

Inhoudsopgave

| | |
|--|----|
| Dankwoord | 6 |
| Hoofdstuk 1 Inleiding..... | 7 |
| 1.1 Situering..... | 7 |
| 1.2 Doelstelling | 7 |
| 1.3 E2S..... | 8 |
| 1.4 Inhoud van de tekst | 8 |
| Hoofdstuk 2 Model Driven Architecture..... | 10 |
| 2.1 Traditionele softwareontwikkeling | 10 |
| 2.1.1 Documentatie..... | 12 |
| 2.1.2 Portabiliteit | 13 |
| 2.1.3 Systeeminteractie..... | 14 |
| 2.2 MDA | 14 |
| 2.2.1 Platform Independent Model | 15 |
| 2.2.2 Platform Specific Model..... | 16 |
| 2.2.3 Code..... | 17 |
| 2.2.4 Voordelen van MDA | 18 |
| Documentatie..... | 18 |
| Portabiliteit | 19 |
| Systeeminteractie..... | 19 |
| Hoofdstuk 3 Standaarden hergebruikt voor MDA | 21 |
| 3.1 Meta Object Facility..... | 21 |
| 3.2 Unified Modeling Language | 22 |
| 3.3 Object Constraint Language..... | 24 |
| 3.3.1 Beperkingen..... | 24 |
| 3.3.2 Invarianten | 25 |
| 3.3.3 Pre- en postcondities..... | 25 |
| 3.3.4 Overige | 26 |
| 3.3.5 Nieuw in OCL 2.0..... | 26 |
| 3.4 Andere technologieën | 26 |
| 3.4.1 XML Metadata Interchange..... | 26 |
| 3.4.2 CWM | 27 |
| Hoofdstuk 4 HOORA | 28 |
| 4.1 Overzicht..... | 28 |
| 4.2 Codegeneratie..... | 32 |
| 4.2.1 Declaratiebestand..... | 33 |
| 4.2.2 Implementatiebestand | 34 |

| | | |
|-------------|--|----|
| 4.2.3 | Constantendeclaraties | 34 |
| 4.2.4 | Pascalbestand..... | 35 |
| 4.2.5 | Codegeneratiepatronen | 36 |
| Hoofdstuk 5 | MDA Transformatiestandaard..... | 38 |
| 5.1 | Query/View/Transformation..... | 39 |
| 5.1.1 | Verband met bestaande specificaties van OMG..... | 39 |
| 5.1.2 | Functionele vereisten..... | 39 |
| 5.1.3 | Non-functionele vereisten..... | 40 |
| 5.1.4 | Evaluatiecriteria..... | 41 |
| 5.2 | RFP voorstellen..... | 41 |
| 5.2.1 | Kennedy Carter..... | 41 |
| | Inhoud van het voorstel | 42 |
| | Evaluatie..... | 42 |
| 5.2.2 | XMOF..... | 43 |
| | Inhoud van het voorstel | 43 |
| | Evaluatie..... | 43 |
| 5.2.3 | Interactive Objects..... | 44 |
| | Inhoud van het voorstel | 44 |
| | Evaluatie..... | 45 |
| 5.2.4 | QVT-Merge Group 1.0..... | 46 |
| | Inhoud van het voorstel | 46 |
| | Evaluatie..... | 47 |
| 5.2.5 | QVT-Merge Group 2.0..... | 48 |
| | Inhoud van het voorstel | 48 |
| | Evaluatie..... | 50 |
| 5.2.6 | Overige | 51 |
| 5.3 | Besluit | 51 |
| Hoofdstuk 6 | Implementatie van transformatieregels..... | 52 |
| 6.1 | Algemeen | 52 |
| 6.1.1 | Hulpmiddelen | 52 |
| 6.1.2 | Transformatieregels..... | 53 |
| 6.2 | ATL..... | 57 |
| 6.2.1 | Overzicht | 58 |
| 6.2.2 | Verhouding QVT-ATL..... | 59 |
| 6.2.3 | Transformatie van PIM naar PSM..... | 59 |
| 6.2.4 | Transformatie van PSM naar code..... | 63 |
| | Pascalbestand..... | 66 |

| | |
|---|-----|
| Overige bestanden | 69 |
| 6.2.5 Integratie van ATL met HOORA | 71 |
| 6.3 MTF | 72 |
| 6.3.1 Overzicht | 72 |
| 6.3.2 Verhouding QVT-MTF | 72 |
| 6.3.3 Transformatie van PIM naar PSM | 72 |
| 6.3.4 Transformatie van PSM naar code..... | 75 |
| 6.3.5 Integratie van MTF met HOORA | 75 |
| Hoofdstuk 7 Conclusie | 76 |
| 7.1 Literatuurstudie | 76 |
| 7.2 Proefimplementatie van transformatieregels..... | 76 |
| 7.3 Moeilijkheden en uitdagingen..... | 77 |
| Appendix A: Code gegenereerd door HOORA | 78 |
| Appendix B: ATL syntax | 86 |
| ATL modules | 86 |
| Hoofding | 86 |
| Importeersectie | 86 |
| Helperfuncties | 86 |
| Transformatieregels | 87 |
| Declaratief | 87 |
| Soorten gecorreleerde transformatieregels..... | 88 |
| Overerving | 88 |
| Imperatief | 89 |
| Appendix C : ATL-implementatie..... | 90 |
| Appendix D: Code gegenereerd door ATL | 107 |
| Appendix E: MTF syntax | 110 |
| Relatie | 110 |
| Afbeelding | 110 |
| Zoekopdracht | 110 |
| Conditie | 110 |
| Gelijkheidscondities | 110 |
| Filtercondities | 111 |
| Samengestelde condities..... | 111 |
| Uitbreiding | 111 |
| Abstracte regels | 112 |
| Opspoorbaarheid..... | 112 |
| Reconciliatie | 112 |

| | |
|---|-----|
| Appendix F: MTF-implementatie..... | 113 |
| Appendix G: Verklarende Woordenlijst..... | 116 |
| Appendix H: Referenties | 117 |

Lijst van figuren

| | |
|--|----|
| Figuur 2.1: Evolutie van softwareontwikkeling | 11 |
| Figuur 2.2: Traditionele softwareontwikkelingcyclus..... | 12 |
| Figuur 2.3: Softwareontwikkelingcyclus m.b.v. MDA | 15 |
| Figuur 2.4: Internetgebaseerde ontbijtdienst | 16 |
| Figuur 2.5: PIM naar PSM transformatie | 17 |
| Figuur 2.6: Transformatiestappen | 18 |
| Figuur 2.7: Systeeminteractie via bridges | 20 |
| Figuur 4.1: HAT | 29 |
| Figuur 4.2: Plugable Objects | 30 |
| Figuur 4.3: E2S codegenerator | 31 |
| Figuur 4.4: E2S documentgenerator..... | 31 |
| Figuur 4.5: Fragment van het declaratiebestand van de Deur-unit..... | 33 |
| Figuur 4.6: Fragment van het declaratiebestand van de Raam-unit | 34 |
| Figuur 4.7: Fragment van het implementatiebestand van de Deur-unit | 34 |
| Figuur 4.8: Fragment van het bestand voor constantendeclaraties voor de Deur-unit | 35 |
| Figuur 4.9: Fragment van het Pascal-bestand voor de Deur-unit | 36 |
| Figuur 5.1: Relatie in QVT 1.0..... | 47 |
| Figuur 6.1: Transformatie van publiek naar privaat attribuut..... | 54 |
| Figuur 6.2: Transformatie van n-op-m associatie naar associatieklasse..... | 55 |
| Figuur 6.3: Transformatie van een associatieklasse naar een klasse | 56 |
| Figuur 6.4: Overzicht van het ATL uitvoeringsmechanisme | 58 |
| Figuur 6.5: Transformatieregel in ATL voor toevoegen van een constructor en destructor | 60 |
| Figuur 6.6: Transformatieregel in ATL voor publieke naar private attributen..... | 61 |
| Figuur 6.7: Transformatieregel in ATL voor n-op-m associatie naar associatieklasse | 62 |
| Figuur 6.8: Transformatieregel in ATL voor associatieklasse naar klasse..... | 63 |
| Figuur 6.9: Transformatie van PSM naar code met ATL..... | 65 |
| Figuur 6.10: Fragment van generatie van Pascalbestand met ATL..... | 66 |
| Figuur 6.11: Helperfunctie voor genereren van Pascalbestand met ATL..... | 66 |
| Figuur 6.12: Fragment van Pascalbestand gegenereerd met ATL..... | 66 |
| Figuur 6.13: Helperfunctie voor genereren van importeerclausule met ATL | 67 |
| Figuur 6.14: Helperfunctie voor genereren van declaratie voor Pascalbestand | 67 |
| Figuur 6.15: Fragment van Pascalbestand gegenereerd met ATL..... | 68 |
| Figuur 6.16: Helperfunctie vor genereren van implementatie van Pascalbestand met ATL | 68 |
| Figuur 6.17: Fragment van Pascalbestand gegenereerd met ATL..... | 69 |
| Figuur 6.18: Transformatieregel in MTF voor toevoegen van constructor en destructor en transformeren van publieke naar private attributen | 74 |

Dankwoord

Graag had ik enkele mensen bedankt die bij de realisatie van dit eindwerk belangrijker waren dan anderen. Allereerst wil ik graag mijn ouders bedanken voor hun voortdurende steun gedurende mijn volledige studiercarrière, waarvan dit eindwerk het kroonstuk vormt. Ook mijn vriendin Sofie wil ik om dezelfde reden bedanken en mijn huidige werkgever, Siemens, voor de flexibiliteit die ik van hen kreeg. Daarnaast ook een hartelijk woord van dank voor mijn thesisbegeleiders, Bert Vanhooff en Stefan Van Baelen, en mijn promotor prof. Dr. ir. Y. Berbers, zonder wie dit eindwerk onmogelijk was geweest.

Hoofdstuk 1

Inleiding

In dit eerste hoofdstuk worden ondermeer de situering, doelstelling en inhoud van dit eindwerk voorgesteld. Allereerst wordt in 1.1 de situering geschetst. Daarop volgend wordt in 1.2 de doelstelling geformuleerd. In 1.3 wordt het bedrijf E2S voorgesteld, waarmee werd samengewerkt om dit eindwerk te realiseren. Tenslotte wordt in 1.4 de inhoud van deze thesistekst behandeld.

1.1 Situering

In onze snel evoluerende wereld ontstaan voortdurend nieuwe technologieën, zoals programmeertalen, middlewaresystemen, besturingssystemen en hardware-platformen. Migreren naar deze nieuwe technologieën en, algemener, de nood aan het bijwerken van bestaande systemen en efficiënter ontwikkelen van grote systemen heeft de *Object Management Group* (OMG, het consortium achter de standaardisatie van ondermeer UML en CORBA) ertoe gebracht een nieuwe aanpak voor te stellen voor het ontwikkelen van software: *Model Driven Architecture* (MDA).

MDA moet een nieuw tijdperk inluiden voor ontwikkelen van software waarbij de ontwikkeling sterk gebaseerd wordt op modellen van de applicatie op verschillende niveaus van abstractie. Hierbij beoogt men de ontwikkeling van een systeem te beginnen met een model op hoog niveau, dat abstractie maakt van zo goed als alle technische en platformafhankelijke details. Naarmate de ontwikkeling vordert wordt dit model telkens getransformeerd naar een model op lager niveau, dat meer en meer van deze details bevat. Dit moet uiteindelijk resulteren in een volledig model dat alle nodige technologische details bevat. Dit model kan vervolgens met een laatste transformatie omgezet worden naar code in een bepaalde programmeertaal voor een bepaald platform, wat het werkende systeem oplevert.

De transformatie tussen deze modellen is een belangrijke sleutel in dit geheel. Voor de specificatie van modellen kunnen bestaande standaarden zoals UML hergebruikt worden. Transformaties worden, volgens het opzet van OMG, gespecificeerd als een set transformatieregels, geschreven in een bepaalde transformatietaal. De mogelijkheden van de taal om deze modellen naar elkaar te transformeren zullen echter de toepasbaarheid van MDA voor reële situaties bepalen.

1.2 Doelstelling

In de standaardisatie van MDA door OMG ontbreekt nog de omschrijving van een geschikte taal voor transformatieregels. Hierdoor kan MDA nog niet zonder meer in softwarewerktuigen geïmplementeerd worden. Het vinden van een geschikte transformatietaal wordt bestudeerd in dit eindwerk. Hiervoor worden huidige standaarden bestudeerd waar MDA en een transformatietaal van gebruik maken. Daarnaast wordt de bruikbaarheid van een aantal transformatietalen nagegaan en worden deze talen met elkaar vergeleken. Hieruit wordt een taal gekozen en een proefimplementatie van transformatieregels in deze taal uitgewerkt. Deze transformatieregels worden afgeleid uit de bestaande software ontwikkelingsnoden van E2S. Op deze manier wordt onderzocht hoe de huidige mogelijkheden van deze softwaresuite met MDA kunnen gerealiseerd worden. Tot slot wordt onderzocht hoe de transformatietaal met de applicatiesuite kan geïntegreerd worden.

De doelstelling van deze thesis kan als volgt samengevat worden:

- Een studie van de codegeneratiemechanismen van de huidige E2S software

- Het identificeren van codegeneratiepatronen hierin
- Aan de hand van een literatuurstudie een geschikte modeltransformatietaal vinden
- Het omvormen van de gevonden codegeneratiepatronen tot een proefimplementatie van transformatieregels in de gekozen transformatietaal
- Het onderzoeken van een integratie van de bestaande E2S software met de transformatietaal

Elk van deze punten worden verderop in dit proefschrift toegelicht.

1.3 E2S

Deze thesis werd gerealiseerd in samenwerking met E2S, een Belgisch software engineering bedrijf gesitueerd in Gent. Via een verscheidenheid van producten en diensten biedt het bedrijf software op maat aan een ruime waaier van klanten in veeleisende toepassingsdomeinen, ondermeer ESA (*European Space Agency*) en Barco. Hiervoor wordt ondermeer gebruik gemaakt van een eigen procesmodel genaamd HOORA (Hierarchical Object ORiented Analysis). De belangrijkste karakteristieken van deze methode zijn objectgerichtheid, een breed toepassingsdomein (inclusief *real time* applicaties of ware-tijd-systemen), ondersteuning voor hiërarchie, nadruk op dynamiek en een krachtig visiemechanisme (meerdere consistente visies op een model). Voor de ondersteuning van dit procesmodel heeft E2S een eigen softwaresuite ontwikkeld die verder toegelicht wordt in Hoofdstuk 4. Hiermee kan via UML-diagrammen software gemodelleerd worden op het niveau van een Platform Specific Model (zie hiervoor verder). Vervolgens kan hiervoor een raamwerk van code gegenereerd worden. Deze methodes voor codegeneratie zijn echter beperkt en weinig flexibel omdat ze hard in de software zijn ingebakken. Met de standaardisatie van MDA door OMG rees de vraag naar MDA als mogelijke oplossing voor deze moeilijkheden.

E2S werkt voor de ontwikkeling van hun softwaresuite eveneens samen met de Avionics-afdeling van Barco N.V. Deze afdeling produceert displaysystemen voor toepassing in militaire en burgerluchtvaart. Systeemgeïntegreerde programmatuur voor dit soort toepassingen, zoals door Barco geproduceerd wordt, is van het niveau van levenskritische software. Het spreekt voor zich dat dit ook voor codegenerators voor deze applicaties bijzondere eisen stelt. De correcte werking van gegenereerde code moet in ieder mogelijk uitvoeringsscenario op voorhand aantoonbaar zijn. Ook hiervoor opent MDA mogelijkheden doordat met MDA code altijd op een deterministische manier gegenereerd wordt.

1.4 Inhoud van de tekst

De inhoud van deze thesistekst is onderverdeeld in zeven hoofdstukken. Hoofdstuk 1, waarvan deze paragraaf het slot vormt, behandelt de situering, doelstelling en inhoud van de thesis en stelt het bedrijf E2S voor. In Hoofdstuk 2 wordt Model Driven Architecture nader verklaard, een nieuwe techniek voor het ontwikkelen van software gebaseerd op transformaties tussen modellen van een applicatie. Model Driven Architecture is deels gestandaardiseerd. De vastgelegde standaarden vormen een aantal bestaande technieken die voor de toepassing van Model Driven Architecture hergebruikt worden. Hoofdstuk 3 vormt een toelichting van de meest belangrijke onder deze standaarden. Niet gestandaardiseerd is de taal waarmee transformaties tussen modellen moeten gedefinieerd worden. Hoofdstuk 4 vormt een analyse van HOORA, een applicatiesuite ontwikkeld door E2S, eerder voorgesteld in Hoofdstuk 1. Voor HOORA is het bedrijf geïnteresseerd in de toepassing van MDA, maar ondervindt hierbij het probleem van het ontbreken van deze standaard voor een transformatietaal. Hiervoor zijn enkel de vereisten waaraan een dergelijke taal zou moeten voldoen gestandaardiseerd. Daarom worden in Hoofdstuk 5 deze gestandaardiseerde vereisten bestudeerd en een aantal talen die in mindere of meerdere mate hieraan voldoen. Als besluit van Hoofdstuk 5 wordt uit de bestudeerde talen een keuze gemaakt. In Hoofdstuk 6 wordt een proefimplementatie van transformatieregels uitgewerkt in de gekozen taal. Deze transformatieregels zijn gebaseerd op de analyse van HOORA uit Hoofdstuk 4. Daarnaast wordt onderzocht hoe deze taal in HOORA kan geïntegreerd worden. Ter vergelijking

wordt ook in de taal van tweede keuze een beperkte proefimplementatie uitgewerkt en worden beide talen met elkaar vergeleken. Tenslotte wordt in Hoofdstuk 7 een besluit geformuleerd over deze vergelijking, en wordt de beste keuze geformuleerd met vermelding van de ondervonden moeilijkheden en toekomstige uitdagingen voor MDA.

Hoofdstuk 2

Model Driven Architecture

In het tweede hoofdstuk worden een aantal inleidende zaken behandeld, die de situering van dit eindwerk uit Hoofdstuk 1 verder toelichten. Binnen traditionele softwareontwikkeling worden een aantal problemen ondervonden waarvoor op korte termijn niet onmiddellijk een oplossing uit de bus valt. Dit wordt behandeld in 2.1. Daarna wordt in 2.2 verklaard waarom en hoe MDA hiervoor mogelijk een oplossing biedt.

2.1 Traditionele softwareontwikkeling

De doelen die met hedendaagse software kunnen bereikt worden zijn enorm. We gebruiken software om rapporten te schrijven, informatie op te zoeken, contact te houden met kennissen, vluchten te boeken naar onze favoriete vakantiebestemming enzovoort. De mogelijkheden die gebruikerstoepassingen bieden stijgen voortdurend. Hierdoor verhoogt ook de complexiteit van deze toepassingen. Dit brengt een nood met zich mee aan nieuwe methoden om deze complexiteit te kunnen beheersen.

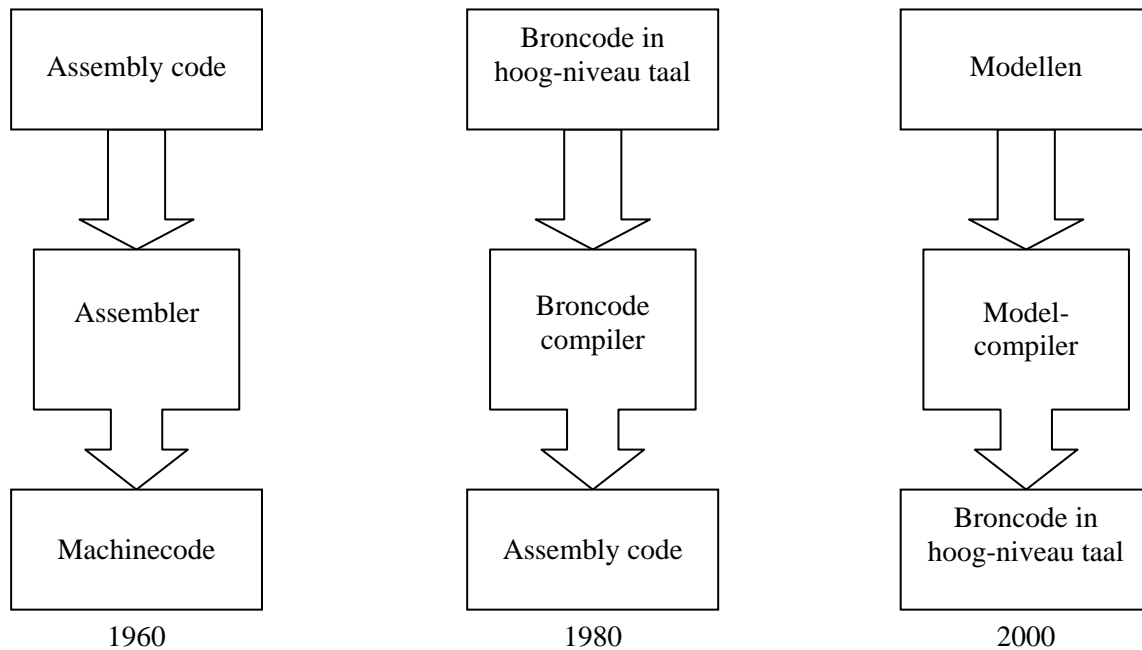
Halfweg de vorige eeuw schreven programmeurs de code voor hun programma's als een lange aaneenschakeling van machine-instructies. Data werd opgeslagen op rollen waarvan de rotatietijd moest in acht genomen worden zodat de kop van de machine de volgende instructie precies op het juiste moment kon lezen. Later kwamen er *assemblers* om de vervelende taak over te nemen van het genereren van rijen nullen en eentjes. Dit gebeurde vanaf een set sleutelwoorden ontworpen voor elk hardwareplatform.

Daarna ontstonden *programmeertalen*, zoals FORTRAN, en werden er standaarden gecreëerd voor COBOL en C zodat portabiliteit (overdraagbaarheid) tussen hardwareplatforms mogelijk werd. De software-industrie ontwikkelde technieken om programma's te structureren zodat deze gemakkelijker konden geschreven, begrepen en onderhouden worden. Tegenwoordig hebben we talen als C++, C#, Smalltalk, Eiffel en Java, elk met de notie van objectoriëntatie, een techniek om data samen met gedrag te structureren in klassen en objecten.

Naarmate men doorheen de tijd van één taal naar een volgende overging, werd over het algemeen het niveau van abstractie verhoogd waarop de softwareontwikkelaar werkt. Daardoor moest de programmeur een nieuwe, hoger-niveau taal leren die dan kon afgebeeld worden op een taal van lager niveau. Zo werd C++ bijvoorbeeld afgebeeld op C, C op *assembly code*, en *assembly code* vervolgens op machinecode en de hardware van het systeem. Een nieuwe laag van abstractie, zoals objectoriëntatie, werd eerst als idee ontwikkeld en bij bewezen nut geformaliseerd. Werktuigen als assemblers, preprocessors en compilers werden gebouwd om deze concepten te ondersteunen. Deze werktuigen automatiseerden het afbeelden van een laag op de volgende (lagere) laag.

Hierin kunnen we een patroon herkennen: we formaliseren onze kennis van een toepassing in een zo hoog mogelijke taal. Naarmate de tijd verstrijkt, leren we deze taal beter gebruiken en we ontwikkelen een set van conventies voor een beter gebruik van de taal. Deze conventies worden geformaliseerd op hoger niveau waarop een nieuwe taal ontstaat die automatisch afgebeeld wordt op de taal van lager niveau. Na een tijd wordt deze hogerniveau taal op haar beurt aanschouwd als lagerniveau, ontwikkelen we een set conventies voor een beter gebruik ervan, het proces herhaalt zich en er wordt een nieuwe taal geboren van nog hoger niveau.

Na onder meer objectoriëntatie en het toepassen van *design patterns* zou de volgende stap het modelgebaseerd ontwikkelen kunnen zijn. Objectoriëntatie groepeerde functionaliteit van ontwikkelde software in objecten en *design patterns* vormen een set van conventies voor hoe dit groeperen moet gebeuren. Modelgebaseerd ontwikkelen kan als volgende stap een hoger niveau vormen waarin we platformafhankelijke softwaremodellen bouwen. Dit betekent echter niet dat op de vorige stappen wordt teruggekomen. Een overzicht van deze evolutie, en een schematische voorstelling van gelaagde compilatie, wordt voorgesteld in Figuur 2.1.



Figuur 2.1: Evolutie van softwareontwikkeling

In bovenstaande figuur zien we de systematische ontwikkeling van hogerniveau talen door de geschiedenis. Waar in de '60-jaren software als *assembly* taal ontwikkeld werd en door een assembler op machine code werd afgebeeld, werd in de '80-jaren een hoog-niveau taal als C door een broncode compiler op *assembly* code afgebeeld. In de jaren 2000 zien we de volgende evolutie als het afbeelden van modellen door een modeltransformator op een software platform. Dit software platform is een geheel van werkende broncode.

Met de introductie van MDA hoopt men als volgende ontwikkeling te kunnen abstractie maken van broncode en bij het ontwikkelen van software enkel met modellen te werken. MDA kan zo aanzien worden als een nieuwe taal van hoger niveau. Zoals vermeld komt deze er altijd om het ontwikkelingsproces te versnellen en als oplossing voor een aantal problemen die met de vorige taal ondervonden worden. We kunnen ons dus afvragen wat de moeilijkheden zijn die programmeurs tegenwoordig ondervinden. Nader onderzoek, volgens [AKJWWB03], brengt voornamelijk drie categorieën van problemen aan het licht. Deze categorieën zijn:

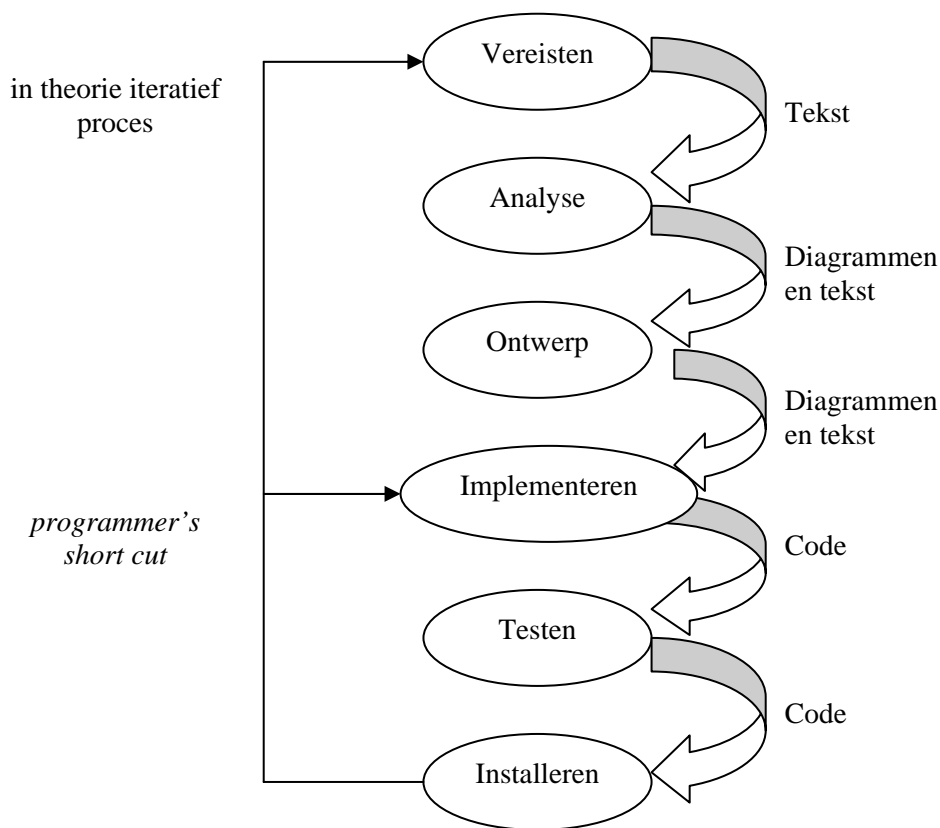
- Documentatie
- Portabiliteit
- Systeeminteractie

Hieronder wordt elk van deze categorieën verder toegelicht.

2.1.1 Documentatie

Het ontwikkelen van software zoals we dat vandaag kennen, wordt vooral gedreven door het produceren van code. Een typisch proces bestaat uit de volgende stappen:

1. Beschrijven van het concept en verzamelen van de vereisten
2. Analyse en functionele beschrijving
3. Ontwerp
4. Implementeren
5. Testen
6. Installeren (*deployment*)



Figuur 2.2: Traditionele softwareontwikkelingcyclus

In Figuur 2.2 wordt het model van het klassieke ontwikkelingsproces, waar ook vaak naar verwezen wordt als het watervalmodel, grafisch geïllustreerd. Stappen 1 tot 6 van het watervalmodel worden klassiek na elkaar gevolgd. Documentatie en informele diagrammen worden geproduceerd in stappen 1 tot 3. Dit houdt onder meer *use cases*, klassendiagrammen en interactiediagrammen in. De stapel papier die geproduceerd wordt is soms indrukwekkend.

Als de vereisten voor het project veranderen, wordt een volgende iteratie, m.a.w. het opnieuw doorlopen van alle stappen, gestart. In theorie zouden programmeurs op dat moment dus een vereistendocument (*use cases*) aanpassen, gevolgd door het opnieuw opstellen van een analyse die nodig is om deze vereisten te implementeren, gevolgd door een laagniveau ontwerp met behulp van klassendiagrammen, om dit te implementeren in code, deze code te testen en een nieuwe versie van het

systeem af te leveren. In de praktijk volgen programmeurs deze uitgebreide route echter niet en gaan meestal meteen over tot het aanpassen van de code (*programmer's short cut*, zie figuur).

Het probleem bestaat erin dat softwareontwikkelaars het schrijven van documentatie veelal beschouwen als “onproductief” en “bijzaak”. Als de code van een systeem verandert, worden documenten meestal pas na een bepaalde tijd bijgewerkt, omdat op het moment dat documenten gemaakt worden er geen code geschreven wordt en programmeurs onproductief lijken te zijn. Het verband tussen code en documentatie gaat daardoor verloren. Hierdoor wordt onderhoud van het systeem moeilijk tot bijna onmogelijk: voor een programmeur die met het bijwerken van een onbekend project belast wordt, is het (door het ontbreken van documentatie) als afgezet worden in een vreemde stad zonder stratenplan (of erger nog in een stad met een verkeerd stratenplan).

Bepaalde softwareontwikkelingsprocessen hebben deze problematiek proberen te aan te pakken door bijvoorbeeld de nadruk te leggen op code, testen, werkende software en individuele kennis van de programmeurs (*Agile Software Development*, waar ondermeer *Extreme Programming*, *XP*, onder valt). Deze hebben echter ook hun problemen. Bij *Extreme Programming* bijvoorbeeld wordt weinig nadruk gelegd op hoogniveau documentatie, zoals modellen als resultaat van analyse. Hierdoor wordt meer tijd doorgebracht met het schrijven van code, maar doordat weinig nadruk op analyse wordt gelegd, kan de totale tijd die aan het project besteed wordt veel groter worden dan het geval kon geweest zijn met een degelijke analyse op voorhand. *Refactoring* wordt gebruikt als de manier om de architectuur en het ontwerp van de software te verbeteren. Als echter door een tekort aan analyse van een verkeerde architectuur wordt uitgegaan, kan de nodige *refactoring* achteraf om tot een goede architectuur te komen enorm ingrijpend en tijdrovend zijn. Bij *Agile Software Development* in het algemeen treden er problemen op wanneer het ontwikkelingsteam ontmanteld wordt, want dan gaat de individuele kennis van de programmeurs verloren aangezien de methodes meestal niet vereisen dat deze kennis op papier gezet wordt. De code zelf is hierbij vaak de enige documentatie. Bij RUP, *Rational Unified Process*, worden projecten onderverdeeld in disciplines zoals documentatie, analyse, design en testen, en fases, zoals verwekking van het project, constructie en productie. Fases bepalen hoe de nadruk op de verschillende disciplines doorheen het project variëren. Deze fases kunnen elk uit één of meer iteraties bestaan. Hier ligt echter een probleem bij het identificeren van de fase waarin het project zich op elk moment bevindt, het ertoe aanzetten van programmeurs om alle disciplines van een fase te volgen, en het afstemmen van disciplines op elkaar binnen een fase.

Er lijkt dus nog een leemte te zijn op het vlak van efficiënt bijhouden van (hoogniveau) documentatie doordat deze met de huidige technologieën voornamelijk met de hand moet bijgehouden worden, waardoor tijd verloren gaat die anders kon besteed worden aan het produceren van code.

Een mogelijke oplossing voor dit probleem is om het genereren van documentatie vanaf code te integreren met het programmeren. Sommige talen, zoals Java, hebben hiervoor een laagniveau hulpmiddel (*Javadoc*). De hoger niveau documentatie (diagrammen en dergelijke) moet echter nog steeds met de hand onderhouden worden waardoor er vraag is naar softwarehulpmiddelen die op dit niveau intelligenter zijn.

2.1.2 Portabiliteit

Er komen voortdurend nieuwe programmeertalen, middleware-systemen, besturingssystemen en dergelijke op de markt. Aanpassing van bestaande software voor een nieuwe technologie vraagt een hoop werk, bovendien verliezen inspanningen voorheen geleverd voor oudere technologieën hun waarde. Recent denkt men aan de aanpassing van software voor de implementatie op het .NET-platform, of het overstappen naar een J2EE-platform. Bovendien komen van deze technologieën ook verschillende versies op de markt naarmate de tijd verstrijkt, zonder de garantie dat deze achterwaarts compatibel zijn. Daarom zou een manier om bestaande software gemakkelijk naar een nieuw platform over te dragen heel waardevol zijn. Bedrijven zouden veel tijd en geld kunnen besparen en bovendien sneller op marktevoluties kunnen reageren.

2.1.3 Systeeminteractie

Systemen leven slechts zelden in isolatie. Een typisch voorbeeld is een webapplicatie: de gebruikersapplicatie, bijvoorbeeld een internetwinkel of bankservice, draait in een internet browser (vb. Internet Explorer of Mozilla), en communiceert met webservers of databasesystemen om data uit te wisselen. Een ander voorbeeld is wanneer een applicatie Enterprise Java Beans (EJB) gebruikt en er nood is aan communicatie met een relationele databank als opslagmechanisme.

Een betrekkelijk groot systeem is altijd beter ontworpen als het bestaat uit kleinere, meer gespecialiseerde mechanismen met geïsoleerde functies (*separation of concerns*). Deze kleinere bouwstenen kunnen gebouwd worden, gebruik makend van de beste technologie die voor die specifieke functie beschikbaar is. Voor een dergelijk systeem is dan echter een mechanisme nodig voor onderlinge communicatie tussen deze bouwstenen, en deze nood aan communicatie stelt zekere eisen aan nieuwe systemen.

Daarnaast kan communicatie met *legacy systems* nodig zijn. Een *legacy system* kan gedefinieerd worden als een oud computersysteem dat technisch voorbijgestreefd is, maar nog steeds in gebruik is. *Legacy* systemen zijn meestal slecht gedocumenteerd door hun ouderdom, maar moeten operationeel blijven. Indien een nieuw systeem met een *legacy* systeem moet interageren is meestal een speciaal mechanisme, onder de vorm van middleware, nodig. Dit komt doordat *legacy* systemen gewoonlijk oudere, incompatibele technologieën gebruiken. Dit mechanisme moet voorzien worden bij het ontwerp van het nieuwe systeem.

Voor interactie tussen systemen die geschreven zijn in dezelfde of verschillende programmeertalen, bestaan reeds een aantal mechanismen die gezamenlijk onder de noemer *middleware* thuishoren. Door middel van een extra softwarelaag wordt communicatie tussen deze systemen mogelijk doordat ze beide een interface in de middlewaretaal implementeren. Op het niveau van de middlewarelaag wordt dan een systeem (bijvoorbeeld bij CORBA de ORB of *Object Request Broker*) voorzien dat een communicatiemechanisme verzorgt tussen objecten die deze middleware-interface implementeren. Communicatiedetails op laag niveau worden zo verborgen voor de programmeur.

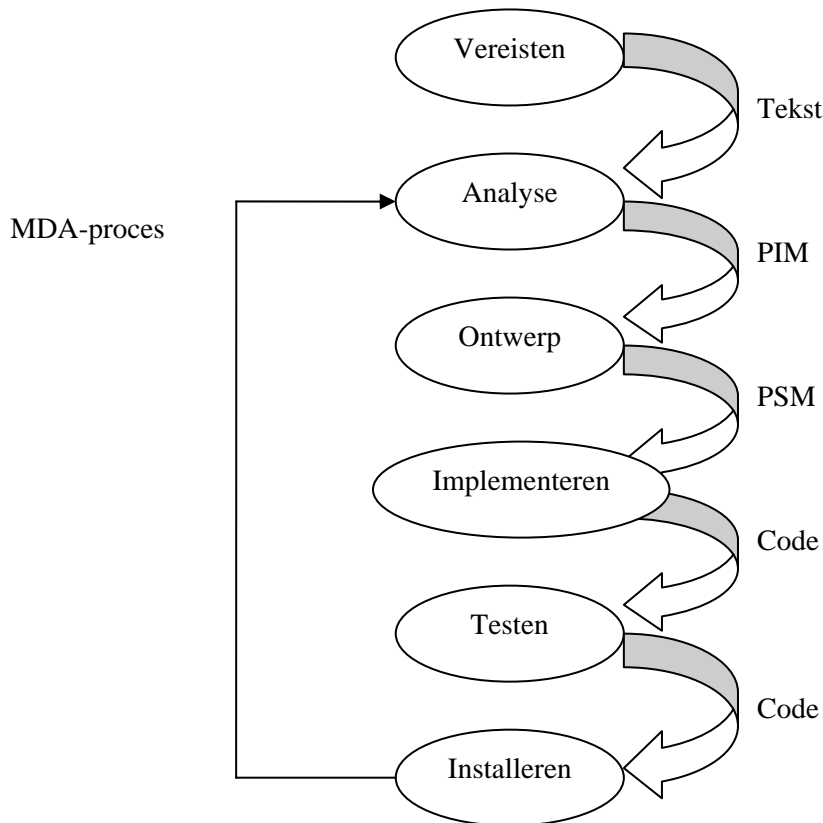
Het is echter mogelijk dat systemen die verschillende middleware-technologieën implementeren (zoals CORBA of .NET) ook onderling moeten kunnen communiceren, bijvoorbeeld omdat deze systemen reeds bestaan en heel moeilijk kunnen aangepast worden naar eenzelfde technologie. Interactie tussen verschillende middleware-platformen is mogelijk, mede door standaarden voor communicatie gedefinieerd door OMG (bijvoorbeeld tussen CORBA en andere middleware-technologieën), maar vraagt veel werk van programmeurs doordat dit allemaal met de hand moet geschreven worden. Momenteel bestaan er zo goed als geen softwarewerktuigen die dit soort interacties ondersteunen en er code voor genereren. Op dit vlak bestaat dus nog een leemte in de huidige stand van de technologie.

2.2 MDA

Model Driven Architecture (MDA) is een raamwerk van verschillende technologieën, gedefinieerd door de *Object Management Group* (OMG) dat aan de problemen uit de traditionele softwareontwikkeling (zoals hierboven opgesomd) een oplossing moet bieden. Na ondermeer objectgeoriënteerd programmeren en OMG's UML (*Unified Modeling Language*) voor het opstellen van klassendiagrammen zou het de volgende evolutie moeten betekenen in softwareland. MDA is een nieuwe manier om specificaties te schrijven en applicaties te ontwikkelen, gebaseerd op platformonafhankelijke en platformspecifieke modellen van de applicatie. Een *platform* definiëren we als een verzameling van subsystemen of technologieën die een coherente verzameling van functionaliteit aanbieden via interfaces en specifieke gebruikspatronen, waar een systeem kan over beschikken zonder zich zorgen te hoeven maken over de details van de implementatie van deze functionaliteit.

Een volledige MDA-toepassing bestaat uit een PIM (*Platform Independent Model*), plus een aantal PSM's (*Platform Specific Model*) en volledige implementaties (codesets) van de toepassing, één voor elk platform die de softwareontwikkelaar wenst te ondersteunen. Een PIM concentreert zich op de

functionaliteit en het gedrag van de applicatie of het systeem, onverstoord door de details van de technologie of technologieën waarmee deze geïmplementeerd gaat worden. Dit wordt verder toegelicht in 2.2.1. Een PSM voegt deze details toe aan het PIM en is een model van de implementatie van een PIM voor een bepaald platform. Het PSM wordt toegelicht in 2.2.2. De transformatie van PIM naar PSM gebeurt door een automatische transformatie van een verzameling transformatieregels, geformuleerd in een transformatietaal. Ook de transformatie van PSM naar codesets gebeurt door een verzameling transformatieregels. Deze transformatie wordt toegelicht in 2.2.3.



Figuur 2.3: Softwareontwikkelingscyclus m.b.v. MDA

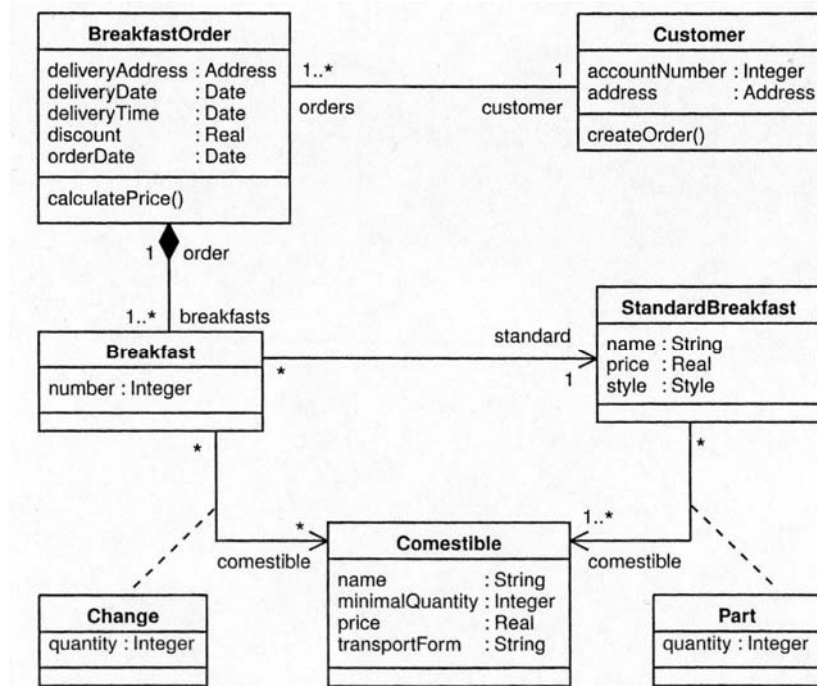
De verschillende stappen waar ontwikkelaars doorheen gaan in de MDA-aanpak zijn niet echt verschillend van het traditionele ontwikkelingsproces, zoals te zien is in Figuur 2.3. Het voornaamste verschil ligt erin dat er meer nadruk gelegd wordt op de hoogniveau artefacten die gecreëerd worden, zoals PIM en PSM die hiervoor reeds aangehaald werden. Doordat meer nadruk op deze documenten wordt gelegd, wordt het “*programmers short cut*”-fenomeen vermeden. De voordelen van MDA ten opzichte van het klassieke proces voor softwareontwikkeling worden toegelicht in 2.2.4.

2.2.1 Platform Independent Model

Het eerste model gedefinieerd door MDA is een hoogniveau abstractie van de applicatie, onafhankelijk van een implementatietechnologie. Dit wordt het Platform Independent Model (PIM) genoemd. In een PIM wordt het systeem gemodelleerd volgens de beste wijze om het *business problem*, de commerciële opdracht waarvoor de software een oplossing moet bieden, op te lossen. Een implementatietechnologie, bijvoorbeeld J2EE of .NET, komt op dit niveau nog niet aan bod. Een PIM komt min of meer overeen met wat klassiek gekend is als een domeinmodel.

Hierna volgt een voorbeeld van een beperkt UML-PIM voor een internetgebaseerde ontbijtdienst, een voorbeeld dat voorkomt in [AKJWWB03]. Dit diagram wordt in een creatief proces door een programmeur gemaakt. Er worden een aantal standaard ontbijten gepresenteerd; daarnaast kunnen

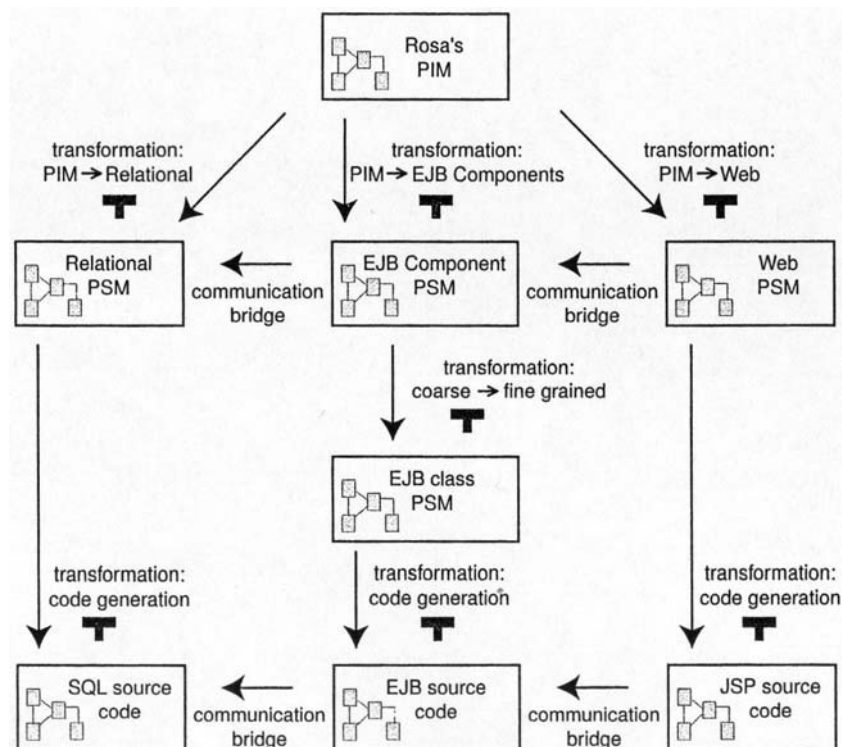
ontbijten zelf samengesteld worden, beide bestaan uit een aantal ingrediënten (*comestible*). De prijs van een bestelling wordt gebaseerd op de gekozen stijl en de prijs van de gekozen ontbijten plus een klein toeleveringsbedrag. Het model beschrijft de ontbijtdienst onafhankelijk van een implementatietechnologie en is daardoor zeker een PIM. Er moet echter opgemerkt worden dat een PIM altijd relatief is ten opzichte van de gekozen abstractie van het platform, het kan dicht aanleunen bij de technologie die uiteindelijk in de PSM gebruikt wordt, of er een heel eind vandaan staan.



Figuur 2.4: Internetgebaseerde ontbijtdienst

2.2.2 Platform Specific Model

De volgende stap is om het PIM te transformeren in één of meerdere *Platform Specific Models* (PSM's). Een PSM beschrijft het systeem in termen van implementatieconcepten, beschikbaar in één bepaalde implementatietechnologie. Een EJB PSM bijvoorbeeld, specificeert het systeem, beschreven in het PIM, in termen van EJB concepten zoals *Beans*. In een PSM, geschikt voor een relationele databank, zouden dan weer elementen als tabel, kolom, sleutel, ... voorkomen. Bij transformatie van een PIM gericht naar een bepaalde programmeertaal, kan bijvoorbeeld rekening gehouden worden met al of niet ondersteunen van meervoudige overerving. Als een PIM waarin meervoudige overerving gebruikt wordt dan getransformeerd wordt naar een taal die dit niet ondersteunt, zal hierbij in de transformatie rekening gehouden moeten worden. Transformatie van PIM naar PSM is de meest complexe stap in het MDA-geheel en kan herhaald worden voor meerdere platformen. De transformatie tussen modellen gebeurt op het niveau van het meta-model: indien een invoermodel en uitvoermodel in UML gespecificeerd worden, specificeren transformaties hoe bijvoorbeeld attributen van de bronklasse op de doelklasse afgebeeld worden. De transformatie heeft het daarbij niet over bepaalde attributen van een bepaalde klasse, maar over alle attributen van alle klassen die aan bepaalde beperkingen voldoen. Het niveau waarop klassen, attributen en dergelijke beschreven kunnen worden als objecten van het type klasse of attribuut is het niveau van het meta-model.



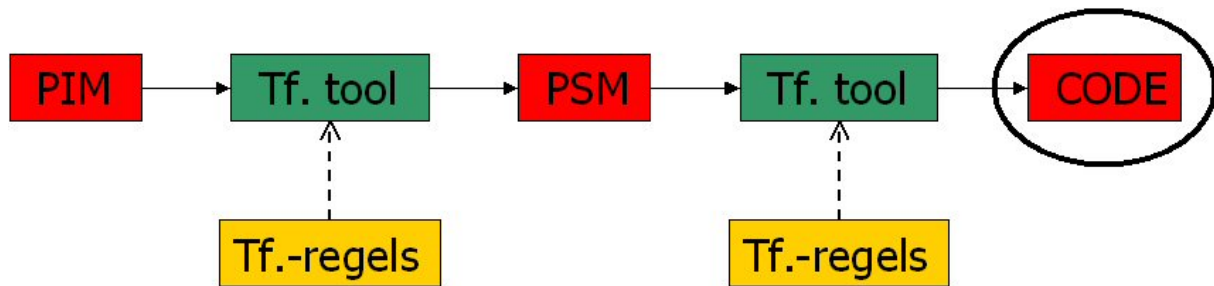
Figuur 2.5: PIM naar PSM transformatie

In Figuur 2.5 wordt voorgesteld hoe de PIM voor het voorbeeld uit 2.2.1 getransformeerd wordt naar drie verschillende PSM's: een PSM voor een relationele databankschema, een PSM voor implementatie met EJB en een PSM voor een internetinterface. Vervolgens wordt uit elke PSM ook code gegenereerd: SQL-broncode voor het relationele databankschema, EJB-broncode voor het EJB-gedeelte, en JSP-broncode voor het webgebaseerde schema. Ook codegeneratie gebeurt met een transformatie (zie ook 2.2.3). Voor de volledige realisatie van het internetgebaseerde gedeelte van de ontbijtdienst, gaan deze verschillende implementaties dan samenwerken en communiceren door middel van *bridges* die ook door een MDA-werktuig gegenereerd worden. Bridges worden verder toegelicht in 2.2.4.

In Figuur 2.5 komt ook een voorbeeld van een intermediaire transformatie voor: vooraleer uit de EJB-Component-PSM code gegenereerd wordt, wordt dit model naar een EJB-Class-PSM getransformeerd. Hierin worden de EJB-componenten gegenereerd door de eerste transformatie, verfijnd naar bijhorende EJB-klassen. Daarna wordt voor de verschillende EJB-klassen code gegenereerd. Dit is een voorbeeld van een PIM-naar-PSM transformatie.

2.2.3 Code

De laatste stap in het ontwikkelingsproces is de transformatie van PSM naar code. Dit soort transformatie is over het algemeen eenvoudiger dan de transformatie van PIM naar PSM doordat een PSM de implementatie reeds van vrij dichtbij beschrijft. Het transformatieproces wordt grafisch voorgesteld in Figuur 2.6.



Figuur 2.6: Transformatiestappen

Transformaties worden uitgevoerd door een transformatiemotor. Hiervoor wordt gebruik gemaakt van een set transformatieregels, welke worden gedefinieerd als een beschrijving van hoe één of meerdere concepten in de brontaal kunnen getransformeerd worden in één of meerdere concepten in de doeltaal. Voor transformatie van PIM naar PSM wordt een bepaalde set transformatieregels gebruikt, voor transformatie van PSM naar code een andere set. Beide transformaties gebruiken als invoer een model, de eerste transformatie echter geeft als uitvoer een nieuw model (PSM) terwijl de tweede transformatie als uitvoer tekstbestanden met code produceert. Ook de transformatie van PSM naar code wordt op het niveau van het metamodel geformuleerd: voor klassen en attributen wordt een bepaald stuk code geproduceerd.

Transformatie van PIM naar PSM of van PSM naar code kan uit meerdere stappen bestaan. Hierbij wordt een PIM naar een volgende intermediaire PIM getransformeerd of een PSM naar een volgende intermediaire PSM tot uiteindelijk een laatste PSM op de gegenereerde code afgebeeld wordt. Deze verschillende stappen komen overeen met verschillende niveaus van platformonafhankelijkheid.

2.2.4 Voordelen van MDA

Op het eerste zicht lijkt de MDA-aanpak misschien niet veel te verschillen van meer traditionele CASE-tools. Er is een uitgebreid gamma tools in de handel beschikbaar die bijvoorbeeld een UML-klassendiagram voor een bepaalde programmeertaal kunnen omzetten in code. Wat nieuw is aan de MDA-aanpak is de omzetting van PIM naar PSM. Vaak wordt veel tijd gespendeerd om een hoogniveau beschrijving van een systeem in bijvoorbeeld een relationeel databankschema om te zetten. Met de MDA-aanpak wordt beoogd om ook dit deel van het ontwikkelingsproces volledig te automatiseren. Een meer gedetailleerde beschrijving van de voordelen van MDA volgt hierna.

Documentatie

Bij MDA wordt de aandacht van het softwareontwikkelingsproces verschoven naar de ontwikkeling van het PIM en transformatieregels. PSM's worden daarna, zoals hoger aangehaald, automatisch gegenereerd door een transformatie van PIM naar PSM. Benodigd hiervoor zijn een set transformatieregels voor PIM naar PSM, gedefinieerd door een expert in het vakgebied. Deze kunnen dus bijvoorbeeld door een ander team in parallel met de PIM ontwikkeld worden. Het voordeel van deze aanpak is echter dat de definitie van transformatieregels slechts één keer hoeft te gebeuren, waarna ze vele malen kunnen toegepast worden.

Het grootste deel van de inspanning voor het ontwikkelen van een nieuwe toepassing gaat dus naar de definitie van het PIM. Doordat dit onafhankelijk van de details van een bepaald platform (besturingssysteem en programmeertaal, middleware enzovoort) kan gebeuren, is er een grote hoeveelheid technische details waar men op dit niveau nog geen rekening mee hoeft te houden. Dit heeft twee voordelen.

Ten eerste hoeven PIM-ontwikkelaars minder werk te verrichten doordat platformspecifieke details niet telkens moeten neergeschreven worden: deze worden toegevoegd door de automatische transformatie van PIM naar PSM. Ten tweede wordt het *business problem* doorgaans beter opgelost, doordat de aandacht van het proces verschuift van pure code naar een hoogniveau beschrijving van het systeem. Zo worden “*accidental complexities*”, zoals Fred Brooks in [FB87] formuleerde, verder

gereduceerd. Hierdoor worden echter ook de eisen die gesteld worden aan de hoogniveau beschrijving (PIM) hoger, het systeem dient hierin tot in detail beschreven te worden. Een hoogniveau beschrijving is niet langer “enkel papier” maar maakt nu integraal deel uit van het ontwikkelingsproces. Hierdoor wordt er ook een soort van “automatische documentatie” gerealiseerd: veranderingen aan het systeem gebeuren rechtstreeks of onrechtstreeks (via het PSM) door de PIM te veranderen. Hierdoor hoeven diagrammen niet meer met de hand bijgewerkt te worden om consistent met de code gehouden te worden zoals tegenwoordig wel het geval is. Ook onderhoud van het systeem wordt eenvoudiger doordat dit niet in de programmacode maar op een hoger niveau van abstractie, PIM of PSM, kan gebeuren.

Portabiliteit

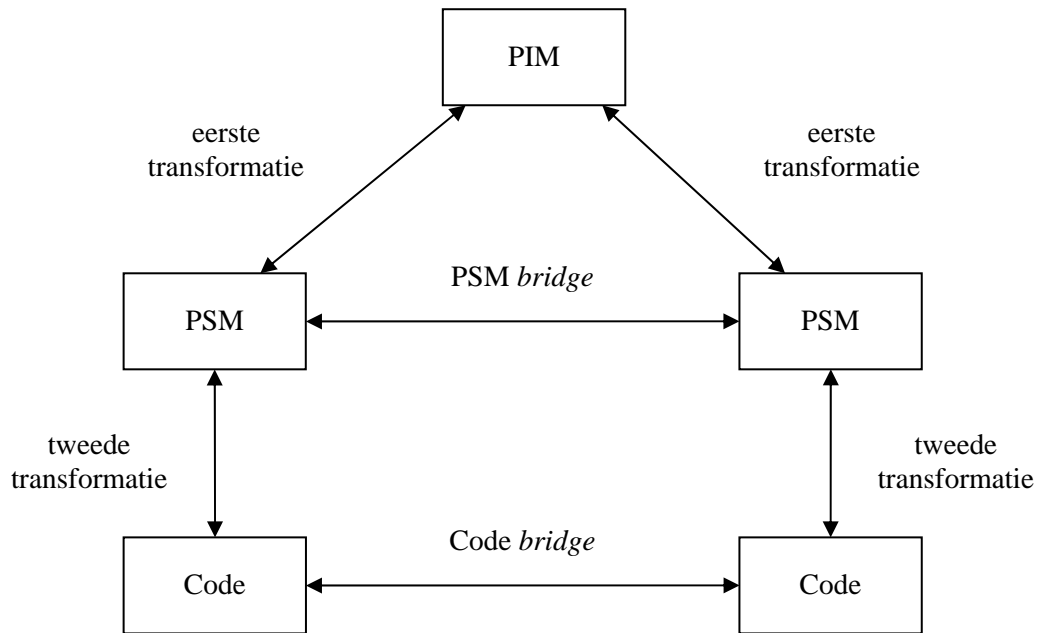
Doordat een PIM door een MDA-transformatiemotor automatisch kan getransformeerd worden in PSM's voor meerdere platformen, is alles wat gespecificeerd wordt op het PIM-niveau volledig overdraagbaar. Als een nieuwe technologie op de markt komt, kan een bestaande PIM ook naar dit platform overgedragen worden zodra transformatieregels voor dit platform beschikbaar gesteld of zelf gedefinieerd worden. Bestaande toepassingen kunnen dus in korte tijd aangepast worden.

Doordat technologiespecifieke zaken niet langer met de eigenlijke logica van een toepassing vermengd worden, wordt ook de intellectuele eigendom van een bedrijf beter beschermd. Als deze apart kan beheerd worden, kan de kennis over een product beter beheerd en behouden worden wanneer van technologie veranderd wordt. Onder intellectuele eigendom verstaan we alles wat volledig door het bedrijf ontwikkeld wordt en niet wordt gerealiseerd door gebruik te maken van een technologie van een derde partij.

Systeeminteractie

Wanneer meerdere PSM's gegenereerd worden vanaf eenzelfde PIM, kunnen er relaties bestaan tussen de resulterende sets code. Codesets, gegenereerd voor verschillende platformen, kunnen over het algemeen niet rechtstreeks met elkaar communiceren. Ook dit wordt in MDA opgelost, door samen met de code, *code-bridges* te genereren. Een niveau hogerop wordt dit gemodelleerd via *PSM-bridges*. De verschillende bridges worden voorgesteld op Figuur 2.7.

Ook voor de problematiek van middleware-interactie biedt MDA een oplossing. Eens de manier geïdentificeerd is waarop een platform middlewareconcepten en -functies implementeert, kunnen specialisten deze informatie in MDA brengen als een afbeelding: een set transformatieregels die een transformatie definiëren. Verschillende middlewarewerktuigen kunnen zo geïntegreerd worden in één systeem. Hoe meer beschrijvingen van middlewareplatformen in MDA aangebracht worden, hoe meer transformaties van één platform naar een ander kunnen gedefinieerd worden, en hoe meer ook de generatie van bridges tussen middlewareplatformen kan geautomatiseerd worden.



Figuur 2.7: Systeeminactie via bridges

Gaan we even terug naar het voorbeeld uit Figuur 2.5. Voor elk element in het PIM van dit voorbeeld weten we naar welk(e) element(en) in de PSM's dit getransformeerd wordt. De transformatie van een "Customer"-element in het PIM zou resulteren in een *Customer*-tabel in het relationele databankmodel enerzijds en een *Customer*-bean in de EJB-implementatie anderzijds. Een bridge bouwen tussen de EJB-implementatie en het relationele databankschema is daardoor rechtlijnig doordat voor elk element uit beide PSM's gekend is van welke elementen uit het PIM deze afgeleid zijn.

Deze bridge kan vervolgens gebruikt worden voor opvraging en opslag van *Customer*-objecten. Om een *Customer*-object uit de databank te halen, gaan we een zoekopdracht op de *Customer*-tabel uitvoeren, een instantie van de *Customer*-bean maken en deze vullen met de data verkregen uit de databank. Om een *Customer*-object op te slaan, kunnen we de data uit de *Customer*-bean opslaan in de *Customer*-tabel van de databank.

Hoofdstuk 3

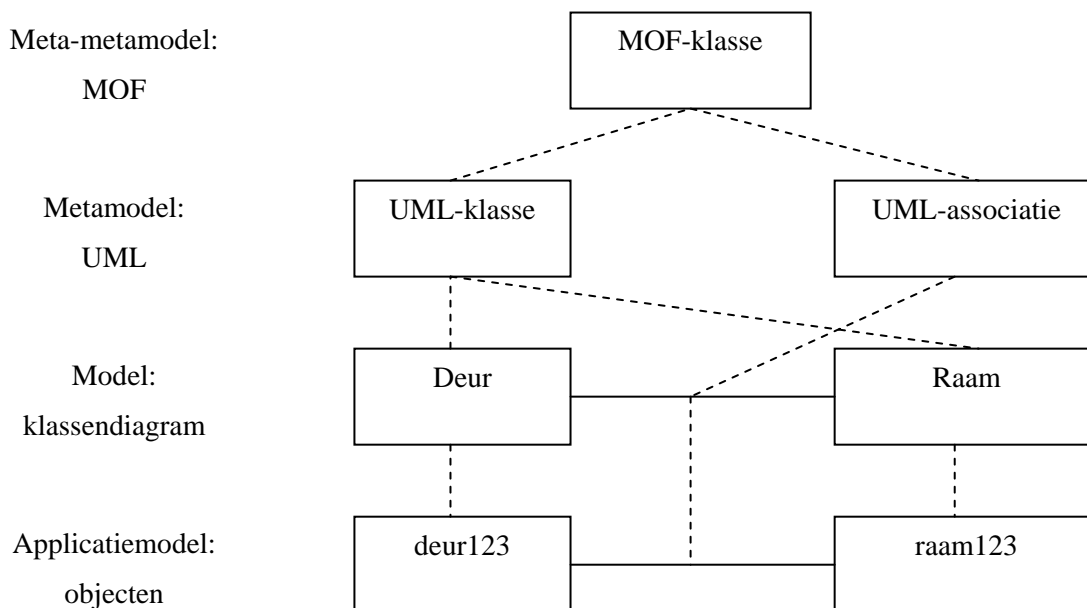
Standaarden hergebruikt voor MDA

Hoewel MDA een revolutionair nieuwe techniek vormt, wil OMG met de standaardisatie van MDA niet het wiel opnieuw uitvinden. Een aantal standaarden en technieken die zich in het verleden reeds als nuttig en betrouwbaar bewezen hebben, worden voor MDA opnieuw gebruikt. Tot deze technieken behoren ondermeer *Meta Object Facility* (zie 3.1), *Unified Modeling Language* (zie 3.2) en *Object Constraint Language* (zie 3.3). Om een volledig beeld van MDA te krijgen, is het nodig deze technieken te begrijpen en te kunnen situeren. Daarom worden deze in dit hoofdstuk toegelicht. De standaardisatie van MDA door OMG waarin duidelijk wordt hoe van deze technieken precies gebruikt gemaakt wordt, wordt verklaard in Hoofdstuk 5.

3.1 Meta Object Facility

Zoals in Hoofdstuk 2 werd toegelicht, werkt ontwikkeling met MDA als een serie van transformaties tussen modellen. Welke modeltypes wordt echter door MDA niet duidelijk gespecificeerd. Een logische keuze voor het ontwerpen van software zou UML-diagrammen zijn, maar dit hoeft niet noodzakelijk het geval te zijn. Voor elke transformatie die uitgevoerd wordt, dient de transformatiemotor van het MDA-werktuig de specificatie te kennen van de modellen waar tussen getransformeerd wordt. Om een maximum aan flexibiliteit te behouden, is de beste manier om dit te doen om het MDA-werktuig van een formele specificatie van het invoer- en uitvoermodel te voorzien. Zo kunnen transformaties tussen alle willekeurige modellen plaatsvinden, op voorwaarde dat de formele specificatie van elk model voorzien is.

Ook deze formele specificatie dient aan een bepaalde standaard te voldoen, zodat het MDA-werktuig dit kan interpreteren. Op dit moment bevinden we ons op het niveau van het meta-metamodel, de taal waarin gebruikte metamodellen gespecificeerd worden. Hiervoor heeft OMG MOF voorzien, de *Meta Object Facility*. MOF wordt beschreven in [formal/2006-01-01]. Een voorbeeld, voor het geval van een klassendiagram, wordt voorgesteld in onderstaande figuur.



Figuur 3.1: Van meta-metamodel naar applicatiemodel

In Figuur 3.1 is te zien hoe een MOF-model zich verhoudt tot de instanties ervan. Het UML-metamodel beschrijft een UML-klasse en een UML-associatie. Dit zijn instanties van een MOF-klasse. Van een UML-klasse zijn twee instanties in het klassendiagram, namelijk Deur en Raam. Tussen Deur en Raam bestaat een associatie. Dit is een instantie van een UML-associatie. Op het niveau van het applicatiemodel, de objecten, is deur123 een instantie van Deur en raam123 een instantie van Raam. De associatie tussen deur123 en raam123 is een instantie van de associatie uit het model. De verhouding tussen de instanties op de verschillende niveaus, het instantiëren, betekent dat de instantie op het lager niveau voldoet aan de syntax op het hoger niveau.

Het is belangrijk op te merken dat transformaties zoals beschreven in Hoofdstuk 2 gebeuren op het niveau van het model. PIM of PSM zijn instanties van modellen, die bijvoorbeeld in UML beschreven worden. Indien men PIM of PSM in een andere modelleringstaal wil specificeren, kan dit door het transformatiewerktuig met dit PIM of PSM te voorzien, samen met een specificatie van deze alternatieve modelleringstaal in MOF. Een applicatiemodel wordt gegenereerd door een aparte transformatie van model naar code. MOF doet dienst als specificatie voor het uitwisselen van metamodelen waar transformaties tussen gebeuren.

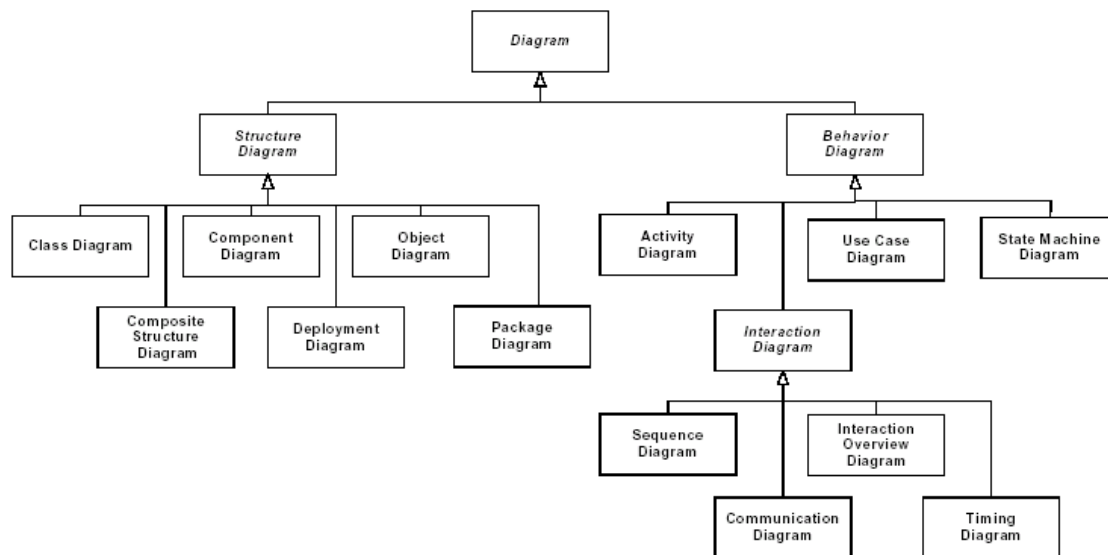
3.2 Unified Modeling Language

UML is gekend als grafische notatie voor het modelleren van de structuur van een applicatie, ook een klassendiagram genoemd. Naast klassendiagrammen kunnen ook gedrag onder de vorm van toestandsdiagrammen, zakenprocessen en datastructuren met UML gemodelleerd worden. De formaliteit van UML, het voldoen aan een onderliggend metamodel, maakt de taal bruikbaar voor het implementeren van softwarewerktuigen die deze diagrammen lezen. Zo kan bijvoorbeeld uit een klassendiagram code gegenereerd worden. Dit concept is heel aantrekkelijk voor MDA, en UML werd met het oog op MDA dan ook grondig vernieuwd. UML is daarom heel geschikt voor het opstellen van een PIM of PSM. Hierin kan conform de UML 2.0-standaard niet alleen een beschrijving van de modelementen opgenomen worden, maar ook informatie in verband met de positionering van deze modelementen. Op deze manier kan een UML-diagram altijd op dezelfde manier weergegeven worden.

De specificatie van UML 2.0 bestaat uit vier delen:

- UML 2.0 Superstructuur: deze definieert de kern van de UML 2.0 taal. Dit bestaat uit een aantal klassen- en interactiediagrammen en de definitie van de elementen die hiertoe behoren. Dit is het deel van de taal dat werktuigen die compatibel zijn met UML 2.0 gaan implementeren. De Superstructuur wordt beschreven in [formal/05-07-04].
- UML 2.0 Infrastructuur: de infrastructuur definieert basisklassen die de basis vormen voor niet alleen de UML 2.0 Superstructuur, maar ook MOF 2.0 (zie ook 3.1). De Infrastructuur wordt beschreven in [formal/03-09-15].
- UML 2.0 *Object Constraint Language* (OCL): dit maakt het definiëren van pre- en postcondities, invarianten en andere condities op elementen van het UML-model mogelijk (zie ook 3.3).
- UML 2.0 Diagramuitwisseling: deze specificatie breidt het UML metamodel uit met graafgeoriënteerde informatie. Hierdoor kunnen modellen uitgewisseld worden en opgeslagen of gelezen, waarna ze weer kunnen voorgesteld worden zoals ze oorspronkelijk waren. Ook Diagramuitwisseling wordt in [formal/05-07-04] beschreven.

Naast de opdeling van de specificatie in deze vier functionele delen, kan de UML-specificatie ook opgedeeld worden volgens types diagrammen. Met UML kan namelijk niet alleen de structuur of substructuur van een systeem gemodelleerd worden via een klassendiagram. Ook de functionaliteit van het systeem kan gemodelleerd worden via een *use case* diagram, het interne gedrag via een sequentiediagram of activiteitsdiagram, en zomeer. In totaal ondersteunt de standaard 13 types diagrammen. De mogelijke diagrammen uit de UML 2.0-standaard worden in onderstaande figuur weergegeven.



Figuur 3.2: de types diagrammen in de UML 2.0-standaard

Het is interessant ook even te kijken welke wijzigingen aan de standaard aangebracht zijn voor versie 2.0. Deze wijzigingen kunnen opgedeeld worden volgens type diagram. Hieronder worden enkele wijzigingen toegelicht die van belang kunnen zijn voor MDA.

Voor klassendiagrammen was voordien de discussie om associaties tussen klassen als associaties in het klassendiagram voor te stellen, of als een attribuut dat aangeduid wordt in een tekstvak van de bijhorende klasse. De nieuwe versie van UML maakt deze twee oplossingen evenwaardig. Dit wordt verder uitgediept bij het opstellen van de transformatieregels in Hoofdstuk 6.

Verder wordt in UML 2.0 ook de notie van een afgeleid attribuut ingevoerd. Dit slaat op een attribuut dat geen reëel attribuut is maar dat kan afgeleid worden uit andere elementen. Zo zou een `Line`-klasse een afgeleid attribuut `length` kunnen hebben waarvan de waarde afgeleid wordt uit de twee eindpunten. Afgeleide attributen zijn eveneens van belang in Hoofdstuk 6.

UML 2.0 bepaalt verder ook drie niveaus van compatibiliteit: *basis*, *tussenliggend* en *volledig*. Bij ieder niveau van compatibiliteit hoort een mate waarin het modelleerwerktuig de UML-standaard implementeert. Dit kan belangrijk zijn voor wie een MDA-werktuig wil implementeren dat werkt met UML 2.0, zoals E2S (zie 1.3).

Sinds UML 1.3 bestaat ook de notie van een UML-profiel. Een profiel is hier een pakket modelementen die worden aangepast voor een specifiek doel, gebruik makend van definitie van (*tags*), beperkingen gedefinieerd in OCL 2.0 (zie 3.3) en uitbreidingsmechanismen zoals stereotypes. Een profiel kan ook bibliotheken definiëren waar het UML-model van afhangt en de deelverzameling modelementen die het verfijnt. Vooral deze laatste toepassing is belangrijk. Doordat de UML-standaard voor de meeste toepassingen te uitgebreid is, worden UML-profielen gebruikt om aan te geven welke deelverzameling modelementen van UML 2.0 het model gebruikt. Anderzijds bieden profielen ook een handige manier om modelementen aan een modelleerwerktuig toe te voegen zonder dat aan de kern van de UML-standaard moet geraakt worden. Ook profielen zullen belangrijk blijken in Hoofdstuk 6.

Andere wijzigingen zijn eerder onbelangrijk in de context van dit eindwerk.

3.3 Object Constraint Language

OCL, kort voor *Object Constraint Language*, is een taal voor analyse, en tegenwoordig ook ontwerp van software. OCL is een declaratieve taal voor het neerschrijven van beperkingen waaraan UML-modellen dienen te voldoen, en daardoor geschikt voor *design by contract*. In het bijzonder wordt OCL gebruikt om beperkingen over MOF-modelementen te formuleren die moeilijk kunnen uitgedrukt worden met de grafische notatie van UML. Zo maakt OCL het in de context van MDA mogelijk om extra detail toe te voegen aan UML. Oorspronkelijk vormde OCL als formele specificatietaal slechts een uitbreiding van UML, maar tegenwoordig kan OCL gebruikt worden om beperkingen te formuleren op eender welk MOF-model. Het evalueren van een beperking verandert echter geen enkele waarde in het systeem. Een beperking stelt dat "dit zou moeten gelden". Als voor een zeker object deze beperking gebroken is, kan in OCL niet uitgedrukt worden wat moet gedaan worden om de verkeerde toestand te verbeteren. OCL is dus een taal zonder neveneffecten. OCL wordt formeel gespecificeerd in [ptc/03-10-14].

Omdat OCL door zijn declaratieve aard modelnavigatie heel gemakkelijk maakt, zijn bestaande talen voor modeltransformatie in MDA vaak gebaseerd op OCL. Daarom is het van belang OCL even in meer detail te bestuderen.

3.3.1 Beperkingen

Er kunnen vier verschillende types beperkingen geformuleerd worden:

- een invariant is een beperking die uitdrukt dat een bepaalde voorwaarde altijd moet gelden bij alle instanties van een klasse, type, of interface. Een invariant wordt beschreven door middel van een uitdrukking die tot waar, `true`, evalueert als aan de invariant voldaan is. Invarianten moeten op elk mogelijk ogenblik waar zijn. Dit wordt besproken in 3.3.2.
- een preconditionie van een operatie is een beperking die waar moet zijn op het ogenblik dat de operatie gaat uitgevoerd worden. Precondities worden besproken in 3.3.3.

- een postconditie van een operatie is een beperking die moet gelden als de operatie zijn uitvoering net beëindigd heeft. Postcondities drukken de verplichtingen van de operatie uit. Ook postcondities worden besproken in 3.3.3.
- een wachtter of *guard*, is een beperking die moet gelden voor een overgang van toestand plaatsvindt.

Deze kunnen gebruikt worden voor het uitdrukken van een aantal zaken in een model:

- de initiële waarde van een attribuut of associatie-einde
- een afleidingsregel voor een attribuut of associatie-einde, wat eigenlijk een zoekopdracht is
- het lichaam van een operatie
- het aanwijzen van een instantie in een dynamisch diagram
- het aanwijzen van een conditie in een dynamisch diagram
- het aanwijzen van de actuele waarden van parameters in een dynamisch diagram

3.3.2 Invarianten

De meest eenvoudige invariant is een invariant op een attribuut. Veronderstel dat ons model een klasse `Klant` bevat met een attribuut `leeftijd`, dan beperkt de volgende beperking de waarde van het attribuut:

```
context Klant inv:
leeftijd >= 18
```

Figuur 3.3: een invariant in OCL

De context van een OCL uitdrukking bepaalt de modelentiteit waarvoor de OCL-uitdrukking, en dus de beperking, gedefinieerd wordt. Dit is gewoonlijk een klasse, interface, datatype of component. In dit geval is dit de klasse `Klant`. De invariant bepaalt dat alle klanten minstens de leeftijd van 18 moeten bereikt hebben.

3.3.3 Pre- en postcondities

Pre- en postcondities worden altijd over operaties uitgedrukt. In pre- en postcondities mogen de parameters van de operatie worden gebruikt. Verder is er een speciaal sleutelwoord `self` dat gebruikt wordt om naar de contextuele instantie te verwijzen. Dit is niet de context van de beperking maar de instantie van de klasse waartoe de operatie behoort. De context van een pre- of postconditie is altijd de operatie waarvoor de beperking wordt uitgedrukt. Het sleutelwoord `result` dat staat voor de terugkeerwaarde van de operatie. Dit sleutelwoord kan enkel gebruikt worden in de postconditie. Zo kunnen we bijvoorbeeld de volgende beperkingen formuleren voor de operatie `verkoop` van de `Verkoper` klasse.

```

context Verkoper::verkoop( item: Item ): Real
pre: self.verkoopbareItems->includes( item )
post: not self.verkoopbareItems->includes( item ) and result =
item.prijs

```

Figuur 3.4: een pre- en postconditie in OCL

3.3.4 Overige

Modellen definiëren vaak afgeleide attributen en associaties. Een afgeleid element staat niet op zichzelf. De waarde van een afgeleid element wordt bepaald uit andere waarden in het model. Dit komt overeen met de notie van een afgeleid attribuut die geïntroduceerd werd in UML 2.0 (zie ook 3.2). Het weglaten van de manier om de waarde van het afgeleide element te bepalen resulteert in een onvolledig model. Met OCL wordt dit ondersteund met de notie van een afleidingsregel.

Meestal is de multipliciteit van een associatie niet 1, maar meer dan 1. Het evalueren van de beperking zal in deze gevallen een verzameling van instanties betrekken van de geassocieerde klasse. Beperkingen kunnen geformuleerd worden op de verzameling zelf of op de elementen van de verzameling. OCL biedt hiervoor de notie van een verzameling, samen met een aantal constructies, ondermeer voor het itereren over of het bepalen van het aantal elementen van een verzameling. Verzamelingen in OCL kunnen al of niet geordend zijn, en al of niet unieke elementen bevatten.

3.3.5 Nieuw in OCL 2.0

OCL werd in het verleden enkel als analysetaal gebruikt. Met het oog op ondermeer MDA, wint OCL aan succes als ontwerptaal. Dit is echter een traag proces. Daarom werden voor versie 2.0 slechts minimale wijzigingen aangebracht aan OCL als taal voor contractspecificatie. De grootste wijzigingen spelen zich af de formele basis van de taal. De wijzigingen die meest van belang zijn voor gebruik met MDA worden hierna kort besproken.

OCL is vanaf versie 2.0 voorzien van een MOF-metamodel (zie ook 3.1). Dit maakt de taal geschikt voor gebruik met MDA-transformaties. Dit metamodel beschrijft de concepten van OCL en legt de semantiek eenduidig vast. De specificatie van OCL als een MOF-metamodel laat ook een vlottere integratie toe met UML 2.0 (zie 3.2). De presentatie van de semantiek is ook formeler en laat zo beter afbeeldingen naar andere semantische domeinen toe (zoals bijvoorbeeld andere talen). OCL 2.0 is de eerste versie die een vaste, gestandaardiseerde semantiek heeft.

Verder werden aan OCL 2.0 ook enkele constructies toegevoegd die het gebruik van OCL als taal voor zoekopdrachten, *queries*, gemakkelijker maken. Zo werd een *tuple* type aan de taal toegevoegd om datatypes te groeperen. Verder kan het nieuwe concept van afgeleid attribuut (zie 3.2), dat gedeeld wordt met UML, ook rechtstreeks als zoekopdracht gebruikt worden.

3.4 Andere technologieën

Naast de hoger vermelde standaarden, worden nog enkele technologieën hergebruikt voor MDA. Deze zijn minder belangrijk dan de vorige, maar keren niettemin terug. Hiertoe behoren XMI (zie 3.4.1), en CWM (3.4.2).

3.4.1 XML Metadata Interchange

XML Metadata Interchange is een OMG-standaard voor het uitwisselen van metadata-informatie via het Extensible Markup Language (XML)-formaat. Het kan gebruik worden voor het uitwisselen van de metadata van elk metamodel dat uitgedrukt is in MOF. XMI wordt meest gebruikt als formaat voor het uitwisselen van UML-modellen, hoewel het ook kan gebruikt worden voor serializatie van modellen die voldoen aan andere metamodellen. Het doel van XMI is om een gemakkelijke uitwisseling van metadata mogelijk te maken tussen UML-gebaseerde modelleringswerktuigen en MOF-gebaseerde

opslagmedia in heterogene omgevingen. XMI wordt volledig gespecificeerd in [formal/05-09-01], XML in [W3C04].

3.4.2 CWM

CWM, kort voor *Common Warehouse Metamodel*, is een OMG-standaard voor de uitwisseling van metadata tussen kennisbanken, kennisbeheer en portaaltechnologieën. CWM biedt een oplossing voor verschillen tussen de metamodellen van deze systemen door een gemeenschappelijke basis te bieden voor deze metamodellen. Indien twee verschillende metamodellen beide MOF-compatibel zijn, kunnen modellen gebaseerd op deze metamodellen in hetzelfde opslagmedium bewaard worden. MOF is daardoor gelijkaardig en gerelateerd aan MOF (zie 3.1). CWM wordt volledig beschreven in [ad/2001-02-01].

Hoofdstuk 4

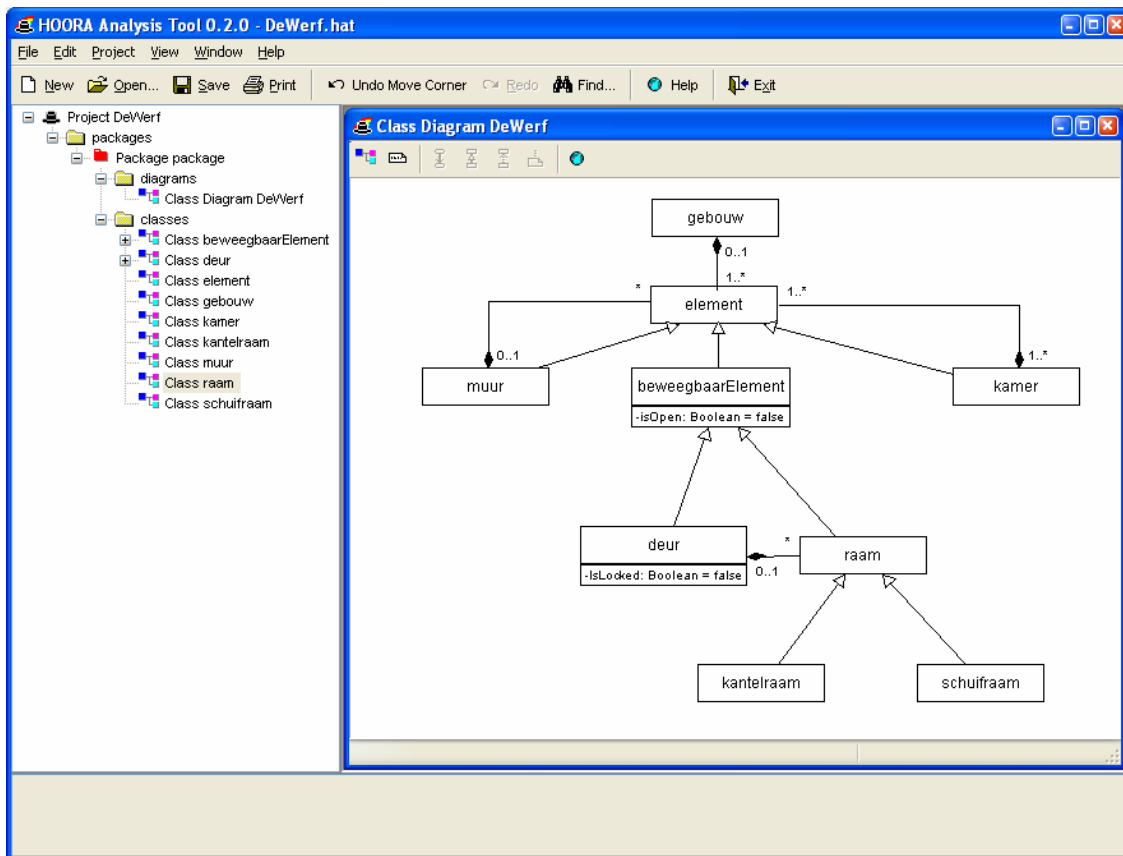
HOORA

Zoals in 1.2 vermeld, is een proefimplementatie van een modeltransformator in de gekozen transformatietaal een onderdeel van deze thesis. Deze modeltransformator zou dezelfde mogelijkheden moeten hebben als de bestaande software van E2S. Daarom wordt in dit hoofdstuk deze applicatiesuite toegelicht. Dit omvat een overzicht van de verschillende onderdelen in 4.1, gevolgd door een meer gedetailleerde uiteenzetting van het codegeneratiemechanisme dat deel uitmaakt van de suite in 4.2.

4.1 Overzicht

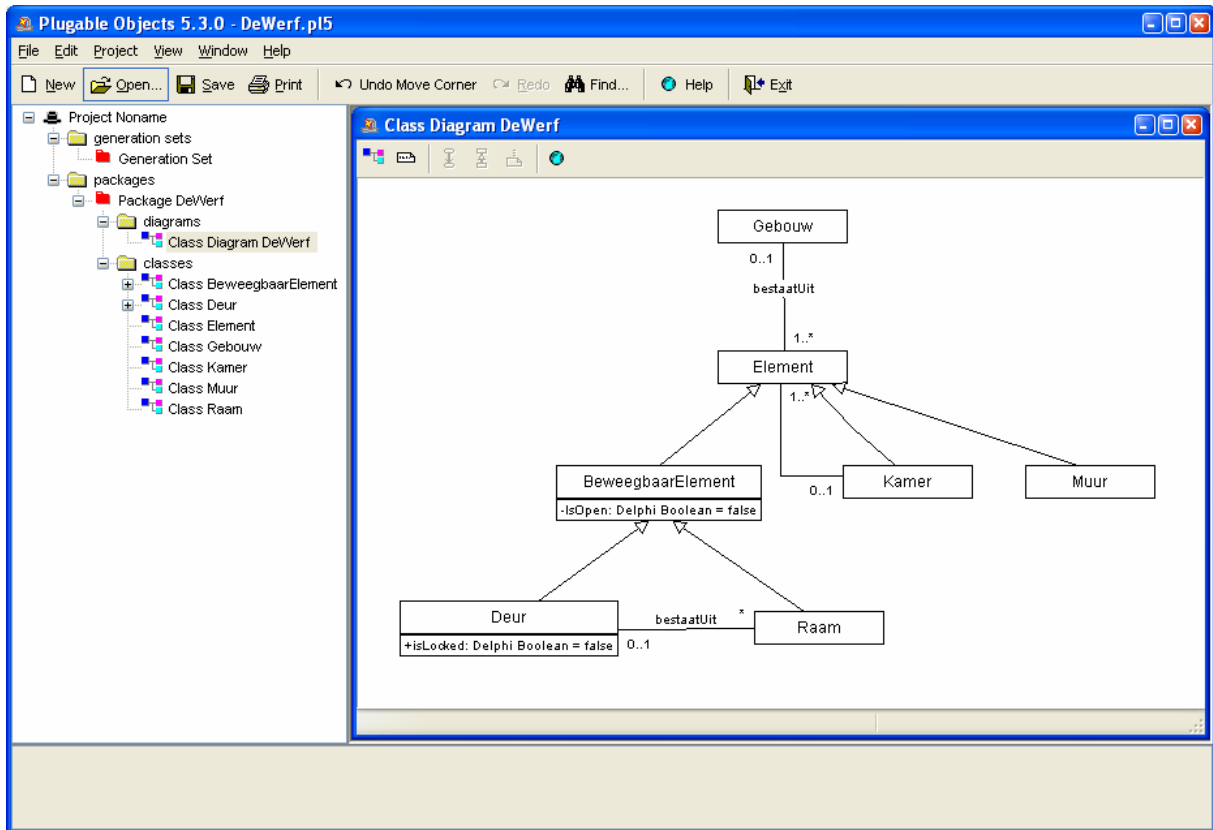
De door E2S ontwikkelde applicatiesuite voor de implementatie van de HOORA-methode, bestaat uit: HAT (*HOORA Analysis Tool*, een modelleringstoepassing), *Plugable Objects* (een bijhorende codegenerator), *HOORA Document Builder* (een documentgenerator) en *HOORA Profile Editor* (profielbewerker).

HAT is het HOORA analysewerkzeug, dat een directe ondersteuning biedt van de HOORA concepten. Het is een handig werktuig voor het traceren van vereisten en ondersteunt de UML 1.3 notatie. HAT moet echter puur als analysehulpmiddel gezien worden doordat het geen mogelijkheden biedt om uit de bekomen diagrammen code te genereren. Nadat het ontwerp van een applicatie voltooid is, kan een programmeur het resultaat dus niet rechtstreeks gebruiken voor het ontwikkelen van een applicatie. Indien men enkel over HAT kan beschikken, moet een UML-ontwerpmodel dus “met de hand” naar bruikbare programmeercode geconverteerd worden. HAT wordt getoond in Figuur 4.1, met een analysemodel als voorbeeld. Aan de linkerkant worden modelementen weergegeven in een boomstructuur, aan de rechterkant het diagram waar deze elementen toe behoren.



Figuur 4.1: HAT

Om deze leemte te vullen, werd aan HAT een codegenerator toegevoegd. De combinatie van HAT met deze codegenerator werd voorzien als een aparte applicatie met de naam Pluggable Objects. Waar de gebruiker bij het aanmaken van een project in HAT een gebruikersprofiel kan kiezen, maakt Pluggable Objects gebruik van een hard in de broncode ingebakken profiel. Met dit profiel wordt bepaald hoe elementen op het UML-diagram (*packages*, klassen, ...) eruit zien en wordt extra informatie zoals constanten voor codegeneratie voorzien. Het ingebakken profiel beperkt ook de modelleringsmogelijkheden om het genereren van code uit het gegeven diagram in alle omstandigheden mogelijk te maken. Zo is het voorlopig bijvoorbeeld niet mogelijk om meervoudige overerving of n-op-m relaties te modelleren. Voorlopig werd het genereren van code via Pluggable Objects enkel mogelijk gemaakt naar de programmeertalen die binnen E2S meest gebruikt worden, nl. C++ en Delphi. De code die gegenereerd wordt door Pluggable Objects maakt gebruik van een vaste set modules, die basismethodes voorzien voor gegevenstoegang en -opslag. Op deze manier worden ook verschillen tussen doelplatformen en opslagmethodes (gegevensbank, bestanden, ...) afgehandeld. De gebruiker moet echter aan de gegenereerde code nog applicatiespecifieke code (de eigenlijke logica van de toepassing) toevoegen.

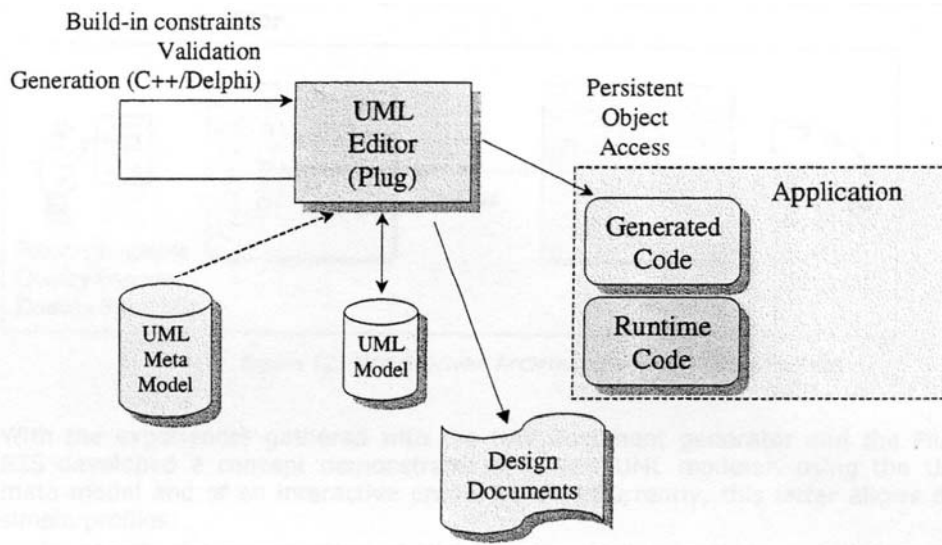


Figuur 4.2: Pluggable Objects

In Figuur 4.2 wordt een overzicht van deze applicatie aangeboden. De applicatie ziet er hetzelfde uit als HAT. Bemerkt echter dat in de boomstructuur aan de linkerkant nu een *Generation Set*, zijnde een bestand met instellingen voor codegeneratie, weergegeven wordt. De verschillen tussen HAT en Pluggable Objects (*PO*) zijn:

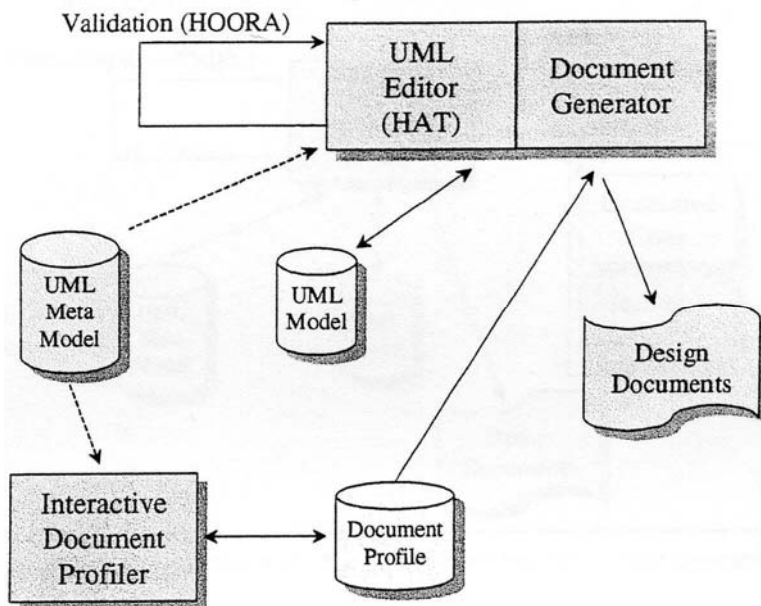
- *PO* bevat ingebouwde beperkingen op UML-modelelementen. Daardoor kan de gebruiker enkel gereduceerde UML-modellen gebruiken (zie ook hoger).
- *PO* laat de definitie toe van attributen met een machineafhankelijk implementatietype (zoals *integers*, *floats* en *Booleans*)
- *PO* voert een syntactische controle uit van het UML-model naar mogelijke problemen voor codegeneratie
- *PO* heeft aandacht voor het modelleren van persistente objecten van het softwareproject (zoals opslaan in een gegevensbank)

Pluggable Objects past in de applicatiesuite zoals op Figuur 4.3 kan gezien worden. *PO* leest een UML-metamodel en UML-model om hier code uit te genereren. Daarnaast gebruikt een aantal hard ingebakken beperkingen voor de generatie van deze code naar Delphi of C++. De generatie van ontwerpdocumenten wordt nader toegelicht in wat volgt.



Figuur 4.3: E2S codegenerator

Om het documenteren van projecten handiger en eenvoudiger te maken, werd aan HAT (en aan Plugable Objects) een documentgenerator toegevoegd. De documentgenerator gebruikt een documentprofiel, waarmee de gebruiker de inhoud van het gecreëerde document kan beschrijven en de volgorde waarin de data van het UML-model zou moeten verschijnen. Om een documentprofiel aan te maken of te bewerken, biedt E2S ook een *document profiler* aan. De documentgenerator verhoudt zich tot de rest van de suite zoals gezien kan worden op Figuur 4.4. De documentgenerator kan gezien worden als een onderdeel van HAT en Plugable Objects, die via deze applicaties toegang tot het UML-metamodel en UML-model verkrijgt. Daarnaast gebruikt de documentgenerator een documentprofiel voor de opmaak van gegenereerde documenten, en geeft als uitvoer de ontwerpdocumenten.



Figuur 4.4: E2S documentgenerator

Met de komst van MDA rees er interesse naar de mogelijkheid voor implementatie hiervan in de HOORA-suite. De voordelen hiervan, zoals aangehaald onder 2.2.4, zijn duidelijk. Specifiek is er vraag naar enkele nieuwe mogelijkheden die met de huidige software moeilijk zijn, zoals het implementeren van n-op-m relaties tussen objecten. Daarnaast kan met MDA ook gemakkelijk

codegeneratie naar nieuwe talen, zoals ondermeer C#, geïmplementeerd worden, door het voorzien van een set transformatieregels naar deze nieuwe doeltaal. Momenteel biedt de HOORA-suite al een beperkte vorm van MDA aan die zich hoofdzakelijk beperkt tot codegeneratie uit het bronmodel. Voor deze codegeneratie wordt geen specifieke taal gebruikt maar de taal waarin de suite geïmplementeerd is. Een specifieke transformatietaal zou de mogelijkheden voor codegeneratie kunnen uitbreiden en meer flexibel maken. Zo kan de suite verder uitgebreid worden naar een meer volledige toepassing van MDA.

4.2 Codegeneratie

In de huidige E2S-applicatiesuite zijn mechanismen voor codegeneratie hard ingebakken. Dit betekent dat zonder de broncode van Plugable Objects te hercompileren, deze mechanismen niet kunnen gewijzigd of uitgebreid worden. Met de implementatie van MDA zou dit wel mogelijk worden door patronen die terug te vinden zijn in de codegeneratie, extern als transformatieregels te specificeren. De codegenerator van de huidige software zou voor dit doel omgevormd worden naar een modeltransformator, waarbij zowel het bronmodel als de code (het doel van de transformatie) als modellen kunnen beschouwd worden. Hiervoor moeten we de codegenerator dieper bestuderen om op zoek te gaan naar patronen in de gegenereerde code. Deze patronen kunnen namelijk omgevormd worden naar MDA-transformatieregels, zodat implementatie van MDA mogelijk wordt. De resultaten van deze studie is wat in deze paragraaf beschreven wordt.

Met Plugable Objects kunnen in een klassendiagram volgende elementen toegevoegd worden die relevant zijn voor codegeneratie:

- klassen
- attributen
- associaties tussen klassen
- generalisaties en specialisaties tussen klassen
- operaties

Commentaar en afhankelijkheden (*dependencies*) kunnen ook gemodelleerd worden, maar worden niet gebruikt voor codegeneratie.

De codegenerator die in het kader van deze thesis bestudeerd werd, genereert Delphi-code. De Delphi-syntax wordt gedetailleerd beschreven in [CGL97]. Voor iedere klasse op het klassendiagram, wordt in de resulterende broncode één unit aangemaakt. Een *unit* in Delphi kan beschouwd worden als een module met daarin één of meerdere klassen, variabelen en functies gedefinieerd. In iedere unit wordt door HOORA één klasse gedeclareerd en geïmplementeerd die overeenkomt met de klasse uit het klassendiagram. Delphi maakt een onderscheid tussen functies en procedures. Een functie komt overeen met een inspector en heeft een terugkeerwaarde. Een procedure heeft geen terugkeerwaarde en komt overeen met een mutator. Klassen kunnen ook attributen hebben, maar dit wordt door HOORA als een waarde in een databank geïmplementeerd (zie verder). Een associatie van een klasse met een andere wordt analoog aan Java geïmplementeerd als een attribuut met als type de geassocieerde klasse. Hiervoor importeert de codegenerator het bronbestand van de geassocieerde klasse in de gegenereerde unit. Voor alle attributen voegt de codegenerator een inspector en mutator toe. In het geval van een meervoudige associatie wordt geen mutator gegenereerd. Voor associaties wordt verder een “zet op nul”-methode toegevoegd en voor attributen een “controleer waarde”-methode. Deze laatste controleert attributen op onder- en bovengrenzen en indien gewenst (dit kan in Plugable Objects aangegeven worden) op verschillend van nul en uniek. Ook overerving is mogelijk in Delphi. Indien een klasse gemodelleerd wordt als afgeleid van een andere klasse, wordt de klasse ook in de broncode afgeleid en wordt het bronbestand van die klasse geïmporteerd in de unit.

Voor iedere unit wordt de gegenereerde code in 4 bestanden opgesplitst. Deze bestanden zijn een .dcl (interface declaratie)-bestand, een .imp (implementatie)-bestand, een .fid-bestand (met declaraties van constanten) en een .pas (Pascal)-bestand. Deze laatste bevat de unit en klasse zelf, overeenkomend met

de klasse op het UML-diagram, en voegt de code in de andere bestanden samen. Deze vier verschillende bestanden worden hierna verder toegelicht.

Om gegevenstoegang platformonafhankelijk te maken, wordt toegang tot attributen verkregen via een databank. Attributen van een klasse uit het klassendiagram worden niet geïmplementeerd als attributen van de Delphi-klasse, maar als waarden in een databank. Vervolgens worden de attributen van een klasse vervangen door een sleutel in een databank. De sleutel voor elke variabele wordt bijgehouden als een constante die gedeclareerd wordt in het .fid-bestand. Bij lezen van en schrijven naar attributen via inspectors en mutators wordt gelezen van en geschreven naar de waarden in deze databank. Zo wordt platformonafhankelijkheid verkregen doordat voor ieder platform dat toegang tot een databank ondersteunt, toegang tot deze variabelen mogelijk is.

Om de beschrijving van het mechanisme volledig te maken, kan het nog interessant zijn te vermelden dat de bronelementen (klassen, attributen, ...) voor codegeneratie gevonden worden door exhaustief zoeken op het model, en dus niet door een vorm van (declaratieve) patrooncorrelatie op het model.

Hierna werden enkele fragmenten ingevoegd van de resulterende code voor het diagram dat op Figuur 4.2 te zien is, meer bepaald van de Deur- en Raam-klasse. De volledige code van deze twee klassen is terug te vinden in Appendix A.

4.2.1 Declaratiebestand

In het declaratiebestand komt de declaratie van publieke methodes van de klasse en de Delphi-unit die ermee overeenkomt. Een voorbeeld voor de Deur-unit wordt gegeven in Figuur 4.5. Voor het attribuut `isLocked` van deze unit werd een mutator (`SetIsLocked`) en een inspector (`GetIsLocked`) gegenereerd. De associatie Deur-Raam werd als 1-op-n gemodelleerd (een deur kan meerdere ramen hebben) en de inspector van deze associatie (`GetAllRaam`) keert dan ook een lijst van objecten terug. Merk op dat voor een Booleaanse waarde geen “controleer waarde”-methode gegenereerd wordt aangezien deze toch enkel de waardes `true` of `false` kan aannemen.

```
public

class function TableIndex: Integer; override;
procedure SetIsLocked(value: Boolean);
function GetIsLocked: Boolean;
function GetAllRaam: ObjectList;
class procedure Initialize(pTblInfo: CTTableInfo; szIDFieldName:
AnsiString = '');
```

Figuur 4.5: Fragment van het declaratiebestand van de Deur-unit

Een voorbeeld van een “zet op nul”-methode vinden we bij de Raam-unit, waarvan hieronder de gegenereerde code gegeven is in Figuur 4.6. Merk op dat “...” duidt op resterende maar gelijkaardige code, die hier niet weergegeven wordt om de essentie te behouden.

```

public
class function TableIndex: Integer; override;
procedure SetDeur(pObj: Pointer);
procedure DropDeur;
function GetDeur: Pointer;
...

class procedure Initialize(pTblInfo: CTableInfo; szIDFieldName:
nsiString = '');

```

Figuur 4.6: Fragment van het declaratiebestand van de Raam-unit

De DropDeur-procedure zet de associatie met Deur op null. Merk op dat hier naar de “1”-kant van de 1-op-n-relatie tussen Deur en Raam gekeken wordt, waardoor in deze klasse wel voor de Deur-associatie een mutator voorzien wordt (SetDeur).

4.2.2 Implementatiebestand

Het implementatiebestand bevat de implementatie van de methodes die in het declaratiebestand (zie 4.2.1) gedeclareerd werden. Voor zowel inspectors (Delphi-functies) als mutators (Delphi-procedures) wordt lichaam voorzien.

```

class function Cls_Deur.TableIndex;
begin
    Result := CID_TBL_DEUR;
end;

procedure Cls_Deur.SetIsLocked;
begin
    SetBooleanValue(FID_TBL_DEUR_ISLOCKED, value);
end;
...

```

Figuur 4.7: Fragment van het implementatiebestand van de Deur-unit

4.2.3 Constantendeclaraties

In Figuur 4.8 wordt het bestand voor constantendeclaraties weergegeven. Het .fid-bestand bevat voor elk attribuut van een klasse de databanksleutel, gedefinieerd als een constante. Zo komt de Boolean isLocked overeen met de databanksleutel FID_TBL_DEUR_ISLOCKED. De implementatie van de databank kan gevarieerd worden van een SQL-databank tot een simpel bestand dat een databank simuleert, zonder dat aan de gegenereerde code iets moet gewijzigd worden. In 4.2.4 wordt op enkele plaatsen naar een Factory-object verwezen, hierin kan de methode voor gegevenstoegang eenvoudig gewijzigd worden.

```
const
START_FID_DEUR = END_FID_BEWEEGBAARELEMENT;
FID_TBL_DEUR_ISLOCKED = START_FID_DEUR + 1;
END_FID_DEUR = START_FID_DEUR + 1;
```

Figuur 4.8: Fragment van het bestand voor constantendeclaraties voor de Deur-unit

4.2.4 Pascalbestand

In Figuur 4.9 wordt het Pascalbestand voor de Deur-unit weergegeven. Het Pascal-bestand bevat de declaratie van de unit en de Delphi-klasse die overeenstemmen met de klasse uit het klassendiagram. In dit bestand worden de vorige bestanden (4.2.1, 4.2.2 en 4.2.3) samengevoegd. Verder wordt aan de klasse in de unit een constructor, destructor en initialisatiemethode toegevoegd. Bij dit bestand vraagt het invoegen van afhankelijke modules (`uses ...`) extra aandacht. De modules die ingevoegd worden zijn:

- Basismodules voor gegevenstoegang en dergelijke
- Module waar de huidige klasse eventueel van afgeleid is
- Modules van klassen waar een associatie mee bestaat

Een afhankelijke module mag slechts eenmaal ingevoegd worden. Dit betekent dat voor een klasse die afgeleid is van een superklasse waar ook een associatie mee bestaat, de overeenstemmende module slechts eenmaal mag ingevoegd worden.

```

unit ClDeur;

interface

uses
  Classes, ObjList, DbObject, ClBeweegbaarElement;

{$I DbDeur.fid}

type
  Cls_Deur = class(Cls_BeweegbaarElement)
  public
    constructor Create(nObjID: Integer; pFactory: CTBaseFactory =
nil); override;
    destructor Destroy; override;

    private
      procedure InitAttributes;
{$I DbDeur.dcl}

  end { class Cls_Deur };

implementation
uses
...

```

Figuur 4.9: Fragment van het Pascal-bestand voor de Deur-unit

4.2.5 Codegeneratiepatronen

Door een analyse van de informatie uit dit hoofdstuk kunnen de codegeneratiepatronen voor de huidige functionaliteit bepaald worden.

- Voor iedere klasse in het PSM wordt een Delphi-unit aangemaakt met hierin de declaratie van een Delphi-klasse (zie 4.2.4), waarbij afhankelijke units slechts eenmaal ingevoegd worden.
- Voor ieder attribuut wordt een sleutel voor een databankelement gemaakt met een publieke inspector en mutator, en “controleer waarde”-methode, behalve indien het over een Boolean gaat (zie 4.2.1)
- Voor iedere associatie wordt een publieke inspector toegevoegd, “zet op nul”-methode en een mutator indien het niet om een meervoudige associatie gaat (zie 4.2)
- Voor iedere Delphi-klasse wordt een constructor (maakt gebruik van een Factory-object), initialisatiemethode en destructor toegevoegd (zie 4.2.4)

Deze patronen worden in Hoofdstuk 6 tot transformatieregels omgevormd. Om niet alleen de huidige functionaliteit te evenaren, maar ook de mogelijkheid te onderzoeken tot het ondersteunen van toegevoegde functionaliteit, worden hier een aantal transformatieregels aan toegevoegd. Zo is er onder meer de vraag naar de omvorming van associatieklassen naar klassen. Dit is niet mogelijk met de huidige codegenerator, door de beperkingen van de implementatietaal van de suite als huidige transformatietaal.

Ook het omvormen van n-op-m associaties tussen klassen wordt niet ondersteund. Het zou interessant zijn te onderzoeken of dit met een MDA-transformatietaal wel kan gerealiseerd worden.

De toegevoegde functionaliteit die nagegaan wordt is dus:

- Omvormen van associatieklassen naar klassen
- Ondersteunen van n-op-m associaties tussen klassen

Geavanceerde UML-constructies worden niet met de huidige functionaliteit ondersteund. Zo wordt meervoudige overerving met de huidige suite niet ondersteund. Deze beperking wordt ingebouwd met behulp van de *HOORA Profile Editor*. Het zou mogelijk zijn te onderzoeken of een transformatietaal meervoudige overerving kan transformeren naar enkelvoudige overerving, maar dit lijkt iets te ver gezocht in deze context.

Hoofdstuk 5

MDA Transformatiestandaard

Uit de algemene beschrijving van Model Driven Architecture in Hoofdstuk 2, de beschrijving van gebruikte standaarden in Hoofdstuk 3 en de beschrijving van de doelstellingen die E2S aan dit systeem stelt in Hoofdstuk 4, kunnen nu de vereisten voor de transformatietaal bepaald worden.

Eerst bepalen we enkele zaken waar een generische modeltransformator over zou moeten beschikken:

- een beschrijving (metamodel) van de brontaal, zodat hieruit kan gelezen worden
- een beschrijving (metamodel) van de doeltaal, zodat hiervoor een model of code kan gegenereerd worden
- een model gespecificeerd in de brontaal
- een model gespecificeerd in de doeltaal
- brontaalcondities: voor elk element in het bronmodel een invariant die de condities definieert die moeten gelden voordat de transformatie op dit element mag toegepast worden
- doeltaalcondities: voor elk element in het doelmodel een invariant die de condities definieert die moeten gelden voor de transformatie op dit element mag toegepast worden
- een set transformatieregels, die definiëren hoe één of meerdere elementen uit het bronmodel op één of meerdere elementen uit het doelmodel moeten afgebeeld worden
- een aantal optionele transformatieparameters, bijvoorbeeld constanten gebruikt in de generatie van het doel, of een waarde die definieert of een bronmodel kan heropgebouwd worden uit het doel

Deze en andere vereisten werden door OMG samengebracht in een *Request For Proposal*, kortweg RFP. Deze RFP (zie [ad/2002-04-10]) bepaalt de vereisten waaraan kandidaat-transformatietalen dienen te voldoen. De MDA-transformatietaal werd door MDA in een algemenere context geplaatst. Naast transformaties bepaalt OMG namelijk dat een transformatietaal ook zoekopdrachten (*queries*) en visies (*views*) moet kunnen definiëren. Daarom vraagt deze RFP naar een *Query/View/Transformation*-taal. Hierna wordt deze RFP verder in detail beschreven.

5.1 Query/View/Transformation

De vraag naar de standaardisatie voor een MDA-transformatietaal gaat samen met de introductie van nieuwe versies van een aantal bestaande specificaties van OMG. Dit wordt uitgelegd in 5.1.1. De vereisten die gesteld worden aan voorstellen ingediend als antwoord op de RFP worden uitgelegd in 5.1.2 en 5.1.3. Verder schetst OMG een aantal evaluatiecriteria van voorstellen. Deze worden in 5.1.4 aangevuld met evaluatiecriteria die belangrijk kunnen zijn in de context van deze thesis.

5.1.1 Verband met bestaande specificaties van OMG

Zoals hoger aangehaald, moeten voorgestelde transformatietalen inwerken op modellen die voldoen aan een metamodel dat gespecificeerd is in MOF 2.0. Dit is niet enkel een verband met een bestaande specificatie, maar ook een vereiste van deze RFP. Daarom wordt MOF 2.0 ook in 5.1.2 vermeld.

Verder werd reeds aangehaald dat MOF OCL gebruikt om beperkingen op modelementen en de semantiek van een operatie te definiëren. MOF 2.0 en UML 2.0 hebben een deels gemeenschappelijke infrastructuur en OCL 2.0 wordt gedefinieerd als onderdeel van UML 2.0. Deze drie specificaties zijn dus verwant. Dit betekent dat OCL in MDA kan hergebruikt worden voor het uitdrukken van beperkingen en zoekopdrachten. In voorstellen kan OCL een belangrijke rol spelen in zoekopdrachten op metadata.

UML *Action Semantics* is een taal die zich richt op de specificatie van acties in UML. Tegenwoordig is er een leemte voor de specificatie van acties in een klassendiagram. Acties, meer bepaald datatoegang, testen, sequenties, oproepen van operaties en dergelijke kunnen in een communicatie- of sequentiediagram, maar niet in een klassendiagram gemodelleerd worden. *Action Semantics* biedt hiervoor een oplossing. Het bevat mechanismen voor het manipuleren van de toestand waarin een object zich bevindt. Het is een taal die net als objectgeoriënteerde talen attributen kan wijzigen, associaties kan zetten en dergelijke. Daardoor is ze gelijkaardig aan talen als Java en C++, maar wordt gebruikt met het oog op modelleren. In het kader van MDA zou dit modelleren van acties echter heel handig kunnen zijn. Door middel van een transformatie kan UML *Action Semantics* dan omgezet worden naar bijvoorbeeld een objectgeoriënteerde taal. Daarom bepaalt de RFP dat dit mechanisme moet overwogen worden voor hergebruik.

Verder vermeldt de RFP verbanden met enkele bestaande transformatiemechanismen. Hiertoe behoort onder meer CWM (zie 3.4.2). CWM heeft ondermeer een transformatiemodel. Verder haalt de RFP nog enkele concepten uit CWM aan die interessant kunnen zijn voor hergebruik in MDA.

5.1.2 Functionele vereisten

De belangrijkste en te verwachten functionele vereiste die aan een transformatietaal kan gesteld worden, is dat ze toelaat transformatiedefinities op te stellen. OMG bepaalt dat transformatiedefinities relaties zullen beschrijven tussen een MOF metamodel A dat als bron optreedt, en een MOF metamodel B dat als doel optreedt. Deze transformatiedefinities kunnen gebruikt worden om een element uit het doelmodel dat aan metamodel B voldoet te genereren uit een element van het bronmodel dat aan metamodel A voldoet. De taal voor transformatiedefinities zal toelaten om volledig te specificeren hoe automatisch een doelelement uit een bronelement te genereren. Verder bepaalt de RFP dat de taal voor transformatiedefinities bij voorkeur moet declaratief zijn. Dit maakt het mogelijk transformaties op te stellen die veranderingen in het bronmodel onmiddellijk naar het doelmodel transformeren, zonder dat het volledige doelmodel dient heropgebouwd te worden.

Voorstellen moeten ook een taal definiëren om zoekopdrachten op modellen op te stellen. De taal voor zoekopdrachten moet zoekopdrachten voor de selectie van modelementen gemakkelijker maken, en dit tevens voor het selecteren van modelementen die de bron van een transformatie moeten worden.

De taal voor transformatiedefinities moet ook toelaten een visie op het model te maken. Met een visie kan een deelverzameling van het model afgebakend worden die gevisualiseerd dient te worden.

Alle mechanismen gespecificeerd in een voorstel moeten inwerken op modellen die voldoen aan een metamodel dat met MOF 2.0 gedefinieerd is. Dit is hoger reeds aangehaald in 5.1.1. De RFP bepaalt verder dat niet alleen modellen die door een transformatie gerelateerd zijn, aan een metamodel dienen te voldoen; ook de syntax van een voorgestelde transformatietaal dient als MOF 2.0 metamodel gedefinieerd te worden.

5.1.3 Non-functionele vereisten

Naast functionele vereisten stelt de OMG ook enkele non-functionele vereisten aan voorgestelde transformatietalen. Makers van voorstellen zijn vrij deze te volgen, al kunnen deze vereisten erg handig zijn.

Zo vraagt de OMG dat voorstellen mogelijk transformaties moeten ondersteunen die kunnen uitgevoerd worden in twee richtingen. Zo wordt het concept van bron- en doelmodel van een transformatie op een hogere abstractie van gerelateerde modellen gebracht. Hiervoor zijn twee mogelijke benaderingen:

- Transformaties kunnen symmetrisch gedefinieerd worden. In dit geval vervalt het concept van bron en doel volledig, en heeft één transformatie zoals vermeld een aantal gerelateerde modellen.
- Twee transformatiedefinities kunnen het inverse zijn van elkaar. In dit geval blijft het concept van bron en doel voor elke transformatie op zich bestaan, maar wordt het doel van één transformatie de bron voor de inverse transformatie.

Voorgestelde transformatietalen kunnen ook de uitvoering van transformaties ondersteunen waarbij het doelmodel hetzelfde is als het bronmodel, om zo transformaties toe te laten om bestaande modellen bij te werken. Dit betekent dat een transformatietaal zou toelaten om de bron van een transformatie te vervangen door het doel ervan.

Voorgestelde transformatietalen kunnen verder mechanismen invoeren voor het hergebruiken en uitbreiden van generische transformaties. Dit geval doet zich voor indien een transformatie gedefinieerd is tussen enkele algemene metaklassen van een model. De transformatietaal zou dan toelaten om deze transformatie te specialiseren naar een transformatie tussen subklassen van deze subklassen. Bovendien houdt dit in dat, naar het objectgeoriënteerd model, significante stukken van transformaties niet telkens hoeven herhaald te worden.

Voor dit eindwerk worden transformaties tussen klassendiagrammen beschouwd. In klassendiagrammen komen met grote waarschijnlijk algemene en afgeleide klassen voor, waardoor we dit als een belangrijke functionele vereiste kunnen beschouwen.

Ook kan opspoorbaarheid, *traceability*, ondersteund worden van transformaties die uitgevoerd worden tussen modelementen van bron- en doelmodel. Zo kan bij een onderbroken transformatie nagegaan worden in hoeverre de transformatie precies uitgevoerd werd, en op welk punt in de uitvoering de transformatie onderbroken werd. Zo kan afgeleid worden in welke toestand de uitvoer van een transformatie zich bevindt. Hiervoor wordt ook het ondersteunen voorgesteld van transacties voor transformaties, waarbij bepaalde delen van een transformatie gedefinieerd worden als geschikt voor een atomaire actie tijdens de uitvoering. Dit lijkt in de context van dit eindwerk echter een minder belangrijke vereiste.

Mogelijk kunnen ook optionele transformatieparameters ondersteund worden die niet in het doelmodel opgenomen zijn. Deze kunnen als invoer voor een transformatie gebruikt worden om het doelmodel te genereren.

Bij de bespreking van voorgestelde transformatietalen (zie 5.2), zullen we zien hoe voorgestelde transformatietalen dit oplossen. Meestal gebeurt dit door een extra model als invoer voor de transformatie te gebruiken. Zo is dit mechanisme voor optionele transformatieparameters transparant met andere transformaties. In de context van dit eindwerk kan deze vereiste gebruikt worden, al komt dit niet echt op de voorgrond door de oplossing die net besproken werd.

5.1.4 Evaluatiecriteria

De evaluatiecriteria voor voorgestelde transformatietalen kunnen we opsplitsen in evaluatiecriteria van OMG enerzijds, en evaluatiecriteria die belangrijk zijn in de context van dit eindwerk anderzijds. Omdat deze criteria gedeeltelijk overlappen, en andere criteria dan weer minder belangrijk zijn voor ons, worden deze hier samen besproken.

Een transformatietaal moet transformaties kunnen ondersteunen van een relatief hoge complexiteit. Een eenvoudige transformatie is een transformatie die één enkel element in het bronmodel transformeert naar één enkel element in het doelmodel. Een voorbeeld van een meer complexe transformatie is bijvoorbeeld het transformeren van een databank met objectgeoriënteerd model naar een databank met relationeel model. Ook dit moet mogelijk zijn met een transformatietaal.

Verder werd reeds in 5.1.3 vermeld dat transformatietalen het hergebruiken en uitbreiden van transformatietalen moeten ondersteunen. Doordat dit eindwerk voornamelijk gericht is op transformaties van klassendiagrammen, en uitbreidingsmogelijkheden hier voor klassendiagrammen zeker van belang zijn, gaan we transformatietalen ook op dit criterium beoordelen.

De manier waarop de invoer of uitvoer voor een transformatie geselecteerd wordt is eveneens een belangrijk criterium. Twee mogelijke manieren zouden zijn het exhaustief zoeken op het model tot invoer van een bepaalde vorm gevonden wordt, of een declaratieve formulering van het invoer- of uitvoerpatroon van de transformatie. In het eerste geval wordt een imperatief model gevolgd. Hoewel dit voor de meeste programmeurs meer vertrouwd is, lijkt dit ons een zeer verboze manier. Een transformatietaal moet dus ook op dit criterium geëvalueerd worden.

Algemeen kunnen we zeggen dat de mate waarin en de manier waarop een voorgestelde transformatietaal aan de functionele en non-functionele vereisten voldoet, een zeer belangrijk evaluatiecriterium is. Dit gaat immers de bruikbaarheid van de taal in reële situaties bepalen. De eisen die E2S aan de transformatietaal stelt, zijn in overeenstemming met de vereisten van OMG. Bovendien beoogt E2S in zekere mate het implementeren van een gekozen taal in hun applicatiesuite, en is het interessant als dit de uiteindelijke standaard wordt. Verder kunnen we hiervoor de opmerkingen in acht nemen in verband met het al of niet relevant zijn van elke vereiste van OMG voor deze thesis.

Als we een taal kiezen, moeten we deze ook effectief gaan gebruiken om er transformatieregels mee te schrijven. Daarom is het belangrijk dat de gekozen taal geen te grote instapdrempel heeft opdat deze voor dit eindwerk op voldoende korte termijn kan aangeleerd en gebruikt worden. Factoren die dit bepalen zijn de complexiteit van de taal zelf, maar ook de kwaliteit en de gedetailleerdheid van het document waarin de transformatietaal beschreven wordt zijn bepalend. Er is immers geen ruimte om de auteurs te raadplegen voor duidelijkheid in verband met bepaalde zaken.

5.2 RFP voorstellen

De RFP beschreven in 5.1 leidde enkele bedrijven en universiteiten geïnteresseerd in Model Driven Architecture, tot het indienen van een voorstel of *submission*. Enkele van deze voorstellen worden hierna toegelicht en beoordeeld. Voor de beoordeling werd ondermeer [GGKH] gebruikt.

5.2.1 Kennedy Carter

Kennedy Carter is een Brits bedrijf dat zich specialiseert in geautomatiseerd ontwerp van software waarvoor prestaties en betrouwbaarheid zeer belangrijk zijn. Hiervoor maakt Kennedy Carter gebruik van xUML, voluit *Executable UML*. Dit is een uitvoerbare versie van de UML-standaard. Hiermee kunnen werkende systemen gemodelleerd, gesimuleerd en getest worden zonder deze te implementeren in een specifieke programmeertaal. Het kan dan ook niet vreemd heten dat het voorstel dat dit bedrijf uitbracht ter ondersteuning van MDA, gebaseerd is op xUML. Het voorstel wordt beschreven in [KC02].

Inhoud van het voorstel

Voor de implementatie van xUML gebruikt Kennedy Carter een implementatie van UML *Action Semantics*. Deze taal werd reeds beschreven in 5.1.1, en de implementatie van Kennedy Carter heet *Action Specification Language* (ASL). Een xUML-model is een UML-klassendiagram aangevuld met specificaties in ASL.

Het ontwerpen van software gebeurt bij Kennedy Carter door het ontwerpen van een aantal xUML-modellen van subsystemen. Het voordeel hiervan is dat deze modellen afzonderlijk kunnen worden uitgevoerd, gesimuleerd en getest. Verschillende deelmodellen kunnen met elkaar verbonden worden om meer complexe systemen te vormen via een volgende ASL-uitdrukking. Doordat deze verbindingen ook uitgedrukt worden met ASL vormen deze verbindingen op zich ook een xUML-model.

Kennedy Carter ziet elk van de modellen van deze subsystemen, en modellen die met elkaar verbonden zijn als een PIM. Deze systemen zijn immers volledig gespecificeerd onafhankelijk van een programmeertaal. Het verbinden van een subsysteem met een ander wordt gezien als een transformatie van PIM naar PIM. Ieder PIM, dus ieder model van een subsysteem of model van een geheel van subsystemen, kan met een transformatie van PIM naar PSM en vandaar naar code omgezet worden.

Voor het formuleren van zoekopdrachten, creëren van visies en definiëren van transformaties stelt Kennedy Carter telkens dezelfde taal voor. Waar andere voorstellen eventueel aparte talen voorstellen voor deze mechanismen, stelt Kennedy Carter telkens ASL voor.

De voordelen hiervan voor het ontwikkelen van objectgeoriënteerde software zijn groot. Doordat gedetailleerde, volledige modellen van een applicatie worden ontwikkeld voor deze te implementeren, dwingt deze methodologie een uitgebreide analysefase af. Daardoor lijkt het voor de hand te liggen dat geïmplementeerde systemen beter ontworpen zijn. Bovendien wordt door het modelleren van subsystemen op zich een logische scheiding van verantwoordelijkheid gelegd in de verschillende delen van het ontwerp. Bovendien zijn modellen, doordat ze kunnen uitgevoerd, worden niet langer onderhevig aan interpretatie door individuele programmeurs. De vraag rest echter of deze aanpak ook bruikbaar is in de context van MDA. Dit wordt nader beschouwd in de volgende paragraaf.

Evaluatie

Na in 5.1.4 de evaluatiecriteria voor een transformatietaal opgesteld te hebben, kunnen we dit voorstel naast deze criteria houden en bepalen in welke mate dit voorstel aan onze criteria voldoet.

- **Complexiteit:** ASL is een volledige taal van imperatieve natuur. In overeenstemming met imperatieve programmeertalen is het daardoor mogelijk transformaties van om het even welke graad van complexiteit op te stellen. Het verschil tussen een relatief eenvoudige en een complexe transformatie zou dan de uitvoerigheid van de definitie van de transformatie zijn.
- **Hergebruik:** doordat ASL in dit geval zo nauw verbonden is met het transformatiemechanisme, berusten de mogelijkheden voor hergebruik en uitbreiding van transformatieregels dan ook op de mogelijkheden die ASL hiervoor biedt. ASL biedt een aantal mechanismen voor het groeperen van functionaliteit in operaties of klassen en voor overerving van klassen. Aan dit criterium is dus voldaan.
- **Selectie van invoer en uitvoer:** dit is meteen het grootste probleem in dit voorstel. Specificaties in ASL zijn door hun imperatieve natuur zo goed als specificaties in om het even welke andere imperatieve programmeertaal. Dit zou er toe leiden dat de specificatie van invoer- en uitvoerelementen in transformatieregels hun eigen methoden moeten implementeren voor navigeren door het bronmodel, en aanpassen van het doelmodel van een transformatie. Dit betekent dat transformatiedefinities al snel heel uitgebreid zouden worden om simpele functionaliteit te implementeren. Bovendien worden transformatiedefinities hierdoor zeer onderhevig aan fouten: wanneer het bronmodel aangepast wordt, moet ook de methode van navigeren door dit bronmodel aangepast worden.

- Voldoen aan functionele en non-functionele vereisten (zie 5.1.2 en 5.1.3): In de functionele vereisten werd bepaald dat de taal voor transformatiedefinities bij voorkeur declaratief dient te zijn. Dit is duidelijk niet het geval. Bij andere voorstellen (zie verder) worden talen voorgesteld die deels declaratief zijn, en een declaratieve taal gebruiken voor navigatie door modellen of visies. Ook dat is in dit voorstel niet het geval en maakt ASL daardoor als transformatietaal, volgens mijn mening, weinig bruikbaar.
- Instapdrempel: doordat veel programmeurs het meest affiniteit hebben met imperatieve programmeertalen, kunnen we veronderstellen dat de instapdrempel voor het volgen van dit voorstel vrij laag is. De instapdrempel blijft beperkt tot het aanleren van de syntax van ASL.

Samengevat kunnen we zeggen dat hoewel Kennedy Carter een interessante benadering heeft op de implementatie van MDA, ASL niet voldoet aan onze noden. Het is duidelijk dat met dit mechanisme transformaties kunnen geïmplementeerd worden zonder een al te hoge leercurve. De nadelen van beperkte navigatiemogelijkheden en slechte onderhoudbaarheid van transformatieregels, zouden echter al snel tot problemen leiden. Daar dit belangrijke criteria zijn lijkt dit voorstel niet geschikt voor implementatie.

5.2.2 XMOF

Een ander belangrijk voorstel is XMOF, een transformatietaal tussen MOF-modellen. XMOF is een taal voorgesteld in 2003 door Compuware Corporation en Sun Microsystems. Het voorstel wordt volledig gespecificeerd in [CWSM03].

Inhoud van het voorstel

XMOF is een declaratieve taal gebaseerd op OCL. Voor zoekopdrachten op modellen wordt OCL volledig hergebruikt. Voor visies op een model stelt XMOF een transformatie van het bronmodel voor. Voor de selectie van modelementen wordt, in overeenstemming met de declaratieve natuur van de taal, patrooncorrelatie gebruikt. Het formuleren van patronen gebeurt eveneens met OCL. Het formuleren van patronen gebeurt in combinatie met condities. Patronen kunnen verder ook rollen en functies hebben. Patrooncondities, -rollen en -functies worden algemeen patroonelementen genoemd.

Ook door de declaratieve natuur ondersteunt de taal symmetrische transformaties. Een transformatie formuleert een declaratief verband tussen een aantal invoer- en uitvoermodellen, de richting waarin de transformatie wordt uitgevoerd wordt echter bij uitvoering bepaald. In plaats van invoer en uitvoer gebruikt XMOF daarom het concept van *poort*.

Een poort is een eigenschap van een klasse. Dit is een attribuut, associatie-einde of operatie. Een poort kan correleren met een deel van een model of ervan afleiden. In het eerste geval is de poort de invoer van een transformatie, en moet de transformatie een uitvoermodel, het doel, definiëren. In het tweede geval is de poort de uitvoer, en moet de transformatie een invoermodel, de bron, definiëren. Poorten kunnen ook genest zijn.

Als oplossing voor transformatie bij incrementele veranderingen van modellen, stelt XMOF voor de verbanden tussen getransformeerde modelementen persistent op te slaan. Dit gebeurt door een MOF-gebaseerd meta-model te definiëren voor afbeeldingen tussen uitgevoerde modellen. Verder ondersteunt XMOF ook compositie en uitbreiding van transformatieregels, en gebruik van transformatiebibliotheken. Ook bijwerken van een model wordt ondersteund. De abstracte syntax van de taal wordt ook als MOF 2.0-meta-model beschreven.

Evaluatie

XMOF is een krachtige taal met een mooie structuur, maar bevat ook een aantal nadelen. In wat hierna volgt vergelijken we dit voorstel met de evaluatiecriteria die in 5.1.4 bepaald werden.

- Complexiteit: de volledig declaratieve aard van XMOF zorgt ervoor dat bij complexe transformaties bepaalde patronen al gauw moeilijk te formuleren zijn. De taal biedt de notie van patroonfuncties die hieraan moet tegemoet komen. Dit lijkt echter een onvolledige

oplossing. Het voorstel maakt notie van externe bibliotheken, maar ook met behulp daarvan kunnen bepaalde transformaties moeilijk worden.

- Hergebruik: XMOF ondersteunt alle mogelijke mechanismen voor hergebruik en uitbreiding.
- Selectie van invoer en uitvoer: Patrooncorrelatie, in combinatie met condities, rollen en functies is een veel betere manier als bijvoorbeeld de imperatieve constructies uit het voorstel van Kennedy Carter (zie 5.2.1). De opmerkingen voor formuleren van complexe transformatieregels (zie hoger) blijven hier echter gelden.
- Voldoen aan functionele en non-functionele vereisten: XMOF voldoet aan alle functionele en non-functionele vereisten die van belang zijn in het kader van deze thesis. Daarnaast wordt nog aanvullende functionaliteit ondersteund.
- Instapdrempel: De instapdrempel voor deze taal is héél hoog. Niet alleen is dit een declaratieve taal, er worden ook heel wat complexe concepten geïntroduceerd die we voor eenvoudige transformaties liever zouden achterwege laten. Doordat ze integraal deel van het transformatiemechanisme uitmaken is dit echter niet mogelijk. Ook de voorbeelden in het voorstel zijn van beperkte bruikbaarheid.

XMOF biedt heel wat interessante mogelijkheden. Helaas is de taal te abstract gedefinieerd en de instapdrempel te hoog voor implementatie op korte tijd. Om die reden lijkt het ook weinig waarschijnlijk dat de taal op termijn als standaard door OMG overgenomen, wat de taal opnieuw minder aantrekkelijk maakt.

5.2.3 Interactive Objects

Interactive Objects is een Duits bedrijf dat producten aanbiedt voor het ontwerpen van objectgeoriënteerde software. De aangeboden producten maken gebruik van moderne technologieën en industriestandaarden voor analyseren, modelleren, ontwerpen en implementeren, zoals UML en J2EE. Vanuit dit standpunt was er interesse in het ondersteunen van Model Driven Architecture. Daarom stelde het bedrijf een transformatietaal met de naam AIM voor. Het voorstel van Interactive Objects wordt beschreven in [IO04].

Inhoud van het voorstel

Interactive Objects ziet transformaties als samenhangende transformatieregels en onderscheidt drie scenario's in het gebruik van modeltransformaties. Deze drie scenario's zijn:

- Constructiescenario: het bouwen van een uitvoermodel door het toepassen van de transformatieregels op een verzameling invoermodellen.
- Wijzigingsscenario: het wijzigen van een bestaand model zodat het voldoet aan de toepassing van transformatieregels op een verzameling invoermodellen.
- Validatiescenario: het valideren van de overeenkomst van een gegeven model aan de toepassing van transformatieregels op een verzameling invoermodellen.

Voor het bepalen van in- en uitvoer-modelelementen gebruikt AIM, de voorgestelde transformatietaal, patrooncorrelatie. Er wordt dus een declaratieve methode gevolgd. Een transformatieregel wordt gedefinieerd als een verzameling van twee patronen: een patroon voor de invoer-modelelementen en een patroon voor de uitvoer-modelelementen. In een transformatieregel wordt niet gedefinieerd hoe elementen moeten gevonden worden die aan dit patroon voldoen. Dit wordt verondersteld als deel van het uitvoeringsmechanisme. Een transformatieregel definieert enkel welke paren patronen met elkaar gerelateerd zijn.

Voor het invoerpatroon wordt een invoer-zoekopdracht gebruikt. Voor het definiëren van deze zoekopdrachten berust AIM op OCL. Transformatieregels kunnen kind-transformatieregels hebben, en AIM biedt voorzieningen waarmee een kind-transformatieregel een patroon van de ouder-transformatieregel kan hergebruiken. Indien een transformatieregel twee verzamelingen modelementen oplevert die voldoen aan de invoer- en uitvoerzoekopdracht, gebeurt het wijzigen van

attributen van de uitvoer-zoekopdracht door kind-transformatieregels. Deze kind-transformatieregels hebben opnieuw een invoer- en uitvoerpatroon. Als attributen gebruikt AIM de vier types attributen die MOF definieert voor een klasse: attribuut, enkelvoudig associatie-einde, meervoudig associatie-einde en operatie.

Voor de uitvoering van transformatieregels worden paren van elementen die voldoen aan invoer- en uitvoerpatroon met elkaar gerelateerd. Deze relaties bestaan tussen één resultaat van de invoer-zoekopdracht en één resultaat van de uitvoer-zoekopdracht, of een attribuut ervan indien het om een kind-transformatieregel gaat. Deze notie is nodig voor het specificeren van transformatieregels in AIM.

Een visie op een model wordt door Interactive Objects gezien als het resultaat van een transformatie. Het afbakenen van de gewenste modelementen in de resulterende visie gebeurt dus door specificeren van het uitvoerpatroon. Een zoekopdracht is in dit voorstel, zoals vermeld, het resultaat van een OCL-zoekopdracht.

Verder wordt in het voorstel een abstracte en concrete syntax voor AIM voorgesteld. Deze syntax komt overeen met de declaratieve natuur van AIM. Verder ingaan op deze syntax in te gaan lijkt hier niet nodig. Interactive Objects biedt een proefimplementatie van AIM met de naam *OptimalJ*.

Evaluatie

We kunnen dit voorstel opnieuw naast de evaluatiecriteria uit 5.1.4 houden en bepalen in welke mate dit voorstel aan onze criteria voldoet.

- **Complexiteit:** door de volledig declaratieve aard van de transformatietaal, kan het moeilijk worden transformaties van aanzienlijke complexiteit te definiëren. De formulering van patronen kan met uitgebreide modellen en complexe transformatieregels al gauw zeer moeilijk worden. Het hangt dan ook sterk van de details van de syntax van deze taal af hoezeer de taal kan toegepast worden voor aanzienlijk complexe transformaties. Voor dit doel biedt AIM echter ook de mogelijkheid patronen te formuleren in andere programmeertalen. Hierbij moet echter dezelfde opmerking gemaakt worden. Formuleren van patronen hoeft in dat geval niet meer, maar hangt dan sterk af van de mogelijkheden van de gebruikte programmeertaal.
- **Hergebruik:** AIM biedt enkele constructies aan voor hergebruik van transformatieregels. Hiertoe behoren het hergebruiken van patronen van een ouder-transformatieregel, zoals hoger vermeld. Verder is er ook ondersteuning voor generische transformatieregels. Ook gebruik van externe bibliotheken wordt ondersteund, op voorwaarde dat deze een MOF-compatibele interface hebben. Ondersteuning voor oververingshiërarchieën van transformatieregels is er echter niet, en dit beschouwen we net als belangrijk in de context van dit eindwerk. De ondersteuning voor hergebruik kunnen we in dit voorstel dus suboptimaal noemen.
- **Selectie van invoer en uitvoer:** in overeenstemming met de declaratieve natuur maakt AIM gebruik van patrooncorrelatie voor de selectie van modelementen. Door de bondige en foutbestendige natuur van patrooncorrelatie kunnen we dit als een sterkte beschouwen.
- **Voldoen aan functionele en non-functionele vereisten:** het voorstel vermeldt niet uitdrukkelijk in welke mate aan deze vereisten wordt voldaan. Wel kunnen we afgaan op de structuur en voorzieningen van de taal om te bepalen hoe aan de functionele en non-functionele vereisten voldaan wordt. Kort samengevat blijkt aan alle functionele vereisten voldaan te zijn, en aan geen enkele non-functionele vereiste. Interactive Objects stelt voor om twee afzonderlijke transformaties te definiëren voor het implementeren van symmetrische transformaties. Constructies om deze transformaties aan elkaar te koppelen zijn er niet. Ook het bijwerken van bestaande modellen wordt slechts in bepaalde situaties ondersteund. Zeker jammer is zoals vermeld het ontbreken van ondersteuning voor uitbreiden van transformatieregels.
- **Instapdrempel:** de instapdrempel voor een taal als AIM is hoog. De taal volgt een volledig declaratief model, wat in combinatie met het meta-niveau waarop gewerkt wordt een

aanzienlijke leercurve vormt. Ook documentatie van de taal en uitgewerkte voorbeelden zijn in het voorstel van lage kwaliteit.

Samengevat kunnen we zeggen dat ook deze taal interessante concepten biedt. De declaratieve manier van het selecteren van modelementen is de weg die we willen volgen. Dit voorstel blijkt echter moeilijk te implementeren door de leercurve van de volledig declaratieve aanpak. Ook E2S heeft weinig ervaring met declaratieve talen. Bovendien zijn de non-functionele vereisten niet vervuld. Daarom richten we onze blik op andere voorstellen.

5.2.4 QVT-Merge Group 1.0

QVT-Merge Group is een groep van bedrijven en academische instellingen die zich groepeerden in functie van een interesse voor Model Driven Architecture. Tot deze groep behoren onder meer bedrijven als IBM en Alcatel, een aantal Franse, Britse, en Amerikaanse universiteiten, en Kennedy Carter. De maker van het voorstel uit 5.2.1 is dus van het xUML-pad afgeweken en heeft zich bij deze groep gevoegd. De ervaringen die Kennedy Carter opdeed met dit voorstel, en andere bedrijven met hun voorstel, werden wel in dit nieuwe voorstel verwerkt.

Het voorstel dat in deze paragraaf besproken wordt, is het eerste voorstel dat door zijn kwaliteiten een serieuze kansmaker vormt voor implementatie. Dit zal in het volgende duidelijk worden. Verder in deze thesis is ook een implementatie van dit voorstel te vinden. Het volledige voorstel wordt beschreven in [QVTMG04].

Inhoud van het voorstel

De inhoud van dit voorstel kan in een aantal duidelijk afgeijnde onderdelen opgesplitst worden. Een eerste classificatie van de voorgestelde concepten gebeurt volgens Q/V/T, de drie vooropgestelde aspecten van transformatietalen.

- Zoekopdrachten: hiervoor gebruikt het voorstel van de QVT-Merge Group een uitbreiding van OCL 2.0 die de bondigheid bevordert. Het niet definiëren van een nieuwe taal en een zekere vertrouwdheid met deze taal kunnen we als aanzienlijke voordelen beschouwen.
- Visies: er wordt voorgesteld om visies op een model te creëren met behulp van een transformatie van het model in kwestie. De voordelen hiervan zijn dat er geen apart mechanisme hoeft gedefinieerd te worden en dat het aantal mogelijke visies onbeperkt is.
- Transformaties: de nadruk van dit voorstel ligt op transformaties. Voor transformaties stelt QVT-Merge Group een taal voor met de naam **QVT**. De mechanismen die voorgesteld worden bevinden zich hoofdzakelijk in dit veld. Hiertoe behoren ondermeer relaties en afbeeldingen, en dit bepaalt ook meteen de volgende classificatie. Dit wordt hieronder in meer detail besproken.

Relaties en afbeeldingen zijn twee mechanismen die worden voorgesteld voor de implementatie van transformaties. Het onderscheid komt overeen met het onderscheid tussen de specificatie en de implementatie van een relatie. Waar relaties definiëren welke modelementen door de transformatie naar elkaar vertaald worden, formuleert de specificatie hoe dit dient te gebeuren.

Een relatie is een multidirectionele transformatiespecificatie. Een relatie is niet in staat een model te creëren of te wijzigen. Wat een relatie echter wel kan is controleren of twee of meer modellen volgens de specificatie van de transformatie consistent zijn met elkaar. Relaties gaan daardoor typisch gebruikt worden in de analysefase van een project, of om de consistentie van een afbeelding te controleren.

Een afbeelding is een mogelijk unidirectionele implementatie van een transformatie. Afbeeldingen kunnen bovendien waarden terugkeren. Een afbeelding kan de implementatie vormen voor een onbeperkt aantal relaties. In de terminologie van dit voorstel heet een implementatie een *verfijning*. Een afbeelding moet altijd consistent zijn met de specificatie van de transformatie, dus de relatie die ze verfijnt. Een groot voordeel hierbij is dat een afbeelding niet enkel in QVT kan geschreven worden, maar in eender welke programmeertaal.

Voor zowel relaties als afbeeldingen wordt QVT gebruikt. QVT is een hybride taal. Ze bevat een declaratief aspect voor de selectie van modelementen. Dit gebeurt via patrooncorrelatie. Omdat een taal voor patrooncorrelatie altijd een compromis is tussen expressiviteit en bondigheid, wordt patrooncorrelatie bij QVT verrijkt met condities. De combinatie van patroon en condities wordt een domein genoemd. Relaties gaan dus niet geformuleerd worden tussen patronen, zoals in 5.2.3, maar tussen domeinen. Verder kunnen condities ook over domeinen heen geformuleerd worden.

De algemene vorm van een relatie wordt in Figuur 5.1 voorgesteld.

```
relation R {
  domain { pattern1 when condition1 }
  ...
  domain { patternn when conditionn }
  when { condition }
}
```

Figuur 5.1: Relatie in QVT 1.0

Bemerk dat een relatie een willekeurig aantal domeinen kan tellen. De laatste conditie is een conditie die over meerdere domeinen heen geformuleerd wordt.

Het imperatieve deel van de QVT-taal wordt gebruikt voor het specificeren van afbeeldingen. Dit deel van de taal ondersteunt het creëren en wijzigen van objecten, iteratieconstructies en dergelijke.

Er wordt ook een grafische syntax voor de taal voorgesteld, als abstractie van de tekstuele vorm. Verder ondersteunt het voorstel ook hergebruik en uitbreiding van transformaties, transformaties die elkaar oproepen, compositie van transformaties, gebruiken van transformaties die in een andere taal geschreven zijn, en meer. De onderliggende structuur van de taal wordt, analoog aan UML 2.0 (zie 3.2) opgedeeld in een Infrastructuur- en Superstructuurgedeelte. Ook wordt de taal als een MOF 2.0-metamodel voorgesteld. IBM heeft van deze taal een proefimplementatie met de naam *Model Transformation Framework* gebouwd, die verderop besproken wordt.

Evaluatie

We vergelijken dit voorstel opnieuw met de evaluatiecriteria die we in 5.1.4 bepaald hebben.

- **Complexiteit:** QVT is uiterst geschikt voor complexe transformaties. De taal bevat een heel aantal constructies ter ondersteuning hiervan, zoals transformatieregels die elkaar aanroepen, compositie en dergelijke. Daarnaast is er ook een interface voorzien met transformatieregels die in andere talen opgesteld zijn voor bijvoorbeeld wiskundige problemen. Verder kan de grafische notatie ook gebruikt worden voor het visualiseren van de samenstelling en interactie van verschillende transformatieregels.
- **Hergebruik:** QVT bevat alle constructies die hergebruik van transformatieregels kunnen bevorderen. Transformatieregels kunnen uitgebreid en gespecialiseerd worden.
- **Selectie van invoer en uitvoer:** hiervoor wordt patrooncorrelatie gebruikt. Dit is niet alleen een heel goede manier, ook wordt patrooncorrelatie met condities aangevuld om de uitdrukingskracht van het patroon te bevorderen.
- **Voldoen aan functionele en non-functionele vereisten:** QVT is niet volledig maar deels declaratief. Dit kan als een voordeel gezien worden. QVT combineert namelijk het beste van twee werelden door een declaratieve manier te gebruiken voor selectie van modelementen, en een imperatieve manier voor het definiëren van afbeeldingen. Relaties zijn volledig

bidirectioneel, afbeeldingen kunnen gedefinieerd worden als het inverse van een andere transformatie.

- Instapdrempel: de instapdrempel voor QVT valt goed mee. Het declaratieve aspect wordt op een manier aangewend die onmiddellijk duidelijk en nuttig is. Het imperatieve specificeren van afbeeldingen verlaagt de instapdrempel. Hoewel het enkele fouten bevat, is het document waarin dit voorstel geformuleerd wordt van heel hoge kwaliteit en bevat enkele uitgebreid uitgewerkte voorbeelden. Het lijkt ons dus doenbaar om met QVT te leren werken.

QVT vormt een interessant voorstel. De taal bevat een uitgebreid gamma aan constructies om aan onze transformatienoden te voldoen. Daarenboven worden een heel aantal constructies voorgesteld die handig kunnen blijken indien het vooropgestelde toepassingsdomein verruimd wordt.

5.2.5 QVT-Merge Group 2.0

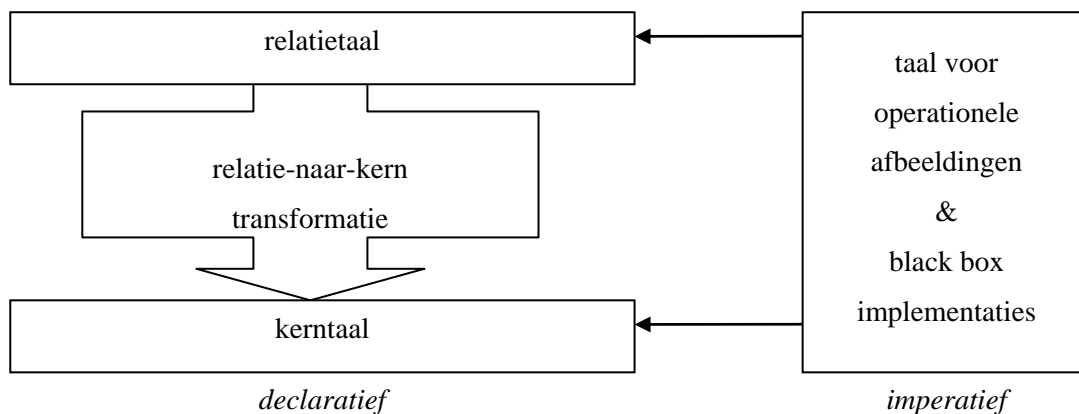
Het voorstel dat op 2 maart 2005 door de QVT Merge Group werd uitgebracht, als versie 2.0, was het resultaat van de bevindingen uit versie 1.0 (zie 5.2.4) en andere voorstellen. Bovendien zijn voor deze versie ook Compuware Corporation en Sun Microsystems, de makers van het voorstel uit 5.2.2, tot de groep toegetreden. Ook zij hebben hun bevindingen aan dit voorstel toegevoegd. QVT 2.0 wordt naar alle verwachtingen de finale standaard. De volledige specificatie kan teruggevonden worden in [QVTMG05].

Inhoud van het voorstel

Hoewel dit voorstel een nieuwe versie is van het voorstel uit 5.2.2, is de overlapping tussen beide beperkt. De specificatie van dit voorstel bevat drie verschillende talen: relatietaal, taal voor operationele mappings en kerntaal. Verder behoort ieder van deze drie talen tot een declaratief of een imperatief deel van de specificatie. In de hierna volgende paragrafen zal elk van de drie delen van de specificatie verder toegelicht worden, het verband ertussen verduidelijkt, en aangehaald of een taal tot het declaratieve of imperatieve deel van de specificatie behoort. Doordat het declaratieve deel de context vormt voor de uitvoering van het imperatieve deel, worden de twee niveaus van de declaratieve architectuur eerst toegelicht.

De declaratieve delen van deze specificatie worden gestructureerd in een tweenniveau-architectuur. De twee lagen zijn:

- relatietaal: laat toe om declaratief en gebruiksvriendelijk de relaties tussen modellen te formuleren. De taal ondersteunt complexe patrooncorrelatie en het creëren van generische objecten gebaseerd op dit patroon. Relaties kunnen ook controleren dat bepaalde relaties gelden tussen modelementen die gecorreleerd worden door de invoerpatronen van de relatie. De semantiek van de relatietaal is een combinatie van Engels en eerste orde predikaatlogica. Daarnaast is in dit voorstel een transformatie van de relatietaal naar de kerntaal voorzien, om zo een relatie uitvoerbaar te maken op een uitvoeringsmechanisme dat enkel de kerntaal ondersteunt.
- kerntaal: is gebaseerd op MOF en OCL, en bevat in realiteit slechts minimale uitbreidingen hiervan. De kerntaal ondersteunt enkel patrooncorrelatie over een vlakke verzameling variabelen, dus niet over variabelen genest in objecten. De kerntaal is even krachtig als de relatietaal, en door zijn relatieve laag niveau kan de semantiek ervan nog eenvoudiger gedefinieerd worden. Dit heeft echter tot gevolg dat transformaties geschreven in de kerntaal verbozer zijn, dus meer tekst nodig hebben om volledig gespecificeerd te worden. De kerntaal kan in een transformatiewerktuig rechtstreeks geïmplementeerd worden.



Figuur 5.2: Relaties tussen QVT-metamodellen

In Figuur 5.2 wordt de relatie tussen deze talen weergegeven. De relatietaal is een niveau hoger dan de kerntaal, maar kan gerealiseerd worden door de kerntaal. Er bestaat een relatie-naar-kern transformatie die dit mogelijk maakt. Deze twee talen zijn declaratief. De talen voor operationele afbeeldingen en *black box* implementaties gebruiken de relatietaal, of rechtstreeks de kerntaal. Zo kunnen declaratieve methodes een imperatieve implementatie hebben.

Er kan een analogie gemaakt worden met de Java architectuur. De kerntaal kan vergeleken worden met Java bytecode, en de semantiek van de kerntaal valt te vergelijken met de gedragsspecificatie van de Java Virtuele Machine. De relatietaal speelt de rol van de Java taal, en de standaard transformatie van relatietaal naar kerntaal is als een Java compiler die bytecode produceert.

Naast de declaratieve relatietaal en kerntaal, die dezelfde semantiek implementeren op een verschillend niveau van abstractie, zijn er nog twee mechanismen voor het uitvoeren van imperatieve transformaties: een taal voor operationele afbeeldingen en een uitvoeringsmechanisme voor *black box* implementaties van MOF-compatibele operaties.

Met de taal voor operationele afbeeldingen kunnen transformaties op een volledig imperatieve manier geschreven worden. De taal voorziet in uitbreidingen van OCL met operaties die neveneffecten uitvoeren. Dit laat een meer procedurele stijl toe, en een syntax die voor imperatieve programmeurs vertrouwd lijkt.

Operationele afbeeldingen kunnen gebruikt worden om een relatie te implementeren, als het moeilijk is om op een volledig declaratieve manier te formuleren hoe de twee domeinen van een relatie (invoer en uitvoer, afhankelijk van de richting waarin de transformatie uitgevoerd wordt) zich tot elkaar verhouden.

MOF-compatibele operaties kunnen voor relaties gebruikt worden, zodat het mogelijk wordt om van een MOF-compatibele operatie verschillende implementaties met dezelfde hoofding (*signature*) te kiezen. Dit maakt andere scenario's mogelijk voor het uitvoeren van transformaties. Hieronder vallen onder meer:

- complexe algoritmen kunnen geschreven worden in eender welke programmeertaal die een MOF-compatibele binding heeft.
- voor het berekenen van waarden die uit de attributen van een model afgeleid worden, kunnen domeinspecifieke bibliotheken gebruikt worden. Domeinen als wiskunde, ingenieurswetenschappen of biowetenschappen kunnen grote bibliotheken hebben die domeinspecifieke algoritmes implementeren. Deze kunnen heel moeilijk, zometert onmogelijk declaratief te formuleren zijn.

- implementaties van sommige delen van een implementatie kunnen transparant blijven.

Transformaties kunnen met eender welk van deze drie talen geïmplementeerd worden, naargelang de verkozen stijl van de programmeur. Een klein verschil treedt op bij de uitvoeringsrichting. De taal voor operationele mappings en black box implementaties beperken de uitvoeringsrichting tot éénrichtingstransformaties. Bidirectionele transformaties zijn voor deze talen mogelijk als voor een transformatie ook een inverse operationele implementatie wordt opgegeven.

Een voorbeeld van een codefragment in QVT 2.0 wordt gegeven in Figuur 5.3.

```

Relation R
{
    Var <R_variable_set> // declaration of variables used in the
                        // Relation
    [checkonly | enforce] Domain:<direction_1>
    <domain_1_variable_set> // subset of <R_variable_set>
    {
        <domain_1_pattern> [<domain_1_condition>]
    }
    [checkonly | enforce]
    Domain:<direction_n><domain_n_variable_set>
    // subset of <R_variable_set>
    {
        <domain_n_pattern>
        [<domain_n_condition>]
    } // n >= 2
    [when <when_variable_set> <when_condition>]
    [where <where_condition>]
}

```

Figuur 5.3: Relatie in QVT 2.0

Deze definitie is gelijkaardig aan het codefragment uit 5.2.4. Een relatie kan nu echter ook variabelen hebben, kan gebruikt worden om enkel te controleren dat bepaalde verbanden gelden, etc. Er zijn een aantal domeinen waartussen transformaties gebeuren. Verder kunnen beperkingen geformuleerd worden voor elk domein apart, of over een aantal domeinen heen.

Verder biedt de taal een gamma aan toegevoegde functionaliteit aan. Dit gamma en de punten die hierboven niet vermeld werden zijn gelijkaardig aan versie 1.0 van het voorstel (zie 5.2.4).

Evaluatie

We vergelijken dit voorstel met de evaluatiecriteria die we in 5.1.4 bepaald hebben.

- Complexiteit: QVT 2.0 is opnieuw uiterst geschikt voor complexe transformaties. De taal bevat alle constructies die mogelijk ter ondersteuning hiervan kunnen dienen. Zie hiervoor ook de evaluatie van QVT 1.0 in 5.2.4.
- Hergebruik: QVT bevat alle constructies die hergebruik van transformatieregels kunnen bevorderen. Transformatieregels kunnen uitgebreid en gespecialiseerd worden doordat hiervoor gesteund wordt op de structuur van MOF 2.0.
- Selectie van invoer en uitvoer: net als in QVT 1.0 wordt patrooncorrelatie gebruikt.
- Voldoen aan functionele en non-functionele vereisten: QVT 2.0 overstijgt alle vereisten. Een volledig declaratieve transformatietaal, zoals gevraagd in de functionele vereisten, wordt

voorzien in de vorm van de relatietaal. Daarnaast is ook een volledig imperatieve taal uitgewerkt in het voorstel. Bidirectionaliteit, opspoorbaarheid, ... worden alle ondersteund.

- Instapdrempel: de instapdrempel voor QVT is zeer redelijk. Programmeurs hebben de vrijheid om een declaratieve of imperatieve stijl te volgen, zelfs een andere programmeertaal kan gebruikt worden. Versie 2.0 is bovendien foutvrij, goed gedocumenteerd en uitgewerkt, en bevat relevante voorbeelden.

QVT 2.0 is door zijn volledigheid een heel aantrekkelijke transformatietaal. De taal biedt de vrijheid tussen een declaratieve of imperatieve programmeerstijl voor het definiëren van transformaties. Softwarebedrijven geïnteresseerd in de implementatie van een transformatietaal, zoals E2S, worden de keuze gelaten welke van deze talen te implementeren. Daarnaast biedt het voorstel, net als QVT 1.0, nog een hele verzameling toegevoegde functionaliteit. Dit maakt dat QVT 2.0 voorlopig als beste voorstel beschouwd wordt.

5.2.6 Overige

Er werden nog een aantal andere voorstellen bestudeerd in het kader van deze thesis. Deze bleken echter zo gebrekkig, onvolledig of foutief geformuleerd dat besloten werd deze voorstellen hier niet te bespreken.

In de context van dit eindwerk werd aanvankelijk overwogen om modeltransformaties op het UML-niveau te realiseren. Dit zou het ontwerpen en implementeren betekenen van een werktuig dat een UML-model inleest onder de vorm van een XMI-bestand, de beschrijving van het klassendiagram heropbouwt, en daarop transformaties uitvoert. Dit idee wordt ook beschreven in [GLRSW02]. Eventueel kon dan XSLT, kort voor *Extensible Stylesheet Language Transformation*, een taal voor transformaties tussen XML of XMI, als transformatietaal gebruikt worden. Dit idee werd echter al gauw achterwege gelaten wegens een gebrek aan abstractie. Een degelijk MDA-werktuig dient zich op een hoger niveau te situeren en niet van een specificatie als XMI afhankelijk te zijn. Het leek veel nuttiger om naar de RFP van OMG te kijken (zie 5.1), en hiervan een proefimplementatie van transformatieregels te realiseren. Deze proefimplementatie gaat dan, conform de QVT RFP, gebruik maken van modellen die voldoen MOF-gebaseerde metamodellen, waaronder UML 2.0.

5.3 Besluit

QVT 2.0 is uit deze vergelijking naar voor gekomen als de transformatietaal die het meest mogelijkheden biedt. Dit niet alleen in een algemene context, maar ook volgens de criteria die we vanuit de context van dit eindwerk aan een transformatietaal stellen.

Daarom werd beslist de transformatieregels van de E2S-software in QVT 2.0 te definiëren. Dit gebeurt niet door het uitschrijven van de transformatieregels op papier. Het aanleren van de syntax van een nieuwe taal door deze enkel te bestuderen op papier, geeft ons geen voeling met de taal. Daarenboven is het te moeilijk aan te nemen dat deze transformatieregels, eenmaal geschreven, correct zijn.

Dit gebeurt echter door middel van een vrij beschikbare proefimplementatie van QVT 2.0 door een derde partij. Doordat ook QVT 1.0 een interessant voorstel lijkt, werd ook een proefimplementatie van QVT 1.0 bestudeerd. De bevindingen van deze experimenten vindt u in het volgende hoofdstuk.

Hoofdstuk 6

Implementatie van transformatieregels

In Hoofdstuk 5 werd uit de waaier van transformatietaalen het QVT 2.0 voorstel als beste kandidaat gekozen. In dit hoofdstuk wordt een proefimplementatie van de transformatieregels die in Hoofdstuk 4 afgeleid werden voor de E2S-software uitgewerkt in deze transformatietaal. We kunnen hiervoor beschikken over een bestaand transformatieraamwerk met de naam ATL [AGLIN06]. De implementatie met ATL wordt besproken in 6.2. Als vergelijking wordt ook de implementatie aangeboden van deze transformatieregels in het QVT 1.0 voorstel. Ook hiervoor bestaat een transformatieraamwerk met de naam MTF [IBM04]. Deze implementatie wordt besproken in 6.3.

6.1 Algemeen

In deze paragraaf worden eerst enkele hulpmiddelen besproken waar ATL en/of MTF gebruik van maken voor de uitvoering van transformatieregels. Daarna worden de transformatieregels die verder geïmplementeerd worden op conceptueel niveau besproken.

6.1.1 Hulpmiddelen

De ontwikkelomgevingen van ATL en MTF staan niet op zichzelf. Ze zijn ontworpen als *plugins* voor het Eclipse-platform [SH04]. Eclipse is een raamwerk voor ontwikkelomgevingen, ontwikkeld door IBM. Door zijn veelzijdigheid, goede ontwerp en platformafhankelijkheid wint het Eclipse-platform sterk aan populariteit.

EMF, voluit *Eclipse Modeling Framework* [BEGMS03], is een faciliteit voor metamodellering en modelopslag. ATL en MTF maken beide gebruik van EMF, hoewel ATL ook een alternatief mechanisme biedt voor metamodellering.

Voor het beschrijven van modellen en metamodellen beschikt EMF over een eigen taal. Deze taal heet *Ecore*. Ecore is niet meer dan een subset van de modelleerconcepten van MOF 1.4. EMF kan elk (meta-)model lezen en door elk (meta-)model navigeren dat met Ecore beschreven is. Als we in ATL of MTF een transformatie willen beschrijven die een UML-model naar een ander UML-model transformeert, hebben we een Ecore-beschrijving van het UML-metamodel nodig. Dit is gelukkig vrij beschikbaar samen met EMF.

In- en uitvoermodellen van transformaties in ATL of MTF stellen we in ons geval als UML-klassendiagrammen op. Ondersteuning hiervoor is nog niet zo courant. Voor dit eindwerk werd gebruik gemaakt van de Eclipse *UML2-plugin* [KH05]. Deze plugin is beperkt doordat er geen ondersteuning is voor het visualiseren van UML 2.0-diagrammen. De plugin biedt echter ondersteuning voor lezen, wijzigen en maken van UML 2.0-diagrammen als een soort boomstructuur. Ook de Ecore-beschrijving van het UML-metamodel maakt deel uit van de Eclipse *UML2-plugin*.

6.1.2 Transformatieregels

De implementatie van de transformatieregels vastgelegd in Hoofdstuk 4 vertonen een aantal gemeenschappelijke punten. Om herhaling te vermijden, en duidelijk te maken wat precies in ATL en MTF moet geformuleerd worden, worden eerst deze gemeenschappelijkheden toegelicht.

De transformatieregels werden als volgt vastgesteld bij de analyse van de E2S-software:

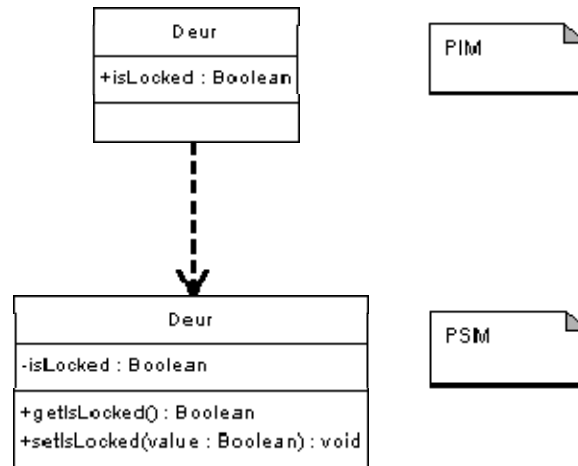
1. Voor iedere klasse in het PSM wordt een Delphi-unit aangemaakt met declaratie van een Delphi-klasse, waarbij afhankelijke units slechts eenmaal ingevoegd worden.
2. Voor ieder attribuut wordt een databankelement gemaakt met een publieke inspector en mutator, en “controleer waarde”-methode behalve indien het over een Booleaanse waarde gaat
3. Voor iedere associatie wordt een publieke inspector toegevoegd, “zet op nul”-methode en een publieke mutator indien het niet om een meervoudige associatie gaat
4. Voor iedere Delphi-klasse wordt een constructor (maakt gebruik van een Factory-object), initialisatiemethode en destructor toegevoegd.

Deze transformatieregels willen we nu modelleren als het geheel van een transformatie van PIM naar PSM, en een transformatie van PSM naar werkende code.

Eerst proberen we de transformatie van PIM naar PSM te formuleren. Hiervoor moeten we eerst het niveau van abstractie bepalen waarop het PIM zich bevindt. Waar met de E2S-suite voordien de beoogde applicatie enkel als PSM werd gemodelleerd en hieruit code gegenereerd, kunnen we nu ter ondersteuning van MDA ook een PIM invoeren. Dit PIM is een UML 2.0-klassendiagram dat enkel klassen met attributen en eventueel toegevoegde operaties bevat.

Vervolgens wordt dit PIM naar een PSM getransformeerd. Ook dit PSM is een UML 2.0-klassendiagram, maar met een aantal wijzigingen ten opzichte van het PIM. Deze wijzigingen zijn het resultaat van de transformatie van PIM naar PSM. Indien een transformatieregel een publiek attribuut uit het PIM privaat maakt en hiervoor een inspector en mutator toevoegt, zijn deze wijzigingen in het PSM terug te vinden. Analoog zijn de wijzigingen van andere transformatieregels in het PSM terug te vinden. Voor de transformatie naar het PSM worden transformatieregels 2 tot en met 4 toegepast. Hiervoor worden enkele implementatiedetails uit de huidige transformatieregels verwijderd.

Eerst beschouwen we de tweede transformatieregel. De details van de implementatie van attributen als databankelementen met een “controleer waarde”-methode kunnen interessant zijn, maar zouden ons in de context van deze proefimplementatie te ver leiden. Daarom wordt de tweede transformatieregel gewijzigd naar het transformeren van een publiek attribuut in het PIM naar een privaat attribuut in PSM waarvoor ook een inspector en mutator toegevoegd wordt. Deze transformatie wordt voorgesteld op Figuur 6.1. Voor het attribuut `isLocked` van de klasse `Deur` wordt een inspector `getIsLocked` toegevoegd en een mutator `setIsLocked`. Een attribuut is in het UML 2.0-metamodel een instantie van `Property`, een operatie is een instantie van `Operation`. Meer gedetailleerd gaat deze transformatie dus eigenschappen van een instantie van `Property`, zoals de visibiliteit, tussen PIM en PSM aanpassen en twee instanties van `Operation` in het PSM toevoegen.

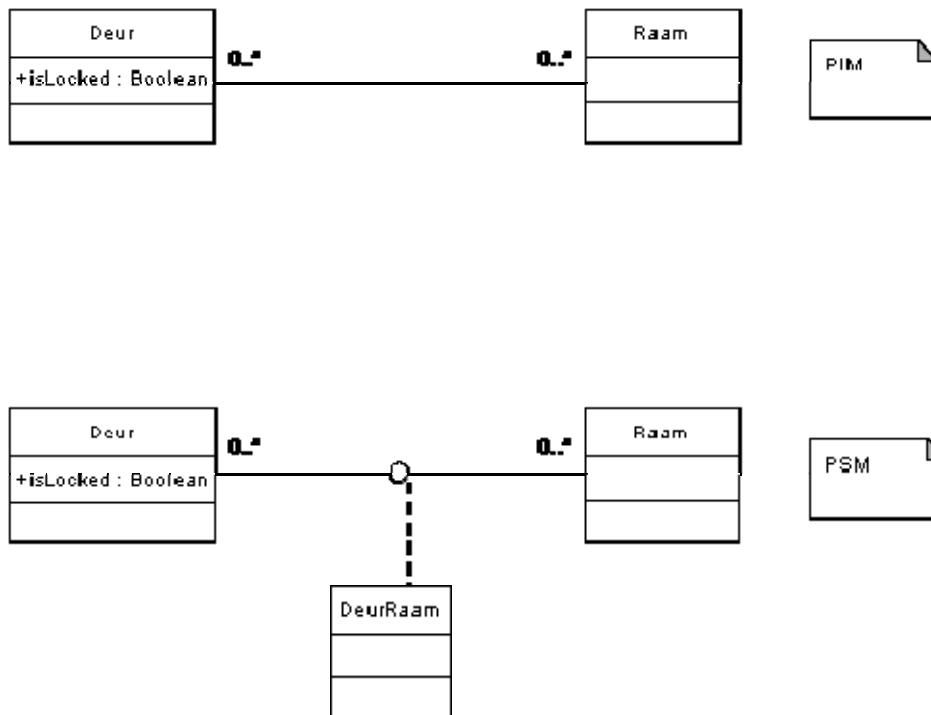


Figuur 6.1: Transformatie van publiek naar privaat attribuut

Vervolgens beschouwen we de derde transformatieregel. Een inspector voor een associatie-einde toevoegen is analoog aan de formulering van de tweede transformatieregel. Een associatie-einde in een klasse is vanaf UML 2.0 net als een attribuut een instantie van `Property`. Daardoor kunnen we voor deze transformatieregel ook de tweede transformatieregel gebruiken. Zo kan ook een mutator voor het associatie-einde toegevoegd worden, wat de “zet-op-nul” methode vervangt.

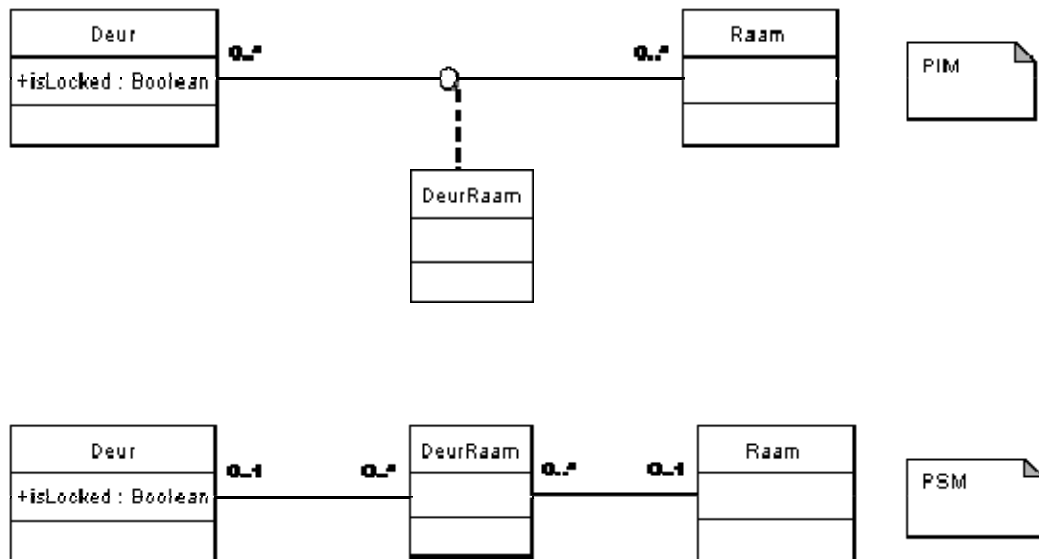
Het niet genereren van een publieke mutator indien het om een meervoudig associatie-einde gaat, heeft een bijzondere reden. Deze beperking werd door E2S aan HOORA toegevoegd omdat men met deze suite geen manier had om n-op-m associaties om te vormen naar code. Met de introductie van MDA is er interesse om dit te verhelpen. MDA zou het mogelijk moeten maken om een n-op-m associatie om te vormen naar één of meerdere modelementen die wel kunnen geïmplementeerd worden als code. Daarom worden enkele extra transformatieregels toegevoegd om n-op-m associaties te transformeren. Deze transformatieregels dienen als een goede test voor de sterkte van een transformatietaal.

De eerste transformatieregel die we toevoegen definieert de transformatie van een n-op-m relatie naar een associatieklasse. Een associatie wordt gemodelleerd als een instantie van `Association` in het UML 2.0-metamodel, een associatieklasse als instantie van `AssociationClass`. De equivalentie wordt voorgesteld in Figuur 6.2. Een naam voor de nieuw ingevoegde associatieklasse kan bijvoorbeeld gevormd worden door de concatenatie van de namen van de twee geassocieerde klassen. Het voordeel van het invoegen van deze transformatieregel wordt duidelijk in combinatie met de volgende transformatieregel.



Figuur 6.2: Transformatie van n-op-m associatie naar associatieklasse

Een volgende transformatieregel die we toevoegen, vormt een associatieklasse uit het UML 2.0-metamodel om naar een klasse uit het UML 2.0-metamodel. Dit kan door een transformatie te definiëren die de associatieklasse omvormt naar een klasse met dezelfde naam en attributen, en twee associaties invoegt tussen de klassen die met de associatieklasse geassocieerd waren en de nieuw ingevoegde klasse. Hierbij wordt de multipliciteit van de associatie-einden passend aangepast. In het UML 2.0-metamodel is een `AssociationClass` een klasse die zowel `Association` als `Class` specialiseert. Daardoor heeft een associatieklasse de eigenschappen van beide. In Figuur 6.3 wordt de equivalentie opnieuw voorgesteld.



Figuur 6.3: Transformatie van een associatieklasse naar een klasse

De combinatie van deze twee toegevoegde transformatieregels vormt meteen een oplossing voor het omvormen van n-op-m relaties. Een n-op-m relatie wordt eerst naar een associatieklasse omgevormd, waarbij de transformatie aan deze klasse de naam van de associatie of een willekeurige naam kan geven. Daarna wordt deze associatieklasse naar een klasse omgevormd waarbij de wijziging van de associaties een oplossing voor het oorspronkelijke probleem vormt. Een n-op-m relatie wordt omgevormd naar twee 1-op-n relaties. Het voordeel van het opsplitsen van deze twee stappen is dat we de transformatie van een associatieklasse naar een klasse ook afzonderlijk kunnen gebruiken.

Vervolgens gaat de transformatie van PIM naar PSM ook een constructor en destructor aan elke klasse toevoegen. Een initialisatie-methode laten we hier weg.

Tenslotte gaan we terug naar de eerste transformatieregel. Elke klasse die uit de transformatie van PIM naar PSM resulteert, gaan we nu transformeren naar code. Om de functionaliteit van de codegenerator van E2S te evenaren, betekent dit dat voor iedere klasse vier bestanden moeten gegenereerd, namelijk een declaratie-, implementatie- en Pascalbestand, en een bestand met constantendeclaraties. Deze bestanden vormen samengevoegd de declaratie van een unit met daarin de declaratie en implementatie van een Delphi-klasse. Deze Delphi-klasse stemt overeen met de UML-klasse uit het PSM. Hierbij moet in acht genomen dat waar afhankelijke units ingevoegd worden, dit slechts eenmaal per unit mag gebeuren. Om de transformatie van PSM naar Delphi niet te complex te maken, worden enkele vereenvoudigingen doorgevoerd ten opzichte van de code gegenereerd door E2S.

Het bestand met constantendeclaraties specificeert welke sleutel een bepaald attribuut van een unit heeft in een databank die waarden van alle attributen uit alle units en alle associaties tussen objecten opslaat. Dit mechanisme biedt een aantal voordelen, maar die zijn in de context van dit eindwerk weinig relevant. We kunnen dus abstractie van de databanksleutels maken, en attributen van en associaties tussen UML-klassen modelleren als attributen van Delphi-klassen. Omdat de databanksleutels met attributen overeenstemmen, declareren we deze attributen in het bestand voor constantendeclaraties. Wanneer dit bestand in het Pascalbestand gevoegd wordt, maken deze ook deel uit van de attributen van de klasse.

Voor het implementatiebestand betekent dit dat de implementatie van methodes niet meer leest van of schrijft naar de databankelementen, maar naar de attributen die deze vervangen.

Verder is elke Delphi-klasse geassocieerd met een *Factory* waarmee het object geconstrueerd wordt, afhankelijk van het beheersysteem van de databank. Ook daar maken we abstractie van, en voegen aan

elke Delphi-klasse een standaard constructor en destructor toe. Ook de initialisatie-procedure uit het declaratiebestand voor de onderliggende databank kunnen we achterwege laten.

Het voordeel van het werken met MDA is dat indien code wil gegenereerd worden in een andere programmeertaal, enkel de laatste transformatieregel dient aangepast te worden. De andere transformatieregels kunnen herbruikt worden. Hierbij merken we op dat OMG een aparte RFP heeft uitgebracht voor dit soort transformaties. Deze RFP heet MOF2Text, een RFP voor transformatie van MOF-modellen naar tekstbestanden, en wordt gespecificeerd in [ad/2004-04-07]. Dit soort transformatietaal kan beter voor de transformatie van PSM naar code gebruikt worden, maar ook de mogelijkheden die de implementaties van QVT voor dit soort transformatie bieden worden even bestudeerd.

Een samenvatting van de transformatieregels die we net opgesteld hebben, wordt hieronder weergegeven.

Transformatie van PIM naar PSM:

- Toevoegen van een constructor en destructor in alle klassen.
- Publieke attributen en associatie-einden naar private instanties, met de toevoeging van inspectors en mutators voor deze instanties: deze transformatieregel transformeert zowel attributen als associatie-einden door de modellering van beide als een instantie `Property` in UML 2.0-metamodel.
- N-op-m associaties naar associatieklassen: deze transformatieregel transformeert instanties van `Association` uit het UML 2.0-metamodel met associatie-einden die een meervoudige multipliciteit hebben, naar instanties van `AssociationClass` met dezelfde multipliciteiten.
- Associatieklasse naar klasse met toevoeging van twee associaties: een instantie van `AssociationClass` wordt hierbij omgevormd naar een instantie van `Class` en twee instanties van `Association`. Hierbij moeten de associatie-einden aangepast worden.

Transformatie van PSM naar code:

Voor elke klasse in het PSM wordt een Delphi-unit met de declaratie en implementatie van een Delphi-klasse gegenereerd. Deze Delphi-klasse bevat alle attributen, methodes, constructors en destructors van de UML-klasse uit het PSM. De Delphi-unit wordt over vier bestanden verdeeld, die samengevoegd worden in één van die bestanden. Waar afhankelijke modules ingevoegd worden, mag dit slechts eenmaal gebeuren per module. Deze transformatie is aanzienlijk verschillend van de vorige. Een MOF2Text-taal is beter geschikt voor dit soort transformaties, maar we willen ook even bestuderen wat voor mogelijkheden de bestudeerde talen voor dit soort transformatie bieden.

In de volgende paragrafen worden de transformatieregels verder uitgewerkt voor twee implementaties van QVT.

6.2 ATL

ATL, de Atlas Transformatie Taal [AGLIN06], is de implementatie van de Franse ATLAS INRIA & LINA onderzoeksgroep van het QVT 2.0 voorstel. ATL wordt ontwikkeld als een onderdeel van AMMA (*ATLAS Model Management Architecture*), een grootschalig project dat zich richt op modelgericht ontwikkelen van software. Het werk dat voor ATL verricht wordt, is een samenwerking tussen de universiteit van Nantes (Frankrijk) en INRIA, een Franse onderzoeksgroep op het vlak van informatica en automatisering. ATL wordt op dit moment gebruikt door verschillende onderzoeksgroepen die in verschillende domeinen werken, en voor onderwijsdoeleinden.

6.2.1 Overzicht

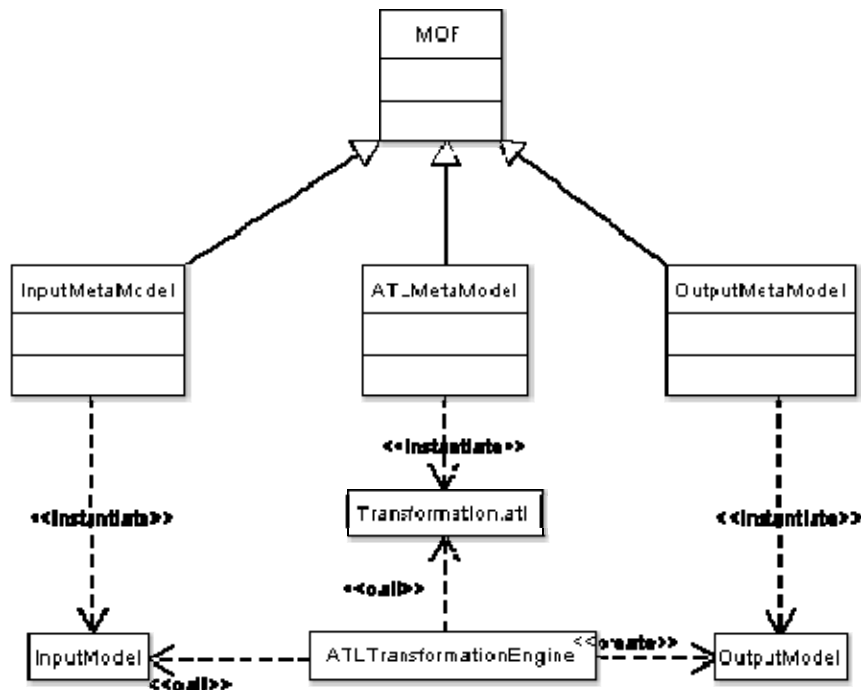
ATL is een taal die gelijkaardig is aan QVT 2.0. In grote lijnen hergebruikt ATL de syntax van OCL. De kleine verschillen tussen de syntax van beide talen worden toegelicht in Appendix B.

ATL-programma's voor transformatie van model naar model worden modules genoemd. Een module bestaat uit een hoofding, een importeersectie, evenals helperfuncties en de transformatieregels (zie Figuur 6.4). De structuur van ATL-programma's wordt verder toegelicht in Appendix B.

Voor een ATL programma kan geschreven worden dienen het metamodel van bron en doel gedefinieerd te zijn. In ATL kan dit met twee types metamodelen: metamodelen afgeleid van MOF, of metamodelen afgeleid van Ecore, de taal die EMF ondersteunt.

Eens het invoer- en uitvoermetamodel gespecificeerd zijn, worden een invoer- en uitvoermodel opgesteld die aan deze metamodelen voldoen. Deze kunnen als in- en uitvoer van de transformatie gespecificeerd worden, waarna de ontwikkelaar enkel nog rest de transformatiemodule zelf te schrijven. Het uitvoeringssysteem van ATL gaat daarop deze module interpreteren, het invoermodel lezen en het uitvoermodel aanmaken of wijzigen. Dit wordt schematisch in Figuur 6.4 voorgesteld.

Er zijn twee uitvoeringsmodes voor de relatietaal: standaardmode en verfijningsmode. In de standaardmode worden elementen enkel gecreëerd als invoerelementen gecorreleerd worden met het invoerpatroon van de transformatie. Aangezien transformaties echter geen modellen kunnen bijwerken, zijn transformaties die slechts een klein deel van een model wijzigen en de rest ongewijzigd laten, moeilijk te schrijven. Dit komt doordat voor het bronmodel enkel lezen toegelaten wordt, en een dergelijke transformatie dus ook transformatieregels zou moeten bevatten om het deel van het model dat niet gewijzigd wordt, te kopiëren. Om dit op te lossen biedt ATL echter een verfijningsmode aan, waarin ongecorrleerde elementen automatisch gekopieerd worden naar het doelmodel. Dit verhoogt het gebruiksgemak van ATL, en kan als een aanzienlijk voordeel gezien worden.



Figuur 6.4: Overzicht van het ATL uitvoeringsmechanisme

6.2.2 Verhouding QVT-ATL

Zowel de relatietaal als de taal voor operationele afbeeldingen uit QVT 2.0 zijn geïmplementeerd in ATL. Onderliggend worden deze beide naar een implementatie van de kerntaal uit QVT 2.0 getransformeerd, zoals ook in QVT 2.0 voorgesteld wordt. Het grootste verschil tussen QVT en ATL is dat bij ATL enkel unidirectionele transformaties toegelaten worden.

Het voldoen aan de QVT-standaard is dan ook geen prioriteit voor ATL. ATL probeert een evenwicht te vinden tussen het voldoen aan het QVT-voorstel en praktisch gebruiksgemak. Dit is ook de reden waarom de concrete syntax van ATL op dit moment gebaseerd is op OCL. In de toekomst plant men ook versies van ATL die andere vormen van modelnavigatie ondersteunen dan OCL. Het is eveneens mogelijk dat ATL andere, extra faciliteiten gaat implementeren dan voorzien in het QVT-voorstel. INRIA, de onderzoeksorganisatie rond ATL, heeft de intentie ATL voornamelijk te gebruiken als een zich ontwikkelend prototype voor het experimenteren met innoverende ideeën in het domein van modeltransformatie en algemeen in het domein van modelmanagement. Dit heeft reeds vruchten afgeworpen. Waar ATL oorspronkelijk een proefimplementatie van QVT 1.0 was, zijn veel concepten uit ATL nu opgenomen in QVT 2.0. Daarom kan ATL gezien worden als een proefimplementatie van QVT 2.0.

6.2.3 Transformatie van PIM naar PSM

In deze paragraaf gaan we de implementatie van de transformatieregels bespreken zoals bepaald in 6.1.2.

Eerst schrijven we de implementatie van een transformatieregel die een constructor toevoegt in alle klassen. Deze implementatie wordt weergegeven op Figuur 6.5. Het UML 2.0-metamodel heeft geen expliciete notie van constructors. Daarom wordt op regels 15 tot en met 17 een operatie toegevoegd met de naam `Create`, als implementatie van een constructor. In sommige programmeertalen, zoals ondermeer Delphi, bestaat ook de notie van een destructor. Ook een destructor wordt aan de klasse toegevoegd. Dit is een operatie die we de naam `Destroy` geven. Deze destructor wordt gedefinieerd op regels 18 tot en met 20. De klasse waartoe de operaties behoren, wordt bepaald door de operatie aan het meervoudig attribuut `ownedOperation` van de klasse in het UML-metamodel toe te voegen. Dit gebeurt op regels 12 en 13. Het sleutelwoord `refining` in de hoofding van de module op regel 2 bepaalt dat de transformatie onder de verfijningsmode van ATL wordt uitgevoerd. Deze uitvoeringsmode wordt ook voor alle volgende transformaties gebruikt. Merk in Figuur 6.5 op hoe transparant en bondig de transformatie met dit mechanisme kan gedefinieerd worden.

```

1 module Class2Constructors;
2 create OUT : UML refining IN : UML;
3
4 rule Class2Constructor {
5 from
6   i : UML!Class
7 to
8   o : UML!Class (
9     name <- i.name,
10    ownedAttribute <- i.ownedAttribute,
11    ownedOperation <- i.ownedOperation,
12    ownedOperation <- oConstructor
13    ownedOperation <- oDestructor
14  ),
15  oConstructor : UML!Operation (
16    name <- 'Create'
17  ),
18  oDestructor : UML!Operation (
19    name <- 'Destroy'
20  )
21 }

```

Figuur 6.5: Transformatieregel in ATL voor toevoegen van een constructor en destructor

Vervolgens schrijven we de implementatie van de publieke-naar-private-attributen transformatie. Deze werkt, zoals vermeld in 6.1.2, in op zowel attributen als associatie-einden. De volledige transformatie in ATL wordt weergegeven in Figuur 6.6.

Op regels 8 tot en met 11 wordt het invoerpatroon van de transformatie bepaald als `Property`-instanties met publieke visibiliteit. De transformatie wijzigt niet alleen de visibiliteit van de `Property`-instantie op regel 15, maar voegt ook een inspector en mutator toe op regels 17 tot en met 21 respectievelijk 27 tot en met 31. Voor een inspector moet daarbij, overeenkomstig het UML 2.0-metamodel, een instantie van `Parameter` aan de operatie gedefinieerd worden met de waarde van het attribuut `direction` gezet op `return`. Deze parameter vormt de terugkeerwaarde van de inspector en heeft dus ook hetzelfde type als de `Property` instantie. De definitie van deze parameter wordt weergegeven op regels 21 tot en met 25.

Voor de specificatie van de mutator, moet analoog een parameter toegevoegd worden. Deze heeft `direction` gezet op `in`, omdat deze een invoerparameter voor de operatie moet vormen. De definitie van de invoerparameter wordt weergegeven op regels 32 tot en met 36.

In de module wordt ook een voorbeeld van een helperfunctie gedefinieerd. Dit is een constructie van ATL om functionaliteit niet altijd opnieuw te hoeven definiëren. Deze helperfunctie converteert de eerste letter van de naam van het attribuut, indien dit een kleine letter was, naar een hoofdletter. Zo kunnen inspectors en mutators, bijvoorbeeld bij een attribuut dat `name` heet, een naam krijgen die eruitziet als `getName` en niet `getname`. De helperfunctie wordt gedefinieerd op regels 4 en 5.

Merk op hoe de helperfunctie bij het bepalen van de naam voor inspector en mutator wordt opgeroepen op regels 18 en 28. Voor converteren van string, concateneren en dergelijke biedt ATL via de OCL-gebaseerde syntax ondersteuning. De syntax is bovendien zo eenvoudig dat de programmeur zich vooral kan toelagen op de transformatie zelf. Dit kunnen we als een groot voordeel beschouwen.

```

1 module Public2Private;
2 create OUT : UML refining IN : UML;
3
4 helper context String def: firstToUpper() : String =
5 self.substring(1, 1).toUpperCase() + self.substring(2, self.size());
6
7 rule Property2Property {
8   from
9     i : UML!Property (
10      (i.visibility = #public)
11    )
12  to
13    o : UML!Property mapsTo i (
14      name <- i.name,
15      visibility <- #private
16    ),
17    og : UML!Operation (
18      name <- 'get' + i.name.firstToUpper(),
19      class_ <- i.class_,
20      ownedParameter <- ogr
21    ),
22    ogr : UML!Parameter (
23      name <- 'return',
24      direction <- #return,
25      type <- i.type
26    ),
27    os : UML!Operation (
28      name <- 'set' + i.name.firstToUpper(),
29      class_ <- i.class_,
30      ownedParameter <- osp
31    ),
32    osp : UML!Parameter (
33      name <- i.name,
34      direction <- #in,
35      type <- i.type
36    )
37 }

```

Figuur 6.6: Transformatieregel in ATL voor publieke naar private attributen

De volgende transformatie is het omvormen van n-op-m associaties naar associatieklassen. De transformatieregel `NToMAssociationToAssociationClass` op regel 16 uit Figuur 6.6 definieert de eigenlijke transformatie van associatie naar associatieklasse. Het invoerpatroon gebruikt op regel 19 de OCL-functie `forall` om te formuleren dat alle einden van de associatie (volgens het UML 2.0 metamodel kunnen dit er immers meer dan twee zijn) een multipliciteit hoger dan 1 moeten hebben. Dit soort multipliciteit wordt door UML met -1 aangegeven, wat de reden is van de vergelijking met '`< 0`' op dezelfde regel. De transformatie gebruikt op regel 23 een helperfunctie `makeAssociationClassName`, gedefinieerd op regels 4 tot en met 13, die de naam voor de nieuw gemaakte associatieklasse bepaalt. Deze helperfunctie neemt de naam van de associatie als naam voor de associatieklasse, of een concatenatie van de namen van de geassocieerde klassen indien de associatie geen naam draagt. Verder worden op regel 24 de associatie-einden van de te transformeren associatie naar de associatieklasse gekopieerd. Doordat, zoals vermeld in 6.1.2, `AssociationClass` zowel `Association` als `Class` uit het UML 2.0-metamodel specialiseert, hebben beide klassen een attribuut `memberEnd` dat de associatie-einden bepaalt.

```

1 module NTOMAssociationToAssociationClass;
2 create OUT : UML refining IN : UML;
3
4 helper context UML!Association def: makeAssociationClassName() :
5 String =
6   if self.name <> ''
7   then
8     self.name
9   else
10    self.memberEnd->iterate( prop; name : String = '' |
11      name.concat( prop.class_.name )
12    )
13  endif;
14
15
16 rule NTOMAssociationToAssociationClass {
17   from
18     i : UML!Association (
19       i.memberEnd->forall( prop | prop.upper < 0 )
20     )
21   to
22     o : UML!AssociationClass (
23       name <- i.makeAssociationClassName(),
24       memberEnd <- i.memberEnd
25     )
26 }

```

Figuur 6.7: Transformatieregel in ATL voor n-op-m associatie naar associatieklasse

De volgende transformatieregel is de transformatie van associatieklassen naar klassen. De implementatie hiervan wordt weergegeven in Figuur 6.8. Deze transformatieregel gebruikt op regel 17 de `foreach` constructie van ATL om voor elk associatie-eind van de associatieklasse een nieuw associatie-eind in de getransformeerde klasse te maken. Daarna wordt op regels 24 tot en met 27 ook voor elk associatie-eind een associatie met één van deze nieuwe associatie-einden gemaakt. Daardoor is de binding tussen de klassen, zoals gedefinieerd in 6.1.2, volledig. Tevens wordt op regels 28 tot en met 35 voor elk associatie-eind een kopie van het multipliciteitselement gemaakt. Dit multipliciteitselement bestaat uit een laagste en een hoogste waarde. Dit multipliciteitselement wordt op regels 21 en 22 aan het nieuw gemaakt associatie-eind, in de klasse die een resultaat is van de getransformeerde associatieklasse, toegekend. Daarna wordt op regels 40 en 41 de multipliciteit in elk associatie-eind in de geassocieerde klassen op '0..1' gezet.

Dit laatste gebeurt in een imperatieve sectie van de transformatieregel. Deze imperatieve sectie op regels 36 tot en met 43 biedt ondersteuning voor iteraties en expliciet wijzigen van waarden.

```

1 module AssociationClass2Class;
2 create OUT : UML refining IN : UML;
3
4 helper context String def: firstToLower() : String =
5 self.substring(1, 1).toLowerCase() + self.substring(2, self.size());
6
7
8 rule AssociationClass2Class {
9 from
10 i : UML!AssociationClass
11 to
12 o : UML!Class mapsTo i (
13   name <- i.name,
14   ownedOperation <- i.ownedOperation,
15   ownedAttribute <- i.ownedAttribute.union( oAttr )
16 ),
17 oAttr : distinct UML!Property foreach ( attr in i.memberEnd )
18 (
19   name <- attr.class_.name.firstToLower(),
20   type <- attr.class_,
21   lowerValue <- oLowerMultiplicity,
22   upperValue <- oUpperMultiplicity
23 ),
24 oAss : distinct UML!Association foreach( attr in i.memberEnd)(
25   memberEnd <- attr,
26   memberEnd <- oAttr
27 ),
28 oLowerMultiplicity : distinct UML!LiteralInteger foreach (
29   attr in i.memberEnd ) (
30   value <- attr.lowerValue.value
31 ),
32 oUpperMultiplicity : distinct UML!LiteralUnlimitedNatural
33   foreach ( attr in i.memberEnd ) (
34   value <- attr.upperValue.value
35 )
36 do {
37   for ( attr in i.memberEnd ) {
38     attr.name <- i.name.firstToLower();
39     attr.type <- i;
40     attr.lowerValue.value <- 0;
41     attr.upperValue.value <- 1;
42   }
43 }
44 }

```

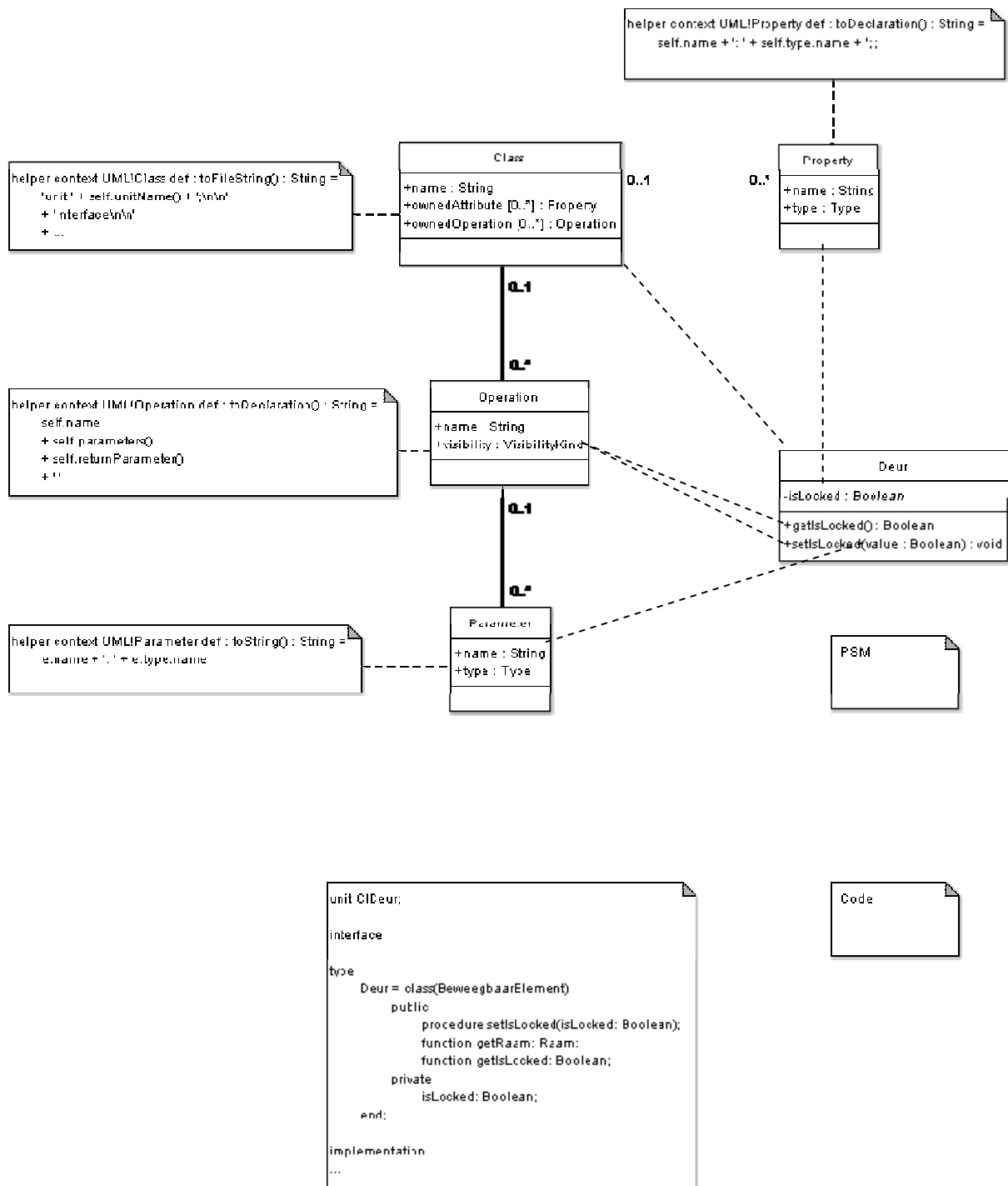
Figuur 6.8: Transformatieregel in ATL voor associatieklasse naar klasse

6.2.4 Transformatie van PSM naar code

De transformatie van het PSM naar code is een speciaal geval van transformatie. De oorzaak hiervoor is dat andere transformaties altijd tussen MOF 2.0-modellen gebeuren, zoals de QVT RFP (zie 5.1) specificeert. Codegeneratie gebeurt echter niet naar een MOF 2.0-model, maar naar een tekstbestand.

ATL biedt een mooie vorm van ondersteuning voor codegeneratie via de mechanismen voor zoekopdrachten en helperfuncties. Voor ieder element uit het metamodel kan een helperfunctie gedefinieerd worden die een element van het type van dit metamodel-element naar tekst omzet. Via een zoekopdracht wordt vervolgens over alle elementen van het bronmodel geïtereerd, waarbij ieder element naar tekst omgezet wordt via de helperfunctie voor het metamodel-element waaraan dit element voldoet. Dit is een gestructureerde en vooral transparante manier om een dergelijk mechanisme met ATL mogelijk te maken. Er zijn immers geen bijzondere constructies nodig om ook codegeneratie te ondersteunen.

De transformatie van PSM naar code met ATL bestaat dus uit de definitie van een zoekopdracht en een aantal helperfuncties. In Figuur 6.9 wordt de transformatie visueel voorgesteld. Er wordt een deel van het UML-metamodel weergegeven dat bestaat uit de UML-metamodel-elementen `Class`, `Operation`, `Parameter` en `Property`. De `Deur`-klasse uit het PSM is een instantie van `Class`, `isLocked` is een instantie van `Property`, `getIsLocked` en `setIsLocked` zijn instanties van `Operation`, en `value` is een instantie van `Parameter`. De stippellijnen tussen de elementen van het UML-metamodel en de `Deur`-klasse geven de “instantie van”-relatie weer. De overige stippellijnen verbinden de elementen van het UML-metamodel met een commentaarvak waarin een helperfunctie weergegeven wordt voor het omzetten van het metamodel-element naar tekst. Daarnaast heeft de transformatie een aantal hoog-niveau helperfuncties die de stukken tekst voor elk element van het bronmodel op de juiste manier samenvoegen. Tenslotte wordt het geheel naar een bestand geschreven.



Figuur 6.9: Transformatie van PSM naar code met ATL

E2S splitst de gegenereerde code over vier bestanden. ATL echter ondersteunt slechts één zoekopdracht per transformatiemodule. Dit betekent dat de transformatie van PSM naar code uit vier modules bestaat. Elke module bevat één zoekopdracht en de helperfuncties die hierin opgeroepen worden zijn telkens licht gewijzigd. Afhankelijk van het bestand (zie 6.1.2) genereren de helperfuncties code voor de declaratie van de unit, klasse, constructors en destructors (Pascalbestand), voor de declaratie van attributen (bestand voor constantendeclaraties), voor de declaratie van functies en operaties (declaratiebestand) of voor de implementatie van deze functies en operaties (implementatiebestand).

Pascalbestand

In Figuur 6.10 wordt de zoekopdracht voor de generatie van het Pascalbestand getoond. De zoekopdracht op regel 1 wordt met het sleutelwoord `query` aangeduid. De formulering van de zoekopdracht bestaat erin op regel 1 over alle instanties van `Class` uit het UML-metamodel te itereren, hiervoor op regel 3 een tekstfragment te genereren met de `toFileString` helperfunctie en dit eveneens op regel 3 naar een bestand te schrijven waarvan het pad bepaald wordt door de `pasFilePath` helperfunctie. Hierbij wordt op regel 2 nog een extra conditie ingevoerd om enkel de klassen te selecteren die een instantie zijn van `Class` zelf, en niet van de sub- of superklassen.

```
1 query UMLtoDelphiPasFile = UML!Class.allInstances()  
2   ->select( c | c.oclcIsTypeOf(UML!Class) )  
3   ->collect( c | c.toFileString().writeTo(c.pasFilePath()) );
```

Figuur 6.10: Fragment van generatie van Pascalbestand met ATL

Het meest interessant is de `toFileString` helperfunctie. Deze wordt getoond in Figuur 6.11. Dit is het eigenlijke beginpunt voor de codegeneratie. Vanaf dit punt kan de implementatie van helperfuncties vergeleken worden met de door E2S gegenereerde code uit Appendix A. Zoals in de code van het `.pas`-bestand in Appendix A te zien is, begint de declaratie van een unit met het `unit`-sleutelwoord, gevolgd door een spatie, de naam van de unit en een `;` (regel 2 in Figuur 6.11). De naam van de unit wordt afgeleid uit de naam van de UML-klasse uit het PSM via een `unitName`-helperfunctie. De implementatie van de `unitName`-helperfunctie voegt overeenkomstig de codegenerator van E2S een prefix `'Cl'` aan de naam van de klasse toe. Hierna volgt de declaratie van de interface van de unit, aangegeven door het `interface`-sleutelwoord. Vervolgens worden op regels 3, 4 en 5 helperfuncties aangeroepen voor het genereren van respectievelijk de importeerclausule, de declaratie en de implementatie van de klasse in de unit.

```
1 helper context UML!Class def : toFileString() : String =  
2   'unit ' + self.unitName() + ';\n\n' + 'interface\n\n'  
3   + self.importClause()  
4   + self.declaration()  
5   + self.implementation();
```

Figuur 6.11: Helperfunctie voor genereren van Pascalbestand met ATL

Het resultaat is een tekstfragment dat in het geval van de `Deur`-unit de vorm heeft als in Figuur 6.12. De importeerclausule, te herkennen aan de `uses`-instructie, en alles wat daarop volgt wordt gegenereerd door helperfuncties die verder beschreven worden.

```
1 unit ClDeur;  
2  
3 interface  
4  
5 uses
```

Figuur 6.12: Fragment van Pascalbestand gegenereerd met ATL

Zoals vermeld in 6.1.2 was één van de uitdagingen voor codegeneratie het verwijderen van dubbel ingevoegde units. Wanneer afhankelijke units ingevoegd worden, mag dit slechts éénmaal gebeuren omdat dit een compilerfout zou geven. In Figuur 6.13 wordt getoond hoe hieraan voldaan wordt. Zoals vermeld in 6.2.1 is ATL gebaseerd op de syntax van OCL. OCL biedt een aantal types aan voor

bijhouden van lijsten. Afhankelijk van het type lijst, wordt het dubbel voorkomen van elementen al of niet toegelaten. Een lijst kan ook naar een ander type omgezet worden. Het vermijden van dubbel ingevoegde units komt hierdoor neer op het verzamelen van alle afhankelijke units in een lijst en het omzetten van deze lijst naar een type waarbij dubbele voorkomens geëlimineerd worden. In Figuur 6.13 wordt getoond hoe dit gerealiseerd wordt. Op regel 3 worden voor een klasse alle geassocieerde klassen met behulp van een helperfunctie verzameld in een lijst. Van deze lijst wordt de unie gemaakt met de superklassen van deze klasse. Het resultaat van deze unie is een lijst van het type `Sequence`. Dit type laat dubbele voorkomens toe. Daarom wordt de lijst met de `asSet`-instructie omgezet naar een lijst van het type `Set`. In dit type komen alle elementen slechts eenmaal voor. Het resultaat is een lijst met unieke voorkomens van afhankelijke klassen. Vervolgens wordt deze lijst in regels 5 tot en met 13 naar tekst omgezet.

```

1 helper context UML!Class def : importClause() : String =
2 let importedClasses : Set(UML!Class) =
3   self.associatedClasses()->union( self.superClass )->asSet()
4 in
5   if importedClasses->notEmpty()
6   then
7     'uses \n\t' + importedClasses->iterate(c; acc : String = '' |
8       acc +
9       if acc = '' then c.unitName()
10      else ',\n\t' + c.unitName()
11      endif ) + ';\n\n'
12  else ''
13  endif;

```

Figuur 6.13: Helperfunctie voor genereren van importeerclausule met ATL

De volgende helperfunctie genereert de declaratie van de Delphi-klasse die deel uitmaakt van de unit. Deze helperfunctie wordt getoond in Figuur 6.14. Op regel 2 wordt het `type`-sleutelwoord ingevoegd, gevolgd door de naam en de eigenlijke declaratie van de klasse op regels 3 en 4 en de declaratie van constructors en destructors op regel 5. Op regels 6 en 7 wordt de code gegenereerd om het bestand voor constantendeclaraties en het declaratiebestand in te voegen. Hiervoor worden helperfuncties gebruikt. Het invoegen van bestanden gebeurt in Delphi met de `{$I <naam bestand>}`-compilerinstructie. Tenslotte wordt op regel 8 de declaratie van de Delphi-klasse afgesloten met het end-sleutelwoord.

```

1 helper context UML!Class def : declaration() : String =
2  'type\n\t'
3  + self.className() + ' = ' + 'class' + self.generalizationClause()
4  + '\n\n'
5  + self.declareConstructorsAndDestructors() + '\n'
6  + self.includeFidFile()
7  + self.includeDclFile()
8  + '\tend;\n\n';

```

Figuur 6.14: Helperfunctie voor genereren van declaratie voor Pascalbestand

Het resultaat voor het geval van de `Deur`-unit wordt weergegeven in Figuur 6.15. Dit tekstfragment sluit in het Pascalbestand aan op het tekstfragment uit Figuur 6.12. In de huidige implementatie bepaalt de `declareConstructorsAndDestructors`-helperfunctie op regel 5 in Figuur 6.14 de constructors en destructors van een UML-klasse uit de naam van de methode. Indien deze naam `Create` is, wordt de operatie als een constructor gedeclareerd, indien de naam `Destroy` is, wordt de operatie als een destructor gedeclareerd. Een betere methode voor het bepalen van constructors en

destructors zou zijn om bij de methodes van een UML-klasse te zoeken naar een methode met bijvoorbeeld een stereotype ‘Create’ of ‘Destroy’. Dit werd echter niet uitgewerkt. De reden hiervoor is dat het toepassen van een stereotype op een modelement via de gebruikte Eclipse UML2-plugin[KH05], het aanmaken van een aantal modelementen vereist en vrij complex is. Doordat een constructor of destructor in Delphi gewoonlijk de vermelde naam draagt, is de gevolgde werkwijze niet noodzakelijk inferieur.

```

5  uses
6      ClRaam,
7      ClBeweegbaarElement;
8
9  type
10     Deur = class(BeweegbaarElement)
11
12     public
13         constructor Create;
14         destructor Destroy;
15
16     {$I DbDeur.fid}
17
18     {$I DbDeur.dcl}
19
20     end;
21

```

Figuur 6.15: Fragment van Pascalbestand gegenereerd met ATL

Voor de generatie van het Pascalbestand rest zoals uit Figuur 6.11 kan afgeleid worden enkel nog de implementatie van de klasse. De helperfunctie die dit tekstfragment genereert, wordt weergegeven in Figuur 6.16. Deze helperfunctie gebruikt opnieuw andere helperfuncties. Doordat voor implementatie enkel de units van geassocieerde klassen hoeven ingevoegd te worden, genereert de implementationImportClause-helperfunctie op regel 5 een andere importeerclausule dan bij de declaratie van de Delphi-klasse het geval was. Vervolgens genereert de implementConstructorsAndDestructors-helperfunctie op regel 6 een lege implementatie voor de constructors en destructors van de klasse. Tenslotte voegt de includeImpFile-helperfunctie op regel 7 de implementatie van overige functies en procedures in via het implementatie(.imp-)bestand.

```

1  helper context UML!Class def : implementation() : String =
2  if self.ownedOperation->isEmpty() then ''
3  else
4      'implementation\n\n'
5      + self.implementationImportClause()
6      + self.implementConstructorsAndDestructors()
7      + self.includeImpFile()
8      + 'end.'
9  endif;

```

Figuur 6.16: Helperfunctie voor genereren van implementatie van Pascalbestand met ATL

Het resultaat voor de Deur-unit wordt weergegeven in Figuur 6.17. Merk op dat dit tekstfragment volgt op het tekstfragment uit Figuur 6.15.

```

21
22 implementation
23
24 uses
25   ClRaam;
26
27 constructor Deur.Create;
28 begin
29 end;
30
31 destructor Deur.Destroy;
32 begin
33 end;
34
35 {$I DbDeur.imp}
36
37 end.

```

Figuur 6.17: Fragment van Pascalbestand gegenereerd met ATL

Het gegenereerde Pascalbestand vormt het geheel van de tekstfragmenten uit Figuur 6.12, Figuur 6.15 en Figuur 6.17, en is terug te vinden in Appendix D. De verschillen tussen dit bestand en het Pascalbestand dat door E2S gegenereerd wordt uit Appendix A, zijn miniem.

Een eerste verschil is dat de naam van de Delphi-klasse bij E2S een prefix 'Cls_' krijgt. Bij het voorbeeld van de Deur-unit declareert E2S dus een klasse met de naam Cls_Deur, terwijl in de eigen implementatie deze klasse de naam Deur draagt. Het is eenvoudig om dit ook met ATL te realiseren. Dit werd echter niet gevolgd om het verband tussen PSM en gegenereerde code duidelijker te maken.

Een tweede verschil bevindt zich in de constructor en destructor gegenereerd met ATL. Waar bij E2S in de constructor een verwijzing voorkomt naar een integer met de naam nObjID en een instantie van Factory, wordt bij ATL een standaard constructor met lege implementatie gegenereerd. De reden hiervoor is dat zoals vermeld in 6.1.2 abstractie gemaakt wordt van het opslagmechanisme via een databank. Deze attributen dienen immers ter referentie naar een databankelement. Het zou mogelijk zijn met ATL iedere instantie van een klasse en de attributen en associaties die ertoe behoren als een object in een databank te modelleren. Dit zou ons echter veel te ver leiden. De destructor wordt bij E2S om dezelfde reden overgeërfd van de superklasse en hergedefinieerd, maar ook daarvan kunnen we abstractie maken en een standaard destructor voorzien.

Een derde verschil bestaat uit de identificatie van het afgesloten codeblok na elk voorkomen van het end-sleutelwoord door een {...}-vermelding. Deze identificatie is toegelaten, maar niet strikt nodig volgens de Delphi-syntax. Om de duidelijkheid van de gegenereerde code te verbeteren, werd deze identificatie bij ATL weggelaten.

Overige bestanden

De transformatiemodules voor generatie van declaratie- en implementatiebestand en het bestand voor constantendeclaraties volgen dezelfde werkwijze als bij de generatie van het Pascalbestand. Er wordt een zoekopdracht geformuleerd die over het bronmodel itereert, in combinatie met een aantal helperfuncties die elk modelement naar tekst omzetten. Het is onnodig deze transformatiemodules opnieuw gedetailleerd toe te lichten. De volledige uitwerking kan teruggevonden worden in Appendix C. De code gegenereerd door de verschillende transformatiemodules is opnieuw heel gelijkaardig aan de code gegenereerd door E2S. De code gegenereerd door de transformatiemodules is terug te vinden

in Appendix D. In wat volgt worden enkel de belangrijkste elementen uit de transformatiemodules en de voornaamste verschillen bij de gegenereerde code toegelicht.

De transformatiemodule voor het genereren van het declaratiebestand brengt geen bijzondere elementen aan ten opzichte van de transformatiemodule voor generatie van het Pascalbestand. De methodes van de UML-klasse uit het PSM worden gerangschikt volgens publieke, *protected* en private methoden. Vervolgens worden de sleutelwoorden *public*, *protected* en/of *private* ingevoegd, indien er methoden met deze visibiliteit aanwezig zijn. Daarna wordt elke van deze methoden omgezet naar een tekstfragment met de declaratie van de functie of procedure die met de methode overeenstemt en worden deze tekstfragmenten op de juiste manier samengevoegd. Bij de gegenereerde code zijn de enige verschillen het weglaten van de methodes voor interactie met het databankmechanisme zoals vermeld in 6.1.2 en de andere naam voor de `getRaam`-functie, een gevolg van het modelleren van raam als een enkelvoudig attribuut.

In het implementatiebestand wordt voor deze functies en procedures een implementatie voorzien. Hierbij is het interessant te vermelden dat de transformatiemodule voor inspectors en mutators ook een lichaam genereert. Dit lichaam gaat de waarde van het attribuut dat met de inspector of mutator overeenstemt terugkeren of wijzigen. Hier treedt een moeilijkheid bij het bepalen van dit attribuut. De transformatiemodule dient immers op een zekere manier te bepalen welk attribuut met de inspector of mutator overeenstemt. In de uitwerking wordt dit gerealiseerd aan de hand van de naam van de inspector of mutator. In de transformatie van publieke naar private attributen die deel uitmaakt van de transformatie van PIM naar PSM (zie 6.2.3) wordt bepaald dat de naam van inspector of mutator samengesteld wordt door 'get' respectievelijk 'set', gevolgd door de naam van het attribuut waarvan de eerste letter naar een hoofdletter omgevormd wordt. De transformatiemodule voor generatie van het implementatiebestand past dit in de omgekeerde richting toe. Als de transformatie van publieke naar private attributen consequent op deze manier toegepast wordt, levert dit geen problemen op. Een betere manier zou opnieuw zijn om een stereotype toe te passen op de methode. Dit werd echter om de vermelde reden niet uitgewerkt. Bij de gegenereerde code merken we opnieuw op dat de functies en procedures voor interactie met de databank weggelaten werden. Dit geldt ook voor de uitwerking van het lichaam van inspectors en mutators, waar niet met databankelementen gewerkt wordt, maar met de attributen van de klasse.

Het bestand voor constantendeclaraties tenslotte voert de declaratie van deze attributen in. Analoog aan de transformatiemodule voor generatie van het declaratiebestand, worden de attributen gerangschikt volgens visibiliteit. Het verband met het bestand voor constantendeclaraties gegenereerd door E2S vervalt, maar dit is enkel een gevolg van het weglaten van het databankmechanisme. De declaratie van attributen zoals gegenereerd door de transformatiemodule is een alternatieve maar gelijkwaardige werkwijze, en syntactisch correct.

Het samenvoegen van deze bestanden gebeurt zoals vermeld bij het Pascalbestand door de `{ $I <naam bestand> }`-compilerinstructie die deel uitmaakt van de Delphi-syntax [CGL97]. Indien de inhoud van declaratie-, implementatiebestand en bestand voor constantendeclaraties manueel zou geplaatst worden waar in het Pascalbestand deze compilerconstructies staan, krijgen we een volledige en syntactisch correcte Delphi-unit en -klasse. Een transformatiemodule die voor iedere klasse uit het PSM de declaratie en implementatie van de Delphi-unit en -klasse in één bestand genereert, werd eveneens uitgewerkt. Deze transformatiemodule is minder complex dan de samenvoeging van de vier hoger besproken transformatiemodules, doordat de volledige generatie in één enkele module wordt uitgewerkt en geen herhalingen of variaties optreden. Een volledige transformatie van UML-klassendiagram, bijvoorbeeld PSM, naar Delphi-code vormt deze module niet. De volledige Delphi-syntax werd niet geïmplementeerd, maar de meest belangrijke elementen zijn net als bij de vier vorige modules aanwezig. In Appendix C is de alternatieve transformatiemodule terug te vinden. De code gegenereerd door deze module voor het beschouwde voorbeeld is terug te vinden in Appendix D.

De flexibiliteit en mogelijkheid tot structureren van het mechanisme voor codegeneratie kunnen we als grote voordelen voor ATL beschouwen. Zelfs al is dit soort transformatie niet wat beoogd wordt met

een taal als ATL, toch blijkt het mogelijk de transformatie op eenvoudige manier te formuleren. Ook de transparantie van dit mechanisme is hoog.

6.2.5 Integratie van ATL met HOORA

Voor dit eindwerk werd ATL als plugin voor Eclipse [SH04] gebruikt. Deze plugin biedt mogelijkheden tot bewerken, compileren, uitvoeren en *debuggen* van ATL-bestanden. Het is echter ook mogelijk de ATL-modeltransformator als Java-bibliotheek aan te spreken. Zo kan dezelfde functionaliteit bereikt worden vanuit andere applicatiesuites, zoals HOORA. Voor E2S is dit een mogelijkheid om ATL rechtstreeks te gebruiken.

ATL steunt ook op het gebruik van EMF en UML2. Beide worden eveneens als Eclipse-plugin (zie 6.1.1) gebruikt, maar het is onduidelijk of ook deze als Java-bibliotheek kunnen gebruikt worden. Aangezien de specificatie ervan niets vermeldt, gaan we ervan uit dat dit niet het geval is. Dit betekent dat E2S een eigen implementatie zou moeten voorzien van EMF voor metamodellering en modelopslag, en van de UML2-plugin om grafisch te modelleren. Momenteel ondersteunt HOORA UML 1.3, dus het laatste aspect komt neer op het aanpassen van de suite naar de UML 2.0-standaard.

Verder moet opgemerkt worden dat met de ondersteuning van MDA, het niveau waarop gemodelleerd wordt zou verschuiven van PSM naar PIM. Minder implementatiedetails zouden weergegeven worden, waardoor de modellering meer platformafhankelijk wordt. Een transformatie van PIM naar PSM en vervolgens van PSM naar code zou de codegeneratie volledig maken.

6.3 MTF

In 5.2.4 werd het QVT 1.0-voorstel nader toegelicht. Hierin worden relaties en afbeeldingen gedefinieerd, waarbij afbeeldingen de implementatie voor relaties vormen. MTF [IBM04] is een proefimplementatie van QVT 1.0. De denkwijze die bij MTF gevolgd wordt is, conform het QVT 1.0-voorstel, declaratief. Dit maakt volledig bidirectionele transformaties mogelijk. Het uitvoeringsmechanisme van MTF laat toe dit te variëren van non-directioneel (controleren of twee modellen aan een transformatie voldoen) over unidirectioneel in elke richting tot bidirectioneel..

6.3.1 Overzicht

MTF werd ontwikkeld door IBM en heet voluit *Model Transformation Framework*. Het concept van relatie in MTF komt overeen met de formulering van het verband tussen twee of meer domeinen. Dit is analoog aan wat we in Java de hoofding van een methode noemen. Wat we analoog in Java het lichaam van een methode noemen, heet in QVT 1.0 een afbeelding, dus de implementatie van een relatie. Dit is ook in MTF zo.

Een afbeeldingregel bestaat in MTF uit een verzameling beperkingen die moeten gecontroleerd worden voor één of meer objecten. Als deze beperkingen één voor één gelden, voldoen de objecten aan de transformatieregel en moeten deze niet gewijzigd worden. Afbeeldingregels worden gecontroleerd binnen een afbeeldingssessie. Deze sessie controleert welke regels voor welke objecten zouden moeten gelden.

Het controleren van een afbeeldingregel kan als gevolg hebben dat ook voor waarden afgeleid van objecten die afgebeeld worden, regels moeten gecontroleerd worden. Als dit voorkomt zijn deze nieuwe afbeeldingen deel van de oorspronkelijke afbeelding, en wordt op deze manier een hiërarchie van afbeeldingen gecreëerd. Het is belangrijk hierbij op te merken dat binnen deze hiërarchie nooit twee afbeeldingen van hetzelfde type dezelfde objecten kunnen afbeelden. Verder is het handig te weten dat dit afbeeldingmechanisme door zijn declaratieve aard volledig bidirectioneel is.

Voor het implementeren van zoekopdrachten wordt in MTF het mechanisme voor patrooncorrelatie gebruikt. Indien een gebruiker dus een waarde wil opvragen, kan dat door een ongebonden variabele als parameter voor een relatie of beperking mee te geven.

6.3.2 Verhouding QVT-MTF

De verhouding tussen QVT en MTF werd reeds in 6.3.1 licht aangehaald. De doelen die bij het ontwerp van MTF vooropgesteld werden zijn de volgende:

- vlotte integratie met EMF modellen.
- het gemakkelijk maken om bidirectionele transformaties te ontwikkelen.
- het ondersteunen van veelrichtingstransformaties.
- toelaten van het schrijven van code in Java of andere talen voor het selecteren, matchen en construeren van delen van het (de) model(len).
- opvolgen van welke elementen waarop werd afgebeeld.

MTF dient niet als een precieze implementatie van QVT 1.0, maar louter als proefimplementatie van de concepten. Enkele van de ideeën die uit deze proefimplementatie geleerd werden, werden in het QVT 2.0 voorstel verwerkt, al geldt ATL (zie 6.2) als een belangrijkere invloed.

6.3.3 Transformatie van PIM naar PSM

Omdat we weten dat QVT 2.0 door ondermeer betere ondersteuning voor imperatieve syntax, een krachtiger voorstel is dan QVT 1.0, werden de transformatieregels in MTF voornamelijk ter

vergelijking opgesteld. De transformatie van PIM naar PSM is in dit deel niet volledig, maar laat ons toe de taal en het QVT 1.0-voorstel te evalueren.

De transformatieregels voor toevoegen van een constructor aan de klassen van het model en omvormen van publieke naar private attributen, met toevoeging van inspectors en mutators, werd in dit geval in één bestand gegroepeerd. MTF biedt namelijk geen ondersteuning voor modules. Bovendien kent MTF het concept van verfijningsmode, uit ATL, niet. Dit betekent dat, zoals aangehaald in 6.2.1, grote delen van het model moeten worden gekopieerd door expliciete transformatieregels. De meest belangrijke aspecten van de implementatie van de twee aangehaalde transformatieregels worden gegeven in Figuur 6.18. Waar ‘...’ voorkomt werden minder belangrijke aspecten weggelaten. De volledige implementatie is te vinden in Appendix F.

```

1 ...
2 relate umlclass2class( uml:Class class when !(util:InstanceOf
3 "uml:Stereotype" (class)), uml:Class outClass )
4 {
5     equals( class.name, outClass.name ),
6     // Attributen transformeren
7     ordered umlproperty2property( over class.ownedAttribute, over
8         outClass.ownedAttribute ),
9     umlproperty2inspector( over class.ownedAttribute, match over
10        outClass.ownedOperation ),
11    umlproperty2mutator( over class.ownedAttribute, match over
12        outClass.ownedOperation ),
13    // Operaties van klasse kopiëren
14    umloperation2operation( over class.ownedOperation, match over
15        outClass.ownedOperation ),
16    // Constructor toevoegen
17    umlclass2constructor( over class, match over outClass.ownedOperation
18        ),
19    // Destructor toevoegen
20    umlclass2destructor( over class, match over outClass.ownedOperation
21        )
22 }
...
30 relate umlproperty2property( uml:Property prop, uml:Property outProp )
31 {
32     equals( prop.name, outProp.name ),
33     equals( prop.class_, outProp.class_ ),
34     equals( prop.type, outProp.type ),
35     equals( outProp.visibility, "private" )
36 }
37
38 relate umlproperty2inspector( uml:Property prop, uml:Operation operation
39 ) when util:MatchString "get_{0}" (prop.name, operation.name)
40 {
41     // Terugkeertype van operatie op type van attribuut zetten
42    setoperationreturntype [1] ( prop, match over
43        operation.ownedParameter )
44 }
45
46 relate setoperationreturntype( uml:Property prop, uml:Parameter
47 parameter when equals( parameter.direction, "return" ) )
48 {
49     equals( prop.type, parameter.type )
50 }
...
59 relate umlproperty2mutator( uml:Property prop, uml:Operation operation )
60 when util:MatchString "set_{0}" (prop.name, operation.name)
61 {
62     ordered umlattribute2parameter( over prop, match over

```

```

63   operation.ownedParameter )
64 }
...
70 relate umlattribute2parameter( uml:Property prop, uml:Parameter
71   parameter)
72 {
73   equals( prop.name, parameter.name ),
74   // Type van parameter van moderator op type van attribuut zetten
75   equals( prop.type, parameter.type )
76 }
...
91 relate umlparameter2parameter( uml:Parameter parameter, uml:Parameter
92   outParameter )
93 {
94   equals( parameter.name, outParameter.name ),
95   equals( parameter.type, outParameter.type ),
96   equals( parameter.direction, outParameter.direction )
97 }
...
105 relate umlclass2constructor( uml:Class class, uml:Operation operation )
106 when equals("Create", operation.name )
...
109 relate umlclass2destructor( uml:Class class, uml:Operation operation )
110 when equals("Destroy", operation.name )

```

Figuur 6.18: Transformatieregel in MTF voor toevoegen van constructor en destructor en transformeren van publieke naar private attributen

De relaties in deze formulering die gelden als transformatieregels, zijn de relaties `umlclass2class`, `umlclass2constructor` en `umlclass2destructor`. De relatie `umlclass2class` (regels 2 tot en met 22) geldt als het beginpunt voor de transformatie en voert de volledige transformatie van PIM naar PSM uit. De relatie maakt hiervoor gebruik van enkele deelrelaties.

De relatie `umlclass2constructor` (regels 105 en 106) voegt aan elke klasse een constructor toe. Deze relatie wordt gevestigd tussen het domein van klassen en het domein van operaties. Het aanroepen van de relatie bij de `umlclass2class`-relatie op regel 17 legt een verband tussen deze domeinen vast, zijnde het domein van klassen en het domein van operaties die tot deze klasse behoren. De formulering van de transformatieregel op regels 105 en 106 is vrij eenvoudig en stelt, net als bij ATL, enkel dat er tot de operaties van de klasse een operatie moet behoren met de naam ‘Create’.

De relatie `umlclass2destructor` (regels 109 en 110) voegt analoog aan elke klasse een destructor toe. Het aanroepen van de relatie bij de `umlclass2class`-relatie op regel 20 legt opnieuw een verband tussen de domeinen van de relatie vast als het domein van klassen en het domein van operaties die tot deze klasse behoren. De formulering van de transformatieregel op regels 109 en 110 stelt dat tot de operaties van de klasse een operatie moet behoren met de naam ‘Destroy’.

De relatie `umlproperty2property` op regels 30 tot en met 36 zet op regel 35 de visibiliteit van een instantie van `Property` instantie op `privaat`. De relatie `umlproperty2inspector` op regels 38 tot en met 44 heeft als domeinen de attributen en operaties van een klasse, en voegt een `inspector` aan de klasse toe. Merk op dat op regel 39 de naam van deze `inspector` via een functie uit de bibliotheek `util` bepaald wordt. Deze bibliotheekfunctie gebruikt opnieuw patrooncorrelatie, maar zeer leesbaar is dit niet. De relatie `setoperationreturntype` op regels 46 tot en met 50 voegt aan deze `inspector` een terugkeertype toe dat hetzelfde type is als het type van het attribuut dat getransformeerd wordt (regel 49). In het operatiedomein wordt op regel 47 een conditie gedefinieerd die bepaalt dat deze parameter de richting “return” moet hebben.

De relatie `umlproperty2mutator` op regels 59 tot en met 64 transformeert een attribuut naar een mutator. Dit gebeurt op analoge wijze als het transformeren naar een inspector. Met de relatie `umlattribute2parameter` op regels 70 tot en met 76 wordt vervolgens via het aanroepen binnen de `umlproperty2mutator`-relatie op regel 62 een parameter aan deze operatie toegevoegd die hetzelfde type heeft als het attribuut dat getransformeerd wordt. De attributen van een klasse zijn een geordende verzameling, waardoor het sleutelwoord **ordered** moeten gebruiken om aan te geven dat er over een georderende verzameling geïtereerd wordt. Aangezien dit ons echter geen voordelen heeft, beschouwen dit als een vorm van lastige syntax.

Verder moet bij het transformeren van een attribuut of associatie-einde naar een inspector of mutator van een klasse aangegeven worden dat deze klassen ook andere parameters kunnen hebben. Dit gebeurt bij het aanroepen binnen de `umlclass2class`-relatie op regels 9 en 11 met het sleutelwoord **match**. Het is echter niet duidelijk wanneer dit mechanisme precies wel of niet moet gebruikt worden. Zo moet voor de correcte werking van de transformatie bij relateren van de terugkeer- en invoerparameter met respectievelijk de inspector of mutator, ook aangegeven worden deze operaties ook andere parameters kunnen hebben. Niettemin zijn deze parameters altijd uniek.

Het kan opgemerkt worden dat er algemeen weinig sleutelwoorden in MTF zijn. De bestaande sleutelwoorden dienen voor het definiëren van relaties (`relate`) en bepalen van de manier waarop over verzamelingen moet geïtereerd worden (`ordered`, `match` en `ref`). Voor het definiëren van de relaties zelf is enkel `equals` nuttig, wat bepaalt dat twee elementen gelijk moeten zijn. Het ontbreken van meer sleutelwoorden leidt al gauw tot het gebruiken van externe bibliotheken, bijvoorbeeld voor het gebruik van de `MatchString` functie uit de `util`-bibliotheek (regels 39 en 60). Deze bibliotheek is echter zeer beperkt en biedt slechts enkele functies aan. De syntax van deze functies zijn bovendien weinig transparant, zoals voor het bepalen van de naam van een inspector of mutator (regels 39 en 60) aangetoond wordt.

6.3.4 Transformatie van PSM naar code

Omdat MTF heel zwakke ondersteuning biedt voor transformatie van modelementen naar tekst, werd deze transformatie hier niet uitgewerkt. De makers van MTF stellen voor om hiervoor een extern mechanisme te gebruiken, zoals een MOF2Text-taal. Dit is uiteraard een nadeel doordat de leercurve van MTF daardoor verhoogd wordt met de leercurve van dit extern mechanisme. Bovendien is het mogelijk dat met een extern mechanisme bepaalde mogelijkheden verloren gaan.

6.3.5 Integratie van MTF met HOORA

Net zoals bij ATL het geval is, kan MTF als Java-bibliotheek aangesproken worden. Een integratie van MTF met HOORA, zou dus neerkomen op het uitvoeren van een aantal operaties in deze bibliotheek.

Ook MTF gebruikt EMF en de UML2-plugin (zie 6.1.1). Opnieuw zou E2S hiervoor een eigen implementatie moeten voorzien.

De opmerking over het verschuiven van PSM naar PIM geldt opnieuw. Het niveau van abstractie kan verhoogd worden, en modelleren op het niveau van het PIM wordt mogelijk.

Hoofdstuk 7

Conclusie

De meest belangrijke doelstelling van dit eindwerk was het vinden van een modeltransformatietaal die aan de vereisten van E2S voldoet, en ruimte laat voor uitbreiding. Na het definiëren van de vereisten in Hoofdstuk 4 zijn we in de literatuur op zoek gegaan naar een geschikte taal. MDA en meer specifiek modeltransformatietalen zijn een zeer recente ontwikkeling in de software-ontwikkelingswereld. De beschikbare transformatietalen bevinden zich hierdoor vaak nog in een experimenteel of zuiver theoretisch stadium, waardoor ze vaak grote gebreken vertonen en een hoge instapdrempel hebben. Het was bijgevolg niet eenvoudig om een geschikte transformatietaal te vinden. De literatuurstudie uit Hoofdstuk 5 werd dan ook vrij uitgebreid, maar met resultaat. Vervolgens maakten we een proefimplementatie van een aantal transformatieregels in twee uitgekozen talen, met als doel de beste kandidaat te selecteren.

7.1 Literatuurstudie

In de literatuurstudie werden een aantal voorstellen bestudeerd die als antwoord op de OMG QVT RFP werden ingediend. Onder de bestudeerde talen bevinden zich xUML, XMOF, AIM, QVT 1.0 en QVT 2.0. Deze talen hebben elk hun sterke en zwakke punten. Een grondige analyse en evaluatie van elke taal leerde ons dat bij de eerste voorstellen de zwakke punten doorwegen. De zuiver theoretische, onvolledige of onduidelijke aard van deze voorstellen maakt implementatie van de voorgestelde taal moeilijk. QVT 1.0 en QVT 2.0 brachten echter betere oplossingen. Vooral de hoge graad van volledigheid van deze voorstellen, en duidelijk gespecificeerde concepten beschouwen we als voordelen. Van elk van deze twee voorstellen is bovendien een implementatie beschikbaar. Deze implementaties werden na de literatuurstudie met elkaar vergeleken.

7.2 Proefimplementatie van transformatieregels

Om de mogelijkheden van elk van deze twee talen te bewijzen, realiseerden we in Hoofdstuk 6 een proefimplementatie van enkele transformatieregels. Deze transformatieregels werden afgeleid uit een studie van de bestaande E2S-software. Het bleek heel goed mogelijk deze transformatieregels in ATL, een implementatie van QVT 2.0, neer te schrijven. We vergeleken bovendien deze proefimplementatie met een proefimplementatie van QVT 1.0, en stelden vast dat QVT 2.0 veel beter werkbaar is, en meer mogelijkheden biedt. Dit kon ook deels verwacht worden doordat QVT 2.0 een verdere ontwikkeling van QVT 1.0 is.

We slaagden erin een platformafhankelijk model te definiëren voor de te ontwerpen applicaties. Dit platformafhankelijk model kan voortaan het niveau van abstractie zijn waarop de E2S-software werkt. Via een transformatie, gespecificeerd met ondermeer de transformatieregels die hier uitgewerkt werden, kan dit model getransformeerd worden naar een model specifiek voor een bepaalde programmeertaal. Enkele details uit de codegeneratiepatronen werden geabstraheerd, maar doordat we transformatieregels van een zekere complexiteit toegevoegd hebben, hebben we bewezen dat ook dit via transformatieregels moet kunnen gespecificeerd worden.

ATL kan rechtstreeks geïmplementeerd worden in werktuigen voor modeltransformatie, dus het type werktuig dat E2S wenst te ontwerpen. ATL biedt behalve de bewezen kwaliteiten als modeltransformatietaal dus ook een oplossing voor integratie. Daarnaast is ook OCL, de basis voor

ATL, reeds gekend bij E2S, en is de syntax eenvoudig zodat een programmeur zich bij het ontwikkelen met ATL vooral kan concentreren op de transformatie zelf.

MTF, als implementatie van QVT 1.0, heeft enkele problemen. De syntax is vrij ontoegankelijk en het uitvoeringsmechanisme biedt weinig oplossing. Doordat de documentatie van MTF van vrij lage kwaliteit is, is het ook moeilijk meer informatie over de precieze werking op te zoeken. In het voordeel spreekt wel de automatische aanvulling van syntax bij MTF. Dit maakt bladeren door een uitgebreid model makkelijker.

Indien er meer tijd was geweest had ik graag nog de implementatie in MTF uitgebreid om het vergelijken van beide talen iets beter te kunnen argumenteren. Het ontwikkelen van transformatieregels met MTF is door de arme syntax en declaratieve aard echter bijzonder tijdsrovend. Dit is op zich eveneens een goed argument om niet voor deze taal te kiezen.

Uiteindelijk denken we met QVT 2.0 en de implementatie in ATL een heel goede taal gevonden te hebben. Het is een heel volwassen taal die niet alleen aan al onze vereisten voldoet, maar ook een instapdrempel heeft die haalbaar is. Daarnaast specificeert de taal heel wat opties die uitbreiding voor allerlei toepassingen toelaat. Ook is het voorstel heel volledig en gedetailleerd, wat implementatie in een MDA-werktuig toelaat.

7.3 Moeilijkheden en uitdagingen

De moeilijkheden die ondervonden werden in dit eindwerk waren enerzijds de experimentele stadia waarin de bestudeerde talen en technieken zich nog bevinden. De bestudeerde voorstellen bevatten veelal fouten en onduidelijkheden, of waren onvolledig. Ook de gebruikte werktuigen waren veelal arm of bevatten fouten in de implementatie. Samen met de beperkt beschikbare of onduidelijke informatie maakt dit een correcte evaluatie van een taal of techniek moeilijk, maar mits de nodige moeite en tijd toch mogelijk. Anderzijds waren enkele bestudeerde technieken zuiver conceptueel en eerder ontoegankelijk.

Als slot kan het passend zijn enkele, vanuit mijn persoonlijk standpunt, toekomstige uitdagingen voor MDA te vermelden.

Een groot voordeel van een transformatietaal als ATL, is dat verschillende metamodellen tegelijk kunnen geladen worden. Zo kan informatie uit een klassendiagram samengevoegd worden met informatie uit een ander UML-diagram, of uit een aantal constanten die in een ander meta-model worden gespecificeerd. Dit laat het genereren toe van code met heel veel informatie.

Code die gegenereerd wordt door een reeks van transformaties, zal typisch aangevuld worden door programmeurs. Het verband tussen een model en de code die ervoor gegenereerd wordt, gaat daarmee verloren. QVT biedt mechanismen aan voor tweerichtingstransformaties, maar het is onduidelijk in hoeverre dit in een dergelijk geval ondersteund wordt. Ook volledig bidirectionele transformaties zijn daarom in mijn mening een toekomstige uitdaging.

Appendix A: Code gegenereerd door HOORA

DbDeur.dcl

```
{ Pluggable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55}

public
class function TableIndex: Integer; override;
procedure SetIsLocked(value: Boolean);
function GetIsLocked: Boolean;
function GetAllRaam: ObjectList;
class procedure Initialize(pTblInfo: CTableInfo; szIDFieldName:
AnsiString = '');
```

DbDeur.imp

```
{ Pluggable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55}

class function Cls_Deur.TableIndex;
begin
    Result := CID_TBL_DEUR
end;

procedure Cls_Deur.SetIsLocked;
begin
    SetBooleanValue(FID_TBL_DEUR_ISLOCKED, value)
end;

function Cls_Deur.GetIsLocked;
begin
    Result := GetBooleanValue(FID_TBL_DEUR_ISLOCKED)
end;
```

```
function Cls_Deur.GetAllRaam;
begin
    Result := GetRelatedByIx(CID_TBL_RAAM, FID_TBL_RAAM_DEUR_REF)
end;

class procedure Cls_Deur.Initialize;
var
    fldInfo: CTFieldInfo;

begin
    if System.Length(szIDFieldName) = 0
    then
        szIDFieldName := 'Fld_ID';
        fldInfo := CTFieldInfo.CreateBoolean('Fld_IsLocked',false,True);
        pTblInfo.AddFieldInfo(FID_TBL_DEUR_ISLOCKED, fldInfo, False);
    end;
```

DbDeur.fid

```
{ Pluggable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55}
```

```
const
```

```
START_FID_DEUR = END_FID_BEWEEGBAARELEMENT;
```

```
FID_TBL_DEUR_ISLOCKED = START_FID_DEUR + 1;
```

```
END_FID_DEUR = START_FID_DEUR + 1;
```

ClDeur.pas

```
{ $B- } {Pluggable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55}
```

```
unit ClDeur;
```

```
interface
```

```
uses
```

```
Classes,
```

```
ObjList,
```

```
DbObject,
```

```
ClBeweegbaarElement;
```

```
{ $I DbDeur.fid }
```

```
type
```

```
ClS_Deur = class(Cls_BeweegbaarElement)
```

```
public
```

```
constructor Create(nObjID: Integer;
```

```
pFactory: CTBaseFactory = nil); override;
```

```
destructor Destroy; override;
```

```
private
```

```
procedure InitAttributes;
```

```
{ $I DbDeur.dcl }
```

```

end { class Cls_Deur };

implementation

uses
    ClRaam,
    Factory;

{$I DbDeur.imp}

constructor Cls_Deur.Create;
begin
    inherited Create(nObjID, pFactory);

    InitAttributes
end { Cls_Deur.Create };

destructor Cls_Deur.Destroy;
begin
    inherited
end { Cls_Deur.Destroy };

procedure Cls_Deur.InitAttributes;
begin
end { Cls_Deur.InitAttributes };

end { unit ClDeur }.

```

DbRaam.dcl

```
{ Pluggable Objects 5.3.0 - WS23\Student 22-09-2004 12:38:56 PM }
public
class function TableIndex: Integer; override;
procedure SetDeur(pObj: Pointer);
procedure DropDeur;
function GetDeur: Pointer;
procedure GetDeur_REF(var nClsID, nObjID: Integer);
class procedure Initialize(pTblInfo: CTTableInfo; szIDFieldName:
AnsiString = '');
```

DbRaam.imp

```
{ Pluggable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55 }
```

```
class function Cls_Raam.TableIndex;
begin
    Result := CID_TBL_RAAM
end;

procedure Cls_Raam.SetDeur;
begin
    SetRelationByIx(FID_TBL_RAAM_DEUR_REF, pObj)
end;

procedure Cls_Raam.DropDeur;
begin
    DropRelationByIx(FID_TBL_RAAM_DEUR_REF)
end;

function Cls_Raam.GetDeur: Pointer;
begin
    Result := GetRelationByIx(FID_TBL_RAAM_DEUR_REF)
end;

procedure Cls_Raam.GetDeur_REF;
begin
```

```

    GetRefValues(FID_TBL_RAAM_DEUR_REF, nClsID, nObjID)
end;

class procedure Cls_Raam.Initialize;
var
    fldInfo: CTFieldInfo;

begin
    if System.Length(szIDFieldName) = 0
    then
        szIDFieldName := 'Fld_ID';
    Cls_BeweegbaarElement.Initialize(pTblInfo, szIDFieldName);

    fldInfo := CTFieldInfo.CreateInteger(
        'Fld_Deur_CLS_REF',
        0,
        True);
    pTblInfo.AddFieldInfo(FID_TBL_RAAM_DEUR_REF, fldInfo);
    fldInfo := CTFieldInfo.CreateInteger(
        'Fld_Deur_OBJ_REF',
        0,
        True);
    pTblInfo.AddFieldInfo(FID_TBL_RAAM_DEUR_REF + 1, fldInfo);
end;

DbRaam.fid

{ Pluggable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55 }
const
START_FID_RAAM = END_FID_BEWEEGBAARELEMENT;
FID_TBL_RAAM_DEUR_REF = START_FID_RAAM + 1;
END_FID_RAAM = START_FID_RAAM + 2;

```

ClRaam.Pas

```
{ $B- } { Plugable Objects 5.3.0 - SOCRATES\Didier 22/02/2005 14:15:55 }  
}
```

unit ClRaam;

interface

uses

```
Classes,  
ObjList,  
DbObject,  
ClBeweegbaarElement;
```

```
{ $I DbRaam.fid }
```

type

```
Cls_Raam = class(Cls_BeweegbaarElement)
```

public

```
constructor Create(nObjID: Integer;  
                  pFactory: CTBaseFactory = nil); override;  
destructor Destroy; override;
```

private

```
procedure InitAttributes;
```

```
{ $I DbRaam.dcl }
```

```
end { class Cls_Raam };
```

implementation

```
uses
    Factory;

{$I DbRaam.imp}

constructor Cls_Raam.Create;
begin
    inherited Create(nObjID, pFactory);

    InitAttributes
end { Cls_Raam.Create };

destructor Cls_Raam.Destroy;
begin
    inherited
end { Cls_Raam.Destroy };

procedure Cls_Raam.InitAttributes;
begin

end { Cls_Raam.InitAttributes };

end { unit ClRaam }.
```

Appendix B: ATL syntax

ATL modules

ATL programma's voor transformatie van model naar model worden **modules** genoemd. Een module bestaat uit een hoofding, een importeersectie, evenals helperfuncties en de eigenlijke transformatieregels. Voor een ATL-module is enkel de hoofding vereist.

Hoofding

De hoofding start met het sleutelwoord `module` gevolgd door de naam van de module. Daarna volgen het bron- en doelmodel die gedeclareerd worden als variabelen getypeerd door hun metamodellen. Het sleutelwoord `create` duidt het doelmodel aan. Het sleutelwoord `from` duidt het bronmodel of de bronmodellen aan. Bij deze declaraties wordt het bronmodel vaak simpelweg `IN` genoemd, het doelmodel `OUT`. De hoofding is, zoals reeds vermeld, de enige verplichte sectie. Een voorbeeld:

```
module UML2RDBMS ;  
create OUT : RDBMS from IN : UML ;
```

Dit voorbeeld neemt dus een UML-schema als invoer en produceert een relationele databankschema (RDBMS) als uitvoer. Dit zou dus de hoofding kunnen zijn van een transformatie van een klassendiagram naar een relationeel databankschema.

Importeersectie

De importeersectie bepaalt welke bibliotheken dienen te worden geïmporteerd. Om bijvoorbeeld de strings bibliotheek te importeren, zou men schrijven:

```
uses strings ;
```

Het 'uses' sleutelwoord bepaalt dus welke bibliotheken dienen te worden ingevoerd.

Helperfuncties

De term *helper* komt van de OCL-specificatie, waarin twee soorten helpers gedefinieerd worden: *helperoperaties* en *-attributen*.

In ATL kan een helperfunctie enkel gedefinieerd worden met als context een OCL-type of een brontype, doordat doelmodellen niet navigeerbaar zijn. Helperfuncties kunnen gedefinieerd worden in de context van een modelement of van een module. Het hoofddoel van een helperoperatie is om de navigatie over een bronmodel gemakkelijker te maken. Helperoperaties kunnen invoerparameters hebben en kunnen recursie gebruiken.

Helperattributen worden gebruikt om alleen-lezen met elementen van het bronmodel te associëren. Gelijkaardig aan helperoperaties hebben ze een naam, een context en een type. Het verschil is dat helperattributen geen invoerparameters kunnen hebben.

Helperattributen lijken op afgeleide opties uit MOF maar kunnen geassocieerd worden met een transformatie en zijn niet altijd aan een metamodel gehecht. Met helperattributen kan dus een attribuut gedefinieerd worden dat niet werkelijk aanwezig is in een model. Een voorbeeld:

```
helper context UML!Class def :  
    allAttributes : Sequence(SimpleClass!Attribute) =  
    self.attrs->union(  
        if not self.parent.oclIsUndefined() then  
        self.parent.allAttributes->select(attr |  
            not self.attrs->exists(at | at.name = attr.name)  
        )  
        else Sequence {}  
        endif  
    )->flatten();
```

Dit voorbeeld gaat dus voor een klasse alle attributen in een verzameling teruggeven, met hierin begrepen alle attributen van de superklasse. Voor een gebruiker van deze klasse lijkt het alsof de klasse `Class` (aangeduid door het sleutelwoord *context*) van het UML-metamodel een attribuut `allAttributes` heeft, terwijl dit in werkelijkheid niet zo is.

Transformatieregels

De basisconstructie in ATL gebruikt om de transformatielogica uit te drukken is die van transformatieregel. Transformatieregels kunnen in ATL in een declaratieve of in een imperatieve stijl uitgedrukt worden. Beide stijlen worden hierna besproken.

Declaratief

Declaratieve transformatieregels worden in ATL gecorreleerde transformatieregels genoemd. Een gematchede transformatieregel bestaat uit een bron- en een doelpatroon.

Het bronpatroon specificeert een brontype (gekozen uit de unie van het bronmetamodel en de OCL-types) en een *wachter* (een booleaanse uitdrukking in OCL). Eerst geeft het brontype een aantal resultaten terug uit het bronmodel, daarna wordt hieruit geselecteerd aan de hand van de wachter.

Het doelpatroon bestaat uit een verzameling elementen. Elk van deze elementen specificeert een doeltype (uit het doelmetamodel) en een set bindingen. Een binding verwijst naar een optie van het type (attribuut, referentie of associatie-einde) en specificeert een uitdrukking waarvan de waarde gebruikt wordt om deze optie te initialiseren. Een voorbeeld:

```
rule Class2Table{  
    from  
        c : UML!Class (  
            c.is_persistent and c.parent.oclIsUndefined()  
        )
```

```

to
    t : RDBMS!Table (
        name <- c.name
    )
}

```

In dit voorbeeld wordt een klasse op een tabel afgebeeld. Het bronpatroon van deze transformatieregel definieert een instantie van `Class` uit het UML-metamodel. De wachter van het invoerpatroon bepaalt dat de klasse persistent dient te zijn en dat enkel klassen zonder superklassen geselecteerd worden voor deze afbeelding.

Het doelpatroon bestaat uit een element van het type `Table` uit het RDBMS-metamodel. Het “<-” symbool bepaalt een initialisatie voor het attribuut `name` van de tabel.

Soorten gecorreleerde transformatieregels

Er zijn verschillende soorten gecorreleerde transformatieregels, afhankelijk van de manier en het ogenblik waarop ze uitgevoerd worden.

- *Standaardregels* worden eenmaal toegepast voor iedere *match* in het bronmodel.
- *Luie regels* worden door andere regels aangeroepen. Ze worden zo vaak op een *match* uitgevoerd er door andere regels naar verwezen wordt. Dit betekent dat een luie regel meerdere malen op één enkele match kan uitgevoerd worden, waarbij deze iedere keer een andere verzameling doelelementen zou kunnen produceren. Luie regels worden met het sleutelwoord ‘lazy’ aangeduid.
- *Unieke luie regels* worden ook door andere regels aangeroepen. Deze worden echter slechts eenmaal voor elke match uitgevoerd. Als een unieke luie regel later nogmaals wordt uitgevoerd op dezelfde match, worden de reeds aangemaakte doelelementen hergebruikt als resultaat. Unieke luie regels worden met de sleutelwoorden ‘unique lazy’ aangeduid.

Overerving

In ATL kan overerving van transformatieregels gebruikt worden als hergebruikmechanisme en als mechanisme voor het specificeren van polymorfische transformatieregels. Een *subregel* kan erven van een *superregel*. Een subregel correleert altijd met een subset van waar zijn superregel mee correleert. Dit impliceert een aantal beperkingen op bronpatronen van subregels. Het bronpatroontype van de subregel moet ofwel hetzelfde zijn als dat van de superregel of een subtype ervan. De wachter van een subregel maakt altijd een selectie uit elementen die reeds gecorreleerd werden door de superregel. De eigenlijke wachter van de subregel is dus de conjunctie van beide wachtters.

Het doelpatroon van de subregel breidt het doelpatroon van de superregel uit door een combinatie van het volgende: afleiden van doeltypes, bindingen toevoegen, bindingen vervangen en nieuwe doelelementen toevoegen. Een binding kan namelijk niet uitgebreid worden, maar moet volledig vervangen worden.

Imperatief

In sommige gevallen kan het nodig zijn in een transformatie complexe algoritmes te verwerken specifiek aan het bron- of doeldomein, bijvoorbeeld het diagonaliseren van een matrix. Enkele mogelijke oplossingen zijn:

- toelaten van interfaces naar een andere programmeertaal: het nadeel hiervan is dat de volgorde van uitvoering uit de semantiek van de transformatietaal beweegt.
- een imperatief deel in de transformatietaal aanbieden. Op deze manier blijft de volgorde van uitvoering in de semantiek van de transformatietaal maar moet de programmeur deze uitvoering expliciet ontwikkelen. Er kunnen mogelijk problemen zijn met de efficiëntie en de optimalisatie van dergelijke transformatiespecificaties. De programmeur kan in dit geval echter nog steeds een manuele optimalisatie uitvoeren.
- Data aan het model ontleen, en deze met een extern hulpmiddel analyseren. Een nadeel van deze aanpak is dat het een zwaar mechanisme vereist, maar dit biedt ook een grote flexibiliteit aan voor het verwerken van de data.

Bij ATL werd gekozen voor een imperatieve aanpak. Deze is gebaseerd op twee constructies:

- aangeroepen regels: een aangeroepen regel is eigenlijk een procedure: deze wordt aangeroepen met zijn naam en kan argumenten nemen. De implementatie ervan kan in ATL gebeuren of eender welke andere taal.
- actieblok: een actieblok is een sequentie van imperatieve statements, en kan gebruikt worden in plaats van of samen met een doelpatroon in gecorreleerde of aangeroepen regels. De imperatieve constructies die beschikbaar zijn in ATL zijn de gekende constructies voor het specificeren van de volgorde van uitvoering, zoals condities, lussen, toekenningen e.d.

Ook de derde optie is in elke vorm van modeltransformatie beschikbaar, aangezien dit niet van de details van de modeltransformatie afhangt, dus ook in ATL.

Appendix C : ATL-implementatie

UMLToDelphiPasFile.atl

```
query UMLtoDelphiPasFile = UML!Class.allInstances()
    ->select( c | c.ocliIsTypeOf(UML!Class) )
    ->collect( c | c.toFileString().writeTo(c.pasFilePath()) );

--
=====
-- Hoog-niveau Helperfuncties voor codegeneratie
--
=====

-- Helperfunctie om naam van Pascalbestand uit klassenaam af te leiden
helper context UML!Class def : pasFilePath() : String =
    'delphi/opgesplitst/' + self.pasFileName();

-- Helperfunctie om naam van Pascalbestand uit klassenaam af te leiden
helper context UML!Class def : pasFileName() : String =
    self.unitName() + '.pas';

-- Helperfunctie om naam van bestand voor constantendeclaraties uit
klassenaam
-- af te leiden
helper context UML!Class def : fidFileName() : String =
    'Db' + self.name + '.fid';

-- Helperfunctie om naam van declaratiebestand uit klassenaam af te leiden
helper context UML!Class def : dclFileName() : String =
    'Db' + self.name + '.dcl';

-- Helperfunctie om naam van implementatiebestand uit klassenaam af te
leiden
helper context UML!Class def : impFileName() : String =
    'Db' + self.name + '.imp';

-- Helperfunctie om naam van Delphi-unit uit naam van UML-klasse af te
leiden
helper context UML!Class def : unitName() : String =
    'Cl' + self.name;

-- Helperfunctie om naam van Delphi-klasse uit naam van UML-klasse af te
leiden
helper context UML!Class def : className() : String =
    self.name;

--
=====
-- Hoog-niveau helperfuncties voor codegeneratie
--
=====

-- Beginpunt van codegeneratie om UML-klasse naar declaratie van Delphi-
unit
-- om te zetten
```

```

helper context UML!Class def : toFileString() : String =
    'unit ' + self.unitName() + ';\n\n' + 'interface\n\n'
    + self.importClause()
    + self.declaration()
    + self.implementation();

-- Helperfunctie om importeerclausule te genereren
helper context UML!Class def : importClause() : String =
    let importedClasses : Set(UML!Class) =
        self.associatedClasses()->union( self.superClass )->asSet()
    in
        if importedClasses->notEmpty()
        then
            'uses \n\t' + importedClasses->iterate(c; acc : String =
'' |
                acc +
                if acc = '' then c.unitName()
                else ',\n\t' + c.unitName()
                endif ) + ';\n\n'
        else ''
        endif;

-- Helperfunctie om declaratie van unit af te leiden uit UML-klasse
helper context UML!Class def : declaration() : String =
    'type\n\t' +
    self.className() + ' = ' + 'class' + self.generalizationClause() +
'\n\n'
    + self.declareConstructorsAndDestructors() + '\n'
    + self.includeFidFile()
    + self.includeDclFile()
    + '\tend;\n\n';

-- Helperfunctie om implementatie van functies en operaties van unit
-- af te leiden uit UML-klasse
helper context UML!Class def : implementation() : String =
    if self.ownedOperation->isEmpty() then ''
    else
        'implementation\n\n'
        + self.implementationImportClause()
        + self.implementConstructorsAndDestructors()
        + self.includeImpFile()
        + 'end.'
    endif;

--
=====
-- Helperfuncties voor importeerclausule
--
=====

-- Helperfunctie om declaratie van unit af te leiden uit UML-klasse
helper context UML!Class def : associatedClasses() : Sequence(UML!Class) =
    self.ownedAttribute->select( attr | not
attr.association.oclIsUndefined() )->collect( attr | attr.type );

--
=====
-- Helperfuncties voor declaratie
--
=====

```

```

-- Helperfunctie om generalizatieclausule af te leiden uit UML-
generalisatie
helper context UML!Class def : generalizationClause() : String =
  if self.generalization->notEmpty()
  then
    self.generalization->iterate(e; acc : String = '' |
      acc +
      if acc = '' then '(' + e.general.name
      else ', ' + e.general.name
      endif
    )
    + ')'
  else ''
  endif
;

-- Helperfunctie om declaratie van constructors en destructors in .pas-
bestand
-- te voegen
helper context UML!Class def : declareConstructorsAndDestructors() : String
=
  self.declarePublicConstructorsAndDestructors()
+ self.declareProtectedConstructorsAndDestructors()
+ self.declarePrivateConstructorsAndDestructors();

-- Helperfunctie om declaratie van publieke constructors en destructors in
.pas-bestand
-- te voegen
helper context UML!Class def : declarePublicConstructorsAndDestructors() :
String =
  let publicConstructorsAndDestructors : Sequence(UML!Operation) =
    self.constructorsAndDestructors()->select( operation |
operation.visibility = #public )
  in
    if publicConstructorsAndDestructors->notEmpty()
    then
      '\tpublic\n'
      + publicConstructorsAndDestructors->iterate( operation;
acc : String = '' |
        acc + '\t\t' + operation.toDeclaration() + '\n' )
    else ''
    endif;

-- Helperfunctie om declaratie van protected constructors en destructors in
.pas-bestand
-- te voegen
helper context UML!Class def : declareProtectedConstructorsAndDestructors()
: String =
  let protectedConstructorsAndDestructors : Sequence(UML!Operation) =
    self.constructorsAndDestructors()->select( operation |
operation.visibility = #protected )
  in
    if protectedConstructorsAndDestructors->notEmpty()
    then
      '\tprotected\n'
      + protectedConstructorsAndDestructors->iterate( operation;
acc : String = '' |
        acc + '\t\t' + operation.toDeclaration() + '\n' )
    else ''
    endif;

```

```

-- Helperfunctie om declaratie van private constructors en destructors in
.pas-bestand
-- te voegen
helper context UML!Class def : declarePrivateConstructorsAndDestructors() :
String =
    let privateConstructorsAndDestructors : Sequence(UML!Operation) =
        self.constructorsAndDestructors()->select( operation |
operation.visibility = #private )
    in
        if privateConstructorsAndDestructors->notEmpty()
        then
            '\tprivate\n'
            + privateConstructorsAndDestructors->iterate( operation;
acc : String = '' |
                acc + '\t\t' + operation.toDeclaration() + '\n' )
        else ''
        endif;

-- Helperfunctie om constructors en destructors te bepalen
helper context UML!Class def : constructorsAndDestructors() :
Sequence(UML!Operation) =
    self.ownedOperation->select( operation | (operation.name = 'Create'
or operation.name = 'Destroy' ) );

-- Helperfunctie om parameters van constructor of destructor in Delphi af
te leiden uit parameters
-- van methode van UML-klasse
helper context UML!Operation def : parameters() : String =
    let parameters : Sequence( UML!Parameter ) =
        self.ownedParameter->select( p | not(p.type.oclIsUndefined()) )
    in
        if parameters->isEmpty() then ''
        else
            '(' + parameters->iterate(e; acc : String = '' | acc +
                if acc = '' then '' else '; ' endif +
                e.name + ': ' + e.type.name )
            + ')'
        endif;

-- Helperfunctie om declaratie van constructor of destructor in .pas-
bestand
-- te voegen
helper context UML!Operation def : toDeclaration() : String =
    if self.name = 'Create' then 'constructor '
        + self.name
        + self.parameters()
        + ';'
    else
        if self.name = 'Destroy' then 'destructor '
            + self.name
            + self.parameters()
            + ';'
        else ''
        endif
    endif;

-- Helperfunctie om .fid-bestand in .pas-bestand te voegen via {$I }
compiler-
-- instructie
helper context UML!Class def : includeFidFile() : String =
    '{$I ' + self.fidFileName() + '}\n\n';

```

```

-- Helperfunctie om .dcl-bestand in .pas-bestand te voegen via {$I }
compiler-
-- instructie
helper context UML!Class def : includeDclFile() : String =
    '{$I ' + self.dclFileName() + '}\n\n';

--
=====
-- Helperfuncties voor implementatie
--
=====

-- Helperfunctie om benodigde geïmporteerde klassen voor implementatie van
-- Delphi-unit af te leiden uit UML-klasse
helper context UML!Class def : implementationImportClause() :
Sequence(UML!Class) =
    let importedClasses : Set(UML!Class) =
        self.associatedClasses()->asSet()
    in
        if importedClasses->notEmpty()
        then
            'uses \n\t' + importedClasses->iterate(c; acc : String =
'' |
                acc +
                if acc = '' then c.unitName()
                else ',\n\t' + c.unitName()
                endif ) + ';\n\n'
        else ''
        endif;

-- Helperfunctie om implementatie van constructors en destructors in .pas-
bestand
-- te voegen
helper context UML!Class def : implementConstructorsAndDestructors() :
String =
    self.constructorsAndDestructors()
        ->iterate( operation; acc : String = '' | acc +
operation.toImplementation() );

-- Helperfunctie om implementatie van constructor of destructor in .pas-
bestand
-- te voegen
helper context UML!Operation def : toImplementation() : String =
    if self.name = 'Create' then 'constructor '
        + self.class_.className() + '.' + self.name + self.parameters()
        + ';\n'
        + 'begin\n'
        + 'end;\n\n'
    else
        if self.name = 'Destroy' then 'destructor '
        + self.class_.className() + '.' + self.name + self.parameters()
        + ';\n'
        + 'begin\n'
        + 'end;\n\n'
        else ''
        endif
    endif;

-- Helperfunctie om .imp-bestand in .pas-bestand te voegen via {$I }
compiler-

```

```
-- instructie
helper context UML!Class def : includeImpFile() : String =
    '{$I ' + self.impFileName() + '}\n\n';
```

UMLToDelphiDclFile.atl

```
query UMLtoDelphiDcl = UML!Class.allInstances()
    ->select( c | c.oclIsTypeOf(UML!Class) )
    ->collect( c | c.toFileString().writeTo(c.dclFilePath()) );
```

```
--
=====
-- Hoog-niveau Helperfuncties voor codegeneratie
--
=====
```

```
-- Helperfunctie om naam van Pascalbestand uit klassenaam af te leiden
helper context UML!Class def : dclFilePath() : String =
    'delphi/opgesplitst/' + self.dclFileName();
```

```
-- Helperfunctie om naam van declaratiebestand uit klassenaam af te leiden
helper context UML!Class def : dclFileName() : String =
    'Db' + self.name + '.dcl';
```

```
--
=====
-- Hoog-niveau helperfuncties voor codegeneratie
--
=====
```

```
-- Beginpunt van codegeneratie om UML-klassen naar Delphi om te zetten
helper context UML!Class def : toFileString() : String =
    self.declareFunctionsAndProcedures();
```

```
--
=====
-- Helperfuncties voor declaratie
--
=====
```

```
-- Beginpunt van codegeneratie om UML-klassen naar Delphi om te zetten
helper context UML!Class def : declareFunctionsAndProcedures() : String =
    self.declarePublicFunctionsAndProcedures()
    + self.declareProtectedFunctionsAndProcedures()
    + self.declarePrivateFunctionsAndProcedures();
```

```
-- Helperfunctie om declaratie van publieke functies en operaties
-- in Delphi af te leiden uit methodes van UML-klasse
helper context UML!Class def : declarePublicFunctionsAndProcedures() :
String =
```

```
    let publicOperations : Set(UML!Operation) =
        self.ownedOperation->select( operation |
            not( operation.name = 'Create' or operation.name =
'Destroy' )
            and operation.visibility = #public )
        ->asSet()

    in
        if publicOperations->isEmpty() then ''
        else
            'public\n' +
```

```

        publicOperations->iterate( f; acc : String = '' | acc +
'\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van protected functies en operaties
-- in Delphi af te leiden uit methodes van UML-klasse
helper context UML!Class def : declareProtectedFunctionsAndProcedures() :
String =
    let protectedOperations : Set(UML!Operation) =
        self.ownedOperation->select( operation |
            not( operation.name = 'Create' or operation.name =
'Destroy' )
            and operation.visibility = #protected )
        ->asSet()

    in
        if protectedOperations->isEmpty() then ''
        else
            'protected\n' +
                protectedOperations->iterate( f; acc : String = '' | acc +
'\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van private functies en operaties
-- in Delphi af te leiden uit methodes van UML-klasse
helper context UML!Class def : declarePrivateFunctionsAndProcedures() :
String =
    let privateOperations : Set(UML!Operation) =
        self.ownedOperation->select( operation |
            not( operation.name = 'Create' or operation.name =
'Destroy' )
            and operation.visibility = #private )
        ->asSet()

    in
        if privateOperations->isEmpty() then ''
        else
            'private\n' +
                privateOperations->iterate( f; acc : String = '' | acc +
'\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van operatie of functie in Delphi af te
leiden uit
-- methode van UML-klasse
helper context UML!Operation def : toDeclaration() : String =
    if self.isQuery = true then 'function '
    else 'procedure '
    endif
    + self.name
    + self.parameters()
    + self.returnParameter()
    + ';;';

-- Helperfunctie om terugkeerparameter van functie in Delphi af te leiden
uit parameters
-- van methode van UML-klasse
helper context UML!Operation def : returnParameter() : String =
    if self.type.oclIsUndefined() then ''
    else ':' + self.type.name
    endif;

```

```

-- Helperfunctie om parameters, uitgezonderd terugkeerparameter van functie
of operatie
-- in Delphi af te leiden uit parameters van methode van UML-klasse
helper context UML!Operation def : parameters() : String =
  let parameters : Sequence( UML!Parameter ) =
    self.ownedParameter->select( p | not(p.type.oclIsUndefined()) )
  in
    if parameters->isEmpty() then ''
    else
      '(' + parameters->iterate(e; acc : String = '' | acc +
        if acc = '' then '' else '; ' endif +
        e.name + ': ' + e.type.name )
      + ')'
    endif;

```

UMLToDelphiImpFile.atl

```
query UMLtoDelphiImp = UML!Class.allInstances()
    ->select( c | c.oclcIsTypeOf(UML!Class) )
    ->collect( c | c.toFileString().writeTo(c.impFileName()) );

--
=====
-- Hoog-niveau Helperfuncties voor codegeneratie
--
=====

-- Helperfunctie om naam van Pascalbestand uit klassenaam af te leiden
helper context UML!Class def : impFileName() : String =
    'delphi/opgesplitst/Db' + self.name + '.imp';

-- Helperfunctie om naam Delphi-klasse uit naam van UML-klasse af te leiden
helper context UML!Class def : className() : String =
    self.name;

--
=====
-- Hoog-niveau helperfuncties voor codegeneratie
--
=====

-- Beginpunt van codegeneratie om UML-klassen naar Delphi om te zetten
helper context UML!Class def : toFileString() : String =
    self.implementation();

-- Helperfunctie om implementatie van functies en operaties van unit
-- af te leiden uit UML-klasse
helper context UML!Class def : implementation() : String =
    if self.ownedOperation->isEmpty() then ''
    else
        self.ownedOperation
        ->select( operation |
            not( operation.name = 'Create' or operation.name =
'Destroy' ) )
        ->iterate( operation; acc : String = '' |
            acc + operation.toImplementation() )
    endif;

--
=====
-- Helperfuncties voor implementatie
--
=====

-- Helperfunctie om implementatie van operatie of functie in Delphi af te
leiden uit
-- methode van UML-klasse
helper context UML!Operation def : toImplementation() : String =
    if self.isQuery = true then 'function '
    else 'procedure '
    endif
    + self.class_.className() + '.' + self.name + self.parameters()
    + self.returnParameter() + ';\n' +
    'begin\n' +
    if self.name->startsWith('get')
    then
```

```

        '\tResult := self.' +
self.getAttributeNameFromInspectorOrMutator() + ';\n'
    else
        if self.name->startsWith('set')
        then
            '\tself.' + self.getAttributeNameFromInspectorOrMutator()
+ ' := '
            + self.getParameterNameFromMutator() + ';\n'
        else
            ''
        endif
    endif
+ 'end;\n\n';

-- Helperfunctie om terugkeerparameter van functie in Delphi af te leiden
uit parameters
-- van methode van UML-klasse
helper context UML!Operation def : returnParameter() : String =
    if self.type.oclIsUndefined() then ''
    else ': ' + self.type.name
    endif;

-- Helperfunctie om parameters, uitgezonderd terugkeerparameter van functie
of operatie
-- in Delphi af te leiden uit parameters van methode van UML-klasse
helper context UML!Operation def : parameters() : String =
    let parameters : Sequence( UML!Parameter ) =
        self.ownedParameter->select( p | not(p.type.oclIsUndefined()) )
    in
        if parameters->isEmpty() then ''
        else
            '(' + parameters->iterate(e; acc : String = '' | acc +
                if acc = '' then '' else '; ' endif +
                e.name + ': ' + e.type.name )
            + ')'
        endif;

-- Helperfunctie om naam van attribuut uit naam van inspector af te leiden
helper context UML!Operation def : getAttributeNameFromInspectorOrMutator()
: String =
    let operationName : String =
        self.name
    in
        operationName.substring( 4, operationName.size()
).firstToLower();

-- Helperfunctie om naam van parameter uit mutator af te leiden
helper context UML!Operation def : getParameterNameFromMutator() : String =
    let parameters : Sequence( UML!Parameter ) =
        self.ownedParameter
    in
        if parameters->isEmpty() then ''
        else
            parameters->iterate(p; acc : String = '' | acc +
                if acc = '' then p.name else '' endif )
        endif;

-- Helper functie om de eerste letter van een String naar een kleine letter
om te zetten.
helper context String def: firstToLower() : String =

```

```
self.substring(1, 1).toLowerCase() + self.substring(2, self.size());
```

UMLToDelphiFidFile.atl

```
query UMLtoDelphiFid = UML!Class.allInstances()
    ->select( c | c.ocIsTypeOf(UML!Class) )
    ->collect( c | c.toFileString().writeTo(c.fidFileName()) );

--
=====
-- Hoog-niveau Helperfuncties voor codegeneratie
--
=====

-- Helperfunctie om naam van bestand voor constantendeclaraties uit
klassenaam
-- af te leiden
helper context UML!Class def : fidFileName() : String =
    'delphi/opgesplitst/Db' + self.name + '.fid';

--
=====
-- Hoog-niveau helperfuncties voor codegeneratie
--
=====

-- Beginpunt van codegeneratie om UML-klassen naar Delphi om te zetten
helper context UML!Class def : toFileString() : String =
    self.declareAttributes();

-- Helperfunctie om declaratie van unit af te leiden uit UML-klasse
helper context UML!Class def : declareAttributes() : String =
    self.declarePublicAttributes()
    + self.declareProtectedAttributes()
    + self.declarePrivateAttributes();

--
=====
-- Helperfuncties voor declaratie
--
=====

-- Helperfunctie om declaratie van publieke variabelen, functies en
operaties
-- in Delphi af te leiden uit attributen en methodes van UML-klasse
helper context UML!Class def : declarePublicAttributes(): String =
    let publicAttributes : Set(UML!Property) =
        self.ownedAttribute->select( attr | attr.visibility = #public
    )->asSet()
    in
        if publicAttributes->isEmpty() then ''
        else
            'public\n' +
                publicAttributes->iterate( f; acc : String = '' | acc +
'\t' + f.toDeclaration() + '\n' )
            endif;

-- Helperfunctie om declaratie van protected variabelen, functies en
operaties
-- in Delphi af te leiden uit attributen en methodes van UML-klasse
```

```

helper context UML!Class def : declareProtectedAttributes() : String =
    let protectedAttributes : Set(UML!Property) =
        self.ownedAttribute->select( attr | attr.visibility =
#protected )->asSet()
    in
        if protectedAttributes->isEmpty() then ''
        else
            'protected\n' +
                protectedAttributes->iterate( f; acc : String = '' | acc +
'\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van private variabelen, functies en
operaties
-- in Delphi af te leiden uit attributen en methodes van UML-klasse
helper context UML!Class def : declarePrivateAttributes() : String =
    let privateAttributes : Set(UML!Property) =
        self.ownedAttribute->select( attr | attr.visibility = #private
)->asSet()
    in
        if privateAttributes->isEmpty() then ''
        else
            'private\n' +
                privateAttributes->iterate( f; acc : String = '' | acc +
'\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van variabele in Delphi af te leiden uit
-- attribuut van UML-klasse
helper context UML!Property def : toDeclaration() : String =
    self.name + ': ' + self.type.name + ';';

```

UMLToDelphi.atl

```
query UMLtoDelphi = UML!Class.allInstances()
    ->select( c | c.oclcIsTypeOf(UML!Class) )
    ->collect( c | c.toFileString().writeTo(c.fileName()) );

--
=====
-- Helperfuncties voor bestands- en padnaam
--
=====

-- Helperfunctie om bestandsnaam voor Delphi unit van klassenaam af te
leiden
helper context UML!Class def : fileName() : String =
    'delphi/samengevoegd/' + self.unitName() + '.unit';

-- Helperfunctie om naam van Delphi-unit uit naam van UML-klasse af te
leiden
helper context UML!Class def : unitName() : String =
    'Cl' + self.name;

-- Helperfunctie om naam van Delphi-klasse uit naam van UML-klasse af te
leiden
helper context UML!Class def : className() : String =
--     'Cls_' + self.name;
    self.name;

--
=====
-- Hoog-niveau helperfuncties voor codegeneratie
--
=====

-- Beginpunt van codegeneratie om UML-klassen naar Delphi om te zetten
helper context UML!Class def : toFileString() : String =
    'unit ' + self.unitName()      + ';\n\n' + 'interface\n\n'
    + self.importClause()
    + self.declaration()
    + self.implementation();

-- Helperfunctie om importeerclausule te genereren
helper context UML!Class def : importClause() : String =
    let importedClasses : Set(UML!Class) =
        self.associatedClasses()->union( self.superClass )->asSet()
    in
        if importedClasses->notEmpty()
        then
            'uses \n\t' + importedClasses->iterate(c; acc : String =
'' |
                acc +
                if acc = '' then c.unitName()
                else ',\n\t' + c.unitName()
                endif) + ';\n\n'
        else ''
        endif;

-- Helperfunctie om declaratie van unit af te leiden uit UML-klasse
helper context UML!Class def : declaration() : String =
    'type\n\t' +
    self.name + ' = ' + 'class' + self.generalizationClause() + '\n'
```

```

+ self.declarePublicFeatures()
+ self.declareProtectedFeatures()
+ self.declarePrivateFeatures()
+ '\tend;\n\n';

-- Helperfunctie om implementatie van functies en operaties van unit
-- af te leiden uit UML-klasse
helper context UML!Class def : implementation() : String =
    if self.ownedOperation->isEmpty() then ''
    else
        'implementation\n\n'
        + self.implementationImportClause()
        + self.ownedOperation->iterate( operation; acc : String = '' |
            acc + operation.toImplementation() )
        + 'end.'
    endif;

--
=====
-- Helperfuncties voor importeerclausule
--
=====

-- Helperfunctie om declaratie van unit af te leiden uit UML-klasse
helper context UML!Class def : associatedClasses() : Sequence(UML!Class) =
    self.ownedAttribute->select( attr | not
    attr.association.oclIsUndefined() )->collect( attr | attr.type );

--
=====
-- Helperfuncties voor declaratie
--
=====

-- Helperfunctie om generalizatieclausule af te leiden uit UML-
generalisatie
helper context UML!Class def : generalizationClause() : String =
    if self.generalization->notEmpty()
    then
        '('
        + self.generalization->iterate(e; acc : String = '' |
            acc +
                if acc = '' then e.general.name
                else ', ' + e.general.name
                endif
        )
        + ')'
    else ''
    endif
    ;

-- Helperfunctie om declaratie van publieke variabelen, functies en
operaties
-- in Delphi af te leiden uit attributen en methodes van UML-klasse
helper context UML!Class def : declarePublicFeatures() : String =
    let publicFeatures : Set(UML!Feature) =
        self.feature->select( feature | feature.visibility = #public )-
>asSet()
    in
        if publicFeatures->isEmpty() then ''
        else

```

```

        '\t\tpublic\n' +
        publicFeatures->iterate( f; acc : String = '' | acc +
'\t\t\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van protected variabelen, functies en
operaties
-- in Delphi af te leiden uit attributen en methodes van UML-klasse
helper context UML!Class def : declareProtectedFeatures() : String =
    let protectedFeatures : Set(UML!Feature) =
        self.feature->select( feature | feature.visibility = #protected
)->asSet()
    in
        if protectedFeatures->isEmpty() then ''
        else
            '\t\tprotected\n' +
            protectedFeatures->iterate( f; acc : String = '' | acc +
'\t\t\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van private variabelen, functies en
operaties
-- in Delphi af te leiden uit attributen en methodes van UML-klasse
helper context UML!Class def : declarePrivateFeatures() : String =
    let privateFeatures : Set(UML!Feature) =
        self.feature->select( feature | feature.visibility = #private
)->asSet()
    in
        if privateFeatures->isEmpty() then ''
        else
            '\t\tprivate\n' +
            privateFeatures->iterate( f; acc : String = '' | acc +
'\t\t\t' + f.toDeclaration() + '\n' )
        endif;

-- Helperfunctie om declaratie van variabele in Delphi af te leiden uit
-- attribuut van UML-klasse
helper context UML!Property def : toDeclaration() : String =
    self.name + ': ' + self.type.name + ';';

-- Helperfunctie om declaratie van operatie of functie in Delphi af te
leiden uit
-- methode van UML-klasse
helper context UML!Operation def : toDeclaration() : String =
    if self.name = 'Create' then 'constructor '
    else
        if self.name = 'Destroy' then 'destructor '
        else
            if self.isQuery = true then 'function '
            else 'procedure '
            endif
        endif
    endif
    + self.name
    + self.parameters()
    + self.returnParameter()
    + ';';

--
=====
-- Helperfuncties voor implementatie

```

```

--
=====

-- Helperfunctie om benodigde geïmporteerde klassen voor implementatie van
-- Delphi-unit af te leiden uit UML-klasse
helper context UML!Class def : implementationImportClause() :
Sequence(UML!Class) =
    let importedClasses : Set(UML!Class) =
        self.associatedClasses()->asSet()
    in
        if importedClasses->notEmpty()
        then
            'uses \n\t' + importedClasses->iterate(c; acc : String =
'' |
                acc +
                if acc = '' then c.unitName()
                else ', ' + c.unitName()
                endif ) + '\n\n'
        else ''
        endif;

-- Helperfunctie om implementatie van operatie of functie in Delphi af te
leiden uit
-- methode van UML-klasse
helper context UML!Operation def : toImplementation() : String =
    if self.name = 'Create' then 'constructor '
    else
        if self.name = 'Destroy' then 'destructor '
        else
            if self.isQuery = true then 'function '
            else 'procedure '
            endif
        endif
    endif
    + self.class_.className() + '.' + self.name + self.parameters()
    + self.returnParameter() + ';\n' +
    'begin\n' +
    if self.name->startsWith('get')
    then
        '\tResult := self.' +
self.getAttributeNameFromInspectorOrMutator() + ';\n'
    else
        if self.name->startsWith('set')
        then
            '\tself.' + self.getAttributeNameFromInspectorOrMutator()
+ ' := '
            + self.getParameterNameFromMutator() + ';\n'
        else
            ''
        endif
    endif
    + 'end;\n\n';

-- Helperfunctie om terugkeerparameter van functie in Delphi af te leiden
uit parameters
-- van methode van UML-klasse
helper context UML!Operation def : returnParameter() : String =
    if self.type.oclIsUndefined() then ''
    else ':' + self.type.name
    endif;

```

```

-- Helperfunctie om parameters, uitgezonderd terugkeerparameter van functie
of operatie
-- in Delphi af te leiden uit parameters van methode van UML-klasse
helper context UML!Operation def : parameters() : String =
    let parameters : Sequence( UML!Parameter ) =
        self.ownedParameter->select( p | not(p.type.ocliIsUndefined()) )
    in
        if parameters->isEmpty() then ''
        else
            '(' + parameters->iterate(e; acc : String = '' | acc +
                if acc = '' then '' else '; ' endif +
                e.name + ': ' + e.type.name )
            + ')'
        endif;

-- Helperfunctie om naam van attribuut uit naam van inspector af te leiden
helper context UML!Operation def : getAttributeNameFromInspectorOrMutator()
: String =
    let operationName : String =
        self.name
    in
        operationName.substring( 4, operationName.size()
).firstToLower();

-- Helperfunctie om naam van parameter uit mutator af te leiden
helper context UML!Operation def : getParameterNameFromMutator() : String =
    let parameters : Sequence( UML!Parameter ) =
        self.ownedParameter
    in
        if parameters->isEmpty() then ''
        else
            parameters->iterate(p; acc : String = '' | acc +
                if acc = '' then p.name else '' endif )
        endif;

-- Helper functie om de eerste letter van een String naar een kleine letter
om te zetten.
helper context String def: firstToLower() : String =
    self.substring(1, 1).toLowerCase() + self.substring(2, self.size());

```

Appendix D: Code gegenereerd door ATL

ClDeur.pas

```
unit ClDeur;

interface

uses
    ClRaam,
    ClBeweegbaarElement;

type
    Deur = class(BeweegbaarElement)

    public
        constructor Create;
        destructor Destroy;

    {$I DbDeur.fid}

    {$I DbDeur.dcl}

    end;

implementation

uses
    ClRaam;

constructor Deur.Create;
begin
end;

destructor Deur.Destroy;
begin
end;

{$I DbDeur.imp}

end.
```

DbDeur.dcl

```
public
    function getIsLocked: Boolean;
    procedure setIsLocked(isLocked: Boolean);
    function getRaam: Raam;
```

DbDeur.imp

```
function Deur.getIsLocked: Boolean;
begin
    Result := self.isLocked;
end;

function Deur.getRaam: Raam;
begin
    Result := self.raam;
end;

procedure Deur.setIsLocked(isLocked: Boolean);
begin
    self.isLocked := isLocked;
end;
```

DbDeur.fid

```
private
    raam: Raam;
    isLocked: Boolean;
```

ClDeur.unit

```
unit ClDeur;

interface

uses
    ClBeweegbaarElement,
    ClRaam;

type
    Deur = class(BeweegbaarElement)
    public
        function getIsLocked: Boolean;
        destructor Destroy;
        constructor Create;
        procedure setIsLocked(isLocked: Boolean);
        function getRaam: Raam;
    private
        isLocked: Boolean;
        raam: Raam;
    end;

implementation

uses
    ClRaam;

function Deur.getIsLocked: Boolean;
begin
    Result := self.isLocked;
end;

function Deur.getRaam: Raam;
begin
    Result := self.raam;
end;

constructor Deur.Create;
begin
end;

destructor Deur.Destroy;
begin
end;

procedure Deur.setIsLocked(isLocked: Boolean);
begin
    self.isLocked := isLocked;
end;

end.
```

Appendix E: MTF syntax

Relatie

MTF maakt slechts in beperkte mate notie maakt van relaties. MTF kan als een implementatie beschouwd worden van de relatietaal uit QVT v2.0. Er zijn echter enkele verschillen tussen beide specificaties te merken. Het concept dat in QVT 2.0 relatie heet, heet in MTF een afbeelding (zie 0). Dit is een gevolg van het oorspronkelijke opzet van MTF als *proof of concept* van QVT 1.0, en niet van QVT 2.0. In QVT v1.0 werd namelijk de formulering van het verband tussen twee of meer domeinen een relatie genoemd. Dit komt min of meer overeen met wat we in Java de “hoofding” van een methode noemen. Wat we analoog in Java het “lichaam” van een methode noemen, heette in QVT 1.0 een afbeelding, dus de “implementatie” van een relatie. Dit is ook in MTF zo. In QVT 1.0 en MTF is het concept van relatie dus vrij zwak: het bestaat enkel uit de specificatie van een afbeelding, met eventueel bijhorende beperking(en) voor patrooncorrelatie.

Afbeelding

Een afbeeldingregel bestaat in MTF uit een verzameling beperkingen die moeten gecontroleerd worden voor één of meer objecten. Als deze beperkingen één voor één gelden, voldoen de objecten aan de transformatieregel en moeten deze niet gewijzigd worden. Afbeeldingregels worden gecontroleerd binnen een afbeeldingssessie. Deze sessie volgt op welke regels voor welke objecten zouden moeten gelden.

Het controleren van een afbeeldingregel kan als gevolg hebben dat ook voor waarden afgeleid van objecten die afgebeeld worden, regels moeten gecontroleerd worden. Als dit voorkomt zijn deze nieuwe afbeeldingen deel van de oorspronkelijke afbeelding, en wordt op deze manier een hiërarchie van afbeeldingen gecreëerd. Het is belangrijk hierbij op te merken dat binnen deze hiërarchie nooit twee afbeeldingen van hetzelfde type dezelfde objecten kunnen afbeelden. Verder is het handig te weten dat dit afbeeldingmechanisme door zijn declaratieve aard volledig bidirectioneel is.

Zoekopdracht

Voor het implementeren van zoekopdrachten wordt in MTF het mechanisme voor patrooncorrelatie gebruikt. Indien een gebruiker dus een waarde wil opvragen, kan dat door een ongebonden variabele als parameter voor een relatie of beperking mee te geven.

Conditie

In MTF kunnen we twee mechanismen voor het formuleren van condities voor patrooncorrelatie onderscheiden: gelijkheidscondities en filtercondities. Deze worden in wat volgt elk apart toegelicht.

Gelijkheidscondities

Gelijkheidscondities zijn, eenvoudig geformuleerd, condities waar het sleutelwoord `equals` in voorkomt. In het formuleren van het lichaam van een afbeelding wordt dit simpelweg als onderdeel van de afbeelding beschouwd. Een gelijkheidsconditie in de hoofding van een afbeeldingsregel wordt op een andere manier behandeld. Tenzij de gelijkheidsconditie van de regel als waar wordt beoordeeld, wordt de afbeeldingregel niet uitgevoerd. Dit wordt, in overeenstemming met het QVT-voorstel als een `when`-clause geformuleerd. Een voorbeeld hiervan kunt u hieronder terugvinden.

```
relate umlclass2ecore( uml:Class class, ecore:EClass eclass)
```

```
when equals(class.name,eclass.name)
```

Filtercondities

Als een alternatief kan een conditie ook geformuleerd worden op een parameter van de afbeeldingregel. Deze conditie volgt op de declaratie van de parameter. Domeinelementen die niet voldoen aan de filterconditie worden niet beschouwd als kandidaten voor de parameter. De filterconditie gedraagt zich dus als een verfijning van het type van de parameter. Hieronder volgt een voorbeeld.

```
relate umlclass2ecore  
  ( uml:Class class when equals(class.isAbstract, "false"),  
    ecore:EClass eclass)
```

Deze filterconditie formuleert dat de afbeelding van een UML-klasse naar een Ecore-klasse (zie ook 6.3.1) enkel mag toegepast worden op niet-abstracte klassen uit het UML-model. Indien hieraan door geen enkel object uit het UML-model voldaan wordt, wordt de afbeelding niet uitgevoerd.

Samengestelde condities

Om condities samen te stellen, beschikt MTF over compositieoperators. Hiervoor worden de volgende symbolen gebruikt:

| | | |
|---|------|-------|
| & | EN | (AND) |
| | OF | (OR) |
| ! | NIET | (NOT) |

Om de voorgaande conditie aan te passen voor het filteren naar abstracte klassen met publieke toegankelijkheid, kunnen we zo de volgende conditie formuleren:

```
equals(class.isAbstract, "false") & equals(class.visibility,  
  "public")
```

Uitbreiding

Om hergebruik te bevorderen, kunnen afbeeldingregels andere regels uitbreiden. Dit gebeurt min of meer op dezelfde manier als we dit uit objectoriëntatie kennen. Een afbeeldingregel kan nul, één of meer afleidingsregels uitbreiden. Hiervoor wordt het sleutelwoord `extends` gebruikt. Een voorbeeld:

```
relate umlclassifier2ecore( uml:Classifier c, ecore:EClassifier ec)  
  when equals(c.name,ec.name)  
  
relate umlclass2ecore extends umlclassifier2ecore( uml:Class class,  
  ecore:EClass eclass)
```

Hierbij gelden een aantal beperkingen. Een subrelatie moet namelijk hetzelfde aantal parameters hebben, gespecificeerd in dezelfde volgorde, als al zijn superrelaties. Gelijkheidscondities worden geërfd van de superklasse, en het lichaam van de afbeeldingregel wordt geërfd van superrelaties. Dit betekent dat bij de uitvoering van een relatie ook de superrelaties uitgevoerd worden. Daarnaast dienen programmeurs ervoor te zorgen dat alle afbeeldingregels die van een bepaalde afbeeldingregel overerven, onderling uitsluitend (*mutually exclusive*) zijn. Waarom dit precies zo is, zou ons op dit moment te ver leiden.

Een programmeur die dit uitbreidingsmechanisme gebruikt, moet verder ook weten dat om polymorfisme te ondersteunen, altijd de meest specifieke afbeeldingregel mogelijk uitgevoerd wordt.

Abstracte regels

In MTF is het mogelijk een basisrelatie te creëren met verschillende gespecialiseerde subrelaties, zonder dat ooit afbeeldingen van de basisrelatie uitgevoerd worden. In Java heet dit mechanisme een overervinghiërarchie met abstracte methodes, in MTF heet dit mechanisme analoog *abstracte regels*. We kunnen een relatie als abstract markeren door voor de (basis)relatie het sleutelwoord `abstract` te plaatsen.

Opspoorbaarheid

Voor het ondersteunen van opspoorbaarheid van de uitvoering van transformaties in MTF, biedt MTF een `TraceMonitor` interface. Objecten die deze interface implementeren, kunnen op de hoogte gebracht worden als significante stappen van de transformatie beginnen en eindigen. Het spreekt voor zich dat dit mechanisme enkel kan gebruikt worden indien MTF als Java-module gebruikt wordt, en dus niet als *Eclipse plugin*.

Reconciliatie

Als een transformatie reeds uitgevoerd werd en een model op een ander afgebeeld werd, kan de transformatie daarna aangepast worden waardoor invoer en uitvoer van de transformatie niet meer met elkaar correleren. Het proces van het aanpassen van de uitvoer van een transformatie, opdat invoer en uitvoer opnieuw zouden overeenkomen, heet *reconciliatie*.

Appendix F: MTF-implementatie

```
import uml http://www.eclipse.org/uml2/1.0.0/UML
import uml http://com.ibm/mtf/uml.ecore
import.ecore http://www.eclipse.org/emf/2002/Ecore
import emf http://com.ibm/mtf/model/emf.ecore
import ws http://com.ibm/mtf/model/workspace.ecore
import util http://com.ibm/mtf/util.ecore

relate uml2uml(ws:IFile file1, ws:IFile file2)
{
    umlmodel2model (over file1.model, over file2.model)
}

relate umlmodel2model( uml:Model model, uml:Model outModel )
{
    equals( model.name, outModel.name ),
    umlpkg2pkg( over model.ownedMember, over outModel.ownedMember )
}

relate umlpkg2pkg( uml:Package pkg when !(util:InstanceOf "uml:Profile"
(pkg)), uml:Package outPkg when !(util:InstanceOf "uml:Profile" (outPkg)) )
{
    equals( pkg.name, outPkg.name ),
    // kinderen afbeelden
    ordered umlclass2class( over pkg.ownedMember, over outPkg.ownedMember
),
    ordered umlprofile2profile( over pkg.ownedMember, over
outPkg.ownedMember ),
    ordered umlstereotype2stereotype( over pkg.ownedMember, over
outPkg.ownedMember ),
    // subpackages afbeelden
    umlpkg2pkg( over pkg.ownedMember, over outPkg.ownedMember )
}

// Stereotype is een subklasse van Class, wat betekent dat ook instanties
van Stereotype door deze transformatieregel worden getransformeerd, maar
dit is gedrag dat we niet willen.

relate umlclass2class( uml:Class class when !(util:InstanceOf
"uml:Stereotype" (class)), uml:Class outClass )
{
    equals( class.name, outClass.name ),
    // Attributen transformereren
    ordered umlproperty2property( over class.ownedAttribute, over
outClass.ownedAttribute ),
    umlproperty2inspector( over class.ownedAttribute, match over
outClass.ownedOperation ),
    umlproperty2mutator( over class.ownedAttribute, match over
outClass.ownedOperation ),
    // Operaties van klasse kopiëren
    umloperation2operation( over class.ownedOperation, match over
outClass.ownedOperation ),
    // Constructor toevoegen
    umlclass2constructor( over class, match over outClass.ownedOperation
),
    // Generalizations transformereren
```

```

        ordered umlgeneralization2generalization( over class.generalization,
over outClass.generalization ),
        // Kindklassen transformeren
        ordered umlclass2class( over class.nestedClassifier, over
outClass.nestedClassifier )
    }

relate umlproperty2property( uml:Property prop, uml:Property outProp )
{
    equals( prop.name, outProp.name ),
    equals( prop.class_, outProp.class_ ),
    equals( prop.type, outProp.type ),
    equals( outProp.visibility, "private" )
}

relate umlproperty2inspector( uml:Property prop, uml:Operation operation )
when util:MatchString "get_{0}" ( prop.name, operation.name )
{
    // Terugkeertype van operatie op type van attribuut zetten
    setoperationreturntype [1] ( prop, match over
operation.ownedParameter )
}

relate setoperationreturntype( uml:Property prop, uml:Parameter parameter
when equals( parameter.direction, "return" ) )
{
    equals( prop.type, parameter.type )
}

relate umlproperty2mutator( uml:Property prop, uml:Operation operation )
when util:MatchString "set_{0}" ( prop.name, operation.name )
{
    ordered umlattribute2parameter( over prop, match over
operation.ownedParameter )
}

relate umlattribute2parameter( uml:Property prop, uml:Parameter parameter )
{
    equals( prop.name, parameter.name ),
    // Type van parameter van moderator op type van attribuut zetten
    equals( prop.type, parameter.type )
}

relate umloperation2operation( uml:Operation operation, uml:Operation
outOperation ) when equals( operation.name, outOperation.name )
{
    ordered umlparameter2parameter( over operation.ownedParameter, over
outOperation.ownedParameter )
}

relate umlparameter2parameter( uml:Parameter parameter, uml:Parameter
outParameter )
{
    equals( parameter.name, outParameter.name ),
    equals( parameter.type, outParameter.type ),
    equals( parameter.direction, outParameter.direction )
}

relate umlclass2constructor( uml:Class class, uml:Operation operation )
when equals( class.name, operation.name )

```

```
relate umlgeneralization2generalization( uml:Generalization
generalization,uml:Generalization outGeneralization ) when ref
umlclass2class( generalization.general,outGeneralization.general )

relate umlprofile2profile( uml:Profile profile, uml:Profile outProfile )
{
    equals( profile.name, outProfile.name )
}

relate umlstereotype2stereotype( uml:Stereotype stereotype, uml:Stereotype
outStereotype )
{
    equals( stereotype.name, outStereotype.name )
}
```

Appendix G: Verklarende Woordenlijst

| Term | Verklaring |
|---|--|
| Model | Een model is een representatie van een deel van de functie, structuur en/of het gedrag van een applicatie of systeem. Een representatie van een model is formeel als het gebaseerd is op een taal met een vastgelegde vorm (“syntax”), betekenis (semantiek), en mogelijk andere beperkingen. De syntax kan grafisch of tekstueel zijn. In MDA is een niet-formele representatie geen model. |
| Platform | Een verzameling subsystemen of technologieën die een gerelateerde functionaliteit aanbieden via interfaces en gebruikspatronen, die elk subsysteem die van het platform afhangt kan gebruiken zonder kennis te hebben van de implementatiedetails. |
| PIM (<i>Platform Independent Model</i>) | Een model van een subsysteem dat geen informatie bevat die specifiek is aan het platform of aan de technologie die gebruikt wordt om dit te realiseren. |
| PSM (<i>Platform Specific Model</i>) | Een model van een subsysteem dat informatie bevat over de specifieke technologie die gebruikt wordt om het subsysteem op een specifiek platform te realiseren, en dus ook elementen bevat die specifiek zijn aan het platform. |
| Afbeelding | Specificatie van een mechanisme voor het transformeren van modelementen die aan een bepaald metamodel voldoen naar elementen van een ander model dat aan een ander (mogelijk hetzelfde) metamodel voldoet. Een afbeelding kan uitgedrukt worden met behulp van associaties, beperkingen, regels, generische mechanismen etc. |
| Relatie | Specificatie voor het aflijnen van welke modelementen naar elementen van een ander model getransformeerd worden. |

Appendix H: Referenties

- [AKJWWB03] Anneke Kleppe, Jos Warmer, Wim Bast: MDA Explained, 2003
- [FB87] Fred Brooks: Essence and Accidents of Software Engineering, 1987
- [formal/2006-01-01] Object Management Group: MOF Core Specification, v2.0, 2006
- [formal/05-07-04] Object Management Group: Unified Modeling Language: Superstructure version 2.0, 2005
- [formal/03-09-15] Object Management Group: Unified Modeling Language (UML) Specification: Infrastructure version 2.0, 2003
- [ptc/03-10-14] Object Management Group: UML 2.0 OCL Specification, 2003
- [formal/05-09-01] Object Management Group: MOF 2.0/XMI Mapping Specification, v2.1, 2005
- [W3C04] W3C: Extensible Markup Language (XML) 1.1, 2004
- [ad/2001-02-01] Object Management Group: Common Warehouse Metamodel (CWM) Specification, 2001
- [CGL97] Marco Cantu, Tim Gooch, John F. Lam: Delphi Developers Handbook, 1997
- [ad/2002-04-10] Object Management Group: Request for Proposal: MOF 2.0 Query/View/Transformation RFP, 2002
- [GGKH] Tracy Gardner, Catherine Griffin, Jana Koehler, Rainer Hauser : A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, 2003
- [KC02] Kennedy Carter Ltd.: Initial Submission to OMG RFP ad/2002-12-10 MOF Query, Views and Transformations, 2003
- [CWSM03] Compuware Corporation, SUN Microsystems: XMOF: Queries, Views and Transformations on Models using MOF, OCL and Patterns, 2003
- [IO04] Interactive Objects Software GmbH: 2nd Revised Submission to MOF Query/View/Transformation RFP, 2004
- [QVTMG04] QVT-Merge Group: Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10), 2004
- [QVTMG05] QVT-Merge Group: Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10), 2005
- [GLRSW02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, Andrew Wood: Transformation: The Missing Link of MDA, 2002
- [AGLIN06] ATLAS Group, LINA & INRIA Nantes:ATL: Atlas Transformation Language, 2006
- [IBM04] IBM: IBM Model Transformation Framework 1.0.0, Programmer's Guide, 2004
- [SH04] Steve Holzner: Eclipse, 2004 (<http://www.eclipse.org>)
- [BEGMS03] Frank Budinsky, Ray Ellersick, Timothy J. Grose, Ed Merks, David Steinberg: Eclipse Modeling Framework, 2003
- [KH05] Kenn Hussey: Getting Started with UML2, 2005
- [ad/2004-04-07] Object Management Group: MOF Model to Text Transformation Language RFP, 2004