleren waar een onderdeel impact op heeft, moet alle code geanalyseerd worden. Grote projecten bestaan uit veel code, waardoor het veel tijd kan kosten om alle code langs te lopen. Als er een analyse voor slechts één component gedaan wordt, moet nog steeds alle code doorlopen worden, dit kost te veel tijd en heeft dit als gevolg dat de CIA voor kleine componenten niet gebruikt zal gaan worden. Dit wordt vooral vervelend als uit de impact analyse blijkt dat er een fout is en men na het oplossen van deze fout weer alle code door moet om te kijken of de situatie goed is opgelost. Dit probleem wordt vermeden door de resultaten van het analyseren van de code op te slaan en alleen de delen die gewijzigd zijn opnieuw te controleren.

9.4.1 Moment van opslaan

De vraag is nu wanneer het analyseren en opslaan van de code moet gebeuren. Dit zou kunnen gebeuren wanneer de ontwikkelaar aangeeft dat hij de impact van een wijziging wil zien. Dit heeft als nadeel dat de opgeslagen gegevens soms verouderd zijn. Het is lastig te beoordelen welke gegevens actueel zijn en welke gegevens opnieuw geanalyseerd moeten worden. Analyse van de code kan daarom het beste tijdens het genereerproces uitgevoerd worden. Wanneer het analyseren van code tijdens het genereren gedaan kan worden, kost het één keer meer tijd. Daarna kunnen de gegevens uit de tabel opgezocht worden. Omdat het analyseren tijdens het genereren gebeurd, is de analyse altijd gelijk met de gegenereerde code.

Bij het aanmaken van een nieuwe projectversie kan het hele code model gekopieerd worden. Dit zorgt ervoor dat de code niet allemaal opnieuw gegenereerd moet worden voor de nieuwe projectversie, terwijl de code zelf gelijk blijft. Zo kan, voordat er gegenereerd wordt, al een impact analyse gedaan worden op basis van het oude code model, omdat de code nog niet gewijzigd is. Bij wijzigingen van de broncode moet er altijd opnieuw gegenereerd worden, zodat zowel de analyse, als het genereren van code alleen voor het gewijzigde deel opnieuw uitgevoerd wordt.

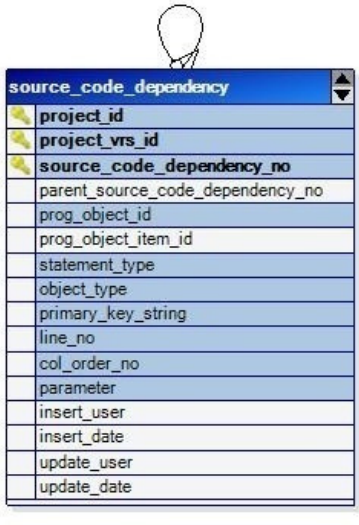
9.4.2 Opslaan van gegevens

De opgeslagen gegevens moeten zo compleet mogelijk zijn, daarom wordt gekeken naar de mogelijke wijzigingen uit hoofdstuk 8.3. Voor mutaties met tabellen, is het nodig dat alle tabellen opgeslagen worden. Als vervolgens een tabelnaam wijzigt, kunnen alle voorkomens van die tabel opgezocht worden. Hetzelfde geldt voor kolommen, als een kolomnaam wijzigt, moeten alle voorkomens hiervan opgezocht worden. Maar bij kolommen is het ook belangrijk te weten bij welke tabel ze horen. Er kunnen in een query ook meerdere tabellen gebruikt worden in een join. Dan is per tabel belangrijk aan welke andere tabel deze gekoppeld is. Ook taken en taakparameters moeten opgeslagen worden. Voor taakparameters geldt net als voor kolommen, dat een parameter alleen binnen de context van een taak een geldige waarde heeft en dus de bijbehorende taak ook opgeslagen moet worden.

Domeinen worden op twee manieren gebruikt:

1. Ten eerste worden aan elke kolom een domein toegekend. Die kolom mag alleen waardes bevatten die voldoen aan de eisen opgeslagen in het domein (bijvoorbeeld alleen getallen, een bit of maximaal twee karakters). Door de kolommen op te zoeken die van een gewijzigd domein gebruik maken, kan hier ook een analyse van gemaakt worden. Een voorbeeld van een domeinwijziging is, als een domein van een *varchar* naar een *int* veranderd. Dit kan toegepast worden als blijkt dat in een veld niet zomaar alles ingevuld mag worden, maar een keuze gemaakt moet worden tussen verschillende teksten. De gebruiker krijgt dan een combobox waaruit de tekst gekozen kan worden, onder water heeft elke tekst een nummer wat in de tabel opgeslagen wordt. Voor deze wijziging moeten alle kolommen die het domein gebruikt opgezocht worden. Zo kunnen overal aanpassingen gemaakt worden waar de kolom toegewezen wordt aan een waarde of waar een vergelijking met de kolom gemaakt wordt. Een ander voorbeeld waarin de kolom opgezocht moet worden als een domeinwaarde veranderd is wanneer er waardes toegevoegd zijn. Bijvoorbeeld bij statussen, als er tussen *initieel* (1) en *afgesloten* (2) een nieuwe status (2) komt, wordt de status *afgesloten* een nummer opgehoogd, overal waar het oude nummer gebruikt wordt, moet deze aangepast worden.
2. Ten tweede kunnen domeinen gebruikt worden bij de declaratie van variabelen. Wanneer de domeinnaam verandert, moeten deze declaraties ook aangepast worden. Ook als de eisen aan het domein wijzigen, moet gecontroleerd worden of de variabelen niet gevuld worden met een onjuiste waarde.

Het code model moet opgeslagen worden in de MSD. Hiervoor is een datamodel gemaakt. Er kan gekozen worden om elk element in een nieuwe tabel op te slaan, zoals bij versiebeheer of alles in één tabel op te slaan. Hier is gekozen om alles in een tabel op te slaan, omdat de gegevens die opgeslagen moeten worden voor elk element hetzelfde zijn. Voor zowel een tabel als een kolom als een taak moet opgeslagen worden waar het vandaan komt en hoe het element in de code te herkennen is. Bij versiebeheer is juist gekozen om alles in losse tabellen op te slaan, omdat deze tabellen ook door gebruikers onderhouden kunnen worden. Bijvoorbeeld, wanneer een kolom van naam veranderd, kan besloten worden dat het gemakkelijker is om de oude kolom te verwijderen en een nieuwe kolom aan te maken. Vervolgens kan in versiebeheer de records aangepast worden, zodat deze herkent dat de kolom van naam veranderd is. Bij source_code_dependency wordt de tabel gelezen en geschreven door het systeem en zal deze tabel dus ook nooit voor gebruikers zichtbaar zijn.



Figuur 20: Impact analyse tabel

Het datamodel bestaat hierdoor uit één tabel die een referentie heeft naar zichzelf. Deze tabel is weergegeven in Figuur 20. Een analyse wordt altijd gedaan in de context van een project en projectversie, daarom zijn de eerste velden toegevoegd. Om elk record uniek te maken is een identifier toegevoegd (*source_code_dependency_no*), deze identifier kan gebruikt worden in de

parent_source_code_dependency_no van de kinderen. Zo kan een tabel aan een andere tabel gekoppeld worden, een kolom aan de bijbehorende tabel toegevoegd worden of een waarde aan een kolom gekoppeld worden bij een toekenning.

Vervolgens wordt opgeslagen waar de tekst vandaan gekomen is. De impact analyse wordt uitgevoerd per stukje gegenereerde code. Elk stuk code stamt af van een programmaobject item. Dit programmaobject item wordt opgeslagen in de source code dependency. Alle programmaobject items vormen samen een programmaobject. Dit zijn complete procedures of triggers. Soms zijn programmaobject items niet syntaxis correct, omdat ze enkel voor kunnen komen samen met andere programmaobject items. In een dergelijk geval wordt het hele programmaobject geparset en het veld *prog_object_item_id* is dan leeg (*null*).

Om dependencies te herkennen in de broncode moet nog de volgende gegevens worden opgeslagen:

- uit welk deel van het statement het afkomstig is (INSERT, UPDATE, DELETE, SELECT, JOIN, WHERE, etc),
- wat voor soort object het is dat opgeslagen wordt (*tab*, *col*, *dom*, *dom_elmnt*, *task*, *task_col*),
- de parent als deze bestaat (bv. de tabel bij een kolom),
- de unieke key (bv. de *tab_id* voor een tabel),
- welke regel in de bron code,
- welk volgnummer in de lijst het element heeft (bv. voor een insert om de veldnaam bij de veldwaarde te vinden)
- en of de gevonden dependency een stukje tekst uit een template is of de waarde van een parameter.

De onderste vier kolommen van de source_code_dependency tabel houden de mutatiegegevens bij, deze kolommen worden altijd aan tabellen van de SF toegevoegd.

9.4.3 De code generator

De code generator is een tool in de SF waarmee een eindproduct uitgegenereerd kan worden. Deze tool kijkt in het MSD welke stukken code geweven moeten worden en schrijft dit vervolgens weg in de verschillende documenten. Nadat een project en projectversie gekozen is, kan het eerste tabblad gebruikt worden om code te genereren. Als dit klaar is, kan op het tweede tabblad verbinding met de database gemaakt worden en kan de code afgevuurd worden. Door het DB script uit te voeren, kan een nieuwe database gemaakt worden. Als er al een database bestaat, wordt juist een Upgrade script afgevuurd. Dit script zorgt ervoor dat de database aangepast wordt aan het nieuwe metaniveau. Hier wordt geen DB script gebruikt, omdat de data die in de database zit met een update wel in tact blijft. De code generator kan alle soorten code genereren, afhankelijk van de taal die in de gebruikte templates staat. Zo wordt naast een eindproduct ook de help voor dit eindproduct gegenereerd, met behulp van templates in HTML code.

Voordat begonnen wordt met genereren, kunnen er eerst een aantal instellingen en filters gezet worden. Zo kan in plaats van alles te genereren een filter gezet worden, zodat een losse

groep, bijvoorbeeld alle triggers, in een keer gegenereerd wordt. Ook kan op deze manier een ontwikkelaar die een template toegevoegd heeft voor alleen deze template de code genereren.

9.4.4 Vullen van gegevens

De impact analyse tabel wordt gevuld tijdens het genereren. Dit wordt gedaan door alle code langs te lopen en de verschillende elementen te ontleden (parse). Daarbij moet geïdentificeerd worden welke elementen belangrijk zijn en deze moeten opgeslagen worden in de database. Omdat het niet het doel van dit onderzoek is om een complete SQL parser te schrijven is, gezocht naar een SQL parser die uit templates de tabellen, kolommen, taken en taakparameters kan herkennen.

Het probleem dat hierbij gevonden werd, is dat normale SQL parsers verwachten dat de SQL code syntaxis correct is. Dit is bij templates niet altijd het geval, onder andere omdat gebruik gemaakt kan worden van parameters. Deze parameters worden tijdens het generatieproces vervangen met echte waardes, zodat correcte SQL code ontstaat. Deze waardes kunnen niet alleen tabellen of kolommen bevatten, maar ook meerdere regels broncode. Een andere reden dat de syntax van één template niet correct is, is omdat deze slechts een deel van een correct stuk code kan zijn. Bijvoorbeeld een procedure die bestaat uit een kop, inhoud en staart, de staart alleen kan incorrect zijn omdat de kop en inhoud missen. Omdat het vullen van het code model gedaan wordt tijdens het generatieproces, is besloten gebruik te maken van de gegenereerde code. Hierdoor wordt niet gebruik gemaakt van de templates, maar van, zoals eerder al aangegeven is, de programmaobjecten.

Voor het vullen van de gegevens is een parser gevonden. Deze parser is geschreven om als bibliotheek in een .NET applicatie gebruikt te kunnen worden. Meer informatie over waarom deze parser gekozen is, staat beschreven in bijlage II. Vervolgens is deze parser gebruikt om de code extractor te maken. Dit is het deel van de impact analyse die de belangrijke elementen uit de parser haalt en in de database zet. Hoe de code extractor precies werkt is weergegeven in hoofdstuk 10.2. Daarin staat beschreven hoe deze gebruikt wordt en hoe de code generator aangepast moet worden om deze code extractor te gebruiken.

9.5 Fase 3: Vergelijken van wijzigingen en code model

De derde fase van de impact analyse is de controle of een element uit het code model ook daadwerkelijk een element is dat gewijzigd moet worden. Voor sommige elementen is makkelijk vast te stellen dat ze gewijzigd moeten worden. Bijvoorbeeld een tabel die van naam gewijzigd is en waarvan de oude naam nog in code gebruikt wordt. Voor andere zal een intelligenter algoritme bedacht moeten worden om precies vast te stellen welke elementen aangepast moeten worden.

9.5.1 Intelligentie van de tool

Het is niet altijd van tevoren te bepalen of een specifiek element gewijzigd moet worden en op welke manier dit gedaan moet worden. Soms is de hulp van de ontwikkelaar die dat onderdeel bedacht heeft nodig. Dit is bijvoorbeeld het geval als je kijkt naar een select query, deze query selecteert een aantal kolommen uit een tabel. Als deze tabel een nieuwe kolom krijgt, is het niet van te voren te beslissen of deze kolom ook in de query toegevoegd moet worden. Dat kan alleen besloten worden door iemand die weet wat het doel van de select query was.

Door de gevallen te melden waarvan niet zeker te zeggen is of ze aangepast moeten worden, komen er *false positives* in het resultaat. Dit zijn meldingen die niet gewijzigd hoeven te worden, terwijl de impact analyse aangeeft dat dit wel zou moeten. Daarentegen zouden er meldingen niet weergegeven worden die wel aangepast moeten worden (*false negatives*) wanneer de *false positives* vermeden zouden worden. Het nadeel van *false positives* is dat het tijd kost om te beslissen of de meldingen correct zijn of aangepast moeten worden. Het nadeel van *false negatives* is dat de meldingen niet weergegeven worden en er dus nog steeds geen controle is over de mogelijke aanpassingen.

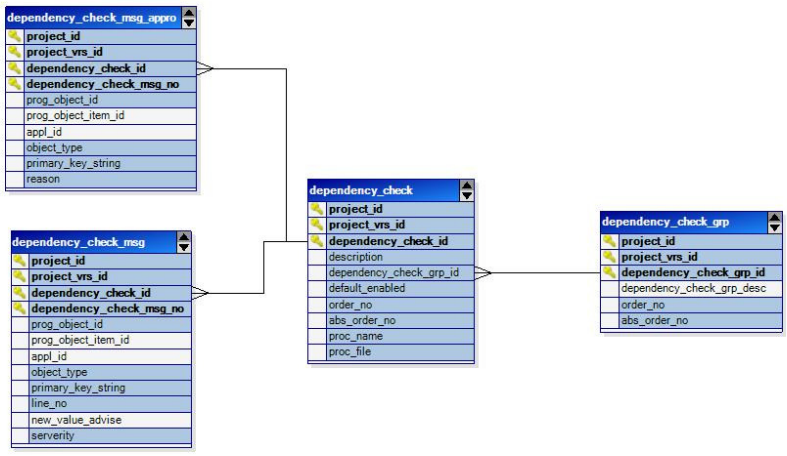
Dus hoe intelligenter de tool wordt, hoe beter ermee gewerkt kan worden. Maar de controles mogen ook weer niet te strikt zijn, want dan ontstaan er weer *false negatives* die ervoor zorgen dat de tool minder bruikbaar is.

9.5.2 "Controle tool" of "Schil van een control tool"

Er is in dit onderzoek gekozen om niet zelf een complete tool te schrijven, maar een schil, zodat ontwikkelaars zelf controles kunnen toevoegen. Deze techniek is al eerder bij de validator gebruikt en is voor deze situatie ook zeer geschikt. Een voordeel van deze techniek is dat niet vooraf alle controles geschreven hoeven te worden, de controles kunnen later als blijkt dat dit nodig is toegevoegd worden. Een ander voordeel is dat de controles beter onderhoudbaar zijn. Ze zijn verdeeld in kleine beheersbare procedures die makkelijk geïdentificeerd en aangepast kunnen worden. Door het flexibel toevoegen van controles wordt de tool steeds intelligenter en kan in steeds meer situaties toegepast worden.

Er zijn ook nadelen aan deze methode die er daarom voor pleiten om zelf alles te programmeren. Het nadeel is dat anderen controles kunnen toevoegen die niet getest zijn of die niet goed werken. Omdat de techniek ervoor zorgt dat de tool flexibeler is, kan deze flexibiliteit ook misbruikt worden. Dit is echter de verantwoordelijkheid van degene die besluit om de controles van anderen te gebruiken. Projectleiders moeten ervoor zorgen dat de kwaliteit van de gebruikte controles goed is, net zoals zij ervoor moeten zorgen dat de kwaliteit van het eindproduct goed is.

Een ander nadeel is dat er minder tijd gestoken wordt in het ontwikkelen van controles, zodat sommige controles nog niet aanwezig zijn en de drempel om de tool te gebruiken hoger ligt. Dit betekent dat er buiten dit project om eerst extra controles gemaakt moeten worden, voordat de tool volledig ingezet kan worden. Het is echter niet het doel van dit project om zo compleet mogelijk impact analyse controles te schrijven. Wanneer dit in een ander project gedaan wordt, kan



Figuur 21: Metamodel dependency check

meer aandacht besteed worden aan het ontwikkelen van nieuwe ideeën of uitbreidingen op de tool die niet flexibel door anderen of op andere momenten gedaan kunnen worden.

9.5.3 De dependency checker

Om de schil van een controle tool te realiseren is het datamodel van de software fabriek uitgebreid en er is een tool ontwikkeld, genaamd de dependency checker. De dependency checker maakt gebruik van het versiebeheer uit fase 1 en het code model uit fase 2 om de ontwikkelaar meldingen te geven van delen uit code die aangepast moet worden. Zowel versiebeheer, als het code model is opgeslagen in tabellen in de database. Deze tabellen zijn met SQL statements te bereiken. Daarom worden de controles die checken of een wijziging in het code model zit, ook in SQL geschreven worden.

Voor elke controle kan een stored procedure geschreven worden. Doordat deze procedures allemaal dezelfde parameters meekrijgen, namelijk project, projectversie, applicatie en dependency check, kan de aanroep generiek opgebouwd worden. Als opgeslagen is welke namen de procedures hebben, kunnen ze om de beurten aangeroepen worden. In Figuur 21 zijn de nieuwe tabellen weergegeven waarin dit opgeslagen kan worden. In dependency check grp worden de verschillende groepen weergegeven. Deze groepen zorgen ervoor dat als er veel controles zijn er toch overzicht blijft.

Elke groep heeft (nul of meer) dependency checks, dit zijn de daadwerkelijke controles. Ze hebben een referentie naar een procedure en deze wordt aangeroepen wanneer de check uitgevoerd wordt.

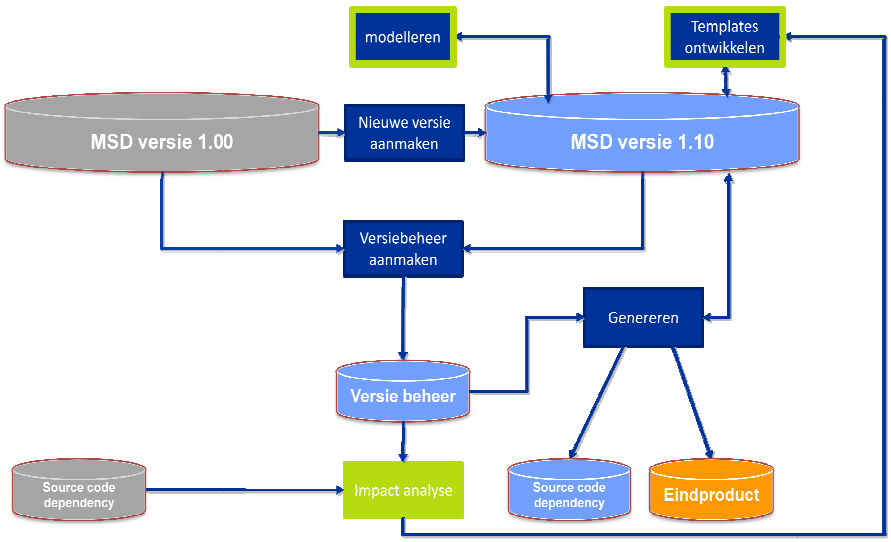
De procedures moeten het resultaat ergens op kunnen slaan, zodat dit door de dependency checker weergegeven kan worden en door de ontwikkelaar verwerkt kan worden. Hiervoor is de tabel *dependency check msg*. In deze tabel wordt opgeslagen waar de dependency vandaan komt en hoe de wijziging mogelijk opgelost kunnen worden. Zo is het mogelijk een nieuwe waarde op te geven, dit geeft aan dat de oude waarde vervangen zou moeten worden met deze nieuwe waarde. Ook kan een status aan een melding gegeven worden. Er zijn drie statussen, namelijk *automatisch, per definitie* of *ter beoordeling*. Als de melding niet terecht is, kan deze afgekeurd worden. De melding wordt dan opgeslagen in de tabel *dependency check msg approved* en komt niet meer in de lijst terecht.

Het moet mogelijk zijn vanuit de dependency check verder te gaan naar schermen waarin de code gelijk aangepast kan worden. Bij de dependency checker kan op een melding geklikt worden, dan moet het resultaat tabblad weergegeven worden. Hierin zijn alle programmaobjecten met programmaobject items weergegeven, deze zijn gefilterd, zodat alleen programmaobjecten die in een melding voorkomen weergegeven worden. De verschillende meldingen moeten hier afgekeurd kunnen worden, automatisch wijzigen of handmatig wijzigen. Daarna moet het mogelijk zijn opnieuw te genereren, zodat de impact analyse nogmaals gecheckt kan worden.

9.6 Resultaat

Voor de impact analyse is het nodig om twee verschillende versies te hebben, omdat tussen deze versies de verschillen worden analyseerd. In Figuur 22 is het uitgebreide kwaliteitsproces weergegeven. Hierin bestaat er een oude versie (1.00). Deze is gebruikt om een nieuwe versie

aan te maken (1.10). Met de input van beide versies is versiebeheer gemaakt. Versiebeheer is opgeslagen in dezelfde database als de rest van de elektronische bouwtekening en deze zijn daarom in dezelfde kleur weergegeven. In de eerste versie is al eerder in het ontwikkelproces de source code dependency gegenereerd. Het genereren gebeurt in elke projectversie, maar voor het overzicht is alleen het genereren van de laatste versie weergegeven en de input die nodig is om de laatste versie te maken. De functionality modeler gebruikt versiebeheer en het MSD om een eindproduct en source code dependency te genereren. Deze source code dependency wordt samen met het versiebeheer gebruikt om meldingen te gegeven die gebruikt kunnen worden bij het ontwikkelen van templates.



Figuur 22: Kwaliteitsproces III

De impact analyse is niet alleen een concept, maar is ook verder ontwikkeld tot een prototype. Deze wordt in het volgende hoofdstuk beschreven.

10 Prototype CIA

Om de change impact analyse te demonstreren is een prototype gemaakt. Dit prototype bestaat uit de drie fases, zoals deze in het vorige hoofdstuk beschreven zijn. Het prototype richt zich nu nog alleen op de analyse van tabellen en kolommen, later kan dit uitgebreid worden, zodat domeinen en taken ook ondersteund worden.

Voor fase 1 van CIA was het voldoende om versiebeheer te gebruiken, zoals het oorspronkelijk ontwikkeld was. Voor fase 2 en fase 3 waren wel nieuwe onderdelen nodig. Voor fase 2 is de code extractor ontwikkeld en voor fase 3 is de dependency checker gemaakt. Later worden dieper op de beschrijving van deze nieuwe onderdelen ingegaan.

10.1 Geïntegreerd in de SF vs. één nieuwe tool

Het is mogelijk om zowel het versiebeheer, als de code extractor, als de dependency checker in één tool te stoppen. Daarentegen kan er ook voor gekozen worden om de drie onderdelen los een plek in het ontwikkelproces te geven. Beide keuzes hebben voor en nadelen, deze zullen hier verder uiteengezet worden.

Voor de eerste fase, versiebeheer maakt het niet veel verschil. Versiebeheer bestaat al en zit al in de schermen waarin project versie en versiebeheer gemanaged worden. Door alles in één tool te stoppen, moet deze functionaliteit slechts een extra keer vanuit de nieuwe tool aangeroepen worden. Dit kan met taken en bestaande componenten snel gerealiseerd worden.

In de tweede fase kan de code extractor aan de ene kant door de code generator aangeroepen worden. Dit heeft als voordeel dat de gegenereerde code direct beschikbaar is. Aan de andere kant geldt dat als het in een los tool zou zitten, het opzoeken van de templates en vervangen van de juiste parameters nog een keer gedaan moet worden. Dit omdat de gegenereerde code niet opgeslagen worden, maar gelijk weggeschreven wordt naar een bestand. Het is niet verstandig om ervan uit te gaan dat dit bestand nog in tact is of zelfs nog beschikbaar is, wanneer het controle gedeelte van de impact analyse uitgevoerd wordt.

Het nadeel van het plaatsen van de code extractor in de code generator is dat het hierdoor meer tijd kost om code te genereren. Bij een klein project, zoals WORKSHOP, kost het genereren van de code 4 seconden, als hier een impact analyse bijgedaan wordt, kost het 6 seconden. Het verschil wordt groter naar mate het project ook groter worden. Het project SQLSERVER_SF is een project waarin het metaniveau opgeslagen zit. Dit project is veel groter dan het WORKSHOP project, maar kleiner dan een groot klantproject. Hier deed de code generator 1 minuut over om de broncode te maken, terwijl met de code extractor erbij er 2,5 minuten voor nodig waren. Doordat de extra functionaliteit die toegevoegd wordt voor de impact analyse er zo lang over doet, zou het verstandig zijn als dit ook in de instellingen uit gezet kan worden.

Het doel van de code generator is om een eindproduct neer te zetten. Het opslaan van code in tabellen heeft echter niets te maken met het neerzetten van een eindproduct. Er is geen functionele reden waarom dit dan in de code generator uitgevoerd moet worden. Als er een instelling wordt weergegeven waarmee een ontwikkelaar de keuze heeft om de code extractor uit te zetten, zou er ook een logische functionele reden moeten zijn waarom de code extractor normaal gesproken wel uitgevoerd wordt en met deze instelling dan niet. Er is echter wel een

technische reden om de impact analyse bij de code generator. Tijdens het genereren van broncode is de broncode ook daadwerkelijk beschikbaar. Als de code extractor later uitgevoerd wordt, dan moet het genereren van deze broncode opnieuw gebeuren.

De dependency checker van de laatste fase maakt geen gebruik van bestaande functionaliteit en zou daarom geschikt zijn als een nieuw tool van de Software Factory. Het zou echter ook in de functionality modeler geplaatst kunnen worden. De functionality modeler heeft als doel het aanpassen van broncode en de dependency checker is ontwikkeld om te helpen met het identificeren van de broncode die aangepast moet worden. Door dit samen in één tool te plaatsen, kan de broncode aangepast worden door eerst te kijken wat er moet gebeuren en daarna het daadwerkelijk uit te voeren. Als de hele impact analyse in één tool zit, zal deze hele tool waarschijnlijk niet in de functionality modeler geplaatst worden, omdat de functionele beschrijving van beide tools niet samen in een tool past. De dependency checker heeft als doel om analyse te doen naar de impact van wijzigingen en de functionality modeler heeft als doel om broncode aan te passen.

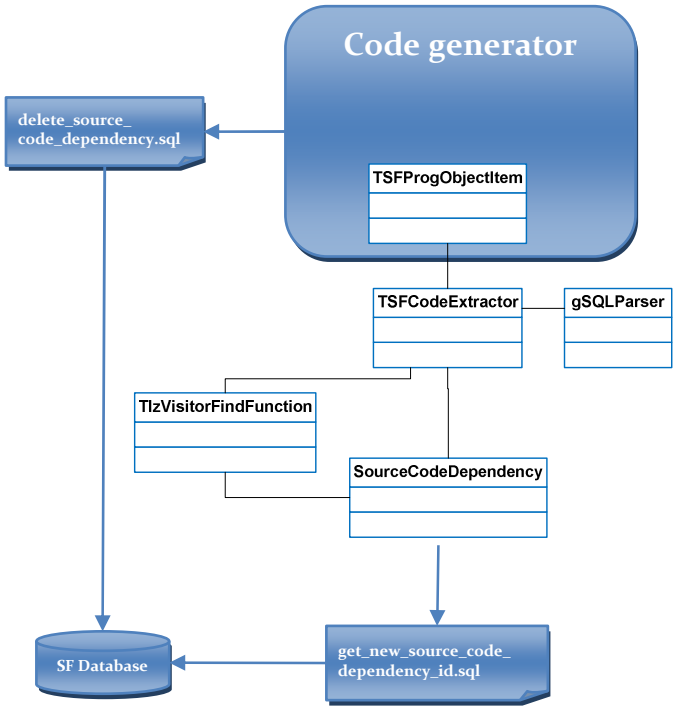
De keuze of de impact analyse in één tool wordt gebruikt of geïntegreerd wordt in het ontwikkelproces, heeft ook effect op de manier waarop ontwikkelaars de impact analyse gaan gebruiken. Voor beide keuzes zijn voor- en nadelen:

1. Wanneer impact analyse in één scherm wordt weergegeven, dan wordt het iets dat men één keer per projectversie uitvoert, van het begin tot het einde. Alle wijzigingen worden opgezocht, daarna wordt alle broncode geanalyseerd en vervolgens wordt gecheckt welke code aangepast moet worden. Dit wordt een voor een aangepast en daarna is die projectversie klaar voor verdere ontwikkeling of uitlevering.
2. Wanneer de losse onderdelen geïntegreerd worden, dan is de impact analyse een techniek die meerdere keren en in kleine stukjes wordt uitgevoerd. Elke keer als een ontwikkelaar iets verandert aan de broncode, dan genereert zij, behalve de broncode ook een code model. Op elk moment in het proces kan dan een check uitgevoerd worden, zodat de analyse kan worden weergegeven. Met (een deel van) de resultaten kunnen aanpassingen gemaakt worden, zodat een ontwikkelaar (een onderdeel van) het project kan aanpassen en afronden. Vervolgens kan de check van de impact analyse opnieuw uitgevoerd worden om te controleren welke onderdelen nog meer aangepast moeten worden.

De software fabriek is opgezet op een manier waarop kleine stukken functionaliteit los uitgevoerd en ontwikkeld kunnen worden. Zo is het mogelijk met meerdere mensen aan verschillende of zelfs dezelfde stukken functionaliteit te werken. In plaats van het hele product in één keer te bouwen, wordt deze opgebouwd in cycli waarin de functionaliteit telkens verder uitgebreid wordt. Daarom is het verstandig om de impact analyse ook op te bouwen, zodat gemakkelijk meerdere malen achter elkaar snel een analyse van een klein stukje of zelfs van alle code weergegeven kan worden.

10.2 Code extractor

In Figuur 23 is weergegeven hoe de code extractor eruit ziet en waar het aan gekoppeld is. De klasse gSQLParser is de parser die gekocht is voor het parsen. Hoe de code generator precies werkt, is in dit onderzoek niet belangrijk. In het scherm van de code generator is een checkbox toegevoegd. Wanneer deze checkbox aangevinkt is, dan wordt ook de impact analyse uitgevoerd. Er is voor gekozen om dit als optie uit te zetten omdat het voor grote projecten redelijk lang kan duren en door het uit te kunnen zetten kan alles sneller gegenereerd worden. Er moet dan echter wel rekening mee gehouden worden dat het code model niet meer up to date is en dat hierdoor fouten pas later in het proces ontdekt kunnen worden.



Figuur 23: Fase 2: code model vullen

Tijdens het genereren zijn er twee activiteiten die zijn toegevoegd om de code extractor te realiseren.

1. Ten eerste wordt er een stored procedure (SP) aangeroepen. Deze SP verwijdert alle oude source code dependencies, rekening houdend met de parameters die opgegeven zijn. De SQL van delete_source_code_ dependency ziet er als volgt uit:

```
delete s
  from source_code_dependency s
  join prog_object p
    on p.project_id      = s.project_id
   and p.project_vrs_id  = s.project_vrs_id
   and p.prog_object_id  = s.prog_object_id
 where s.project_id      = @project_id
   and s.project_vrs_id  = @project_vrs_id
   and p.control_proc_id like @control_proc_id
   and p.prog_object_id  like @prog_object_id
```

2. Ten tweede wordt in de klasse TSFProgObjectItem de code extrator aangeroepen. De klasse TSFCodeExtrator bevat de procedure FillDependency, deze procedure wordt aangeroepen met een bepaald programmaobject item, een programmaobject en een stuk code dat bij dat programmaobject item hoort. Deze procedure gebruikt de parser om de database te vullen.

Hieronder is een aantal belangrijke regels van FillDependency weergegeven.

```
sourceCodeDependency dummy;
List<sourceCodeDependency> dependencies;

TGSqlParser sqlparser;
sqlparser = new TGSqlParser(TDbVendor.DbVMssql);

sqlparser.SqlText.Text = source_code.ToString();
sqlparser.Parse();

dummy = new sourceCodeDependency(_projectId, _projectVrsId, "", null, null, _progObj,
          _progObjItem, 0, 0, false, sourceCodeDependency.statementType.SELECT);

for (int j = 0; j <= sqlparser.SqlStatements.Count() - 1; j++)
{
    TCustomSqlStatement statement = sqlparser.SqlStatements[j];

    dependencies = _fillDependency(new List<sourceCodeDependency>(), statement, dummy);
    storeStatement(dependencies);
}
```

Bij deze code wordt een parser gedefinieerd. De parser leest nu enkel Microsoft SQL Server code, maar in de toekomst moet van het MSD uitgelezen worden welke databasetaal gebruikt wordt. Dit onderscheid is vanuit de parser gemaakt, omdat elke database een ander dialect van SQL gebruikt. De grote lijnen komen redelijk samen, maar sommige constructies mogen in de ene taal niet, terwijl ze in de andere taal juist verplicht zijn.

Door bij de parser een tekst (de te parsen broncode) toe te voegen en de *parse* procedure aan te roepen, wordt de SQL tekst geparset in een SQL representatie. Uit deze representatie moeten de elementen gehaald worden die voor de impact analyse belangrijk zijn en deze elementen moeten opgeslagen worden in de database. Hiervoor is een dataklasse gemaakt waarin de elementen tijdelijk opgeslagen kunnen worden, genaamd sourceCodeDependency. In deze klasse wordt onder andere opgeslagen wat het project en projectversie is en uit welke programmaobject en programmaobject item de tekst komt, hier wordt later nog verder op ingegaan.

Met een for-lus wordt door de SQL representatie van de statements gelopen om voor elk statement de elementen eruit te halen. De lijst met elementen worden vervolgens per statement in de database opgeslagen.

De procedure *_fillDependency* bestaat uit een generieke en een aantal specifieke procedures. De generieke procedure kijkt naar het type statement en roept vervolgens een specifieke procedure aan. Als bijvoorbeeld een statement van type sstInsert is, dan kan dat statement gecast worden naar TInsertSQLStatement. Een procedure die ook *_fillDepdenency* heet, maar die een TInsertSQLStatement verwacht wordt uitgevoerd om uit een insert statement de tabellen en kolommen te halen.

Een aantal statement hebben nu nog geen interessante elementen, dit statement resulteren daarom gelijk de dependencies die al gevonden waren. Als het statement type gelijk is aan sstErrorStmt, dan betekent dit dat het statement niet syntaxis correct was. Dit kan komen, doordat de tekst deel uitmaakt van een groter geheel of omdat er echt een fout in de code zit. Als er een fout geconstateerd wordt, dan wordt een boolean genaamd *Error* gezet. Vervolgens wordt het hele programmaobject verwerkt. Als blijkt dat er bij het hele programmaobject nog steeds een fout in zit, dan is dit echt een syntaxis fout en krijgt de gebruiker een foutmelding.

Alleen het meest voorkomende statement types zijn geïmplementeerd. Als een programmaobject item een statement type bevat wat niet geïmplementeerd is, dan wordt er nu een foutmelding weergegeven. Later kan besloten worden of alle mogelijk statements

geïmplementeerd moeten worden of dat de foutmelding weggehaald moet worden, zodat het statement die niet herkend worden ook niet geanalyseerd worden.

```
// every statement has a different fill function.
switch (statement.SqlStatementType)
{
    case (TSqlStatementType.sstInsert):
        return _fillDependency(dependencies, (TInsertSqlStatement)statement, dummy);
    case (TSqlStatementType.sstSelect):
        return _fillDependency(dependencies, (TSelectSqlStatement)statement, dummy);
    case (TSqlStatementType.sstUpdate):
        return _fillDependency(dependencies, (TUpdateSqlStatement)statement, dummy);
    case (TSqlStatementType.sstDelete):
        return _fillDependency(dependencies, (TDeleteSqlStatement)statement, dummy);
    case (TSqlStatementType.sstMssqlSet):
        return _fillDependency(dependencies, (TMssqlSet)statement, dummy);
    case (TSqlStatementType.sstMssqlIf):
        return _fillDependency(dependencies, (TMssqlIfElse)statement, dummy);
    case (TSqlStatementType.sstMssqlBlock):
        return _fillDependency(dependencies, (TMssqlBlock)statement, dummy);
    case (TSqlStatementType.sstRaiserror):
        return _fillDependency(dependencies, (TMssqlRaiserror)statement, dummy);
    // There are no interesting parts identified for these statements (yet)
    case (TSqlStatementType.sstMssqlUse):
    case (TSqlStatementType.sstMssqlGo):
    case (TSqlStatementType.sstMssqlDeclare):
    case (TSqlStatementType.sstMssqlExec):
    case (TSqlStatementType.sstMssqlReturn):
    case (TSqlStatementType.sstBeginTran):
    case (TSqlStatementType.sstMssqlCommit):
    case (TSqlStatementType.sstMssqlRollback):
    case (TSqlStatementType.sstRollback):
        return dependencies;
    // If there is an error, it is not possible to further evaluate
    case (TSqlStatementType.sstErrorStmt):
        {
            if (Error)
                throw (new Exception("Het is niet mogelijk dit statement te parsen: '" +
                        statement.AsText + "' "));
            else
                Error = true;
            return dependencies;
        };
}
// Throw exception if an unidentified statement has been used
throw (new NotImplementedException(statement.SqlStatementType.ToString() + "stmt is: " +
        statement.AsText + " was not implemented"));
```

De Parser

Er kan op twee manieren gebruik gemaakt worden van de SQL representatie die de parser weergeeft. Ten eerste kan gebruik gemaakt worden van de tabellen en kolommen, zoals deze door de parser weergegeven worden. Zo wordt geïdentificeerd wat een tabel is, wat een kolom is en aan welke tabel een kolom hangt. De tweede manier is door een iterator te schrijven, deze iterator voert bij elk element uit de representatie een procedure uit.

Omdat beide methode voor en nadelen hebben, zijn ze allebei gebruikt in andere plekken, zodat voor beide methodes van de sterke punten gebruik gemaakt kan worden. Het gebruik van de tabellen die door de parser al geïdentificeerd zijn, wordt bijvoorbeeld bij de *_fillDependency* procedure voor een select statement gebruikt.

```
foreach (TLzTable tab in statement.Tables)
{
    sourceCodeDependency sTab = null;
    if (tab.OwnerJoin != null)
    {
        // Table is joined with other tables
        …
    }
    else if (tab.OwnerJoinItem == null)
    {
        // Table is a regular table
        …
    }
}
```

In bovenstaande code wordt voor elke tabel uit een bepaald statement gekeken of dit een tabel is die een join heeft met een andere tabel of dat het een simpele from-clause is. Stel dat we de volgende query hebben:

```
select h.description, p.activity_code, h.activity_id
  from hour h
  join project_activity p
    on h.activity_id = p.activity_id
   and h.project_id  = p.project_id
```

De lijst van statement.Tables bestaat dan uit twee elementen, hours en project_activity. Door een foreach-lus te gebruiken kan elke tabel opgeslagen worden. Het is echter niet genoeg om te weten dat deze twee tabellen gebruikt worden, het is ook belangrijk om te weten dat ze aan elkaar gekoppeld zijn. Daarom moet bij *project_activity* aangegeven worden dat de parent *hour* is. Dit is het nadeel van deze methode, het is wel mogelijk om de tokens van de join-items te vinden, maar niet de tabellen.

Hiervoor is de andere methode interessant. Deze methode krijgt tokens en voert per token een procedure uit. In deze procedure kan gecontroleerd worden wat voor een type dit token is. Wanneer het token een tabel (ofwel een `ttObjTable`) is, dan kan deze opgeslagen worden met een referentie naar de parent. Dit is hieronder weergegeven, de procedure krijgt een pVisited item mee, als dit een token is, dan kan dit een interessant element zijn en wordt deze verder verwerkt.

```
if (pVisited is TSourceToken)
{
    TSourceToken token = (TSourceToken)pVisited;
    …
    if (token.DBObjType == TDBObjType.ttObjTable)
    {
        …
    }
}
```

Het zou wel mogelijk zijn om alles in de iterator te contoleren en niet gebruik te maken van de tabellen die al door de parser geïdentificeerd zijn. Dit heeft echter als nadeel dat er in de parser al meer intelligentie zit, dit moet dan allemaal opnieuw gemaakt worden. Zo kan bijvoorbeeld gebruik gemaakt worden van deze intelligentie om de where-clause van een query te bepalen of te bepalen of een veld een subquery is of een toekenning is.

SourceCodeDependency

De gevonden elementen worden bijgehouden in een nieuwe klasse die sourceCodeDependency heet. Deze klasse bestaat uit gegevens en functies. De gegevens zijn dezelfde als opgeslagen worden in de tabel, aangevuld met een prefix. Bij tabellen wordt in de prefix aangegeven wat

de tabelalias was, zo kan gezocht worden naar de juiste tabel die bij een kolom hoort. Dit wordt gedaan met behulp van de functie *isParentTable*. De volgende twee regels zorgen ervoor dat de lijst met dependencies doorzocht worden.

```
findTable = new Predicate<sourceCodeDependency>(value.isParentTable);
tab = dependencies.Find(findTable);
```

De functie *isParentTable* krijgt een dependency mee en controleert of deze de parent is van *value*. De functie Find, die standaard in een lijst gedefinieerd is, zoek de eerst die aan *isParentTable* voldoet. Een dependency voldoet hieraan wanneer de context gelijk is, het objecttype “tab” is en de prefixen gelijk zijn of de prefix van de child leeg is. Dit wordt met onderstaande code weergegeven.

```
// If context is equal, and item is a table
if (_projectId.Equals(d._projectId)
    && _projectVrsId.Equals(d._projectVrsId)
    && _progObjectId.Equals(d._progObjectId)
    && ((_progObjectItemId == null && d._progObjectItemId == null)
        || _progObjectItemId.Equals(d._progObjectItemId)))
{
    if (d._objectType.Equals("tab"))
    {
        if (_prefix == null)
            // if the prefix is empty, then all tables are true
            return true;
        else
            // if the prefixes are equal, then it is the parent table, otherwise not
            return _prefix.Equals(d._prefix);
    }
}
```

In de tabel wordt van een parent alleen de unieke sleutel opgeslagen. In sourceCodeDependency is deze unieke sleutel nog niet bekend (deze moet nog uitgegeven worden bij het opslaan) en wordt daarom een referentie naar de parent (een andere instantie van sourceCodeDependency) opgeslagen. De unieke sleutel wordt gevuld met de functie *storeDependency*. Deze functie roept een stored procedure aan die de database vult. De functie krijgt een datarow en een connectie met de database mee. De velden van deze datarow worden gevuld met de gegevens uit de sourceCodeDependency. Vervolgens wordt onderstaande code uitgevoerd.

```
// stored procedure will add the record and return the new id
conn.ExecuteStoredProcedure("get_new_source_code_dependency_id", ID);

// store the id so it can be used if this item is a parent.
_sourceCodeDependencyId = ID["source_code_dependency_id"].ToString();
```

De stored procedure krijgt de parameters mee die ingevuld moeten worden en insert deze in de tabel *source_code_dependency*. Deze tabel heeft een identifier als unieke sleutel, deze wordt door de SP uitgelezen en weer teruggegeven. Door deze identifier in een variabele op te slaan, kan een andere dependency deze dependency gebruiken als parent. Er moet dan wel voor gezorgd worden dat eerst de parents opgeslagen worden en later pas relaties die hiervan afhankelijk zijn. Dit wordt gedaan, doordat tijdens het toevoegen van dependencies aan de lijst, ook eerst de parent en daarna de child toegevoegd wordt. Als een child een parent heeft waarvan de id onbekend is, dan wordt deze id niet ingevuld, net zoals zou gebeuren wanneer een element helemaal geen parent heeft.

10.2.1 Verdere uitbreidingen

De code extractor vult de source code dependency nu alleen nog met alle tabellen en kolommen. Dit moet nog uitgebreid worden voor taken, taakparameters, domeinen en domein elementen.

De uitbreidingen kunnen gerealiseerd worden door te identificeren in welk type statements deze elementen voor kunnen komen en de specifieke _fillDependency procedure van dit type statement aan te passen. Bijvoorbeeld, als er gekeken wordt naar domeinen, deze worden alleen gebruikt in declaraties, dus dan moeten de declaratiestatements aangepast worden. Vervolgens worden de gedeclareerde variabelen ergens toegekend, dit kan zijn bij een set of een select statement. Op deze manier kan de parser telkens wanneer nodig verder uitgebreid worden.

10.2.2 Keuzes aan de hand van statements

Zoals al eerder beschreven, wordt het code model opgebouwd in één platgeslagen tabel. Er is in de code extractor een keuze gemaakt welke elementen belangrijk waren om op te slaan en hoe deze elementen opgeslagen moesten worden. Dit is gedaan aan de hand van verschillende statements. Er zijn vier soorten statements die geanalyseerd zijn. Dit zijn:

- select
- update
- delete
- insert

Door de voorbeeldstatements en de resultaten in de tabel weer te geven wordt beter duidelijk welke keuzes gemaakt zijn, dan dit zou zijn met het uitleggen van regels code. De statements zijn voorbeelden die uitgevoerd kunnen worden op de workshop database. Het datamodel hiervan is al eerder weergegeven in Figuur 6 op pagina 24.

select

Het eerste statement is een simpel SQL statement.

```
select sum(amount)
  from hour h
 where h.sales_invoice_id = 1
```

Hierin wordt de tabel *hour* gebruikt met de kolommen *amount* en *sales_invoice_id*. De inhoud van de source_code_dependency tabel zal er dan als volgt uitzien.

id	parent	statement type	object type	primary key string	col order no	line no
9421		select	tab	hour	0	2
9422	9421	select	col	amount	0	1
9423	9421	where	col	sales_invoice_id	0	3

Hierbij verwijzen de beide kolommen naar de tabel. De tabel *hour* en kolom *amount* maken deel uit van de select, terwijl *sales_invoice_id* specifiek is voor de where-clause. Bij deze query is de kolom volgnummer niet van belang. Het nummer van de regel is wel weergegeven, zodat de elementen later weer teruggevonden kunnen worden.

Een lastigere query is een query met een join erin. Hieronder is een join gemaakt tussen *hour* en *project_activity*.

```
select h.description, p.activity_code, h.activity_id
  from hour h
  join project_activity p
    on h.activity_id      = p.activity_id
   and h.project_id       = p.project_id
```

Hierbij worden drie kolommen geselecteerd. Twee ervan verwijzen naar *hour* en de derde verwijst naar *project_activity*. De records die deze kolommen representeren, verwijzen daarom ook netjes naar de bijbehorende tabellen. Hetzelfde geldt voor de kolommen die in de join gebruikt worden.

id	parent	statement type	object type	primary key string	col order no	line no
9424		select	tab	hour	0	6
9425	9424	join	tab	project_activity	0	7
9426	9424	join	col	activity_id	0	8
9427	9425	join	col	activity_id	0	8
9428	9424	join	col	project_id	0	9
9429	9425	join	col	project_id	0	9
9430	9424	select	col	description	0	5
9431	9425	select	col	activity_code	0	5
9432	9424	select	col	activity_id	0	5

Update

Een update query wordt gebruikt om de gegevens uit een tabel te wijzigen, wanneer deze gegevens voldoen aan een bepaalde conditie.

```
update hour
   set hourly_rate = 100
 where employee_id = 'esprik'
```

Bovenstaande query verandert van een werknemer het tarief naar 100. Nu worden in de tabel alleen de gebruikte kolommen en tabellen weergegeven, later kan ook de toekenning (100, ‘esprik’) toegevoegd worden bijvoorbeeld voor controles met domeinen. Als aan een kolom dan een waarde gekoppeld wordt die niet in deze kolom opgeslagen mag worden, kan een melding deze situaties eruit filteren.

id	parent	statement type	object type	primary key string	col order no	line no
9433		update	tab	hour	0	11
9434	9433	select	col	hourly_rate	0	12
9435	9433	where	col	employee_id	0	13

Het wordt lastiger wanneer alleen gegevens gewijzigd mogen worden die een relatie hebben met een andere tabel. Om te herkennen uit welke tabellen de gegevens moeten komen, kan een alias gebruikt worden. Deze alias moet dan geüpdate worden, niet de hele tabel.

```
update h
   set hourly_rate       = 100
  from hour h
  join project_activity p
    on p.project_id      = h.project_id
   and p.activity_id     = h.activity_id
 where employee_id       = 'esprik'
```

Daarom is er voor bovenstaand stuk code eerst een alias opgeslagen. De tabel zelf verwijst naar deze alias. Van kolommen die zelf geen alias hebben, wordt verwacht dat deze bij de alias en dus in dit geval bij “*h*” horen.

id	parent	statement type	object type	primary key string	col order no	line no
9436		update	tab	h	0	15
9437	9436	update	tab	hour	0	17
9438	9437	join	tab	project_activity	0	18
9439	9438	join	col	project_id	0	19
9440	9437	join	col	project_id	0	19
9441	9438	join	col	activity_id	0	20
9442	9437	join	col	activity_id	0	20
9443	9436	select	col	hourly_rate	0	16
9444	9436	where	col	employee_id	0	21

Employee en *hourly_rate* hebben geen alias, deze worden dus gekoppeld aan "*h*". *Activity_id* en *project_id* hebben wel een alias en worden dus aan de opgegeven tabel (*hour* of *project_activity*) gekoppeld.

Delete

Delete lijkt erg op update, hier wordt ook een hoofdtabel opgegeven, eventueel gebruik makend van een join. Het verschil met delete is dat daar geen kolommen gezet (set) kunnen worden.

```
delete from hour
 where employee_id <> 'esprik'
```

De inhoud van de source_code_dependency tabel na het verwerken van deze query is daarom kort. De tabel *hour* wordt weergegeven en de kolom *employee_id* die gebruikt wordt om een filter op de selectie te zetten.

id	parent	statement type	object type	primary key string	col order no	line no
9445		delete	tab	hour	0	23
9446	9445	where	col	employee_id	0	24

Wanneer een join gebruikt wordt, moet net zoals bij een update een alias gebruikt worden om de juiste rijen te verwijderen. Hierbij geldt ook weer dat *employee_id* and de alias "*h*" gekoppeld is, terwijl *project_id* en *activity_id* wel aan de juiste tabel gekoppeld zijn.

```
delete h
  from hour h
  join project_activity p
    on p.project_id      = h.project_id
   and p.activity_id     = h.activity_id
 where employee_id       <> 'esprik'
```

id	parent	statement type	object type	primary key string	col order no	line no
9447		delete	tab	h	0	26
9448	9447	delete	tab	hour	0	27
9449	9448	join	tab	project_activity	0	28
9450	9449	join	col	project_id	0	29
9451	9448	join	col	project_id	0	29
9452	9449	join	col	activity_id	0	30
9453	9448	join	col	activity_id	0	30
9454	9447	where	col	employee_id	0	31

Insert

De insert is een stuk lastiger, omdat hiervoor verschillende schrijfwijzen zijn. Niet elke manier mag op elk systeem gebruikt worden. Thinkwise ondersteunt op dit moment klanten die

gebruik maken van Microsoft SQL Server (MSSql) en DB2. Bij MSSql is het mogelijk om een alias toe te kennen aan een selectie. Bij een insert kan dit gebruikt worden om voor de lezer duidelijk te maken in welke kolom de waarde gevuld moet worden, bijvoorbeeld `date = getdate()`. Deze syntax mag bij DB2 niet, hierbij mag een toekenning alleen gebruikt worden in combinatie met een *set* statement en moet een alias gebruikt worden, zoals `getdate() as date`. Omdat Thinkwise ontwikkelt op zowel MSSql, als DB2, wordt de laatste mogelijkheid gehanteerd. Dit is tevens de ANSI standaard. Hieronder staat een insert query die op deze manier weergegeven is.

```
insert into hour
select  getdate()          as date,
        1                  as employee_id,
        1                  as project_id,
        1                  as activity_id,
        100                as amount_of_hours,
        'test'             as description,
        100                as hourly_rate,
        1                  as amount,
        null               as sales_invoice_id,
        null               as insert_user,
        null               as insert_date_time,
        null               as update_user,
        null               as update_date_time
```

Een nadeel van het opgeven van aliassen (en ook van toekenningen) is dat de database hier niets mee doet. Als twee kolommen van volgorde wijzigen, geeft deze manier van weergeven geen foutmelding. Zo kan in bovenstaand voorbeeld *hourly_rate* en *amount* omgedraaid worden, dit zou dan nog steeds syntaxis correct zijn. Terwijl de inhoud van de database hierdoor corrupt wordt.

Een andere keuze is om de kolomnamen toe te voegen, deze kolommen mogen dan wel omgedraaid worden en de select moet dezelfde volgorde aanhouden als de opgegeven kolommen. De kolomnamen komen achter de tabelnaam en voor de select te staan. Dit is bijvoorbeeld zo bij onderstaand statement, dat hetzelfde resultaat heeft als het eerder weergegeven insert statement.

```
insert into hour (activity_id, amount, amount_of_hours, date, description,
                  employee_id, hourly_rate,project_id)
select  1, 1, 100, getdate(), 'test', 1, 100, 1
```

Door het opgeven van kolomnamen, is het behalve velden op een andere volgorde weer te geven ook mogelijk om velden niet te specificeren. Het bewust weglaten van velden heeft echter als nadeel dat er niet altijd een syntaxis foutmelding komt. Bijvoorbeeld in de situatie dat er een nieuwe kolom toegevoegd kan worden, die niet verplicht is. Als dit veld wel gevuld moest worden, komt er geen foutmelding. In tegenstelling tot wanneer kolommen niet gespecificeerd zijn, want dan komt er wel een foutmelding die de ontwikkelaar eraan herinnert dat een kolom nog gevuld moet worden.

id	parent	statement type	object type	primary key string	col order no	line no
9455		insert	tab	hour	0	33
9456	9455	assign	col	getdate()	0	0
9457	9456	assign	alias	date	0	34
9458	9455	assign	col	1	1	0
9459	9458	assign	alias	employee_id	0	35
…	…	…	…	…	…	…
9480	9455	assign	col	null	12	0

9481	9480	assign	alias	update_date_time	0	46

Bij een insert wordt eerst de tabel weergegeven die gevuld wordt. Vervolgens worden alle waardes waarmee deze tabel gevuld wordt aan dit record gekoppeld (in dit geval *getdate()*, *1 …*, *null*). Elke kolom heeft een alias die de kolomnaam bevat (dus *date*, *employee_id, …*, *update_date_time*). Bij alle kolommen wordt weergegeven in de kolom *col_order_no* welk volgnummer ze hebben, op deze manier kan gecontroleerd worden of de volgorde van kolommen goed is. Als de achtste kolom naar de derde plek verschuift, dan kan gecontroleerd worden of aan de derde regel een alias gekoppeld is die overeen komt met de nieuwe kolom.

```
insert into hour
select  getdate()         as date,
        (select top 1 employee_id from employee)  as employee_id,
        p.project_id      as project_id,
        p.activity_id     as activity_id,
        100               as amount_of_hours,
        'test'            as description,
        100               as hourly_rate,
        p.amount          as amount,
        null              as sales_invoice_id,
        null              as insert_user,
        null              as insert_date_time,
        null              as update_user,
        null              as update_date_time
from project_activity p
where p.activity_id = 1
```

De select query van een insert statement kan ook verder uitgebreid zijn. Zo is in bovenstaand voorbeeld een subquery gebruikt om het eerste *employee_id* te selecteren. Er zijn ook een aantal velden uit een andere tabel gebruikt. De subquery is in totaal weergegeven in de *primary_key_string* als invulling die wordt toegekend aan een kolom. Later (in record 9510 en 9511) wordt de subquery geanalyseerd en worden de tabellen en kolommen hiervan als een nieuwe select query opgeslagen.

id	parent	statement type	object type	primary key string	col order no	line no
9482		insert	tab	hour	0	48
9483	9482	insert	tab	project_activity	0	63
9484	9483	assign	col	getdate()	0	0
9485	9484	assign	alias	date	0	49
9486	9483	assign	col	(Select top 1 employee_id From employee)	1	0
9487	9486	assign	alias	employee_id	0	51
9488	9483	assign	col	project_id	2	0
9489	9488	assign	alias	project_id	0	52
9490	9483	assign	col	activity_id	3	0
9491	9490	assign	alias	activity_id	0	53
9492	9483	assign	col	amount_of_hours	4	0
9493	9492	assign	alias	amount_of_hours	0	54
…	…	…	…	…	…	…
9508	9483	assign	col	null	12	0
9509	9508	assign	alias	update_date_time	0	62
9510		select	tab	employee	0	50
9511	9510	select	col	employee_id	0	50
9512	9483	where	col	activity_id	0	64

Doordat er gebruik gemaakt wordt van een andere tabel, worden de kolommen niet aan *hour,* maar aan *project_activity* gekoppeld. *Project_activity* heeft vervolgens weer een relatie met *hour*, zodat duidelijk is dat dit dezelfde query is.

Wanneer er bij de insert statement ook kolomnamen opgegeven worden, dan worden deze kolommen gekoppeld aan *hour*. Hierbij wordt de kolomvolgorde ingevuld, zodat gecontroleerd

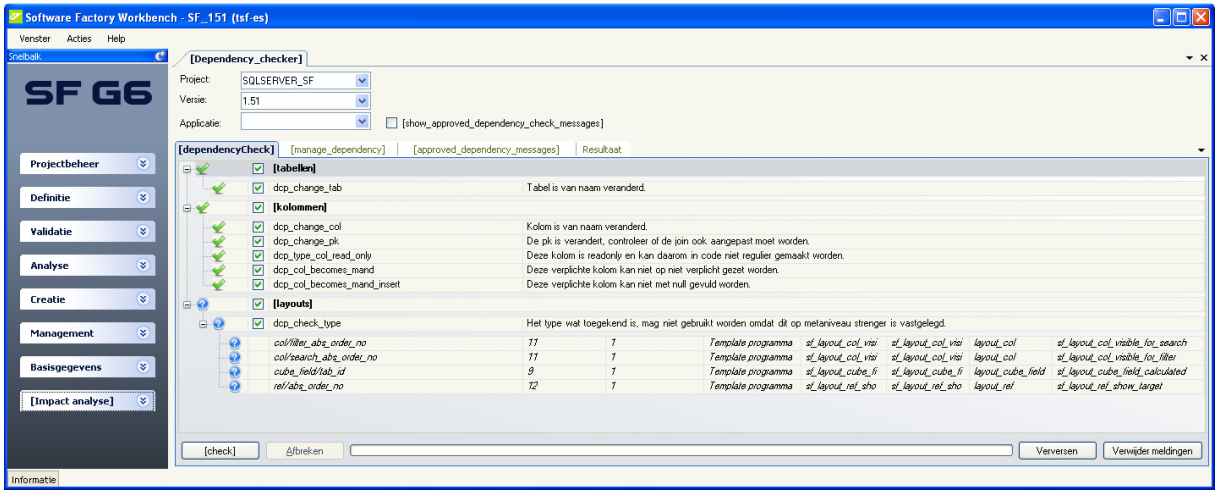
kan worden of men kolommen vergeten is en zodat de waardes aan de kolomnamen gekoppeld kunnen worden om te kijken of er niet onjuiste waardes toegekend worden. Een andere controle die door het opslaan van de kolomnamen uitgevoerd kan worden is de controle of een veld dat verplicht wordt, wel in de lijst staat. Als een kolom niet verplicht is kan deze bewust weggelaten zijn, dus wanneer het wel verplicht wordt, moet de query aangepast worden.

10.3 Dependency checker

De dependency checker is een uniek scherm dat is gekoppeld in het metaniveau van de Software Factory, zodat deze met de uniforme GUI geopend kan worden. De dependency checker is bereikbaar door in de linker balk, genaamd de snelbalk, op het dependency checker item te klikken.

Omdat de dependency checker gebaseerd is op de techniek de gebruikt is voor validaties, kon het scherm ook grotendeels overgenomen worden. Het verschil tussen beide schermen is dat er andere views aangeroepen worden, zodat er andere data weergegeven wordt, namelijk de gegevens uit *dependency_check_msg* en *dependency_check_approved_msg*. Behalve de labels en teksten verschilt dit scherm ook nog van het validatie scherm in de acties die gebeuren wanneer op een melding geklikt wordt. Bij de dependency checker moet dan het stukje code dat gecontroleerd is weergegeven worden. Als blijkt dat dit geen code, maar een parameter is, moet de control procedure die de parameter vult weergegeven worden. Dit scherm zal nog gemaakt moeten worden, maar zonder dit scherm is het prototype ook duidelijk genoeg.

In Figuur 24 is een screenshot van de dependency checker weergegeven. Nadat project en projectversie geselecteerd zijn, wordt in het eerste tabblad alle groepen weergegeven. In dit voorbeeld zijn de groepen *tabellen*, *kolommen* en *layouts* weergegeven. Elke groep heeft een aantal checks, in het voorbeeld heeft de groep *kolommen* de checks: *dcp_change_col, dcp_change_pk, dcp_type_col_read_only, dcp_col_becomes_mand* en *dcp_col_becomes_mand _insert*. Deze checks hebben representatieve records in de tabel dependency check. In deze records staat beschreven welke stored procedures aangeroepen zullen worden om de controle uit te voeren. De naam van de procedure hoeft niet gelijk te zijn aan de naam van de check, omdat deze wordt opgeslagen in de tabel, maar het is gemakkelijker als dit een herkenbare naam is.



Figuur 24: Dependency checker

In Figuur 24 is alles goedgekeurd, behalve de controle voor de layouts. Als een controle meldingen oplevert, dan komen deze onder de check te staan. Elke melding heeft een status, om aan te geven hoe waarschijnlijk het is dat deze melding zorgt voor een aanpassing. Het kan zijn dat een wijziging zo zeker is, dat het eigenlijk automatisch gewijzigd zou kunnen worden. Dit wordt weergegeven met een kruis (). Als een melding minder waarschijnlijk voor een aanpassing zorgt, dan wordt het een uitroepteken () weergegeven. Dit geeft aan dat er wel altijd gewijzigd moet worden, maar hoe het gewijzigd moet worden is niet door de tool te bepalen. De melding waarvan het niet zeker is of dit wel een aanpassing tot gevolg heeft, wordt weergegeven met een vraagteken() en geeft aan dat de ontwikkelaar zelf moet beslissen of deze melding terecht is of niet.

De kracht van deze techniek is dat ontwikkelaars zelf dependency checks toe kunnen voegen. Dit doen ze door in het tweede tabblad een check toe te voegen. Daarna moet een stored procedure geschreven worden die de dependency check msg tabel vult. De check komt vervolgens automatisch in de lijst terecht en wordt automatisch gecontroleerd als de checkbox aan staat en men op de checkknop klikt. Het tweede tabblad bevat een TSFForm en een TSFGrid component. Deze componenten lezen de MSD uit en bouwen aan de hand hiervan automatisch het scherm op met een lijst en daaronder het formulier. In het volgende tabblad kunnen afgekeurde meldingen bekeken worden. Indien de afgekeurde meldingen niet terecht zijn, kunnen ze in dit tabblad weer geannuleerd worden. Wanneer op een melding dubbel geklikt wordt, dan wordt het laatste tabblad weergegeven met daarop een tekstveld die de code weergeeft. Hierin wordt de regel geselecteerd die gewijzigd kan worden. Later kan dit nog verder uitgebreid worden. Dit is anders dan bij het valideren, omdat hierbij een ander standaard scherm geopend wordt waarin het onderwerp dat gewijzigd moet worden centraal staat (bijvoorbeeld het scherm met tabellen of het scherm met kolommen).

10.3.1 Dependency checks

Ondanks dat dependency checks heel gemakkelijk bijgemaakt kunnen worden, moeten er toch al een aantal checks gemaakt zijn, voordat de dependency checker daadwerkelijk in gebruik genomen kan worden. Daarom zijn de volgende dependency checks gemaakt:

- Dcp_change_tab
- Dcp_change_col
- Dcp_change_pk
- Dcp_type_col_read_only
- Dcp_col_becomes_mand
- Dcp_col_becomes_mand_insert
- Dcp_check_type

Deze dependency checks worden hieronder verder uitgelegd.

Dcp_change_tab

Deze dependency check procedure controleert of een tabel van naam gewijzigd is. De procedure die dit uitvoert bestaat uit één insert statement, dit is hieronder weergegeven.

```
insert into dependency_check_msg
select   project_id                = s.project_id,
           project_vrs_id          = s.project_vrs_id,
           dependency_check_id     = @dependency_check_id,
           prog_object_id          = s.prog_object_id,
           prog_object_item_id     = s.prog_object_item_id,
           appl_id                 = @appl_id,
           object_type             = case when s.parameter = 1
                                     then 'poi'
                                          else 'template_prog_object_item'
                                     end,
           primary_key_string      = s.primary_key_string,
           line_no                 = s.line_no,
           new_value_advise        = v.to_tab_id,
           severity                = 0,
           insert_user             = user_name(),
           insert_date             = getdate(),
           update_user             = user_name(),
           update_date             = getdate()
 from source_code_dependency s
 join vrs_control_tab v
   on s.project_id                 = v.project_id
  and s.project_vrs_id             = v.to_project_vrs_id
  and s.primary_key_string         = v.from_tab_id
where s.object_type                = 'tab'
  and v.from_tab_id                <> v.to_tab_id
  and s.project_id                 = @project_id
  and s.project_vrs_id             = @project_vrs_id
```

De check controleert dat als een record uit source_code_dependency van het type “tab” is, de naam van de dependency komt voor in versiebeheer als from_tab_id en het is een record dat van naam verandert (dus de *from* tabelnaam is anders dan de *to* tabelnaam), dan wordt er een melding toegevoegd. Deze melding heeft de sterkst status, deze geeft aan dat de tekst altijd gewijzigd moet worden en dat dit zelfs automatisch kan. In de melding wordt opgeslagen welke tekst dit is en waar deze te vinden is. Als de tekst deel is van een parameter, dan staat de tekst in een programmaobject item (poi) en anders in een template (template programmaobject item). De nieuwe waarde uit versiebeheer kan gebruikt worden om de oude waarde te wijzigen.

Dcp_change_col

De procedure *dcp_change_col* bestaat net als de andere procedures uit een insert die een record toevoegt in *dependency check msg*. Het toekennen van de kolommen met waardes is voor alle procedures ongeveer hetzelfde, daarom worden alleen de interessante onderdelen eruit gelicht. Het belangrijkste van elke procedure is de from/where-clause en deze wordt daarom telkens weergegeven.

De procedure *dcp_change_col* controleert of een kolom die gewijzigd is, nog gebruikt wordt in de code. Als dit het geval is dan moet de code aangepast worden (de melding heeft dus status *automatisch aanpassen*), zodat de oude waarde vervangen wordt met de nieuwe. Een kolom wordt altijd gezocht in de context van een tabel, daarom is een string opgebouwd deze bestaat uit de string van de parent en de kolom.

```
primary_key_string = dbo.get_parent_string(@project_id,@project_vrs_id,
                         s.parent_source_code_dependency_no) + '/' + s.primary_key_string,
```

Zo wordt bijvoorbeeld bij het veld *project_id* uit de tabel *project* de string “*project/project_id*” opgebouwd. Er wordt gekeken naar alle kolommen die van naam wijzigen. Hierbij wordt gekeken of combinatie van de oude tabel en oude kolom in de code voorkomt. Dit heeft als gevolg dat als zowel de tabel als de kolom van naam wijzigt, deze ook tegelijk in de code

aangepast moet worden. In een andere controle kan gekeken worden of een kolom niet aan een tabel hangt die niet meer bestaat.

```
from vrs_control_col c
 join source_code_dependency s
   on s.project_id                = c.project_id
  and s.project_vrs_id            = c.to_project_vrs_id
  and s.primary_key_string        = c.from_col_id
 join source_code_dependency s2
   on s2.project_id               = s.project_id
  and s2.project_vrs_id           = s.project_vrs_id
  and s2.source_code_dependency_no= s.parent_source_code_dependency_no
  and s2.primary_key_string       = c.from_tab_id
where c.from_col_id               <> c.to_col_id
  and s.object_type               = 'col'
  and s2.object_type              = 'tab'
  and s.project_id                = @project_id
  and s.project_vrs_id            = @project_vrs_id
```

Dcp_change_pk

De procedure *dcp_change_pk* controleert of een kolom een primaire sleutel wordt. Als dit het geval is, dan moet voor elke join die met deze tabel gedaan wordt gecontroleerd worden of deze join niet ook dat veld moet bevatten.

In de check wordt gecontroleerd van elke tabel die een join tabel is of aan een join tabel gekoppeld is of dit een brontabel van een relatie is. Daarnaast wordt gecontroleerd of van deze tabel een veld primary key wordt. Voor alle velden die pk worden, wordt daarna gekeken of het veld van niet primary key naar wel primary key verandert en niet in de join-clause voorkomt. Dit natuurlijk weer binnen de context van een project versie.

```
from source_code_dependency s
left outer join source_code_dependency s2
  on s.project_id                         = s2.project_id
 and s.project_vrs_id                     = s2.project_vrs_id
 and ((s.parent_source_code_dependency_no          is null
     and s2.parent_source_code_dependency_no       = s.source_code_dependency_no)
      or s.parent_source_code_dependency_no        = s2.source_code_dependency_no)
join vrs_control_col v
  on s.project_id                         = v.project_id
 and s.project_vrs_id                     = v.to_project_vrs_id
 and s.primary_key_string                 = v.to_tab_id
join col c1
  on c1.project_id                        = v.project_id
 and c1.project_vrs_id                    = v.from_project_vrs_id
 and c1.tab_id                            = v.from_tab_id
 and c1.col_id                            = v.from_col_id
join col c2
  on c2.project_id                        = v.project_id
 and c2.project_vrs_id                    = v.to_project_vrs_id
 and c2.tab_id                            = v.to_tab_id
 and c2.col_id                            = v.to_col_id
join ref r
  on s.project_id                         = r.project_id
 and s.project_vrs_id                     = r.project_vrs_id
 and s.primary_key_string                 = r.source_tab_id
 and s2.primary_key_string                = r.target_tab_id
where s.project_id                        = @project_id
  and s.project_vrs_id                    = @project_vrs_id
  and v.vrs_control_status                = 2
  and c1.primary_key                      = 0
  and c2.primary_key                      = 1
  and (s2.statement_type                  = 4
       or s.statement_type                = 4)
  and s.object_type                       = 'tab'
  and s2.object_type                      = 'tab'
  and not exists (select top 1 1
                    from source_code_dependency d
                   where d.project_id                        = s.project_id
                     and d.project_vrs_id                    = s.project_vrs_id
                     and d.object_type                       = 'col'
                     and d.primary_key_string                = c2.col_id
                     and d.statement_type                    = 4
                     and d.parent_source_code_dependency_no= s.source_code_dependency_no
                 )
```

Deze check controleert of de tabel voorkomt in een relatie en hier een brontabel is, omdat in de doeltabel de primary key niet overgenomen hoeft te worden. Neem bijvoorbeeld de volgende join met de tabellen *hour* en *employee* van de WORKSHOP database.

```
select *
  from hour h
  join employee e
    on h.employee_id = e.employee_id
```

Wanneer er bij *hour* een primaire sleutel veld bijkomt, dan hoeft dit veld niet aan de on-clause toegevoegd te worden. Maar wanneer er een primaire sleutel veld bij *employee* komt, dan moet deze wel in de join toegevoegd worden. Dit komt omdat *hour* de doeltabel is van de relatie tussen *employee* en *hour*, terwijl *employee* de brontabel is.

Dcp_type_col_read_only

De procedure *Dcp_type_col_read_only* controleert of een kolom in code op regulier wordt gezet, terwijl deze in het alleen lezen is. Hieronder wordt weergegeven hoe de procedure eruit ziet. Hiervan is alleen de from/where-clause weergegeven, omdat deze de interessante informatie biedt.

```
…
from source_code_dependency s
 join prog_object p
   on p.project_id                = s.project_id
  and p.project_vrs_id            = s.project_vrs_id
  and p.prog_object_id            = s.prog_object_id
 join vrs_control_col v
   on v.project_id                = s.project_id
  and v.to_project_vrs_id         = s.project_vrs_id
   -- string begint met @ en daarna de kolomnaam en eindigd op _type
  and s.primary_key_string        = '@' + v.to_col_id + '_type'
  and v.to_tab_id                 = p.tab_id
 join col c1
   on c1.project_id               = v.project_id
  and c1.project_vrs_id           = v.from_project_vrs_id
  and c1.tab_id                   = v.from_tab_id
  and c1.col_id                   = v.from_col_id
 join col c2
   on c2.project_id               = v.project_id
  and c2.project_vrs_id           = v.to_project_vrs_id
  and c2.tab_id                   = v.to_tab_id
  and c2.col_id                   = v.to_col_id
where v.vrs_control_status        = 2
  and c1.type_of_col              = 0  -- regulier
  and c2.type_of_col              = 3  -- verborgen
  and exists (select top 1 1
        from source_code_dependency d
       where d.project_id                        = s.project_id
         and d.project_vrs_id                    = s.project_vrs_id
         and d.parent_source_code_dependency_no  = s.source_code_dependency_no
         and d.primary_key_string                = '0')
  and s.project_id                = @project_id
  and s.project_vrs_id            = @project_vrs_id
```

Op metaniveau wordt voor kolommen weergegeven welk type ze zijn. Een kolom kan regulier, alleen lezen of verborgen zijn. Het type kan in de layouts gewijzigd worden, bijvoorbeeld het veld datum_uitdienst, dat onzichtbaar gezet wordt, tenzij de checkbox uit_dienst aangevinkt is. Het mag echter niet zo zijn, dat een veld op metaniveau gedefinieerd wordt als onzichtbaar en vervolgens in de code op regulier gezet wordt. Nu is het zo, dat het metaniveau de layouts overruled, dat wil zeggen dat als het type wat in de code gezet wordt sterker is dan wat op metaniveau aangegeven is, dan wordt de waarde van het metaniveau gebruikt. De code die bijvoorbeeld aangeeft dat een veld regulier moet worden, is onnodig geworden wanneer het veld op metaniveau alleen lezen is en kan weggehaald worden. Het weghalen lijkt geen

consequenties te hebben voor de applicatie, maar zorgt wel degelijk voor een verhoogde onderhoudbaarheid. Als de code niet aangepast zou worden en later wordt besloten dat het veld toch weer regulier moet worden, dan zou de code weer actief worden, terwijl dit niet verwacht werd. Dit zorgt dus voor foutsituaties.

In bovenstaande code wordt gekeken naar alle kolommen die eindigen op _type en beginnen met een apenstaartje (@) en een kolomnaam, bijvoorbeeld *@project_id_type*. Als aan het type een o toegekend wordt, betekent dit dat het veld regulier wordt. Hier wordt gecontroleerd of er een gewijzigde kolom is die van regulier naar verborgen gaat en of deze kolom ergens in de code op regulier gezet wordt.

Dcp_col_becomes_mand

Naast het type, kan ook aangegeven worden of een kolom verplicht is of niet. Ook hiervoor geldt dat een kolom die op metaniveau verplicht is, niet in code zwakker gezet kan worden, dus het is niet mogelijk deze kolom niet verplicht te maken. Als dit in code wel voorkomt wordt het genegeerd en blijft de kolom verplicht.

```
…
   -- string begint met @ en daarna de kolomnaam en eindigd op _mand
  and s.primary_key_string        = '@' + v.to_col_id + '_mand'
…
where v.vrs_control_status        = 2
  and c1.mand                     = 0      -- niet verplicht
  and c2.mand                     = 1      -- verplicht
  and exists (select top 1 1
                from source_code_dependency d
               where d.project_id                         = s.project_id
                 and d.project_vrs_id                     = s.project_vrs_id
                 and d.parent_source_code_dependency_no   = s.source_code_dependency_no
                 and d.primary_key_string                 = '0')
  and s.project_id                = @project_id
  and s.project_vrs_id            = @project_vrs_id
```

Hiervoor wordt ongeveer dezelfde controle gebruikt, alleen wordt nu niet op het woord *'_type'*, maar op *'_mand'* gezocht. Het gaat er nu niet om dat het type verandert, maar de veldverplichting (mandatory). Met andere woorden, als een kolom eerst niet verplicht was en in de volgende versie wel verplicht wordt. Code waarin deze kolom op niet verplicht (o) gezet is, moet aangepast worden.

Dcp_col_becomes_mand_insert

Als een kolom niet verplicht is, kan deze gevuld worden met de waarde *null*. Wanneer vervolgens de kolom op verplicht gezet wordt, mag deze waarde niet meer ingevuld worden. Daarom moet gecontroleerd worden waar *null* ingevuld was. Er moet tevens gecontroleerd worden of de waarde wel toegekend wordt, dit is het geval als bij de insert de kolomnamen opgegeven worden. De kolomnamen die niet opgegeven worden, vult de database engine dan automatisch met de standaard waarde of als deze niet bestaat met null. Echter, dit heeft als nadeel dat wanneer een kolom verplicht wordt en er geen standaard waarde opgegeven is, dit zorgt voor een syntaxisfout. Daarom moet gecontroleerd worden of deze kolom wel aan de insert is toegevoegd. Met onderstaande code wordt gecontroleerd of er een toekenning is die *null* toekent aan een kolom die verplicht is geworden. De controle of de kolom wel bij de opgegeven velden staat, zal later nog geïmplementeerd moeten worden.

```
…
where v.vrs_control_status = 2
and c1.mand = 0
and c2.mand = 1
and s.object_type = 'col'
and s2.object_type = 'tab'
and s2.statement_type = 1
and exists (select top 1 1
         from source_code_dependency d
         where d.project_id = s.project_id
         and d.project_vrs_id = s.project_vrs_id
         and dbo.root(d.project_id, d.project_vrs_id, d.parent_source_code_dependency_no)
                 = dbo.root(s.project_id, s.project_vrs_id, s.source_code_dependency_no)
         and d.line_no = s.line_no
         and d.primary_key_string = 'null')
and s.project_id = @project_id
and s.project_vrs_id = @project_vrs_id
```

Dcp_check_type

De vorige controles waren gebaseerd op een wijziging in het model. Een neveneffect van de impact analyse is dat er ook controles gemaakt kunnen worden die zich alleen richten op het code model en het metamodel. Hierdoor is het mogelijk om te controleren welke tabellen gebruikt worden die niet op metaniveau gedefinieerd zijn of welke kolommen aan een tabel gekoppeld zijn en daarin niet bestaan. In onderstaande code is een dergelijke controle geschreven zonder versiebeheer die het hele model controleert.

```
 from source_code_dependency s
 join prog_object p
   on p.project_id                          = s.project_id
  and p.project_vrs_id                      = s.project_vrs_id
  and p.prog_object_id                      = s.prog_object_id
 join col c
   on c.project_id                          = s.project_id
  and c.project_vrs_id                      = s.project_vrs_id
   -- string begint met @ en daarna de kolomnaam en eindigd op _type
  and s.primary_key_string                  = '@' + v.col_id + '_type'
  and c.tab_id                              = p.tab_id
 join source_code_dependency d
   on d.project_id                          = s.project_id
  and d.project_vrs_id                      = s.project_vrs_id
  and d.parent_source_code_dependency_no    = s.source_code_dependency_no
  and isnumeric(d.primary_key_string)       = 1
  and cast(d.primary_key_string as int)     < c.type_of_col
where s.project_id                          = @project_id
  and s.project_vrs_id                      = @project_vrs_id
```

Deze controle kijkt of er een object is dat eindigt op '*_type*', als dit veld een toekenning heeft dan wordt gecontroleerd of de *primary_key_string* hiervan een nummer is. Als dit een nummer is, dan mag het nummer niet kleiner zijn dan het type kolom. Dit is om dezelfde reden als bij dcp_check_type werd beschreven, dat een sterker metamodel de code overruled.

10.4 Impact analyse op de SF

Om de impact analyse te demonstreren, is deze uitgevoerd op het metamodel van de software factory. Dit metamodel is geüpgrade van versie 1.40 naar versie 1.50 (later versie 1.51). Omdat de wijzigingen in de broncode al gemaakt waren, zijn er niet veel dependency check meldingen gevonden. Er zijn drie verschillende type meldingen gevonden.

De eerste melding is een voorbeeld van een melding die door de ontwikkelaar goedgekeurd moet worden. Het gaat hierbij om een kolom die van naam veranderd is, dus de oude naam zou niet meer voor mogen komen. Dit gaat echter niet op, omdat een andere kolom ook van naam is verandert, naar de nieuwe naam van de andere kolom. De melding geeft aan dat *template_id* van *prog_object_item* verandert moet worden in *control_prog_id*. Terwijl

template_item_id veranderd is in *template_id*. De checks die hieruit voortkomen wijzen op code die doelt op de oude *template_item_id* en niet op de nieuwe *control_prog_id*.

De tweede melding is van de check *dcp_col_becomes_mand*. Het veld *abs_order_no* was in versie 1.40 niet verplicht en is in versie 1.50 wel verplicht. Vervolgens wordt de volgende code gebruikt om de kolom niet verplicht te maken. Deze code geeft niet gelijk een fout, maar het is een bug die later voor een fout kan zorgen en waarvan het netter is om de code te verwijderen.

```
-- Volgnummers van tabblad verplicht indien tabblad wordt getoond.
if @show_target = 1
begin
   set @order_no_type = 0
   set @order_no_mand = 1

   set @abs_order_no_type = 0
   set @abs_order_no_mand = 1
end
else
begin
   set @order_no_type = 2
   set @order_no_mand = 0

   set @abs_order_no_type = 2
   set @abs_order_no_mand = 0
end
```

De laatste melding is van een controle die niet kijkt naar versiebeheer, maar de hele applicatie controleert. Deze controle *dcp_check_type* vond in de software factory vier plekken waarin het type regulier was gezet, terwijl deze kolommen op metaniveau als alleen lezen gedefinieerd waren. Dit zijn:

- *search_abs_order_no* van de tabel *col*
- *filter_abs_order_no* van de tabel *col*
- *tab_id* van de tabel *cube_field*
- *abs_order_no* van de tabel *ref*

Door deze meldingen te verbeteren wordt de onderhoudbaarheid en dus ook de kwaliteit van het metamodel de software fabriek verbeterd. Dit was ook het hoofdzakelijke doel van de impact analyse.

10.5 Multi user

Aangezien de projecten in software zo groot zijn, dat deze niet door één persoon ontwikkeld kunnen worden, is het belangrijk dat ook de impact analyse geschikt is om met meerdere mensen tegelijk aan te werken.

Dit is één van de redenen waarom de (tussen)resultaten van de impact analyse in de database opgeslagen worden. Door het code model op te slaan, kunnen anderen een impact analyse doen zonder eerst een code model te genereren. Door de resultaten van de dependency checker op te slaan, kunnen meerdere mensen deze resultaten bereiken, zonder zelf de controles uit te voeren. Hierdoor kan met meerdere mensen aan verschillende delen van de impact analyse gewerkt worden, zonder dat dit elkaar hindert. Het geeft pas problemen als er tegelijk hetzelfde onderdeel uitgevoerd wordt.

10.5.1 Code extractor

In de code extractor wordt eerst het vorige model weggegooid, waarna het nieuwe model toegevoegd wordt. Stel twee ontwikkelaars proberen een code model te genereren. Deze starten allebei een proces op, deze processen worden proces 1 en proces 2 genoemd. Proces 1 begint met het weggooien van het code model en het toevoegen van een nieuwe code model. Tijdens het toevoegen van dit code model wordt proces 2 gestart. Deze gooit alles weg, dus ook het nieuwe code model van proces 1. Proces 1 gaat verder vanaf dat punt met de wijzigingen die nog niet verwerkt zijn toe te voegen. Wanneer dit klaar is, is er nog geen compleet code model en kan deze ontwikkelaar in de volgende stappen delen van impact missen. Vervolgens komt proces 2 op het deel van het code model dat proces 1 al toegevoegd had. Deze zal een pk foutmelding krijgen, maar het code model is nu wel compleet. Het resultaat is dus dat de ontwikkelaar die als eerste begint met het proces, als deze gelijk door gaat met checken, een onvolledige analyse krijgt. Terwijl de tweede ontwikkelaar wel altijd een volledige analyse ziet, maar dan met foutmeldingen tussendoor.

10.5.2 Versiebeheer

Versiebeheer bestaat uit twee taken, versiebeheer aanmaken en versiebeheer verwijderen. Het zou fout kunnen gaan wanneer iemand versiebeheer aanmaakt, terwijl iemand anders versiebeheer probeert te verwijderen. Aangezien dit beide relatief korte handelingen zijn, moeten ontwikkelaars, wanneer de taken niet de gewenste effecten hadden, de taak gewoon opnieuw uitvoeren.

10.5.3 Dependency checker

Als de ontwikkelaars allebei besluiten direct verder te gaan met de analyse en ook de dependency checker uitvoeren, dan kan deze op verkeerde waardes van het code model of van versiebeheer uitgevoerd worden. Doordat de andere ontwikkelaar verder gaat met vullen, zal uiteindelijk wel een goede versie ontstaan. De eerste ontwikkelaar zal dan of een keer opnieuw moeten checken of opnieuw het scherm openen nadat de tweede ontwikkelaar gecheckt heeft. Als twee ontwikkelaars tegelijk de dependency checker aanzetten, dan geldt net als bij de code extrator dat het beginresultaat van de eerste ontwikkelaar weggegooid wordt door de tweede ontwikkelaar. Het verschil is echter dat de tweede ontwikkelaar bij het toevoegen geen foutmelding krijgt, maar dat de meldingen twee keer in de database opgeslagen worden. Het scherm controleert vervolgens dat elke melding die weergegeven wordt wel uniek is.

10.5.4 Een geïntegreerd scherm

De vorige subhoofdstukken gingen er vanuit dat de onderdelen die voor de impact analyse gebruikt worden, in losse schermen zitten. Doordat het losse onderdelen zijn, is het niet vreemd dat sommige onderdelen opnieuw uitgevoerd moeten worden, omdat ze op dat moment niet meer recent genoeg zijn. Als de impact analyse in één scherm zit en het wordt eenmaal per projectversie uitgevoerd, moet wel zeker zijn dat het resultaat ook volledig is. Hiervoor zal een controle ingebouwd kunnen worden die controleert dat als het proces in werking gesteld wordt, deze niet nogmaals uitgevoerd kan worden. Dit zou gedaan kunnen worden door de hele impact analyse in een transactie uit te voeren. Wanneer de impact analyse in losse delen zit, dan is het lastig als anderen niet verder kunnen werken, omdat een bepaald deel gelockt is. Maar als het in één scherm is, dan is het juist de bedoeling dat er niet aan de impact analyse gewerkt wordt, totdat de gehele analyse klaar is.

11 Validatie

Aan het begin van dit project is een architectuur bedacht. In deze architectuur werden punten aangewezen waar de kwaliteit van software, ontwikkeld met Thinkwise technologie, mogelijk verbeterd zou kunnen worden. Een van deze punten is de validatie. De validatie is ontwikkeld om de elektronische bouwtekening te controleren. Situaties die waarschijnlijk fouten op zullen leveren, worden in de validatie weergegeven. Bijvoorbeeld een tabel waar geen kolommen in zitten. Dit kan geen controle zijn die in de triggers uitgevoerd wordt, omdat bij het toevoegen van een tabel, deze wel leeg mag zijn. Pas bij het genereren is het belangrijk dat de elektronische bouwtekening daadwerkelijk valide is.

De functionaliteit van validatie bestond al. De vraag was echter, hoe deze uitgebreid kan worden. Dit kan op twee manieren, toevoegen van extra validaties en ontwikkelen van controles op basis van impact analyse.

11.1 Extra validaties

Zoals in hoofdstuk 10.3 al beschreven is, kan aan de validator gemakkelijk nieuwe validaties toegevoegd worden. Het enige wat hiervoor gedaan hoeft te worden, is een stored procedure te schrijven die mogelijke foutsituaties in een tabel wegschrijft en een bijbehorend record toe te voegen in de tabel *validation*.

Het is alleen niet altijd even gemakkelijk om de inhoud van de stored procedures te bedenken. Er is al een groot aantal procedures ontwikkeld, waardoor het lastiger is om nieuwe, nuttige validaties te ontwikkelen. Toch zijn er een aantal voorstellen voor nieuwe validaties gedaan. Deze validaties zijn ontdekt doordat er een testdatabase gemaakt is. Een software fabriek is met behulp van een data generator gevuld met random data. Van deze data is vervolgens een testdatabase gecreëerd. Hierdoor kwamen een aantal mogelijke bugs naar voren. Dit kwam omdat 'ontwerpkeuzes' werden gemaakt die normaal gesproken niet zo snel gemaakt worden. Bijvoorbeeld een foreign key over een datum. Sommige situaties waren zo zeldzaam, dat ze waarschijnlijk nooit voorkomen. Hiervoor zijn daarom geen validaties geschreven. Andere situaties zouden nog wel eens voor kunnen komen, daarom ontstonden er ideeën voor nieuwe validaties die deze situaties kunnen herkennen.

Een voorbeeld van een nieuwe controle die ontdekt is, is dat het aantal kolommen dat vastgezet is, niet groter mag zijn dan het totaal aantal kolommen. Het "aantal kolommen vastgezet" is een optie die in een tabel opgeslagen wordt en consequenties heeft op de lijst. In het scherm dat bij die tabel hoort, wordt in een lijst alle gegevens getoond. Soms is het wenselijk dat als naar rechts verschoven wordt, dat de kop nog wel steeds zichtbaar blijft. Als de kop de eerste kolom is, dan kan de optie op 1 gezet worden, zodat deze ene kolom altijd zichtbaar blijft, zelfs als de gebruiker naar rechts in de lijst schuift. Het aantal mag natuurlijk nooit meer zijn dan het aantal kolommen dat in de lijst staat. Hier kan geen trigger opgezet worden omdat een kolom dan niet verwijdert mag worden, als het aantal kolommen toevallig gelijk is aan het aantal kolommen dat vastgezet is. Terwijl ontwikkelaars misschien eerst een aantal records weg zouden willen gooien en daarna pas het aantal kolommen vastzetten. Door achteraf te controleren of het aantal goed staat ingesteld, kunnen fouten in de eindapplicatie voorkomen worden.

11.2 Debug tool

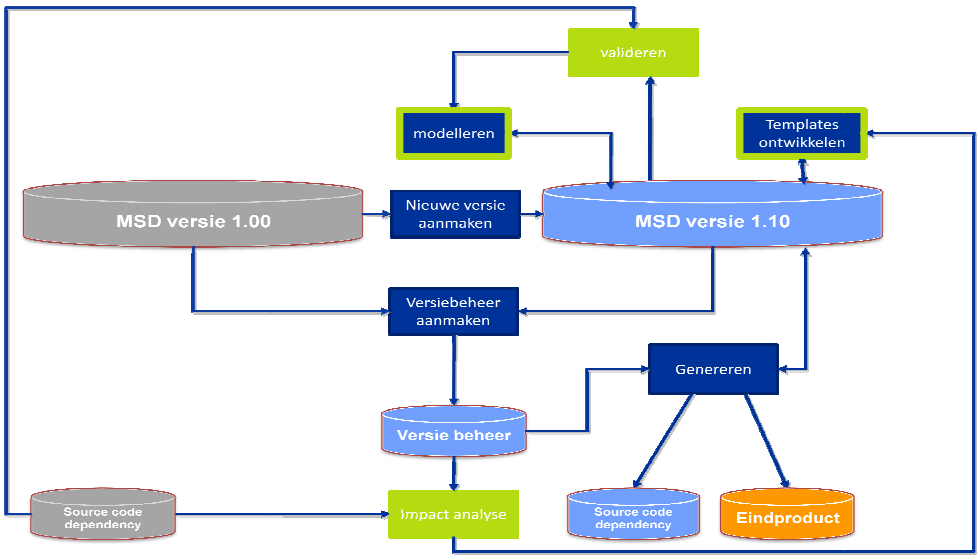
Voor de impact analyse wordt voor elk project een code model gemaakt. Aan de hand van dit code model en de wijzigingen in versiebeheer worden vervolgens controles gedaan. Sommige van deze controles zouden ook uitgevoerd kunnen worden zonder dat er een wijziging, opgeslagen in versiebeheer, aan vooraf gaat. Bijvoorbeeld de controle dat als een kolom van type veranderd, dat deze dan niet meer aan een minder sterk type gekoppeld mag worden. Deze controle zou goed zijn om overal uit te voeren (zie Dcp_check_type uit hoofdstuk 10.3.1).

Op deze manier kunnen nog meer interessante controles toegevoegd worden. Zo kan bijvoorbeeld gecontroleerd worden of er in een tabel ook kolommen aangeroepen worden, die niet bestaan. Er kan zelfs gecontroleerd worden of er tabellen gebruikt worden in de code, die op metaniveau niet bestaan.

Het voordeel van deze controles is, dat ze uitgevoerd kunnen worden voordat het eindproduct gecompileerd is. Fouten worden daardoor nog vroeger in het ontwikkelproces opgespoord. Er kan ook gecontroleerd worden welke tabellen uit een ander tabellen gebruikt worden, bijvoorbeeld voor de koppeling tussen verschillende systemen. Deze koppeling wordt hierdoor ook beter onderhoudbaar.

11.3 Resultaat

In **Fout! Verwijzingsbron niet gevonden.** is de validatie aan het kwaliteitsproces oegevoegd. Het valideren heeft als input het MSD en zorgt voor input bij het modelleren. Een onverwacht resultaat dat in dit deel ontdekt is, is dat het ook mogelijk was om te valideren op basis van de source code dependency. In plaats van het vergelijken tussen source_code_dependency en versiebeheer zoals dit bij impact analyse gebeurd, kunnen controles geschreven worden die vergeleken tussen de elektronische bouwtekening (MSD) en de source code dependency (debug tool).



Figuur 25: Kwaliteitproces IV

12 Test tools

Kwaliteit kan gemeten worden door te testen. Door de fouten die uit de testen komen, op te lossen, wordt de kwaliteit van de software beter. Het testen kan handmatig, maar er zijn ook technieken om automatisch te testen. De oudste manier van testen is handmatig alle mogelijkheden van de software uit te voeren en het resultaat te controleren..

De software factory van Thinkwise zorgt ervoor dat er anders getest kan worden dan bij normale applicaties. Bij traditionele software applicaties wordt elk scherm handmatig gemaakt. De programmeur sleept invoervelden en knoppen op het scherm en deze worden op de juiste plek uitgelijnd. Vervolgens worden alle knoppen gekoppeld aan een dataset zodat deze gevuld worden met de gegevens uit de database. Als laatst moet achter elke knop functionaliteit geschreven worden. Elk scherm moet daarom ook getest worden om te controleren of elke knop en elk veld dat op het scherm gezet is, werkt zoals bedoeld.

Bij Thinkwise worden de schermen automatisch opgebouwd. Dit betekent dat niet elk scherm meer getest hoeft te worden. Het scherm wordt immers altijd op dezelfde manier automatisch opgebouwd. Het automatisch opbouwen van schermen moet wel getest worden. De functionaliteit van de applicatie is voornamelijk geprogrammeerd op de database. Zo zijn er twee aparte onderdelen van testen ontstaan. Ten eerste moeten de GUI's (dit is het automatisch opbouwen van schermen) getest worden. Ten tweede moet het eindproduct van een klant getest worden, waarbij het vooral gaat om de business rules die specifiek voor dat project geschreven zijn.

Zowel het testen van GUI's als het testen van business rules is erg belangrijk. Als er in een GUI een fout gevonden wordt, dan is de impact hiervan groter dan bij een fout in de business rules van één project. Alle klanten die deze GUI gebruiken, kunnen de fout dan tegenkomen, terwijl fouten die gevonden worden in de business rules alleen maar gelden voor dat betreffende project. Door een testtechniek op te zetten die bij de software fabriek past, is het echter wel mogelijk er gelijk voor te zorgen dat er in alle project aandacht aan het testen van de applicatie besteed wordt.

In dit onderzoek is ervoor gekozen om op de business rules te richten. Deze zijn voor het onderzoek interessanter omdat de business rules op de database zitten en database testen is minder bekend dan User interface testen. Ook wordt met het ontwikkelen van een tool voor het testen van business rules grotere delen in de klantproject geraakt, zodat de impact van de resultaten ook groter is.

Dit neemt niet weg dat het testen van de GUI's ook erg belangrijk is. Dit zal in een vervolgproject onderzocht moeten worden. Er moet dan een testproject gemaakt worden, waar de meeste situaties die ook bij klanten kunnen voorkomen in opgeslagen zijn. Verder zal er een testtechniek met een testtool gekozen moeten worden waarmee getest kan worden. Vervolgens zal er begonnen moeten worden met het testen van de GUI's en het opslaan van de resultaten.

12.1 Business rules

In een project worden business rules geschreven, die getest moeten worden. Deze business rules worden geschreven in SQL (Structural Query Language). Business rules die zorgen voor

de integriteit van de database worden uitgevoerd in de triggers. De business rules die gebruikers sturen in wat wel en wat niet ingevuld mag worden, worden weergegeven in defaults en layouts. Naast triggers, defaults en layouts zijn er ook nog stored procedures en functies die gebruikt kunnen worden en getest zullen moeten worden.

12.1.1 Databases testen

Chan en Cheung [31] geven aan dat het testen van database applicaties opgedeeld kan worden in drie subproblemen: genereren van test cases, voorbereiden van test data en verificatie van de uitkomst. Ze behandelen twee verschillende type technieken die gebruikt worden bij testen van applicaties, *black box* en *white box*. De *black box* techniek wordt vooral gebruikt om te controleren of een applicatie voldoet aan de specificaties. Voorbeelden van black box technieken zijn *Equivalence Partitioning*, *Boundary Value Analysis* en *Cause-effect Grahping*. Het nadeel van black box testen is dat men niet weet hoeveel van het systeem getest is. Sommige type fouten worden door deze techniek niet gevonden. Een voorbeeld dat Chan en Cheung geven is dat wanneer een aantal functies in een vaste volgorde goed werken, dit niet hoeft te betekenen dat dit in een andere volgorde ook het geval is.

Bij *white box* technieken wordt gekeken welke code er tijdens het uitvoeren van de testen gebruikt wordt. Het doel is een zo groot mogelijk percentage van de code te testen en voor de code die niet getest is alsnog testcases te ontwikkelen. Voorbeelden van white box technieken zijn *statement testing*, *branch testing*, *condition coverage* en *path testing*. Volgens Chan en Cheung is deze techniek niet bruikbaar, omdat SQL ingebakken is in programma's en vervolgens slechts één keer uitgevoerd worden. Chan en Cheung geven aan dat normale testtechnieken niet van toepassing zijn op database applicaties en beschrijven de techniek WHODATE (WHite bOx Database Applicatie TEsting). Daarbij wordt de ingebakken SQL statements omgezet naar GPL (general-purpose programming language). Vervolgens kunnen er normale testtechnieken op de GPL code uitgevoerd worden.

In het vervolgwerk van Cheung met Zhang en Xu [32] wordt nog meer de nadruk gelegd op het testen van database applicaties. Zoals wij in het onderzoek geïnteresseerd zijn in de database die door de applicatie gebruikt wordt, ligt bij Zhang et al de nadruk op de applicatie die de database gebruikt. Ze ontwikkelen een tool waarbij het schema, het SQL statement en de benodigde bewering (bijvoorbeeld "er moet een record bestaan" of "er mogen geen records bestaan") worden meegegeven, zodat constraints ontstaan die bestaande tools kunnen gebruiken om een database met testdata te vullen. Vervolgens kan met behulp van deze testdatabase de applicatie die ervan gebruik maakt getest worden. Dit is voor Thinkwise niet geschikt, omdat er bij de software fabriek gebruik gemaakt wordt van een generieke GUI met algemene componenten. Afhankelijk van de gegevens uit de MSD worden verschillende queries opgebouwd. Het belangrijkste is dus dat de database in een project getest wordt, terwijl Zhang et al de applicatie wil testen, eventueel met behulp van een onderliggende database.

Chays, Dan, Frankl, Vokolos en Weyuker [33] definiëren testen van een programma als het uitvoeren van dat programma met geselecteerde invoer om te controleren of het resultaat overeen komt met de gespecificeerde waardes. Het verschil tussen gewoon testen en het testen van database applicaties, geven ze aan, is dat bij database applicaties ook een instantie van een

database als input gegeven moet worden. Als resultaat moet ook gecontroleerd worden of de nieuwe instantie van de database voldoet aan de specificaties. In het artikel richten Chays et al zich vooral op het vullen van de database met testdata. Bij meerdere testen worden er niet meerdere instanties van een database meegegeven, maar wordt in de instantie meerdere gebruikers aangemaakt waarvoor verschillende testinstellingen gelden.

Chays et al baseren zich op category-partition testing, een black box techniek. Hierbij identificeert de tester belangrijke categorieën van data. Voor elke categorie worden voorbeeld waardes gegeven en deze worden opgedeeld in partities. Een tool kan vervolgens testdata genereren door verschillende waardes te combineren. De tester kan met behulp van constraints sturen in de gebruikte data.

Tuya, Dolado, Suárez-Cabal en de la Riva [34] behandelen de white box techniek van Suárez-Cabal en Tuya [35]. Tuya et al willen onderzoeken of er effectievere test cases gemaakt kunnen worden wanneer de tester gebruik maakt van een coverage criterium. Met een coverage criterium wordt gekeken of alle criteria de verschillende type waardes hebben gehad. Een criterium kan true, false of null zijn. Met null wordt de NULL bedoeld die in een database gebruikt wordt om aan te geven dat er geen waarde bekend is. De coverage wordt uitgerekend door een coverage tree te maken van de JOIN en WHERE condities in de query. Elke conditie wordt gerepresenteerd met een level uit de boom. Voor elke evaluatie zijn er zes situaties, Fl, Fr, T welke respectief staan voor false van links naar rechts, false van rechts naar links en true en Nl, Nr en Nb waarbij de waarde voor het linker, rechter of beide kant van de vergelijking onbekend (null) is. Als alle mogelijke situaties getest zijn, is er een condition coverage van 100%. Het resultaat is dat als een coverage techniek gebruikt wordt, er effectievere testen gemaakt worden waarbij meer SQL gerelateerde fouten opgespoord worden.

Ambler [36] geeft aan dat wanneer een database getest wordt, er onderscheid gemaakt kan worden tussen het testen van de interface (black box) en het testen van de functionaliteit in de database (Clear box of white box). Als voorbeeld kijkt Ambler naar hoe de data in de database opgeslagen wordt, hoe de data geëxtraheerd wordt en hoe de database met de applicatie samenwerkt. Als clear-box testing worden de voorbeelden gegeven van stored procedures/functies, triggers, views, constraints, etc. De black box technieken van Ambler zijn voor Thinkwise niet erg interessant, omdat het opslaan en extraheren van data door een database engine (SQL Server of DB2) afgehandeld worden en de interface met de GUI in de GUI afgehandeld (en getest) wordt. Het interessante aan het opslaan van data is of de triggers die afgaan de juiste wijzigingen op de data aanbrengen. Dit wordt ook met de clear-box techniek getest tijdens het testen van de triggers. Ambler geeft een aantal tools die gebruikt kunnen worden voor unit testen, load testen (tools die een zware druk op de database simuleren) en test data generatie. Met unit test kunnen functies en stored procedures getest worden, zoals normale Java functies en procedures ook getest zouden worden.

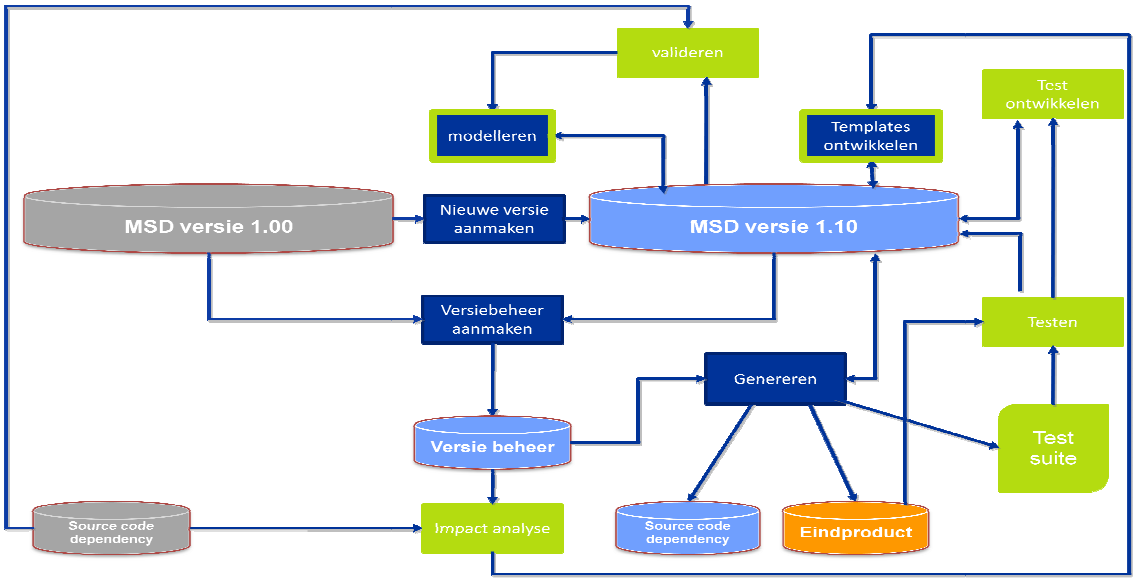
12.1.2 Resultaat

Er zal nog meer onderzoek gedaan moeten worden, naar welke techniek het beste gebruikt kan worden om de database te testen. Wel is duidelijk dat deze testcases zullen bestaan uit een aantal regels code. Het zou het mooiste zijn als de code die geschreven wordt om templates te testen zelf ook in templates opgeslagen kan worden. Zo kan een testsuite (dit zijn een aantal

testcases bij elkaar) gegenereerd kan worden. Met het testen wordt dan ook de kracht van de software fabriek behouden, namelijk dat je een stuk code maar één keer schrijft en dit vervolgens automatisch op verschillende plekken terecht komt.

Het testen van databases kan uitgevoerd worden door de verschillende objecten aan te roepen met een vaste database opstelling en te controleren of het resultaat klopt. Zo kan een soort unit test geschreven worden, waarbij de unit één stored procedure of één tabel is. Deze wordt aangeroepen (of gevuld) met een aantal parameters en vervolgens moet in de database gecontroleerd worden of de situatie veranderd is zoals verwacht.

In Figuur 26 wordt weergegeven hoe dit uitgewerkt zou moeten worden. Met behulp van het MSD worden test templates geschreven. Deze worden tijdens het genereren gebruikt om een test suite te maken. De test suite wordt weer gebruikt tijdens het testen van het eindproduct. De resultaten hiervan moeten weer in de elektronische bouwtekening opgeslagen worden. Tevens kunnen de resultaten gebruikt worden om nieuwe test templates te ontwikkelen.



Figuur 26: Kwaliteitproces V

13 Conclusie en aanbevelingen

Aan het begin van dit onderzoek is de vraag gesteld hoe de kwaliteit van de elektronische bouwtekening verhoogd kan worden, zodat het eindproduct, geautomatiseerd gebouwd met Thinkwise technologie, van betere kwaliteit is. De kwaliteit kan op verschillende manieren verhoogd worden. Daarom zijn er vijf deelvragen gesteld die ervoor gezorgd hebben dat er structuur in het onderzoek zat. Deze vragen zijn gebaseerd op de architectuur die bij de probleemstelling beschreven is. Door de deelvragen te beantwoorden zijn er verschillende manieren beschreven waarop de kwaliteit van de bouwtekening verhoogd kan worden.

13.1 Verbetering van het kwaliteitsproces

Om de kwaliteit van de software stabiel te houden, moet een kwaliteitsysteem toegepast worden. Daarom is er gekeken naar het kwaliteitsproces. Dit is gedaan om antwoord te kunnen geven op de volgende deelvraag.

Hoe kan de kwaliteit van een eindproduct, dat gebouwd wordt met Thinkwise technologie, gemanaged worden?

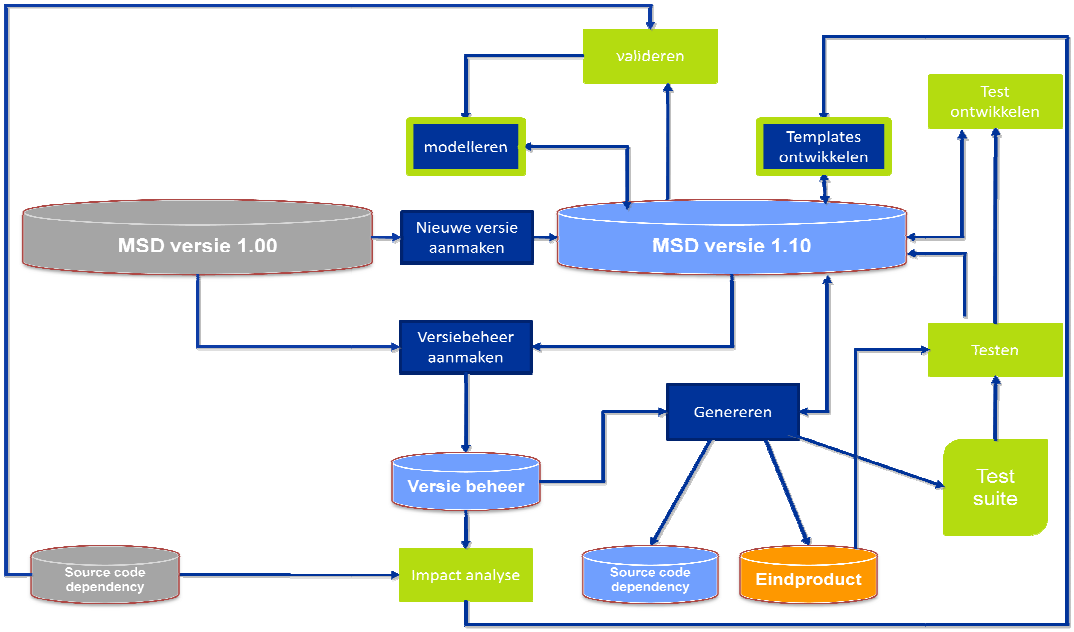
Om de kwaliteit te kunnen managen zal een kwaliteitssysteem opgezet moeten worden. Hiervoor moet de kwaliteit gemeten worden. Voor elke kwaliteitseigenschap die gemeten wordt, moeten de gegevens geanalyseerd worden op basis van een norm. Als een project niet aan de norm voldoet, dan moeten er maatregelen genomen worden, zodat het project er wel aan gaat voldoen. Daarnaast moet periodiek gekeken worden of de norm wel realistisch is en kan de norm waar nodig aangepast worden.

Om kwaliteit te meten is gekeken naar een aantal kwaliteitstandaarden. Een deel van deze standaarden geeft aan hoe kwaliteit beschreven of gemeten kan worden. Een ander deel geeft aan hoe het kwaliteitsproces verbeterd zou kunnen worden. De kwaliteitstandaarden zijn vooral erg formeel. Er is veel papierwerk en tijd mee gemoeid en de opbrengsten zijn niet gelijk terugverdiend. De filosofie achter de standaarden is dat het kwaliteitsproces in kaart gebracht moet worden. Voor een software bedrijf, zoals Thinkwise, zal eerst onderzoek gedaan moeten worden om te kijken of er vraag is naar certificering. Als potentiële klanten op zoek zijn naar een leverancier die gecertificeerd is, dan is het verstandig om een kwaliteitstandaard te gebruiken. Het voordeel van een kwaliteitsstandaard is wel dat de processen goed in kaart gebracht (moeten) worden. Dit zorgt ervoor dat de onderneming het proces kan beheren en verbeteren.

Een manier om kwaliteit te meten, is door de software te testen. Er zijn verschillende aspecten van kwaliteit die getest kunnen worden. In dit onderzoek is vooral aandacht besteed aan functionaliteit en onderhoudbaarheid. De testresultaten moeten opgeslagen worden zodat deze geanalyseerd kunnen worden. Hiervoor is in dit onderzoek een datamodel gemaakt. Dit kan toegevoegd worden aan het metaniveau van de software fabriek zodat in de software fabriek opgeslagen kan worden welke testobjecten wanneer getest zijn.

13.2 Kwaliteitsverbetering in het ontwikkelproces

Naast dit kwaliteitsysteem zijn er ook een aantal andere manieren onderzocht om de kwaliteit te verbeteren. De groene onderdelen in Figuur 26 geven de plekken aan waarop de kwaliteit verbeterd wordt.



Figuur 27: Kwaliteitproces

13.2.1 Protocollen

"Voorkomen is beter dan genezen", dit gezegde representeert hetgeen dat in deze subvraagstelling naar boven moest komen. Namelijk, slechte kwaliteit moet ergens ontstaan, daarna kan het weer opgelost worden, maar beter zou zijn om te voorkomen dat het ontstaat. Hierdoor is de volgende deelvraag ontstaan:

Hoe kunnen protocollen ervoor zorgen dat tijdens de invoer van de elektronische bouwtekening slechte kwaliteit vermeden worden?

Protocollen zijn gedragsvoorschriften, die ervoor zorgen dat acties op een eenduidige manier uitgevoerd worden. Er zijn drie plekken geïdentificeerd waar protocollen onderzocht konden worden. Dit zijn in de Meta Solution Definition (MSD), bij de control procedures en code templates en in de GUI.

Voor de GUI is geen manier gevonden om protocollen te introduceren. Dit komt omdat deze manier van ontwikkelen het meeste lijkt op de traditionele manier van ontwikkelen. Hiervoor worden dus traditionele methodes, zoals compont based development, en bestaande ontwikkelomgevingen gebruikt. De MSD gebruikt al protocollen, deze kunnen verder uitgebreid worden, door het gebruik van de SF technieken verder door te trekken. Dit zijn modelprotocollen (zoals primaire sleutels, relaties met integriteit controles), database protocollen (zoals maximale en minimale waarde, trigggers, identity), SF protocollen (zoals layouts, defaults, processtappen) en projectafhankelijke protocollen. De protocollen van control procedures en code templates zijn normaal gesproken te realiseren in een

stijldocument. Dit is echter statische en ontwikkelaars moeten vervolgens zelf de opmaak regelen, daarom is gekeken hoe de opmaak automatisch aangepast kan worden. Dit kan toegevoegd worden aan de huidige tools van de software fabriek.

13.2.2 Impact analyse

De deelvraag over impact analyse bleek in dit onderzoek het meeste potentie te hebben en is daarom naar voren in het onderzoek geschoven. Dit heeft invloed gehad op de manier waarop de deelvraag over validatie (in het volgende subhoofdstuk) is beantwoord. De derde deelvraag is als volgt geformuleerd.

Hoe kan change impact analyse gebruikt worden, om ervoor zorgen dat aanpassingen in de elektronische bouwtekening en het eindproduct kwalitatief beter zijn?

De change impact analyse is eerst bestudeerd in de literatuur. Vervolgens is er een ontwerp ontwikkeld, om een impact analyse voor Thinkwise te maken. Met deze impact analyse is het makkelijker om een systeem, gemaakt met de software fabriek, te onderhouden, waardoor de algemene kwaliteit van het systeem verbetert. Tussen twee versies is het mogelijk een impact analyse uit te voeren die aangeeft welke stukken code moeten veranderen en waarom deze code niet meer correct zou kunnen zijn.

Voor de impact analyse is een prototype gemaakt. Dit is in drie stappen gedaan. Ten eerste is versiebeheer verder uitgebreid zodat hierin opgeslagen wordt welke modelwijzigingen er tussen de twee versies zitten. Daarna is met behulp van een parser functionaliteit gemaakt die de belangrijkste elementen uit de code haalt en in een code model in de database opslaat (de code extractor). Als laatste is er een tool gemaakt waarmee geanalyseerd kan worden welke wijzigingen in de code voorkomen en aangepast moeten gaan worden.

Voordat het prototype ingezet kan worden, zal door het management een keuze gemaakt moeten worden hoe de impact analyse neergezet wordt. Er kan gekozen worden voor een impact analyse die bestaat uit één tool. Deze tool bevat alle onderdelen die nodig zijn om een analyse van begin tot eind uit te voeren. Het is daarbij de bedoeling dat deze analyse eenmalig uitgevoerd wordt tussen twee versies in.

Aan de andere kant is er de keuze om de impact analyse frequenter uit te voeren. Een ontwikkelaar kan de impact analyse dan op elk moment in het traject uitvoeren en nadat er wijzigingen gemaakt zijn, opnieuw controleren wat er uit de analyse komt. Dit heeft als gevolg dat niet alles in één tool weergegeven wordt, maar op verschillende plekken. De impact analyse wordt dan opgedeeld in versiebeheer, code extraction en dependency checken.

13.2.3 Validatie

Thinkwise maakt gebruik van een elektronische bouwtekening. Als deze bouwtekening getest zou kunnen worden, dan kunnen fouten eerder in het ontwikkelproces ontdekt worden. Daarom wordt de elektronische bouwtekening gevalideerd. Bij het valideren wordt er gekeken naar situaties die waarschijnlijk voor fouten zorgen. Het valideren wordt gedaan door de validator. Deze is al eerder gemaakt en kan in projecten gebruikt worden. De vraag die in dit onderzoek gesteld was, is dan ook:

Hoe kan de bestaande validatie uitgebreid worden, zodat slechte kwaliteit eerder in het proces zichtbaar wordt?

Er is in dit onderzoek onderscheid gemaakt tussen twee manieren om de validatie uit te breiden. De ene manier is het toevoegen van nieuwe validaties aan de validator. De andere manier is door technieken te ontwikkelen die ook foutsituaties opsporen, maar dit op een andere manier doet dan de validator. Met behulp van een testdatabase zijn een aantal validaties ontdekt, die toegevoegd kunnen worden om de validator uit te breiden.

De tweede manier, het ontwikkelen van nieuwe technieken waarbij gevalideerd wordt, is toegepast door het code model van de impact analyse te gebruiken. Vervolgens kan met controles het code model afgezet worden tegen de elektronische bouwtekening. Als deze niet overeenkomen, kan besloten worden welk model niet klopt en waar dit model aangepast moet worden. Zo kan bijvoorbeeld gecontroleerd worden welke tabellen in code aangeroepen worden, maar in het metaniveau niet bestaan. Deze techniek heeft de naam debug tool gekregen, dit omdat ermee eerder in het ontwikkelproces bepaalde situaties gedebugd kunnen worden.

13.2.4 Testen

De vorige deelvragen worden eerder in het ontwikkelproces gebruikt. Deze kunnen veel fouten detecteren, maar toch zal er op het einde getest moeten worden. Uit de eerste deelvraag is al gebleken dat niet alles getest kan worden. Er moet daarom gezocht worden waar het meest waarschijnlijk is dat er bugs gevonden worden. Daarom moet er rekening gehouden worden met de Thinkwise technologie en is dit onderzocht met de volgende deelvraag:

Hoe kan het eindproduct, dat gebouwd wordt met Thinkwise technologie, getest worden?

Het testen van de software fabriek kan opgedeeld worden in twee onderdelen. Ten eerste moet de GUI's die gebruikt worden, getest worden. Deze worden door alle klanten gebruikt, daarom hebben fouten in de GUI's een groter impact. Ten tweede moet het eindproduct voor elke specifieke klant getest worden. Hiervoor moeten de business rules, die opgeslagen zijn in triggers, stored procedures en taken, getest worden.Het testen van de GUI's is niet onderzocht binnen dit project en zal in een vervolgonderzoek gerealiseerd moeten worden.

Het testen van business rules is anders dan het testen van traditioneel geprogrammeerde applicaties. Bij traditionele applicaties staat alle SQL die gebruikt wordt van tevoren in de code opgeschreven. Er hoeft dus alleen voor deze statements getest te worden. Omdat de database elke keer anders aangeroepen wordt, kan niet vanuit de GUI's getest worden. Het gaat namelijk juist om de database zelf.

Er zal eerst nog meer onderzoek gedaan moeten worden naar automatisch testen voordat dit bij Thinkwise ingezet kan worden. Het concept dat in dit project bedacht is, zal dan verder uitgewerkt moeten worden. Het is namelijk zo dat dan test cases geschreven kunnen worden die bestaan uit templates. Zo kan de testsuite, net zoals het eindproduct, gegenereerd worden. Als er nieuwe templates op een tabel komen, dan staan er ook gelijk nieuwe testregels in de testsuite. Hiervoor zal eerst uitgezocht moeten worden welke tools en welke technieken

hiervoor het beste gebruikt kunnen worden. Vervolgens kunnen er in projecten testen geschreven worden, zodat de kwaliteit van het eindproduct gewaarborgd kan worden.

13.3 Aanbevelingen

In dit onderzoek is gekeken hoe de kwaliteit verbeterd kan worden. De resultaten die gevonden zijn, zijn niet de enige manieren waarop de kwaliteit verbeterd kan worden. Dit is slechts een eerste opzet die zorgt voor de bewustwording van kwaliteit. Het is daarna aan het bedrijf om de kwaliteitsbeheer op te pakken en verder door te voeren. Hiervoor worden de volgende aanbevelingen gedaan:

- Uit het onderzoek is gebleken dat procesbeschrijvingen kunnen helpen met het verbeteren van het kwaliteitsproces. Er zijn echter geen procesbeschrijvingen geschreven, dit valt buiten de scope van het onderzoek. Aangezien het bedrijf steeds groter wordt, zou het wel verstandig zijn deze procesbeschrijvingen te gaan schrijven. Wanneer processen beschreven zijn, zorgt dit voor een kortere inwerktijd van nieuwe medewerkers. Daarnaast krijgen de huidige medewerkers een beter idee hoe anderen in het bedrijf werken en kunnen hiervan leren. Zo worden fouten niet telkens opnieuw gemaakt in verschillende projecten bij verschillende ontwikkelaars.
- De eerste stap bij het invoeren van een kwaliteitsysteem is het opslaan van (handmatige) testen. Dit geeft beter inzicht in hoe getest is en welke delen meer aandacht nodig hebben. De testen worden opgeslagen in nieuwe tabellen in het MSD. Doordat dit met een software fabriek bereikt kan worden, kunnen kubussen gebruikt worden om de juiste managementinformatie eruit te halen. Op deze manier kan geanalyseerd worden of in het project aan een eigen opgestelde standaard voldoet en kan het management periodiek beslissen of de standaard goed is of aangepast moet worden.
- Voor protocollen is het belangrijkste punt de projectafhankelijke afspraken. Deze zullen in eerste instantie in een document bijgehouden kunnen worden. Later kan besloten worden dit geautomatiseerd te doen. In een nieuwe tabel kunnen verschillende eigenschappen van een project opgegeven worden. Deze eigenschappen kunnen later uitgelezen worden, zodat de controles alleen af gaan bij protocollen die van toepassing zijn in het project waar de tabel gebruikt wordt. De huidige protocollen in de software fabriek zouden verder uitgebreid moeten wordt, zodat deze het bedrijfsproces van Thinkwise, namelijk software ontwikkelen, nog meer ondersteunen.
- Het automatisch opmaken van de code kan bij het scherm dat de templates en control procedures beheert toegevoegd worden. Ontwikkelaars kunnen vervolgens zelf beslissen wanneer ze de opmaak toepassen. Voor het ontwikkelen van automatische opmaak is een parser gebruikt. De aanroep van deze parser zal eerst nog verder uitgebreid en getest moeten worden voordat deze daadwerkelijk in gebruik genomen kan worden. Dit kan samen genomen worden met de ontwikkeling van een editor in de functionality weaver.
- Bij de impact analyse kan een keuze gemaakt worden tussen één nieuwe tool of geïntegreerd in de software fabriek. Voor Thinkwise is de laatste keuze meer geschikt, omdat er flexibel en met meerdere mensen ontwikkeld wordt. Deze personen kunnen dan onafhankelijk van elkaar de impact analyse uitvoeren.

- De impact analyse zal verder uitgebreid moeten worden. Door de gekozen opzet is dit makkelijk te realiseren. Voor het prototype is alleen gekeken naar tabellen en kolommen. Later zal ook naar taken en domeinen gekeken moeten worden. Hiervoor moeten de volgende uitbreidingen geïmplementeerd worden:
    - Versiebeheer, hiervoor moeten extra tabellen toegevoegd worden. Daarnaast moet de taak waarmee de tabellen gevuld worden aangepast worden, zodat deze op meer wijzigingen controleert en zodat de nieuwe tabellen ook gevuld worden.
    - Code extractor, deze zal uitgebreid moeten worden, zodat ook de belangrijke elementen uit de code die nodig zijn om taken en domeinen uit te breiden, toe worden gevoegd. Daarnaast moet een extra controle geschreven worden, zodat de functionaliteit niet alleen voor Microsoft SQL server geschikt is, maar ook voor DB2 code.
    - Dependency checker, de dependency checker is wel compleet en kan geheel gebruikt worden. Hiervoor is het verstandig om eerst meerdere checks toe te voegen.
- In dit onderzoek zijn tijdens het ontwikkel van een testproject nieuwe validaties en protocollen ontdekt, deze moeten toegevoegd worden.
- Zoals eerder beschreven is de debug tool een manier van valideren tussen code en model. Deze zal ontwikkeld moeten worden en kan vervolgens ingezet worden om van tevoren te valideren of de code overeen komt met het metamodel.
- Voor het testen zijn de concepten beschreven. Er moet echter nog gekozen worden welke testtechniek gebruikt wordt om daadwerkelijk te kunnen testen. Voor deze testtechnieken zijn daarna tools nodig om de code aan te sturen en te gebruiken. Er zou een onderzoek naar gedaan kunnen worden welke tools het beste bij Thinkwise passen.

13.4 Evaluatie

Het onderzoek naar protocollen heeft niet zoveel opgeleverd als verwacht was. Het was wel nuttig om dit te onderzoeken zodat nu duidelijk geworden is dat hier verder niet al te veel aandacht aan besteed hoeft te worden. De plekken waar protocollen toegepast kunnen worden, bestonden voor een deel al. Het belangrijkste dat nog niet bestond, is dat de code automatisch opgemaakt kan worden.

Door dit onderzoek naar impact analyse is een concept ontstaan wat verder uitgebreid kan worden. Doordat het makkelijk uitbreidbaar is, geeft het niet dat de tool zich alleen op tabellen en kolommen richt. Het kostte te veel tijd om een volledig afgeronde tool te implementeren. Door alleen een prototype te maken, kon ook op andere manieren van kwaliteit toevoegen worden.

Het resultaat van de validatie vind ik het meest verrassend. In eerste instantie was deze toegevoegd om het plaatje compleet te maken zodat in het onderzoek een duidelijk beeld geschetst werd van de manieren waarom kwaliteit verbeterd kan worden. Tijdens het onderzoek bleek dat impact analyse ook gebruikt kon worden voor het valideren. Zodat met minimale moeite meer resultaat uit de tools gehaald kan worden.

Het onderdeel testen is ook belangrijk voor het verbeteren van de kwaliteit, maar dit is wel altijd achteraf, als het eindproduct al gebouwd is. Er is geprobeerd om zoveel mogelijk de manieren te onderzoeken die eerder in het ontwikkelproces zitten en daarom meer vooruitstrevend zijn. Hierdoor liggen er nog veel mogelijkheden open op het gebied van testen. Er is geprobeerd om hetgeen dat wel onderzocht is zo goed mogelijk tot één geheel af te ronden met toekomstvisies hoe het geïmplementeerd zou kunnen worden. Als er meer tijd was geweest dan had ik graag hier dieper in gegaan.

Bibliografie

1. **Greenfield, Jack en Short, Keith.** Software factories: assembling applications with patterns, models, frameworks and tools. *Conference on Object Oriented Programming Systems Languages and Applications, Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* . 2003, pp. 16-27.

2. **Li, C., Li, H. en Li, M.** A Software Factory Model Based on ISO9000 and CMM for Chinese Small Organizations. *Second Asia-Pacific Conference on Quality Software (APAQS'01).* 2001, p. 0288.

3. **Cusumano, Michael A.** The Software Factory: A Historical Interpretation. *IEEE Software, vol. 06, no. 2.* 1989, pp. 23-30.

4. **Verkooijen, Peter.** Softwarefabriek vereist domeinspecifieke. *Computable.* 38, Vol. 2007.

5. **Demir, Ahmet.** Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. *Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06).* 2006, pp. 75-83.

6. **Mannaert, Herwig, Verelst, Jan en Ven, Kris.** Towards Rules and Laws for Software Factories and Evolvability: A Case-Driven Approach. *International Conference on Software Engineering Advances (ICSEA'06).* 2006, p. 35.

7. **Parigot, Didier.** Towards domain-driven development: the smartTools software factory. *Conference on Object Oriented Programming Systems Languages and Applications, Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* 2004, pp. 37-38.

8. **Sprik, Ester.** *Afstudeerscriptie UML.* Apeldoorn : Thinkwise Software, 2006.

9. Metadata; Protocol. *Wikipedia.* [Online] [Citaat van: 22 02 2008.] http://nl.wikipedia.org/.

10. **Runeson, Per en Isacsson, Peter.** Software Quality Assurance — Concepts and Misconceptions . *EUROMICRO Conference* . 2, 1998, Vol. 24.

11. **Rao, Ashok, et al.** *Total Quality Management: A Cross Functional Perspective.* sl : John Wiley & Sons, 1996.

12. Deming Prize. *The W.Edwards Deming Institute.* [Online] [Citaat van: 28 Feb 2008.] http://www.deming.org/demingprize/.

13. **Kan, S. H., Basili, V.R. en Shapiro, L.N.** Software quality: an overview from the perspective of total quality management. *IBM Systems Journal, vol.33, no 1.* 1994, pp. 4-18.

14. **Brooks, Frederick P.** No Silver Bullet Essence and Accidents of Software Engineering. *Computer.* April 1987, pp. 10-19.

15. **Saiedian, Hossein en McClanahan, Laura.** A study of two frameworks for quality software process. *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing.* February 1995.

16. **Zuser, Wolfgang, Heil, Stefan en Grechenig, Thomas.** Software quality development and assurance in RUP, MSF and XP: a comparative study. *ACM SIGSOFT Software Engineering Notes* . 30, 2005.

17. **Keleman, Mihaela L.** *Managing Quality.* sl : SAGE Publications, 2003. pp. 87-97.

18. **Stevenson, Thomas H. en Barnes, Frank C.** Fourteen years of ISO 9000: impact, criticisms, costs, and benefits. *Business Horizons.* May - June 2001, Vol. 44, 3, pp. 45-51.

19. **Jung, Ho-Won, Kim, Seung-Gweon en Chung, Chang-Shin.** Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE SOFTWARE.* September/October 2004, pp. 88-92.

20. **Jung, Ho-Won.** Validating the external quality subcharacteristics of software products according to ISO/IEC 9126. *Computer Standards & Interfaces.* September 2007, Vol. 29, 26, pp. 653-661.

21. **Herbsleb, James, et al.** Software quality and the Capability Maturity Model. *Communications of the ACM.* June 1997, Vol. 40, 6, pp. 30-40.

22. **Ward, William A.** Some Observations on Software Quality. *ACM Southeast Regional Conference - Proceedings of the 37th annual Southeast regional conference (CD-ROM).* 1999.

23. **Humphrey, Watts S.** Using A Defined and Measured Personal Software Process . *IEEE Software vol. 13 no. 3.* May 1996, pp. 77-88.

24. **Bohner, Shawn A.** Extending Software Change Impact Analysis into COTS Components. *27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02).* December 2002, p. 175.

25. **Hewitt, Jacqueline en Rilling, Juergen.** A Light-Weight Proactive Software Change Impact Analysis Using Use Case Maps. *IEEE International Workshop on Software Evolvability (Software-Evolvability'05).* September 2005, pp. 41-48.

26. **Li, Li en Offutt, A. Jefferson.** Algorithmic Analysis of the Impact of Changes to Object-Oriented Software. *12th International Conference on Software Maintenance (ICSM'96).* November 1996, p. 171.

27. **Bose, Prasanta.** Change analysis in an architectural model: a design rationale based approach. *Proceedings of the third international workshop on Software architecture.* 1998, pp. 5-8.

28. **Briand, L. C., Labiche, Y. en O'Sullivan, L.** Impact Analysis and Change Management of UML Models. *19th IEEE International Conference on Software Maintenance (ICSM'03).* September 2003, p. 256.

29. **Mao, Chengying, Zhang, Jinlong en Lu, Yansheng.** Using Dependence Matrix to Support Change Impact Analysis for CBS. *The 2007 International Conference Computational Science and its Applications (ICCSA 2007).* Augustus 2007, pp. 192-200.

30. **Xiao, Hua, Guo, Jin en Zou, Ying.** Supporting Change Impact Analysis for Service Oriented Business Applications. *International Workshop on Systems Development in SOA Environments (SDSOA'07: ICSE Workshops 2007).* Mei 2007, p. 6.

31. **Chan, M.Y. en Cheung, S.C.** Testing Database Applications with SQL Semantics. *Proceedings of 2nd International SYmposium on Cooperative Database SYstems for Advanced Appications (CODAS'99).* Maart 1999, pp. 363-374.

32. **Zhang, Jian, Xu, Chen en Cheung, S.C.** Automatic Generation of Database Instances for White-box Testing. *25th Annual International Computer Software and Applications Conference (COMPSAC'01).* 2001.

33. **Chays, David, et al.** A framework for testing database applications. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '00).* Augustus 2000, pp. 147-157.

34. **Tuya, Javier, et al.** A controlled experiment on white-box database testing. *ACM SIGSOFT Software Engineering Notes,.* Volume 33, Januari 2008, Issue 1.

35. **Suárez-Cabal, María José en Tuya, Javier.** Using an SQL coverage measurement for testing database applications. *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* . Oktober 2004.

36. **Ambler, Scott W.** Database Testing: How to Regression Test a Relational Database. *Agile Data.* [Online] [Citaat van: 31 Maart 2008.] http://www.agiledata.org.

Bijlage I. Verklarende woordenlijst

Business rules – Operaties, definities en beperkingen die voor een organisatie gelden, zodat deze haar doel kan bereiken.

Capability Maturity Model (CMM) - Model dat aangeeft hoe volwassen een organisatie is op het gebied van software ontwikkeling.

Code extractor – In dit onderzoek ontwikkelde functie om belangrijke gegevens uit de code te herkennen en op te slaan in de database

Code templates – Kleine delen van code die door de functionality weaver gebruikt worden om broncode te weven.

Componenten – Delen van een systeem die een bepaald stuk functionaliteit uitvoeren en hergebruikt kunnen worden voor andere systemen.

Control procedure – Procedure die aangeeft tijdens het weven van broncode, waar bepaalde templates moeten komen.

Database – Een elektronisch archief waarin gegevens opgeslagen kunnen worden.

Datamodel - Model waarin beschreven wordt hoe de gegevens in een informatiesysteem gestructureerd zijn.

DB2 – Een relationeel database management systeem. Dit is een systeem waarmee databases opgeslagen en gebruikt worden.

Default procedure – Een procedure die gebruikt wordt door de GUI, om tijdens het verlaten van een veld of opslaan van een record automatisch velden te vullen.

Dependency checker – In dit onderzoek ontwikkelde tool waarmee gecontroleerd kan worden waar wijzigingen weergegeven in versiebeheer invloed hebben op de reeds ontwikkelde code.

Domeinen – Hierin kan in de elektronische bouwtekening opgeslagen worden op welke manier een kolom gevuld mag worden.

Entity Relationship Diagram (ERD) – Een diagram die het datamodel op een grafische manier representeert.

ERD Modeller – Tool van Thinkwise waarmee een ERD diagram gemaakt kan worden.

Functionality Weaver – Tool van Thinkwise waarmee een eindproduct gegeneerd kan worden.

Grafische User Interface (GUI) – Applicatie om de gebruiker met de computer te laten communiceren. Deze wordt bij Thinkwise dataGUI of kortweg GUI genoemd.

GUI Renderer – Tool van Thinkwise waarmee de eigenschappen van de GUI aangepast kunnen worden.

Change Impact Analyse (CIA) – Ontwikkeld in dit onderzoek om de impact tussen versies weer te geven

ISO 9000 – Standaard die vastlegt hoe een organisatie haar kwaliteit kan waarborgen.

ISO/IEC 9126 – Standaard waarin kwaliteitsaspecten van software worden beschreven.

Kolommen – Definities van de velden in een tabel.

Layout procedure – Een procedure die gebruikt wordt door de GUI om aan te geven wanneer velden verplicht, niet

verplicht, zichtbaar, alleen lezen en regulier moeten worden.

Meta Solution Definition (MSD) – De elektronische bouwtekening die wordt opgeslagen in de database.

Microsoft SQL server – Een relationeel database management systeem, net als DB2.

Parsen – Het automatisch syntactisch analyseren van code.

Personal Software Process (PSP) – Een subset van de standaard CMM, die zich richt op de kwaliteit geleverd door individuele software ontwikkelaars.

Programmaobject – Objecten waarin business functionaliteit ondergebracht kan worden, zoals default, lay-out, taken en triggers.

Protocollen - Gedragsvoorschriften, die ervoor zorgen dat acties op een eenduidige manier uitgevoerd worden.

Record – Een enkele rij van gegevens uit een tabel. Bijvoorbeeld 1 klant of 1 dossier.

Referenties – Relatie tussen tabellen, bijvoorbeeld een contactpersoon werkt voor een bedrijf.

SQL – Structured Query Language, een taal waarin queries geschreven worden. Dit is code waarmee informatie uit de databas gehaald kan worden.

Software factory (SF) - Software waarmee vanuit een elektronische bouwtekening of model de ontwikkeling en het onderhoud van software geautomatiseerd wordt.

Stored procedure – Een functie die opgeslagen wordt in de database, meestal geschreven in SQL.

Tabellen – Object in de database waarin gegevens opgeslagen worden.

Template – zie code templatess.

Triggers – Programmacode dat bij het bewerken van gegevens nagaat of er in die bewerking aan een vastgestelde voorwaarde wordt voldaan en eventueel actie hierop onderneemt.

Validatie – Controle dat de elektronische bouwtekening voldoet aan de eisen. Bij Thinkwise uitgevoerd door de Definition validator.

Versiebeheer – Functionaliteit van Thinkwise waarin opgeslagen wordt welke wijzigingen er tussen versie plaatsgevonden hebben.

Workbench – Verzamelnaam van de tools die gebruikt kunnen worden om de elektronische bouwtekening te vullen.

Bijlage II. SQL Parsers

General SQL parser

Website: http://www.sqlparser.com/
Taal: C#, VB.NET, VC++, VB, Delphi, free pascal, Kylix
Kosten: US$ 79.95 single license, 139,95 site license, additional source code + US$300
Voordeel:

- mogelijk tabellen, kolomen en functies uit sql te halen.
- Mogelijkheid om sql-code op te maken, met behulp van instellingen
- Werkt door dll aan het project toe te voegen.
- Forum om vragen te stellen

Nadeel:

- werkt alleen als de code syntax correct is

SQLinForm

Website: http://www.sqlinform.com/
Taal: Java, dotNet
Kosten: 85 Euro
Voordeel:

- Kan ook sql aan die niet syntax correct is
- Mogelijkheid om sql-code op te maken

Nadeel:

- ontwikkelt voor opmaak, niet voor parsen (email gestuurd om te vragen of het ook kan parsen)

Zql

Website: http://www.experlog.com/gibello/zql/
Taal: Java
Kosten: geen, mogelijk source te kopen.
Voordeel:

- Uitgebreide api omschrijving
- Heeft functies om tabellen en kolommen op te halen

Nadeel:

- Niet commercieel, dus geen garanties of het goed is.
- Geen opmaakfunctionaliteit

SQL4J

Website: http://www.cs.toronto.edu/~jglu/sql4j/index.htm
Taal: Java
Kosten: geen
Voordeel:

- goede api omschrijving
- Functies om tabellen en kolommen op te halen

Nadeel:

- Vanaf de website wordt niet goed duidelijk welke queries ondersteund worden.
- Project van een professor begonnen rond 2000 (niet commercieel)
- Geen opmaak functionaliteit