MASTER THESIS "TOOLS FOR TOOLS"

Mark Jans

Scriptienummer: 577
Supervisor: Rinus Plasmeijer

Radboud Universiteit Nijmegen

MASTER THESIS "TOOLS FOR TOOLS"

Scriptienummer: 577

Student:
Mark Jans
Studentnummer s0438596

Supervisor:
Rinus Plasmeijer
Radboud Universiteit Nijmegen

Opdrachtgever:
Peter Hendriks
Info Support B.V.

Titel	Master Thesis
Project/Onderwerp	Tools for Tools
Versie	1.0
Status	Definitief
Datum	20-2-2008
Bestand	Master Thesis
Bedrijf	Info Support B.V.

Kruisboog 42, 3905 TG Veenendaal, Tel. +31 (0) 318 55 20 20 Fax +31 (0) 318 55 23 55
De Smalle Zijde 39, 3903 LM Veenendaal, Tel. +31 (0) 318 50 11 19 Fax +31 (0) 318 51 83 59
Rabobank Bergh 30.59.52.889, Postbank 47.38.593

Historie

Versie	Status	Datum	Auteur	Verandering
1.0	Concept	7-12-2007	Mark Jans	Creatie
1.0	Voorstel Definitief	12-2-2008	Mark Jans	-
1.0	Definitief	20-2-2008	Mark Jans	-

Distributie

Versie	Status	Datum	Aan
1.0	Voorstel Definitief	12-2-2008	Rinus Plasmeijer
1.0	Voorstel Definitief	12-2-2008	Peter Achten
1.0	Definitief	20-2-2008	Rinus Plasmeijer
1.0	Definitief	20-2-2008	Peter Achten
1.0	Definitief	20-2-2008	Onderwijsbureau Informatica – RU
1.0	Definitief	20-2-2008	Peter Hendriks
1.0	Definitief	20-2-2008	Marieke Keurntjes

Samenvatting

Projectspecifieke tools zijn tools die het ontwikkelproces van een bepaald software project ondersteunen, maar geen onderdeel van het project zelf zijn. Ze worden ontwikkeld door een teamlid van het project als er geen geschikte 3rd party tool beschikbaar is. Bij dit soort tools moet gedacht worden aan bijvoorbeeld code generators, data converters en tools om systeemadministratieve zaken te (semi)automatiseren.

Het ontwikkelen van projectspecifieke tools gebeurt over het algemeen zeer ad-hoc, omdat in eerste instantie verwacht wordt dat de tool maar op beperkte schaal ingezet gaat worden. Echter, later blijkt vaak dat de tool nuttig blijft tijdens de lifecycle van het project, bijvoorbeeld tijdens het testen of beheren. Omdat het aantal gebruikers groeit, er meer functionaliteit gewenst is en/of de gebruiksomgeving verandert moet de tool steeds aangepast worden. Er wordt niet goed over het ontwerp nagedacht, waardoor het doorvoeren van wijzigingen vaak lastig of zelfs helemaal niet mogelijk is.

Info Support B.V. is een IT-dienstverlener en ontwikkelt en beheert kantoorautomatiserende software projecten voor haar klanten. Hierbij streven ze ernaar om dit op een zo professioneel mogelijke manier te doen. Het competence center Java is een werkgroep binnen Info Support waar kennis en standaarden worden ontwikkeld en gedeeld voor de Java projecten.

De manier waarop projectspecifieke tools ontwikkeld worden sluit niet aan bij de professionele manier van werken. Deze thesis geeft de resultaten van een onderzoek naar het ontwikkelen van projectspecifieke tools weer, dat in opdracht van het Copmpetence Center Java van Info Support is uitgevoerd. De focus van het oplossingsgebied ligt dan ook op Java en Eclipse, de ontwikkelomgeving die binnen dit Competence center gebruikt wordt.

Door een inventarisatie onder verschillende werknemers van Info Support uit projecten voor verschillende klanten zijn een zestal knelpunten geïdentificeerd, die vaak optreden tijdens het ontwikkelen van projectspecifieke tools: “Niet geschikt voor evolutie”, “Niet geschikt voor build proces”, “Weinig tot geen documentatie”, “Te weinig overdracht aan PDC”, “Project wordt geremd door afhankelijkheid tool” en “Geen simpele richtlijnen”.

Omdat het probleem zeer algemeen is, zullen er waarschijnlijk talloze oplossingen voor de knelpunten te vinden zijn, maar door het tijdsbestek waarin het onderzoek gedaan is, is er gefocust op model driven engineering. Er is een veelzijdigheid aan projectspecifieke tools en om de knelpunten af te dekken zal hier iets generieks over gezegd moeten worden. Dit kan door middel van een abstract model. Om de ontwikkelaar te helpen, zonder allerlei richtlijnen te moeten lezen en in acht te moeten houden, kan model driven engineering ingezet worden. Dit biedt namelijk een lijdraad, waarbij een aantal stappen geautomatiseerd kunnen worden, maar toch dingen als nadenken over ontwerp afgedwongen worden.

Bij model driven engineering is een model nodig. Hiervoor hebben we naar de wat bekendere en gemakkelijk te leren diagrammen gekeken, namelijk het class diagram, data flow diagram en state machine diagram. Verder worden een aantal Eclipse plugins, wat in feite tools voor de Eclipse ontwikkelomgeving zijn, geëvalueerd. Dit zijn: “Cheat sheets”, “UML2 Tools”, “Eclipse Modeling Framework” en “Smart Developer Environment for Eclipse”. Geen van deze bestaande mogelijkheden dekt alle knelpunten af.

Uit de inventarisatie is gebleken dat de meeste tools iets met data doen, zoals bijvoorbeeld “code generators” en “data converters”. We definiëren het “tool data flow diagram”, waarmee de informatie stromen binnen een projectspecifieke tool gemodelleerd kunnen worden. Dit diagram kan ingezet worden in combinatie met model driven engineering en is gemakkelijk te leren, wat een ontwikkelaar van Info Support aangetoond heeft.

Een willekeurige projectspecifieke tool, de “stored procedure generator”, is ontwikkeld als proof of concept. Voor deze tool is in meerdere iteraties een tool data flow diagram gemodelleerd en aan de hand hiervan een implementatie ontwikkeld. Dit is een interessante case study, want deze tool is eerder ontwikkeld en toen kwamen alle knelpunten voor. We tonen aan dat het invoeren van model driven engineering in combinatie met het tool data flow diagram alle knelpunten afdekt, terwijl het ontwikkelproces er niet moeilijker van wordt.

De conclusie is dat projectspecifieke tools op een ad-hoc wijze ontwikkeld worden, omdat er gedacht wordt dat ze maar een gering aantal keren nodig zullen zijn. In de praktijk blijken ze echter veel langer nuttig, maar dan is er niet nagedacht over ontwerp en is het aanpassen ervan lastig of zelfs niet mogelijk. Door model driven engineering in combinatie met het tool data flow diagram toe te passen in het ontwikkelproces worden alle knelpunten afgedekt.

Inhoudsopgave

1 INLEIDING 8

1.1 Info Support B.V. 8
1.2 Probleemstelling 9
1.3 Structuur van de Thesis 9

2 PROJECTSPECIFIEKE TOOLS 11

2.1 Achtergronden 11
2.1.1 I-CASE environment 12
2.1.2 Software Ontwikkelstraat 12
2.1.3 Endeavour 13
2.1.4 Categorisatie op Herkomst 13
2.1.5 Tagging Tools 15
2.1.6 Categorisatie op Functie 17
2.2 Ontwikkelproces 17
2.2.1 Initiatief tot Ontwikkelen 17
2.2.2 Ontwikkelmethode 18
2.2.3 Requirements Opstellen 19
2.2.4 Implementatie 20
2.2.5 Testen 20
2.2.6 Beheren 20
2.3 Levenscyclus en Aantal Gebruikers 20
2.4 Borging van Kennis 22
2.4.1 Documentatie 22
2.4.2 Bestandslocatie 23
2.5 Eisen 23
2.6 Knelpunten 23

3 BESTAANDE OPLOSSINGEN 25

3.1 Model Driven Engineering 26
3.1.1 Class Diagram 28
3.1.2 Data Flow Diagram 29
3.1.3 State Machine Diagram 33
3.2 Eclipse Plugins 36
3.2.1 Cheat Sheets 37
3.2.2 UML2 Tools 38
3.2.3 Eclipse Modeling Framework 40
3.2.4 Smart Developer Environment for Eclipse 41
3.3 Conclusies 44

4 TOOL DATA FLOW DIAGRAM 45

4.1 Data Objecten, Processen en Data Flows 45
4.2 Decoupling 46
4.3 Executievolgorde van Processen 47
4.4 Model Driven Engineering met Tool Data Flow Diagram 48
4.4.1 Modelleren 49
4.4.2 Afbeelding op Source Code 50
4.4.3 Implementeren 50
4.4.4 Backend en Frontend 51
4.4.5 Aanpassingen in het Model 51
4.5 Voor- en Nadelen 52

5 PROOF OF CONCEPT 53
5.1 Tool Data Flow Eclipse Plugin 54
5.1.1 Grafische Editor 54
5.1.2 Code Generator 56
5.1.3 Externe Data Objecten 56
5.1.4 Interne Data Objecten 56
5.1.5 Abstracte Processen 58
5.1.6 Procesimplementaties 58
5.1.7 Applicatie 60
5.1.8 Builden en Uitvoeren 63
5.2 Stored Procedure Generator 63
5.2.1 Knelpunten 63
5.2.2 Eerste Iteratie 64
5.2.3 Tweede Iteratie 68
5.2.4 Derde Iteratie 70
5.2.5 Builden en Uitvoeren 73
5.3 Afgedekte Knelpunten 73
5.4 Ervaringen Ontwikkelaar Info Support 74
5.5 Conclusies 75
6 GERELATEERD ONDERZOEK 77
7 CONCLUSIES EN AANBEVELINGEN 79
7.1 Conclusies 79
7.2 Aanbevelingen 80
REFERENTIES 82
A VRAGENLIJST INVENTARISATIE PROJECTSPECIFIEKE TOOLS 84
B FEATURE LIST ECLIPSE PLUGIN 85
C TECHNISCH ONTWERP TOOL DATA FLOW PLUGIN 86
C.1 Grafische Editor 86
C.1.1 Graphical Modeling Framework (GMF) 86
C.1.2 Wijzigingen in Gegenereerde Code 90
C.2 Code Generator 90
C.2.1 Generator Aanroepen Vanuit Grafische Editor 91
C.3 Eclipse Plugin 93
C.3.1 Dependencies Eclipse Plugin (Compile-time) 93
C.3.2 Dependencies Tool Data Flow Project (Runtime) 93

1 Inleiding

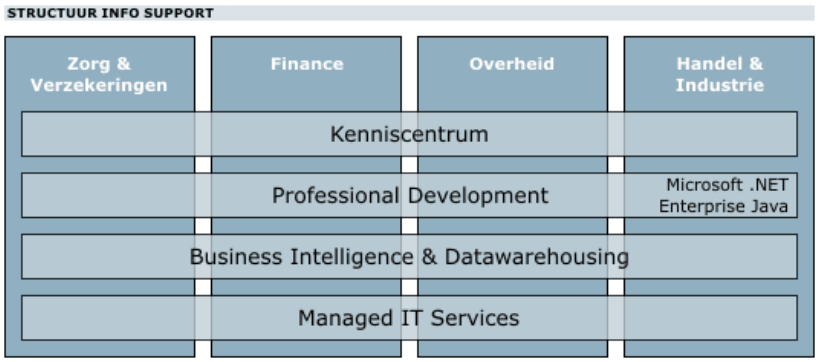
Deze master thesis presenteert mijn onderzoek op het gebied van het ontwikkelen van projectspecifieke tools. Dit onderzoek is gedaan in de vorm van een afstudeeropdracht bij Info Support B.V. Sectie 1.1 van dit hoofdstuk geeft achtergrondinformatie over Info Support, waarna in sectie 1.2 de initiële probleemstelling uitgelegd wordt. Tot slot vertelt sectie 1.3 de opbouw van de thesis.

1.1 Info Support B.V.

Info Support B.V. is een IT-dienstverlener met meer dan 250 medewerkers in Nederland. Haar dienstenpakket omvat Training, Professional Development, Business Intelligence, Life Cycle Management en Hosting.

Sinds de oprichting in 1986 wordt er continu geïnvesteerd in kennis. Deze kennis wordt gewaarborgd door middel van een eigen kenniscentrum, waar niet alleen HBO/TU geschoolde informatici verder getraind worden, maar ook de medewerkers van relaties van Info Support.

Deze kennis wordt toegepast in het ontwikkelen en onderhouden van software. Het gaat hierbij om zowel transactieverwerkende als Business Intelligence systemen, waarbij gebruik gemaakt wordt van moderne technologieën, zoals Enterprise Java en Microsoft .NET. Een eigen software ontwikkelstraat, Endeavour genaamd (zie sectie 2.1.3) staat garant voor hoge productiviteit en een beheersbaar ontwikkelproces. De afdeling Managed IT Services beheert wat er gebouwd wordt en levert verschillende hostingdiensten.



Figuur 1. Structuur Info Support

Om de dienstverlening volledig naar de wensen van de markt te kunnen inzetten werkt Info Support vanuit vier marktgerichte business units (zie ; vertikaal). Diverse competence centers (zie ; horizontaal) zorgen ervoor dat specifieke technologische kennis optimaal wordt geborgd en verder uitgewerkt.

Het onderzoek werd gedaan vanuit het competence center Professional Development, en wel de Enterprise Java groep hierbinnen, vaak ook competence center Java genoemd.

1.2 Probleemstelling

In de software-ontwikkelprojecten die door Info Support gedaan worden, bestaat regelmatig de wens om eigen aanvullende projectspecifieke tools te ontwikkelen, bijvoorbeeld voor de volgende situaties:

1. Beheren van grote hoeveelheden opgestelde (meta)gegevens die in het project gebruikt worden.
2. Inzichtelijk maken/manipuleren van run-time eigenschappen van een applicatie, bijvoorbeeld voor testdoeleinden.
3. Ondersteunen van een werkproces door geautomatiseerde validaties, -aanvullingen en -processtappen.

In de praktijk blijkt dat dergelijke tools vaak als ad-hoc oplossing worden gemaakt, met beperkte aandacht voor requirements, architectuur en onderhoudbare code. Ook vereist de ontwikkeling van dergelijke tools kennis en kunde die meestal niet direct voorhanden is. Ten slotte wordt basisfunctionaliteit vaak iedere keer opnieuw ontwikkeld, i.p.v. hergebruikt.

Het is mogelijk om eisen op te stellen waar tools aan moeten voldoen. Hierin kunnen we twee categorieën onderscheiden: (1) tools uit een software ontwikkelstraat, welke zo generiek opgezet zijn dat ze voor meerdere projecten gebruikt kunnen worden en (2) tools die voor maar één project gebruikt hoeven/kunnen te worden, de zogenaamde projectspecifieke tools.

Het onderzoek gaat over de tweede categorie. Deze tools worden op dit moment ad-hoc gemaakt, en er zijn op dit moment geen richtlijnen om ze op te zetten. Het resultaat van het onderzoek zal een aanbeveling zijn over hoe het ontwikkelen van projectspecifieke tools aangepakt kan worden.

Info Support pakt het ontwikkelproces van haar projecten professioneel aan. Hiervoor hebben ze onder ander hun eigen software ontwikkelstraat, Endeavour genaamd (zie sectie 2.1.3). De huidige manier van het ontwikkelen van projectspecifieke tools druist tegen deze professionaliteit in, waardoor het resultaat van dit onderzoek ook het ontwikkelen van de projectspecifieke tools naar een professioneler niveau kan brengen. Omdat het onderzoek vanuit het competence center Java uitgevoerd wordt, ligt de focus voor het oplossingsgebied op Java en de Eclipse [ECLIPSE] ontwikkelomgeving, die binnen dit competence center gebruikt worden.

1.3 Structuur van de Thesis

Aangezien het onderzoek over het ontwikkelen van projectspecifieke tools gaat moest er eerst duidelijk gemaakt worden wat projectspecifieke tools nu eigenlijk zijn en waar we dit in het grotere geheel kunnen plaatsen. Verder moesten de knelpunten van projectspecifieke tools geïdentificeerd worden. Een inventarisatie onder drie verschillende werknemers van Info Support met verschillende rollen in projecten bij verschillende klanten heeft hier meer inzicht in gebracht. Om de inventarisatie breed te houden is er niet gefilterd op een bepaalde techniek of gebruiksomgeving. Hoofdstuk 2 geeft de resultaten van de inventarisatie weer, met als afsluitende sectie wat de huidige knelpunten zijn bij het ontwikkelen van projectspecifieke tools.

Na uitgelegd te hebben wat projectspecifieke tools zijn en wat de huidige knelpunten ervan zijn worden in hoofdstuk 3 oplossingen besproken die gebruikt kunnen worden binnen projecten die gebaseerd zijn op Java en de Eclipse ontwikkelomgeving. Hierbij ligt de focus op model driven engineering. Er is een veelzijdigheid aan projectspecifieke tools en om de knelpunten af te dekken zal hier iets generieks over gezegd moeten worden. Dit kan door middel van een abstract

model. Om de ontwikkelaar te helpen, zonder allerlei richtlijnen te moeten lezen en in acht te moeten houden, kan model driven engineering ingezet worden. Dit biedt namelijk een lijdraad, waarbij een aantal stappen geautomatiseerd kunnen worden, maar toch dingen als nadenken over ontwerp afgedwongen worden.

De oplossingen uit hoofdstuk 3 dekken niet alle knelpunten uit hoofdstuk 2 af, waardoor er naar een uitgebreidere oplossing gezocht moest worden. Hiervoor introduceren we het "tool data flow diagram" in hoofdstuk 4. Hiermee kunnen projectspecifieke tools uit de categorieën "data converters" en "code generators" op een platform/ontwikkelomgeving onafhankelijke manier gemodelleerd worden. Behalve uitleg over wat het diagram inhoudt, welke knelpunten het afdekt en welke juist niet zal er in dit hoofdstuk een aanbeveling gedaan worden hoe projectspecifieke tools ontwikkeld kunnen worden door het tool data flow diagram te gebruiken in combinatie met model driven engineering.

Om te laten zien dat de tool uit hoofdstuk 4 de knelpunten uit hoofdstuk 2 afdekken wordt in hoofdstuk 5 een proof of concept besproken.

Tot slot wordt er in hoofdstuk 6 verwezen naar gerelateerd onderzoek en staan in hoofdstuk 7 de conclusies van het onderzoek. in dit laatste hoofdstuk zullen tevens aanbevelingen voor verder onderzoek gedaan worden.

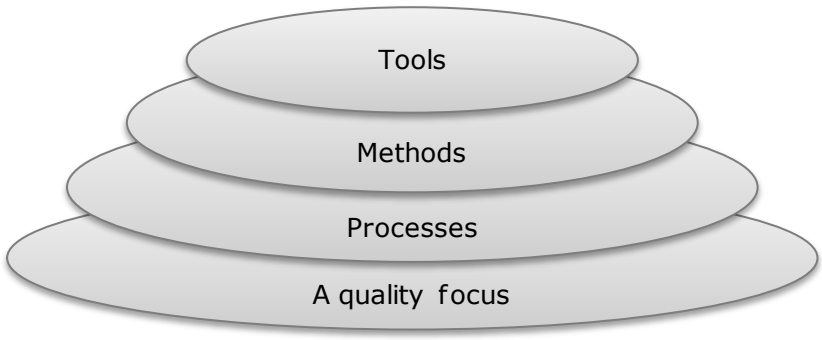
2 Projectspecifieke Tools

Om een beter inzicht in projectspecifieke tools te krijgen zijn drie werknemers van Info Support geïnterviewd aan de hand van de vragenlijst zoals te vinden is in Bijlage A. Als eerste is iemand van het Professional Development Center (PDC) van Info Support geïnterviewd. Hij werkt mee aan de Endeavour .NET ontwikkelstraat (zie sectie 2.1.3) en komt daar regelmatig met projectspecifieke tools in aanraking. De tweede geïnterviewde werknemer is ontwikkelaar in een groot .NET-project, wat ten tijde van het interview in acceptatiefase verkeert. Voor dit project zijn een aantal projectspecifieke tools ontwikkeld. Als laatste is een teamleider van een groot Java-project geïnterviewd. Hij heeft de leiding over ongeveer 20 ontwikkelaars. Het project loopt nu ongeveer twee jaar en ook in dit project zijn meerdere projectspecifieke tools ontwikkeld.

Dit hoofdstuk is een uitwerking van deze interviews. In sectie 2.1 wordt vanuit software engineering steeds dieper ingezoomd om uit te leggen wat projectspecifieke tools zijn en waar we ze in het grotere plaatje moeten plaatsen. Sectie 2.2 laat zien hoe het ontwikkelproces van projectspecifieke tools eruit ziet. Hierbij wordt er onderscheid gemaakt tussen tools die binnen een halve dag geïmplementeerd kunnen worden en tools die meer ontwikkeltijd nodig hebben en er wordt onderscheid gemaakt in het aantal (te verwachten) gebruikers. In sectie 2.3 wordt de levenscyclus van projectspecifieke tools gepresenteerd. Hierna wordt in sectie 2.4 ingegaan op de borging van kennis over de projectspecifieke tools: welke documentatie er meestal beschikbaar is en waar de (source code van de) tools opgeslagen worden, wordt hier besproken. Sectie 2.5 stelt een aantal eisen waaraan projectspecifieke tools zouden moeten voldoen. Afsluitend worden de huidige knelpunten die optreden bij het ontwikkelen van projectspecifieke tools opgesomd in sectie 2.6.

2.1 Achtergronden

"Software engineering is een gelaagde technologie en komt voort uit het doel om de kwaliteit en volwassenheid van het software ontwikkelproces continu te verhogen" [PRESS]. Een organisatie die software engineering toepast zal dan ook altijd proberen om het productieproces te verbeteren en van eerder gemaakte fouten te leren. Deze "quality focus" is de basis van software engineering (zie Figuur 2).



Figuur 2. Software Engineering Layers [PRESS]

Om de kwaliteit van het productieproces zo hoog mogelijk te houden worden er verschillende "processen" binnen het project onderscheiden. Vaak worden deze processen ook wel fasen van het project genoemd. Ook deze fasen zouden weer opgedeeld kunnen worden in subfasen, wat dan weer processen op zich zijn. Een

software engineering proces kan als volgt gedefinieerd worden: "A set of interrelated activities, which transform inputs into outputs" [IEEE-12207]. Deze inputs zijn dan bijvoorbeeld informatie, (meta)data en/of een output van een eerder proces. Voorbeelden van outputs zijn documenten, een proof of concept, een deliverable en/of het eindproduct.

De processen zeggen niets over de technische invulling en zijn wat dat betreft dus zeer generiek. Techniek ontwikkelt zich zeer snel en we willen de processen, die de basis van het project zijn, liever niet aanpassen gedurende het project. Door de processen, wat betreft techniek, generiek te houden kan dit gerealiseerd worden.

Software engineering "methoden" zijn de technische "how-to's" over hoe de processen uitgevoerd moeten worden. Een voorbeeld van een software engineering methode is Rational Unified Process [RUP] als ontwikkelmethode, maar bijvoorbeeld ook analyse- en management-, test- en maintenance-methoden vallen hieronder. Door bewezen methoden te gebruiken kan de kwaliteit van zowel het productieproces als de (eind)producten zo hoog mogelijk gehouden worden.

Software engineering "tools" (semi)automatiseren processen enerzijds en zorgen er anderzijds voor dat er volgens de gekozen methoden gewerkt wordt. We kunnen allerlei soorten tools onderscheiden, zoals in de secties 2.1.4, 2.1.5 en 2.1.6 uitgediept zal worden.

2.1.1 I-CASE environment

Om software op een professionelere manier te produceren kunnen de lagen van software engineering (gedeeltelijk) gestandaardiseerd worden. Begin jaren '90 richtten veel organisaties een I-CASE environment in. Dit is een omgeving waarin Integrated Computer Aided Software Engineering (I-CASE) tools centraal staan. Binnen de I-CASE environment wordt een repository van CASE tools bijgehouden, die het productieproces van de te bouwen software ondersteunen.

De I-CASE environment zorgt ervoor dat er binnen de organisatie meer bekend is over de gebruikte tools en deze tools beter op elkaar afgestemd zijn. Deze afstemming (vooral als er met 3rd party tools gewerkt wordt) zorgt er voor dat er een beperking op de te gebruiken methoden/technieken ontstaat.

Wanneer het productieproces sterk afhankelijk is van een bepaalde I-CASE tool kunnen deze beperkingen in een later stadium van het ontwikkel- en/of beheersproces tot problemen leiden. In dat geval zal er een keuze gemaakt moeten worden tussen een "workaround" of het aanpassen van de software, zodat de afhankelijkheid verdwijnt.

2.1.2 Software Ontwikkelstraat

In een software ontwikkelstraat wordt een omgeving gecreëerd waar de processen en de tools sterk gestandaardiseerd en gestructureerd zijn. Hierdoor wordt "hacking" voorkomen. De software ontwikkelstraat wordt ook gezien als het middel om nieuwe technologieën te introduceren [KRAN].

Organisaties houden zich vaak binnen een bepaald marktsegment bezig en hebben voorkeuren voor bepaalde methoden en technieken. Daarom kunnen sommige keuzes binnen de software engineering lagen organisatiebreed gemaakt worden. Hiervoor kan dan een ontwikkelstraat ingericht worden, waardoor de software engineering zoals eerder geschetst versmald wordt. Eventueel kunnen er meerdere ontwikkelstraten ingericht worden, waarbij voor iedere ontwikkelstraat een andere set aan methoden en technieken vastgelegd wordt. Het grote verschil tussen een I-CASE environment en een software ontwikkelstraat is dat een I-CASE

environment zich alleen op de tools-laag van software engineering richt, terwijl de software ontwikkelstraat zich op alle lagen richt.

Doordat bepaalde keuzes vast liggen kunnen er richtlijnen opgesteld worden over hoe een project opgezet, ontwikkeld en beheerd moet worden. Verder kunnen tools, die bij deze keuzes aansluiten, ter beschikking gesteld worden.

Omdat het uiteindelijke doel van software engineering het verbeteren van de kwaliteit en het verhogen van volwassenheid is, zal een ontwikkelstraat zo ingericht zijn dat fouten en ervaringen geregistreerd worden, wat ertoe zal leiden dat de richtlijnen en tools continu aangepast/verbeterd zullen worden en de kans op herhaling van eerder gemaakte fouten kleiner wordt.

2.1.3 Endeavour

Endeavour is een binnen Info Support ontwikkelde ontwikkelstraat en kent twee versies, namelijk één voor Microsoft .NET-projecten en één voor Java-projecten. Endeavour biedt richtlijnen voor het opzetten van projecten, met welke rollen er binnen het project zijn, welke processen onderscheiden worden en wie waarvoor verantwoordelijk is. Verder worden voor alle onderdelen relevante cursussen en tools aanbevolen en aangeboden. Voor alle documenten worden templates aangeboden. Deze informatie is beschikbaar via een zogenaamde "Digitale Coach", die via het intranet voor alle werknemers van Info Support te bereiken is.

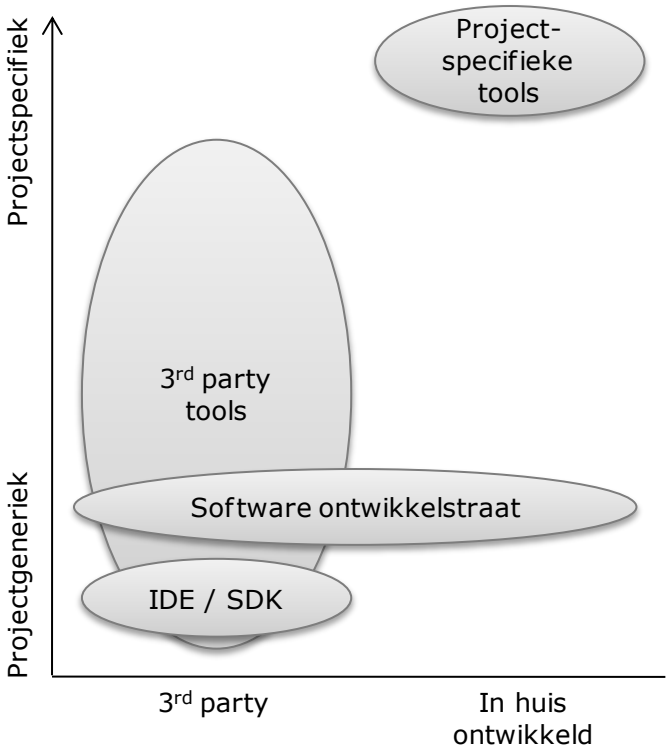
De Digitale Coach is zo opgezet dat er gemakkelijk en snel genavigeerd kan worden, waardoor de informatie op een snelle en overzichtelijke manier beschikbaar is. De informatie kan gemakkelijk aangepast worden, wat ervoor zorgt dat iedereen snel gebruik kan maken van de laatste ervaringen van collega's en de volwassenheid van het productieproces continu verhoogd wordt.

Verder biedt Endeavour een versiebeheersysteem, een geautomatiseerd build- en testproces en een zogenaamd "Project Portal", waar de statussen van de projecten en resultaten van het builden en testen weergegeven worden. Op deze manier heeft iedereen de beschikking over meest recente status en informatie over het project.

2.1.4 Categorisatie op Herkomst

In het productieproces wordt een veelzijdigheid aan tools gebruikt. Deze tools kunnen op veel manieren gecategoriseerd worden, bijvoorbeeld op leverancier (3rd party of in-house), op herkomst, op projectspecifiekheid, op functie, etc.

Tools komen voort uit een bepaald softwarepakket, kunnen als "losse" tool worden aangeschaft of zelf worden geschreven. Uit de inventarisatie is gebleken dat tools op de volgende herkomsten te categoriseren zijn.



Figuur 3. Software Engineering Tools

IDE/SDK. Een Integrated Development Environment (IDE) bestaat uit een set development tools voor een specifiek platform of architectuur. Vroeger waren dit alleen compilers en debuggers, maar tegenwoordig bieden ze een complete set tools met editors, compilers, debuggers, refactoring tools, etc. Deze tools zijn allemaal met elkaar geïntegreerd. Een Software Development Kit (SDK) bundelt development tools als libraries, compilers, debuggers en API documentatie, waarmee software voor een bepaald platform of package gebouwd kan worden.
IDE's en SDK's zijn ontwikkeld om ondersteuning te bieden bij het software productieproces, wanneer er gebruik gemaakt wordt van een specifieke techniek. Deze ondersteuning wordt op een zeer project- en organisatie-generieke manier aangeboden.

Software ontwikkelstraat. Een software ontwikkelstraat kan in-house worden ontwikkeld, maar kan ook aangekocht worden. In het eerste geval kunnen er behalve zelf ontwikkelde tools ook 3rd party tools aanbevolen en/of aangeboden en in het tweede geval kan de ontwikkelstraat aangepast worden naar de wensen van de organisatie, waardoor de grens tussen aangekocht en in-house ontwikkeld niet zo duidelijk is. In Figuur 3 wordt de ontwikkelstraat daarom zeer breed gevisualiseerd.
De ontwikkelstraat is iets projectspecifieker dan IDE's/SDK's, omdat binnen organisaties vaak dezelfde soorten projecten draaien. Dit vanwege het feit dat ze vaak vooral binnen een bepaald marktsegment opereren. Hierdoor zal de ontwikkelstraat iets projectspecifieker zijn dan IDE's/SDK's. De ontwikkelstraat biedt aanvullende tools die niet beschikbaar zijn binnen de IDE/SDK's.
Het idee van de ontwikkelstraat is een basis vormen van hoe het productieproces uitgevoerd moet worden, maar dit gebeurt zeer projectgeneriek, want alle projecten binnen de organisatie moeten van de ontwikkelstraat gebruik kunnen maken.

3rd party tools. Binnen projecten is er vaak behoefte aan aanvullende tools, die niet beschikbaar zijn in de gebruikte IDE/SDK's of ontwikkelstraat. Wanneer deze tools aangekocht worden bij een derde partij vallen ze onder deze categorie. De project-specifiekheid van 3rd party tools kunnen variëren per tool. Zo kunnen ze soms bij meerdere projecten gebruikt worden en soms is het maar voor één bepaald project handig om een bepaalde tool te gebruiken.

Projectspecifieke tools. Projectspecifieke tools zijn een aanvulling op de tools die beschikbaar zijn in de gebruikte IDE/SDK's, ontwikkelstraat en worden ontwikkeld wanneer geen geschikte 3rd party tool bestaat of beschikbaar is. Projectspecifieke tools worden door het projectteam ontwikkeld en kunnen niet zomaar bij andere projecten ingezet worden, omdat deze categorie aan tools meestal niet generiek genoeg opgezet zijn. Verder hebben andere projecten vaak geen weet van het bestaan van deze tools. Wanneer een tool projectgeneriek genoeg is, of projectgeneriek te maken is, kan de tool onderdeel van de ontwikkelstraat worden op het moment dat de tool volwassener wordt, waardoor de tool ook voor andere projecten beschikbaar wordt. In sectie 2.3 zal hier dieper op ingegaan worden.

2.1.5 Tagging Tools

Een andere categorisatie van tools is categorisatie op functie. Voor de volledigheid is er gekeken naar twee grote download sites[1], die ieder duizenden tools aanbieden, en hoe zij de tools ingedeeld hebben. Het blijkt dat zij de tools niet categoriseren, maar "taggen". Dit heeft als voordeel dat wanneer een tool zich in het grensgebied tussen twee categorieën bevindt, beide tags aan de tool toegekend kunnen worden.

Elke tool op deze download sites heeft één of meerdere tags. De lijst van tags waaruit gekozen kan worden is door de download site opgesteld. Eventueel kan er een suggestie voor een nieuwe tag worden voorgelegd, waardoor deze na goedkeuring aan de lijst toegevoegd zal worden.

Als we de lijst met tags bestuderen zien we dat er twee typen tags zijn, namelijk tags die iets zeggen over de functie van de tool en tags die iets zeggen over de omgeving waarin de tool gebruikt kan worden. Deze tags worden hieronder opgesomd.

2.1.5.1 Functie Tags

Assemblers/disassemblers. Omzetters van "assembly" source code naar machine code of omgekeerd.

Build tools. Het build proces van een project is vaak omvangrijk en heeft een hiërarchische structuur. Build tools automatiseren alle stappen van het build proces.

Code generators. Dit zijn tools die, meestal na input van (meta)data, code genereren. Zo kan er bijvoorbeeld programmacode gegenereerd vanuit een bepaald model. Het woord "code" moet vrij breed opgevat worden. Zo kan er ook code voor een Domain Specific Language (DSL) gegenereerd worden.

Compilers/Cross compilers/Interpreters/Pre-processors. Dit zijn tools die source code omzetten naar machinetaal. Dit kan stap-voor-stap door een interpreter of in zijn geheel door een compiler.

Data converters. Data converters converteren het ene datatype in het andere. In principe kunnen code generators gezien worden als speciale converters die bepaalde informatie omzetten naar een vorm van code.

Debuggers. Dit zijn tools om bugs op te sporen door het programma stap-voor-stap uit te voeren.

[1] SourceForge.net [SF] en freshmeat.net [FM]

Design/modeling. Tools die helpen bij het ontwerpen en modelleren. Een voorbeeld is een tool waarmee UML diagrammen gemaakt kunnen worden [UML].

Frameworks. Een framework biedt een herbruikbaar gedeelte van een applicatie, waardoor alleen details nog ingevuld hoeven te worden.

Libraries/algorithms. Een library of algoritme (meestal verpakt in een library) dat ingezet kan worden, zodat de functionaliteit ervan niet zelf geschreven hoeft te worden.

Profiling. Profilers analyseren de uitvoer van een applicatie, zodat achterhaald kan worden waar de meest uitgevoerde code zich bevind, zodat wanneer de code geoptimaliseerd moet worden gekeken kan worden waar het eerst begonnen moet worden.

Refactoring. Refactoring tools wijzigen kleine dingen in een set source code. Bijvoorbeeld het aanpassen van naamgeving van variabelen, of code beautifiers.

System administration. Tools om het ontwikkel-, acceptatie- en/of productiesysteem mee te beheren.

Testing/analyse. Allerhande testtools. Ook metrieken vallen onder deze categorie.

Version control. Tools om versies bij te houden. Dit kunnen bijvoorbeeld versiebeheersystemen zijn zoals CVS, SubVersion of Microsoft Visual SourceSafe, maar ook diff- en merge-tools.

Widget sets. Een speciaal type library, wat ingezet kan worden bij het implementeren van een grafische user interface. Voorbeelden zijn wxWindows en GTK.

2.1.5.2 Gebruiksomgeving Tags

CASE. Tools die onderdeel zijn van een I-CASE environment.

Documentation. Tools die iets met documentatie doen. Bijvoorbeeld documentatie uit (geannoteerde) source code genereren of controleren dat alle source code van commentaar voorzien is.

Embedded systems. Allerhande tools die gebruikt kunnen worden om embedded systems te ontwikkelen.

I18N/L10N. Allerhande tools die ondersteunen bij het ontwikkelen van meertalige applicaties.

Object Oriented. Tools die in een object georiënteerde omgeving werken. Een voorbeeld is een modelleerapplicatie om object georiënteerd systeem mee te modelleren.

Quality assurance. Tools die helpen om de kwaliteit te verbeteren. Bijvoorbeeld een test of refactoring tool.

Usability. Tools die helpen om een product gebruiksvriendelijker te maken. Bijvoorbeeld een modelleerapplicatie.

User Interface. Tools die iets met de user interface doen. Bijvoorbeeld een modelleerapplicatie of een code generator, die van een model, wat de user interface van de applicatie beschrijft, programma code genereert.

2.1.6 Categorisatie op Functie

De vier categorieën software engineering tools (sectie 2.1.4) bieden ieder haar eigen set aan functionaliteiten. Daarom kunnen we de functie tags als hierboven geschetst op de volgende manier verdelen onder de categorieën:

IDE/SDK

- Assemblers/Disassemblers
- Build tools
- Compilers/Cross compilers/Interpreters/ Pre-processors
- Debuggers
- Libraries/algorithms
- Profiling
- Refactoring

Software ontwikkelstraat

- Build tools
- Code generators
- Data converters
- Design/modeling
- Frameworks
- Libraries/algorithms
- Profiling
- Refactoring
- System administration
- Testing/Analysis
- Version control
- Widget sets

3rd party tools

- Build tools
- Code generators
- Converters
- Design/modeling
- Libraries/algorithms
- Profiling
- Refactoring
- System administration
- Testing/Analysis
- Version control
- Widget sets

Projectspecifieke tools

- Build tools
- Code generators
- Converters
- Design/modeling
- Refactoring
- System administration
- Testing/Analysis

2.2 Ontwikkelproces

Deze sectie beschrijft het ontwikkelproces van projectspecifieke tools en wie er bij de verschillende stappen betrokkenen zijn.

2.2.1 Initiatief tot Ontwikkelen

Op een gegeven moment ontstaat er binnen een project behoefte aan een projectspecifieke tool. Uit de inventarisatie onder werknemers van Info Support is gebleken dat deze behoefte over het algemeen komt vanuit een ontwikkelaar, die een bepaalde actie geautomatiseerd zou willen zien. Hierdoor worden routinewerkzaamheden gereduceerd, tijd worden bespaard en kwaliteit worden verbeterd. Ook binnen het management kan er behoefte zijn aan een tool,

bijvoorbeeld om statistieken over het project op een geautomatiseerde manier te verzamelen.

We zien vooral dat de toekomstige gebruiker van de tool het initiatief neemt om de tool te (laten) ontwikkelen. Deze gebruiker kan dan ook het beste overzien of het nuttig is om de tool te ontwikkelen, want het ontwikkelen en gebruiken van de tool moet uiteindelijk niet meer tijd kosten dan wanneer het werk handmatig gedaan zou worden, behalve wanneer de tool voor een significante performance- of kwaliteitsverbetering van het product zou kunnen zorgen, terwijl dit niet het geval is als het werk handmatig gedaan zou worden.

Binnen Info Support geldt "Use before Buy before Make" als richtlijn, wat inhoudt dat voor elke benodigde tool vooraf moet worden gekeken of deze niet al via een prefered supplier, open source project beschikbaar is of wordt geleverd door een third party. Tevens moet worden nagegaan of er niet al een alternatief van Info Support is. Als zelf- en of bijbouw noodzakelijk is dan moet dit worden beargumenteerd [KWALOS].

Wanneer het naar schatting minder dan een halve dag inneemt om een projectspecifieke tool te ontwikkelen zal een ontwikkelaar dat vaak doen zonder aan de projectleider te vragen of de ontwikkelaar daar toestemming voor krijgt. Als het langer duurt om een tool te ontwikkelen zal de projectleider daar altijd eerst toestemming voor moeten geven.

2.2.2 Ontwikkelmethode

Een tool wordt meestal door één persoon ontwikkeld, namelijk de initiatiefnemer of een ontwikkelaar binnen het projectteam in het geval dat iemand van het management behoefte heeft aan een bepaalde tool.

Het ontwikkelen van tools gebeurt op een incrementele manier. Dit betekent dat de tool origineel voor een bepaalde functionaliteit geschreven wordt en later eventueel aangepast wordt met nieuwe functionaliteit. We kunnen zeggen dat ze een evolutionair karakter hebben.

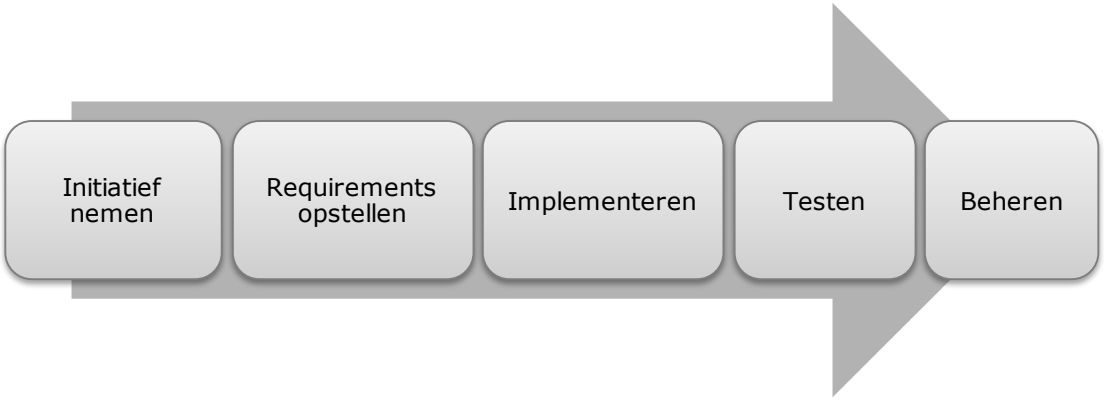
De gebruikte ontwikkelmethode binnen de incrementen is sterk afhankelijk van het aantal gebruikers dat de tool uiteindelijk zullen gaan gebruiken. Wanneer de tool op een later stadium in de levens-cyclus door meerdere mensen gebruikt gaat worden zien we dan ook dat de aanpak van ontwikkelen professioneler wordt. Meer over deze verandering is te vinden in sectie 2.3.

	Eén gebruiker (ontwikkelaar)	Alleen gebruikers binnen projectteam	Gebruiker(s) buiten projectteam
Implementatie kan in enkele uren	Ad-hoc	Ad-hoc	Ad-hoc
Implementatie neemt meer dan een halve dag in	Ad-hoc	Ad-hoc	Gefaseerd

Tabel 1. Ontwikkelmethode

In het geval dat de toekomstige gebruikers zich allemaal in het projectteam bevinden zal de ontwikkelmethode ad-hoc zijn. Er worden geen fasen onderscheiden en het ontwikkelen gebeurt vooral door middel van "trial and error".

Wanneer de tool ook door gebruikers buiten het projectteam gebruikt zal gaan worden of wanneer de werkplek van een aantal gebruikers niet dichtbij de werkplek van de ontwikkelaar is, wordt het ontwikkelen professioneler aangepakt. Het ontwikkelproces zal in verschillende fasen opgedeeld worden, er zullen duidelijke requirements opgesteld worden en er zal gebruikersdocumentatie geschreven worden.



Figuur 4. Gefaseerde ontwikkelmethode

Ten opzichte van een softwareproject is het ontwikkelen van een tool zeer kleinschalig. Daarom zullen de fasen van ontwikkeling nooit erg strak zijn. Ze moeten daarom ook meer als stappen van het ontwikkelproces gezien worden dan als echte fasen die ieder iets op moeten leveren.

Behalve het aantal gebruikers is de ontwikkelmethode ook afhankelijk van de complexiteit van de tool. Wanneer de tool binnen enkele uren geïmplementeerd kan worden is het niet nuttig om een gefaseerde ontwikkelmethode te gebruiken.
Het is belangrijk dat er een balans wordt gevonden tussen hoeveel tijd de implementatie kost en hoeveel tijd er geïnvesteerd wordt in het opstellen van requirements en het testen van de tool.

2.2.3 Requirements Opstellen

De omgang met requirements is sterk afhankelijk van het aantal gebruikers. Wanneer de ontwikkelaar de enige gebruiker is, zal hij/zij de requirements duidelijk in zijn/haar hoofd hebben en is het niet nodig om deze formeel op te schrijven. Eventueel zullen er collega's geraadpleegd worden of de tool aanvullende functionaliteit moet kunnen bieden of later gemakkelijk uit te breiden moet zijn met een bepaalde functionaliteit. In dat laatste geval zal de tool dus op één of andere manier generieker opgezet moeten worden.

Als de tool ontwikkeld wordt met behulp van een gefaseerde ontwikkelmethode zal er netjes een functioneel en technisch ontwerp opgesteld worden. Het is dan duidelijk dat de tool vaker dan één keer gebruikt gaat worden en de kans groot is dat deze later aangepast moet worden. Het maken van documentatie is in dat geval noodzakelijk.

Tools die geadopteerd worden in de Endeavour software ontwikkelstraat moeten aan een aantal eisen voldoen, bijvoorbeeld dat een functioneel en technisch ontwerp beschikbaar is. Meer over de eisen van projectspecifieke tools is te vinden in sectie 2.5.

2.2.4 Implementatie

Projectspecifieke tools worden meestal door één ontwikkelaar geïmplementeerd. Dit is vooral omdat de omvang van tools niet zo groot is dat het werk verdeeld kan worden en/of dat het niet wenselijk is om een verdeling te maken. Het feit dat de requirements lang niet altijd formeel opgeschreven worden helpt mee dat alleen de ontwikkelaar, die precies weet wat er moet gebeuren, de tool zal implementeren.

Het implementeren van de tool gebeurt over het algemeen door de initiatiefnemer, aangezien hij/zij juist de behoefte aan de tool heeft, een gebruiker zal zijn en de requirements heeft opgesteld en/of in zijn hoofd heeft. Wanneer het initiatief vanuit het management komt zien we dat de projectleider meestal een ontwikkelaar aanwijst, die de tool zal implementeren.

2.2.5 Testen

Ook bij het testen is het zeer afhankelijk van hoe breed de tool ingezet zal worden of er veel of weinig getest wordt. Wanneer de ontwikkelaar de enige gebruiker is zal deze de tool zelf testen totdat hij/zij er tevreden over is. Wanneer er bij het gebruik een bug opduikt kan hij/zij het zelf oplossen. Eventueel wordt er gebruik gemaakt van unit-testen, wanneer de tool wat complexer is.

Wanneer er meerdere gebruikers zijn, maar alle gebruikers zich binnen het projectteam bevinden zal in de meeste gevallen alleen de ontwikkelaar de tool testen. Eventueel test een mede-projectlid de tool voordat de rest het gaat gebruiken. Dit testen is niet al te intensief, aangezien de ontwikkelaar altijd dicht in de buurt is, en deze de tool snel aan kan passen bij problemen.

We zien dat het testproces veel intensiever is wanneer ook mensen buiten het projectteam de tool gebruiken. Dit omdat de ontwikkelaar dan over het algemeen minder gemakkelijk te benaderen is en fouten dus minder adequaat opgelost kunnen worden.

Over het algemeen wordt er, als er getest wordt, alleen maar op functioneel niveau getest, om het aantal bugs zoveel mogelijk te reduceren. Het testen van performance wordt niet vaak gedaan, omdat het niet nuttig lijkt, aangezien de tools geen onderdeel van het uiteindelijke product zijn en er van tevoren gedacht wordt dat ze niet frequent uitgevoerd zullen gaan worden. Het zelfde geldt voor stress- en regressietests.

2.2.6 Beheren

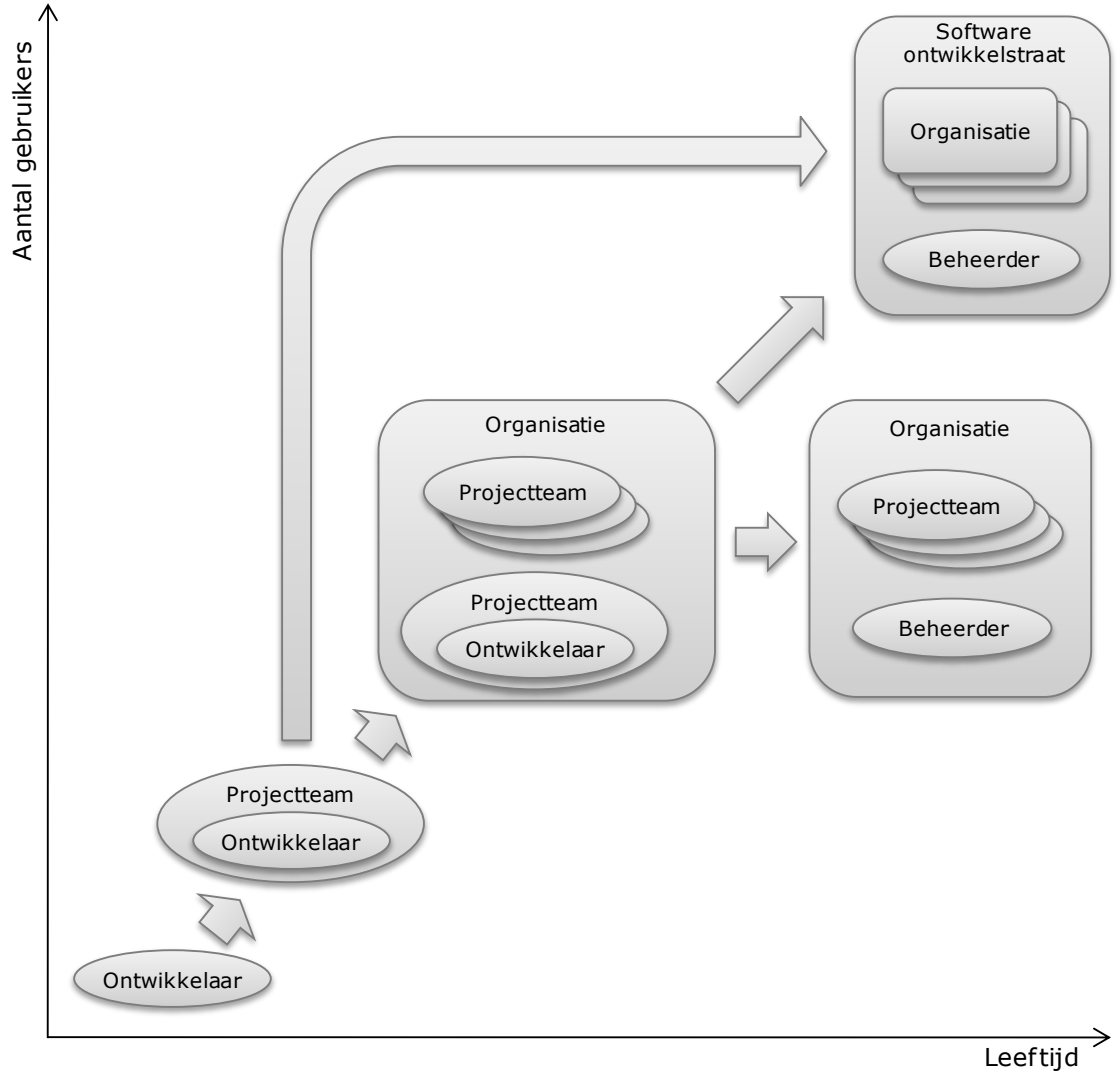
We zien dat de ontwikkelaar die de tool implementeert over het algemeen ook het beheer op zich neemt. Hiervoor is geen overdracht van kennis nodig. Als er wel een overdracht noodzakelijk is, wordt deze vaak bemoeilijkt doordat er geen of slechte documentatie is.

Wanneer het aantal gebruikers van de tool toeneemt wordt het beheer vaak een te grote last voor de ontwikkelaar, waardoor er alsnog een overdracht plaats zal vinden. Meer hierover is te lezen in sectie 2.3.

2.3 Levenscyclus en Aantal Gebruikers

Projectspecifieke tools starten meestal als een "gehackte" tool. Er is snel behoefte aan een bepaalde tool, die nog niet beschikbaar is en waarvoor besloten is om deze in-house te gaan ontwikkelen. Omdat de tool geen onderdeel van het op te leveren product is wil men hier zo min mogelijk tijd en geld in steken.

In eerste instantie is de initiatiefnemer ook de ontwikkelaar en enige gebruiker. Naar mate een tool langer bestaat zien we dat het aantal gebruikers toeneemt. Figuur 5 visualiseert dit. Een voetnoot hierbij is echter dat maar een beperkt aantal projectspecifieke tools de volledige cyclus zal doorlopen.



Figuur 5. Aantal gebruikers naarmate tool ouder wordt

Als de tool een tijdje in gebruik is, de ontwikkelaar/gebruiker er enthousiast over is en de tool niet al te gebruiksonvriendelijk is, kan er een behoefte ontstaan om de tool breder in te zetten. Mede-projectleden zullen de tool dan gaan gebruiken. De ontwikkelaar blijft de tool beheren en naar behoefte uit blijven breiden met nieuwe functionaliteit.

Vaak zien we dat de tool later in het ontwikkelproces van het project op een andere manier ingezet kan/gaat worden. De tool wordt bijvoorbeeld gebuikt als ondersteuning van het implementatieproces van het product, maar blijkt ook handig te zijn om in te zetten bij het beheren van het product. In dat geval wordt de tool niet alleen in het projectteam van de ontwikkelaar ingezet, maar zal het ook ingezet worden door andere afdelingen binnen de organisatie.

Wanneer de functionaliteit van de tool uitgekristalliseerd is en blijkt dat de ontwikkelaar teveel tijd kwijt is met het beheer ervan of het niet meer wenselijk is dat

de ontwikkelaar de tool blijft beheren kan het beheer overgedragen worden aan een centrale beheerafdeling of aan systeembeheer. In deze twee gevallen blijft de tool alleen in de organisatie inzetbaar.

Sommige tools zouden veel breder ingezet kunnen worden dan alleen in het project waarvoor het ontwikkeld is. In dat geval kan er een overdracht plaatsvinden naar de software ontwikkelstraat. Het beheer wordt hiermee overgedragen en andere projecten die op de ontwikkelstraat gebaseerd zijn kunnen ook profiteren van de tool. De tool zal hiervoor vaak generieker gemaakt moeten worden.

In het geval van Endeavour als software ontwikkelstraat kan een "request for adoption" ingediend worden, waarna de ontwikkelaars van Endeavour zullen beslissen of het mogelijk en nuttig is om de tool op te nemen in de ontwikkelstraat. Voor tools die onderdeel van Endeavour zijn gelden bepaalde eisen. De ontwikkelaars van Endeavour die het beheer overnemen zullen de tool en de documentatie ervan aan moeten passen om ervoor te zorgen dat aan deze eisen voldaan wordt. Meer over deze eisen is te vinden in sectie 2.5.

Niet alle projectspecifieke tools zullen het gehele traject doormaken. Echter, de meeste projectspecifieke tools zullen niet door meer gebruikers dan het projectteam gebruikt worden. Dit komt onder andere omdat te weinig mensen betrokken zijn bij de ontwikkelaars van de software ontwikkelstraat en er dus niet bij nadenken dat de tool ook handig zou kunnen zijn voor andere projecten.

Een mogelijkheid waarom een projectspecifieke tool niet meer gebruikt wordt is bijvoorbeeld dat een nieuwe versie van een 3rd party tool voldoende functionaliteit biedt. Een andere mogelijkheid is bijvoorbeeld een data conversie tool die tijdelijk ingezet wordt om data van het oude systeem naar een gedeelte van een nieuw systeem te converteren. Wanneer het gehele nieuwe systeem af is, is deze conversie niet meer nodig, omdat alle data zich dan in het nieuwe systeem bevindt.

2.4 Borging van Kennis

In deze sectie wordt besproken hoe de kennis over projectspecifieke tools geborgd wordt. Eerst wordt er aandacht besteed aan welke documentatie over het algemeen beschikbaar is. Verder zal er besproken worden waar de (source code van de) tools fysiek opgeslagen worden.

2.4.1 Documentatie

Een ontwikkelaar heeft een bepaalde manier van werken, waarbij de hoeveelheid commentaar die hij in de source code schrijft redelijk constant is. Als het om een tool gaat die op een ad-hoc manier ontwikkeld wordt, zal deze hoeveelheid commentaar tot een minimum beperkt worden, maar niet geheel verdwijnen.

Behalve de source code van commentaar voorzien is het sterk afhankelijk hoeveel documentatie er beschikbaar is. Wanneer er veel of veel wisselende gebruikers zijn, of wanneer gebruikers zich fysiek niet in de buurt van de ontwikkelaar bevinden zijn er over het algemeen altijd installatie- en gebruikershandleidingen beschikbaar.

Zoals eerder in sectie 2.2.3 werd verteld worden lang niet altijd requirements opgesteld. Daarom is er maar zelden een functioneel- en/of technisch ontwerp te vinden. Deze worden alleen gemaakt als vanaf het begin af aan duidelijk is dat de tool breed ingezet zal gaan worden. Eventueel wordt er later wel een functioneel- en/of technisch ontwerp gemaakt. Dit bijvoorbeeld wanneer de tool opgenomen gaat worden

in de software ontwikkelstraat en dan ineens aan bepaalde eisen moet gaan voldoen. De ontwerpen worden dan meestal gemaakt door middel van reverse engineering.

2.4.2 Bestandslocatie

Meestal wordt de source code van de tools opgeslagen in het versiebeheersysteem van het project waarvoor de projectspecifieke tool geschreven wordt. Dit kan dan bijvoorbeeld in een bepaalde submap zijn. Hierdoor loopt de tool mee in de life-cycle van het op te leveren product.

Hele kleine tools, zoals bijvoorbeeld batch-scripts, waar niet echt onderscheid gemaakt kan worden tussen source code en de tool zelf, zien we vaak opgeslagen worden op een netwerkschijf. Alle projectleden kunnen bij deze netwerkschijf en er wordt regelmatig een backup van de netwerkschijf gemaakt, waardoor de ontwikkelde tools niet verloren zullen gaan bij een systeem/harde schijf-crash.

Binnen een project leven altijd een aantal tools die een ontwikkelaar ooit heeft gemaakt en alleen zelf gebruikt. Het ontwikkelen heeft dan zo weinig tijd gekost dat de tool geheel op eigen initiatief ontwikkeld is en behalve de ontwikkelaar niemand van de tool af weet. In dat geval zullen dergelijke tools op de harde schijf van de ontwikkelaar staan en waarschijnlijk verloren gaan bij een systeem/harde schijf-crash. Dergelijke tools lossen over het algemeen maar één heel klein, heel specifiek probleem op en kunnen maar eenmalig ingezet worden, waardoor het niet erg van belang is of het bewaren van de tool veilig gesteld is.

2.5 Eisen

In principe gelden er geen eisen of richtlijnen voor het ontwikkelen van projectspecifieke tools. Echter, als een projectspecifieke tool op een gegeven moment overgedragen gaat worden naar het Professional Developer Center (PDC) van Info Support, zal de projectspecifieke tool aangepast moeten worden, zodat deze aan een aantal eisen voldoet. Het PDC beheert de tools binnen de Endeavour ontwikkelstraat. De eisen die dan aan tools gesteld worden staan vast in een document [KWALOS].

2.6 Knelpunten

Door middel van de inventarisatie zijn enkele knelpunten die tijdens het ontwikkelen van projectspecifieke tools optreden naar voren gekomen. Deze sectie is een opsomming van deze knelpunten.

1. **Niet geschikt voor evolutie.** Projectspecifieke tools worden over het algemeen op een ad-hoc manier ontwikkeld, waardoor er geen duidelijk ontwerp is. Hier staat recht tegenover dat deze tools een evolutionair karakter hebben, wat wil zeggen dat er regelmatig iets bij gemaakt wordt. Omdat er geen duidelijk ontwerp is en het voor de ontwikkelaar alweer langer geleden is dat hij/zij er aan gewerkt heeft moet iedere keer opnieuw uitgezocht worden waar de aanpassing plaats moet vinden. Verder wordt er initieel geen rekening gehouden met het feit dat er later features toegevoegd moeten kunnen worden.
2. **Niet geschikt voor build proces.** Het komt regelmatig voor dat het wenselijk zou zijn om een tool die regelmatig handmatig uitgevoerd wordt als stap in het build proces op te nemen. Een voorbeeld is een code generator, die aan de hand van bepaalde (meta)data programmacode genereert. Elke keer dat de (meta)data verandert moet de tool opnieuw handmatig uitgevoerd worden. Het zou in dit geval handiger zijn als de tool elke keer dat het product gebuild wordt, net voor de compile-fase uitgevoerd zou worden en dus een stap in het build proces zou zijn. Echter is de source code van de user interface vaak door de source code

van de business logica verweven, waardoor het erg lastig is om de user interface van de tool te verwijderen.

3. **Weinig tot geen documentatie.** Voor de meeste projectspecifieke tools is weinig tot geen documentatie aanwezig. Over het algemeen houdt de documentatie op bij wat commentaar in de source code. Wanneer de tool later aangepast moet worden of moet worden overgedragen aan een andere ontwikkelaar of beheerder moet iedere keer opnieuw uitgezocht worden wat de structuur van de tool is.
4. **Te weinig overdracht aan PDC.** Projectspecifieke tools worden te weinig overgedragen aan het PDC. Dit komt vooral omdat de ontwikkelaars, die vooral bij klanten gedetacheerd zijn, niet zoveel met het PDC te maken hebben en er daardoor niet zo altijd aan denken dat een bepaalde tool met wat aanpassingen generieker gemaakt zou kunnen worden en daardoor veel breder ingezet zou kunnen worden.
5. **Project wordt geremd door afhankelijkheid tool.** Een tool die in eerste instantie de wat minder leuke klusjes automatiseren helpen de ontwikkelaar bij het ontwikkelen van het product en het te ontwikkelen product is hier totaal niet afhankelijk van. Echter blijkt vaak dat het ontwikkelproces steeds afhankelijker wordt van de tool, naarmate de tool vaker gebruikt wordt. Omdat de omgeving continu verandert moeten de tools steeds aangepast worden om goed mee te kunnen draaien. Doordat er weinig tot geen documentatie is en de tools op een ad-hoc manier ontwikkeld zijn is het lastig om de tool later aan te passen.
6. **Geen simpele richtlijnen.** Er zijn geen richtlijnen voor het ontwikkelen van projectspecifieke tools. Door op eerdere momenten rekening te houden met de toekomst kan er veel tijd bespaard worden, terwijl het rekening houden weinig extra tijd kost. Bij zulke richtlijnen zou bijvoorbeeld horen dat er bij elke wijziging nagedacht moet worden of de tool later eventueel onderdeel zou kunnen worden van een software ontwikkelstraat. Hierdoor kan de "specifiekheid" continu ingeschat worden en kunnen bepaalde ontwerpbeslissingen gemaakt worden, die later veel tijd kunnen besparen. Als er richlijnen opgesteld worden moeten deze ook weer niet te uitgebreid zijn, dat een ontwikkelaar ervan "afschrikt" en de projectspecifieke tool alsnog op de huidige ad-hoc manier ontwikkelt.

3 Bestaande Oplossingen

In hoofdstuk 2 zijn een aantal knelpunten geïdentificeerd die regelmatig optreden bij het ontwikkelen van projectspecifieke tools. Bepaalde diagrammen en tools kunnen een oplossing voor deze knelpunten bieden of in ieder geval te beperken. In dit hoofdstuk worden bestaande hulpmiddelen besproken waarbij telkens aangegeven wordt welke knelpunten ermee afgedekt worden en wat de voor- en nadelen van deze oplossingen zijn.

Het verbeteren van het ontwikkelproces van projectspecifieke tools is eigenlijk het verbeteren van het software engineering proces, waarbij we alleen naar dit proces voor projectspecifieke tools kijken. Het software engineering proces kunnen we in de verschillende lagen, zoals in sectie 2.1 besproken, verbeteren. Bij dit verbeteren moet er wel rekening mee gehouden worden dat de verbetering in verhouding staat tot het op te lossen probleem. Het heeft immers geen zin om een zeer uitgebreide gefaseerde aanpak ("methods"-laag) te gebruiken voor het ontwikkelen van een bepaalde tool, terwijl de te verwachte ontwikkeltijd maar een aantal uur is. Vaak blijkt achteraf dat er te lichtzinnig is gedacht over deze ontwikkeltijd, maar een ontwikkelaar die denkt dat hij een tool in een paar uur ontwikkelt zal niet snel geen uitgebreide aanpak gebruiken. Daarom zullen we de oplossingen voor de eerdergenoemde knelpunten vooral in de lagen "processes" en "tools" moeten zoeken.

De geïdentificeerde knelpunten zijn problemen van een algemene aard, waardoor er talloze bestaande oplossingen te vinden zijn. Hierdoor zullen we ons moeten focussen op bepaalde methoden/technieken. Om de "processes"-laag te verbeteren wordt er gefocust op model driven engineering. Er is een veelzijdigheid aan projectspecifieke tools en om de knelpunten af te dekken zal hier iets generieks over gezegd moeten worden. Dit kan door middel van een abstract model. Om de ontwikkelaar te helpen, zonder allerlei richtlijnen te moeten lezen en in acht te moeten houden, kan model driven engineering ingezet worden. Dit biedt namelijk een lijdraad, waarbij een aantal stappen geautomatiseerd kunnen worden, maar toch dingen als nadenken over ontwerp afgedwongen worden.

Andere, niet-model driven engineering, oplossingen zouden gezocht kunnen worden in bijvoorbeeld frameworks, die voorzien in bepaalde basisfunctionaliteit van een tool en waarbij dingen als directorystructuren vast staan.

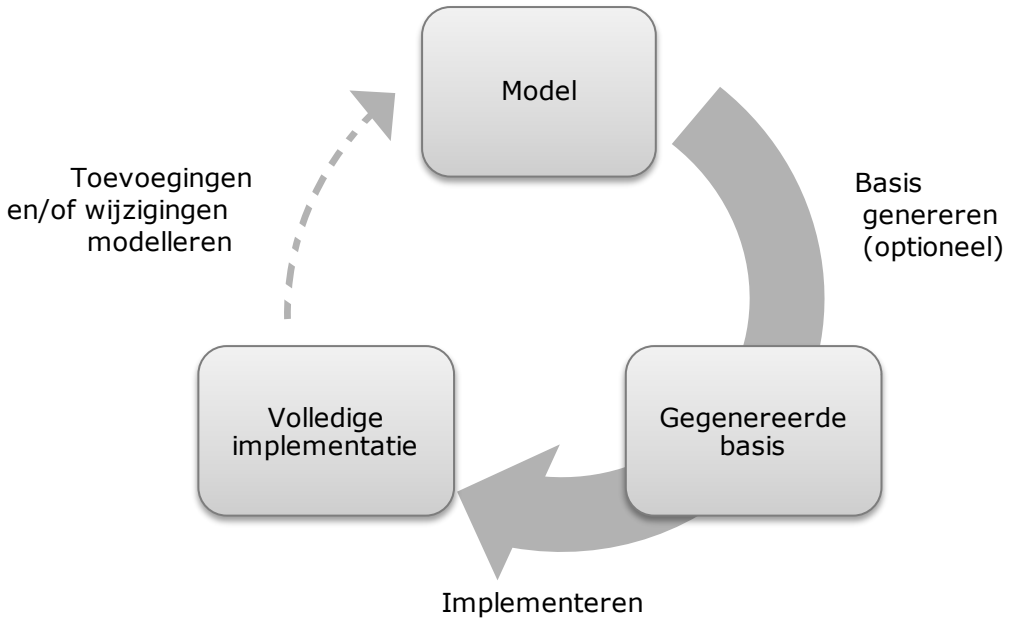
Ontwikkelomgevingen worden steeds uitgebreider, waardoor er tegenwoordig allerhande mogelijkheden gegeven worden om model driven engineering eenvoudig toe te passen. Hierdoor is het invoeren van dit principe in het ontwikkelproces van projectspecifieke tools gemakkelijk te realiseren. Sectie 3.1 bespreekt oplossingen met betrekking tot model driven engineering. Verder worden een aantal diagrammen besproken die als model gebruikt kunnen worden.

Ook wanneer er naar oplossingen in de "tools"-laag van software engineering gezocht gaat worden kan dit op talloze manieren. Het onderzoek had van het begin af aan een focus op Java/Eclipse wanneer er naar oplossingen gezocht moest worden. Door middel van plugins kan de Eclipse ontwikkelomgeving uitgebreid worden met allerlei tools. Voor oplossingen in de "tools"-laag is er gefocust op Eclipse plugins die betrekking hebben op model driven engineering. Hierdoor wordt aangesloten bij zowel de focus van het onderzoek als de oplossingen voor de "processes"-laag. Sectie 3.2 bespreekt deze plugins.

3.1 Model Driven Engineering

Bij model driven engineering [MDE] worden modellen op een systematische manier ingezet als primaire stap van het software engineering proces. Hierbij wordt het systeem eerst heel abstract gemodelleerd en hierna worden nieuwe, minder abstracte modellen gemodelleerd, die steeds meer detail bevatten, totdat ze gedetailleerd genoeg zijn dat ze in source code geïmplementeerd kunnen worden [RTSE].

Voor projectspecifieke tools is het niet van belang dat deze door middel van steeds minder abstractere diagrammen gemodelleerd worden. We zouden graag één diagram willen, dat abstract genoeg is om de tool mee te modelleren, zonder dat dit alleen een visualisatie van de code is, maar niet te abstract is, zodat het wel mogelijk is om source code vanuit het diagram te genereren. We definiëren in Figuur 6 hoe de roundtrip van model driven engineering eruit zal zien wanneer we één model op één abstractieniveau maken en deze implementeren. Eventueel kan een gedeelte van de source code gegenereerd worden. Deze stap hebben we zelf toegevoegd om beter aan te geven hoe het implementeren in zijn werk gaat.



Figuur 6. Roundtrip bij gebruik van model driven engineering

Het ontwikkelen van software start bij een model. Hiervoor kunnen allerhande diagrammen of Domain Specific Languages (DSL) gebruikt worden. Diagrammen zijn grafische modellen zoals bijvoorbeeld een class diagram. Een DSL wordt ontworpen om een specifiek domein door middel van tekst te modelleren. Dit is in feite hetzelfde principe als bij bijvoorbeeld een class diagram, die alleen classes met bijbehorende attributen, operaties en relaties kan modelleren, maar bijvoorbeeld geen processen of informatie stromen.

Vaak is het mogelijk om vanuit het model source code te genereren. Bijvoorbeeld wanneer het model een class diagram is, kunnen deze classes (inclusief attributen, operaties en relaties) gegenereerd worden. Natuurlijk is het model een abstractie van de werkelijkheid, waardoor de gegenereerde code nog geïmplementeerd moet worden. Het genereren van source code vanuit het model is optioneel. Het staat de

ontwikkelaar vrij om geheel zelf een implementatie aan de hand van het model te maken.

Bij model driven engineering is het belangrijk dat wanneer er iets in het ontwerp wijzigt altijd eerst het model aangepast dient te worden. Vanuit hier kan dan eventueel weer source code gegenereerd worden. Hierna kan de implementatie aangepast/aangevuld worden naar de wijzigingen in het model. In figuur 6 is de pijl een gestippeld om aan te geven dat deze stap niet geautomatiseerd verloopt. Voor deze stap zou reverse engineering ingezet kunnen worden, waarna we geen model driven engineering maar roundtrip engineering toepassen.

Omdat er altijd begonnen wordt met een model en de implementatie aan de hand van dit model gemaakt wordt, lopen deze altijd in sync. Hierdoor kan het model als documentatie gebruikt worden en blijft het overzichtelijk waar later eventueel aanpassingen gedaan moeten worden. Het genereren van source code betekent behalve tijdsbesparing ook een vermindering van de kans op fouten. De namen en id's uit het model worden namelijk één-op-één overgenomen, wat bij handmatig overtypen nog wel eens mis wil gaan.

Door model driven engineering toe te passen worden er richtlijnen over hoe het ontwikkelproces van projectspecifieke tools eruit dient te zien geïntroduceerd. Het ontbreken van richtlijnen is één van de geïdentificeerde knelpunten. Het voordeel van het gebruik van model driven engineering is dat er wel richtlijnen ontstaan, maar dat de ontwikkelaar nog steeds vrij is om "onder water" zijn eigen technieken/frameworks te gebruiken. Het model is immers een representatie op een bepaald abstractieniveau.

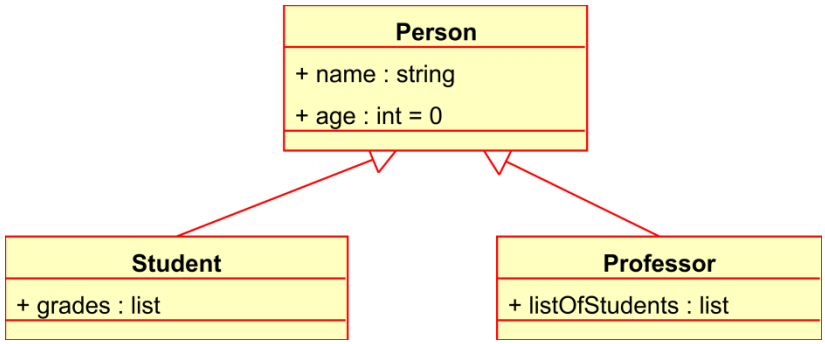
Het concept van model driven engineering zegt niets over welk model er gebruikt dient te worden. Om een keuze te maken welk modellen geschikt zijn om te gebruiken bij het ontwikkelen van projectspecifieke tools worden in dit hoofdstuk een aantal diagrammen besproken. De besproken diagrammen zijn de wat meer algemeen bekende diagrammen, omdat het niet de bedoeling is dat een ontwikkelaar van projectspecifieke tools veel tijd in het leren en/of maken van een diagram steekt. De diagrammen moeten dus gemakkelijk te doorgronden en modelleren zijn.

Als we naar de huidige toepassing van model driven engineering kijken zien we dat alleen source code gegenereerd wordt uit class diagrams, state machine diagrams en door middel van DSL gespecificeerde modellen. Om de leercurve van het model driven engineering zo vlak mogelijk te houden worden grafisch gevisualiseerde modellen geprefereerd boven tekst-gebaseerde modellen. Daarom worden in ieder geval het class diagram en state machine diagram besproken. Omdat de projectspecifieke tools die het vaakst ontwikkeld worden iets met data doen (bijvoorbeeld "code generators" of "data converters") is het data flow diagram ook zeer interessant, hoewel er op dit moment geen tool beschikbaar is die source code uit dit diagram kan genereren.

In sectie 3.1.1 wordt het class diagram besproken. Dit diagram uit de UML specificatie is bekend bij alle ontwikkelaars en staat heel dicht op de source code [UML]. Met dit diagram kan de technische specificatie gemodelleerd worden. Het data flow diagram, wat in sectie 3.1.2 besproken wordt, stamt uit de jaren '70 en modelleert de informatie stromen binnen een applicatie, wat betekent dat dit diagram de applicatie op functioneel niveau modelleert. Als laatste wordt in sectie 3.1.3 het state machine diagram besproken. Ook dit diagram modelleert software op functioneel niveau, maar dan als we naar processen kijken in plaats van naar informatie.

3.1.1 Class Diagram

Wanneer een object georiënteerde programmeertaal gebruikt wordt om een applicatie te ontwikkelen is het mogelijk om een class diagram te maken. Het class diagram geeft een visualisatie van classes en de relaties ertussen. Hierdoor zit het class diagram wat abstractieniveau betreft zeer dicht op de techniek. Het class diagram is onderdeel van de Unified Markup Language [UML] specificatie en eigenlijk alle ontwikkelaars zijn bekend met dit diagram.



Figuur 7. Class diagram

Figuur 7 laat een voorbeeld van een class diagram zien. We zien hier drie classes, waarbij "Person" een generalisatie is van "Student" en "Professor". "Person" heeft twee attributen, namelijk "name" en "age". De types staan hierbij aangegeven. De classes "Student" en "Professor" hebben ieder 1 attribuut.

3.1.1.1 Afbeelden op Java Code

Vanuit het class diagram kan rechtstreeks Java code gegenereerd worden door middel van een één-op-één afbeelding. Dit komt omdat het diagram op hetzelfde abstractie niveau als de source code zit. Echter, de implementatie van functies of methoden moet natuurlijk nog gedaan worden. Omdat het class diagram isomorf met de source code is, is reverse engineering van code naar model ook door middel van een één-op-één afbeelding mogelijk.

3.1.1.2 Afgedekte Knelpunten

3. **Weinig tot geen documentatie.** Het class diagram voorziet in technische documentatie. Wanneer de projectspecifieke tool later aangepast moet worden kan hierdoor gemakkelijk opgezocht worden waar de wijziging plaats zal moeten vinden of waar iets toegevoegd moet worden.

3.1.1.3 Niet-afgedekte knelpunten

1. **Niet geschikt voor evolutie.** Het class diagram biedt de mogelijkheid om op technisch niveau software te modelleren. Waar echter meer behoefte aan is bij het ontwikkelen van projectspecifieke tools is het feit dat er geen functionele specificatie is en dat deze verandert naarmate de tool ouder wordt. Functionele specificaties kunnen niet gemodelleerd worden met deze plugin.
2. **Niet geschikt voor build proces.** Het class diagram geeft evenveel vrijheid over hoe de tool opgezet wordt als wanneer de plugin niet gebruikt wordt. Hierdoor dwingt het de ontwikkelaar nog steeds niet om bijvoorbeeld user interface source code los te koppelen de source code van de processen.

4. **Te weinig overdracht aan PDC.** Het class diagram helpt niet om het geheel abstracter te bekijken. Hoewel er wel documentatie beschikbaar komt, door het gebruik van dit model, maakt het overdracht niet gemakkelijker.
5. **Project wordt geremd door afhankelijkheid tool.** Omdat er documentatie is, is het gemakkelijker om de tool aan te passen wanneer de gebruiksomgeving verandert. Echter, de documentatie staat zo dicht op de source code dat het te weinig toevoegt om gemakkelijk functionaliteit te kunnen aanpassen, vervangen en/of toe te voegen.
6. **Geen simpele richtlijnen.** Het gebruik van het class diagram op zich biedt geen richtlijnen om projectspecifieke tools te ontwikkelen. De enige richtlijn die opgesteld zou kunnen worden is dat dit diagram gebruikt dient te worden. Echter, hierdoor zal er geen aandacht aan de overige knelpunten gegeven worden, behalve dat er meer documentatie beschikbaar is.

3.1.1.4 Voor- en Nadelen

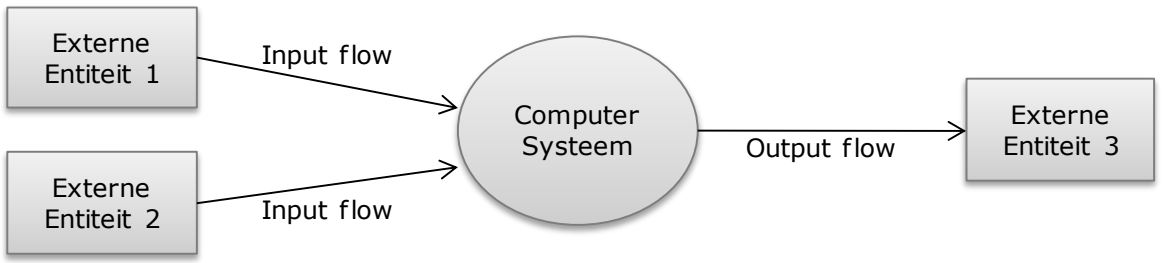
Model driven engineering waarbij een class diagram gebruikt wordt biedt het grote voordeel dat er altijd technische documentatie beschikbaar is. Omdat het model door middel van reverse-engineering geüpdate kan worden zullen model en source code nooit out-of-sync lopen.

Het nadeel van model driven engineering met een class diagram als model is dat er alleen maar technische documentatie beschikbaar komt, terwijl er juist meer behoefte is aan een functioneel ontwerp. Ook bij het gebruik van een class diagram is het namelijk mogelijk om code van verschillende functionaliteiten te mengen, waardoor het later lastig of misschien zelfs onmogelijk is om functionaliteit toe te voegen. Het gebruik van dit diagram voegt dus alleen inzicht toe hoe de class structuur in elkaar zit, maar brengt geen structuur in het ontwikkelproces.

3.1.2 Data Flow Diagram

Het Data Flow Diagram (DFD) brengt informatie zoals deze door software stroomt in kaart. Door middel van een serie transformaties wordt informatie verplaatst en gewijzigd, waardoor uiteindelijk de input van het computer systeem getransformeerd wordt in output. DFD's worden ook wel Data Flow Graphs of Bubble Charts genoemd [PRESS].

DFD's kunnen op meerdere abstractieniveaus getekend worden, waarbij een level 0 DFD het meest abstract is en het computer systeem als een "black box" ziet. In deze level 0 DFD worden alleen het computer systeem als één proces en de externe entiteiten met haar in- en outputs gedefinieerd.



Figuur 8. Level 0 Data Flow Diagram

Figuur 8 geeft een voorbeeld van een level 0 DFD. Hierbij krijgt het computer systeem informatie vanuit twee externe entiteiten en zal het output opleveren aan de derde

externe entiteit. Het diagram vertelt alleen dat deze informatiestromen er zijn, maar het zegt niets over wanneer welke informatieoverdracht plaatsvindt. Zo kan het zijn dat het computer systeem informatie verkrijgt van beide input entiteiten en daarna output geeft. Bij dit diagram zou het ook kunnen zijn dat het computer systeem informatie verkrijgt van één van de twee input entiteiten en daarna output geeft aan de output entiteit. Weer een totaal andere optie zou zijn dat er elke 10 seconden output gegeven wordt over een bepaalde status aan de output entiteit en af en toe input krijgt van de input entiteiten.

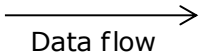
3.1.2.1 Elementen van het Diagram

In het Data Flow Diagram worden de volgende elementen onderscheiden:

Een externe entiteit consumeert en/of produceert informatie van/naar het gemodelleerde computer systeem, waarbij de entiteit zich buiten het computer systeem bevindt. Externe entiteiten kunnen andere computer systemen, hardware, software of mensen zijn, kortom alles wat acteert op het computer systeem.

Een proces bevindt zich binnen het gemodelleerde computer systeem en is een transformeerder van informatie. Het kan daarom ook als "functie" van het computer systeem gezien worden.

Data stores dienen als opslag van informatie binnen het te modelleren computer systeem en worden gebruikt door één of meerdere processen. Data stores kunnen zo simpel zijn als een buffer of simpele datastructuur, maar kunnen ook zo uitgebreid zijn als een RDBMS[2].



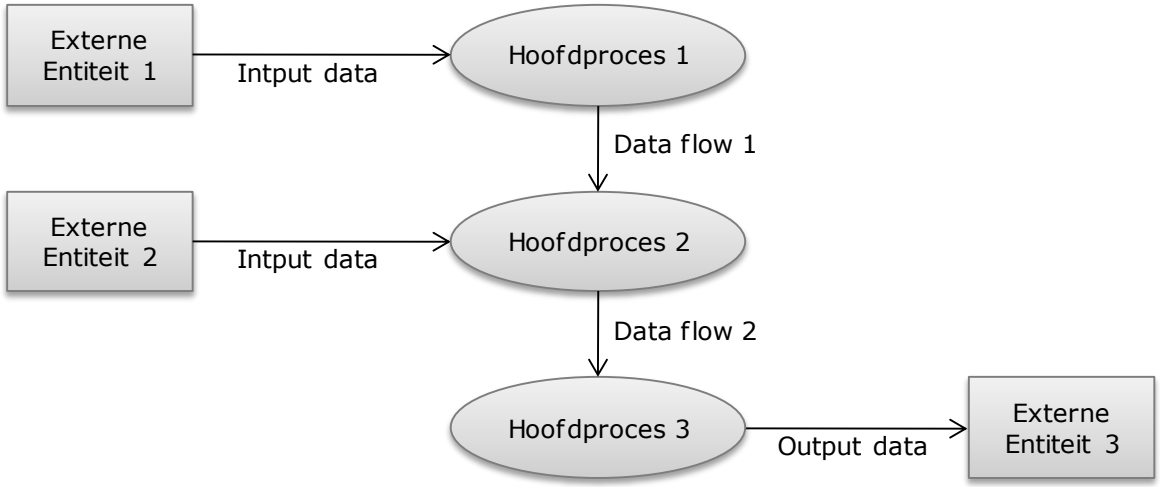
Data flows geven een informatiestroom met bijbehorende richting weer. Deze informatiestromen kunnen van en naar alle processen binnen het diagram getekend worden. Externe entiteiten kunnen alleen verbonden worden met processen, want informatiestromen tussen externe entiteiten vallen buiten het te modelleren systeem. Verder is er een proces nodig om informatie van een externe entiteit naar een data store over te brengen en vice versa, waardoor een informatiestroom tussen een externe entiteit en een data store niet mogelijk is.

3.1.2.2 Abstractieniveaus

Vanuit een level 0 DFD kan er "ingezoomd" worden op het proces dat het computer systeem representeert. Na dit inzoomen kunnen de belangrijkste functionaliteiten van het systeem gemodelleerd worden in de vorm van hoofdprocessen. Het diagram na één keer inzoomen wordt een level 1 DFD genoemd, na nogmaals inzoomen een level 2 DFD, etc. Het modelleren van de onderliggende niveaus wordt ook wel verfijning van de informatiestromen genoemd.

[2] RDBMS: Relational Database Management System

Bij het maken van een diagram dat één level hoger ligt dan een bestaand diagram, gaan we uit van het proces waarop ingezoomd wordt en nemen alle verbonden elementen en datastromen over. Hiervan maken we een nieuw diagram, zonder dat er nieuwe externe entiteiten toegevoegd worden. Figuur 9 is een voorbeeld van een level 1 DFD als er ingezoomd wordt op het "Computer Systeem" proces van het level 0 DFD uit bovenstaand voorbeeld.



Figuur 9. Level 1 Data Flow Diagram

Wanneer we nu een level 2 DFD van "Hoofdproces 1" willen maken, starten we met een diagram waar alleen "Externe Entiteit 1", "Hoofdproces 2" en de twee verbonden data flows op staan. Nu kunnen de subprocessen van "Hoofdproces 1" in dit nieuwe diagram gemodelleerd worden.

3.1.2.3 Afbeelden op Java Code

Om een DFD in combinatie met model driven engineering te gebruiken is een afbeelding van model naar source code noodzakelijk. In het geval van dit onderzoek dient het model naar Java code afgebeeld te worden.

Het afbeelden van een DFD op Java code is niet praktisch. Smart Developer Environment [SDE], zoals besproken zal worden in sectie 3.2.4, beeld processen af op een class met één gegenereerde functie die aangeroepen kan worden om het proces uit te voeren. Deze functie kan dan door de ontwikkelaar geïmplementeerd worden. Echter, alleen processen waarop niet meer ingezoomd kunnen worden kunnen op source code afgebeeld worden, want een proces dat subprocessen bevat is niet af te beelden op objectgeoriënteerde code. Dit komt omdat de pijlen informatiestromen aangeven en niet een volgorde waarop de processen uitgevoerd worden. Hierdoor kan het voor komen dat sommige processen uit het level 1 DFD op Java code afgebeeld kunnen worden, terwijl op sommige andere processen op hetzelfde niveau eerst dieper ingezoomd moet worden om bij de processen te komen die niet meer op te delen zijn en op source code afgebeeld kunnen worden.

In het diagram wordt wel gespecificeerd hoe data door de software loopt en er wordt met tekst aangegeven welke data van welk element naar een ander element van het diagram gaat. Echter, hoe deze data er precies uit ziet wordt niet gemodelleerd.

Hierdoor is het niet mogelijk om iets over de data te zeggen en kan hiervoor niets meer dan ongetypeerde objecten of commentaar in de code voor gegenereerd worden.

3.1.2.4 Afgedekte Knelpunten

1. **Niet geschikt voor evolutie.** Vanuit het model is duidelijk welke processen er zijn. Wanneer de projectspecifieke tool later aangepast moet worden kan hierdoor gemakkelijk opgezocht worden waar de wijziging plaats zal moeten vinden of waar iets toegevoegd moet worden. Aangezien er vanaf het begin af aan in termen van "processen" en "informatiestromen" binnen de tool gedacht wordt kan een proces, wat redelijk op zichzelf staand is, gemakkelijk vervangen worden of een nieuw proces worden toegevoegd.
2. **Niet geschikt voor build proces.** Met het DFD worden processen en de informatiestromen ertussen gemodelleerd. Het zegt niets over de user interface. De user interface dient apart ontwikkeld te worden en is ervoor verantwoordelijk de juiste processen op de juiste momenten uit te voeren. Dat het DFD niets over de user interface zegt, zorgt ervoor dat de user interface loosly coupled is. Hierdoor kan in plaats van een user interface ook een build server de processen, waarin het "werk" van de tool gedaan wordt, uitvoeren.
6. **Geen simpele richtlijnen.** Zoals eerder beschreven leidt het gebruik van model driven engineering automatisch tot een richtlijn, namelijk dat bij elke iteratie van het ontwikkelproces eerst het model wordt aangepast, welke op source code afgebeeld wordt en hierna kan de source code bijgewerkt en/of geïmplanteerd worden. Door een DFD als model te gebruiken wordt de ontwikkelaar er op een soepele manier toe gedwongen om de tool te modelleren in de vorm van processen waar informatie doorheen stroomt. Dit is alleen maar op functioneel niveau. De gegenereerde code zal de ontwikkelaar niet tegenhouden om bepaalde (eigen) technieken of frameworks te gebruiken.

3.1.2.5 Niet-afgedekte Knelpunten

3. **Weinig tot geen documentatie.** Door model driven engineering toe te passen tijdens het ontwikkelen van een projectspecifieke tool, met een DFD als model is er continu documentatie beschikbaar. Omdat elke wijziging begint met een wijziging in het model en daarna in de source code, blijft het model, en dus de documentatie, up-to-date. De documentatie is echter niet compleet, aangezien alleen processen en subprocessen, met bijbehorende informatiestromen gemodelleerd kunnen worden. Er is namelijk nog aanvullende documentatie nodig over hoe de informatie die er stroomt er precies uit ziet.
4. **Te weinig overdracht aan PDC.** Om een projectspecifieke tool gemakkelijk over te kunnen dragen is het belangrijk dat er goede documentatie beschikbaar is. Alles wat niet (goed) gedocumenteerd is zal persoonlijk overgedragen moeten worden. Het zou handig zijn wanneer het model, in dit geval een DFD, met één druk op de knop naar het PDC gemaild zou kunnen worden. Echter, het model is geen volledige documentatie omdat er wel gemodelleerd wordt welke informatiestromen er zijn, maar niet wat voor informatie in elke stroom zit, waardoor de overdracht hiermee niet versimpeld wordt en dit knelpunt blijft bestaan.
5. **Project wordt geremd door afhankelijkheid tool.** Om dit knelpunt op te lossen moeten projectspecifieke tools gemakkelijk aan te passen zijn aan de omgeving wanneer de omgeving verandert. Dit is maar gedeeltelijk het geval. Wanneer een proces aangepast of toegevoegd moet worden is er geen probleem. Het model voorziet echter niet in de mogelijkheid om te definiëren hoe de informatie die tussen de processen stroomt te modelleren, waardoor een eventuele verandering hierin niet anders is dan in de huidige situatie. Om dit knelpunt op

te lossen zullen de data flows en data stores dus preciezer gedefinieerd moeten worden.

3.1.2.6 Voor- en Nadelen

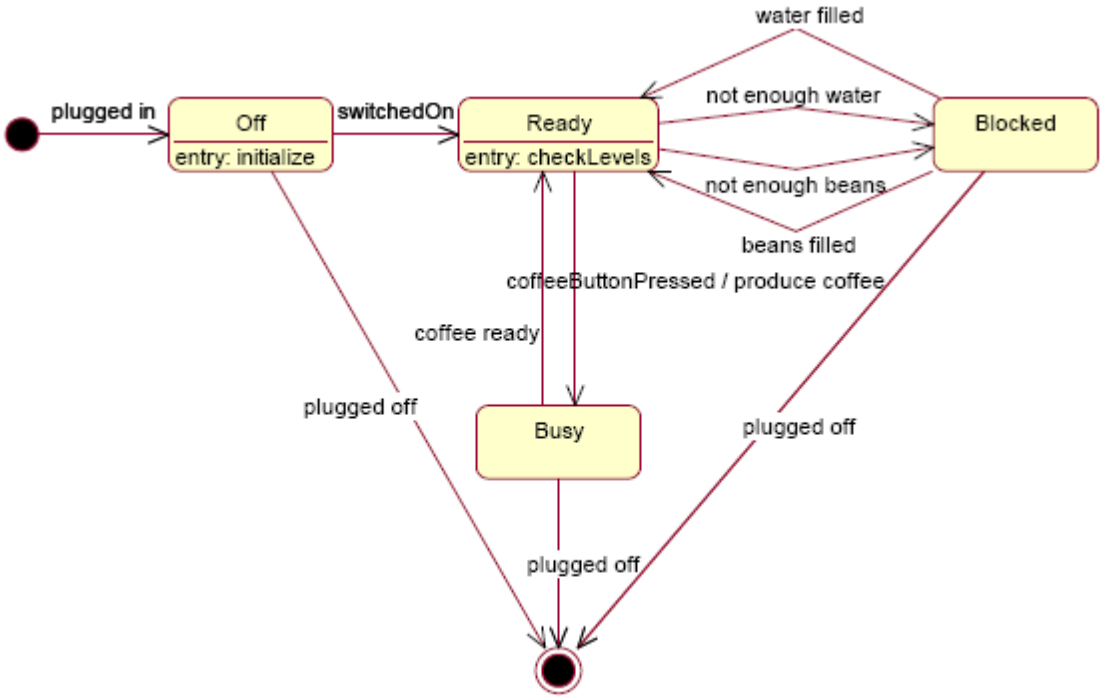
Het grote voordeel van model driven engineering, waarbij een DFD als model gebruikt wordt is dat de informatiestromen binnen de projectspecifieke tools gemodelleerd kunnen worden. We zagen al dat de meestgebruikte projectspecifieke tools, zoals "code generators" en "data converters" data als in- en output hebben en intern data transformeren. Hiervoor is het inzetten van een DFD ideaal.

Omdat alleen de stromen gemodelleerd kunnen worden, maar niet hoe de data per stroom eruit ziet is het model niet volledig genoeg om alle knelpunten op te lossen. Dit geldt voor zowel de data flows als de data stores. Hiervoor zouden aanvullende diagrammen gebruikt moeten worden. Echter, het koppelen van andere diagrammen aan een DFD, waarbij voor elk pijltje en elke data store een apart diagram ontstaat maakt het geheel niet overzichtelijker.

Zoals eerder gezegd kennen DFD's meerdere abstractieniveaus. Hierdoor zal het regelmatig voor komen dat sommige processen uit een level 0 DFD "atomair" zijn en niet verder op te delen in subprocessen, terwijl andere processen wel verder opgedeeld kunnen worden. Deze laatste eventueel zelfs in nog meer niveaus. Hierdoor zien we dat op bijna alle lagen atomaire processen voor komen. Alleen atomaire processen kunnen op source code afgebeeld worden, waardoor het diagram eigenlijk te ver van source code af staat. Het overzicht is dan namelijk te snel zoek.

3.1.3 State Machine Diagram

Met een state machine diagram kunnen applicaties gemodelleerd worden waarbij de applicatie bij het uitvoeren zich steeds in een bepaalde toestand bevindt. Door het uitvoeren van een proces komt de applicatie in een volgende (eventueel dezelfde) toestand terecht. Dit proces wordt een transitie genoemd. Het state diagram modelleert een applicatie, net als het Data Flow Diagram, op functioneel niveau.



Figuur 10. State machine diagram

Figuur 10 geeft een voorbeeld van een state machine diagram die een bepaald koffiezetapparaat modelleert. We zien hier dat het koffiezetapparaat in 4 "toestanden" kan verkeren. De pijlen geven de transities aan. De tekst bij de transities zijn van de vorm "A / B", waarbij de slash en B regelmatig weggelaten wordt. De tekst bij de transitie moet gelezen worden als "de transitie is alleen beschikbaar als A. Tijdens de transitie wordt B uitgevoerd. Bijvoorbeeld bij de transitie van "Ready" naar "Busy": wanneer er op de coffee button gedrukt wordt zal er koffie geproduceerd gaan worden en is het koffiezetapparaat "Busy". Als de koffie klaar is gaat het koffiezetapparaat weer van "Busy" naar "Ready". Het dichte rondje geeft aan waar de machine start en het rondje met dubbele rand geeft een "final state" aan. Van deze laatste kunnen er meerdere voor komen in één diagram.

Behalve het modelleren van apparaten kan ook software gemodelleerd worden met state machine diagrams. Als we naar projectspecifieke tools kijken, dan zien we dat het regelmatig mogelijk is om een state machine diagram te tekenen. Zoals eerder gezegd doen de meeste projectspecifieke tools iets met data (zoals "code generators" en "data converters"). Als het generatie- of conversie-proces meerdere stappen beslaat, waarbij de volgorde niet per definitie vast staat kan het erg handig zijn om een machine state diagram te modelleren om de stappen in kaart te brengen. Veel tools voeren een proces uit dat uit een aantal stappen bestaat, maar waarbij de volgorde van de stappen vast staat. In dat geval is het state machine diagram niet nuttig.

3.1.3.1 Afbeelden op Java Code

Handmatig een state machine implementeren zal de eerste keer niet gemakkelijk zijn. Er moet namelijk continu geadministreerd worden in welke toestand de applicatie zich bevindt, welke transities beschikbaar zijn en/of welke transitie uitgevoerd wordt.

Gelukkig zijn er allerlei tools beschikbaar die code kunnen genereren uit een state machine diagram. Deze zorgen ervoor dat behalve code voor de toestanden ook code voor de administratie gegenereerd wordt.

De toestanden kunnen ieder als class gegenereerd worden. Transities zijn member-functies van de bron-toestand en leveren een bestemmings-toestand op. Op elke toestand kunnen acties gedefinieerd worden, waarbij de "entry"-actie altijd uitgevoerd wordt bij het binnenkomen van een toestand. Vanuit de acties moet beslist worden wanneer welke transitie uitgevoerd moet worden.

De ontwikkelaar zal zorgvuldig op moeten letten dat hij geen dubbele code schrijft. Het komt namelijk vaak voor dat er meerdere transities zijn die naar een bepaalde toestand toe gaan en eigenlijk precies dezelfde functionaliteit implementeren. De "plugged off" transitie in het voorbeeld demonstreert dit. Er moet voor gezorgd worden dat deze functionaliteit niet in iedere toestand opnieuw geïmplementeerd wordt.

3.1.3.2 Afgedekte Knelpunten

1. **Niet geschikt voor evolutie.** Door middel van het diagram wordt duidelijk gemaakt welke toestanden en transities tussen toestanden er zijn. Wanneer de projectspecifieke tool later aangepast moet worden kan hierdoor gemakkelijk opgezocht worden waar de wijziging plaats zal moeten vinden of waar iets toegevoegd moet worden. Aangezien er vanaf het begin af aan in termen van toestanden en transities binnen de tool gedacht wordt kan een toestand of transitie gemakkelijk vervangen worden of een nieuwe worden toegevoegd.
2. **Niet geschikt voor build proces.** Met het state machine diagram worden toestanden en transities ertussen gemodelleerd. Het zegt niets over de user interface. De user interface dient apart ontwikkeld te worden en is ervoor verantwoordelijk de juiste processen op de juiste momenten uit te voeren. Dat het diagram niets over de user interface zegt, zorgt ervoor dat de user interface loosly coupled is. Hierdoor kan in plaats van een user interface ook een build server de transities, waarin het "werk" van de tool gedaan wordt, uitvoeren.
6. **Geen simpele richtlijnen.** Zoals eerder beschreven leidt het gebruik van model driven engineering automatisch tot een richtlijn, namelijk dat bij elke iteratie van het ontwikkelproces eerst het model wordt aangepast, welke op source code afgebeeld wordt en hierna kan de source code bijgewerkt en/of geïmplanteerd worden. Door een state machine diagram als model te gebruiken wordt de ontwikkelaar er op een soepele manier toe gedwongen om de tool te modelleren in de vorm van toestanden en transities. Dit is alleen maar op functioneel niveau. De gegenereerde code zal de ontwikkelaar niet tegenhouden om bepaalde (eigen) technieken of frameworks te gebruiken.

3.1.3.3 Niet-afgedekte Knelpunten

3. **Weinig tot geen documentatie.** Door model driven engineering toe te passen tijdens het ontwikkelen van een projectspecifieke tool, met een machine state diagram als model is er continu documentatie beschikbaar. Omdat elke wijziging begint met een wijziging in het model en daarna in de source code, blijft het model, en dus de documentatie, up-to-date. De documentatie is echter niet compleet. Aangezien de meeste projectspecifieke tools iets met informatie doen zal er aanvullende documentatie nodig zijn van de benodigde datastructuren en informatie stromen.
4. **Te weinig overdracht aan PDC.** Om een projectspecifieke tool gemakkelijk over te kunnen dragen is het belangrijk dat er goede documentatie beschikbaar is.

Alles wat niet (goed) gedocumenteerd is zal persoonlijk overgedragen moeten worden. Het zou handig zijn wanneer het model, in dit geval een state machine diagram, met één druk op de knop naar het PDC gemaild zou kunnen worden. Echter, het model biedt, zoals bij punt 3 aangegeven is, geen volledige documentatie, waardoor de overdracht hiermee niet versimpeld wordt en dit knelpunt blijft bestaan.

5. **Project wordt geremd door afhankelijkheid tool.** Om dit knelpunt op te lossen moeten projectspecifieke tools gemakkelijk aan te passen zijn aan de omgeving wanneer de omgeving verandert. Dit is maar gedeeltelijk het geval. Wanneer een toestand of transitie aangepast of toegevoegd moet worden is er geen probleem. Het model voorziet echter niet in de mogelijkheid om te definiëren hoe de informatie en informatiestromen eruit zien, waardoor een eventuele verandering hierin niet anders is dan in de huidige situatie. Om dit knelpunt op te lossen zal dus meer documentatie beschikbaar moeten zijn en deze moet constant up-to-date gehouden worden.

3.1.3.4 Voor- en Nadelen

Voor redelijk wat projectspecifieke tools kan het nuttig zijn om een state machine diagram in te zetten. Hierbij worden de mogelijke volgorde van het uitvoeren van de processen gemodelleerd, waardoor het de projectspecifieke tool op een goed abstractieniveau kan modelleren. Een nadeel is dat dit geen volledig ontwerp geeft. De datastructuren worden bijvoorbeeld niet gemodelleerd.

Dat het mogelijk is om Java code vanuit een state machine diagram te modelleren is een groot voordeel. Het moeilijke gedeelte van de implementatie, zoals administratie bijhouden wordt gegenereerd. Echter, het denken in toestanden en transities zal niet voor elke ontwikkelaar meteen intuïtief zijn, waardoor er een drempel zal zijn om dit diagram te gaan gebruiken.

3.2 Eclipse Plugins

Behalve het ontwikkelproces aan te passen kunnen ook bestaande “tools” worden ingezet om de eerdergenoemde knelpunten op te lossen, waardoor we naar oplossingen in de “tools”-laag van software engineering zoeken. Voor aanvang van het onderzoek was gesteld dat de focus bij het zoeken naar oplossingen op Java/Eclipse zou moeten liggen, omdat het onderzoek gedaan wordt vanuit het competence center Java van Info Support. In dit gebied zijn alle belangrijke tools uitgevoerd als Eclipse plugin [ECLIPSE]. Vandaar dat alle besproken tools hier dan ook Eclipse plugins zijn.

Om aan te sluiten bij de oplossingen uit de “processes”-laag van software engineering hebben we bij het zoeken naar Eclipse plugins die oplossingen voor de eerder geïdentificeerde knelpunten kunnen bieden, gefocust op plugins die model driven engineering ondersteunen.

De Eclipse ontwikkelomgeving bestaat zelf uit een kleine basis, die standaard aangevuld is met een groot aantal plugins, die sterk geïntegreerd zijn. De gebruiker van Eclipse zal dan ook niet zomaar zien wat in welke plugin te vinden is. Er kunnen verschillende versies van Eclipse gedownload worden, waarbij alleen het aantal standaard meegeleverde plugins verschilt.

Een plugin voor Eclipse die standaard met de ontwikkelomgeving mee geïnstalleerd wordt, is een plugin die voorziet in zogenaamde cheat sheets. Behalve het gebruiken van cheat sheats kunnen hier ook cheat sheets mee gemaakt worden. Hoewel deze cheat sheets in principe niets met model driven engineering te maken hebben, kunnen ze wel richtlijnen bieden, zodat de ontwikkelaar verteld kan worden wat hij wanneer

dient te doen, zonder dat hier veel documentatie voor gelezen hoeft te worden. Sectie 3.2.1 bespreekt cheat sheets en zal ook laten zien dat het meer voordelen biedt dan alleen richtlijnen. Op deze manier kan de ontwikkelaar begeleid worden met het maken van het model en het genereren van source code uit dit model.

Wanneer Eclipse geïnstalleerd wordt bevat deze maar een zeer beperkt aantal plugins. Vanuit de ontwikkelomgeving kunnen aanvullende plugins geïnstalleerd worden. Dit kunnen plugins zijn die door het Eclipse-ontwikkelteam ondersteund worden of 3rd party plugins. Het is natuurlijk aan te bevelen om eerst naar plugins te kijken die ondersteund worden door het Eclipse-ontwikkelteam, oftewel beschikbaar zijn via de "Eclipse Discovery Site" (Eclipse 3.3). Hier vinden we een tweetal plugins die een oplossing voor de geïdentificeerde knelpunten zouden kunnen bieden en aansluiten bij het model driven engineering principe, namelijk "UML2 Tools" en het "Eclipse Modeling Framework". Deze twee plugins worden in respectievelijk sectie 3.2.1 en sectie 3.2.3 besproken. Alle plugins van de "Eclipse Discovery Site" zijn net zoals de Eclipse ontwikkelomgeving open-source.

Verschillende websites[3] bieden een repository met 3rd party plugins. De meeste plugins zijn in de hobby-sfeer ontwikkeld en bieden weinig functionaliteit en/of is er een lange tijd niet meer aan ontwikkeld. "Smart Developer Environment for Eclipse" is een commerciële 3rd party Eclipse-plugin en is ontwikkeld door het bedrijf Visual Paradigm. Dit pakket blijft in ontwikkeling en biedt veel mogelijkheden op een grafische manier diagrammen te modelleren en om model driven engineering toe te passen. Deze plugin is goed onderhouden en er zijn op dit moment geen redenen om eraan te twijfelen dat dit in de nabije toekomst gaat veranderen.

3.2.1 Cheat Sheets

Cheat sheets zijn een hulpmiddel in Eclipse om documentatie aan te bieden, zonder dat er pagina's aan tekst gelezen hoeft te worden. Een cheat sheet is een checklist van wat een ontwikkelaar moet doen om tot een bepaald doel te komen. Zo is er bijvoorbeeld een cheat sheet om een EMF Project op te zetten.

Aan elke stap kan een help-pagina met meer uitleg en een actie gekoppeld worden. Deze actie doet iets voor de gebruiker in de Eclipse ontwikkelomgeving. Een voorbeeld van een actie is dat de "New EMF project" wizard opgestart wordt door op de link van de actie in de cheat sheet te klikken. Normaal gesproken zou de ontwikkelaar via het menu "File" > "New project..." > "EMF Project" hetzelfde effect kunnen bereiken. Deze acties bieden een serieuze meerwaarde dan alleen een lijstje van punten die uitgevoerd dienen te worden.

Het is ook mogelijk om zelf cheat sheets te ontwikkelen. Hierdoor kan het ontwikkelproces van projectspecifieke tools meer gestandaardiseerd worden en kunnen er punten aangegeven worden waarop gelet moet worden. Zo kan er bij elke cheat sheet aangegeven worden dat het misschien wel handig is als de tool op dat moment overgedragen wordt aan het PDC. De ontwikkelaar kan hier dan over nadenken.

3.2.1.1 Afgedekte Knelpunten

6. Geen simpele richtlijnen. Door cheat sheets te ontwikkelen kunnen richtlijnen opgesteld worden en hierdoor kan om aandacht gevraagd worden, waar een ontwikkelaar normaal even niet aan zou denken. Verder kunnen er tips gegeven worden die het ontwikkelproces versnellen.

[3] Bijvoorbeeld http://www.Eclipse-plugins.info en http://www.Eclipseplugincentral.com

3.2.1.2 Niet-afgedekte Knelpunten

1. **Evolutionaire karakter.** Cheat sheets kunnen helpen om tips te geven, maar ze zijn en blijven een vorm van statische documentatie, waardoor ze het knelpunt niet geheel afdekken.
2. **Niet geschikt voor build proces.** Wanneer het ontwikkelproces van projectspecifieke tools vaster staat, kan er een gerichtere richtlijn opgesteld worden over hoe een tool in het build proces opgenomen kan worden. Echter, alleen richtlijnen zijn niet genoeg om af te dwingen dat bijvoorbeeld user interface code losgekoppeld is van de achterliggende code. Hiervoor is meer nodig, waardoor dit punt niet geheel afgedekt is.
3. **Weinig tot geen documentatie.** Cheat sheets bieden geen documentatie over de te ontwikkelen projectspecifieke tools. Ze geven alleen documentatie voor het ontwikkelproces.
4. **Te weinig overdracht aan PDC.** Door richtlijnen voor het ontwikkelproces in te voeren wordt de overdracht aan PDC versimpeld, maar er zal ook nog goede documentatie nodig zijn. Daarom helpen cheat sheets bij de overdracht, maar ze zijn niet voldoende om het hele knelpunt af te dekken.
5. **Project wordt geremd door afhankelijkheid tool.** Omdat de knelpunten "te weinig tot geen documentatie" en "evolutionaire karakter" niet afgedekt zijn, wordt dit knelpunt ook niet afgedekt.

3.2.2 UML2 Tools

UML2 Tools is een plugin voor Eclipse waarmee de volgende UML-diagrammen grafisch gemodelleerd kunnen worden: class diagram, state machine diagram, profile diagram, component diagram, activity diagram en deployment diagram. Alleen vanuit het class diagram kan Java code gegenereerd worden [UML2T].

UML2 Tools implementeren de volledige UML 2.1 standaard voor de ondersteunde diagrammen. Dit betekent dat de mogelijkheden die deze plugin biedt zeer uitgebreid zijn.

3.2.2.1 Class Diagram

Het class diagram zit wat betreft abstractieniveau het dichtste op de code. Vanuit dit diagram is het dan ook mogelijk om code te genereren. Dit kan echter niet rechtstreeks. Hiervoor wordt gebruik gemaakt van het Eclipse Modeling Framework [EMF], wat in sectie 3.2.3 uitgelegd wordt.

Het class diagram uit de UML2 Tools zijn vooral bedoeld om op UML-gebied zo compleet mogelijk te zijn. Zo kunnen een UML class diagram van een andere applicatie ingelezen worden, of kan het diagram als uitgebreide documentatie gebruikt worden. Wanneer er code uit het diagram gegenereerd gaat worden, kan maar een beperkte set van het diagram gebruikt worden, namelijk de set die door EMF ondersteund wordt.

3.2.2.2 State Machine Diagram

Het state machine diagram geeft alle "toestanden" in het computer systeem weer. Verder worden mogelijke transities van toestand naar toestand weergegeven. Deze transities kunnen gezien worden als processen. Deze processen zijn echter niet uniek. Om een klein voorbeeld te geven: we hebben een web-based applicatie. We zouden elk scherm als toestand kunnen zien. Op elk scherm (behalve het aanmeld-scherm) staat een knop "afmelden". De functionaliteit voor deze knop is elke keer hetzelfde proces. Echter, in het state diagram zijn dit verschillende transities.

Aangezien de processen uiteindelijk geïmplementeerd worden, is het overzicht over welke processen er zijn al snel zoek in het state machine diagram. Dit maakt de afbeelding van diagram op source code lastig.

Het state machine diagram zou wel ingezet kunnen worden als aanvulling op een data flow diagram, waarbij voor de transities in het state machine diagram alleen uit de processen uit het data flow diagram gekozen kan worden. Echter, een data flow diagram is geen onderdeel van de UML standaard en wordt niet ondersteund door UML2 Tools.

3.2.2.3 Profile Diagram

Het profile diagram beschrijft welke services er organisatie-breed in een te ontwikkelen computer systeem beschikbaar moeten zijn en hoe deze aan elkaar gekoppeld zijn. Elke service kan weer gezien worden als apart computersysteem.

Dit diagram is vooral bedoeld voor architecten, waarbij de services later door designers ingevuld kunnen worden. Dit maakt dat het profile diagram niet geschikt is om te gebruiken bij het ontwikkelen van projectspecifieke tools.

3.2.2.4 Component Diagram

Het component diagram heeft veel weg van het profile diagram, met het verschil dat het niet alleen bedoeld is om de verschillende componenten van het computer systeem te identificeren, maar juist om de interfaces tussen de componenten vast te leggen. Ook bij dit diagram geldt dat het bedoeld is om grote systemen mee te ontwerpen, en het dus onbruikbaar is als ondersteuning van het ontwikkelproces van projectspecifieke tools.

3.2.2.5 Activity Diagram

Een Activity diagram is bedoeld om een bedrijfsproces of een gedeelte hiervan te modelleren. Zo kunnen hier informatie stromen en object stromen mee gedefinieerd worden. Aangezien projectspecifieke tools niet als bedrijfsprocessen gezien kunnen worden is dit diagram onbruikbaar om het ontwikkelen van projectspecifieke tools te ondersteunen.

3.2.2.6 Deployment Diagram

Door middel van het deployment diagram kan gemodelleerd worden welk gedeelte van het te ontwikkelen/ontwikkelde computer systeem waar geïnstalleerd/deployed moet worden. Dit diagram is alleen geschikt wanneer een computer systeem over meerdere lokaties, omgevingen en/of computers verdeeld moet worden. Aangezien projectspecifieke tools altijd op één computer draaien heeft het gebruiken van het deployment diagram geen nut bij dit soort software.

3.2.2.7 Afgedekte Knelpunten

3. **Weinig tot geen documentatie.** Vanuit het model is duidelijk welke processen er zijn. Wanneer de projectspecifieke tool later aangepast moet worden kan hierdoor gemakkelijk opgezocht worden waar de wijziging plaats zal moeten vinden of waar iets toegevoegd moet worden. Aangezien er vanaf het begin af aan in termen van "processen" en "informatiestromen" binnen de tool gedacht wordt kan een proces, wat redelijk op zichzelf staand is, gemakkelijk vervangen worden of een nieuw proces worden toegevoegd.

3.2.2.8 Niet-afgedekte knelpunten

1. **Niet geschikt voor evolutie.** UML2 Tools biedt de mogelijkheid om op technisch niveau software te modelleren. Waar echter meer behoefte aan is bij het ontwikkelen van projectspecifieke tools is het feit dat er geen functionele specificatie is en dat deze verandert naarmate de tool ouder wordt. Functionele specificaties kunnen niet gemodelleerd worden met deze plugin.
2. **Niet geschikt voor build proces.** De UML2 Tools plugin geeft evenveel vrijheid over hoe de tool opgezet wordt als wanneer de plugin niet gebruikt wordt. Hierdoor dwingt het de ontwikkelaar nog steeds niet om bijvoorbeeld user interface code los te koppelen van de source code van de processen.
4. **Te weinig overdracht aan PDC.** Deze plugin helpt niet om het geheel abstracter te bekijken. Hoewel er wel documentatie beschikbaar komt, door het gebruik van deze plugin, maakt het overdracht niet erg veel gemakkelijker.
5. **Project wordt geremd door afhankelijkheid tool.** Omdat er documentatie is, is het gemakkelijker om de tool aan te passen wanneer de gebruiksomgeving verandert. Echter, de documentatie staat zo dicht op de source code dat het te weinig toevoegt om gemakkelijk functionaliteit te kunnen aanpassen, vervangen of toe te voegen.
6. **Geen simpele richtlijnen.** Het gebruik van de UML2 Tools plugin op zich biedt geen richtlijnen om projectspecifieke tools te ontwikkelen. De enige richtlijn die opgesteld zou kunnen worden is dat een bepaald diagram gebruikt dient te worden. Echter, hierdoor zal er geen aandacht aan de overige knelpunten gegeven worden, behalve dat er meer documentatie beschikbaar is.

3.2.2.9 Voor- en Nadelen

Het grote voordeel van UML2 Tools is dat de ontwerper/ontwikkelaar een aantal modellen in kan zetten, welke ook weer als documentatie gebruikt kunnen worden. Echter, de modellen zijn vooral bedoeld voor grote software systemen of ze staan zo dicht op de code dat ze geen richtlijnen afdwingen.

3.2.3 Eclipse Modeling Framework

Met behulp van het Eclipse Modeling Framework [EMF] kan een class diagram gemodelleerd worden. In tegenstelling tot het class diagram uit de UML2 Tools, zoals besproken in sectie 3.2.1, ondersteunt dit diagram niet alle UML specificaties.

Het diagram uit deze plugin voor Eclipse is er echter wel geheel op toegespitst dat alles wat gemodelleerd kan worden ook daadwerkelijk naar Java code vertaald kan worden. Ook de andere kant op is mogelijk met dit pakket: vanuit geannoteerde Java code kan een model gegenereerd worden. Deze weg is echter vooral geschikt om van eerder gegenereerde code opnieuw een model te maken, aangezien er bij het annoteren op een aantal zaken gelet moet worden.

Het proces van model naar Java code gaat als volgt. Eerst wordt er een Ecore model gemaakt. Door een grafische editor te gebruiken kan dit visueel. Vanuit het Ecore model kan een gen model afgeleid worden. Dit gen model heeft een verwijzing naar het Ecore model, waardoor geen informatie redundant opgeslagen wordt. De reden van de scheiding tussen het Ecore model en het gen model is dat in het Ecore model alleen properties van de elementen in het class diagram te vinden zijn, terwijl de properties voor het genereren van Java code in het gen model staan. Vanuit het gen model kan Java code gegenereerd worden.

Zoals gezegd kan een grafische editor gebruikt worden om het Ecore model visueel te bewerken. Wanneer het Graphical Modeling Framework [GMF] geïnstalleerd wordt, is

direct een grafische editor voor Ecore bestanden beschikbaar. Dit is namelijk een example van de GMF plugin. TOPCASED Ecore Editor is een uitgebreidere, net als GMF open-source, grafische editor voor Ecore bestanden [TOPCASED].

EMF is zeer geschikt om data modellen mee te ontwerpen en genereren. Vooral aangezien dit vaak routinewerkzaamheden zijn, maar wel precies komt. Het grote voordeel van EMF is dat wanneer twee classes een relatie hebben, bijvoorbeeld een aggregatie, dan wordt ervoor gezorgd dat er automatisch verwijzingen en/of lijsten van verwijzingen gemaakt worden. Zo kunnen objecten uit een één-op-veel relatie ook maar één ouder hebben, wat door de gegenereerde code van het framework afgedwongen wordt.

3.2.3.1 Afgedekte en Niet-afgedekte Knelpunten

De afgedekte en niet-afgedekte knelpunten zijn hetzelfde als bij de UML2 Tools plugin, zoals te vinden in secties 3.2.2.7 en 3.2.2.8.

3.2.3.2 Voor- en Nadelen

EMF is te vergelijken met het class diagram van UML2 Tools, met dien verstande dat EMF toegespitst is op Java/Eclipse, terwijl er bij het ontwikkelen van UML2 Tools uitgegaan is van UML diagrammen.

Een groot voordeel van EMF is dat vanuit elk Ecore model Java code gegenereerd kan worden. Niet alle elementen uit het class model van de UML2 Tools plugin kunnen naar code gegenereerd worden.

Behalve voor projectspecifieke tools, kan EMF ook voor andere (grotere) projecten ingezet worden. Het is vrij gemakkelijk om te leren en haalt routinewerkzaamheden uit handen. Verder is de kans op fouten kleiner wanneer code gegenereerd wordt.

Een nadeel van EMF is dat er standaard geen grafische editor voor Ecore modellen meegeleverd wordt. Het Graphical Modeling Framework of de TOPCASED Ecore Editor installeren zijn hiervoor een oplossing.

3.2.4 Smart Developer Environment for Eclipse

Smart Developer Environment is een commercieel, niet open-source product van Visual Paradigm [VP]. Het is beschikbaar voor een aantal ontwikkelomgevingen, waaronder Eclipse en biedt allerlei tools en diagrammen om model driven engineering aan de ontwikkelomgeving toe te voegen. Dit gebeurt door middel van plugins.

Op het moment van schrijven kunnen 24 verschillende typen diagrammen gemodelleerd worden met dit pakket. Net zoals bij UML2 Tools zijn de meeste diagrammen vooral bedoeld voor grote software pakketten. Sommige diagrammen waardoor er weinig diagrammen over blijven die nuttig kunnen zijn bij het ontwikkelen van projectspecifieke tools. Deze diagrammen worden hieronder besproken.

3.2.4.1 Class Diagram

Het class diagram is het welbekende diagram zoals in de UML specificaties te vinden is. Vanuit dit diagram kan Java code gegenereerd worden. Deze gegenereerde code is wat aan de summiere kant. Bijvoorbeeld wanneer we twee classes modelleren met een aggregatie relatie ertussen, dan worden er netjes variabelen gegenereerd: één als reference naar de containments (een lijst) en één als reference naar de container. Echter, er worden geen functies gegenereerd om items aan de container toe te

voegen, waarbij de containment-variabele ook automatisch toegewezen wordt. EMF, zoals besproken in sectie 3.2.3, doet dit wel.

3.2.4.2 State Machine Diagram

Voor elke class in een class diagram kan een state machine diagram gemodelleerd worden. Natuurlijk worden ook op zichzelf staande state machine diagrammen ondersteund, maar als ze aan een class gekoppeld zijn is het mogelijk om er Java code uit te genereren. In de gegenereerde class code komt een verwijzing naar een zogenaamde "context", waarin het state machine diagram geïmplementeerd wordt.

Het state machine diagram modelleert het proces dat door de gebruiker van de te ontwikkelen projectspecifieke tool doorlopen kan worden en is daarom vooral handig wanneer een projectspecifieke tool ontwikkeld gaat worden die een bepaald proces implementeert.

Data en informatie stromen kunnen niet gemodelleerd worden met dit diagram. Wanneer de projectspecifieke tool veel met data doet, bijvoorbeeld omdat het om een "code generator" of een "data converter" gaat, dan is dit diagram niet zo handig, aangezien dit niet gemodelleerd kan worden.

3.2.4.3 Data Flow Diagram

Om projectspecifieke tools waarin een informatie stroom gemodelleerd zou kunnen worden te ontwikkelen kan het data flow diagram gebruikt worden. Meer over dit diagram staat uitgelegd in sectie 3.1.2.

Met Smart Developer Environment kunnen data flow diagrams gemodelleerd worden. De plugin biedt veel opties om het diagram als documentatie te gebruiken. Eventueel kunnen andere diagrammen gekoppeld worden. Wat echter niet kan is code genereren vanuit het diagram. Hierdoor kan het niet gebruikt worden in een omgeving waar model driven engineering wordt toegepast.

3.2.4.4 Afgedekte Knelpunten

1. **Evolutionaire karakter.** Door model driven development toe te passen dwing je jezelf om een diagram te maken als ontwerp. Hierdoor werk je automatisch op een manier die meer gestructureerd is dan de ad-hoc werkwijze die nu vooral wordt toegepast. Het dwingt je verder om beter over het ontwikkelproces na te denken. Later kan er altijd op het diagram teruggevallen worden om snel een inzicht te krijgen in het systeem, waardoor aanpassingen gemakkelijker zullen zijn.
2. **Niet geschikt voor build proces.** Als de ontwikkelaar met de juiste diagrammen (bijvoorbeeld data flow diagram of state machine diagram) werkt, zal hij er over nadenken hoe de verschillende onderdelen van de tool gescheiden moeten worden. Immers, als de onderdelen niet gescheiden worden, bestaat het diagram maar uit zo weinig elementen dat het diagram geen toegevoegde waarde heeft en dat zal een goede ontwikkelaar niet toelaten. Hierdoor zal de ontwikkelaar er gemakkelijker zorg voor dragen dat het gedeelte van de user interface losgekoppeld is en de tool later gemakkelijk in het build proces opgenomen kan worden.

3. **Weinig tot geen documentatie.** Door model driven engineering toe te passen tijdens het ontwikkelen van een projectspecifieke tool, met een bepaald diagram als model is er continu documentatie beschikbaar. Omdat elke wijziging begint met een wijziging in het model en daarna in de source code, blijft het model, en dus de documentatie, up-to-date. Als de juiste en voldoende diagrammen uit deze plugin gebruikt worden is de documentatie volledig.
4. **Te weinig overdracht aan PDC.** Omdat het evolutionaire karakter geen knelpunt is en er voldoende documentatie beschikbaar is, is het gemakkelijk om een projectspecifieke tool over te dragen aan het PDC.
5. **Project wordt geremd door afhankelijkheid tool.** Omdat er documentatie is, is het gemakkelijker om de tool aan te passen wanneer de gebruiksomgeving verandert. Dit geldt natuurlijk alleen als de juiste en voldoende diagrammen ingezet zijn om de projectspecifieke tool ermee te documenteren.

3.2.4.5 Niet-afgedekte Knelpunten

6. **Geen simpele richtlijnen.** Het gebruik van deze plugin op zich biedt geen richtlijnen om projectspecifieke tools te ontwikkelen. De enige richtlijn die opgesteld zou kunnen worden is dat een bepaald diagram gebruikt dient te worden. Echter, welk diagram en tot welk detail een diagram gemodelleerd dient te worden zal aangegeven moeten worden. De diagrammen en de mogelijkheden zijn zeer uitgebreid, maar de mogelijkheden om hiervan code te genereren is juist weer zeer beperkt, waardoor er uitgebreide richtlijnen gemaakt zullen worden. Te uitgebreide richtlijnen lezen en eraan houden staat niet in verhouding tot de hoeveelheid ontwikkeltijd gemiddeld nodig is om een projectspecifieke tool te ontwikkelen.

3.2.4.6 Voor- en Nadelen

Smart Developer Environment biedt vele mogelijkheden om visueel te ontwikkelen en model driven engineering toe te passen. Het aantal diagrammen en mogelijkheden per diagram is zeer uitgebreid en bijna alle diagrammen kunnen aan elkaar of aan een element van een diagram gekoppeld worden.

De meeste diagrammen zijn echter vooral geschikt voor grote software systemen, waar projectspecifieke tools niet onder vallen. Het class diagram en state machine diagram zijn wel geschikt en vanuit deze twee diagrammen kan Java code gegenereerd worden. De plugin is gemakkelijk in gebruik, waardoor het niet veel tijd zal kosten om te leren hoe de plugin gebruikt kan worden.

Een nadeel van dit pakket is dat er geen code gegenereerd kan worden vanuit een data flow diagram, waardoor dit diagram alleen voor documentatiedoeleinden ingezet kan worden.

3.3 Conclusies

Geen van de bestaande oplossingen dekt alle knelpunten af. In de volgende tabel wordt een overzicht gegeven van de oplossingen en de afgedekte knelpunten.

	1. Niet geschikt voor evolutie	2. Niet geschikt voor build proces	3. Weinig tot geen documentatie	4. Te weinig overdracht aan PDC	5. Project wordt geremd door afhankelijkheid tool	6. Geen simpele richtlijnen
Model driven engineering						
Class diagram			•			
Data flow diagram	•	•				•
State machine diagram	•	•				•
Eclipse plugins						
Cheat sheets						•
UML2 Tools [UML2T]			•			
Eclipse Modeling Framework [EMF]			•			
Smart Developer Environment [SDE]	•	•	•	•	•	

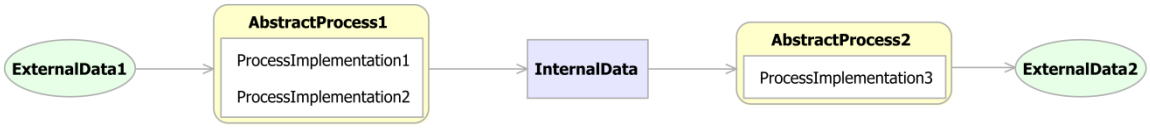
Tabel 2. Overzicht afgedekte knelpunten door bestaande oplossingen

De Smart Developer Environment dekt vijf van de zes knelpunten af. Omdat de mogelijkheden van dit pakket zo uitgebreid zijn dienen er uitgebreide en duidelijke richtlijnen opgesteld te worden om dit pakket goed in te kunnen zetten voor het ontwikkelen van projectspecifieke tools. Hiervoor zouden bijvoorbeeld cheat sheets gebruikt kunnen worden. Echter, een ontwikkelaar zal veel te snel geneigd zijn om toch meer functionaliteit van de diagrammen te gebruiken, die later bij het implementeren nutteloos blijken te zijn. Hier komt bij dat het aantal diagrammen waaruit Smart Developer Environment source code kan genereren zeer beperkt is en de gegenereerde code zeer summier is, waardoor de Smart Developer Environment vooral geschikt is voor het ontwikkelen van grote software projecten en niet zozeer voor projectspecifieke tools.

4 Tool Data Flow Diagram

Uit de inventarisatie is gebleken dat projectspecifieke tools waar een data flow doorheen loopt het vaakst voor komen. Het betreft hier vooral "data converters" en "code generators".

In deze thesis definiëren we het "tool data flow diagram". Hiermee kunnen data objecten, processen en data flows in kaart gebracht worden. Hierbij is ernaar gestreefd om een balans te vinden tussen enerzijds het bieden van een abstractie van de te ontwikkelen projectspecifieke tool, terwijl de objecten anderzijds één-op-één op object georiënteerde source code afgebeeld kunnen worden. De leercurve van het diagram is erg vlak, wat in sectie 5.4 aangetoond zal worden.



Figuur 11. Voorbeeld van een tool data flow diagram

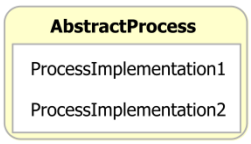
In sectie 4.1 worden de verschillende onderdelen van het diagram uitgelegd, waarna in sectie 4.2 uitgelegd wordt hoe de verschillende elementen decoupled zijn in verschillende lagen. De volgorde van executie is niet van het het diagram af te lezen. Waarom dit is wordt uitgelegd in sectie 4.3. Het diagram kan gebruikt worden als model bij model driven engineering. Hoe dit werkt wordt uitgelegd in sectie 4.4. Als laatste worden de voor- en nadelen van het tool data flow diagram en het gebruik ervan bij model driven engineering besproken in sectie 4.5.

4.1 Data Objecten, Processen en Data Flows

Figuur 11 laat een voorbeeld zien van een tool data flow diagram. Dit moet als volgt gelezen worden: AbstractProcess1 leest/gebruikt data uit een externe bron (ExternalData1) en schrijft vervolgens data in data object InternalData. AbstractProcess2 leest/gebruikt de eerder geschreven data uit InternalData en zal bepaalde data exporteren naar ExternalData2.

Data objecten zijn onderverdeeld in interne en externe data objecten. Externe data objecten "leven" buiten de projectspecifieke tool. Dit kan bijvoorbeeld een database zijn, een bestand op schijf, een bericht dat via een server/service binnen komt, een parameter van de command line, een bericht dat van/naar de console gelezen/geschreven wordt, etc. Een extern data object moet gezien worden als een "black box" en wordt vooral gemodelleerd voor documentatiedoeleinden.

Interne data objecten representeren een data model waarin informatie tijdelijk opgeslagen wordt. Dit kan een simpele buffer zijn om informatie van het ene naar het andere proces door te geven, maar het kan ook een uitgebreide datastructuur zijn. Interne data objecten zouden in een apart class diagram gemodelleerd kunnen worden.



Processen transformeren data. Zowel de bron als de bestemming moet één of meerdere data objecten zijn, waarbij het niet uit maakt of dit interne of externe data objecten zijn. Op functioneel niveau zijn er abstracte processen en op technisch niveau kunnen de abstracte processen op één of meerdere manieren geïmplementeerd worden. Deze implementaties worden in het diagram binnen abstracte processen getekend, zoals in het voorbeeld "ProcessImplementation1" en "ProcessImplementation2". Abstracte processen kunnen één-op-één afgebeeld worden naar abstracte classes en proces implementaties op classes die overerven van het abstracte proces.

Tussen de objecten worden data flows gedefinieerd. Een data flow van een proces naar een data object moet gelezen worden als "Proces X schrijft data naar data object Y" en een data flow van data object naar een proces moet gelezen als "Proces X haalt informatie uit data object Y". Hierbij zien we dus dat de data objecten statisch zijn en de processen een koppeling naar de verschillende data objecten hebben. De pijltjes moeten dus niet als "call" gezien worden, want in dat geval zouden de pijlen van data object naar proces de verkeerde kant op staan.

Data flows lopen altijd van een data object naar een proces of omgekeerd. Het is dus niet mogelijk om een data flow te definiëren tussen twee processen of tussen twee data objecten. De reden hiervoor is dat data objecten statisch zijn en zelf dus geen data naar een ander data object over kunnen hevelen. Hier is een proces voor nodig. Omdat de pijlen een data flow voorstellen en geen call zijn is het onlogisch om een data flow te definiëren tussen twee processen, want dan wordt de informatie die van het ene naar het andere proces gaat niet gemodelleerd. Hier moet dus altijd een data object tussen om aan te geven welke informatie van het ene naar het andere proces gaat.

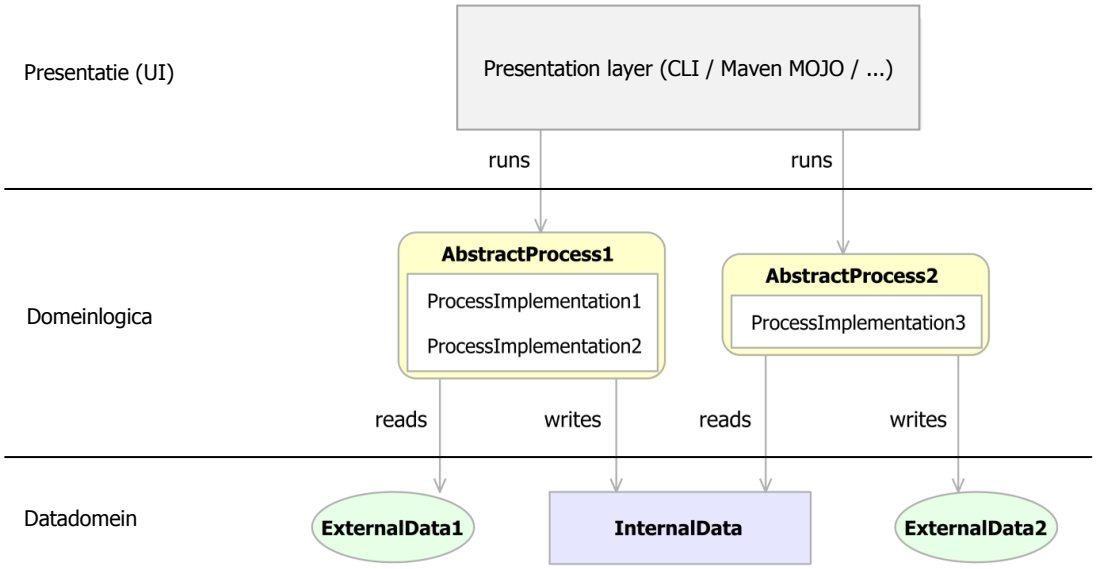
4.2 Decoupling

Door decoupling toe te passen kunnen we een software product opdelen in drie lagen: de presentatie laag, de domeinlogica laag en de datadomein laag. De datadomein laag bevat alle data-elementen van het software product. Dit kunnen datastructuren zijn, maar ook databases, bestanden of een koppeling naar een extern systeem. Kortom, alles waar data in opgeslagen en/of uit gelezen kan worden.

De elementen uit de domeinlogica laag kunnen bewerkingen doen op de data uit de datadomein laag. In deze laag bevinden zich de processen binnen het software product dus. De bovenste laag is de presentatie laag. Deze laag wordt ook wel de user interface genoemd. De presentatie laag roept functionaliteit uit de domeinlogica laag aan.

Een projectspecifieke tool die met een tool data flow diagram gemodelleerd wordt is automatisch decoupled. Figuur 12 laat dit zien. In deze figuur zien we alle elementen uit een tool data flow diagram, waarbij de pijlen nu geen data flows zijn, maar functie aanroepen. We zien hier dat de interne en externe data objecten zich in de datadomein laag bevinden en processen in de domeinlogica laag. Dit kunnen we

samen als de "backend" van de tool zien. Hier bovenop hebben we de "frontend". Dit is de presentatie laag die de processen executeert.



Figuur 12. Tool data flow diagram is decoupled

Het grote voordeel van decoupling zien we wanneer een nieuwe frontend nodig is. Zoals één van de geïdentificeerde knelpunten aangeeft bestaat er vaak de wens om een projectspecifieke tool die regelmatig ingezet wordt om te bouwen tot stap in het build proces, zodat de tool bij elke build uitgevoerd wordt. Om dit te bewerkstelligen zal een nieuwe frontend ontwikkeld moeten worden. Wanneer de source code van de frontend met de code van de backend verweven is, is dit erg lastig en vaak zelfs niet mogelijk. Door decoupling toe te passen is de frontend helemaal losgekoppeld en altijd direct te vervangen.

Een ander voordeel van decoupling is dat het mogelijk is om de processen geïsoleerd te testen. De functionaliteit van de verschillende processen staat geheel op zichzelf. Door de verbonden interne data objecten met pijl richting het proces te vullen met bepaalde en daarna het proces uit te voeren kan naderhand in de verbonden interne data objecten met pijlen richting de interne data objecten gecontroleerd worden of het proces juist is geïmplementeerd.

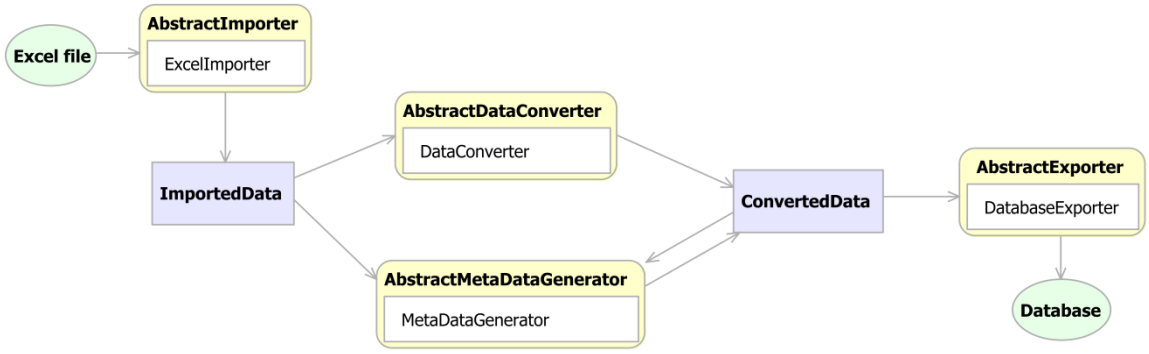
4.3 Executievolgorde van Processen

Processen zijn op zichzelf staande onderdelen van de applicatie. De frontend is er verantwoordelijk voor om de juiste processen op de juiste tijdstippen uit te voeren. Het lijkt of het voor de hand ligt welke processen wanneer uitgevoerd dienen te worden, aangezien de pijlen in het diagram gevolgd moeten worden. Dit is echter niet altijd waar.

Figuur 13 modelleert een projectspecifieke tool die informatie uit een Excel-bestand importeert, converteert naar een ander data model en daarna exporteert naar een database. Optioneel kan er na de conversieslag metadata gegenereerd worden. Voor dit genereren van metadata heeft het generator-proces informatie uit zowel

"ImportedData" als "ConvertedData" nodig. De gegenereerde metadata zal dan worden toevoegd aan "ConvertedData".

Zoals eerder gezegd is het aan de frontend welke processen wanneer uitgevoerd worden. Zo zou deze tool over een grafische user interface kunnen beschikken met drie knoppen: "Importeer", "Genereer metadata" en "Exporteer". Wanneer er op de knop "Importeer" geklikt wordt zal een venster met "Selecteer Excel-bestand" verschijnen en na het selecteren van een bestand zullen achtereenvolgens de procesimplementaties "ExcelImporter" en "DataConverter" uitgevoerd worden. Na het klikken op de knop "Genereer metadata" zal procesimplementatie "MetaDataGenerator" uitgevoerd worden en na het klikken op "Exporteer" wordt procesimplementatie "DatabaseExporter" uitgevoerd.



Figuur 13. Tool Data flow diagram waarbij de volgorde waarop de processen uitgevoerd dienen te worden niet per definitie hetzelfde is als de informatiestroom

Nu ligt het eraan of alle drie de knoppen te allen tijden enabled zijn, of dat alleen de eerste knop enabled is en de tweede pas enabled wordt op het moment dat de processen achter de eerste knop succesvol uitgevoerd is. Als alle knoppen altijd enabled zijn kan er dus geëxporteerd worden, ook al is er niets geïmporteerd. Het exporteer-proces zal dan dus wel uitgevoerd worden, maar niets exporteren.

Het tool data flow diagram voorziet er dus niet in om de volgorde dat processen uitgevoerd dienen te worden te modelleren. Hier zou eventueel een sequence diagram of een state machine diagram voor ingezet kunnen worden, maar de vraag is of dit nuttig is voor projectspecifieke tools, aangezien het modelleren dan waarschijnlijk meer tijd in zal nemen dan het handmatig uitprogrammeren in de frontend. Dit vraagstuk, of het nuttig is om dit te modelleren en zoja, welke diagrammen hiervoor ingezet zouden kunnen worden valt buiten de scope van het onderzoek.

4.4 Model Driven Engineering met Tool Data Flow Diagram

Het tool data flow diagram is ontworpen om gebruikt te kunnen worden als model wanneer model driven engineering wordt toegepast. Door middel van het diagram wordt de projectspecifieke tool gemodelleerd. Daarna kan vanuit het diagram Java code gegenereerd worden, die een aantal "gaten" bevat, welke door de ontwikkelaar ingevuld dienen te worden.

Het proof of concept, wat in hoofdstuk 5 wordt uitgelegd, maakt gebruik van model driven engineering, waarbij een tool data flow diagram als model gebruikt wordt. In dat hoofdstuk zijn aanvullende voorbeelden te vinden.

Met het invoeren van model driven engineering bieden we richtlijnen over hoe projectspecifieke tools ontwikkeld dienen te worden. Deze richtlijnen zorgen ervoor dat altijd er in termen van informatie stromingen, processen en data objecten gedacht wordt. We zullen zien dat eventuele latere aanpassingen hierdoor versimpeld worden. Verder zorgt het principe van model driven engineering ervoor dat er altijd een model van de ontwikkelde projectspecifieke tool beschikbaar is. Eén van de geïdentificeerde knelpunten is dat er geen simpele richtlijnen zijn voor het ontwikkelen van projectspecifieke tools. Het invoeren van model driven engineering in combinatie met het tool data flow biedt richtlijnen aan.

4.4.1 Modelleren

Wanneer begonnen wordt met het ontwikkelen van een projectspecifieke tool, dient eerst een tool data flow diagram gemodelleerd te worden. Hiervoor is het het handigste om eerst de externe data objecten te modelleren. Als we bijvoorbeeld een data converter die data uit verschillende ASCII-bestanden naar een Excel-bestand converteert willen ontwikkelen, starten we met twee externe data objecten: "ASCII-bestanden" en "Excel-bestand". Natuurlijk is het ook mogelijk dat we meer dan twee externe data objecten modelleren, of juist maar één.

Nu kunnen de abstracte processen en de interne data objecten gemodelleerd worden. Het is niet de bedoeling dat er maar één proces in de projectspecifieke tool aanwezig is, want dan wordt de tool als een soort van "black box" gemodelleerd en vervallen alle positieve eigenschappen van model driven engineering. Er zal dus geïdentificeerd moeten worden welke "stappen" of "data transformaties" de tool bevat.

De abstracte processen dienen generieke namen te hebben, zodat de naam nog steeds logisch is als er meerdere implementaties van zijn. Zo zouden we bijvoorbeeld een abstract proces "AbstractImporter" kunnen definiëren. Later kunnen we hier gemakkelijk meerdere implementaties voor modelleren, zoals bijvoorbeeld "ACSIIImporter", "ExcelFileImporter" en "CSVImporter". Hierdoor zal de tool later ook gemakkelijker aan te passen zijn, bijvoorbeeld door een nieuwe implementatie aan een abstract proces toe te voegen.

Na het modelleren van de abstracte processen en interne data objecten kunnen de procesimplementaties toegevoegd worden, hoewel deze ook ook direct na het modelleren van elk abstract proces toegevoegd kunnen worden. Tussendoor dienen de abstracte processen en de interne en externe data objecten met elkaar verbonden te worden door data flows te definiëren. Elk abstract proces en intern data object heeft als het goed is minimaal één ingaande data flow en minimaal één uitgaande data flow. Externe data objecten zijn met minimaal één data flow verbonden met een abstract proces.

Als het tool data flow model naar wens is, zijn de interne data objecten aan de beurt. Voor elk intern data object dient een class diagram gemodelleerd te worden om te definieren hoe de data in deze interne data objecten eruit ziet. In dit class diagram dient één class de "top level class" te zijn. Een instantie hiervan kan dan beschikbaar zijn in de verbonden abstracte processen in het tool data flow diagram. De andere classes dienen daarom dan ook (direct of indirect) verbonden te zijn met deze top level class.

4.4.2 Afbeelding op Source Code

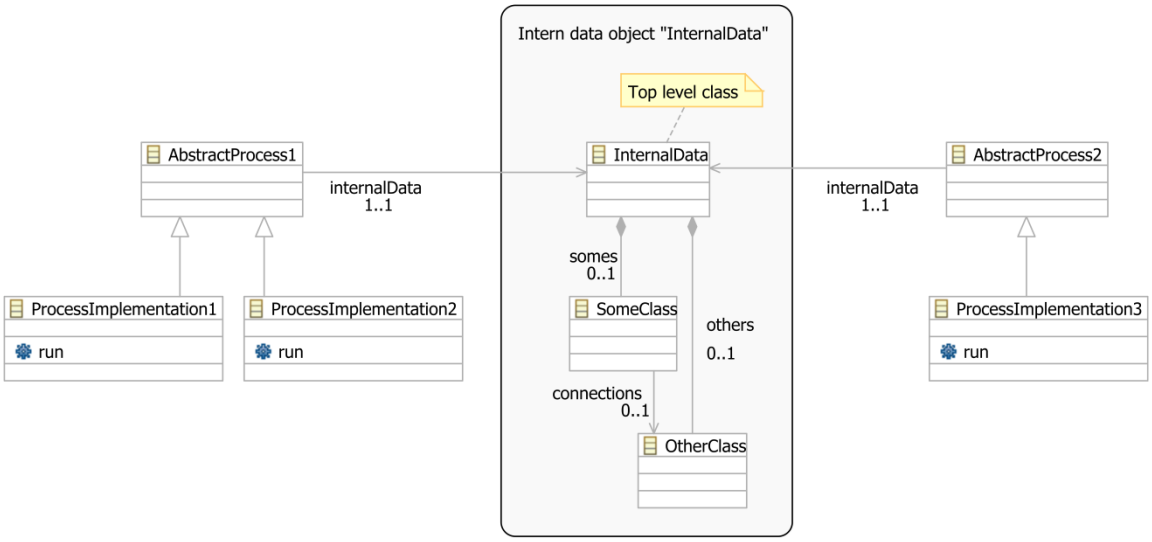
Het tool data flow diagram is zo ontworpen dat deze processen, data objecten en data flows op een abstracte manier gemodelleerd kunnen worden, maar dat het model toch niet ver van object georiënteerde code af ligt.

Externe data objecten zijn puur voor documentatie-doeleinden. Zij hoeven dus niet op source code afgebeeld te worden. Het is wel handig dat ze vernoemd worden in het commentaar bij de code voor de verbonden abstracte processen.

Interne data objecten zijn van binnen al een class diagram, die rechtstreeks op source code afgebeeld kan worden. Class diagrams staan zo dicht bij de code dat ze eigenlijk alleen een visualisatie van de code zijn. Behalve de inhoud van de functies en het commentaar in de code is alles in dit diagram terug te vinden.

Abstracte processen kunnen rechtstreeks afgebeeld worden op een abstract class. De bijbehorende procesimplementaties kunnen ieder afgebeeld worden op een class die overerft van de abstract class. Abstracte processen hebben een associatie met de top level classses van alle verbonden interne data objecten.

De afbeelding van een tool data flow diagram op source code is te demonstreren aan de hand van een afbeelding van een tool data flow diagram op een class diagram. Figuur 11 op bladzijde 45 laat een simpel tool data flow diagram zien, waarbij alle mogelijke typen elementen gebruikt worden. Figuur 14 is een class diagram, dat afgeleid is van het tool data flow diagram.



Figuur 14. Afgeleid class diagram

De afbeelding van model naar source code is simpel en kan geautomatiseerd worden door een code generator. Hierdoor wordt de structuur van het project gegenereerd en hoeft alleen de daadwerkelijke technische implementatie nog ontwikkeld te worden.

4.4.3 Implementeren

De procesimplementaties dienen geïmplementeerd te worden. Dit gebeurt door het implementeren van de "run" functie. Omdat de procesimplementaties overerven van de abstracte processen, is de associatie naar de top level class van de verbonden interne data objecten automatisch beschikbaar.

Verbindingen met externe data objecten moet handmatig ontwikkeld worden. Dit soort data objecten kunnen bijvoorbeeld een database of bestanden op de harde schijf zijn. De verbonden procesimplementatie zal ervoor moeten zorgen dat de verbinding gemaakt wordt of het bestand ingelezen/weggeschreven wordt. Dit is niet te automatiseren.

Object georiënteerde programmeertalen zoals Java en C# kennen zogenaamde "packages". Dit is vergelijkbaar met directories in een bestandssysteem en hiermee kunnen classes in een package bij elkaar gehouden worden. Elk abstract proces dient samen met zijn implementaties in een aparte package ondergebracht te worden. Een procesimplementatie hoeft namelijk niet per definitie in één class geïmplementeerd te worden. De "run" functie dient namelijk gezien te worden als de "main" van een normale applicatie. Door alle andere classes die bij deze processen horen in dezelfde package onder te brengen houden we de source bestanden netjes geordend.

4.4.4 Backend en Frontend

Het tool data flow diagram kan de backend van een projectspecifieke tool modelleren. De frontend dient hierna nog geïmplementeerd te worden. Dit kan gemakkelijk door op de juiste momenten, of door bepaalde acties van de gebruiker, de "run" functie van de juiste procesimplementaties uit te voeren.

De backend en frontend is dus helemaal gescheiden, waardoor de frontend gemakkelijk te vervangen is. Dit is bijvoorbeeld handig wanneer de projectspecifieke tool later aangepast moet worden, zodat het als stap in het build proces opgenomen kan worden. Dit was één van de geïdentificeerde knelpunten.

4.4.5 Aanpassingen in het Model

Bij model driven engineering gaat het altijd om een roundtrip. Eerst wordt een model gemaakt van de te ontwikkelen software. Dit model wordt hierna geïmplementeerd. Wanneer er hierna iets aangepast moet worden, dient eerst teruggegaan naar het model, om hier de wijzigingen door te voeren. Hierna moet ervoor gezorgd worden dat de implementatie bijgewerkt aan de hand van het nieuwe model.

Het aanpassen van het model kan op verschillende manieren. Op de eerste plaats kunnen we een element toevoegen aan het diagram. Dit zal ervoor zorgen dat er meer classes nodig zijn. Deze kunnen van het diagram afgeleid worden.

Wanneer een naam in het diagram wijzigt, wijzigt de naam van de class mee. Dit heeft als gevolg dat alle code, die gebruik maakt van die class-naam aangepast dient te worden. Als het een abstract proces betreft, zal de naam waarvan overgeërfd wordt mee aangepast moeten worden. Als het een procesimplementatie betreft dienen de aanroepen in de frontend aangepast te worden. Naamsveranderingen van classes binnen een intern data object kunnen gevolgen hebben voor de implementaties van de "run" functie en alles wat in deze functie aangeroepen wordt.

Als een element van het diagram verwijdert wordt, dienen de bijbehorende classes ook verwijdert te worden. Dit kan gevolgen hebben voor alle code. Eventueel kan bij een implementatie van model driven engineering en het tool data flow diagram ervoor gekozen worden dat de classes bewaard blijven als een element van het diagram verwijderd wordt. De ontwikkelaar zal dan zelf de bijbehorende classes kunnen verwijderen. Dit is minder gevaarlijk.

De knelpunten "Niet geschikt voor evolutie" en "project is afhankelijk van de tool" zouden hiermee opgelost moeten worden. Wanneer er iets aangepast moet worden in de tool, bijvoorbeeld omdat meer functionaliteit gewenst is of omdat de gebruiksomgeving verandert, kan dit gemakkelijk in het model aangepast worden. Vanuit het model is het geen probleem om de aanpassingen in de code door te voeren.

4.5 Voor- en Nadelen

Het toepassen van model driven engineering in combinatie met het tool data flow diagram biedt veel voordelen. Zo kunnen informatiestromen binnen een projectspecifieke tool op een abstracte manier gemodelleerd worden, terwijl het implementeren aan de hand van het model gemakkelijk is, omdat de afbeelding van model op source code simpel is.

Het tool data flow diagram kan alleen ingezet worden voor projectspecifieke tools die iets met data doen, zoals bijvoorbeeld "code generators" of "data converters". Dit zijn echter wel de meest ontwikkelde projectspecifieke tools. Andere tools, zoals bijvoorbeeld systeemadministratieve tools kunnen niet gemodelleerd worden. Deze tools zijn meestal niet meer dan een script, waardoor het bieden van een abstractielaag geen voordeel zou hebben. De ontwikkeltijd zou hier alleen meer mee toenemen in plaats van afnemen.

De volgorde dat de processen uitgevoerd dienen te worden kan niet gemodelleerd worden. Het is de vraag of het handig is dat om dit te modelleren of dat dit meer werk met zich meebrengt dan wanneer de frontend zonder enig model ontwikkeld wordt. Een nieuw onderzoek zou antwoord op deze vraag kunnen geven. Eventueel zou een sequence diagram of een state machine diagram ingezet kunnen worden om de frontend te modelleren.

5 Proof of Concept

Hoofdstuk 4 bespreekt het tool data flow diagram en hoe dit in te zetten is als model bij model driven engineering. Verder wordt er gesteld dat dit alle in hoofdstuk 2 geïdentificeerde knelpunten op moet lossen. Om dit aan te tonen is een proof of concept ontwikkeld, wat in dit hoofdstuk besproken wordt.

Het tool data flow diagram moet een willekeurige projectspecifieke tool waarin een data flow te definiëren is kunnen modelleren. Dit geldt in het bijzonder voor tools uit de categorieën "code generators" en "converters". Als proof of concept hebben we een projectspecifieke tool ontwikkeld, waarin alle knelpunten naar voren komen. Op de eerste plaats laten we zien dat alle knelpunten opgelost worden. Verder laten we zien dat het gemakkelijk is om deze projectspecifieke tool te modelleren door middel van een tool data flow diagram en dat het mogelijk is om de Eclipse omgeving uit te breiden zodat dit diagram gemodelleerd en gebruikt kan worden en model driven engineering wordt toegevoegd.

Het uitbreiden van de Eclipse ontwikkelomgeving gebeurt door middel van plugins. Daarom hebben we een Eclipse plugin geschreven die de ontwikkelomgeving uitbreidt met een grafische editor voor tool data flow diagrammen. Verder zorgt deze plugin ervoor dat er vanuit dit diagram Java code gegenereerd wordt. Met de plugin kunnen we laten zien dat het mogelijk is om een tool data flow diagram binnen de Eclipse ontwikkelomgeving te modelleren en model driven engineering hiermee toe te passen.

Hiermee is nog niet aangetoond dat het tool data flow diagram ook daadwerkelijk toepasbaar is op een willekeurige projectspecifieke tool. Daarom wordt de ontwikkelde plugin gebruikt om een projectspecifieke tool, de "stored procedure generator" genaamd, te ontwikkelen. Er is voor gezorgd dat alle knelpunten voor komen, zodat er nagegaan kan worden of deze opgelost worden door het gebruik van de Eclipse plugin.

In de eerste fase van het onderzoek zijn medewerkers van Info Support geïnterviewd over welke projectspecifieke tools ze zoal ontwikkeld en gebruikt hebben. Eén van deze tools is de stored procedure generator. De originele tool is ontwikkeld door één medewerker en is geschreven in VBScript. Aangezien deze medewerker niet meer werkzaam is en er geen ontwerp of documentatie beschikbaar is, is het erg lastig om aanpassingen aan de tool te doen. Het project is sterk afhankelijk van de tool, waardoor er af en toe tools om de tool heen geschreven worden, zodat de stored procedure generator maar niet aangepast hoeft te worden. Aangezien alle gedefinieerde knelpunten te vinden zijn, wordt deze tool opnieuw ontwikkeld, maar deze keer door gebruik te maken van model driven engineering in combinatie met het tool data flow diagram. Deze nieuwe versie zal niet alle functionaliteit van de originele tool bevatten, omdat dan teveel ingegaan moet worden op implementatiedetails van de tool zelf, welke niet relevant zijn voor het proof of concept.

Sectie 5.1 beschrijft de Eclipse plugin die model driven engineering in combinatie met het tool data flow diagram toevoegt aan de Eclipse ontwikkelomgeving. In sectie 5.2 wordt het ontwikkelproces van de stored procedure generator, waarbij de plugin gebruikt wordt, beschreven. Sectie 5.3 laat zien dat in het alle knelpunten in het proof of concept afgedekt worden. Behalve zelf met de plugin te werken is een medewerker van Info Support gevraagd om er eens mee te werken. Zijn bevinden zijn opgenomen in sectie 5.4. Tenslotte worden de conclusies van het proof of concept beschreven in sectie 5.5.

5.1 Tool Data Flow Eclipse Plugin

De Eclipse ontwikkelomgeving beschikt standaard alleen over op tekst gebaseerde editors, zoals bijvoorbeeld een tekst- en Java-editor. Er zijn verschillende plugins beschikbaar die grafische editors voor bepaalde diagrammen toevoegen aan de ontwikkelomgeving.

Het ontwikkelen van een grafische editor kan op verschillende manieren. Zo is het mogelijk om het Eclipse framework direct te gebruiken en alleen de API daarvan te gebruiken om de editor te ontwikkelen. Aangezien dit niet handig is, is het Graphical Editing Framework [GEF] ontwikkeld. Dit framework, wat ontwikkeld is door Eclipse Foundation, biedt ondersteuning voor het ontwikkelen van grafische editors. Ook nu moet de editor met de hand ontwikkeld worden, maar omdat het framework een brede ondersteuning biedt, wordt het ontwikkelproces vergemakkelijkt.

Weer een andere mogelijkheid is om gebruik te maken van het Graphical Modeling Framework [GMF]. Dit framework kan de Java code voor een grafische editor genereren uit een aantal modellen. Bij deze modellen moet gedacht worden aan modellen voor de achterliggende datastructuur, de grafische elementen van de editor, het palette waarmee nieuwe grafische elementen aan het diagram toegevoegd kunnen worden, etc. De gegenereerde code maakt op zijn beurt weer gebruik van de API uit het Graphical Editing Framework. Het gebruik van GMF kan het ontwikkelen van een grafische editor aanzienlijk vergemakkelijken.

Model driven engineering houdt in dat er begonnen wordt met een model. Hierna wordt een implementatie gemaakt die aan het model voldoet. Dit kan handmatig, maar het is natuurlijk gemakkelijker wanneer (een deel van) de implementatie gegenereerd kan worden. Om dit te bewerkstelligen is een code generator ontwikkeld, die Java code genereert aan de hand van het gemodelleerde tool data flow diagram.

Het technisch ontwerp van de tool data flow Eclipse plugin is te vinden in bijlage B.

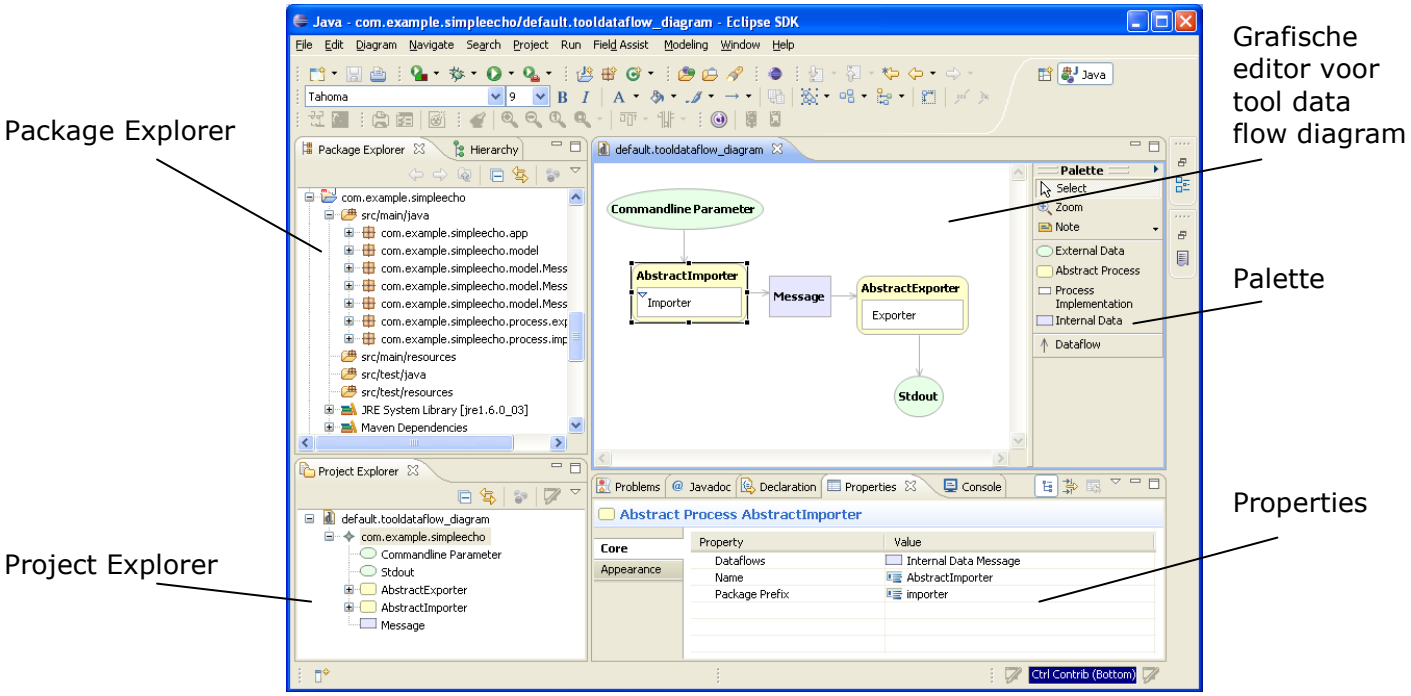
5.1.1 Grafische Editor

Bij het ontwikkelen van de grafische editor voor tool data flow diagrammen is gebruik gemaakt van het Graphical Modeling Framework [GMF]. Hiermee kan de grafische editor door middel van model driven engineering worden ontwikkeld. Eén van de belangrijkste modellen is de achterliggende datastructuur. Deze wordt gemodelleerd met het Eclipse Modeling Framework [EMF]. In principe houdt dit in dat een class diagram gemaakt wordt van de achterliggende datastructuur.

We hebben de grafische editor ontwikkeld door een model van de achterliggende datastructuur te ontwerpen. In dit model zitten classes voor de verschillende elementen van het diagram: InternalData, ExternalData, AbstractProcess, ProcessImplementation, wat gegeneraliseerde classes, etc. Verder hebben we modellen ontwikkeld over hoe elk element er grafisch uit dient te zien (rondje, vierkantje, achtergrondkleur, etc) en welke elementen kunnen worden toegevoegd door middel van een palette. Als laatste is een model ontwikkeld over de samenhang van alle eerdergenoemde modellen. GMF kan vanuit deze modellen, die speciaal voor GMF ontwikkeld zijn, een implementatie van de grafische editor genereren.

De door ons gemodelleerde achterliggende datastructuur kan door de gegenereerde code worden geserialiseerd naar een *.tooldataflow bestand. De grafische editor is niet meer dan een grafische “view” op deze datastructuur. In principe kunnen er meerdere views op één datastructuur gedefinieerd worden, maar dit zal in de praktijk weinig voor komen. Views worden opgeslagen in een *.tooldataflow_diagram bestand

en bevatten een verwijzing naar het datastructuur-bestand, zodat geen informatie dubbel opgeslagen wordt.



Figuur 15. Eclipse ontwikkelomgeving met tool data flow plugin

Figuur 15 laat een screenshot van de Eclipse ontwikkelomgeving met de ontwikkelde grafische editor voor tool data flow diagrammen zien. Hier zien we het palette waarmee nieuwe objecten aan het diagram toegevoegd kunnen worden. Aan de onderkant zien we de "Properties" view. Het palette is gegenereerd vanuit de GMF-modellen. De properties view is een standaard-view van Eclipse die de eigenschappen van een geselecteerd item, waar dan ook in de ontwikkelomgeging, toont. Deze eigenschappen kunnen hier, indien toegestaan, ook bewerkt worden. De getoonde properties zijn de attributen van de classes uit de door ons ontworpen achterliggende datastructuur van het model.

Links onderin hebben we de "Project Explorer" view geopend. Deze view is standaard in Eclipse aanwezig en is ongeveer hetzelfde als de standaard geopende "Package Explorer", die ook standaard in Eclipse aanwezig is. Het verschil tussen deze twee views zit in het gedrag van editors die geopend worden wanneer er gedubbelklikt wordt op een bestand. De "Package Explorer" is op Java gericht, terwijl de "Project Explorer" algemener is en eerder een normale tekst-editor zal openen in plaats van een Java-editor.

Een ander verschil tussen de "Project Explorer" en de "Package Explorer" is dat bestanden in de "Package Explorer" nooit uitgeklapt kunnen worden en bestanden in de "Project Explorer", afhankelijk van het bestand, wel. Voor tool flow diagrammen hebben we gezorgd dat deze bestanden uitgeklapt kunnen worden, om zo alle elementen van het diagram aan deze boomstructuur toe te voegen. Dit hebben we bewerkstelligd door dit in het juiste GMF-model te modelleren. Dubbelklikken op een item hier doet hetzelfde als dubbelklikken op een object in het diagram. De "Project Explorer" wordt pas bijgewerkt op het moment dat het diagram opgeslagen wordt, waardoor deze niet altijd in sync met het diagram is.

5.1.2 Code Generator

Vanuit de verschillende elementen van het diagram dient code gegenereerd te worden. Hiervoor hebben we een code generator ontwikkeld, waarbij we gebruik hebben gemaakt van zogenaamde Java Emitter Templates (JET). JET is een onderdeel van het Eclipse Modeling Framework [EMF], wat ontwikkeld is door de Eclipse Foundation. Met JET is het gemakkelijk om een code generator te ontwikkelen. Een groot voordeel van het gebruik van JET, is dat de code generator naar annotaties kan kijken om zo bepaalde onderdelen niet te overschrijven.

De code generator genereert voor elk element van het diagram (externe data objecten uitgezonderd) een Java class, die ieder in een apart bestand worden opgeslagen. Omdat dit een één-op-één afbeelding is en er aangegeven kan worden dat bepaalde stukken code niet overschreven mogen worden, zal het opnieuw genereren van eerder gegenereerde en daarna aangepaste code niet snel een probleem opleveren.

Immers, als een element aan het diagram wordt toegevoegd zal dit een nieuwe nog niet eerder bestaande class genereren. Een naam van een element wijzigen zal de naam van de class wijzigen. Als een element verwijderd wordt, zal de class (en dus het bestand van de class) blijven bestaan.

Het kan zijn dat code die door de ontwikkelaar zelf is toegevoegd aangepast moet worden nadat het diagram aangepast is. Zo zullen we later zien dat data flows beschikbaar komen als variabelen binnen processen. Als de ontwikkelaar een data flow verwijdert, verdwijnt deze variabele. Als in de code van het proces gebruik wordt gemaakt van deze variabele, zal deze code aangepast moeten worden. De Eclipse ontwikkelomgeving zal zien dat hier een fout is en een probleem in de “Problems” view tonen. Deze problems view is een standaard view van Eclipse en is te vinden in een tabblad aan de onderkant van het scherm, waar ook de properties view te vinden is.

De ontwikkelaar zal natuurlijk altijd na moeten denken wat een verandering in het diagram betekent voor de achterliggende code, om de impact van een wijziging in te kunnen schatten. Aangezien de elementen van het diagram allemaal behoorlijk op zichzelf staan, zullen wijzigingen in/van het ene element niet snel gevolgen hebben voor andere elementen.

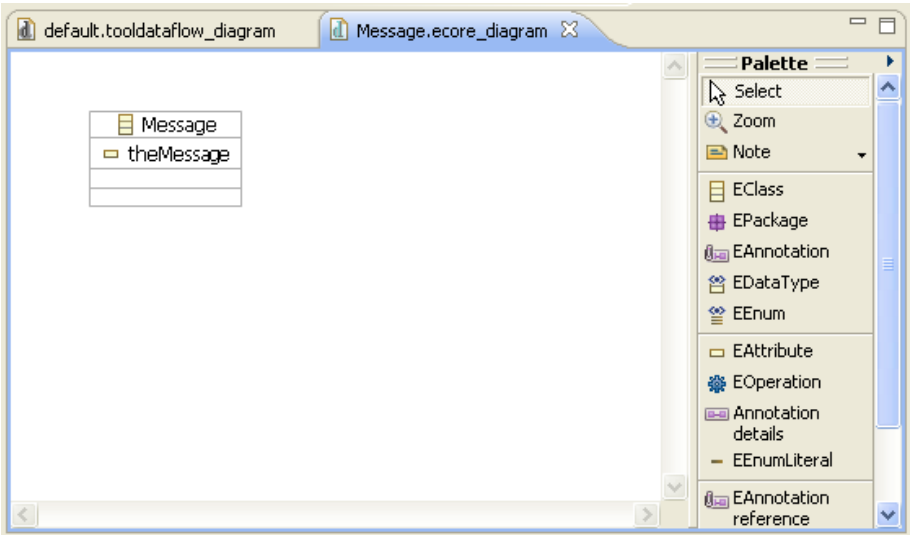
Behalve een class voor elk element van het diagram worden ook een aantal aanvullende bestanden gegenereerd door de code generator. Deze worden in de volgende secties besproken.

5.1.3 Externe Data Objecten

De externe data objecten zijn puur voor documentatiedoeleinden. Wanneer Java code gegenereerd gaat worden uit het diagram zullen de externe objecten terug komen in de code van de abstracte processen en van de procesimplementaties, en wel als Javadoc commentaar bij deze gegenereerde classes. Dit gegenereerde commentaar wordt besproken in secties 5.1.5 en 5.1.6.

5.1.4 Interne Data Objecten

Interne data objecten zijn in feite een datastructuur. Deze datastructuur is te modelleren als class diagram. Hiervoor wordt het Eclipse Modeling Framework [EMF] gebruikt. Een class diagram van EMF wordt een Ecore diagram genoemd. Hiervoor zijn verschillende grafische editors beschikbaar. Wij gebruiken de Ecore editor die automatisch bij het Graphical Modeling Framework [GMF] geïnstalleerd wordt.



Figuur 16. Dubbelklikken op een intern data object opent een Ecore editor

Door te dubbelklikken op een intern data object zal een grafische editor voor het Ecore diagram geopend worden. Figuur 16 laat hier een screenshot van zien. Elke datastructuur heeft een zogenaamde "top level class". Dit is een class waarvan per applicatie één instantie gecreëerd wordt. Dit object is beschikbaar voor alle processen die via een data flow verbonden zijn met het intern data object. Andere classes zullen doorgaans, direct of indirect, een relatie hebben met deze top level class.

Vanuit het tool data flow diagram zal Java code gegenereerd worden. Vanuit de Ecore diagrammen zal dit genereren aan EMF overgelaten worden. Deze gegenereerde code is zeer volledig. In het geval van het diagram uit de screenshot zal er dus een class "Message" gegenereerd worden. In deze class is het attribuut "theMessage" te vinden. Hiervoor zullen getters en setters gegenereerd worden. Voor eventuele relaties worden lijsten gegenereerd. Verder wordt een zogenaamde "factory" gegenereerd, waarmee instanties van de verschillende classes gecreëerd kunnen worden. Codevoorbeeld 7 in sectie 5.2.2.4 demonstreert het gebruik van de gegenereerde code.

Behalve bovenstaande worden vanuit het model ook zogenaamde content providers gegenereerd. Wanneer de projectspecifieke tool over een grafische user interface beschikt, waarbij JFace als widget toolkit gebruikt wordt, kan een content provider ingezet worden om bepaalde widgets van content te voorzien. Hierdoor wordt de koppeling tussen user interface en model versimpeld.

Zoals gezegd wordt per tool één instantie van de top level class gecreëerd, welke beschikbaar is voor de processen waarop een data flow met het interne data object gedefinieerd is. Deze instantie van de top level class wordt in de interface ModelsSingleton vastgehouden. Deze interface wordt gegenereerd uit het tool data flow diagram en ziet er als volgt uit:

```java
package com.example.simpleecho.model;

import com.example.simpleecho.model.Message.MessageFactory;
import com.example.simpleecho.model.Message.Message;

/**
 * Interface containing singletons of top level elements of all Internal Data objects
 *
 * @generated
 */
public interface ModelsSingleton {

    /**
     * Singleton of top level element of Internal Data object 'Message'
     *
     * @generated
     */
    static final Message message = MessageFactory.eINSTANCE.createMessage();
}
```

Codevoorbeeld 1. Gegenereerde singleton om referenties naar interne data objecten vast te houden

5.1.5 Abstracte Processen

Abstracte processen implementeren de gedeelde functionaliteit uit de onderliggende procesimplementaties. Door op een abstract proces te dubbelklikken zal de implementatie in een Java-editor geopend worden. De gegenereerde code achter een abstract proces ziet er als volgt uit:

```java
package com.example.simpleecho.process.importer;

import com.example.simpleecho.model.ModelsSingleton;
import com.example.simpleecho.model.Message.Message;

/**
 * Abstract Process containing singletons of top level elements of all
 * Internal Data objects
 *
 * Inputs:
 * - External Data object 'Commandline Parameter'
 * Outputs:
 * - Internal Data object 'Message', accessible via variable 'message'
 *
 * @generated
 */
public abstract class AbstractImporter {

    /**
     * Output: Singleton of top level element of Internal Data object 'Message'
     *
     * @generated
     */
    protected final Message message = ModelsSingleton.message;
}
```

Codevoorbeeld 2. Gegenereerde code voor AbstractImporter

Zoals te zien is in de code worden de relaties naar interne data objecten automatisch gegenereerd in de vorm van variabelen. Dit geldt voor zowel inkomende als uitgaande data flows. Verder wordt er Javadoc commentaar voor alle classes en variabelen gegenereerd. Hier zien we de externe data objecten ook in terug komen.

5.1.6 Procesimplementaties

Procesimplementaties implementeren een abstract proces. Ook voor dit object van het diagram geldt dat een Java-editor geopend wordt zodra erop gedubbelklikt wordt. De gegenereerde code achter een procesimplementatie ziet er als volgt uit:

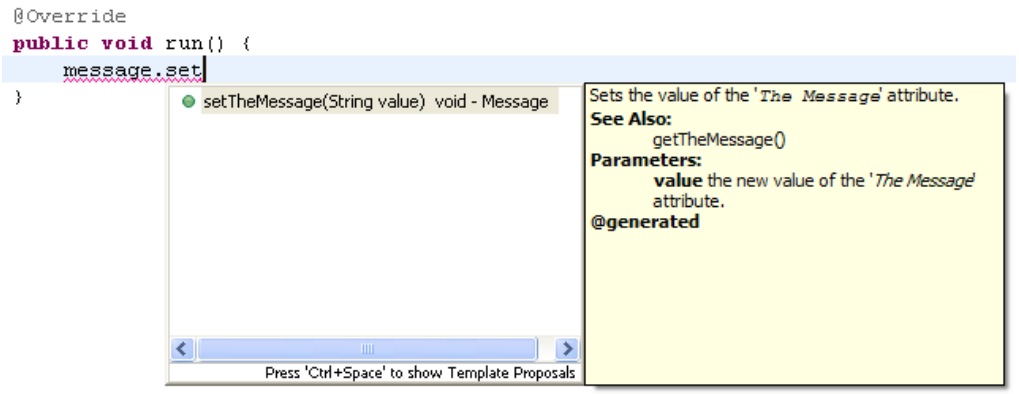
```
package com.example.simpleecho.process.importer;

/**
 * A Process Implementation of AbstractImporter
 *
 * Inputs:
 * - External Data object 'Commandline Parameter'
 * Outputs:
 * - Internal Data object 'Message', accessible via variable 'message'
 *
 * @generated
 */
public class Importer extends AbstractImporter implements Runnable {

    /**
     * Thread runner
     *
     * @generated NOT
     */
    @Override
    public void run() {
        // TODO: Implement your process here.
        // Internal Data objects are already available via variables,
        // see the class javadoc comment for more information.
    }
}
```

Codevoorbeeld 3. Gegenereerde code voor Importer

Omdat procesimplementaties overerven van abstracte processen zijn de protected variabelen van het abstracte proces ook hier beschikbaar. Omdat de code van interne data objecten gegenereerd is, zijn alle getters en setters direct beschikbaar. Eclipse helpt de ontwikkelaar hier door middel van code completion. Figuur 17 laat dit zien.



Figuur 17. Code completion voor verbonden interne data objecten

Ook in de gegenereerde class van een procesimplementatie zien we weer dat Javadoc commentaar gegenereerd wordt, zodat de ontwikkelaar direct kan zien welke data flows er op het proces gedefinieerd zijn. Zowel inkomende als uitgaande en zowel interne als externe verbonden data objecten worden hier genoemd.

Procesimplementaties implementeren de “Runnable” interface. Hierdoor kan een proces gemakkelijk als thread uitgevoerd worden. Runnables worden in Java vaak gebruikt en door veel frameworks en libraries ondersteund. Hierdoor is de gegenereerde code generiek te gebruiken, ook wanneer de ontwikkelaar een aanvullend framework wenst te gebruiken.

5.1.7 Applicatie

Door de processen te implementeren hebben we allerhande losstaande functionaliteiten gecreëerd. Deze zouden we de backend van de projectspecifieke tool kunnen noemen. Natuurlijk hebben we ook een frontend nodig. Zo'n frontend kan een (grafische) user interface zijn, maar ook een command line interface of een stap in een geautomatiseerd build-proces.

De frontend is verantwoordelijk voor het starten van de processen uit de backend. Zoals in sectie 4.3 te lezen is, is het niet mogelijk om de volgorde dat de processen uitgevoerd moeten worden af te lezen van het tool data flow diagram. Dit diagram modelleert namelijk de data flow en niet de executie volgorde. De aanroepen naar de processen kunnen wel gegenereerd worden, maar de ontwikkelaar zal deze dan nog op de goede volgorde moeten zetten. De ontwikkelaar kan er ook voor kiezen om bepaalde processen parallel uit te voeren.

Omdat de processen de "Runnable" interface implementeren beschikken ze allemaal over de "run" functie. Deze kan aangeroepen worden om het proces uit te voeren. Omdat processen normale classes zijn, is het mogelijk om functies toe te voegen aan de class en kan het proces "ingesteld" worden alvorens de "run" functie aangeroepen wordt. Dit aanroepen zal later gedemonstreerd worden. Als voorbeeld breiden we de procesimplementatie "Importer" uit met het private attribuut "messageParam" en public getters en setters voor dit attribuut:

```
public class Importer extends AbstractImporter implements Runnable {

     private String messageParam;

     public String getMessageParam() {
          return messageParam;
     }

     public void setMessageParam(String messageParam) {
          this.messageParam = messageParam;
     }

    @Override
    public void run() {
         message.setTheMessage(messageParam);
    }
}
```

Codevoorbeeld 4. Parameter toegevoegd aan procesimplementatie Importer

De code zonder gele achtergrond is gegenereerde code. De code met gele achtergrond zijn aanpassingen ten opzichte van de gegenereerde code.

Voor elk tool data flow diagram worden twee soorten frontends gegenereerd. Op de eerste plaats wordt een command line interface gegenereerd en op de tweede plaats een Maven MOJO. In secties 5.1.7.1 en 5.1.7.2 worden deze gegenereerde frontends besproken. Natuurlijk staat het de ontwikkelaar vrij om de gegenereerde frontends niet te gebruiken en een geheel eigen frontend te ontwikkelen, die de processen uit de backend uitvoert.

5.1.7.1 Command Line Interface

Voor elk tool data flow diagram waar Java code voor gegenereerd wordt, wordt een command line interface frontend gegenereerd. Zoals eerder gezegd is de volgorde waarop de processen uitgevoerd worden in de gegenereerde code bijna nooit juist. De code wordt vast gegenereerd, zodat de ontwikkelaar hier een houvast aan heeft. Door

een annotatie in de code kan aangegeven worden of de frontend opnieuw gegenereerd moet worden (en dus overschreven) of niet.

De gegenereerde command line interface frontend ziet er als volgt uit. De code zonder gele achtergrond is gegenereerd, de code met gele achtergrond is handmatig toegevoegd.

```
package com.example.simpleecho.app;

import com.example.simpleecho.process.importer.Importer;
import com.example.simpleecho.process.exporter.Exporter;

/**
 * Command Line Interface
 *
 * @generated
 */
public class CLI {

    /**
     * @param args
     *
     * @generated NOT
     */
    public static void main(String[] args) {

        if (args.length != 1) {
            System.err.println("Usage: simpleecho message");
            System.exit(1);
        }

        Importer importer = new Importer();
        importer.setMessageParam(args[0]);
        importer.run();

        Exporter exporter = new Exporter();
        exporter.run();
    }
}
```

Codevoorbeeld 5. Command line interface

De backend wordt het eerst ontwikkeld en daarna pas de frontend. Daarom is het handig dat de "main" functie elke keer opnieuw gegenereerd wordt zolang de ontwikkelaar nog met de backend bezig is. Pas als de ontwikkelaar met de frontend aan de slag gaat mag de "main" functie niet meer overschreven worden door de code generator. De ontwikkelaar beslist of dit wel of niet gebeurt door de annotatie "@generated" aan te passen naar "@generated NOT". Nu zal de code generator deze functie niet meer overschrijven.

5.1.7.2 Maven MOJO

Eén van de geïdentificeerde knelpunten is dat wanneer de projectspecifieke tool een tijdje in gebruik is, dat de wens ontstaat om deze in te zetten als stap van het geautomatiseerde build-proces. In de Endeavour Java ontwikkelstraat van Info Support wordt Maven 2 gebruikt voor het builden van de ontwikkelde software.

Maven maakt gebruik van plugins. Elke plugin beschikt over implementaties voor één of meerdere "goals" die ieder een taak van het build proces op zich nemen. Zo zijn er de standaard bekende goals als "clean" en "compile". Iedereen is vrij om eigen plugins voor Maven te ontwikkelen, en hiermee nieuwe goals ter beschikking te stellen. De implementatie van een goal wordt een MOJO genoemd. Dit komt van het woord POJO (Plain-Old-Java-Object), waarbij de P vervangen is door de M van Maven.

Een MOJO is niet veel anders dan een class die overerft van "AbstractMojo", de "execute" functie implementeert en over een aantal annotaties beschikt. De door onze plugin gegenereerde MOJO ziet er als volgt uit:

```
package com.example.simpleecho.app;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;

import com.example.simpleecho.process.importer.Importer;
import com.example.simpleecho.process.exporter.Exporter;

/**
 * Maven Mojo
 *
 * For information see:
 * http://maven.apache.org/guides/plugin/guide-java-plugin-development.html
 *
 * @goal simpleecho
 * @generated
 */
public class MavenMojo extends AbstractMojo {

     /**
      * @parameter
      * @required
      */
     private String messageParam;

    /**
     * @generated NOT
     */
    @Override
    public void execute() throws MojoExecutionException, MojoFailureException {

        Importer importer = new Importer();
        importer.setMessageParam(messageParam);
        importer.run();

        Exporter exporter = new Exporter();
        exporter.run();
    }
}
```

Codevoorbeeld 6. Maven MOJO

Wederom is code zonder gele achtergrond gegenereerd en code met gele achtergrond aangepast. Zoals zojuist bij de command line interface frontend besproken is, wordt de backend eerst ontwikkeld en daarna de frontend. Tijdens het ontwikkelen van de backend is het handig dat de "execute" functie steeds overschreven wordt door de code generator. Wanneer de ontwikkelaar met de frontend aan de slag gaat mag deze functie niet meer overschreven worden. Om dit te bewerkstelligen kan de ontwikkelaar de annotatie "@generated" vervangen door "@generated NOT".

We zien hier een MOJO die de goal "simpleecho" implementeert. Dit is aangegeven door de @goal annotatie in het Javadoc commentaar van de class.

MOJO's kunnen parameters hebben, die per project dat gebuild wordt aangepast kunnen worden. Op deze manier zijn MOJO's generiek te gebruiken in meerdere projecten. De goal "compile" kent bijvoorbeeld een parameter om de versie van de Java compiler in te stellen. In het geval van het voorbeeld hebben we de parameter "messageParam" gedefinieerd. Dit wordt gedaan door een private variabele te maken en te annoteren met "@param". De annotatie "@required" maakt de parameter verplicht. Wanneer deze laatste annotatie weggelaten wordt, wordt de parameter optioneel.

5.1.8 Builden en Uitvoeren

De door onze plugin gegenereerde code is erop voorbereid om behalve door Eclipse ook door Maven 2 gebuild te kunnen worden. Dit geldt dus niet alleen voor de Maven MOJO frontend, maar ook voor de command line interface frontend en eventueel zelf ontwikkelde frontends. Hiervoor zijn alle bestandspaden in het project al op de juiste wijze ingesteld en is het bestand POM.xml aanwezig. Dit bestand vertelt aan Maven op welke wijze de projectspecifieke tool gebuild moet worden.

Het builden met Maven levert een executable JAR op in de "target" directory. Wanneer de projectspecifieke tool over een Maven MOJO frontend beschikt dient de "install" goal uitgevoerd te worden tijdens het builden. Vanaf dat moment is de geïmplementeerde goal beschikbaar voor alle projecten in dezelfde build omgeving.

5.2 Stored Procedure Generator

Nu de tool data flow Eclipse plugin beschreven is en duidelijk is welke functionaliteiten het biedt, gaan we de plugin gebruiken om een daadwerkelijke projectspecifieke tool te ontwikkelen, genaamd de "stored procedure generator".

De stored procedure generator genereert, zoals de naam zegt, stored procedures. Dit wordt gedaan door een verbinding met een Microsoft SQL Server database te maken en uit te lezen welke tabellen zich in deze database bevinden. Per tabel worden vier stored procedures gegenereerd, namelijk stored procedures voor CRUD (Create, Read, Update en Delete) operaties. De originele stored procedure generator genereert nog een aantal andere generiek te gebruiken stored procedures voor de tabellen, maar deze zijn niet van belang voor het proof of concept. De gegenereerde stored procedures worden opgeleverd als een SQL bestand.

5.2.1 Knelpunten

Om aan te tonen dat de knelpunten niet aanwezig zijn als model driven engineering in combinatie met het tool data flow diagram toegepast wordt, oftewel gebruik maken van de tool data flow Eclipse plugin, moeten we er zeker van zijn dat ze wel aanwezig zouden zijn als we de stored procedure generator op de originele manier ontwikkelen. Aangezien deze projectspecifieke tool een praktijkvoorbeeld is weten we dat alle knelpunten voor komen. In deze sectie wordt per knelpunt besproken hoe dit knelpunt voorkwam in de originele versie van de stored procedure generator.

1. **Niet geschikt voor evolutie.** De stored procedure generator is ooit geschreven om hele simpele stored procedures te genereren en is steeds verder uitgebreid om ook complexere taken voor alle tabellen als stored procedure geautomatiseerd beschikbaar te stellen. Hierdoor is de code zeer onoverzichtelijk geworden. In het proof of concept genereren gaan we alleen stored procedures voor CRUD operaties genereren. De tool eerst ontwikkelen voor alleen Create operaties en daarna drie keer uitbreiden zal dit knelpunt niet simuleren, aangezien deze functionaliteiten zo dicht bij elkaar zitten, dat we er in het model niets van terug zien. Om dit knelpunt te simuleren ontwikkelen we de tool in drie iteraties. In de eerste iteratie lezen we nog niet welke tabellen er in de database zitten, maar gebruiken we "hardcoded" informatie. De uitvoer zal afgedrukt worden op de console. In de tweede iteratie gaan we daadwerkelijk verbinding maken met de database om uit te lezen welke tabellen er zijn. Verder zal er behalve uitvoer op de console ook een SQL bestand opgeleverd worden. Als derde iteratie zorgen we dat de tool als stap in het build proces uitgevoerd kan worden.

2. **Niet geschikt voor build proces.** Zoals verteld was het de wens om de stored procedure generator als stap in het build proces uit te voeren. Dit ging niet omdat het uitvoeren van de tool te complex was om het op een goede manier te kunnen automatiseren. In de derde iteratie gaan we ervoor zorgen dat onze command line interface versie uitgebreid wordt, zodat deze als stap in het build proces uitgevoerd kan worden.
3. **Weinig tot geen documentatie.** Bij de originele versie heeft de ontwikkelaar het project verlaten zonder documentatie achter te laten. Mede hierdoor durven andere ontwikkelaars het niet aan om wijzigingen in de tool door te brengen.
4. **Te weinig overdracht aan PDC.** Een stored procedure generator zou niet alleen voor dit project, maar ook zeker voor andere projecten nuttig kunnen zijn. Waarschijnlijk worden een aantal stored procedures gegenereerd die alleen voor dit ene project nuttig zijn, maar het concept van de tool zou zo op andere plaatsen ingezet kunnen worden. Omdat er geen documentatie beschikbaar is en niemand van het project zelf aandurft om een wijziging door te voeren, is het niet mogelijk om de tool over te dragen aan het PDC.
5. **Project wordt geremd door afhankelijkheid tool.** Zoals gezegd is het project afhankelijk van de tool en zou deze zelfs in willen zetten als stap in het build proces. Echter, omdat niemand het aandurft om de tool aan te passen moeten er af en toe ad-hoc oplossingen om de tool heen ontwikkeld worden om de tool te kunnen blijven gebruiken in de gebruiksomgeving die niet stil staat.
5. **Geen simpele richtlijnen.** De tool is gebouwd om snel wat stored procedures te genereren en op dat tijdstip had niemand gedacht dat de tool nog zoveel uitgebreid en gebruikt zou gaan worden. De tool is toentertijd ad-hoc ontwikkeld, zonder enige richtlijn in acht te nemen, waardoor er te weinig aan de toekomst gedacht is.

5.2.2 Eerste Iteratie

Zoals hierboven bij het eerste knelpunt is aangegeven, wordt de stored procedure generator in drie iteraties ontwikkeld om het evolutionaire karakter te simuleren. In de eerste iteratie wordt vooral geconcentreerd op het proces dat van gegeven tabellen stored procedures genereert. De tabellen worden dus "hardcoded" ingevoerd. Hierna worden stored procedures gegenereerd voor deze tabellen. Deze stored procedures worden afgedrukt op de console. In de eerste iteratie zal de tool dus over een command line interface frontend beschikken.

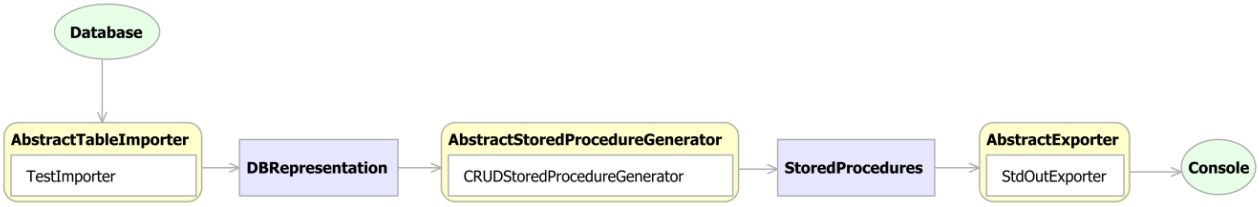
5.2.2.1 Nieuw Maven Project

Om ervoor te zorgen dat de tool door Maven gebuild kan worden, moeten we een Maven project aanmaken. Een standaard Java project in Eclipse heeft enigszins andere bestandspaden. Verder moeten alle dependencies beschikbaar zijn in de Maven build omgeving, waardoor ook in het project de juiste dependency-directories ingesteld moeten staan. De Maven2 plugin voor Eclipse voegt de functionaliteit om een nieuw Maven project aan te maken toe aan Eclipse.

5.2.2.2 Tool Data Flow Diagram

We gebruiken model driven engineering om de tool te ontwikkelen. Daarom starten we met het maken van een model, wat in ons geval een tool data flow diagram is. We zien dat we drie processen nodig hebben. Op de eerste plaats moet een proces een connectie met de database maken en informatie over de tabellen opvragen. Een tweede proces genereert uit deze informatie stored procedures. Een derde proces drukt de stored procedures af op de console.

AbstractTableImporter maakt een verbinding met de database en leest uit welke tabbellen er zich in deze database bevinden. De informatie over deze tabellen zal worden opgeslagen in een intern data object, genaamd DBRepresentation. Het volgende proces kan hier de informatie dan weer uit lezen. Dit volgende proces is de AbstractStoredProcedureGenerator, welke op zijn beurt de gegenereerde stored procedures in het interne data object StoredProcedures wegschrijft. Een AbstractExporter leest de informatie over de stored procedures uit en exporteert dit.



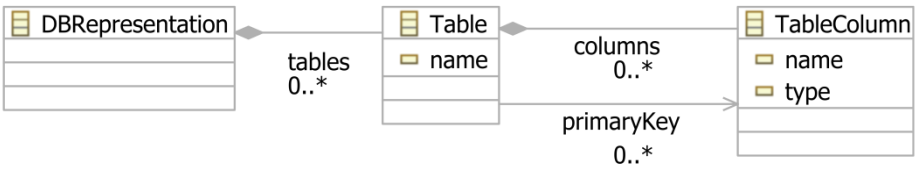
Figuur 18. Tool data flow diagram voor eerste iteratie

Om zo generiek mogelijk voor de toekomst te blijven geven we de abstracte processen een generieke naam, zoals bijvoorbeeld "AbstractStoredProcedureGenerator". De namen van de procesimplementaties zijn concreter, bijvoorbeeld "CRUDStoredProcedureGenerator". Hierdoor kunnen andere implementaties later op een gemakkelijke manier aan dezelfde abstracte processen worden toegevoegd. Bij deze tool zou het bijvoorbeeld kunnen zijn dat later ook andere stored procedures gegenereerd dienen te worden. Hiervoor zou dan een nieuwe implementatie aan het abstracte proces "AbstractStoredProcedureGenerator" toegevoegd kunnen worden.

5.2.2.3 Interne Data Objecten

Nu het tool data flow diagram gemodelleerd is, kunnen de interne data objecten gemodelleerd worden. Deze kunnen geopend worden door erop te dubbelklikken. We zien dan dat één top level class gedefinieerd is. De rest van het data model moet bereikbaar zijn vanaf deze class.

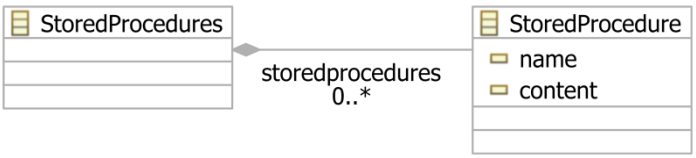
We modelleren het interne data object "DBRepresentation" als volgt:



Figuur 19. Initiële Ecore diagram van intern data object DBRepresentation

De database-representatie heeft nul of meerdere tabellen en elke tabel heeft nul of meerdere kolommen. In de praktijk moet een tabel minimaal één kolom hebben. Dit kan ook gemodelleerd worden, maar dan zal er gebruik gemaakt moeten worden van een zogenaamde validator om dat ook daadwerkelijk af te vangen. Omdat wij geen validator gaan gebruiken, is "nul of meerdere", wat de standaard property is, voldoende. Hetzelfde geldt voor de relatie primaryKey.

Het interne data object "StoredProcedure" ziet er als volgt uit:



Figuur 20. Initiële Ecore diagram van intern data object StoredProcedures

Hier zien we dat de top level class "StoredProcedures" nul of meerdere stored procedures bevat. Een stored procedure heeft een naam en een content. De content is een string met de T-SQL code van de stored procedure.

5.2.2.4 Procesimplementaties

Vanuit bovenstaande modellen wordt allerlei code gegenereerd, die we nu zelf kunnen implementeren. We beginnen met de procesimplementatie "TestImporter". Dit proces schrijft "hardcoded" tabelinformatie in het interne data object "DBRepresentation". De code om dit te bewerkstelligen ziet er als volgt uit:

```
public void run() {
    Table myTableOne = DBRepresentationFactory.eINSTANCE.createTable();
    myTableOne.setName("MyTableOne");
        TableColumn intColumn = DBRepresentationFactory.eINSTANCE.createTableColumn();
        intColumn.setName("IntColumn");
        intColumn.setType("INT");
        myTableOne.getColumns().add(intColumn);
        myTableOne.getPrimaryKey().add(intColumn);

        TableColumn strColumn = DBRepresentationFactory.eINSTANCE.createTableColumn();
        strColumn.setName("StrColumn");
        strColumn.setType("varchar(100)");
        myTableOne.getColumns().add(strColumn);
    dBRepresentation.getTables().add(myTableOne);

    Table myTableTwo = DBRepresentationFactory.eINSTANCE.createTable();
    myTableTwo.setName("MyTableTwo");
        TableColumn timestampColumn =
            DBRepresentationFactory.eINSTANCE.createTableColumn();
        timestampColumn.setName("TimestampColumn");
        timestampColumn.setType("TIMESTAMP");
        myTableTwo.getColumns().add(timestampColumn);
        myTableTwo.getPrimaryKey().add(timestampColumn);

        TableColumn moneyColumn = DBRepresentationFactory.eINSTANCE.createTableColumn();
        moneyColumn.setName("MoneyColumn");
        moneyColumn.setType("MONEY");
        myTableTwo.getColumns().add(moneyColumn);
    dBRepresentation.getTables().add(myTableTwo);
}
```

Codevoorbeeld 7. "run" functie van procesimplementatie TestImporter

Het is heel even wennen om de door EMF gegenereerde code van het interne data object te gebruiken. De objecten moeten namelijk geïnstantieerd worden door middel van een zogenaamde "factory" in plaats van met het "new" keyword. Na het zien van een voorbeeldje spreekt het verder voor zich hoe het interne data object gebruikt dient te worden.

Nu implementeren we de CRUDStoredProcedureGenerator. We laten hier niet alle inhoudelijke code zien, maar alleen de "run" functie. Deze ziet er als volgt uit:

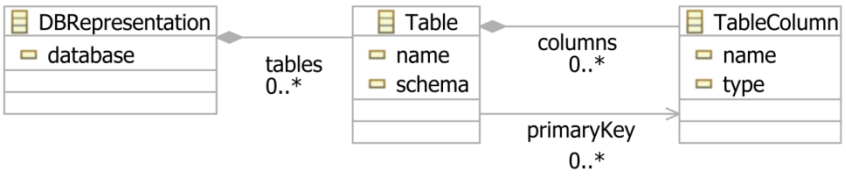
```
public void run() {
    for (Iterator<Table> i = dBRepresentation.getTables().iterator(); i.hasNext(); ) {
        Table table = i.next();

        storedProcedures.getStoredprocedures().add(createInsertSP(table));
        storedProcedures.getStoredprocedures().add(createSelectSP(table));
        storedProcedures.getStoredprocedures().add(createUpdateSP(table));
        storedProcedures.getStoredprocedures().add(createDeleteSP(table));
    }
}
```

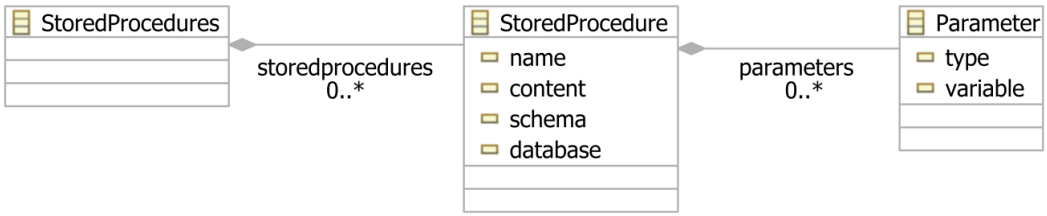
Codevoorbeeld 8. "run" functie van procesimplementatie CRUDStoredProcedureGenerator

Hier zien we dat wanneer we voor alle objecten uit een relatie iets moeten doen we heel handig een zogenaamde iterator kunnen definiëren. Hierdoor hoeven we niet met lijsten en lengtes daarvan om te gaan. Verder is de performance bij het gebruik van een iterator hoger dan elke keer het i-de element opvragen.

Als laatste implementeren we de StdOutExporter. Deze haalt de informatie uit het interne data object StoredProcedures en drukt de gegenereerde stored procedures af op de console. Omdat we redelijk ad-hoc aan het ontwikkelen zijn en blijkbaar niet goed genoeg over de interne data objecten nagedacht hebben blijken we één en ander te missen. Zo weten we niet meer hoe de database heet, welke schema's bij welke tabellen en stored procedures horen en stored procedures hebben parameters nodig. Hiervoor passen we de Ecore diagrammen van de interne data objecten als volgt aan:



Figuur 21. Aangepast Ecore diagram van intern data object DBRepresentation



Figuur 22. Aangepast Ecore diagram van intern data object StoredProcedures

Het aanpassen van de interne data objecten levert geen problemen op. We passen de Ecore diagrammen aan, laten vanuit het tool data flow diagram opnieuw code genereren en passen de reeds geïmplementeerde processen zonder problemen aan. Vooral het feit dat code completion direct na het genereren van de code beschikbaar is vergemakkelijkt het implementeren/aanpassen van de processen.

5.2.2.5 Command Line Interface

De backend is geïmplementeerd en nu is de frontend aan de beurt. We implementeren een command line interface. In de gegenereerde code hiervoor hoeven we alleen de volgorde dat de processen uitgevoerd dienen te worden aan te passen. In ons geval

hadden we de processen in de goede volgorde in het diagram getekend en voldoet de gegenereerde code.

5.2.2.1 Builden en Uitvoeren

De stored procedure generator kan nu gemakkelijk gebuild worden door een opdracht prompt te openen, naar de juiste directory toe te gaan en "mvn clean dependency:copy-dependencies package" aan te roepen. Dit levert een executable JAR bestand op in de "target" directory.

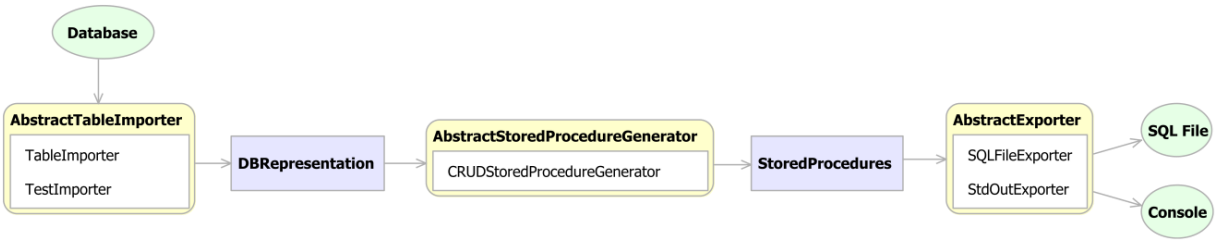
5.2.3 Tweede Iteratie

In de tweede iteratie wordt de functionaliteit van de stored procedure generator uitgebreid. Er zal nu een daadwerkelijke verbinding met de Microsoft SQL Server database gemaakt worden en uitgelezen worden welke tabellen zich in de database bevinden. Dit wordt geïmporteerd in de DBRepresentation.

Verder zullen de gegenereerde stored procedures behalve afgedrukt worden op de console, ook weggeschreven worden naar een SQL bestand. Dit zal parallel gedaan worden.

5.2.3.1 Tool Data Flow Diagram

Bij model driven engineering staat het model centraal, dus we passen als eerste het model, oftewel het tool data flow diagram aan. We hebben een vervanging voor de TestImporter nodig en de AbstractExporter dient uitgebreid te worden met een SQLFileExporter. Het nieuwe tool data flow diagram ziet er als volgt uit:



Figuur 23. Tool data flow diagram voor tweede iteratie

TestImporter is in deze iteratie overbodig geworden, maar het staat niemand in de weg, want als deze implementatie niet gestart wordt door een frontend gebeurt er niets mee. Wel moet er opgelet worden, dat wanneer de verbonden interne data objecten (in dit geval alleen DBRepresentation) aangepast wordt, dat het mogelijk is dat de code niet meer compileert. Het is natuurlijk wel een goed idee om in het commentaar van de class TestImporter aan te geven dat deze niet meer gebruikt wordt en wat er verwacht kan worden wanneer deze later toch weer nuttig zou blijken te zijn.

5.2.3.2 Interne Data Objecten

Er hoeft niets aangepast te worden aan de interne data objecten.

5.2.3.3 Procesimplementaties

Het implementeren van de TableImporter is niets anders dan zorgen dat in de "run" functie van die class een verbinding met de database wordt maakt, meta-informatie over de tabellen uitgelezen wordt en dit aan het interne data object DBRepresentation

wordt toegevoegd. De code voor deze functionaliteit is te irrelevant om hier te beschrijven.

Bij het toevoegen van de procesimplementatie SQLFileExporter zien we dat de functionaliteit voor het grootste gedeelte overeenkomt met StdOutExporter. In beide processen moet namelijk een string opgebouwd worden van de gegenereerde stored procedures. Deze string moet in het ene geval afgedrukt worden naar de console en in het andere geval weggeschreven worden naar een SQL bestand.

We passen de implementatie van StdOutExporter aan, zodat het opbouwen van de string in een aparte functie gebeurd. Nu kunnen we deze in Eclipse heel gemakkelijk naar AbstractExporter verplaatsen door de cursor ergens op of in de functie te plaatsen en in het menu "Refactor" voor "Pull Up..." te kiezen. Nu is deze functie voor beide procesimplementaties beschikbaar en hoeft er dus geen dubbele code geschreven te worden.

5.2.3.4 Command Line Interface

De interne data objecten hoefden niet aangepast te worden en sommige abstracte processen zijn uitgebreid met procesimplementaties. Nu hoeven we er alleen nog maar voor te zorgen dat de frontend de nieuwe processen aanroept. In plaats van de TestImporter dient nu de TableImporter uitgevoerd te worden en verder dient de SQLFileExporter parallel uitgevoerd te worden met StdOutExporter. We passen de code aan, zodat deze er als volgt uit ziet:

```java
package com.infosupport.proc.spgenerator.app;

import com.infosupport.proc.spgenerator.process.exporter.SQLFileExporter;
import com.infosupport.proc.spgenerator.process.exporter.StdOutExporter;
import com.infosupport.proc.spgenerator.process.....CRUDStoredProcedureGenerator;
import com.infosupport.proc.spgenerator.process.tableimporter.TableImporter;

/**
 * Command Line Interface
 *
 * @generated
 */
public class CLI {

    /**
     * @param args
     *
     * @generated NOT
     */
    public static void main(String[] args) {

        // Check for server address parameter
        if (args.length != 1) {
            System.err.println("Usage: spgenerator serverAddress");
            System.exit(1);
        }
        String serverAddress = args[0];

        // Import table information into DBRepresentation
        TableImporter tableImporter =
            new TableImporter(serverAddress, 1381, "AdventureWorksLT");
        tableImporter.setDbschema("SalesLT");
        tableImporter.setUsername("myUsername");
        tableImporter.setPassword("myPassword");
        tableImporter.run();

        // Generate stored procedures for CRUD operations
        CRUDStoredProcedureGenerator cRUDStoredProcedureGenerator =
            new CRUDStoredProcedureGenerator();
        cRUDStoredProcedureGenerator.run();

        // Export to both SQL file and console (parallel)
        Thread sqlFileExporter = new Thread(new SQLFileExporter("stored_procedures.sql"));
        Thread stdOutExporter = new Thread(new StdOutExporter());

        sqlFileExporter.start();
        stdOutExporter.start();

        try {
            sqlFileExporter.join();
            stdOutExporter.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Codevoorbeeld 9. Command line interface

Hier zien we wederom hoe parameters aan een procesimplementatie meegegeven kunnen worden. Verder zien we hier dat we de procesimplementaties gemakkelijk als thread uitgevoerd kunnen worden.

Het gegenereerde SQL bestand is in de Microsoft SQL Server geladen en alle stored procedures werden met succes toegevoegd. Hiermee is de tweede iteratie voltooid.

5.2.4 Derde Iteratie

In de derde iteratie wordt de stored procedure generator aangepast zodat deze als stap in het build proces opgenomen kan worden. De standaard build omgeving die

gebruikt wordt wanneer een project ontwikkeld wordt met behulp van de Endeavour Java ontwikkelstraat is Maven. Een stap in het buildproces van Maven wordt geïmplementeerd als Maven MOJO, zoals in sectie 5.1.7.2 te lezen is.

5.2.4.1 Maven MOJO

Een Maven MOJO lijkt wat structuur betreft veel op die van de command line interface. We moeten bij beide een soort van "main" functie implementeren. In een Maven MOJO is dit de "execute" functie. Verder moeten we een goal en eventueel parameters definiëren. Net zoals in de command line interface versie gebruiken we altijd dezelfde database en hoeven we dit niet als parameter mee te geven.

We kunnen de inhoud van de "main" functie uit de command line interface frontend bijna één-op-één overnemen naar de "execute" functie. Het verschil is dat de parameters van de applicatie bij de "main" functie als "args" parameters meegegeven wordt. Bij een Maven MOJO dienen hier geannoteerde private variabelen voor gedefinieerd te worden. De Maven MOJO ziet er als volgt uit:

```java
package com.infosupport.proc.spgenerator.app;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;

import com.infosupport.proc.spgenerator.process.exporter.SQLFileExporter;
import com.infosupport.proc.spgenerator.process.exporter.StdOutExporter;
import com.infosupport.proc.spgenerator.process.....CRUDStoredProcedureGenerator;
import com.infosupport.proc.spgenerator.process.tableimporter.TableImporter;

/**
 * Maven Mojo
 *
 * @see http://maven.apache.org/guides/plugin/guide-java-plugin-development.html
 *
 * @goal spgenerate
 * @generated
 */
public class MavenMojo extends AbstractMojo {

    /**
     * @parameter
     * @required
     */
    private String serverAddress;

    /**
     * @generated NOT
     */
    @Override
    public void execute() throws MojoExecutionException, MojoFailureException {

        // Import table information into DBRepresentation
        TableImporter tableImporter =
            new TableImporter(serverAddress, 1381, "AdventureWorksLT");
        tableImporter.setDbschema("SalesLT");
        tableImporter.setUsername("myUsername");
        tableImporter.setPassword("myPassword");
        tableImporter.run();

        // Generate stored procedures for CRUD operations
        CRUDStoredProcedureGenerator cRUDStoredProcedureGenerator =
            new CRUDStoredProcedureGenerator();
        cRUDStoredProcedureGenerator.run();

        // Export to both SQL file and console (parallel)
        Thread sqlFileExporter = new Thread(new SQLFileExporter("stored_procedures.sql"));
        Thread stdOutExporter = new Thread(new StdOutExporter());

        sqlFileExporter.start();
        stdOutExporter.start();

        try {
            sqlFileExporter.join();
            stdOutExporter.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

We zien dat het gemakkelijk is om een projectspecifieke tool met command line interface frontend als stap in het build proces uit te voeren. We kunnen de frontend namelijk gemakkelijk herschrijven als Maven MOJO.

Het mag duidelijk zijn dat wanneer we een interactieve tool hebben, het nog niet moeilijk is om hier een Maven MOJO voor te schrijven. We moeten immers de juiste processen achter elkaar, met je juiste parameters, aanroepen. Deze parameters worden met de oude frontend interactief ingesteld. Aangezien het build proces niet

interactief is, zullen deze parameters of vast staan, of als parameter van de Maven MOJO meegegeven moeten worden.

5.2.5 Builden en Uitvoeren

Het builden gaat hetzelfde als bij de command line interface versie. Alleen nu moet de "install" goal toegevoegd worden om de Maven plugin, waar de Maven MOJO onderdeel van is, toe te voegen aan de build omgeving. Hierna kan de tool gebruikt worden tijdens het builden van een ander project door de gemaakte plugin/goal op te nemen in de configuratie van deze build. Dit gebeurt door de POM.xml van het project aan te passen.

5.3 Afgedekte Knelpunten

In sectie 5.2.1 zagen we dat de alle knelpunten voor kwamen als we de stored procedure generator op de huidige manier ontwikkelen. We bespreken hier wederom alle knelpunten om aan te tonen dat deze knelpunten opgelost zijn als we de Eclipse plugin, die het concept uit hoofdstuk 4 implementeert, gebruiken.

1. **Evolutionaire karakter.** Het tool data flow diagram biedt het juiste abstractieniveau. Zo staat het diagram niet zo dicht op de code als bijvoorbeeld een class diagram, waarbij het diagram alleen de code visualiseert maar niet gemakkelijk te zien is waar nieuwe wijzigingen doorgevoerd zouden moeten worden. Het diagram is ook weer niet te abstract, zodat er geen code uit gegenereerd kan worden. Hierdoor kan het diagram ingezet worden in combinatie met model driven engineering, wat het ontwikkelproces versimpeld en later gemakkelijk gezien kan worden waar de tool aangepast dient te worden als een wijziging nodig en/of gewenst is.
   Zo zien we bij het ontwikkelen van de stored procedure generator dat we het exporteren heel gemakkelijk uit kunnen breiden met een SQL file exporter. Als we andere soorten stored procedures willen genereren, kunnen we gemakkelijk een procesimplemenatie aan AbstractStoredProcedureGenerator toevoegen. Eventueel kunnen we ook gemakkelijk een geheel nieuw proces toevoegen
2. **Niet geschikt voor build proces.** Doordat de gegenereerde code van de plugin een duidelijk onderscheid maakt tussen de backend en de frontend is het gemakkelijk om de frontend te vervangen. Dit zien we in de derde iteratie in het proof of concept. Er wordt namelijk code voor een Maven MOJO gegenereerd, zodat de tool daarmee als stap in het build proces opgenomen kan worden.
3. **Weinig tot geen documentatie.** Doordat model driven engineering wordt toegepast is het model, wat in ons geval een tool data flow diagram is, altijd beschikbaar. Dit diagram biedt documentatie over welke processen en data stores er zijn. De data stores worden op hun beurt weer gemodelleerd door een class diagram, waardoor ook hiervoor documentatie beschikbaar is. Verder is alle gegenereerde code voorzien van Javadoc commentaar, wat ook een vorm van documentatie is. Deze documentatie is voldoende om de tool over te dragen aan een andere ontwikkelaar.
4. **Te weinig overdracht aan PDC.** Omdat er genoeg documentatie is en de tool gemakkelijk aangepast kan worden, zoals we al bij het knelpunt "Niet geschikt voor evolutie" zagen, is de overdracht aan het PDC een stuk versimpeld. De overdracht zelf is namelijk versimpeld vanwege de beschikbare documentatie. Meestal zal het PDC de tool generieker maken, zodat deze bij meerdere projecten ingezet kan worden. Deze aanpassing is versimpeld omdat het knelpunt "Niet geschikt voor evolutie" opgelost wordt.
5. **Project wordt geremd door afhankelijkheid tool.** Wanneer de gebruiksomgeving verandert en de tool mee moet veranderen kan teruggevallen worden op het diagram om snel in te zien waar de verandering plaats moet

vinden. Omdat de processen op zichzelf staan zal het niet snel voor komen dat allerlei code door elkaar gemixt is, waardoor kleine wijzigingen grote effecten kunnen veroorzaken. Door het tool data flow diagram te gebruiken kan het effect van een verandering uit het diagram opgemaakt worden.
Stel, er komt een nieuwe versie versie van Microsoft SQL Server uit, waardoor de stored procedure generator niet meer werkt. Nu kan in het diagram snel gezien worden dat de TableImporter aangepast moet worden. Eventueel kan er ook een nieuwe implementatie gemaakt worden, waarbij vanaf de frontend gekozen kan worden of er gebruik wordt gemaakt van een oude of nieuwe versie database en aan de hand daarvan kan dan de juiste implementatie uitgevoerd worden.

6. **Geen simpele richtlijnen.** Het gebruik van model driven engineering biedt al richtlijnen. De ontwikkelde Eclipse plugin maakt het gebruik hiervan simpeler. Zo biedt het de mogelijkheid om een diagram te tekenen waarbij constraints, zoals data flows die alleen van data object naar proces of omgekeerd getekend mogen worden, afgevangen zullen worden. Verder wordt een gedeelte van de implementatie gegenereerd uit het diagram, waardoor het zeker is dat de implementatie aan het model voldoet en het ontwikkelproces vergemakkelijkt wordt. Hierdoor worden er richtlijnen ingevoerd die het de ontwikkelaar niet moeilijker, maar gemakkelijker maken, maar waarbij ze wel de overige knelpunten oplossen.

5.4 Ervaringen Ontwikkelaar Info Support

Een medewerker van Info Support, die zelf eerder een aantal projectspecifieke tools ontwikkeld heeft, heeft de Eclipse plugin gebruikt om een fictieve projectspecifieke tool te ontwikkelen. Dit om te onderzoeken of het tool data flow diagram en de Eclipse plugin gemakkelijk in gebruik zijn. Zijn bevindingen zijn in deze sectie opgenomen.

Het aanmaken van een Maven project en hier een tool data flow diagram aan toevoegen is gemakkelijk. Hierna werden externe data objecten toegevoegd aan het diagram.

Na de externe data objecten kwamen de abstracte processen en de interne data objecten aan de beurt. Hierbij wordt opgemerkt dat processen en interne data objecten nog redelijk vaak een nieuwe naam krijgen. In de ontwikkelde Eclipse plugin wordt de gegenereerde code niet refactored, maar worden nieuwe processen met de nieuwe naam naast de processen met de oude naam gegenereerd. Hierdoor lijkt al eerder geïmplementeerde code te verdwijnen. Voordat de Eclipse plugin grootschalig in een productie omgeving ingezet kan gaan worden zal ervoor gezorgd moeten worden dat wanneer een naam in het model aangepast wordt, dit een refactoring van de code oplevert in plaats van dat nieuwe processen ernaast gegenereerd worden.

Het tool data flow diagram vangt automatisch af dat data flows altijd tussen een data object en een proces of andersom gemodelleerd worden en nooit tussen twee data objecten of tussen twee processen. Dit helpt bij het begrijpen van het diagram.

De namen van abstracte processen dienen te beginnen met “Abstract”, anders wordt er foutieve code gegenereerd. Op dit moment wordt deze beperking in naamgeving niet afgedwongen, waardoor dit uit documentatie gehaald moet worden. Verder worden de namen van de elementen van het diagram gebruikt als class namen in de gegenereerde code. Hierdoor zijn spaties en speciale tekens verboden. Ook dit wordt niet afgedwongen door het diagram. Voordat de Eclipse plugin in een productie omgeving gebruikt gaat worden zullen constraints op de namen van de verschillende

elementen uit het diagram gelegd moeten worden, zodat dit niet uit documentatie gehaald hoeft te worden.

Wanneer gedubbelklikt wordt op een intern data object wordt een Ecore editor met een class diagram voor dit object geopend. Het modelleren van de datastructuur is gemakkelijk. Echter, vanuit deze view is het niet snel duidelijk hoe nieuwe source code gegenereerd kan worden. Hier moet namelijk eerst voor teruggegaan worden naar de grafische editor van het tool data flow diagram, of met de rechter muisknop op het bestand in de "Package explorer" geklikt worden. Voordat de Eclipse plugin in een productie omgeving gebruikt gaat worden zal het genereren van code ook aan de Ecore editor toegevoegd moeten worden.

Het lezen en schrijven van en naar een intern data object is gemakkelijk. Het is even wennen dat nieuwe objecten door middel van een "factory" geïnstantieerd moeten worden, maar dit wijst zichzelf snel. Hierna biedt de code completion van Eclipse veel hulp bij het gebruik van de datastructuur in de procesimplemenaties. Zo zien we namelijk door middel van een lijstje welke getters, setters en bijvoorbeeld "add" en "contains" functies bij elk object beschikbaar zijn.

De gegenereerde code staat in een vaste directory-structuur, welke gedeeltelijk aan te passen is in het diagram. Dit kan door de properties van de elementen te wijzigen. De ervaring is dat dit nog uitgebreider zou mogen, zodat de ontwikkelaar hier meer zeggenschap over heeft. Dit zal toegevoegd moeten worden voordat de Eclipse plugin in een productie omgeving ingezet kan gaan worden.

We concluderen dat het leren van het tool data flow diagram niet moeilijk is, vooral omdat de meeste beperkingen, zoals waartussen data flows gemodelleerd mogen worden, afgevangen worden. Een ontwikkelaar zal wel even de omschakeling moeten maken dat dat de pijlen geen functie-calls representeren, maar informatie stromen. Het gebruik van het diagram in de Eclipse ontwikkelomgeving werkt, maar mist nog een aantal basisfunctionaliteiten voordat het in een productie omgeving ingezet kan worden. Het toevoegen van de missende functionaliteiten zullen geen aangrijpende verandering van de Eclipse plugin zijn, maar vooral het doorontwikkelen van het proof of concept naar een plugin die in te zetten is in een productie omgeving.

5.5 Conclusies

Het ontwikkelen van een grafische editor voor Eclipse was niet zo simpel als het in eerste instantie leek. Gelukkig hoeft dit maar eenmalig te gebeuren en kan het daarna overal gebruikt worden. Om een grafische editor te ontwikkelen zijn een aantal frameworks (Eclipse, [EMF], [GEF] en [GMF]) benodigd. Het leren omgaan met deze frameworks neemt redelijk wat tijd in beslag. Dit komt vooral ook omdat we vast zitten aan deze frameworks (als we niet alles handmatig willen ontwikkelen) en niet alle frameworks even volwassen zijn.

Het Eclipse Modeling Framework [EMF] is zeer volwassen en gemakkelijk te leren, mits niet alle uitbreidingen zoals het validatie framework, commands met undo/redo functionalteit enzovoorts gebruikt worden, want dan neemt de leercurve snel toe. Met EMF kan een datamodel op gemakkelijke wijze gemodelleerd worden en hier kan code voor gegenereerd worden. Het framework laat het toe om opnieuw code te genereren wanneer het model wordt aangepast. Het is aan te bevelen om EMF behalve bij het ontwikkelen van projectspecifieke tools ook bij het ontwikkelen van andere projecten in te zetten.

Het Graphical Editing Framework [GEF] is volwassen genoeg om te gebruiken. De ontwikkelaar hoeft hier bijna geen kennis over te hebben als het Graphical Modeling Framework [GMF] gebruikt wordt, omdat bijna alle code die gebruik maakt van GEF gegenereerd wordt. Alleen kleine details die niet met GMF te modelleren zijn moeten handmatig geïmplementeerd worden.

Het Graphical Modeling Framework [GMF] is nog niet volwassen genoeg om op grote schaal te gebruiken. GMF genereert source code voor een grafische editor. Er wordt echter regelmatig foutieve code gegenereerd. Zo werd er soms code gegenereerd die code uit internals van GEF aanroept. Wanneer je deze code probeert te compileren zal dit een aantal errors opleveren zoals “Access denied due to access restrictions”. De code wordt gegenereerd aan de hand van een aantal door de ontwikkelaar gedefinieerde modellen. Als deze modellen helemaal juist zijn, levert GMF goede source code op en kan dit prima gebruikt worden. Het kost echter redelijk wat tijd om de modellen helemaal juist te krijgen, vooral aangezien onjuiste modellen vaak als valide valideren.

Jave Emitter Templates (JET), een onderdeel van het Eclipse Modeling Framework [EMF], is gebruikt om de code generator te ontwikkelen. Het is redelijk gemakkelijk te leren en is volwassen genoeg om te gebruiken.

De stored procedure generator toont aan dat een willekeurige projectspecifieke tool, waar bij een eerdere implementatie alle knelpunten naar voren kwamen, op de nieuwe manier van ontwikkelen geïmplementeerd kan worden, terwijl de knelpunten niet meer naar voren komen. Het aanpassen van de tool, bijvoorbeeld het toevoegen van functionaliteit, is vergemakkelijkt door het gebruik van het tool data flow diagram en het feit dat hieruit code ge(her)genereerd kan worden.

Hoe de processen intern geïmplementeerd worden staat de ontwikkelaar helemaal vrij. Daardoor kunnen de kleine losstaande functionaliteiten toch nog op een redelijk “ad-hoc” wijze ontwikkeld worden en staat het de ontwikkelaar vrij om eigen frameworks in te zetten. Hierdoor is het tool data flow diagram gemakkelijk in te zetten en loopt de ontwikkelaar niet continu tegen restricties aan.

6 Gerelateerd Onderzoek

Tools zijn een onderdeel van software engineering. In Pressman's "Software Engineering – A Practitioner's Approach" [PRESS] en in Sommerville's "Software Engineering" [SSE] wordt dan dus ook verteld dat tools nodig zijn bij het ontwikkelen van software. Echter, hoe deze tools te categoriseren zijn of hoe het ontwikkelproces eruit ziet wordt niet beschreven.

In de jaren '90 kwamen de zogenaamde CASE tools op, waarbij tools gestandaardiseerd, georganiseerd en geïntegreerd werden. Wasserman geeft aan hoe verschillende CASE tools geïntegreerd kunnen worden [TISE]. Artikelen van Kranenbug [KRAN] en Greenfield en Short [GSSF] geven aan waar tools zich in een software ontwikkelstraat bevinden en hoe ze daar geïntegreerd zouden moeten worden.

Naar ons weten is er nog niet eerder onderzoek gedaan naar het ontwikkelen van projectspecifieke tools. Ze worden gezien als het ontwikkelen van een normaal software project, maar uit ons onderzoek is gebleken dat de aanpak geheel anders is, namelijk zeer ad-hoc. Dit is niet te vinden in andere boeken of artikelen.

Het tool data flow diagram is afgeleid van het Data Flow Diagram. Pressman geeft aan hoe een Data Flow Diagram gebruikt dient te worden [PRESS]. Dit model is abstract en beschrijft niet hoe dit af te beelden zou zijn op source code. In overige artikelen, zoals van Brown [MDACP] en Medvidovic, Egyed en Rosenblum [RTSE] wordt de combinatie van meerdere diagrammen, waaronder het Data Flow Diagram, gebruikt om source code te genereren/implementeren.

We passen model driven engineering [MDE] toe in onze aanbeveling hoe projectspecifieke tools ontwikkeld zouden moeten worden. We hebben dus geen onderzoek naar model driven engineering zelf gedaan, maar het toegepast. De basisprincipes hiervan wordt uitgelegd door Bézivin [BPMDE] en Brown [MDACP]. Hierbij wordt het systeem eerst heel abstract gemodelleerd. Hierna wordt twee keer in een nieuwe diagram meer detail toegevoegd en deze laatste wordt geïmplementeerd. Omdat projectspecifieke meestal niet complex zijn is het niet nuttig om meta-meta-modellen of meta-modellen in te zetten. We gebruiken dus alleen de onderste laag: "model representeert implementatie".

Ook uit het artikel van Medvidovic, Egyed en Rosenblum [RTSE], waarbij roundtrip engineering wordt uitgelegd, passen we alleen de stap van model naar implementatie toe. In dit paper wordt de stap van implementatie naar model (al dan niet via reverse engineering) gemaakt. Dit doen wij niet, omdat handmatig een proces of intern data object toevoegen in de implementatie niet "per ongeluk" kan gebeuren. Hiervoor moeten namelijk een aantal packages aangemaakt worden, met een juiste implementatie van verschillende (abstracte) classes en interfaces. Deze roundtrip heeft in ons geval geen toegevoegde waarde.

We hebben het Graphical Modeling Framework [GMF] gebruikt om een grafische editor voor het tool data flow diagram te ontwikkelen. Java Emitter Templates uit het Eclipse Modeling Framework [EMF] werden ingezet om uit dit diagram Java code te genereren. Ehrig, Ermel, Hänsgen en Taentzer [GVE] beschrijven hoe dit werkt. Zij gebruiken Tiger [TIGER] om te zorgen dat het model ten allen tijde valid is. We hebben er in deze thesis voor gekozen om een minimum aantal plugins te gebruiken, en hebben constraints door middel van Object Constraint Language [OCL] geïmplementeerd, die standaard aanwezig zijn in Eclipse. OCL is minder uitgebreid en

wat lastiger implementeren dan Tiger, maar aangezien dit alleen te maken heeft met het ontwikkelen van de Eclipse plugin en de ontwikkelaars van projectspecifieke tools hier dus niets mee te maken hebben is dit geen probleem.

7 Conclusies en Aanbevelingen

In dit hoofdstuk worden sectie 7.1 algemene conclusies getrokken. Hierna worden in sectie 7.2 aanbevelingen gedaan over het ontwikkelen van projectspecifieke tools en toekomstig onderzoek.

7.1 Conclusies

Projectspecifieke tools worden op dit moment vaak op een ad-hoc wijze ontwikkeld. Wanneer de ontwikkelaar begint met het ontwikkelen van de tool wordt er vaak gedacht dat de tool maar een beperkt aantal keer ingezet zal gaan worden, waardoor het niet zo precies komt hoe de tool geïmplementeerd wordt. In de praktijk zien we echter dat de projectspecifieke tools nuttig blijven en dat ze uitgebreid en/of aangepast aan de gebruiksomgeving dienen te worden. Nu is er te weinig documentatie beschikbaar en de tools zijn zo geïmplementeerd dat deze bijna niet aan te passen zijn. Verder wordt code van frontend en backend door elkaar gebruikt, waardoor de fontend niet vervangbaar is. Hierdoor is het dan ook vaak onmogelijk om de tool op te nemen als stap in het geautomatiseerde build proces.

Model driven engineering wordt steeds meer en meer toegepast, maar we zien dat eigenlijk alleen in de vorm van model driven architecture [MDA]. Dit is model driven engineering met een class diagram als model. Dit diagram staat echter heel dicht bij de code en wat abstractere diagrammen zouden gewenst zijn. Unified Modeling Language [UML] heeft een aantal modellen die abstracter zijn, maar hier kan dan geen code uit gegenereerd worden, waardoor het in sync houden van model en code lastig blijft en het ontwikkelproces niet echt versimpeld wordt. De implementatie moet immers nog steeds vanaf scratch ontwikkeld worden. Pas wanneer meer abstractere diagrammen beschikbaar komen om als model gebruikt te worden bij model driven engineering zal het laatste zijn nut pas echt laten zien.

Het inzetten van het in deze thesis ontwikkelde tool data flow diagram in combinatie met model driven engineering biedt de gewenste extra abstractie, plus dat er code vanuit het model gegenereerd kan worden. Hierdoor wordt het de ontwikkelaar gemakkelijker gemaakt, terwijl het leren omgaan met de Eclipse plugin gemakkelijk is, zoals is gebleken toen een ontwikkelaar van Info Support er gebruik van heeft gemaakt. Tevens biedt het het voordeel dat de ontwikkelaar wordt gedwongen om de tool eerst te modelleren, waardoor de geïdentificeerde knelpunten opgelost worden.

Tijdens de categorisatie in hoofdstuk 2 zagen we dat “code generators” en “data converters” niet de enige categorieën projectspecifieke tools zijn. Dit zijn wel de meest voorkomende projectspecifieke tools. Projectspecifieke tools uit de categorie “build tools” zijn tools die opgenomen zijn in het build proces van een project. We hebben gezien dat deze tools te ontwikkelen zijn door middel van het tool data flow diagram en model driven engineering. Hier zijn we uitgegaan van Maven als build omgeving, maar omdat de backend geheel los staat van de frontend kunnen ook andere build omgevingen zoals bijvoorbeeld Apache Ant [ANT] of MSBuild [MSBUILD] gebruikt worden.

Bij projectspecifieke tools uit de categorie “refactoring” wordt een model of een set aan source code ingevoerd. Hier worden bepaalde elementen aangepast, bijvoorbeeld de namen van bepaalde functies in source code. Daarna wordt alle invoer, maar nu met de wijzigingen als output opgeleverd. Hiervoor is een informatiestroom modelleerbaar, dus ook deze categorie wordt afgedekt door het tool data flow diagram.

Een aantal projectspecifieke tools kunnen we indelen in de categorieën "design/modeling" en "testing/analysis". Ook tools uit deze categorieën kunnen door middel van een tool data flow diagram gemodelleerd kan worden, aangezien bij zowel modelleren als testen en analyse redelijk wat informatie verzameld wordt. De verzamelprocessen, de verzamelde informatie en processen die iets met de verzamelde informatie doen kunnen door middel van het tool data flow diagram gemodelleerd worden.

Projectspecifieke tools uit de categorie "system administration" zijn bijna altijd batch- of shell-scripts, waarbij achtereenvolgens commando's uitgevoerd worden. Hiervoor kan geen tool data flow diagram gemodelleerd worden. Dit is niet erg, aangezien de source code van een tool uit deze categorie vaak maar één of twee a4-tjes beslaat en overzichtelijk genoeg is.

Bij ontwikkelen van het tool data flow diagram en het proof of concept lag de focus op Java en Eclipse. Er zijn geen redenen om het tool data flow diagram niet bij een andere ontwikkelomgeving in te zetten, mits deze omgeving object georiënteerd is. Zo is het bijvoorbeeld mogelijk om dezelfde functionaliteit als nu aan Eclipse is toegevoegd aan de Microsoft Visual Studio toe te voegen, zodat het ingezet kan worden met C++, C# en alle andere .NET programmeertalen.

7.2 Aanbevelingen

Het is aan te bevelen om bij nieuwe projectspecifieke tools gebruik te maken van de ontwikkelde Eclipse plugin. Hiervoor zal de plugin hier en daar nog wel aangepast moeten worden. Zo zal er voor gezorgd moeten worden dat ook vanuit het Ecore diagram, waar interne data objecten mee gemodelleerd worden, source code gegenereerd kan worden, wanneer de namen van objecten in het tool data flow diagram wijzigen dit in de gegenereerde source code meeverandert en dat meer directorystructuren van alle gegenereerde code aan te passen is. Dit zal de plugin bruikbaar maken voor alle Java ontwikkelaars.

De ontwikkelde Eclipse plugin is, zoals is aangetoond, gemakkelijk in gebruik. Dit is echter wel nadat er uitleg is gegeven over de volgorde dat deze gebruikt dient te worden. Er moet namelijk eerst een project aangemaakt worden, dan een tool data flow diagram worden toegevoegd. Het diagram moet gemodelleerd worden en hieruit wordt dan source code gegenereerd bij het opslaan van het diagram, of wanneer er met de rechtermuisknop in het diagram geklikt wordt en voor "generate source" gekozen wordt. Deze source code kan nu volledig geïmplementeerd worden. Elke wijziging dient eerst in het tool data flow diagram te worden doorgevoerd. Hierna kan hier weer source code van gehergenereerd worden en pas dan hoort de source code aangepast te worden. Het is aan te bevelen om behalve een schriftelijke handleiding ook cheat sheets te maken. Zo kan een ontwikkelaar door een soort van afvinklijst een projectspecifieke tool ontwikkelen, zonder dat hij er een handleiding voor hoeft te lezen. Doordat aan elk punt in de cheat sheet acties gekoppeld kunnen worden, kunnen bepaalde taken zoals het aanmaken van een nieuw project, het toevoegen van het tool data flow diagram en het genereren van source code met één druk op de knop uitgevoerd worden.

Veel ontwikkelaars van Info Support werken op het kantoor van de klant waarvoor ze met het project bezig zijn. Zij maken wel regelmatig gebruik van de Endeavour software ontwikkelstraat, maar zijn weinig betrokken bij het ontwikkelen ervan. Zo wordt er te weinig nagedacht of een projectspecifieke tool misschien niet breder, in andere projecten ingezet kan worden en dus eventueel beschikbaar gesteld zou

moeten worden via de software ontwikkelstraat. Het is aan te bevelen om in zowel de cheat sheets zoals hierboven vermeld een punt op te nemen om de ontwikkelaar eraan te herinneren dat de tool misschien wel nuttig zou kunnen zijn om op te nemen in de software ontwikkelstraat. Verder dient het PDC actiever met de ontwikkelaars van Info Support te communiceren over welke tools beschikbaar zijn en dat het mogelijk is om projectspecifieke tools uit de projecten te adopteren.

Het is aan te bevelen om een optie aan het tool data flow diagram toe te voegen waarmee de diagrammen en de source code gemakkelijk naar het PDC gemaild kan worden. Bij deze optie moet er goed worden nagedacht hoe het diagram en de source code gecensureerd kan worden, zodat er eventuele bedrijfsinformatie eruit gefilterd wordt alvorens het opgestuurd wordt.

Interne data objecten worden als class diagram gemodelleerd en hieruit wordt code gegenereerd. Hier wordt dus eigenlijk MDA toegepast. We gebruiken hiervoor het Eclipse Modeling Framework [EMF]. Dit framework bleek simpel in gebruik en goede resultaten op te leveren. Daarom kunnen we dan ook aanraden om dit framework ook tijdens het ontwikkelen van projecten in te zetten en niet alleen tijdens het ontwikkelen van projectspecifieke tools. Het framework is vooral nuttig om een datastructuur te modelleren en hiervoor code te genereren.

De frontend wordt niet gemodelleerd. Bij elk tool data flow diagram wordt echter wel een command line interface frontend en een Maven MOJO frontend gegenereerd. Hierbij wordt wel code gegenereerd om de processen uit te voeren, maar deze staan niet per definitie in de goede volgorde, omdat de executievolgorde van de processen niet uit het diagram uit te lezen is. We verwachten dat het modelleren van deze executievolgorde, het genereren van code hiervoor en het handmatig toevoegen van alle andere code voor de frontend zelf, zoals bijvoorbeeld code voor een grafische user interface geen significante tijdwinst oplevert ten opzichte van het geheel handmatig ontwikkelen van de frontend. Een nieuw onderzoek zou hier meer inzicht in kunnen geven. Eventueel kan er dan gekeken worden of het inzetten van een state machine diagram of een sequence diagram hiervoor nuttig kan zijn.

We kunnen voorstellen dat sommige gemodelleerde processen in het tool data flow diagram vaak voor komen. Hierbij moet gedacht worden aan processen die een XML-bestand inlezen of wegschrijven, een databaseverbinding maken, interne dataobjecten (de)serialiseren etc. Hiervoor zou een centrale repository voor opgezet kunnen worden, zodat deze processen als “template” aan het diagram toegevoegd kunnen worden en nog minder code handmatig ontwikkeld hoeft te worden. Hierbij zal goed nagedacht moeten worden over hoe dit te centraliseren is, wie dit gaat onderhouden, wie er wel of niet iets aan de repository toe mag voegen en hoe alles te beveiligen is. Deze laatste punten vielen buiten de scope van dit onderzoek en kan in een volgend onderzoek kan duidelijk maken hoe dit te realiseren is en of dit een toegevoegde waarde is.

Referenties

Code	Bron
ANT	Apache Ant, http://ant.apache.org/
ECLIPSE	Eclipse, http://www.eclipse.org/
EMF	Eclipse Modeling Framework (EMF), http://www.Eclipse.org/modeling/emf/
END	Endeavour Software Ontwikkelstraat, http://www.infosupport.nl/Softwareontwikkelstraat
FM	freshmeat.net, Browse project tree: Software Development, http://www.infosupport.nl/Softwareontwikkelstraat
GEF	Graphical Editing Framework (GEF), http://www.Eclipse.org/gef/
GMF	Graphical Modeling Framework (GMF), http://www.Eclipse.org/gmf/
GSSF	Jack Greenfield en Keith Short, Software Factories, Assembling Applications with Patterns, Models, Frameworks and Tools, http://www.lore.ua.ac.be/Teaching/SSPEC2LIC/ SoftwareFactoriesOopsla.pdf
GVE	Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer, Generation of Visual Editors as Eclipse Plugins, Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp. 134-143
IEEE-12207	IEEE Computer Society/Software & Systems Engineering Standards Committee, Standard for Information Technology - Software Life Cycle Processes, 1996
KRAN	Kees Kranenburg, Een Software Ontwikkelstraat Inrichten, Gepubliceerd in "Informatie", mei 2002
KWALOS	Professional Development Center, Info Support, Kwaliteitseisen Ontwikkelstraat, versie 1.2, 2004
MDA	Object Management Group, Model Driven Architecture (MDA), http://www.omg.org/mda/
MDACP	Alan W. Brown, Model Driven Architecture: Concepts and Practice, http://www.haifa.ibm.com/Workshops/ple2004/papers/mda-concepts-practice-v5.pdf
MDDP	Markus Völter en Jorn Bettin, Patterns for Model-Driven Software-Development, Version 1.4, 2004, http://www.voelter.de/data/pub/MDDPatterns.pdf
MDE	Douglas. C. Schmidt, Model-Driven Engineering, Computer, vol. 39, no. 2, pp. 25-31, Feb., 2006
MSBUILD	Microsoft MSBuild, http://msdn2.microsoft.com/en-us/library/wea2sca5.aspx
OCL	Object Constraint Language, http://www.omg.org/technology/documents/ modeling_spec_catalog.htm#OCL

PRESS	Roger S. Pressman, Software Engineering, A practitioner's Approach, Fourth Edition, ISBN 0-07-709-411-5, 1997
RTSE	Nenad Medvidovic, Alexander Egyed en David S. Rosenblum, Round-Trip Software Engineering Using UML: From Architecture to Design and Back, Gepubliceerd in Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR), France, September 1999, pp. 1-8
RUP	Rational Unified Process, IBM, http://www-306.ibm.com/software/awdtools/rup/
SF	SourceForge.net, Software Map: Software Development, http://sourceforge.net/softwaremap/trove_list.php?form_cat=45
SSE	Ian Sommerville, Software Engineering, Pearson Education, 2006, ISBN 0321313798
TIGER	Tiger Project, http://tfs.cs.tu-berlin.de/tigerprj/
TISE	Anthony I. Wasserman, Software Engineering: Proceedings, Lecture Notes in Computer Science 467, Springer-Verlag, pp. 137-149
TOPCASED	TOPCASED Ecore Editor, http://topcased-mm.gforge.enseeiht.fr
UML	Object Management Group, Unified Modeling Language (UML), http://www.uml.org/
UML2T	UML2 Tools, http://www.eclipse.org/modeling/mdt/?project=uml2tools
VP	Visual Paradigm, http://www.visual-paradigm.com

A Vragenlijst Inventarisatie Projectspecifieke Tools

1. **Geef een beschrijving van een projectspecifiek tool, waarmee je in je project in aanraking bent gekomen.**
2. **In welke functiecategorie zou je de tool indelen?**
   (Kies een functiecategorie zoals eerder beschreven. Mocht dit niet toerijkend zijn, kunnen er hier opmerkingen over gemaakt worden)
3. **Wie heeft het initiatief genomen om de tool te gaan ontwikkelen en wat is zijn/haar rol in het project?**
   (Het is niet noodzakelijk om namen te noemen. Beschrijf eventueel de relatie tot andere projectleden)
4. **Wie heeft de requirements voor de tool opgesteld en wat is zijn/haar rol in het project?**
   (Het is niet noodzakelijk om namen te noemen. Beschrijf eventueel de relatie tot andere projectleden)
5. **Wie heeft de tool geïmplementeerd en wat is zijn/haar rol in het project?**
   (Het is niet noodzakelijk om namen te noemen. Beschrijf eventueel de relatie tot andere projectleden)
6. **Wie neemt het testen van de tool voor zijn/haar rekening en wie beslist dat de tool klaar is om in de productieomgeving te gebruiken? Geef de rollen in het project aan.**
   (Het is niet noodzakelijk om namen te noemen. Beschrijf eventueel de relatie tot andere projectleden)
7. **Wie zijn de gebruikers van de tool? Geef per gebruiker aan wat de rollen in het project zijn.**
   (Het is niet noodzakelijk om namen te noemen. Beschrijf eventueel de relatie tot andere projectleden)
8. **Wie beheert de tool en wat is zijn/haar rol in het project?**
   (Het is niet noodzakelijk om namen te noemen. Beschrijf eventueel de relatie tot andere projectleden)
9. **Wat is de invoer en uitvoer van de tool?**
   (Beschrijf met wat voor soort gegevens, processtappen en/of functionaliteiten de tool omgaat. Bijvoorbeeld voor een code generator: wat voor (meta)informatie dient als input en welke (programma)taal heeft de uitvoer?)
10. **Welke technologie/technologieën zijn gebruikt om de tool te ontwikkelen?**
    (Voorbeelden: Java, XML, Database, Eclipse RCP, Windows batch-bestand, Microsoft Excel, XSLT, ...)
11. **Beschrijf het ontwikkelproces en welke fasen hierin te onderscheiden zijn.**
12. **Beschrijf het test-proces.**
    (Probeert de ontwikkelaar of het werkt, of worden er unit-testen geschreven en uitgevoerd? Worden de resultaten door iemand anders geverifieerd?)
13. **Hoeveel documentatie is er beschikbaar?**
    (Denk hierbij aan ontwerp, commentaar in code, installatie- en gebruikshandleidingen, ...)
14. **Wie weten van het bestaan van de tool af?**
    (Zijn alleen sommige/alle projectleden op de hoogte van het bestaan van de tool of ook daarbuiten, zoals collega's in andere projecten of in de overkoepelende organisatie?)
15. **Hoe staat het nu met de tool en wat is de toekomst ervan?**
    (Wordt de tool daadwerkelijk gebruikt? Is de tool zo goed dat het ook in andere projecten gebruikt zou kunnen worden? Zou dit veel werk met zich meebrengen?)
16. **Wat zijn (achteraf) de voordelen van de tool?**
17. **Wat zijn (achteraf) de nadelen van de tool?**
18. **Heb je eventueel nog opmerkingen of tips over de tool, projectspecifieke tools in het algemeen of over mijn onderzoek?**
19. **Wordt er gebruik gemaakt van een versiebeheersysteem tijdens het ontwikkelen van de tool?**
20. **Generiekheid? Misschien te generiek? Moment om tool richting ontwikkelstraat/product te 'duwen' goed gekozen?**
21. **Hoe afhankelijk is het project van de tool?**
22. **Zou de tool later in het build proces gehangen kunnen worden? Zonee, zou hier eigenlijk wel behoefte aan zijn?**
23. **Is de User Interface loosly coupled? Zou het mogelijk zijn om de tool om te bouwen, zodat deze in het build proces gehangen kan worden?**

B Feature list Eclipse Plugin

In deze feature list worden de volgende termen gebruikt:
Plugin: de tool waarmee tools op een gemakkelijke en visuele wijze ontwikkeld kunnen worden
Tool: een tool die ontwikkeld is met behulp van de plugin

Plugin

P-1	De plugin wordt geschreven voor Eclipse 3.3.	Must have
P-2	De plugin is een grafische editor voor een Tool Dataflow diagram (TDF diagram).	Must have
P-3	Als er vanuit het context menu van het TDF diagram (zowel in de grafische editor als van het bestand in de Package Explorer) voor "generate code" gekozen wordt, zal er voor alle nodes code gegenereerd/geüpdate worden.	Must have
P-4	De gegenereerde code moet een zo min mogelijk dependencies hebben	Must have
P-5	De tool moet gemakkelijk als Maven build task uit te voeren zijn.	Must have
P-6	De tool moet standalone kunnen draaien. De plugin is een hulpmiddel tijdens de ontwikkeling, de tool is er niet afhankelijk van tijdens de uitvoering.	Must have
P-7	Door middel van een wizard in File > New… > Project... kan een nieuw project worden aangemaakt, waarbij de directory-structuur en het model gemakkelijk aangemaakt kunnen worden en een aantal variabelen gezet kunnen worden.	Should have
P-8	Vanuit het TDF diagram en de achterliggende code moet een rapportage gegenereerd kunnen worden, welke voor documentatie gebruikt kan worden en/of opgestuurd naar het PDC kan worden.	Should have
P-9	Als het diagram opgeslagen wordt, zal er voor alle nodes code gegenereerd/geüpdate worden.	Could have
P-10	Alle nodes (behalve external data nodes en note nodes) hebben een context menu, waarmee alleen code behorende bij deze node gegenereerd/geüpdate wordt.	Could have
P-11	Er moet een Unit-test-suite gegenereerd worden, zodat unit tests gemakkelijk te implementeren zijn.	Could have
P-12	Opgeslagen TDF diagrammen moeten in Subversion op te slaan zijn.	Must have

Internal Data nodes

ID-1	Voor alle internal data nodes wordt een Ecore Model, een Ecore Diagram (grafische weergave van het Ecore Model) en een EMF genmodel gegenereerd.	Must have
ID-2	Het Ecore Diagram kan worden opgeroepen door op een internal data node te dubbelklikken.	Must have
ID-3	Alle internal nodes dienen een toplevel class te hebben, waarvan een instantie als singleton binnen de tool draait. Deze toplevel class wordt gegenereerd.	Must have
ID-4	De naam van de data nodes kan later aangepast (refactored) worden.	Should have
ID-5	Vanuit een Ecore Diagram kan standaard geen code gegenereerd worden, hiervoor moet een toolbar-button/menu/view-met-button beschikbaar zijn.	Could have
ID-6	Internal Data nodes kunnen gekopieerd en geplakt worden.	Would have

Process Interface nodes

Pint-1	Voor elke geassocieerde internal data node (zowel in- als uitgaand) wordt een variabele aangemaakt, zodat deze direct beschikbaar zijn in de implementations.	Must have

Process Implementation nodes

Pimpl-1	Een process implementation is een Runnable, waardoor er meerdere processen parallel uitgevoerd kunnen worden als thread en ze niet in de GUI-thread draaien.	Must have
Pimpl-2	Process implementations die langer dan 2 seconden nodig hebben om uitgevoerd te worden dienen hun voortgang door te geven. De gegenereerde code moet hierop voorbereid zijn.	Must have
Pimpl-3	Voor alle geassocieerde external data nodes wordt een stukje commentaar gegenereerd als documentatie.	Must have
Pimpl-4	Een codevoorbeeld voor het doorgeven van de voortgang wordt als commentaar binnen het process gegenereerd.	Could have
Pimpl-5	Process implementation nodes kunnen gekopieerd en geplakt worden.	Would have

User Interface

UI-1	Naar keuze in de New…-wizard worden er classes voor een commandline interface, een visuele interface (Eclipse RCP project) en/of een Maven build task gegenereerd, waarin de processen als codevoorbeeld aangeroepen worden.	Should have

Cheat sheets

CS-1	Nieuwe tool maken.	Could have
CS-2	Bestaande tool aanpassen.	Could have

C Technisch Ontwerp Tool Data Flow Plugin

De tool data flow plugin is een plugin voor Eclipse 3.3 (Europa) en bestaat uit twee onderdelen: Op de eerste plaats een grafische editor waarmee een tool data flow diagram gemodelleerd kan worden. Verder bevat de plugin een Java code generator om vanuit het diagram Java code te genereren.

Het primaire doel van dit technisch ontwerp is uitleg geven over de opbouw en gebruikte technieken van de plugin. Aangezien de gebruikte technieken voor de meeste medewerkers van Info Support nieuw zullen zijn, wordt er dieper op ingegaan en zal er uitgelegd worden wat ze inhouden en in welke context geplaatst worden. Dit document kan daarom ook als developers manual gebruikt worden.

In sectie C.1 staat het technisch ontwerp van de grafische editor. Hierna is het technisch ontwerp van de code generator in sectie C.2 te vinden. Sectie C.3 vertelt hoe de grafische editor en de code generator samengevoegd zijn tot een Eclipse plugin. Hier is tevens meer informatie over de plugin zelf te vinden.

C.1 Grafische Editor

De Eclipse ontwikkelomgeving kent zogenaamde "editors" en "views". De bestanden waarin gewerkt wordt worden geopend door een editor. Deze editor staat over het algemeen op een centrale positie in de user interface. Views geven additionele informatie en/of de context van hetgeen in de editor geopend is of over het project waarin het geopende bestand zich bevind. Zo is er een view die alle bestanden in het project als boomstructuur weergeeft (met het geopende bestand geselecteerd), er is een view om de layout van het geopende bestand weer te geven en weer een andere view somt eigenschappen op van hetgeen in de editor geselecteerd is.

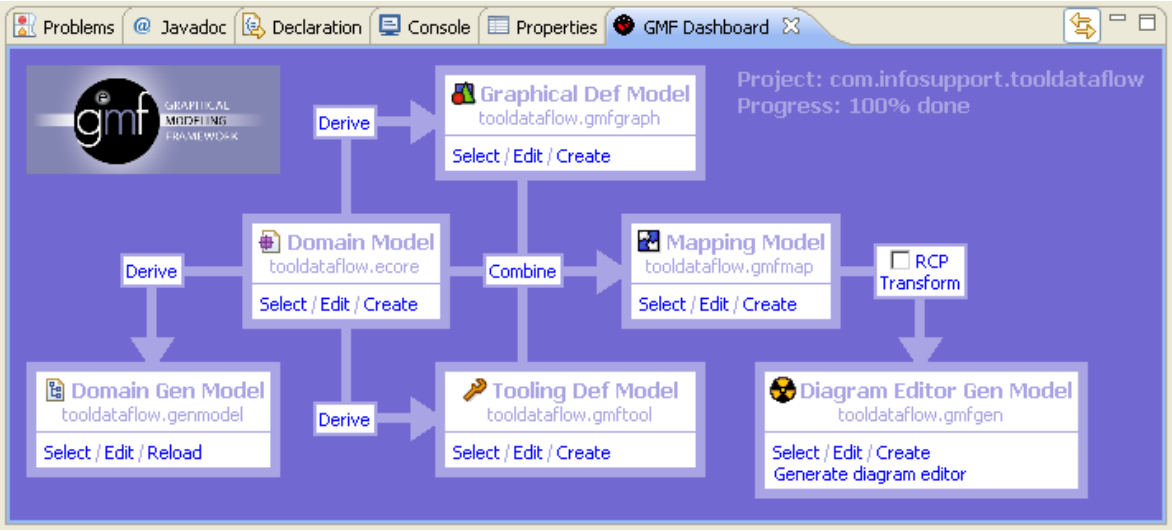
Behalve editors voor tekst-bestanden zoals bijvoorbeeld Plain Text-bestanden, Java-bestanden en XML-bestanden is het ook mogelijk om een grafische editor te gebruiken of zelf te ontwikkelen. Grafische editors worden voornamelijk gebruikt om een model of diagram visueel te kunnen bewerken.

Grafische editors voor Eclipse kunnen ontwikkeld worden met behulp van het Eclipse Graphical Editor Framework (GEF). GEF is zeer uitgebreid en biedt de ontwikkelaar veel mogelijkheden om een krachtige grafische editor te ontwikkelen. Het nadeel van GEF is dat de leercurve erg steil is en het de eerste keer veel tijd zal kosten om een grafische editor te ontwikkelen.

C.1.1 Graphical Modeling Framework (GMF)

Eclipse biedt buiten GEF ook het Graphical Modeling Framework (GMF). Hiermee kan een grafische editor gemodelleerd worden. Vanuit deze modellen wordt Java code gegenereerd, die de editor implementeert met behulp van GEF. Deze code is later eventueel te customizen als hetgeen gewenst is niet met de modellen gedefinieerd kan worden.

De grafische editor voor het tool data flow diagram wordt gemaakt door middel van een GMF project. Hierdoor krijgen we de beschikking over het zogenaamde "GMF Dashboard", wat een view voor Eclipse is en een roadmap geeft over welke modellen er nodig zijn om de grafische editor te ontwikkelen en in welke volgorde ze gemaakt dienen te worden. Dit laatste vanwege de afhankelijkheid. Figuur 24 geeft het GMF Dashboard weer.



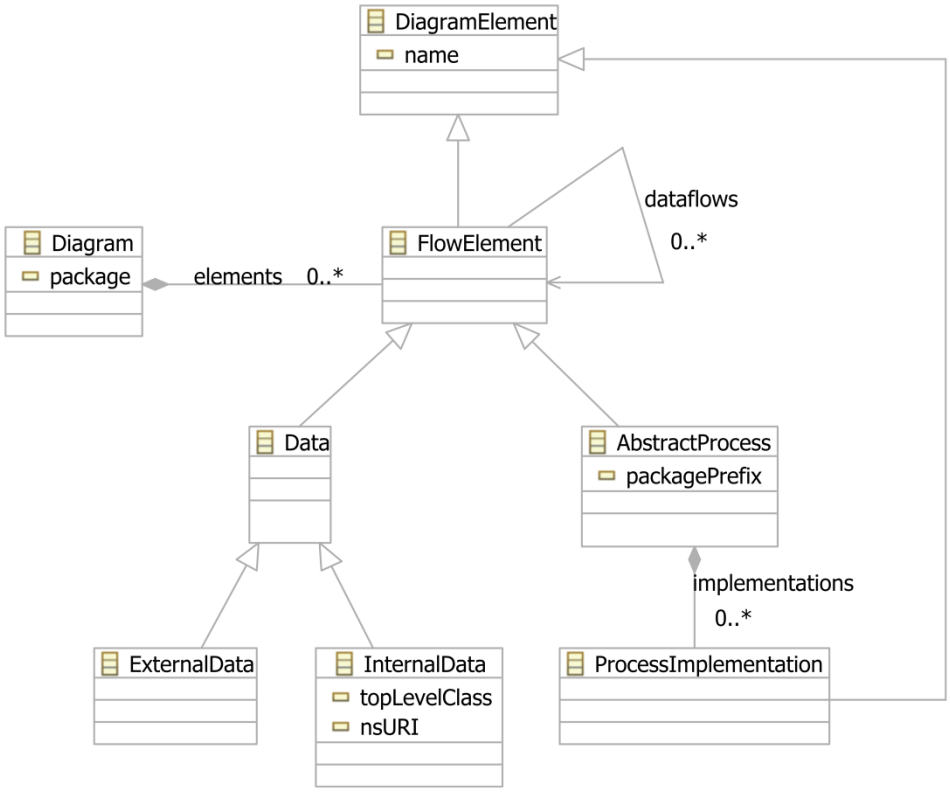
Figuur 24. GMF Dashboard

C.1.1.1 Domain Model (tooldataflow.ecore)

Het domain model definieert de structuur van de informatie achter diagram dat met de grafische editor gemodelleerd kan worden. Deze structuur wordt gemodelleerd als een Ecore-bestand: tooldataflow.ecore. Dit bestand moet gezien kan worden als een class diagram wat geheel op Eclipse en Java toegespitst is. De standaard-editor van een ecore-bestand geeft het class diagram weer als een boom-structuur.

Het Ecore diagram komt uit het Eclipse Modeling Framework (EMF). GMF maakt dan ook gebruik van EMF om de Java code voor het domain model te genereren.

Wanneer GMF geïnstalleerd is kan er een tweede bestand aangemaakt worden, wat een grafische view van het Ecore-bestand is. De grafische versie van het domain model is te vinden als “tooldataflow.ecore_diagram”. Figuur 25 geeft het domain model weer.



Figuur 25. Domain Model

Wanneer de grafische editor voor het tool data flow diagram gebruikt gaat worden zal er in elk diagram-bestand één object van de class "Diagram" bevatten. Dit object houdt referenties bij naar de elementen op het diagram.

We maken onderscheid in elementen van het diagram (class "Element") en elementen waartussen data flows gedefinieerd kunnen worden (class "FlowElement"). De classes "ExternalData", "InternalData" en "AbstractProcess" zijn afgeleidt van "FlowElement". "ExternalData" en "InternalData" leiden ook af van de class "Data". Dit laatste is om later te kunnen definiëren dat alleen data flows tussen "Data"-objecten en "AbstractProcess"-objecten geoorloofd zijn. Deze restrictie is niet in het domain model gedefinieerd, maar zal door de grafische editor afgedwongen worden.

C.1.1.2 Domain Gen Model (tooldataflow.genmodel)

Het maken van domein modellen en het werken met Ecore-bestanden heeft allemaal te maken met het Eclipse Modeling Framework (EMF). Dit framework voorziet erin om vanuit een grafische weergave van een model Java code te genereren, welke als implementatie van het model gebruikt kan worden.

Om Java code te genereren dient een domain gen model van het domain model afgeleidt te worden. Dit kan door in het GMF Dashboard op "derive" te klikken. Vanuit het domain gen model wordt Java code gegenereerd.

Door rechts te klikken in de editor van het domain gen model kan Java code gegenereerd worden. Model code is de implementatie van het model. Edit code geeft extra functionaliteit zoals allerlei providers voor koppeling tussen user interface en het model. Met generate editor code wordt een niet-grafische editor voor het model

gegenereerd. Generate test code levert een aantal test cases op, waar tests in geïmplementeerd kunnen worden. Er wordt geen enkele test gegenereerd.

Voor de tool data flow plugin zijn alleen model code en edit code nodig.

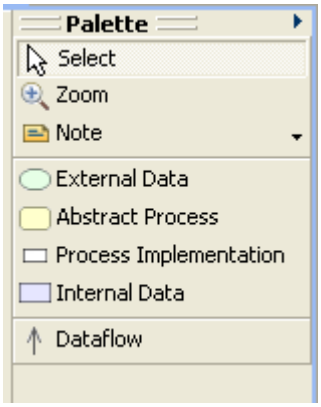
C.1.1.3 Graphical Def Model (tooldataflow.gmfgraph)

Het graphical def model definieert welke grafische elementen op wat voor manier op het diagram dienen te verschijnen. Een standaard-definitie kan derived worden. Deze definitie is een basis van waaruit gewerkt kan worden. Bijna alle elementen (nodes) moeten aangepast worden naar hoe ze eruit moeten zien.

C.1.1.4 Tooling Def Model (tooldataflow.gmftool)

In het tooling def model wordt het "Palette" van de grafische editor gedefinieerd (zie Figuur 26). De definitie kan afgeleidt worden van het domain model. Een paar kleine aanpassingen zijn nodig.

De pictogrammen bevinden zich niet in dit model. Zij bevinden zich namelijk in de "icons"-directory van de gegenereerde "edit code".



Figuur 26. Uiteindelijke Pallette

C.1.1.5 Mapping Model (tooldataflow.gmfmap)

Het mapping model combineert de informatie uit het domain model, het graphical def model en het tooling def model. Zo wordt er in dit mapping model aangegeven welk grafisch object (node) getekend moet worden bij welk item uit het "Palette". Verder koppelt het een class uit het domain model hieraan.

Het mapping model kan gegenereerd worden door op "combine" in het GMF Dashboard te klikken. Het is belangrijk dat het gegenereerde model gecontroleerd wordt, aangezien mijn ervaring is dat het niet altijd klopt. Omdat de process implementations in een compartment van een abstract process zitten moet hier wat handmatig werk gedaan worden.

Restricties kunnen in dit model opgelegd worden. De restrictie dat data flows alleen van abstract process naar een data element of van data element naar abstract process kan lopen wordt in dit model gedefinieerd. Dit wordt gedaan met behulp van Object Constraint Language (OCL).

C.1.1.6 Diagram Editor Gen Model (tooldataflow.gmfgen)

Het diagram editor gen model combineert alle eerdere modellen (al dan niet door ernaar te verwijzen). In dit model staat precies welke onderdelen en classes gegenereerd moeten worden om Java code van de grafische editor op te leveren. Deze Java code kan vanuit het GMF Dashboard gegenereerd worden.

C.1.2 Wijzigingen in Gegenereerde Code

Bij alle gegenereerde classes en functies staat een "@generated" annotatie in de Javadoc van die class of functie. Wanneer code aangepast wordt, wordt deze annotatie vervangen door "@generated NOT". Wanneer er opnieuw gegenereerd wordt zal de gewijzigde functie overgeslagen worden.

De gewijzigde functies ten opzichte van de gegereerde code kan altijd gevonden worden door in het gehele project te zoeken op de string "@generated NOT".

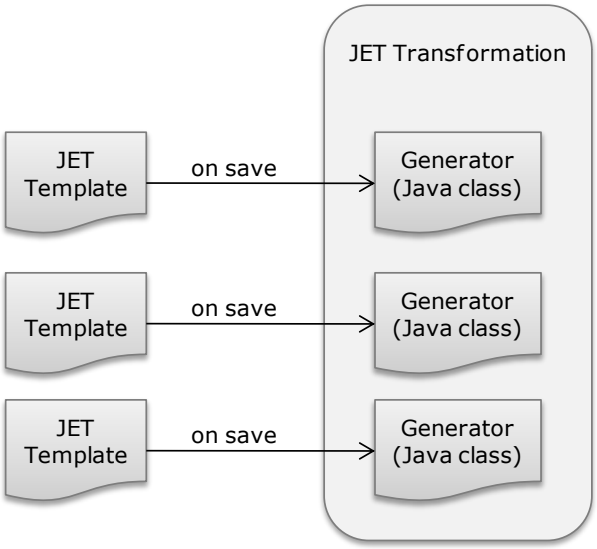
C.2 Code Generator

Met de grafische editor kan een tool data flow diagram gemodelleerd worden. Hierna kan vanuit dit diagram Java code gegenereerd worden, die als basis van de te ontwikkelen tool gebruikt kan worden.

Voor het genereren van Java code worden Java Emitter Templates (JET) gebruikt. JET is een onderdeel van Eclipse. De code generator is een apart project en wel een EMFT JET Transformation Project. In dit project is een map "templates" te vinden.

JET Templates zijn JSP-achtige templates, waarin de te genereren Java-class gedefinieerd wordt. Door de Java code worden JET-instructies verweven om onder andere class namen, imports en variabelen in te vullen.

Tijdens het ontwikkelen van dergelijke templates zal er elke keer wanneer een template opgeslagen wordt, vanuit de gedefinieerde template een Java class gegenereerd worden, waarin de generator zoals gedefinieerd in de template geïmplementeerd wordt. Deze gegenereerde classes samen leveren de uiteindelijke code generator (JET Tranformation) op. Figuur 27 geeft dit weer.



Figuur 27. Ontwikkelen van een JET Transformation

C.2.1 Generator Aanroepen Vanuit Grafische Editor

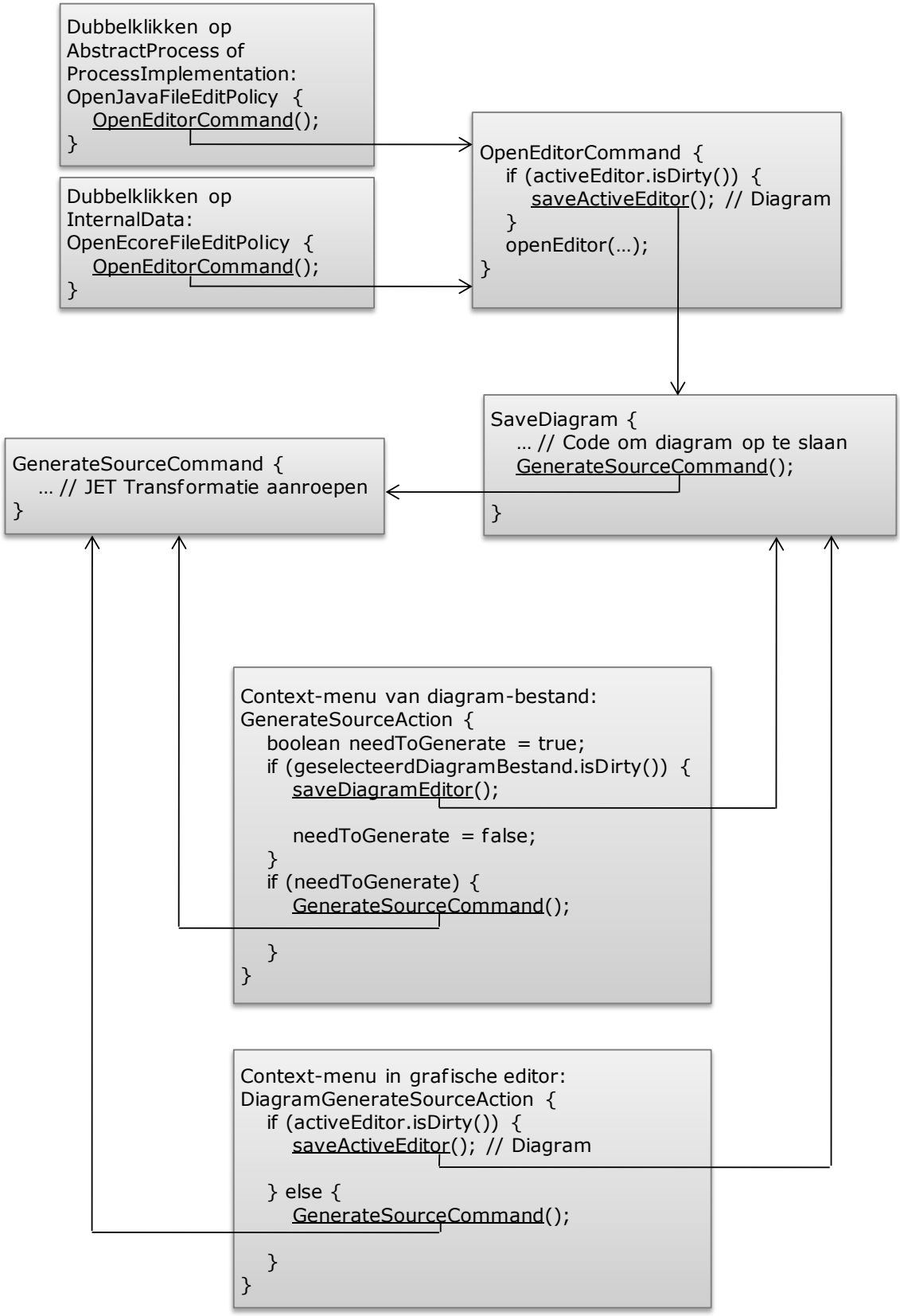
De code generator (in de vorm van een JET Transformation) heeft als input een instantie van het domain model van de grafische editor. Als output zal het de juiste Java classes toevoegen aan het project waar het diagram zich in bevindt.

Om ervoor te zorgen dat model en achterliggende Java code continu in sync zijn wordt de generator elke keer gestart als het model opgeslagen wordt.

Wanneer er gedubbelklikt wordt op een element van het diagram wordt het bijbehorende bestand geopend. Om er zeker van te zijn dat het bestand bestaat wordt de code generator aangeroepen alvorens het bestand geopend wordt.

Vanuit het context-menu van het diagram en vanuit de context-menu van het diagram-bestand kan de code generator handmatig aangeroepen worden.

Figuur 28 geeft weer wanneer er de generator aangeroepen wordt en wanneer het tool data flow diagram opgeslagen wordt. De pijlen moeten gezien worden als functie-call. In sommige gevallen is de pijl geen directe call, maar bestaat de aanroep doordat een event opgeworpen wordt.



Figuur 28. Pseudo-code over wanneer code gegenereerd wordt en wanneer het diagram opgeslagen wordt.

C.3 Eclipse Plugin

De Eclipse plugin bestaat uit de volgende vier projecten:

com.infosupport.tooldataflow Dit is een GMF project. Hierin staan alle modellen (models directory) en de source code van het domain model.

com.infosupport.tooldataflow.edit Bij het genereren van het domainmodel door EMF wordt edit-code gegenereerd. Alle source code in dit project is dus gegenereerd. De grafische editor maakt gebruik van deze code om wijzigingen in het domain model door te voeren. Zaken zoals undo-functionaliteit zitten in dit project. De icoontjes die in het pallette en de project explorer getoond worden bevinden zich ook in dit project.

com.infosupport.tooldataflow.diagram Vanuit de modellen in het GMF project wordt dit project gegenereerd. Dit is de bron code van de grafische editor. Deze code is hier en daar aangepast (zoals aanroepen code generator). Alle custom code is te vinden door in het project naar "@generated NOT" te zoeken.

com.infosupport.tooldataflow.codegen Dit is een JET project. Het gehele project is een "JET tranfsormatie". In de directory "templates" staan de templates voor de te genereren code.

C.3.1 Dependencies Eclipse Plugin (Compile-time)

Om de Eclipse plugin te compilen en/of te gebruiken dienen eerst GMF en JET geïnstalleerd te worden. Via het menu "Help" > "Software Updates" > "Find and Install…" kunnen beide dependencies worden toegevoegd aan de Eclipse 3.3 installatie. Zowel GMF als JET zijn open-source en worden ontwikkeld en beheerd door de Eclipse Foundation. Ze zijn daarom te vinden op de "Europa Discovery Site" in het "Install" scherm.

Beide dependencies vallen onder de categorie "Models and Model Development":

- Graphical Modeling Framework (Europa Edition)
- Java Emitter Templates (JET) (Incubation)

Deze twee dependencies hebben zelf ook een aantal dependencies, die automatisch geselecteerd kunnen worden door op "Select Required" te klikken.

Verder dient de eigen gemaakte Eclipse plugin te worden geïnstalleerd. Dat kan door de vier JAR-files naar de "plugins" directory van Eclipse te kopiëren.

C.3.2 Dependencies Tool Data Flow Project (Runtime)

Tools die ontwikkeld worden aan de hand van een tool data flow diagram kunnen gebuild worden met Maven2. De volgende Maven dependencies moeten dan worden opgegeven:

- org.eclipse.emf.common
- org.eclipse.emf.ecore

Om gebruik te kunnen maken van de gegenereerde Maven MOJO frontend dient ook "org.apache.maven-api" als dependency toegevoegd te worden.

Runtime is de plugin die het tool data flow diagram toevoegt aan Eclipse niet nodig. Deze kunnen gemakkelijk aan Maven2 worden toegevoegd. Wanneer het project voor de eerste keer gebuild wordt zal Maven2 het juiste commando weergeven. De benodigde bestanden zijn in de "plugins" directory van Eclipse te vinden.